

**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-21  
Терещенко Олексій Ігорович

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**

21. Пул потоків містить дві черги, перша обслуговується 3-ма потоками, а друга – 1-им. Кожна черга виконання має обмежений розмір в 15 задач. Задачі додаються в першу чергу виконання через один інтерфейс (користувач не має явного доступу до черг виконання) в порядку «складності» (коротші за часом задачі мають більший пріоритет), або відкидаються, якщо вони не поміщаються в першу чергу виконання. Якщо задача з першої черги не виконується на протязі подвійного часу, необхідного для її виконання, вона переноситься в другу чергу, або відкидається, якщо вони не поміщаються в другу чергу виконання. Задача береться на виконання з обох черг одразу за наявності відповідного вільного робочого потоку. Задача займає випадковий час від 3 до 14 секунд.

**ХІД РОБОТИ**

Напишемо програму, що вирішить задану задачу.

Код програми:

#include <iostream>

#include <deque>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <chrono>

#include <functional>

#include <random>

#include <atomic>

#include <vector>

#include <limits>

using namespace std;

using namespace std::chrono;

struct Task {

    function<void()> func;

    int expectedDuration;

    steady\_clock::time\_point enqueuedTime;

};

struct QueueStats {

    long long maxFullTimeMs;

    long long minFullTimeMs;

};

class TaskQueue {

public:

    TaskQueue(size\_t *capacity*) : capacity(capacity), currentlyFull(false) {

        maxFullTimeMs = 0;

        minFullTimeMs = numeric\_limits<long long>::max();

    }

    bool push(const Task& *task*) {

        lock\_guard<mutex> lock(mtx);

        if (queue.size() >= capacity) {

            return false;

        }

        auto it = queue.begin();

        while (it != queue.end() && it->expectedDuration <= task.expectedDuration)

            ++it;

        queue.insert(it, task);

        cv.notify\_one();

        if (queue.size() == capacity && !currentlyFull) {

            currentlyFull = true;

            fullStartTime = steady\_clock::now();

        }

        return true;

    }

    bool pop(Task& *task*) {

        unique\_lock<mutex> lock(mtx);

        cv.wait(lock, [this]() { return !queue.empty() || terminate; });

        if (queue.empty())

            return false;

        bool wasFull = currentlyFull;

        task = queue.front();

        queue.pop\_front();

        if (wasFull && queue.size() < capacity) {

            currentlyFull = false;

            auto fullDuration = duration\_cast<milliseconds>(steady\_clock::now() - fullStartTime).count();

            if (fullDuration > maxFullTimeMs)

                maxFullTimeMs = fullDuration;

            if (fullDuration < minFullTimeMs)

                minFullTimeMs = fullDuration;

        }

        return true;

    }

    vector<Task> getTasksSnapshot() {

        lock\_guard<mutex> lock(mtx);

        return vector<Task>(queue.begin(), queue.end());

    }

    bool removeTask(const Task& *task*) {

        lock\_guard<mutex> lock(mtx);

        for (auto it = queue.begin(); it != queue.end(); ++it) {

            if (it->enqueuedTime == task.enqueuedTime && it->expectedDuration == task.expectedDuration) {

                queue.erase(it);

                if (currentlyFull && queue.size() < capacity) {

                    currentlyFull = false;

                    auto fullDuration = duration\_cast<milliseconds>(steady\_clock::now() - fullStartTime).count();

                    if (fullDuration > maxFullTimeMs)

                        maxFullTimeMs = fullDuration;

                    if (fullDuration < minFullTimeMs)

                        minFullTimeMs = fullDuration;

                }

                return true;

            }

        }

        return false;

    }

    void setTerminateFlag(bool *term*) {

        lock\_guard<mutex> lock(mtx);

        terminate = term;

        cv.notify\_all();

    }

    QueueStats getStats() {

        lock\_guard<mutex> lock(mtx);

        QueueStats stats;

        stats.maxFullTimeMs = maxFullTimeMs;

        stats.minFullTimeMs = (minFullTimeMs == numeric\_limits<long long>::max() ? 0 : minFullTimeMs);

        return stats;

    }

private:

    deque<Task> queue;

    size\_t capacity;

    mutex mtx;

    condition\_variable cv;

    bool terminate = false;

    bool currentlyFull;

    steady\_clock::time\_point fullStartTime;

    long long maxFullTimeMs;

    long long minFullTimeMs;

};

class ThreadPoolVariant21 {

public:

    ThreadPoolVariant21() : primaryQueue(15), secondaryQueue(15), stopFlag(false),

        tasksAddedPrimary(0), tasksDiscardedPrimary(0), tasksTransferredToSecondary(0),

        tasksExecutedPrimary(0), tasksExecutedSecondary(0), tasksDiscardedSecondary(0),

        totalTaskWaitTimeMs(0), tasksCountForWaitTime(0), totalWorkerWaitTimeMs(0), workerWaitCount(0)

    {

    }

    void start() {

        for (int i = 0; i < 3; i++) {

            primaryWorkers.emplace\_back(&ThreadPoolVariant21::primaryWorker, this, i);

        }

        secondaryWorker = thread(&ThreadPoolVariant21::secondaryWorkerFunc, this);

        monitorThread = thread(&ThreadPoolVariant21::monitorFunction, this);

    }

    void stop() {

        stopFlag = true;

        primaryQueue.setTerminateFlag(true);

        secondaryQueue.setTerminateFlag(true);

        for (auto& t : primaryWorkers) {

            if (t.joinable())

                t.join();

        }

        if (secondaryWorker.joinable())

            secondaryWorker.join();

        if (monitorThread.joinable())

            monitorThread.join();

    }

    bool addTask(int *expectedDuration*, function<void()> *func*) {

        Task task;

        task.func = func;

        task.expectedDuration = expectedDuration;

        task.enqueuedTime = steady\_clock::now();

        tasksAddedPrimary++;

        bool pushed = primaryQueue.push(task);

        if (!pushed) {

            cout << "[AddTask] Задачу відхилено: первинна черга переповнена." << endl;

            tasksDiscardedPrimary++;

        }

        return pushed;

    }

    void printStatistics() {

        cout << "\n==== Статистика виконання ====\n";

        cout << "Загальна кількість задач, доданих до первинної черги: " << tasksAddedPrimary.load() << "\n";

        cout << "Задач відхилено при додаванні (черга первинна): " << tasksDiscardedPrimary.load() << "\n";

        cout << "Задач виконано первинними потоками: " << tasksExecutedPrimary.load() << "\n";

        cout << "Задач виконано вторинним потоком: " << tasksExecutedSecondary.load() << "\n";

        cout << "Задач перенесено до вторинної черги: " << tasksTransferredToSecondary.load() << "\n";

        cout << "Задач відхилено при перенесенні (черга вторинна): " << tasksDiscardedSecondary.load() << "\n";

        if (tasksCountForWaitTime.load() > 0) {

            cout << "Середній час очікування задач (від моменту додавання до початку виконання): "

                << totalTaskWaitTimeMs.load() / tasksCountForWaitTime.load() << " мс\n";

        }

        else {

            cout << "Середній час очікування задач: N/A\n";

        }

        if (workerWaitCount.load() > 0) {

            cout << "Середній час очікування потоку (в pop()): "

                << totalWorkerWaitTimeMs.load() / workerWaitCount.load() << " мс\n";

        }

        else {

            cout << "Середній час очікування потоку: N/A\n";

        }

        QueueStats primaryStats = primaryQueue.getStats();

        QueueStats secondaryStats = secondaryQueue.getStats();

        cout << "Первинна черга - максимальний час заповнення: " << primaryStats.maxFullTimeMs

            << " мс, мінімальний час заповнення: " << primaryStats.minFullTimeMs << " мс\n";

        cout << "Вторинна черга - максимальний час заповнення: " << secondaryStats.maxFullTimeMs

            << " мс, мінімальний час заповнення: " << secondaryStats.minFullTimeMs << " мс\n";

        cout << "Кількість створених робочих потоків: " << (int)(primaryWorkers.size() + 1) << "\n"; *// 3 + 1*

        cout << "Кількість інших потоків (монітор, продюсер): 2\n";

        cout << "================================\n";

    }

private:

    TaskQueue primaryQueue;

    TaskQueue secondaryQueue;

    vector<thread> primaryWorkers;

    thread secondaryWorker;

    thread monitorThread;

    atomic<bool> stopFlag;

    atomic<int> tasksAddedPrimary;

    atomic<int> tasksDiscardedPrimary;

    atomic<int> tasksTransferredToSecondary;

    atomic<int> tasksExecutedPrimary;

    atomic<int> tasksExecutedSecondary;

    atomic<int> tasksDiscardedSecondary;

    atomic<long long> totalTaskWaitTimeMs;

    atomic<int> tasksCountForWaitTime;

    atomic<long long> totalWorkerWaitTimeMs;

    atomic<int> workerWaitCount;

    void primaryWorker(int *workerId*) {

        while (!stopFlag) {

            Task task;

            auto waitStart = steady\_clock::now();

            if (primaryQueue.pop(task)) {

                auto waitEnd = steady\_clock::now();

                long long workerWait = duration\_cast<milliseconds>(waitEnd - waitStart).count();

                totalWorkerWaitTimeMs += workerWait;

                workerWaitCount++;

                long long taskWait = duration\_cast<milliseconds>(waitEnd - task.enqueuedTime).count();

                totalTaskWaitTimeMs += taskWait;

                tasksCountForWaitTime++;

                cout << "[Первинний поток " << workerId << "] Початок виконання задачі (очікуваний час "

                    << task.expectedDuration << " сек, час очікування " << taskWait << " мс)." << endl;

                task.func();

                tasksExecutedPrimary++;

                cout << "[Первинний поток " << workerId << "] Завершено виконання задачі." << endl;

            }

        }

    }

    void secondaryWorkerFunc() {

        while (!stopFlag) {

            Task task;

            auto waitStart = steady\_clock::now();

            if (secondaryQueue.pop(task)) {

                auto waitEnd = steady\_clock::now();

                long long workerWait = duration\_cast<milliseconds>(waitEnd - waitStart).count();

                totalWorkerWaitTimeMs += workerWait;

                workerWaitCount++;

                long long taskWait = duration\_cast<milliseconds>(waitEnd - task.enqueuedTime).count();

                totalTaskWaitTimeMs += taskWait;

                tasksCountForWaitTime++;

                cout << "[Вторинний поток] Початок виконання задачі (очікуваний час "

                    << task.expectedDuration << " сек, час очікування " << taskWait << " мс)." << endl;

                task.func();

                tasksExecutedSecondary++;

                cout << "[Вторинний поток] Завершено виконання задачі." << endl;

            }

        }

    }

    void monitorFunction() {

        while (!stopFlag) {

            auto tasks = primaryQueue.getTasksSnapshot();

            for (auto& task : tasks) {

                auto waited = duration\_cast<seconds>(steady\_clock::now() - task.enqueuedTime).count();

                if (waited > 2 \* task.expectedDuration) {

                    bool removed = primaryQueue.removeTask(task);

                    if (removed) {

                        bool pushed = secondaryQueue.push(task);

                        if (pushed) {

                            tasksTransferredToSecondary++;

                            cout << "[Монітор] Перенесено задачу (очікуваний час "

                                << task.expectedDuration << " сек) з первинної до вторинної черги, "

                                << "час очікування: " << waited << " сек." << endl;

                        }

                        else {

                            tasksDiscardedSecondary++;

                            cout << "[Монітор] Вторинна черга переповнена. Задача відкинута." << endl;

                        }

                    }

                }

            }

            this\_thread::sleep\_for(seconds(1));

        }

    }

};

void simulatedTask(int *execTime*) {

    cout << "    [Task] Виконується задача на " << execTime << " сек." << endl;

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

}

int main() {

    ThreadPoolVariant21 pool;

    pool.start();

    thread taskProducer([&*pool*]() {

        random\_device rd;

        mt19937 gen(rd());

        uniform\_int\_distribution<> distr(3, 14);

        for (int i = 0; i < 50; i++) {

            int execTime = distr(gen);

            pool.addTask(execTime, [*execTime*]() {

                simulatedTask(execTime);

                });

            this\_thread::sleep\_for(seconds(1));

        }

        });

    taskProducer.join();

    this\_thread::sleep\_for(seconds(120));

    pool.stop();

    pool.printStatistics();

    return 0;

}

Подивимося на результат виконання програми:

A screenshot of a computer program

AI-generated content may be incorrect.

Рисунок 1 — Початок роботи програми.

A screenshot of a computer

AI-generated content may be incorrect.

Рисунок 2 — Статистичні дані.

**ВИСНОВКИ**

В результаті проведених експериментів можна зробити висновок, що розроблений пул потоків відповідає вимогам лабораторної роботи. Реалізовано два рівні черг із обмеженим розміром, де первинну чергу обслуговують три робочих потоки, а вторинну – один. Механізм перенесення задач при перевищенні критерію часу очікування працює коректно, що підтверджується відсутністю відкинутих задач. Проте, через обмежений час тестування, декілька задач могли залишитися невиконаними.

Статистичні дані також свідчать про належну роботу синхронізаційних примітивів (м’ютексів та умовних змінних), що забезпечує коректність паралельного виконання задач. Таким чином, розроблений пул демонструє правильне функціонування у контексті розгляду примітивів синхронізації та принципів роботи пулу потоків, що є ключовим аспектом для побудови ефективного багатопотокового програмного забезпечення.