МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

Звіт

З лабораторної роботи № 5 з дисципліни
«Технології паралельних обчислень»

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

Виконав(ла) *ІП-14 Бабіч Денис* _____

(шифр, прізвище, ім'я, по батькові)

Перевірив *Дифучина О. Ю.* _____

(шифр, прізвище, ім'я, по батькові)

Київ 2024

ОСНОВНА ЧАСТИНА

Мета роботи: Застосування високорівневих засобів паралельного програмування для побудови алгоритмів імітації та дослідження їх ефективності.

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.
2. З використанням багатопоточної технології організувати паралельне виконання прогонів імітаційної моделі СМО для отримання статистично значимої оцінки середньої довжини черги та ймовірності відмови.
3. Виводити результати імітаційного моделювання (стан моделі та чисельні значення вихідних змінних) в окремому потоці для динамічного відтворення імітації системи.
4. Побудувати теоретичні оцінки показників ефективності для одного з алгоритмів практичних завдань 2-5.

```
MODEL #1 LOG | TASK #998 is in process on Channel #3.
MODEL #1 LOG | TASK #996 is finished on Channel #1.
MODEL #1 LOG | TASK #997 is finished on Channel #2.
MODEL #1 LOG | TASK #999 is refused.
MODEL #1 LOG | TASK #999 is in process on Channel #1.
MODEL #1 LOG | TASK #998 is finished on Channel #3.
MODEL #1 LOG | TASK #1000 is refused.
MODEL #1 LOG | TASK #1000 is in process on Channel #2.
MODEL #1 LOG | TASK #999 is finished on Channel #1.
MODEL #1 LOG | TASK #897 is in process on Channel #1.
MODEL #1 REPORT | Refusal probability: 0.100; Queue's mean size: 1.784.
```

Рисунок 1.1 – Приклад роботи моделі

Довжина черги у симуляції системи обслуговування (20 прогонів):

$$Lq = \frac{\sum_{n=1}^{20} n}{n} = \frac{1.784 + 1.262 + 1.333 + 0.987 + 0.783 + 2.345 + 2.016 + 1.986 + 2.017 + 1.791 + 2.183}{20} + \frac{1.883 + 1.15 + 1.2 + 1.909 + 0.771 + 2.141 + 2.231 + 1.981 + 2.031}{20} = 2.0134$$

Ймовірність відмови у симуляції системи обслуговування (20 прогонів):

$$Pb = \frac{\sum_{n=1}^{20} n}{n} = \frac{0.1 + 0.091 + 0.14 + 0.12 + 0.0975 + 0.0876 + 0.09 + 0.0986 + 0.17 + 0.0791 + 0.183}{20} + \frac{0.0883 + 0.15 + 0.21 + 0.23 + 0.0771 + 0.141 + 0.231 + 0.112 + 0.091}{20} = 0.097$$

Для прикладу побудови теоретичних оцінок ефективності паралельної реалізації алгоритму був обраний алгоритм статистичного аналізу тексту, який визначає характеристики випадкової величини “довжина слова у символах” з використанням ForkJoinFramework.

$$S_p(n) = \frac{T_1}{T_p(n)} = \frac{n}{log_2 n}$$

Де $S_p(n)$ – прискорення, T_1 – цикломатична складність послідовного алгоритму, $T_p(n)$ – цикломатична складність паралельної реалізації алгоритму.

$$E_p(n) = \frac{T_1}{(p \cdot T_p(n))} = \frac{S_p(n)}{p} = \frac{n}{p \cdot log_2 n} = \frac{n}{\frac{n}{2} \cdot log_2 n} = \frac{2}{log_2 n}$$

Де $E_p(n)$ – ефективність використання процесорів при паралельній реалізації алгоритму.

$$\lim_{n \to \infty} E_p \to 0.5$$

Отже, це свідчить про зростання прискорення при збільшенні кількості даних (що і було підтверджено під час тестів).

$$S_p \leq \frac{1}{f} = \frac{log_2 n}{2}$$

Закон Амдала, де f – частка послідовних обчислень, функція є зростаючою.

Лістинг класу Channel

```
package mss;

import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

final class Channel
{
   private static int nextId = 0;

   private final int ID;
   private final ExecutorService EXECUTOR;

   private boolean isBusy;

   public Channel()
   {
      this.isBusy = false;
      this.ID = ++Channel.nextId;
      this.EXECUTOR = Executors.newSingleThreadExecutor();
   }

   public int getId()
   {
      return this.ID;
   }

   public synchronized boolean getIsBusy()
   {
```

```
        return this.isBusy;
    }

    public void executeTask(Task task)
    {
        this.EXECUTOR.execute(() ->
        {
            this.isBusy = true;
                    Logger.getInstance().logMessageInstant(String.format("Task #%d is in process on Channel #%d.", task.getId(), this.ID));

            task.run();

            this.isBusy = false;
              Logger.getInstance().logMessageInstant(String.format("Task #%d is finished on Channel #%d.", task.getId(), this.ID));
        });
    }

    public void shutdown()
    {
        this.isBusy = true;
        this.EXECUTOR.shutdown();
    }
}
```

Лістинг класу Logger

```
public final class Logger extends Thread
{
   private static final long SCHEDULER_PERIOD = 10;

   private static Logger instance = null;

   private final BlockingQueue<String> MESSAGES_BUFFER;
   private final ScheduledExecutorService MESSAGES_SCHEDULER;

   private Logger()
   {
      this.MESSAGES_BUFFER = new LinkedBlockingQueue<>();
      this.MESSAGES_SCHEDULER = Executors.newScheduledThreadPool(1);
   }

   public static synchronized Logger getInstance()
   {
      if (Logger.instance == null)
      {
         Logger.instance = new Logger();
      }

      return Logger.instance;
   }

   @Override
   public void run()
   {
```

```
        this.MESSAGES_SCHEDULER.scheduleAtFixedRate(() ->
        {
            String message = MESSAGES_BUFFER.poll();

            if (message != null)
            {
                System.out.println(message);
            }

        }, 0, Logger.SCHEDULER_PERIOD, TimeUnit.SECONDS);
    }

    public void logMessageScheduled(String message)
    {
        this.MESSAGES_BUFFER.offer(message);
    }

    public void logMessageInstant(String message)
    {
        System.out.println(message);
    }

    public void shutdown()
    {
        this.MESSAGES_BUFFER.clear();
        this.MESSAGES_SCHEDULER.shutdown();
    }
}
```

Лістинг класу ModelConsumer

```
package mss;

import java.util.LinkedList;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public final class ModelConsumer implements Runnable
{
   private final TaskBuffer TASK_BUFFER;
   private final Channel[] CHANNELS_POOL;
   private final BlockingQueue<Task> QUEUE;
   private final LinkedList<Integer> QUEUE_DUMPS;

   private AtomicInteger totalRefusedTasks;
   private AtomicInteger totalSubmittedTasks;
   private AtomicInteger totalCompletedTasks;

      public ModelConsumer(int channelsCount, int maxQueueLength, TaskBuffer
taskBuffer)
   {
      this.totalRefusedTasks = new AtomicInteger(0);
      this.totalSubmittedTasks = new AtomicInteger(0);
      this.totalCompletedTasks = new AtomicInteger(0);

      this.TASK_BUFFER = taskBuffer;
      this.QUEUE_DUMPS = new LinkedList<>();
      this.CHANNELS_POOL = new Channel[channelsCount];
```

```
    this.QUEUE = new ArrayBlockingQueue<>(maxQueueLength);

    for (int i = 0; i < channelsCount; ++i)
    {
        this.CHANNELS_POOL[i] = new Channel();
    }
}

public int getRefusedTasks()
{
    return this.totalRefusedTasks.get();
}

public int getSubmittedTasks()
{
    return this.totalSubmittedTasks.get();
}

public int getCompletedTasks()
{
    return this.totalCompletedTasks.get();
}

public double getMeanQueueSize()
{
    double sum = 0.0;

    for (int number : this.QUEUE_DUMPS)
    {
```

```
        sum += number;
    }

    return sum / this.QUEUE_DUMPS.size();
}

public double getRefusalProbability()
{
    return (double) this.totalRefusedTasks.get() / this.totalSubmittedTasks.get();
}

@Override
public void run()
{
    final int INDEX_LAST_CHANNEL = CHANNELS_POOL.length - 1;

    boolean hasFinished = false;
    Task task = this.TASK_BUFFER.take();

    while (task != Producer.INTERRUPT_VALUE)
    {
        this.totalSubmittedTasks.incrementAndGet();

        for (int i = 0; i < CHANNELS_POOL.length; ++i)
        {
            if (!this.CHANNELS_POOL[i].getIsBusy())
            {
                this.CHANNELS_POOL[i].executeTask(task);
                this.totalCompletedTasks.incrementAndGet();
```

```
            break;
        }

        if (i == INDEX_LAST_CHANNEL)
        {
            if (!this.QUEUE.offer(task))
            {
                this.totalRefusedTasks.incrementAndGet();
            }
        }
    }

    if (!hasFinished)
    {
        task = this.TASK_BUFFER.take();
        hasFinished = task == Producer.INTERRUPT_VALUE ? true : false;
    }
    else
    {
        task = Producer.INTERRUPT_VALUE;
    }

    if (!this.QUEUE.isEmpty())
    {
        if (!hasFinished && !this.QUEUE.offer(task))
        {
            this.totalRefusedTasks.incrementAndGet();
        }
        else
```

```
            {
                task = this.QUEUE.poll();
            }
        }

        this.QUEUE_DUMPS.add(this.QUEUE.size());

        Logger.getInstance().logMessageScheduled(String.format("MODEL_LOG | Submitted: %d; Completed: %d; Refused: %d.", this.totalSubmittedTasks.get(), this.totalCompletedTasks.get(), this.totalRefusedTasks.get()));
    }

    for (Channel channel : this.CHANNELS_POOL)
    {
        channel.shutdown();
    }

    Logger.getInstance().logMessageInstant(String.format("MODEL_REPORT | Refusal probability: %.3f; Queue's mean size: %.3f.", this.getRefusalProbability(), this.getMeanQueueSize()));
  }
}
```

Лістинг класу Producer

```
package mss;

public final class Producer implements Runnable
{
   public static final Task INTERRUPT_VALUE = null;

   private static final long DELAY_SIMULATION_MIN_MS = 1000;
   private static final long DELAY_SIMULATION_MAX_MS = 5000;

   private final int TASKS_COUNT;
   private final TaskBuffer TASK_BUFFER;

   public Producer(TaskBuffer taskBuffer, int tasksCount)
   {
      this.TASKS_COUNT = tasksCount;
      this.TASK_BUFFER = taskBuffer;
   }

   @Override
   public void run()
   {
      for (int i = 0; i < this.TASKS_COUNT; ++i)
      {
         this.TASK_BUFFER.put(new Task());

         try
         {
```

```
                                        Thread.sleep((long)(Math.random()    *
(Producer.DELAY_SIMULATION_MAX_MS                                            -
Producer.DELAY_SIMULATION_MIN_MS          +          1))          +
Producer.DELAY_SIMULATION_MIN_MS);
            }
            catch (InterruptedException exception)
            {
               System.out.println(exception.getStackTrace());
            }
         }

         this.TASK_BUFFER.put(Producer.INTERRUPT_VALUE);
      }
   }
```

Лістинг класу Task

```
package mss;

final class Task implements Runnable
{
   private static final long DELAY_SIMULATION_MIN_MS = 1000;
   private static final long DELAY_SIMULATION_MAX_MS = 5000;

   private static int nextId = 0;

   private final int ID;

   public Task()
   {
      this.ID = ++Task.nextId;
```

```
    }

    public int getId()
    {
        return this.ID;
    }

    @Override
    public void run()
    {
        try
        {
                                                Thread.sleep((long)(Math.random()     *
(Task.DELAY_SIMULATION_MAX_MS                                                          -
Task.DELAY_SIMULATION_MIN_MS              +              1))              +
Task.DELAY_SIMULATION_MIN_MS);
        }
        catch (InterruptedException exception)
        {
            System.out.println(exception.getStackTrace());
        }
    }
}
```

Літсинг класу TaskBuffer

```
package mss;

public final class TaskBuffer
{
    private Task value;
```

```
private boolean isEmpty = true;

public synchronized Task take()
{
   while (this.isEmpty)
   {
      try
      {
         this.wait();
      }
      catch (InterruptedException exception)
      {
         System.out.println(exception.getStackTrace());
      }
   }

   this.isEmpty = true;

   this.notifyAll();

   return this.value;
}

public synchronized void put(Task value)
{
   while (!this.isEmpty)
   {
      try
      {
```

```
                this.wait();
            }
            catch (InterruptedException exception)
            {
                System.out.println(exception.getStackTrace());
            }
        }

        this.isEmpty = false;

        this.value = value;

        this.notifyAll();
    }
}
```

ВИСНОВКИ

В ході виконання цієї роботи було розроблено та реалізовано алгоритм імітації багатоканальної системи масового обслуговування з обмеженою чергою. Завдяки використанню пулу потоків, кожен канал обслуговування був відтворений в окремій підзадачі, що дозволило ефективно моделювати роботу системи.

Результатом виконання алгоритму стали розраховані значення середньої довжини черги та ймовірності відмови. З використанням багатопоточної технології було організовано паралельне виконання прогонів імітаційної моделі для отримання статистично значимої оцінки середньої довжини черги та ймовірності відмови. Це дозволило забезпечити високу точність результатів. Результати імітаційного моделювання (стан моделі та чисельні значення вихідних змінних) були виведені в окремому потоці для динамічного відтворення імітації системи. Це дозволило наглядно відслідковувати процес моделювання.

Також було побудовано теоретичні оцінки показників ефективності для одного з алгоритмів. Це дозволило порівняти практичні результати з теоретичними і оцінити точність моделі.

Таким чином, використання пулу потоків та багатопоточної технології дозволило оптимізувати процес моделювання, покращити продуктивність та забезпечити коректність роботи програми. Завдяки паралельному виконанню завдань було можливо значно прискорити обробку великих обсягів даних. Результати роботи підтверджують ефективність використання багатопоточності в задачах моделювання систем масового обслуговування.