МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ

Факультет менеджменту

Кафедра економічної кібернетики, комп'ютерних наук та інформаційних технологій

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

методичні рекомендації

для практичних занять та самостійної
роботи здобувачів першого (бакалаврського) рівня вищої освіти
ОПП «Комп'ютерні науки» спеціальності F3(122) «Комп'ютерні науки» денної форми здобуття вищої освіти

МИКОЛАЇВ
2025

УДК 004.75:004.272
Т38

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету від 27 березня 2025 року, протокол № 7.

Укладачі:

С. І. Тищенко	к.п.н., доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету
О. Ю. Пархоменко	к.ф.-м.н., доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету
О.О. Жебко	асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій, Миколаївський національний аграрний університет

Рецензенти:

В.В.Базаренко	заступник начальника Миколаївської обласної військової адміністрації з питань цифрового розвитку, цифрових трансформацій і цифровізації (CDTO)
Д.Л.Кошкін	к.т.н., доцент, доцент кафедри електроенергетики, електротехніки та електромеханіки Миколаївського національного аграрного університету

ЗМІСТ

Передмова

Курс дисципліни «Технології розподілених систем та паралельних обчислень» має важливе значення в теоретичній підготовці майбутніх фахівців і є обов'язковою компонентою підготовки здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спеціальності 122 «Комп'ютерні науки».

Мета дисципліни: сформувати у здобувачів необхідний обсяг теоретичних і практичних знань про основні поняття та архітектурні принципи розподілених систем, такі як мережеві комунікації, розподілені бази даних, сервіс-орієнтовані архітектури та методи паралельного програмування, включаючи концепції багатозадачності, потоків виконання, розподілених алгоритмів та технік синхронізації.

Основними завданнями, що мають бути вирішенні у процесі викладення дисципліни, є надання здобувачам вищої освіти:

- теоретичних відомостей про концепції розподілених систем, включаючи архітектурні моделі, механізми комунікації та синхронізації;

- практичних навичок реалізації методів паралельного програмування, концепцій багатозадачності, потоків виконання, розподілених алгоритмів та технік синхронізації;

- досвіду розробки та налагодження розподілених та паралельних застосунків;

- знань про архітектуру сучасних мікропроцесорів та мультипроцесорів.

Предмет дисципліни: багатопроцесорні обчислювальні системи, розробка багатопотокових додатків, класичні задачі синхронізації, моделювання паралельних обчислень.

Загальна мета полягає у підготовці здобувачів до професійної діяльності у галузі розробки програмного забезпечення, де розподілені та паралельні обчислення відіграють важливу роль у розв'язанні складних завдань обробки даних та обчислень.

Практична робота № 1

Тема: Вступ до багатопоточного (паралельного) програмування. Клас Thread. Створення, виконання потоків. Завершення за допомогою методу sleep().

Основні теоретичні відомості

Більшість мов програмування підтримують таку важливу функціональність як багатопотоковість, і Java не виняток. За допомогою багатопоточності ми можемо виділити у застосунку кілька потоків, які виконуватимуть різні завдання одночасно. Якщо у нас, припустимо, графічний додаток, який посилає запит до якогось сервера або зчитує та обробляє величезний файл, то без багатопоточності у нас блокувався б графічний інтерфейс на час виконання завдання. А завдяки потокам ми можемо виділити відправку запиту або будь-яке інше завдання, яке може довго оброблятись, в окремий потік. Тому більшість реальних застосунків, які багатьом із нас доводиться використовувати, практично неможливо використовувати без багатопоточності.

Клас Thread

У Java функціональність окремого потоку полягає у класі **Thread**. І щоб створити новий потік, нам треба створити об'єкт цього класу. Але не всі потоки створюються вручну програмістом. Коли програма запускається, починає працювати головний потік цієї програми. Від цього головного потоку породжуються й інші дочірні потоки.

За допомогою статичного методу **Thread.currentThread()** ми можемо отримати поточний потік виконання:

```
1  public static void main(String[] args) {
2
3      Thread t = Thread.currentThread(); //
4      System.out.println(t.getName()); // ma
5  }
```

За замовчуванням ім'ям головного потоку буде main.

Для керування потоком клас Thread має ще низку методів. Найбільш використовувані з них:

- getName(): повертає ім'я потоку.
- setName(String name): встановлює ім'я потоку.
- getPriority(): повертає пріоритет потоку
- setPriority(int proirity): встановлює пріоритет потоку.

Пріоритет є одним із ключових факторів для вибору системою потоку з купи потоків для виконання. Цей метод як параметр передається числове значення пріоритету - від 1 до 10. За замовчуванням головному потоку виставляється середній пріоритет - 5.

- isAlive(): повертає true, якщо потік активний.
- isInterrupted(): повертає true, якщо потік був перерваний.
- join(): очікує завершення потоку.
- run(): визначає точку входу в потік.
- sleep(): призупиняє потік на задану кількість мілісекунд.
- start(): запускає потік, викликаючи його метод run().

Ми можемо вивести всю інформацію про потік:

```
public static void main(String[] args) {

    Thread t = Thread.currentThread(); /
    System.out.println(t); // main
}
```

Консольний вивід:

```
Thread[main,5,main]
```

Перше main буде представляти ім'я потоку (що можна отримати через *t.getName()*), друге значення 5 надає пріоритет потоку (також можна отримати через *t.getPriority()*), і останнє main представляє ім'я групи потоків, до якого належить поточний - замовчуванням також main (також можна отримати через *t.getThreadGroup().getName())*

Недоліки при використанні потоків

Далі ми розглянемо, як створювати та використовувати потоки. Це досить просто. Однак при створенні багатопотокового застосунку нам слід враховувати низку обставин, які негативно можуть позначитися на роботі програми.

На деяких платформах запуск нових потоків може сповільнити роботу програми. Що може мати велике значення, якщо нам є критична продуктивність програми.

Для кожного потоку створюється свій власний стек у пам'яті, куди поміщаються всі локальні змінні та ряд інших даних, пов'язаних із виконанням потоку. Відповідно, що більше потоків створюється, то більше пам'яті використовується. При цьому треба пам'ятати, у будь-якій системі розміри пам'яті, що використовується, обмежені. Крім того, у багатьох системах може бути обмеження кількості потоків. Але навіть якщо такого обмеження немає, то у будь-якому випадку є природне обмеження у вигляді максимальної швидкості процесора.

Для створення нового потоку ми можемо створити новий клас або наслідуючи його від класу Thread або реалізуючи в класі інтерфейс Runnable.

Спадкування від класу Thread

Створимо свій клас на основі Thread:

```
class JThread extends Thread {

    JThread(String name){
        super(name);
    }

    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s fiished... \n", Thread.currentThread().getName());
    }
}

public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        new JThread("JThread").start();
        System.out.println("Main thread finished...");
    }
}
```

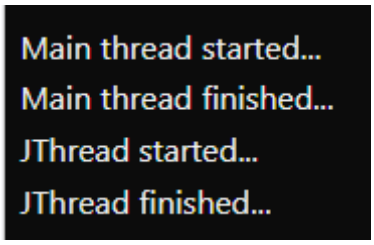
Клас потоку називається JThread. Передбачається, що конструктору класу передається ім'я потоку, яке потім передається конструктор базового класу. У конструктор свого класу потоку ми можемо передати різні дані, але головне, щоб у ньому викликався конструктор базового класу Thread, в який передається ім'я потоку.

І також JThread перевизначається метод *run()*, код якого власне і буде представляти весь той код, який виконується в потоці.

У методі main для запуску потоку JThread у нього викликається метод start(), після чого починається виконання коду, який визначений у методі run:

```
new JThread("JThread").start();
```

Консольний вивід:



Тут у методі main у конструктор JThread передається довільна назва потоку, і потім викликається метод start(). По суті, цей метод якраз і викликає перевизначений метод run() класу JThread.

Зауважимо, що головний потік завершує роботу раніше, ніж породжений ним дочірній потік JThread. Аналогічно до створення одного потоку ми можемо запускати відразу кілька потоків:

```
public static void main(String[] args) {

    System.out.println("Main thread started...");
    for(int i=1; i < 6; i++)
        new JThread("JThread " + i).start();
    System.out.println("Main thread finished...");
}
```

Консольний вивід:

Очікування завершення потоку

При запуску потоків у прикладах вище Main thread завершувався до дочірнього потоку. Як правило, більш поширеною ситуацією є випадок, коли Main thread завершується останнім. Для цього необхідно застосувати метод join(). У цьому випадку поточний потік чекатиме завершення потоку, для якого викликаний метод join:

```
public static void main(String[] args) {

    System.out.println("Main thread started...");
    JThread t= new JThread("JThread ");
    t.start();
    try{
        t.join();
    }
    catch(InterruptedException e){

        System.out.printf("%s has been interrupted", t.getName());
    }
    System.out.println("Main thread finished...");
}
```

Метод join() змушує потік (у цьому випадку Main thread) очікувати завершення викликаного потоку, для якого і застосовується метод join (у разі JThread).

Консольний вивід:

```
Main thread started...
JThread  started...
JThread  finished...
Main thread finished...
```

Якщо у програмі використовується декілька дочірніх потоків, і треба, щоб Main thread завершувався після дочірніх, то для кожного дочірнього потоку треба викликати метод join.

Реалізація інтерфейсу Runnable

Інший спосіб визначення потоку є реалізація інтерфейсу Runnable. Цей інтерфейс має один метод run:

```
interface Runnable{

    void run();
}
```

У методі run() власне визначається весь код, який виконується при запуску потоку. Після визначення об'єкта Runnable він передається в один із конструкторів класу Thread:

```
Thread(Runnable runnable, String threadName)
```

Для реалізації інтерфейсу визначимо наступний клас MyThread:

```
class MyThread implements Runnable {


    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        try{
            Thread.sleep(500);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}

public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        Thread myThread = new Thread(new MyThread(),"MyThread");
        myThread.start();
        System.out.println("Main thread finished...");
    }
}
```

Реалізація інтерфейсу Runnable багато в чому аналогічна до перевизначення класу Thread. Також у методі run визначається найпростіший код, який призупиняє потік на 500 мілісекунд.

У методі main викликається конструктор Thread, який передається об'єкт MyThread. І щоб запустити потік, викликається метод start(). У результаті консоль виведе щось на кшталт наступного:

```
Main thread started...
Main thread finished...
MyThread started...
MyThread finished...
```

Оскільки Runnable фактично представляє функціональний інтерфейс, який визначає один метод, то об'єкт цього інтерфейсу ми можемо уявити як лямбда-вираз:

```
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        Runnable r = ()->{
            System.out.printf("%s started... \n", Thread.currentThread().getName());
            try{
                Thread.sleep(500);
            }
            catch(InterruptedException e){
                System.out.println("Thread has been interrupted");
            }
            System.out.printf("%s finished... \n", Thread.currentThread().getName());
        };
        Thread myThread = new Thread(r,"MyThread");
        myThread.start();
        System.out.println("Main thread finished...");
    }
}
```

Індивідуальне завдання:

Розробити багатопоточні консольні застосунки, використовуючи клас Thread та інтерфейс Runnable. Задати та встановити нові ім’я та пріоритети потокам, вивести дані про потоки. Використати методи isAlive(), join(), run(), sleep(), start().

Практична робота № 2

Тема: Завершення та переривання потоку. Методи interrupt та isInterrupted(). Обробка виключення InterruptedException**.**

Основні теоретичні відомості

Приклади потоків являли собі потік як послідовний набір операцій. Після виконання останньої операції завершувався потік. Однак нерідко має місце й інша організація потоку у вигляді нескінченного циклу. Наприклад, потік сервера в нескінченному циклі прослуховує певний порт щодо отримання даних. І в цьому випадку ми також можемо передбачити механізм завершення потоку.

Завершення потоку. Поширений спосіб завершення потоку представляє опитування логічної змінної. І якщо вона дорівнює, наприклад, false, то потік завершує нескінченний цикл та закінчує своє виконання. Визначимо наступний клас потоку:

```
class MyThread implements Runnable {

    private boolean isActive;

    void disable(){
        isActive=false;
    }

    MyThread(){
       isActive = true;
    }

    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1;
        while(isActive){
            System.out.println("Loop " + counter++);
            try{
                Thread.sleep(400);
            }
            catch(InterruptedException e){
                System.out.println("Thread has been interrupted");
            }
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}
```

Змінна isActive вказує на активність потоку. За допомогою методу disable() ми можемо скинути стан цієї змінної. Тепер використовуємо цей клас:

```
public static void main(String[] args) {

    System.out.println("Main thread started...");
    MyThread myThread = new MyThread();
    new Thread(myThread,"MyThread").start();

    try{
        Thread.sleep(1100);

        myThread.disable();

        Thread.sleep(1000);
    }
    catch(InterruptedException e){
        System.out.println("Thread has been interrupted");
    }
    System.out.println("Main thread finished...");
}
```

Спочатку запускається дочірній потік: *new Thread(myThread,"MyThread").start()*. Потім на 1100 мілісекунд зупиняємо Main thread і потім викликаємо метод myThread.disable(), який перемикає в потоці прапор isActive. І дочірній потік завершується.

Метод interrupt. Ще один спосіб виклику завершення або переривання потоку є методом *interrupt()*. Виклик цього методу встановлює потоку статус, що він перерваний. Сам метод повертає true, якщо потік може бути перерваний, інакше повертається false.

При цьому сам виклик цього методу не завершує потік, він лише встановлює статус: зокрема, метод isInterrupted() класу Thread повертатиме значення true. Ми можемо перевірити значення, що повертається даним методом і зробити деякі дії. У класі, який успадкований від Thread, ми можемо отримати статус поточного потоку за допомогою методу, isInterrupted(). І доки цей метод повертає false, ми можемо виконувати цикл. А після того, як буде викликаний метод interrupt, isInterrupted() поверне true, і відповідно відбудеться вихід із циклу. Наприклад:

```java
class JThread extends Thread {

    JThread(String name){
        super(name);
    }
    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1;
        while(!isInterrupted()){

            System.out.println("Loop " + counter++);
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        JThread t = new JThread("JThread");
        t.start();
        try{
            Thread.sleep(150);
            t.interrupt();

            Thread.sleep(150);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.println("Main thread finished...");
    }
}
```

Можливий результат у консолі:

```
Main thread started...
JThread started...
Loop 1
Loop 2
Loop 3
Loop 4
JThread finished...
Main thread finished...
```

Якщо основна функціональність закодована у класі, що реалізує інтерфейс Runnable, то там можна перевіряти статус потоку за допомогою методу *Thread.currentThread().isInterrupted()*:

```
class MyThread implements Runnable {

    public void run(){

        System.out.printf("%s started... \n", Thread.currentThread().getName());
        int counter=1; /
        while(!Thread.currentThread().isInterrupted()){

            System.out.println("Loop " + counter++);
        }
        System.out.printf("%s finished... \n", Thread.currentThread().getName());
    }
}
public class Program {

    public static void main(String[] args) {

        System.out.println("Main thread started...");
        MyThread myThread = new MyThread();
        Thread t = new Thread(myThread,"MyThread");
        t.start();
        try{
            Thread.sleep(150);
            t.interrupt();

            Thread.sleep(150);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.println("Main thread finished...");
    }
}
```

Однак при отриманні статусу потоку за допомогою методу isInterrupted() слід враховувати, що якщо ми обробляємо в циклі виключення **InterruptedException** у блоці catch, то при перехопленні виключення статус потоку автоматично скидається, і після цього isInterrupted буде повертати false.

Наприклад, додамо в цикл потоку затримку за допомогою методу sleep:

```
public void run(){

    System.out.printf("%s started... \n", Thread.currentThread().getName());
    int counter=1; ,
    while(!isInterrupted()){

        System.out.println("Loop " + counter++);
        try{
            Thread.sleep(100);
        }
        catch(InterruptedException e){
            System.out.println(getName() + " has been interrupted");
            System.out.println(isInterrupted());    // false
            interrupt();
        }
    }
    System.out.printf("%s finished... \n", Thread.currentThread().getName());
}
```

Коли потік викликає метод interrupt, метод sleep згенерує виняток InterruptedException, і управління перейде до блоку catch. Але якщо ми перевіримо статус потоку, то побачимо, що isInterrupted повертає false. Як варіант, у разі ми можемо повторно перервати поточний потік, знову ж таки викликавши метод interrupt(). Тоді при новій ітерації циклу при методі, isInterrupted, поверне true, і буде вихід з циклу.

Або ми можемо відразу в блоці catch вийти з циклу за допомогою break:

```
while(!isInterrupted()){

    System.out.println("Loop " + counter++);
    try{
        Thread.sleep(100);
    }
    catch(InterruptedException e){
        System.out.println(getName() + " has been interrupted");

        break;
    }
}
```

Якщо нескінченний цикл поміщений у конструкцію try...catch, достатньо обробити InterruptedException:

```java
public void run(){

    System.out.printf("%s started... \n", Thread.currentThread().getName());
    int counter=1;
    try{
        while(!isInterrupted()){
            System.out.println("Loop " + counter++);
            Thread.sleep(100);
        }
    }
    catch(InterruptedException e){
        System.out.println(getName() + " has been interrupted");
    }

    System.out.printf("%s finished... \n", Thread.currentThread().getName());
}
```

Індивідуальне завдання:

Розробити багатопоточний консольний застосунок, використовуючи клас Thread чи інтерфейс Runnable. Зупинити виконання потоків використавши методи interrupt та isInterrupted(). Обробити виключення InterruptedException. У створеному потоці вивести номер залікової книжки студента.

Практична робота № 3

Тема: Синхронізація потоків. Оператор synchronized.

Основні теоретичні відомості

Синхронізація відноситься до багатопоточності. Синхронізований блок коду може бути виконаний лише одним потоком одночасно. Java підтримує кілька потоків для виконання. Це може призвести до того, що два або більше потоків отримають доступ до одного і того ж поля або об'єкта. Синхронізація це процес, який дозволяє виконувати всі паралельні потоки у програмі синхронно. Синхронізація дозволяє уникнути помилок узгодженості пам'яті, викликаних через непослідовний доступ до спільної пам'яті.

Синхронізація досягається Java використовуючи зарезервоване слово *synchronized*. Ви можете використовувати його у своїх класах, визначаючи синхронізовані методи або блоки. Ви не зможете використовувати synchronized у змінних або атрибутах у визначенні класу.

Під час роботи потоки нерідко звертаються до якихось загальних ресурсів, які визначені поза потоком, наприклад, звернення до якогось файлу. Якщо одночасно кілька потоків звернуться до загального ресурсу, результати виконання програми можуть бути несподіваними і навіть непередбачуваними. Наприклад, визначимо наступний код:

```
1  public class Program {
2
3      public static void main(String[] args) {
4
5          CommonResource commonResource= new CommonResource();
6          for (int i = 1; i < 6; i++){
7
8              Thread t = new Thread(new CountThread(commonResource));
9              t.setName("Thread "+ i);
10             t.start();
11         }
12     }
13 }
14
```

```java
class CommonResource{

    int x=0;
}

class CountThread implements Runnable{

    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }
    public void run(){
        res.x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);
            res.x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

Тут визначено клас CommonResource, який представляє загальний ресурс і в якому визначено ціле число x.

Цей ресурс використовується класом потоку CountThread. Цей клас просто збільшує у циклі значення x на одиницю. Причому при вході в потік значення x = 1: `res.x=1;`

Тобто в результаті ми очікуємо, що після виконання циклу res.x дорівнюватиме 4.

У головному класі програми запускається п'ять потоків. Тобто, ми очікуємо, що кожен потік буде збільшувати res.x з 1 до 4 і так п'ять разів. Але якщо ми подивимося на результат роботи програми, то він буде іншим:

```
Thread 1 1
Thread 2 1
Thread 3 1
Thread 5 1
Thread 4 1
Thread 5 6
Thread 2 6
```

```
Thread 1 6
Thread 3 6
Thread 4 6
Thread 4 11
Thread 2 11
Thread 5 11
Thread 3 11
Thread 1 11
Thread 4 16
Thread 1 16
Thread 3 16
Thread 5 16
Thread 2 16
```

Тобто, поки один потік не закінчив роботу з полем res.x, з ним починає працювати інший потік.

Щоб уникнути подібної ситуації, треба синхронізувати потоки. Одним із способів синхронізації є використання ключового слова *synchronized*. Цей оператор передує блоку коду або методу, який підлягає синхронізації. Для його застосування змінимо клас CountThread:

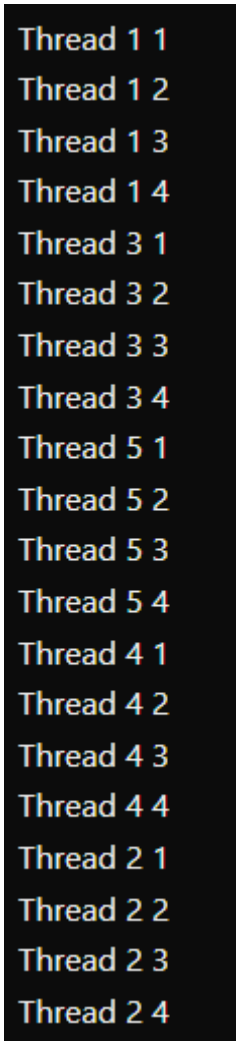
```
1  class CountThread implements Runnable{
2
3      CommonResource res;
4      CountThread(CommonResource res){
5          this.res=res;
6      }
7      public void run(){
8          synchronized(res){
9              res.x=1;
10             for (int i = 1; i < 5; i++){
11                 System.out.printf("%s %d \n", Thread.currentThread().getName(), res.x);
12                 res.x++;
13                 try{
14                     Thread.sleep(100);
15                 }
16                 catch(InterruptedException e){}
17             }
18         }
19     }
20 }
```

Під час створення синхронізованого блоку коду після оператора synchronized йде об'єкт-заглушка: synchronized(res). Причому як об'єкт може

використовуватися тільки об'єкт якогось класу, але не примітивного типу.

Кожен об'єкт Java має асоційований з ним **монітор**. Монітор представляє свого роду інструмент управління доступу до об'єкту. Коли виконання коду доходить до оператора synchronized, монітор об'єкта res блокується, і на його блокування монопольний доступ до блоку коду має лише один потік, який і зробив блокування. Після закінчення роботи блоку коду монітор об'єкта res звільняється і стає доступним для інших потоків.

Після звільнення монітора його захоплює інший потік, а решта потоків продовжують чекати його звільнення. У результаті консольний вивід зміниться на наступний:



При застосуванні оператора synchronized до методу, поки цей метод не завершить виконання, монопольний доступ має тільки один потік - перший, який почав його виконання. Для застосування synchronized до методу, змінимо

класи програми:

Результат роботи в даному випадку буде аналогічним прикладу вище з

```
public class Program {

    public static void main(String[] args) {

        CommonResource commonResource= new CommonResource();
        for (int i = 1; i < 6; i++){

            Thread t = new Thread(new CountThread(commonResource));
            t.setName("Thread "+ i);
            t.start();
        }
    }
}
```

```
class CommonResource{

    int x;
    synchronized void increment(){
        x=1;
        for (int i = 1; i < 5; i++){
            System.out.printf("%s %d \n", Thread.currentThread().getName(), x);
            x++;
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
        }
    }
}
```

```
class CountThread implements Runnable{

    CommonResource res;
    CountThread(CommonResource res){
        this.res=res;
    }

    public void run(){
        res.increment();
    }
}
```

блоком synchronized. Тут знову у справу вступає монітор об'єкта CommonResource - загального об'єкта всіх потоків. Тому синхронізованим оголошується не метод run() у класі CountThread, а метод increment класу

CommonResource. Коли перший потік починає виконання методу increment, він захоплює монітор об'єкта CommonResource. А всі потоки також продовжують чекати на його звільнення.

Індивідуальне завдання

Розробити багатопоточний консольний застосунок, використовуючи клас Thread чи інтерфейс Runnable. Синхронізувати доступ до потоків та допомогою оператору synchronized. У кожному потоці вивести номер студентського квитка.

Практична робота № 4

Тема: Організація міжпотокової взаємодії за допомогою монітора. Використання методів wait(), notify() та notifyAll().

Теоретичні відомості

У багатопоточних додатках, для вирішення супутніх задач, головна програма у процесі своєї роботи постійно породжує дочірні потоки. Дочірні потоки, протягом деякого часу, виконують наданий їм об'єм роботи та завершуються. Головна програма завжди (у будь-який момент часу) повинна читко знати, які з задач чекають свого вирішення, які вирішуються, які вже вирішені, і їх результати доступні для подальшого використання.

Для виконання цієї роботи призначений спеціальний програмний механізм, який зветься монітором. За допомогою монітора головна програма завжди має актуальну інформацію про те, скільки і які потоки працюють у даний час та скільки і які потоки уже завершені.

Кожен об'єкт в Java має асоційований з ним монітор. Монітор представляє свого роду інструмент для управління доступу до об'єкта. Коли виконання коду доходить до оператора synchronized, монітор об'єкта m блокується, і на час його блокування монопольний доступ до блоку коду має тільки один потік, який і зробив блокування. Після закінчення роботи блоку коду, монітор об'єкта m звільняється і стає доступним для інших потоків.

Іноді при взаємодії потоків постає питання про повідомлення одних потоків про дії інших. Наприклад, дії одного потоку залежать від результату дій іншого потоку, і треба якось сповістити один потік, що другий потік виконав якусь роботу. І для подібних ситуацій у класу Object визначено ряд методів:

wait()— звільняє монітор і переводить викликаний потік в стан очікування до тих пір, поки інший потік не викличе метод notify();

notify()— продовжує роботу потоку, у якому раніше був викликаний метод wait();

notifyAll()— відновлює роботу всіх потоків, у яких раніше був

викликаний метод wait();

Монітор — в мовах програмування, це високорівнева конструкція для отримання ексклюзивного доступу до спільних ресурсів.Монітор реалізується за допомогою м'ютекса та умовних змінних.

Монітор був винайдений П.Б. Хансеном та Тоні Гоаром та вперше був застосований в мові Concurrent Pascal.

ЛІСТИНГ:

```
class TestThread extends Thread {

    String threadName;
    Monitor m;

    TestThread(String name, Monitor mm) {
        threadName = name;
        m = mm;
        System.out.println(threadName + " - Created");
    }
    public void run() {
        m.procIncrement();
        System.out.println(threadName + " - Start of Work");
        try {
            Thread.sleep(100);
        } catch (InterruptedException ie) {
        }
        m.procDecrement();
        synchronized (m) {
            m.notify();
            System.out.println(threadName + " - Signal sended");
        }
        System.out.println(threadName + " - End of Work");
    }
}

public class Monitor implements Runnable {

    int procNum = 0;

    Monitor(int procNumber) {
        TestThread t[] = new TestThread[procNumber];
        for (int i = 0; i < procNumber; ++i) {
            t[i] = new TestThread("Proc:" + i, this);
            t[i].start();
        }
    }
```

```
public void run() {
    System.out.println("Monitor - Started: " + getProcNum());
    try {
        while (getProcNum() != 0) {
            synchronized (this) {
                System.out.println("Monitor - Waiting: " + getProcNum());
                wait();
                System.out.println("Current process: "+procNum);
                if (getProcNum() == 3) {
                    System.out.println("Oleksandr Zhebko");
                }
                System.out.println("Monitor - Signal received: " + getProcNum());
            }
        }
    } catch (InterruptedException ee) {
        System.out.println("Monitor - Interrupted Exception: " + ee.toString());
    }
    System.out.println("Monitor - Ended: " + getProcNum());
}
```

```
    public synchronized void procIncrement() {
        ++procNum;
    }

    public synchronized void procDecrement() {
        --procNum;
    }

    public synchronized int getProcNum() {
        return (procNum);
    }

    public static void main(String argc[]) {
        System.out.println("Main process started");
        Monitor m = new Monitor(5);
        new Thread(m).start();
        System.out.println("Main process ended");
    }
}
```

Індивідуальне завдання

Розглянути, відкомпілювати та запустити на виконання наведений приклад. З'ясувати принципи багатопоточної синхронізації за допомогою монітора процесів. З'ясувати особливості програмування та застосування монітора процесів. У другому потоці вивести ім’я, прізвище студента та у третьому розрахувати суму двох чисел.

Практична робота № 5

Тема: Організація міжпотокової взаємодії за допомогою засувок.

Мета: Опанувати принципи міжпотокової взаємодії за допомогою засувок.

Теоретичні відомості

Звичайна синхронізація забезпечує надійне оновлення множинних загальних змінних у багатопотоковому застосунку, попереджаючи гонку або пошкодження даних і гарантує, що паралельні потоки, які синхронізуються належним чином, побачать останні значення цих змінних. Але така синхронізація має деякі функціональні обмеження:

- неможливо перервати потік, який очікує на блокування;
- неможливо опитувати чи намагатися отримати блокування, не будучи готовим до тривалого очікування;
- блокування має бути знято в тому ж стековому фреймі, в якому було розпочато.

Пакет java.util.concurrent.locks включає класи, які можна використовувати для блокування ресурсів з певними умовами, які суттєво відрізняються від вбудованої синхронізації та моніторів. Цей пакет дозволяє набагато більшу гнучкість у використанні блокувань без умов та з умовами. Класи пакету реалізують такі інтерфейси:

• **Lock** - інтерфейс підтримує порядок блокування та дозволяє використовувати багаторазово пов'язаний умовний об'єкт Condition;

• **Condition -** інтерфейс описує пов'язані з блокуванням змінні, які можуть виконувати функції монітора об'єкта;

• **ReadWriteLock** - інтерфейс підтримує пару пов'язаних блокувань: одне для читання та одне для запису.

Для управління доступом до загального ресурсу в якості альтернативи оператору synchronized ми можемо використовувати блокування.

Спочатку потік намагається отримати доступ до загального ресурсу. Якщо він вільний, то на нього накладається блокування. Після завершення роботи, блокування з загального ресурсу знімається. Якщо ж ресурс не вільний

і на нього вже накладено блокування, то потік очікує, поки це блокування не буде знято.

Класи блокувань ресурсів реалізують інтерфейс **Lock**, який визначає наступні методи:

void lock(): очікує, поки не буде отримано блокування,

void lockInterruptibly() throws InterruptedException: очікує, поки не буде отримано блокування, якщо потік не перерваний,

boolean tryLock(): намагається отримати блокування, якщо блокування отримано, то повертає true. Якщо блокування не отримано, то повертає false. На відміну від методу lock() не очікує отримання блокування, якщо воно недоступне,

void unlock(): знімає блокування,

Condition newCondition(): повертає Об'єкт Condition, який пов'язаний з поточним блокуванням.

Організація блокування в загальному випадку досить проста: для отримання блокування викликається метод lock(), а після закінчення роботи з загальними ресурсами викликається метод unlock(), який знімає блокування. Об'єкт Condition дозволяє управляти блокуванням. Як правило, для роботи з блокуванням ресурсів використовується клас ReentrantLock з пакету java.util.concurrent.locks. Даний клас реалізує інтерфейс Lock.

Нерекурсивна версія засувки. У тому випадку, коли нема необхідності контролювати кількість потоків, які одночасно мають доступ до загального ресурсу, можна використати засувки Lock.

Приклад використання нерекурсивного класу засувки:

ЛІСТИНГ:

```
class TestThread extends Thread {
  String threadName;
  Counter c;
  int delay;

  TestThread( String name, int t, Counter cc) {
    threadName = name;
    c = cc;
    delay = t;
    System.out.println( threadName + " - Created");
    }

  public void run() {
    System.out.println( threadName + " - Start of Work");
    for (int i = 0; i < 10; ++i) {
      if (delay % 2 == 0) {
        System.out.println( threadName + " - increment " + c.inc());
        } else {
        System.out.println( threadName + " - decrement " + c.dec());
        }
      try {
        Thread.sleep( (int)((double)delay * 10.0 * Math.random()));
        }
      catch (InterruptedException ie) {
        }
      }
    System.out.println( threadName + " - End of Work");
    }
  }


 class Lock {
   private boolean isLocked = false;

   public synchronized void lock() throws InterruptedException {
     while(isLocked) {
       wait();
       }
     isLocked = true;
     }

   public synchronized void unlock() {
     isLocked = false;
     notify();
     }
  }
```

```
public class Counter {
  private Lock lock = new Lock();
  private int count = 0;
  private int newCount;

  public int inc() {
    try {
      lock.lock();
      newCount = ++count;
      lock.unlock();
      }
    catch (InterruptedException ie) {
      }
    return newCount;
    }

  public int dec() {
    try {
      lock.lock();
      newCount = --count;
      lock.unlock();
      }
    catch (InterruptedException ie) {
      }
    return newCount;
    }

  public static void main( String argc[]) {
    System.out.println( "Main process started");
    Counter cnt = new Counter();
    TestThread t[] = new TestThread[10];
    for (int i = 0; i < 10; ++i) {
      t[i] = new TestThread( "Proc:" + i, i, cnt);
      t[i].start();
      }
    System.out.println( "Main process ended");
          }
}
```

Завдання № 1. Зробити налагодження коду нерекурсивної версії засувки, навести результати у звіті виконання ПР № 5.

Рекурсивна версія засувки. Синхронізовані блоки у мові Java є рекурсивними. А це, в свою чергу, дає можливість рекурсивного блокування ресурсів. Таким чином код засувки можна переписати для рекурсивного використання.

```
class TestThread extends Thread {
  String threadName;
  CounterReentrant c;
  int delay;

  TestThread( String name, int t, CounterReentrant cc) {
    threadName = name;
    c = cc;
    delay = t;
    System.out.println( threadName + " - Created");
    }

  public void run() {
    System.out.println( threadName + " - Start of Work");
    for (int i = 0; i < 10; ++i) {
      if (delay % 2 == 0) {
        System.out.println( threadName + " - increment " + c.inc());
        } else {
        System.out.println( threadName + " - decrement " + c.dec());
        }
      try {
        Thread.sleep( (int)((double)delay * 10.0 * Math.random()));
        }
      catch (InterruptedException ie) {
        }
      }
    System.out.println( threadName + " - End of Work");
    }
  }
class Lock {
  boolean isLocked = false;
  Thread lockedBy = null;
  int lockedCount = 0;
  public synchronized void lock() throws InterruptedException {
    Thread callingThread = Thread.currentThread();
    while(isLocked && lockedBy != callingThread) {
      wait();
      }
    isLocked = true;
    lockedCount++;
    lockedBy = callingThread;
    }

  public synchronized void unlock() {
    if (Thread.currentThread() == this.lockedBy) {
      lockedCount--;
      if(lockedCount == 0) {
        isLocked = false;
        notify();
        }
      }
    }
  }
```

```
public class CounterReentrant {
  private Lock lock = new Lock();
  private int count = 0;
  private int newCount;

  public int inc() {
    try {
      lock.lock();
      newCount = ++count;
      lock.unlock();
      }
    catch (InterruptedException ie) {
      }
    return newCount;
    }

  public int dec() {
    try {
      lock.lock();
      newCount = --count;
      lock.unlock();
      }
    catch (InterruptedException ie) {
      }
    return newCount;
    }

  public static void main( String argc[]) {
    System.out.println( "Main process started");
    CounterReentrant cnt = new CounterReentrant();
    TestThread t[] = new TestThread[10];
    for (int i = 0; i < 10; ++i) {
      t[i] = new TestThread( "Proc:" + i, i, cnt);
      t[i].start();
      }
  System.out.println( "Main process ended");
  }
}
```

Завдання № 2. Зробити налагодження коду рекурсивної версії засувки, навести результати у звіті виконання ПР № 5.

Ключове слово volatile. *Volatile* - говорить потоку що змінна може змінюватися, і інформує потік про необхідність звертатися до останньої версії, а не до хешированої копії і своєчасно поширювати зміни.

Індивідуальне завдання до захисту роботи

Написати паралельну програму із загальною змінною типу String та двома робочими процесами. Загальній змінній на початку роботи програми

надати значення "Start:". Кожен робочий процес під час роботи програми повинен п'ять разів звернутися до загальної зм_нної і кожного разу додавати у її кінець по одній літері. Перший процес повинен додавати літеру "A", а другий "B". Таким чином у кінці роботи програми загальна змінна повинна містити приблизно таке значення "Start:ABBAAABABB". Процеси повинні працювати паралельно і незалежно один від одного. Програма повинна працювати вірно навіть якщо під час виконання операції додавання ввести випадкову часову затримку.

Практична робота № 6

Тема: Організація міжпотокової взаємодії за допомогою семафорів.

Мета: Опанувати принципи міжпотокової взаємодії за допомогою семафорів.

Теоретичні відомості

Семафори представляють ще один засіб синхронізації для доступу до ресурсів. В Java семафори представлені класом Semaphore, який розташовується в пакеті java.util.concurrent.

Для управління доступом до ресурсу семафор використовує лічильник, що представляє кількість дозволів. Якщо значення лічильника більше нуля, то потік отримує доступ до ресурсу, при цьому лічильник зменшується на одиницю. Після закінчення роботи з ресурсом потік звільняє семафор, і лічильник збільшується на одиницю. Якщо ж лічильник дорівнює нулю, то потік блокується і чекає, поки не отримає дозвіл від семафора.

Встановити кількість дозволів для доступу до ресурсу можна за допомогою конструкторів класу Semaphore:

Semaphore(int permits)

Semaphore(int permits, boolean fair)

Параметр permits вказує на кількість допустимих дозволів для доступу до ресурсу. Параметр fair у другому конструкторі дозволяє встановити черговість отримання доступу. Якщо він дорівнює true, то дозволи будуть надаватися потокам, що очікують в тому порядку, в якому вони запитували доступ. Якщо ж він дорівнює false, то дозволи будуть надаватися в невизначеному порядку.

Для отримання дозволу у семафора треба викликати метод acquire (), який має дві форми:

void acquire() throws InterruptedException

void acquire (int permits) throws InterruptedBxception

Для отримання одного дозволу застосовується перший варіант, а для отримання декількох дозволів - другий варіант. Після виклику цього методу поки потік не отримає дозвіл, Він блокується. Після закінчення роботи з

ресурсом отриманий раніше дозвіл треба звільнити за допомогою методу release():

void release()

void release(int permits)

Перший варіант методу звільняє один дозвіл, а другий варіант - кількість дозволів, зазначених у permits. Семафори відмінно підходять для вирішення завдань, де треба обмежувати доступ.

Простий семафор

Простий семафор має два методи take() і release(). Метод take() отримує сигнал та зберігає його у внутрішньому буфері. Метод release() постійно перебуває у стані чекання цього сигналу, і коли його отримує, відразу звільняє внутрішній буфер. Це дозволяє не втратити сигнал і підвищити стабільність системи. Навіть якщо виклик take() відбудеться раніше ніж release(), сигнал не буде втрачено, бо він зберігається у внутрішньому буфері семафора. Фактично методи take() і release() замінюють notify() і wait(), але завдяки буферизації, забезпечують більшу захищність та стабільність системи.

Використання семафорів для сигналізації

Семафори дуже зручно використовувати для передачі сигналів між процесами.

ЛІСТИНГ:

```
class SendingThread extends Thread {
  Semaphore semaphore = null;

  SendingThread( Semaphore semaphore) {
    this.semaphore = semaphore;
    System.out.println( "SendingThread created");
    }

  public void run() {
    System.out.println( "SendingThread started");
    for (int i = 0; i < 6; ++i) {
      System.out.println( "SendingThread do(" + i + ")");
      this.semaphore.take( i);
      }
    System.out.println( "SendingThread stoped");
    }
  }
```

```
class ReceivingThread extends Thread {
  Semaphore semaphore = null;

  ReceivingThread( Semaphore semaphore) {
    this.semaphore = semaphore;
    System.out.println( "ReceivingThread created");
    }

  public void run() {
    System.out.println( "ReceivingThread started");
    while(true) {
      try {
        int m = this.semaphore.release();
        System.out.println( "ReceivingThread do(" + m + ")");
        }
      catch (InterruptedException ie) {
        }
      }
    }
  }

class Semaphore {
  private boolean signal = false;
  private int message = 0;

  public synchronized void take( int m) {
    this.signal = true;
    this.notify();
    message = m;
    System.out.println( "Semaphore take");
    }

  public synchronized int release() throws InterruptedException{
    while(!this.signal) {
      wait();
      }
    this.signal = false;
    System.out.println( "Semaphore release");
    return( message);
    }
  }

 public class SignalingSemaphore {
   public static void main( String argc[]) {
     System.out.println( "Main process started");
     Semaphore semaphore = new Semaphore();
     SendingThread sender = new SendingThread( semaphore);
     ReceivingThread receiver = new ReceivingThread( semaphore);

     receiver.start();
     sender.start();
     System.out.println( "Main process ended");
     }
   }
```

Завдання № 1. Зробити налагодження коду семафору для сигналізації, навести результати у звіті виконання ПР № 6 та висновки.

Рахуючий семафор. Іноді виникає необхідність у підрахунку кількості надісланих / переданих сигналів.Для цього використовують так звані “рахуючі семафори”. У таких семафорах замість булевого буферу використовується цілочисельний.

ЛІСТИНГ:

```
class SendingThread extends Thread {
  CountingSemaphore semaphore = null;

  SendingThread( CountingSemaphore semaphore) {
    this.semaphore = semaphore;
    System.out.println( "SendingThread created");
    }

  public void run() {
    System.out.println( "SendingThread started");
    for (int i = 0; i < 6; ++i) {
      System.out.println( "SendingThread do(" + i + ")");
      this.semaphore.take( i);
      }
    System.out.println( "SendingThread stoped");
    }
  }

class ReceivingThread extends Thread {
  CountingSemaphore semaphore = null;

  ReceivingThread( CountingSemaphore semaphore) {
    this.semaphore = semaphore;
    System.out.println( "ReceivingThread created");
    }

  public void run() {
    int sum = 0;
    System.out.println( "ReceivingThread started");
    while(true) {
      try {
        int m = this.semaphore.release();
        sum += m;
        System.out.println( "ReceivingThread do(" + m + "): " + sum);
        }
      catch (InterruptedException ie) {
        }
      }
    }
  }
```

```
class CountingSemaphore {
  private int signals = 0;
  private int message = 0;

  public synchronized void take( int m) {
    this.signals++;
    message = m;
    this.notify();
    System.out.println( "CountingSemaphore(take): " + this.signals);
    }

  public synchronized int release() throws InterruptedException {
    while(this.signals == 0) {
      wait();
      }
    this.signals--;
    System.out.println( "CountingSemaphore(release): " + this.signals);
    return( message);
    }
  }

public class CountingSemaphoreDemo {
  public static void main( String argc[]) {
    System.out.println( "Main process started");
    CountingSemaphore semaphore = new CountingSemaphore();
    SendingThread sender1 = new SendingThread( semaphore);
    SendingThread sender2 = new SendingThread( semaphore);
    ReceivingThread receiver = new ReceivingThread( semaphore);

    receiver.start();
    sender1.start();
    sender2.start();
    System.out.println( "Main process ended");
    }
  }
```

Завдання № 2. Зробити налагодження коду рахуючого семафору, навести результати у звіті виконання ПР № 6 та висновки.

Обмежуючий семафор

Рахуючий семафор лише підраховує кількість комуніаційних повідомлень. Прицьому він не накладає ніяких обмежень на кількість процесів, які обмінюються повідомленнями або мають доступ до критичного ресурсу. Але дуже часто виникає необхідність у обмеженні кількості комунікацій між процесами, або між процесом і критичним ресурсом. Це можна здійснити за допомогою так званого “обмежуючого семафору”.

ЛІСТИНГ:

```
class SendingThread extends Thread {
  BoundedSemaphore semaphore = null;
  String name;
  long procNum;

  SendingThread( String n, long p, BoundedSemaphore semaphore) {
    this.semaphore = semaphore;
    name = n;
    procNum = p;
    System.out.println( name + ": created");
    }

  public void run() {
    System.out.println( name + ": started");
    for (int i = 0; i < 4; ++i) {
      System.out.println( name + ": do(" + (i * procNum) + ")");
      try {
        this.semaphore.take( i * procNum);
        }
      catch (InterruptedException ie) {
        }
      }
    System.out.println( name + ": stoped");
    }
  }

class ReceivingThread extends Thread {
  BoundedSemaphore semaphore = null;
  String name;
  int procNum;
  long sum = 0;

  ReceivingThread( String n, int p, BoundedSemaphore semaphore) {
    this.semaphore = semaphore;
    name = n;
    procNum = p;
    System.out.println( name + ": created");
    }

  public void run() {
    System.out.println( name + ": started");
    while(true) {
      try {
        long m = this.semaphore.release();

       sum += m;
       System.out.println( name + ": do(" + m + "), " + sum);
       }
     catch (InterruptedException ie) {
       }
     }
   }
 }
```

```
class BoundedSemaphore {
  private int signals = 0;
  private int bound = 0;
  private long message = 0;

  public BoundedSemaphore( int upperBound) {
    this.bound = upperBound;
    }

  public synchronized void take( long m) throws InterruptedException {
    while(this.signals == bound) {
      wait();
      }
    this.signals++;
    System.out.println( "BoundedSemaphore(take): " + this.signals);
    message = m;
    this.notify();
    }

  public synchronized long release() throws InterruptedException {
    while(this.signals == 0) {
      wait();
      }
    this.signals--;
    System.out.println( "BoundedSemaphore(release): " + this.signals);
    long m = message;
    message = 0;
    this.notify();
    return( m);
    }
  }
public class BoundedSemaphoreDemo {
  public static void main( String argc[]) {
    System.out.println( "Main process started");
    BoundedSemaphore semaphore = new BoundedSemaphore( 3);
    SendingThread sender1 = new SendingThread( "Sender 1", 1, semaphore);
    SendingThread sender2 = new SendingThread( "Sender 2", 10, semaphore);
    SendingThread sender3 = new SendingThread( "Sender 3", 100, semaphore);
    SendingThread sender4 = new SendingThread( "Sender 4", 1000, semaphore);
    SendingThread sender5 = new SendingThread( "Sender 5", 10000, semaphore);
    SendingThread sender6 = new SendingThread( "Sender 6", 100000, semaphore);
    ReceivingThread receiver = new ReceivingThread( "Receiver", 1, semaphore);

    receiver.start();
    sender1.start();
    sender2.start();
    sender3.start();
    sender4.start();
    sender5.start();
    sender6.start();
    System.out.println( "Main process ended");
    }
  }
```

Завдання № 3. Зробити налагодження коду обмежуючого семафору, навести результати у звіті виконання ПР № 6 та висновки.

Практична робота № 7

Тема роботи: Часова синхронізація паралельних процесів.

Мета роботи: Опанувати методику створення розподілених програм з використанням часової синхронізації паралельних процесів.

1.1 *Теоретичні відомості*

Клас **CountDownLatch** пакету **java.util.concurrent** використовується для того, щоб один чи декілька процесів змогли дочекатися виконання певної кількості операцій у інших процесах. Цей клас працює за принципом таймеру. Викопується ініціалізація його деяким початковим значенням і зворотній відлік. Під час виклику методу **await** цього класу деяким процесом, він переходить у етап чекання моменту досягнення рахівником таймера значення 0. На практиці цей клас зручно використовувати для координації моменту початку та закінчення певної кількості процесів. Це дозволяє зробити наступне:

- Запускати декілька процесів у один і той же момент часу.

- Відслідковувати момент закінчення роботи декількох процесів.

У наведеному прикладі (Лістинг 1) розглянемо початок виконання декількох процесів у один і той же момент часу. При створенні об’єкту **CountDownLatch,** його рахівник ініціалізується значенням 1. Усі процеси

```
import java.util.concurrent.*;

class LatchedThread extends Thread {
  private final CountDownLatch startLatch;
  private int procNumber;

  public LatchedThread( CountDownLatch s, int n) {
    startLatch = s;
    procNumber = n;
    System.out.println( "Thread(" + procNumber + "): created");
    }

  public void run() {
    System.out.println( "Thread(" + procNumber + "): started");
    try {
      startLatch.await();
      System.out.println( "Thread(" + procNumber + "): running");
      }
    catch(InterruptedException e) {
      System.err.println( e.toString());
      }
    System.out.println( "Thread(" + procNumber + "): stoped");
    }
  }
```

чекають моменту переходу цього рахівника у етап 0, після чого починають виконання свого коду.

```
public class CountDownLatchDemo {
  public static void main( String args[]) {
    System.out.println( "Main process started");
    CountDownLatch startLatch = new CountDownLatch( 1);
    for (int i = 0; i < 4; ++i) {
      LatchedThread t = new LatchedThread(startLatch, i+1);
      t.start();
      }
    try {
      Thread.sleep( 200);
      }
    catch( InterruptedException e) {
      System.err.println( e.toString());
      }
    startLatch.countDown();
    System.out.println( "Main process stoped");
    }
  }
```

Лістинг 1. Приклад використання класу CountDownLatch для запуску декількох процесів у один і той же момент часу

У наведеному прикладі (Лістинг 2) розглянемо процес відстеження моменту закінчення роботи декількох процесів. При створенні об'єкту **CountDownLatch,** його рахівнику привласнюється значення кількості процесів. Після цього головний процес переводиться до етапу чекання за допомогою методу **await.** Усі дочірні процеси перед закінченням виконання свого коду за допомогою методу **countDown** зменшують значення рахівника па одиницю. Після того, як значення рахівника етапе дорівнювати 0 головний процес виходить зі етапу чекання та продовжує виконання свого коду.

```
import java.util.concurrent.*;

class StopLatchedThread extends Thread {

    private final CountDownLatch stopLatch;
    private int procNumber;

    public StopLatchedThread(CountDownLatch s, int n) {
        stopLatch = s;
        procNumber = n;
        System.out.println(" Thread (" + procNumber + "): created ");
    }

    public void run() {
        System.out.println(" Thread (" + procNumber + "): started ");
        System.out.println(" Thread (" + procNumber + "): running ");
        stopLatch.countDown();
        System.out.println(" Thread (" + procNumber + "): stoped ");
    }
}
```

```
public class StopLatchedThreadDemo {

    public static void main(String args[]) {
        CountDownLatch stopLatch = new CountDownLatch(3);
        for (int i = 0; i < 3; ++i) {
            StopLatchedThread t = new StopLatchedThread(stopLatch, i + 1);
            t.start();
        }
        try {
            stopLatch.await();
        } catch (InterruptedException e) {
            System.err.println(e.toString());
        }
        System.out.println(" Main process stoped ");
    }
}
```

Лістинг 2. Приклад використання класу CountDownLatch для відстеження моменту закінчення роботи декількох процесів.

Завдання

1. Розглянути, відкомпілювати та запустити па виконання наведені приклади.

2. З'ясувати принципи вземодії та часової синхронізації процесів при використанні класу **CountDownLatch.**

3. З'ясувати особливості програмування та застосування класу **CountDownLatch.**

Індивідуальні варіанти завдань до захисту роботи

Примітка. Для синхронізації робочих процесів використати клас **CountDownLatch.** Кількість робочих процесів *n* ввести як параметр у командному рядку під час запуску програми.

1. Заповнити одновимірний масив значенням *True*, використовуючи n робочих процесів.

2. Підрахувати кількість елементів, які мають значення істина, у одновимірному булевому масиві, використовуючи n робочих процесів.

3. Підрахувати кількість елементів, які мають значення істина, у

двовимірній булевій матриці, використовуючи n робочих процесів.

4. Підрахувати кількість елементів, які мають значення істина, у тривимірному булевому кубі даних, використовуючи n робочих процесів.

5. У цілочисельному масиві знайти середнє арифметичне, використовуючи n робочих процесів.

6. У цілочисельному масиві знайти медіану (середнє найменшого й найбільшого значень), використовуючи n робочих процесів.

7. Виконати інверсію елементів одновимірного масиву за допомогою n паралельних процесів.

8. Підрахувати суму елементів цілочисельної матриці за допомогою n паралельних процесів.

9. У двовимірному масиві знайти задане число за допомогою n паралельних процесів.

10. Дано одновимірний масив цілих чисел. За допомогою n паралельних процесів здійснити у ньому такі зміни: 3->6, 8->*12*, 10->5. Вивести на екран початковий масив, модифікований масив і загальну кількість змін. Результат подати у вигляді таблиці.

11. За допомогою n паралельних процесів у послідовності цифр знайти номер першого елементу, який менше заданого числа.

12. Дано масив натуральних чисел. За допомогою n паралельних процесів перевірити, чи не складається він з нулів.

13. Дано послідовність упорядкованих по зростанню натуральних чисел. За допомогою n паралельних процесів визначити, чи можна вставити у неї елемент.

14. За допомогою n паралельних процесів у послідовності цифр знайти номер останнього негативного числа.

15. Дано рядок якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів знайти усі слова, у яких літера ’а’ входить не менше двох разів.

16. Дано рядок, за допомогою n паралельних процесів знайти найбільшу

кількість цифр, які йдуть у ньому підряд.

17. За допомогою n паралельних процесів знайти найдовшу групу цифр у рядку. Якщо таку довжину мають декілька груп, то взяти першу за чергою.

18. Дано рядок, якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів у словах, які закінчуються на ’ing’ змінити закінчення на ’ed’ і розташувати їх у кінці рядка у черзі, яка зворотна їх появі у вхідному рядку.

19. Дано рядок, якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів знайти слово у якому кількість літер ’a’ та ’b’ максимальна.

20. Дано рядок, якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів знайти слово, яке містить найбільшу кількість голосних літер (на початку та на кінці рядка проміжків нема).

21. Дано рядок. За допомогою n паралельних процесів видалити кожен символ, який зустрічається підряд більш, ніж один раз.

22. Дано рядок у якому є однакові символи, що йдуть один за одним. За допомогою n паралельних процесів визначити, скільки разів зустрічається таке поєднання.

23. За допомогою n паралельних процесів знайти суму цілих позитивних чисел кратних 3, що більше *А* і менше *Б (А < Б).*

24. За допомогою n паралельних процесів знайти суму цілих позитивних непарних чисел, менше 1000.

25. За допомогою n паралельних процесів знайти суму залишків від ділення на 5 усіх цілих позитивних чисел менше A (A >> 5).

26. Написати паралельну програму для транспонування матриці розміром 4n x 4n елементів n процесами.

Практична робота № 8

Тема: Відстеження моменту закінчення роботи декількох потоків

Мета: Опанувати методику створення розподілених програм з використанням часової синхронізації паральних потоків.

Теоретичні відомості

Клас **CountDownLatch** пакету **java.util.concurrent** використовується для того, щоб один чи декілька потоків змогли дочекатися виконання певної кількості операцій у інших потоках. Цей клас працює за принципом таймеру. Виконується ініціалізація його деяким початковим значенням і зворотній відлік.

Під час виклику методу **await()** цього класу деяким потоком, він переходить у стан чекання моменту досягнення рахівником таймера значення 0.

На практиці цей клас зручно використовувати для координації моменту початку та закінчення певної кількості потоків.

Це дозволяє зробити наступне:

- Запускати декілька потоків у один і той же момент часу.
- Відслідковувати момент закінчення роботи декількох потоків.

Приклад використання класу CountDownLatch

У наведеному прикладі розглянемо початок виконання декількох потоків у один і тойже момент часу.

При створенні об'єкту **CountDownLatch**, його рахівник ініціалізується значенням 1. Усі потки чекають моменту переходу цього рахівника у стан 0, після чого починають виконання свого коду.

ЛІСТИНГ:

```
import java.util.concurrent.*;

class LatchedThread extends Thread {
  private final CountDownLatch startLatch;
  private int procNumber;

  public LatchedThread( CountDownLatch s, int n) {
    startLatch = s;
    procNumber = n;
    System.out.println( "Thread(" + procNumber + "): created");
    }
```

```
  public void run() {
    System.out.println( "Thread(" + procNumber + "): started");
    try {
      startLatch.await();
      System.out.println( "Thread(" + procNumber + "): running");
      }
    catch(InterruptedException e) {
      System.err.println( e.toString());
      }
    System.out.println( "Thread(" + procNumber + "): stoped");
    }
  }

public class CountDownLatchDemo {
  public static void main( String args[]) {
    System.out.println( "Main process started");
    CountDownLatch startLatch = new CountDownLatch( 1);
    for (int i = 0; i < 4; ++i) {
      LatchedThread t = new LatchedThread(startLatch, i+1);
      t.start();
      }
    try {
      Thread.sleep( 200);
      }
    catch( InterruptedException e) {
      System.err.println( e.toString());
      }
    startLatch.countDown();
    System.out.println( "Main process stoped");
    }
  }
```

РЕЗУЛЬТАТ:

```
run:
Main process started
Thread(1): created
Thread(2): created
Thread(3): created
Thread(4): created
Thread(1): started
Thread(2): started
Thread(3): started
Thread(4): started
Main process stoped
Thread(4): running
Thread(3): running
Thread(3): stoped
Thread(4): stoped
Thread(1): running
Thread(1): stoped
Thread(2): running
Thread(2): stoped
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

У наведеному прикладі (Лістинг 2) розглянуто процес відстеження моменту закінчення роботи декількох процесів. При створенні об'єкту CountDownLatch, його рахівнику привласнюється значення кількості процесів. Після цього головний процес переводиться до стану чекання за допомогою методу await. Усі дочірні процеси перед закінченням виконання свого коду за допомогою методу countDown зменшують значення рахівника на одиницю. Після того, як значення рахівника стане дорівнювати 0 головний процес виходить зі стану чекання та продовжує виконання свого коду.

ЛІСТИНГ 2:

```
import java.util.concurrent.*;

class StopLatchedThread extends Thread {

    private final CountDownLatch stopLatch;
    private int procNumber;

    public StopLatchedThread(CountDownLatch s, int n) {
        stopLatch = s;
        procNumber = n;
        System.out.println(" Thread (" + procNumber + "): created ");
    }

    public void run() {
        System.out.println(" Thread (" + procNumber + "): started ");
        System.out.println(" Thread (" + procNumber + "): running ");
        stopLatch.countDown();
        System.out.println(" Thread (" + procNumber + "): stoped ");
    }
}

public class StopLatchedThreadDemo {

    public static void main(String args[]) {
        CountDownLatch stopLatch = new CountDownLatch(3);
        for (int i = 0; i < 3; ++i) {
            StopLatchedThread t = new StopLatchedThread(stopLatch, i + 1);
            t.start();
        }
        try {
            stopLatch.await();
        } catch (InterruptedException e) {
            System.err.println(e.toString());
        }
        System.out.println(" Main process stoped ");
    }
}
```

РЕЗУЛЬТАТ:

```
run:
 Thread (1): created
 Thread (2): created
 Thread (3): created
 Thread (1): started
 Thread (1): running
 Thread (2): started
 Thread (2): running
 Thread (1): stoped
 Thread (2): stoped
 Thread (3): started
 Thread (3): running
 Thread (3): stoped
 Main process stoped
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Іноді потрібно, щоб потік-виконання знаходився в режимі очікування до тих пір, поки не настане одна (або більше) подія.

Для цих цілей в паралельному API наявний клас CountDownLatch, що реалізує самоблокування зі зворотним відліком. Об'єкт цього класу спочатку створюється з кількістю подій, які повинні відбутися до того моменту, як буде зняте самоблокування. Кожного разу, коли відбувається подія, значення лічильника зменшується.

Як тільки значення лічильника досягне нуля, самоблокування буде зняте.У класі CountDownLatch є наведений нижче конструктор, де параметр число визначає кількість подій, які повинні відбутися до того, як буде зняте самоблокування.

CountDownLatch (int число)

Для очікування самоблокування в потоці-виконання викликається метод await(), загальні форми якого наведені нижче.

void await() throws InterruptedException

boolean await (long очікування, TimeUnit одиниця_часу) throws InterruptedException

У першій формі очікування триває до тих пір, поки відлік, пов'язаний з викликає об'єктом типу CountDownLatch, не досягне нуля. А в другій формі очікування триває тільки протягом певного періоду часу, що визначається параметром очікування.

Час очікування вказується в одиницях, що позначаються параметром **одиниця_часу**, який приймає об'єкт перерахування **TimeUnit**.

Метод **await()** повертає логічне значення false, якщо досягнута межа часу очікування, або логічне значення true, якщо зворотний відлік досягає нуля.

Щоб сповістити про подію, слід викликати метод **countDown()**. Кожного разу, коли викликається метод countDown(), відлік, пов'язаний з об'єктом, що викликається, зменшується на одиницю.

void countDown()

У тілі методу main() встановлюється самоблокування у вигляді об'єкта cdl типу CountDownLatch з вихідним значенням зворотного відліку, рівним 5.

Потім створюється екземпляр класу MyThread, який починає виконання нового потоку. Зверніть увагу на те, що об'єкт cdl передається в якості параметра конструктору класу MyThread і зберігається в змінній екземпляра latch.

Далі в головному потоці виконання викликається метод await() для об'єкта cdl, в результаті чого виконання головного потоку припиняється до тих пір, поки зворотний відлік самоблокування не зменшиться на одиницю п'ять разів в об'єкті cdl.

У тілі методу run() з класу MyThread організовується цикл, який повторюється п'ять разів. На кожному кроці цього циклу викликається метод countDown() для змінної екземпляра latch, яка посилається на об'єкт cdl в методі main(). По завершенні п'ятого кроку циклу самоблокування знімається, дозволяючи відновити головний потік виконання.

Клас **CountDownLatch** є ефективним і простим у використанні засобом синхронізації, який виявиться корисним в тих випадках, коли потік виконання повинен перебувати в стані очікування до тих пір, допоки не відбудеться одна або кілька подій.

Індивідуальні варіанти завдань до захисту роботи

Примітка. Для синхронізації робочих процесів використати клас **CountDownLatch.** Кількість робочих процесів *n* ввести як параметр у командному рядку під час запуску програми.

1. Заповнити одновимірний масив значенням *True,* використовуючи n робочих процесів.
2. Підрахувати кількість елементів, які мають значення істина, у одновимірному булевому масиві, використовуючи n робочих процесів.
3. Підрахувати кількість елементів, які мають значення істина, у двовимірній булевій матриці, використовуючи n робочих процесів.
4. Підрахувати кількість елементів, які мають значення істина, у тривимірному булевому кубі даних, використовуючи n робочих процесів.
5. У цілочисельному масиві знайти середнє арифметичне, використовуючи n робочих процесів.
6. У цілочисельному масиві знайти медіану (середнє найменшого й найбільшого значень), використовуючи n робочих процесів.
7. Виконати інверсію елементів одновимірного масиву за допомогою n паралельних процесів.
8. Підрахувати суму елементів цілочисельної матриці за допомогою n паралельних процесів.
9. У двовимірному масиві знайти задане число за допомогою n паралельних процесів.
10. Дано одновимірний масив цілих чисел. За допомогою n паралельних процесів здійснити у ньому такі зміни: 3->6, 8->*12*, 10->5. Вивести на екран початковий масив, модифікований масив і загальну кількість змін. Результат подати у вигляді таблиці.
11. За допомогою n паралельних процесів у послідовності цифр знайти номер першого елементу, який менше заданого числа.

12. Дано масив натуральних чисел. За допомогою n паралельних процесів перевірити, чи не складається він з нулів.

13. Дано послідовність упорядкованих по зростанню натуральних чисел. За допомогою n паралельних процесів визначити, чи можна вставити у неї елемент.

14. За допомогою n паралельних процесів у послідовності цифр знайти номер останнього негативного числа.

15. Дано рядок якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів знайти усі слова, у яких літера ’a’ входить не менше двох разів.

16. Дано рядок, за допомогою n паралельних процесів знайти найбільшу кількість цифр, які йдуть у ньому підряд.

17. За допомогою n паралельних процесів знайти найдовшу групу цифр у рядку. Якщо таку довжину мають декілька груп, то взяти першу за чергою.

18. Дано рядок, якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів у словах, які закінчуються на ’ing’ змінити закінчення на ’ed’ і розташувати їх у кінці рядка у черзі, яка зворотна їх появі у вхідному рядку.

19. Дано рядок, якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів знайти слово у якому кількість літер ’a’ та ’b’ максимальна.

20. Дано рядок, якій складається з слів (послідовність символів, які обмежені проміжками). За допомогою n паралельних процесів знайти слово, яке містить найбільшу кількість голосних літер (на початку та на кінці рядка проміжків нема).

21. Дано рядок. За допомогою n паралельних процесів видалити кожен символ, який зустрічається підряд більш, ніж один раз.

22. Дано рядок у якому є однакові символи, що йдуть один за одним. За допомогою n паралельних процесів визначити, скільки разів зустрічається таке поєднання.

23. За допомогою n паралельних процесів знайти суму цілих позитивних чисел кратних 3, що більше *А* і менше *Б (А < Б)*.

24. За допомогою n паралельних процесів знайти суму цілих позитивних непарних чисел, менше 1000.

25. За допомогою n паралельних процесів знайти суму залишків від ділення на 5 усіх цілих позитивних чисел менше A (A >> 5).

26. Написати паралельну програму для транспонування матриці розміром 4n x 4n елементів n процесами.

Практична робота № 9

Тема роботи: Використання обмінників.

Мета роботи: Опанувати методику створення розподілених програм з використанням обмінників.

Теоретичні відомості

Обмінники— це засоби синхронізації, які використовуються для гарантованого обміну інформацією між двома процесами. Обмінники є, також, синхронними засобами комунікації між двома процесами. Це означає, що процес-відправник даних буде знаходитися у стані чекання доти, доки процес-отримувач даних їх не отримає. Обмінник є дуже простим класом, який може пов'язати тільки два процеси, та за один сеанс зв'язку може передати лише один об'єкт даних.

У мові Java обмінники представлені класом Exchanger. Обмін даними відбувається під час виклику його методу exchange. При цьому, обмінник може працювати у таких режимах:

- режим передачі даних,
- режим прийому у даних,
- режим одночасного прийому та передачі даних.

У режимі передачі даних у якості параметра методу exchange треба передати дані у вигляді примірника інтерфейсного класу (DataClass). Наприклад так:

```
exchanger.exchange( new DataClass( parametersList ));
```

або так:

```
DataClass data = new DataClass(parametersList );
exchanger.exchange(data);
```

Значення, що повертає метод exchange, при цьому можна не приймати.

У режимі прийому даних, навпаки важливе саме значення, яке поверта№ метод exchange:

```
DataClass newData = exchanger.exchange(null);
```

У режимі одночасного прийому та передачі даних метод exchange можна

застосувати так:

```
DataClass data = new DataClass( new DataClass(parametersList));
```

або так:

```
DataClass data = new DataClass( parametersList );
DataClass newData = exchanger.exchange( data);
```

Розглянемо приклад організації синхронного обміну даними між двома процесами loop1 і loop2. Вони е примірниками класу Loop (лістинг 1). Особливість тут полягає у тому, що спочатку треба визначитись з даними, які підлягають обміну та форматом їх передачі. Для цього треба створити інтерфейсний клас. У наведеному лістингу це DataClass. Під час обміну буде передаватися числове значення (int value;) та рядок тексту (String message;). Примірнику інтерфейсного класу вони будуть передаватися під час його створення за допомогою конструктора (new DataClass(value, textString);). Після отримання інтерфейсного класу процес-приймач зможе отримати ці дані за допомогою методів getValue та getMassage.

ЛІСТИНГ:

```
import java.util.concurrent.*;
class DataClass {
    int value;
    String message;
    DataClass(int v, String s) {
        value = v;
        message = s;
    }
    int getValue() {
        return (value);
    }
    String getMassage() {
        return (message);
    }
}
class Loop implements Runnable {
    int counter;
    String name;
    Exchanger<DataClass> exchanger;
    Loop(int startValue, String id, Exchanger< DataClass> ex) {
        counter = startValue;
        name = id;
        exchanger = ex;
        System.out.println(name + ": created ");
    }
```

```
    public void run() {
        System.out.println(name + ": started ");
        DataClass data = new DataClass(counter, name);
        for (int i = 0; i < 3; ++i) {
            try {
                DataClass newData = exchanger.exchange(data);
                counter += newData.getValue();
                System.out.println(name + ": from "
                        + newData.getMassage() + ": data : "
                        + newData.getValue() + ": state = " + counter);
            } catch (InterruptedException e) {
                System.err.println(e.toString());
            }
        }
        System.out.println(name + ": ended ");
    }
}
public class ExchangerDemo {

    public static void main(String args[]) {
        System.out.println(" Main process started ");
        Exchanger< DataClass> exchanger = new Exchanger< DataClass>();
        Loop loop1 = new Loop(1, " First ", exchanger);
        Loop loop2 = new Loop(2, " Second ", exchanger);
        new Thread(loop1).start();
        new Thread(loop2).start();
        System.out.println(" Main process ended ");
    }
}
```

Робота головного процесу (ExchangerDemo) складається з трьох головних етапів. На першому етапі створюється примірник обмінника exchanger для інтерфейсного класу DataClass. На другому створюються два робочі процеси loop1 і loop2, які є примірниками класу Loop. При ініціалізації їм у якості параметрів надається деяке числове значення (у лістингу 1 це номер процесу, який є одночасно і початковим значенням рахівника сеансів обміну даними) та текстовий рядок (у лістингу 1 це власна назва процесу), а також посилання на примірник обмінника, створений раніше. На третьому етапі процеси loop1 і loop2 запускаються на виконання. Під час роботи програми відбуваються три сеанси обміну даними між робочими процесами loop1 і loop2. Кожен з них створює свої власні примірники інтерфейсного класу DataClass для прийому та передачі даних та тричі викликає метод exchange для синхронного обміну даними з іншим процесом.

Завдання

1. Розглянути, відкомпілювати та запустити па виконання наведений приклад.

2. З’ясувати принципи взаємодії та часової синхронізації процесів при використанні обмінників.

3. З’ясувати особливості програмування та застосування обмінників.

Індивідуальні варіанти завдань до захисту роботи

1. Створити три процеси та організувати обмін даними між ними за топологією «Лінійка». Тобто перший процес передає дані другому, а другий третьому.

2. Створити три процеси та організувати циклічний обмін даними між ними. Тобто перший процес повинен передавати дані до другого, другий до третього, а третій до першого.

3. Створити чотири процеси та організувати обмін даними між ними за топологією «Зірка».

Тобто один (головний) процес повинен роздати дані іншим трьом (підлеглим) процесам.

4. Розробити паралельну програму конвеєр даних для табулювання функції $y = 1 + 1/(1 + 1/(1 + 1/(1 + 1/x)))$.

5. Розробити паралельну програму конвеєр даних для табулювання функції $y = a * (1 + a * (1 + a * (1 + a * (1 + a))))$.

6. Розробити паралельну програму конвеєр даних для табулювання функції $y = a * x/(a * x/(a * x/(a * x)))$.

7. Розробити паралельну програму конвеєр даних для табулювання функції $y = a/(x + a/(x + a/(x + a/(x + a/x))))$.

8. Розробити паралельну програму конвеєр даних для табулювання функції $y = a * b/(x + a * b/(x + a * b/(x + a * b/(x + a * b/x))))$.

9. Розробити паралельну програму конвеєр даних для табулювання функції $y = 1 + (1 + (1 + (1 + x)/x)/x)/x$.

10. Розробити паралельну програму конвеєр даних для табулювання функції $y = 2 * (2 * (2 * (2 * x - 1) * x - 1) * x - 1) * x - 1.$

11. Розробити паралельну програму конвеєр даних для табулювання функції $y = a/(a/(a/(a/x - b) - b) - b) - b.$

12. Розробити паралельну програму конвеєр даних для табулювання функції $y = a * (b/(a * (b/(a * (b/x) + c)) + c)) + c.$

13. Написати паралельну програму, яка побайтно читає вхідний файл filename1 і запису його у стандартний вихід, а також у файл filename2, тобто створює дві копії вхідних даних.

Розпаралелити програму так, щоб вона використовувала три процеси:

а) читання з файлу filename1,

б) запису у стандартний вивід,

в) запису у файл filelename2.

Для передачі даних між процесами використати обмінники.

Практична робота № 10

Тема роботи: “Використання портфеля задач”.

Мета роботи: Опанувати методику створення розподілених програм з використанням портфеля задач.

1.1 Теоретичні відомості

Портфель задач - неупорядковане сховище задач, що чекають на виконання. Задачі кладуться у портфель, що розподілений між декількома робочими процесами. Кожен робочий процес виконує такий основний код:

```
while (true) {
отримати задачу з портфеля;
if (задач більш нема)
break; // вихід з циклу while
виконати задачу, можливо, породжуючи нові задачі;
}
```

Рис. 1. Алгоритм робочого процесу для парадигми портфелю задач

Цей підхід можна використовувати для:

- реалізації рекурсивного паралелізму;

- вирішення ітеративних проблем з фіксованою кількістю незалежних задач.

Парадигма портфелю задач має такі корисні властивості: Вона дуже проста у використанні. Достатньо визначити представлення задачі, реалізувати портфель, запрограмувати виконання задачи і визначити умови завершення роботи алгоритму. Програми, що використовують портфель задач є масштабуємими у тому сенсі, що їх можна використовувати з будь-якою кількістю процесорів. Для цього достатньо просто змінити кількість робочих процесів.

Спрощується реалізація балансування навантаження. Якщо час виконання різних задач різний, то деякі з задач будуть виконуватися довше інших. Але поки задач більше ніж робочих процесів (у 2 - 3 рази), загальні об'єми обчислень, що виконуються робочими процесорами, будуть приблизно однаковими.

1.2 Приклад використання портфеля задач за допомогою

ExecutorService

У лістингу 1 за допомогою виклику Executors.newFixedThreadPool було створено пул на 5 процесів. Тепер, у разі появи великої кількості задач, вони будуть виконуватися вже існуючими процесами. Тим самим, буде економитись час на створення нових процесів за рахунок більш ефективного використання вже існуючих процесів з цього пулу.

ЛІСТИНГ:

```
import java.util.concurrent.*;
import java.util.Random;
class CallableImpl implements Callable<Integer> {
    private static final Random rand = new Random();
    int threadNumber = 0;
    public void setThreadNumber(int num) {
        threadNumber = num;
    }
    public Integer call() {
        System.out.println(" Callable task (" + threadNumber+ ") begin ");
        busy();
        System.out.println(" Callable task (" + threadNumber+ ") end ");
        return new Integer(threadNumber);
    }
    private void busy() {
        try {
            Thread.sleep(rand.nextInt(500));
        } catch (InterruptedException e) {
        }
    }
}
public class ExecutorServiceDemo {
    public static void main(String args[]) {
        CallableImpl callable[] = new CallableImpl[10];
        Future future[] = new Future[10];
        ExecutorService executor
                = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; ++i) {
            callable[i] = new CallableImpl();
            callable[i].setThreadNumber(i + 1);
            future[i] = executor.submit(callable[i]);
        }
        for (int i = 0; i < 10; ++i) {
            try {
                System.out.println(" Future value : "
                        + future[i].get());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        executor.shutdown();
    }
}
```

Задача від процесу відрізняється тим, що може повертати результат обчислення до батьківського процесу. Для цього її треба імплементувати від інтерфейсу **Callable**. Цей інтерфейс вимагає, щоб усі дії, пов'язані з обчисленнями задачі виконувалися у публічному методі **call**. Цей метод, на відміну від методу **run** може повертати значення результат обчислення.

Для того, щоб батьківський процес отримав результат обчислення від задачі, використовується синхронний об'єкт класу **Future**. Кожен примірник цього класу пов'язується з відповідною задачею за допомогою методу **submit** класу **ExecutorService** під час додавання цієї задачі до портфелю.

Після того, як задача додана до портфелю, її може виконати будь-який з його робочих процесів. Це може відбутися у буд-який момент часу. Тому виклик методу **get** у відповідного об'єкта типу **Future** змушує текучий процес очікувати результат виконання робочим процесом цієї задачі.

Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.

2. З'ясувати принципи взамодії та синхронізації процесів при використанні портфеля задач.

3. З'ясувати особливості програмування та застосування портфеля задач.

Індивідуальні варіанти завдань до захисту роботи

1. Написати паралельну програму для генерації послідовності чисел, які менше за 10000 та діляться на 3, 5, 7, 9. Використати два процеси та чотири задачі.

2. Обчислити визначений інтеграл для функції $y = \frac{x^2}{4}$ у діапазоні [2; 10]. Використати два процеси та чотири задачі.

3. Обчислити довжину шляху, пройденого матеріальною точкою за t = 48 секунд, яка рухається за таким законом:

$$\begin{cases} y = 16 * \sin(2 * t), \\ x = 4 * t. \end{cases}$$

Використати три процеси та дев'ять задач.

4. Гра «Хто більше». Використовуючи механізм синхронізації портфель задач, створити два процеси та десять задач. Кожна задача генерує випадкове ціле число у діапазоні від 0 до 100 та закінчує свою роботу. Після вичерпання портфелю задач головна програма збирає згенеровані дані, та з'ясовує, яка із задач згенерувала найбільше число та виводить його у консоль разом із номером цієї задачі.

5. Дано масив, заповнений випадковими цілими числами від 0 до 10000. Використовуючи механізм синхронізації портфель задач, створити три процеси та десять задач для пошуку у цьому масиві заданого числа. Довжина масиву повинна бути кратна кількості задач. Забезпечити обробку ситуацій, коли шукане число не буде знайдене, або буде знайдено декілька таких чисел.

6. Дано масив, заповнений цілими числами від 0 до 10000. Використовуючи механізм синхронізації портфель задач, з'ясувати, чи відсортований він за зростанням. Створити чотири процеси та десять задач. Забезпечити вивід у консоль з головної програми повідомлення про те у якому з сегментів масиву порушується порядок (якщо таке є).

7. Дано масив, заповнений випадковими цілими числами від 0 до 15000. Використовуючи механізм синхронізації портфель задач, підрахувати кількість парних чисел у ньому. Створити три процеси та п'ятнадцять задач. Забезпечити вивід діагностики роботи програми у консоль.

8. Дано кардіоїда вписана у квадрат. Обрахувати співвідношення площин кардіоїди та квадрата методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

9. Дано коло вписане у квадрат. Обрахувати співвідношення площин кола та квадрата методом Монте-Карло. Використати два процеси та вісім задач.

10. Дано квадрат вписаний у коло. Обрахувати співвідношення площин кола та квадрата методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

11. Дано трикутник вписаний у коло. Обрахувати співвідношення площин кола та трикутника методом Монте-Карло. Використати три процеси та шістнадцять задач.

12. Дано коло вписане у трикутник. Обрахувати співвідношення площин кола та трикутника методом Монте-Карло. Використати три процеси та шістнадцять задач.

13. Дано зірка вписана у коло. Обрахувати співвідношення площин кола та зірки методом Монте-Карло. Використати три процеси та дев'ять задач.

14. Дано сфера вписана у куб. Обрахувати співвідношення об'ємів сфери та куба методом Монте-Карло. Використати три процеси та дев'ять задач.

15. Дано куб вписаний у сферу. Обрахувати співвідношення об'ємів сфери та куба методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

16. Дано сфера вписана у тетраедр1. Обрахувати співвідношення об'ємів сфери та тетраедра методом Монте-Карло. Використати три процеси та дев'ять задач.

17. Дано тетраедр вписаний у сферу. Обрахувати співвідношення об'ємів сфери та тетраедра методом Монте-Карло. Використати чотири процеси та шістнадцять задач.

Практична робота № 11

Тема роботи: Розподілене програмування на мові Java з використанням графа операції-операнди

Мета роботи: Опанувати методику побудови моделі обчислень у вигляді графа операції-операнди.

1.1 Теоретичні відомості

1.1.1 Модель обчислень у вигляді графа „операції-операнди“ дуже часто використовуватися для опису існуючих інформаційних залежностей у паралельних алгоритмах. Уявімо множину операцій, які виконуються в досліджуваному алгоритмі рішення обчислювальної задачі, і існуючі між операціями інформаційні залежності у вигляді ациклічного орієнтованого графа G = (V,E), де V = 1,…,*V* - є множина вершин графа, що уявляє виконувані операції алгоритму, а E множина дуг графа (при цьому дуга e = (i, j) належить графу тільки, якщо операція j використовує результат виконання операції *i)*. На рисунку Рис. 2 наведено граф алгоритму обчислення площі прямокутника, заданого координатами двох кутів (Рис. 1) за формулою

s = (x2*y2 - x2*y1) + (x1*y1 - x1*y2).

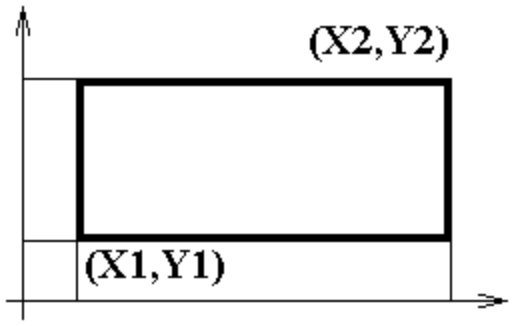


Рис. 11.1 - Прямокутник, заданий координатами своїх кутів

1.1.2 Приклад паралельного алгоритму для обчислення площі прямокутника

Представлена тут програма ілюструє простий, але цікавий приклад використання обчислювальної моделі алгоритму у вигляді графа "операції-операнди" для побудови конвейера команд. Розглянемо задачу обчислення площі прямокутника, заданого координатами своїх кутів (Рис.1). Площу

такого прямокутника можна обрахувати за такою формулою:

s = (x2 - x1)*(y2 - y1).

Розпаралелимо її:

s = (x2 -x1)*(y2 - y1) = (x2*y2-x2*y1) + (x1*y1-x1*y2).

Побудуємо граф "операції-операнди" (Рис. 2).

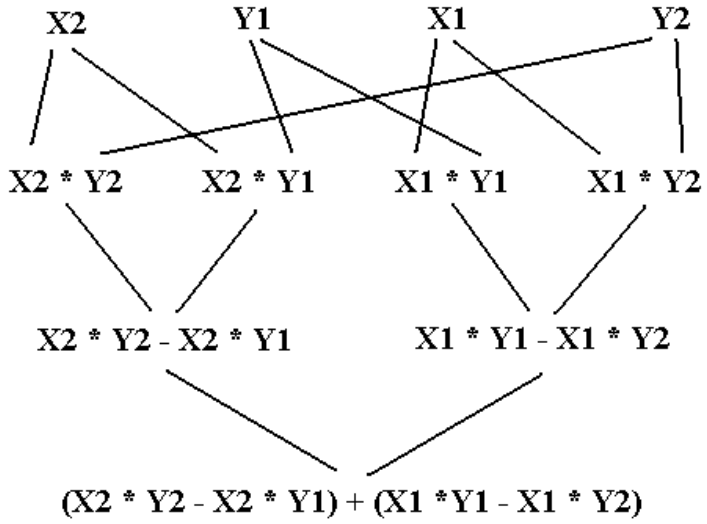


Рис.11.2 - Граф "операції-операнди" для обчислення площі прямокутника

У верхньому рядочку графа розташовані операнди, значення яких подаються на вхід процесів, що виконують операцію множення. Результати цих операцій передаються процесам, які виконують операцію віднімання. На заключному етапі роботи результати віднімання збирає останній процес і виконує операцію їх складання. Для реалізації такого алгоритму створимо клас Calculator, який буде виконувати елементарні математичні дії. Для того, щоб цей клас можна було запускати як окремий процес, він повинен імплементувати інтерфейс Runnable. Математичні операції, які необхідно виконати визначаються константами opSum, opSub, opMult та opDiv. У об'єкт класу Calculator код операції передається за допомогою параметру cop. Початкові дані, проміжні та кінцеві результати зберігаються у лінійному масиві data класу RectangleSquareCalc. Цей масив є розподіленим ресурсом. Класу Calculator під час створення передаються лише індекси відповідних комірок у цому масиві. А саме:

- op1 - індекс першого операнду;

- op2 - індекс другого операнду;

- res - індекс результату виконання операції.

Крім того, кожному примірнику класа Calculator передається його власне ім'я name та посилання на батьківський об'єкт-менеджер RectangleSquareCalc. Програмна реалізація класу Calculator на алгоритмічній мові Java наведена у лістингу 1.

ЛІСТИНГ 1:

```
import java.util.concurrent.*;
class Calculator implements Runnable {
    public final static int opSum = 1;
    public final static int opSub = 2;
    public final static int opMult = 3;
    public final static int opDiv = 4;
    int codeOp;
    int operator1;
    int operator2;
    int result;
    String threadName;
    RectangleSquareCalc manager;
    Calculator(String name, int cop, int op1, int op2, int res,
            RectangleSquareCalc rsc) {
        threadName = name;
        codeOp = cop;
        operator1 = op1;
        operator2 = op2;
        result = res;
        manager = rsc;
        System.out.println(threadName + " - Created ");
    }
    public void run() {
        System.out.println(threadName + " - Start of Work ");
        while (!manager.stopFlag) {
            if (manager.flags[operator1] && manager.flags[operator2]) {
                switch (codeOp) {
                    case opSum:
                        manager.data[result] = manager.data[operator1] + manager.data[operator2];
                        System.out.println(threadName + ": Sum : " + operator1 + " + "
                                + operator2 + " ==> " + result);
                        break;
                    case opSub:
                        manager.data[result] = manager.data[operator1] - manager.data[operator2];
                        System.out.println(threadName + ": Sub : " + operator1 + " + "
                                + operator2 + " ==> " + result);
                        break;
```

```
                    case opDiv:
                        manager.data[result] = manager.data[operator1] / manager.data[operator2];
                        System.out.println(threadName + ": Div : " + operator1 + " + "
                                + operator2 + " ==> " + result);
                        break;
                    default:
                        break;
                }
                manager.flags[operator1] = false;
                manager.flags[operator2] = false;
                manager.flags[result] = true;
            }
            Thread.yield();
        }
    }
}
```

У процесі роботи кожен примірник класу Calculator аналізує наданий йому код операції, бере операнди із загального масиву, виконує над ними вказану операцію та кладе результат у загальний масив за його індексом. Налагодження портфеля задач згідно з наведеним вище графом операції-операнди виконаємо за допомогою класу RectangleSquareCalc.

```
public class RectangleSquareCalc {
    double data[];
    boolean flags[];
    boolean stopFlag = false;
    RectangleSquareCalc(double x1, double y1, double x2, double y2) {
        data = new double[15];
        flags = new boolean[15];
        int pos = 0;
        flags[pos] = true;
        data[pos++] = x2;
        flags[pos] = true;
        data[pos++] = y2;
        flags[pos] = false;
        data[pos++] = 0.0;
        flags[pos] = true;
        data[pos++] = x2;
        flags[pos] = true;
        data[pos++] = y1;
        flags[pos] = false;
        data[pos++] = 0.0;
        flags[pos] = true;
        data[pos++] = x1;
        flags[pos] = true;
        data[pos++] = y2;
        flags[pos] = false;
        data[pos++] = 0.0;
        flags[pos] = true;
        data[pos++] = x1;
        flags[pos] = true;
        data[pos++] = y1;
        flags[pos] = false;
        data[pos++] = 0.0;
    }
```

```
    public static void main(String argc[]) {
        System.out.println(" Main process started ");
        RectangleSquareCalc rsc = new RectangleSquareCalc(1.0, 2.0, 3.0, 4.0);
        ExecutorService execSvc = Executors.newFixedThreadPool(7);
        execSvc.execute(new Calculator(" Mult_1 ", Calculator.opMult, 0, 1, 2, rsc));
        execSvc.execute(new Calculator(" Mult_2 ", Calculator.opMult, 3, 4, 5, rsc));
        execSvc.execute(new Calculator(" Mult_3 ", Calculator.opMult, 6, 7, 8, rsc));
        execSvc.execute(new Calculator(" Mult_4 ", Calculator.opMult, 9, 10, 11, rsc)
        );
        execSvc.execute(new Calculator(" Sub_5 ", Calculator.opSub, 2, 5, 12, rsc));
        execSvc.execute(new Calculator(" Sub_6 ", Calculator.opSub, 11, 8, 13, rsc));
        execSvc.execute(new Calculator(" Sum_7 ", Calculator.opSum, 12, 13, 14, rsc));
        while (!rsc.flags[14]) {
            try {
                Thread.sleep(2);
            } catch (InterruptedException e) {
            }
        }
        rsc.stopFlag = true;
        System.out.println();
        for (int i = 0; i < 4; ++i) {
            for (int j = 0; j < 3; ++j) {
                System.out.print("( " + (3 * i + j) + " ): " + rsc.data[3 * i + j] + ",");
            }
            System.out.println();
        }
        System.out.println("( " + 12 + " ): " + rsc.data[12] + ",");
        System.out.println("( " + 13 + " ): " + rsc.data[13] + ",");
        System.out.println("( " + 14 + " ): " + rsc.data[14] + ",");
        execSvc.shutdown();
        System.out.println(" Main process ended ");
    }
}
```

Примірник класу RectangleSquareCalc є об'єктом-менеджером, який керує усім процесом обчислення. На нього покладені такі завдання:

- Розміщення та зберігання початкових, проміжних та кінцевих результатів обчислень. Для цього призначено масив data подвійної точності.
- Зберігання статусів початкових, проміжних та кінцевих результатів обчислень. Для цього призначено масив flags логічного типу.
- Організація портфеля задач та призначення процесів для його обслуговування.
- Створення та налагодження задач відповідно до графу операції-операнди та розміщення їх у портфелі задач.
- Очікування результатів обрахунку та виведення їх на консоль.
- Вивільнення ресурсів, зайнятих при організації портфеля задач.

Портфель задач, примірник класу ExecutorService, створимо виділивши для цього фіксований пул процесів з об'єкту Executors. У якості параметра, при

цьому, вкажемо кількість необхідних процесів. Після створення портфеля задач, за допомогою його методу execute, додамо у нього робочі задачі. У нашому прикладі такими задачами є примірники класу Calculator.

Кожен такий примірник створюється оператором new, і у якості параметрів приймає свою власну назву, код виконуваної ним операції, індекси її двох операндів, індекс результату, та посилання на батьківський об'єкт. По мірі наповнення портфеля задач, його робочі процеси виконують обчислення. Об'єкту-менеджеру RectangleSquareCalc залишається лише дочекатися результату, прийняти і вивести його на консоль.

Наведений приклад програмного коду належить до класу програм, що керуються даними. Тобто порядок обрахунків залежить виключно від структури графу операції-операнди. При цьому, операції визначаються під час створення робочих задач, а операнди розташовуються у лінійному загальнодоступному масиві. Відповідно до готовності даних, кожна комірка цього масиву може мати два стани: дані у комірці не готові, та дані готові до початку обчислення. Таким чином робоча задача може бути виконана тільки якщо дані обох операндів отримують стан готовності. Якщо, хоча б один з операндів не готовий до обрахунку, задача залишається у стані чекання. У наведеному прикладі, за стан готовності даних відповідає булевий масив flags. Початковий стан масивів data та flags наведено у таблиці 1.

№	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
data	x_2	y_2	0	x_2	y_1	0	x_1	y_1	0	x_1	y_2	0	0	0	0
flags	T	T	F	T	T	F	T	T	F	T	T	F	F	F	F

Табл. 1. Стан масивів **data** та **flags** перед початком обчислення

1.2 Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.

2. З'ясувати принципи взаємодії та синхронізації процесів при використанні моделі обчислень у вигляді графа операції-операнди.

3. З'ясувати особливості програмування та застосування моделі обчислень у вигляді графа операції-операнди.

1.3 Індивідуальні варіанти завдань до захисту роботи

Побудувати граф операції-операнди та реалізувати його програмно для обрахунку такого виразу:

1. $y = \frac{(a+b+c+d)\cdot\frac{e+f}{g+h}}{\frac{\frac{i-j}{k\cdot l}}{m\cdot n+\frac{q}{r}}}$

2. $y = \frac{\frac{a}{b}-(c+d)+\frac{e+f}{g\cdot h}}{i\cdot j\cdot\frac{k}{l}-(m\cdot n-p+q)}$

3. $y = \frac{(\frac{a}{b}+c+d)\cdot(e\cdot f-(g+h))}{(i\cdot j-\frac{k}{l})+(m+n)\cdot\frac{p}{q}}$

4. $y = \frac{\frac{a+b}{c-d}+(e-f)\cdot(g-h)}{(i\cdot j+\frac{k}{l})\cdot((m-n)+(p-q))}$

5. $y = \frac{\frac{a\cdot b}{c+d}\cdot((e-f)+\frac{g}{h})}{\frac{i+j}{k\cdot l}-(\frac{m}{l}-\frac{p}{q})}$

6. $y = \frac{(\frac{a}{b}+\frac{c}{d})\cdot(\frac{e}{f}+\frac{g}{h})}{\frac{i-j}{k-l}\cdot\frac{m\cdot n}{p-q}}$

7. $y = \frac{\frac{a\cdot b}{c+d}+\frac{e-f}{g\cdot h}}{(\frac{i}{j}\cdot\frac{k}{l})\cdot\frac{m-n}{p-q}}$

8. $y = \frac{\frac{a+b}{c\cdot d}}{\frac{e\cdot f}{g-h}}\cdot\frac{\frac{i-j}{k-l}}{\frac{m}{n}-\frac{p}{q}}$

9. $y = \frac{a\cdot b-c\cdot d-e\cdot f+\frac{g}{h}}{(i+j)\cdot\frac{k}{l}\cdot(\frac{m}{n}-\frac{p}{q})}$

10. $y = \frac{\frac{a}{b}+\frac{c}{d}}{(e-f)\cdot\frac{g}{h}}+\frac{\frac{i}{j}-k\cdot l}{\frac{m-n}{p\cdot q}}$

11. $y = \frac{(a-b)\cdot\frac{c}{d}-\frac{e}{f}\cdot\frac{g}{h}}{\frac{i+j}{k\cdot l}\cdot\frac{m-n}{p+q}}$

12. $y = (\frac{a+b}{c+d}-\frac{e}{f}\cdot(g+h))\cdot((i\cdot j+k\cdot l)+(\frac{m}{n}-(p-q))$

13. $y = (((a + b) + c \cdot d) - (e + f) + (g + h)) - (i \cdot j + \frac{k}{l} - (m \cdot n - p \cdot q)$

14. $y = ((a \cdot b - \frac{c}{d}) - (e \cdot f - g \cdot h)) + (i \cdot j + k \cdot l + \frac{m \cdot n}{p \cdot q})$

15. $y = ((a - b) \cdot c \cdot d) + \frac{e+f}{\frac{g}{h}} + (i \cdot j + k + l) \cdot \frac{m+n}{p-q}$

16. $y = (a \cdot b - \frac{c}{d} \cdot \frac{e+f}{g-h}) \cdot ((i + j) \cdot (k + l) - (\frac{m}{n} + p \cdot q))$

17. $y = \frac{(a+b) \cdot (c-d)}{(e+f) \cdot (g-h)} \cdot (i \cdot j - (k + l)) \cdot \frac{m+n}{p-q}$

18. $y = \frac{(a-b) \cdot \frac{c}{d}}{\frac{e}{f} - g \cdot h} + \frac{i-j}{k \cdot l} \cdot ((m + n) - p \cdot g)$

19. $y = \frac{a+b-c \cdot d}{\frac{e}{f} + g \cdot h} \cdot (\frac{i+j}{k \cdot l} + m \cdot n \cdot \frac{p}{q})$

20. $y = \frac{a \cdot b}{c-d} - \frac{e-f}{g \cdot h} + (\frac{i}{j} + k \cdot l) \cdot \frac{m \cdot n}{p \cdot q}$

21. $y = \frac{a \cdot b + c \cdot d}{e \cdot f - g \cdot h} \cdot \frac{\frac{i}{j} - \frac{k}{l}}{(m-n) \cdot (p+q)}$

22. $y = \frac{a-b}{c-d} \cdot (\frac{e}{f} + \frac{g}{h}) \cdot (\frac{i \cdot j}{k+l} + \frac{m}{n} \cdot \frac{p}{q})$

23. $y = \frac{a \cdot b}{c+d} \cdot \frac{e \cdot f}{g-h} - \frac{i}{j} \cdot \frac{k}{l} \cdot (\frac{m}{n} + \frac{p}{q})$

24. $y = (\frac{a+b}{c-d} + \frac{e \cdot f}{g \cdot h}) \cdot ((i + j) \cdot \frac{k}{l} + \frac{m}{n} \cdot \frac{p}{q})$

25. $y = (\frac{a+b}{c \cdot d} - (\frac{e}{f} - \frac{g}{h})) \cdot (\frac{i-j}{k-l} + \frac{m}{n} \cdot \frac{p}{q})$

Практична робота № 12

Тема роботи: Розподілене програмування на мові Java з використанням бар'єрної синхронізації.

Мета роботи: Опанувати методику створення розподілених програм з використанням бар'єрної синхронізації.

1.1 Теоретичні відомості

Бар’єр - це засіб синхронізації, який використовується для того, щоб деяка кількість процесів очікували один одного у деякому місці програми, яке зветься бар'єром або **точкою синхронізації**. Після того, як усі процеси досягли точки синхронізації, вони розблокуються і зможуть продовжувати виконання. На практиці бар’єри використовуються для збору результатів виконання деякої розпаралеленої задачі.

ЛІСТИНГ:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

class MatrixThread extends Thread {
    Thread mThread;
    int ma[][];
    int mb[][];
    int mc[][];
    int lo, hi;
    CyclicBarrier barrier;
    MatrixThread(int a[][], int b[][], int c[][], int l, int h, CyclicBarrier cb
    ) {
        ma = a;
        mb = b;
        mc = c;
        lo = l;
        hi = h;
        barrier = cb;
    }
    public void run() {
        for (int i = lo; i < hi; ++i) {
            for (int j = 0; j < mc[i].length; ++j) {
                ma[i][j] = 0;
                for (int k = 0; k < mb[j].length; ++k) {
                    ma[i][j] += mb[j][k] * mc[i][j];
                }
            }
        }
        try {
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException ex) {
        }
    }
}
```

```
public class MatrixCalcCyclicBarrier {
    static int ma[][];
    static int mb[][];
    static int mc[][];

    public static void main(String[] args) {
        int mSize = 10;
        ma = new int[mSize][mSize];
        mb = new int[mSize][mSize];
        mc = new int[mSize][mSize];
        for (int i = 0; i < mSize; ++i) {
            for (int j = 0; j < mSize; ++j) {
                ma[i][j] = 0;
                mb[i][j] = mc[i][j] = 1;
            }
        }
        int workersNumber = mSize;
        CyclicBarrier barrier = new CyclicBarrier(workersNumber, () -> {
            for (int i = 0; i < ma.length; ++i) {
                for (int j = 0; j < ma[i].length; ++j) {
                    System.out.print(ma[i][j] + " ");
                }
                System.out.println(" ");
            }
        });
        for (int i = 0; i < workersNumber; i++) {
            (new MatrixThread(ma, mb, mc, i, i + 1, barrier)).start();
        }
    }
}
```

Розглянемо приклад використання класу **CyclicBarrier**, який є реалізацією бар'єра. Він є складовою пакета **java.util.concurrent**. У якості такого прикладу можна розглянути задачу множення двох матриць (лістинг 1). При розпаралелюванні цієї задачі кожному процесу буде доручено множення визначених рядків на визначені стовпці. В точці синхронізації отримані результати збираються від усіх процесів, і будується кінцева матриця.

Матриці MB і MC містять початкові дані. Матриця MA, після завершення обрахунків буде містити результат. Для того, щоб вони були доступні усім процесам, їх розміщено у публічному об'єкті, породженому від класу MatrixCalcCyclicBarrier, і продекларовано за допомогою модифікатора доступу **static**. Це означає, що матриці є статичними даними. Тобто вони повинні існувати протягом усього процесу обрахунку.

Процес налагодження програми складається з чотирьох етапів (див.

функцію *main* лістингу 1). На першому етапі матриці MA, MB і MC розміщуються у статичній пам'яті. Їм виділяється об'єм пам'яті відповідно до їх розмірності. На другому етапі матриці MB і MC ініціалізуються одиницями, а матриці MA нулями. На третьому етапі відбува№ться створення та налагодження бар'єра. Бар'єр створюється як примірник класу CyclicBarrier. При ініціалізації йому передається кількість робочих процесів (workersNumber) та анонімний клас типу Runnable з визначеним обробником події, яка відбудеться коли коли до цього бар'єру підійдуть усі його workersNumber робочих процесів. На четвертому етапі створюються та запускаються на виконання дочірні робочі процеси. Їх кількість повинна точно дорівнювати кількості процесів, вказаної при створенні бар'єра. Кожному процесу у якості параметрів передаються посилання на матриці даних та результату, початковий та кінцевий індекси частин матриць, які будуть оброблятися цим процесом, та посилання на примірник бар'єру.

Кожен дочірній процес є примірником класу **MatrixThread** і складається з конструктора та методу **run**. У конструкторі відбувається прийняття та переприсвоєння формальних параметрів. У методі **run** процес обчислення. Кожен процес обраховує лише свою ділянку матриці MA та зупиняється у бар'єра за допомогою метода **await**. Оскільки під час чекання інших процесів може виникнути немасковане переривання або аварійний прорив бар'єра, то метод await може викинути переривання InterruptedException та/або BrokenBarrierException. Їх треба перехопити та обробити. Для цього використовується конструкція try catch.

1.2 Завдання

1. Розглянути, відкомпілювати та запустити на виконання наведений приклад.

2. З'ясувати принципи взаємодії та синхронізації процесів при використанні бар'єрної синхронізації.

3. З'ясувати особливості програмування та застосування бар'єрної

синхронізації.

1.3 Індивідуальні варіанти завдань до захисту роботи

1. Написати паралельну програму для транспонування матриці розміром *4n* x *4n* елементів *n* процесами. Для збору даних від робочих процесів та виводу кінцевого результату використати метод бар'єрної синхронізації.

2. Ввести загальну змінну з початковим значенням 0. Зробити так, щоб кожен процес після свого старту збільшував її на 1. Після зупинки усіх процесів у бар'єра, він повинен вивести її значення на екран. Перед закінченням роботи кожен процес повинен зменшити її значення на 1 і вивести результат на екран.

3. Створити загальний масив розмірності *N*. Зробити так, щоб кожен процес після свого старту збільшував відповідний елемент цього масиву на одиницю. Після зупинки усіх процесів у бар'єра він повинен вивести у консоль усі елементи цього масиву. Перед закінченням роботи кожен процес повинен зменшити на одиницю значення відповідної комірки масиву та вивести результат у консоль. Розмірність масиву задати як параметр у командному рядку під час запуску програми.

4. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси A і B. Процес A генерує матрицю A та ініціалізує її одиницями. Процес B генерує матрицю B та ініціалізує її двійками. Після чого обидва процеси зупиняються у першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу процеси A і B обраховують матрицю C, яка є поелементною сумою матриць A і B (тобто C = A + B). Причому процес A обраховує першу половину матриці C, а процес B другу. Після закінчення обрахунку обидва процеси зупиняються у другого бар'єра. Другий бар'єр

виводить значення матриці C у консоль. Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

5. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси A і B. Перший процес генерує матрицю відношення R1 : x ≥ y (більше чи дорівнює). Другий процес генерує матрицю відношення R2 : x ≤ y (менше чи дорівнює). Після чого обидва процеси зупиняються у першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу процеси A і B обраховують композицію цих відношень. Причому перший процес обраховує першу половину матриці відношення, а другий процес другу. Після закінчення обрахунку обидва процеси зупиняються у другого бар'єра. Другий бар'єр виводить значення отриманої матриці у консоль. Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

6. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси A і B. Процес A генерує множину A. Процес B генерує множину B. Після чого обидва процеси зупиняються у першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу процеси A і B обраховують декартовий добуток цих множин. Причому процес A обраховує першу половину матриці декартового добутку, а процес B другу. Після закінчення обрахунку обидва процеси зупиняються у другого бар'єра. Другий бар'єр виводить значення матриці декартового добутку у консоль. Після цього процеси припиняють роботу. Потужність множин задати як параметр у командному рядку під час запуску програми.

7. Розробити паралельну програму з двома робочими процесами та двома

бар'єрами. Головний процес створює два дочірніх робочих процеси A і B. Процес A генерує множину A. Процес B генерує множину B. Після чого обидва процеси зупиняються у першого бар'єра. Перший бар'єр виводить значення обох матриць у консоль. Після закінчення виводу про- цеси A і B обраховують переріз цих множин. Причому процес A обраховує першу половину перерізу, а процес B другу. Після закінчення обрахунку обидва процеси зупиняються у другого бар'єра. Другий бар'єр виводить значення множини перерізу у консоль. Після цього процеси припиняють роботу. Потужність множин задати як параметр у командному рядку під час запуску програми.

8. Розробити паралельну програму з трьома робочими процесами та двома бар'єрами. Головний процес створює три дочірніх робочих процеси A, B і C. Процес A генерує матрицю MA та ініціалізує її одиницями. Процес B генерує матрицю MB та ініціалізує її двійками. Процес C генерує матрицю MC та ініціалізує її трійками. Після чого усі процеси зупиняються у першого бар'єра. Перший бар'єр виводить значення усіх матриць у консоль. Після закінчення виводу процеси A, B і C обраховують матрицю MD, яка є результатом такого виразу: MD = MA + MB * MC. Матриця MC загальний ресурс, а матриці MA і MB розбиваються на три частини та розподіляються між робочими процесами. Після закінчення обрахунку усі робочі процеси зупиняються у другого бар'єра. Другий бар'єр виводить значення матриці MD у консоль. Після цього процеси припиняють роботу. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

9. Розробити паралельну програму з двома робочими процесами та бар'єром. Головний процес створює два дочірніх робочих процеси A і B. Процес A генерує множину A. Процес B генерує множину B. Після чого обидва процеси зупиняються у бар'єра. Бар'єр виконує об'єднання цих множин. Після цього процес A виводить у консоль першу половину об'єднаної множини, а процес

В другу. Потужність множин задати як параметр у командному рядку під час запуску програми.

10. Розробити паралельну програму з трьома робочими процесами та бар'єром. Головний процес створює три дочірніх робочих процеси A, B і C. Процес A генерує матрицю MA та ініціалізує її одиницями. Процес B генерує матрицю MB та ініціалізує її двійками. Процес C генерує матрицю MC та ініціалізує її трійками. Після чого усі процеси зупиняються у першого бар'єра. Перший бар'єр збільшує кожен елемент матриць MA, MB, та MC на одиницю. Після закінчення роботи першого бар'єра процеси A, B і C здійснюють вивід матриць. Розмірність матриць задати як параметр у командному рядку під час запуску програми.

11. Розробити паралельну програму з двома робочими процесами та двома бар'єрами. Головний процес створює два дочірніх робочих процеси A і B. Процес A генерує матрицю MA та ініціалізує її одиницями. Процес B генерує матрицю MB та ініціалізує її двійками. Після чого обидва процеси зупиняються у першого бар'єра. Перший бар'єр виконує введення значень матриці MC з дискового файлу. Після закінчення вводу процеси A, B і C обраховують матрицю MD, яка є результатом такого виразу MD = MA + MB * MC. Матриця MC загальний ресурс, а матриці MA і MB розбиваються на дві частини та розподіляються між робочими процесами. Після закінчення обрахунку усі робочі процеси зупиняються у другого бар'єра. Другий бар'єр виводить значення матриці MD у консоль. Після цього процеси припиняють роботу. Розмірність матриць та назву файлу даних з матрицею MC задати як параметр у командному рядку під час запуску програми.

Практична робота № 13

Тема: Блокуючи черги.

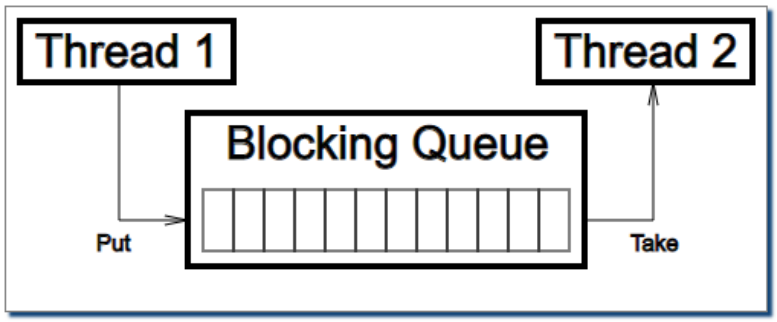
Мета: Опанувати методику синхронізації паралельних процесів з використанням блокуючої черги.

Теоретичні відомості.

Блокуюча черга - механізм гарантованого обміну даними між процесами шляхом їх блокування при повному її заповненні, або спорожненні. Блокуюча черга є одним з дієвих механізмів синхронізації процесів.

Процес, який намагається отримати дані з порожньої черги буде заблакований доти, доки деякий інший процес не покладе дані у цю чергу. Процес, який намагається покласти дані у вже заповнену чергу блокується доти, доки деякий інший процес не звільнить у неї місце шляхом виборки з неї даних. Крім виборки, місце може бути звільнене також знищенням частини, або, навідь усіх даних, що зберігаються у блокуючої черзі.

Взаємодія двох процесів за допомогою блокуючої черги може бути проілюстрована таким чином:



Починаючи з п'ятої версії реалізація блокуючої черги включена до пакету java.util.concurrent.

Але для більш єфективного використання цього механізму може бути корисно знати деталі його реалізації.

Приклад реалізації блокуючої черги

От проста реалізація блокуючої черги:

```
public class BlockingQueue {
private List queue = new LinkedList();
private int limit = 10;
public BlockingQueue(int limit){
this.limit = limit;
```

```
}
public synchronized void enqueue(Object item)
throws InterruptedException {
while(this.queue.size() == this.limit) {
wait();
}
if(this.queue.size() == 0) {
notifyAll();
}
this.queue.add(item);
}
public synchronized Object dequeue()
throws InterruptedException{
while(this.queue.size() == 0){
wait();
}
if(this.queue.size() == this.limit){
notifyAll();
}
return this.queue.remove(0);
}
}
```

Зверніть увагу, що метод notifyAll() викликається у enqueue() і dequeue() тільки у випадках, коли розмір черги дорівнює або 0, або максимуму.

В усіх інших випадках, коли розмір черги знаходиться у межах між нулем та максимумом, ніякі процеси не блокуються і передача даних відбувається без затримок.

Приклад використання блокуючої черги

```
import java.util.concurrent.*;

class Producer implements Runnable {

    private final BlockingQueue<Integer> queue;
    int procNumber;
    Producer(int p, BlockingQueue<Integer> q) {
        queue = q;
        procNumber = p;
    }
    public void run() {
        try {
            for (int i = 0; i < 10; ++i) {
                queue.put((Integer) produce(i));
            }
            queue.put(0);
            queue.put(0);
        } catch (InterruptedException ex) {
            System.out.println("Process (" + procNumber + "): " + ex.toString());
        }
    }
    Object produce(int i) {
        System.out.println("Producer(" + procNumber + "): " + i);
        return (i + 10);
    }
}
```

```
class Consumer implements Runnable {
    private final BlockingQueue<Integer> queue;
    int procNumber;
    Consumer(int p, BlockingQueue<Integer> q) {
        queue = q;
        procNumber = p;
    }
    public void run() {
        try {
            while (true) {
                consume(queue.take());
            }
        } catch (InterruptedException ex) {
            System.out.println("Process (" + procNumber + "): " + ex.toString());
        }
    }
    void consume(Integer x) throws InterruptedException {
        System.out.println("Result(" + procNumber + "): " + x);
        if (x == 0) {
            throw new InterruptedException();
        }
    }
}

public class BlockingQueueDemo {
    public static void main(String args[]) {
        System.out.println("Main process started");
        SynchronousQueue<Integer> q = new SynchronousQueue<Integer>(true);
        Producer p = new Producer(1, q);
        Consumer c1 = new Consumer(1, q);
        Consumer c2 = new Consumer(2, q);
        new Thread(p).start();
        new Thread(c1).start();
        new Thread(c2).start();
    }
}
```

У багатопотокових додатках черги повинні обробляти кілька одночасних сценаріїв виробники-споживачі. **Правильний вибір одночасної черги може мати вирішальне значення для досягнення хорошої продуктивності в наших алгоритмах.**

По-перше, ми побачимо деякі важливі відмінності між блокуючою чергою та неблокуючою. Потім ми розглянемо деякі впровадження та найкращі практики.

2. Блокування проти неблокуючої черги

BlockingQueue пропонує **простий захищений від потоків механізм** . У цій черзі потоки повинні чекати доступності черги. Виробники будуть чекати доступної потужності перед додаванням елементів, тоді як споживачі чекатимуть, поки черга не порожня. У цих випадках неблокуюча черга видає виняток або повертає спеціальне значення, наприклад *null* або *false* .

Для досягнення цього механізму *блокування* інтерфейс *BlockingQueue* надає дві функції поверх звичайних функцій *черги* : *put* і *take* . Ці функції єквівалентом *додавання* та *видалення* у стандартній *черзі*.

3. Одночасні реалізації *черги*

3.1. *ArrayBlockingQueue*

Як випливає з назви, ця черга використовує масив внутрішньо. Як наслідок, це **обмежена черга, тобто вона має фіксований розмір** .

Проста черга роботи - це приклад використання. Цей сценарій часто є низьким відношенням виробника до споживача, коли ми розподіляємо трудомісткі завдання між кількома працівниками. Оскільки ця черга не може зростати нескінченно, **обмеження розміру виступає як поріг безпеки, якщо проблема з пам'яттю** .

Говорячи про пам'ять, важливо зазначити, що черга попередньо розподіляє масив. Хоча це може покращити пропускну здатність, воно **також може споживати більше пам'яті, ніж потрібно** . Наприклад, черга великої місткості може тривати порожньою протягом тривалого періоду часу.

Крім того, *ArrayBlockingQueue* використовує єдиний замок як для операцій *введення, так* і для *прийому* . Це гарантує відсутність перезапису записів ціною хіта продуктивності.

3.2. *LinkedBlockingQueue*

LinkedBlockingQueue використовує *LinkedList* варіант, в якому кожен елемент черги являє собою новий вузол. Хоча це робить чергу в принципі необмеженою, вона все ще має жорсткий ліміт *Integer.MAX_VALUE* .

З іншого боку, ми можемо встановити розмір черги, використовуючи конструктор *LinkedBlockingQueue (int capacity)* .

Ця черга використовує окремі блокування для операцій *путівки* та *прий мання* Як наслідок, обидві операції можна виконувати паралельно та покращувати пропускну здатність.

Оскільки *LinkedBlockingQueue* може бути обмеженим або необмеженим, чому

б ми використовували *ArrayBlockingQueue* над цим? ***LinkedBlockingQueue*** **по винен розподіляти та звільняти вузли кожного разу, коли елемент додається або вилучається з черги** . З цієї причини *ArrayBlockingQueue* може бути кращою альтернативою, якщо черга швидко зростає і швидко скорочується.

Продуктивність *LinkedBlockingQueue* вважається непередбачуваною. Іншими словами, нам завжди потрібно скласти профіль наших сценаріїв, щоб забезпечити правильну структуру даних.

3.3. *PriorityBlockingQueue*

PriorityBlockingQueue наш йти до вирішення , **коли ми повинні споживати**

елементи в певному порядку . Для цього *PriorityBlockingQueue* використову є двійкову купу на основі масиву. У той час як внутрішньо він використовує єдиний механізм блокування, то *взяття* операція може відбуватися одночасно з *пут* операції. Використання простого блокування робить це можливим.

Типовим випадком використання є споживання завдань з різними пріоритетами. **Ми не хочемо, щоб завдання з низьким пріоритетом замінювало завдання з високим пріоритетом** .

3.4. *DelayQueue*

Ми використовуємо *DelayQueue,* **коли споживач може взяти лише прострочений товар** . Цікаво, що він використовує *PriorityQueue* внутрішньо для впорядкування елементів до закінчення терміну їх дії.

Оскільки це не черга загального призначення, вона не охоплює стільки сценаріїв, як *ArrayBlockingQueue* або *LinkedBlockingQueue* . Наприклад, ми можемо використовувати цю чергу для реалізації простого циклу подій, подібного до того, що знаходиться в NodeJS. Ми розміщуємо асинхронні завдання в черзі для подальшої обробки, коли вони закінчуються.

3.5. *LinkedTransferQueue*

LinkedTransferQueue вводить *передачі* методу. Хоча інші черги, як правило, блокуються під час виробництва або споживання

предметів, *LinkedTransferQueue* **дозволяє виробнику чекати споживання елемента** .

Ми використовуємо *LinkedTransferQueue,* коли нам потрібна гарантія того, що певний елемент, який ми помістили в чергу, кимось був прийнятий. Крім того, ми можемо реалізувати простий алгоритм зворотного тиску, використовуючи цю чергу. Дійсно, блокуючи виробників до споживання, **споживачі можуть керувати потоком вироблених повідомлень** .

3.6. *SynchronousQueue*

Хоча черги, як правило, містять багато елементів, *SynchronousQueue* завжди матиме, щонайбільше, один елемент. Іншими словами, нам потрібно розглядати *SynchronousQueue* як **простий спосіб обміну деякими даними між двома потоками** .

Коли у нас є два потоки, які потребують доступу до спільного стану, ми часто синхронізуємо їх із *CountDownLatch* або іншими механізмами синхронізації. Використовуючи *SynchronousQueue* , ми можемо **уникнути цієї ручної синхронізації потоків** .

3.7. *ConcurrentLinkedQueue*

ConcurrentLinkedQueue є єдиним неблокіруючіх черзі даного керівництва. Отже, він забезпечує алгоритм "без очікування", коли ***додавання*** **та** ***опитування*** **гарантують безпеку потоку та негайне повернення** . Замість блокувань у цій черзі використовується CAS (Порівняння та обмін).

Внутрішньо він базується на алгоритмі *простих, швидких та практичних неблокуючих та блокуючих паралельних алгоритмів черги* Магеда М. Майкла та Майкла Л. Скотта.

Це **ідеальний кандидат для сучасних реактивних систем** , де використання блокуючих структур даних часто заборонено.

З іншого боку, якщо наш споживач в кінцевому підсумку чекає в циклі, нам, мабуть, слід вибрати чергу блокування як кращу альтернативу.

Індивідуальні завдання до захисту роботи

Розробити розподілену програму для поелементного складання двох матриць розміром N x N. Перший та другий процеси генерують початкові матриці та поелементно передають їх до третього процесу. Третій процес реалізує процедуру складення елементів матриць та по мірі отримання результату передає його до четвертого процесу. Четвертий процес виводить результат на консоль. Розмірність матриці задати як параметр у командному рядку під час запуску програми.

Практична робота № 14

Тема: Обрахунок визначеного інтегралу з використанням механізму синхронізації CountDownLatch

Мета: опанувати методику створення паралельних ітеративних програм та чисельний метод обрахунку визначеного інтеграла з використанням лівосторонніх прямокутників.

Теоретичні відомості.

Клас CountDownLatch пакету java.util.concurrent. Клас CountDownLatch пакету java.util.concurrent використовується для того, щоб один чи декілька потоків змогли дочекатися виконання певної кількості операцій у інших потоках. Цей клас працює за принципом таймеру. У головному процесі виконується ініціалізація його деяким початковим значенням procNumber.

```
CountDownLatch latch = new CountDownLatch(procNumber);
```

У дочірніх процесах, за допомогою викликів методу countDown(), відбувається зворотній відлік.

```
latch.countDown();
```

Під час виклику методу await() цього класу деяким потоком, він переходить у стан чекання моменту досягнення лічильником таймера значення 0.

```
try{
latch.await();
} catch (InterruptedException e) {}
```

На практиці цей клас зручно використовувати для координації моменту початку та закінчення певної кількості потоків.

Це дозволяє зробити наступне:

- Запускати декілька потоків у один і той же момент часу.
- Відслідковувати момент закінчення роботи декількох потоків.

Визначений інтеграл

Визначений інтеграл являє собою площину, обмежену кривою f(x), віссю x та прямими x = a і x = b, та позначається таким чином:

$$S = \int_a^b f(x)\mathrm{dx}.$$

Графічна інтерпретація інтегралу виглядає так:

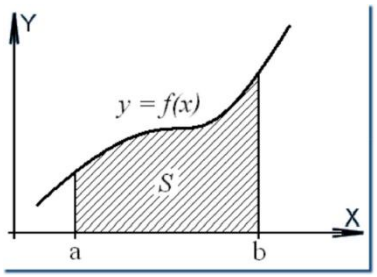


Рис. 14.1 – Визначений інтеграл

Чисельне інтегрування основане на тому, що відрізок інтегрування [a, b] розподілюється на n менших відрізків [xi-1, xi], кожен з яких є основою геометричної фігури, площу якої знаходять наближено як Si, а значення інтегралу визначають як суму таких площин Si, тобто $S = \sum_{i=0}^{n} S_i$.

У теорії розглядаються два способи розбиття відрізка інтегрування на менші:

1. Розбиття відрізка інтегрування відбувається раніше за аналіз результатів інтегрування, крім того, відрізки вибирають рівними (метод прямокутників, трапецій, Сімпсона);

2. Розташування та довжину відрізків визначають і послідовно уточнюють під час інтегрування за умови досягнення найбільшої точності обчислень (метод Гаусса, Ньютона-Котеса, Чебишева).

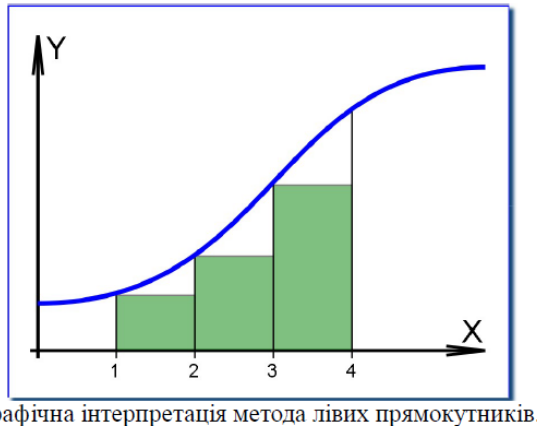
Чисельні методи знаходження визначеного інтегралу

У залежності від форми елементарної геометричної фігури, площа якої обчислюється, розрізняють такі методи: a. прямокутників, b. трапецій, c. Сімпсона.

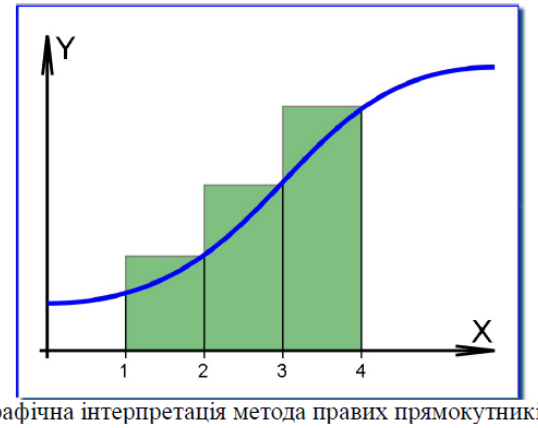
Метод прямокутників. Метод прямокутників є найпростішим методом наближеного обчислення інтеграла. За цим методом інтеграл обчислюється як сума площ N прямокутників з висотою f(x) та основою $\Delta x = x_{i+1} - x_i$, отриманих шляхом розбиття відрізка інтегрування [a, b] на N рівних частин.

Тут можливі два випадки:

1. $S_i = f(x_i) * (x_{i+1} - x_i)$ - метод лівих прямокутників,
2. $S_i = f(x_{i+1}) * (x_{i+1} - x_i)$ - метод правих прямокутників.



Графічна інтерпретація метода лівих прямокутників.



Графічна інтерпретація метода правих прямокутників.

Рис. 14.2 – Метод прямокутників

Метод трапецій

За цим методом інтеграл обчислюється як сума площ Si прямокутних трапецій. Площа кожної такої трапеції визначається як:

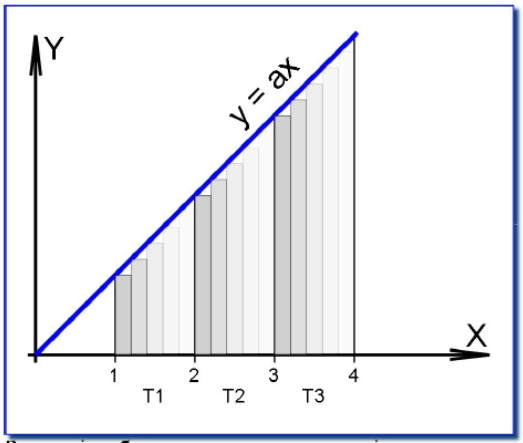
$$S_i = \frac{f(x_i) + f(x_{i+1})}{2} * (x_{i+1} - x_i).$$

Метод Сімпсона (метод парабол або криволінійних трапецій)

За цим методом інтеграл обчислюється як сума площ Si криволінійних трапецій. Площа кожної криволінійної трапеції визначається за формулою Сімпсона:

$$S_i = \frac{(x_{i+2} - x_i)}{3}[f(x_i) + 4 * f(x_{i+1}) + f(x_{i+2})]$$

Приклад реалізації ітеративного обрахунку визначеного інтеграла з використанням лівосторонніх прямокутників.



Розподіл обчислювальних задач між процесами.

Рис. 14.3 – Метод лівосторонніх прямокутників

ЛІСТИНГ:

```
import java.util.concurrent.CountDownLatch;

class IntegralThread extends Thread {
    int procNumber;
    IntegralDemo manager;
    double lo;
    double hi;
    double epsilon;
    IntegralThread(int num, IntegralDemo mgr, double l, double h, double eps) {
        procNumber = num;
        manager = mgr;
        lo = l;
        hi = h;
        epsilon = eps;
    }

    @Override
    public void run() {
        System.out.println(" IntegralIteration: " + procNumber + ": lo= " + lo + ", hi= " + hi);
        manager.result[procNumber] = IntegralIteration(lo, hi, epsilon);
        manager.latch.countDown();
    }
    double userFunction(double a, double b, double x) {
        return (a * x + b);
    }
    double IntegralStep(double lo, double hi, double step) {
        double s = 0.0;
        for (double x = lo; x < hi; x += step) {
            double f = userFunction(1, 0, x);
            s += f * step;
        }
        System.out.println(" " + procNumber + " s= " + s);
        return (s);
    }
    double IntegralIteration(double lo, double hi, double epsilon) {
        double step = (hi - lo) / 10;
        double s1 = 0.0, s2 = 0.0;
        do {
            s1 = s2;
            s2 = IntegralStep(lo, hi, step);
            step /= 2;
        } while (Math.abs(s1 - s2) > epsilon);
        return (s2);
    }
}
public class IntegralDemo {
    CountDownLatch latch;
    double result[];
    IntegralDemo(double lo, double hi, double epsilon, int procNumber) {
        result = new double[procNumber];
        latch = new CountDownLatch(procNumber);
        double step = (hi - lo) / procNumber;
        double l = lo;
        for (int i = 0; i < procNumber; ++i) {
            double h = l + step;
            IntegralThread integral = new IntegralThread(i, this, l, h, epsilon / procNumber);
            integral.start();
            System.out.println(i + ") lo= " + l + ", hi= " + h);
            l = h;
        }
        try {
            latch.await();
        } catch (InterruptedException e) {
        }

        double s = 0.0;
        for (int i = 0; i < procNumber; ++i) {
            s += result[i];
        }
        System.out.println("F= " + s);
    }
    public static void main(String args[]) {
        IntegralDemo demo = new IntegralDemo(1.0, 4.0, 0.01, 3);
    }
}
```

РЕЗУЛЬТАТ:

```
run:
0) lo= 1.0, hi= 2.0
 IntegralIteration: 0: lo= 1.0, hi= 2.0
 0 s= 1.4500000000000006
1) lo= 2.0, hi= 3.0
2) lo= 3.0, hi= 4.0
 0 s= 1.4750000000000005
 0 s= 1.537499999999998
 0 s= 1.5187499999999983
 0 s= 1.496875000000007
 0 s= 1.4984375000000072
 IntegralIteration: 1: lo= 2.0, hi= 3.0
 1 s= 2.45
 1 s= 2.6249999999999982
 1 s= 2.5624999999999982
 1 s= 2.4937500000000075
 1 s= 2.4968750000000064
 IntegralIteration: 2: lo= 3.0, hi= 4.0
 2 s= 3.4500000000000006
 2 s= 3.674999999999999
 2 s= 3.587499999999998
 2 s= 3.4937500000000075
 2 s= 3.496875000000007
F= 7.49218750000002
```

Індивідуальні завдання до захисту роботи

1. Обчислити визначений інтеграл функції однієї змінної виду $y = 4 * \sin(\pi * x)$ методом лівих прямокутників на інтервалі від 3 до 5 з точністю ε = 0.000001.

2. Обчислити визначений інтеграл функції однієї змінної виду $y = 6 * \sin(\pi * x + 4)$ методом правих прямокутників на інтервалі від 5 до 7 з точністю ε = 0.000001.

3. Обчислити визначений інтеграл функції однієї змінної виду $y = 10 * \cos(2 * \pi * x)$ методом трапецій на інтервалі від 7 до 9 з точністю ε = 0.000001.

4. Обчислити визначений інтеграл функції однієї змінної виду $y = 4 * \sin(\pi + x)$ методом лівих прямокутників на інтервалі від 1 до 3 з точністю ε = 0.000001.

5. Обчислити визначений інтеграл функції однієї змінної виду $y = 9 * \sin(\pi * x + 10)$ методом правих прямокутників на інтервалі від 5 до 7 з точністю

ε = 0.000001.

6. Обчислити визначений інтеграл функції однієї змінної виду y = 7 * cos (6 * π * x) методом трапецій на інтервалі від 7 до 9 з точністю ε = 0.000001.

7. Обчислити визначений інтеграл функції однієї змінної виду y = 3 * sin (π * x) методом лівих прямокутників на інтервалі від 3 до 6 з точністю ε = 0.000001.

8. Обчислити визначений інтеграл функції однієї змінної виду y = 2 * sin (π * x + 4) методом правих прямокутників на інтервалі від 3 до 7 з точністю ε = 0.000001.

9. Обчислити визначений інтеграл функції однієї змінної виду y = 7 * cos (3 * π * x) методом трапецій на інтервалі від 4 до 8 з точністю ε = 0.000001.

10. Обчислити визначений інтеграл функції однієї змінної виду y = 3 * sin (π * x) методом лівих прямокутників на інтервалі від 3 до 8 з точністю ε = 0.000001.

11. Обчислити визначений інтеграл функції однієї змінної виду y = 6 * sin (π * x + 2) методом правих прямокутників на інтервалі від 3 до 7 з точністю ε = 0.000001.

12. Обчислити визначений інтеграл функції однієї змінної виду y = 7 * cos (5 * π * x) методом трапецій на інтервалі від 6 до 9 з точністю ε = 0.000001.

Практична робота № 15

Тема: Вирішення обчислювальних задач методом Монте-Карло (функціональний стиль)

Мета: Опанувати методику програмування у функціональному стилі.

Теоретичні відомості

Пошук рішення методом Монте-Карло відбувається протягом певного часу у певному просторі значень. Тому можливі два способи розпаралелювання:

- за часом;
- за простором.

Перший спосіб передбачає створення цілком однакових процесів, які паралельно й незалежно один від одного шукають рішення у всьому просторі. Другий спосіб передбачає розподіл всього простору рішень на частини й надання кожної такої частини відповідному процесу, якій і буде шукати там рішення.

Програмування у функціональному стилі

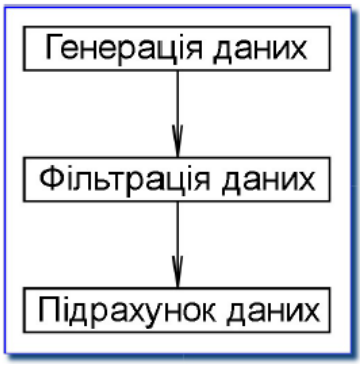


Рис 15.1 – Схема функціонального стилю

У версії 8 мови програмування Java введено новий пакет java.util.stream, якій містить класи, що можуть бути використані при програмуванні у функціональному стилі.

Функціональні потоки:

- **IntStream** - поток даних типу Integer.
- **LongStream** - поток даних типу Long.
- **DoubleStream** - поток даних типу Double.

Основні методи функціональної обробки даних:

– **generate(IntSupplier s)** - генерація потоку даних за допомогою постачальника s.

– **of(int ... values)** - побудова потоку даних з переліку елементів.

– **iterate(int seed, IntUnaryOperator f**) - ітеративна генерація потоку даних.

– **skip(long n)** - відкидання (пропуск) перших n елементів даних потока.

– **limit(long maxSize)** - обмеження генерації даних у потоці до maxSize елементів.

– **range(int startInclusive, int endExclusive)** - вібір з потоку певної послідовності даних за їх номерами від *startInclusive* до *endExclusive*.

– **parallel()** - розпаралелювання потока даних.

– **sequential()** - об'єднання потоків даних.

– **filter(IntPredicate predicate)** - фільтрація даних згідно зі значенням предиката.

– **sorted()** - сортування потоку даних.

– **count()** - підрахунок кількості даних, що надійшли.

– **min()** - повернення мінімального значення з потоку даних.

– **max()** - повернення максимального значення з потоку даних.

– **sum()** - підрахунок суми.

– **average()** - підрахунок середнього значення.

– **reduce(IntBinaryOperator op)** - об'єднання даних за певним законом або правилом.

– **map(IntUnaryOperator mapper)** - Мапінг, або ремапінг даних - процес, при якому одні дані замінюються іншими за певним законом або правилом.

– **mapToInt(DoubleToIntFunction mapper)**

– **mapToLong(IntToLongFunction mapper)**

– **mapToDouble(IntToDoubleFunction mapper)**

– **mapToObj(IntFunction mapper)**

– **peek(IntConsumer action)** - виконання дії action над кожним елементом потоку. Проміжна обробка, результати видаються у вихідний потік і у подальшому можуть бути оброблені іншим фільтром.

– **forEach(IntConsumer action**) - виконання дії action над кожним елементом потоку. Фінальна обробка.

– **findFirst()** - пошук першоко елемента даних.

– **findAny()** - пошук будь-якого елемента даних.

– **toArray()** - перетворення потоку даних у масив

Приклад обрахунку числа π методом Монте-Карло

Площу кола можна обрахувати за формулою $S_{\text{кола}} = \pi r^2$, площу квадрата - за формулою $S_{\text{квадрата}} = (2r)^2$, де r - радіус кола, яке вписане у квадрат.

Тоді співвідношення цих площ буде: $\frac{S_{circle}}{S_{square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4} \approx 0.785$

Таким чином число π можна обрахувати так: $\pi = 4 * \frac{S_{circle}}{S_{square}}$

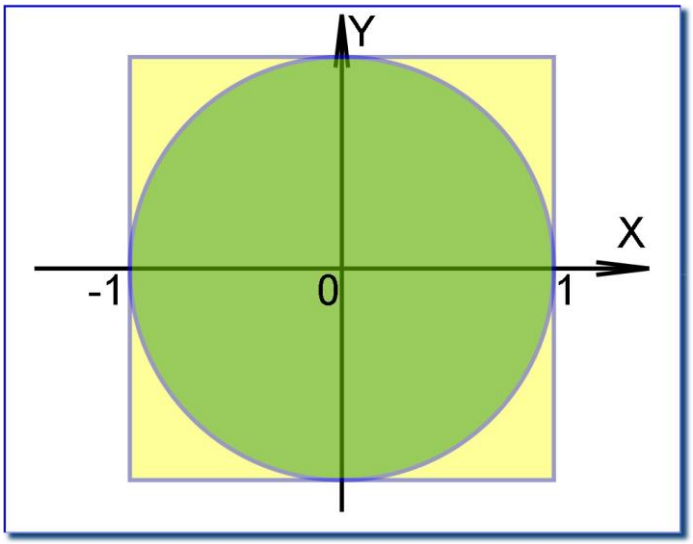


Рис. 15.2 – Графічна інтерпретація обрахунку числа π методом Монте-Карло

Тепер залишається написати програму, яка випадковим чином генерує множину точок у діапазоні від -1 до 1 за координатами X і Y, та визначає, скільки з них потрапляє до кола з радіусом 1. Відношення кількості точок, що потрапили у коло до кількості усіх згенерованих точок помножене на 4 і буде шуканим числом π.

ЛІСТИНГ:

```
import static java.lang.System.out;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;

public class MonteCarloPI {

    public static double range() {
        return (Math.random() * 2) - 1;
    }
    public static double pi(int numThrows) {
        long inCircle = DoubleStream.generate(
                () -> Math.hypot(range(), range())
        ).limit(numThrows)
                .parallel()
                .filter(d -> d < 1)
                .count();
        return (4.0 * inCircle) / numThrows;
    }
    public static void main(String[] args) {
        IntStream.of(10000, 100000, 1000000)
                .mapToDouble(MonteCarloPI::pi)
                .forEach(out::println);
    }
}
```

Обрахунок співвідношення об’ємів куба та вписаної у нього сфери

Об'єм сфери можна обрахувати за формулою $V_{\text{сфери}} = \frac{4}{3}\pi r^3$, об'єм куба - за формулою $V_{\text{куба}} = (2r)^3$, де r - радіус сфери, яка вписана у куб. Тоді співвідношення цих об’ємів буде: $\frac{\text{Vсфери}}{\text{Vкуба}} = \frac{\frac{4}{3}\pi r^3}{(2r)3} = \frac{\pi}{6} \approx 0.5233$.

ЛІСТИНГ:

```
import static java.lang.System.out;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;

public class MonteCarloSphereInCube {
    public static double pos() {
        return 2.0 * Math.random() - 1.0;
    }
    public static double range() {
        double x = pos();
        double y = pos();
        double z = pos();
        return Math.sqrt(x * x + y * y + z * z);
    }
    public static double test(int numThrows) {
        long inSphere = DoubleStream.generate(() -> range())
                .limit(numThrows)
                .parallel()
                .filter(r -> r < 1)
                .count();
        return (double) inSphere / (double) numThrows;
    }
    public static void main(String[] args) {
        IntStream.of(10000, 100000, 1000000)
                .mapToDouble(MonteCarloSphereInCube::test)
                .forEach(out::println);
    }
}
```

Завдання:

1. Розглянути, відкомпілювати та запустити на виконання наведені приклади.

2. З'ясувати принципи взємодії та синхронізації потоків при використанні функціонального стилю програмування.

3. З'ясувати особливості програмування у функціональному стилі.

Індивідуальні завдання до захисту роботи

1. Дано кардіоїда вписана у квадрат. Обрахувати співвідношення площин кардіоїди та квадрата методом Монте-Карло.

2. Дано коло вписане у квадрат. Обрахувати співвідношення площин кола та квадрата методом Монте-Карло.

3. Дано квадрат вписаний у коло. Обрахувати співвідношення площин кола та квадрата методом Монте-Карло.

4. Дано трикутник вписаний у коло. Обрахувати співвідношення площин кола та трикутника методом Монте-Карло.

5. Дано коло вписане у трикутник. Обрахувати співвідношення площин кола та трикутника методом Монте-Карло.

6. Дано зірка вписана у коло. Обрахувати співвідношення площин кола та зірки методом Монте-Карло.

7. Дано сфера вписана у куб. Обрахувати співвідношення об’ємів сфери та куба методом Монте-Карло.

8. Дано куб вписаний у сферу. Обрахувати співвідношення об’ємів сфери та куба методом Монте-Карло.

9. Дано сфера вписана у тетраедр. Обрахувати співвідношення об’ємів сфери та тетраедра методом Монте-Карло.

10. Дано тетраедр вписаний у сферу. Обрахувати співвідношення об’ємів сфери та тетраедра методом Монте-Карло

Перелік питань для підсумкового контролю знань

1. Послідовні обчислення.
2. Паралельні обчислення.
3. Засоби для здійснення паралельних обчислень.
4. Паралельні комп’ютери.
5. Актуальність та перспективи використання паралельних обчислень.
6. Сфери застосування паралельних обчислень.
7. Рівні розпаралелювання.
8. Способи обробки даних в обчислювальних системах.
9. Послідовна обробка даних.
10. Конвеєрна обробка даних.
11. Характеристики систем функціональних пристроїв.
12. Класифікація паралельних обчислювальних систем.
13. Клас Thread. Створення, виконання потоків.
14. Завершення та переривання потоку.
15. Переваги та недоліки при використанні потоків.
16. Спадкування від класу Thread та інтерфейс Runnable.
17. Синхронізація потоків.
18. Концепція необмеженого паралелізму
19. Граф алгоритму
20. Внутрішній паралелізм
21. Загальне визначення монітору при організації міжпотокової взаємодії.
22. Блокуюча та неблокуюча умовні змінні. Взаємне виключення.
23. Дисципліни сигналізації: оператор signal.
24. Загальне визначення засувок при організації міжпотокової взаємодії.
25. Інтерфейс Lock та клас ReentrantLock. Блокування.
26. Нерекурсивна версія засувки.
27. Рекурсивна версія засувки.
28. Ключове слово volatile в Java.

29. Загальне визначення семафорів при організації міжпотокової взаємодії.

30. Клас Semaphore. Конструктори та методи класу Semaphore.

31. Простий семафор.

32. Використання семафорів для сигналізації.

33. Рахуючий семафор.

34. Обмежуючий семафор.

35. Принцип синхронізації робочих процесів за часом.

36. Алгоритми розподілених систем: централізований алгоритм.

37. Розподілений алгоритм та алгоритм Token Ring.

38. Обмін даними між потоками. Клас Exchanger та метод exchange.

39. Процес відправник та процес-отримувач.

40. Режим одночасного прийому та передачі даних.

41. Парадигма портфелю задач: принципи взаємодії та синхронізації процесів при використанні портфеля задач.

42. Особливості програмування та застосування портфелю задач.

43. Модель обчислень у вигляді графа «операції-операнди».

44. Ациклічний орієнтований граф.

45. ExecutorService та метод execute.

46. Засіб синхронізації бар’єр або точка синхронізації.

47. Принципи взаємодії та синхронізації при використанні бар'єрної синхронізації.

48. Особливості програмування та застосування бар'єрної синхронізації.

49. Блокуюча черга, як дієвий механізм синхронізації процесів.

50. Відмінності між блокуючою та неблокуючою чергами.

51. Види черг: ArrayBlockingQueue.

52. Види черг: LinkedBlockingQueue.

53. Види черг: SynchronousQueue.

54. Принципи взаємодії та часової синхронізації процесів під час використання класу CountDownLatch.

55. Метод прямокутників при реалізації ітеративного обрахунку визначеного інтегралу

56. Метод трапецій при реалізації ітеративного обрахунку визначеного інтегралу.

57. Паралелізм у алгоритмі множення матриць

58. Методика програмування у функціональному стилі програмування.

59. Генерація, фільтрація та підрахунок даних.

60. Пакет java.util.stream. Функціональні потоки.

Список рекомендованих та використаних джерел

Основна

1. Архітектура обчислювальних систем : навчальний посібник до виконання розрахунково-графічної роботи / уклад. В. Г. Артюхов та ін. Київ : КПІ ім. Ігоря Сікорського, 2023. 85 с. URL: https://ela.kpi.ua/server/api/core/bitstreams/f0a58483-7a52-4bd5-9ee5-66831d54c445/content
2. Бородкіна І. Л. Теорія алгоритмів : посібник. Київ : ЦУЛ, 2022. 184 с.
3. Гороховатський В. О., Творошенко І. С. Методи інтелектуального аналізу та оброблення даних : навч. посібник. Харків : ХНУРЕ, 2021. 92 с. URL: https://openarchive.nure.ua/server/api/core/bitstreams/2e55d639-52fd-48d9-b7b7-14989f49f291/content
4. Корочкін О. В., Русанова О. В. Паралельні та розподілені обчислення. Вибрані розділи : навч. посібник. Київ : КПІ ім. Ігоря Сікорського, 2020. 123 с.
5. Коцовський В. М. Теорія паралельних обчислень : навчальний посібник. Ужгород : ПП «АУТДОР-Шарк», 2021. 188 с. URL: http://surl.li/seeca
6. Кузьма К. Т., Мельник О. В. Паралельні та розподілені обчислення : навчальний посібник для вищих закладів освіти. Миколаїв : ФОП Швець В.М., 2020. 172 с. URL: http://surl.li/pnljs
7. Малашонок Г. І., Сідько А. А. Паралельні обчислення на розподіленій пам’яті: OpenMPI, Java, Math Partner : підручник. Київ : НаУКМА, 2020. 266 с.
8. Минайленко Р. М. Паралельні та розподілені обчислення : навч. посіб. Кропивницький : ЦНТУ, 2021. 153 с. URL: https://dspace.kntu.kr.ua/server/api/core/bitstreams/396e02d2-725b-47b5-a1c0-ae07a9bec326/content
9. Паулін О. М. Розподілені обчислення : навчальний посібник. Одеса : Одеська політехніка, 2022. 62 с.
10. Юрчишин В. Я. Проєктування сучасних високопродуктивних обчислювальних систем : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2022. 279 с. https://ela.kpi.ua/server/api/core/bitstreams/781c88bf-0f18-4af9-aed2-c92d952e3f01/content
11. Луцків А. М., Лупенко С. А., Пасічник В. В. Паралельні та розподілені обчислення : посібник. Львів : ПП "Магнолія 2006", 2025. 566 с.
12. Бочкарьов О. Ю. Паралельне програмування в ОС Linux : навчальний посібник. Львів : ПП "Магнолія 2006", 2024. 201 с.
13. Минайленко Р. М. Паралельні та розподілені обчислення: навч. посіб. Кропивницький: ЦНТУ, 2021. 153 с.

Додаткова література

1. Argonne National Laboratory, Center for Computational Science and Technology. URL: http://www.mcs.anl.gov
2. Czarnul P. Parallel Programming for Modern High Performance Computing Systems. CRC Press, 2018. 304p.

3. Kurgalin S., Borzunov S. A Practical Approach to High-Performance Computing. Springer, 2019. 206 p

4. Parallel Computing. Architectures, Algorithms and Applications / Bischof C., Bücker M., Gibbon P., Joubert G.R., Lippert T., Mohr B., Peters F.

5. Richard Gerber, Aart J.C. Bik, Kevin B. Smith, and Xinmin Tian The Software Optimization Cookbook, Second Edition.Intel Press, 404p.

6. S. Akhter, J. Roberts. Multi-Core Programming. – Intel Press, 344p.

7. Synchronization of Parallel Programs / Andre J., Herman D., Verjus J.-P. Oxford: North Oxford Academic Publishing Company Limited, 1985. 110 p.

8. Аксак Н. Г. Руденко О. Г., Гуржій А. М. Паралельні та розподілені обчислення: підручник. Харків : Компанія СМІТ, 2009. 480с.

9. Дорошенко А.Ю. Паралельні обчислювальні системи : методичний посібник і конспект лекцій. Київ : Видавничий дім «КМ Академія», 2013. 46 с.

10. Жуков І., Корочкін О. Паралельні та розподілені обчислення : навчальний посібник. Київ : Корнійчук, 2014. 284 с.

11. Кузьменко Б. В., Чайковська О. А. Технологія розподілених систем та паралельних обчислень. Конспект лекцій, частина 1. Розподілені об’єктні системи, паралельні обчислювальні системи та паралельні обчислення, паралельне програмування на основі MPI : навчальний посібник. Київ : Видавничий центр КНУКІМ, 2011. 126 с.

12. Лазарович І. М. Паралельні обчислювальні середовища. Лабораторний практикум. Івано-Франківськ : Видавництво ПНУ імені Василя Стефаника, 2014. 65 с.

13. Організація паралельних обчислень : навчальний посібник / уклад. Є. Ваврук, О. Лашко. Львів : Національний університет “Львівська політехніка”, 2018. 70 с.

14. Паралелізм в Java. URL: https://uk.wikipedia.org/wiki/Паралелізм_в_Java

15. Паралельна обробка і паралелізм в .NET Framework. URL: http://msdn.microsoft.com/ruru/library/hh156548(v=vs.110).aspx

16. Рольщиков В. Б. Технології розподілених систем та паралельних обчислень : конспект лекцій. Одеса : ОДЕКУ 2016.155 с.

17. Сайт Української команди розподілених обчислень. URL: http://distributed.org.ua/.

Навчальне видання

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Методичні рекомендації

Укладачі: **Тищенко** Світлана Іванівна
Пархоменко Олександр Юрійович
Жебко Олександр Олегович

Формат 60х84 1/16. Ум. друк. арк. 4.25.
Наклад 50 прим. Зам. № _____

Надруковано у видавничому відділі
Миколаївського національного аграрного університету
54020, м. Миколаїв, вул. Георгія Гонгадзе, 9

Свідоцтво суб'єкта видавничої справи ДК № 4490 від 20.02.2013