

**Національний Технічний Університет України КПІ**

Факультет Інформатики та Обчислювальної Техніки  
Кафедра інформатики та програмної інженерії

**Лабораторна робота №1**

З дисципліни «Технології паралельних обчислень»

Розробка потоків та дослідження пріоритету запуску потоків

**Перевірила:**

Асистент

Дифучина Олександра Юріївна

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

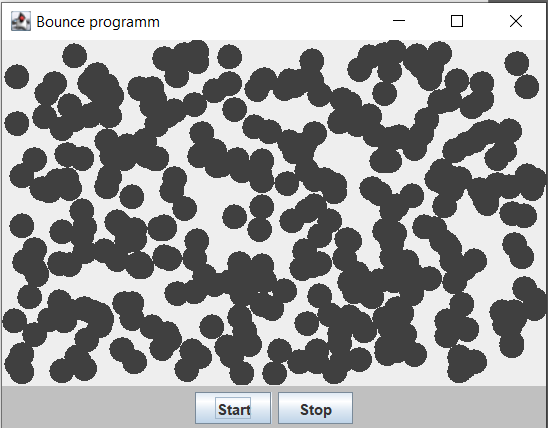
Студент групи ІТ-92

Бондаренко Д.С.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_

Завдання 1

Реалізуйте програму імітації руху більярдних кульок, в якій рух кожної кульки відтворюється в окремому потоці. Спостерігайте роботу програми при збільшенні кількості кульок. Поясніть результати спостереження. Опишіть переваги потокової архітектури програм.



При збільшенні кількості кульок програма починає працювати повільніше. Це пояснюється тим, що кількість потоків, які можуть бути виділені для виконання роботи програми, обмежено потужністю процесора, а саме кількістю ядер та кількістю потоків.

Перевагою потокової архітектури програми є покращення продуктивності самої програми, тобто багатопоточність відкриває можливість одночасно виконувати багато різних обчислень на процесорі. У нашому випадку, кожна кулька рухається незалежно одна від одної, тобто те, як буде переміщатися кулька обраховується у кожному потоці окремо, а тому ми можемо бачити рух усіх створених кулько одночасно. Якщо б ми опрацьовували рух кульки лише у головному потоці самої програми (тобто не створювали б на кожну кульку окремий потік), то ми б могли спостерігати рух лише однієї кульки.

Лістинг програми розміщений за посиланням: <https://github.com/Agupnik/PDC/tree/master/PDCLab1/PDCLab1_Start>

Завдання 2

Модифікуйте програму так, щоб при потраплянні в «лузу» кульки зникали, а відповідний потік завершував свою роботу. Кількість кульок, яка потрапила в «лузу», має динамічно відображатись у текстовому полі інтерфейсу програми.

Потік завершує роботу, якщо ми виходимо із методу run() який ми перевизначаємо у класі нашого потоку:

@Override  
public void run(){  
 try{  
 while(b.isAlive()){  
 b.move();  
 b.checkPocketIntersections();  
  
 System.*out*.println("Thread name = "  
 + Thread.*currentThread*().getName());  
 Thread.*sleep*(5);  
  
 }  
 } catch(InterruptedException ex){  
 System.*out*.println("Thread finished! Thread name = "  
 + Thread.*currentThread*().getName());  
 }  
}

Ми додали флаг до кульки isAlive. Якщо відбувається колізія, то цей флаг приймає значення True:

public void makeDead()  
{  
 alive = false;  
 canvas.ballsInPocket++;  
}  
  
public void checkPocketIntersections()  
{  
 boolean isInteracted = false;  
 for(int i=0; i<xPocketsArray.length; i++){  
 int pocketX = xPocketsArray[i];  
 int pocketY = yPocketsArray[i];  
 double sqrtDistance = Math.*pow*(x - pocketX, 2) + Math.*pow*(y - pocketY, 2);  
 if (sqrtDistance <= *XSIZE* \* *XSIZE*) {  
 isInteracted = true;  
 break;  
 }  
 }  
 if (isInteracted)  
 {  
 makeDead();  
 }  
}

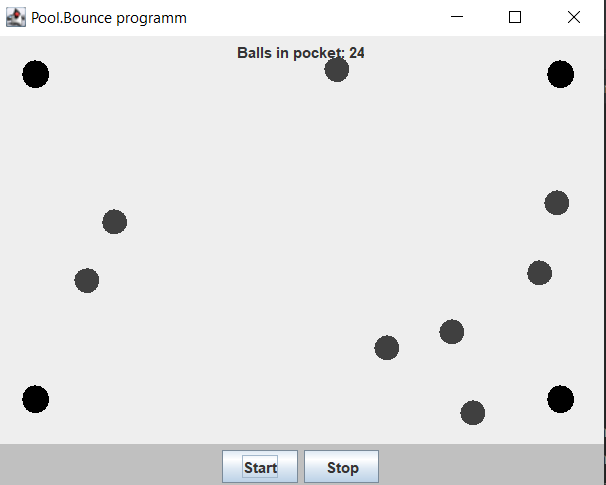
Для того, щоб перестати малювати кульку, ми редагували метод paintComponent() у BallCanvas:

@Override  
public void paintComponent(Graphics g){  
 super.paintComponent(g);  
 Graphics2D g2 = (Graphics2D)g;  
 balls.removeIf(b -> !b.isAlive());  
 for(int i=0; i<balls.size(); i++){  
 Ball b = balls.get(i);  
 b.draw(g2);  
 }  
 for(int i=0; i<pockets.size(); i++){  
 Pocket p = pockets.get(i);  
 p.draw(g2);  
 }  
 label.setText("Balls in pocket: " + ballsInPocket);  
 //repaint element to set pocket count  
 repaint();  
}

Також, ми створили та додали JLabel, для виводу кількості кульок у лузах:

public ArrayList<Ball> balls = new ArrayList<>();  
private final ArrayList<Pocket> pockets;  
private final JLabel label = new JLabel();  
public int ballsInPocket;  
  
public void add(Ball b){  
 this.balls.add(b);  
}  
  
public BallCanvas(ArrayList<Pocket> pockets){  
 label.setText("Balls in pockets: " + ballsInPocket);  
 add(label, BorderLayout.*SOUTH*);  
 this.pockets = pockets;  
}

Результат:



Повний лістинг програми наведений за посиланням:

<https://github.com/Agupnik/PDC/tree/master/PDCLab1/PDCLab1>

Завдання 3

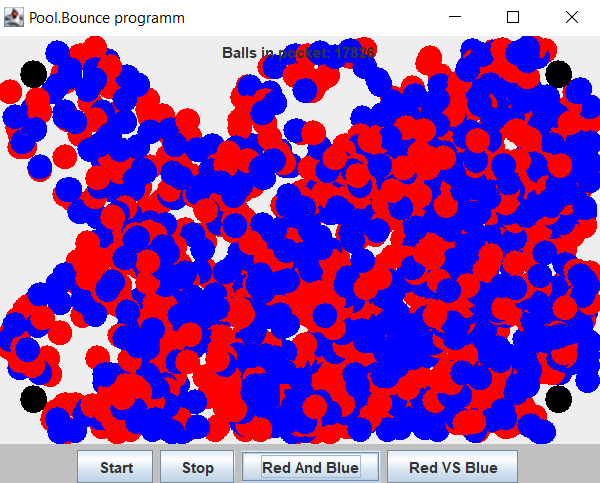
Виконайте дослідження параметру priority потоку. Для цього модифікуйте програму «Більярдна кулька» так, щоб кульки червоного кольору створювались з вищим пріоритетом потоку, в якому вони виконують рух, ніж кульки синього кольору. Спостерігайте рух червоних та синіх кульок при збільшенні загальної кількості кульок. Проведіть такий експеримент. Створіть багато кульок синього кольору (з низьким пріоритетом) і одну червоного кольору, які починають рух в одному й тому ж самому місці більярдного стола, в одному й тому ж самому напрямку та з однаковою швидкістю. Спостерігайте рух кульки з більшим пріоритетом. Повторіть експеримент кілька разів, значно збільшуючи кожного разу кількість кульок синього кольору. Зробіть висновки про вплив пріоритету потоку на його роботу в залежності від загальної кількості потоків.

Додаймо ще один конструктор потоку кульок для того, щоб встановлювати пріоритет кульки в залежності від переданих параметрів та встановлених параметрів кульки:

public BallThread(Ball ball, Thread thread){  
 b = ball;  
 this.thread = thread;  
 setPriority(getBallPriority(ball));  
}  
  
public BallThread(Ball ball){  
 b = ball;  
 setPriority(getBallPriority(ball));  
}  
  
private int getBallPriority(Ball ball)  
{  
 Priority priority = ball.priority;  
 return priority == Priority.*Low* ? 4 : *MAX\_PRIORITY*;  
}

Також додамо конструктори до кульок, щоб ми могли створювати різні кульки в залежності від пріоритету:

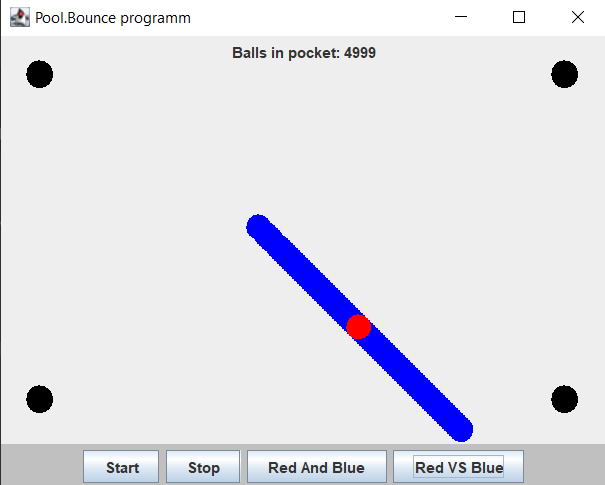
public Ball(BallCanvas c, int[] xPocketsArray, int[] yPocketsArray, Priority priority) {  
 this(c, xPocketsArray, yPocketsArray);  
 this.priority = priority;  
 if(priority == Priority.*High*){  
 color = Color.*RED*;  
 }  
 else if(priority == Priority.*Low*){  
 color = Color.*BLUE*;  
 }  
}  
  
public Ball(BallCanvas c, int[] xPocketsArray, int[] yPocketsArray, Priority priority, int x, int y) {  
 this(c, xPocketsArray, yPocketsArray);  
 this.x = x;  
 this.y = y;  
 this.priority = priority;  
 if(priority == Priority.*High*){  
 color = Color.*RED*;  
 }  
 else if(priority == Priority.*Low*){  
 color = Color.*BLUE*;  
 }  
}



Ми можемо помітити, що кульки, які мають менший пріоритет рухаються повільніше і рвано, ніж ті, що мають високий пріоритет. При цьому, ми можем спостерігати протилежний результат, тобто сині кульки будуть рухатися швидше. Іноді планувальник потоків може вибрати потоки з низьким пріоритетом для виконання щоб збалансувати навантаження.

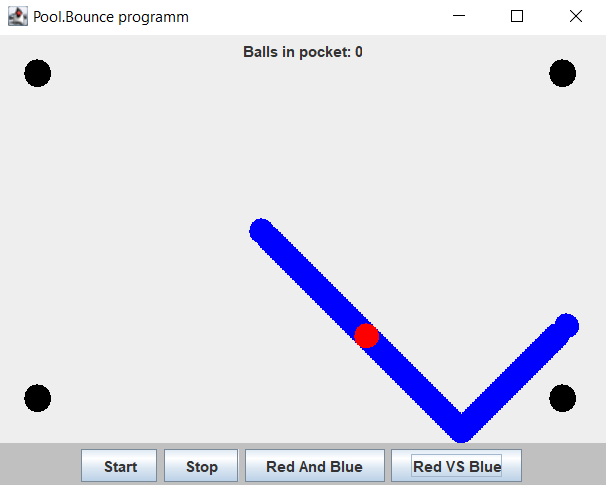
Створимо багато кульок синього та одну кульку червоного кольору:

Кількість синіх кульок: 5000



Не дивлячись на те, що червона кулька додавалася та починала рух останньою серед усіх кульок, вона пересувається швидке, ніж більшість синіх кульок.

Кількість синіх кульок 15000:



В цілому, ситуація, залишилася такою ж, але тепер червона кулька рухається дещо повільніше.

Отже, роблячи висновок, із проведених експериментів, можна сказати, що потоки з більш високим пріоритетом більш важливі для програми, і їм у першу чергу повинен виділятися процесорний час. Проте пріоритети потоків неспроможні гарантувати порядок, у якому виконуються потоки, і це дуже залежить від платформи і від кількості потоків (чим більше потоків, тим більше програма намагається рівномірно розподілити процесорний час між потоками спираючись на їх пріоритет).

Більш детальний лістинг програми можна знайти за посиланням:

<https://github.com/Agupnik/PDC/tree/master/PDCLab1/PDCLab1>

Завдання 4

Побудуйте ілюстрацію для методу join() класу Thread з використанням руху більярдних кульок різного кольору. Поясніть результат, який спостерігається.

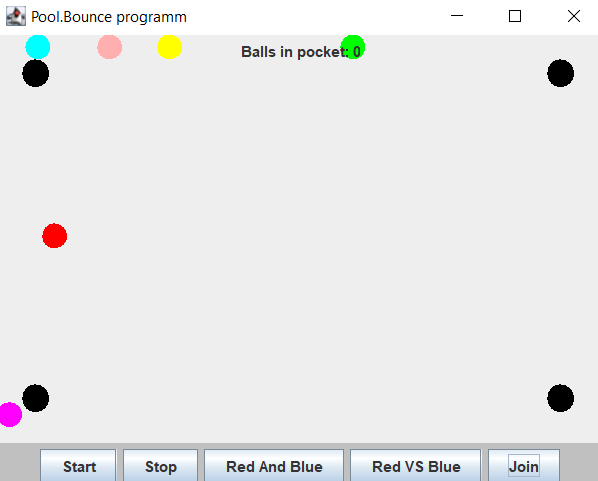
Для цього ми додали конструктор до класу BallThread який приймає попередній потік:

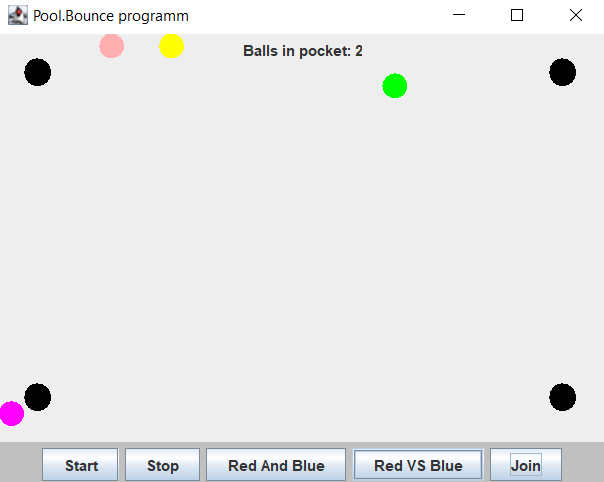
public BallThread(Ball ball, Thread thread){  
 b = ball;  
 this.thread = thread;  
 setPriority(getBallPriority(ball));  
}

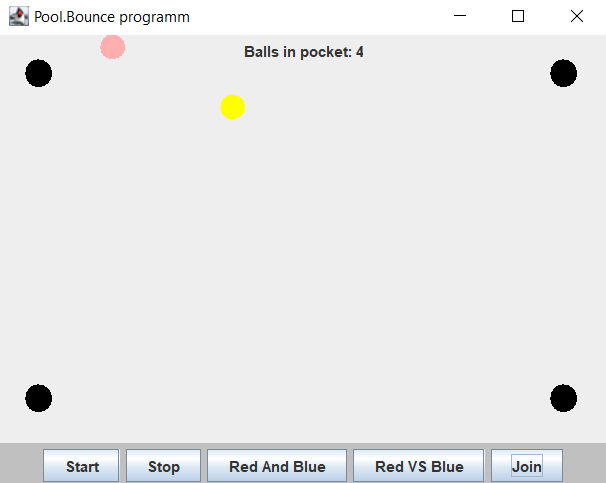
Також ми відредагували сам метод run(), щоб він викликав метод join на попередньому потоці:

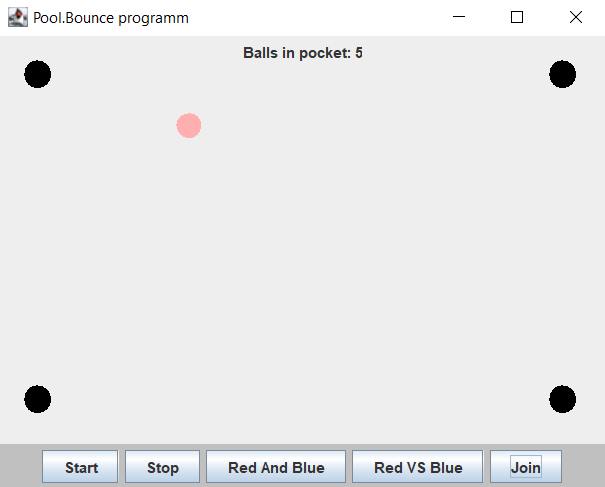
@Override  
public void run(){  
 try{  
 if (thread != null)  
 {  
 thread.join();  
 }  
 while(b.isAlive()){  
 b.move();  
 b.checkPocketIntersections();  
  
 System.*out*.println("Thread name = "  
 + Thread.*currentThread*().getName());  
 Thread.*sleep*(5);  
  
 }  
 } catch(InterruptedException ex){  
 System.*out*.println("Thread finished! Thread name = "  
 + Thread.*currentThread*().getName());  
 }  
}

Додаймо 6 кульок та спостерігаємо наступний результат:









Тепер рухається лише одна кулька на полі і як тільки, вона потрапляє у лузу (тобто знищується потік, у якому знаходиться опрацювання кульки) відразу починає рухатися інша кулька.

Так відбувається тому, що метод join() призупиняє виконання поточного потоку доти, доки інший потік не закінчить виконання, тобто, кожного разу, коли ми створювали нову кульку, ми призупиняли потік для попередньо створеної кульки, саме тому, першою починає рухатися остання створена кулька, а всі інші будуть «чекати» поки вона потрапить у лузу.

Більш детальний лістинг програми можна знайти за посиланням:

<https://github.com/Agupnik/PDC/tree/master/PDCLab1/PDCLab1>

Завдання 5

Створіть два потоки, один з яких виводить на консоль символ ‘-‘, а інший – символ ‘|’. Запустіть потоки в основній програмі так, щоб вони виводили свої символи в рядок. Виведіть на консоль 100 таких рядків. Поясніть виведений результат. Використовуючи найпростіші методи управління потоками, добийтесь почергового виведення на консоль символів.

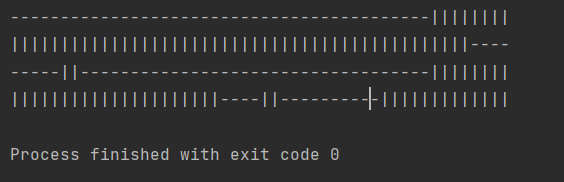
Лістинг друкуючого класу:

public class PrinterUnsync implements IPrinter {  
  
 private int characterCount;  
 public void Write(char character)  
 {  
 System.*out*.print(character);  
 characterCount++;  
 if (characterCount % 50 == 0) {  
 System.*out*.print('\n');  
 }  
 }  
}

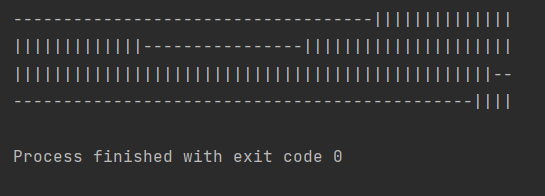
Лістинг потоку, який буде друкувати:

public class SymbolsThread extends Thread  
{  
 private final IPrinter printer;  
 private final int numberOfSymbols;  
 private final char symbol;  
  
 public SymbolsThread(IPrinter printer, int numberOfSymbols, char symbol)  
 {  
 this.printer = printer;  
 this.numberOfSymbols = numberOfSymbols;  
 this.symbol = symbol;  
 }  
  
 @Override  
 public void run()  
 {  
 for (int i = 0; i < numberOfSymbols; i++)  
 {  
 printer.Write(symbol);  
 }  
 }  
}

Результат без синхронізації потоків:



Як можемо побачити результат виводу ми отримали доволі випадковий. Кожного разу, коли ми запускаємо програму, ми будемо отримувати різні результати:



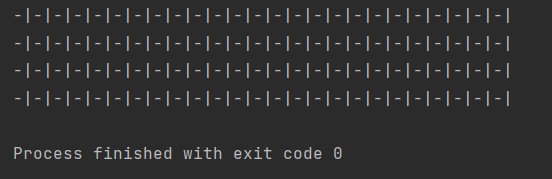
При роботі потоки нерідко звертаються до якихось загальних ресурсів, визначених поза потоком, наприклад, звернення до спільного об’єкту IPrinter, який ми визначаємо у методі main(). Якщо одночасно кілька потоків звернуться до загального ресурсу, результати виконання програми можуть бути несподіваними і навіть непередбачуваними.

Щоб уникнути подібної ситуації, треба синхронізувати потоки.

Використаємо для цього блок synchronized а блокування будемо робити на загальному об’єкті, який створимо раніше:

public class PrinterSync implements IPrinter {  
 private final Object lock = new Object();  
 private char currentCharacter;  
 private int characterCount;  
  
 public void Write(char character)  
 {  
 synchronized (lock)  
 {  
 if (currentCharacter == character)  
 {  
 try  
 {  
 lock.wait();  
 } catch (InterruptedException ignored) {}  
 }  
 System.*out*.print(character);  
 characterCount++;  
 if (characterCount % 50 == 0) {  
 System.*out*.print('\n');  
 }  
 currentCharacter = character;  
 lock.notify();  
 }  
 }  
}

Логіка, така: якщо у нас співпадають символи, які мають друкуватися, то ми переводимо потік у очікування методом wait(), якщо символи не співпадають, то ми друкуємо символ і викликаємо метод notify(), який буде продовжувати роботу потоку, у якого раніше був викликаний метод wait(). Таким чином ми досягнемо почергового виведення на консоль символів:



Більш детальний лістинг програми можна знайти за посиланням:

<https://github.com/Agupnik/PDC/tree/master/PDCLab1/PDCLab1>

Завдання 6

Створіть клас Counter з методами increment() та decrement(), які збільшують та зменшують значення лічильника відповідно. Створіть два потоки, один з яких збільшує 100000 разів значення лічильника, а інший – зменшує 100000 разів значення лічильника. Запустіть потоки на одночасне виконання. Спостерігайте останнє значення лічильника. Поясніть результат. 10 балів. Використовуючи синхронізований доступ, добийтесь правильної роботи лічильника при одночасній роботі з ним двох і більше потоків. Опрацюйте використання таких способів синхронізації: синхронізований метод, синхронізований блок, блокування об’єкта. Порівняйте способи синхронізації.

Створимо клас Counter:

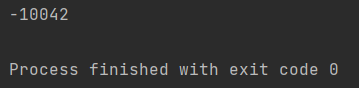
public class Counter  
{  
 protected int count;  
  
 public void increment()  
 {  
 count++;  
 }  
  
 public void decrement()  
 {  
 count--;  
 }  
  
 public int getCount()  
 {  
 return count;  
 }  
}

Створимо два різні потоки, які будуть збільшувати та зменшувати значення лічильника:

public class DecrementThread extends Thread  
{  
 private final Counter counter;  
 private final int numberOfOperations;  
  
 public DecrementThread(Counter counter, int numberOfOperations)  
 {  
 this.counter = counter;  
 this.numberOfOperations = numberOfOperations;  
 }  
  
 @Override  
 public void run()  
 {  
 for (int i = 0; i < numberOfOperations; i++)  
 {  
 counter.decrement();  
 }  
 }  
}

public class IncrementThread extends Thread  
{  
 private final Counter counter;  
 private final int numberOfOperations;  
  
 public IncrementThread(Counter counter, int numberOfOperations)  
 {  
 this.counter = counter;  
 this.numberOfOperations = numberOfOperations;  
 }  
  
 @Override  
 public void run()  
 {  
 for (int i = 0; i < numberOfOperations; i++)  
 {  
 counter.increment();  
 }  
 }  
}

Запустимо ці два потоки на одночасне виконання і отримаємо наступний результат:



Спостерігаємо ту саму ситуацію, що і у минулому завданні. Замість того, щоб отримати 0, ми отримали зовсім не те, що очікували через звернення до спільного ресурсу.

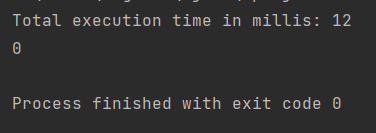
Для синхронізації, використаємо синхронізований метод:

При застосуванні оператора synchronized до методу, поки цей метод не завершить виконання, монопольний доступ має тільки один потік - перший, який почав його виконання.

public class CounterSynchronizedMethod extends Counter{  
 @Override  
 public synchronized void increment()  
 {  
 count++;  
 }  
  
 @Override  
 public synchronized void decrement()  
 {  
 count--;  
 }  
}

Створимо 4 потоки – 2 на додавання та 2 на віднімання

Отримаємо наступний результат:



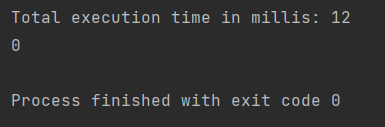
Для синхронізації використаємо синхронізований блок:

Коли виконання коду доходить до оператора synchronized, монітор об'єкта res блокується, і на час його блокування монопольний доступ до блоку коду має лише один потік, який і зробив блокування. Після закінчення роботи блоку коду монітор об'єкта res звільняється і стає доступним для інших потоків.

public class CounterSynchronizedBlock extends Counter  
{  
 @Override  
 public void increment()  
 {  
 synchronized (this)  
 {  
 count++;  
 }  
 }  
  
 @Override  
 public synchronized void decrement()  
 {  
 synchronized (this)  
 {  
 count--;  
 }  
 }  
}

Створимо 4 потоки – 2 на додавання та 2 на віднімання

Отримаємо наступний результат:



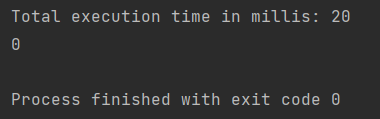
Для синхронізації використаємо блокування об’єкта:

Для керування доступом до спільного ресурсу як альтернатива оператору synchronized ми можемо використовувати блокування об’єкта. Клас ReentrantLock, що реалізує інтерфейс Lock, також, як і synchronized, забезпечує багатопоточність, але має додаткові можливості, пов'язані з опитуванням про блокування (lock polling), очікуванням блокування протягом певного часу та перериванням очікування блокування. Крім того, ReentrantLock пропонує набагато більшу ефективність функціонування в умовах жорсткої змагальності. Іншими словами, коли кілька потоків намагаються отримати доступ до спільно використовуваного ресурсу, віртуальній машині JVM потрібно менше часу на встановлення черговості потоків і більше часу на її виконання.

import java.util.concurrent.locks.ReentrantLock;  
  
public class CounterObjectLock extends Counter  
{  
 private final ReentrantLock locker = new ReentrantLock();  
  
 @Override  
 public void increment()  
 {  
 locker.lock();  
 try{  
 count++;  
 }  
 finally{  
 locker.unlock();  
 }  
 }  
  
 @Override  
 public synchronized void decrement()  
 {  
 locker.lock();  
 try{  
 count--;  
 }  
 finally{  
 locker.unlock();  
 }  
 }  
}

Створимо 4 потоки – 2 на додавання та 2 на віднімання

Отримаємо наступний результат:



Порівнюючи всі ці 3 методи, можна сказати, що самим повільним по часу виявилося блокування об’єкта, хоча воно і є більш ефективним у задачах де потрібні переривання очікування блокування та де є жорсткі умови “race condition”. Різниці у часі між синхронізованим методом та синхронізованим блоком немає. Різниця між ними лише у тому, що вони будуть блокувати.

Синхронізований метод набуває блокування на весь об'єкт. Це означає, що жоден інший потік не може використовувати будь-який синхронізований метод у всьому об’єкті, поки метод виконується одним потоком.

Синхронізований блок набуває блокування в об'єкті між дужками після ключового слова synchronized. Це означає, що жоден інший потік не може отримати блокування на заблокованому об’єкті, поки не відбудеться вихід із синхронізованого блоку.

Більш детальний лістинг програми можна знайти за посиланням:

<https://github.com/Agupnik/PDC/tree/master/PDCLab1/PDCLab1>

Висновок

На цій лабораторній роботі ми розібрали, що таке потік, навчилися створювати потоки та працювати із ними, розібрали способи синхронізації потоків та блокування. Вивчили методи join(), wait(), notify() – як з ними працювати, та коли використовувати.