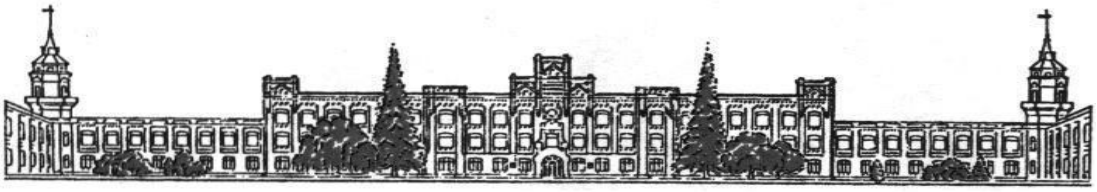


Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Лабораторна робота №2

з дисципліни “Паралельні обчислення”

на тему: “**Дослідження базових примітивів синхронізації**”

Виконала:

студентка III курсу, групи ДА-01

Балуматкіна О. В.

Варіант 4

Прийняла(в):

Київ-2023

Зміст

Мета роботи 3

Завдання 3

Варіант 4 4

Хід роботи 4

Теоретична складова 4

Реалізація власного пулу потоків 6

Реалізація повної програми відповідно до варіанту 7

Аналіз результату роботи програми 10

Висновок 15

Додаток А 16

Додаток Б 18

Додаток В 19

Додаток Г 20

Мета роботи

Розглянути базові примітиви синхронізації та їх особливості, в залежності від обраної мови програмування. Розглянути підходи до побудови ПЗ з використанням паралелізму та ознайомитися з класичною задачею паралелізму у вигляді пулу потоків.

Завдання

1. Ознайомитися з визначенням поняття пул потоків (thread pool), використовуючи даний методичний посібник, або ж сторонні джерела.
2. Реалізувати власний пул потоків з характеристиками, зазначеними в обраному варіанті. Зі спільних характеристик: пул потоків повинен бути написаним коректно відносно обраної мови програмування, повинен мати можливість коректного завершення своєї роботи (моментально, з покиданням всіх активних задач, так і з завершенням активних задач), можливість тимчасової зупинки своєї роботи. Операції ініціалізації та знищення пулу, додавання та вилучення задач в чергу повинні бути потіко-безпечними.
3. Створити програму, котра буде виконувати задачі за обраним варіантом, використовуючи написаний студентом пул потоків. Зі спільних характеристик: код, відповідальний за додавання задач в пул потоків, та сам пул потоків повинні знаходитися в різних потоках виконання.
4. Перевірити та довести коректність роботи програми з використанням системи вводу/виводу інформації в консоль. Зробити обмежене за часом тестування та розрахувати кількість створених потоків та середній час знаходження потоку в стані очікування. Визначити середню довжину кожної черги та середній час виконання задачі (для необмежених черг). Або визначити максимальний та мінімальний час, поки черга була заповнена, кількість відкинутих задач (для обмежених черг).

Варіант 4

Спільні характеристики: програма обов’язково має мінімум один пул потоків котрий обслуговує програму й виконує задачі (вибір механізму генерації задач віддається на вибір студента). Задачі створюються та додаються з окремих потоків виконання (не в тому ж потоці, що й знаходиться пул потоків). Кожна задача займає випадковий реальний час на виконання:

4. Пул потоків обслуговується 4-ма робочими потоками й має одну чергу виконання. Задачі додаються відразу в кінець черги виконання. Черга задач виконується з інтервалом в 45 секунд (буфер наповнюється задачами протягом 45-ти секунд, котрі потім виконуються). Задачі, що додаються під час виконання черги задач відкидаються. Задача займає випадковий час від 6 до 12 секунд.

Хід роботи

Теоретична складова

Відповідно до завдання лабораторної роботи, початково необхідно реалізувати власний пул потоків, що міститиме також характеристики, притаманні варіанту, закріпленому за студентом. Перед безпосереднім початком роботи викладемо теоретичну складову, необхідну для розуміння коду, що був реалізований у ході виконання завдання.

Пул потоків (Thread pool) – механізм управління потоками, який дозволяє перевикористовувати потоки, що вже були створені, замість того, щоб створювати нові потоки кожен раз, коли потрібно виконати деяку задачу. Такий підхід дозволяє значно зменшити витрати ресурсів та покращити продуктивність програми. Загалом в обраній мові програмування Java пул потоків можна використовувати за допомогою класу ThreadPoolExecutor з пакету java.util.concurrent. (дозволяє створювати пули потоків з різними параметрами, такими як кількість потоків, які можуть бути одночасно активні, максимальний розмір черги завдань та час життя потоків).

Пули потоків розрізняються в основному за певними характеристиками:

- За кількістю черг задач;
- За алгоритмом вибору задачі для подальшого виконання;
- За підходом до виконання задач в черзі;
- За можливістю отримання результату виконання.

Монітор – деякий об’єкт, який може використовуватись для управління доступом до спільних ресурсів між потоками, щоб уникнути некоректної поведінки, такої як змінення значень змінних одночасно з різних потоків. У мові Java монітор можна реалізувати за допомогою ключового слова synchronized та блоку synchronized. Якщо потік входить в блок synchronized, то блокується доступ до даного блоку з інших потоків, поки даних потік не покине блок. Також ключове слово можна використовувати до методів, які забезпечують доступ до відповідних полів об’єкту.

М’ютекс – механізм синхронізації доступу до ресурсу між потоками в паралельних обчисленнях. В основі його роботи лежить ідея блокування доступу до ресурсу з боку інших потоків в той момент, коли він використовується одним потоком. Класичний інтерфейс м’ютекса складається з двох методів: lock та unlock. Потік виконання, що викликає метод lock, стає власником м’ютекса, і допоки власник не викликає метод unlock, виклики до методу lock з інших потоків змусить їх чекати виклику unlock від потоку власника.

Семафор – примітив синхронізації, який є схожим на м’ютекс, але обмежує доступ до секції коду не для одного потоку виконання, а для декількох.

Умовна змінна – примітив синхронізації, який блокує виконання потоку (або декількох потоків), допоки інший потік не виконає певну умову (condition) та не оголосить про це через повідомлення потоків виконання, які знаходяться у стані очікування.

Реалізація власного пулу потоків

Реалізований власний пул потоків (CustomThreadPool.java) використовує самостійно реалізовану блокуючу чергу для зберігання завдань, що генеруються спеціальним класом-генератором, відповідно до заздалегідь встановленої кількості, клас Thread використовується для створення масиву робочих потоків всередині пулу потоків, булева змінна running використовується для контролю роботи потоків всередині пулу потоків, а змінні numTasksExecuted та totalExecutionTime використовуються для надання базової статистики по виконаній роботі.

У класі CustomThreadPool реалізовані наступні методи та класи:

- Метод execute(task) – використовується для додання завдання в чергу виконання для пулу потоків;
- Метод taskFinished() – використовується для зміни статистичної змінної, по закінченню виконання певного завдання додає 1 до загального лічильника виконаних за сесію задач. В даному методі використано ключове слово synchronized для заблокування доступу до зміни спільної змінної numTasksExecuted;
- Метод getNumTasksExecuted() – аналогічно використовується для роботи з змінною, що фігурує в статистиці по роботі пулу потоків, метод повертає наявне значення кількості виконаних пулом потоків завдань за дану сесію. В даному методі використано ключове слово synchronized для заблокування доступу до зчитування спільної змінної numTasksExecuted;
- Метод getTotalExecutionTime() – аналогічно використовується для роботи з змінною, що фігурує в статистиці по роботі пулу потоків, метод повертає загальний час роботи програми над даною сесією завдань. . В даному методі використано ключове слово synchronized

для заблокування доступу до зчитування спільної змінної totalExecutionTime;

- Метод stop() – використовується для зупинки даного пулу потоків, переривання всіх робочих потоків та очистки черги завдань, після чого й скидання статистичних змінних в значення 0;
- Клас WorkerThread – розширює Thread, використовується для створення робочих потоків, що виконують завдання (перевизначено метод run()), задані пулу потоків, та виводу інформації про те, який потік приймає завдання на виконання;
- Метод printStatistics() – відповідно до своєї назви, використовується для виводу в консоль невеликої статистики по роботі пулу потоків.

Повний код до класу CustomThreadPool.java наведений у додатку А.

Реалізація повної програми відповідно до варіанту

Як було зазначено у попередньому пункті, власна реалізація пулу потоків також використовує власну реалізацію блокуючої черги, яка використовує чергу на двозв'язному списку та монітори для обмеження доступу до черги декількох потоків, а також надає можливість помістити у чергу завдання будь-якого типу даних. Завдання блокуючої черги – забезпечити можливість блокування (чекання) на операції додавання (put) та видалення (take) елементів, якщо черга порожня або заповнена відповідно. Розберемо реалізовані в класі власної блокованої черги (CustomBlockingQueue.java) методи та конструктор, повний код даного класу наведено у додатку Б.

- Конструктор CustomBlockingQueue<T> - в конструкторі створюється стандартна черга на основі двозв'язного списку, що дозволяє ефективно додавати та видаляти елементи (в нашому випадку – деякі завдання) зі списку;

- Метод put(T element) – додає елемент до нашої реалізованої черги. Ключове слово synchronized визначає, що метод буде виконуватись у потоці, який отримає блокування на об'єкті this (тобто цього класу). Першим кроком методу put(T element) перевіряється, чи була черга порожньою перед додаванням нового елемента. Якщо так, тоді метод notifyAll() повідомляє всі потоки, які чекають на доступ до черги (тобто виконують метод take()), що елемент був доданий, і тепер можна продовжити виконання. Метод offer(T element) додає елемент до кінця черги;
- Метод take() – повертає та видаляє перший елемент з черги (робота за принципом First In First Out). Оскільки черга може виявитись порожньою, метод знаходиться у нескінченному циклі while, який очікує на отримання завдань в чергу. Як тільки в чергу потрапляють якісь завдання, метод видаляє та повертає перший елемент черги та викликає метод notifyAll(), що повідомляє всім потокам про звільнений монітор, тобто інші потоки тепер можуть отримати собі завдання на виконання;
- Метод clear() – простий метод, що використовується для очищення черги, для якої пул потоків вже припинив свою роботу та не буде в подальшому виконувати ці завдання, а також для сповіщення потоків про те, що черга пуста.

Додатково, для роботи програми за отриманим завданням лабораторної роботи, було реалізовано клас-генератор завдань (TaskGenerator.java), повний код якого наведено у додатку В.

- Конструктор TaskGenerator(CustomThreadPool threadPool) – конструктор приймає об’єкт класу CustomThreadPool, який буде

виконувати завдання з блокованої черги, які будуть відповідно згенеровані поточним класом-генератором завдань;

- Метод addTasks(int numTasks) – приймає кількість завдань, які необхідно додати до пула потоків, та додає завдання з використанням методу execute() з класу власного пулу потоків. Кожне завдання створюється як анонімний об’єкт класу Runnable, має випадковий час виконання від 6 до 12 секунд (у ході виконання програми насправді виконання завдання лише імітується з використанням методу speel() для потоку, що отримує поточне завдання), а також інформація по кожному завданню (його номер, коли його виконання розпочате, скільки часу було виділено на імітацію виконання, коли завдання було виконане) та повідомлення про переривання виконання завдання виводиться до консолі в поточному методі.

Останнім класом, призначеним викликати написаний в решті класів програми код, було реалізовано Main (Main.java), повний код якого наведено у додатку Г. Клас приймає фіксовані значення для кількості потоків в пулі потоків, загальної кількості завдань на виконання, час сну для одної сесії виконання задач та кількість повторів програми, якими оперує для подальшого створення пулу потоків на 4 потоки, створення об’єкту класу TaskGenerator для відповідної генерації встановленої кількості завдань на виконання пулом потоків. Головний клас запускає пул потоків на виконання, використовуючи метод sleep() для створення 45-секундного інтервалу, за який пул потоків має виконати якомога більшу кількість завдань. У випадку, якщо пул потоків встиг виконати усі постановлені у сесії завдання, буде просто надрукована коротка статистика по роботі пулу в даній сесії, після чого виконання програми, за умови виконання всіх ітерацій, буде успішно завершено. Якщо ж деякі з завдань не встигнуть вкластись в проміжок 45 секунд, пул потоків перерве своє виконання та відкине отримані

задачі й не стане виконувати ті, що залишаються у черзі на виконання. У випадку, якщо виникають переривання потоків, головний клас виводить відповідні повідомлення в консоль.

Аналіз результату роботи програми

Для прикладу, встановимо кількість повторень виконання програми на 3, як зазначено в фінальному коді. Запустимо програму на виконання та проаналізуємо отримані виводи у консоль. На прикладі першої ітерації розберемо логіку виводу:

```
Iteration #1

THREAD: Task 1 added to thread pool
THREAD: Task 2 added to thread pool
THREAD: Task 3 added to thread pool
THREAD: Task 4 added to thread pool
THREAD: Task 5 added to thread pool
THREAD: Task 6 added to thread pool
THREAD: Task 7 added to thread pool
THREAD: Task 8 added to thread pool
THREAD: Task 9 added to thread pool
THREAD: Task 10 added to thread pool
Task is given to: Thread-0
Task is given to: Thread-2
Task is given to: Thread-1
Task is given to: Thread-3
THREAD: Task 11 added to thread pool
THREAD: Task 12 added to thread pool
THREAD: Task 13 added to thread pool
THREAD: Task 14 added to thread pool
THREAD: Task 15 added to thread pool
THREAD: Task 16 added to thread pool
THREAD: Task 17 added to thread pool
THREAD: Task 18 added to thread pool
```

```
THREAD: Task 19 added to thread pool
THREAD: Task 20 added to thread pool
THREAD: Task 21 added to thread pool
THREAD: Task 22 added to thread pool
THREAD: Task 23 added to thread pool
THREAD: Task 24 added to thread pool
THREAD: Task 25 added to thread pool
THREAD: Task 26 added to thread pool
THREAD: Task 27 added to thread pool
THREAD: Task 28 added to thread pool
THREAD: Task 29 added to thread pool
THREAD: Task 30 added to thread pool
TASK START AND DATA: Task 2 has execution time of 7002ms
TASK START AND DATA: Task 3 has execution time of 6591ms
TASK START AND DATA: Task 4 has execution time of 7113ms
TASK START AND DATA: Task 1 has execution time of 10579ms
TASK DONE: Task 3 finished in 6592ms
Task is given to: Thread-1
TASK START AND DATA: Task 5 has execution time of 8794ms
TASK DONE: Task 2 finished in 7003ms
Task is given to: Thread-3
TASK START AND DATA: Task 6 has execution time of 7441ms
TASK DONE: Task 4 finished in 7115ms
Task is given to: Thread-2
TASK START AND DATA: Task 7 has execution time of 11529ms
TASK DONE: Task 1 finished in 10581ms
Task is given to: Thread-0
TASK START AND DATA: Task 8 has execution time of 10473ms
TASK DONE: Task 6 finished in 7443ms
Task is given to: Thread-3
TASK START AND DATA: Task 9 has execution time of 11035ms
TASK DONE: Task 5 finished in 8794ms
Task is given to: Thread-1
TASK START AND DATA: Task 10 has execution time of 9023ms
TASK DONE: Task 7 finished in 11532ms
```

```
Task is given to: Thread-2
TASK START AND DATA: Task 11 has execution time of 9531ms
TASK DONE: Task 8 finished in 10473ms
Task is given to: Thread-0
TASK START AND DATA: Task 12 has execution time of 7418ms
TASK DONE: Task 10 finished in 9024ms
Task is given to: Thread-1
TASK START AND DATA: Task 13 has execution time of 11519ms
TASK DONE: Task 9 finished in 11036ms
Task is given to: Thread-3
TASK START AND DATA: Task 14 has execution time of 6637ms
TASK DONE: Task 11 finished in 9532ms
Task is given to: Thread-2
TASK START AND DATA: Task 15 has execution time of 9640ms
TASK DONE: Task 12 finished in 7421ms
Task is given to: Thread-0
TASK START AND DATA: Task 16 has execution time of 7602ms
TASK DONE: Task 14 finished in 6638ms
Task is given to: Thread-3
TASK START AND DATA: Task 17 has execution time of 11047ms
TASK DONE: Task 13 finished in 11520ms
Task is given to: Thread-1
TASK START AND DATA: Task 18 has execution time of 10867ms
TASK DONE: Task 16 finished in 7603ms
Task is given to: Thread-0
TASK START AND DATA: Task 19 has execution time of 10730ms
TASK DONE: Task 15 finished in 9642ms
Task is given to: Thread-2
TASK START AND DATA: Task 20 has execution time of 11741ms
TASK DONE: Task 17 finished in 11047ms
Task is given to: Thread-3
TASK START AND DATA: Task 21 has execution time of 8978ms

------------------------------
Thread pool finished in 45002ms
```

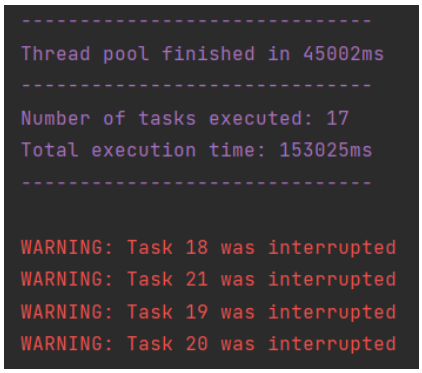
```
------------------------------
Number of tasks executed: 17
Total execution time: 153025ms
------------------------------

WARNING: Task 18 was interrupted
WARNING: Task 21 was interrupted
WARNING: Task 19 was interrupted
WARNING: Task 20 was interrupted
```

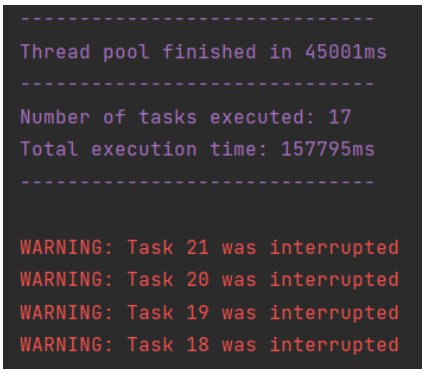
В самому кінці виводу одразу можна побачити вищезгадану коротку статистику по поточній сесії роботу пулу потоків: кількість виконаних завдань, кількість витраченого загалом часу на сесію та кількість часу, скільки працював сам пул.

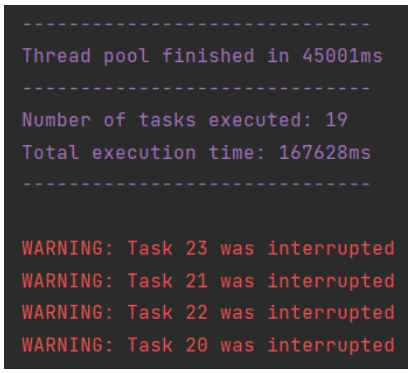
1. «WARNING: Task 18 was interrupted» - приклад виводу-повідомлення про перерване виконаня завдання в потоці в силу того, що ліміт в 45 секунд роботи пулу потоків вже вичерпано. Як можна побачити, в цій ітерації програма перервала виконання завдань 18-21, тобто 4 завдання самостійно було відкинуто пулом потоків, а решта з 9 з 30 завдань були відкинуті ще буфером черги, для них навіть не був згенерований час імітації виконання;
2. «Task is given to: Thread-0» - приклад повідомлення про те, що конкретний потік у пулі потоків отримав собі завдання на виконання, відповідність номеру завдання та часу його виконання можна проаналізувати за подальшим результатом виводу;
3. «THREAD: Task 1 added to thread pool» - приклад повідомлення про те, що завдання було додано до буферу черги, а якщо бути більш точним, то завдання було додано в чергу на генерацію часу виконання та можливу передачу на виконання пулу потоків;

4. «TASK START AND DATA: Task 2 has execution time of 7002ms» - приклад повідомлення про успішно згенерований час виконання для завдання №2 з зазначенням відповідного часу. Також дане повідомлення свідчить про те, що виконання завдання розпочато, тобто якийсь з потоків з пулу потоків працює над ним;
5. «TASK DONE: Task 2 finished in 7003ms» - приклад повідомлення про успішно виконане завдання одним з потоків з пулу потоків, сам по собі потік, що звільнився від цього завдання та вже готовий отримати наступне, зазначається одразу після цього повідомлення: «Task is given to: Thread-3», після даного повідомлення вже повідомляється інформація про нове завдання, передане на виконання даному потоку.

Загальні результати по роботі трьох ітерацій програми представлено на скріншотах нижче. За ними можна побачити, що найбільш ефективно пул потоків виконав свою роботу на третьому повторенні, виконавши загалом 19 завдань. В усіх трьох випадках пул потоків самостійно відкидує виконання чотирьох отриманих завдань (через сплин сесії в 45 секунд), але в перших двох випадках кількість буферно відкинутих завдань складатиме 9, у той час як у третьому випадку на два менше – 7.







Висновок

У ході виконання другої лабораторної роботи були розглянуті базові примітиви синхронізації та їх особливості в залежності від обраної мови програмування Java, розглянуто підходи до побудови програмного забезпечення з використанням паралелізму. Було реалізовано власну версію популярного в програмах з паралелізацією пулу потоків, який дозволяє повторно використовувати створені потоки для роботи над завданнями, без необхідності повторно створювати потік для кожного окремого завдання, що, за умови використання готового рішення (або самостійно написаного рішення) доволі сильно полегшує роботу над створенням та розумінням паралелізованої програми. Для реалізації власного пулу потоків також було написано скорочену версію власної блокованої черги, що дозволяє керувати доступом потоків до черги завдань, які мають бути виконані потоками з пулу потоків. Таким чином, глибоко вивчено логіку відповідних інструментів по роботі з паралелізованими програмами, вивчено роботу з моніторами через використання спеціалізованого слова synchronized, розвинуто загальні навички по роботі з паралелізованими програмами, замірами часу виконання завдань та, звичайно, розвинуто навички по роботі з мовою програмування Java.

Найбільшою складністю по виконанню даної роботи можу зазначити лише відносну складність реалізації завдань (з точки зору новизни мови Java для студентки, що виконувала роботу), наведених у лабораторній, та постійну необхідність в підвищеному рівні уваги до будь-якого написаного слова та символу, оскільки паралелізація виявляється не такою й простою, коли немає вже готових інструментів.

Посилання на гіт: https://github.com/balumatkina/parallel-computing-kpi.git

Додаток А

```
public class CustomThreadPool {
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_PURPLE = "\u001B[35m";

    private final CustomBlockingQueue<Runnable> taskQueue;
    private final Thread[] threads; //working threads array
    // -> workerthread
    private volatile boolean running;

    //statistics
    private volatile int numTasksExecuted;
    private volatile long totalExecutionTime;

    public CustomThreadPool(int numWorkers) {
        this.taskQueue = new CustomBlockingQueue<>();
        this.threads = new Thread[numWorkers];
        this.running = true;
        this.numTasksExecuted = 0;
        this.totalExecutionTime = 0;
        for (int i = 0; i < numWorkers; i++) {
            threads[i] = new WorkerThread();
            threads[i].start();
        }
    }

    public void execute(Runnable task) throws InterruptedException {
        taskQueue.put(task);
    }

    public void taskFinished() {
        synchronized (this) {
            numTasksExecuted++;
        }
    }

    public int getNumTasksExecuted() {
        synchronized (this) {
            return numTasksExecuted;
        }
    }

    public long getTotalExecutionTime() {
        synchronized (this) {
            return totalExecutionTime;
        }
    }

    public void stop() { //to stop the thread pool
        running = false;
        for (Thread thread : threads) {
            thread.interrupt();
        }
        taskQueue.clear();
        for (Thread thread : threads) {
            try {
                thread.join();
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        synchronized (this) {
            numTasksExecuted = 0;
            totalExecutionTime = 0;
        }
    }


    private class WorkerThread extends Thread {
        @Override
        public void run() {
            while (running && !isInterrupted()) {
                try {
                    Runnable task = taskQueue.take();
                    System.out.println("Task is given to: " +
Thread.currentThread().getName());
                    long startTime = System.currentTimeMillis();
                    task.run();
                    long endTime = System.currentTimeMillis();
                    synchronized (CustomThreadPool.this) {
                        totalExecutionTime += endTime - startTime;
                    }
                    taskFinished();
                } catch (InterruptedException e) {
                    interrupt(); // re-interrupting the thread
                }
            }
        }
    }

    public void printStatistics() {
        System.out.println(ANSI_PURPLE + "-".repeat(30));
        System.out.println("Number of tasks executed: " +
getNumTasksExecuted());
        System.out.println("Total execution time: " + getTotalExecutionTime() +
"ms");
        System.out.println("-".repeat(30) + ANSI_RESET);
        System.out.println();
    }
}
```

Додаток Б

```java
import java.util.LinkedList;
import java.util.Queue;

public class CustomBlockingQueue<T> {

    private final Queue<T> queue;

    public CustomBlockingQueue() {
        this.queue = new LinkedList<>();
    }

    public synchronized void put(T element) {
        boolean wasEmpty = queue.isEmpty();
        queue.offer(element);
        if (wasEmpty) {
            notifyAll();
        }
    }

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T element = queue.poll();
        notifyAll();
        return element;
    }

    public synchronized void clear() {
        queue.clear();
        notifyAll();
    }
}
```

Додаток В

```
import java.util.Random;

public class TaskGenerator {
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_RED = "\u001B[31m";

    private final CustomThreadPool threadPool;
    private final Random random;
    private int taskIdCounter;

    public TaskGenerator(CustomThreadPool threadPool) {
        this.threadPool = threadPool;
        this.random = new Random();
        this.taskIdCounter = 1;
    }

    public void addTasks(int numTasks) {
        for (int i = 0; i < numTasks; i++) {
            final int taskId = taskIdCounter++;
            Runnable task = () -> {
                int executeTime = random.nextInt(6000) + 6000;
                System.out.println("TASK START AND DATA: Task " + taskId + " has
execution time of " + executeTime + "ms");

                long startTime = System.currentTimeMillis();
                try {
                    Thread.sleep(executeTime);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt(); // set the interrupted
status
                    System.out.println(ANSI_RED + "WARNING: Task " + taskId + "
was interrupted" + ANSI_RESET);
                    return;
                } catch (Throwable t) {
                    System.out.println(ANSI_RED + "WARNING: Task " + taskId + "
threw an unexpected error: " + t.getMessage() + ANSI_RESET);
                return;
            }
                long endTime = System.currentTimeMillis();

                System.out.println("TASK DONE: Task " + taskId + " finished in "
+ (endTime - startTime) + "ms");
            };
            try {
                threadPool.execute(task);
                System.out.println("THREAD: Task " + taskId + " added to thread
pool");
            } catch (Throwable t) {
                System.err.println(ANSI_RED + "THREAD: Task " + taskId + " threw
an unexpected error when added to thread pool: " + t.getMessage() + ANSI_RESET);
            }
        }
    }
}
```

Додаток Г

```
public class Main {
    public static final String ANSI_RESET = "\u001B[0m";
    public static final String ANSI_RED = "\u001B[31m";
    public static final String ANSI_PURPLE = "\u001B[35m";

    public static final int NUM_THREADS_IN_THREAD_POOL = 4;
    public static final int NUM_TOTAL_TASKS = 30;
    public static final int THREAD_SLEEP_TIME = 45000;
    public static final int NUM_PROGRAM_ITERATION = 3;

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < NUM_PROGRAM_ITERATION; i++) {

            System.out.println();
            System.out.println(ANSI_PURPLE + "Iteration #" + (i + 1) +
ANSI_RESET);
            System.out.println();

            CustomThreadPool threadPool = new
CustomThreadPool(NUM_THREADS_IN_THREAD_POOL);
            TaskGenerator taskGenerator = new TaskGenerator(threadPool);
            taskGenerator.addTasks(NUM_TOTAL_TASKS);

            try {
                long startTime = System.currentTimeMillis();
                Thread.sleep(THREAD_SLEEP_TIME);
                //45 seconds wait before stopping the thread pool
                long endTime = System.currentTimeMillis();
                long elapsedTime = endTime - startTime;

                System.out.println();
                System.out.println(ANSI_PURPLE + "-".repeat(30));
                System.out.println("Thread pool finished in " + elapsedTime +
"ms" + ANSI_RESET);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println(ANSI_RED + "WARNING: Interrupted Thread." +
ANSI_RESET);
            }

            if (Thread.interrupted()) {
                threadPool.stop();
                throw new InterruptedException();
            }

            threadPool.printStatistics();
            threadPool.stop();
        }
    }
}
```