**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**

**імені Ігоря Сікорського»**

**Лабораторна робота № 3**

**з курсу «Паралельні обчислення»**

**на тему «Дослідження базових примітивів синхронізації»**

**Виконав студент ІПСА групи ДА-22**

**Котляр Єгор Ярославович**

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 14**: Пул потоків обслуговується 6-ма робочими потоками й має одну чергу виконання. Черга виконання має обмежений розмір в 15 задач. Задачі додаються одразу в кінець черги виконання, або відкидаються, якщо вони не поміщаються в чергу виконання. Задача береться на виконання з буферу одразу за наявності вільного робочого потоку. Задача займає випадковий час від 5 до 10 секунд.

Посилання на [репозиторій](https://github.com/Kott1/PC).

**Теоретичні відомості**

**Пул потоків** – це набір заздалегідь створених потоків-«робітників», які отримують із загальної черги завдань функціональні об’єкти й виконують їх.

Примітиви синхронізації:

**М’ютекс** – забезпечує взаємовиключний доступ одного потоку до критичної секції коду.

Інтерфейс:

lock() – захоплення м’ютекса; якщо м’ютекс вже захоплено іншим потоком, потік-блокувальник чекає.

unlock() – звільнення м’ютекса; дозволяє одному з очікуючих потоків увійти в критичну секцію.

**Семафор** – обмежує одночасну кількість потоків, які можуть виконувати критичну секцію, до певного числа N.

Інтерфейс (зазвичай):

wait() або acquire() – зменшення внутрішнього лічильника; якщо лічильник ≤ 0, потік чекає.

signal() або release() – збільшення лічильника; будить один із очікуючих потоків.

**Умовна змінна (Condition Variable)** – організовує очікування до настання деякої умови, що визначена користувачем.

Інтерфейс:

wait(mutex) – атомарно звільняє м’ютекс і блокує потік до виклику notify.

notify\_one() / notify\_all() – розблоковує один або всі потоки, що очікували на цій змінній.

**Спін-лок** – реалізується за допомогою циклу «очікування за спіном»: потік у циклі перевіряє стан замка, не віддаючи керування планувальнику.

Підходить для дуже коротких критичних секцій, щоб уникнути витрат на контекстні переключення.

**Read–Write Lock (Shared–Unique Lock)** – дозволяє одночасний доступ для читачів (shared), але виключний доступ для писачів (unique).

Проблеми, пов’язані з примітивами синхронізації:

**Deadlock (взаємне блокування)** – ситуація, коли два або більше потоків чекають захоплення примітива, утворюючи цикл залежностей.

**Livelock** – потоки постійно змінюють свій стан у відповідь одне одному, але прогрес не досягається.

**Starvation (голодування)** – потік може ніколи не отримати доступ до ресурсу через пріоритет інших.

**Priority inversion** – низькопріоритетний потік утримує ресурс, необхідний високопріоритетному, внаслідок чого останній «засинає».

**False sharing** – конкуренція не на логічному ресурсі, а на кеш-лініях процесора, призводить до зниження продуктивності.

**Хід виконання роботи**

За варіантом створимо програму, де буде реалізовано власний пул потоків:

**Код програми:**

#include <iostream>  
#include <queue>  
#include <thread>  
#include <vector>  
#include <shared\_mutex>  
#include <mutex>  
#include <condition\_variable>  
#include <functional>  
#include <chrono>  
#include <atomic>  
#include <random>  
#include <windows.h>  
  
using namespace std;  
using Clock = chrono::steady\_clock;  
using read\_write\_lock = shared\_mutex;  
using read\_lock = shared\_lock<read\_write\_lock>;  
using write\_lock = unique\_lock<read\_write\_lock>;  
  
template<typename F>  
class task\_queue {  
 queue<F> tasks;  
 mutable read\_write\_lock rw;  
public:  
 bool empty() const {  
 read\_lock lk(rw);  
 return tasks.empty();  
 }  
  
 size\_t size() const {  
 read\_lock lk(rw);  
 return tasks.size();  
 }  
  
 void clear() {  
 write\_lock lk(rw);  
 while (!tasks.empty()) tasks.pop();  
 }  
  
 bool pop(F &f) {  
 write\_lock lk(rw);  
 if (tasks.empty()) return false;  
 f = move(tasks.front()); tasks.pop();  
 return true;  
 }  
  
 bool push(F&& f, size\_t cap) {  
 write\_lock lk(rw);  
 if (tasks.size() >= cap) return false;  
 tasks.push(move(f));  
 return true;  
 }  
  
 F top() const {  
 read\_lock lk(rw);  
 if (tasks.empty()) throw std::runtime\_error("Черга пуста");  
 return tasks.top();  
 }  
};  
  
class thread\_pool {  
public:  
 thread\_pool(size\_t workers = 6, size\_t capacity = 15): WORKERS(workers), CAP(capacity) {  
 for(size\_t i = 0; i < WORKERS; ++i)  
 workers\_vec.emplace\_back(&thread\_pool::worker, this);  
 created = WORKERS;  
 }  
 ~thread\_pool() { shutdown(); }  
  
 void addTask(function<void()> f) {  
 attempted++;  
  
 if (!q.push(move(f), CAP)) {  
 rejected++;  
 return;  
 }  
  
 accepted++;  
  
 auto now = Clock::*now*();  
 if (q.size() == CAP) {  
 write\_lock lg(metrics\_mtx);  
 if (!is\_full) {  
 is\_full = true;  
 full\_start = now;  
 }  
 }  
  
 cv.notify\_all();  
 }  
  
 void shutdown() {  
 stop = true;  
 cv.notify\_all();  
 for(auto &t : workers\_vec) {  
 if (t.joinable()) t.join();  
 }  
 }  
  
 void show\_metrics() const {  
 double avgWait = waitCycles ? (totalWaitNs / static\_cast<double>(waitCycles) / 1e9) : 0;  
  
 vector<double> durations\_copy;  
 {  
 read\_lock lg(metrics\_mtx);  
 durations\_copy = full\_durations;  
 }  
  
 double minFull = 0, maxFull = 0;  
 if (!durations\_copy.empty()) {  
 minFull = \*min\_element(durations\_copy.begin(), durations\_copy.end());  
 maxFull = \*max\_element(durations\_copy.begin(), durations\_copy.end());  
 }  
  
 cout << "Кількість робочих потоків: " << created << "\n";  
 cout << "Спроб додати задач: " << attempted << "\n";  
 cout << "Завершено задач: " << completed << "\n";  
 cout << "Відкинуто задач: " << rejected << "\n";  
 cout << "Середній час простою потоків (с): " << avgWait << "\n";  
 cout << "Найкоротший час, коли черга була повністю заповнена (с): " << minFull << "\n";  
 cout << "Найдовший час, коли черга була повністю заповнена (с): " << maxFull << "\n";  
 }  
  
private:  
 void worker() {  
 while (true) {  
 function<void()> task;  
 {  
 unique\_lock<mutex> lk(cv\_m);  
 auto start\_wait = Clock::*now*();  
 cv.wait(lk, [&]{ return stop || !q.empty(); });  
 auto end\_wait = Clock::*now*();  
 totalWaitNs += chrono::duration\_cast<chrono::nanoseconds>(end\_wait - start\_wait).count();  
 waitCycles++;  
 }  
 if (stop && q.empty()) break;  
  
 if (q.pop(task)) {  
 bool record = false;  
 Clock::time\_point start\_tp;  
 {  
 write\_lock lg(metrics\_mtx);  
 if (is\_full) {  
 is\_full = false;  
 start\_tp = full\_start;  
 record = true;  
 }  
 }  
 if (record) {  
 auto now2 = Clock::*now*();  
 double dt = chrono::duration<double>(now2 - start\_tp).count();  
 write\_lock lg(metrics\_mtx);  
 full\_durations.push\_back(dt);  
 }  
  
 completed++;  
 task();  
 }  
 }  
 }  
  
 const size\_t WORKERS;  
 const size\_t CAP;  
 task\_queue<function<void()>> q;  
 vector<thread> workers\_vec;  
  
 mutex cv\_m;  
 condition\_variable\_any cv;  
 atomic<bool> stop{false};  
  
 size\_t created{0};  
 atomic<size\_t> attempted{0}, accepted{0}, completed{0}, rejected{0};  
  
 mutable read\_write\_lock metrics\_mtx;  
 bool is\_full{false};  
 Clock::time\_point full\_start;  
 vector<double> full\_durations;  
  
 atomic<uint64\_t> totalWaitNs{0}, waitCycles{0};  
};  
  
int main() {  
 SetConsoleOutputCP(CP\_UTF8);  
  
 const int PRODUCERS = 5;  
 const int TASKS\_PER\_PRODUCER = 10;  
 mutex cout\_mtx;  
  
 thread\_pool pool(6, 15);  
  
 vector<thread> producers;  
 for(int p = 0; p < PRODUCERS; ++p) {  
 producers.emplace\_back([&, p](){  
 mt19937 rng(random\_device{}());  
 uniform\_int\_distribution<int> dist(5, 10);  
 for(int i = 0; i < TASKS\_PER\_PRODUCER; ++i) {  
 int id = p \* TASKS\_PER\_PRODUCER + i + 1;  
 int dur = dist(rng);  
 pool.addTask([id, dur, &cout\_mtx](){  
 { lock\_guard<mutex> lg(cout\_mtx);  
 cout << "Завдання #" << id << " виконується " << dur << " сек\n"; }  
 this\_thread::sleep\_for(chrono::seconds(dur));  
 { lock\_guard<mutex> lg(cout\_mtx);  
 cout << "Завдання #" << id << " завершено\n"; }  
 });  
 this\_thread::sleep\_for(chrono::milliseconds(500));  
 }  
 });  
 }  
 for(auto &t : producers) t.join();  
  
 pool.shutdown();  
 pool.show\_metrics();  
 return 0;  
}

Вхідні дані – це кількість виробників та їх завдань. Нехай буде 5 робітників, які мають по 10 завдань. Завдання виконується на проміжку від 5 до 10 секунд, шістьма потоками (робітниками), але не всі завдання будуть виконанні. Черга обмежена розміром – 15, тому ті завдання, що не попали до черги будуть відкинуті:

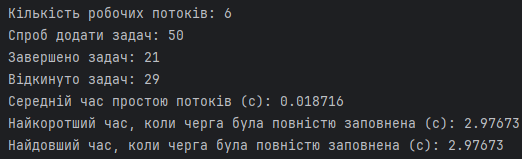


Рисунок 1 – Результат програми.

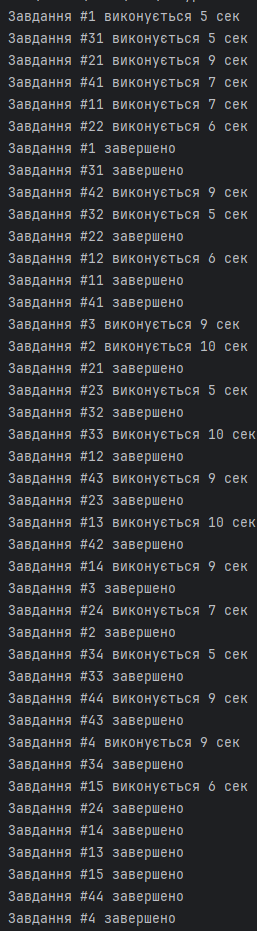


Рисунок 2 – Виконання програми.

У реалізації пулу потоків публічний інтерфейс складається з:

* Конструктора thread\_pool(size\_t workers, size\_t capacity), який створює фіксовану кількість робочих потоків і чергу з обмеженою місткістю.
* Методу addTask(function<void()>), що додає завдання в кінець черги або відкидає його, якщо черга повна.
* Методу shutdown(), який коректно зупиняє всі потоки.
* Методу show\_metrics(), що виводить у консоль статистику виконання (кількість створених потоків, спроб додати задачі, успішно виконаних і відкинутих, середній час простою потоків і діапазон інтервалів, коли черга була заповнена).

Після створення пулу кожен робочий потік заходить у функцію worker(), де в циклі чекає на появу завдань або сигнал зупинки. Перед входом у стан очікування й одразу після пробудження вимірюється час простою, який накопичується для обчислення середнього. Коли завдання з’являється, потік витягає його з черги (за принципом FIFO), збільшує лічильник завершених задач і виконує функціональний об’єкт. Якщо під час вибірки завдання черга тільки що була заповнена до межі, то фіксується інтервал «повного» стану черги.

Тестування ж проводилося за участю п’яти виробників, кожен із яких додавав у пул по десять задач із випадковою тривалістю виконання від 5 до 10 секунд. Загалом було зроблено 50 спроб додати задачі.

**Висновок**

У висновку можна зазначити, що під час помірного навантаження пул обробляє всі задачі без відмов і з дуже низьким простоєм потоків, проте при піковому додаванні задач із швидкістю, вищою за швидкість обробки, значна частина завдань втрачається через обмежену місткість черги. Це свідчить про необхідність або збільшення числа робочих потоків, або використання динамічного масштабування, щоб зменшити кількість відкинутих завдань у пікові моменти. Ще одним шляхом підвищення надійності може стати реалізація повторного додавання відкинутих задач або введення пріоритетів для критичних операцій.