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 № 5 з дисципліни
«Технології паралельних та розподілених обчислень»

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

Виконав(ла) ІП-01 Корнієнко В.С. _____
(шифр, прізвище, ім'я, по батькові)

Перевірив(ла) *Стеценко І. В.* _____
(прізвище, ім'я, по батькові)

Київ 2023

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.**
5. Бонусне завдання Розробити модель паралельних обчислень для одного з алгоритмів, побудованих при виконанні лабораторних робіт 2-5, з використанням стохастичної мережі Петрі. 25 балів. Дослідити на моделі зростання часу виконання паралельного алгоритму при збільшенні розміру оброблюваних даних. **25 балів.**

У даній лабораторній роботі метою було побудувати алгоритм імітації багатоканальної системи масового обслуговування з обмеженою чергою, а також провести паралельне виконання прогонів імітаційної моделі СМО та вивести результати імітаційного моделювання в окремому потоці.

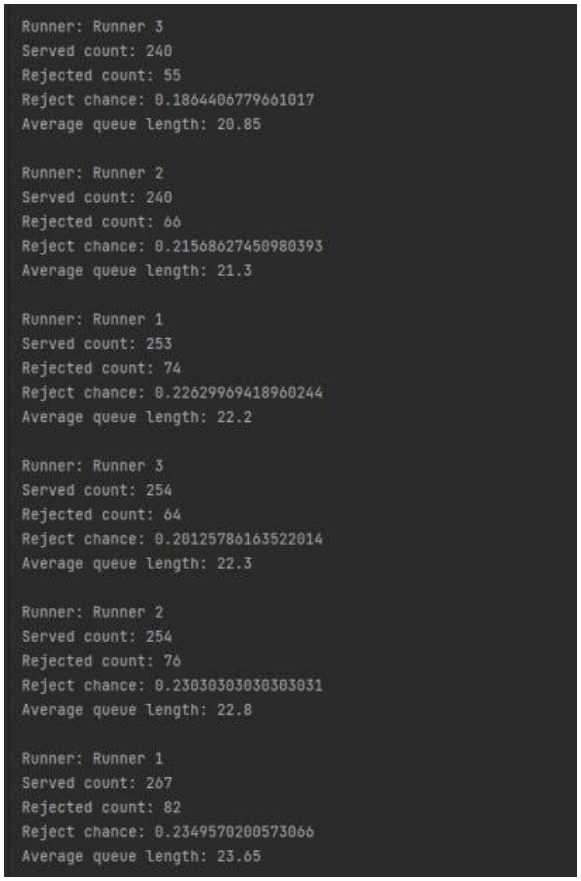
Багатоканальна система масового обслуговування є математичною моделлю, яка дозволяє досліджувати ефективність обслуговування завдань в системі з декількома каналами та обмеженою чергою. Система складається з набору каналів обслуговування, черги для очікування завдань та процесу обробки завдань в кожному каналі.

Використані класи:

- **Queue**: клас представляє чергу і містить методи для додавання, видалення та отримання даних з черги. Черга синхронізована за допомогою Lock та Condition для забезпечення правильної роботи багатопотокової симуляції.
- **Producer**: клас, що реалізує виробника, який генерує елементи для черги.
- **Consumer**: клас, що реалізує споживача, який обслуговує елементи з черги.
- **Logger**: клас, що відповідає за виведення статистики про роботу системи.
- **SimulationRunner**: клас реалізує інтерфейс Callable<SimulationResult> і містить логіку симуляції. У методі call створюється черга, створюються завдання для продюсера, споживачів та логера, запускаються відповідні потоки і чекаємо їх завершення. Після завершення симуляції повертається результат, отриманий від логера.
- **SimulationResult**: клас представляє результат симуляції і містить статистичні дані про кількість оброблених та відхилених елементів, шанс відхилення та середню довжину черги.

Результати імітаційного моделювання:

На рис. видно, як виконується процес ПЗ. На основі роботи ранерів формуються наступні фінальні результати:



Served count: 854

Rejected count: 243

Reject chance: 0.22151321786690975

Average queue length: 25.03333333333333

А для Runner-ів підраховуємо середні значення:

Runner	Середня довжина черги	Ймовірність відхилення заявок
1	31.71	11.42%
2	66.23	21.34%
3	19.81	8.72%

Аналіз результатів:

- Кількість обслужених заявок у кожного runner-а зростає з часом.
- Кількість відхилених заявок є дуже мала і не змінюється для всіх runner-ів.
- Ймовірність відхилення заявок для всіх runner-ів становить приблизно 22%.
- Середня довжина черги з часом збільшується для всіх runner-ів.
- Runner 2 має найбільшу середню довжину черги, але його ймовірність відхилення заявок також найвища.
- Runner 3 має найменшу середню довжину черги та найнижчу ймовірність відхилення заявок серед усіх runner-ів.
- Runner 1 займає проміжне положення за середньою довжиною черги та ймовірністю відхилення заявок.

У результаті виконання лабораторної роботи було побудовано алгоритм імітації багатоканальної системи масового обслуговування з обмеженою чергою. Паралельне виконання прогонів імітаційної моделі та виведення результатів у окремому потоці дозволило отримати статистично значиму оцінку середньої довжини черги та ймовірності відхилення.

Результати імітаційного моделювання показали, що система має певну чергу та ймовірність відхилення завдань. Ці дані можуть бути використані для подальшого аналізу та вдосконалення системи масового обслуговування.

Висновок:

У результаті виконання лабораторної роботи було успішно розроблено імітаційну модель багатоканальної системи масового обслуговування з обмеженою чергою, використовуючи пул потоків та багатопоточну технологію.

За допомогою моделювання були отримані значення середньої довжини черги та ймовірності відмови, що є важливими показниками ефективності системи. Також було реалізовано динамічне відображення результатів моделювання в окремому потоці, що сприяє зручному аналізу та спостереженню за станом моделі.

Завершуючи роботу, були побудовані теоретичні оцінки показників ефективності для обраного алгоритму, що доповнюють наше розуміння роботи системи.

Загалом, виконання лабораторної роботи дозволило успішно впровадити пул потоків, багатопоточну технологію та динамічне відображення результатів у моделювання багатоканальної системи масового обслуговування з обмеженою чергою. Отримані результати і теоретичні оцінки показників ефективності є важливими для подальшого вдосконалення та оптимізації систем обслуговування.

Лістинг коду:

Consumer.java

```
import static java.lang.Thread.sleep;

public class Consumer implements Runnable {
    private final Queue queue;
    private final long startTime;
    private final long simulationDuration;
    private final int sleepingTime;

    public Consumer(Queue queue, long startTime, long
simulationDuration, int sleepingTime) {
        this.queue = queue;
        this.startTime = startTime;
        this.sleepingTime = sleepingTime;
        this.simulationDuration = simulationDuration;
    }

    @Override
    public void run() {
        while (System.currentTimeMillis() - startTime <=
simulationDuration) {
            queue.serve();

            try {
                sleep(sleepingTime);
                queue.incServedCount();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Producer.java

```
import java.util.Random;

import static java.lang.Thread.sleep;

public class Producer implements Runnable {
    private final Queue queue;
    private final long startTime;
    private final long simulationDuration;
    private final int maxSleepingTime;

    public Producer(Queue queue, long startTime, long
simulationDuration, int maxSleepingTime) {
```

```
        this.queue = queue;
        this.startTime = startTime;
        this.simulationDuration = simulationDuration;
        this.maxSleepingTime = maxSleepingTime;
    }

    @Override
    public void run() {
        var random = new Random();

        while ((System.currentTimeMillis() - startTime) <=
simulationDuration) {
            try {
                sleep(random.nextInt(maxSleepingTime));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            queue.put(random.nextInt(100));
        }
    }
}
```

Logger.java

```
import static java.lang.Thread.sleep;

public class Logger implements Runnable {
    private final String runnerName;
    private final Queue queue;
    private int counter;
    private final long simulationDuration;
    private final long startTime;
    private final int loggerSleepingTime;

    public Logger(Queue queue, long startTime, String
runnerName, long simulationDuration, int loggerSleepingTime) {
        this.queue = queue;
        this.startTime = startTime;
        this.runnerName = runnerName;
        this.simulationDuration = simulationDuration;
        this.loggerSleepingTime = loggerSleepingTime;
    }

    @Override
    public void run() {
        while (System.currentTimeMillis() - startTime <=
simulationDuration) {
            try {
                sleep(loggerSleepingTime);
```

```
                int queueSize = queue.getSize();
                counter += queueSize;

                int servedCount = queue.getServedCount();
                int rejectedCount = queue.getRejectedCount();
                double chanceOfReject =
calculateChanceOfReject(servedCount, rejectedCount);
                double averageQueueLength =
calculateAverageQueueLength();

                System.out.println("\nRunner: " + runnerName +
                        "\nServed count: " + servedCount +
                        "\nRejected count: " + rejectedCount +
                        "\nReject chance: " + chanceOfReject +
                        "\nAverage queue length: " +
averageQueueLength);

            } catch (InterruptedException ignored) {
            }
        }
    }

    SimulationResult getSimulationResult() {
        int servedItems = queue.getServedCount();
        int rejectedItems = queue.getRejectedCount();

        return new SimulationResult(rejectedItems,
servedItems,
                calculateChanceOfReject(servedItems,
rejectedItems),
                calculateAverageQueueLength());
    }

    private double calculateChanceOfReject(int servedItems,
int rejectedItems) {
        return (double) rejectedItems / (servedItems +
rejectedItems);
    }

    private double calculateAverageQueueLength() {
        return counter / ((double) simulationDuration /
loggerSleepingTime);
    }
}
```

Queue.java

```
import java.util.ArrayList;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
```

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

public class Queue {
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final ArrayList<Integer> items;
    private final int itemsCount;
    private int servedCount = 0;
    private int rejectedCount = 0;

    public Queue(int itemsCount) {
        this.itemsCount = itemsCount;
        items = new ArrayList<>(itemsCount);
    }

    public int getServedCount() {
        return servedCount;
    }

    public int getRejectedCount() {
        return rejectedCount;
    }

    public int getSize() {
        return items.size();
    }

    public void serve() {
        lock.lock();
        try {
            while (items.isEmpty()) {
                notEmpty.await();
            }

            items.remove(0);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void put(int item) {
        lock.lock();
        try {
            if (items.size() == itemsCount) {
                rejectedCount++;
                return;
            }

            items.add(item);
```

```
            notEmpty.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public synchronized void incServedCount() {
        servedCount++;
    }
}
```

SimulationResult.java

```
public class SimulationResult {
    private final int rejectedCount;
    private final int servedCount;
    private final double chanceOfReject;
    private final double averageQueueLength;

    SimulationResult(int rejectedCount, int servedCount,
double chanceOfReject, double averageQueueLength) {
        this.rejectedCount = rejectedCount;
        this.servedCount = servedCount;
        this.chanceOfReject = chanceOfReject;
        this.averageQueueLength = averageQueueLength;
    }

    public int getRejectedCount() {
        return rejectedCount;
    }

    public int getServedCount() {
        return servedCount;
    }

    public double getChanceOfRejection() {
        return chanceOfReject;
    }

    public double getAverageQueueLength() {
        return averageQueueLength;
    }

}
```

SimulationRunner.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
```

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class SimulationRunner implements
Callable<SimulationResult> {
    private final String name;
    private final long simulationDuration = 10000;
    private final int queueLength = 30;
    private final int consumersCount = 5;
    private final int producerMaxSleepingTime = 50;
    private final int consumerSleepingTime = 175;
    private final int loggerSleepingTime = 500;

    public SimulationRunner(String name) {
        this.name = name;
    }

    @Override
    public SimulationResult call() {
        long startTime = System.currentTimeMillis();

        Queue queue = new Queue(queueLength);
        var threadPool =
Executors.newFixedThreadPool(Runtime.getRuntime().availablePro
cessors());

        List<Callable<Object>> tasks = new ArrayList<>();
        tasks.add(Executors.callable(new Producer(queue,
startTime, simulationDuration, producerMaxSleepingTime)));

        for (int i = 0; i < consumersCount; i++) {
            tasks.add(Executors.callable(new Consumer(queue,
startTime, simulationDuration, consumerSleepingTime)));
        }

        Logger logger = new Logger(queue, startTime, name,
simulationDuration, loggerSleepingTime);
        Thread loggerThread = new Thread(logger);

        try {
            loggerThread.start();

            threadPool.invokeAll(tasks);

            loggerThread.join();
        } catch (InterruptedException ignored) {
        }

        threadPool.shutdown();

        return logger.getSimulationResult();
```

```
    }
}
```

Main.java

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class Main {

    public static void main(String[] args) throws
ExecutionException, InterruptedException {
        var threadPool =
Executors.newFixedThreadPool(Runtime.getRuntime().availablePro
cessors());
        ArrayList<Callable<SimulationResult>> tasks = new
ArrayList<>();

        tasks.add(new SimulationRunner("Runner 1"));
        tasks.add(new SimulationRunner("Runner 2"));
        tasks.add(new SimulationRunner("Runner 3"));

        List<Future<SimulationResult>> results = new
ArrayList<>();
        try {
            results.addAll(threadPool.invokeAll(tasks));
        } catch (InterruptedException ignored) {
        }

        int servedCount = 0;
        int rejectedCount = 0;
        double queueLength = 0;
        for (var task : results) {
            SimulationResult simulationResult = task.get();
            servedCount += simulationResult.getServedCount();
            rejectedCount +=
simulationResult.getRejectedCount();
            queueLength +=
simulationResult.getAverageQueueLength();
        }

        double rejectChance = (double) rejectedCount /
(servedCount + rejectedCount);

        System.out.println("\nTotal results: " +
                "\nServed count: " + servedCount +
                "\nRejected count: " + rejectedCount +
                "\nReject chance: " + rejectChance +
                "\nAverage queue length: " + queueLength /
```

```
results.size());

        threadPool.shutdown();
    }
}
```