

**Національний Технічний Університет України КПІ**

Факультет Інформатики та Обчислювальної Техніки  
Кафедра інформатики та програмної інженерії

**Лабораторна робота №3**

З дисципліни «Технології паралельних обчислень»

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

**Перевірила:**

Асистент

Дифучина Олександра Юріївна

Оцінка:\_\_\_\_\_\_\_\_\_\_\_**Виконав:**

Студент групи ІТ-92

Бондаренко Д.С.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_

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.

Лістинг програми

Клас та клас-потік, які є потоконебезпечним

package AsyncBank;  
  
public class BankUnsafe implements Bank  
{  
 private final int[] accounts;  
 private long transactions;  
  
 public BankUnsafe(int numberOfAccounts, int initialBalance)  
 {  
 accounts = new int[numberOfAccounts];  
 int i;  
 for (i = 0; i < accounts.length; i++)  
 {  
 accounts[i] = initialBalance;  
 }  
 transactions = 0;  
 }  
  
 public void transfer(int from, int to, int amount) throws InterruptedException  
 {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transactions++;  
 if (transactions % *TESTS* == 0)  
 {  
 test();  
 }  
 }  
  
 public void test()  
 {  
 int sum = 0;  
 for (int account : accounts)  
 {  
 sum += account;  
 }  
 System.*out*.println("Transactions:" + transactions + " Sum: " + sum);  
 }  
  
 public int size()  
 {  
 return accounts.length;  
 }  
}

package AsyncBank;  
  
class TransferThread extends Thread  
{  
 private static final int *REPS* = 1000;  
  
 private final Bank bank;  
 private final int fromAccount;  
 private final int maxAmount;  
  
 public TransferThread(Bank bank, int from, int max)  
 {  
 this.bank = bank;  
 fromAccount = from;  
 maxAmount = max;  
 }  
  
 public void run()  
 {  
 try  
 {  
 while (!*interrupted*())  
 {  
 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);  
 }  
 }  
 }  
 catch (InterruptedException ignored)  
 {  
 }  
 }  
}

Потокобезпечний клас-потік з використанням ReentrantLock

package AsyncBank;  
  
import java.util.concurrent.locks.ReentrantLock;  
  
public class TransferThreadSafe extends Thread  
{  
 private static final int *REPS* = 1000;  
  
 private final Bank bank;  
 private final int fromAccount;  
 private final int maxAmount;  
 private final ReentrantLock lock;  
  
 public TransferThreadSafe(Bank bank, int from, int max, ReentrantLock lock)  
 {  
 this.bank = bank;  
 fromAccount = from;  
 maxAmount = max;  
 this.lock = lock;  
 }  
  
 public void run()  
 {  
 try  
 {  
 while (!*interrupted*())  
 {  
 for (int i = 0; i < *REPS*; i++)  
 {  
 lock.lock();  
 int toAccount = (int) (bank.size() \* Math.*random*());  
 int amount = (int) (maxAmount \* Math.*random*() / *REPS*);  
 bank.transfer(fromAccount, toAccount, amount);  
 lock.unlock();  
 }  
 }  
 }  
 catch (InterruptedException ignored)  
 {  
 }  
 }  
}

Реалізація через Semaphore

package AsyncBank;  
  
import java.util.concurrent.Semaphore;  
  
public class BankSemaphore implements Bank  
{  
 private final int[] accounts;  
 private long transactions;  
  
 private final Semaphore semaphore = new Semaphore(1);  
  
 public BankSemaphore(int numberOfAccounts, int initialBalance)  
 {  
 accounts = new int[numberOfAccounts];  
 int i;  
 for (i = 0; i < accounts.length; i++)  
 {  
 accounts[i] = initialBalance;  
 }  
 transactions = 0;  
 }  
  
 public void transfer(int from, int to, int amount) throws InterruptedException  
 {  
 semaphore.acquire();  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transactions++;  
 if (transactions % *TESTS* == 0)  
 {  
 test();  
 }  
 semaphore.release();  
 }  
  
 public void test()  
 {  
 int sum = 0;  
 for (int account : accounts)  
 {  
 sum += account;  
 }  
 System.*out*.println("Transactions:" + transactions + " Sum: " + sum);  
 }  
  
 public int size()  
 {  
 return accounts.length;  
 }  
}

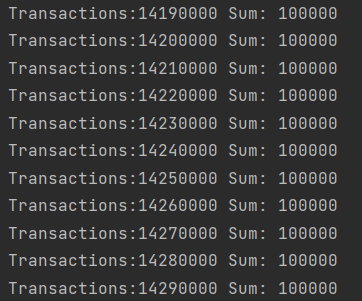
Реалізація через synchronized метод

package AsyncBank;  
  
public class BankSynchronized implements Bank  
{  
 private final int[] accounts;  
 private long transactions;  
  
 public BankSynchronized(int numberOfAccounts, int initialBalance)  
 {  
 accounts = new int[numberOfAccounts];  
 int i;  
 for (i = 0; i < accounts.length; i++)  
 {  
 accounts[i] = initialBalance;  
 }  
 transactions = 0;  
 }  
  
 public synchronized void transfer(int from, int to, int amount) throws InterruptedException  
 {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transactions++;  
 if (transactions % *TESTS* == 0)  
 {  
 test();  
 }  
 }  
  
 public void test()  
 {  
 int sum = 0;  
 for (int account : accounts)  
 {  
 sum += account;  
 }  
 System.*out*.println("Transactions:" + transactions + " Sum: " + sum);  
 }  
  
 public int size()  
 {  
 return accounts.length;  
 }  
}

Запустимо варіант програми із наведеного лістингу. Як ми можемо побачити, через те, що різні потоки намагаються прочитати та записати дані на однакові облікові записи, у нас виникає Race Condition, при цьому ми втрачаємо дані, та чим більше транзакцій ми проведемо, то більше даних втратимо:



Для того, щоб такого не відбувалося, ми маємо використати методи управління потоків. У цій роботі ми розглянемо ReentrantLock, synchronized метод та Semaphore. Synchronized метод є одним із самих простих методів синхронізації потоків. Із ним ми отримаємо гарантію, що код, який знаходиться у synchronized блоці буде безпечно виконаний лише одним потоком. ReentrantLock є більш гнучким ніж synchronized метод. З ReentrantLock отримання та зняття блокування здійснюється користувачем за допомогою методів lock() і unlock(). ReentrantLock надає додаткову функціональність щодо використання синхронізованих методів і операторів, надаючи опцію для спроби отримати блокування (tryLock()), спроби отримати блокування, яке можна перервати (lockInterruptibly(), і спроби отримати блокування, яке може зробити тайм-аут (tryLock(long, TimeUnit)). Semaphore є більш гнучким ніж ReentrantLock. Semaphore забезпечує механізм синхронізації вищого рівня, дозволяючи використовувати налаштовану реалізацію механізму блокування та відновлення з тупикової ситуації. Таким чином, це дає більше контролю розробникам. Використовуючи різні варіанти керування потоками ми отримали однакові результати виконання програми:



Як можна побачити, тепер транзакції працюють правильно і ми отримали очікуваний результат.

Завдання 2

Реалізуйте приклад Producer-Consumer application. 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.

Клас Consumer

package ProducerConsumer;  
  
import java.util.Random;  
  
public record Consumer(Drop drop) implements Runnable {  
  
 public void run() {  
 Random random = new Random();  
 for (int number = drop.take(); number >= 0; number = drop.take()) {  
 System.*out*.println("MESSAGE RECEIVED:" + number);  
 try {  
 Thread.*sleep*(random.nextInt(160));  
 } catch (InterruptedException ignored) {  
 }  
 }  
 }  
}

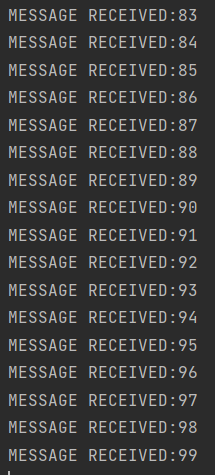
Клас Drop

package ProducerConsumer;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Drop  
{  
 private final List<Integer> buffer;  
 private final int size;  
  
 public Drop(int size)  
 {  
 this.size = size;  
 buffer = new ArrayList<>(size);  
 }  
  
 public synchronized int take()  
 {  
 while (buffer.isEmpty())  
 {  
 try  
 {  
 wait();  
 } catch (InterruptedException ignored)  
 {  
 }  
 }  
 notifyAll();  
 return buffer.remove(0);  
 }  
  
 public synchronized void put(int number)  
 {  
 while (size == buffer.size())  
 {  
 try  
 {  
 wait();  
 } catch (InterruptedException ignored)  
 {  
 }  
 }  
 buffer.add(number);  
 notifyAll();  
 }  
}

Клас Producer

package ProducerConsumer;  
  
import java.util.Random;  
  
public record Producer(Drop drop, int size) implements Runnable {  
  
 public void run() {  
 Random random = new Random();  
  
 for (int i = 0; i < size; i++) {  
 drop.put(i);  
 try {  
 Thread.*sleep*(random.nextInt(160));  
 } catch (InterruptedException ignored) {  
 }  
 }  
 }  
}

Даний код представляє із себе систему, у якому існує «буфер» фіксованого розміру (Drop) у який один код заносить дані (Producer), а інший забирає (Consumer). Завдяки системі синхронізації (synchronized метод) цей код є потокобезпечним. Завдяки методу notifyAll() ми даємо іншим потокам знати, що статус буфера був змінений. Приклад роботи програми:



Після того, як Producer успішно передав усі дані у буфер та перестав генерувати нові, Drop перейшов у статус очікування додавання нових даних. Всі значення були опрацьовані. Програма працює вірно.

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.

Клас Student

package Journal;  
  
public class Student  
{  
 private final String name;  
 private String group;  
  
 public Student(String name)  
 {  
 this.name = name;  
 }  
  
 public void setGroup(String group)  
 {  
 this.group = group;  
 }  
  
 @Override  
 public String toString()  
 {  
 return name + " from group " + group;  
 }  
}

Клас Group

package Journal;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Group  
{  
 private final String name;  
 private final List<Student> students;  
  
 public Group(String name)  
 {  
 this.name = name;  
 students = new ArrayList<>();  
 }  
  
 public void addStudent(Student student)  
 {  
 students.add(student);  
 student.setGroup(name);  
 }  
  
 public String getName()  
 {  
 return name;  
 }  
  
 public List<Student> getStudents()  
 {  
 return students;  
 }  
}

Клас Journal

package Journal;  
  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
  
public class Journal  
{  
 private final List<Group> groups;  
 private final HashMap<Student, Integer> marks;  
  
 public Journal()  
 {  
 groups = new ArrayList<>();  
 marks = new HashMap<>();  
 }  
  
 public void addGroup(Group group)  
 {  
 groups.add(group);  
 }  
  
 public List<Group> getGroups()  
 {  
 return groups;  
 }  
  
 public List<Student> getStudents()  
 {  
 var students = new ArrayList<Student>();  
 var groups = getGroups();  
 for (Group group: groups) {  
 students.addAll(group.getStudents());  
 }  
 return students;  
 }  
  
 public synchronized void setMark(Student student, int mark)  
 {  
 marks.put(student, mark);  
 }  
  
 @Override  
 public String toString()  
 {  
 StringBuilder builder = new StringBuilder();  
 for (Group group : groups)  
 {  
 builder.append(group.getName());  
 builder.append(":");  
 builder.append("\n");  
 for (Student student : group.getStudents())  
 {  
 builder.append(student.toString());  
 builder.append(" - ");  
 if (marks.containsKey(student))  
 {  
 builder.append(marks.get(student));  
 }  
 builder.append("\n");  
 }  
 builder.append("\n");  
 }  
 return builder.toString();  
 }  
}

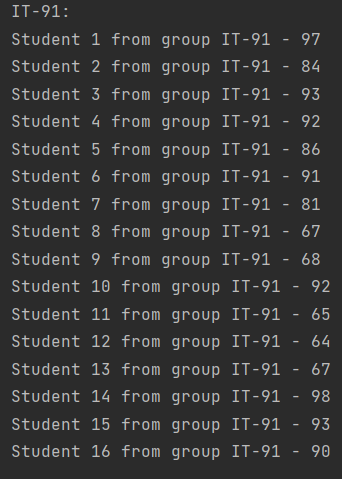
Клас JournalThread

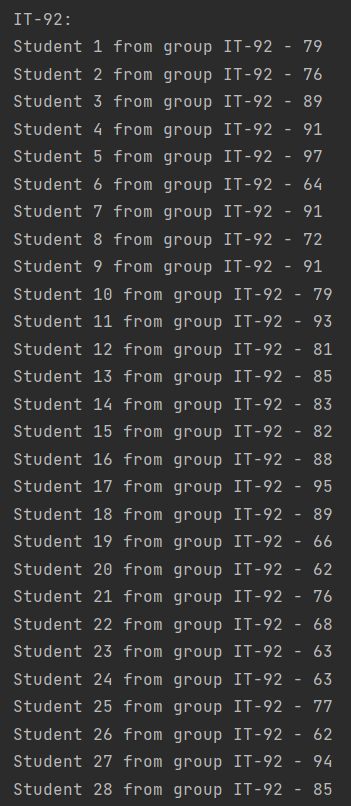
package Journal;  
  
import java.util.List;  
import java.util.Random;  
  
public class JournalThread extends Thread  
{  
 private static final int *MIN\_MARK* = 60;  
 private static final int *MAX\_MARK* = 100;  
  
 private final Journal journal;  
 private final int weeks;  
 private final Random random;  
  
 public JournalThread(Journal journal, int weeks)  
 {  
 this.journal = journal;  
 this.weeks = weeks;  
 random = new Random();  
 }  
  
 @Override  
 public void run()  
 {  
 for (int i = 0; i < weeks; i++)  
 {  
 var count = random.nextInt(countStudents());  
 for (int j = 0; j < count; j++)  
 {  
 int mark = random.nextInt(*MAX\_MARK* - *MIN\_MARK*) + *MIN\_MARK*;  
 journal.setMark(getRandomStudent(), mark);  
 }  
 }  
 }  
  
 private int countStudents()  
 {  
 return journal.getStudents().size();  
 }  
  
 private Student getRandomStudent()  
 {  
 List<Student> students = journal.getStudents();  
 return students.get(random.nextInt(students.size()));  
 }  
}

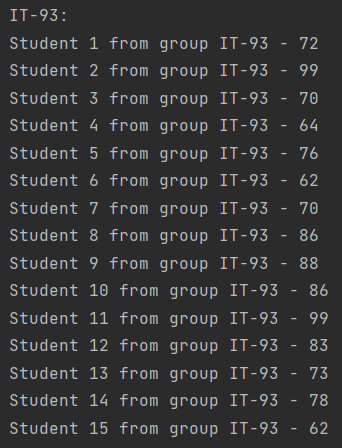
Клас Main

package Journal;  
  
import java.util.Random;  
  
public class Main  
{  
 private static final int *GROUPS* = 3;  
 private static final int *MIN\_STUDENT\_NUMBER* = 15;  
 private static final int *MAX\_STUDENT\_NUMBER* = 30;  
 private static final int *THREADS* = 4;  
 private static final int *WEEKS* = 7;  
  
 private static final Random *random* = new Random();  
  
 public static void main(String[] args) throws InterruptedException  
 {  
 Journal journal = *createJournal*();  
 Thread[] threads = new Thread[*THREADS*];  
 for (int i = 0; i < *THREADS*; i++)  
 {  
 Thread thread = new JournalThread(journal, *WEEKS*);  
 threads[i] = thread;  
 thread.start();  
 }  
 for (int i = 0; i < *THREADS*; i++)  
 {  
 threads[i].join();  
 }  
 System.*out*.println(journal);  
 }  
  
 private static Journal createJournal()  
 {  
 var journal = new Journal();  
 for (int i = 0; i < *GROUPS*; i++)  
 {  
 journal.addGroup(*createGroup*(i + 1));  
 }  
 return journal;  
 }  
  
 private static Group createGroup(int groupNumber)  
 {  
 Group group = new Group("IT-9" + groupNumber);  
 var count = *random*.nextInt(*MAX\_STUDENT\_NUMBER* - *MIN\_STUDENT\_NUMBER*) + *MIN\_STUDENT\_NUMBER*;  
 for (int i = 0; i < count; i++)  
 {  
 group.addStudent(*createStudent*(i + 1));  
 }  
 return group;  
 }  
  
 private static Student createStudent(int i)  
 {  
 return new Student("Student " + i);  
 }  
}

Кількість потоків представляють із себе лектора та його асистентів. Потокобезпечніть даних нам забезпечує synchronized метод. Студенти та їх оцінки зберігаються у вигляді словника HashMap. Приклад роботи програми:







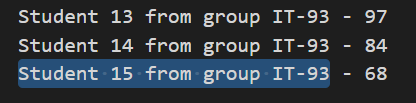
Synchronized метод гарантує нам, що оновлення оцінок завжди буде вірним і ми побачимо саме останнє оновлення кожної оцінки для студента. Наприклад, передостаннє оновлення оцінки студента:



Останнє оновлення студента:



Результат:



Тобто, ми отримали очікуваний та вірний результат.

Висновок

У даній лабораторній роботі ми дослідили різні механізми синхронізації потоків. Такі як синхронізовані методи, локери та спеціальні типи. Як ми можемо побачити, механізмів управління потоками є доволі багато, і який метод ми будемо обирати залежить від задачі та потрібної гнучкості керування. Дані методи використовуються, коли ми бачимо, що декілька потоків будуть одночасно працювати із загальнодоступними даними на читання та оновлення/додавання і при цьому у нас виникає Race condition. Для всіх задач, які ми опрацювали у цій роботі, нам вистачило синхронізованого методу, який є одним із самих легких способів синхронізації потоків. Якщо ж нам треба щось більше, то можна використати Reentrant lock, Semaphore.

Повний лістинг програми можна знайти на GitHub: https://github.com/Agupnik/PDC/tree/master/PDCLab3/PDCLab3