**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**

Комп‘ютерного практикуму № 3 з дисципліни

«Технології паралельних та розподілених обчислень»

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





Київ 2022

## 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.**

## Хід роботи

У рамках даного завдання було реалізовано функціонал банку, який дозволяє виконувати операції переказу коштів між рахунками. Однак, під час тестування коду, було виявлено проблему, коли сума на рахунках зменшувалась з часом, незважаючи на правильність виконання операцій переказу (рис. 1). Сума на рахунках постійно зменшувалась, хоча вона повинна бути константною. Це свідчило про проблему зі синхронізацією доступу до спільних ресурсів.

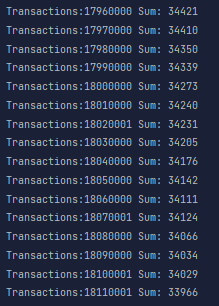


Рис. 1 – Вивід суми при несинхронізованому виконанні

Для вирішення проблеми, було запропоновано кілька способів модифікації програми з метою забезпечення правильності результатів (рис. 2):

1. Використання synchronized у методі syncMethodTransfer(). Це дозволяє забезпечити взаємовиключення між потоками, що працюють зі спільними ресурсами.
2. Обгортання ділянки читання та запису в синхронізований блок у методі syncBlockTransfer(). Також гарантує взаємовиключення під час доступу до спільних ресурсів.
3. Використання локерів ReentrantLock у методі lockTransfer(). Дозволяє захоплювати та звільняти захоплення ресурсів, забезпечує більш гнучкий контроль над синхронізацією.

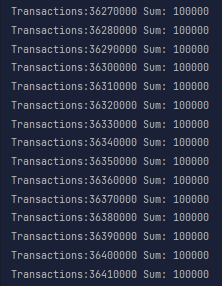


Рис. 2 – Вивід суми при синхронізованому виконанні

## 

## Завдання 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.**

## Хід роботи

Клас `Drop`, який представляє собою буфер. Усередині цього класу було створено синхронізовані методи з умовою та wait(), notifyAll(), що використовуються для синхронної роботи з буфером. Умова `pointerOnEmpty` вказує на те, коли буфер не повний або не порожній.

Далі було реалізовано два методи - `put` та `take` . У цих методах виконується перевірка, чи не вийшла кількість поточних елементів за максимальну довжину масиву, а також чи не дорівнює нулю (пустий або порожній). Якщо ці умови виконуються, потік чекає, поки інший потік забере або положить елемент і повідомляє про це.

В результаті тестування не виникло помилок виходу за межі масиву, що свідчить про коректну роботу програми (рис. 3). Використання відповідного функціоналу дозволило забезпечити синхронізацію роботи зі спільним буфером та запобігти гонкам за ресурсами.

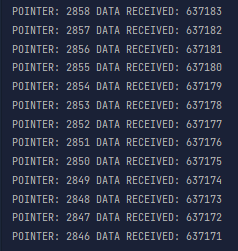


Рис. 3 – Коректне виконання Producer-Consumer для масиву розміру 5000

## 

## Завдання 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.**

## Хід роботи

У коді були реалізовані наступні класи:

1. "SubjectJournal” – електронний журнал дисципліни. Використовує `ConcurrentHashMap` для збереження оцінок студентів. Містить методи для додавання оцінок та обчислення середнього балу для студента.
2. “Group” – група студентів. Має унікальний ідентифікатор та містить список студентів.
3. “Student” – студент із унікальним ідентифікатором.
4. “Teacher” – викладач із унікальним ідентифікатором має змогу заповнювати журнал оцінок у декількох груп.
5. “addMarkThread”: реалізує потік для додавання оцінок до журналу. Генерує випадкові оцінки та додає їх до відповідних студентів у журналі.

Вивід результату:

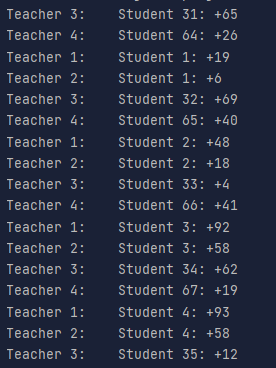


Рис. 4 – Демонстрація роботи потоків

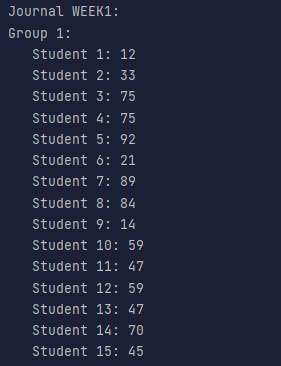


Рис. 5 – Вивід середніх оцінок першого тижня

Результати відображають успішну реалізацію електронного журналу. Середні оцінки для студентів були виведені правильно.

## 

## Висновок

Використання методів управління потоками в Java є важливою та потужною функціональністю для розробки багатопотокових програм. Ось декілька висновків:

1. Синхронізація потоків дозволяє забезпечити взаємовиключення доступу до спільних ресурсів. Використання ключового слова `synchronized`, синхронізованих блоків або об'єктів `Lock` допомагає уникнути гонок за ресурсами.

2. Методи очікування і сповіщення (“wait()”, “notifyAll()”), дозволяють потокам ефективно спілкуватися між собою та синхронізувати свою роботу. Вони дозволяють потокам чекати на певні умови, поки інші потоки не виконають певні дії чи не змінять стан.

4. Управління потоками дозволяє досягти кращої продуктивності та швидкодії шляхом використання багатопотоковості. Також дозволяє використовувати ефективніше ресурси системи та знижувати час виконання завдань.

5. З мінусів: може створюватися складність, пов'язана зі синхронізацією, гонками за ресурсами та дедлоками. Неправильне використання синхронізації може призводити до некоректних результатів або невизначеного стану програми.

## 

## 

## Код завдання 1

public class **AsyncBankTest** {

public static final int *NACCOUNTS* = 10;

public static final int *INITIAL\_BALANCE* = 10000;

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

**Bank** b = new Bank(*NACCOUNTS*, *INITIAL\_BALANCE*);

for (int i = 0; i < *NACCOUNTS*; i++) {

**TransferThread** t = new TransferThread(b, i, *INITIAL\_BALANCE*);

t.setPriority(**Thread**.*NORM\_PRIORITY* + i % 2);

t.start();

}

}

}

public class **TransferThread** extends **Thread** {

private **Bank** bank;

private int fromAccount;

private int maxAmount;

private static final int *REPS* = 1000;

public TransferThread(**Bank** b, int from, int maxAmount) {

bank = b;

fromAccount = from;

this.maxAmount = maxAmount;

}

@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*);

// bank.transfer(fromAccount, toAccount, amount);

// bank.syncMethodTransfer(fromAccount, toAccount, amount);

// bank.syncBlockTransfer(fromAccount, toAccount, amount);

bank.lockTransfer(fromAccount, toAccount, amount);

}

}

}

}

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

public class **Bank** {

public static final int *NTEST* = 10000;

private final int[] accounts;

private long transacts = 0;

private final **ReentrantLock** bankLock = new ReentrantLock();

public Bank(int n, int initialBalance) {

accounts = new int[n];

int i;

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

transacts = 0;

}

public void transfer(int from, int to, int amount) {

accounts[from] -= amount;

accounts[to] += amount;

transacts++;

if (transacts % *NTEST* == 0) test();

}

public void syncBlockTransfer(int from, int to, int amount) {

synchronized (this) {

accounts[from] -= amount;

accounts[to] += amount;

transacts++;

if (transacts % *NTEST* == 0) test();

}

}

public synchronized void syncMethodTransfer(int from, int to, int amount) {

transfer(from, to, amount);

}

public void lockTransfer(int from, int to, int amount) {

bankLock.lock();

try {

accounts[from] -= amount;

accounts[to] += amount;

transacts++;

if (transacts % *NTEST* == 0) test();

} finally {

bankLock.unlock();

}

}

public void test() {

int sum = 0;

for (int account : accounts) sum += account;

**System**.*out*.println("Transactions:" + transacts + " Sum: " + sum);

}

public int size() {

return accounts.length;

}

}

## Код завдання 2

## public class ProducerConsumerExample {

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

## Drop drop = new Drop(5000);

## (new Thread(new Producer(drop))).start();

## (new Thread(new Consumer(drop))).start();

## }

## }

## 

## import java.util.Random;

## 

## public class Producer extends Thread {

## public static int *value* = 1;

## private Drop drop;

## 

## public Producer(Drop drop) {

## this.drop = drop;

## }

## 

## @Override

## public void run() {

## while (true) {

## drop.put(*value*++);

## }

## }

## }

## 

## import java.util.Random;

## 

## public class Consumer extends Thread {

## private Drop drop;

## 

## public Consumer(Drop drop) {

## this.drop = drop;

## }

## 

## @Override

## public void run() {

## while (true) {

## int data = drop.take();

## }

## }

## }

## 

## public class Drop {

## private int[] data;

## private int pointerOnEmpty = 0;

## 

## public Drop(int size) {

## data = new int[size];

## }

## 

## public synchronized int take() {

## while (pointerOnEmpty == 0) {

## try {

## wait();

## } catch (InterruptedException e) {}

## }

## int value = data[pointerOnEmpty-1];

## pointerOnEmpty--;

## notifyAll();

## 

## System.*out*.format("POINTER: %-4s DATA RECEIVED: %s%n", pointerOnEmpty, value);

## return value;

## }

## 

## public synchronized void put(int value) {

## while (pointerOnEmpty >= data.length) {

## try {

## wait();

## } catch (InterruptedException e) {}

## }

## data[pointerOnEmpty++] = value;

## notifyAll();

## }

## }

## 

## Код завдання 3

import **java.util.ArrayList**;

public class **GradesJournalTest** {

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

**SubjectJournal** mathJournal = new SubjectJournal();

**Group** group1 = new Group(30);

**Group** group2 = new Group(33);

**Group** group3 = new Group(35);

mathJournal.addGroup(group1);

mathJournal.addGroup(group2);

mathJournal.addGroup(group3);

**Teacher** lector = new Teacher(mathJournal, group1);

lector.addGroup(group2);

lector.addGroup(group3);

**Teacher** assistant1 = new Teacher(mathJournal, group1);

**Teacher** assistant2 = new Teacher(mathJournal, group2);

**Teacher** assistant3 = new Teacher(mathJournal, group3);

**ArrayList**<**Thread**> threads = new ArrayList<>();

for (int i = 0; i < 3; i++) {

threads.add(lector.fillJournal());

threads.add(assistant1.fillJournal());

threads.add(assistant2.fillJournal());

threads.add(assistant3.fillJournal());

try {

for (**Thread** thread : threads) {

thread.join();

}

} catch (**InterruptedException** e) {

e.printStackTrace();

}

**System**.*out*.println("Journal WEEK" + (i+1) + ":");

mathJournal.printJournalAverage();

}

}

}

import **java.util.ArrayList**;

public class **Teacher** {

public static int *ID* = 1;

private final int PersonalID;

private **SubjectJournal** journal;

private **ArrayList**<**Group**> groups = new ArrayList<>();

public Teacher(**SubjectJournal** journal, **Group** group) {

PersonalID = *ID*++;

this.journal = journal;

groups.add(group);

}

public void addGroup(**Group** group) {

journal.addGroup(group);

}

public **Thread** fillJournal() {

**addMarkThread** thread = new addMarkThread(journal, groups, PersonalID);

thread.start();

return thread;

}

}

import **java.util.ArrayList**;

public class **Group** {

public static int *GroupsID* = 1;

private **ArrayList**<**Student**> students;

private final int ID;

public Group(int size) {

ID = *GroupsID*++;

students = new ArrayList<>(size);

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

students.add(new Student());

}

}

public void addStudent(**Student** student) {

students.add(student);

}

public void removeStudent(**Student** student) {

students.remove(student);

}

public int getSize() {

return students.size();

}

public **Student** getStudent(int i) {

return students.get(i);

}

public **Integer** getID() {

return ID;

}

}

public class **Student** {

public static int *CurrentID* = 1;

private final int PersonalID;

public Student() {

PersonalID = *CurrentID*++;

}

public **Integer** getID() {

return PersonalID;

}

}

import **java.util.ArrayList**;

import **java.util.concurrent.BlockingQueue**;

import **java.util.concurrent.ConcurrentHashMap**;

public class **SubjectJournal** {

**ConcurrentHashMap**<**Student**, **ArrayList**<**Integer**>> journal = new ConcurrentHashMap<>();

**ArrayList**<**Group**> groups = new ArrayList<>();

public void addGroup(**Group** group) {

groups.add(group);

for (int i = 0; i < group.getSize(); i++) {

if (!journal.containsKey(group.getStudent(i))) {

journal.put(group.getStudent(i), new ArrayList<>());

}

}

}

public synchronized void addMark(**Student** student, int mark) {

journal.get(student).add(mark);

}

public int getAverageMark(**Student** student) {

int sum = 0;

for (int mark : journal.get(student)) {

sum += mark;

}

return sum / journal.get(student).size();

}

public void printJournalAverage() {

for (**Group** group : groups) {

**System**.*out*.println("Group " + group.getID() + ":");

for (int i = 0; i < group.getSize(); i++) {

**System**.*out*.println(" Student " + group.getStudent(i).getID() + ": " +

getAverageMark(group.getStudent(i)));

}

}

}

}

import **java.util.ArrayList**;

import **java.util.Random**;

public class **addMarkThread** extends **Thread** {

private **Random** random = new Random();

private **SubjectJournal** journal;

private **ArrayList**<**Group**> groups;

private int TeacherID;

public addMarkThread(**SubjectJournal** journal, **ArrayList**<**Group**> groups, int TeacherID) {

this.journal = journal;

this.groups = groups;

this.TeacherID = TeacherID;

}

@Override

public void run() {

for (**Group** group : groups) {

for (int i = 0; i < group.getSize(); i++) {

int mark = random.nextInt(100) + 1;

journal.addMark(group.getStudent(i), mark);

**System**.*out*.print("Teacher " + TeacherID + ": " +

"Student " + group.getStudent(i).getID() + ": +" + mark + "**\n**");

}

}

}

}