**КПІ ім. Ігоря Сікорського**

**Інститут прикладного системного аналізу**

**Кафедра Системного проектування**

Лабороторна робота - 1

з дисципліни «Паралельні обчислення»

Виконав:

Студент групи ДА-12

Коваленко Олександр

Київ – 2024

**Лабораторна робота -2**

**Дослідження базових примітивів синхронізації**

**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

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: 6** Пул потоків містить дві черги, кожна з яких обслуговується 2-ма

робочими потоками. Задачі в чергу виконання додаються через один

інтерфейс (користувач не має явного доступу до черг виконання). Задачі

додаються відразу в кінець найменш завантаженої по часу (черга з

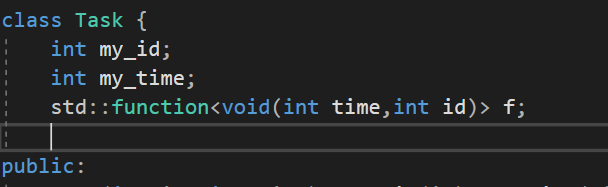
задачами, чий сумарний час виконання менший) черги виконання. Задача

займає випадковий час від 2 до 15 секунд.

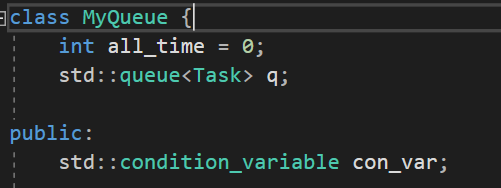
**Хід роботи**

Для виконання лабороторної роботибуло створено 3 класи: Task,MyQueue та ThreadPool.

В класі Task зберігається айді, час виконання та функція, яку необхідно виконати.



В класі MyQueue зберігається std::queue<Task> - стандартна черга, яка зберігає об’єкти типа Task; ще зберігається зміна all\_time, в якій записано час, який необхідний для виконнання всіх задач з черги. Крім того в черзі зберігається std::condition\_variable con\_var – вона необхідна для блокування(потоки будуть спати, не буде простою) потоків, коли в черзі нема задач і розблокування , коли задача в черзі з’являється.



В класі ThreadPool рядок

Рядок std::vector<std::thread> producer\_threads; - це масив потоків, які відповідають за створення задач та додавання в необхідну чергу;

Рядок std::vector<std::thread> consumer\_threads; - це масив потоків, які відповідають за виконання задач з черги за яку вони відповідають;

Рядок int created\_tasks = 0; - це зміна де зберігається айді задачі яка буде створенна, а також кількість створенніх задач – 1;

Рядки int num\_producers; int num\_consumers; - це відповідно кількість потоків, які додають задачі до черги і потоки, які забирають задачі з черги;

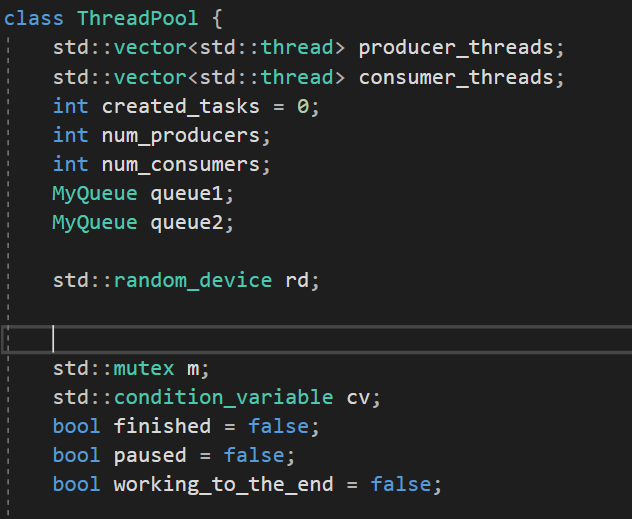
Рядки MyQueue queue1; MyQueue queue2; - це відповідно перша та друга черга з задачами;

Рядок std::random\_device rd; - він необхідний для створення випадкового часу виконання задач;

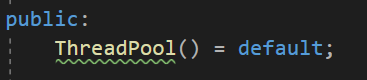
Рядок std::mutex m – необхіден для блокування доступу до спільної пам’яті (черг);

Рядок std::condition\_variable cv; - необхіден для блокування потоків, які створюють задачі;

Рядки bool finished = false; bool paused = false; bool working\_to\_the\_end = false; - необхідні для того, щоб знати стан ThreadPool, коли всі false програма працює в звичайному режимі, коли paused = true; прогрма на паузі, bool finished = true; програма завершує всі задачі, які вже почала робити і закінчує роботу(consumers),а producers – перестають додавати нові завданя в чергу. Коли working\_to\_the\_end = true; producers перестають додавати нові завдання, а consumers працюють до моменту закінчення задач в черзі.

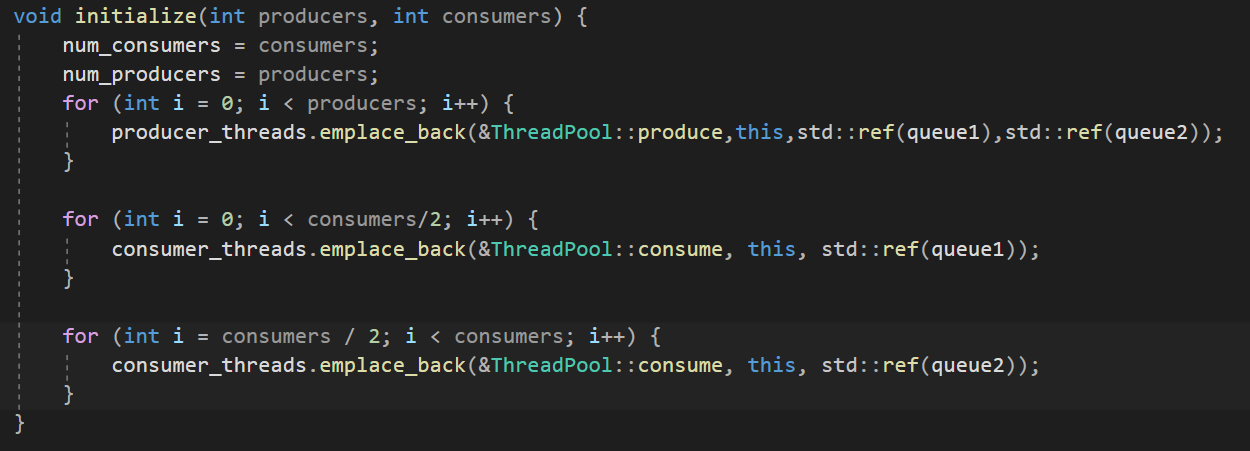


При створенні пулу потоків визивається дефолтний конструктор.

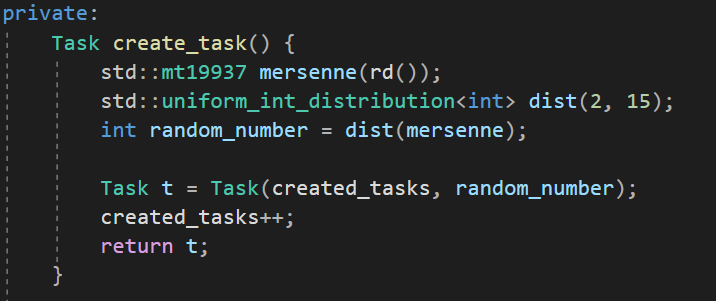


Для того, щоб пул потоків почав працювати треба визвати функцію ініціалізації. Для цього необхідно в функцію передати кількість потоків, які будуть додавати завдання в чергу, та кількість потоків, які будуть виконувати задачі відповідно producers та consumers.

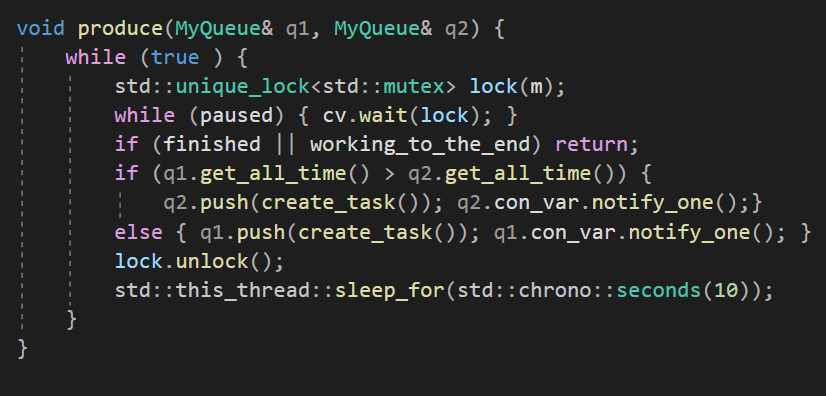
В циклі for додаємо потоки до масива потоків, для producers передаємо обидві черги, а для consumers тільки чергу за яку вони відповідають.



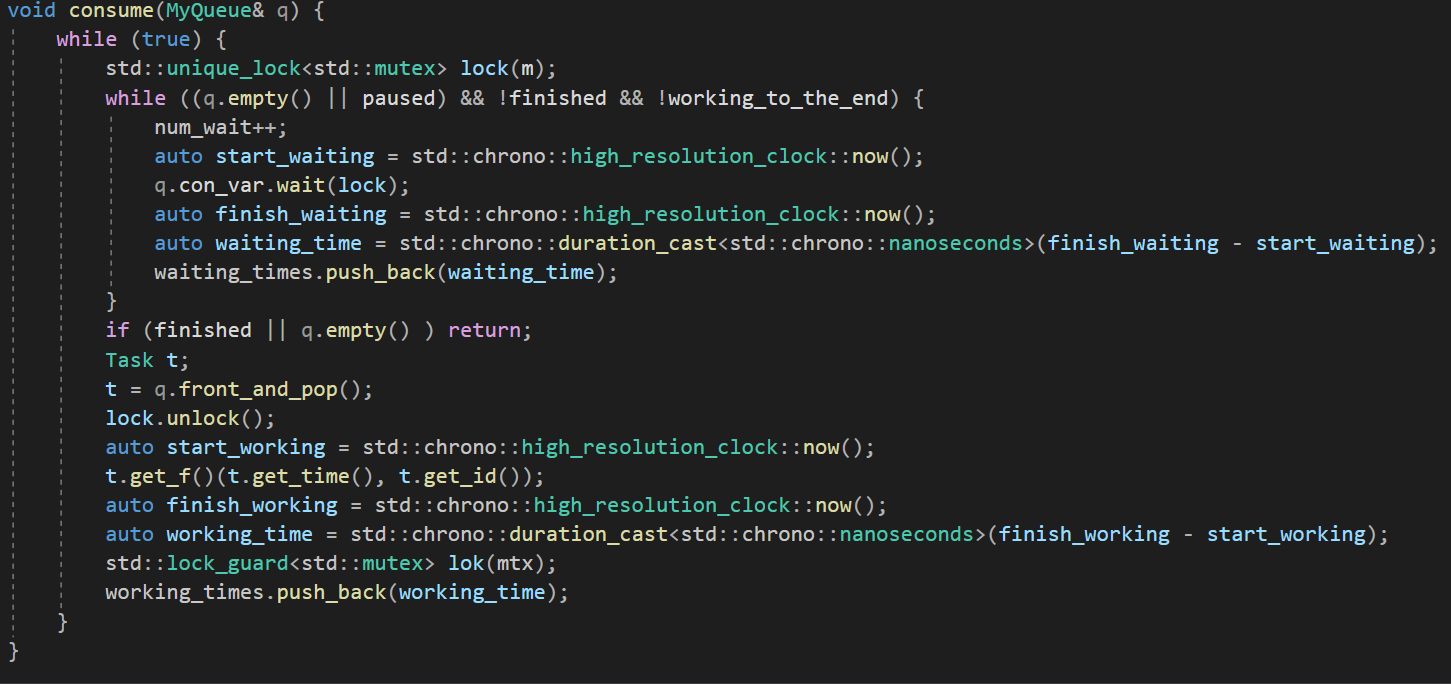
В функції create\_task() створюємо задачу, за допомогою Task t = Task(created\_tasks, random\_number); - перший аргумент це айді задачі, а другий це випадкове число в промужку між 2 та 15.



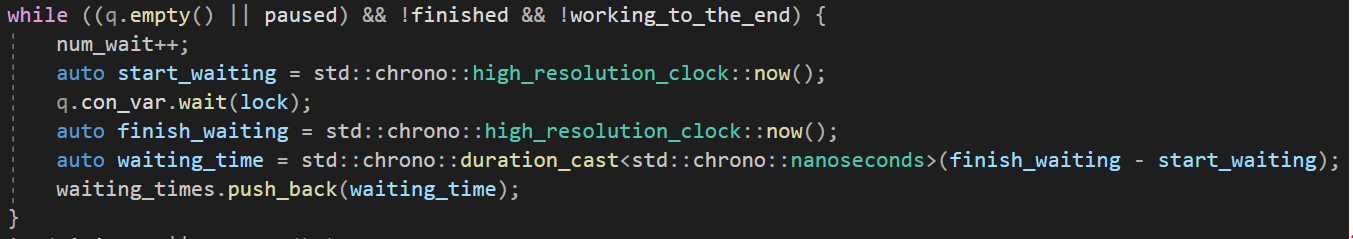
В функції produce() – створюємо задачу за допомогою create\_task() та заносимо ії в необхідну чергу, в залежності від того де час виконання всіх задач із черги менший. Якщо paused = true, то потоки, які виконують цю функцію встануть на паузу(заснуть/ блокуються). Якщо finished = true або working\_to\_the\_end = true, то потоки завершують свою роботу.



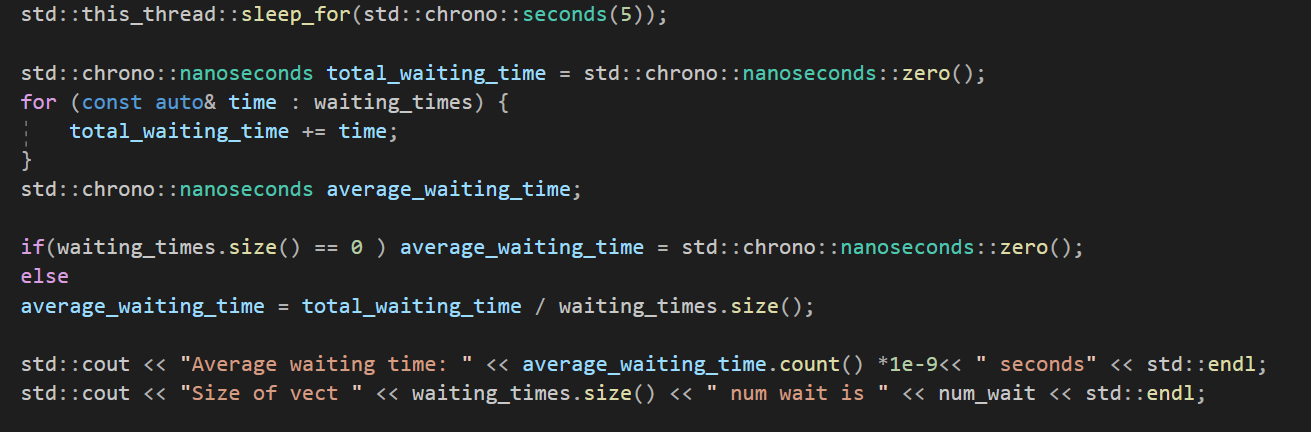
В функції consume береться задача з певної черги та виконуються. Якщо черга пуста або paused = true то потік зупиняє свою роботу. Якщо finished = true, то потік закінчує свою роботу.



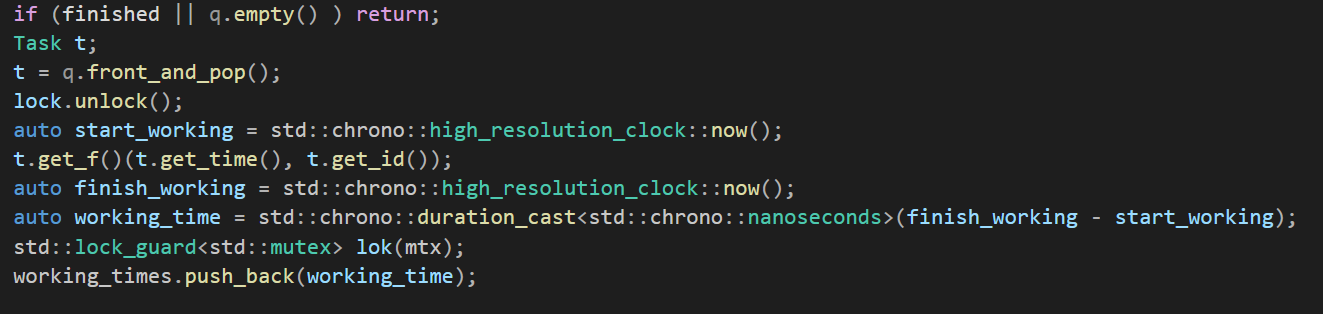
Num\_wait – це число скільки раз потоки були в стані очікування. Auto start\_waiting та інші необхідно щоб порахувати час, коли потоки чекали, після цього цей час заносить до масиву waiting\_times.



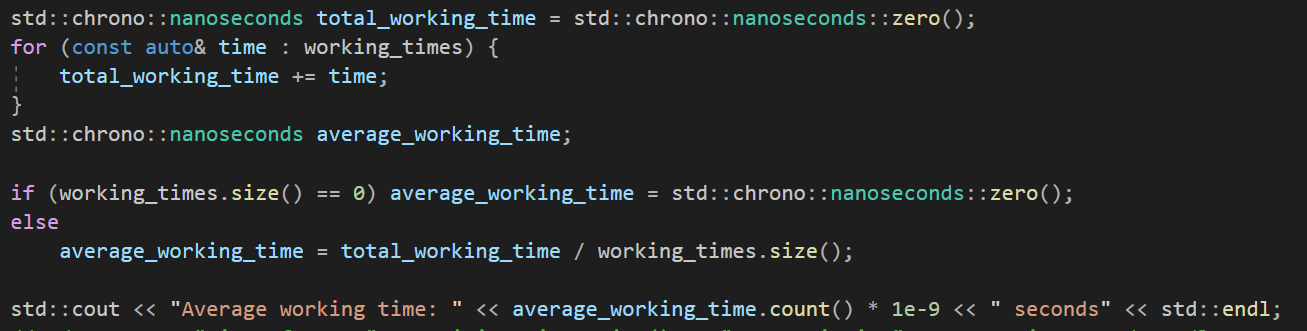
Після цього рахуємо середній час очікування для потока та виводимо.



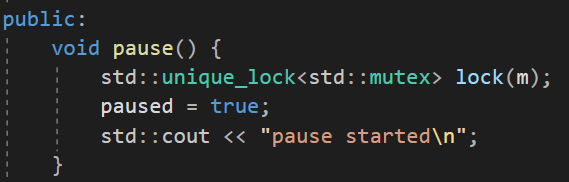
За допомогою auto start\_working та інших рахуємо час роботи задачі і заносимо ці значення до масиву.



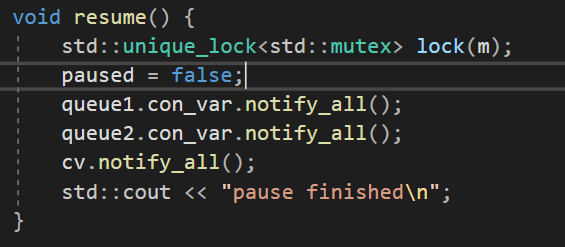
Після цього рахуємо середній час роботи(час виконання задачі) та виводимо.



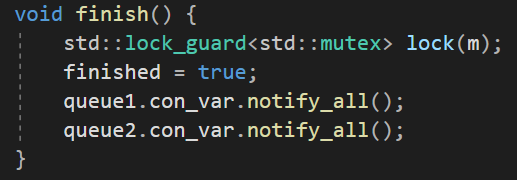
Функія pause необхідна для на певний час призупинити виконання пулу потоків.



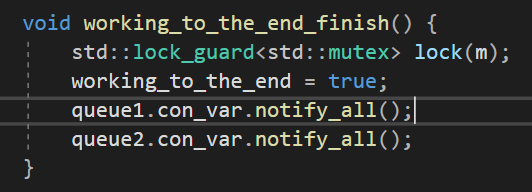
Функія resume необхідна щоб поновити роботу після виклику функії pause.



Функція finish необхідна для завершення роботи пулу потоків. Нові задачі в чергу перестають додаватися, потоки, які вже виконують роботу дороблюють та завершують свою роботу.

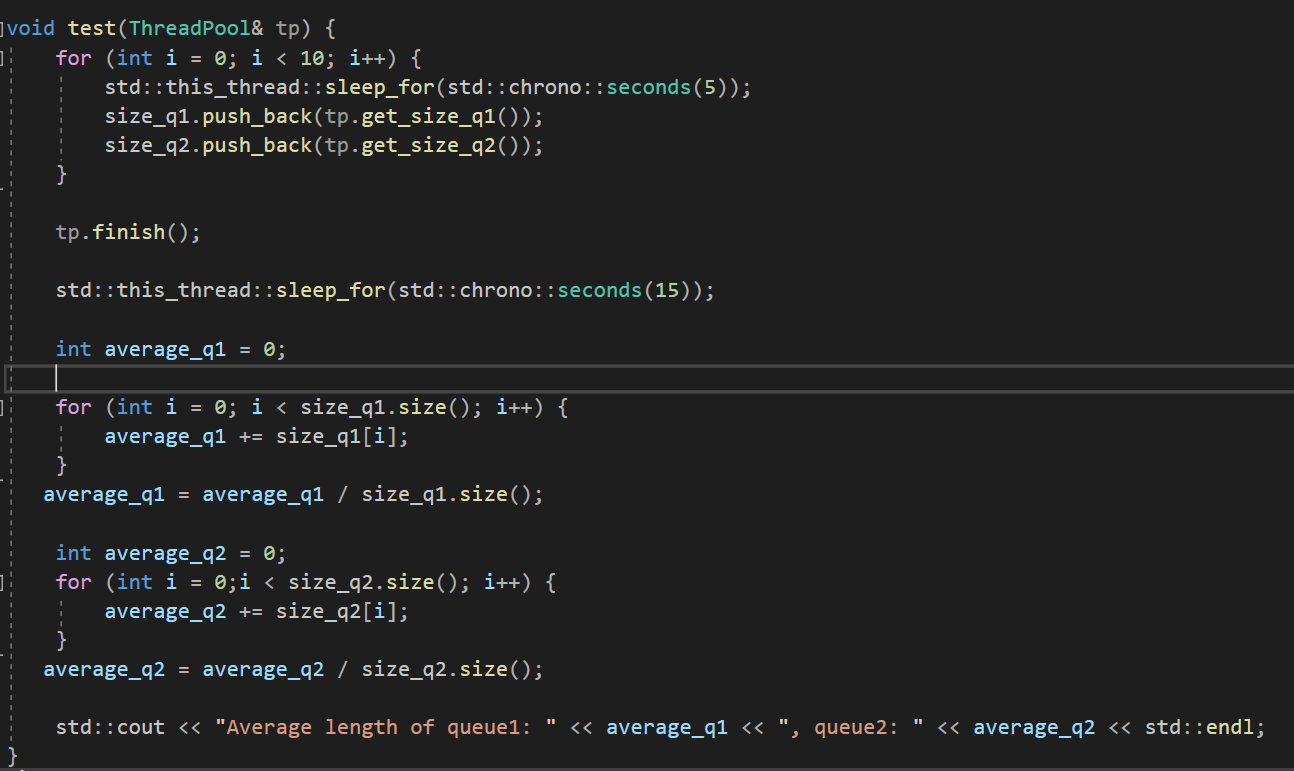


Функція working\_to\_the\_end\_finish необхідна для завершення роботи так само як і функція finish, але на відміність від finish, всі задачі, які є в чергах дороблюються.



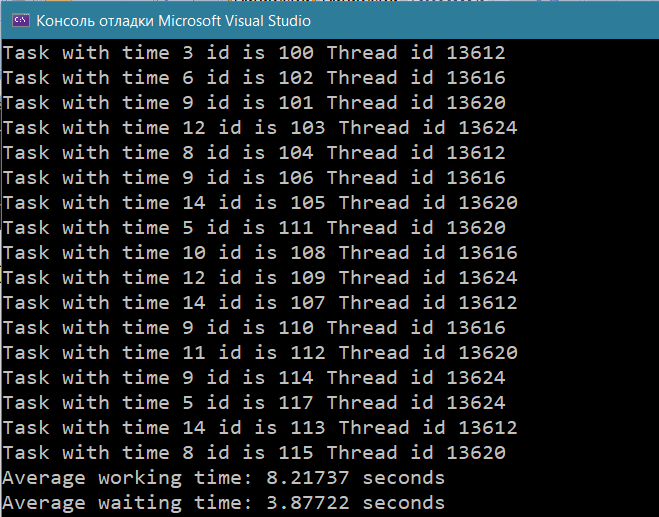
**Тестування**

Функція необхідна для знаходження середньої довжини черги. Інші тести для виявлення середнього часу очікування та середнього часу написані ввише( під поясненням функції consume)

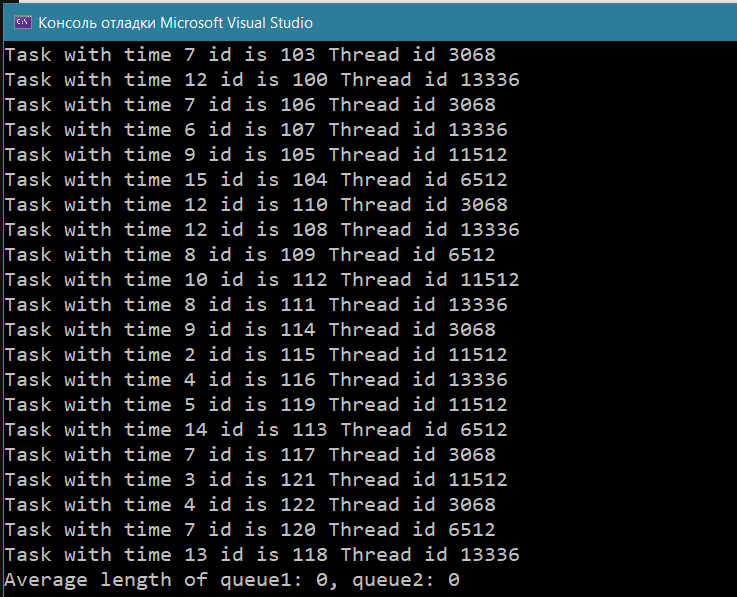


Середній час виконання задач: 8.22 секунди

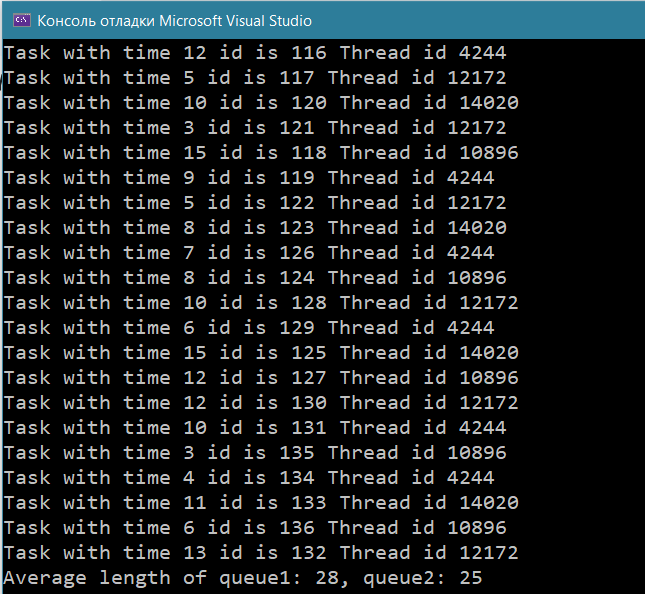
Середній час очікування: 3,88 секунди

Тестування проводилося протягом 5 хвилин.

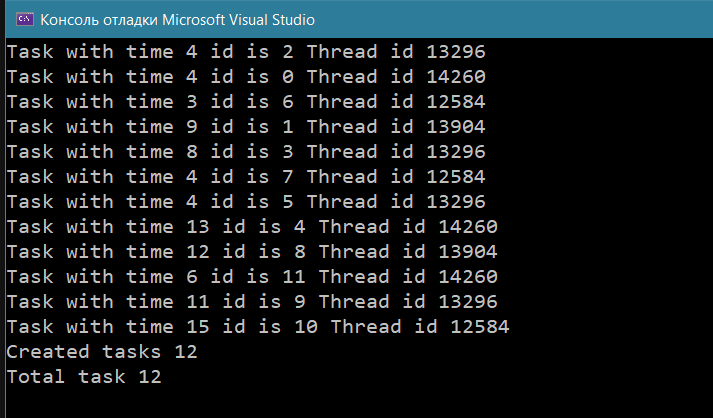
Середня довжина черги:



Так як середній час виконання задачі 8,2 секунди, а продюсер додає задачу раз в 10 секунд, то черги майже завжди пусті. Зробимо повторне тестування зменшивши проміжки між занесеннями задач(зменшимо час з 10 секунд до 5)



Тестування правильності роботи:



Created tasks - це кількість створених задач total task – це кількість виконаних задач. Як видно всі завдання були виконанні

**Висновок:** був створенний пул потоків з 4 потоками,які додають задачі в чергу та 4 потоками, які їх виконують та проведені тести для перевірки правильності його роботи, всі тести були пройдені і записані результати. Середню довжини черги залежить від того, як часто продюсери створюють задачи та середнього часу виконання задачі, на друге впливати ми не можемо, а в залежності від першого можемо побачити як змінюються довжина черги. Щоб підрахувати середній час очікування потоків програму не можна ставити на паузу, бо час паузу буде рахуватися як час очікування.

**Додаток (Код програми):**

#include <iostream>

#include <functional>

#include <vector>

#include <queue>

#include <thread>

#include <mutex>

#include <chrono>

#include <random>

int total\_task = 0 ;

std::mutex m\_out;

std::vector<std::chrono::nanoseconds> waiting\_times;

int num\_wait = 0;

std::vector<int> size\_q1;

std::vector<int> size\_q2;

std::mutex mtx;

std::vector<std::chrono::nanoseconds> working\_times;

class Task {

int my\_id;

int my\_time;

std::function<void(int time,int id)> f;

public:

Task(int id, int time) : my\_id(id), my\_time(time) {

f = [](int my\_time,int my\_id) {

std::this\_thread::sleep\_for(std::chrono::seconds(my\_time));

//std::this\_thread::sleep\_for(std::chrono::seconds(4));

std::lock\_guard<std::mutex> lock(m\_out);

std::cout << "Task with time " << my\_time << " id is " << my\_id <<" Thread id "<< std::this\_thread::get\_id()<< std::endl;

total\_task++;

};

}

Task():my\_id(0),my\_time(0){}

int get\_time() { return my\_time; }

int get\_id() { return my\_id; }

std::function<void(int time,int id)> get\_f() { return f; }

};

class MyQueue {

int all\_time = 0;

std::queue<Task> q;

public:

std::condition\_variable con\_var;

void push(Task t) {

q.push(t);

set\_all\_time(t.get\_time() + get\_all\_time());

}

Task front\_and\_pop() {

Task t = q.front();

q.pop();

set\_all\_time(get\_all\_time() - t.get\_time() );

return t;

}

int size() {

return q.size();

}

bool empty() {

return q.empty();

}

Task front() {

return q.front();

}

int get\_all\_time() {

return all\_time;

}

void set\_all\_time(int time) {

all\_time = time;

}

};

class ThreadPool {

std::vector<std::thread> producer\_threads;

std::vector<std::thread> consumer\_threads;

int created\_tasks = 0;

int num\_producers;

int num\_consumers;

MyQueue queue1;

MyQueue queue2;

std::random\_device rd;

std::mutex m;

std::condition\_variable cv;

bool finished = false;

bool paused = false;

bool working\_to\_the\_end = false;

public:

ThreadPool() = default;

void initialize(int producers, int consumers) {

num\_consumers = consumers;

num\_producers = producers;

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

producer\_threads.emplace\_back(&ThreadPool::produce,this,std::ref(queue1),std::ref(queue2));

}

for (int i = 0; i < consumers/2; i++) {

consumer\_threads.emplace\_back(&ThreadPool::consume, this, std::ref(queue1));

}

for (int i = consumers / 2; i < consumers; i++) {

consumer\_threads.emplace\_back(&ThreadPool::consume, this, std::ref(queue2));

}

}

private:

Task create\_task() {

std::mt19937 mersenne(rd());

std::uniform\_int\_distribution<int> dist(2, 15);

int random\_number = dist(mersenne);

Task t = Task(created\_tasks, random\_number);

created\_tasks++;

return t;

}

void produce(MyQueue& q1, MyQueue& q2) {

while (true ) {

std::unique\_lock<std::mutex> lock(m);

while (paused) { cv.wait(lock); }

if (finished || working\_to\_the\_end) return;

if (q1.get\_all\_time() > q2.get\_all\_time()) {

q2.push(create\_task()); q2.con\_var.notify\_one();}

else { q1.push(create\_task()); q1.con\_var.notify\_one(); }

lock.unlock();

std::this\_thread::sleep\_for(std::chrono::seconds(10));

}

}

void consume(MyQueue& q) {

while (true) {

std::unique\_lock<std::mutex> lock(m);

while ((q.empty() || paused) && !finished && !working\_to\_the\_end) {

num\_wait++;

auto start\_waiting = std::chrono::high\_resolution\_clock::now();

q.con\_var.wait(lock);

auto finish\_waiting = std::chrono::high\_resolution\_clock::now();

auto waiting\_time = std::chrono::duration\_cast<std::chrono::nanoseconds>(finish\_waiting - start\_waiting);

waiting\_times.push\_back(waiting\_time);

}

if (finished || q.empty() ) return;

Task t;

t = q.front\_and\_pop();

lock.unlock();

auto start\_working = std::chrono::high\_resolution\_clock::now();

t.get\_f()(t.get\_time(), t.get\_id());

auto finish\_working = std::chrono::high\_resolution\_clock::now();

auto working\_time = std::chrono::duration\_cast<std::chrono::nanoseconds>(finish\_working - start\_working);

std::lock\_guard<std::mutex> lok(mtx);

working\_times.push\_back(working\_time);

}

}

public:

void pause() {

std::unique\_lock<std::mutex> lock(m);

paused = true;

std::cout << "pause started\n";

}

void resume() {

std::unique\_lock<std::mutex> lock(m);

paused = false;

queue1.con\_var.notify\_all();

queue2.con\_var.notify\_all();

cv.notify\_all();

std::cout << "pause finished\n";

}

void finish() {

std::lock\_guard<std::mutex> lock(m);

finished = true;

queue1.con\_var.notify\_all();

queue2.con\_var.notify\_all();

}

void working\_to\_the\_end\_finish() {

std::lock\_guard<std::mutex> lock(m);

working\_to\_the\_end = true;

queue1.con\_var.notify\_all();

queue2.con\_var.notify\_all();

}

int get\_created\_tasks() { return created\_tasks; }

int get\_size\_q1() { return queue1.size(); }

int get\_size\_q2() { return queue2.size(); }

~ThreadPool() {

for (int i = 0; i < num\_producers; i++) {

producer\_threads[i].join();

}

for (int i = 0; i < num\_consumers; i++) {

consumer\_threads[i].join();

}

std::cout << "Created tasks " << this->get\_created\_tasks() << "\n" << "Total task " << total\_task << "\n";

}

};

void test(ThreadPool& tp) {

for (int i = 0; i < 10; i++) {

std::this\_thread::sleep\_for(std::chrono::seconds(5));

size\_q1.push\_back(tp.get\_size\_q1());

size\_q2.push\_back(tp.get\_size\_q2());

}

tp.finish();

std::this\_thread::sleep\_for(std::chrono::seconds(15));

int average\_q1 = 0;

for (int i = 0; i < size\_q1.size(); i++) {

average\_q1 += size\_q1[i];

}

average\_q1 = average\_q1 / size\_q1.size();

int average\_q2 = 0;

for (int i = 0;i < size\_q2.size(); i++) {

average\_q2 += size\_q2[i];

}

average\_q2 = average\_q2 / size\_q2.size();

std::cout << "Average length of queue1: " << average\_q1 << ", queue2: " << average\_q2 << std::endl;

}

int main()

{

ThreadPool tp;

tp.initialize(4, 4);

test(std::ref(tp));

std::this\_thread::sleep\_for(std::chrono::seconds(300));

tp.finish();

std::chrono::nanoseconds total\_working\_time = std::chrono::nanoseconds::zero();

for (const auto& time : working\_times) {

total\_working\_time += time;

}

std::chrono::nanoseconds average\_working\_time;

if (working\_times.size() == 0) average\_working\_time = std::chrono::nanoseconds::zero();

else

average\_working\_time = total\_working\_time / working\_times.size();

std::cout << "Average working time: " << average\_working\_time.count() \* 1e-9 << " seconds" << std::endl;

std::chrono::nanoseconds total\_waiting\_time = std::chrono::nanoseconds::zero();

for (const auto& time : waiting\_times) {

total\_waiting\_time += time;

}

std::chrono::nanoseconds average\_waiting\_time;

if(waiting\_times.size() == 0 ) average\_waiting\_time = std::chrono::nanoseconds::zero();

else

average\_waiting\_time = total\_waiting\_time / waiting\_times.size();

std::cout << "Average waiting time: " << average\_waiting\_time.count() \*1e-9<< " seconds" << std::endl;

std::cout << "Size of vect " << waiting\_times.size() << " num wait is " << num\_wait << std::endl;

}