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

Кафедра інформатики та програмної інженерії

Звіт

з комп’ютерного практикуму №3 з дисципліни
«Технології паралельних обчислень»

«Розробка паралельних програм з використанням механізмів синхронізації: синхронізовані методи, локери, спеціальні типи»

Виконав(ла) *ІП-11 Прищепа В. С.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Дифучин А. Ю.* _____
(прізвище, ім'я, по батькові)

Київ 2024

Завдання

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. 30 балів.

2. Реалізуйте приклад Producer-Consumer application (див. https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html). 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. 20 балів.

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. 40 балів.

4. Зробіть висновки про використання методів управління потоками в java. 10 балів.

Виконання

1 Завдання

Лістинг

AsyncBankTest.java

```
package org.example.task1;

public class AsyncBankTest {
    //    VARIANT can range from 1 to 3
    public static final int VARIANT = 3;
    public static final int NACCOUNTS = 10;
    public static final int INITIAL_BALANCE = 10000;

    public static void main(String[] args) {
        System.out.println("\nVariant #" + VARIANT +
                "\n-----------------------------------"+
                "\nNACCOUNTS = " + NACCOUNTS +
                "\nINITIAL_BALANCE = " + INITIAL_BALANCE +
                "\nInitial total bank: " + NACCOUNTS * INITIAL_BALANCE +
                "\n-----------------------------------");
        Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
```

```
        for (int i = 0; i < NACCOUNTS; i++) {
            TransferThread thread = new TransferThread(bank, i, INITIAL_BALANCE,
VARIANT);
            thread.setPriority(Thread.NORM_PRIORITY + i % 2);
            thread.start();
        }
    }
}
```

Bank.java

```
package org.example.task1;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import java.util.Arrays;
import java.util.concurrent.atomic.AtomicLong;

public class Bank {
    public static final int NTEST = 10000;

    private final int[] accounts;
    private final AtomicLong ntransacts = new AtomicLong(0);

    public Bank(int len, int initBalance) {
        accounts = new int[len];
        Arrays.fill(accounts, initBalance);
    }

    //    variant 1
    public synchronized void transferV1(int from, int to, int amount) {
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts.incrementAndGet();
        if (ntransacts.get() % NTEST == 0) {
            test();
            Thread.currentThread().interrupt();
        }
    }

    //    variant 2
    public void transferV2(int from, int to, int amount) {
        synchronized (accounts) {
            accounts[from] -= amount;
            accounts[to] += amount;
            if (ntransacts.incrementAndGet() % NTEST == 0) {
```

```
            test();
            Thread.currentThread().interrupt();
        }
    }
}

//    variant 3
private final Lock lock = new ReentrantLock();
public void transferV3(int from, int to, int amount) {
    try {
        lock.lock();
        accounts[from] -= amount;
        accounts[to] += amount;
        if (ntransacts.incrementAndGet() % NTEST == 0) {
            test();
            Thread.currentThread().interrupt();
        }
    } finally {
        lock.unlock();
    }
}

public void test() {
    int sum = 0;
    for (int account : accounts) {
        sum += account;

    }
    System.out.println("Transactions: " + ntransacts + " | Sum: " + sum);
}

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

TransferThread.java

```
package org.example.task1;

public class TransferThread extends Thread {
    private final Bank bank;
    private final int fromAccount;
    private final int maxAmount;
    private final int variant;
```

```
    private static final int REPS = 1000;

    public TransferThread(Bank bank, int fromAccount, int maxAmount, int variant) {
        this.bank = bank;
        this.fromAccount = fromAccount;
        this.maxAmount = maxAmount;
        this.variant = variant;
    }

    @Override
    public void run(){
        while (!Thread.currentThread().isInterrupted()) {
            int toAccount = (int) (bank.size() * Math.random());
            int amount = (int) (maxAmount * Math.random() / REPS);
            switch (variant) {
                case 1:
                    bank.transferV1(fromAccount, toAccount, amount);
                    break;
                case 2:
                    bank.transferV2(fromAccount, toAccount, amount);
                    break;
                case 3:
                    bank.transferV3(fromAccount, toAccount, amount);
                    break;
                default:
                    System.out.println("Wrong variant provided!");
                    return;
            }
        }
    }
}
```

Результат

```
Variant #1
-----------------------------------
NACCOUNTS = 10
INITIAL_BALANCE = 10000
Initial total bank: 100000
-----------------------------------
Transactions: 10000 | Sum: 100000
Transactions: 20000 | Sum: 100000
Transactions: 30000 | Sum: 100000
Transactions: 40000 | Sum: 100000
Transactions: 50000 | Sum: 100000
Transactions: 60000 | Sum: 100000
Transactions: 70000 | Sum: 100000
Transactions: 80000 | Sum: 100000
Transactions: 90000 | Sum: 100000
Transactions: 100000 | Sum: 100000
```

Варіант із синхронізованим методом.

```
Variant #2
-----------------------------------
NACCOUNTS = 10
INITIAL_BALANCE = 10000
Initial total bank: 100000
-----------------------------------
Transactions: 10000 | Sum: 100000
Transactions: 20000 | Sum: 100000
Transactions: 30000 | Sum: 100000
Transactions: 40000 | Sum: 100000
Transactions: 50000 | Sum: 100000
Transactions: 60000 | Sum: 100000
Transactions: 70000 | Sum: 100000
Transactions: 80000 | Sum: 100000
Transactions: 90000 | Sum: 100000
Transactions: 100000 | Sum: 100000
```

Варіант із синхронізованим об’єктом

```
Variant #3
-----------------------------------
NACCOUNTS = 10
INITIAL_BALANCE = 10000
Initial total bank: 100000
-----------------------------------
Transactions: 10000 | Sum: 100000
Transactions: 20000 | Sum: 100000
Transactions: 30000 | Sum: 100000
Transactions: 40000 | Sum: 100000
Transactions: 50000 | Sum: 100000
Transactions: 60000 | Sum: 100000
Transactions: 70000 | Sum: 100000
Transactions: 80000 | Sum: 100000
Transactions: 90000 | Sum: 100000
Transactions: 100000 | Sum: 100000
```

Варіант із локером.

Усі три запропоновані модифікації видали один і той самий коректний результат. Бачимо, що сума завжди залишається 100000, оскільки лише один потік в одиницю часу модифікує масив.

2 Завдання

Лістинг

Consumer.java

```
package org.example.task2;

public class Consumer implements Runnable {
    private final Object terminator;
    private final Drop<?> drop;
    private final boolean isRandom;

    public Consumer(Drop<?> drop, Object terminator, boolean isRandom) {
        this.drop = drop;
        this.terminator = terminator;
        this.isRandom = isRandom;
    }

    public void run() {
        for (Object message = drop.take(); !message.equals(terminator); message =
drop.take()) {
            if (isRandom) System.out.format("Received message: %s%n", message);
            else System.out.format("Received message #%s%n", message);
        }
    }
}
```

Drop.java

```
package org.example.task2;

public class Drop<T> {
    // Message sent from producer
    // to consumer.
    private T 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 T take() {
```

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

    public synchronized void put(T message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException ignored) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

Producer.java

```
package org.example.task2;

import java.util.Random;

public class Producer implements Runnable {
    private final int size;
    private final Drop<Integer> drop;
    private final Integer terminator;
    private final boolean isRandom;
    private final Random random = new Random();
```

```
    public Producer(Drop<Integer> drop, int size, Integer terminator, boolean
isRandom) {
        this.drop = drop;
        this.size = size;
        this.terminator = terminator;
        this.isRandom = isRandom;
    }

    public void run() {
        for (int i = 0; i < size; i++) {
            if (isRandom) drop.put(random.nextInt(Integer.MAX_VALUE));
            else drop.put(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException ignored) {}
        }
        drop.put(terminator);
    }

}
```

ProducerConsumerExample.java

```
package org.example.task2;

public class ProducerConsumerExample {
    private static final int SIZE = 100;
    private static final Integer TERMINATOR = -1;
    private static final boolean ISRANDOM = false;

    public static void main(String[] args) {
        final Drop<Integer> drop = new Drop<>();

        (new Thread(new Producer(drop, SIZE, TERMINATOR, ISRANDOM))).start();
        (new Thread(new Consumer(drop, TERMINATOR, ISRANDOM))).start();
    }
}
```

Результат

```
Received message #88
Received message #89
Received message #90
Received message #91
Received message #92
Received message #93
Received message #94
Received message #95
Received message #96
Received message #97
Received message #98
Received message #99

Process finished with exit code 0
```

У другому завданні масив рядків був замінений на масив цілих чисел.

Результатом стало правильне виводження чисел.

3 Завдання

Лістинг

Group.java

```java
package org.example.task3;

import java.util.ArrayList;
import java.util.Iterator;
public class Group implements Iterable<Student> {
    private final ArrayList<Student> students;
    private final String name;
    public Group(String name, int totalStudents) {
        this.name = name;
        this.students = new ArrayList<>(totalStudents);
        for(var i = 0; i < totalStudents; i++) {
            students.add(new Student("Student " + i, this));
        }
    }
    public String getName() {
        return name;
    }
    public static synchronized void printGrades(Group g) {
        for(var s : g) {
            s.printGrades();
        }
    }
    @Override
  public Iterator<Student> iterator() {
```

```
        return students.iterator();
    }
}
```

Student.java

```
package org.example.task3;

import java.util.ArrayList;
public class Student {
    private final ArrayList<Integer> grades = new ArrayList<>();
    private final String name;
    private final Group group;
    public Student(String name, Group group) {
        this.group = group;
        this.name = name;
    }
    public synchronized void addGrade(int grade) {
        grades.add(grade);
    }
    public synchronized void printGrades() {
        System.out.println(group.getName() + " " + name + ": ");
        grades.stream().limit(grades.size() - 1).toList().forEach(g ->
                System.out.print(g + ","));
        System.out.print(grades.get(grades.size() - 1) + "\n");
    }
}
```

StudentTeacherExample.java

```
package org.example.task3;

public class StudentTeacherExample {
    private static final int TOTAL_TEACHERS = 4;
    private static final int TOTAL_GROUPS = 3;
    private static final int TOTAL_STUDENTS_IN_GROUP = 5;
    public static void main(String[] args) {
        var groups = new Group[TOTAL_GROUPS];
        for(var i = 0; i < TOTAL_GROUPS; i++) {
            groups[i] = new Group("Group " + i,
                    TOTAL_STUDENTS_IN_GROUP);
        }
        var teachers = new TeacherThread[TOTAL_TEACHERS];
        for(var i = 0; i < TOTAL_TEACHERS; i++) {
            teachers[i] = new TeacherThread(groups);
            teachers[i].start();
        }
```

```
        for(var i = 0; i < TOTAL_TEACHERS; i++) {
            try {
                teachers[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        }
}
```

TeacherThread.java

```
package org.example.task3;

public class TeacherThread extends Thread {
    private final Group[] groups;

    public TeacherThread(Group[] groups) {
        this.groups = groups;
    }

    @Override
    public void run() {
        while (true) {
            for (var g : groups) {
                for (var s : g) {
                    s.addGrade((int) (100 * Math.random()));
                }
                Group.printGrades(g);
            }
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Результат

У третьому завданні синхронізація досягається синхронізованим методом addGrade.

Для синхронізованого виведення оцінок групи використав статичний метод printGrades, тобто монітор береться не на об’єкт класу, а на сам клас, тобто може викликатися один статичний синхронізований метод в одиницю часу.

Висновок: Управління потоками в Java може бути здійснене за допомогою ключових слів synchronized, використання блоків synchronized, а також замків (locks), таких як ReentrantLock.

Ключове слово synchronized: Простий у використанні, вона автоматично забезпечує блокування на рівні методу або об'єкта. Може бути застосовано до методів, конструкторів та блоків коду.

Блок synchronized: Дозволяє заблокувати тільки певний фрагмент коду, а не весь метод або об'єкт. Це дає більшу гнучкість у контролі за блокуванням.

Замки (locks): Надають ще більшу гнучкість та контроль за блокуванням, особливо за допомогою ReentrantLock. Вони можуть бути застосовані для складних сценаріїв синхронізації та дозволяють здійснювати спеціальні операції, такі як очікування на спеціальних умовах. Однак вони також

вимагають вручного керування викликами lock() та unlock(), що може збільшити складність коду та його вразливість до помилок.

Вибір методу управління потоками залежить від конкретних потреб програми, складності сценаріїв синхронізації та простоти використання.