ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

(повне найменування назва факультету)

Кафедра програмування

(повна назва кафедри)

ДИПЛОМНА РОБОТА

на тему: «Пул потоків у C++»

Студента IV курсу, групи ПМІ-41,

напряму підготовки *122 Комп'ютерні науки*

Полянського Андрія

(прізвище та ініціали)

Керівник *доцент Клакович Леся Миронівна*

(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Національна шкала _____

Кількість балів: _____ Оцінка: ECTS _____

Львів 2023

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

Кафедра програмування

Спеціальність 122 Комп'ютерні науки

«ЗАТВЕРДЖУЮ»

Завідувач кафедри Ярошко С.А.

" _ "_____ 2023 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Полянського Андрія

(прізвище, ім'я, по батькові)

1. Тема роботи «Пул потоків у С++»

керівник роботи Клакович Леся Миронівна, доцент кафедри програмування

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені Вченою радою факультету від "13"_вересня 2022 року №_15_.

2. Строк подання студентом роботи 12 червня 2023 року

3. Вихідні дані до роботи літературні джерела, інтернет-ресурси, постановка задачі, наукові статті

4. Зміст дипломної роботи (перелік питань, які потрібно розробити)

1. Вивчити концепти операційних систем для розробки асинхронних програм;
2. Вивчити основні підходи до розробки пулу потоків;
3. Розробити архітектуру та імплементацію бібліотеки мовою С++;
4. Провести тестування бібліотеки;

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Ознайомлення з предметною областю	05.09.2022 – 05.11.2022	
2	Аналіз аналогів на ринку	05.11.2022 – 15.11.2022	
3	Написання специфікації вимог	15.11.2022 – 17.11.2022	
4	Вибір технологій	17.11.2022 – 18.11.2022	
5	Проєктування архітектури бібліотеки	18.11.2022 – 26.12.2022	
6	Імплементація бібліотеки	15.01.2023 – 23.04.2023	
7	Написання тестів	23.04.2023 – 14.05.2023	
8	Виправлення багів	14.05.2023 – 20.05.2023	
9	Оформлення дипломної роботи	20.05.2023 – 10.06.2023	
10	Подання дипломної роботи	12.06.2023	

Студент _____ Полянський А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Клакович Л. М.
(підпис) (прізвище та ініціали)

ЗМІСТ

1. КОНЦЕПТИ ОПЕРАЦІЙНОЇ СИСТЕМИ 8
1.1 Процеси 8
1.2 Потоки 9
1.2.1 Види потоків 9
1.2.1.1 Потоки рівня ядра (англ. kernel-level threads) 9
1.2.1.2 Потоки рівня користувача (англ. user-level threads) 10
1.2.2 Використання потоків 10
1.2.3 Керування потоками 10
1.3 Синхронізаційні примітиви 11
1.3.1 Основні типи синхронізаційних примітивів 11
1.3.1.1 Блокування (англ. locks) 11
1.3.1.2 Семафори (англ. semaphores) 11
1.3.1.3 М'ютекси (англ. mutexes) 11
1.3.1.4 Умовні змінні (англ. condition variables) 12
1.3.1.5 Бар'єри (англ. barriers) 12
1.4 Планувальник задач 12
1.4.1 Основні функції планувальника задач 12
1.4.2 Алгоритми планування 13
1.4.2.1 First-Come, First-Served (FCFS) 13
1.4.2.2 Shortest-Job-First (SJF) 14
1.4.2.3 Пріоритетне планування 14
1.4.2.4 Round Robin (RR) 14
1.4.2.5 Багаторівневе планування черги 14
1.4.2.6 Багаторівневе планування черги зі зворотним зв'язком 15
1.4.2.7 Shortest Remaining Time First (SRTF) 15
1.4.2.8 Longest Remaining Time First (LRTF) 15
1.4.2.9 Highest Response Ratio Next (HRRN) 15
1.4.3 Типи алгоритмів 15
1.4.3.1 Випереджувальні алгоритми планування 16
1.4.3.2 Не випереджувальні алгоритми планування 16
1.4.4 Однопроцесорне/багатопроцесорне планування 16
1.4.4.1 Однопроцесорне планування (англ. single-processor scheduling) 16
1.4.4.2 Багатопроцесорне планування (англ. multiprocessor scheduling) 17
1.4.4.2.1 Симетричне та асиметричне планування 17
1. 5 Балансувальник навантаження 19

1.5.1 Стратегії балансування 20
2. ПУЛ ПОТОКІВ 22
2.1 Проблеми при розробці 23
3. АСИНХРОННЕ ПРОГРАМУВАННЯ 25
4. ОГЛЯД РИНКУ 26
6.1 Intel Threading Building Blocks (TBB) 26
6.2 Boost.Thread 26
6.3 PPL (Parallel Patterns Library) 26
6.4 ThreadPool (C++11) 27
5. ТЕХНОЛОГІЇ 28
4.1 C++ 28
4.2 CMake 29
4.3 gTest 31
4.4 Visual Studio 32
6. АРХІТЕКТУРА 34
5.1 WorkerThread 35
5.2 IThreadPoolTask 36
5.3 ITaskScheduler 37
5.4 IThreadPool 38

АНОТАЦІЯ

Ця дипломна робота присвячена детальному дослідженню, проєктуванню та реалізації пулу потоків мовою програмування C++. Метою роботи є розуміння концепції пулу потоків, аналіз існуючих реалізацій та розробка оптимального пулу потоків для використання в різноманітних програмах.

У першому розділі роботи проводиться детальний огляд концептів операційної системи, зокрема потоків, процесів та планувальника. Розглядається важливість ефективного управління потоками та вплив цього на продуктивність програм. Розглядаються різні стратегії планування завдань та методи синхронізації.

Другий розділ присвячений розгляду поняття пулу потоків. Розглядаються основні принципи та функції пулу потоків та складнощі при розробці.

Третій розділ розглядає асинхронне програмування та його переваги.

Четвертий розділ присвячений огляду ринку та аналізу існуючих рішень у галузі пулу потоків. Розглядаються комерційні та відкриті реалізації пулу потоків, їх функціональні можливості та переваги.

У п’ятому розділі описується архітектура розробленої бібліотеки пулу потоків мовою C++. Розглядаються ключові компоненти бібліотеки, методи синхронізації та керування потоками. Показуються важливі аспекти реалізації пулу потоків та його використання у реальних програмних проєктах.

Ключові слова: пул потоків, багатозадачність, багатопотоковість, синхронізація, планування, продуктивність, оптимізація, масштабованість.
Посилання на опубліковану роботу: https://github.com/ppolyanskiyy/ThreadPool
Загальний обсяг роботи - 40 сторінок

ВСТУП

Сучасні програми все більше стикаються з необхідністю ефективно використовувати обчислювальні ресурси, працювати з асинхронними завданнями та забезпечувати швидкий відгук користувачам. Одним із способів досягнення цих цілей є використання паралельного програмування та ефективного управління потоками в програмах.

У контексті мови програмування C++, одним із потужних інструментів для роботи з потоками є пул потоків (англ. thread pool). Пул потоків є популярним концептом, який дозволяє керувати набором потоків та ефективно виконувати асинхронні завдання в програмах. Використання пулу потоків може полегшити розробку паралельних програм, спростити управління потоками, зменшити витрати пам'яті та покращити продуктивність.

Ця дипломна робота присвячена детальному дослідженню, проєктуванню та реалізації пулу потоків мовою програмування C++. Метою роботи є розуміння концепції пулу потоків, аналіз існуючих реалізацій та розробка оптимального пулу потоків для використання в різноманітних програмах.

Дипломна робота складатиметься з кількох розділів, включаючи огляд концептів операційних систем, аналіз поняття пулу потоків, розробку архітектури та реалізацію пулу потоків у C++. Основні завдання дипломної роботи включають:

a) Дослідження теоретичних основ пулу потоків та аналіз існуючих реалізацій.

b) Проєктування та розробка архітектури пулу потоків у C++.

c) Реалізація розробленого пулу потоків з використанням сучасних можливостей мови C++.

d) Валідація та тестування реалізації пулу потоків.

Очікується, що результати цієї дипломної роботи допоможуть розробникам програмних продуктів з легкістю використовувати пул потоків у своїх проєктах, забезпечуючи покращену продуктивність та швидкий відгук користувачам.

1. КОНЦЕПТИ ОПЕРАЦІЙНОЇ СИСТЕМИ

1.1 Процеси

В операційних системах процес є фундаментальною концепцією, яка представляє виконавчу частину програми. Процес є ізольованим середовищем виконання, що містить код програми, дані, пам'ять, регістри та контекст виконання. Це забезпечує ізоляцію між процесами, тобто один процес не може напряму змінювати дані або виконувати операції в іншому процесі.

Операційна визначає, які процеси отримують доступ до процесора та на який період часу. Планування забезпечує справедливий розподіл процесорного часу між процесами та оптимальне використання ресурсів.

Також, операційна система надає процесам доступ до системних ресурсів, таких як процесорний час, пам'ять, введення-виведення та інші пристрої. Керування ресурсами включає розподіл ресурсів між процесами, управління чергами завдань та механізми синхронізації для уникнення конфліктів та змагань за ресурси.

Коли операційна система перемикає виконання між процесами, вона зберігає поточний стан виконання процесу у контексті та завантажує контекст нового процесу. Це називається контекстним перемиканням, і воно відбувається дуже швидко для забезпечення безперервної роботи системи.

Процеси можуть взаємодіяти один з одним через механізми міжпроцесної взаємодії. Це можуть бути спільна пам'ять, канали зв'язку, сигнали, повідомлення та синхронізаційні примітиви. Це дозволяє процесам обмінюватись даними, спілкуватись та спільно вирішувати завдання.

Процеси мають свій життєвий цикл, який складається зі станів, таких як створення, готовність, виконання, блокування та завершення. Операційна система керує переходами між цими станами залежно від подій та дій процесу.

Коли процес потребує виконання операцій, що займають багато часу або можуть блокувати процес, він може передати керування іншому процесу. Як вже було згадано, це називається перемикання контексту та дозволяє операційній системі продовжувати виконання інших процесів.

Процеси є основою для багатозадачності, де декілька процесів можуть виконуватись одночасно, та забезпечують ефективне використання ресурсів операційної системи. Вони дозволяють програмам працювати паралельно, взаємодіяти та забезпечувати правильну послідовність виконання завдань.

1.2 Потоки

Операційні системи використовують потоки (англ. threads) для реалізації паралельної обробки та багатозадачності. Потоки є основною одиницею виконання в рамках процесу та дозволяють програмам виконувати кілька послідовних або паралельних шляхів виконання. Кожен процес може мати один або більше потоків. Вони спільно використовують пам'ять та ресурси процесу.

Потоки використовуються для виконання паралельних або асинхронних операцій, що дозволяють програмам більш ефективно використовувати доступні ресурси. Всі потоки в процесі ділять адресний простір процесу, але мають власні стеки викликів та регістри.

Потоки взаємодіють між собою через спільну пам'ять, що може призводити до проблем зі змаганням за ресурси та синхронізацією.

1.2.1 Види потоків

1.2.1.1 Потоки рівня ядра (англ. kernel-level threads)

Це потоки, які керуються операційною системою. Операційна система відповідає за створення, планування та керування цими потоками. Вони надають більшу надійність, але меншу продуктивність через значні накладні витрати на перемикання контексту між різними рівнями виконання.

1.2.1.2 Потоки рівня користувача (англ. user-level threads)

Це потоки, які керуються самою користувацькою програмою або бібліотекою. Операційна система несвідома про наявність таких потоків, і вони можуть бути реалізовані безпосередньо у програмному коді. Вони забезпечують більшу продуктивність, але меншу надійність, оскільки проблеми в одному потоці можуть вплинути на інші.

1.2.2 Використання потоків

Потоки дозволяють програмам виконувати кілька задач одночасно. Кожен потік може виконувати свої операції та отримувати доступ до спільної пам'яті.

Також, потоки дозволяють програмі розділити обчислення на менші частини та виконувати їх паралельно на різних процесорах або ядрах. Це забезпечує покращену продуктивність та швидкодію програми.

Потоки дозволяють виконувати операції асинхронно, не блокуючи інші потоки. Це особливо корисно для розробки програм, які взаємодіють з мережею або виконують введення-виведення.

1.2.3 Керування потоками

Програми можуть створювати потоки за допомогою спеціальних функцій або методів, наданих операційною системою або програмними бібліотеками.

Операційна система відповідає за планування виконання потоків та розподіл ресурсів. Вона вирішує, які потоки будуть виконуватись в який момент часу та на яких процесорах.

У разі спільного доступу до ресурсів потоки можуть потребувати синхронізації, щоб уникнути проблем зі змаганням за ресурси або забезпечити правильну послідовність виконання.

Загалом, потоки є потужним інструментом для реалізації паралельної обробки та багатозадачності в операційних системах. Вони дозволяють програмам ефективно використовувати ресурси, покращувати продуктивність та забезпечувати швидку та гнучку обробку завдань.

1.3 Синхронізаційні примітиви

Синхронізаційні примітиви в операційних системах є інструментами, які дозволяють керувати та координувати взаємодію між процесами або потоками. Вони забезпечують синхронізацію доступу до спільних ресурсів та уникнення проблем зі змаганням за ресурси. Користування синхронізаційними примітивами вимагає уважності та правильного проєктування, щоб уникнути ситуацій, які можуть призвести до зациклення або взаємоблокування програми.

Синхронізаційні примітиви охоплюють різні механізми, такі як блокування, семафори, м'ютекси, умовні змінні та бар'єри.

1.3.1 Основні типи синхронізаційних примітивів

1.3.1.1 Блокування (англ. locks)

Блокування є найпоширенішими синхронізаційними примітивами. Вони забезпечують ексклюзивний доступ до ресурсу, що означає, що тільки один процес або потік може мати доступ до ресурсу в певний момент часу. Блокування мають два стани: заблокований та незаблокований. Якщо ресурс заблокований, інші процеси чекають, поки ресурс стане доступним.

1.3.1.2 Семафори (англ. semaphores)

Семафори є синхронізаційними об'єктами, які використовуються для керування доступом до обмежених ресурсів. Вони містять лічильник, який може збільшуватись або зменшуватись потоками. Потік, що намагається отримати доступ до ресурсу, перевіряє значення семафора. Якщо значення більше нуля, він отримує доступ, інакше потік блокується поки ресурс не звільниться.

1.3.1.3 М'ютекси (англ. mutexes)

М'ютекси є спеціальним типом блокування, який забезпечує ексклюзивний доступ до ресурсу. Тільки один потік може утримувати м'ютекс одночасно. Якщо інший потік намагається отримати м'ютекс, він блокується і чекає, поки перший потік не звільнить його.

1.3.1.4 Умовні змінні (англ. condition variables)

Умовні змінні використовуються для координування виконання потоків у визначених умовах. Вони дозволяють потокам чекати на певну умову і повідомляти про цю умову іншим потокам. Умовні змінні зазвичай використовуються разом з м'ютексами для забезпечення безпеки при зміні умови.

1.3.1.5 Бар'єри (англ. barriers)

Бар'єри використовуються для синхронізації групи потоків. Вони встановлюють точку синхронізації, де всі потоки зупиняються та чекають, поки всі інші потоки досягнуть цієї точки. Після того, як всі потоки досягли бар'єру, вони продовжують своє виконання.

1.4 Планувальник задач

Планувальник задач (також відомий як планувальник процесів) є важливою складовою операційних систем і виконує ключову роль у керуванні виконанням процесів. Його основна відповідальність полягає в розподілі процесорного часу між процесами або потоками, а також в прийнятті рішень про пріоритети та послідовність виконання задач.

1.4.1 Основні функції планувальника задач

Планувальник задач визначає, які процеси або потоки отримують доступ до процесора та на який проміжок часу. Він вирішує, яку задачу виконувати в даному моменті і яким чином перемикати контекст між процесами.

Кожному процесу або потоку може бути призначений пріоритет, який вказує, наскільки важливою є задача. Планувальник задач використовує ці пріоритети для визначення порядку виконання процесів. Задачі з вищим пріоритетом отримують більше процесорного часу або пріоритетніше виконуються.

Планувальник задач реагує на різні події, які можуть впливати на виконання процесів. Це можуть бути введення-виведення, таймери, запити від користувача та інші події. Він вирішує, коли та як перемикати виконання задач, щоб ефективно реагувати на події та забезпечувати продуктивність системи.

Планувальник задач використовує різні алгоритми для визначення порядку виконання процесів. Це можуть бути алгоритми першого прийняття (First-Come, First-Served), найкоротшого часу виконання (Shortest Job First), пріоритетного планування, алгоритми зворотного зв'язку та багато інших. Вибір алгоритму залежить від вимог до системи, характеристик задач та цілей оптимізації. Огляд цих алгоритмів буде наведено далі.

Планувальник задач також відповідає за керування іншими системними ресурсами, такими як пам'ять та ввід-вивід. Він контролює доступ до цих ресурсів та забезпечує їх ефективне використання серед процесів або потоків.

Планувальник задач може працювати в різних режимах, таких як планування зі строгою пріоритетністю (англ. strict priority scheduling), планування з часовим квантом (англ. time quantum scheduling), планування в реальному часі (англ. real-time scheduling) та інші. Вибір планувальника та його режиму залежить від потреб конкретної операційної системи та ситуацій, в яких вона використовується.

Ефективний планувальник задач грає важливу роль у забезпеченні продуктивності, надійності та відклику операційної системи. Він дозволяє раціонально використовувати процесорний час, уникати заторів та перенавантажень системи, а також забезпечує справедливий доступ до ресурсів для всіх процесів або потоків.

1.4.2 Алгоритми планування

В операційних системах алгоритми планування відповідають за розподіл обчислювальних ресурсів, таких як процесорний час та вводи/виводи, між різними процесами або потоками. Ці алгоритми грають ключову роль у досягненні оптимальної продуктивності, справедливого використання ресурсів та задоволення вимог користувачів. Наступні алгоритми планування є найпоширенішими в операційних системах.

1.4.2.1 First-Come, First-Served (FCFS)

FCFS (дослівно “першим прийшов, першим обслужений”) є найпростішим алгоритмом планування. Завдання обробляються в порядку їх прибуття. Якщо

завдання А прибуває першим, воно виконується до закінчення, а потім переходить до наступного завдання у черзі. Цей алгоритм не враховує тривалості завдань, тому може виникати проблема "відрядження" (англ. convoy effect), коли довге завдання блокує виконання коротких завдань.

1.4.2.2 Shortest-Job-First (SJF)

SJF (дослівно “найкоротша робота перша”) вибирає найкоротше завдання для виконання. Приблизна тривалість завдань відома заздалегідь або оцінюється. Планувальник віддає перевагу завданню з найменшою очікуваною тривалістю. Цей алгоритм забезпечує мінімізацію часу очікування для завдань, але вимагає знання про тривалість завдань заздалегідь.

1.4.2.3 Пріоритетне планування

Пріоритетне планування використовується, коли кожному завданню присвоюється пріоритет. Завдання з вищим пріоритетом виконуються першими. Цей алгоритм може бути реалізований як неперервний (пріоритети надаються в режимі реального часу) або дискретний (пріоритети присвоюються під час запуску процесу). Проблема оптимізації полягає в правильному призначенні пріоритетів між завданнями.

1.4.2.4 Round Robin (RR)

RR (Round Robin - це вид дитячої каруселі в США) використовує кванти часу, де кожне завдання отримує обмежений час виконання, відомий як квант. Якщо завдання не встигає завершитися за відведений квант, воно переходить до черги інших завдань. Цей алгоритм забезпечує справедливе розподіл часу процесора між завданнями, але може бути неефективним для довгих завдань.

1.4.2.5 Багаторівневе планування черги

Багаторівневе планування черги (англ. multilevel queue scheduling) використовує декілька черг з різними пріоритетами. Кожна черга може мати свій власний алгоритм планування, наприклад, FCFS або RR. Завданням присвоюється певний пріоритет, і воно розміщується у відповідній черзі. Завдання з вищим пріоритетом виконуються першими.

1.4.2.6 Багаторівневе планування черги зі зворотним зв'язком

Багаторівневе планування черги зі зворотним зв'язком (англ. multilevel feedback queue scheduling) подібний до багаторівневого планування черги, але має можливість переміщення завдань між чергами на основі їхнього виконання та пріоритету. Завдання може бути переведено в чергу з більшим або меншим пріоритетом залежно від його характеристик та роботи. Це дозволяє адаптувати планування до змінних потреб системи.

1.4.2.7 Shortest Remaining Time First (SRTF)

SRTF (дослівно “спочатку найкоротший час, що залишився”) є варіацією SJF, де завдання з найменшим часом виконання поступають в чергу першими. Якщо прибуває нове завдання з ще коротшим часом виконання, воно може перервати виконання поточного завдання.

1.4.2.8 Longest Remaining Time First (LRTF)

LRTF (дослівно “спочатку найдовший час, що залишився”) також є варіацією SJF, де завдання з найбільшим часом виконання поступають в чергу першими. Цей алгоритм надає перевагу довгим завданням, що може призвести до затримки коротких завдань.

1.4.2.9 Highest Response Ratio Next (HRRN)

HRRN (дослівно “найвищий коефіцієнт відповіді наступний”) вибирає завдання з найвищим співвідношенням часу очікування до тривалості. Завдання, яке має довгий час очікування, але має короткий час виконання, буде мати високий пріоритет. Це дозволяє забезпечити більш справедливе розподілення часу процесора між завданнями.

1.4.3 Типи алгоритмів

В контексті планування в операційних системах існують два основних типи алгоритмів: випереджувальні (англ. preemptive) та не випереджувальні (англ. non-preemptive).

Обираючи між випереджувальними та не випереджувальними алгоритмами планування, розробники операційних систем мають враховувати особливості

системи, вимоги до часу відклику, пріоритети завдань та ефективність використання ресурсів. Кожен тип алгоритму має свої переваги та обмеження, і вибір залежить від конкретних потреб та цілей системи.

1.4.3.1 Випереджувальні алгоритми планування

Випереджувальні алгоритми дозволяють переривати виконання поточного процесу або потоку, якщо з'являється процес або потік з вищим пріоритетом або якась інша подія, що вимагає негайної уваги. У таких алгоритмах планування виконання процесу або потоку може бути призупинене, а його ресурси виділені іншому процесу або потоку. Випереджувальні алгоритми надають більш гнучкий контроль над виконанням, дозволяють відповідати на пріоритетні події негайно та забезпечують швидший відклик системи. Приклади випереджувальних алгоритмів включають Round Robin, Shortest Remaining Time First (SRTF) та Highest Response Ratio Next (HRRN).

1.4.3.2 Не випереджувальні алгоритми планування

Не випереджувальні алгоритми дозволяють процесу або потоку виконуватися без переривання до завершення або до тих пір, поки вони добровільно не передадуть управління іншому процесу або потоку. В таких алгоритмах процес або потік утримують ресурси до завершення або до тих пір, поки не стане доступним інший ресурс або відбудеться певна подія. Не випереджувальні алгоритми прості у реалізації та використанні ресурсів, але можуть мати проблеми з реагуванням на пріоритетні події та забезпеченням низької латентності системи. Приклади не випереджувальних алгоритмів включають First Come First Serve (FCFS), Shortest-Job-First (SJF) та Пріоритетне планування.

1.4.4 Однопроцесорне/багатопроцесорне планування

1.4.4.1 Однопроцесорне планування (англ. single-processor scheduling)

Однопроцесорне планування належить до планування завдань у системі з одним процесором. У такій системі лише один процесор виконує задачі у послідовному режимі. Завдання, які надходять до системи, чергуються і

виконуються по одному на процесорі відповідно до обраного алгоритму планування. Приклади алгоритмів планування для однопроцесорних систем включають FCFS, SJF, Round Robin і пріоритетне планування. Однопроцесорне планування використовується в багатьох стандартних настільних комп'ютерах і серверах з одним процесором.

1.4.4.2 Багатопроцесорне планування (англ. multiprocessor scheduling)

Багатопроцесорне планування стосується систем, в яких наявність більше одного процесора, що можуть виконувати завдання паралельно. У таких системах надходять багато завдань, і їх слід розподілити між доступними процесорами для оптимального використання ресурсів і покращення продуктивності системи. Багатопроцесорне планування може включати різні алгоритми, такі як балансування навантаження (англ. load balancing), роздільне планування (англ. partitioned scheduling), глобальне планування (англ. global scheduling) та інші. Завдання можуть бути розподілені між процесорами на основі їхнього пріоритету, часу виконання, навантаження та інших факторів. Багатопроцесорне планування використовується в потужних серверних системах, кластерах обчислювальних вузлів, суперкомп'ютерах та інших системах, що мають кілька процесорів для паралельного виконання завдань.

1.4.4.2.1 Симетричне та асиметричне планування

Є два підходи до багатопроцесорного планування:

а) **Симетрична обробка**. У симетричній обробці всі процесори мають однаковий доступ до пам'яті та виконують однаковий набір операцій. Кожен процесор може виконувати будь-яку задачу з загального пулу із процесами. Планувальник розподіляє завдання рівномірно між доступними процесорами. Процесори можуть спілкуватися та синхронізуватися між собою для обміну даними та координації роботи. До того ж є декілька сценаріїв для реалізації черги завдань:

1) *Глобальна черга*. У системі з глобальною чергою всі процесори спільно використовують одну чергу завдань. Кожен

процесор може вибирати наступне завдання з глобальної черги для виконання. Цей підхід дозволяє розподіляти завдання рівномірно між процесорами та забезпечувати балансування навантаження

2) *Локальна черга*. У системі з локальною чергою кожен процесор має свою власну чергу завдань. Кожен процесор вибирає завдання зі своєї власної черги для виконання. Цей підхід дозволяє зменшити взаємне блокування та конфлікти при доступі до черги завдань, оскільки кожен процесор має локальний доступ до своєї черги.

3) *Гібридний*. Гібридний підхід поєднує як глобальну, так і локальну чергу в залежності від контексту та потреб системи. Наприклад, можна використовувати глобальну чергу для розподілу завдань між процесорами, а потім перемикатися на локальну чергу для збільшення швидкодії та зниження накладних витрат.

Симетрична обробка зазвичай є більш гнучкою, оскільки кожен процесор може виконувати будь-яке завдання з пулу процесів. Вона дозволяє досягти кращої масштабованості та розподілу навантаження.

b) **Асиметрична обробка**. У асиметричній обробці різні процесори мають різні ролі та функції. Один процесор виконує завдання операційної системи та розподіляє завдання між іншими процесорами. Основний процесор (англ. master server) приймає рішення про розподіл завдань та координує роботу інших процесорів (англ. slaves). Слейв-процесори виконують завдання, які їм видав мастер-процесор. Асиметрична обробка може бути ефективною для конкретних випадків, де деякі процесори мають спеціалізовані функції або завдання. Вона може забезпечувати краще управління розподілом ресурсів та використанням процесорів.

Вибір між симетричною та асиметричною обробкою залежить від конкретних потреб, характеру завдань та архітектури системи. Обидва підходи використовуються в різних сферах, включаючи сервери, вбудовані системи та паралельні обчислення.

1. 5 Балансувальник навантаження

Балансувальник навантаження (англ. load balancer) є важливим компонентом операційних систем та мережевих інфраструктур, призначеним для розподілу робочого навантаження між різними ресурсами з метою підвищення продуктивності, доступності та масштабованості системи. Основна ідея балансування навантаження полягає в тому, щоб розподілити навантаження між різними серверами, комп'ютерами процесами чи навіть потоками таким чином, щоб уникнути перевантаження одних ресурсів та забезпечити рівномірно розподілене навантаження.

Балансувальник навантаження використовує різні алгоритми для розподілу навантаження між ресурсами. Це може бути розподіл навантаження за круговим методом (англ. round robin), використання ваги (англ. weighted distribution), вибір найменш завантаженого ресурсу (англ. least loaded), аналіз процесорного часу або пропускної здатності ресурсу та багато інших алгоритмів.

Балансувальник навантаження постійно моніторить стан ресурсів, до яких він розподіляє навантаження. Це може включати моніторинг використання процесора, пам'яті, пропускної здатності мережі, навантаження сервера та інші метрики. На підставі цих метрик балансувальник приймає рішення про розподіл навантаження.

Балансувальник навантаження може виявляти збої або відмову в роботі ресурсів та реагувати на них. Наприклад, якщо сервер стає недоступним або перевантаженим, балансувальник може автоматично виключити його з розподілу навантаження та перенаправити роботу на інші ресурси. Це забезпечує високу доступність та надійність системи.

Балансувальники навантаження дозволяють системі масштабуватися горизонтально, тобто додавати або видаляти ресурси з мережі без необхідності зміни архітектури або конфігурації програмного забезпечення. Це дозволяє динамічно змінювати розподіл навантаження в залежності від змін потреб користувачів.

Балансувальники навантаження можуть бути реалізовані як апаратне або програмне забезпечення, а також входити в склад мережевих пристроїв, таких як

маршрутизатори або комутатори. Вони є важливим елементом для побудови високопродуктивних, масштабованих та надійних систем, що забезпечують розподіл навантаження між різними ресурсами та оптимальне використання ресурсів.

1.5.1 Стратегії балансування

У контексті балансувальника навантаження, push і pull міграції відталування (англ. push) та притягування (англ. pull) є двома різними стратегіями переміщення задач або ресурсів між вузлами системи з метою забезпечення оптимального розподілу навантаження. Розгляньмо кожен підхід детальніше:

a) **Push-міграції**. У стратегії push-міграції балансувальник навантаження активно передає задачі з одного вузла до іншого. Це означає, що балансувальник навантаження самостійно визначає, які задачі слід перемістити з перевантаженого вузла на менш завантажений. Такий підхід дозволяє розподілити навантаження від активних вузлів до менш активних та забезпечити більш рівномірне розподілення роботи.

b) **Pull-міграції**. У стратегії pull-міграції вузли-отримувачі активно запитують у балансувальника навантаження про задачі, які вони можуть виконати. Балансувальник навантаження відповідає запитам, надаючи задачі, які потрібно виконати. Вузли-отримувачі самостійно вирішують, які задачі вони приймають. Цей підхід дозволяє вузлам-отримувачам динамічно адаптувати своє навантаження залежно від їх поточного стану та можливостей.

Кожна стратегія має свої переваги та обмеження. Push-міграція дозволяє активно контролювати розподіл роботи та швидко реагувати на зміни навантаження. Pull-міграція забезпечує більш гнучкий підхід та дозволяє вузлам самостійно приймати рішення щодо прийняття роботи. Вибір між цими стратегіями залежить від характеристик системи, типу роботи, пропускної здатності мережі та інших факторів.

Ефективне використання push і pull міграцій в балансувальнику навантаження може допомогти забезпечити високу продуктивність та оптимальне використання ресурсів у багатопроцесорних системах.

2. ПУЛ ПОТОКІВ

Пул потоків (англ. thread pool) - це механізм у програмуванні, який дозволяє ефективно керувати потоками для виконання асинхронних завдань. Він складається з певної кількості заздалегідь створених потоків, які готові для обробки завдань. Пул потоків дозволяє уникнути накладних витрат на створення та знищення потоків при кожному завданні, що сприяє покращенню продуктивності та зменшенню затрат пам'яті. Використання пулу потоків може покращити продуктивність програми та забезпечити більш плавну та швидку обробку завдань.

Основною ідеєю пулу потоків є те, що певна кількість потоків створюється на початку роботи програми та зберігається для майбутнього використання. Замість того, щоб створювати новий потік для кожного завдання, пул потоків використовує доступні потоки з пулу для обробки завдань. Це дозволяє уникнути затримок, пов'язаних зі створенням нових потоків, та забезпечити постійне виконання завдань.

Розглянемо основні компоненти пулу потоків:

a) **Потоки**. Пул потоків містить фіксовану кількість потоків. Ці потоки вже створені та готові для виконання завдань. Кількість потоків у пулі може бути налаштована залежно від вимог програми та обмежень ресурсів.

b) **Черга завдань**. Це структура даних, яка зберігає завдання, які потрібно виконати. Коли нове завдання надходить до пулу потоків, воно додається до черги. Потоки з пулу забирають завдання з черги та виконують їх послідовно.

c) **Управління ресурсами**. Пул потоків автоматично керує кількістю потоків та ресурсами, які використовуються. Він забезпечує оптимальне використання доступних потоків та уникнення перевантаження системи. Наприклад, якщо певний потік завершив виконання завдання, він може взяти нове завдання з черги, що дозволяє уникнути простою потоку та забезпечує постійну обробку завдань.

d) **Керування завданнями**. Пул потоків забезпечує спосіб посилання завдань на виконання та отримання результатів. Зазвичай це виконується за допомогою викликів функцій або інтерфейсів, які дозволяють додавати завдання до пулу та отримувати результати виконання. Для керування розподілу завдань можна використовувати алгоритми планування та балансувальник навантаження.

До переваг використання пулу потоків належать:

a) **Ефективне використання ресурсів.** Пул потоків дозволяє керувати кількістю потоків, що забезпечує ефективне використання обчислювальних ресурсів. Він уникне зайвого навантаження системи, яке виникає при створенні нових потоків для кожного завдання.

b) **Зменшення накладних витрат**. Перевага використання пулу потоків полягає в тому, що створення та видалення потоків здійснюється один раз під час ініціалізації пулу. Це зменшує накладні витрати на створення нових потоків для кожного завдання, що допомагає покращити продуктивність програми.

c) **Керування навантаженням**. Пул потоків забезпечує контроль навантаження системи, оскільки він може обмежити кількість одночасно виконуваних потоків. Це дозволяє уникнути перевантаження системи в умовах великого обсягу завдань.

2.1 Проблеми при розробці

Під час розробки бібліотеки пулу потоків можуть виникати різноманітні проблеми, які потребують уваги та вирішення.

Один з головних аспектів розробки бібліотеки пулу потоків - це ефективне управління потоками. Потрібно вирішити питання про створення, запуск, зупинку та видалення потоків у пулі. Окремі потоки повинні бути ефективно розподілені між завданнями та звільнені після завершення роботи. Потрібно також враховувати

можливість контролювати кількість потоків у пулі, залежно від навантаження та ресурсів системи.

Крім того, бібліотека потребує механізму для додавання та виконання завдань в потоковому пулі. Важливо розробити ефективну чергу завдань або іншу структуру даних для зберігання та управління невиконаними завданнями. Також потрібно мати можливість пріоритезувати завдання, щоб важливіші завдання виконувалися першими.

Розробка потокового пулу потребує обережного управління синхронізацією та безпекою. Багатопотокове середовище може призвести до перегонів за ресурсами, взаємного виключення та інших проблем, які впливають на коректність та продуктивність системи. Потрібно використовувати відповідні синхронізаційні примітиви, такі як блокування, м’ютекси або семафори, для запобігання конфліктам та забезпечення безпеки.

І звісно що, перед випуском бібліотеки потрібно виконати належне тестування та налагодження. Потрібно переконатися, що пул потоків працює правильно, ефективно та стабільно в різних сценаріях та навантаженнях. Тестування може включати симуляцію різних завдань, перевірку відповідності результатів, аналіз продуктивності та знайдення та усунення можливих помилок.

Ці проблеми вимагають уважного аналізу, проєктування та реалізації. Правильне розв'язання цих проблем допоможе створити ефективну та надійну бібліотеку.

3. АСИНХРОННЕ ПРОГРАМУВАННЯ

Асинхронне програмування - це підхід до розробки програмного забезпечення, де виконання завдань розділяється на незалежні фрагменти, які можуть виконуватися паралельно та асинхронно, без блокування основного потоку виконання програми. У традиційному синхронному програмуванні виконання одного завдання затримує виконання наступних, тоді як в асинхронному програмуванні можна продовжувати виконання інших завдань, не чекаючи завершення попереднього.

Асинхронні операції дозволяють виконувати задачі паралельно, що зменшує загальний час виконання. Крім того, використання асинхронного вводу/виводу (I/O) дозволяє програмі не затримуватися на операціях вводу/виводу, а продовжувати виконання інших завдань.

Також асинхронне програмування дозволяє легко масштабувати систему, оскільки можна виконувати багато операцій паралельно. Це особливо корисно в системах, які отримують велику кількість запитів від користувачів.

Асинхронне програмування дозволяє ефективно використовувати ресурси системи, оскільки воно не блокує виконання інших задач під час очікування відповіді на запит. Вільні ресурси можуть бути використані для виконання інших завдань.

Крім того, асинхронне програмування надає зручні механізми для керування паралельними завданнями, такими як обробка помилок, синхронізація та координація виконання.

Асинхронне програмування використовується в різних областях, включаючи мережеву комунікацію, веб-розробку, обробку подій, обчислення в реальному часі та багато іншого. Відповідно до особливостей конкретного додатка, розробники можуть використовувати різні інструменти та бібліотеки для реалізації асинхронного програмування, такі як асинхронні фреймворки, засоби мультипотоковості або асинхронні розширення мов програмування.

4. ОГЛЯД РИНКУ

Існує декілька популярних бібліотек для роботи з пулом потоків які дозволяють ефективно керувати та розподіляти виконання завдань у багатопотоковому середовищі. Нижче наведено огляд декількох з них.

4.1 Intel Threading Building Blocks (TBB)

TBB - це потужна кросплатформова бібліотека, розроблена компанією Intel, яка надає високорівневі конструкції для паралельного програмування. У складі TBB присутній компонент пулу потоків, який автоматично керує створенням та використанням потоків. Він надає абстракцію від низькорівневих операцій з потоками та дозволяє легко створювати паралельні версії алгоритмів. TBB забезпечує ефективне планування роботи та автоматичне масштабування залежно від кількості доступних ядер процесора.

4.2 Boost.Thread

Boost.Thread - це розширена версія стандартної бібліотеки потоків C++, яка надає додаткові можливості для паралельного програмування. У складі Boost.Thread присутній клас thread_pool, який дозволяє створювати пул потоків та керувати їх виконанням. Бібліотека Boost також надає різні інші корисні інструменти для роботи з потоками, включаючи синхронізаційні примітиви та алгоритми для паралельного обчислення.

4.3 PPL (Parallel Patterns Library)

PPL - це бібліотека, розроблена Microsoft, яка надає високорівневі шаблони та інструменти для паралельного програмування на C++. Вона включає компонент task_scheduler, який використовує пул потоків для виконання задач. PPL дозволяє легко створювати паралельні операції, використовуючи шаблони Parallel For, Parallel For Each та інші. Вона також надає зручний інтерфейс для роботи зі

структурами даних, такими як вектори та хеш-таблиці, забезпечуючи автоматичне розпаралелювання обчислень.

4.4 ThreadPool (C++11)

ThreadPool - це проста бібліотека, яка надає реалізацію пулу потоків за допомогою нових функцій C++11. Вона має простий інтерфейс та легка у використанні. Бібліотека дозволяє створювати пул потоків заданого розміру, надсилати завдання на виконання та отримувати результати.

5. ТЕХНОЛОГІЇ

5.1 C++

C++ - це високорівнева, загальнопризначена мова програмування, яка поєднує в собі можливості мови C з додатковими функціями та функціональністю, що дозволяє робити більш ефективну та зручну розробку програмного забезпечення. Мова C++ була розроблена у 1980-х роках Бьорном Страуструпом як розширення мови C з метою підвищення продуктивності, масштабованості та зручності програмування.

Основні риси C++:

a) **Об'єктноорієнтований підхід**. C++ підтримує парадигму об'єктноорієнтованого програмування (ООП), що дозволяє створювати класи, об'єкти, спадковість, поліморфізм та інші концепції ООП. Це дозволяє розробникам організовувати код у вигляді логічних модулів та забезпечувати більшу структурованість та повторне використання коду.

b) **Низькорівневі можливості**. C++ дозволяє безпосередньо керувати ресурсами системи та працювати з пам'яттю, вказівниками, адресами, бітовими операціями тощо. Це дає можливість оптимізувати програми для досягнення кращої продуктивності та керувати ресурсами системи.

c) **Шаблони (англ. Templates)**. C++ включає потужну систему шаблонів, яка дозволяє створювати загальні алгоритми та контейнери, що працюють з різними типами даних. Шаблони дозволяють писати гнучкий та перевикористовуваний код.

d) **Ефективність**. C++ дозволяє розробникам створювати ефективне програмне забезпечення. Вона надає можливість безпосередньо маніпулювати пам'яттю та оптимізувати код для кращої продуктивності. C++ підтримує статичну типізацію та компіляцію, що дозволяє виявляти помилки на етапі компіляції та генерувати ефективний машинний код.

e) **Розширюваність**. C++ дозволяє використовувати код мови C та інших мов програмування, а також розширювати мову за допомогою власних бібліотек та фреймворків.

C++ широко використовується для розробки різноманітних програм, включаючи системне програмування, розробку вбудованого програмного забезпечення, веб-додатки, графічні ігри, наукові обчислення та багато іншого. Вона є однією з найпопулярніших мов програмування та знаходить застосування у багатьох сферах програмування.

5.2 CMake

CMake - це крос-платформова система автоматизованої збірки та конфігурації програмного забезпечення. Вона надає простий та ефективний спосіб описувати, керувати та здійснювати збірку проєктів у різних операційних системах та середовищах розробки. CMake заснована на декларативному підході до конфігурації проєктів, що дозволяє програмістам описувати залежності, налаштування компілятора та інші параметри у незалежному від платформи форматі.

Основні переваги використання CMake:

a) **Крос-платформовість**. CMake підтримує багато операційних систем, включаючи Windows, macOS, Linux та інші. Це дозволяє розробляти проєкти на різних платформах, не залежно від конкретного середовища розробки.

b) **Простота використання**. CMake використовує простий та лаконічний синтаксис, що дозволяє легко описувати проєкти та їх залежності. Файли конфігурації CMake пишуться у текстовому форматі, що робить їх легкими для зберігання та версійності.

c) **Модульність**. CMake надає можливість організовувати проєкти у модулі та бібліотеки. Це дозволяє розбити великі проєкти на менші компоненти, що спрощує розробку, тестування та підтримку коду.

d) **Інтеграція з різними середовищами розробки**. CMake підтримує інтеграцію з різними середовищами розробки, такими як Visual Studio, Xcode, Eclipse та інші. Це дозволяє розробникам використовувати звичні інструменти для розробки та налагодження своїх проєктів.

e) **Підтримка зовнішніх бібліотек**. CMake дозволяє легко включати зовнішні бібліотеки до проєкту. Це спрощує розробку проєктів, які використовують сторонні компоненти, і дозволяє легко керувати залежностями.

f) **Налаштування компілятора та оптимізації**. CMake надає можливість встановлювати параметри компіляції, оптимізації та діагностичних інструментів для проєкту. Це дозволяє використовувати оптимальні налаштування компілятора для конкретної платформи та вимог проєкту.

g) **Система модулів**. CMake має розширену систему модулів, що дозволяє використовувати готові модулі для розробки різних функціональностей. Це включає підтримку пакетного менеджера, збірку документації, тестування, інсталяцію програмного забезпечення та багато іншого.

h) **Підтримка декількох конфігурацій**. CMake дозволяє визначати різні конфігурації для збірки проєкту, такі як режими відладки (англ. debug) та релізу (англ. release), різні платформи та набори параметрів. Це спрощує управління різними варіантами збірки та полегшує розповсюдження готового продукту.

Узагальнюючи, CMake є потужним інструментом для автоматизації збірки та конфігурації проєктів. Він надає простоту використання, крос-платформову підтримку та розширені можливості для розробки програмного забезпечення на різних платформах. CMake є популярним вибором серед розробників C++ та інших мов програмування, які цінують ефективну та гнучку систему збірки.

5.3 gTest

gTest, або Google Test, є потужним фреймворком для автоматизованого тестування програмного забезпечення мовою програмування C++. Він розроблений компанією Google і надає зручні інструменти для написання, виконання та аналізу тестових сценаріїв.

Основні особливості та переваги gTest:

a) **Простота використання**. gTest надає чіткий та легко зрозумілий інтерфейс, що спрощує процес написання тестів. Він використовує простий синтаксис, що дозволяє швидко створювати тести та перевіряти очікувані результати.

b) **Багатофункціональність**. gTest підтримує широкий спектр функціональності для тестування. Це включає стандартні перевірки (наприклад, перевірка рівності, нерівності, включення тощо), асерти (англ. assert) та перевірку виключень.

c) **Розширені можливості**. gTest надає розширені можливості для організації тестів у набори (англ. test suites), групи (англ. test groups) та категорії (англ. test categories). Це дозволяє гнучко організувати тести та виконувати їх у зручному порядку.

d) **Параметризовані тести**. gTest підтримує параметризовані тести, що дозволяють виконувати однаковий тестовий сценарій з різними вхідними параметрами. Це забезпечує широкі можливості для тестування різних варіантів вхідних даних.

e) **Вбудовані інструменти для аналізу**. gTest надає вбудовані засоби для аналізу результатів тестів, включаючи звіти про успішність, виведення деталей про помилки та час виконання кожного тесту. Це допомагає легко виявляти проблеми та швидко знаходити їх причини.

f) **Інтеграція з CMake**. gTest добре інтегрується з системою збірки CMake, що спрощує процес включення тестів у проєкт. Він може бути

використаний для автоматичного виконання тестів під час збірки проєкту або окремо як окрема ціль.

g) **Крос-платформова підтримка**. gTest підтримує роботу на різних платформах, включаючи Windows, Linux та macOS. Це дозволяє використовувати його в різних проєктах, незалежно від обраної платформи розробки.

Узагальнюючи, gTest є потужним фреймворком для автоматизованого тестування мовою програмування C++. Він надає зручні інструменти для написання, виконання та аналізу тестових сценаріїв, дозволяючи розробникам легко перевіряти правильність роботи свого програмного забезпечення.

5.4 Visual Studio

Visual Studio є інтегрованою середовищем розробки (англ. intergrated development environment), розробленим компанією Microsoft, яке надає розширені можливості для розробки програмного забезпечення на різних мовах програмування, включаючи C++, C#, Visual Basic і багато інших. Вона стала одним з найпопулярніших інструментів розробки програмного забезпечення та заслужила репутацію потужного та універсального середовища.

Основні особливості Visual Studio:

a) **Мови програмування**. Visual Studio підтримує широкий спектр мов програмування, що дозволяє розробникам вибирати мову, яка найкраще підходить для їх проєкту. Це включає популярні мови, такі як C++, C#, Visual Basic, а також JavaScript, Python, F# та багато інших.

b) **Інтегрована розробка**. Visual Studio надає повноцінну розробку, що включає редактор коду з підсвічуванням синтаксису, автодоповненням, перевіркою правильності коду та іншими корисними функціями. Вона також надає зручний інтерфейс для керування проєктами, компіляції, налагодження, відстеження версій та інших аспектів розробки.

c) **Візуальний дизайнер**. Visual Studio має вбудовані візуальні дизайнери для різних типів програмного забезпечення, таких як Windows Forms, WPF, ASP.NET та інші. Це дозволяє розробникам швидко створювати інтерфейси користувача за допомогою перетягування та розміщення елементів.

d) **Налагодження**. Visual Studio надає потужні засоби для налагодження програмного забезпечення. Розробники можуть встановлювати точки зупину, стежити за змінними, виконувати крок за кроком та аналізувати стек викликів, щоб виявити й усунути помилки.

e) **Розширення та плагіни**. Visual Studio дозволяє розробникам розширювати його функціональність за допомогою плагінів та розширень. Є велика кількість сторонніх розширень, які додають нові можливості, покращують продуктивність розробки та працюють з різними фреймворками та інструментами.

f) **Інтеграція з іншими інструментами Microsoft**. Visual Studio легко інтегрується з іншими інструментами Microsoft, такими як Azure, Team Foundation Server, SQL Server та інші. Це спрощує розробку, розгортання та управління проєктами.

Узагальнюючи, Visual Studio є потужним інтегрованим середовищем розробки, яке надає широкі можливості для розробки програмного забезпечення на мові C++ та інших мовах програмування. Вона має вбудовані інструменти для розробки, налагодження, тестування та аналізу програмного забезпечення, що допомагає розробникам підвищити продуктивність та якість своїх проєктів.

6. АРХІТЕКТУРА

На рисунку 1 представлена повна діаграма класів бібліотеки з усіма взаємодіями. Більшість компонентів буде розглянуто далі. Цей рисунок відображає лише високорівневу архітектуру бібліотеки.

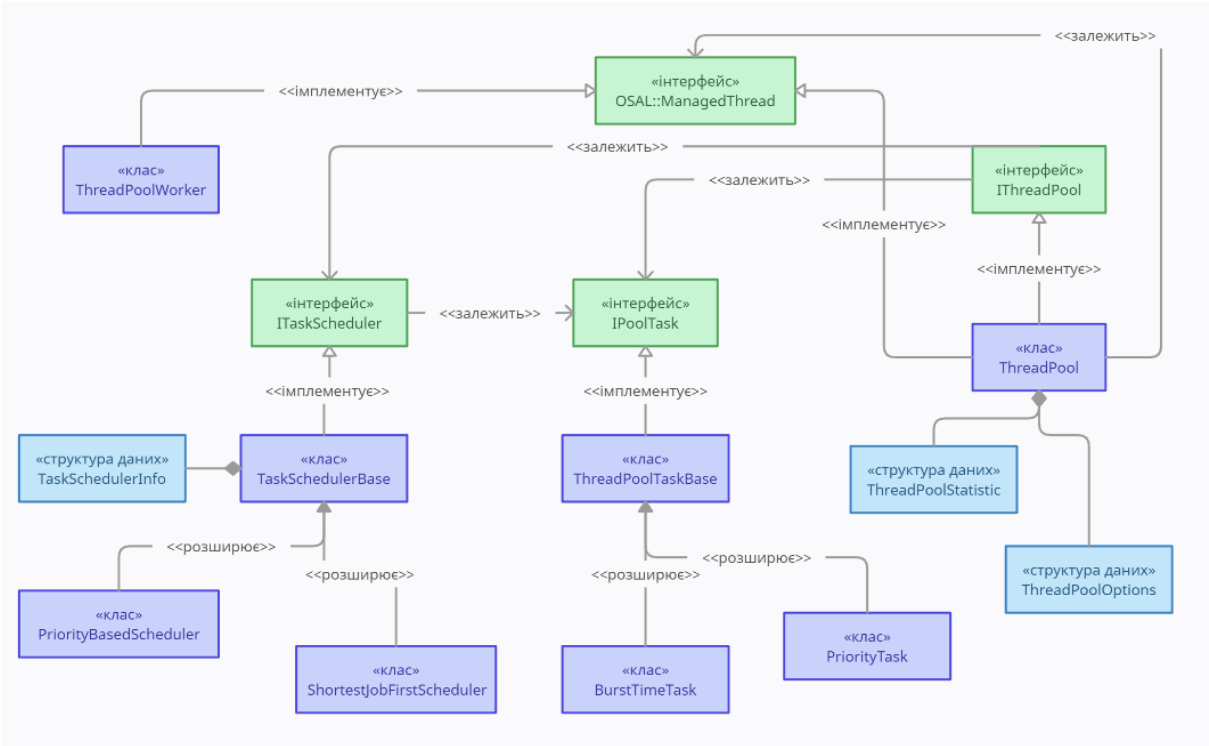

Рисунок 1 - Діаграма класів

Для бібліотеки пулу потоків я використовую контейнер “спеціальних потоків”, які відповідають за виконання “спеціальних завдань”.

Цей “спеціальний потік” називається ThreadPoolWorker, а “спеціальні завдання” представлені реалізацією ієрархії класів з інтерфейсом IThreadPoolTask.

6.1 ThreadPoolWorker

ThreadPoolWorker, це користувацький потік, який керує завданнями за допомогою різних алгоритмів планування, які інкапсульовані всередині реалізації ієрархії класів з інтерфейсом ITaskScheduler. На рисунку 2.1 представлено діаграму станів, через які проходить ThreadPoolWorker під час виконання, а на рисунку 2.2 деталі класу.

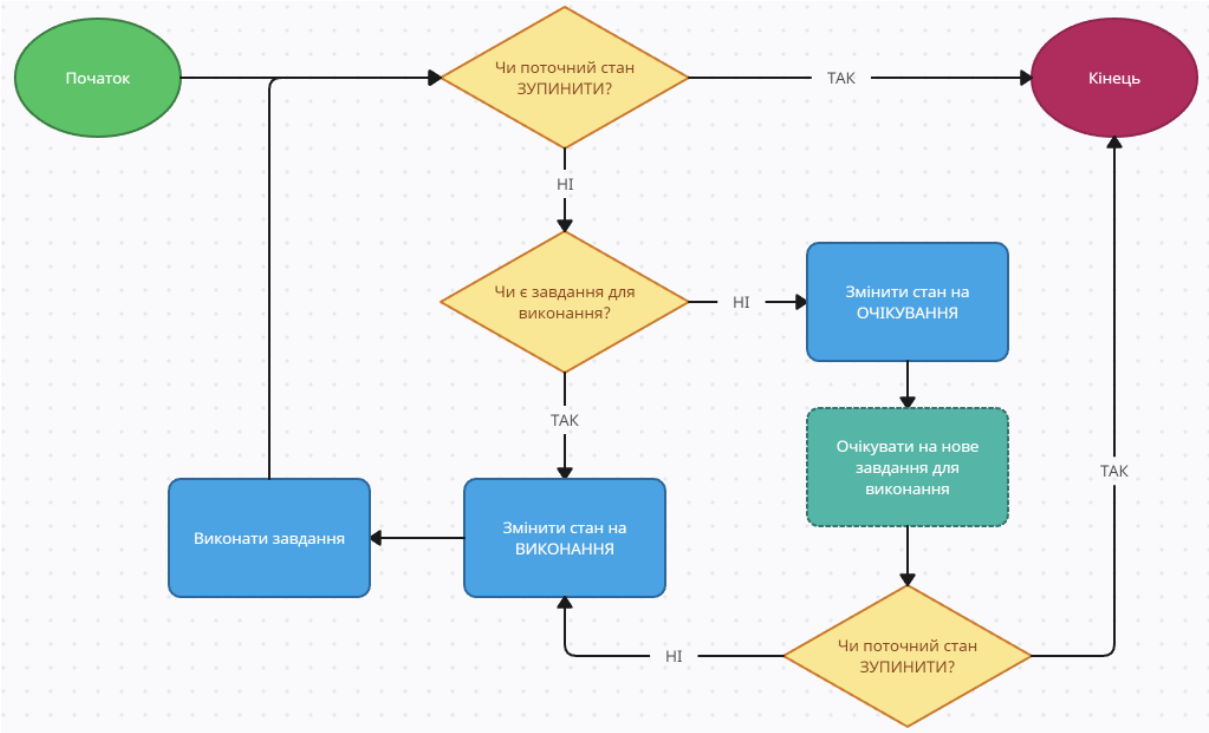

Рисунок 2.1 - Діаграма станів ThreadPoolWorker

ThreadPoolWorker реалізує інтерфейс OSAL::ManagedThread, оскільки в ThreadPool я хочу мати повний контроль над поточним потоком (призупинення, відновлення тощо).

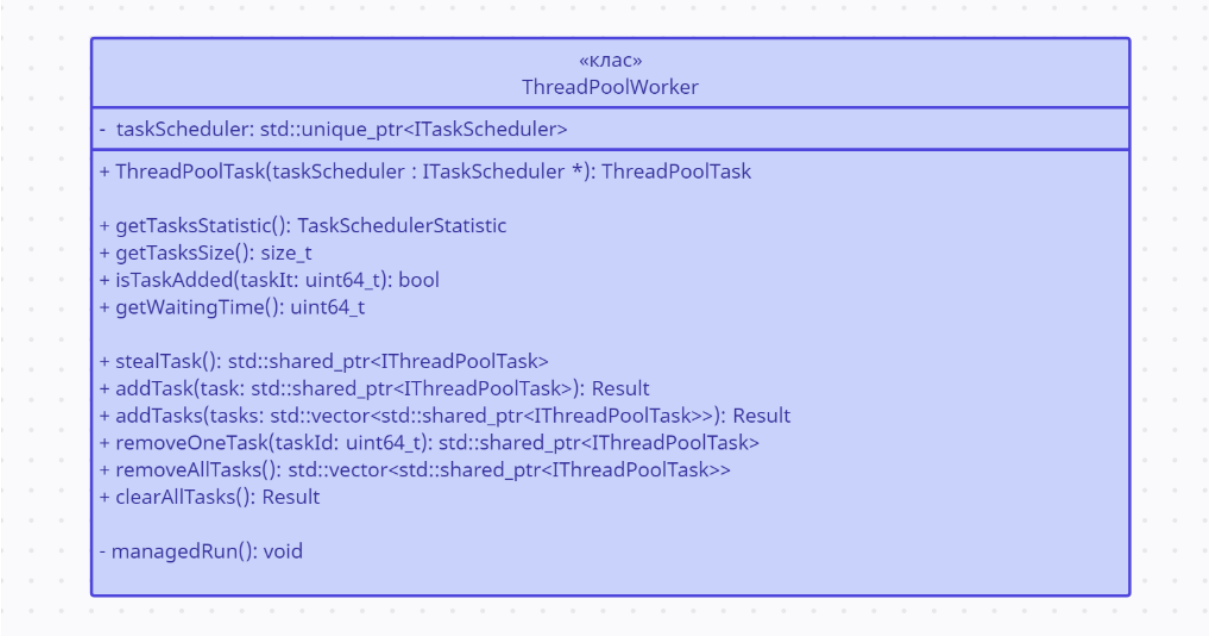

Рисунок 2.2 – Діаграма деталей класу ThreadPoolWorker

6.2 IThreadPoolTask

Інтерфейс IThreadPoolTask надає методи для виконання, скасування та отримання внутрішніх властивостей виконуваного завдання.

Реалізація заснована на std::function і std::future, тому користувач може помістити будь-яку функцію в конструктор IThreadPoolTask і зможе отримати результат виконання після цього.

Є кілька властивостей, які допомагають контролювати виконання та відстежувати завдання:

a) стан (Виконання, Скасування, тощо)

b) ідентифікатор

Очікується кілька реалізацій IThreadPoolTask, які базуються на пріоритеті або часу виконання. На рисунку 3 зображено деталі ієрархії класів.

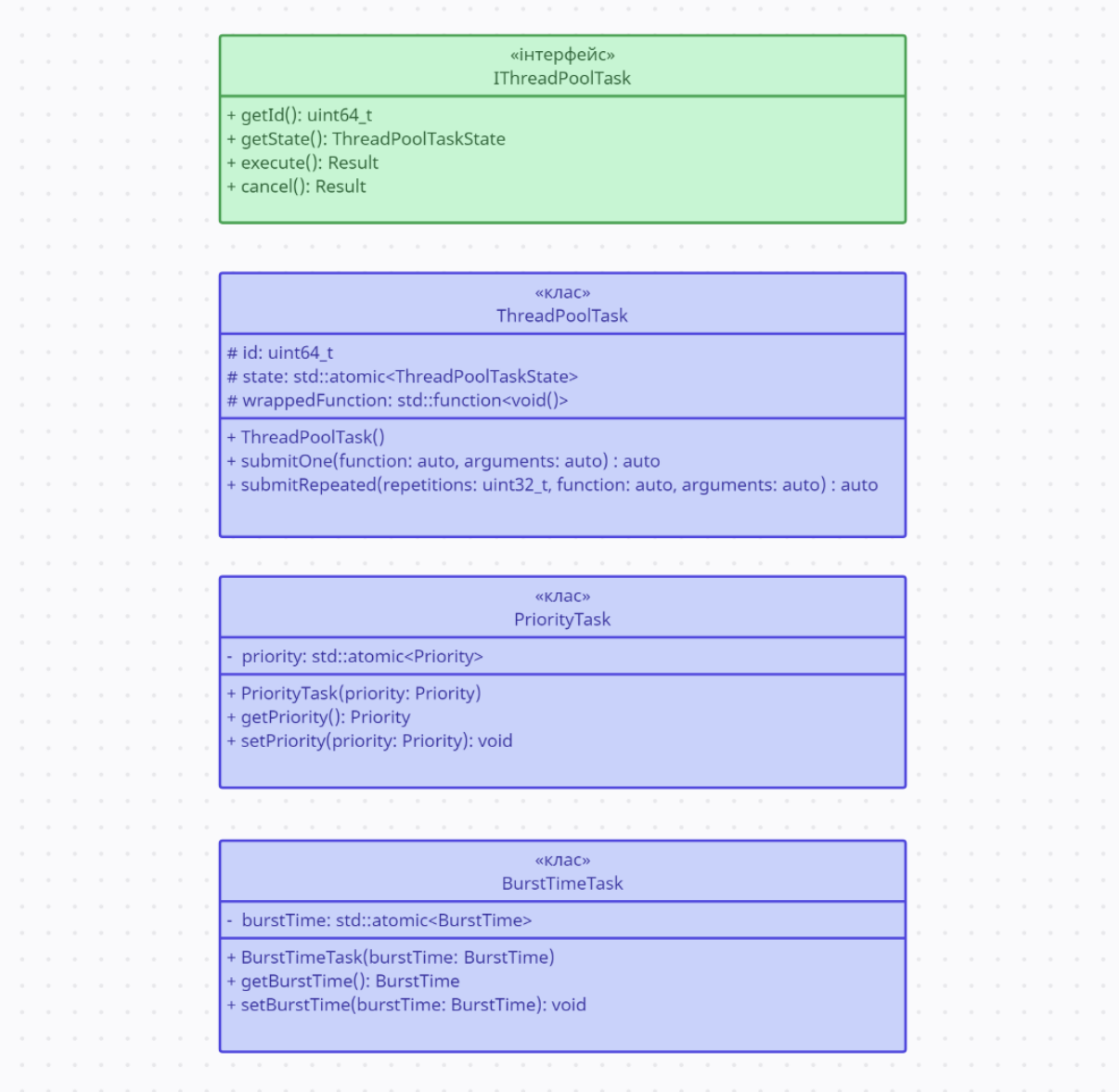

Рисунок 3 – Діаграми ієрархії класів IThreadPoolTask

6.3 ITaskScheduler

Інтерфейс ITaskScheduler надає методи для планування/скасування завдання та отримання наступного завдання для виконання.

Тобто, ITaskScheduler не виконує завдання, а просто надає їх для виконання. А виконання поставленого завдання є обов'язком клієнта (в цьому випадку клієнтом буде ThreadPoolWorker).

Різні алгоритми планування вже були розглянуті, проте для цієї роботи було вибрано лише декілька із них (FCFS, SJF, Пріоритетне планування). Кожен алгоритм використовуватиме власну реалізацію IThreadPoolTask.

На рисунку 4 зображено деталі інтерфейсу ITaskScheduler.

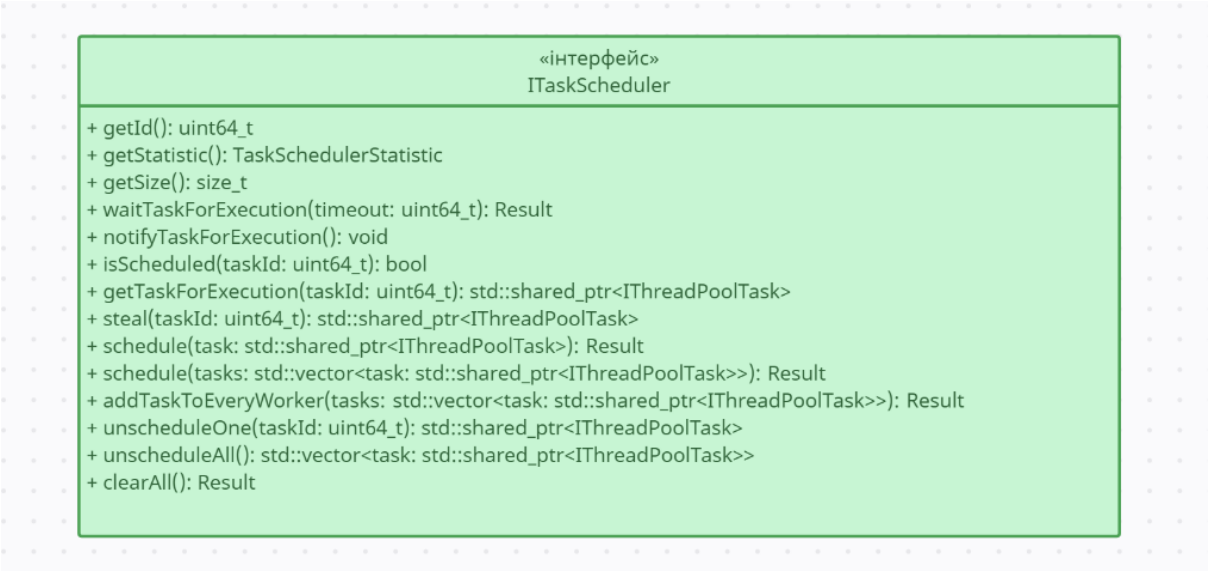

Рисунок 4 - Діаграма деталей інтерфейсу ITaskScheduler

ITaskScheduler має 3 різні сценарії методів, які видаляють завдання з локальної черги:

a) unscheduleOne/unscheduleAll - видаляє завдання за певним ідентифікатором і повертає його.

b) getTaskForExecution - видаляє завдання, яке має бути виконано наступне за певним алгоритмом і повертає його.

c) steal - видаляє завдання, яке повинне виконуватися останнє відповідно до певного алгоритму і повертає його.

6.4 IThreadPool

ThreadPool також реалізує інтерфейс OSAL::ManagedThread для балансування навантаження в окремому потоці. Тобто, у перевизначеному методі managedRun я виконую метод loadBalance з ThreadPool.

Таким чином я можу залишити інтерфейс незмінним. Будь-який успадкований від ThreadPool клас повторно реалізує метод loadBalance, навіть не знаючи про OSAL::ManagedThread.

Коли користувач хоче створити ThreadPool, він може передати певні параметри, а одним із цих параметрів є відкладення виконання. Це означає, що

запуск ThreadPool буде відкладено, доки користувач не викличе startExecution у ThreadPool. На рисунку 5.1 зображено діаграму деталей інтерфейсу IThreadPool та похідного класу ThreadPool.

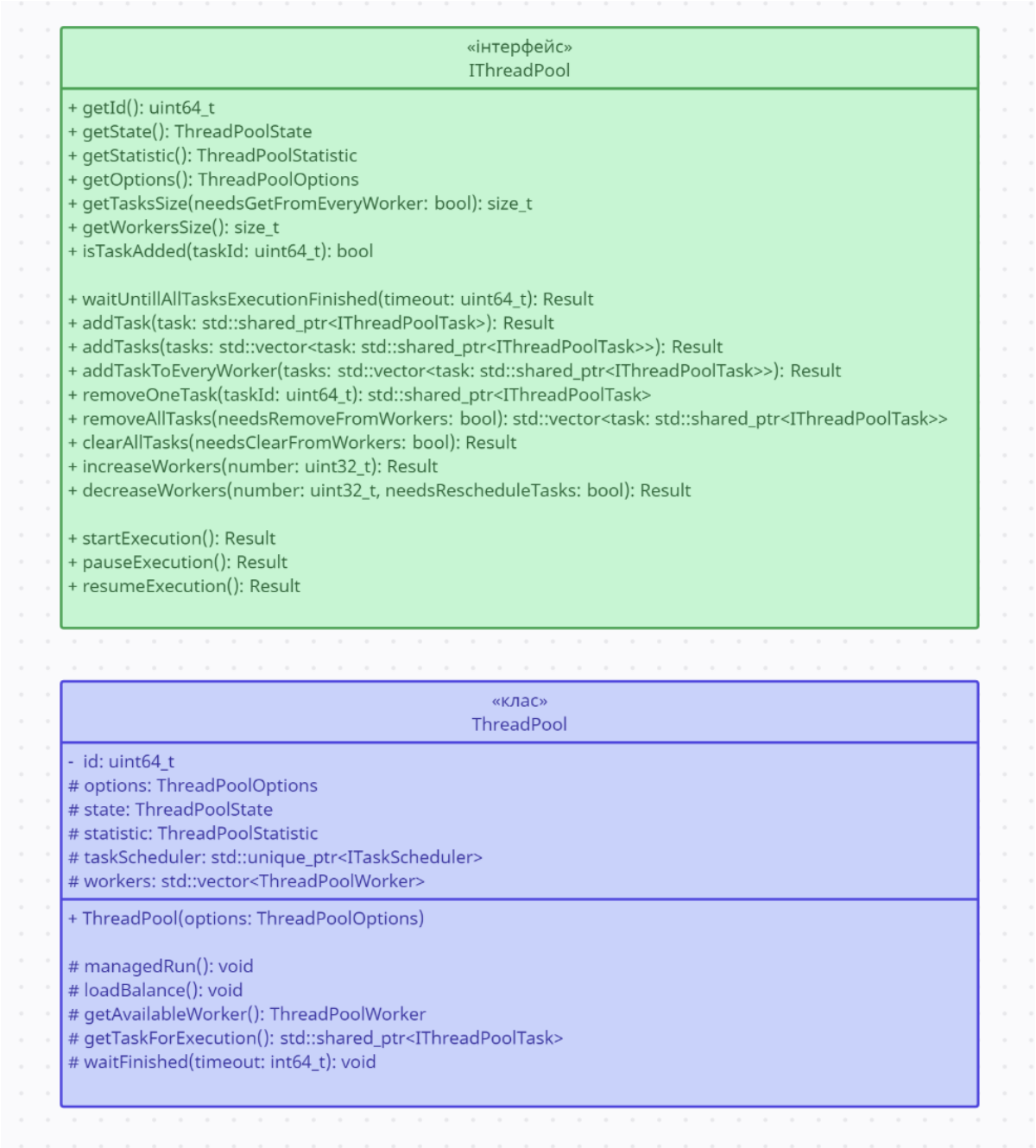

Рисунок 5.1 - Деталі ієрархії класів IThreadPool

Також, на рисунку 5.2 зображена блок-схема балансування навантаження, яка присутня в ThreadPool.

Усього є одне очікування та декілька умов розблокування:

a) З'являється нове завдання

b) Деякі потоки повідомляють про доступність

Якщо час очікування минув (означає відсутність нового завдання та нового доступного потоку), виконується періодичне балансування навантаження. Також балансування навантаження виконується після кожного нового прибулого повідомлення.

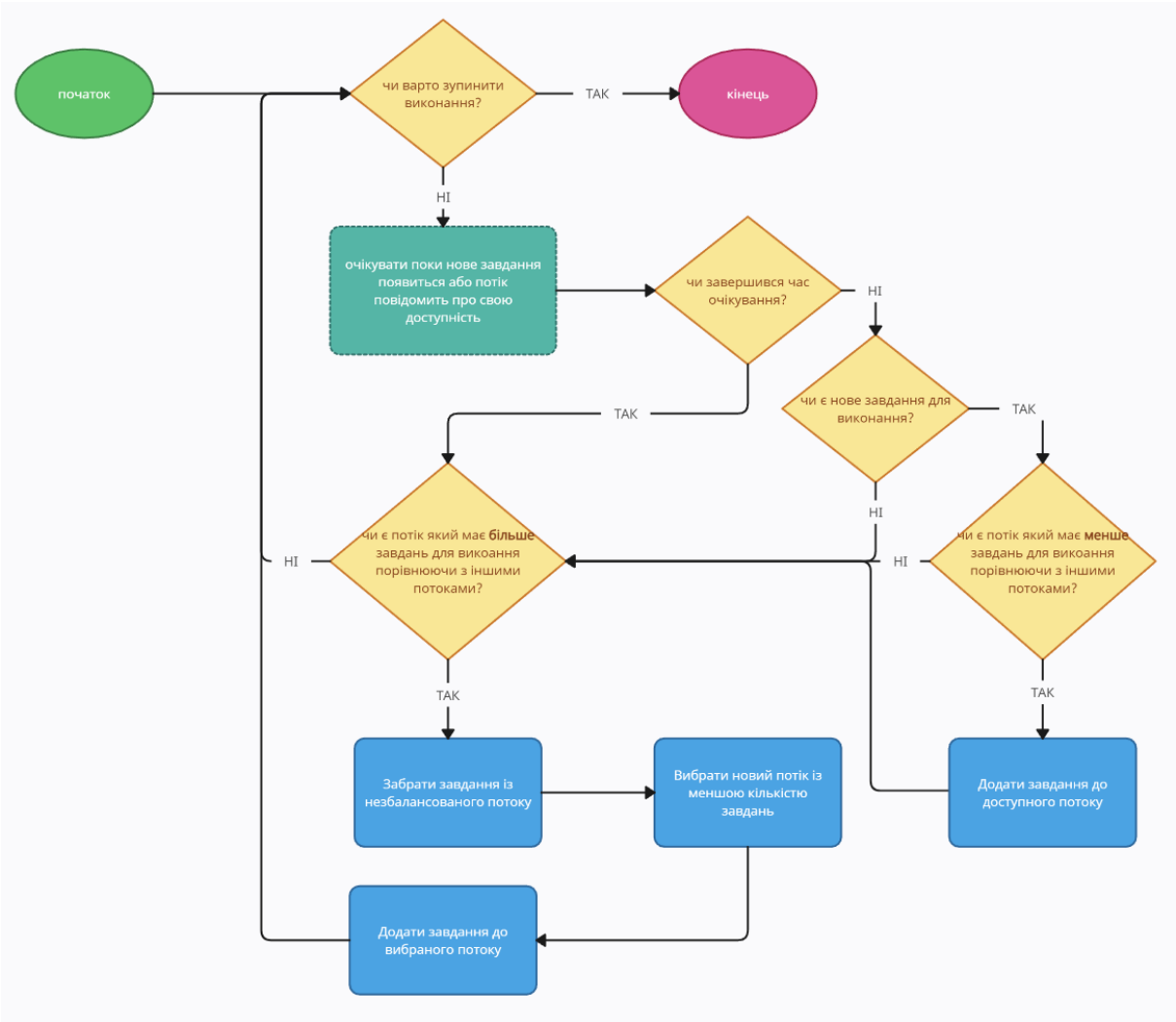

Рисунок 5.2 - Діаграма балансування навантаження в ThreadPool

Наступні діаграми зображені на рисунках 5.3 та 5.4 зображають послідовність створення об’єкту ThreadPool та одного виконання завдання.

Як згадувалося раніше, балансування навантаження в ThreadPool виконується в окремому потоці (це не показано на схемі). Крім того, тут чітко показано керування ThreadPoolWorker і взаємодію з ITaskScheduler у формі

реального окремого потоку операційної системи (ОС). Оскільки ThreadPoolWorker є просто оболонкою над потоком реальної ОС.

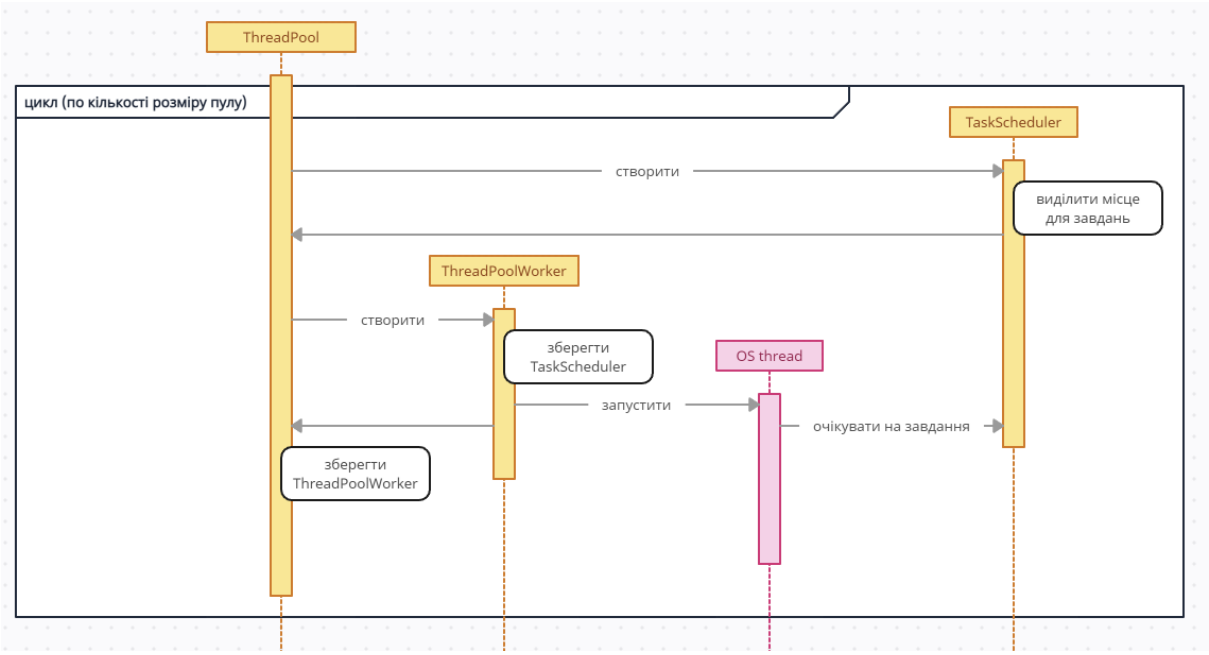

Рисунок 5.3 - Створення об'єкту класу ThreadPool

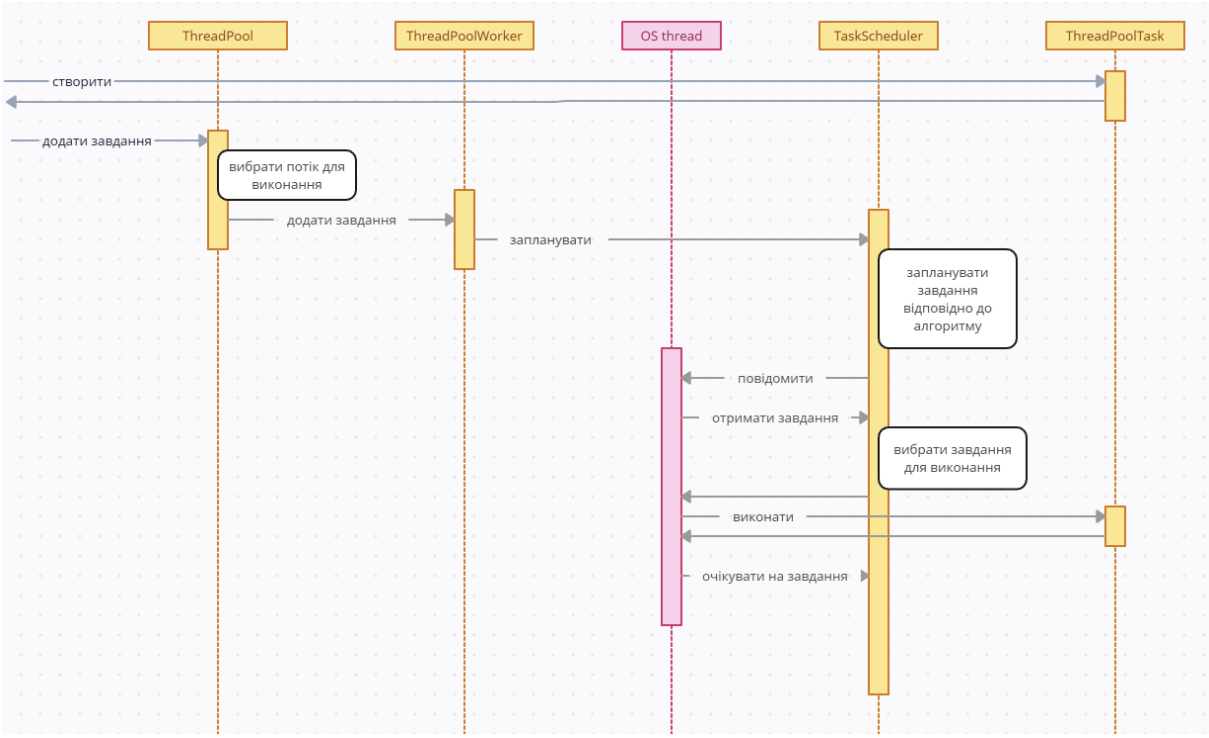

Рисунок 5.4 - Виконання завдання у створеному ThreadPool

ВИСНОВКИ

У цій дипломній роботі була розроблена та реалізована бібліотека пулу потоків мовою програмування C++. Виконані експерименти та аналіз показали, що використання потокового пулу може покращити продуктивність багатопотокових програм, забезпечуючи ефективну роботу з паралельними завданнями.

Основні переваги розробленої бібліотеки включають простий та зручний інтерфейс, автоматичне керування потоками, гнучкі налаштування та можливість контролювати навантаження системи.

Процес розробки бібліотеки включав проєктування архітектури, реалізацію основних функціональних модулів, валідацію та тестування. Застосування бібліотеки може значно спростити роботу з багатопотоковими програмами та покращити їх продуктивність. Далі розробка може включати розширення функціональності, оптимізацію та підтримку інших мов програмування.

ВИКОРИСТАНІ РЕСУРСИ

a) Processes and Threads [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/processes-and-threads
b) About Processes and Threads [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads
c) Multitasking [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/multitasking
d) Adavantanges of Multitasking [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/advantages-of-multitasking
e) When to Use Multitasking [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/when-to-use-multitasking
f) Multitasking Considerations [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/multitasking-considerations
g) Scheduling [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling
h) Scheduling Priorities [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities
i) Context Switches [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/context-switches
j) Multiple Processors [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/multiple-processors
k) Multiple Threads [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/multiple-threads

l) Thread Stack Size [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/thread-stack-size
m) Synchronizing Execution of Multiple Threads [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/synchronizing-execution-of-multiple-threads
n) Thread Local Storage [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/thread-local-storage
o) Thread Pools [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pools
p) Thread Pool API [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pool-api
q) Thread Pooling [Електронний ресурс] / Bridge K., Sharkey K., Satran M. // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pooling
r) Concurrency support library [Електронний ресурс] // cppreference. - 2022. – Режим доступу: https://en.cppreference.com/w/cpp/thread
s) Windows with C++ - The Evolution of Synchronization in Windows and C++ [Електронний ресурс] / Kerr K. // Microsoft Documentation. - 2012. – Режим доступу: https://learn.microsoft.com/en-us/archive/msdn-magazine/2012/november/windows-with-c-the-evolution-of-synchronization-in-windows-and-c
t) Operating System - Process Scheduling [Електронний ресурс] // tutorialspoint. – Режим доступу: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm
u) Operating System - Scheduling algorithms [Електронний ресурс] // tutorialspoint. – Режим доступу: https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm
v) Operating System - Multi-Threading [Електронний ресурс] // tutorialspoint. – Режим доступу: https://www.tutorialspoint.com/operating_system/os_multi_threading.htm
w) CPU Scheduling in Operating Systems [Електронний ресурс] // geeksforgeeks. – Режим доступу: https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/

x) Load balancing (computing) [Електронний ресурс] // Wikipedia. – Режим доступу: https://en.wikipedia.org/wiki/Load_balancing_(computing)

y) Load Balancing Approach in Distributed System [Електронний ресурс] // geeksforgeeks. – Режим доступу: https://www.geeksforgeeks.org/load-balancing-approach-in-distributed-system/

z) How to design a thread pool in C++ [Електронний ресурс] // SoByte. - 2022. – Режим доступу: https://www.sobyte.net/post/2022-05/design-a-thread-pool/

aa) Operating Systems [Електронний ресурс] / Joshy J. // YouTube. - 2017. – Режим доступу: https://www.youtube.com/watch?v=vBURTt97EkA&list=PLBlnK6fEyqRiVhbXDGLXDk_OQAeuVcp2O&pp=iAQB

bb) ThreadPool [Електронний ресурс] / Progsch J., Haisman V. // GitHub. - 2013. – Режим доступу: https://github.com/progschj/ThreadPool

cc) thread_pool [Електронний ресурс] / Christopher M. // Boost Documentation. - 2021. – Режим доступу: https://www.boost.org/doc/libs/1_78_0/doc/html/boost_asio/reference/thread_pool.html

dd) Intel Threading Building Blocks [Електронний ресурс] // Wikipedia. – Режим доступу: https://en.wikipedia.org/wiki/Threading_Building_Blocks

ee) Parallel Patterns Library (PPL) [Електронний ресурс] / Whitney T., Sharkey K., Robertson C., Jones M., Blome M., Hogenson G., Cai S // Microsoft Documentation. - 2021. – Режим доступу: https://learn.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170

ff) C++ [Електронний ресурс] // Wikipedia. – Режим доступу: https://en.wikipedia.org/wiki/C%2B%2B

gg) CMake Tutorial [Електронний ресурс] // CMake Documentation. – Режим доступу: https://cmake.org/cmake/help/latest/guide/tutorial/index.html

hh) Googletest [Електронний ресурс] // GitHub. - 2023. – Режим доступу: https://github.com/google/googletest

ii) Visual Studio [Електронний ресурс] // Wikipedia. – Режим доступу: https://en.wikipedia.org/wiki/Visual_Studio