НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра технічної кібернетики

Звіти до комп’ютерних практикумів з кредитного модуля

“Backend”

**Виконав**

**Студенти групи ІТ-02**

**Терешкович М.О.**

**Перевірив:**

Київ – 2022

**Комп‘ютерний практикум No 3**

**Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності**

ЗАВДАННЯ

**Text

Description automatically generated**

**ВИКОНАННЯ**

Повний код лаби тут та у лістингу до звіту: https://github.com/mcmcmax437/Parallel-Computing-Technologies-Labs/tree/master/Laba3\_TPO

Отже 1 завданя звучить так:

«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**.»

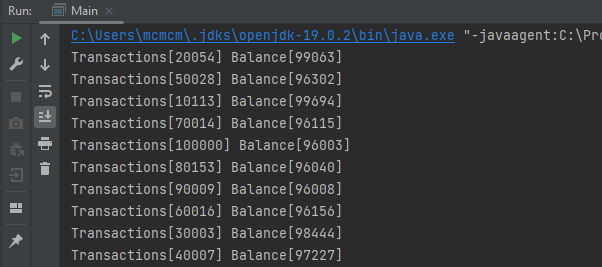
Програма складається з двох файлів: ***Main*** і ***Bank.***

Файл Main:

package T1;  
  
public class Main {  
 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*);  
 int i;  
 for (i = 0; i < *NACCOUNTS*; i++) {  
 TransferThread thread = new TransferThread(b, i, *INITIAL\_BALANCE*);  
 thread.setPriority(Thread.*NORM\_PRIORITY* + i % 2);  
 thread.start();  
 }  
 }  
}  
  
class TransferThread extends Thread {  
 private final Bank bank;  
 private final int fromAccount;  
 private final int maxAmount;  
 private static final int *REPS* = 1000;  
  
 public TransferThread(Bank b, int from, int max) {  
 bank = b;  
 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.BasicTransfer(fromAccount, toAccount, amount);  
// bank.syncTransfer(fromAccount, toAccount, amount);  
 // bank.syncBlockTransfer(fromAccount, toAccount, amount);  
 bank.LockTransfer(fromAccount, toAccount, amount);  
 }  
 }  
 } catch (InterruptedException ignored) {  
 }  
 }  
}

Основна задача була створити 3 різних варіанти управління потоками. При базовому методі пересилання грошей через стан гонки або Race Condition гроші поступово почнуть зникати з рахунків:

public void BasicTransfer(int from, int to, int amount) throws InterruptedException {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
  
 if (transaction % *NTEST* == 0) {  
 test();  
 }  
}



Ось так з самої першої транзакції при базовому трансфері, який не синхронізований, зникають гроші з рахунку.

Зараз перейдемо до 1 метода управління потоками для коретної роботи програми:

public synchronized void syncTransfer(int from, int to, int amount) throws InterruptedException {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
 if (transaction % *NTEST* == 0){  
 test();  
 }  
 }

Цей метод (**syncTransfer**) є синхронізованим на рівні методу, що означає, що лише один потік може виконувати його в один момент часу. Вхід в цей метод може бути заблокований, якщо інший потік вже зайняв блокування на його виконання.

Цей метод приймає три параметри - from, to та amount, які вказують на рахунки, між якими потрібно здійснити транзакцію та її суму.

У тілі методу спочатку зменшується баланс рахунку з індексом from на величину amount, а потім збільшується баланс рахунку з індексом to на ту саму величину amount. Після цього збільшується лічильник транзакцій transaction. Якщо кількість транзакцій ділиться на **NTEST** (константу, визначену на рівні класу Bank), то викликається метод test() для перевірки стану банку.

Зазвичай синхронізація на рівні методу дозволяє досягти потрібного рівня безпеки, але у цьому конкретному випадку можуть виникнути проблеми з продуктивністю, оскільки блокування буде здійснюватись на рівні методу. Також варто зазначити, що в методі не передбачена обробка винятків, яка може призвести до неочікуваної поведінки програми.

A screenshot of a computer

Description automatically generated

Як бачимо програма працює коректно.

Тепер перейдемо до 2 методу управління потоками для коректної роботи програми:

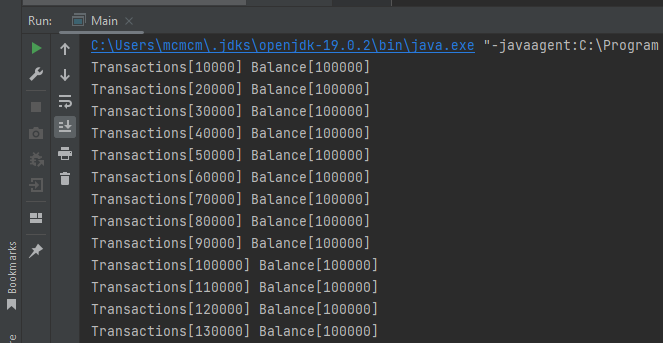
public void syncBlockTransfer(int from, int to, int amount) throws InterruptedException {  
 synchronized (sync) {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
 if (transaction % *NTEST* == 0) {  
 test();  
 }  
 }  
}

Цей метод (**syncBlockTransfer**) також забезпечує синхронізацію доступу до критичної секції, але використовує блок синхронізації замість синхронізованого методу.

У тілі методу спочатку виконується блок синхронізації на об'єкті sync, який є об'єктом класу **Bank**, в якому знаходиться цей метод. Всі потоки, що викликають цей метод, будуть блокуватись на об'єкті sync до того часу, поки інший потік не звільнить його.

Далі метод працює аналогічно до попереднього методу (**syncTransfer**): зменшує баланс рахунку from на величину amount, збільшує баланс рахунку to на ту саму величину amount, збільшує лічильник транзакцій transaction і, якщо кількість транзакцій ділиться на NTEST, викликає метод test().

Використання блоку синхронізації в методі **syncBlockTransfer** дозволяє зменшити витрати на блокування, оскільки потік блокується лише на час виконання критичної секції, а не на весь час виконання методу, як у випадку з синхронізованим методом. Однак, як і у випадку з попереднім методом, можуть виникнути проблеми з продуктивністю, якщо критична секція займає багато часу, що може призвести до блокування потоків на довгий час.



Як бачимо, програма працює коректно з даним методом.

Тепер перейдемо до 3 методу управління потоками для коректної роботи програми:

public void LockTransfer(int from, int to, int amount) throws InterruptedException {  
 Lock.lock();  
 try {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
  
 if (transaction % *NTEST* == 0) {  
 test();  
 }  
 } finally {  
 Lock.unlock();  
 }  
 }

Тут було використано **Reentrantlock** для реалізації управління потоками.  
**Reentrantlock** - це клас в мові програмування Java, який використовується для створення блокувань (локів) та забезпечення потокобезпеки у багатопотокових програмах.

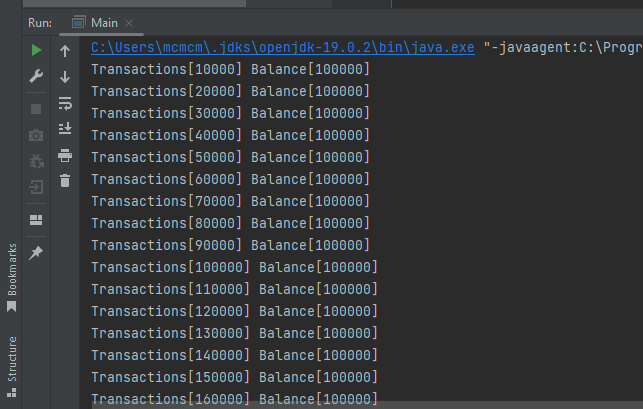
Цей клас дозволяє програмісту вручну керувати блокуваннями та має багато функцій, які дозволяють точно настроювати блокування. Він може бути використаний як альтернатива ключовому слову synchronized в мові Java для керування доступом до критичних секцій коду у багатопотокових програмах.

Особливістю **ReentrantLock** є те, що він може бути повторно використаний тим же потоком, що вже має блокування. Це означає, що поток може входити у критичну секцію коду, захоплену **ReentrantLock**, декілька разів перед тим, як звільнити блокування. Це дає можливість уникнути блокування потоку, який вже має блокування на критичну секцію коду, що може викликати непередбачувану поведінку програми. Також код буде заблокований до моменту, як об’явлено unlock. Навіть якщо локер захоплює декілька методів.

Цей код містить метод **LockTransfer**, який є одним з методів класу **Bank**. Цей метод передає дві змінні from та to, які відповідають номерам рахунків, з яких будуть здійснені перекази коштів, і змінну amount, яка відповідає сумі коштів, що будуть переказані. Цей метод використовує **ReentrantLock** для забезпечення потокобезпеки.

Коли потік викликає цей метод, він блокує лок у монітора **Lock**, який забезпечує доступ лише для одного потоку. Після отримання локу, потік віднімає вказану суму з рахунку from і додає її на рахунок to. Потім метод збільшує значення лічильника transaction на одиницю.

Якщо лічильник transaction ділиться націло на константу NTEST, то метод викликає test(), який перевіряє правильність балансів на всіх рахунках і виводить повідомлення про будь-які помилки. Після виконання операцій метод повертає лок, щоб інші потоки могли отримати доступ до нього.



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**.»

Основний код програми був представлений і розписаний на сайті, тому вдаватися в базові деталі я не буду. Опиши в цілому тільки.

Ця програма демонструє взаємодію між виробником і споживачем через спільну змінну типу **Drop**. Клас **Drop** містить два методи - **put()** та **take()**, які синхронізовані, тобто виконуються в потоці безпечним для потоків способом.

Клас **Producer** генерує інформацію та передає її в змінну типу **Drop**, використовуючи метод **put().** Клас **Consumer** витягує інформацію з змінної типу Drop, використовуючи метод **take()**. Класи **Producer** та **Consumer** запускаються в окремих потоках, щоб вони могли працювати паралельно.

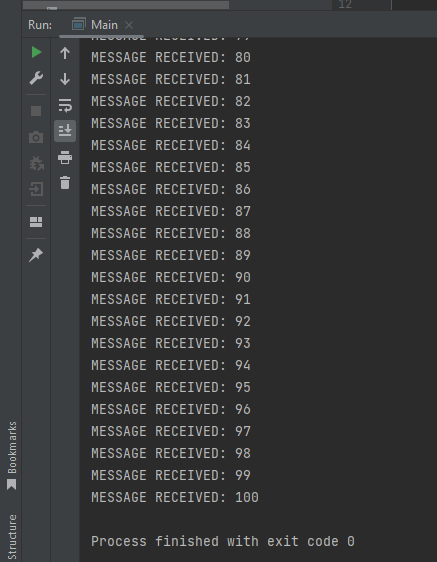
Клас **Main** містить точку входу в програму, в якій створюються об’єкти **Producer** та **Consumer** та запускаються їх потоки.

Після запуску програми, виробник генерує інформацію та передає її у змінну **Drop**, яку отримує споживач. Якщо змінна **Drop** використана споживачем, виробник чекає на використання змінної споживачем. Коли інформація передана споживачу, виробник знову може використовувати змінну **Drop** для передачі інформації, і процес продовжується далі до тих пір, поки виробник не передасть повідомлення 0, що означає кінець передачі інформації.

public synchronized int take() {  
 while (empty) {  
 try {  
 wait();  
 } catch (InterruptedException ignored) {  
 }  
 }  
 empty = true;  
 notifyAll();  
 return message;  
 }  
  
 public synchronized void put(int message) {  
 while (!empty) {  
 try {  
 wait();  
 } catch (InterruptedException ignored) {  
 }  
 }  
 empty = false;  
 this.message = message;  
 notifyAll();  
 }  
}

Метод **take()** забезпечує безпечне вилучення повідомлення з об'єкта **Drop**. Поки змінна empty є true, тобто поки об'єкт порожній, потік викликає wait() і звільняє монітор. Це означає, що потік блокується і чекає, доки інший потік не викличе notify() або notifyAll() для того, щоб просигналізувати інші потоки, що об'єкт може бути готовим для використання. Після того, як об'єкт був взятий, змінна empty встановлюється на true і повідомлення повертається з методу.

Метод **put()** забезпечує безпечне зберігання повідомлення в об'єкті **Drop**. Поки змінна empty є false, тобто поки об'єкт не порожній, потік викликає wait() і звільняє монітор. Це означає, що потік блокується і чекає, доки інший потік не викличе notify() або notifyAll() для того, щоб просигналізувати інші потоки, що об'єкт може бути готовим для використання. Після того, як повідомлення було збережене в об'єкті, змінна empty встановлюється на false і викликається notifyAll() для того, щоб просигналізувати інші потоки про те, що об'єкт може бути готовим для використання.



Як результат програма працює коректно.

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**.»

Основний код програми:

package T3;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Random;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class ElectronicJournal {  
  
 private static final int *NUM\_GROUPS* = 3;  
 private static final int *NUM\_STUDENTS\_PER\_GROUP* = 20;  
 private static final int *NUM\_LECTURERS* = 4;  
 private static final int *MAX\_GRADE* = 100;  
  
 private final Lock[] groupLocks;  
 private final int[][] journal;  
  
 public ElectronicJournal() {  
 groupLocks = new Lock[*NUM\_GROUPS*];  
 Arrays.*setAll*(groupLocks, i -> new ReentrantLock());  
 journal = new int[*NUM\_GROUPS*][*NUM\_STUDENTS\_PER\_GROUP*];  
 }  
  
 public void run() {  
 ArrayList<Thread> threads = new ArrayList<>();  
  
 for (int i = 0; i < *NUM\_LECTURERS*; i++) {  
 int i1 = i;  
 Thread thread = new Thread(() -> giveGrades(i1));  
 threads.add(thread);  
 thread.start();  
 }  
  
 for (Thread thread : threads) {  
 try {  
 thread.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 for (int i = 0; i < *NUM\_GROUPS*; i++) {  
 System.*out*.print("Group" + (i+1) + " -- ");  
 for (int j = 0; j < *NUM\_STUDENTS\_PER\_GROUP*; j++) {  
 System.*out*.print(journal[i][j] + " ");  
 }  
 System.*out*.println();  
 }  
 }  
  
 private void giveGrades(int lecturerId) {  
 Random random = new Random();  
  
 while (true) {  
  
 boolean allGraded = true;  
 for (int[] group : journal) {  
 for (int grade : group) {  
 if (grade == 0) {  
 allGraded = false;  
 break;  
 }  
 }  
 if (!allGraded) {  
 break;  
 }  
 }  
 if (allGraded) {  
 break;  
 }  
  
  
 int groupId = random.nextInt(*NUM\_GROUPS*);  
 Lock groupLock = groupLocks[groupId];  
  
  
 if (groupLock.tryLock()) {  
 try {  
  
 if (Arrays.*stream*(journal[groupId]).allMatch(grade -> grade > 0)) {  
 continue;  
 }  
  
 int studentId;  
 do {  
 studentId = random.nextInt(*NUM\_STUDENTS\_PER\_GROUP*);  
 } while (journal[groupId][studentId] > 0);  
  
 int grade = random.nextInt(*MAX\_GRADE*) + 1;  
 journal[groupId][studentId] = grade;  
 System.*out*.println("Lec" + (lecturerId + 1) + " give " + grade +  
 " Student " + (studentId + 1) + " Group" + (groupId + 1));  
  
 } finally {  
 groupLock.unlock();  
 }  
 }  
 }  
 }  
  
  
  
  
 public static void main(String[] args) {  
 ElectronicJournal journal = new ElectronicJournal();  
 journal.run();  
 }  
}

Даний код створює електронний журнал, до якого 4 викладачі додають оцінки для студентів 3 груп. Використовується мультипотоківсть з використанням замків ReentrantLock. Про ReentrantLock вже пояснювалося в Задачі 1.

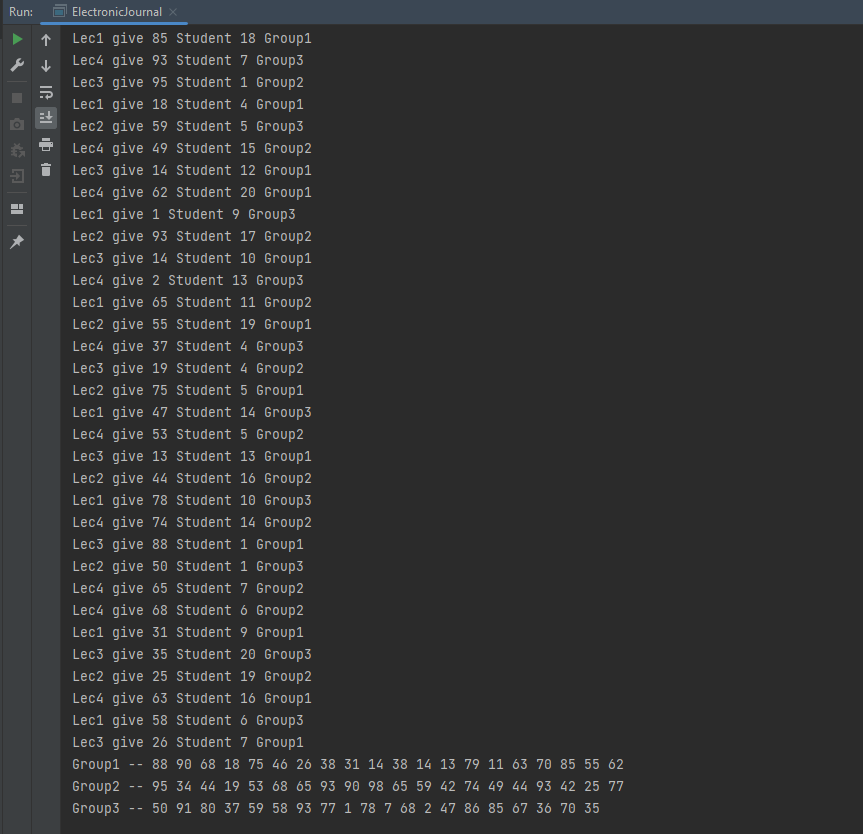
У методі **giveGrades** відбувається надання оцінки для студента. Кожен викладач випадковим чином вибирає групу. Якщо група не в лоці то йдемо далі, якщо ж вона присутня то чекаємо поки звільниться.

Lock groupLock = groupLocks[groupId];

Потім рандомно обираємо студента з цієї групи*. Лок відбувається за групою, бо студенти та сама група обирається рандомно – якщо цього не зробити існує ймовірність того, що два лектора отримують одного і того самого студента з однієї і тої самої групи.* Якщо обраний студент ще не отримав оцінку, то викладач додає його до журналу. Якщо студент вже має оцінку, то викладач продовжує випадково обирати інших студентів.

Для уникнення одночасної зміни журналу різними викладачами використовується механізм блокування замками. Кожна група має свій замок **groupLocks**, який блокує доступ до журналу для інших викладачів, які хочуть додати оцінку. Якщо замок групи є заблокованим, викладач продовжує випадково обирати іншу групу та студента.

У методі **run** запускаються потоки для кожного викладача, які виконують метод **giveGrades**. Потім чекаємо, поки всі потоки завершаться, і виводимо журнал з оцінками для кожної групи.



Як ми бачимо все працює відмінно, всі отримали оцінки. Жоден зі студентів з жодної групи не отримав оцінку двічі – отже все правильно.

**ВИСНОВОК**

***\*Я НЕ ПЛАНУЮ ЗАХИЩАТИ РОБОТУ, ОЦІНІТЬ БУДЬ ЛАСКА ДАННИЙ ЗВІТ\****

Методи управління потоками в Java дозволяють програмістам ефективно керувати виконанням багатопоточних програм. Найбільш поширеними методами є синхронізація, взаємодія потоків за допомогою wait/notify, взаємовиключення за допомогою блокування на основі Lock і ReentrantLock.

Синхронізація дозволяє забезпечити взаємовиключення між потоками під час доступу до спільних ресурсів. Взаємодія потоків за допомогою wait/notify дозволяє потокам чекати на сповіщення про зміну стану об'єкту або сигналізувати інші потоки про такі зміни. Lock і ReentrantLock забезпечують більш гнучкий механізм блокування, який дозволяє програмістам контролювати, які потоки мають доступ до ресурсів.

Використання методів управління потоками може бути складним, оскільки неправильне використання може призвести до дедлоків, голодування потоків і непередбачуваної поведінки програми. Тому важливо ретельно планувати та тестувати багатопоточний код для забезпечення його коректності.

**Лістинг**

Тask1:

Main.java

package T1;  
  
public class Main {  
 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*);  
 int i;  
 for (i = 0; i < *NACCOUNTS*; i++) {  
 TransferThread thread = new TransferThread(b, i, *INITIAL\_BALANCE*);  
 thread.setPriority(Thread.*NORM\_PRIORITY* + i % 2);  
 thread.start();  
 }  
 }  
}  
  
class TransferThread extends Thread {  
 private final Bank bank;  
 private final int fromAccount;  
 private final int maxAmount;  
 private static final int *REPS* = 1000;  
  
 public TransferThread(Bank b, int from, int max) {  
 bank = b;  
 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.BasicTransfer(fromAccount, toAccount, amount);  
// bank.syncTransfer(fromAccount, toAccount, amount);  
 // bank.syncBlockTransfer(fromAccount, toAccount, amount);  
 bank.LockTransfer(fromAccount, toAccount, amount);  
 }  
 }  
 } catch (InterruptedException ignored) {  
 }  
 }  
}

Bank.java

package T1;  
  
import java.util.concurrent.locks.ReentrantLock;  
  
public class Bank {  
 public final Object sync = new Object();  
 public static final int *NTEST* = 10000;  
 private final int[] accounts;  
 private long transaction = 0;  
 private final ReentrantLock Lock = new ReentrantLock();  
  
 public Bank(int n, int initialBalance) {  
 accounts = new int[n];  
 int i;  
 for (i = 0; i < accounts.length; i++) accounts[i] = initialBalance;  
 transaction = 0;  
 }  
  
 public void BasicTransfer(int from, int to, int amount) throws InterruptedException {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
  
 if (transaction % *NTEST* == 0) {  
 test();  
 }  
 }  
  
public synchronized void syncTransfer(int from, int to, int amount) throws InterruptedException {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
 if (transaction % *NTEST* == 0){  
 test();  
 }  
 }  
  
 public void syncBlockTransfer(int from, int to, int amount) throws InterruptedException {  
 synchronized (sync) {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
 if (transaction % *NTEST* == 0) {  
 test();  
 }  
 }  
 }  
  
 public void LockTransfer(int from, int to, int amount) throws InterruptedException {  
 Lock.lock();  
 try {  
 accounts[from] -= amount;  
 accounts[to] += amount;  
 transaction++;  
  
 if (transaction % *NTEST* == 0) {  
 test();  
 }  
 } finally {  
 Lock.unlock();  
 }  
 }  
  
 public void test() {  
 int sum = 0;  
 for (int account : accounts) sum += account;  
 System.*out*.println("Transactions[" + transaction +"]" + " Balance[" + sum + "]");  
 }  
  
 public int size() {  
 return accounts.length;  
 }  
}

Task2:

Main.java

package T2;  
  
public class Main {  
 public static final int *ITER* = 100;  
 public static void main(String[] args) {  
  
 Drop drop = new Drop();  
 (new Thread(new Producer(drop))).start();  
 (new Thread(new Consumer(drop))).start();  
 }  
}

Drop.java

package T2;  
  
public class Drop {  
 private int message;  
 private boolean empty = true;  
  
 public synchronized int take() {  
 while (empty) {  
 try {  
 wait();  
 } catch (InterruptedException ignored) {  
 }  
 }  
 empty = true;  
 notifyAll();  
 return message;  
 }  
  
 public synchronized void put(int message) {  
 while (!empty) {  
 try {  
 wait();  
 } catch (InterruptedException ignored) {  
 }  
 }  
 empty = false;  
 this.message = message;  
 notifyAll();  
 }  
}

Consumer.java

package T2;  
  
import java.util.Random;  
  
public class Consumer implements Runnable {  
 private final Drop drop;  
  
 public Consumer(Drop drop) {  
 this.drop = drop;  
 }  
  
 public void run() {  
 Random random = new Random();  
 for (int message = drop.take(); message != 0; message = drop.take()) {  
 System.*out*.format("MESSAGE RECEIVED: %d%n", message);  
 try {  
 Thread.*sleep*(random.nextInt(50));  
 } catch (InterruptedException ignored) {  
 }  
 }  
 }  
}

Producer.java

package T2;  
import java.util.Random;  
import static T2.Main.*ITER*;  
  
public class Producer implements Runnable {  
 private final Drop drop;  
  
 public Producer(Drop drop) {  
 this.drop = drop;  
 }  
  
 public void run() {  
 int[] importantInfo = new int[*ITER*];  
 for (int i = 0; i < importantInfo.length; i++){  
 importantInfo[i] = i + 1;  
 }  
 Random random = new Random();  
  
 for (int i : importantInfo) {  
 drop.put(i);  
 try {  
 Thread.*sleep*(random.nextInt(50));  
 } catch (InterruptedException ignored) {  
 }  
 }  
 drop.put(0);  
 }  
}

Task3:

package T3;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Random;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class ElectronicJournal {  
  
 private static final int *NUM\_GROUPS* = 3;  
 private static final int *NUM\_STUDENTS\_PER\_GROUP* = 20;  
 private static final int *NUM\_LECTURERS* = 4;  
 private static final int *MAX\_GRADE* = 100;  
  
 private final Lock[] groupLocks;  
 private final int[][] journal;  
  
 public ElectronicJournal() {  
 groupLocks = new Lock[*NUM\_GROUPS*];  
 Arrays.*setAll*(groupLocks, i -> new ReentrantLock());  
 journal = new int[*NUM\_GROUPS*][*NUM\_STUDENTS\_PER\_GROUP*];  
 }  
  
 public void run() {  
 ArrayList<Thread> threads = new ArrayList<>();  
  
 for (int i = 0; i < *NUM\_LECTURERS*; i++) {  
 int i1 = i;  
 Thread thread = new Thread(() -> giveGrades(i1));  
 threads.add(thread);  
 thread.start();  
 }  
  
 for (Thread thread : threads) {  
 try {  
 thread.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 for (int i = 0; i < *NUM\_GROUPS*; i++) {  
 System.*out*.print("Group" + (i+1) + " -- ");  
 for (int j = 0; j < *NUM\_STUDENTS\_PER\_GROUP*; j++) {  
 System.*out*.print(journal[i][j] + " ");  
 }  
 System.*out*.println();  
 }  
 }  
  
 private void giveGrades(int lecturerId) {  
 Random random = new Random();  
  
 while (true) {  
  
 boolean allGraded = true;  
 for (int[] group : journal) {  
 for (int grade : group) {  
 if (grade == 0) {  
 allGraded = false;  
 break;  
 }  
 }  
 if (!allGraded) {  
 break;  
 }  
 }  
 if (allGraded) {  
 break;  
 }  
  
  
 int groupId = random.nextInt(*NUM\_GROUPS*);  
 Lock groupLock = groupLocks[groupId];  
  
  
 if (groupLock.tryLock()) {  
 try {  
  
 if (Arrays.*stream*(journal[groupId]).allMatch(grade -> grade > 0)) {  
 continue;  
 }  
  
 int studentId;  
 do {  
 studentId = random.nextInt(*NUM\_STUDENTS\_PER\_GROUP*);  
 } while (journal[groupId][studentId] > 0);  
  
 int grade = random.nextInt(*MAX\_GRADE*) + 1;  
 journal[groupId][studentId] = grade;  
// if(lecturerId+1 == 1) {  
// System.out.println("Lec " + (lecturerId + 1) + " give " + grade +  
// " Student " + (studentId + 1) + " Group " + (groupId + 1));  
// }  
 System.*out*.println("Lec" + (lecturerId + 1) + " give " + grade +  
 " Student " + (studentId + 1) + " Group" + (groupId + 1));  
  
 } finally {  
 groupLock.unlock();  
 }  
 }  
 }  
 }  
  
  
  
  
 public static void main(String[] args) {  
 ElectronicJournal journal = new ElectronicJournal();  
 journal.run();  
 }  
}