

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
УЖГОРОДСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІНЖЕНЕРНО-ТЕХНІЧНИЙ ФАКУЛЬТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

Мамай Л. М. Самусь Є.І.

Паралельне програмування для комп'ютерних систем зі спільною пам'яттю

**Методичні вказівки і завдання до лабораторних робіт з курсу
«Паралельні та розподілені обчислення»**

*для студентів 4-го курсу інженерно-технічного факультету,
спеціальність 123 — «Комп'ютерна інженерія»*

Методичні вказівки і завдання до лабораторних робіт з курсу «Паралельні та розподілені обчислення», для студентів 4-го курсу інженерно-технічного факультету спеціальності 123 — «Комп'ютерна інженерія»

Укладачі: Мамай Л.М., канд. фіз.-мат наук, доцент каф. КСМ,
Самусь Є.І. старший викладач каф КСМ

© Мамай Л. М., Самусь Є.І. 2021

ВСТУП

Паралельна обробка інформації створює передумови для суттєвого підвищення продуктивності засобів обчислювальної техніки. Тому останні десятиріччя пов'язані з швидким розвитком паралельних комп'ютерних систем (ПКС). Поява та швидке розповсюдження мов програмування, таких як Java, C# визначили новий напрям в дослідженнях проблем програмування багатоплатформних розподілених та паралельних застосувань. Тому для інженера або програміста важливо знати й використовувати основні поняття та засоби мультипроцесорної обробки й паралельного програмування для підвищення ефективності та надійності обчислень.

Метою дисципліни «Паралельні та розподілені обчислення» є придбання студентами знань про теорію та методи паралельних обчислень; ознайомлення з новими досягненнями в розвитку паралельних обчислювальних систем, а також придбання ними практичних навичок в розробці сучасного паралельного програмного забезпечення.

Дані методичні вказівки і завдання до лабораторних робіт з курсу «Паралельні та розподілені обчислення» розроблено для студентів 4-го курсу інженерно-технічного факультету спеціальності 123 — «Комп'ютерна інженерія» для вивчення основних принципів побудови паралельних алгоритмів та додатків для паралельних обчислювальних систем зі спільною пам'яттю. Завдання, які пропонуються для виконання — це побудова алгоритмів та програм для виконання матрично-векторних операцій.

Запропоновано п'ять лабораторних робіт: перші дві стосуються застосування семафорів та м'ютексів для вирішення завдання взаємного виключення та синхронізації процесів, три наступних — використання моніторів та їх реалізація у мовах програмування *Ada*, *C#* та *Java*. Наведено короткі теоретичні відомості, які необхідні для виконання завдань кожної з лабораторних робіт та типовий приклад.

Завдання для лабораторних робіт зведені в дві таблиці і знаходяться в додатку. При застосуванні семафорів та м'ютексів як засобів взаємодії процесів, комп'ютерна система, для якої потрібно розробити програму – двопроцесорна; у випадку використання моніторів — чотирипроцесорна.

Виконуючи завдання лабораторних робіт, студенти вчаться розробляти паралельні алгоритми та алгоритми роботи процесів (потоків), будувати структурні схеми взаємодії потоків (задач), розробляти паралельні програми на мовах програмування *Ada*, *C#*, *Java*, що мають вбудовані засоби роботи з потоками.

Лабораторна робота №1

Тема: Семафори в мові *Ada*.

Мета роботи: Вивчення можливостей мови *Ada* для роботи з процесами; Застосування семафорів мови *Ada* для вирішення завдання взаємного виключення та синхронізації процесів.

Теоретичні відомості

1.1 Процеси в мові *Ada*

Ada — одна з перших мов програмування, які мають вбудовані засоби роботи з процесами. Процеси в мові *Ada* реалізовані у вигляді спеціальних модулів - задач (*task*). Задачний модуль *task* має стандартну форму у вигляді специфікації і тіла. Специфікація визначає інтерфейс задачі, де задається ім'я задачі, а також у разі потреби, пріоритет задачі, засоби взаємодії з іншими задачами, місце розташування в пам'яті. Простіший вид специфікації задачі містить тільки ім'я задачі і називається виродженою:

task A; - - специфікація задачі *A* (вироджена)

task A is

pragma Priority(4); - - специфікація задачі *A* з вказаним пріоритетом
end A;

Тіло задачі визначає дії задачі під час виконання:

task body A is

begin

 -- оператори,

 -- тіло задачі

end A;

1.2 Семафори в мові *Ada*

Семафор — системно-апаратний механізм і включає 3 речі: тип *Semaphore* і дві неподільні операції над змінною *s* цього типу: *P(s)* і *V(s)*. Неподільність операції означає, що якщо якийсь процес працює з семафором, ніякий інший не може з ним працювати (її не можна переривати, поки не завершиться її виконання).

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

Розв'язання завдання взаємного виключення полягає в попередженні одночасного звернення процесів до одного і того ж спільного ресурсу (СР), що призводить до конфлікту процесів.

У мові *Ada95* механізм семафорів подано у вигляді пакета *Synchronous_Task_Control*. Семафорний тип забезпечується приватним типом *Suspension_Object* і набуває значень *false* і *true*, тобто є бінарним логічним семафором.

Операції $P(S)$, $V(S)$ реалізовані за допомогою процедур

$P(S)$ — *Suspend_Until_True (S)*

$V(S)$ — *SetTrue (S)*.

Зауваження. При створенні семафору за замовчуванням $S = 0$ (*false*).

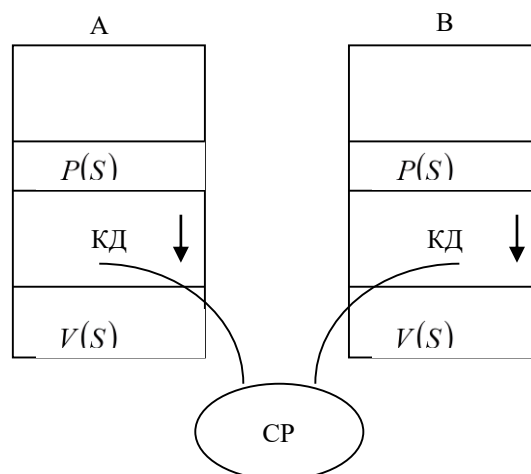


Рис. 1.1. Загальна схема вирішення ЗВВ з використанням семафорів



Рис. 1.2 Загальна схема вирішення СП з використанням семафорів

Приклад виконання завдання

Постановка задачі. Реалізувати обчислення за формулою $a = \alpha(B \cdot C)$ у двопроцесорній системі зі спільною пам'яттю (рис. 1.3) з використанням семафорів мови *Ada*.

Розробити: а) паралельний алгоритм; б) алгоритми роботи процесів; в) структурну схему взаємодії задач; в) програму на мові *Ada*.

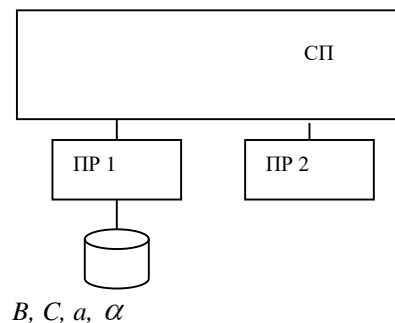


Рис. 1.3 Структурна схема двопроцесорної системи

1 Етап. Побудова паралельного алгоритму

$$a = \underbrace{\alpha \cdot (b_1 \cdot c_1 + b_2 \cdot c_2 + \dots + b_H \cdot c_H)}_{a_1} + \underbrace{\alpha \cdot (b_{H+1} \cdot c_{H+1} + \dots + b_N \cdot c_N)}_{a_2};$$

$$a = a + a_i, i = 1, 2;$$

СП: a, α

$$H = \frac{N}{P}; P = 2, P — \text{число процесорів.}$$

2 Етап. Розробка алгоритмів роботи процесів

Таблиця 1.1

T_1		T_2	
1. Ввід B, C, α ;		1. Чекати сигнал від T_1 про завершення введення;	W_{11}
2. Сигнал T_2 про завершення введення даних;	S_{21}	2. Копіювати α $\alpha_2 := \alpha$;	КД
3. Копіювати α $\alpha_1 := \alpha$;	КД	3. Обчислення $a_2 = \alpha_2 \cdot (B_H \cdot C_H)$;	
4. Обчислення $a_1 = \alpha_1 \cdot (B_H \cdot C_H)$;		4. Обчислення $a = a + a_2$;	КД
5. Обчислення $a = a + a_1$	КД	5. Сигнал T_1 про завершення обчислень;	S_{11}
6. Чекати сигнал від задачі T_2 про завершення обчислень.	W_{21}		
7. Виведення a .			

3 Етап. Розробка структурної схеми взаємодії задач

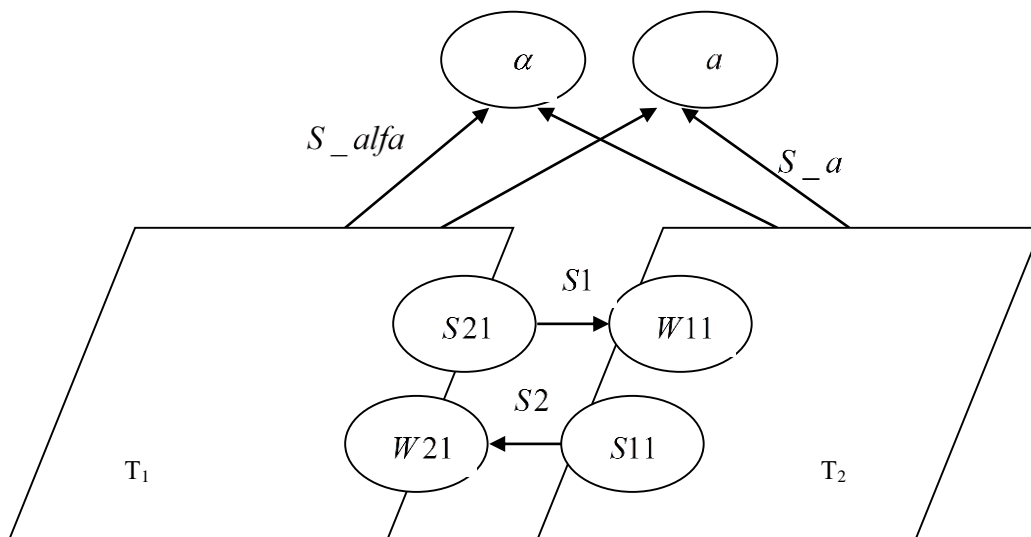


Рис. 1.4 Структурна схема взаємодії задач

На структурній схемі взаємодії задач S_alfa , S_a , $S1$, $S2$ — семафори:

S_alfa — для керування доступом до спільного ресурсу α ;

S_a — для керування доступом до спільного ресурсу a ;

S1 — для синхронізації із завершення введення в *T1*;

S2 — для синхронізації із завершення обчислень в *T2* і виведення результату.

4 Етап. Розробка програми

```
With Ada.Text_IO, Ada.Integer_Text_IO,  
Ada.Synchronous_task_control;  
Use Ada.Text_IO, Ada.Integer_Text_IO,  
Ada.Synchronous_task_control;
```

```
Procedure Lab2 is
```

```
  N:integer:=100;  
  P:integer:=2;  
  H:integer:=N/p;  
  type vect is array (1..N) of integer;  
  
  S_a, S_alfa, S1, S2: Suspension_Object;  
  a, alfa : integer:=0;  
  B, C : vect;
```

```
Procedure Zapusk is
```

```
  Task T1;  
  Task body T1 is  
  a1, alfa1:integer:=0;  
begin  
  put_line (" T1 started");  
  new_line;  
  
  for i in 1..N loop  
    B(i):=1;  
    C(i):=1;  
  End loop;  
  alfa:=2;  
  Set_True(S1);  
  Suspend_Until_True(S_alfa);  
  alfa1:= alfa;  
  Set_True(S_alfa);  
  
  for i in 1..H loop  
    a1:=a1+C(i)*B(i);  
  End loop;  
  a1:=a1*alfa1;
```



```

                                Suspend_Until_True(S_a);
                                a:= a+a1;
                                Set_True(S_a);

                                Suspend_Until_True(S2);
                                put("a"); put(a,3);
                                put (" T1 finished");
                                new_line;
end T1;

Task T2;
Task body T2 is
    alfa2, a2 :integer:=0;
begin

    put (" T2 started");
    Suspend_Until_True(S1);

                                Suspend_Until_True(S_alfa);
                                alfa2:= alfa;
                                Set_True(S_alfa);

                                for i in H+1..N loop
                                    a2:=a2+C(i)*B(i);
                                End loop;
                                a2:= alfa2* a2;

                                Suspend_Until_True(S_a);
                                a:= a+a2;
                                Set_True(S_a);

                                Set_True(S2);

                                put (" T2 finished");
end T2;

Begin
    null;
    end Zapusk;

Begin

    put("Main started");
    Set_True(S_a);
    Set_True(S_alfa);

```

```

        Zapusk;
    put("Main finished");
    -- new_line;

```

End Lab2;

Завдання для самостійного виконання.

Реалізувати обчислення за поданою у таблиці 1 додатку формулою у двопроцесорній ПОС зі спільною пам'яттю з використанням семафорів мови *Ada*.

Розробити:

- а) паралельний алгоритм;
- б) алгоритми роботи процесів;
- в) структурну схему взаємодії задач;
- г) програму на мові *Ada*.

Номери варіантів та відповідні їм номери завдань з таблиці 1 додатка наведено у таблиці 1.2

Таблиця 1.2

Номер варіант	Номер завдання	Номер варіанту	Номер завдання
1	1	11	11
2	2	12	12
3	3	13	13
4	4	14	14
5	5	15	15
6	6	16	16
7	7	17	17
8	8	18	18
9	9	19	19
10	10	20	20

Лабораторна робота №2

Тема: Семафори та м'ютекси мови C#

Мета роботи: Вивчення можливостей мови C# для роботи з потоками: створення потоків; вирішення завдання взаємного виключення та синхронізації, використовуючи синхронізуючі об'єкти — семафори та м'ютекси.

Теоретичні відомості

2.1 Створення потоків в C#. Клас *Thread*

Система багатопотокової обробки ґрунтується на класі *Thread*, який інкапсулює потік виконання. У класі *Thread* визначений ряд методів і властивостей, призначених для управління потоками. Для створення потоку досить отримати екземпляр об'єкту типу *Thread*.

Проста форма конструктора класу *Thread* :

```
public Thread(ThreadStart запуск)
```

де

запуск — це ім'я методу, що викликається з метою почати виконання потоку, *ThreadStart* — делегат, визначений в середовищі .NET Framework:

```
public delegate void ThreadStart().
```

Метод, що вказується в якості точки входу в потік, повинен мати тип значення, що повертається — *void* і не приймати ніяких аргументів.

Існує ще одна форма створення потоку:

```
Thread myThread = new Thread(Count).
```

Створений новий потік не почне виконуватися до тих пір, поки не буде викликаний його метод *Start()*, визначений в класі *Thread*.

одна з форм оголошення методу *Start()* :

```
public void Start().
```

Почавшись, потік виконуватиметься до тих пір, поки не станеться повернення з методу, на який вказує запуск.

Часто використовується статичний метод *Thread.Sleep()*, визначений в класі *Thread*:

```
public static void Sleep(int мілісекунд_простоя),
```

де *мілісекунд_простоя* означає період часу, на який призупиняється виконання потоку. Цей метод обумовлює призупинення того потоку, з якого він

був викликаний, на певний період часу, що вказується в мілісекундах. Якщо вказана кількість мілісекунд_простою дорівнює нулю, то потік, що викликає призупиняється лише для того, щоб надати можливість для виконання потоку, що очікує своєї черги на виконання.

2.2 Керування потоками

М'ютекс. М'ютекс є синхронізуючим об'єктом, що використовується для задачі взаємного виключення. М'ютекс призначений для тих ситуацій, в яких спільний ресурс може бути використаний тільки в одному потоці.

М'ютекс реалізовано в класі `System.Threading.Mutex` і має декілька конструкторів. Розглянемо конструктор:

```
public Mutex() .
```

В цій формі конструктора створюється м'ютекс, яким спочатку ніхто не володіє. Для того, щоб отримати м'ютекс, в коді програми слід викликати метод `WaitOne()` для цього м'ютекса `public bool WaitOne()`. Метод `WaitOne()` чекає до тих пір, поки не буде отриманий м'ютекс, для якого він був викликаний. Отже, цей метод блокує виконання потоку, який викликав метод до тих пір, поки не стане доступним вказаний м'ютекс. Він завжди повертає логічне значення `true`.

Коли ж в коді більше не вимагається володіти м'ютексом, він звільняється за допомогою виклику методу `ReleaseMutex()`:

```
public void ReleaseMutex()
```

В цій формі метод `ReleaseMutex()` звільняє м'ютекс, для якого він був викликаний, що дає можливість іншому потоку отримати цей м'ютекс.

Для застосування м'ютекса з **метою синхронізувати доступ до спільного ресурсу** методи `WaitOne()` і `ReleaseMutex()` використовуються так:

```
Mutex myMtx = new Mutex();
```

```
// код...
myMtx.WaitOne(); // очікувати отримання м'ютекса

// Отримати доступ до СР (Критична ділянка)

myMtx.ReleaseMutex(); // звільнити м'ютекс

// код...
```

При виклику методу `WaitOne()` виконання відповідного потоку призупиняється до тих пір, поки не буде отриманий м'ютекс `myMtx`, а при виклику методу `ReleaseMutex()` м'ютекс звільняється і потім може бути отриманий іншим потоком. Завдяки такому підходу до синхронізації одночасний доступ до спільного ресурсу обмежується тільки одним потоком.

Семафор. Семафор подібний мютексу, за винятком того, що він надає одночасний доступ до спільного ресурсу не одному, а декільком потокам. Тому семафор придатний для синхронізації цілого ряду ресурсів.

Семафор управляє доступом до спільного ресурсу, використовуючи для цієї мети лічильник:

- якщо значення **лічильника більше нуля**, то доступ до ресурсу **дозволений**;
- якщо це значення **дорівнює нулю**, то доступ до ресурсу **заборонений**.

Якщо значення лічильника семафора більше нуля, то потік отримує дозвіл, а лічильник семафора декрементується. Інакше потік блокується до тих пір, поки не отримає дозвіл. Коли ж потоку більше не потрібно доступ до спільного ресурсу, він звільняє дозвіл, а лічильник семафора інкрементується.

Зауваження. Якщо створити семафор, що одночасно дозволяє тільки один доступ, то такий семафор діятиме як м'ютекс.

Семафор реалізується в класі `System.Threading.Semaphore`. Найпростіша форма конструктора цього класу:

```
public Semaphore(int initialCount, int maximumCount),
```

де `initialCount` - це первинне значення для лічильника дозволів семафора, тобто кількість спочатку доступних дозволів, `maximumCount` — максимальне значення цього лічильника, тобто максимальна кількість дозволів, які може дати семафор.

Семафор може застосовуватися так само, як і описаний раніше м'ютекс.

Для отримання доступу до спільного ресурсу в коді програми викликається метод `WaitOne()`. Метод `WaitOne()` очікує до тих пір, доки не буде отриманий семафор для якого він викликається. Таким чином він блокує виконання викликаючого потоку до тих пір поки вказаний семафор не дозволить доступ до СР.

Якщо коду більше не потрібно володіти семафором, він його звільняє, викликаючи метод `Release()`. Нижче приведені дві форми цього метода:

`public int Release()` – звільняє тільки один дозвіл.

`public int Release (int releaseCount)` – кількість дозволів визначається параметром `releaseCount`.

Метод `WaitOne ()` допускається викликати в потоці кілька разів перед викликом методу `Release ()`. Але кількість викликів методу `WaitOne ()` має дорівнювати кількості викликів методу `Release ()` перед вивільненням дозволу. З іншого боку, можна скористатися формою виклику методу `Release (int num)`, щоб передати кількість дозволів, що вивільняються, рівну кількості викликів методу `WaitOne ()`.

Нижче наведений приклад програми, в якій демонструється застосування м'ютекса `mtx` для синхронізованого доступу до спільного ресурсу *a* двох потоків (відбувається додавання числа 5 до *a* кожним потоком) та семафора `sem` для синхронізації процесів щодо введення даних другим потоком. На рисунку 2.1 Зображено структурну схему при даній взаємодії потоків.

```
using System;
using System.Threading;
using System.Text;

namespace Semaphore_mutex
{
    class Shared
    {
        // public static int Count;
        public static int a;
        public static Mutex mtx = new Mutex();
        public static Semaphore sem = new Semaphore(0, 1);
    }
    class ClThread1
    {
        public Thread thr1;
        public ClThread1(string name)
        {
            thr1 = new Thread(new ThreadStart(this.Run));
            thr1.Name = name;
            thr1.Start();
        }
        void Run()
        {
            Console.WriteLine("thr1 started");
            // Критична ділянка. Використання м'ютекса mtx.
            Shared.mtx.WaitOne();
        }
    }
}
```

```

        Shared.a += 5;
        Shared.mtx.ReleaseMutex();

        // «сигнал» про завершення операції додавання.
        // Використання семафора sem.

        Shared.sem.Release();
    }
}

class ClThread2
{
    public Thread thr2;
    public ClThread2(string name)
    {
        thr2 = new Thread(new ThreadStart(this.Run));
        thr2.Name = name;
        thr2.Start();
    }
    void Run()
    {

        // Критична ділянка. Використання м'ютекса mtx.
        Shared.mtx.WaitOne();
        Shared.a += 5;
        Shared.mtx.ReleaseMutex();

        // Очікування завершення обчислення у першому потоці.
        // Використання семафора sem.

        Shared.sem.WaitOne();

        Console.WriteLine("Print a={0}", Shared.a);
        Console.WriteLine("thr2 finished");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main started");

        ClThread1 one = new ClThread1("first");
        ClThread2 two = new ClThread2("second");
    }
}

```

```

        one.thr1.Join();
        two.thr2.Join();

        Console.WriteLine("Main finished");
        Console.ReadLine();
    }
}

```

На рисунку 2.1 Зображено структурну схему взаємодії потоків при вирішенні задачі взаємного виключення, використовуючи м'ютекс. Count — спільний ресурс.

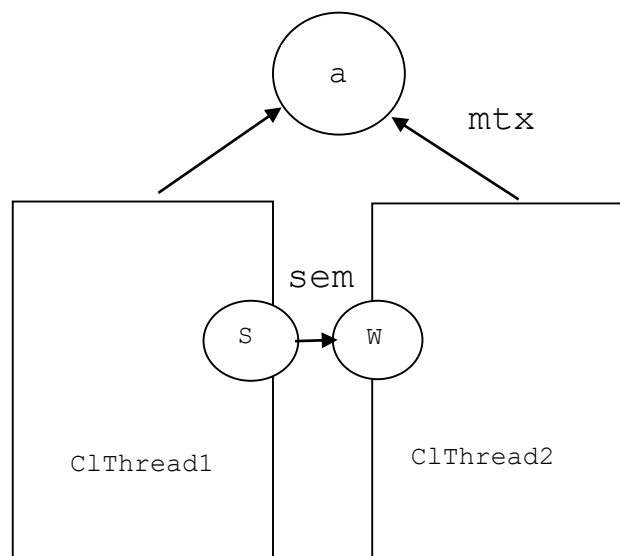


Рис 2.1. Структурна схема взаємодії потоків

`Join()` — це метод синхронізації, який блокує потік, який викликає цей метод до тих пір, поки не завершиться потік, метод `Join()` якого був викликаний. Використовуйте цей метод, щоб переконатися, що потік був завершений.

Завдання для самостійного виконання.

Реалізувати обчислення за поданою у таблиці 2 додатка формулою у двопроцесорній ПОС зі спільною пам'яттю з використанням семафорів та м'ютексів мови C#.

Розробити:

- а) паралельний алгоритм;
- б) алгоритми роботи процесів;
- в) структурну схему взаємодії потоків;
- г) програму на мові C#.

Номери варіантів та відповідні їм завдання із таблиці 2 додатка наведено у таблиці 2.1.

Таблиця 2.1

Номер варіант	Номер завдання	Номер варіанту	Номер завдання
1	11	11	1
2	12	12	2
3	13	13	3
4	14	14	4
5	15	15	5
6	16	16	6
7	17	17	7
8	18	18	8
9	19	19	9
10	20	20	10

Лабораторна робота №3

Тема: Захищені модулі мови *Ada*

Мета роботи: Вивчення можливостей захищених модулів мови *Ada* для вирішення проблеми доступу до спільних ресурсів та синхронізації процесів.

Теоретичні відомості

3.1 Монітори в мові *Ada*

Монітор в мові *Ada* називається захищеним модулем *Protected Unit*. В захищеному модулі об'єднуються спільні дані і операції над ними (захищені операції). Доступ до спільних ресурсів можливий тільки через захищені операції.

Захищенні операції-це:

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

Захищені функції забезпечують доступ тільки до читання чи копії захищених елементів. Тіло захищеної функції може містити виклик іншої захищеної функції але не виклик захищеної процедури.

Захищені процедури Виклик захищеної процедури дозволяє процесу як читати, так і змінювати інформацію в захищеному модулі. На відміну від захищеної функції під час виконання захищеної процедури дозволяється змінювати дані. Захищені процедури забезпечують ексклюзивний доступ до захищених елементів через читання і запис. У тілі захищеної процедури дозволено виклик як захищеної функції, так і захищеної процедури.

Приклад реалізації процедур та функцій монітора

Protected Box is

```
Function Read return integer;  
Procedure Write (x: in integer );  
    Private  
    Z:integer:=10      -- CP
```

End Box;

Protected body Box is

```
Function Read return integer is  
    Begin  
        Return z;  
    End Read;  
Procedure Write (x: in integer )is  
    Begin  
        Z:=x;          -- CP
```

```
End Write
End Box;
```

Виклик процедур та функцій монітора задачами *A* та *B*

Task A

Task B

```
Box.Write(S)
```

```
X:=Box.Read
```

Захищені входи забезпечують ті самі функції, що й захищені процедури, додатково реалізуючи за допомогою *бар'єрів* ексклюзивний (умовний) доступ до тіла захищеного входу. Це дозволяє реалізувати за допомогою входів вирішення завдання синхронізації. Конструкція **When Умова** — це бар'єр, де “**Умова**” — логічний вираз, який визначає *відкритий* або *закритий* вхід.

Приклад реалізації Захищеного входу:

```
Protected Box is
  Entry Wait;
  Procedure Signal;
  Private
    F:integer:=0;      -- CP
End Box2;
```

```
Protected body Box is
  Entry Wait when F=1 is
  Begin
    Null;
  End Wait;

  Procedure Signal is
  Begin
    F:=1;
  End Signal;
End Box2;
```

Нижче наведено приклад виклику входу *Wait* та процедури *Signal* задачами *A* та *B*:

Task A

Task B

```
Box.Signal
```

```
Box.Wait
```

Типовий приклад виконання завдання.

Постановка задачі. Реалізувати обчислення за формулою $A = B + \alpha \cdot C$ у чотирипроцесорній системі зі спільною пам'яттю (рис. 3.1) з використанням захищеного модуля мови *Ada*.

Розробити: а) паралельний алгоритм; б) алгоритми роботи процесів; в) структурну схему взаємодії задач; г) програму на мові *Ada*.

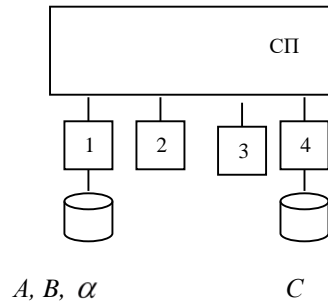


Рис. 3. 1 Структурна схема чотирипроцесорної системи

1 Етап. Побудова паралельного алгоритму

$$A = B_H + \alpha \cdot C_H; \quad \text{CP: } \alpha; \quad H = \frac{N}{P}; \quad P = 4.$$

2 Етап. Розробка алгоритмів роботи процесів

T_1		T_2		T_3	T_4
1. Ввід B, α . 2. Сигнал T_2-T_4 про завершення введення B, α . 3. Чекати від задачі T_4 сигнал про завершення введення C . 4. Копіювати α $\alpha_1 := \alpha$. 5. Обчислення $A_H = B_H + \alpha \cdot C_H$. 6. Чекати від задач T_2-T_4 сигнал про завершення обчислень. 8. Виведення A .	КД	1. Чекати від задачі T_1 сигнал про завершення введення B, α . 2. Чекати від задачі T_4 сигнал про завершення введення C . 3. Копіювати α $\alpha_2 := \alpha$. 4. Обчислення $A_H = B_H + \alpha \cdot C_H$. 5. Сигнал T_1 про завершення обчислень.	КД		1. Ввід C . 2. Сигнал T_1-T_3 про завершення введення C . 3. Чекати від задачі T_1 сигнал про завершення введення B, α . 4. Копіювати α $\alpha_4 := \alpha$. 5. Обчислення $A_H = B_H + \alpha \cdot C_H$. 6. Сигнал T_1 про завершення обчислень.

3 Етап. Розробка структурної схеми взаємодії задач

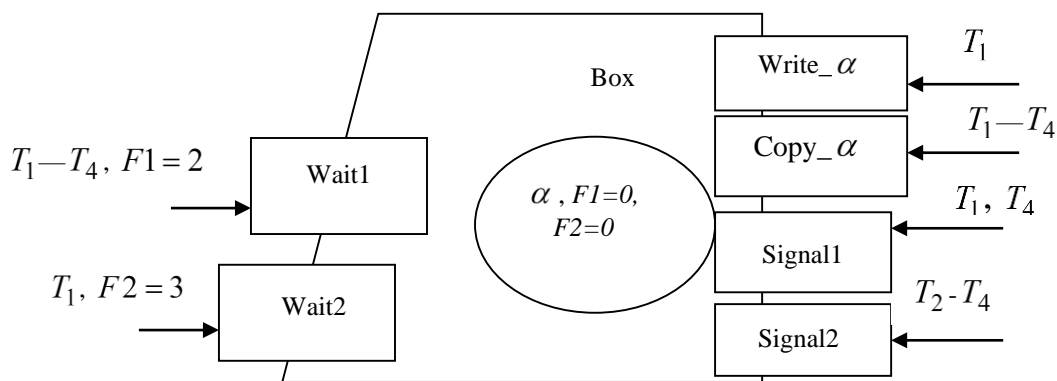


Рис. 3.2 Захищений модуль в операції $A = B + \alpha \cdot C$

Захищений модуль *Box* включає три захищені елементи: α , $F1$, $F2$ а також вхід *Wait1* — для синхронізації з введення даних в $T1$ та $T2$, вхід *Wait2* — для синхронізації з завершення обчислень в задачах $T2-T4$, функцію *Write_alpha* для копіювання спільного ресурсу α , процедуру *Signal1* для сигналу про завершення введення даних, процедуру *Signal2* для сигналу про завершення обчислень у задачах $T2-T4$.

3 Етап. Розробка програми

```
With Ada,Text_Io, Ada,Integer_Text_Io;
Use Ada,Text_Io, Ada,Integer_Text_Io;
Procedure Lab3 is
  N:integer:=8;
  P:integer:=4;
  H:integer:=N/P;
  Type Vect is array (1..
N) of integer;
  A, B, C : Vect;
  Protected Box is
    Procedure Write_a (x: in integer);
    Function Copy_a return integer;
    Procedure Signal1;
    Procedure Signal2;
    Entry Wait1;
    Entry Wait2;
  Private
    a: integer:=0;
    F1: integer:=0;
    F2: integer:=0;
  End Box;
  Protected body Box is
    Procedure Write_a (x: in integer) is
      Begin
        a:=x;
```

```

    End Write_a;

Function Copy_a return integer is
    Begin
        return a;
    End Copy_a;
Procedure Signall is
    Begin
        F1:=F1+1;
    End Signall;

Procedure Signal2 is
    Begin
        F2:=F2+1;
    End Signal2;

Entry Wait1 when F1=2 is
    Begin
        null;
    End Wait1;
Entry Wait2 when F2=3 is
    Begin
        null;
    End Wait2;
End Box;
Task T1;
Task body T1 is
    -- A1, B1, C1: Vect;
    aal, aa: integer:=0;
    Begin
        put("T1 started");
        new_line;

        -- ввід а
        aa:=7;

        -- запис CP у BOX
        Box,Write_a(aa);

        -- ввід B
        For i in 1,,N loop
            B(i):=1;
        End loop;

        -- Сигнал задачам (T1-T4) про завершення вводу B, а
        Box,Signall;

        -- Чекати сигнал про завершення вводу від задачам T4
        Box,Wait1;

        -- копія CP

```

```

        aa1:=Box,Copy_a;
    -- обчислення
        For i in 1,,N loop
            A(i):=B(i)+aa1*C(i);
        End loop;
    -- Чекати сигнал про завершення обчислень від задач (T2-T4)
        Box,Wait2;

    -- вивід результату
        new_line;
        For i in 1,,N loop
            Put(A(i));
        End loop;
    new_line;
        put("T1 finished");
End T1;

```

```

Task T2;
Task body T2 is
    aa2: integer:=0;
    Begin
        put("T2 started");
        new_line;

    -- Чекати сигнал про завершення вводу від задач T1, T4
        Box,wait1;
    -- копія CP
        aa2:=Box,Copy_a;
    -- обчислення своєї частини матриці
        For i in N+1,,2*N loop
            A(i):=B(i)+aa2*C(i);
        End loop;
    -- Сигнал про закінчення обчислень
        Box,Signal2;
        new_line;
        put("T2 finished");
End T2;
Task T3;
Task body T3 is

```

```

    aa3: integer:=0;
    Begin
        put("T3 started");
        new_line;
        -- Чекати сигнал про завершення введення від задач T1,
T4
        Box,Wait1;

    -- копія CP
        aa3:=Box,Copy_a;

    -- обчислення своєї частини матриці
    For i in 2*N+1,,3*N loop

```

```

        A(i):=B(i)+aa3*C(i);
End loop;

-- Сигнал про закінчення обчислень
    Box,Signal2;
    new_line;
    put("T3 finished");
End T3;

Task T4;
    Task body T4 is
    -- A4, B4, C4: Vect;
    aa4:integer:=0;
    Begin
        put("T4 started");
        new_line;

        -- Ввід C
        For i in 1,,N loop
            C(i):=1;
        End loop;

        -- Сигнал задачам про завершення введення

            Box,Signal1;

        -- Чекати сигнал про завершення введення
            Box,Wait1;
            new_line;

        -- копія CP
            aa4:=Box,Copy_a;

        -- обчислення своєї частини матриці

            For i in 3*N+1,,N loop
                A(i):=B(i)+aa4*C(i);
            End loop;

        -- Сигнал про закінчення обчислень
            Box,Signal2;
            new_line;
            put("T4 finished"); End T4;

Begin
    new_line;
    put("main started");
End Lab3;

```


Завдання для самостійного виконання

Реалізувати обчислення за поданою у таблиці 2 додатку формулою у відповідній чотирипроцесорній ПОС зі спільною пам'яттю з використанням захищеного модуля мови *Ada*.

Розробити:

- а) паралельний алгоритм;
- б) алгоритми роботи процесів;
- в) структурну схему взаємодії задач;
- в) програму на мові *Ada*.

Номери варіантів та відповідні їм завдання із таблиці 2 додатка наведено у таблиці 3.1.

Таблиця 3.1

Номер варіант	Номер завдання	Номер варіанту	Номер завдання
1	1	11	11
2	2	12	12
3	3	13	13
4	4	14	14
5	5	15	15
6	6	16	16
7	7	17	17
8	8	18	18
9	9	19	19
10	10	20	20

Лабораторна робота №4

Тема: Монітори в мові *Java*

Мета роботи: Вивчення можливостей мови *Java* для роботи з потоками: створення потоків, вирішення завдання взаємного виключення та засоби синхронізації потоків, за допомогою синхронізованих методів мови *Java*.

Теоретичні відомості

4.1 Процеси в мові *Java* та методи для керування потоком. Створення потоку.

Процес в мові *Java* реалізований у вигляді *потоку* (thread). Класи-потоки створюються з використанням класу *Thread*.

Методи для керування потоком:

Метод *start()* дозволяє запустити потік на виконання.

Метод *sleep()*: дозволяє призупинити потік на заданий відрізок часу:

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

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

Метод *join()* дозволяє організовувати очікування завершення викликаного потоку:

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

Можливе використання в головному потоці, якщо потрібно, щоб він був завершений останнім після завершення усіх запусчених ним потоків.

Існують два підходи до створення потоку: за допомогою розширення класу *Thread* та з використанням інтерфейсу *Runnable*.

Створення потоку: за допомогою розширення класу *Thread*.

Для створення потоку потрібно оголосити підклас через розширення класу *Thread* і створити екземпляр цього підкласу. Підклас має перевизначити метод *run()*, що є точкою входу в потік. Необхідно також викликати метод *start()*, щоб почати виконання потоку.

Нижче наведений приклад створення потоків шляхом розширення класу *Thread*.

```

Class Thread1 extends Thread {

    // перевизначення методу run()

    public void run() {
        System.out.println("Process Thread1 ");
    }
}

class Thread2 extends Thread {

    public void run() {
        System.out.println("Process Thread2 ");
    }
}

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

// оголошення екземплярів потоків

        Thread1 T1 = new Thread1( );

        Thread2 T2 = new Thread2( );

// запуск потоків

        T1.start();
        T2.start();

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

    } // main

} // MainThread

```

Використання інтерфейсу *Runnable*.

Потік можна створити використовуючи клас `java.lang.Runnable`. Цей інтерфейс має єдиний метод `public void run()`, який необхідно перевизначити в створюваному класі – потоці. У методі `run()` слід описати код, який визначає дії створюваного потоку.

Нижче наведений приклад створення потоків з використанням інтерфейсу `Runnable`.

```

class Denon implements Runnable{

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

    public void run(){

        try{
            for (int i=1; i<5; i++){
                System.out.println("Start of process " + Name);

Thread.sleep(500); } }
                catch(InterruptedException e){
                    System.out.println("Error in process");

                    System.out.println("Finish of process " + Name);
                }
            }
        // Головний потік, що використовує клас Denon

    public class Titan {

        // точка входу в основний клас

        public static void main(String args[]){
            System.out.println("Process Titan started");

            // оголошення екземплярів класу Denon

            Denon A = new Denon ("A");

            Denon B = new Denon ("B");

            // чекати завершення потоку A і B

            try

            {
                A.t.join() ;
                B.t.join() ;
            }
        }
    }
}

```

```

catch (InterruptedException e)
{
    System.out.println("Error in Titan process");
    System.out.println("Process Titan finished ")
}

```

4.2 Монітори в мові Java

Монітор — програмний модуль, що містить змінні та процедури для роботи з ними, причому доступ до змінних можливий тільки через процедури монітора.

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

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

Приклади синхронізованих методів:

```

synchronized void f1 (double x)
{
    // оператори
}

synchronized int f2 (int x, int y)
{
    // оператори
    return c}

```

Приклад реалізації монітора в мові Java

```

class Box
{
    private int a;        // спільний ресурс
    // синхронізовані процедури доступу до CP
}

```

```
synchronized int Read ( )

{ return a; }

synchronized void Write (int x) {
a= x; }
} // Box
```

Вирішення завдання синхронізації процесів в мові Java

Синхронізація процесів в мові Java виконується за допомогою методів `wait()`, `notify()` або `notifyAll()`. Ці методи оголошені в класі `Object` і доступні всім класам.

Метод `notify()` розблоковує потік, який заблоковано методом `wait()`.

Метод `notifyAll()` розблоковує *усі* потоки, які заблоковані методом `wait()` в одному об'єкті.

Якщо на момент виконання методів `notify()` і `notifyAll()` не існує заблокованих потоків, то методи ігноруються.

Якщо треба методом `notify()` обов'язково повідомити процес про подію, яка відбулася до того, як він буде її чекати, необхідно застосувати додаткову спільну змінну, значення якої треба перевіряти разом з використанням методу `wait()`.

Зауваження. Методи `wait()`, `notify()` і `notifyAll()` взаємодіють тільки тоді, коли вони прив'язані до одного і того ж об'єкта. Крім того, всі три методи можуть бути викликані тільки всередині синхронізованого методу.

Типовий приклад виконання завдання.

Постановка задачі. Реалізувати обчислення за формулою $a = \min[(B + \alpha \cdot C)MD]$ у трипроцесорній системі зі спільною пам'яттю (рис. 4.1) з використанням синхронізованих методів мови *Java*.

Розробити: а) паралельний алгоритм; б) алгоритми роботи процесів; в) структурну схему взаємодії потоків; г) програму на мові *Java*.

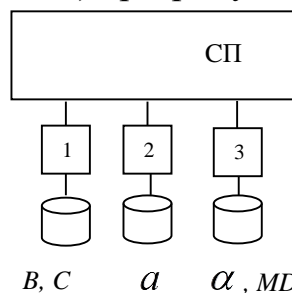


Рис. 4.1. Структурна схема трипроцесорної системи

1 Етап. Побудова паралельного алгоритму

$$T_H = B_H + \alpha \cdot C_H,$$

$$a_i = \min(T * MD_H),$$

$$a = \min a_i, \quad i = 1, 2, 3.$$

CP: α, T ; $H = \frac{N}{P}; P = 3.$

2 Етап. Розробка алгоритмів роботи процесів

aThread	bThread	cThread
<ol style="list-style-type: none"> 1. Ввід B, c. 2. Сигнал про завершення введення; 3. Чекати сигнал про завершення введення. 4. Копіювати α $\alpha1 := \alpha$. 5. Обчислення $T_H = B_H + \alpha1 \cdot C_H$. 6. Чекати сигнал про завершення обчислення T 7. Запис T. 8. Копіювати T: $T1 := T$. 9. Обчислення $a1 = \min(T1 * MD_H),$ $a = \min(a1, a)$ 10. Сигнал про завершення обчислень 	<ol style="list-style-type: none"> 1. Чекати сигнал про завершення введення. 2. Копіювати α $\alpha2 := \alpha$. 3. Обчислення $T_H = B_H + \alpha2 \cdot C_H$. 4. Сигнал про завершення обчислення T_H 5. Копіювати T: $T2 := T$. 6. Обчислення $a2 = \min(T2 * MD_H),$ $a = \min(a2, a)$ 7. Чекати сигнал про завершення обчислень 8. Вивід a. 	<ol style="list-style-type: none"> 1. Ввід α. 2. Запис α. 3. Сигнал про завершення введення; 4. Чекати сигнал про завершення введення. 5. Копіювати α $\alpha3 := \alpha$. 6. Обчислення $T_H = B_H + \alpha3 \cdot C_H$. 7. Сигнал про завершення обчислення T_H 8. Копіювати T: $T3 := T$. 9. Обчислення $a3 = \min(T3 * MD_H),$ $a = \min(a3, a)$ 10. Сигнал про завершення обчислень

3 Етап. Розробка структурної схеми взаємодії потоків

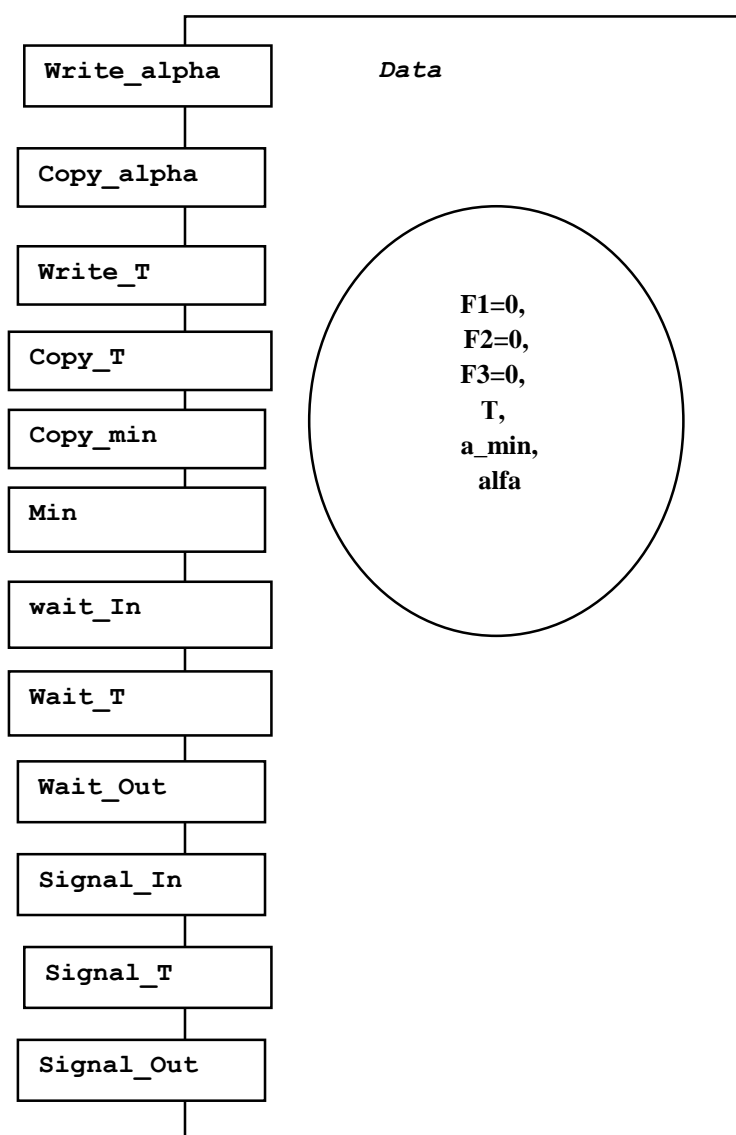


Рис 4.2 Структура класу *Data*

4 Этап. Розробка програми

```
public class Data {

    int N=900;
    int H=N/3;

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

    private int F1=0, F2=0, F3=0;
    private int a_min=10000000, alpha=0;
    private int T[]= new int[N];

    public synchronized void Write_alpha (int alpha3){
        alpha=alpha3; }

    public synchronized int Copy_alpha (){
        return alpha; }

    public synchronized void Write_T (int T_[]){
        for (int i=0; i<N; i++) {
            T[i]=T_[i]; }
    }
    public synchronized int[] Copy_T (){
        return T;
    }
    public synchronized int Copy_min (){
        return a_min; }

    public synchronized void Min (int a){
        if (a_min>a) a_min=a; }

    public synchronized void wait_In() {
        try{
            if (F1<2) wait();
        }
        catch (Exception e) { System.out.println(e);
        } }
    public synchronized void Wait_T(){
        try{
            if (F2<2) wait();
        }
        catch (Exception e)
        {System.out.println(e);}
    }
    public synchronized void Wait_Out(){
        try{
            if (F3<2) wait();
        }
        catch (Exception e)
        {System.out.println(e);}
    }
}
```

```

    }
    public synchronized void Signal_In() {
        F1++; notify();
    }
    public synchronized void Signal_T() {
        F2++; notify();
    }
    public synchronized void Signal_Out() {
        F3++; notify();
    }
}
public class aThread extends Thread {
    int a1=+100000000, alpfa1=0;
    Data Z;
    public aThread (Data q){
        Z=q; }

    public void run() {

        for (int i=0; i<Z.N; i++) {
            Z.B[i]=1;
            Z.C[i]=1;
        }
        Z.Signal_In();
        Z.wait_In();

        alpfa1=Z.Copy_alpha();

        for (int i=0; i<Z.H; i++){
            Z.T1[i]=Z.C[i]+alpfa1*Z.B[i];
        }

        Z.Wait_T();

        int T1_1[]=new int [Z.N];

        Z.Write_T(Z.T1);

        T1_1 =Z.Copy_T();

        for (int j=0; j<Z.H; j++){
            Z.V[j]=0;
            for (int i=0; i<Z.H; i++){
                Z.V[j]=Z.V[j]+T1_1[i]*Z.MD[i][j];
            }
        }

        for (int i=0; i<Z.H; i++){
            if (Z.V[i]<a1) a1=Z.V[i];
        }
        Z.Min(a1);
        Z.Signal_Out();
    }
}

```

```

}
public class bThread extends Thread {
    int a2=+1000000000, alpha2=0, aa=0;
    Data Z;
    public bThread (Data q){
        Z=q; }

    public void run(){
        Z.wait_In();

        alpha2=Z.Copy_alpha();

        for (int i=Z.H; i<2*Z.H; i++){
            Z.T1[i]=Z.C[i]+alpha2*Z.B[i];
        }
        Z.Signal_T();
        int T1_2[]=new int [Z.N];

        T1_2 =Z.Copy_T();

        for (int j=Z.H; j<2*Z.H; j++){
            Z.V[j]=0;
            for (int i=0; i<Z.H; i++){
                Z.V[j]=Z.V[j]+T1_2[i]*Z.MD[i][j];
            } }

        for (int i=Z.H; i<2*Z.H; i++){
            if (Z.V[i]<a2) a2=Z.V[i];
        }
        Z.Min(a2);

        Z.Wait_Out();
        aa=Z.Copy_min();

        System.out.println(aa);
}
}

```

```

public class cThread extends Thread {
    Data Z;
    int a3=+1000000000, alpha=0, alpha3=0 ;
    public cThread (Data q){
        Z=q;
    }
    public void run(){

        for (int i=0; i<Z.N; i++)
            for(int j=0; j<Z.N; j++){
                Z.MD[i][j]=1; }
    }
}

```

```

        alpha=1;
        Z.Write_alpha(alpha);
        Z.Signal_In();
        Z.wait_In();
        alpha3=Z.Copy_alpha();

        for (int i=2*Z.H; i<Z.N; i++){
            Z.T1[i]=Z.C[i]+alpha3*Z.B[i];
        }
        Z.Signal_T();
        int T1_3[]=new int [Z.N];

        T1_3 =Z.Copy_T();

        for (int j=2*Z.H; j<Z.N; j++){
            Z.V[j]=0;
            for (int i=0; i<Z.H; i++){
                Z.V[j]=Z.V[j]+T1_3[i]*Z.MD[i][j];
            }
        }

        for (int i=2*Z.H; i<Z.N; i++){
            if (Z.V[i]<a3) a3=Z.V[i];
        }
        Z.Min(a3);
        Z.Signal_Out();
    }
}

public class MainClass {

    public static void main(String[] args) {

        Data D= new Data();

        aThread P1 =new aThread (D);
        bThread P2 =new bThread (D);
        cThread P3 =new cThread (D);

        P1.start();
        P2.start();
        P3.start();

    }

}

```

Завдання для самостійного виконання

Реалізувати обчислення за поданою у таблиці 2 додатку формулою у відповідній чотирипроцесорній ПОС зі спільною пам'яттю, використовуючи синхронізовані методи мови *Java*.

Розробити:

- а) паралельний алгоритм;
- б) алгоритми роботи процесів;
- в) структурну схему взаємодії задач;
- г) програму на мові *Java*.

Номери варіантів та відповідні їм номери завдань із таблиці 2 додатку подано у таблиці 4.1.

Таблиця 4.1

Номер варіант	Номер завдання	Номер варіанту	Номер завдання
1	11	11	1
2	12	12	2
3	13	13	3
4	14	14	4
5	15	15	5
6	16	16	6
7	17	17	7
8	18	18	8
9	19	19	9
10	20	20	10

Лабораторна робота №5

Тема: Монітори в мові C#

Мета роботи: Вивчення можливостей класу `System.Threading.Monitor` для роботи з потоками.

Теоретичні відомості

5.1 Синхронізація потоків в мові C#

У основу синхронізації в мові C# покладено поняття блокування, за допомогою якого організовується управління доступом до кодового блоку в об'єкті. Синхронізація організовується за допомогою ключового слова `lock`. Загальна форма блокування :

```
private object threadLock = new object();  
lock (threadLock)  
{      // код      },
```

де `threadLock` означає посилання на об'єкт, що синхронізується. Об'єкт, що блокується, не має бути загальнодоступним.

Оператор `lock` гарантує, що фрагмент коду, захищений блокуванням для цього об'єкту, використовуватиметься тільки в потоці, що має це блокування. Усі інші потоки блокуються до тих пір, поки блокування не буде знято. Блокування знімається після закінчення фрагмента коду, що захищається нею. Блокованим вважається такий об'єкт, який представляє ресурс, що синхронізується.

5.2 Клас `Interlocked`.

Клас `Interlocked` дозволяє створювати прості оператори для атомарних операцій зі змінними. Клас `Interlocked` надає методи, що дозволяють виконувати інкремент, декремент, обмін і зчитувати значення безпечним для потоків способом. Деякі методи класу `Interlocked` наведені в таблиці 5.1.

Таблиця 5.1. Методи класу `Interlocked`

Член	Призначення
<code>CompareExchange()</code>	Безпечно перевіряє два значення на еквівалентність. Якщо вони еквівалентні, замінює одне із значень на третє
<code>Decrement()</code>	Безпечно зменшує значення на 1

Exchange()	Безпечно змінює два значення місцями
Increment()	Безпечно збільшує значення на 1

Використовуючи клас **Interlocked**, код

```
public void AddOne()
{
    lock(myLockToken)
    {
        int Val++;
    }
}
```

Можна переписати наступним чином:

```
public void AddOne()
{int newVal = Interlocked.Increment(ref intVal); }
```

Метод `Increment()` не лише змінює значення вхідного параметра, але також повертає отримане нове значення:

5.3 Клас *Monitor*

Якщо потік *T* виконується в кодовому блоці `lock`, і йому потрібно доступ до ресурсу *R*, який тимчасово недоступний, можна тимчасово звільнити об'єкт і тим самим дати можливість виконуватися іншим потокам. Такий підхід ґрунтується на деякій формі повідомлення між потоками, яка організовується в *C#* за допомогою методів `Wait()`, `Pulse()` і `PulseAll()`.

Зауваження. Методи `Wait()`, `Pulse()` і `PulseAll()` визначені в класі `Monitor` і можуть викликатися тільки із заблокованого фрагмента блоку. Якщо викликаються з коду, що знаходиться за межами синхронізованого коду, наприклад поза блоком `lock`, то генерується виключення `SynchronizationLockException`.

Метод `Wait()`. Коли виконання потоку тимчасово заблоковане, він викликає метод `Wait()`. У результаті потік переходить в стан очікування, а блокування з відповідного об'єкту знімається, що дає можливість використати цей об'єкт в іншому потоці.

Дві найчастіше використовувані форми методу `Wait()`:

```
public static bool Wait(object obj)
    public static bool Wait(object obj, int
        мілісекунд_простою)
```

У першій формі очікування триває аж до повідомлення про звільнення об'єкту, а в другій формі — як до повідомлення про звільнення об'єкту, так і до вичерпування періоду часу, на який вказує кількість `мілісекунд_простою`. У обох формах `obj` означає об'єкт, звільнення якого очікується.

Методи `Pulse()` і `PulseAll()`. При виклику методу `Pulse()` поновлюється виконання першого потоку, що очікує своєї черги на отримання блокування. А виклик методу `PulseAll()` сигналізує про зняття блокування усіх очікуючих потоків.

Загальні форми методів `Pulse()` і `PulseAll()`:

```
public static void Pulse(object obj)
public static void PulseAll(object obj),
```

де `obj` означає об'єкт, що звільняється.

Типовий приклад виконання завдання.

Постановка задачі. Реалізувати обчислення за формулою $A = B + \alpha \cdot C$ у двопроцесорній системі зі спільною пам'яттю (рис. 5.1), використовуючи ключове слово `lock` та методи `Wait()`, `Pulse()`, `PulseAll()` класу `Monitor` для синхронізації потоків в мові `C#`.

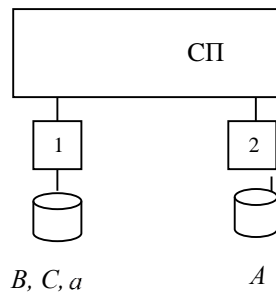


Рис. 5.1. Структурна схема двопроцесорної системи

1 Етап. Побудова паралельного алгоритму

$$A_H = B_H + \alpha \cdot C_H,$$

СП: α ;

$$H = \frac{N}{P}; P = 2.$$

2 Етап. Розробка алгоритмів роботи процесів

Clthread1	Clthread2
1. Ввід B, C, α . 2. Запис α . 3. Сигнал про завершення введення; 4. Копіювати α $\alpha1 := \alpha$. 5. Обчислення $A_H = B_H + \alpha1 \cdot C_H$. 6. Сигнал про завершення обчислення	1. Чекати сигнал про завершення введення. 2. Копіювати α . $\alpha2 := \alpha$. 3. Обчислення $A_H = B_H + \alpha2 \cdot C_H$. 4. Чекати сигнал про завершення обчислень 5. Вивід A .

3 Етап. Розробка структурної схеми взаємодії задач

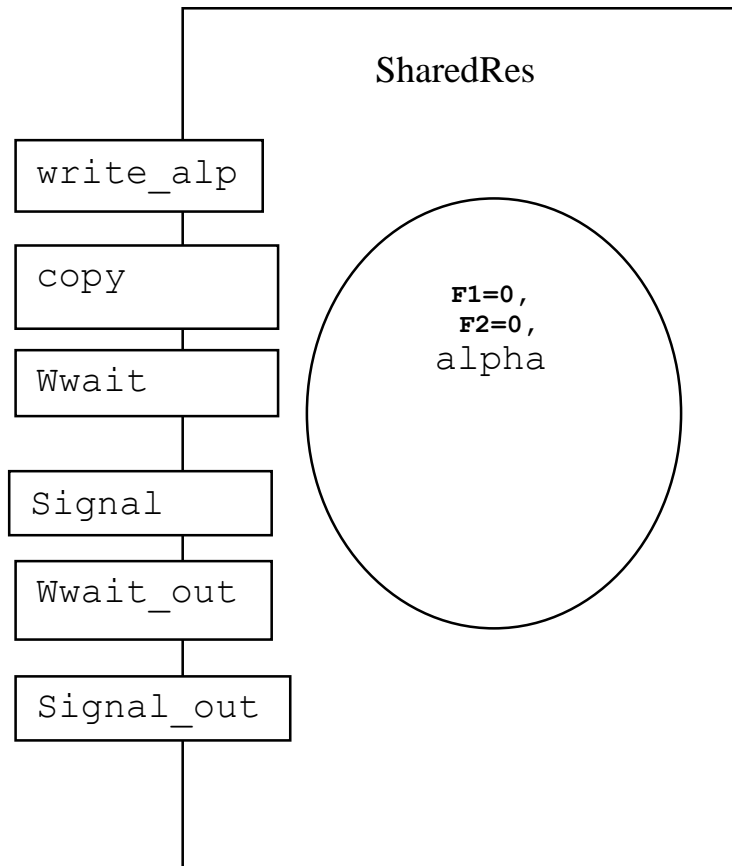


Рис. 5.2. Структура класу SharedRes

4 Етап. Розробка програми

```
class SharedRes
{
    public static int N = 10;
    public static int P = 2;
    public static int H = N / P;
    private int F1 = 0, F2 = 0;
    private object lockOn = new object();
    static private int alpha;
    public int[] B = new int[N];
    public int[] C = new int[N];
    public int[] A = new int[N];
    public static void write_alpha(int alpha_)
    {
        alpha = alpha_;
    }
    public int copy()
    {
        lock (lockOn)
```

```

        {
            return alpha;
        }
    }
    public void Wwait_out()
    {
        lock (lockOn)
        {
            if (F2 < 1) Monitor.Wait(lockOn);
        }
    }
    public void Signal_out()
    {
        lock (lockOn)
        {
            ++F2;
            if (F2 == 1) Monitor.Pulse(lockOn);
            else return;
        }
    }
    public void Wwait()
    {
        lock (lockOn)
        {
            if (F1 < 1) Monitor.Wait(lockOn);
        }
    }
    public void Signal()
    {
        lock (lockOn)
        {
            ++F1;
            if (F1 == 1) Monitor.Pulse(lockOn);
            else return;
        }
    }
}

class Clthread1
{
    public Thread thr1;
    SharedRes obj1;
    int alpha = 0;

    public Clthread1(string name, SharedRes obj)
    {
        thr1 = new Thread(this.Run);
        thr1.Name = name;
        obj1 = obj;
        thr1.Start();
    }
}

```

```

void Run()
{
    Console.WriteLine(thr1.Name + "   started");

    for (int i = 0; i < SharedRes.N; i++)
    {
        obj1.B[i] = 1;
        obj1.C[i] = 1;
    }
    alpha = 5;

    SharedRes.write_alpha(alpha);
    obj1.Signal();          // сигнал про завершения введения
    int alpha1 = obj1.copy();

    for (int i = 0; i < SharedRes.H; i++)

        obj1.A[i] = obj1.B[i] + alpha1 * obj1.C[i];

    obj1.Signal_out();
    Console.WriteLine(thr1.Name + " finished");
    Console.WriteLine();
}
}

class Clthread2
{
    public Thread thr2;
    SharedRes obj2;

    public Clthread2(string name, SharedRes obj)
    {
        thr2 = new Thread(new ThreadStart(this.Run));
        thr2.Name = name;
        obj2 = obj;
        thr2.Start();
    }

    void Run()
    {
        Console.WriteLine(thr2.Name + "   started");
        obj2.Wait();
        int alpha2 = obj2.copy();

        for (int i = SharedRes.H; i < SharedRes.N; i++)

            obj2.A[i] = obj2.B[i] + alpha2 * obj2.C[i];

        obj2.Wait_out();
    }
}

```

```

        for (int i = 0; i < SharedRes.N; i++)

        { Console.WriteLine(obj2.A[i]); }

        Console.WriteLine("second finished");
    }

}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Main started");
        SharedRes mon = new SharedRes();
        Clthread1 one = new Clthread1("first", mon);
        Clthread2 two = new Clthread2("second", mon);

        one.thr1.Join();
        two.thr2.Join();
        Console.WriteLine("Main finished");
        Console.ReadLine();
    }
}

```

Завдання для самостійного виконання

Реалізувати обчислення за поданою у таблиці 2 додатку формулою у відповідній чотирипроцесорній ПОС зі спільною пам'яттю на мові *C#*, використовуючи методи класу *Monitor*.

Розробити: а) паралельний алгоритм; б) алгоритми роботи процесів; в) структурну схему взаємодії потоків; г) програму на мові *C#*.

Номери варіантів та відповідні їм номери завдань із таблиці 2 додатку наведено у таблиці 5.1.

Таблиця 5.1

Номер варіант	Номер завдання	Номер варіанту	Номер завдання
1	2	11	13
2	12	12	11
3	1	13	14
4	3	14	15
5	4	15	16
6	5	16	17
7	10	17	18
8	7	18	19
9	8	19	20
10	9	20	6

СПИСОК ЛІТЕРАТУРИ

1. Богачев К.Ю. Основы параллельного программирования / К.Ю. Богачев — Изд-во лаборатория знаний, Бином, 2003. — 342 с.
2. Бэкон Дж. Операционные системы. Параллельные и распределенные системы / Дж. Бэкон, Т. Харрис — СПб: Питер; Киев: Издательская группа BHV, 2004. — 800с.
3. Воеводин В.В. Параллельные вычисления / В.В. Воеводин, Вл. В. Воеводин — Санкт-Петербург, "БХВ-Петербург", 2002 — 608 с.
4. Джехани Н. Язык Ада / Н. Джехани — М.: Мир, 1988.—552 с.
5. Жуков І.А. Паралельні та розподілені обчислення / І.А. Жуков, О.В. Корчкін — К.: „Корнійчук”, 2005. — 224с.
6. Корнеев В.Д. Параллельное программирование в MPI / В.Д. Корнеев — Новосибирск, 2002. — 215 с.
7. Таненбаум Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен — СПб.: Питер, 2003 — 877 с.
8. Хьюз К. Параллельное и распределённое программирование с использованием C++ / Камерон Хьюз, Треиси Хьюз — М.: Издательский дом "Вильямс", 2004 — 672 с .
9. Эндрюс Г.Р. Основы многопоточного параллельного и распределенного программирования / Г.Р. Эндрюс — М.: Изд. Дом Вильямс, 2003 — 330 с.

ДОДАТОК

Варіанти можливих матрично-векторних операцій та відповідні їм паралельні алгоритми:

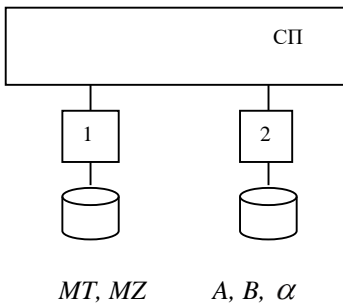
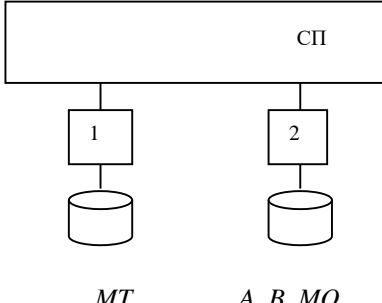
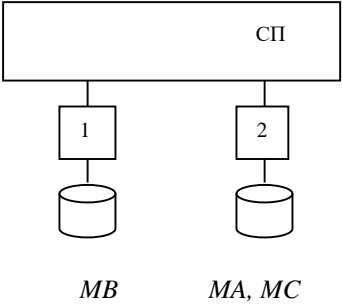
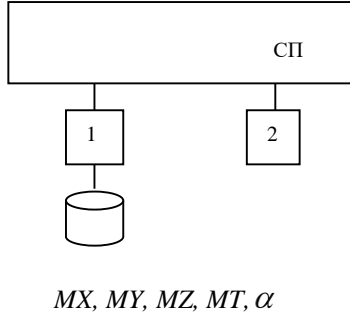
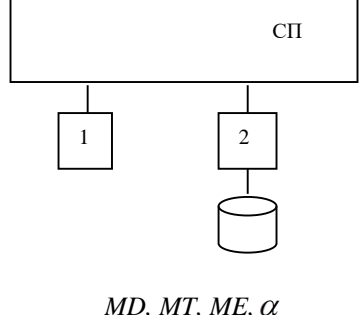
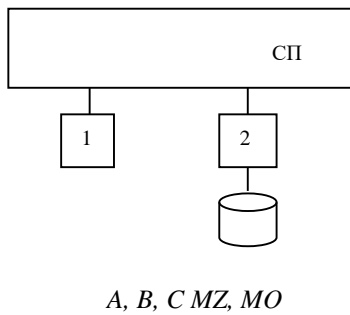
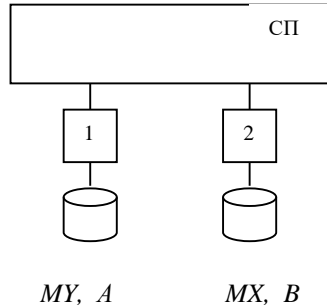
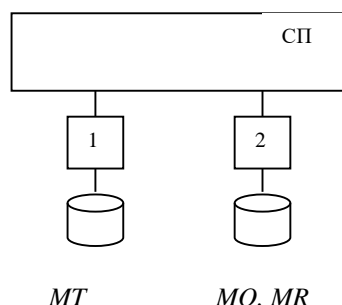
№	Задача	Паралельний алгоритм	Спільні ресурси (СР)	Примітки
1	$A = B + C$	$A_H = B_H + C_H$	—	B_H, C_H, A_H — H елементів відповідно векторів B, C, A
2	$A = B + \alpha \cdot C$	$A_H = B_H + \alpha \cdot C_H$	α	
3	$A = B \cdot MC$	$A_H = B \cdot MC_H$	B	MC_H, MZ_H, MA_H — H стовпців матриць MC, MZ, MA
4	$A = \alpha \cdot B + C \cdot MZ$	$A_H = \alpha \cdot B_H + C \cdot MZ_H$	α, C	
5	$MA = MB \cdot MC$	$MA_H = MB \cdot MC_H$	MB	
6	$MX = \alpha \cdot MY + \beta \cdot MZ \cdot MT$	$MX_H = \alpha \cdot MY_H + \beta \cdot MZ \cdot MT_H$	α, β, MZ	Позначення аналогічні попереднім випадкам (див, вище)
7	$A = (B \cdot MZ) \cdot MX$	$T_H = B \cdot MZ_H$ $A_H = T \cdot MX_H$	B, T	
8	$MA = MB \cdot (MC \cdot MZ)$	$MA_H = MB \cdot (MC \cdot MZ_H)$	MB, MC	

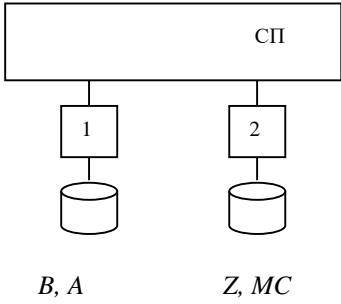
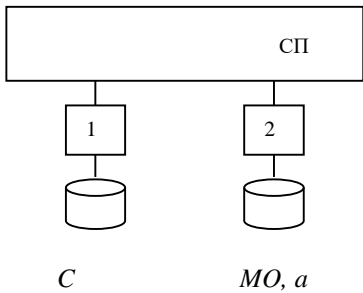
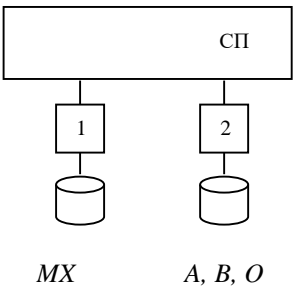
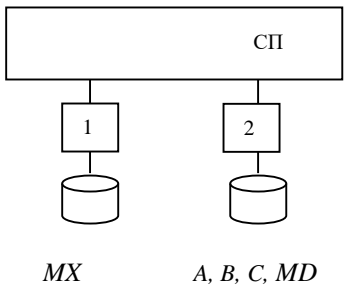
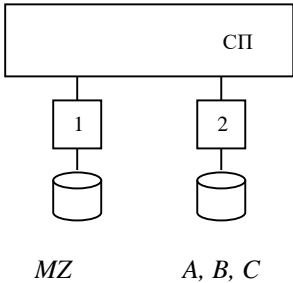
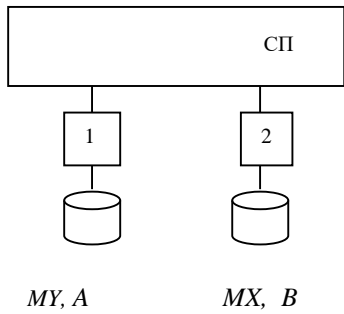
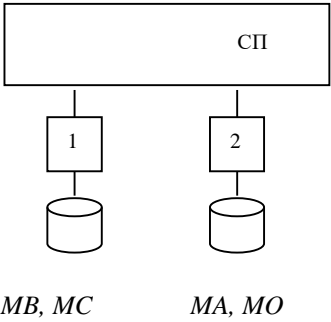
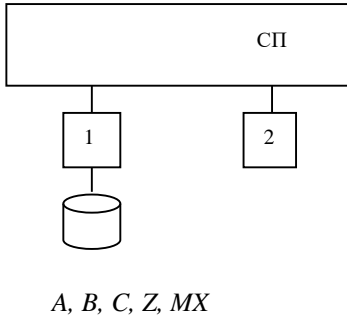
a, α, β, \dots — числа;

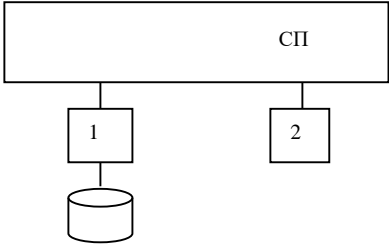
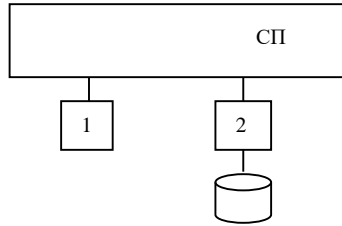
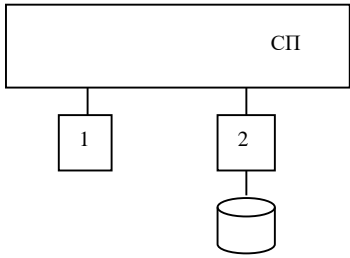
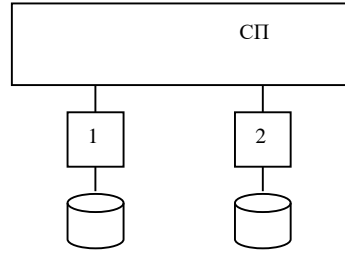
A, B, C, \dots — вектори;

$MA, MB, MC, MZ, MX, \dots$ — матриці

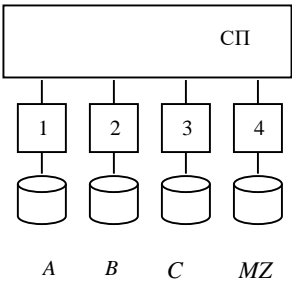
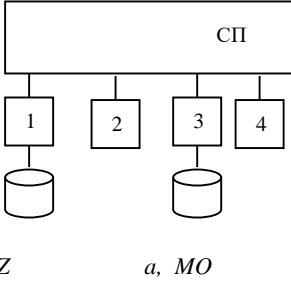
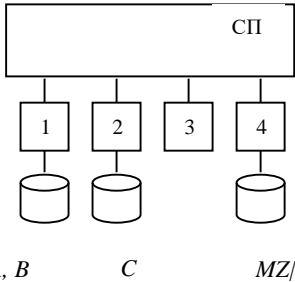
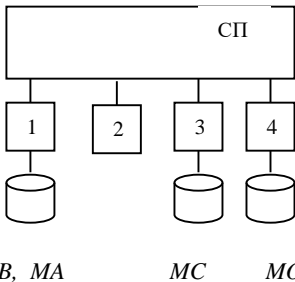
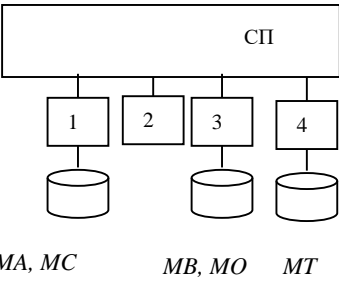
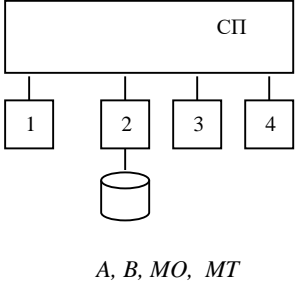
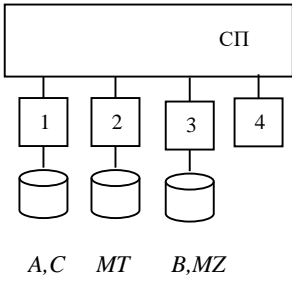
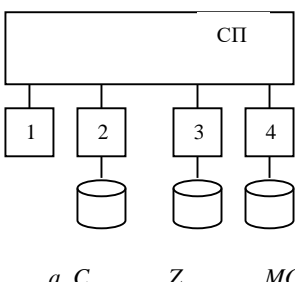
Таблиця 1. Завдання для лабораторних робіт №1 та №2

Номер завдання	Завдання	Номер завдання	Завдання
1	$A = B \cdot (MX + \alpha \cdot MZ)$  <p><i>MT, MZ A, B, α</i></p>	5	$A = B \cdot (MO \cdot MT)$  <p><i>MT A, B, MO</i></p>
2	$MA = MB \cdot MC$  <p><i>MB MA, MC</i></p>	6	$MX = MY + \alpha \cdot MZ \cdot MT$  <p><i>MX, MY, MZ, MT, α</i></p>
3	$MD = MT \cdot ME \cdot \alpha$  <p><i>MD, MT, ME, α</i></p>	7	$A = (B + C) \cdot (MZ + MO)$  <p><i>A, B, C, MZ, MO</i></p>
4	$A = B(MX \cdot MY)$  <p><i>MY, A MX, B</i></p>	8	$MT = MO \cdot MR$  <p><i>MT MO, MR</i></p>

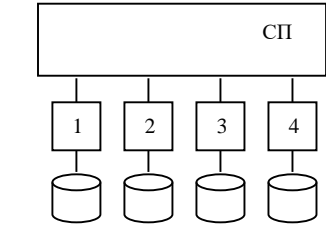
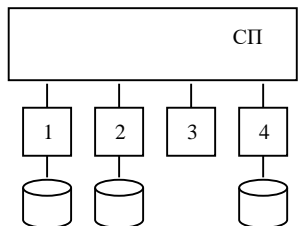
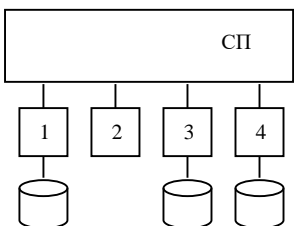
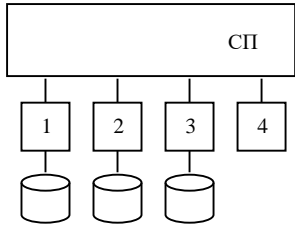
9	$A = B \cdot MC - Z$ 	13	$a = \min(MO) + C$ 
10	$A = B \cdot MX + O$ 	14	$A = B \cdot MX + C \cdot MD$ 
11	$A = B + C \cdot MZ$ 	15	$A = B \cdot (MX \cdot MY)$ 
12	$MA = MO \cdot \alpha + MB \cdot MC$ 	16	$A = B + C - Z \cdot MX$ 

17	$MA = MB \cdot MC - MO$  <p><i>MA, MB, MC, MO</i></p>	19	$A = B \cdot (MO + MR) - T$  <p><i>A, B, MO, MR, T</i></p>
18	$A = B \cdot (MX \cdot MZ)$  <p><i>A, B, MX, MZ</i></p>	20	$a = \min(MO \cdot MT)$  <p><i>a, MO</i> <i>MT</i></p>

Таблиця 2. Завдання для лабораторних робіт №3, №4 та №5

Номер завдання	Математична формула, структура ПОС	Номер завдання	Математична формула, структура ПОС
1	$A = B + C \cdot MZ$ 	5	$a = \max(MO \cdot MZ)$ 
2	$A = B \cdot MZ + C$ 	6	$MA = MB \cdot MC - MO$ 
3	$MA = MB \cdot MC + MO \cdot MT$ 	7	$A = B \cdot (MO \cdot MT)$ 
4	$A = B + C(MZ \cdot MT)$ 	8	$a = \max(C + Z \cdot MO)$ 

9	$A = B(MO \cdot MT)$ <p style="text-align: center;"><i>A B MO MT</i></p>	13	$MA = MB + MC \cdot MZ$ <p style="text-align: center;"><i>MA, MC MB MZ</i></p>
10	$MA = MB \cdot MC + MZ \cdot MO$ <p style="text-align: center;"><i>MA, MB, MC, MZ, MO</i></p>	14	$MA = \alpha \cdot MB \cdot MZ$ <p style="text-align: center;"><i>α , MZ MA, MB</i></p>
11	$A = B + C \cdot MZ$ <p style="text-align: center;"><i>A, B, C, MZ</i></p>	15	$MA = MB \cdot MC - MO$ <p style="text-align: center;"><i>MA, MB, MC, MO</i></p>
12	$MA = MB \cdot MC - a \cdot MT$ <p style="text-align: center;"><i>MA, MC MB, MT, a</i></p>	16	$A = B + C \cdot MZ$ <p style="text-align: center;"><i>A, B C MZ</i></p>

17	$MA = MB \cdot (MC + MO \cdot MX)$  <p style="text-align: center;"><i>MA, MB MC MO MX</i></p>	19	$A = B + C - Z \cdot MX$  <p style="text-align: center;"><i>A, B C, Z MX/</i></p>
18	$A = B + \alpha \cdot C - Z \cdot MT$  <p style="text-align: center;"><i>C, Z B, α MT, A</i></p>	20	$A = B \cdot (MO + MT)$  <p style="text-align: center;"><i>A, MT B MO</i></p>

ЗМІСТ

ВСТУП.....	3
Лабораторна робота № 1. Семафори в мові <i>Ada</i>	4
Лабораторна робота № 2. Семафори та м'ютекси мови <i>C#</i>	11
Лабораторна робота № 3. Захищені модулі мови <i>Ada</i>	18
Лабораторна робота № 4. Монітори в мові <i>Java</i>	26
Лабораторна робота № 5. Монітори в мові <i>C#</i>	37
СПИСОК ЛІТЕРАТУРИ.....	45
ДОДАТОК.....	46