



Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики і програмної інженерії

Лабораторна робота №3
з дисципліни
Технології паралельних обчислень

Виконав:

студент групи ПІ-13:
Бабашев О. Д.

Перевірив:

ст.викл.
Дифучин А. Ю.

Київ 2024

Завдання до комп'ютерного практикуму 3 «Розробка паралельних програм з використанням механізмів синхронізації: синхронізовані методи, локери, спеціальні типи»

1. Реалізуйте програмний код, даний у лістингу, та протестуйте його при різних значеннях параметрів. Модифікуйте програму, використовуючи методи управління потоками, так, щоб її робота була завжди коректною. Запропонуйте три різних варіанти управління.
2. Реалізуйте приклад Producer-Consumer application (див. <https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>)
_12 Модифікуйте масив даних цієї програми, які читаються, у масив чисел заданого розміру (100, 1000 або 5000) та протестуйте програму. Зробіть висновок про правильність роботи програми.
3. Реалізуйте роботу електронного журналу групи, в якому зберігаються оцінки з однієї дисципліни трьох груп студентів. Кожного тижня лектор і його 3 асистенти виставляють оцінки з дисципліни за 100-бальною шкалою.
4. Зробіть висновки про використання методів управління потоками в java.

Завдання 1

Лістинг коду:

AsynchBankTest.java

```
package Task1;

import java.util.Scanner;

/**
 * author Cay Horstmann
 */

public class AsynchBankTest {

    public static final int NACCOUNTS = 10;
    public static final int INITIAL_BALANCE = 10000;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```
System.out.print("Enter 1 for Default, 2 for Lock, 3 for SyncBlock, 4 for  
SyncMethod: ");
```

```
int chosenMethod = scanner.nextInt();
```

```
Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
```

```
int i;
```

```
for (i = 0; i < NACCOUNTS; i++){
```

```
    TransferThread t = new TransferThread(b, i,
```

```
        INITIAL_BALANCE,chosenMethod);
```

```
    t.setPriority(Thread.NORM_PRIORITY + i % 2);
```

```
    t.start () ;
```

```
    }
```

```
}
```

```
}
```

Bank.java

```
package Task1;
```

```
import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReentrantLock;
```

```
public class Bank {
```

```
    public static final int NTEST = 10000;
```

```
    private final int[] accounts;
```

```
    private long ntransacts = 0;
```

```
    private final Lock transLock = new ReentrantLock();
```

```
    public Bank(int n, int initialBalance){
```

```
        accounts = new int[n];
```

```

    int i;
    for (i = 0; i < accounts.length; i++)
        accounts[i] = initialBalance;
    ntransacts = 0;
}

public void transfer(int from, int to, int amount) {
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % NTEST == 0)
        test();
}

//модифікація
public void transferLock(int from, int to, int amount) {
    transLock.lock();
    try{
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts++;
        if (ntransacts % NTEST == 0)
            test();
    } finally {
        transLock.unlock();
    }
}

public void transferSyncB(int from, int to, int amount) {
    synchronized (this) {
        accounts[from] -= amount;

```

```

        accounts[to] += amount;
        ntransacts++;
        if (ntransacts % NTEST == 0)
            test();
    }
}

```

```

public synchronized void transferSyncM(int from, int to, int amount) {
    accounts[from] -= amount;
    accounts[to] += amount;
    ntransacts++;
    if (ntransacts % NTEST == 0)
        test();
}

```

```

public void test(){
    int sum = 0;
    for (int i = 0; i < accounts.length; i++)
        sum += accounts[i] ;
    System.out.println("Transactions:" + ntransacts
        + " Sum: " + sum);
}

public int size(){
    return accounts.length;
}
}

```

TransferThread.java

```

package Task1;

```

```

public class TransferThread extends Thread{

```

```
private final Bank bank;

private final int fromAccount;

private final int maxAmount;

private static final int REPS = 1000;

private final int manageMethod;

public TransferThread(Bank b, int from, int max, int manageMethod){

    bank = b;

    fromAccount = from;

    maxAmount = max;

    this.manageMethod = manageMethod;

}
```

@Override

```
public void run(){

    while (true) {

        for (int i = 0; i < REPS; i++) {

            int toAccount = (int) (bank.size() * Math.random());

            int amount = (int) (maxAmount * Math.random()/REPS);

            // Check the chosen transfer method

            switch (manageMethod) {

                case 1:

                    bank.transfer(fromAccount, toAccount, amount);

                    break;

                case 2:

                    bank.transferLock(fromAccount, toAccount, amount);

                    break;

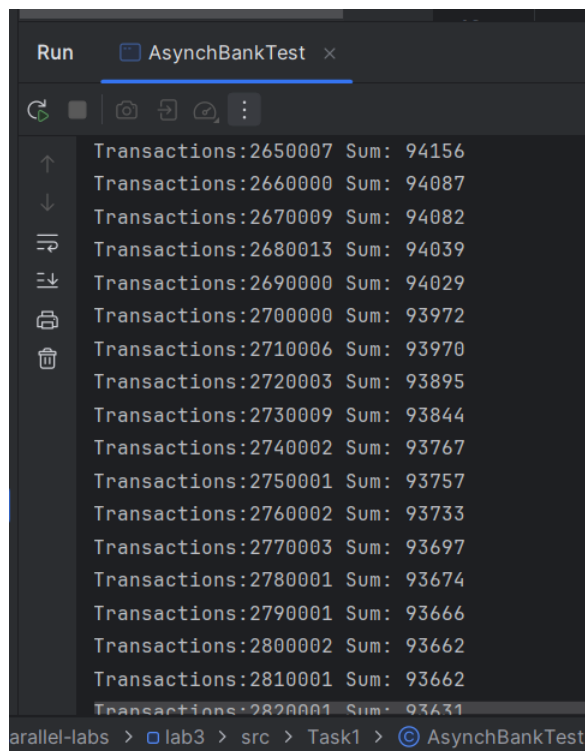
                case 3:
```

```

        bank.transferSyncB(fromAccount, toAccount, amount);
        break;
case 4:
        bank.transferSyncM(fromAccount, toAccount, amount);
        break;
default:
        bank.transfer(fromAccount, toAccount, amount); // Default method
        break;}}}}}}

```

Результати виконання коду:



```

Run AsynchBankTest x
Transactions:2650007 Sum: 94156
Transactions:2660000 Sum: 94087
Transactions:2670009 Sum: 94082
Transactions:2680013 Sum: 94039
Transactions:2690000 Sum: 94029
Transactions:2700000 Sum: 93972
Transactions:2710006 Sum: 93970
Transactions:2720003 Sum: 93895
Transactions:2730009 Sum: 93844
Transactions:2740002 Sum: 93767
Transactions:2750001 Sum: 93757
Transactions:2760002 Sum: 93733
Transactions:2770003 Sum: 93697
Transactions:2780001 Sum: 93674
Transactions:2790001 Sum: 93666
Transactions:2800002 Sum: 93662
Transactions:2810001 Sum: 93662
Transactions:2820001 Sum: 93431
parallel-labs > lab3 > src > Task1 > AsynchBankTest

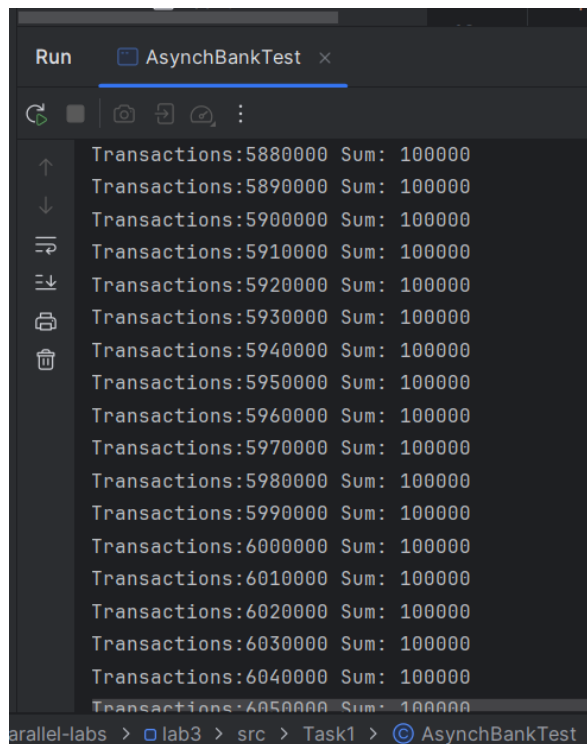
```

Рисунок 1.1 – Результат виконання коду.

Код протестовано при різних параметрах. При виведенні інформації про транзакції спостерігаємо помилкову кількість транзакцій(вона повинна бути кратна 10000) і помилкову загальну суму(вона повинна статичною і не змінювати незалежно від кількості здійснених транзакцій).

Модернізуємо код застосувавши 3 різні методи управління потоками, так, щоб робота програми була коректною.

Застосуємо `ReentrantLock`. Цей спосіб синхронізації доступу використовує явне створення об'єкта `ReentrantLock` та методи `lock()` і `unlock()` для заблокування та розблокування критичної секції. Це дає більшу гнучкість і контроль над блокуванням, так як можна заблокувати кілька ресурсів одночасно.



```
Run AsynchBankTest x
Transactions:5880000 Sum: 100000
Transactions:5890000 Sum: 100000
Transactions:5900000 Sum: 100000
Transactions:5910000 Sum: 100000
Transactions:5920000 Sum: 100000
Transactions:5930000 Sum: 100000
Transactions:5940000 Sum: 100000
Transactions:5950000 Sum: 100000
Transactions:5960000 Sum: 100000
Transactions:5970000 Sum: 100000
Transactions:5980000 Sum: 100000
Transactions:5990000 Sum: 100000
Transactions:6000000 Sum: 100000
Transactions:6010000 Sum: 100000
Transactions:6020000 Sum: 100000
Transactions:6030000 Sum: 100000
Transactions:6040000 Sum: 100000
Transactions:6050000 Sum: 100000
parallel-labs > lab3 > src > Task1 > AsynchBankTest
```

Рисунок 1.2 – Результат виконання коду з застосуванням ReentrantLock

Спостерігаємо правильні результати виводу інформації про перекази, маємо статичну загальну суму та кількість транзакцій кратні 10000.

Далі застосуємо блокування через об'єкт. Цей метод використовує ключове слово `synchronized`, щоб заблокувати виконання методу всередині поточного об'єкта. Це означає, що одночасно тільки один потік може виконувати цей метод на даному об'єкті. Однак, цей підхід менш гнучкий, оскільки потік може заблокувати весь об'єкт, не зважаючи на те, які саме ресурси він потребує.


```
Run AsynchBankTest x
Transactions:1550000 Sum: 100000
Transactions:1560000 Sum: 100000
Transactions:1570000 Sum: 100000
Transactions:1580000 Sum: 100000
Transactions:1590000 Sum: 100000
Transactions:1600000 Sum: 100000
Transactions:1610000 Sum: 100000
Transactions:1620000 Sum: 100000
Transactions:1630000 Sum: 100000
Transactions:1640000 Sum: 100000
Transactions:1650000 Sum: 100000
Transactions:1660000 Sum: 100000
Transactions:1670000 Sum: 100000
Transactions:1680000 Sum: 100000
Transactions:1690000 Sum: 100000
Transactions:1700000 Sum: 100000
Transactions:1710000 Sum: 100000
Transactions:1720000 Sum: 100000
parallel-labs > lab3 > src > Task1 > © AsynchBankTe
```

Рисунок 1.3 – Результат виконання коду з застосуванням блокування через об'єкт

Спостерігаємо правильні результати виводу інформації про перекази, маємо статичну загальну суму та кількість транзакцій кратні 10000.

Синхронізований метод. Цей метод використовує ключове слово `synchronized` для забезпечення потокобезпечного виклику методу. Коли один потік викликає цей метод на об'єкті, інші потоки будуть блоковані до тих пір, поки викликаний метод не завершить своє виконання. Це подібно до блокування через об'єкт, але відрізняється тим, що весь метод вважається синхронізованим, тобто всі його виклики будуть блокувати інші потоки, які намагаються отримати доступ до будь-яких синхронізованих методів цього об'єкта.

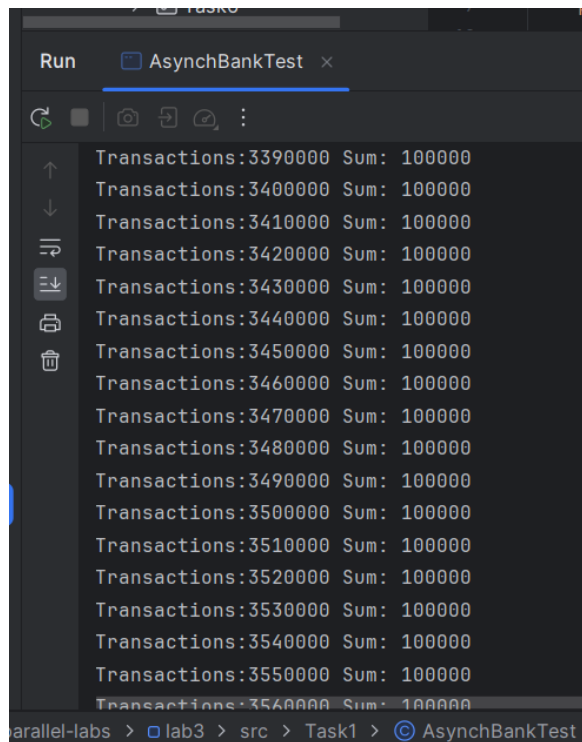


Рисунок 1.4 – Результат виконання коду з застосуванням синхронізованого методу

Спостерігаємо правильні результати виводу інформації про перекази, маємо статичну загальну суму та кількість транзакцій кратні 10000.

Всі три методи забезпечують потокобезпечний доступ до операцій з рахунками, уникнення гонок за ресурсами (race conditions) та інших проблем синхронізації даних між потоками.

Завдання 2

Лістинг коду:

Drop.java

```
package Task2;
```

```
public class Drop {
    // Message sent from producer
    // to consumer.
    private int message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
```

```
// consumer to retrieve message.
private boolean empty = true;

public synchronized int take() {
    // Wait until message is
    // available.
    while (empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = true;
    // Notify producer that
    // status has changed.
    notifyAll();
    return message;
}

public synchronized void put(int message) {
    // Wait until message has
    // been retrieved.
    while (!empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = false;
    // Store message.
```

```

        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}

```

Consumer.java

```

package Task2;

import java.util.Random;

public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (int message = drop.take(); message != -1; message = drop.take()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(100));
            } catch (InterruptedException e) {}
        }
    }
}

```

Producer.java

```

package Task2;

```

```
import java.util.Random;

public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {

        int size = 100;
        int[] importantInfo = new int[size];
        for (int i = 0; i < size; i++)
            importantInfo[i] = i+1;

        Random random = new Random();

        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(100));
            } catch (InterruptedException e) {}
        }
        drop.put(-1);
    }
}
```

ProducerConsumerExample.java

```
package Task2;
```

```
public class ProducerConsumerExample {  
    public static void main(String[] args) {  
        Drop drop = new Drop();  
        (new Thread(new Producer(drop))).start();  
        (new Thread(new Consumer(drop))).start();  
    }  
}
```

Результати виконання коду:

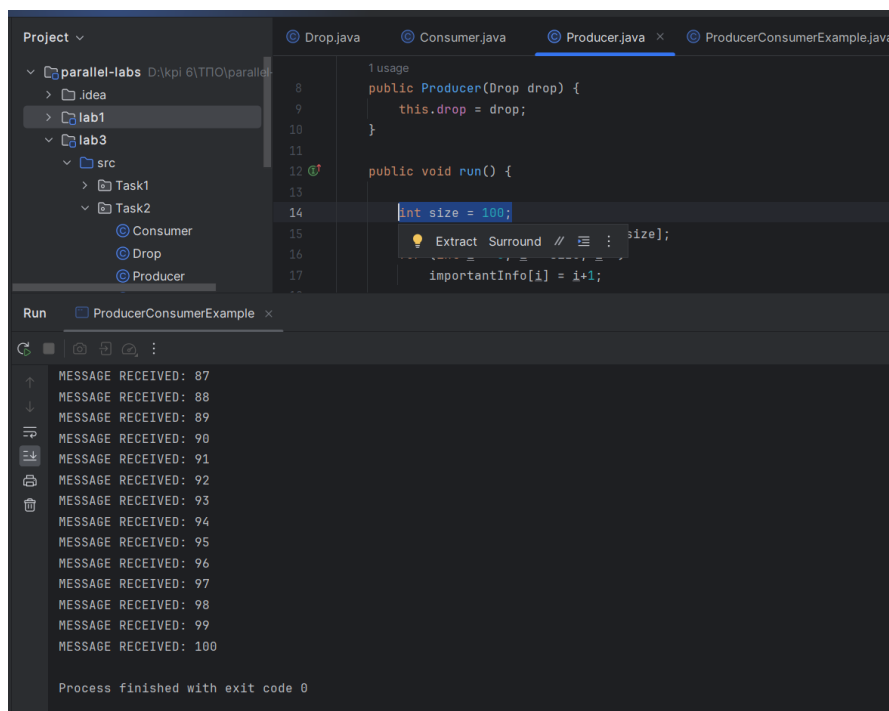


Рисунок 2.1 – Тестування коду з масивом чисел від 1 до 100

З результатів тестування спостерігаємо, що наша програма відпрацьовує правильно, забезпечуючи взаємодію між виробником та споживачем з використанням синхронізації потоків та обміном повідомленнями через об'єкт `Drop`. Виробник відправляє послідовно числа від 1 до 100, а потім відправляє спеціальне повідомлення -1, що сигналізує про завершення виробництва. Споживач отримує та виводить отримані повідомлення у консоль, поки не отримає спеціальне повідомлення про завершення виробництва.

Завдання 3

Лістинг коду:

Group.java

```
package Task3;
```

```
import java.util.ArrayList;
```

```
public record Group(int id, ArrayList<Student> students) {  
    public Student getStudent(int id) {  
        return students.get(id);  
    }  
    public int getStudentsNumber() {  
        return students.size();  
    }  
  
    public int getGroupNumber(){  
        return id+11;  
    }  
    public int getgroupid(){  
        return id;  
    }  
}
```

Journal.java

```
package Task3;
```

```
import java.util.ArrayList;
```

```
import java.util.concurrent.ConcurrentHashMap;
```

```
public class Journal {  
    private final ArrayList<Group> groups;
```

```
private final int weeks;

private final ConcurrentHashMap<String, ConcurrentHashMap<Integer,
Integer>> marks = new ConcurrentHashMap<>();
```

```
public Journal(ArrayList<Group> groups, int weeks) {
    this.groups = groups;
    this.weeks = weeks;
}
```

```
public int getWeeks() {
    return weeks;
}
```

```
public ArrayList<Group> getGroups() {
    return groups;
}
```

```
public void addMark(int mark, int week, int groupId, int studentId) {
    Group group = groups.get(groupId);
    Student student = group.getStudent(studentId);

    String key = student.getId() + "-" + group.id();
```

```
    ConcurrentHashMap<Integer, Integer> weekMarks =
marks.computeIfAbsent(key, k -> new ConcurrentHashMap<>());
    weekMarks.putIfAbsent(week, mark);
}
```

```
public int getMark(int week, int groupId, int studentId) {
    Group group = groups.get(groupId);
    Student student = group.getStudent(studentId);
```



```

String key = student.getId() + "-" + group.id();
ConcurrentHashMap<Integer, Integer> weekMark = marks.get(key);

return weekMark.getDefault(week, -1);
}

public void printMarks() {

    System.out.println("\n-----\n");

    for (Group group : groups) {
        System.out.println("III-" + group.getGroupNumber());
        for (int week = 1; week <= weeks; week++) {
            if (week == 1) {
                System.out.printf("%-12s", " ");
            }
            System.out.printf("%-10d", week);
        }

        System.out.println();
        System.out.println();

        for (Student student : group.students()) {
            System.out.print("Студент " + student.getId() + ": ");
            for (int week = 1; week <= weeks; week++) {
                int mark = getMark(week, group.id(), student.getId());
                System.out.printf("%-10d", mark);
            }
            System.out.println();
        }
    }
}

```

```
    }  
    System.out.println();  
}  
}  
}
```

Main.java

```
package Task3;  
  
import java.util.ArrayList;  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        ArrayList<Group> groups = generateGroups(3, 3);  
  
        Journal journal = new Journal(groups, 3);  
  
        PuttingMarksThread markThread1 = new PuttingMarksThread("Лектор",  
journal, groups);  
        PuttingMarksThread markThread2 = new PuttingMarksThread("Асистент1",  
journal, groups);  
        PuttingMarksThread markThread3 = new PuttingMarksThread("Асистент2",  
journal, groups);  
        PuttingMarksThread markThread4 = new PuttingMarksThread("Асистент3",  
journal, groups);  
  
        markThread1.start();  
        markThread2.start();  
        markThread3.start();  
        markThread4.start();  
    }  
}
```

```

        try {
            markThread1.join();
            markThread2.join();
            markThread3.join();
            markThread4.join();
        } catch (InterruptedException e) {
            //
        }

        journal.printMarks();

    }

    public static ArrayList<Student> generateStudents(int amountOfStudents) {
        ArrayList<Student> students = new ArrayList<>();
        for (int i = 0; i < amountOfStudents; i++) {
            students.add(new Student("Student " + i, i));
        }
        return students;
    }

    public static ArrayList<Group> generateGroups(int amountOfGroups, int
amountOfStudents) {
        ArrayList<Group> groups = new ArrayList<>();
        for (int i = 0; i < amountOfGroups; i++) {
            groups.add(new Group(i, generateStudents(amountOfStudents)));
        }
        return groups;
    }

```

```
}
```

PuttingMarksThread.java

```
package Task3;
```

```
import java.util.ArrayList;
```

```
public class PuttingMarksThread extends Thread {
```

```
    String teacherPost;
```

```
    ArrayList<Group> groups;
```

```
    private final Journal journal;
```

```
    public PuttingMarksThread(String teacherPost, Journal journal,  
        ArrayList<Group> groups) {
```

```
        this.groups = groups;
```

```
        this.journal = journal;
```

```
        this.teacherPost = teacherPost;
```

```
    }
```

```
    public int generateMark() {
```

```
        return (int) (Math.random() * 101);
```

```
    }
```

```
@Override
```

```
public void run() {
```

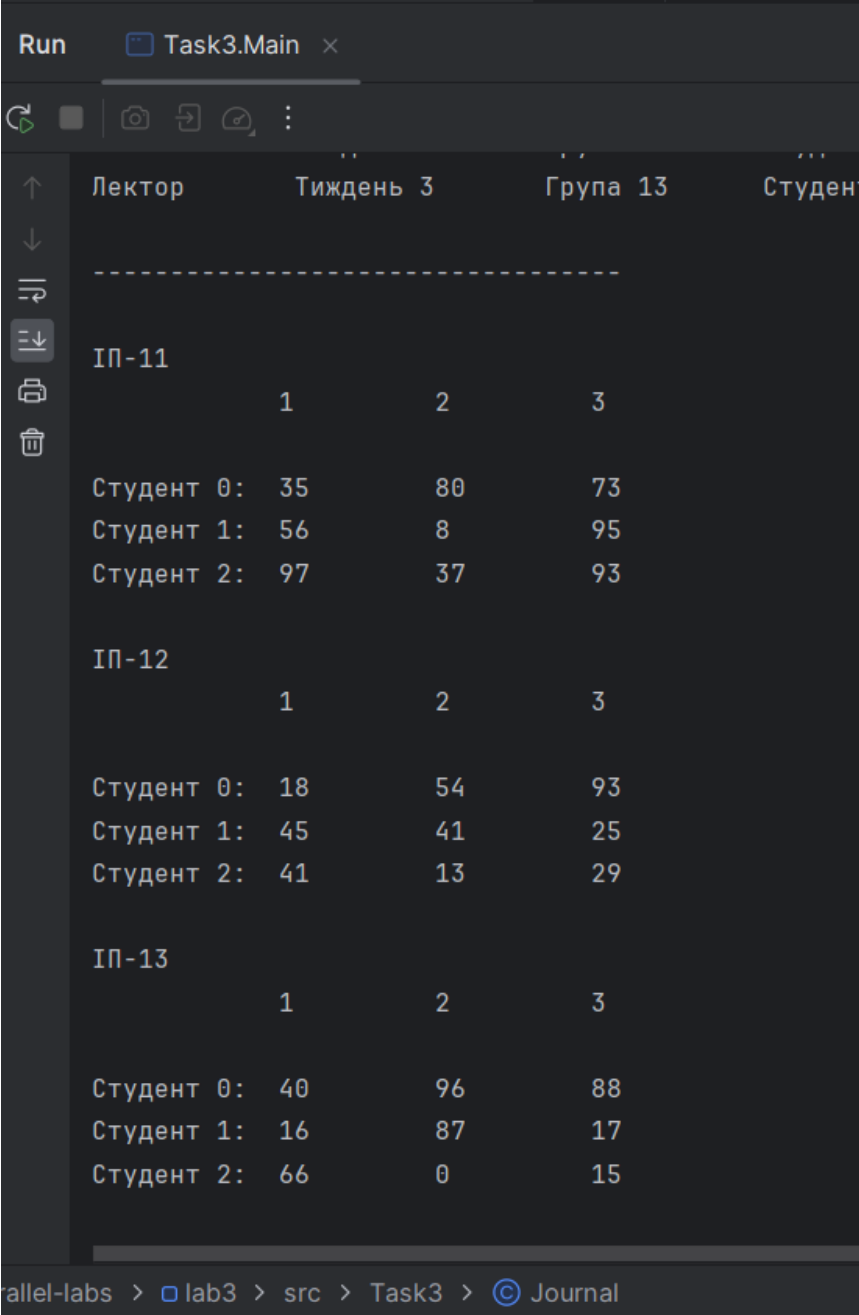
```
    for (int week = 1; week <= journal.getWeeks(); week++) {
```

```
        for (int groupid = 0; groupid < groups.size(); groupid++) {
```

```
            int numStudents =
```

```
            journal.getGroups().get(groupid).getStudentsNumber();
```


Результати виконання коду:



Лектор	Тижень 3	Група 13	Студент

IP-11			
	1	2	3
Студент 0:	35	80	73
Студент 1:	56	8	95
Студент 2:	97	37	93
IP-12			
	1	2	3
Студент 0:	18	54	93
Студент 1:	45	41	25
Студент 2:	41	13	29
IP-13			
	1	2	3
Студент 0:	40	96	88
Студент 1:	16	87	17
Студент 2:	66	0	15

Рисунок 3.1 – Результат виконання коду

Створено електронний журнал, в якому виставляються оцінки паралельно 4 потоками(лектор та 3 асистенти) протягом кількох тижнів. Для синхронізації потоків та забезпечення багатопотокової безпеки було використано ConcurrentHashMap це одна з імплементацій інтерфейсу Map у Java, що забезпечує потокобезпечність при одночасному доступі з різних потоків.

Завдання 4

Використання потоків у Java може значно поліпшити продуктивність програми, дозволяючи виконувати кілька завдань одночасно. Однак необхідно ретельно керувати потоками, оскільки може виникати гонка за ресурси, що може призвести до некоректної роботи програми. Для уникнення таких ситуацій існують різні методи управління потоками.

Синхронізація - один з основних методів управління потоками, який допомагає уникнути гонок за ресурсами. Вона може бути реалізована за допомогою різних механізмів, таких як синхронізація методу, блоку, локери, блокування об'єкту тощо. Для цього використовуються ключові слова, такі як `synchronized`, `wait`, `notify`, `lock` та інші. Також існує `ConcurrentHashMap` це одна з імплементацій інтерфейсу `Map` у Java, що забезпечує потокобезпечність при одночасному доступі з різних потоків. `ConcurrentHashMap` забезпечує атомарні операції для багатьох своїх методів. Атомарність означає, що операція виконується як єдине ціле, не розбивається на окремі кроки, і неможливо перервати її в процесі виконання або втратити її результат через одночасний доступ з іншого потоку.

Правильне використання механізмів синхронізації дозволяє уникнути гонок за даними, забезпечити взаємовиключення та відновлення потоків. Це допомагає створити ефективні, надійні та безпечні багатопотокові програми, які можуть працювати коректно та ефективно в умовах конкуренції за ресурси.

Висновок

При виконанні лабораторної роботи ми ознайомились з основними механізмами синхронізації потоків у Java та вивчили їхнє практичне використання. Під час виконання роботи ми зрозуміли важливість управління потоками для уникнення гонок за ресурсами та забезпечення коректної роботи програми в умовах конкуренції. Освоєння різних методів синхронізації дало нам можливість ефективно керувати взаємодією між потоками та забезпечити правильну синхронізацію доступу до спільних ресурсів. Для забезпечення безпеки багатопоточної реалізації також було застосовано імплементацію інтерфейсу `Map` у Java, що забезпечує потокобезпечність при одночасному доступі з різних потоків.

Отримані знання та практичні навички з синхронізації потоків у Java виявляться корисними при розробці багатопотокових додатків, де необхідно забезпечити безпеку та надійність при одночасному доступі до спільних даних.