**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  
Моренець Сергій Андрійович

Прийняв:

асистент Яременко В. С.

Київ – 2025

***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. 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.

***Варіант 20:***

Пул потоків обслуговується 2-ма робочими потоками й має одну чергу виконання. Задачі додаються відразу в кінець черги виконання. Черга задач виконується з інтервалом в 30 секунд (буфер наповнюється задачами на протязі 30-ти секунд, котрі потім виконуються). Задачі, що додаються під час виконання черги задач відкидаються. Задача займає випадковий час від 2 до 8 секунд.

***Код програми:***

#include <iostream>

#include <chrono>

#include <vector>

#include <queue>

#include <mutex>

#include <shared\_mutex>

#include <thread>

#include <functional>

#include <atomic>

#include <random>

#include <condition\_variable>

#include <numeric>

#include <algorithm> // DELETE DELETE DELETE DELETE

#include <cassert>   // DELETE DELETE DELETE DELETE

#define THREAD\_COUNT 2

#define EXECUTION\_INTERVAL 30

#define MIN\_TASK\_TIME 2

#define MAX\_TASK\_TIME 8

#define GENERATOR\_THREAD\_COUNT 2

#define GENERATOR\_INTERVAL\_MIN 2

#define GENERATOR\_INTERVAL\_MAX 8

using namespace std;

using namespace chrono;

using read\_lock = shared\_lock<shared\_mutex>;

using write\_lock = unique\_lock<shared\_mutex>;

atomic<size\_t> prev\_id = 0;

shared\_mutex sync\_mtx;

static mutex cout\_mutex;

// Statistic variables

atomic<size\_t> created\_threads\_count = 0;

vector<duration<double>> wait\_times;

shared\_mutex wait\_times\_mutex;

vector<size\_t> queue\_sizes;

mutex queue\_stat\_mutex;

vector<seconds> execution\_times;

mutex execution\_times\_mtx;

double get\_average\_wait\_time()

{

    lock\_guard lock(wait\_times\_mutex);

    if (wait\_times.empty())

    {

        return 0.0;

    }

    duration<double> total\_wait\_time = accumulate(wait\_times.begin(), wait\_times.end(), duration<double>(0));

    return total\_wait\_time.count() / wait\_times.size();

}

double get\_average\_queue\_length()

{

    lock\_guard lock(queue\_stat\_mutex);

    if (queue\_sizes.empty())

    {

        return 0.0;

    }

    size\_t sum = accumulate(queue\_sizes.begin(), queue\_sizes.end(), 0);

    return static\_cast<double>(sum) / queue\_sizes.size();

}

double get\_average\_execution\_time()

{

    lock\_guard lock(execution\_times\_mtx);

    if (execution\_times.empty())

        return 0.0;

    duration<double> total\_wait\_time = accumulate(execution\_times.begin(), execution\_times.end(), duration<double>(0));

    return total\_wait\_time.count() / execution\_times.size();

}

class task

{

public:

    task() : task\_time(random\_time()) {};

    task(size\_t id, seconds time) : task\_id(id), task\_time(time) {}

    ~task() = default;

    size\_t task\_id;

    size\_t get\_id() const { return task\_id; }

    seconds get\_time() const { return task\_time; }

    void execute()

    {

        this\_thread::sleep\_for(task\_time);

        {

            lock\_guard lock(cout\_mutex);

            // cout << get\_id() << " task is executed time: " << get\_time().count() << endl;

        }

    }

private:

    seconds task\_time;

    seconds random\_time()

    {

        static thread\_local mt19937 generator(random\_device{}());

        uniform\_int\_distribution<int> distribution(MIN\_TASK\_TIME, MAX\_TASK\_TIME);

        return seconds(distribution(generator));

    }

};

class task\_queue

{

public:

    task\_queue() = default;

    ~task\_queue() { clear(); };

    bool empty()

    {

        read\_lock \_(rw\_lock);

        return tasks.empty();

    }

    size\_t size()

    {

        read\_lock \_(rw\_lock);

        return tasks.size();

    }

    void clear()

    {

        write\_lock \_(rw\_lock);

        while (!tasks.empty())

        {

            tasks.pop();

        }

        {

            lock\_guard lock(queue\_stat\_mutex);

            queue\_sizes.push\_back(tasks.size());

        }

    }

    bool pop(task &task)

    {

        write\_lock \_(rw\_lock);

        if (tasks.empty())

        {

            return false;

        }

        else

        {

            task = move(tasks.front());

            tasks.pop();

            {

                lock\_guard lock(queue\_stat\_mutex);

                queue\_sizes.push\_back(tasks.size());

            }

            return true;

        }

    }

    void emplace(task &&task)

    {

        write\_lock \_(rw\_lock);

        tasks.emplace(move(task));

        {

            lock\_guard lock(queue\_stat\_mutex);

            queue\_sizes.push\_back(tasks.size());

        }

    }

    void push\_front(task &&new\_task)

    {

        write\_lock \_(rw\_lock);

        queue<task> new\_queue;

        new\_queue.push(move(new\_task));

        while (!tasks.empty())

        {

            new\_queue.push(move(tasks.front()));

            tasks.pop();

        }

        tasks.swap(new\_queue);

        {

            lock\_guard lock(queue\_stat\_mutex);

            queue\_sizes.push\_back(tasks.size());

        }

    }

private:

    queue<task> tasks;

    mutable shared\_mutex rw\_lock;

};

class thread\_pool

{

public:

    thread\_pool() = default;

    ~thread\_pool() { terminate(); };

    bool working()

    {

        read\_lock \_(rw\_lock);

        return working\_unsafe();

    }

    bool working\_unsafe()

    {

        return initialized && !terminated;

    }

    void initialize(size\_t worker\_count)

    {

        write\_lock \_(rw\_lock);

        if (initialized || terminated)

        {

            return;

        }

        workers.reserve(worker\_count);

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

        {

            workers.emplace\_back(routine, this);

            created\_threads\_count++;

        }

        initialized = !workers.empty();

    }

    void routine()

    {

        while (true)

        {

            bool task\_accquired = false;

            task task;

            auto wait\_start = steady\_clock::now();

            {

                write\_lock \_(rw\_lock);

                auto wait\_condition = [this, &task\_accquired, &task]

                {

                    if (execution\_paused)

                    {

                        return false;

                    }

                    task\_accquired = tasks.pop(task);

                    return terminated || task\_accquired;

                };

                task\_waiter.wait(\_, wait\_condition);

            }

            auto wait\_end = steady\_clock::now();

            duration<double> wait\_duration = wait\_end - wait\_start;

            {

                lock\_guard lock(wait\_times\_mutex);

                wait\_times.push\_back(wait\_duration);

            }

            if (terminated && !task\_accquired)

            {

                return;

            }

            seconds exec\_time;

            if (task\_accquired)

            {

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

                task.execute();

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

                exec\_time = duration\_cast<seconds>(end\_time - start\_time);

            }

            if (terminated)

            {

                return;

            }

            {

                write\_lock \_(rw\_lock);

                if (!execution\_paused) // If TP is executing yet

                {

                    lock\_guard lock(cout\_mutex);

                    cout << task.get\_id() << " task is executed time: " << task.get\_time().count() << endl;

                    {

                        lock\_guard lock(execution\_times\_mtx);

                        execution\_times.push\_back(exec\_time);

                    }

                }

                else // didn`t have time to execute

                {

                    tasks.push\_front(move(task));

                }

            }

        }

    }

    void add\_task(task &&task)

    {

        {

            read\_lock \_(rw\_lock);

            if (!working\_unsafe())

            {

                return;

            }

        }

        {

            write\_lock \_(rw\_lock);

            tasks.emplace(move(task));

        }

        if (!execution\_paused)

        {

            task\_waiter.notify\_one();

        }

    }

    void terminate()

    {

        {

            write\_lock \_(rw\_lock);

            if (working\_unsafe())

            {

                terminated = true;

            }

            else

            {

                workers.clear();

                terminated = false;

                initialized = false;

                return;

            }

        }

        task\_waiter.notify\_all();

        for (thread &worker : workers)

        {

            if (worker.joinable())

            {

                worker.join();

            }

        }

        workers.clear();

        terminated = false;

        initialized = false;

    }

    void execution\_pause()

    {

        write\_lock \_(rw\_lock);

        if (working\_unsafe() && !execution\_paused)

        {

            execution\_paused = true;

        }

    }

    void execution\_resume()

    {

        {

            write\_lock \_(rw\_lock);

            if (execution\_paused)

            {

                execution\_paused = false;

            }

        }

        task\_waiter.notify\_all();

    }

private:

    bool initialized = false;

    bool terminated = false;

    bool execution\_paused = false;

    task\_queue tasks;

    vector<thread> workers;

    mutable shared\_mutex rw\_lock;

    mutable condition\_variable\_any task\_waiter;

};

class task\_manager

{

public:

    task\_manager() = default;

    ~task\_manager() { terminate(); };

    void initialize()

    {

        phase = poolPhase::Accumulation;

        tp.initialize(THREAD\_COUNT);

        tp.execution\_pause();

        terminated = false;

        phase\_changer = thread(&task\_manager::chagePhase, this);

        created\_threads\_count++;

        generators.reserve(GENERATOR\_THREAD\_COUNT);

        for (size\_t id = 0; id < GENERATOR\_THREAD\_COUNT; id++)

        {

            generators.emplace\_back(&task\_manager::generate\_and\_try\_to\_insert\_task, this);

            created\_threads\_count++;

        }

    }

    void terminate()

    {

        {

            lock\_guard \_(tm\_mtx);

            terminated = true;

        }

        if (phase\_changer.joinable())

        {

            phase\_changer.join();

        }

        tp.terminate();

        for (thread &gen : generators)

        {

            if (gen.joinable())

            {

                gen.join();

            }

        }

        generators.clear();

    }

private:

    void chagePhase()

    {

        while (!terminated)

        {

            {

                lock\_guard lock(cout\_mutex);

                cout << "Phase: " << (phase == poolPhase::Execution ? "Execution" : "Accumulation") << endl;

            }

            this\_thread::sleep\_for(seconds(EXECUTION\_INTERVAL));

            if(terminated)

            {

                return;

            }

            if (phase == poolPhase::Execution)

            {

                phase = poolPhase::Accumulation;

                tp.execution\_pause();

            }

            else

            {

                phase = poolPhase::Execution;

                tp.execution\_resume();

            }

        }

    }

    void generate\_and\_try\_to\_insert\_task()

    {

        static thread\_local mt19937 randTime(random\_device{}());

        uniform\_int\_distribution<int> distribution(GENERATOR\_INTERVAL\_MIN, GENERATOR\_INTERVAL\_MAX);

        while (!terminated)

        {

            seconds waitTime = seconds(distribution(randTime));

            this\_thread::sleep\_for(seconds(waitTime));

            if (terminated)

            {

                return;

            }

            task task;

            task.task\_id = prev\_id++;

            {

                lock\_guard \_(tm\_mtx);

                if (phase == poolPhase::Accumulation)

                {

                    {

                        tp.add\_task(move(task));

                    }

                    {

                        lock\_guard lock(cout\_mutex);

                        cout << "Add task with id: " << task.get\_id() << ", time: " << task.get\_time().count() << endl;

                    }

                }

            }

        }

    }

    enum class poolPhase

    {

        Accumulation,

        Execution

    };

    poolPhase phase = poolPhase::Accumulation;

    thread phase\_changer;

    vector<thread> generators;

    thread\_pool tp;

    mutable shared\_mutex tm\_mtx;

    bool terminated = false;

};

int main()

{

    task\_manager tm;

    tm.initialize();

    cin.get();

    cout << "Counts of created threads: " << created\_threads\_count.load() << endl;

    cout << "Average threads wait time: " << get\_average\_wait\_time() << " seconds" << endl;

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

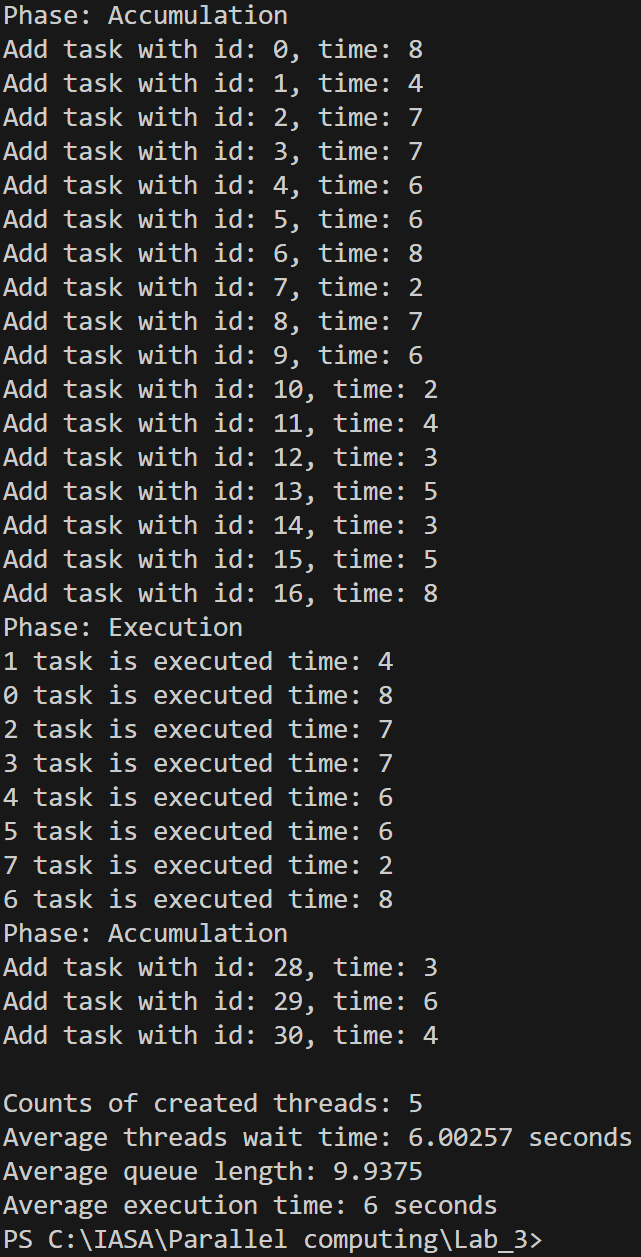
    cout << "Average execution time: " << get\_average\_execution\_time() << " seconds" << endl;

    tm.terminate();

    return 0;

}

Приклад виконання:



***Висновок:***

В ході роботи було реалізовано пул потоків, що при роботі має 2 стани: стан запису задач в чергу та стан виконання задач.

Програму реалізовано, як цикл наповнення та виконання задач за допомогою додаткового класу task\_manager. При ініціалізації цього класу створюється і ініціалізується клас пулу потоків.

Черга в пулі потоків заповнюється за допомогою метода класу task\_manager generate\_and\_try\_to\_insert\_task(), який з випадковою затримкою генерує випадкову задачу та намагається додати її до черги. Цей метод виконуєтся паралельно у визначеній кількоті потоків.

Кількість потоків для заповнення і виконання, тривалість затримки створення задачі і тривалість виконання задачі можна змінювати програмно змінивши константи.

Git: <https://github.com/m0renets/PC_Lab_3.git>