Національний технічний університет України  
«Київський політехнічний інститут»

Кафедра обчислювальної техніки

(повна назва кафедри, циклової комісії)

Члени комісії

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

Студентки 3 курсу групи ІО-01

напряму підготовки 050102

«Комп’ютерна інженерія»

Заяць Ю.С.

(прізвище та ініціали)

Керівник доцент Корочкін О.В.

Національна оцінка

Кількість балів:

Оцінка: ECTS

**Курсова робота**

з дисципліни «Паралельні та розподілені обчислення»

(назва дисципліни)

на тему: «Розробка програмного забезпечення   
для паралельних комп’ютерних систем»

Київ – 2013 рік

Національний технічний університет України  
«Київський політехнічний інститут»

Факультет (інститут) інформатики та обчислювальної техніки

(повна назва)

Кафедра обчислювальної техніки

(повна назва)

Освітньо-кваліфікаційний рівень бакалавр

Напрям підготовки 6.050102 «Комп’ютерна інженерія»

(шифр і назва)

**Завдання**

на курсову роботу студенту

Заяць Юлія Степанівна

(прізвище, ім’я, по батькові)

1. Тема роботи «Розробка програмного забезпечення для паралельних комп’ютерних систем».
2. Керівник роботи к. т. н., доцент Корочкін Олександр Володимирович.

(вчене звання, науковий ступінь, прізвище та ініціали)

1. Строк подання студентом роботи 11 травня 2013 р.
2. Вихідні дані для роботи:

* порівняння реалізації механізму семафорів в мовах і бібліотеках паралельного програмування;
* математична задача ;
* структури паралельної комп’ютерної системи з спільною пам’яттю (ПКС СП) та паралельної комп’ютерної системи з локальною пам’яттю (ПКС ЛП);
* мови програмування: C#, Ada.Рандеву .

1. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):

* порівняння реалізації механізму семафорів в мовах і бібліотеках паралельного програмування;
* розробка і тестування програми ПРГ1 для ПКС СП;
* розробка і тестування програми ПРГ2 для ПКС ЛП.



Рисунок 1.1 – Структура ПКС СП



Рисунок 1.2 – Структура ПКС ЛП

1. Перелік графічного матеріалу:

* структурна схема ПКС СП;
* структурна схема ПКС ЛП;
* схеми алгоритмів процесів і головної програми для ПРГ1;
* схеми алгоритмів процесів і головної програми для ПРГ2.

1. Дата видачі завдання

КАЛЕНДАРНИЙ ПЛАН

|  |  |  |
| --- | --- | --- |
| № з/п | Назва етапів виконання КР | Строк виконання етапів КР |
|  | Виконання огляду для розділу 1 | 20.03.2013 |
|  | Розробка паралельного алгоритму рішення задачі | 01.04.2013 |
|  | Розробка алгоритмів процесів | 06.04.2013 |
|  | Розробка схем взаємодії процесів | 13.04.2013 |
|  | Розробка програм | 20.04.2013 |
|  | Тестування програм | 30.04.2013 |
|  | Оформлення КР | 10.05.2013 |
|  | Захист КР | 18.05.2013 |

Студент

(підпис) (прізвище та ініціали)

Керівник роботи

(підпис) (прізвище та ініціали)

**ПОЯСНЮВАЛЬНА  
ЗАПИСКА**

ЗМІСТ

[ВСТУП…… 5](#_Toc356215913)

[Розділ 1. Порівняння реалізації механізму семафорів в мовах і бібліотеках паралельного програмування. 6](#_Toc356215914)

[1.1 Вступ. 6](#_Toc356215915)

[1.2 Семафори. 7](#_Toc356215916)

[1.3 Реалізація механізму. 8](#_Toc356215917)

[1.3.1 Win32. 8](#_Toc356215918)

[1.3.2 POSIX. 9](#_Toc356215919)

[1.3.3 C#. 11](#_Toc356215920)

[1.3.4 Java. 14](#_Toc356215921)

[1.3.5 Ada. 16](#_Toc356215922)

[1.4 Висновки до Розділу 1. 20](#_Toc356215923)

[Розділ 2. Розробка програми ПРГ1 для ПКС СП. 21](#_Toc356215924)

[2.1 Розробка паралельного математичного алгоритму. 21](#_Toc356215927)

[2.2 Розробка алгоритмів процесів. 21](#_Toc356215928)

[2.3 Розробка схеми взаємодії процесів. 23](#_Toc356215929)

[2.4 Розробка програми ПРГ1. 25](#_Toc356215930)

[2.5 Тестування програми ПРГ 1. 26](#_Toc356215931)

[2.6 Висновки до Розділу 2. 31](#_Toc356215932)

[Розділ 3. Розробка програми ПРГ2 для ПКС ЛП. 32](#_Toc356215933)

[3.1 Розробка паралельного математичного алгоритму. 32](#_Toc356215935)

[3.2 Розробка алгоритмів процесів. 32](#_Toc356215936)

[3.3 Розробка схеми взаємодії процесів. 34](#_Toc356215937)

[3.4 Розробка програми ПРГ2. 36](#_Toc356215938)

[3.5 Тестування програми ПРГ 2. 36](#_Toc356215939)

[3.6 Висновки до Розділу 3. 41](#_Toc356215940)

[Основні результати і висновки по роботі. 43](#_Toc356215941)

[Список використаних джерел. 44](#_Toc356215942)

[ДОДАТКИ… 45](#_Toc356215943)

[Додаток А. Структурна схема ПКС СП. 46](#_Toc356215944)

[Додаток Б. Схема алгоритму головної програми зі зазначенням паралельних ділянок для ПРГ1. 48](#_Toc356215945)

[Додаток В. Схема алгоритму процесів для програми ПРГ1. 50](#_Toc356215946)

[Додаток Г. Лістинг програми ПРГ1. 52](#_Toc356215947)

[Додаток Д. Структурна схема ПКС ЛП. 57](#_Toc356215948)

[Додаток Е. Схема алгоритму головної програми зі зазначенням паралельних ділянок для ПРГ2. 59](#_Toc356215949)

[Додаток Ж. Схема алгоритму процесів для програми ПРГ2. 61](#_Toc356215950)

[Додаток З. Лістинг програми ПРГ2. 63](#_Toc356215951)

ВСТУП

Курсова робота по дисципліні «Паралельні і розподілені обчислення» складається з трьох розділів.

В першому розділі «Порівняння реалізації механізму семафорів в мовах і бібліотеках паралельного програмування» описаний механізм семафорів в паралельних програмах, а також його застосування в різних мовах та бібліотеках програмування.

Другий та третій розділи присвячені розробці програми для обчислення математичної задачі в паралельній комп’ютерній системі зі спільною та локальною пам’яттю відповідно. Програмне забезпечення для комп’ютерної системи зі спільною пам’яттю розроблено на мові C#, а для комп’ютерної системи з локальною пам’яттю – Аdа використовуючи механізм Рандеву. Проведено тестування отриманих програмних продуктів і зроблено висновки по їх ефективності.

Лістинги та алгоритми розроблених програм наведено у додатках.

# 

# Розділ 1. Порівняння реалізації механізму семафорів в мовах і бібліотеках паралельного програмування.

* 1. **Вступ.**

Синхронізація є основою паралельних програм, тому для розробки правильних протоколів синхронізації бажано мати спеціальні засоби, які можна використовувати для блокування призупинених процесів. Першим таким засобом синхронізації, який не втратив актуальності і сьогодні, стали *семафори*. Вони полегшують захист критичних секцій та можуть використовуватися систематично для реалізації планування і сигналізації. Крім того, семафори допускають різні способи реалізації, як за допомогою активного очікування, так і за допомогою ядра. Ідея семафора відповідно до назви взята з методу синхронізації руху поїздів, прийнятого на залізниці. Залізничний семафор - це "сигнальний прапорець", що показує, вільний шлях попереду або зайнятий іншим поїздом. У міру руху поїзда семафори встановлюються і скидаються. Семафор залишається встановленим на час, достатній, щоб при необхідності зупинити інший потяг. Таким чином, залізничні семафори можна розглядати як пристрої, які сигналізують про умови, щоб забезпечити взаємовиключне проходження поїздів по критичних ділянках шляху. Семафори в паралельних програмах аналогічні - вони надають базовий механізм сигналізації та використовуються для реалізації взаємного виключення і умовної синхронізації.

Повертаючись до історії. У середині 1960-х років Едсгер Дейкстра і п'ять його колег з Технічного університету Ейндховена (Нідерланди) розробили одну з перших мультипрограмних операційних систем. Ця система має елегантну структуру, що складається з ядра і рівнів віртуальних машин, реалізованих процесами. У ній були представлені семафори, які Дейкстра винайшов як корисний засіб реалізації взаємного виключення і вироблення сигналів про такі події, як переривання. Дейкстра також ввів термін *приватний семафор*.

Оскільки Дейкстра голландець, то назви операцій Р і V походять від голландських слів. Р - це перша буква голландського слова *passeren* (пропустити), а V-перша літера слова *vrijeven* (звільнити). Дейкстра і його група пізніше вирішили пов'язати букву Р зі словом *prolagen*, складеного з нідерландських слів *prolagen* (спробувати) та *ver1аgеn* (зменшити), а букву V-зі словом *verhogen* (збільшити). Приблизно в цей же час Дейкстра написав важливу роботу по взаємодії послідовних процесів. У цій роботі було показано, як використовувати семафори для вирішення різних завдань синхронізації. У своїй роботі по моніторам Тоні Хоар представив ідею розділеного двійкового семафора і показав, як його використовувати для реалізації моніторів. Однак саме Дейкстра пізніше, дав цьому методу назву і довів його практичність. Дейкстра описав використання розділених двійкових семафорів для вирішення задачі про читачів та письменників. Він також показав, як реалізувати звичайні семафори, використовуючи лише розділені двійкові семафори.

* 1. **Семафори.**

Механізм семафорів є одним із класичних засобів забезпечення цілісності поділюваних даних при зверненні до неподільного ресурсу. Семафори не допускають виконання деякої ділянки коду кількома задачами одночасно. Найпростіший семафор - замок (*lock*) або mutex (від англійського mutually exclusive, взаємовиключний). Для того щоб задача могла виконати критичну ділянку коду, вона повинна спочатку захопити замок. Після захоплення замку задача виконує критичну ділянку коду і потім звільнює замок, щоб інша задача могла його отримати і виконати критичну ділянку, яка охороняється замком, свого коду. Задача, зіткнувшись із зайнятим замком іншою задачею, звичайно чекає його звільнення.

Семафор - це особливий тип поділюваної змінної, яка обробляється лише двома неподільними операціями P() і V(). Семафор можна вважати екземпляром класу «семафор», а операції P() і V() - методами цього класу з додатковим атрибутом, котрий визначає їх неподільність.

Значення семафору - невід'ємне ціле число.

Операція V() використовується для сигналізації, що подія відбулася, тому вона збільшує значення семафора на одиницю.

Операція P() припиняє потік до моменту, коли станеться деяка подія, тому вона, дочекавшись, коли значення семафору стане позитивним, зменшує його значення на одиницю. Виконання операції P() не може бути призупинено.

* 1. **Реалізація механізму.**
     1. **Win32.**

По суті, механізм мьютекс - це одна з реалізацій семафору. Об'єкти Mutex знаходяться в дозволеному стані, якщо вони не належать жодному з потоків, і в забороненому стані, якщо будь-якої потік володіє об'єктом. В поточний момент часу лише одному з потоків може належати мьютекс, за рахунок чого і реалізуються функції синхронізації паралельних потоків. Наприклад, якщо існують два потоки, які виконують запис у розподілену пам'ять, то один з них захоплює цей об'єкт, виконує операції доступу до пам'яті, після чого звільняє мьютекс, тим самим дозволяючи іншому процесу виконати необхідні операції по доступу до вашої пам'яті. Потік може використовувати одну з функцій очікування для запиту на володіння об'єктом. У випадку, якщо мьютекс зайнятий іншим потоком, функція очікування блокує викликає потік до визволення об'єкта. Якщо потік вже володіє мьютексом, то виклик функцій очікування не призведе до тупику цього ж процесу.

У бібліотеці Win32 мьютекс необхідно створити за допомогою функції CreateMutex , надалі уже існуючий мьютекс можна захоплювати функцією OpenMutex та звільняти (для цього слугує функція ReleaseMutex ).

Семафор в реалізації Win32 визначається як лічильник, який приймає значення від нуля до певного значення. Лічильник зменшується на одиницю кожен раз, коли потік закінчує очікування семафору і виконує функцію очікування його і інкрементується при виклику функції ReleaseSemaphore. Якщо лічильник дорівнює нулю, то вважається, що він знаходиться у забороненому стані, і при виклику функцій очікування останні будуть блокувати процес до того моменту, поки один з потоків не збільшить значення семафору (інкрементує лічильник). Семафор застосовується для контролю над спільними ресурсами, доступ до яких дозволений певному числу користувачів. Лічильник семафору не може бути менше нуля (стан блокування) і більше максимального значення.

Функція використовується для створення об'єкта семафор у бібліотеці Win32:

*HANDLE CreateSemaphore (*

*LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes, / / вказівник на security attributes*

*LONG lInitialCount, / / початкове значення*

*LONG lMaximumCount, / / максимальне значення*

*LPCTSTR lpName / / вказівник на ім'я об'єкта*

*);*

Функція управління семафором в Win32 мають наступний вигляд: ReleaseSemaphore.

* + 1. **POSIX.**

Семафори лічильники ресурсів, що розділяються між потоками. Основні операції на семафори є: збільшуємо лічильник атомарно, і чекати, поки значення лічильника не дорівнює нуль, та зменшити його атомарно.

Семафори мають минулі максимальні значення, яке не можуть бути збільшені. Макрос SEM\_VALUE\_MAX визначимо, що це максимальне значення. У бібліотеці GNU C, SEM\_VALUE\_MAX є еквівалентним INT\_MAX, але це може бути набагато менше в інших системах.

Бібліотека Pthreads реалізує POSIX 1003.1b семафори. Її не слід плутати з System V семафорів (МПК, semctl і semop). Всі функції і макроси семафорів визначені в semaphore.h.

Функції:

*int sem\_init (sem\_t \*sem, int pshared, unsigned int value)*

Ініціалізує семафор об'єкта, на який вказує на sem. Кількість пов'язаних з семафором задається спочатку через value. Аргумент Pshared вказує, чи є семафор є локальним для поточного процесу (pshared дорівнює нулю), або повинен бути розділений між декількома процесами (pshared не дорівнює нулю).

У разі успіху sem\_init повертає 0. У разі невдачі повертає -1 та встановлює ERRNO до однієї з наступних значень: EINVAL - вартість перевищує максимальне значення лічильника; SEM\_VALUE\_MAX ENOSYS - pshared не дорівнює нулю. LinuxThreads в даний час не підтримують процес спільних семафорів. (Це в кінцевому рахунку зміниться.)

*int sem\_destroy (sem\_t \* sem)*

Руйнує об'єкт семафору, звільняючи ресурси, які він може утримувати. Якщо будь-який поток, які очікують на семафор, коли sem\_destroy викликається, вона не виконується івстановлює ERRNO в EBUSY.

У реалізації LinuxThreads, немає ресурсів, пов'язаних з семафором об'єктів, thussem\_destroy насправді нічого, крім перевірки того, що жоден потік очікує семафор. Це зміниться, коли процес спільного використання семафорів буде реалізований.

*int sem\_wait (sem\_t \* sem)*

Призупиняє виклик потіку, поки семафор, на який вказує sem  має ненульовий рахунок. Потім він атомарно зменшує лічильник семафора. sem\_wait є точкою скасування. Вона завжди повертає 0.

*int sem\_trywait (sem\_t \* sem)*

sem\_trywait не є варіантом блокування sem\_wait. Якщо семафор, на який вказує sem має не нульове значення, лічильник атомарно зменшується і sem\_trywait негайно повертає 0. Якщо лічильник семафора дорівнює нулю, sem\_trywait відразу повертає -1 та встановлює ERRNO в EAGAIN.

*int sem\_post (sem\_t \* sem)*

Атомарно збільшує лічильник семафора, на який вказує sem. Ця функція ніколи не блокується. sem\_post завжди успішний і повертає 0, якщо лічильник семафора буде перевищувати SEM\_VALUE\_MAX після того, як збільшиться. У цьому випадку sem\_post повертає -1 та встановлює ERRNO у EINVAL. Лічильник семафора залишається незмінним.

*int sem\_getvalue (sem\_t \* sem, int \* sval)*

sem\_getvalue зберігає в позиції, на яку вказує SVAL поточний лічильник семафора sem. Вона завжди повертає 0.

* + 1. **C#.**

Схема роботи з мютексами на мові C# абсолютно аналогічна схемі роботи з мютексами бібліотеки Win32 . Різниця полягає в використовуванні методів класу Mutex – конструктор та методи WaitOne () та ReleaseMutex (). Також у C# існують функції очікування декількох мьтексів (метод WaitAll).

Реалізація семафору у С# ідентична його реалізації у бібліотеці Win32.

Функція використовується для створення об'єкта семафор у бібліотеці C#:

*static Semaphore name = // ім’я семафору*

*new Semaphore( // конструктор класу*

*int init\_value, // початкове значення лічильника*

*int max\_value // максимальне значення семафору*

*);*

У C# функція управління семафором виглядає наступнис чином: Release.

Структура роботи семафорів в Win32 та C# однакова.

Ключове слово lock використовується для того, щоб виконання блоку коду не переривалося кодом, виконуваним в інших потоках. Для цього потрібно отримати взаємовиключне блокування для даного об'єкта на час тривалості блоку коду. Оператор lock починається з ключового слова lock, якому як аргумент вказується об'єкт, і за яким слідує блок коду, який повинен виконуватися одночасно тільки в одному потоці. Аргумент, що надається ключовим словом lock, повинен бути об'єктом на основі посилального типу; він використовується для визначення області блокування. Строго кажучи, об'єкт, що надається для lock, використовується тільки для того, щоб унікальним чином визначити ресурс, до якого надається доступ для різних потоків, тому це може бути довільний екземпляр класу. В дійсності цей об'єкт зазвичай представляє ресурс, для якого потрібна синхронізація потоків. Наприклад, якщо об'єкт контейнера повинен використовуватися в декількох потоках, то контейнер можна передати блокуванню, а блок синхронізованого коду після блокування повинен отримати доступ до контейнера. Якщо інші потоки блокуються для того ж контейнера перед доступом до нього, забезпечується безпечна синхронізація доступу до об'єкта.

Як правило, рекомендується уникати блокування типу public або екземплярів об'єктів, якими не управляє код вашого застосунку. Наприклад, використання lock (this) може призвести до несправності, якщо екземпляру дозволений відкритий доступ, оскільки зовнішній код також може блокувати об'єкт. Це може призвести до створення ситуацій взаємного блокування, коли два або кілька потоків будуть чекати вивільнення одного і того ж об'єкта. З цієї ж причини блокування відкритого типу даних (на відміну від об'єктів) може призвести до несправності. Блокування строкових літералів найбільш небезпечна, оскільки рядкові літерали інтернується середовищем CLR. Це означає, що якщо у всій програмі є один екземпляр будь-якого строкового літерала, точно такий же об'єкт буде представляти літерал у всіх запущених доменах додатки і у всіх потоках. В результаті блокування, включена для рядка з однаковим вмістом у всьому додатку, блокує всі екземпляри цього рядка в додатку. З цієї причини краще використовувати блокування закритих або захищених членів, для яких інтернування не застосовується. У деяких класах є члени, спеціально призначені для блокування. Наприклад, в типі Array є SyncRoot. У багатьох типах колекцій є член SyncRoot.

Приклад:

*Class Example{*

*Static object LC = new object ();*

*Static int x1;*

*Static void proc\_example() {*

*Lock (LC) {*

*x1 = 0;*

*}*

*}*

*}*

* + 1. **Java.**

Семафори часто розуміється неправильно і є недовикористованим інструментом для обмеження доступу до ресурсів. Вони ігнорують інші способи контролю доступу до ресурсів. Але семафори дають набір інструментів, який виходить за рамки нормальної синхронізації та інших інструментів. В Java клас Semaphore введено з версії 1.5, який розташований в пакеті java.util.concurrent.

Злічуваний семафор. Концептуально, семафор підтримує набір дозволів. Кожне acquire() захоплює блок, якщо необхідно, поки доступ відкритий, а потім забирає його. Кожен release() додає доступ, потенційно звільнюючи блокуючий потік. Однак жодне блокування об'єктів не використовуються; Семафор просто продовжує підрахунок кількості доступних об'єктів і діє відповідно.

Семафори часто використовуються для обмеження кількості потоків, які можуть отримати доступ до деяких (фізичних або логічних) ресурсів.

Конструктор:

*public Semaphore(int permits,*

*boolean fair)*

Створюється семафор із заданим числом дозволів і справедливість даного параметра. Параметри: *permits* - початкова кількість доступних дозволів. Це значення може бути негативним, у цьому випадку реліз повинен відбутися до того, як доступ буде наданий. *fair* - вірно, якщо семафор буде гарантувати умову FIFO надання дозволів відповідно до твердження, інше помилково.

Методи:

*public void acquire()*

*throws* [*InterruptedException*](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/InterruptedException.html)

Отримує дозвіл від цього семафора, блокування, поки один доступні, або потік переривається. Отримує дозвіл, якщо такий є, і негайно повертається, зменшуючи кількість доступних дозволів на один. Якщо немає дозволу доступу, то поточний потік стає недоступним для планування цілей потоку і очікує, доки один з двох випадків не виконається: деякий інший потік викликає метод Release ()для цього семафора і поточний потік стане наступним у призначені дозволу, або деякий інший потік перериватиме поточний потік.

*public void release()*

Повертає дозвіл, повернувши його в семафор. Повертає дозвіл, збільшуючи кількість доступних дозволів на один. Якщо який-небудь потік, який намагаються отримати дозвіл, то вибирається та надається дозвіл, який тільки що був повернений. Не потрібно умов, щоб потік, який повертає дозвіл, повинен був отримати цей дозвіл за допомогою callingacquire(). Правильне використання семафора встановлюється за допомогою програмування конвенції в додатку.

*public int availablePermits()*

Повертає поточну кількість дозволів, доступних в цьому семафорі. Цей метод зазвичай використовується для налагодження і тестування. Повертає кількість дозволів, доступних в цьому семафорі.

*public int drainPermits()*

Отримати і повернути всі дозволи, які відразу ж стають доступними. Повертає кількість дозволів.

*protected void reducePermits(int reduction)*

Зменшує кількість доступних дозволів на зазначений зменшення. Цей метод може бути корисним у підкласах, які використовують семафори для відстеження ресурсів, які стають недоступними. Цей метод відрізняється від *fromacquire* тим, що він не блокує очікування дозволу, щоб стати доступними.

* + 1. **Ada.**

Одним із способів запобігання одночасного доступу декількох потоків до ресурсів – це асоціювати семафор з ресурсом. Потік, який вимагає ексклюзивний доступ до ресурсу захоплює семафор, що змушує потік чекати, поки потоки попереду нього в черзі звільнять семафор. Коли потік завершить роботу з ресурсом, він звільняє семафор, дозволяючи іншим очікуючим потокам продовжити.

Реалізація

Тип семафору дуже простий та має необхідність в лише парі операцій захоплення і звільнення:

*package Binary\_Semaphores is*

*type Semaphore\_Type is limited private;*

*procedure Seize (Semaphore : in out Semaphore\_Type);*

*procedure Release (Semaphore : in out Semaphore\_Type);*

*...*

*end Binary\_Semaphores;*

Повний вигляд типу реалізований, як захищений тип:

*package Binary\_Semaphores is*

*...*

*private*

*protected type Semaphore\_Type is*

*procedure Release;*

*entry Seize;*

*private*

*In\_Use : Boolean := False;*

*end Semaphore\_Type;*

*end Binary\_Semaphores;*

Порівнювати часткову і повну декларації типу обережно. Частковий вид оголошений як обмежені приватні: *type Semaphore\_Type is limited private;* в той час як повний вид реалізований у вигляді захищеного типу: *protected type Semaphore\_Type is ...;*

Захищений тип оголошує Release як захищену процедуру, яка означає, що незважаючи на те що виклики синхронізовані, фактично не відбувається блокування.

Тим не менш, Seize оголошена як захищений елемент, тому що ми хочемо блокувати потоки, якщо ресурс вже використовується. Єдиний стан семафору має флажок, який вказує коли ресурс знаходиться у використанні. Ми використовуємо флажок в якості бар'єру для Seize:

*entry Seize when not In\_Use is*

*begin*

*In\_Use := True;*

*end;*

Якщо ресурс In\_Use, бар'єр стає помилковим, так що потоки блокуються

до зміни стану бар'єру. Коли це відбудеться, тіло Seize виконується, установка прапора повертається до істинного (який блокує інші потоки). Захищена процедура Release просто встановлює In\_Use фляжок назад помилковим:

*procedure Release is*

*begin*

*In\_Use := False;*

*end;*

Це відразу змушує бар'єр для Seize, щоб бути переоцінені, дозволяючи наступнону потоку продовжувати. Основна проблема, з семафором в тому, що це дуже легко, щоб не випустити семафор, коли ви завершите з ресурсом. Це може привести до взаємного блокування всієї системи, тому що кожен потік, який намагається захопити ресурс буде заблокованим назавжди, в очікуванні звільнення, якого ніколи не відбудеться.

Для того, щоб зробити семафор безпечним, нам потрібен спосіб, щоб переконатися, що він звільняється незалежно від того, як структура завершується. Рішення полягає у використанні "виділення ресурсів ініціалізації". Ми використовуємо контрольований об'єкту, щоб захопити семафор під час ініціалізації та звільнення семафора при його остаточному доопрацюванні. Об'єкт управління семафором зв'язується з семафором за допомогою доступу дискримінанта:

*type Semaphore\_Control (Semaphore : access Semaphore\_Type) is*

*limited private;*

Тип реалізований, як вивід, що Limited\_Controlled перевизначення Initialize та Finalize:

*type Semaphore\_Control (Semaphore : access Semaphore\_Type) is*

*new Limited\_Controlled with null record;*

*procedure Initialize (Control : in out Semaphore\_Control);*

*procedure Finalize (Control : in out Semaphore\_Control);*

Об'єкт управління реалізує Initialize, викликавши Seize, при вході в об’єкт семафору він зобов'язаний:

*procedure Initialize (Control : in out Semaphore\_Control) is*

*begin*

*Control.Semaphore.Seize;*

*end;*

Він реалізує Finalize, викликавши процедуру Release семафору:

*procedure Finalize (Control : in out Semaphore\_Control) is*

*begin*

*Control.Semaphore.Seize;*

*end;*

Операції Finalize для контрольованого об'єкта завжди викликається не

має зважаючи, як струкрура закінчується, тому це гарантує, що ресурс

буде випущений. Щоб використовувати семафор об'єкта управління, потрібно просто оголосити його:

*Resource : Resource\_Type;*

*Semaphore : aliased Semaphore\_Type;*

*procedure Op (...) is*

*Control : Semaphore\_Control (Semaphore'Access);*

*begin*

*<do op>*

*end;*

Операції семафору Seize і Release викликаються автоматично в процесі будівництва та реконструкції об'єкту Control:

*procedure Do\_Something (Resource : in out T) is*

*Control : Semaphore\_Control (Resource.Semaphore);*

*begin*

*...*

*if Done then*

*return;*

*end if;*

*...*

*end Do\_Something;*

Семафори можна використовувати для реалізації абстракцій, які доступні для декількох потоків одночасно. Семафор буде синхронізувати потоки, тому стан не може отримати пошкодження через чергування виконання. Щоб додати підтримку паралелізму, можна включити семафор, як частина

подання стека:

*type Stack\_Type is*

*limited record*

*Items : Item\_Array;*

*Top : Natural := 0;*

*Sema : aliased Semaphore\_Type;*

*end record;*

*end Stacks;*

Кожна операція з стеком здійснюється шляхом оголошення об'єкту управління семафором, і прив'язки його до семафору стека:

*procedure Push*

*(Item : in Item\_Type;*

*On : in out Stack\_Type) is*

*Control : Semaphore\_Control (On.Sema'Access);*

*begin*

*On.Top := On.Top + 1;*

*On.Items (On.Top) := Item;*

*end;*

* 1. **Висновки до Розділу 1.**

Семафори в паралельних програмах надають базовий механізм сигналізації та використовуються для реалізації взаємного виключення і умовної синхронізації. Вони полегшують захист критичних секцій та можуть використовуватися систематично для реалізації планування і сигналізації. Однією з реалізації семафору, або найпростішим семафором є замок або мьютекс.

Зробивши аналіз реалізації механізму семафорів у різних мовах програмування (C#, Java, Ada) та бібліотеках паралельного програмування (POSIX, Win 32), можна зробити висновок що сама організація роботи механізму дуже схожа. В більшості випадках відрізняються лише синтаксисом функцій, а також ініціалізацією.

# Розділ 2. Розробка програми ПРГ1 для ПКС СП.

2. 1. **Розробка паралельного математичного алгоритму.**

Паралельний математичний алгоритм можна подати у вигляді наступних етапів:

1. 
2. , 
3. , 
4. 

де:

– розмірність матриць і векторів;

– кількість ядер;

;

 –  рядків матриці ;

 –  рядків матриці ;

 –  елементів вектора ;

 –  елементів вектора ;

 –  рядків матриці ;

Спільні ресурси: .

* 1. **Розробка алгоритмів процесів.**

|  |  |  |
| --- | --- | --- |
|  | **Т1** | **КД і ТС** |
| 1. | Ввід MB. |  |
| 2. | Сигнал T2,T3,T4 про завершення вводу MB. | S11 |
| 3. | Чекати сигнали від T2, T3 про завершення вводу даних. | W21,W31 |
| 4. | Копія . | КД |
| 5. | Обчислення |  |
| 6. | Обчислення , |  |
| 7. | Сигнал T2,T3,T4 про завершення обчислення а. | S12 |
| 8. | Чекати сигнали від T2,T3,T4 про завершення обчислення а. | W22,W32,W42 |
| 9. | Копія . | КД |
| 10. | Обчислення |  |
| 11. | Сигнал T3 про завершення обчислення МА. | S13 |
|  | **Т2** | **КД і ТС** |
| 1. | Ввід MС, T. |  |
| 2. | Сигнал T1,T3,T4 про завершення вводу MС, T. | S21 |
| 3. | Чекати сигнали від T1, T3 про завершення вводу даних. | W11,W31 |
| 4. | Копія . | КД |
| 5. | Обчислення |  |
| 6. | Обчислення , |  |
| 7. | Сигнал T1,T3,T4 про завершення обчислення а. | S22 |
| 8. | Чекати сигнали від T1,T3,T4 про завершення обчислення а. | W12,W32,W42 |
| 9. | Копія . | КД |
| 10. | Обчислення |  |
| 11. | Сигнал T3 про завершення обчислення МА. | S23 |
|  | **Т3** | **КД і ТС** |
| 1. | Ввід Z. |  |
| 2. | Сигнал T1,T2,T4 про завершення вводу Z. | S31 |
| 3. | Чекати сигнали від T1, T2 про завершення вводу даних. | W11,W21 |
| 4. | Копія . | КД |
| 5. | Обчислення |  |
| 6. | Обчислення , |  |
| 7. | Сигнал T1,T2,T4 про завершення обчислення а. | S32 |
| 8. | Чекати сигнали від T1,T2,T4 про завершення обчислення а. | W12,W22,W42 |
| 9. | Копія . | КД |
| 10. | Обчислення |  |
| 11. | Чекати сигнали від T1,T2,T4 про завершення обчислення МА. | W33 |
| 12. | Вивід MА. |  |
|  | **Т4** | **КД і ТС** |
| 1. | Чекати сигнали від T1, T2, T3 про завершення вводу даних. | W11,W21,W31 |
| 2. | Копія . | КД |
| 3. | Обчислення |  |
| 4. | Обчислення , |  |
| 5. | Сигнал T1,T2,T3 про завершення обчислення а. | S42 |
| 6. | Чекати сигнали від T1,T2,T3 про завершення обчислення а. | W12,W22,W32 |
| 7. | Копія . | КД |
| 8. | Обчислення |  |
| 9. | Сигнал T3 про завершення обчислення МА. | S43 |

* 1. **Розробка схеми взаємодії процесів.**

На основі алгоритму, наведеному в попередньому пункті, була розроблена структурна схема взаємодії задач (рис.2.1). Схема дозволяє наочно контролювати зв’язки належних точок синхронізації (сигналів S та очікувань W). 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 і події, що будуть використовуватись в програмі.

Для демонстрації взаємодій між задачами вибрано всі задачі.

На структурній схемі взаємодії задач введено такі засоби для вирішення задачі взаємного виключення та синхронізації:

* критична секція *Mut* для керування доступом до спільного ресурсу ;
* критична секція *lockobj* для керування доступом до спільного ресурсу ;
* семафори *Sem1, Sem2, Sem3* для синхронізації із завершенням вводу в T1, Т2, Т3 відповідно;
* події *Ev1, Ev2, Ev4* для синхронізації задачі T3 із завершенням всіх обчислень в інших задачах;
* семафори *Sem11, Sem21, Sem31, Sem41* для синхронізації всіх задач із завершенням обчислення  в усіх задачах;

 Рисунок 2.1 – Структурна схема взаємодії задач для ПРГ1

* 1. **Розробка програми ПРГ1.**

Програма написана на мові C# і складається з одного модуля. Даний модуль містить п’ять функцій: *Main* – точка входу в програму, формує ідентифікатори задач, запускає задачі, а також вимірює час виконання програми ПРГ1 та *Т1, Т2, Т3, Т4* – кожна з цих функцій відповідно реалізує алгоритм, описаний в пункті 2.2**Ошибка! Источник ссылки не найден.**, а також об’явлені такі змінні: N – розмірність матриць, P – кількість процесорів. Засоби для розв’язку задачі взаємного виключення та синхронізації даного модуля описані в розділі 2.3.

Лістинг програми ПРГ1 наведено у додатку Г.

* 1. **Тестування програми ПРГ 1.**

Для тестування використовувалась паралельна обчислювальна система з наступними апаратними характеристиками:

* процесор: AMD Phenom II X6 1055T Processor(6 ядер по 2.8 Ггц, 6 МБ кешу третього рівня);
* оперативна пам'ять: DDR3 1333 МГц, 3327 МБ.

В якості програмного забезпечення виступали:

* операційна система: Microsoft Windows 7 x86 SP1 (Version 6.7.7601);
* середовище розробки і компіляції C# програми: Microsoft Visual Studio 2010.

Для вимірювання часу виконання програми використовувався таймер з допоміжного класу System.Diagnostics.Stopwatch мови програмування C#.

Для оцінки ефективності програми використовуються коефіцієнти прискорення та ефективності.

Коефіцієнт прискорення  показує скорочення часу виконання паралельної програми в паралельній системі з ** процесорами ** в порівнянні з часом виконання послідовної програми в однопроцесорній системі :



Коефіцієнт ефективності  застосування комп’ютерної системи показує ступінь використання ** процесорів системи:



Результати тестування і проведених досліджень ефективності розробленої програми наведено в табл. 2.1-2.3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **Т1, с** | **Т2, с** | **Т3, с** | **Т4, с** |
| **900** | 20.498 | 10.649 | 7.203 | 6.489 |
| **1800** | 158.764 | 79.792 | 58.837 | 40.564 |
| **2400** | 400.347 | 199.825 | 138.797 | 99.936 |

Таблиця 2.1 – Час виконання програми ПРГ1

На основі даних із таблиці 2.1 виконано розрахунок значень коефіцієнтів прискорення, які наведені в таблиці 2.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Кількість процесорів (P)** | | | |
| **N** | **1** | **2** | **3** | **4** |
| **900** | 1.000 | 1.925 | 2.846 | 3.159 |
| **1800** | 1.000 | 1.990 | 2.698 | 3.914 |
| **2400** | 1.000 | 2.003 | 2.884 | 4.006 |

Таблиця 2.2 – Коефіцієнти прискорення для програми ПРГ1

Коефіцієнти ефективності (таблиця 2.3) обчислено за даними таблиці 2.3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Кількість процесорів (P)** | | | |
| **N** | **1** | **2** | **3** | **4** |
| **900** | 100.00% | 96.24% | 94.86% | 78.97% |
| **1800** | 100.00% | 99.49% | 89.95% | 97.85% |
| **2400** | 100.00% | 100.00% | 96.15% | 100.00% |

Таблиця 2.3 – Коефіцієнти ефективності для програми ПРГ1

Використовуючи табл. 2.2‑2.3 побудовано графіки зміни коефіцієнтів прискорення і ефективності в залежності від *N* і .

Рисунок 2.2 – Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер при N=900

Рисунок 2.3 – Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер при N=1800

Рисунок 2.4 – Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер при N=2400

Рисунок 2.5 – Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер при N=900

Рисунок 2.6 – Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер при N=1800

Рисунок 2.7 – Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер при N=2400

* 1. **Висновки до Розділу 2.**

Слідуючи вимогам у технічному завданні була написана багатопотокова програма ПРГ1 для ПКС СП на високорівневій, об’єктно-орієнтованій мові програмування з безпечною системою типізації для платформи .NET – C#. В високопродуктивних обчислювальних системах зі спільною пам’яттю процеси організовують обмін спільними даними і виконують синхронізацію з іншими потоками за допомогою спеціальних засобів. Спільні ресурси оголошуються в якості глобальних змінних у даній багатопотоковій програмі. Для того, щоб процес міг отримати доступ до глобальних даних необхідно у програмі використовувати відповідні синхронізовані операції. Коли *і* - тий процес матиме доступ до певного спільного ресурсу в пам’яті, то всі решта процеси, при спробі отримання доступу до цього ж ресурсу, будуть заблоковані аж до звільнення цього спільного ресурсу іншим процесом який зайняв його. Під час цих дій виникає зниження швидкості виконання програми.

Мова програмування C# має власні вбудовані засоби синхронізації, тому не потрібно використовувати сторонні бібліотеки. Можемо використати такі засоби синхронізації : мьютекси, події, семафори, об’єкти блокування, а також волатайл змінні. Використовуючи об’єкти відповідних класів у мові С# вдалось безпроблемно, легко та ефективно організувати взаємодію між процесами. Описана програма забезпечує значно вищу швидкодію, ніж послідовна її альтернатива. Після тестування паралельної програми можна спостерігати наступні результати :

- максимальне значення коефіцієнту прискорення рівне 4.006. Цей результат забезпечує ПКС з Р=4 при N = 2400

- мінімальне значення коефіцієнту прискорення рівне 1.925. Цей результат забезпечує ПКС з Р=2 при N = 900

# Розділ 3. Розробка програми ПРГ2 для ПКС ЛП.

1. 1. **Розробка паралельного математичного алгоритму.**

Паралельний математичний алгоритм можна подати у вигляді наступних етапів:

1. 
2. 

де:

– розмірність матриць і векторів;

– кількість ядер;

;

 –  рядків матриці ;

 –  рядків матриці ;

 –  рядків матриці ;

Спільні ресурси: .

* 1. **Розробка алгоритмів процесів.**

|  |  |
| --- | --- |
|  | **Т1** |
| 1. | Ввід МВ. |
| 2. | Передати Т2 . |
| 3. | Прийняти від Т2 . |
| 4. | Прийняти від Т2 . |
| 5. | Обчислення . |
| 6. | Обчислення . |
| 7. | Передати Т2 . |
|  | **Т2** |
| 1. | Ввід МС, Т. |
| 2. | Прийняти від Т1 . |
| 3. | Передати Т3 . |
| 4. | Передати Т1 . |
| 5. | Передати Т3 . |
| 6. | Прийняти від Т3 . |
| 7. | Передати Т1 . |
| 8. | Обчислення . |
| 9. | Обчислення . |
| 10. | Прийняти від Т1 . |
| 11. | Передати Т3 . |
|  | **Т3** |
| 1. | Ввід Z. |
| 2. | Прийняти від Т2 . |
| 3. | Передати Т4 . |
| 4. | Прийняти від Т2 . |
| 5. | Передати Т4 . |
| 6. | Передати Т2 . |
| 7. | Передати Т4 . |
| 8. | Обчислення . |
| 9. | Обчислення . |
| 10. | Прийняти від Т2 . |
| 11. | Прийняти від Т4 . |
| 12. | Вивід результату . |
|  | **Т4** |
| 1. | Прийняти від Т3. |
| 2. | Передати Т5 . |
| 3. | Прийняти від Т3. |
| 4. | Передати Т5 . |
| 5. | Прийняти від Т3. |
| 6. | Передати Т5 . |
| 7. | Обчислення . |
| 8. | Обчислення . |
| 9. | Прийняти від Т5 . |
| 10. | Передати Т3 . |
|  | **Т5** |
| 1. | Прийняти від Т4. |
| 2. | Прийняти від Т4. |
| 3. | Прийняти від Т4. |
| 4. | Обчислення . |
| 5. | Обчислення . |
| 6. | Передати Т4 . |

* 1. **Розробка схеми взаємодії процесів.**

Задача 1 вводить матрицю МВ. Кожна частина матриці МВ буде неодмінно брати участь в обрахунках. Відповідно кожна задача отримує свою частину матриці та виконує пересилку інших частин далі. Задача 2 вводить матрицю МС та вектор Т. Для реалізації вказаної математичної функції кожна задача повинна мати свою власну копію для вектора Т та матриці МС. Для цього задача 2 надсилає задачі 1 копії вектора Т та матриці МС. Те саме відбувається і для задачі 3. Задача 3 в свою чергу отримує дані, копіює їх та надсилає наступній задачі. І так відбувається з усіма наступними задачами. Задача 3 вводить вектор Z. Далі дії відбуваються аналогічно як в задачі 2 тому що вектор Z необхідний всім задачам в повній мірі. Після закінчення пересилки та отримання всіх даних починається обрахунок. Кожна задача виконує обрахунок отриманих даних і отримує частину вихідної матриці. Після закінчення всіх обрахунків всі отримані результати надсилаються до задачі 3.

 Рисунок 3.1 – Структурна схема взаємодії задач для ПРГ2

* 1. **Розробка програми ПРГ2.**

Програма написана на мові Ada з використанням механізму Рандеву та складається з трьох модулів : основного модуля main.adb, опису допоміжного модуля data.ads та допоміжного модуля data.adb.

Основний модуль main.adb має тільки процедуру main в якій містяться оголошені задачі які виконують пересилку, відправлення повідомлень в залежності від номера задачі а також виклик процедур з допоміжного модуля. Для надсилання та приймання повідомлень використовуються входи (entry).

Допоміжний модуль, який використовується в основному модулі main.adb, містить в собі реалізацію алгоритмів множення векторів, матриць та ін. А модуль опису допоміжного модулю містить лише опис процедур реалізованих алгоритмів що дозволяю основному модулю main.adb безпроблемно використовувати реалізовані процедури з допоміжного модулю data.adb.

Лістинг програми ПРГ2 наведено у додатку З.

* 1. **Тестування програми ПРГ 2.**

Для тестування використовувалась паралельна обчислювальна система з наступними апаратними характеристиками:

* процесор: AMD Phenom II X6 1055T Processor(6 ядер по 2.8 Ггц, 6 МБ кешу третього рівня);
* оперативна пам'ять: DDR3 1333 МГц, 3327 МБ.

В якості програмного забезпечення виступали:

* операційна система: Microsoft Windows 7 x86 SP1 (Version 6.7.7601);
* середовище розробки і компіляції C# програми: Microsoft Visual Studio 2010.

Для вимірювання часу виконання програми використовувався функція Clock мови програмування Ада.

Для оцінки ефективності програми використовуються коефіцієнти прискорення та ефективності.

Коефіцієнт прискорення  показує скорочення часу виконання паралельної програми в паралельній системі з ** процесорами ** в порівнянні з часом виконання послідовної програми в однопроцесорній системі :



Коефіцієнт ефективності  застосування комп’ютерної системи показує ступінь використання ** процесорів системи:



Результати тестування і проведених досліджень ефективності розробленої програми наведено в табл. 3.1-3.3.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **N** | **Т1, с** | **Т2, с** | **Т3, с** | **Т4, с** | **Т5, с** |
| **900** | 26.015 | 13.650 | 9.971 | 7.985 | 5.934 |
| **1800** | 177.773 | 91.928 | 60.970 | 48.310 | 39.248 |
| **2400** | 463.937 | 235.910 | 193.880 | 121.275 | 99.210 |

Таблиця 3.1 – Час виконання програми ПРГ2

На основі даних із таблиці 3.1 виконано розрахунок значень коефіцієнтів прискорення, які наведені в таблиці 3.2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Кількість процесорів (P)** | | | | |
| **N** | **1** | **2** | **3** | **4** | **5** |
| **900** | 1.000 | 1.906 | 2.609 | 3.258 | 4.384 |
| **1800** | 1.000 | 1.934 | 2.916 | 3.680 | 4.529 |
| **2400** | 1.000 | 1.967 | 2.393 | 3.825 | 4.676 |

Таблиця 3.2 – Коефіцієнти прискорення для програми ПРГ2

Коефіцієнти ефективності (таблиця 3.3) обчислено за даними таблиці 3.2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Кількість процесорів (P)** | | | | |
| **N** | **1** | **2** | **3** | **4** | **5** |
| **900** | 100.00% | 95.29% | 86.97% | 81.45% | 87.68% |
| **1800** | 100.00% | 96.69% | 97.19% | 92.00% | 90.59% |
| **2400** | 100.00% | 98.33% | 79.76% | 95.64% | 93.53% |

Таблиця 3.3 – Коефіцієнти ефективності для програми ПРГ2

Використовуючи табл. 3.2‑3.3 побудовано графіки зміни коефіцієнтів прискорення і ефективності в залежності від *N* і .

Рисунок 3.2 – Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер при N=900

Рисунок 3.3 – Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер при N=1800

Рисунок 3.4 – Графік зміни коефіцієнту прискорення програми ПРГ1 в залежності від кількості ядер при N=2400

Рисунок 3.5 – Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер при N=900

Рисунок 3.6 – Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер при N=1800

Рисунок 3.7 – Графік зміни коефіцієнту ефективності програми ПРГ1 в залежності від кількості ядер при N=2400

* 1. **Висновки до Розділу 3.**

Згідно до технічного завдання була виконана реалізація програми ПРГ2 для ПКС ЛП використовуючи мову програмування Ada та механізм Рандеву. В системах з локальною пам’яттю процеси обмінюються повідомленнями за допомогою спеціальних засобів. Для надсилання та приймання повідомлень використовуються входи (entry). При цьому виникає зниження швидкості виконання програми.

Завдяки даному механізму вдалось легко організувати взаємодію між процесами. Вищеописана паралельна програма забезпечує значно вищу швидкість виконання, ніж послідовна програма. З отриманих результатів можна зробити наступні висновки :

* максимальне значення коефіцієнту прискорення рівне 4.676. Цей результат забезпечує ПКС з Р=5 при N = 2400.
* мінімальне значення коефіцієнту прискорення рівне 1.906. Цей результат забезпечує ПКС з Р=2 при N = 900.
* спостерігається явне зменшення часу виконання скомпільованого програмного коду при збільшенні кількості ядер в багатоядерній обчислювальній системі.
* середній час виконання програми реалізованої за допомогою високорівневої об’єктно-орієнтованої мови програмування С# менший ніж час виконання програми реалізованої за допомогою мови програмування Ada та механізму Рандеву.
* графіки коефіцієнтів прискорення виконання програм мають майже лінійний характер.

# Основні результати і висновки по роботі.

Проведено аналіз, огляд та детальний опис потужних засобів синхронізації – семафорів. Приведена історична довідка щодо потреб, виникнення та реалізації в різних мовах програмування. Наведені детальні приклади та вичерпна інформація щодо застосування цього засобу.

Розроблено алгоритми паралельних програм для багатоядерних обчислювальних систем, які вирішують поставлені математичні задачі з технічного завдання.

Розроблено структурні схеми паралельних програм, які показують взаємодію процесів у багатоядерній обчислювальній системі.

Розроблені програмні реалізації для поставлених математичних задач на високорівневій, об’єктно-орієнтованій мові програмування з безпечною системою типізації для платформи .NET – C# , а також на мові програмування Ada з використанням механізму Рандеву. Мову С# доцільно використовувати в багатоядерних обчислювальних системах з спільною пам’яттю, а мову Ada та механізм Рандеву доцільно використовувати в системах з локальною пам’яттю.

Результат виконання програм показав, що програма написана на мові C# має менший час виконання ніж програма написана на мові Ada.

# Список використаних джерел.

1. Карепова Е.Д., Кузьмин Д.А., Легалов А.И., Редькин А.В., Удалова Ю.В., Федоров Г.А. «СРЕДСТВА РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ», Учебное пособие, Красноярск 2007
2. Хьюз Камерон, Хьюз Трейси. Параллельное и распределенное программирование на C++.: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. – 672 с.
3. Грегори Р. Эндрюс - Основы многопоточного, параллельного и распределенного программирования
4. Корочкин А. В. Киев. Ада 95. Введение в программирование ,"Світ" 1998 ISBN 966-952-82-0-8
5. <http://www.csc.villanova.edu/~mdamian/threads/posixsem.html>
6. <http://bibliofond.ru/view.aspx?id=564475>
7. <http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Semaphore.html>
8. Жуков И.А., Корочкин А.В. Параллельные и распределенные вычисления. Лабораторный практикум. Учебно – методическое пособие. К.: «Корнейчук», 2008. – 224 с.
9. Нэш Трей. C# 2008. Ускоренный курс для профессионалов. Пер. с англ. – М.: ООО «И.Д. Вильямс». 2008. – 576.
10. Конспект лекцій
11. Жуков І.А., Корочкін О.В. Паралельні та розподілені обчислення: Навч. посібник [Текст]. – К.: Корнійчук, 2005. – 226 с. – ISBN 996-7599-36-1.

**ДОДАТКИ**

1. Структурна схема ПКС СП.

1. Схема алгоритму головної програми зі зазначенням паралельних ділянок для ПРГ1.

1. Схема алгоритму процесів для програми ПРГ1.

1. Структурна схема ПКС ЛП.

1. Схема алгоритму головної програми зі зазначенням паралельних ділянок для ПРГ2.

1. Схема алгоритму процесів для програми ПРГ2.