

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

Кафедра прикладної математики

Дипломна робота

Створення каналів як методу комунікації у багатопроцесорних системах

Виконав: студент групи ПМП-42
спеціальності
113 - прикладна математика

Волобуєв А.І.

(прізвище та ініціали)

Керівник Дяконюк Л.М.

(прізвище та ініціали)

Рецензент _____

(прізвище та ініціали)

Львів - 2023

Анотація

Предмет цієї дипломної роботи полягає у створенні програми, яка реалізує примітив синхронізації, відомий як канал. Для реалізації буде використано відкриту бібліотеку YASLib. Новизна роботи полягає у тому, щоб перевірити і проаналізувати нові можливості мови C++20, а саме корутини, на основі яких будуть створені канали. Також буде здійснене порівняння з кількома іншими реалізаціями і аналіз lockfree алгоритму реалізації каналів.

Зміст

Зміст	3
Вступ	5
Розділ 1. Загальні положення	7
1.1. Корутини	7
1.2. Організація кешів процесора	9
1.3. Протоколи підтримки когерентності	11
1.3.1. Стани MESI	11
1.3.2. Повідомлення протоколу MESI	12
1.4. Канали	14
Розділ 2. Аналіз різних підходів до реалізації	16
2.1. Lockfree (без блокувань) канали	16
2.2. Go-канали	21
2.2.1. Небуферизований канал	21
2.2.2. Буферизований канал	21
Розділ 3. Реалізація програми	23
3.1. Корутини C++20	23
3.1.1. Awaiters	24
3.2. Код програми	25
3.3. Аналіз реалізації	29
3.3.1. Небуферизований канал	29
3.3.2. Буферизований канал	30
3.4. Швидкодія програми	32
3.4.1. Результати замірів	32
3.4.2. Код замірів	34

Висновок.....	36
Список використаних джерел.....	37

Вступ

Програмування багатопотокових систем завжди було складним викликом для будь-якого програміста. Історично склалося, що складності виникли з кількох причин:

1. Велика ціна і рідкісність багатоядерних систем.
2. Типова відсутність у дослідника чи програміста досвіду з багатоядерними системами.
3. Недостатність публічно доступного багатопотокового коду.
4. Відсутність широкодоступної навчальної програми з багатопотокового програмування.
5. Велика ціна комунікації відносно до ціни обробки даних, навіть в системах з розподіленням пам'яті.

Більша частина з цих складностей на поточний момент усунена. За останні кілька десятиліть ціна багатопроцесорних систем різко впала, дякуючи закону Мура[1]. Перші дослідження на тему переваг багатоядерних комп'ютерів з'явилися ще у 1996 р.[2]. Вже в 2008-му році стало складно знайти домашній комп'ютер з менш ніж двома ядрами. Починаючи з 2012-го року, навіть в смартфонах почали з'являтися багатоядерні системи.

Наявність багатоядерних систем невеликої ціни призвела до того, що колись рідкісна можливість отримати доступ до такої системи стала можливою майже для кожного. По суті це є по кишені навіть студенту.

Окрім того, якщо в 20-му столітті більша частина багатопотокових програм була комерційною, у 21-му столітті з'явилася велика кількість публічно відкритих подібних систем. Прикладом цього може слугувати операційна система Linux[3], бази даних[4], системи передачі повідомлень[5].

На жаль, велика ціна комунікації між потоками відносно обробки даних все ще залишається важкою перешкодою для розвитку багатоядерних систем.

Метою цієї роботи є реалізація ефективного способу комунікації, який би враховував внутрішнє влаштування процесора.

Метою дослідження є ефективні способи реалізації примітиву синхронізації, відомого як канали. Саме канали є таким способом комунікації, який має бути найбільш сприятливим для сучасного процесора.

Предметом дослідження є нові засоби для побудови каналів та їх ефективність.

Практична цінність отриманих результатів полягає у перевірці нових інструментів для створення подібних примітивів синхронізації, використаних для реалізації, а також аналізі відомих алгоритмів і підходів.

Розділ 1. Загальні положення

1.1. Корутини

Термін корутина був уперше вжитий Мелвіном Конвеєм у 1958 році [6]. Для того щоб перейти до нього, спершу потрібно розглянути термін «функція» з точки зору асемблеру. Для цього введемо кілька понять.

У більшості сучасних систем існує таке поняття як стек. У теорії обчислювальних систем – це структура даних, яка працює за принципом LIFO, що зберігає інформацію для повернення з функцій. Стек використовується також для передачі аргументів функції та виділення локальної пам'яті.

Функція – це по суті набір команд. Кожна функція має свій стековий фрейм. При виклику функції відбувається наступне: в залежності від угоди про виклики, аргументи функції та адреса її повернення зберігаються на стековому фреймі функції, коли вона завершує свої інструкції, відбувається повернення по адресі, записаній на стеці, та очистка усієї інформації.

Для опису доступних дій для функції уведемо наступні терміни: `suspend`, `activate`, `terminate`, `yield`. `suspend` – це призупинення виконання поточної програми. `activate` – це активація нового стекового фрейму. `terminate` – це очистка стекового фрейму, «знищення» функції. `yield` – це повернення управління з функції.

Очевидно, виклик функції можна описати у вищезазначених термінах таким чином: `suspend + activate`. Ми призупиняємо виконання програми, а потім активуємо стековий фрейм нової функції. Повернення з функції ж описується так: `terminate + yield`. Ми знищуємо поточний стековий фрейм і віддаємо управління.

Щоб перейти до корутин, уведемо нову операцію: `suspend + yield`. Практичне значення цієї операції полягає в тому, щоб призупинити виконання і повернути управління. Наприклад, призупинити виконання поточної функції і повернути з неї управління. Повернення ж до виконання цієї функції здійснюється так само: призупиняється виконання поточного блоку інструкцій(`suspend`) і здійснюється передача управління(`yield`). Наявність окрім операції виклику та операції повернення подібної операції робить зі звичайної функції корутину.

Використання корутин приводить нас до нового підходу до багатозадачності. Такий підхід у багатоядерних системах, зазвичай, називають кооперативною багатозадачністю: функція виконується до тих пір, поки не вирішить здійснити `suspend + yield`, тобто добровільно передати управління якійсь іншій функції.

1.2. Організація кешів процесора

Розглянемо влаштування сучасних багатоядерних систем. Сучасні процесори набагато швидші, ніж сучасні системи пам'яті. Ще у 2006-му році процесор міг здійснити 10 операцій в наносекунду, в той час як виконання читання з пам'яті вимагає кілька десятків мілісекунд. Ця різниця у швидкості призвела до появи кешів у сучасних процесорах (рис. 1)[7]. Ці кеші прив'язані до конкретної обчислювальної одиниці (CPU) і, зазвичай, доступ до них здійснюється за кілька процесорних циклів.

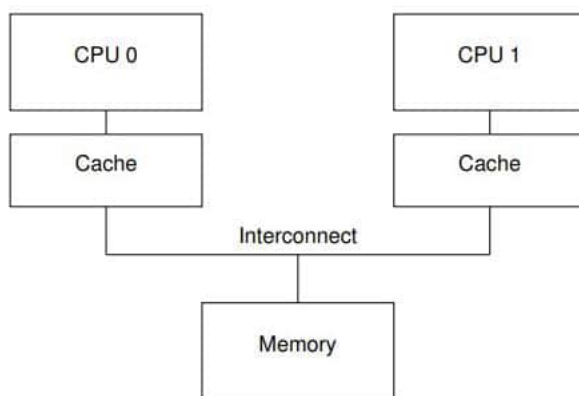


Рисунок 1. Структура кешів у сучасній комп'ютерній системі

Дані передаються між кешами обчислювальних одиниць блоками фіксованого розміру, які називаються кеш-лініями. Зазвичай такі блоки мають розмір степені двійки, від 16 до 256 байт. Коли ядро вперше звертається до якихось даних у пам'яті комп'ютера, ці дані будуть відсутні у його кеші. Відтак, подібна ситуація називається кеш-місом (cache miss). Кеш-міс означає, що цій обчислювальній одиниці доведеться очікувати кілька сотень процесорних циклів, поки дані будуть витягнуті з пам'яті. Але, коли дані будуть завантажені до відповідного кешу, наступні спроби отримати до них доступ візьмуть їх з кешу, що відбудеться набагато швидше.

Після певного часу роботи процесору, певні кеші можуть бути заповненими. Таким чином, якщо будуть потрібні нові дані з пам'яті, старі дані будуть відкинуті з кешу, а нові кеш-лінії завантажені. Щоб уникнути довгого завантаження відкинутих даних, зазвичай роблять багаторівневу систему кешів, де витіснені дані спершу потрапляють у кеш нижчого рівня.

Тим не менш, поки що ми розглядали лише ситуації, у котрих необхідно читати з пам'яті. Проте після обробки певних даних процесором виникає необхідність у записі цих даних до основної пам'яті комп'ютера. А відтак виникає питання, яке саме значення одних і тих самих даних записати у пам'ять з різних кешів процесору.

Таким чином виникає необхідність у операції інвалідації кешу. Якщо якимось ядром хоче записати певні дані у визначену ділянку пам'яті, спершу інші копії цієї пам'яті у різних кешах мають бути інвалідовані, або видалені. Як тільки ця операція була здійснена, певна обчислювальна одиниця може спокійно здійснювати запис до певної комірки пам'яті.

Пізніше, якщо у іншій обчислювальній одиниці виникає необхідність доступу до цієї ж пам'яті, відбудеться кеш-міс, який зазвичай називають комунікаційним кеш-місом. Така назва через те, що він відбувається зазвичай через спроби кількох ядер комунікувати між собою через певну комірку пам'яті.

Очевидно, при такому підході потрібно докласти багато зусиль для того, щоб значення одних і тих самих даних залишилося когерентним. Коли ми дістаємо дані з пам'яті, інвалідуємо, робимо записи, нескладно уявити, що дані можуть зникнути. Ще гірше може бути, якщо у різних кешах процесору знаходяться різні значення одних і тих самих даних. А відтак у дію вступають протоколи підтримки когерентності.

1.3. Протоколи підтримки когерентності

Протоколи підтримки когерентності існують для того, щоб запобігти неконсистентності даних у кешах або навіть їх втрати. Ці протоколи можуть бути дуже складними, з багатьма десятками станів, проте для спрощення і стислості ми розглянемо найпростіший протокол з чотирьох станів, який називається MESI. Цей протокол ми розглядатимемо і надалі в цій роботі як наближення реальної роботи процесора.

1.3.1. Стани MESI

MESI розшифровується як M – modified(модифікований), E – exclusive(ексклюзивний), S – shared(спільний), I – invalid(невалідний) для чотирьох можливих станів конкретної кеш-лінії. Відтак, кеші, які використовують цей протокол, мусять підтримувати певну комірку пам'яті для кожної кеш-лінії, де зберігається її поточний стан.

Якщо кеш-лінія знаходиться в модифікованому стані, це означає, що нещодавно по цій кеш-лінії був запис від певного ядра. Відповідно, пам'ять під цією кеш-лінією не може знаходитися у кеші іншого ядра. Фактично, обчислювальна одиниця володіє цією кеш-лінією. Вона також є відповідальною за усі операції з цією кеш-лінією і за те, щоб записати дані з кеш-лінії у пам'ять, якщо вона буде витіснена з кешу.

Кеш-лінія в ексклюзивному стані це по суті те саме, що й кеш-лінія у модифікованому стані, проте ще не було здійснено жодного запису по цій кеш-лінії. Відповідно, якщо така кеш-лінія має бути витіснена з ядра процесору, це може бути здійснено без будь-яких записів у основну пам'ять комп'ютера.

Кеш-лінія в спільному стані знаходиться у спільному володінні як мінімум двох обчислювальних одиниць. Таким чином, жодна з них не може

здійснити вільний запис без комунікації з іншою. Як і з ексклюзивним станом, кеш-лінія може бути відкинута без будь-яких записів до основної пам'яті.

Кеш-лінія в невалідному стані є порожньою, тобто не містить жодних даних. Коли нові дані потрапляють до кешу, вони записуються до однієї з вільних кеш-ліній у невалідному стані, якщо такі є. Так відбувається через те, що заміна кеш-лінії, яка вже тримає певні дані, призвела би до важкого для процесору кеш-місу, якби ці дані були знову за потреби.

Оскільки усі обчислювальні одиниці мають зберігати цілісність даних у своїх кешах, протоколи підтримки когерентності надають певні повідомлення, які координують стани кеш-ліній у всій системі.

1.3.2. Повідомлення протоколу MESI

Усі переходи між станами, описаними у секції вище, вимагають комунікації. Якщо усі процесори розташовані на спільній шині даних, то можливо ввести такі повідомлення:

1. Read – це повідомлення несе в собі адресу кеш-лінії, яка має бути прочитана.
2. Read response – містить у собі дані, про які було відправлено запит на отримання у Read запиті. Дані можуть бути як з пам'яті, так і з кешу іншого ядра. Якщо вони знаходились у кеші, інше ядро мусить відправити Read response.
3. Invalidate – це повідомлення містить адресу тої кеш-лінії, яку потрібно інвалідувати. Усі інші обчислювальні одиниці мусять прибрати цю кеш-лінію з своїх кешів та відповісти Invalidate acknowledge.
4. Invalidate acknowledge – процесор, який отримав повідомлення Invalidate має відповісти цим повідомленням після того, як відповідна кеш-лінія була усунута з його кешу.

5. Read invalidate – це повідомлення містить адресу кеш-лінії для читання, водночас воно повідомляє інші процесори про необхідність усунути кеш-лінію з їх кешу. У відповідь вимагається отримати як Read response, так і Invalidate acknowledge повідомлення.
6. Writeback – це повідомлення містить адресу і дані кеш-лінії, яка має бути записана назад у основну пам'ять. Це повідомлення дозволяє процесорам відкинути лінії у модифікованому стані з свого кешу.

Детальна діаграма переходу станів може бути знайдена у [7].

1.4. Канали

Зазвичай, комунікація і синхронізація у багатопотокових програмах здійснюється через розділювані комірки пам'яті. Як приклад можуть слугувати м'ютекси, які впорядковують дії, що знаходяться до початку критичної секції і після за допомогою певних відношень. В середині ж критичної секції комунікація здійснюється саме через звернення до розділюваної пам'яті. Прикладом також можуть слугувати певні комірки пам'яті, до яких здійснюються атомарні читання або записи.

Розглядаючи протоколи підтримки когерентності можна помітити, що їх основна ціль складається в тому, щоб зберегти цілісність пам'яті і при тому не втрачати швидкості доступу до неї. Проте через необхідність зберігати цілісність пам'яті неминуче з'являються певні обмеження, які призводять до втрати продуктивності. Таким чином, щоб здійснити запис до певної комірки пам'яті з якогось потоку, процесору на якому виконується даний потік, доведеться виконувати дорогу операцію інвалідації кеш-лінії у інших процесорах, чекати на відповідь і т.д. Однак самі процесори надають більш ефективний інструмент – повідомлення.

Канал – це такий примітив синхронізації, який заснований на принципі передачі повідомлень. «Do not communicate by sharing memory; instead, share memory by communicating» – один з девізів мови програмування Go. Фактично, канал це якась черга, до якої можуть прийти або sender – той, хто бажає надіслати повідомлення, або receiver – той, хто бажає повідомлення отримати.

Існує два типи каналів:

1. Небуферизовані канали. Вони не потребують ніякої буферизації і sender просто додається до черги або зразу ж віддає своє значення конкретному receiver'у.

2. Буферизовані канали. Це традиційні producer-consumer черги, які базуються на циклічному буфері. В даному випадку, якщо буфер є заповненим, можлива необхідність очікування.

Іноді виділяється також третій тип каналів. Це буферизовані канали, елементами яких слугують структури нульового розміру, тобто `void`. Такі канали по суті своїй є семафорами і не будуть розглядатися у цій роботі.

Для реалізації каналів використовуватимуться корутини. Це потрібно для ефективного використання ресурсів процесору. Наприклад, уявімо, що ми є `receiver`'ом каналу. В такому випадку нам може бути необхідним час, щоб до каналу прийшов який-небудь `sender` і відправив значення. При звичайному підході ми мусили б заблокуватися на очікуванні значення. Використовуючи корутини ми можемо призупинити виконання до того моменту, коли нам надійде значення, в той самий час виконуючи інші операції.

Розділ 2. Аналіз різних підходів до реалізації

У цьому розділі будуть розглянуті різні підходи до реалізації каналів. А саме: lockfree реалізація з мови програмування Котлін та реалізація з мови програмування Go.

2.1. Lockfree (без блокувань) канали

Аналіз реалізації lockfree каналів буде здійснено на основі реалізації в мові програмування Котлін. Матеріали для аналізу взято з [8].

Канали, які розглядатимуться, є небуферизованими. Вони будуть засновані на абстрактному нескінченному масиві, який використовуватиметься як черга. Кожна комірка такого масиву буде занумерована. Елементами масиву слугуватиме інша абстрактна структура даних – waiter. Вона складатиметься з корутини та самого елемента, який необхідно відправити чи отримати з каналу. Для підтримки черги вводяться два індекси – `deqIdx` та `enqIdx`, від слів `dequeue` та `enqueue` відповідно. `deqIdx` вказуватиме на першу зайняту комірку, в той час як `enqIdx` на першу вільну, куди можна встати до черги.

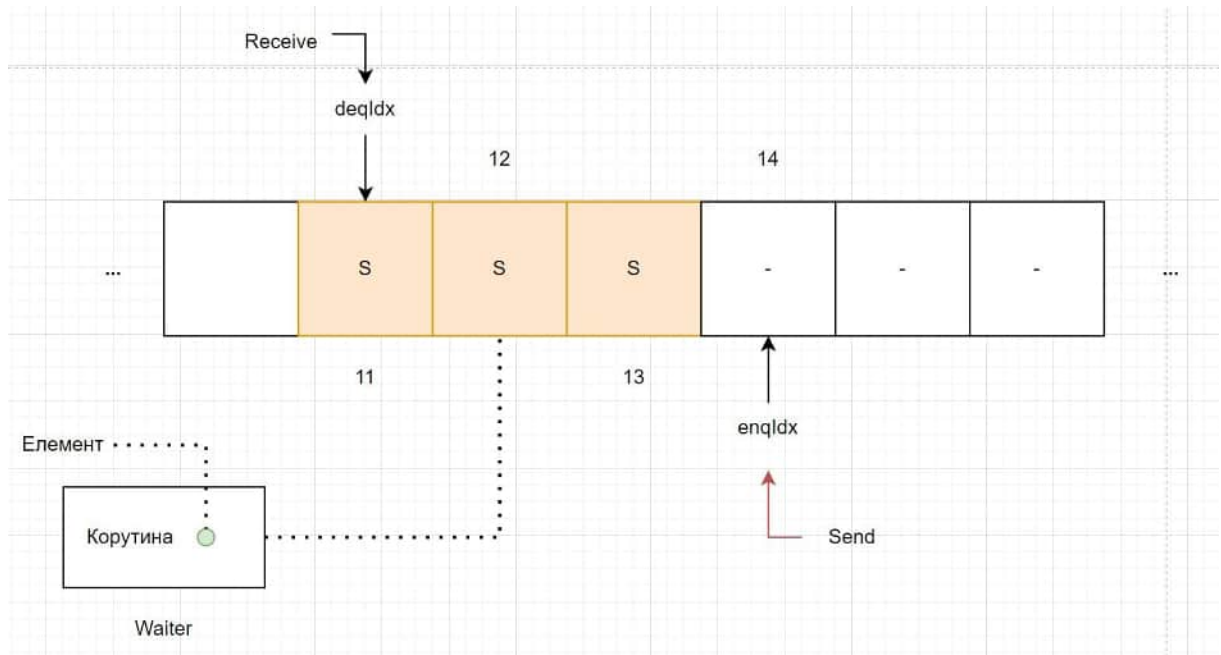


Рисунок 2. Структура lockfree каналу

На рис.2 зображено таку чергу, у якій вже є три елементи – sender`и. Вони очікують на receiver`а, котрому зможуть віддати своє значення.

Алгоритм роботи такого каналу доволі нескладний. Припустимо в черзі вже лежить один елемент і це sender. Якщо ми receiver, у будь якому випадку, нам необхідно зрозуміти чим наповнена черга – sender`ами чи receiver`ами. Тому ми перевіряємо що знаходиться по `deqIdx` та `enqIdx`. Якщо sender`ами, то ми маємо просто забрати елемент першого з них, якщо ж receiver`ами, то додати себе у кінець черги. Схему такого випадку зображено на рис. 3.

Тим не менше, це все відбувається у багатопотоковому кодї, тому receiver може бути не один, а лише один з багатьох, хто конкурує за отримання поточного sender`а. Тому необхідно атомарно додати до `enqIdx` одиницю, що здійснюється за допомогою операції CAS – compare and swap. Значення атомарно порівнюється з очікуваним і якщо співпало, замінюється на нове. Алгоритм для додавання sender`а такий самий, тільки збільшується `enqIdx`. Після пересування індексу можна безпечно брати елемент або записувати себе до черги.

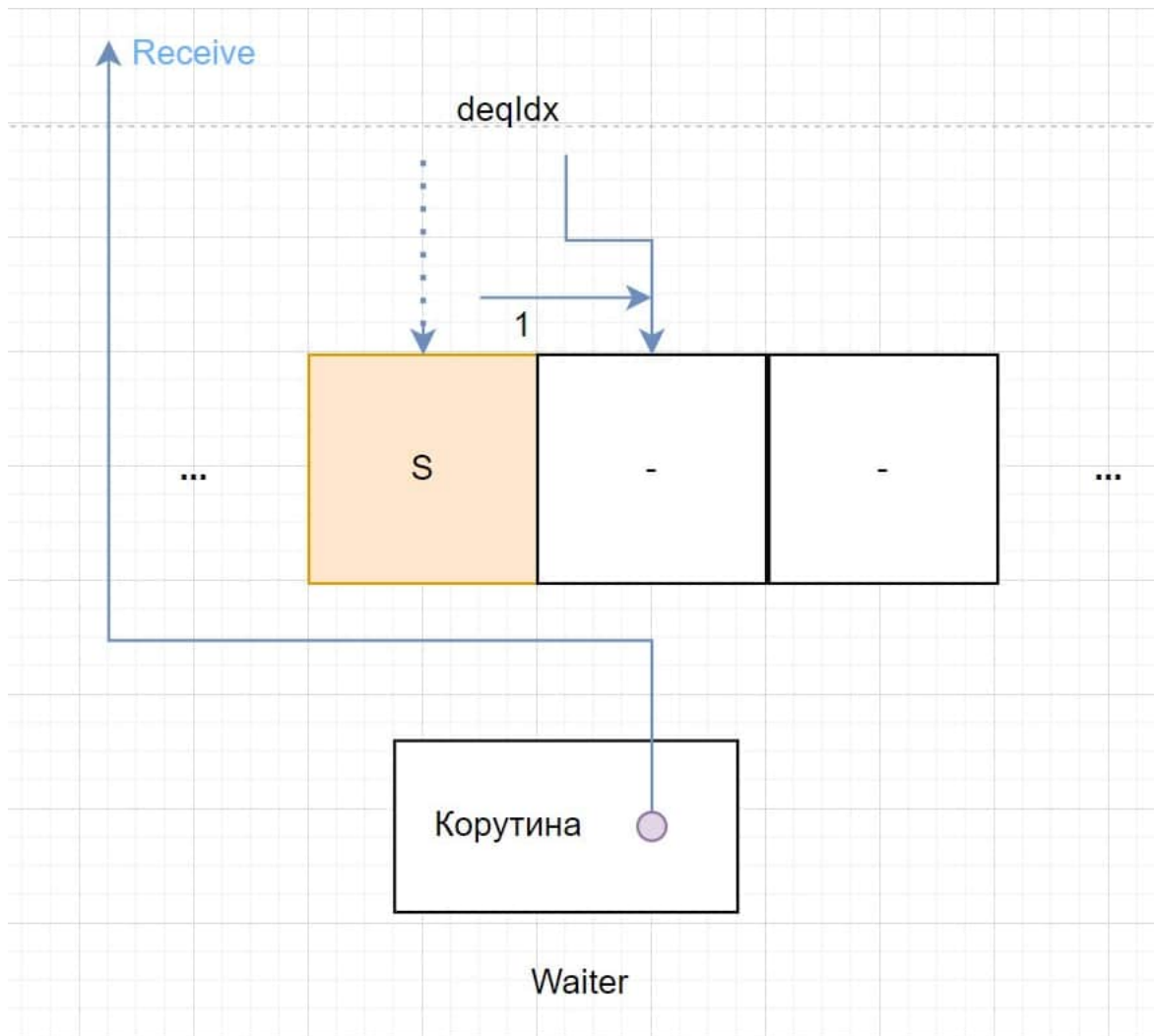


Рисунок 3. Алгоритм роботи lockfree каналу

Проте з такою реалізацією виникають певні проблеми. Припустимо ситуацію, де якийсь sender додається до черги. Він здійснив збільшення `enqIdx`, а потім заснув. Таке можливо у багатопотокових програмах. Час на виконання цього потоку закінчився і процесор почав виконувати щось інше. Поки цей sender спав, receiver`и розібрали чергу до цього індексу, в який мав бути записаним sender. Receiver мав би прочитати значення, проте його просто немає, через що реалізація ламається (рис. 4).

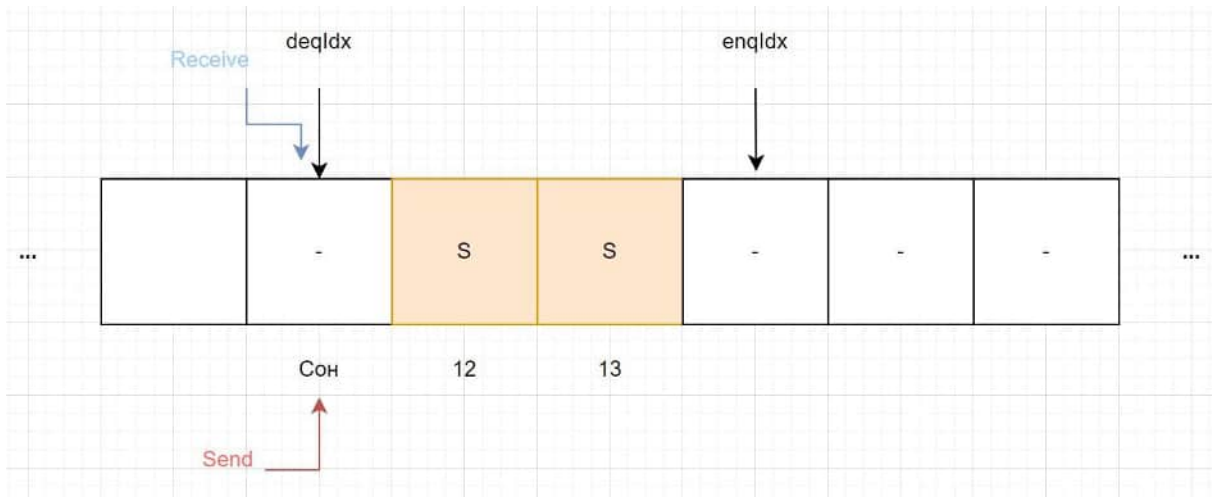


Рисунок 4. Помилка реалізації lockfree каналу

Рішення такої проблеми доволі просте. Receiver спробує здійснити запис спеціального значення до комірки. Таке значення називатиметься отруєним – poisoned. Тоді sender, якщо побачить таке значення, вже не намагатиметься записати себе до комірки, а просто знову намагатиметься себе додати до кінця черги. Якщо ж sender встигне раніше receiver`а, то останній побачить наявність значення і просто прочитає його (рис. 5).

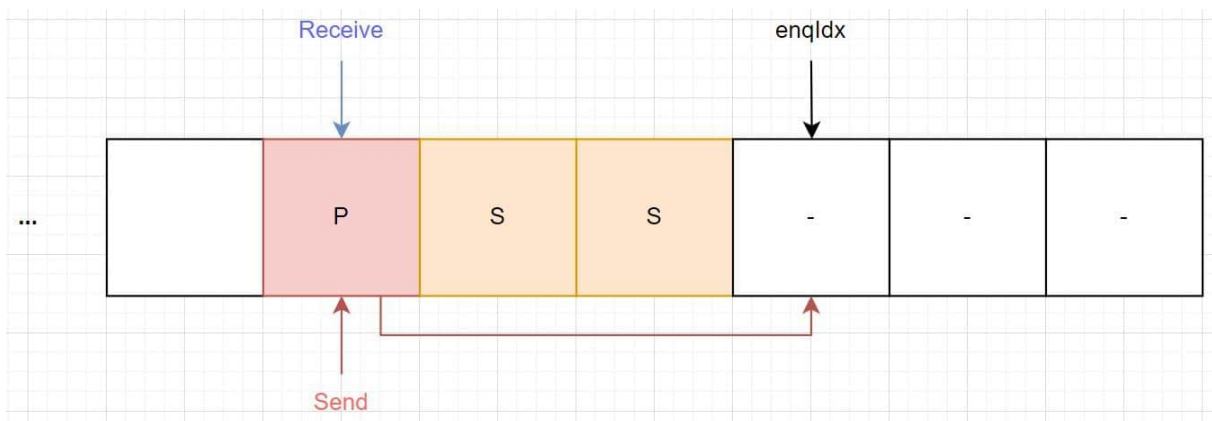


Рисунок 5. "Отруєні" значення

Така реалізація теж має свою проблему. Можливо припустити сценарій, де кожен sender встигає лише пересунути індекс у черзі, а потім засинає. Потім до черги приходять receiver і не бачить значення, внаслідок чого записує отруєне. Sender прокидається, бачить отруєне значення, пересовує індекс і знову засинає, після чого приходять receiver... Здавалосьь

би якась робота виконується, але жодного прогресу насправді не досягнута. Така ситуація у lockfree алгоритмах називається livelock.

Щоб вирішити цю проблему достатньо лише підібрати правильну структуру даних. Припустимо, для реалізації звичайного нескінченного масиву використовується черга Майкла-Скота[9]. Її суть полягає в створенні нового елемента кожен раз, коли необхідно розширити чергу. Існує також інший варіант – сегментована черга Майкла-Скота[9]. Різниця з попередньою чергою полягає у тому, що коли недостатньо місця додаються не елементи, а одразу цілі сегменти фіксованої довжини.

Ідея вирішення livelock`у полягає в тому, щоб при такій структурі даних в кожному сегменті при створенні зразу класти перший елемент – sender чи receiver. Таким чином, коли якийсь waiter намагатиметься перевірити стан черги і записати себе або отримати елемент, черга вже буде не пустою, а відтак жодної проблеми з отруєним елементом не виникне.

Заміри продуктивності таких каналів можна знайти у [8].

2.2. Go-канали

Аналіз реалізації каналів у мові програмування Go базується на [10]. Автором цієї реалізації є Дмитрій В'юков – один з розробників мови програмування Go. Буде розглянутий як небуферизований, так і буферизований варіанти.

2.2.1. Небуферизований канал

Структура даних каналу буде дуже простою: списки `sender`ів` та `receiver`ів`, булева змінна чи закритий канал та м'ютекс. Алгоритм для відправки значення складатиметься з наступних кроків:

1. Перевірити чи канал відкритий. Якщо ні, операція невдала.
2. Якщо ми не блокуємось на очікування `receiver`а`, якщо `receiver` відсутній у черзі, операція невдала.
3. Заблокуватися.
4. Знову здійснити крок 2, але вже під блокуванням.
5. Здійснити спробу отримати `receiver` і якщо це вдалося передати йому значення, операція вдала. Якщо ні, тоді крок 6.
6. Перехід до цього кроку значить, що поки ми намагалися отримати `receiver`, інший більш вдалий потік вже зміг отримати його, тому якщо ми не блокуємося на очіванні, операція невдала. Якщо блокуємося, крок 7.
7. Блокуємося і чекаємо на появу `receiver`а`, завершуємо операцію успіхом.

2.2.2. Буферизований канал

Структура даних каналу зазнає певних змін відносно попередньої реалізації. А саме, з'являється змінна для збереження розміру каналу, буфер вказаного розміру та дві 64-бітні змінні. Перші 32 байти цих змінних позначатимуть позицію у буфері, в той час як наступні 32 поточне «коло»

для sender`ів і receiver`ів відповідно. Самі елементи будуть структурою даних, де зберігаються дані, які користувач хотів передати та поточне «коло». Значення «кола» – парні елементи якщо елемент готовий до запису на цьому колі, непарні якщо готовий до читання. Алгоритм для відправки значення складатиметься з наступних кроків:

1. Отримати з каналу змінну, яка позначає позицію та «коло» для sender`ів та прочитати відповідні значення.
2. Отримати наступний елемент з буфера та його «коло».
3. Якщо «коло» sender`а збігається з колом елемента, тоді можна здійснити спробу відправити значення і перейти на крок 4. Якщо ні крок 7.
4. Необхідно визначити нову наступну позицію у каналі. Якщо наступна позиція все ще менша розміру каналу, тоді просто збільшуємо позицію на 1, якщо ні, переходимо до початку циклічного буфера.
5. Намагаємося збільшити позицію у змінній для sender`ів для самого каналу за допомогою CAS. Якщо не вдалося, переходимо до кроку 1.
6. Робимо запис до елемента і збільшуємо його «коло» на 1, щоб він був доступним для читання.
7. Якщо «коло» операції відправки більше, ніж коло поточного елемента, значить елемент ще не був прочитаний на минулому колі і канал повний, операція невдала. В інакшому випадку крок 8.
8. Більш вдалий потік вже здійснив запис на цьому кроці. Перехід до кроку 1.

Заміри продуктивності таких каналів можна знайти у [10].

Розділ 3. Реалізація програми

Для власної реалізації використовується мова програмування C++. Особливістю реалізації є використання корутин, які додали в останній стандарт C++20. Реалізація здійснюється за допомогою відкритої сторонньої бібліотеки YACLib. Це здійснюється з метою поширення власної реалізації як відкритого коду, доступного кожному для читання і навчання.

3.1. Корутини C++20

З виходом стандарту C++20 у мові програмування C++ з'явилися корутини. Їх підтримка має особливе значення для будь-яких програм, що працюють з асинхронністю. Наприклад, уявімо ситуацію, де програма робить запит до інтернет-ресурсу. Це може зайняти певний час, доволі довгий, який у звичайному випадку доведеться просто чекати, не виконуючи жодної корисної роботи. Використовуючи корутини ж можливо призупинити виконання функції, яка здійснювала запит, і повернутися до неї коли відповідь на запит вже прийде.

При створенні каналів корутини мають своє значення. У випадку небуферизованого каналу з чергою необмеженої довжини, корутини можуть знадобитися при отриманні значення з каналу. Такого значення може просто не бути і без корутин доведеться блокуватися і чекати. З корутинами ж можна призупинити виконання і повернутися вже коли з'явиться значення.

Для буферизованого каналу очікування можливе як у випадку надсилання значення, так і у випадку отримання. У разі надсилання значення, черга каналу може вже бути повною. У разі отримання значення все так само, його може просто не бути. Очікування в обох цих випадках дозволяють уникнути корутини.

3.1.1. Awaiters

Для підтримки корутин у C++20 додали три ключові слова: `co_await`, `co_yield` та `co_return`. Найважливішим з них є `co_await`, який дозволяє здійснювати гнучке очікування поки щось не буде готовим. Здійснюється таке за допомогою спеціальних об'єктів, які називаються `awaiter`.

`Awaiter` підтримує інтерфейс з трьох функцій: `await_ready`, `await_suspend` і `await_resume`. `await_ready` завжди повертає булеве значення, яке вказує на те, чи дійсно необхідно призупинити корутину, чи ні. `await_suspend` приймає на вхід саму корутину, щоб здійснювати нею управління. Він може повертати `void`, у випадку якщо корутина буде неодмінно призупинена, `bool`, який означає те саме, що і в `await_ready` або іншу корутину, до якої повернеться управління замість поточної. `await_resume` повертає те, що повертається після виконання `co_await`. Таким чином можна не тільки призупинити корутину, а й повернути певне значення за необхідності.

3.2. Код програми

Нижче наведено код, який реалізує буферизований та небуферизований канали з використанням корутин C++20.

```
#pragma once

#include "yaclib/algo/detail/base_core.hpp"
#include "yaclib/fwd.hpp"

#include <yaclib/coro/coro.hpp>
#include <yaclib/log.hpp>
#include <yaclib/util/detail/intrusive_list.hpp>
#include <yaclib/util/detail/node.hpp>

#include <mutex>
#include <queue>
#include <utility>

namespace yaclib {
template <typename T>
class BoundedChannel {
    class PushAwaiter : public detail::Node {
    public:
        PushAwaiter() = default;

        template <typename... Args>
        explicit PushAwaiter(std::unique_lock<std::mutex>& lock, BoundedChannel&
channel, Args&&... args)
            : _channel{&channel}, _value{std::forward<Args>(args)...} {
            lock.release();
        }

        bool await_ready() noexcept {
            return _channel == nullptr;
        }

        template <typename Promise>
        void await_suspend(yaclib_std::coroutine_handle<Promise> handle) noexcept
        {
            _core = &handle.promise();
            _channel->_senders.PushFront(*this);
            _channel->_mutex.unlock();
        }

        constexpr void await_resume() noexcept {
```

```

}

void Resume(T& value) {
    value = std::move(_value);
    _core->_executor->Submit(*_core);
}

void Resume(std::unique_lock<std::mutex>& lock) {
    _channel->_queue.push(std::move(_value));
    lock.unlock();
    _core->_executor->Submit(*_core);
}

private:
    BoundedChannel* _channel{nullptr};
    detail::BaseCore* _core{nullptr};
    T _value;
};

class PopAwaiter : public detail::Node {
public:
    explicit PopAwaiter(BoundedChannel& channel) : _channel{channel} {
    }

    bool await_ready() {
        std::unique_lock lock{_channel._mutex};
        if (_channel._queue.empty()) {
            if (_channel._senders.Empty()) {
                lock.release();
                return false;
            }
            auto& sender = _channel._senders.PopFront();
            lock.unlock();
            static_cast<PushAwaiter&>(sender).Resume(_value);
        } else {
            _value = std::move(_channel._queue.front());
            _channel._queue.pop();
            if (!_channel._senders.Empty()) {
                auto& sender = _channel._senders.PopFront();
                static_cast<PushAwaiter&>(sender).Resume(lock);
            }
        }
        return true;
    }
};

template <typename Promise>
void await_suspend(yacolib_std::coroutine_handle<Promise> handle) noexcept
{
    _core = &handle.promise();
    _channel._receivers.PushFront(*this);
}

```

```

    _channel._mutex.unlock();
}

T await_resume() {
    return std::move(_value);
}

template <typename... Args>
void Produce(Args&&... args) {
    _value = T{std::forward<Args>(args)...};
    _core->_executor->Submit(*_core);
}

private:
    BoundedChannel& _channel;
    detail::BaseCore* _core{};
    T _value;
};

public:
    BoundedChannel(size_t size) : _size{size} {
    }

    template <typename... Args>
    PushAwaiter Push(Args&&... args) {
        std::unique_lock lock{_mutex};
        if (_receivers.Empty()) {
            if (_queue.size() == _size) {
                return PushAwaiter{lock, *this, std::forward<Args>(args)...};
            }
            _queue.emplace(std::forward<Args>(args)...);
        } else {
            YACLIB_ASSERT(_queue.empty());
            auto& receiver = _receivers.PopFront();
            lock.unlock();
            static_cast<PopAwaiter&>(receiver).Produce(std::forward<Args>(args)...);
        }
        return {};
    }

    PopAwaiter Pop() {
        return PopAwaiter{*this};
    }

private:
    std::mutex _mutex;
    std::queue<T> _queue;
    std::size_t _size;
    detail::List _senders;
    detail::List _receivers;

```

```

};

template <typename T>
class Channel {
    class Awaiter : public detail::Node {
    public:
        explicit Awaiter(Channel& channel) : _channel{channel} {
        }

        bool await_ready() {
            std::unique_lock lock{_channel._mutex};
            if (_channel._queue.empty()) {
                lock.release();
                return false;
            }
            _value = std::move(_channel._queue.front());
            _channel._queue.pop();
            return true;
        }

        template <typename Promise>
        void await_suspend(yaclib_std::coroutine_handle<Promise> handle) noexcept
        {
            _core = &handle.promise();
            _channel._consumers.PushFront(*this);
            _channel._mutex.unlock();
        }

        T await_resume() {
            return std::move(_value);
        }

        template <typename... Args>
        void Produce(Args&&... args) {
            _value = T{std::forward<Args>(args)...};
            _core->_executor->Submit(*_core);
        }

    private:
        detail::BaseCore* _core{};
        Channel& _channel;
        T _value;
    };

public:
    template <typename... Args>
    void Push(Args&&... args) {
        std::unique_lock lock{_mutex};
        if (_consumers.Empty()) {
            _queue.emplace(std::forward<Args>(args)...);

```

```

    } else {
        YACLIB_ASSERT(!_queue.empty());
        auto& consumer = _consumers.PopFront();
        lock.unlock();
        static_cast<Awaiter&>(consumer).Produce(std::forward<Args>(args)...);
    }
}

Awaiter Pop() {
    return Awaiter{*this};
}

private:
    std::mutex _mutex;
    std::queue<T> _queue;
    detail::List _consumers;
};
}

```

3.3. Аналіз реалізації

Для порівняння власної реалізації з іншими наведеними у роботі реалізаціями буде здійснений розбір алгоритму та замір швидкодії.

3.3.1. Небуферизований канал

Як це вже зазначалось у цій роботі, небуферизований канал вимагає `awaiter` лише для операції читання з каналу. Сам канал складається з м'ютекса, який використовується для захисту даних при роботі з каналом з багатьох потоків, черги, до якої додаються елементи та списку споживачів значень з каналу. Також реалізується внутрішня структура `awaiter`a` для отримання значень.

Надсилання значення до каналу реалізується у методі `Push`. При спробі додати значення до каналу існують два можливі випадки. Або жодних споживачів немає і в такому випадку значення просто додається до

черги, або споживачі є, і тоді після отримання споживача в нього записується значення, яке буде витягнуто при виклику Pop.

Метод Pop просто повертає awaіter від каналу. Це робиться для того, щоб якщо в каналі не було значень, можна було чекати на їх появу без блокувань. Це досягається за допомогою реалізації Awaіter.

У методі await_ready перевіряється чи черга є пустою. Якщо це так, тоді потрібно повернути управління. Якщо ні, тоді можна не повертати управління, а одразу отримати значення і записати його всередину awaіter`у, щоб повернути у await_resume.

У await_suspend корутина завжди призупиняється. Сам awaіter додається до списку споживачів каналу і чекає на появу в ньому значення. Таким чином, коли вона прокинеться, в awaіter`і завжди буде значення, яке і повернеться через await_resume.

Окрім звичайних методів awaіter`ів наявний також метод Produce. Його призначення саме в тому, щоб «розбудити» корутину. Він використовується у методі Push, який у випадку наявності споживача одразу ж пише в нього значення через Produce. В цьому методі ж відбувається повернення управління до корутини через запис залишку її коду на виконання у пулі потоків. Після продовження виконання корутини через await_resume повернеться значення, записане в awaіter`і.

3.3.2. Буферизований канал

Реалізація буферизованого каналу поскладнюється тим, що операція Push більше не є завжди неблокуючою. Іноді, якщо буфер каналу заповнений, необхідно чекати на появу вільного місця. Сам канал складатиметься з м'ютекса, черги, розміру каналу та списків тих, хто надсилає значення і отримує.

Буферизований канал складніше зробити також тому, що він має бути «чесним». Це значить, що при отриманні значень з каналу вони мають бути повернуті у тому ж порядку, у якому й надіслані. Можливо реалізувати і «нечесний» канал, але подібне призведе до того, що якісь значення можуть довше очікувати на отримання, ніж інші. Ці проблеми виникають тому, що не ясно звідки брати наступне значення: з черги чи з списку відправників.

Розглянемо метод `Pop` і спосіб вирішення проблеми «чесності» каналу. Метод `Pop` повертає `awaiter`, який має відмінну від небуферизованого каналу семантику.

Метод `await_ready` перевіряє наявність у черзі значень. Якщо їх немає, а також немає жодного відправника, яким є `awaiter`, що повертається при спробі записати значення до заповненого каналу, тоді корутина призупиняється. Якщо ж відправники існують, викликається метод `Resume` відправника, який має наступну семантику: віддати значення `awaiter` у споживача і продовжити виконання корутини, які здійснювала відправку.

Якщо ж черга каналу не є пустою, тоді можна отримати звідти значення. При тому, якщо список відправників каналу є не пустим, можна для одного з них викликати метод `Resume`, який має нову семантику. А саме переміщення значення з відправника до черги каналу, оскільки на попередньому кроці було звільнене одне місце. Таким чином досягається «чесність» каналу, бо наступні значення переміщуються до його черги.

Якщо корутина все ж була призупинена у `await_ready`, викликається `await_suspend`, який додає її до списку споживачів каналу. Повернення до виконання цієї корутини виконується як і у випадку небуферизованого каналу, а саме через метод `Produce`.

Реалізація методу `Push` доволі нескладна. Якщо немає жодних споживачів і черга заповнена, тоді повертається `awaiter`. У випадку якщо черга не заповнена, значення додається до неї. Якщо ж споживачі є, один з них повертається на виконання через метод `Produce`.

У методі `await_ready` `awaiter`у` для `Push` корутина призупиняється завжди. `await_suspend` ж записує корутину у список відправників каналу. `await_resume` не повертає нічого, оскільки це операція надсилання значення, а не отримання.

Також реалізовані два методи `Resume`, семантика яких була згадана при розборі `awaiter`у` для `Pop`.

3.4. Швидкодія програми

Нижче зазначені результати заміру швидкодії каналів. Усі заміри відбуваються у випадку двох ядер. Аналіз проводиться для передачі булевого значення, цілочисельного значення і пустої структури.

3.4.1. Результати замірів

Небуферизований канал:

Тип	Кількість ітерацій	Результат, нс/операція
-----	--------------------	------------------------

int	1 000 000	34
bool	1 000 000	39
struct{ }	1 000 000	33
int	10 000 000	34
bool	10 000 000	42
struct{ }	10 000 000	34

Буферизований канал. Заміри проводились з різною ємністю каналу, проте результати приблизно однакові, незалежно від розміру буферу.

Тип	Кількість ітерацій	Результат, нс/операція
int	1 000 000	52
bool	1 000 000	52
struct{ }	1 000 000	43
int	10 000 000	45
bool	10 000 000	44
struct{ }	10 000 000	43

3.4.2. Код замірів

```

yaclib::FairThreadPool tp{2};
yaclib::Channel<S> ch;
auto producer = [&]() -> yaclib::Future<> {
    co_await On(tp);
    auto start = std::chrono::system_clock::now();
    for (int i = 0; i != 10000000; ++i) {
        ch.Push(S{});
    }
    auto end = std::chrono::system_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start) ;
    std::cout << "duration " << (elapsed.count() / 10000000) << std::endl;
    co_return{};
};
auto consumer = [&]() -> yaclib::Future<> {
    co_await On(tp);

    for (int i = 0; i != 10000000; ++i) {
        auto r = co_await ch.Pop();
    }
    co_return{};
};
auto c = consumer();
auto p = producer();

yaclib::Wait(c, p);

tp.HardStop();
tp.Wait();

yaclib::FairThreadPool tp{2};
yaclib::BoundedChannel<int> ch{100};
auto producer = [&]() -> yaclib::Future<> {
    co_await On(tp);
    auto start = std::chrono::system_clock::now();
    for (int i = 0; i != 10000000; ++i) {
        co_await ch.Push(42);
    }
    auto end = std::chrono::system_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start) ;
    std::cout << "duration " << (elapsed.count() / 10000000) << std::endl;
    co_return{};
};
auto consumer = [&]() -> yaclib::Future<> {
    co_await On(tp);

```

```
    for (int i = 0; i != 10000000; ++i) {
        auto r = co_await ch.Pop();
    }
    co_return{};
};
auto c = consumer();
auto p = producer();

yaclib::Wait(c, p);

tp.HardStop();
tp.Wait();
```

Висновок

У дипломній роботі було розглянуто і проаналізовано основні підходи до реалізації каналів. Також для побудови каналів були використані корутини мови програмування C++20 та перевірена їх ефективність. Порівняно з lockfree реалізацією обидві інші виграють. Ймовірні причини складаються в тому, хоча lockfree алгоритми забезпечують прогрес без блокувань, однак це відбувається за рахунок багатьох додаткових кроків, яких нема у звичайних реалізаціях. Власна реалізація програє для буферизованих каналів, проте має вигреш для небуферизованого варіанту. Результати замірів швидкодії власної реалізації свідчать про масштабованість небуферизованого варіанту, оскільки швидкодія практично не змінюється. Швидкодія буферизованих каналів нестабільна, що може свідчити про необхідність подальшого покращення. Ймовірним напрямком руху буде реалізація з мови програмування Go, яка в цьому випадку виграє по швидкодії.

Список використаних джерел

1. Cramming more components onto integrated circuits [Електронний ресурс] – Режим доступу до ресурсу: <https://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf>
2. The case for a single-chip multiprocessor [Електронний ресурс] – Режим доступу до ресурсу: <https://www.ece.ucdavis.edu/~akella/270W05/reading/p2-olukotun.pdf>
3. Linux 2.6 [Електронний ресурс] – Режим доступу до ресурсу: <https://kernel.org/pub/linux/kernel/v2.6/>
4. PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: <https://www.postgresql.org/>
5. The Open MPI Project [Електронний ресурс] – Режим доступу до ресурсу: <https://www.open-mpi.org/software/>
6. Knuth D. Fundamental Algorithms / Knuth Donald – Addison-Wesley. Section 1.4.5: History and Bibliography, 1997. – 666 с.
7. Is Parallel Programming Hard, And, If So, What Can You Do About It? [Електронний ресурс] – Режим доступу до ресурсу: <https://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git>
8. N. Koval. Scalable fifo channels for programming via communicating / Nikita Koval, Dan Alistarh, and Roman Elizarov – Springer, 2019. – 17 с.
9. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms [Електронний ресурс] – Режим доступу до ресурсу: https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf
10. Go channels on steroids [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.google.com/document/d/1yIAymbvL3JxOKOjuCyon7JhW4cSv1wy5hC0ApeGMV9s/pub>
11. Charles H. Communicating Sequential Processes / Charles Antony Richard Hoare – Communications of the ACM, 1978. – 666-677 с.