

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ  
 НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ   
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу  
Кафедра системного проектування**

**Звіт**

**про виконання практичної роботи №3  
з дисципліни «Паралельні обчислення»**

Виконав:  
студент III курсу, групи ДА-22  
Жадько Микита Сергійович

Прийняв:

асистент Яременко В. С.

Київ – 2025

**ПОСТАНОВКА ЗАДАЧІ**

1. Ознайомитися з визначенням поняття пул потоків (thread pool), використовуючи даний методичний посібник, або ж сторонні джерела. Ознайомитися з примітивами синхронізації та проблемами, пов’язаними з ними.
2. Реалізувати власний пул потоків з характеристиками, зазначеними в обраному варіанті. Обов’язкові для кожного варіанту характеристики: 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, працювати з використанням умовних змінних)
3. Операції ініціалізації та знищення пулу, додавання та вилучення задач в чергу повинні бути безпечними з точки зору паралелізму.
4. Створити програму, котра буде виконувати задачі за обраним варіантом, використовуючи написаний студентом пул потоків. Код відповідальний за додавання задач в пул потоків, та сам пул потоків повинні знаходитися в різних потоках виконання. Задачі в пул потоків повинні додаватися з декількох потоків.
5. Перевірити та довести коректність роботи програми з використанням системи вводу/виводу інформації в консоль (або іншого засобу – профайлера).
6. Зробити обмежене за часом тестування та розрахувати кількість створених потоків та середній час знаходження потоку в стані очікування. Для задач з необмеженою чергою – визначити середню довжину кожної черги та середній час виконання задач. Для обмежених за розміром черг – визначити максимальний та мінімальний час, поки черга була заповнена, кількість відкинутих задач.
7. В протоколі роботи описати ключові моменти реалізації пулу потоків в цілому, а також ключові моменти при імплементації конкретного варіанту завдання. Описати публічний інтерфейс пулу потоків та механізм його роботи. Описати механізм тестування розробленого рішення. Значення метрик, перечисллений в пункті 6 для різних проміжків часу, обраних в довільному порядку самим студентом.
8. Надати висновок, що повинен містити аналіз отриманих результатів.

Завдання за варіантом (№9)

Пул потоків обслуговується 4-ма робочими потоками й має одну пріоритезовану чергу виконання. Задачі додаються в чергу виконання в порядку «складності» (коротші за часом задачі мають більший пріоритет, в даному варіанті допускається можливість того, що певна задача ніколи не виконається). Задача береться на виконання з буферу одразу за наявності вільного робочого потоку. Задача займає випадковий час від 5 до 10 секунд.

**ХІД РОБОТИ**

Напишемо програму, що вирішить задану задачу. Для генерації тасок використовуватимемо окремі потоки, кількість та період яких регулюватимемо константами. Також симуляція житиме лише певний час, після чого всі потоки будуть змушені припинити роботу.

Код програми:

#include <iostream>

#include <queue>

#include <thread>

#include <vector>

#include <mutex>

#include <condition\_variable>

#include <functional>

#include <chrono>

#include <shared\_mutex>

#include <atomic>

**using** **namespace** std**;**

**using** read\_write\_lock **=** shared\_mutex**;**

**using** read\_lock **=** shared\_lock**<**read\_write\_lock**>;**

**using** write\_lock **=** unique\_lock**<**read\_write\_lock**>;**

atomic**<**bool**>** stop\_program**(false);**

atomic**<**int**>** global\_task\_id**(**0**);**

mutex cout\_mutex**;**

struct PrioritizedTask **{**

int priority**;**

function**<**void**()>** task**;**

bool **operator<(**const PrioritizedTask**&** other**)** const **{**

**return** priority **>** other**.**priority**;**

**}**

**};**

template **<**typename task\_type\_t**>**

class task\_queue **{**

**using** task\_queue\_implementation **=** priority\_queue**<**task\_type\_t**>;**

public**:**

task\_queue**()** **=** **default;**

**~**task\_queue**()** **{** clear**();** **}**

bool empty**()** const **{**

read\_lock lock**(**m\_rw\_lock**);**

**return** m\_tasks**.**empty**();**

**}**

task\_type\_t top**()** const **{**

read\_lock lock**(**m\_rw\_lock**);**

**return** m\_tasks**.**top**();**

**}**

size\_t size**()** const **{**

read\_lock lock**(**m\_rw\_lock**);**

**return** m\_tasks**.**size**();**

**}**

void clear**()** **{**

write\_lock lock**(**m\_rw\_lock**);**

**while** **(!**m\_tasks**.**empty**())** **{**

m\_tasks**.**pop**();**

**}**

**}**

bool pop**(**task\_type\_t**&** task**)** **{**

write\_lock lock**(**m\_rw\_lock**);**

**if** **(**m\_tasks**.**empty**())** **{**

**return** **false;**

**}** **else** **{**

task **=** m\_tasks**.**top**();**

m\_tasks**.**pop**();**

**return** **true;**

**}**

**}**

bool pop**()** **{**

write\_lock lock**(**m\_rw\_lock**);**

**if** **(**m\_tasks**.**empty**())** **{**

**return** **false;**

**}** **else** **{**

m\_tasks**.**pop**();**

**return** **true;**

**}**

**}**

template **<**typename**...** arguments**>**

void emplace**(**arguments**&&...** parameters**)** **{**

write\_lock lock**(**m\_rw\_lock**);**

m\_tasks**.**emplace**(**forward**<**arguments**>(**parameters**)...);**

**}**

private**:**

mutable read\_write\_lock m\_rw\_lock**;**

task\_queue\_implementation m\_tasks**;**

**};**

class thread\_pool **{**

public**:**

thread\_pool**()** **=** **default;**

**~**thread\_pool**()** **{** terminate**();** **}**

void initialize**(**const size\_t worker\_count**)** **{**

write\_lock lock**(**m\_rw\_lock**);**

**if** **(**m\_initialized **||** m\_terminated**)** **{**

**return;**

**}**

m\_workers**.**reserve**(**worker\_count**);**

**for** **(**size\_t id **=** 0**;** id **<** worker\_count**;** **++**id**)** **{**

m\_workers**.**emplace\_back**(&**thread\_pool**::**routine**,** **this);**

**}**

m\_initialized **=** **!**m\_workers**.**empty**();**

**}**

void terminate**()** **{**

**{**

write\_lock lock**(**m\_rw\_lock**);**

**if** **(**working\_unsafe**())** **{**

m\_terminated **=** **true;**

m\_force\_terminate **=** **true;**

**}** **else** **{**

m\_workers**.**clear**();**

m\_terminated **=** **false;**

m\_initialized **=** **false;**

**return;**

**}**

**}**

m\_task\_waiter**.**notify\_all**();**

**for** **(**thread**&** worker **:** m\_workers**)** **{**

**if** **(**worker**.**joinable**())** **{**

worker**.**join**();**

**}**

**}**

m\_workers**.**clear**();**

m\_terminated **=** **false;**

m\_initialized **=** **false;**

**}**

template **<**typename task\_t**,** typename**...** arguments**>**

void add\_task**(**int priority**,** task\_t**&&** task**,** arguments**&&...** parameters**)** **{**

**{**

read\_lock lock**(**m\_rw\_lock**);**

**if** **(!**working\_unsafe**())** **{**

**return;**

**}**

**}**

auto bind **=** std**::**bind**(**forward**<**task\_t**>(**task**),** forward**<**arguments**>(**parameters**)...);**

m\_tasks**.**emplace**(**PrioritizedTask**{**priority**,** bind**});**

m\_task\_waiter**.**notify\_one**();**

**++**total\_tasks\_created**;**

**}**

void pause**()** **{**

write\_lock lock**(**m\_rw\_lock**);**

m\_paused **=** **true;**

**}**

void resume**()** **{**

**{**

write\_lock lock**(**m\_rw\_lock**);**

m\_paused **=** **false;**

**}**

m\_task\_waiter**.**notify\_all**();**

**}**

size\_t get\_total\_tasks\_created**()** const **{** **return** total\_tasks\_created**.**load**();** **}**

size\_t get\_total\_tasks\_completed**()** const **{** **return** total\_tasks\_completed**.**load**();** **}**

double get\_average\_queue\_length**()** const **{**

**return** **static\_cast<**double**>(**total\_queue\_length**.**load**())** **/** total\_tasks\_created**.**load**();**

**}**

double get\_average\_wait\_time**()** const **{**

**return** **static\_cast<**double**>(**total\_wait\_time**.**load**())** **/** **(**total\_tasks\_completed**.**load**()** **\*** 1000**);**

**}**

private**:**

mutable read\_write\_lock m\_rw\_lock**;**

mutable condition\_variable\_any m\_task\_waiter**;**

vector**<**thread**>** m\_workers**;**

task\_queue**<**PrioritizedTask**>** m\_tasks**;**

bool m\_initialized **=** **false;**

bool m\_terminated **=** **false;**

bool m\_paused **=** **false;**

bool m\_force\_terminate **=** **false;**

atomic**<**size\_t**>** total\_tasks\_created**{**0**};**

atomic**<**size\_t**>** total\_tasks\_completed**{**0**};**

atomic**<**size\_t**>** total\_queue\_length**{**0**};**

atomic**<**size\_t**>** total\_wait\_time**{**0**};**

void routine**()** **{**

**while** **(true)** **{**

PrioritizedTask task**;**

**{**

write\_lock lock**(**m\_rw\_lock**);**

auto wait\_condition **=** **[this]** **{**

**return** m\_terminated **||** **(!**m\_tasks**.**empty**()** **&&** **!**m\_paused**);**

**};**

m\_task\_waiter**.**wait**(**lock**,** wait\_condition**);**

**if** **(**m\_force\_terminate**)** **{**

**return;**

**}**

**if** **(!**m\_tasks**.**empty**())** **{**

task **=** m\_tasks**.**top**();**

m\_tasks**.**pop**();**

total\_queue\_length **+=** m\_tasks**.**size**();**

**}**

**}**

**if** **(**task**.**task**)** **{**

auto start\_time **=** chrono**::**steady\_clock**::**now**();**

task**.**task**();**

auto end\_time **=** chrono**::**steady\_clock**::**now**();**

total\_wait\_time **+=** chrono**::**duration\_cast**<**chrono**::**milliseconds**>(**end\_time **-** start\_time**).**count**();**

**++**total\_tasks\_completed**;**

**}**

**if** **(**m\_force\_terminate**)** **{**

**return;**

**}**

**}**

**}**

bool working**()** const **{**

read\_lock lock**(**m\_rw\_lock**);**

**return** working\_unsafe**();**

**}**

bool working\_unsafe**()** const **{**

**return** m\_initialized **&&** **!**m\_terminated**;**

**}**

**};**

void executeTask**(**int taskId**,** int duration**)** **{**

**{**

lock\_guard lock**(**cout\_mutex**);**

cout **<<** "Task " **<<** taskId **<<** " started, duration: " **<<** duration **<<** " seconds.\n"**;**

**}**

this\_thread**::**sleep\_for**(**chrono**::**seconds**(**duration**));**

**{**

lock\_guard lock**(**cout\_mutex**);**

cout **<<** "Task " **<<** taskId **<<** " completed.\n"**;**

**}**

**}**

int getRandomDuration**(**int min**,** int max**)** **{**

**return** min **+** rand**()** **%** **(**max **-** min **+** 1**);**

**}**

int getRandomInterval**(**int min**,** int max**)** **{**

**return** min **+** rand**()** **%** **(**max **-** min **+** 1**);**

**}**

void autoTerminateAfterTime**(**int seconds**)** **{**

this\_thread**::**sleep\_for**(**chrono**::**seconds**(**seconds**));**

stop\_program **=** **true;**

**}**

int main**()** **{**

const int workers\_amount **=** 4**;**

const int min\_task\_time **=** 5**;**

const int max\_task\_time **=** 10**;**

const int generation\_rate\_freq **=** 2**;**

const int num\_generators **=** 4**;**

const int simulation\_duration **=** 20**;**

srand**(static\_cast<**unsigned**>(**time**(nullptr)));**

thread\_pool pool**;**

pool**.**initialize**(**workers\_amount**);**

atomic**<**bool**>** stop\_generation**(false);**

auto generate\_tasks **=** **[&**pool**,** **&**stop\_generation**,** generation\_rate\_freq**]()** **{**

**while** **(!**stop\_generation**)** **{**

int duration **=** getRandomDuration**(**min\_task\_time**,** max\_task\_time**);**

int task\_id **=** global\_task\_id**.**fetch\_add**(**1**);**

pool**.**add\_task**(**duration**,** executeTask**,** task\_id**,** duration**);**

int interval **=** getRandomInterval**(**1**,** 2**)** **\*** generation\_rate\_freq**;**

this\_thread**::**sleep\_for**(**chrono**::**seconds**(**interval**));**

**}**

**};**

vector**<**thread**>** generators**;**

**for** **(**size\_t i **=** 0**;** i **<** num\_generators**;** **++**i**)** **{**

generators**.**emplace\_back**(**generate\_tasks**);**

**}**

thread timer\_thread**(**autoTerminateAfterTime**,** simulation\_duration**);**

**while** **(!**stop\_program**)** **{**

this\_thread**::**sleep\_for**(**chrono**::**milliseconds**(**100**));**

**}**

stop\_generation **=** **true;**

pool**.**terminate**();**

**for** **(**auto**&** generator **:** generators**)** **{**

**if** **(**generator**.**joinable**())** **{**

generator**.**join**();**

**}**

**}**

**if** **(**timer\_thread**.**joinable**())** **{**

timer\_thread**.**join**();**

**}**

cout **<<** endl**;**

cout **<<** "Total tasks created: " **<<** pool**.**get\_total\_tasks\_created**()** **<<** endl**;**

cout **<<** "Total tasks completed: " **<<** pool**.**get\_total\_tasks\_completed**()** **<<** endl**;**

cout **<<** "Average queue length: " **<<** pool**.**get\_average\_queue\_length**()** **<<** endl**;**

cout **<<** "Average task execution time: " **<<** pool**.**get\_average\_wait\_time**()** **<<** " s" **<<**endl**;**

**return** 0**;**

**}**

Подивимося на результат виконання програми:

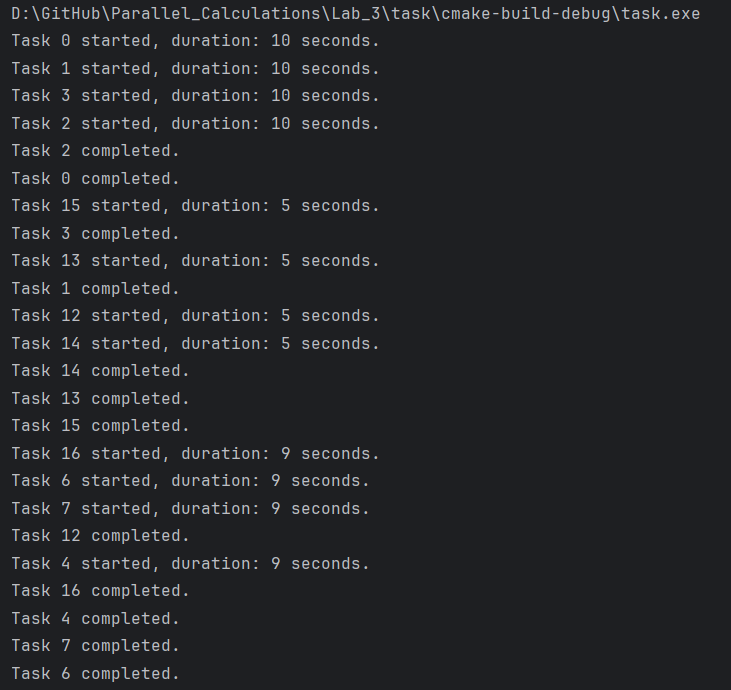


Рисунок 1 — Логи взаємодії потоків між собою.

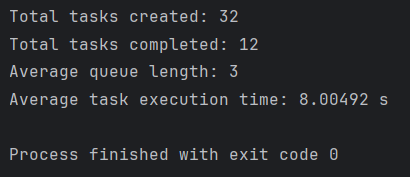


Рисунок 2 — Кінець програми з певною інформацією.

**ВИСНОВКИ**

Було виконано лабораторну роботу, у якій створено програму, що вирішує задану задачу. Використано спільний доступ до пріоритетної черги за допомогою пулу потоків. Для генерації завдань до виконання потоками використано також декілька потоків.

Для кінця програми використано таймер, що завершує всі активні таски через певний період часу після початку роботи програми.

Також [посилання на репозиторій](https://github.com/nikk0308/Parallel_Calculations).