НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

\_\_\_\_\_\_\_\_\_\_Кафедра обчислювальної техніки\_\_\_\_\_\_\_\_\_\_\_\_

(повна назва кафедри, циклової комісії)

**КУРСОВИЙ ПРОЕКТ**

з дисципліни «Паралельне програмування»

(назва дисципліни)

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

Студента 3 курсу \_\_ІП-42\_\_ групи

напряму підготовки 050103 «Програмна інженерія»

\_ Водотійця Денис Ігоровчича \_\_\_\_\_\_\_\_\_

(прізвище та ініціали)

Керівник доцент Корочкін О.В.

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_\_\_\_\_\_\_

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

Київ- 2017 р.

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 6.050103 «Програмна інженерія»

# (шифр і назва)

## ***З А В Д А Н Н Я***

### НА КУРСОВИЙ ПРОЕКТ СТУДЕНТУ

\_\_\_\_\_\_\_\_\_\_Водотійцю Денису Ігоровичу\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я, по батькові)

1. Тема роботи «Розробка програмного забезпечення для паралельних

комп’ютерних систем»

керівник роботи Корочкін Олександр Володимирович к.т.н.**,** доцент

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

2. Строк подання студентом роботи 11 травня 2017 р.

3. Вхідні дані до роботи

* Математична задача МА = (B \* C) \* МO + max(Z) \* (MR \* MK)
* Структури ПКС ОП та ПКС ЛП
* Мови і бібліотеки програмування: Java

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

* порівняння реалізації механізму моніторів секції в мовах і бібліотеках паралельного програмування
* розробка і тестування програми ПРГ1 для ПКС ОП
* розробка і тестування програми ПРГ2 для ПКС ЛП

5. Перелік графічного матеріалу

* Структурна схема ПКС СП
* Структурна схема ПКС ЛП
* Структура класу TaskControl ПРГ1
* Структурна схема взаємодії задач ПГР2

7. Дата видачі завдання \_\_\_\_\_­\_2.03.2017\_\_\_\_\_

#### **КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |
| --- | --- | --- |
| №  з/п | Назва етапів виконання КР | Строк виконання етапів КР |
| 1 | Виконання розділу 1 | 13.03.2017 |
| 2 | Виконання розділу 2 | 03.04.2016 |
| 3 | Виконання розділу 3 | 24.04.2016 |
| 4 | Оформлення КР | 8.05.2016 |
| 5 | Перевірка КР викладачем | 11.05.2016 |
| 6 | Захист КР | 18.05.2016 |

**Студент \_\_\_\_\_\_\_\_\_ \_\_ Водотієць Д.І.\_\_\_\_**

( підпис ) (прізвище та ініціали)

**Керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_Корочкін О.В.\_\_\_\_\_\_**

( підпис ) (прізвище та ініціали)

ЗМІСТ

ВСТУП……………………………………………………………………………… 5

РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ З ПОТОКАМИ В МОВІ C#…………... 6

* 1. Потік ……………………………………….…………………………….. 6
  2. Порівняння потоків з процесами ………………………………………. 7
  3. Багатопотоковість……………………………………………………...... 7
  4. Взаємодія потоків ……………………………………………………….. 8
  5. Засоби роботи з потоками в мові C# …………………………………… 9
  6. Висновки до розділу 2 ………………………………………………….. 13

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС ОП……………………

* 1. Огляд паралельно обчислювальної системи…………………………..
  2. Розробка паралельного математичного алгоритму……………………
  3. Аналіз задачі з точки зору КНП………………………………………..
  4. Розробка алгоритмів процесів………………………………………….
  5. Розробка схеми взаємодії процесів…………………………………….
  6. Розробка програми ПРГ1……………………………………………….
  7. Тестування програми ПРГ1 …………………………………………….
  8. Висновки до розділу 2 …………………………………………………..

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМИ ПРГ2 ДЛЯ ПКС ЛП ……………………

* 1. Розробка паралельного математичного алгоритму………………….
  2. Розробка алгоритмів процесів…………………………………………
  3. Розробка схеми взаємодії процесів…………………………………..
  4. Розробка програми ПРГ2………………………………………………
  5. Тестування програми ПРГ2……………………………………………
  6. Висновки до розділу 3…………………………………………………

ОСНОВНІ РЕЗУЛЬТАТИ І ВИСНОВКИ ДО РОБОТИ……………………..

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ………………………………………

ДОДАТКИ…………………………………………………….............................

**ВСТУП**

Темою даної курсової роботи є «Розробка програмного забезпечення для паралельних комп’ютерних систем», що передбачає собою розробку програм для обчислення математичної задачі у паралельних комп’ютерних системах з загальною та локальною пам’яттю, а також розбір теоретичних відомостей за варіантом.

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

В першому розділі «Огляд засобів роботи з потоками в мові С#» розглянуті засоби роботи з потоками в мові C#.

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 Java. Проведено тестування отриманих програмних продуктів і зроблено висновки по їх ефективності.

Лістинги та алгоритми розроблених програм наведено у додатках.

**РОЗДІЛ 1. ОГЛЯД ЗАСОБІВ РОБОТИ З ПОТОКАМИ В МОВІ C#**

Листів

Лист

**1**

Літера

**6.050103**

**ІП-42**

**Розробка програмного забезпечення для паралельних комп’ютерних систем**

**НТУУ КПІ 17 4203 - 000 ПЗ**

Зм.

Виконав

Керівник

Конс.

Н. контр.

Зав. каф.

№ докум.

Водотієць Д.І.

Корочкін О.В.

Корочкін О.В.

Стіренко С.Г

Лит

Підпис

Дата

**1**

**1.1. Потік**

Потік[1] — в [інформатиці](https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) так називається спосіб програми розділити себе на дві чи більше паралельні задачі. Реалізація потоків та [процесів](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) відрізняються в різних [операційних системах](https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), але загалом потік міститься всередині процесу і різні потоки одного [процесу](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) спільно розподіляють деякі ресурси, в той час як різні [процеси](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) ресурси не розподіляють.

В системах з одним [процесором](https://uk.wikipedia.org/wiki/CPU) багатопотоковість реалізується загалом поділом часу виконання («кванти часу»), дуже подібно до паралельного виконання багатьох задач: процесор послідовно переключається між різними потоками. Це переключення контексту відбувається настільки швидко, що у кінцевого користувача створюється ілюзія одночасного виконання. На [багатопроцесорних](https://uk.wikipedia.org/w/index.php?title=%D0%91%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0&action=edit&redlink=1) чи на [багатоядерних системах](https://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D1%8F%D0%B4%D0%B5%D1%80%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) робота потоків здійснюється справді одночасно, оскільки різні потоки і [процеси](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) виконуються буквально одночасно різними процесорами або ядрами процесора.

Багато сучасних операційних систем прямо підтримують квантування часу і багатопроцесорну роботу потоків через [планувальник процесів](https://uk.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%BD%D1%83%D0%B2%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%96%D0%B2). [Ядро операційної системи](https://uk.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%BE%D1%97_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B8) дозволяє програмісту маніпулювати потоками через інтерфейс [системних викликів](https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%BD%D0%B8%D0%B9_%D0%B2%D0%B8%D0%BA%D0%BB%D0%B8%D0%BA). Деякі реалізації викликають потоки ядра, оскільки [легковагові процеси](https://uk.wikipedia.org/w/index.php?title=%D0%9B%D0%B5%D0%B3%D0%BA%D0%BE%D0%B2%D0%B0%D0%B3%D0%BE%D0%B2%D1%96_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%B8&action=edit&redlink=1) ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) lightweight process, LWP) є спеціальним типом потоків ядра, що розподіляють деякі стани та інформацію.

Поза тим, програма може емулювати роботу потоків, використовуючи таймер, сигнали або інші методи, щоб перервати власне виконання і послідовно виконувати різні задачі власним квантуванням часу. Такий спосіб іноді зветься потоками користувацького простору ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) user-space threads) або [волокнами](https://uk.wikipedia.org/wiki/%D0%92%D0%BE%D0%BB%D0%BE%D0%BA%D0%BD%D0%BE_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)).

**1.2. Порівняння потоків з процесами**

Потоки відрізняються від традиційних [процесів](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D0%BA%D0%BE%D0%BC%D0%BF%27%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%96_%D0%BD%D0%B0%D1%83%D0%BA%D0%B8)) [багатозадачних](https://uk.wikipedia.org/wiki/%D0%91%D0%B0%D0%B3%D0%B0%D1%82%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D1%96%D1%81%D1%82%D1%8C) [операційних систем](https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), в тому що [процеси](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)):

* на загал, незалежні;
* дублюють значну частину інформації про [стан](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B8_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%83);
* мають окремий [адресний простір](https://uk.wikipedia.org/w/index.php?title=%D0%90%D0%B4%D1%80%D0%B5%D1%81%D0%BD%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%96%D1%80&action=edit&redlink=1);
* взаємодіють тільки через системні [міжпроцесорні механізми комунікацій](https://uk.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D1%94%D0%BC%D0%BE%D0%B4%D1%96%D1%8F_%D0%BC%D1%96%D0%B6_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%B0%D0%BC%D0%B8).

Потоки всередині [процесу](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), з іншої сторони, розподіляють інформацію про стан [процесу](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), і прямо доступаються до спільної пам'яті та інших ресурсів. Переключення контексту між потоками процесу на загал швидше, ніж переключення контексту між процесами. Описуючи ситуацію такі системи, як [Windows NT](https://uk.wikipedia.org/wiki/Windows_NT) та [OS/2](https://uk.wikipedia.org/wiki/OS/2), кажуть, що мають «дешеві» потоки та «дорогі» процеси; в інших операційних системах ситуація не дуже відмінна.

**1.3. Багатопотоковість**

Багатопотоковість[2] ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *multi-threading*),  — властивість [операційної системи](https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D1%96%D0%B9%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) або [застосунку](https://uk.wikipedia.org/wiki/%D0%97%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA), яка полягає в тому, що [процес](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)), породжений в операційній системі, може складатися з кількох [нитей](https://uk.wikipedia.org/wiki/%D0%9D%D0%B8%D1%82%D1%8C), що виконуються паралельно, або навіть одночасно на багатопроцесорних системах. При виконанні деяких завдань таке розділення може досягти ефективнішого використання ресурсів [комп'ютера](https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%27%D1%8E%D1%82%D0%B5%D1%80).

Головною метою багатопотоковості є квазі-багатозадачність на рівні одного виконуваного процесу, тобто всі потоки виконуються в адресному просторі процесу. Окрім цього, всі потоки процесу мають не тільки спільний адресний простір, але і спільні дескриптори файлів. Процес, що виконується, має як мінімум однин (головний) потік.

Головні переваги в багатопотоковості:

* Спрощення програми в деяких випадках, за рахунок використання загального адресного простору;
* Менші відносно процесу часові витрати на створення ниті і взаємодію між нитями;
* Підвищення продуктивності процесу за рахунок розпаралелювання процесорних обчислень і операцій вводу/виводу.

**1.4. Взаємодія потоків**

У багатонитевому середовищі часто виникають проблеми, зв'язані з використанням паралельними виконуваними птоками одних і тих же даних або пристроїв. Для вирішення подібних проблем використовуються такі методи взаємодії потоків, як взаємовиключення ([м'ютекси](https://uk.wikipedia.org/wiki/%D0%9C%27%D1%8E%D1%82%D0%B5%D0%BA%D1%81)), [семафори](https://uk.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)), [критичні секції](https://uk.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%BD%D1%96_%D1%81%D0%B5%D0%BA%D1%86%D1%96%D1%97) і [події](https://uk.wikipedia.org/w/index.php?title=%D0%9F%D0%BE%D0%B4%D1%96%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)&action=edit&redlink=1).

* Взаємовиключення (mutex, м'ютекс) — це об'єкт синхронізації, який встановлюється в особливий сигнальний стан, коли не зайнятий жодним потоком. Тільки один поток володіє цим об'єктом у будь-який момент часу, звідси і назва таких об'єктів (від англійського mutually exclusive access — взаємно виключний доступ) — одночасний доступ до загального ресурсу виключається. Після всіх необхідних дій м'ютекс звільняється потоком, надаючи іншим потокам доступ до загального ресурсу.
* Семафори - є доступні ресурси, які можуть займатися кількома потоками в один і той же час, поки обсяг ресурсів не спустіє. Тоді додаткові потоки повинні чекати, поки необхідна кількість ресурсів не буде знову доступна. Семафори дуже ефективні, оскільки вони дозволяють одночасний доступ до ресурсів.
* Події - корисні в тих випадках, коли необхідно послати повідомлення потоку, що відбулося певна подія. Наприклад, при асинхронних операціях вводу/виводу з одного пристрою, система встановлює подію в сигнальний стан, коли закінчується якась з цих операцій. Один поток може використовувати кілька різних подій в декількох операціях, що перекриваються, а потім чекати приходу сигналу від будь-якого з них.
* Критичні секції забезпечують синхронізацію подібно м'ютексам за винятком того, що об'єкти, що представляють критичні секції, доступні в межах одного процесу.

Події, м'ютекси і семафори також можна використовувати в однопроцесному застосунку, проте критичні секції забезпечують швидший і ефективніший механізм синхронізації взаємного виключення.

**1.5. Засоби роботи з потоками в мові С#**

C # підтримує паралельне виконання коду через багатопоточність.

Програма на C# запускається як єдиний потік, автоматично створюваний CLR(Common Language Runtime - загальномовне виконуюче середовище) і операційною системою ("головний" потік), і стає багатопотоковою за допомогою створення додаткових потоків. Ось простий приклад і його висновок:

class ThreadTest

{

static void Main()

{

Thread thread = new Thread(Print);

thread.Start(); // Виконати Print() у новому потоці

while (true)

{

Console.Write("Hi!"); // Друкувати в косоль «Hi!»

}

}

static void Print()

{

while (true)

{

Console.Write("Hello, world!");

// Друкувати в косоль «Hello, world!»

}

}

}

У головному потоці створюється новий потік thread, виконуючий метод, який безперервно друкує «Hi!». Одночасно головний потік безперервно друкує «Hello, world!». CLR призначає кожному потоку свій стек, так що локальні змінні зберігаються окремо.

**Управління багатопоточністю** здійснює планувальник потоків, цю функцію CLR зазвичай делегує операційній системі[4]. Планувальник потоків гарантує, що активним потокам виділяється відповідний час на виконання, а потоки, які очікують або блоковані, наприклад, на очікуванні ексклюзивної блокування, або користувача введення - не споживають часу CPU.

На однопроцесорних комп'ютерах планувальник потоків використовує квантування часу - швидке перемикання між виконанням кожного з активних потоків. Це призводить до непередбачуваного поведінки, як в першому прикладі, де кожна послідовність «Hi!» і «Hello, world!» відповідає кванту часу, виділеного потоку. У

На багатопроцесорних комп'ютерах багатопоточність реалізована як суміш квантування часу і справжнього паралелізму, коли різні потоки виконують код на різних CPU. Необхідність квантування часу все одно залишається, тому що операційна система повинна обслуговувати як свої власні потоки, так і потоки інших додатків.

Кажуть, що потік витісняється, коли його виконання призупиняється через зовнішніх факторів типу квантування часу. У більшості випадків потік не може контролювати, коли і де він буде витіснений.

Для **створення потоків** використовується конструктор класу Thread, який приймає як параметр делегат типу ThreadStart, який вказує метод, який потрібно виконати. Делегат ThreadStart визначається так:

*public delegate void ThreadStart();*

Виклик методу Start() починає виконання потоку. Потік триває до виходу з виконуваного методу. Ось приклад, який використовує повний синтаксис C# для створення делегата ThreadStart:

*class ThreadTest*

*{*

*static void Main(){*

*Thread t = new Thread(new ThreadStart(Run));*

*t.Start(); // Виконати Run() в новому потоці*

*Run (); // Одночасно запустити Run() у головному потоці*

*}*

*static void Run ()*

*{*

*Console.WriteLine("Hello, world!");*

*}*

У цьому прикладі потік виконує метод Run () одночасно з головним потоком. Результат - два майже одночасних «Hello, world!»

Потік має властивість IsAlive, що повертає true після виклику Start () і до завершення потоку.

Потік, який закінчив виконання, не може бути початий знову.

Потоку можна задати ім’я, використовуючи властивість Name. Це надає велику зручність при налагодженні: імена потоків можна вивести в Console.WriteLine() і побачити у вікні Debug - Threads в Microsoft Visual Studio. Ім'я потоку може бути призначено в будь-який момент, але тільки один раз - при спробі змінити його буде згенеровано виключення.

Головному потоку також можна призначити ім'я - в наступному прикладі доступ до головного потоку здійснюється через статичну властивість CurrentThread класу Thread:

*class ThreadNaming*

*{*

*static void Main()*

*{*

*Thread.CurrentThread.Name = "main";*

*Thread worker = new Thread(Go);*

*worker.Name = "worker";*

*worker.Start();*

*Go();*

*}*

*static void Go()*

*{*

*Console.WriteLine("Hello from " + Thread.CurrentThread.Name);*

*}*

*}*

Властивість Priority визначає, скільки часу на виконання буде виділено потоку щодо інших потоків того ж процесу. Існує 5 градацій пріоритету потоку:

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

Значення пріоритету стає істотним, коли одночасно виконуються декілька потоків.

Установка пріоритету потоку на максимум ще не означає роботу в реальному часі (real-time), так як існують ще пріоритет процесу додатки. Щоб працювати в реальному часі, потрібно використовувати клас Process з простору імен System.Diagnostics для підняття пріоритету процесу:

Process.GetCurrentProcess (). PriorityClass = ProcessPriorityClass.High;

Від ProcessPriorityClass.High один крок до найвищого пріоритету процесу - Realtime. Встановлюючи пріоритет процесу в Realtime, ви говорите операційній системі, що хочете, щоб ваш процес ніколи не витіснявся. Якщо ваша програма випадково потрапить в нескінченний цикл, операційна система може бути повністю заблокована. Врятувати вас в цьому випадку зможе тільки кнопка вимкнення живлення. З цієї причини ProcessPriorityClass.High вважається максимальним пріоритетом процесу, придатним до вживання.

Якщо real-time програма має користувальницький інтерфейс, може бути не бажано піднімати пріоритет його процесу, так як оновлення екрану буде з'їдати занадто багато часу CPU - гальмуючи весь комп'ютер, особливо якщо UI досить складний. Зменшення пріоритету головного потоку в поєднанні з підвищенням пріоритету процесу гарантує, що real-time потік не буде витіснятися перемальовуванням екрану, але не рятує від гальм весь комп'ютер, тому що операційна система все ще буде виділяти багато часу CPU всьому процесу в цілому. Ідеальне рішення полягає в тому, щоб тримати роботу в реальному часі і користувальницький інтерфейс в різних процесах (з різними пріоритетами), що підтримують зв'язок через Remoting або shared memory.

**1.6. Висновки до розділу 1**

У даному розділі розглядалася багатопотоковість в мові програмування C. Був виконаний аналіз багатопотоковості в даній мові програмування. Короткий огляд даної мови багатопотокового програмування можна виконати за такими основними параметрами:

1. Створення потоку. Первинний (головний) потік створюється автоматично , решта - за допомогою класу Thread, якому в якості параметрів передається делегат типу ThreadStart, який вказує метод, який потрібно виконати. Виклик методу Start починає виконання потоку.

2. Засоби синхронізації потоків. У C # для синхронізації використовуються: lock, Mutex, Semaphore, EventWaitHandle, Wait and Pulse, Interlocked, volatile (для безпечного не блокує доступу до полів).

В C# існує стандартний набір функцій для повноцінної роботи багатопоточних систем. При виборі конкретного засобу програмування потрібно керуватися специфічними завданнями, які необхідно вирішити в даній задачі.

**РОЗДІЛ 2. РОЗРОБКА ПРОГРАМИ ПРГ1 ДЛЯ ПКС ОП**

**2.1. Огляд паралельної обчислюваної системи**

Листів

Лист

1

Літера

**6.050102**

**ІП-42**

**Розробка програмного забезпечення для паралельних комп’ютерних систем**

**НТУУ КПІ 17 4203 - 000 ПЗ**

Зм.

Виконав

Керівник

Конс.

Н. контр.

Зав. каф.

№ докум.

Кафтанатій Б.С.

Корочкін О.В.

Корочкін О.В.

Стіренко С.Г

Лит

Підпис

Дата

1

Дана паралельна комп’ютерна система, що складається з Р процесорів, двох пристроїв вводу/виводу та спільної пам’яті. Для даної комп’ютерної системи розробити програмне забезпечення для обчислення виразу:

МА = (B \* C) \* МO + max(Z) \* (MR \* MK)

Структурна схема ПКС СП наведена в додатку В. Вхідні та вихідні дані знаходяться на пристроях вводу/виводу так, як показано на схемі.

Мова для розробки програмного забезпечення – Java.

**2.2. Розробка паралельного математичного алгоритму**

1. a*i* = max(ZH), де *i* = ;

2. a = max(a, a*i*), s = s + si, *i* = ;

3. s­­i = BH \* CH, де *i* = ;

4. s = s + si, *i* = ;

5. MAH = s \* MOH + a \* (MR \* MKH) , де:

Н = N / P, P – кількість процесів,

MAH – Н рядків матриці MA,

MOH – Н рядків матриці MO,

MRH – Н рядків матриці MR,

MKH – Н рядків матриці MK,

Спільні ресурси: a, s, MR.

**2.3. Аналіз задачі з точки зору концепції необмеженого паралелізму (КНП)**

Для оцінки необхідного часу обчислень використаємо теорему Мунро-Петерсена, яка для комп’ютерної системи з необмеженим числом процесорів формулюється наступним чином: якщо виконується обчислення скалярної величини, яке потребує *m* бінарних операцій, то необхідний час обчислень tp:

tp ≥ [log2(m + 1)].

Для обчислення а = max(Z) необхідно виконати N бінарних операцій порівняння. Тому час виконання буде:

tp1 ≥ [log2(N + 1)].

Для обчислення одного елементу добутку матриць a \* MR \* MK необхідно виконати N + 1 множень та N – 1 операцій додавання. Час виконання:

tp2 ≥ [log2(2N + 1)].

Для обчислення одного елементу матриці (B \* C) \* MO, необхідно виконати  N  + 1 операцій множення та N – 1 операцій додавання. Тому час виконання буде:

tp3 ≥ [log2(2N + 1)].

Так як другий та третій етап незалежні, то вони можуть виконуватись паралельно. Тому сумарний час їх виконання буде рівний максимальному з двох:

tp2,3 ≥ max(tp2, tp3) = tp3 = tp2 = [log2(2N + 1)].

Для обчислення одного елементу суми матриць (B \* C) \* MO + a \* MR \* MK необхідно виконати одну операцію додавання. Час виконання:

tp4 ≥ [log2(2)] = 1.

Сумарний час виконання всіх етапів обчислень буде виражатись наступною формулою:

tp ≥ tp1 + tp2,3 + tp4 = [log2(N + 1)] + [log2(2N + 1)] + 1.

**2.4. Розробка алгоритмів процесів**

Так як розроблюване програмне забезпечення має бути масштабованим, тобто має працювати на системі з будь-якою кількістю процесорів, то зручним варіантом реалізації є написання єдиного алгоритму для всіх задач.

|  |  |
| --- | --- |
| **Крок алгоритму** | **ТС, КД** |
| 1. Якщо *tid = 0*, ввести MA, B, MR |  |
| 2. Якщо *tid = 0*, сигнал задачам *1…Р-1* про завершення *вводу 1*. | S*i,1* i = |
| 3. Якщо *tid = Р-1*, вести C, MO, Z, MK. |  |
| 4. Якщо *tid = Р-1*, сигнал задачам *0…Р-2* про завершення *вводу 2*. | S*i,2* i = |
| 5. Якщо *tid != 0*, чекати сигналу про завершення *вводу 1* від задачі *0*. | W*0,1* |
| 6. Якщо *tid != Р-1*, чекати сигналу про завершення *вводу 2* від задачі *Р-1* | W*P-1,2* |
| 7. Обчислення a*i* = max(Z*H*), si = BH \* CH |  |
| 8. Обчислення a = max(a, a*i*), s = s + si. | КД |
| 9. Сигнал всім задачам про завершення обчислення. | S*i,3* i = |
| 10. Чекати сигналів про завершення обчислення від всіх задач. | W*i3* i = |
| 11. Копія МR*i* = MR, s*i* = s, a*i* = a. | КД |
| 12. Обчислення MAH = si \* MOH + ai \* (MRi \* MKH). |  |
| 13. Якщо *tid != 0*, сигнал задачі *0* про завершення обчислень. | S*0,4* |
| 14. Якщо *tid = 0*, чекати сигналів про завершення обчислення від задач *1…Р-1* | W*i4* i = |
| 15. Якщо *tid = 0*, вивести *МА*. |  |

**2.5. Розробка схеми взаємодії процесів**

Під час виконання даного етапу була розроблена структура класу-монітора TaskControl. Він використовується для синхронізації паралельних потоків. На даному етапі визначався набір захищених елементів, що будуть знаходитись у моніторі, а також множина захищених операцій. Набір захищених елементів визначається множиною спільних ресурсів (див. паралельний математичний алгоритм) та множиною змінних, що використовуються в якості умов. Семантика захищених операцій обиралася виходячи з завдання мiнiмiзацiї кількості захищених операцій.

Клас TaskControl містить поля *а, MR, s* для зберігання відповідних спільних ресурсів, а також поля inputCount, calculationCount, preparationCount для організації умов виконання методів монітору. Структура класу TaskController наведена в додатку А. В класі містяться наступні методи:

– waitInput — для очікування введення даних в потоках T(0) та T(P-1);

– waitCalculation — для очікування закінчення обчислень;

– waitPreparation — для очікування закінчення пошуку максимального елемета вектора та добутку векторів;

– inputMR — для введення MО;

– getMR — для копіювання спільного ресурсу MR;

– getA — для копіювання спільного ресурсу a;

– getS — для копіювання спільного ресурсу s;

– setA — для виконання операцій над спільним ресурсом a;

– setS — для виконання операцій над спільним ресурсом s;

– signalInputDone — для сигналу про завершення вводу даних;

– signalCalculationDone — для сигналу про завершення обчислень;

– signalPreparationDone — для сигналу про завершення пошуку максимального елемента вектора та добутку векторів;

**2.6. Розробка програми ПРГ1**

Програма для системи з спільною пам’яттю написана на мові Java. Основні класи програми:

– Director — основний клас. Містить головний метод, що запускається JVM при старті програми. Головний метод формує ідентифікатори потоків, запускає потоки та вимірює час їх виконання. В основному класі знаходиться константа P, змінюючи яку можна виконати налаштування програми під конкретну комп’ютерну систему;

– TaskWorker — задачний тип, реалізує інтерфейс Runnable;

– TaskController — клас-монітор, який вирішує задачі синхронізації та взаємного виключення, а також зберігає спільні ресурси;

Повний лістинг програми наведено у додатку Д.

**2.7. Тестування програми ПРГ1**

Для тестування використовувалась паралельна обчислювальна система з наступним апаратним забезпеченням:

- процесор: Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz

- оперативна пам'ять: 6 Гб DDR3;

В якості програмного забезпечення використовувались:

- операційна система: Windows 10;

- компілятор та віртуальна машина Java: Sun Java 1.8.0\_92, 64-бітна версія.

Таблиця 2.1. Час виконання програми з спільною пам’яттю(значення в мілісекундах)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | T1 | T2 | T3 | T4 |
| 1000 | 10850 | 6584 | 5036 | 4503 |
| 1500 | 50941 | 30895 | 24761 | 20804 |
| 2000 | 146102 | 85492 | 72218 | 61645 |
| 2500 | 313007 | 193823 | 155367 | 135717 |

Підрахунок коефіцієнту прискорення (КП) виконується за формулою

КП = Т1 / ТР

Таблиця 2.2. Значення КП для програми з спільною пам’яттю

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Р | | | |
| 1 | 2 | 3 | 4 |
| 1000 | 1 | 1,64793439 | 2,1545 | 2,4095 |
| 1500 | 1 | 1,64884285 | 2,0573 | 2,4486 |
| 2000 | 1 | 1,70895522 | 2,0231 | 2,3701 |
| 2500 | 1 | 1,61491154 | 2,0146 | 2,3063 |

Підрахунок коефіцієнту ефективності (КЕ) відбувається за формулою

КЕ = КП / Р

Таблиця 2.3. Значення КЕ для програми зі спільною пам’яттю

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Р | | | |
| 1 | 2 | 3 | 4 |
| 1000 | 1 | 0,82397 | 0,7182 | 0,6024 |
| 1500 | 1 | 0,82442 | 0,6858 | 0,6122 |
| 2000 | 1 | 0,85448 | 0,6744 | 0,5925 |
| 2500 | 1 | 0,80746 | 0,6715 | 0,5766 |

Рисунок 2.1. Графік залежності КП від кількості процесорів при N = 1000

Рисунок 2.2. Графік залежності КП від кількості процесорів при N = 1500

Рисунок 2.3. Графік залежності КП від кількості процесорів при N = 2000

Рисунок 2.4. Графік залежності КП від кількості процесорів при N = 2500

Рисунок 2.5. Графік залежності КЕ від кількості процесорів при N = 1000

Рисунок 2.6. Графік залежності КЕ від кількості процесорів при N = 1500

Рисунок 2.7. Графік залежності КЕ від кількості процесорів при N = 2000

Рисунок 2.8. Графік залежності КЕ від кількості процесорів при N = 2500

**2.8. Висновки до розділу 2**

– У даному розділі досліджені результати тестування паралельної програми для системи з загальною пам’яттю, написаної на мові Java. Тестування проводилось для 2, 3 та 4 потоків. Для розрахунку коефіцієнтів прискорення та ефективності була протестована окремо створена послідовна програма.

– Коефіцієнт прискорення приймає значення у проміжку від 1.61 до 2.44. Найвищі значення цього коефіцієнту приходяться на систему з чотирма потоками, найменше прискорення отримане для системи з двома потоками. Характер графіків коефіцієнтів прискорення однаковий для систем з 2, 3 та 4 ядрами.

– При збільшенні кількості ядер P > 2 коефіцієнт прискорення зменшується. Це в основному пов’язано з тим, що пропускна здатність пам’яті ділиться між всіма ядрами та під час запитів до пам’яті ядра простоюють. Крім того, деякий час витрачається на синхронізацію.

– Результати дослідження мають похибку, зумовлену тим, що процесори виділяються операційною системою не на монопольне використання, тобто, процесорний час може бути в будь-який час передано сторонній програмі. Чим менший час виконання програми, тим більша вірогідність виникнення досить значущої похибки.

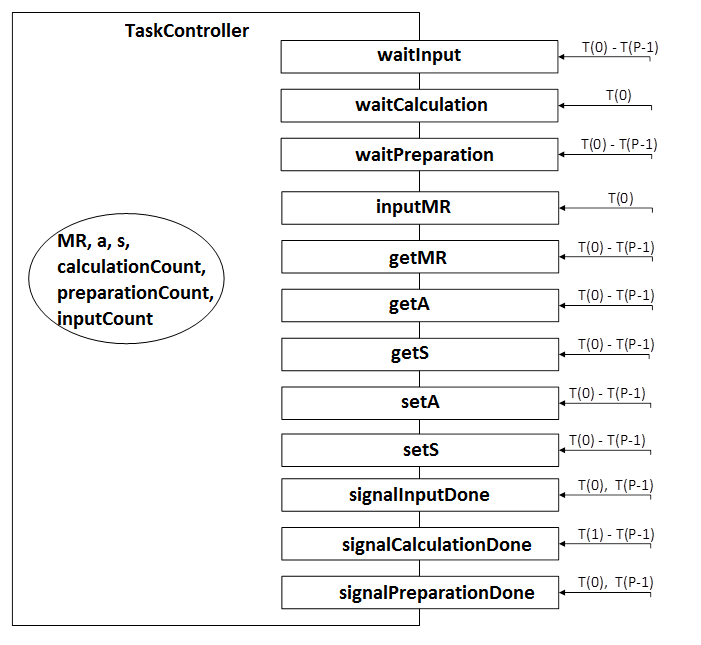
**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ**

1. Введение в многопоточность [Електронний ресурс] – Режим доступу: http://www.codenet.ru/progr/cpp/threads.php
2. Ars Technica article about multithreading, etc [Електронний ресурс] – Режим доступу: http://archive.arstechnica.com/paedia/images/m-hyperthreading-1.html
3. Багатопотоковість [Електронний ресурс] – Режим доступу: http://qps.ru/GWbLr
4. Работа с потоками в C# [Електронний ресурс] – Режим доступу: http://rsdn.ru/article/dotnet/CSThreading2.xml
5. Жуков И. А., Корочкін О. В. Паралельні та розподілені обчислення – К.: «Корнійчук», 2014. – 102-111 с.
6. Шильд.Г. Java 8 издание. Полное руководство. Москва-Санкт-Петербург-Киев. 2012.-280 с.
7. Корочкін О., Жужель М. Авдієв А., Корочкін Д. Захищений модуль як універсальний засіб синхронізації процесів – Вісн. НТУУ«КПІ», Інформатика, управління та обчислювальна техніка, 2001, К.: - №34, С. 137 – 145

**ДОДАТКИ**

**Додаток А**

**Структура класу TaskController ПРГ1**

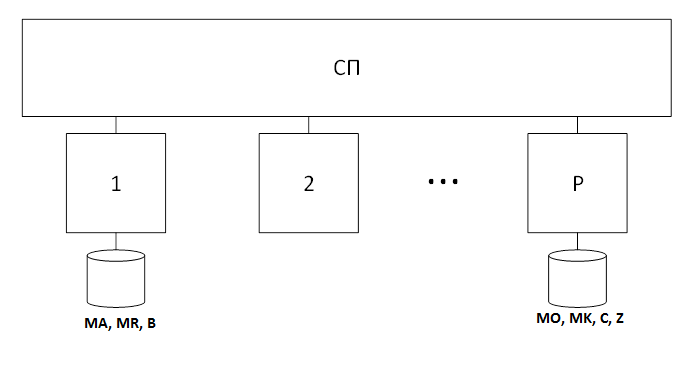


**Додаток Б**

**Структурна схема взаємодії задач ПГР2**

**Додаток В**

**Структурна схема ПКС СП**



**Додаток Г**

**Структурна схема ПК ЛП**

**Додаток Д**

**Лістинг програми ПРГ1**

Director.java

package com.vodotiiets.directors;  
  
import com.vodotiiets.controllers.TaskController;  
import com.vodotiiets.primitives.Matrix;  
import com.vodotiiets.primitives.Vector;  
import com.vodotiiets.workers.TaskWorker;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class Director {  
  
 private static int *N* = 2500;  
 private static int *P* = 1;  
 private static int *H* = *N* / *P*;  
  
 private static int *maxValue* = 5;  
 private static long *startTime*;  
 private static long *endTime*;  
  
 private static Matrix *MA* = new Matrix(*N*);  
 private static Matrix *MO* = new Matrix(*N*);  
 private static Matrix *MK* = new Matrix(*N*);  
  
 private static Vector *B* = new Vector(*N*);  
 private static Vector *C* = new Vector(*N*);  
 private static Vector *Z* = new Vector(*N*);  
  
 public static void main(String[] args) {  
  
 if(*P* > *N*) {  
 *P* = *N*;  
 *H* = 1;  
 }  
  
 Thread[] tasks = new Thread[*P*];  
 TaskController monitor = new TaskController();  
  
 for (int i = 0; i < *P*; i++) {  
 tasks[i] = new Thread(new TaskWorker(i, monitor));  
 }  
  
 *startTime* = System.*currentTimeMillis*();  
 for (int i = 0; i < *P*; i++) {  
 tasks[i].start();  
 }

}  
  
 public static int getN() {  
 return *N*;  
 }  
  
 public static int getP() {  
 return *P*;  
 }  
  
 public static int getH() {  
 return *H*;  
 }  
  
 public static int getMaxValue() {  
 return *maxValue*;  
 }  
  
 public static void setEndTime(long endTime) {  
 Director.*endTime* = endTime;  
 }  
  
 public static long getTime() {  
 return *endTime* - *startTime*;  
 }  
  
 public static Matrix getMA() {  
 return *MA*;  
 }  
  
 public static Matrix getMO() {  
 return *MO*;  
 }  
  
 public static Matrix getMK() {  
 return *MK*;  
 }  
  
 public static Vector getB() {  
 return *B*;  
 }  
  
 public static Vector getC() {  
 return *C*;  
 }  
  
 public static Vector getZ() {  
 return *Z*;  
 }  
}

TaskController.java

package com.vodotiiets.controllers;  
  
import com.vodotiiets.directors.Director;  
import com.vodotiiets.primitives.Matrix;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class TaskController {  
  
 private int inputCount;  
 private int preparationCount;  
 private int calculationCount;  
  
 private int a = Integer.*MIN\_VALUE*;  
 private int s;  
 private Matrix MR = new Matrix(Director.*getN*());  
  
 public synchronized void waitInput() {  
 while (inputCount < 2) {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 System.*out*.println("Error has occurred in " + getClass().getSimpleName()  
 + "when running waitInput()!" + e.getMessage());  
 }  
 }  
 }  
  
 public synchronized void waitCalculation() {  
 while (calculationCount < Director.*getP*() - 1) {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 System.*out*.println("Error has occurred in " + getClass().getSimpleName()  
 + "when running waitCalculation()!" + e.getMessage());  
 }  
 }  
 }  
  
 public synchronized void waitPreparation() {  
 while (preparationCount < Director.*getP*()) {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 System.*out*.println("Error has occurred in " + getClass().getSimpleName()  
 + "when running waitPreparation()!" + e.getMessage());  
 }  
 }  
 }  
  
 public synchronized void inputMR() {  
 MR.generate(Director.*getMaxValue*());  
 }  
  
 public synchronized void setA(int a) {  
 if(a > this.a) {  
 this.a = a;  
 }  
 }  
  
 public synchronized void setS(int s) {  
 this.s += s;  
 }  
  
 public synchronized Matrix getMR() {  
 return new Matrix(MR);  
 }  
  
 public synchronized int getA() {  
 return a;  
 }  
  
 public synchronized int getS() {  
 return s;  
 }  
  
 public synchronized void signalInputDone() {  
 inputCount++;  
 if(inputCount == 2) {  
 notifyAll();  
 }  
 }  
  
 public synchronized void signalCalculationDone() {  
 calculationCount++;  
 if (calculationCount == Director.*getP*() - 1) {  
 notifyAll();  
 }  
 }  
  
 public synchronized void signalPreparationDone() {  
 preparationCount++;  
 if (preparationCount == Director.*getP*()) {  
 notifyAll();  
 }  
 }  
  
  
}

TaskWorker.java

package com.vodotiiets.workers;  
  
import com.vodotiiets.ParallelTasks.MaxTask;  
import com.vodotiiets.ParallelTasks.VectorMultipleTask;  
import com.vodotiiets.controllers.TaskController;  
import com.vodotiiets.directors.Director;  
import com.vodotiiets.primitives.Matrix;  
  
import java.util.concurrent.ForkJoinPool;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class TaskWorker implements Runnable {  
 private int tid;  
 private TaskController monitor;  
  
 private Matrix MRcopy;  
 private int aCopy;  
 private int sCopy;  
  
 public TaskWorker(int tid, TaskController monitor) {  
 this.monitor = monitor;  
 this.tid = tid;  
 }  
  
 @Override  
 public void run() {  
 if (tid == 0) {  
 monitor.inputMR();  
 Director.*getMA*().generate(Director.*getMaxValue*());  
 Director.*getB*().generate(Director.*getMaxValue*());  
  
 monitor.signalInputDone();  
 }  
  
 if (tid == Director.*getP*() - 1) {  
 Director.*getC*().generate(Director.*getMaxValue*());  
 Director.*getMO*().generate(Director.*getMaxValue*());  
 Director.*getMK*().generate(Director.*getMaxValue*());  
 Director.*getZ*().generate(Director.*getMaxValue*());  
  
 monitor.signalInputDone();  
 }  
  
 monitor.waitInput();  
  
 int startIndex = tid \* Director.*getH*();  
 int endIndex = (tid != Director.*getP*() - 1) ? (tid + 1) \* Director.*getH*() : Director.*getN*();  
  
 MaxTask maxTask = new MaxTask(Director.*getZ*().getPart(startIndex, endIndex));  
 int a = new ForkJoinPool().invoke(maxTask);  
 monitor.setA(a);  
  
 VectorMultipleTask vectorMultipleTask = new VectorMultipleTask(Director.*getB*().getPart(startIndex, endIndex),  
 Director.*getC*().getPart(startIndex, endIndex));  
 int s = new ForkJoinPool(Director.*getP*()).invoke(vectorMultipleTask);  
  
 monitor.setS(s);  
  
 monitor.signalPreparationDone();  
 monitor.waitPreparation();  
  
 MRcopy = monitor.getMR();  
 aCopy = monitor.getA();  
 sCopy = monitor.getS();  
  
 for (int i = 0; i < Director.*getN*(); i++) {  
 for (int j = startIndex; j < endIndex; j++) {  
 int sum = 0;  
 for (int k = 0; k < Director.*getN*(); k++) {  
 sum += MRcopy.get(i, k) \* Director.*getMK*().get(k, j);  
 }  
  
 int value = sCopy \* Director.*getMO*().get(i, j) + aCopy \* sum;  
 Director.*getMA*().set(i, j, value);  
 }  
 }  
  
 if (tid == 0) {  
 monitor.waitCalculation();  
 } else {  
 monitor.signalCalculationDone();  
 }  
  
 if (tid == 0) {  
 Director.*setEndTime*(System.*currentTimeMillis*());  
 System.*out*.println("All threads ended calculations. Result time(ms): " + Director.*getTime*());  
  
 if(Director.*getMA*().getDimension() < 10) {  
 System.*out*.println("Result MA:\n" + Director.*getMA*());  
 } else {  
 System.*out*.println("Result was calculated. Matrix is too large");  
 }  
 }  
  
 }  
}

MaxTask.java

package com.vodotiiets.ParallelTasks;  
  
import com.vodotiiets.directors.Director;  
import com.vodotiiets.primitives.Vector;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.RecursiveTask;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class MaxTask extends RecursiveTask<Integer> {  
 private Vector vector;  
  
 public MaxTask(Vector vector) {  
 this.vector = vector;  
 }  
  
 @Override  
 protected Integer compute() {  
 if(vector.getDimension() > Director.*getH*() / Director.*getP*()) {  
 List<MaxTask> subtasks = createSubtasks();  
  
 for(MaxTask subtask : subtasks) {  
 subtask.fork();  
 }  
  
 int result = Integer.*MIN\_VALUE*;  
 for(MaxTask subtask : subtasks) {  
 int temp = subtask.join();  
 if(temp > result) {  
 result = temp;  
 }  
 }  
  
 return result;  
 } else {  
 return vector.getMaxElement(0, vector.getDimension());  
 }  
 }  
  
 private List<MaxTask> createSubtasks() {  
 List<MaxTask> subtasks = new ArrayList<>();  
  
 MaxTask subtask1 = new MaxTask(this.vector.getPart(0, this.vector.getDimension()/2));  
 MaxTask subtask2 = new MaxTask(this.vector.getPart(this.vector.getDimension()/2 + 1, this.vector.getDimension()));  
  
 subtasks.add(subtask1);  
 subtasks.add(subtask2);  
  
 return subtasks;  
 }  
}

VectorMultiple.java

package com.vodotiiets.ParallelTasks;  
  
import com.vodotiiets.directors.Director;  
import com.vodotiiets.primitives.Vector;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.RecursiveTask;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class VectorMultipleTask extends RecursiveTask<Integer> {  
  
 private Vector vector1;  
 private Vector vector2;  
 private int dimension;  
  
 public VectorMultipleTask(Vector vector1, Vector vector2) {  
 this.vector1 = vector1;  
 this.vector2 = vector2;  
  
 dimension = vector1.getDimension();  
 }  
  
 @Override  
 protected Integer compute() {  
 if(dimension > Director.*getH*() / Director.*getP*()) {  
 List<VectorMultipleTask> subtasks = createSubtasks();  
  
 for(VectorMultipleTask subtask : subtasks) {  
 subtask.fork();  
 }  
  
 int result = 0;  
 for(VectorMultipleTask subtask : subtasks) {  
 result += subtask.join();  
 }  
  
 return result;  
 } else {  
 int result = 0;  
 for (int i = 0; i < dimension; i++) {  
 result += vector1.get(i) \* vector2.get(i);  
 }  
 return result;  
 }  
 }  
  
 private List<VectorMultipleTask> createSubtasks() {  
 List<VectorMultipleTask> subtasks = new ArrayList<>();  
  
 VectorMultipleTask subtask1 = new VectorMultipleTask(vector1.getPart(0, vector1.getDimension() / 2),  
 vector2.getPart(0, vector2.getDimension() / 2));  
 VectorMultipleTask subtask2 = new VectorMultipleTask(vector1.getPart(vector1.getDimension() / 2 + 1,  
 vector1.getDimension()), vector2.getPart(vector2.getDimension() / 2 + 1, vector2.getDimension()));  
  
 subtasks.add(subtask1);  
 subtasks.add(subtask2);  
  
 return subtasks;  
 }  
}

Vector.java

package com.vodotiiets.primitives;  
  
import java.util.Random;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class Vector {  
 private int dimension;  
 private int[] array;  
  
 public Vector(int dimension) {  
 this.dimension = dimension;  
 array = new int[dimension];  
 }  
  
 public Vector(int[] array) {  
 this.array = array;  
 this.dimension = array.length;  
 }  
  
 public void generate(int maxValue) {  
 Random generator = new Random();  
  
 for (int i = 0; i < dimension; i++) {  
 array[i] = generator.nextInt(maxValue);  
 }  
 }  
  
 public int getMaxElement(int startIndex, int endIndex) {  
 if (startIndex < 0 || endIndex > dimension || startIndex >= endIndex)  
 throw new IllegalStateException();  
  
 int result = Integer.*MIN\_VALUE*;  
 for (int i = startIndex ; i < endIndex; i++) {  
 if(array[i] > result) {  
 result = array[i];  
 }  
 }  
 return result;  
 }  
  
 public Vector getPart(int startIndex, int endIndex) {  
 if(endIndex <= startIndex || endIndex > dimension || startIndex < 0)  
 throw new IllegalStateException();  
  
 int length = endIndex - startIndex;  
 int[] resultArray = new int[length];  
 System.*arraycopy*(this.array, startIndex, resultArray, 0, length);  
  
 return new Vector(resultArray);  
 }  
  
 public int getDimension() {  
 return dimension;  
 }  
  
 public int get(int i) {  
 return array[i];  
 }  
  
 @Override  
 public String toString() {  
 String result = "";  
  
 for (int i = 0; i < dimension; i++) {  
 result += array[i] + "\t";  
 }  
  
 return result;  
 }  
}

Matrix.java

package com.vodotiiets.primitives;  
  
import java.util.Random;  
  
*/\*\*  
 \* Created by Denys Vodotiiets.  
 \*/*public class Matrix {  
 private int dimension;  
 private int[][] array;  
  
 public Matrix(int dimension) {  
 this.dimension = dimension;  
 array = new int[dimension][dimension];  
 }  
  
 public Matrix(Matrix other) {  
 this.dimension = other.dimension;  
 array = new int [this.dimension][this.dimension];  
 for (int i = 0; i < this.dimension; i++) {  
 for(int j = 0; j < this.dimension; j++) {  
 this.array[i][j] = other.array[i][j];  
 }  
 }  
 }  
  
 public void generate(int maxValue) {  
 Random generator = new Random();  
 for (int i = 0; i < dimension; i++){  
 for (int j = 0; j < dimension; j++){  
 array[i][j] = generator.nextInt(maxValue);  
 }  
 }  
 }  
  
 public int get(int i, int j) {  
 return array[i][j];  
 }  
  
 public void set(int i, int j, int value) {  
 array[i][j] = value;  
 }  
  
 public int getDimension() {  
 return dimension;  
 }  
  
 @Override  
 public String toString() {  
 String result = "";  
 for (int i = 0; i < dimension; i++){  
 for (int j = 0; j < dimension; j++){  
 result += array[i][j] +"\t";  
 }  
 result += "\n";  
 }  
 return result;  
 }  
}

**Додаток Є**

**Лістинг програми ПРГ2**