МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО» ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

Звіт

З компʼютерного практикуму № 5 з дисципліни

«Технології паралельних обчислень»

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

**Виконав(ла)** *ІП-15 Мєшков Андрій*

(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Дифучина О. Ю.*

(шифр, прізвище, ім'я, по батькові)

Київ 2024

# 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.

# ХІД РОБОТИ

Лістинг коду:

App.java

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.*\**;

public class App {

public static void main(String[] args) throws InterruptedException, ExecutionException {

int testsNum = 5;

int channelsCount = 3;

int queueLength = 4;

int averageTaskTime = 120;

int averageWaitTime = 20;

ArrayList<Callable<Statistic>> tests = *new* ArrayList<>();

ArrayList<QueuingSystem> systems = *new* ArrayList<>();

ExecutorService pool = Executors.newFixedThreadPool(testsNum + 1);

Analyzer analyzer = *new* Analyzer(systems, queueLength);

tests.add(analyzer);

*for* (int i = 0; i < testsNum; i++) {

QueuingSystem system = *new* QueuingSystem(channelsCount, queueLength);

Tester tester = *new* Tester(system, analyzer, averageTaskTime, averageWaitTime, 1000);

tests.add(tester);

systems.add(system);

}

List<Future<Statistic>> failureProbabilities = pool.invokeAll(tests);

pool.shutdown();

Double failure = (double) 0;

Statistic statistic = failureProbabilities.get(0).get();

*for* (int systemIndex = 1; systemIndex < failureProbabilities.size(); systemIndex++) {

Statistic futureResult = failureProbabilities.get(systemIndex).get();

Double failureProbability = futureResult.bounceRate;

failure += failureProbability;

}

System.out.println("Ймовірність відмови: " + (failure / systems.size()));

System.out.println("Середнє значення довжини черги: " + statistic.averageQueueLength);

pool.shutdownNow();

}

}

Analyzer.java

import java.util.ArrayList;

import java.util.concurrent.Callable;

public class Analyzer implements Callable<Statistic> {

private ArrayList<QueuingSystem> systems;

private int queueSize;

public boolean isTerminated = false;

Analyzer(ArrayList<QueuingSystem> systems, int queueSize) {

*this*.systems = systems;

*this*.queueSize = queueSize;

}

@Override

public Statistic call() throws Exception {

double testsCount = 0;

double summaryLength = 0;

double numberOfFailures = 0;

int iteration = 0;

*while* (!isTerminated) {

iteration++;

int currentLengths = 0;

*for* (QueuingSystem system *:* systems) {

testsCount++;

int currentLength = system.queue.size();

summaryLength += currentLength;

*if* (currentLength >= queueSize) {

numberOfFailures++;

}

currentLengths += currentLength;

}

*if* (iteration % 10 == 0) {

System.out.print("Поточна довжина: ");

System.out.println((double) currentLengths / (double) systems.size());

System.out.print("Середня довжина черги: ");

System.out.println((double) summaryLength / (double) testsCount);

System.out.print("Ймовірність відмови: ");

System.out.println((double) numberOfFailures / (double) testsCount);

System.out.println();

System.out.println();

}

Thread.sleep(5);

}

double average = (double) summaryLength / testsCount;

double bounceRate = (numberOfFailures / (double) testsCount);

*return* *new* Statistic(

bounceRate,

average

);

}

}

Statistic.java

public class Statistic {

double bounceRate;

double averageQueueLength;

Statistic(double bounceRate, double averageQueueLength) {

*this*.bounceRate = bounceRate;

*this*.averageQueueLength = averageQueueLength;

}

}

import java.util.ArrayList;

import java.util.concurrent.*\**;

QueuingSystem.java

public class QueuingSystem {

public BlockingQueue<Integer> queue;

private int channelCount;

private int queueLength;

private ExecutorService pool;

private ArrayList<Channel> channels;

public QueuingSystem(int channelCount, int queueLength) {

queue = *new* ArrayBlockingQueue<Integer>(queueLength);

*this*.channelCount = channelCount;

*this*.queueLength = queueLength;

pool = Executors.newFixedThreadPool(channelCount);

channels = *new* ArrayList<>();

}

public void start() {

*for* (int i = 0; i < channelCount; i++) {

Channel channel = *new* Channel(queue);

pool.execute(channel);

channels.add(channel);

}

pool.shutdown();

}

public boolean add(Integer workTime) {

*if* (queue.size() >= queueLength) {

*return* false;

}

queue.add(workTime);

*return* true;

}

public void stop() throws InterruptedException {

*for* (Channel channel *:* channels) {

channel.isTerminated = true;

*if* (queue.size() < queueLength) {

queue.add(0);

}

}

pool.awaitTermination(10, TimeUnit.SECONDS);

}

}

Channel.java

import java.util.concurrent.BlockingQueue;

public class Channel implements Runnable {

private final BlockingQueue<Integer> queue;

public boolean isTerminated = false;

public Channel(BlockingQueue<Integer> queue) {

*this*.queue = queue;

}

@Override

public void run() {

*while* (!isTerminated) {

*try* {

Integer taskTime = queue.take();

Thread.sleep(taskTime);

} *catch* (InterruptedException e) {

*throw* *new* RuntimeException(e);

}

}

}

}

Tester.java

import java.util.Random;

import java.util.concurrent.*\**;

public class Tester implements Callable<Statistic> {

private QueuingSystem system;

private Analyzer analyzer;

private int averageTaskTime;

private int averageWaitTime;

private int workTime;

public Tester(QueuingSystem system, Analyzer analyzer, int averageTaskTime, int averageWaitTime, int workTime) {

*this*.system = system;

*this*.analyzer = analyzer;

*this*.averageTaskTime = averageTaskTime;

*this*.averageWaitTime = averageWaitTime;

*this*.workTime = workTime;

}

@Override

public Statistic call() {

long startTime = System.currentTimeMillis();

Random random = *new* Random();

system.start();

int testsNum = 0;

int failureCount = 0;

*while* (startTime + workTime > System.currentTimeMillis()) {

*// int waitTime = random.nextInt(2 \* averageWaitTime - 1) + 1;*

*// int taskTime = random.nextInt(2 \* averageTaskTime - 1) + 1;*

int waitTime = averageWaitTime;

int taskTime = averageTaskTime;

boolean isAdded = system.add(taskTime);

testsNum++;

*if* (!isAdded) {

failureCount++;

}

*try* {

Thread.sleep(waitTime);

} *catch* (InterruptedException e) {

*throw* *new* RuntimeException(e);

}

}

*try* {

analyzer.isTerminated = true;

system.stop();

} *catch* (InterruptedException e) {

*throw* *new* RuntimeException(e);

}

*return* *new* Statistic((double) failureCount / (double) testsNum, 0);

}

}

**Результат:**

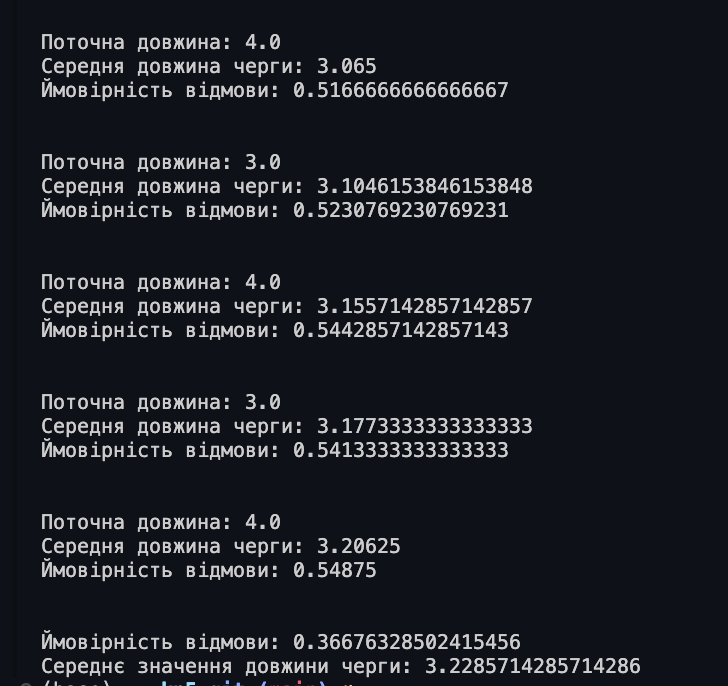


Рисунок 1 – Результат запуску програми

# ВИСНОВКИ

В результаті роботи над комп’ютерним практикумом було розроблено програму, що імітує роботу багатоканальної СМО