НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Лабораторна робота №5

із дисципліни «Технології паралельних та розподілених обчислень»

на тему «Застосування високорівневих засобів паралельного програмування для побудови алгоритмів імітації та дослідження їх ефективності»

Підготував:

Студент ФІОТ, гр. ІТ-83,

Троян О.С,

Перевірив:

пос. Дифучина О. Ю.

Київ 2020

Мета роботи: застосувати високорівневі засоби паралельного програмування для побудови алгоритмів імітації та дослідження їх ефективності.

Завдання:

1. З використанням пулу потоків побудувати алгоритм імітації багатоканальної системи масового обслуговування з обмеженою чергою, відтворюючи функціонування кожного каналу обслуговування в окремій підзадачі. Результатом виконання алгоритму є розраховані значення середньої довжини черги та ймовірності відмови.

   40 балів.

2. З використанням багатопоточної технології організувати паралельне виконання прогонів імітаційної моделі СМО для отримання статистично значимої оцінки середньої довжини черги та ймовірності відмови.

   20 балів.

3. Виводити результати імітаційного моделювання (стан моделі та чисельні значення вихідних змінних) в окремому потоці для динамічного відтворення імітації системи.

   20 балів.

4. Побудувати теоретичні оцінки показників ефективності для одного з алгоритмів практичних завдань 2-5.

   20 балів.

Бонусне завдання

1. Розробити модель паралельних обчислень для одного з алгоритмів, побудованих при виконанні лабораторних робіт 2-5, з використанням стохастичної мережі Петрі.

   25 балів.

2. Дослідити на моделі зростання часу виконання паралельного алгоритму при збільшенні розміру оброблюваних даних.

   25 балів.

Хід роботи

1. Створюємо чергу яка буде містити в собі задачі, а також методи які будут добавляти та викликати ці задачі. Черга має обмежений об'єм.

```
public class QueueStorage<Task extends ChannelTask> {
    private static final int MAX_QUEUE_SIZE = 500;
    private final BlockingQueue<Task> queue = new LinkedBlockingQueue<>();

    private int deniesCount;
    private int processedCount;

    public synchronized void addTask(Task task){
        if (queue.size() > MAX_QUEUE_SIZE){
            deniesCount++;
            return;
        }

        queue.add(task);
        processedCount++;

        notify();
    }

    public synchronized Task take() throws InterruptedException {
        while (queue.isEmpty()){
            wait();
        }
        return queue.take();
    }

    public int getDeniesCount() {
        return deniesCount;
    }

    public int getProcessedCount() {
        return processedCount;
    }

    public int getCurrentQueueSize(){
        return queue.size();
    }
}
```

В випадку якщо додається елемент і черга переповнена збільшуємо лічильник відмов, в іншому ж разі збільшуємо лічильник відмов та добавляємо транзакцію в чергу на обробку.

2. Створимо класс який буде моніторити стан черги

```java
public class StorageStatistics<Task extends ChannelTask> extends Thread {
    private final QueueStorage<Task> storage;
    private ArrayList<Integer> measures = new ArrayList<>();

    public StorageStatistics(QueueStorage<Task> storage) {
        this.storage = storage;
    }

    @Override
    public void run() {
        while (true){
            measures.add(storage.getCurrentQueueSize());
            System.out.println("Avg queue size: " + getAverageQueueSize());
            System.out.println(storage.getProcessedCount());
            System.out.println(storage.getDeniesCount());
            System.out.println("Denies possibility: " + (double) storage.getDeniesCount() / (storage.getDeniesCount() + s
            System.out.println( x: "==============");
            try {
                Thread.sleep( millis: 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public int getAverageQueueSize(){
        int sum = 0;

        for (Integer measure : measures) {
            sum += measure;
        }

        return sum / measures.size();
    }
}
```

Цей клас отримує кількість прийнятих та забракованих транзакцій і вираховує відсоток від загальної кількості, а також зберігає поточну кількість транзакцій в черзі і вираховує їх середнє число. Всю цю інформацію виводить у зрозумілому для користувача виді.

3. Класс для створення транзакцій

```java
public class Producer {
    private final QueueStorage<ChannelTask> storage;
    private final Random random = new Random();

    public Producer(QueueStorage<ChannelTask> storage) {
        this.storage = storage;
    }

    public void produce() {
        for (; ;){
            storage.addTask(new ChannelTask() { });
            try {
                Thread.sleep((long)(Math.abs(random.nextGaussian()) * 10));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Безкінечний цикл створює нову транзакцію в випадковий момент часу від 1 до 10 сек.

4. Створюємо интерфейс воркера. Воркери будут виконувати задачі паралельно. Імплементація буде далі.

```java
public interface ChannelWorker<Task extends ChannelTask> extends Runnable {
    void setTask(Task task);
}
```

5. Задача яка передається для виконання. Немає методів, тому як ми лише імітуємо роботу.

```java
public interface ChannelTask {
}
```

6. Менеджер який виконує задачі.

```java
public class ChannelManager<Worker extends ChannelWorker<Task>, Task extends ChannelTask> {
    private final ExecutorService executor = Executors.newFixedThreadPool( nThreads: 8);

    private final QueueStorage<Task> storage;
    private final ConcurrentHashMap<ChannelWorker<Task>, Future<?>> workers = new ConcurrentHashMap<>();

    public ChannelManager(QueueStorage<Task> storage, List<ChannelWorker<Task>> workers) {
        this.storage = storage;
        for (var worker : workers) {
            this.workers.put(worker, CompletableFuture.completedFuture( value: null));
        }
    }

    public void run() {
        while (true){
            Task task = null;
            try {
                task = storage.take();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Optional<Map.Entry<ChannelWorker<Task>, Future<?>>> optional;
            do {
                optional = workers.entrySet().stream().filter(x -> x.getValue().isDone()).findFirst();
            } while (optional.isEmpty());

            var worker = optional.get().getKey();

            worker.setTask(task);
            var future = executor.submit(worker);
            workers.put(worker, future);
        }
    }
}
```

Через конструктор отримує чергу та воркерів які виконують задачі. Кожен воркер має відповідний стан, який сигналізує чи виконав він роботу. З черги береться задача, очікується воркер у якого задача виконана і він виконує наступну.

7. Ініціалізуємо дані

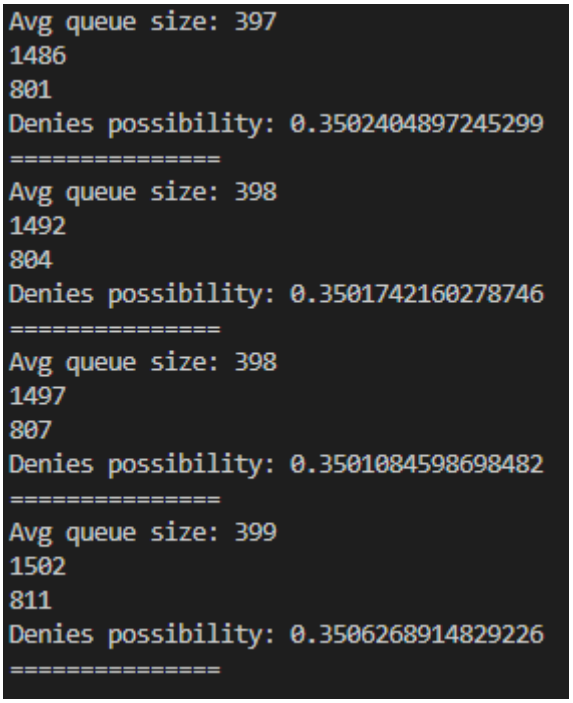
```java
public static void main(String[] args) throws InterruptedException {
    QueueStorage<ChannelTask> storage = new QueueStorage<>();
    List<ChannelWorker<ChannelTask>> workers = new ArrayList<>();
    for (int i = 0; i < 8; i++) {
        workers.add(new ChannelWorker<>() {
            @Override
            public void setTask(ChannelTask task) {
            }

            @Override
            public void run() {
                try {
                    Thread.sleep( millis: 160);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
    ChannelManager<ChannelWorker<ChannelTask>, ChannelTask> manager = new ChannelManager<>(storage, workers);
    Producer producer = new Producer(storage);

    (new Thread(manager::run)).start();
    var t = new Thread(producer::produce);
    var s = new StorageStatistics<>(storage);
    t.start();
    s.start();
    t.join();
}
```

Імплементація воркера, так як ми імітуємо виконання задачі то замість неї ми просто будем призупиняти потік. Таким чином ми ініціалізуємо 8 воркерів.

Далі створюємо менеджер, і передаємо туди чергу и все воркери, а також створюємо виконавця (клас який створює транзакції), і монітор черги.

8. Запускаємо всі задачи в окремих потоках.

   Результат:



Висновок: застосував високорівневі засоби паралельного програмування для побудови алгоритмів імітації та дослідив їх ефективность.