Содержание

Список сокращений и условных обозначений 4
Введение 5
1 Цели и задачи работы 6
2 Введение в предметную область 7
2.1 Актуальность и практическая значимость 7
2.2 Многопоточная синхронизация 8
2.2.1 Блокирующая синхронизация 9
2.2.2 Неблокирующая синхронизация 12
2.2.3 Построение неблокирущей синхронизации 13
2.2.4 Обзор MWCAS 16
2.2.5 Обзор алгоритма работы MWCAS 17
2.2.6 Синхронизация с использованием транзакционной памяти 21
3 Обзор модуля широковещательной рассылки Tokio 24
3.1 Внутреннее состояние канала 24
3.2 Алгоритм канала 27
4 Анализ оптимального дизайна для очереди 31
4.1 Выделение памяти 31
4.2 Использование неблокируюещей синхронизации 32
5 Новый алгоритм модуля broadcast 34
5.1 Организация общего состояния канала 34
5.2 Алгоритм канала 36
6 Сравнительный анализ производительности 40
6.1 Режим доступа SPSC 41
6.2 Режим доступа SPMC 42
6.3 Режим доступа MPSC 43
6.4 Режим доступа MPMC 44
7 Заключение 46
8 Список использованных источников 47

Приложение 49

Список сокращений и условных обозначений

- CAS - Compare And Swap
- MWCAS - Multi Word Compare And Swap
- ОС - Операционная Система
- MPMC - Multiple Producer Mulptiple Consumer
- MPSC - Multiple Producer Single Consumer
- SPMC - Single Producer Mulptiple Consumer
- SPSC - Single Producer Single Consumer

Введение

На сегодняшний день в основе большинства разрабатываемых приложений и систем: веб-серверов, кластеров обработки данных, блокчейн сетей и тому подобных, лежит инфрастуктурная основа в виде надёжной и производительной среды предоставления асинхронного исполнения - runtime исполнения асинхронных задач.

От него требуется обеспечение эффективной обработки задач, предоставления инструментов композиции и коммуникации между ними. Такая основа может быть встроена в сам язык, как например в Erlang и Go, или использоваться как отдельная библиотека. Примером такого подхода являются библиотеки Kotlin Coroutines и Rust Tokio.

Любой такой инструмент строит свои абстракции исполнения поверх процессов или потоков, предоставляемых операционной системой, поэтому во время его использования возникает потребность в синхронизации между ними. Особенно эта проблема актуальна для примитивов синхронизации, предоставляемыми данными инструментами.

Синхронизация по своей природе может быть нескольких типов. Каждый из них предоставлет как преимущества, так и недостатки. Современные архитектуры имеют тенденцию распараллеливать вычислительные процессы. Однако в большом числе случаев для синхронизации до сих пор используются инструменты крайне неэффективно масштабирующиеся вслед за архитектурой процессоров. В большинстве случаев - это блокирующая синхронизация, несмотря на то, что существуют способы производить операции в неблокирующем режиме, которые открывают возможность к существенному масштабированию вычислений. Связано это с тем, что неблокирующая синхронизация довольно сложна в реализации, в отличие

от блокирующей, а для комплексных структур данных эффективная и простая реализация становится практически невозможной.

В рамках данной работы рассмотрена возможная оптимизация для примитивов синхронизации фреймворка Tokio в среде языка Rust с использованием неблокирующей синхронизации.

1 Цели и задачи работы

Целью работы является оптимизация модуля широковещательной рассылки фреймворка Tokio.

Для выполнения цели были выделены следующие задачи:

- Определить оптимальный дизайн очереди
- Реализовать полученную модель
- Провести сравнительное тестирование

2 Введение в предметную область

2.1 Актуальность и практическая значимость

Фреймворк Tokio на сегодняшний день является основным инструментом для построения асинхронных приложений в экосистеме языка Rust. Он предоставляет разработикам приложений и систем, не только среду исполнения асинхронных задач, но также и примитивы синхронизации, которые могут быть использованы для реализации задач коммуникации между исполняемыми задачами.

Так как Tokio в общем виде представляет собой среду исполнения, спроектированную для работы на многоядерных процессорах, его планировщик а также инструменты, поставляемые вместе с ним, должны поддерживать соответствующее масштабирование. Сам планировщик и часть примитивов синхронизации уже поддерживают эффективное масштабирование, однако остаётся часть, которая использует для этого неэффективные техники.

Один из таких инструментов, канал, предоставляющий семантику широковещательной рассылки, при применении которого, отправленное сообщение должны увидеть все потоки, подписавшиеся на рассылку сообщений. В рамках работы производится исследование возможных оптимизаций данного примитива синхронизации.

Актуальность добавляемых оптимизаций обуславливается возможностью повышения производительности предоставляемых фреймворком Tokio инструментов, что повысит производительность клиентских приложений, использующих его в качестве своей основы.

2.2 Многопоточная синхронизация

Существует несколько общих подходов к построению многопоточной синхронизации. Каждый из них предоставляет пользователю следующие параметры:

- Гарантии прогресса многопоточного исполнения (liveness)
- Относительная степень масштабирования вычислений
- Простота реализации
- Платформенная / аппаратная поддержка

Эти подходы можно разделить на три больших класса:

- Блокирующая синхронизация
- Неблокирующая синхронизация
- Транзакционная память

На данный момент не существует универсального: простого и одновременно эффективного способа синхронизации. Вместе с каждым классом возникают как свои определённые достоинства, так и недостатки, выражающиеся в отсутвие одного или нескольких из перечисленных параметров.

Далее будут подробно рассмотрены все классы синхронизации.

2.2.1 Блокирующая синхронизация

Первый рассматриваемый класс представляет собой способ построения синхронизации вокруг критических секций - участков программы, одновременное исполнение которых возможно только одним, в случае модификации состояния, или несколькими определёнными выделенными потоками, в случае чтения. При этом происходит блокирование прогресса всех остальных исполняемых задач до момента завершения выполнения выделенных задач.

У такого типа синхронизации есть большое преимущество - он простой и не требует специального подхода к построению структуры данных, над которой производятся операции. Также при относительно небольшом числе параллельных задач, выполняющих критические секции, достигается оптимальное использование ресурсов, необходимых для координации задач.

Если рассматривать поддержку, то на всех современных многоядерных архитектурах, операционные системы предоставляют необходимые примитивы для осуществления данного типа синхронизациию.

Примерами могут выступать:

- futex syscall (Linux)
- WaitOnAddress (Windows)

Однако данный подход крайне неэффективно масштабируется, так как в единицу времени может выполняться только одна критическая секция. Остальным потокам необходимо ждать освобожнения блокировки. Это время может стать очень продолжительным из-за размера очереди ожидания. Ожидание может быть сопряжено с дополнительными системными вызовами, например с futex syscall. Или

же могут возникать затраты на переключение контекста и координацию в очередях ожидания, в случае использования корутин поверх потоков.

Если присутствует большое число потоков оперирующих над критической секцией, появляется большое число подобных накладных расходов. В худшем случае, при таком использовании блокирующей синхронизации, число исполняемых критических секций в единицу времени может быть меньше, чем если бы они исполнялись последовательно одним ядром процессора. Отдельно стоит также сказать о проблеме, при которой поток или процесс захвативший блокировку и исполняющий критическую секцию, будет временно снят с исполнения планировщиком задач операционной системы до момента снятия блокировки, например по истечению выделенного ему временного кванта на выполнение. В данном случае возникает риск полной остановки прогресса исполнения системы до конечной разблокировки.

Самый простой пример такой синхронизации - использование мьютекса:

```
// Располагается в общей для потоков памяти
let mutex = Mutex::new()

// Вызывается разными потоками
fn doCriticalSection(){
    {
        mutex.lock()
        // Исполнение критической секции
        mutex.unlock()
    }
}


fn main() {
```

```
    let thread_1 = spawn(doCriticalSection)
    let thread_2 = spawn(doCriticalSection)

    thread_1.join();
    thread_2.join();
}
```

Таблица 1 – Применение блокирующей синхронизации.

2.2.2 Неблокирующая синхронизация

Для избавления от проблем присутствующих у блокирующей синхронизации, существует альтернативный подход, при котором, операции над данными осуществляются в неблокирующем режиме. При этом исчезает понятие критической секции.

Неблокирующая синхронизация предоставляет следующие преимущества по сравнению с блокирующей:

- Гарантия прогресса системы в целом - означает, что при любом многопоточном исполнении, всегда есть поток или потоки успешно завершающие свои операции. Решения планировщика ОС теперь не могут привести к полной остановке системы.

- Сравнительно высокая масштабируемость по сравнению с блокирующей синхронизацией - при использовании неблокирующей синхронизации потоки не обязаны ждать друг друга, поэтому операции могут работать в параллель. Вся координация между потоками образуется в специальных точках синхронизации. В большинстве языков программирования - это атомарные переменные.

- Уменьшение накладных расходов на синхронизацию. Использование атомарных переменных не требует обращения к ядру операционной системы. Операции над атомарными переменными напрямую упорядочивают исполнение через L2 кэш ядер процессора, за счёт чего достигается требуемая линеаризация операций.

2.2.3 Построение неблокирущей синхронизации

При использовании неблокирующей синхронизации изчезает понятие критической секции. Вместо этого появляется понятие транзакции, совершаемой над состоянием структуры данных. Транзакция представляет собой серию операций чтения и записи в определённые ячейки памяти. Эти ячейки памяти - атомарны.

Выделяются специальные операции над атомарными ячеейками памяти:

- read - чтение
- store - запись
- cas - условная запись
- Иные операции в зависимости от архитектуры процессора

Главная особенность этих операций состоит в том, что они определяют порядок обращений потоков к определённой ячейке, за счёт чего достигается синхронизация.

В большинстве случаев образуется общий подход при построении структуры даннных и операций над ней - необходима модификация структуры на основе её текущего состояния. Для синхронизации выделяется общее состоянии, которое становится атомарной ячейкой памяти, в том плане, что все операции над ней линеаризуемы и образуют некоторый порядок обращений.

Все операции абстрактно в рамках атомарной транзакции разбиваются на три этапа:

1. Копирование текущего состояния (snapshot).
2. Локальная модификация полученного состояния.

3. Попытка замена общего состояния на модифицированную копию, в случае, если общее состояние за время модификации не изменилось. Если состояние успело измениться - начать заного с шага №1.

В псеводокоде это можно представить так:

```
// Располагается в общей для потоков памяти
let state = Atomic<State>

// Вызывается из разных потоков
fn doLockFreeOperation(){
    while(true){
        let old_state = state.atomic_read()

        let modified_state = modify(old_state)

        if(state.atomic_cas(old_state,modified_state)){
            // За время транзакции состояние не изменилось
            // Поток успешно завершает транзакцию
            break;
        } else{
              // Операция по замене неуспешна
              // Поток повторяет цикл
            continue;
        }
    }
}

fn main() {
    let thread_1 = spawn(doLockFreeOperation)
    let thread_2 = spawn(doLockFreeOperation)

    thread_1.join();
    thread_2.join();
}
```

Таблица 2 – Применение неблокирующей синхронизации.

Очевидно, что при таком подходе обязательно будет существовать поток или потоки, успешно завершающие свои транзакции, при этом остальным потокам нужно будет лишь повторить попытку. Синхронизация в описанном примере происходит в точках `state.atomic_read()` и `state.atomic_cas()`. При этом модификация состояния может происходить в параллель.

Необходимо также отметить доступность этого способа синхронизации на разных аппаратных платформах. Атомарные операции доступны только над машинными словами определённого размера. На большинстве архитектур, они не провосходят её битность (размера указателя). На данный момент самый распространённый размер - 64 бит, но также встречается и 128 бит.

Несмотря на свои плюсы, такой подход обладает одним серъездным недостатком. Необходимая линеаризуемость и следующая из неё синхронизация, образуется лишь вокруг одной ячейки памяти. Однако в большинстве структур данных чаще всего требуется атомарная замена сразу нескольких ячеек памяти. Любые прямые изменения хотя бы двух ячеек памяти влекут за собой потерю порядка исполнения, так как между двумя атомарными операциями может произойти произвольное число сторонних событий, в зависимости от решения приоритета исполнения планировщика операционной системы.

Для решения описанной проблемы была представлена транзакционная память. Её можно реализовать как на уровне процессора, так и программно. Так как сейчас процессоры в большинстве случаев не поддерживают подобную опцию, будет

рассмотрено использование программной реализации в виде примитива Multi Word Compare And Swap (MWCAS).

2.2.4 Обзор MWCAS

MWCAS - это операция для синхронизации памяти, которая считывает состояние N произвольных ячеек памяти, поддерживающих атомарные операции, описанные ранее, после чего сравнивает их с определённым множеством значений, и, если все из из этих значений совпадают со значениями, располагающимися в заданных ячейках памяти, заменяет их на новое множество.

При использовании программной транзакционной памяти, выделяются два типа операций, которые атомарны и линеаризуемы относительно друг-друга:

- mwcas_transaction()
- Специальная операция чтения mwcas_read()

Стандартные атомарные операции над одной ячейкой памяти в данном случае неприменимы.

Преимуществом использования программной транзакционной памяти является то, что она поддерживается на любых платформах, которые поддерживают атоманые операции над одной ячейкой памяти. Она также наследует все преимущества неблокирующей синхронизации.

Существенным недостатком этого вида синхронизации является то, что транзакционная память может оперировать только над ячейками одного типа, определяемым языком программирования из-за чего возникает необходимость в небезопасной конвертации типов. Не существует единого стандарта реализации транзакционной памяти. Также для использования транзакций доступ ко всем атомарным

ячейкам должен осуществляться в едином глобальном порядке, что обусловлено алгоритмом работы mwcas. Также стоит отметить, что в отличие от классическойй неблокирующей синхронизации, для работы программной транзакционной памяти требуются дополнительные аллокации, обусловленные алгоритмом её работы.

2.2.5 Обзор алгоритма работы MWCAS

Как уже было сказано раньше, программная транзакционная память в классическом варианте предоставляет две операции: mwcas_transaction, операцию множественного CAS, и дополнительную операцию чтения из одной атомарной ячейки - mwcas_read.

Каждая отдельная множественная транзакция условной замены ассоциируется с уникальным дескриптором mwcas, который заранее аллоцируется в общей памяти потоков. Даже если в рамках одной операции первая транзакция завершаилась неуспешно и поток повторяет операцию, ему понадобится новый дескриптор.

Дескриптор представляет собой объект, заключающий в себе всю информациию о совершаемой транзакции:

- Ссылки на набор атомарных ячеек памяти
- Список ожидаемых значений
- Список новых значений
- Статус транзакции: SUCCESS | UNDECIDED | FAILED

Алгоритм транзакции работает в две фазы:

В первой фазе транзакция пытается последовательно захватить все одиночные ячейки памяти, для последующего использования, путём размещения в них указателя на дескриптор (именно поэтому необходимо, чтобы он был аллоцирован в общей памяти),

предварительно сравнивая значения со старым множеством, используя операции единичного CAS.

При этом может возникнуть три исхода:

- Поток успешно захватывает все ячейки памяти, далее он переходит ко второй фазе.

- Поток может вместо старого значения в ячейке увидеть указатель на другой дескриптор. Это означает, что другая транзакция сейчас находится в процессе захвата своего множества атомарных ячеек. Поток обнаруживший указатель на дескриптор другой транзакции использует его для того, чтобы повторить первую фазу и помочь завершиться транзакции другого потока. Именно поэтому важно, чтобы доступ к ячейкам памяти был глобально упорядочен. В противном случае, если ячейки захватываются в разном порядке, может произойти ситауция взаимной блокировки, с последующими бесконечнымми повторами перекрёстных транзакций (livelock).

- Во время захвата ячеек поток обнаруживает, что есть хотя бы одна ячейка, старое значение в которой не совпадает, с проверяемым. Это означает, что во время фазы захвата текущей транзакции, произошла другая транзакция, успевшая завершиться. В данном случае, поток помечает транзакцию, как неуспешную, и возвращает пользователю значение false, что означает необходимость в повторении транзакции.

Во второй фазе, поток , в случае, если первая фаза прошла успешно. Последовательно меняет значение в атомарных ячейках с указателя на дескриптор своей транзакции на новое значение. Иначе, так как поток мог занять часть ячеек своей блокировкой,

он последовательно снимает эти блокировки, возвращая ячейкам их старые значения.

```
fn mwcas(cd: *MWCASDesciptor) -> bool{
    // Фаза 1
    if (cd.status == UNDECIDED){
        let status = SUCCESS
        for i = 0; i < cd.N; i++{
            if status != SUCCESS{
                break;
            }
        }
        loop{
            let atomic = cd.atomics[i]
            let value = (atomic.address, atomic.old_value,
cd) // ???

            if isDescriptor(value){
                // Другая транзакция в прогрессе
                if(value != cd){
                    // Помощь в завершении
                    mwcas(value);
                    // Продолжение текущей транзакции
                    continue;
                }
            }else{
                if value != atomic.old_value{
                // Другой поток успел завершить транзакцию
                    status = FAILED
                }
                // Захват ячейки прошёл успешно
                break;
            }
        }
        cd.status.cas(UNDECIDED, status)
    }
```

```
    //  Фаза  2
    let  suc  =  cd.status.load()  ==  SUCCESS
    for  i  =  0  ;  i  <  cd.N;  i++  {
        //  В  зависимости  от  результата  первой  фазы
        //  значениие  устанавливается  на  новое
        //  или  возвраащается  старое.
        let  new_value  =  if  suc  {
            cd.atomics[i].new_value
        }  else{
            cd.atomics[i].old_value
        }
        cd.atomics[i].address.cas(
            cd,
            new_value
        )
    }
}
```

Таблица 3 – Алгоритм MWCAS программной транзакционной памяти.

Операция mwcas_read отличается от обычной операции атомарно чтения в том, что во время считывания значения, она может увидеть адрес, являющийся адресом дескриптора транзакции, которая находится в прогрессе исполнения. В этом случае, операция чтения помогает другой транзакции завершиться, после чего пробует считать значение ещё раз.

```
fn  mwcas_read(atomic:  AtomicPtr<T>)  ->  T{
    loop{
        let  value  =  atomic.read()
        if  isDesciptor(value){
            //  Помощь  в  завершении  транзакции
```

```
            mwcas(value)
        }  else{
            return  (*value)
        }
    }
}
```

Таблица 4 – Алгоритм чтения в программной транзакционной памяти.

2.2.6 Синхронизация с использованием транзакционной памяти

Пример исполнения транзакции для трёх атомарных ячеек памяти, где каждый поток использует лишь часть из них может выглядеть следующим образом:

```
//  Ячейки  располагаются  в  общей  для  потоков  памяти.
//  Порядок  обращений  соостетвует  порядку
//  определения  ячеек  в  программе.
let  state_A  =  Atomic<State>
let  state_B  =  Atomic<State>
let  state_C  =  Atomic<State>

fn  doABTransaction(){
    while(true){
        let  old_state_A  =  mwcas_read(&state_A)
        let  old_state_B  =  mwcas_read(&state_B)

        let  modified_state_A  =  modify(old_state_A)
        let  modified_state_B  =  modify(old_state_B)

        //  На  каждую  транзакцию  нужна
        //  отдельная  аллокация  дескриптора
        let  mwcas  =  Mwcas::new()
```

```
        // Транзакция атомарно заменяет ожидаемые значения
        // на модифицированные копии.
        // Ссылки на ячейки передаются в нужном порядке
        // В случае, если наблюдаемое старое значение
        // изменилось, транзакция помогает завершиться другой
        // возможной транзакции и сообщает о
        // неуспешном завершении
        if(mwcas.transaction(
            memory_cell = [state_A, state_B]
            expected_states = [old_state_A, old_state_B],
            new_states = [modified_state_A, modified_state_B],
        )){
            // Поток успешно заменил все значения
            break;
        } else{
              // Транзакция прошла неуспешно
              // Поток повторяет цикл
            continue;
        }
    }
}

fn doBCTransaction(){
    while(true){
        let old_state_B = mwcas_read(&state_B)
        let old_state_C = mwcas_read(&state_C)

        let modified_state_B = modify(old_state_B)
        let modified_state_C = modify(old_state_C)

        let mwcas = Mwcas::new()

        if(mwcas.transaction(
            memory_cell = [state_B, state_C]
            expected_states = [old_state_B, old_state_C],
            new_states = [modified_state_B, modified_state_C],
```

```
        )){
            break;
        }  else{
            continue;
        }
    }
}

fn  main()  {
    let  thread_1  =  spawn(doABTransaction);
    let  thread_2  =  spawn(doBCTransaction);

    thread_1.join();
    thread_2.join();
}
```

Таблица 5 – Применение программной транзакционной памяти.

Можно видеть, что данный алгоритм не иммеет принципиальных различий с классической неблокирующей синхронизацией. Добавляется дополнительный объект координирующий множественную транзацкию - дескриптор mwcas. Единственное необходимое условие использования - соблюдение общего глобального порядка использования ячеек памяти в транзакциях.

3 Обзор модуля широковещательной рассылки Tokio

Модуль широковещательной рассылки фреймворка Tokio, является частью предоставляемого им пакета инструментов для синхронизации sync.

Модуль представляет собой программный канал для пересылки сообщений между между потоками программы в режиме многопоточного доступа Multiple Producer Multiple Consumer (MPMC): в канал конкуррентно могут отправлять сообщения сразу несколько потоков. При этом каждое значение доступно для чтения всем подписавшимся до момента отправки очередного сообщения потокам-читателям, чтение при этом может происходить конкурентно. достигается это за счёт дополнительного счётчика, устанавливаемого вместе с сообщением. Счётчик обновляется с добавлением или удалением потоков-читателей. При определённых обстоятельствах медленный поток-читатель может пропустить некоторые сообщения. В таком случае он переходит на последние актуальные сообщения. В случае если сообщений нет, поток-читатель становится в очередь ожидания очередного нового значения. Из-за этой особенности канала, операция отправки сообщения синхронная, в то время как ожидание потоками-читателями - асинхронное.

3.1 Внутреннее состояние канала

С точки зрения программной реализации канал представляет собой общее состояние в памяти, обращения к которому совершаются с помощью интерфейсов структур двух типов: Sender и Reciever. Каждая структура предоставляет лёгковесный способ управления каналом через определённый интерфейс.

```
pub struct Sender<T> {
    // Ссылка на общее состояние
    shared: Arc<Shared<T>>,
}

pub struct Receiver<T> {
    // Ссылка на общее состояние
    shared: Arc<Shared<T>>,

    // Локальная позиция для следующего чтения
    next: u64,
}
```

Таблица 6 – Структуры управляющие каналом.

Общее состояние содержит в себе:

1. Память самой очереди: она представляется в виде обычного массива слотов (структура типа Slot), логически представленного в виде кольцевого буффера (buffer). Каждому слоту в соответствие ставится RwLock - блокирующий примитив синхронизации позволяющий производить параллельное чтение, только при отсутвии одновременной записи. Доступ ко всем полям отдельного слота синхронизируется с помощью данного примитива.

Каждый слот содержит в себе:

- Текущее сообщение (val)
- Ассоциированное число потоков читателей, для которых доступно сообщение (rem)
- Логическую позицию слота (pos)

2. Битовая маска для быстрого определения позиции (mask). При создании канала, его длина округляется до ближайшей степени двойки.

3. Основная информация для координации операций над очередью, представлена структурой Tail, это:
   1. Логическая позиция нового следующего сообщения (pos)
   2. Текущее число потоков-читателей (rx_cnt)
   3. Состояние канала (открыт / закрыт) (closed)
   4. Глобальная очередь ожидания следующего значения, представленная в виде связного списка задач потоков-читателей, осуществляющих проверку канала (waiters)

Синхронизация доступа к этим полям осуществляется с помощью мьютекса. На данный примитив синхронизации приходится основная нагрузка при параллельном доступе.

4. Текущеее число потоков-отправителей (num-tx)

5. Примитив оповещения о закрытии последнего потока-читателя (notify_last_rx_drop)

```
struct Shared<T> {
        buffer: Box<[RwLock<Slot<T>>]>,

        mask: usize,

        tail: Mutex<Tail>,

        num_tx: AtomicUsize,

        notify_last_rx_drop: Notify,
```

```
}

struct Tail {
        pos: u64,

        rx_cnt: usize,

        closed: bool,

        waiters: LinkedList<Waiter, <Waiter as
linked_list::Link>::Target>,
}

struct Slot<T> {
        rem: AtomicUsize,

        pos: u64,

        val: UnsafeCell<Option<T>>,
}
```

Таблица 7 – Внутреннее состояние канала.

3.2 Алгоритм канала

Массив слотов ограничен в длине значением, задаваемым пользователем. В этом возникает необходимость из-за проблемы следующей из архитектуры канала: так как значение должно быть доставлено всем получателям, оно должно всё это время оставаться в памяти, и, если для каждого нового сообщения будет выделяться новая память, наличие лишь одного медленного потока-читателя может привести к переполнению памяти.

Две основные операции канала - отправление и получение сообщений.

Операция отправления - синхронная. Потоки-писатели не обязаны ждать потоков-получателей. В связи упомянутой раньше проблемой, при которой выделение памяти может привести к её неконтролируемогу росту, буффер очереди ограничен, и при этом новые сообщения, превышающие длину очереди перезаписывают старые. Потоки-читатели, не успевшие получить отправленные ранее сообщения, с помощью своего собственного локального счётчика позиции сообщений, обнаруживают неконсистентность и переводят счётчик на текущую актуальную позицию в очереди, добавляя к нему один круг длины очереди.

Псевдокод операции:

```
fn send(newValue: T){
    // Захват общей блокировки очереди
    let tail = tail.lock()

    // Захват блокировки слота,
    // в который будет произведена запись
    let slot = buffer[nextId].lockWrite()

    // Запись нового значения
    buffer[nextId].value = newValue

    // Обновление мета-информации и необходимых счётчиков
    slot.pos = tail.pos + 1
    slot.rem = tail.rx_cnt
    slot.val = newValue

    // Освобождение блокировки слота
    buffer[nextId].unlock()

    // Запуск на исполнение
    // всех ожидающих задач потоков-читателей
```

```
    queue.notify_all()

    // Общая разблокировка очереди
    tail.unlock()
}
```

Таблица 8 – Блокирующий алгоритм отправки сообщения.

Операция получения - асинхронная. При попытке получить сообщение, поток-читатель может обнаружить ситуацию при которой готовых сообщений в очереди нет. Это может произойти как сразу при совпадении локального счётчика потока-отправителя и счётчика позиции слота, так и после добавления дополнительного круга длины, в случае, описанном ранее. В этом случае поток-читатель добавляет задачу на очередную проверку очереди в специальную очередь ожидания. Потоки-отправители после отправки сообщений проверяют эту очередь и запланирую на исполнение все оставшиейся в ней задачи.

Псевдокод выполнения операции:

```
// taskHandler - структура, отвечающая за планирование
// задачи, вызвавшей recv, на исполнение

async fn recv(taskHandler) -> T {

    // Следующий слот на чтение значения
    let slot = buffer[nextId]

    // Блокировка слота на чтение
    // Допускается параллельное чтение
    slot.lockRead()
```

```
    //  Поток-читатель  отстал  от  актуальной  информации
    //  как  минимум  на  один  круг  очереди
    if  slot.pos  !=  self.next  {

        //  Блокировка  состояния  канала
        tail.lock()

        //  Добавление  одного  круга  очереди
        let  next_pos  =  slot.pos  +  buffer.len()

        //  Позиция  потока-читателя  совпадает
        //  с  суммой  позициии  слота
        //  и  длины  одного  круга  очереди.
        //  В  таком  случае  готового  значения  ещё  нет.
        //  Поток  оставляет  задачу  в  очереди  ожидания.
        if  self.next  ==  next_pos{
            qeueu.park(taskHandler)
        }

        //  Поток-читатель  утанавливает  указатель
        //  на  самое  старое  текущее  значение  в  канале.
        //  Все  остальные  значения  считаются  пропущенными.
        let  next  =  tail.pos  -  buffer.len()
        let  missed  =  next  -  self.next
        let  self.next  =  next

        //  Блокировка  состояния  канала  и  слота.
        slot.unlockRead()
        tail.unlock()

    }  else{
        //  Если  первая  проверка  показала,
        //  что  текущее  значение  self.next
        //  совпадает  с  позицией  слота,  это  означает,
        //  что  поток  читает  актуальную  информацию.
        //  В  таком  случае  он  инкрементирует  локальный
```

```
счётчик
        //  и  возвращает  значение.
        let  result  =  slot.value
        slot.unlock()
        self.next++
        return  result
    }

}
```

Таблица 9 – Блокирующий алгоритм получения сообщения.

Новый способ синхронизации позволит полностью избавиться от блокирующих примитивов Mutex и RwLock, составляющих основные проблемные точки при масштабировании использования канала.

4 Анализ оптимального дизайна для очереди

4.1 Выделение памяти

Существует два возможных варианта использования выделения памяти под сообщения:

1. Выделять под каждое новое сообщение новую ячейку памяти

2. Выделить ограниченное число ячеек с перезаписью (используется сейчас)

Использование первого варианта гарантирует то, что ни одно сообщение не будет потеряно. При этом, несмотря на то, что можно аллоцировать память группами (аллоцировать сразу несколько ячеек), это не избавит в принципе от необходимости данных аллокаций, что может существенно повлиять на производительность. Кроме того,

при наличие медленных потоков-читателей можно прийти к ситуации переполнения памяти.

При использовании второго способа присутсвует только одна аллокация - при инициализации буффера памяти и при этом отсутствует проблема переполнения памяти, однако остаётся проблема перезаписи уже с заполненных ячеек памяти и потенциальная потеря определённых сообщений потоками-читателями.

Так как риск потери сообщений и последующая обработка таких случаев, возложенная на потоки - читатели, во втором случае существенно меньше риска потери функционала программы в целом в первом случае, второй вариант предпочтителен.

4.2 Использование неблокируюещей синхронизации

Перед потоками-отправителелями становится задача атомарно произвести следующие операции:

- [Событие А] Обновить состояние следующего слота, в который производится очередная запись. Включает также обновление глобального счётчика позиции слотов, разделяемого между всеми потоками-отправителями.

- [Событие Б] Запланировать на исполнения все существующие в очереди ожидания задачи потоков-читателей, не увидивших значения при прошлом получении и ожидающих его сейчас.

Потокам-читателям необходимо атомарно произвести следующие операции во время очередной проверки очереди канала:

- [Событие В] Проанализировать значение последнего актуального слота: в случае, если поток-читатель отстал от хвоста очереди, вернуть

пользователю число пропущенных сообщений, переведя локальный счётчик на последнее актуальное сообщение, определяемое глобальным счётчиком канала.

- [Событие Г] В случае если актуальных слотов нет, в том числе если до этого произошло сокращение отсавания от актуального хвоста, описанное ранее, добавить задачу на повторную проверку в очередь ожидания.

Применение классической неболокирующей синхронизации с использованием одной атомарной переменной подразумевает создание единой точки синхронизации вокруг которой будут упорядочиваться обращения к структуре данных. В данном случае этого можно достичь путём создания указателя (атомарные операции возможны над максимальной битовой размерностью указателя архитектуры) на общее аллоцированное состояние.

Проблема возникает в том, что для такого подхода необходима полная копия глобального состояния на каждую отдельную операцию, включающее копию всей очереди ожидания и всего буффера очереди. В обоих случаях это приводит к дополнительным аллокациям памяти, что при большом числе потоков-читателей и/или большом размере буффера, становится нецелесообразно с точки зрения эффективного использования памяти.

Если же попробовать разделить состояние на несколько атомарных ячеек: Сделать отдельный указатель на очередь, а также отдельные указатели на каждый из слотов буффера, что решит проблему полного копирования буффера и очереди ожидания, может произойти ситуация, при которой исчёезнет общий порядок операций над совершаемой структурой данных и следующая из этого потерясинхронизации.

Если события А, Б, В, Г, описанные выше произойдут для двух потоков, один из которых поток-читатель, а другой отправитель, в следующем порядке: В, А, Б, Г, что допускается при наличии нескольких атомарных переменных, возникнет ситуация при которой состояние канала, успеет обновиться потоком-отправителем (поток-отправитель полностью завершит свою операцию включая планирование, всех задач потоков-читателей в очереди ожидания) до постановки задачи потока-читателя в очередь исполнения. В итоге получится ситуация при которой поток-читатель оставит задачу на повторную проверку в очереди, увидев, перед этим прочитав состояния канала и заключив, что актуальных сообщений нет, которая никогда уже не будет запланирована на исполнение.

В описанной проблеме естественным образом возникает потребность в возможности атомарно заменить сразу несколько ячеек памяти, которую удовлетворяет программная транзакционная память. Новый алгоритм канала будет использовать её в качестве своей основы для синхронизации.

5 Новый алгоритм модуля broadcast

5.1 Организация общего состояния канала

Для реализации алгоритма, основанного на использовании транзакционной памяти, в первую очередь выделим атомарные ячейки, участвующиев в транзакциях. Используемый тип атомарных ячеек - u64, так как большинство архитектур имеет в качестве размера указателя и соответсвующую аппаратную поддержку атомарных операций над машинными словами длиной в 64 бита. Все поля при этом выполняют те же функции, что и в блокирующей реализации алгоритма.

Стоит отдельно уточнить, что в поле waiters будет находится значение представляющее указатель на голову очереди в виде интрузивного списка задач.

```
struct Shared<T> {
        buffer: Box<[Slot<T>]>,

        mask: usize,

        pos: U64Pointer,

        rx_cnt: U64Pointer,

        closed: U64Pointer,

        waiters: U64Pointer,

        num_tx: U64Pointer,

        num_weak_tx: U64Pointer,

        notify_last_rx_drop: Notify,
}

struct Slot<T> {

        rem: U64Pointer,

        pos: U64Pointer,

        // Тип-обёртка над U64Pointer
        // С возможносьтю аллокации любого объекта
        val: HeapPointer<Option<T>>,
}
```

Таблица 10 – Внутреннее состояние канала с использованием транзакционной памяти.

Теперь необходимо определить порядок обращений к ним. Он должен быть глобально единым для всех операций над каналом.

Порядок определяется в соответсвии со следованием определения полей в структуре, начиная с глобального счётика позиции слота (Shared.pos). Внутри отдельного слота парядок обращений также последовательный. В общем порядке для структуры Shared, обращение к буфферу очереди (buffer) осуществляется в последнюю очередь и продолжается в порядке, определённом в рамке одного слота. При этом нет необходимости выстраивать единый порядок между разными слотами, так как в рамках любой операции у любого типа потока выполняется транзакция с сопуствующим изменением состояния максимум одного слота.

5.2 Алгоритм канала

В новом алгоритме вместо блокировки используется последовательное чтение состояния в порядке, определённом раньше, после чего оно модифицируется также, как и в блокирующей реализации, и производится множестввенная транзакция, в случае успеха которой происходит оповещение всей очереди ожидания.

```
pub fn send(value: T) {
    let shared = &self.shared;

    // Транзакционной памяти необходимо
    // закрепление в памяти
    // до использования в операциях.
    let guard = new Guard;
```

```
        loop {
                if shared.rx_cnt.read(&guard) == 0 {
                        return Err;
                }

                // Считывание необходимого состояние
                // всех ячеек

                let pos = shared.pos.read(&guard);
                let rem = shared.rx_cnt.read(&guard);
                let idx = pos & shared.mask;

                let slot = &shared.buffer[idx];
                let slot_rem = slot.rem.read(&guard);
                let slot_pos = slot.pos.read(&guard);
                let slot_val = slot.val.read(&guard);
                let new_val = value.clone();

                let waiters = shared.waiters.read(&guard);

                let mwcas = new Mwcas;

                // Описание и исполнение транзакции
                mwcas.cas(&shared.pos, pos, pos + 1);
                mwcas.cas(&slot.pos, slot_pos, pos);
                mwcas.cas(&slot.rem, slot_rem, rem);
                mwcas.cas(&slot.val, slot_val, Some(new_val));
                mwcas.cas(&shared.waiters, waiters, nullprt);

                if mwcas.exec(&guard) {
                        shared.queue.notify_all();
                        return rem;
                }
        }
}
```

Таблица 11 – Алгоритм неблокирующей отправки сообщения с исопльзованием транзакционной памяти.

```
fn try_recv(taskHandler) -> T {
    // Закрепление потока в памяти для транзакции
    let guard = new Guard;

    let idx = self.next & shared.mask;
    let slot = &shared.buffer[idx];
    loop {
            let slot_pos = slot.pos.read(&guard);

            // Произошло отставание
            // Поток-ситатель добавляет один круг очереди
            if slot_pos != self.next {
                let mut old_waker = None;

                let next_pos = slot_pos+shared.buffer.len();

                // Если позиция совпала
                // это означает одно из двух:
                // 1) Канал - закрыт
                // 2) Нет новых значений
                if next_pos == self.next {
                    // Канал для данного получателя пустой
                    // происходит отмена операции
                    if self.shared.closed.read(&guard) == true{
                            return Err(Closed);
                    }

                    // В противном случае происходит
                    // попытка добавления задачи
                    // в голову очереди
                    let waiters = shared.waiters.read(&guard);
                    waiters.next = taskHandler
```

```
                    let  cas  =  new  Mwcas;
                    cas.cas(
                            waiters,
                            (*waiter_ptr).next  as  u64,
                            waiter_ptr  as  u64,
                    );

                //  Если  поток  успешно  выполнил
                //  транзакцию,  ничего  больше  не  происходит.
                //  В  противном  случае,  необходимо  проверить
                //  канал  на  наличие  нового  значения
                    if  cas.exec(&guard)  {
                            return  Err(Empty);
                    }
                    continue;
                }

    //  На
        let  next  =  shared
                .pos
                .read(&guard)
                -  shared.buffer.len();

        let  missed  =  next  -  self.next;

        //  The  receiver  is  slow  but  no  values  have  been
missed
        if  missed  ==  0  {
                self.next  =  self.next.wrapping_add(1);
                return  Ok((*slot.val.read(&guard)).clone());
        }

        self.next  =  next;

        return  Err(TryRecvError::Lagged(missed));
        }
```

```
    self.next  =  self.next.wrapping_add(1);
    break;
}

Ok((*slot.val.read(&guard)).clone())
}
```

Таблица 12 – Алгоритм неблокирующего получения сообщения с использованием транзакционной памяти.

6 Сравнительный анализ производительности

Для benchmark-тестирования использовалась библиотека “criterion”. На каждый тест создаётся N задач для потоков-отправителей и K задач для потоков получателей. Длина буффера для канала в каждом тесте составляет 1000 элементов. Тестирование выполнялось с использованием 8 потоков ОС на 8 процессорных ядрах.

Тестирование заключается в том, что каждый из потоков отправителей отправляет по 10 сообщений, при этом процесс теста дожидается доставку всем потокам-читателям. Отдельный тестовый случай с определённым числом задач для потоков-отправителей и получателей исполняется по 100 раз. Перед началом создаются все необходимые задачи, ожидающие создания друг друга с помощью барьера (необходимо для того, чтобы создание задач не влияло на результаты тетсирования), после чего начинается отправка сообщений.

Представленные графики для каждого режима работы отображают распределение времени тестирования на отдельный тестовый случай, а также медианное значение.

Для удобства введём обозначение режима работы: “N/K” - режим при котором в канал отправляют сообщения N потоков и получают - K

6.1 Режим доступа SPSC

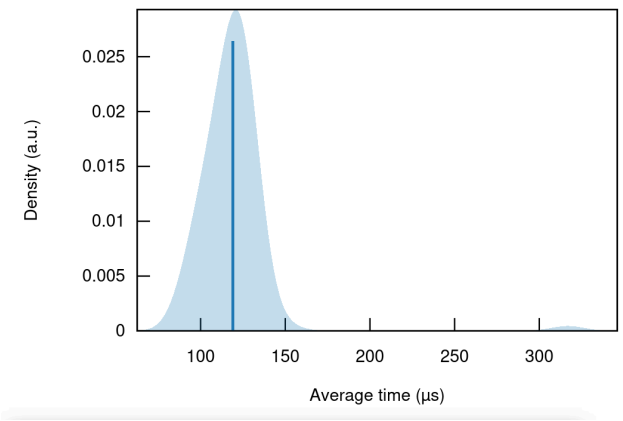

Рисунок 1 — График времени выполнения с неблокирующим каналом в режиме 1/1

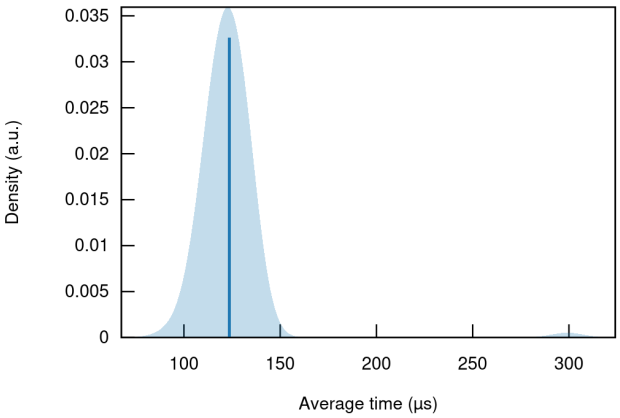

Рисунок 2 — График времени выполнения с блокирующим каналом в режиме 1/1

В режиме доступа 1/1, медианные значения составили:

- Блокирующая версия - 146.88 микросекунд
- Неблокирующая версия - 122.00 микросекунд

Прирост производительности - 20,39%

В данном тесте прирост относительно небольшой, так как при малом числе конкурирующих потоков, использование блокирующей синхронизации достаточно эффективно.

6.2 Режим доступа SPMC

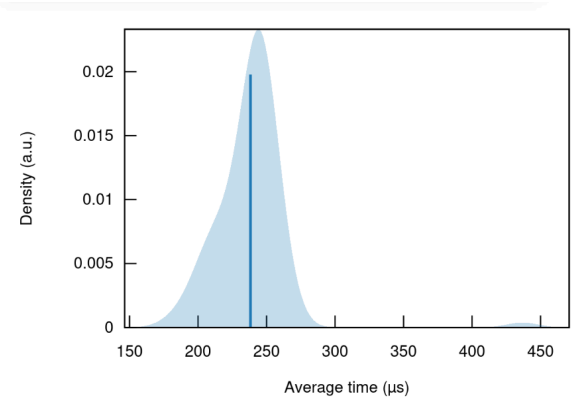

Рисунок 3 — График времени выполнения с неблокирующим каналом в режиме 1/4

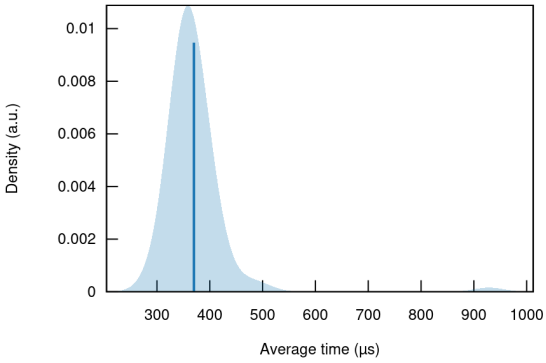

Рисунок 4 — График времени выполнения с блокирующим каналом в режиме 1/4

В режиме доступа 1/4, медианные значения составили:

- Блокирующая версия - 364.52 микросекунд
- Неблокирующая версия - 239.59 микросекунд

Прирост производительности - 52,14%

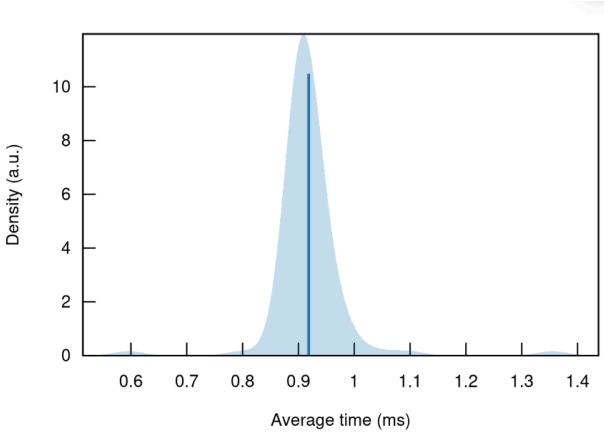

Рисунок 1 — График времени выполнения с неблокирующим каналом в режиме 1/32

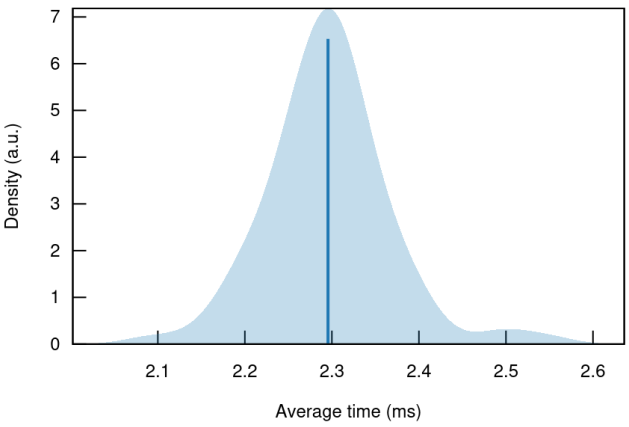

Рисунок 2 — График времени выполнения с блокирующим каналом в режиме 1/32

В режиме доступа 1/32, медианные значения составили:

- Блокирующая версия - 2.2955 миллисекунд
- Неблокирующая версия - 910.08 микросекунд

Прирост производительности - 151,23%

6.3 Режим доступа MPSC

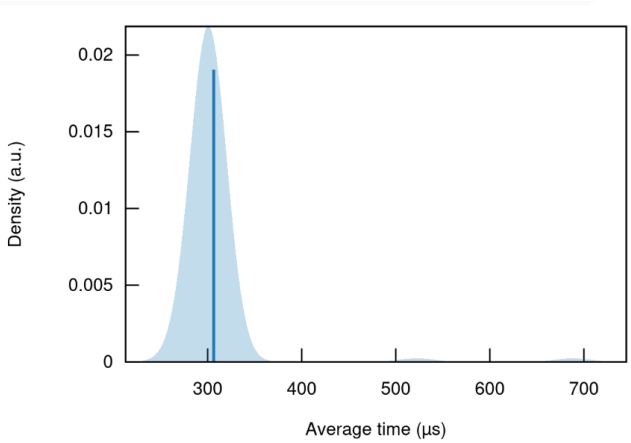

Рисунок 1 — График времени выполнения с неблокирующим каналом в режиме 4/1

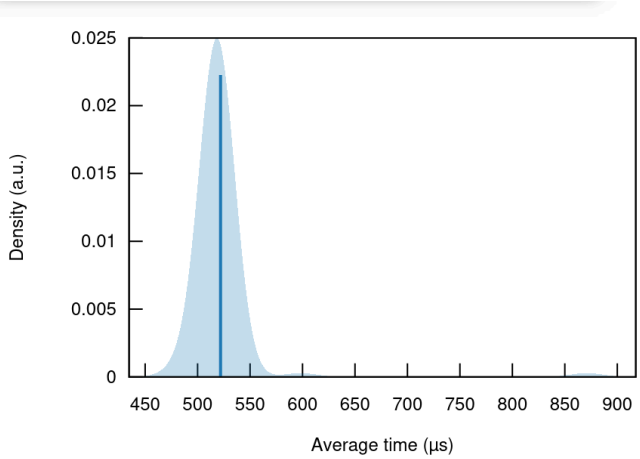

Рисунок 2 — График времени выполнения с блокирующим каналом в режиме 4/1

В режиме доступа 4/1, медианные значения составили:

- Блокирующая версия - 519.81 микросекунд
- Неблокирующая версия - 300.76 микросекунд

Прирост производительности - 72,83%

Можно дополнительно заметить, что прирост в данном случае больше чем, при сравнении с режимом 1/4. Связано это с тем, что потоки-писатели создают большую нагрузку на канал за счёт оперирования с разделяемым состоянием через Mutex, вместо параллельного RwLock для потоков-читателей в отдельных слотах. Это же замечание справедливо для тестов в режимах 1/32 и 32/1

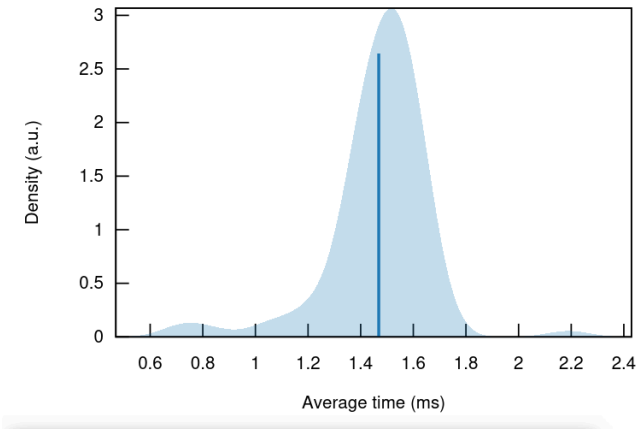

Рисунок 1 — График времени выполнения с неблокирующим каналом в режиме 32/1

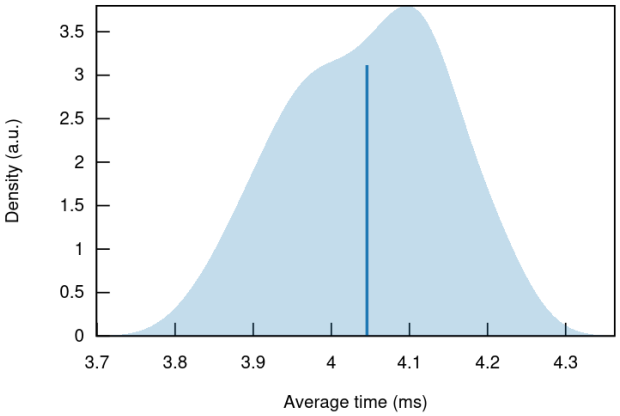

Рисунок 2 — График времени выполнения с блокирующим каналом в режиме 32/1

В режиме доступа 32/1, медианные значения составили:

- Блокирующая версия - 4.0457 миллисекунд
- Неблокирующая версия - 1.4837 миллисекунд

Прирост производительности - 173,67%

6.4 Режим доступа MPMC

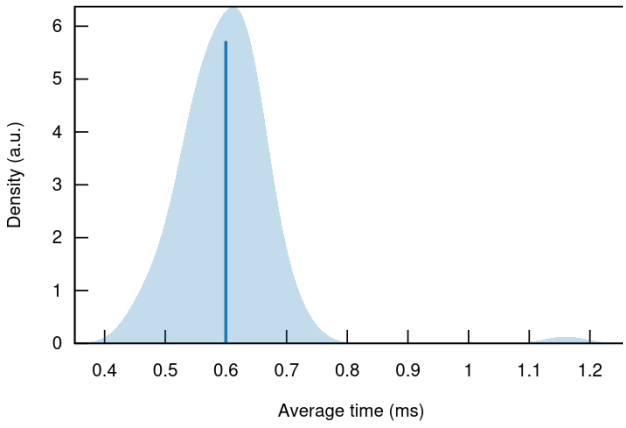

Рисунок 1 — График времени выполнения с неблокирующим каналом в режиме 4/4

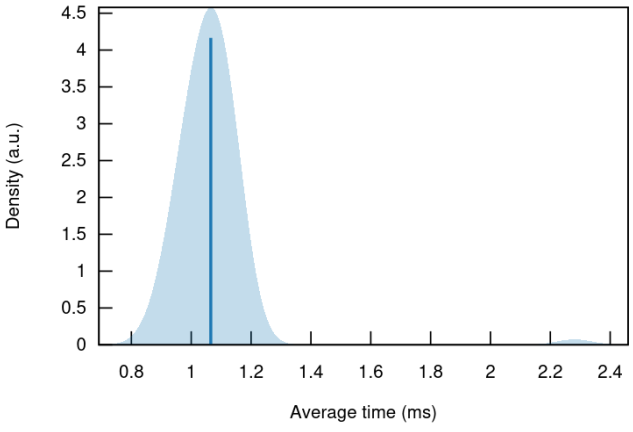

Рисунок 2 — График времени выполнения с блокирующим каналом в режиме 4/4

В режиме доступа 4/4, медианные значения составили:

- Блокирующая версия - 1.0613 миллисекунд
- Неблокирующая версия - 604.56 микросекунд

Прирост производительности - 75,54%

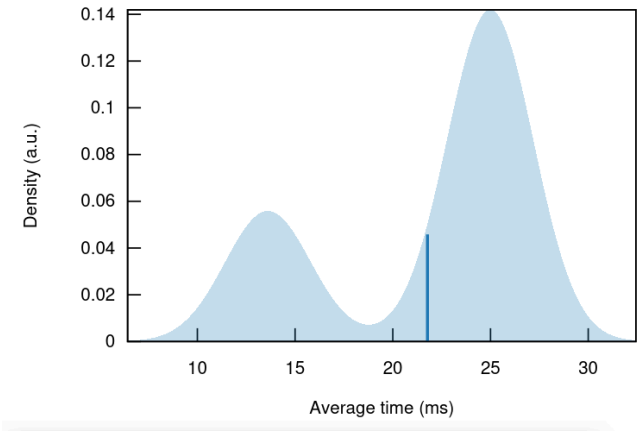

Рисунок 1 — График времени выполнения с неблокирующим каналом в режиме 32/32

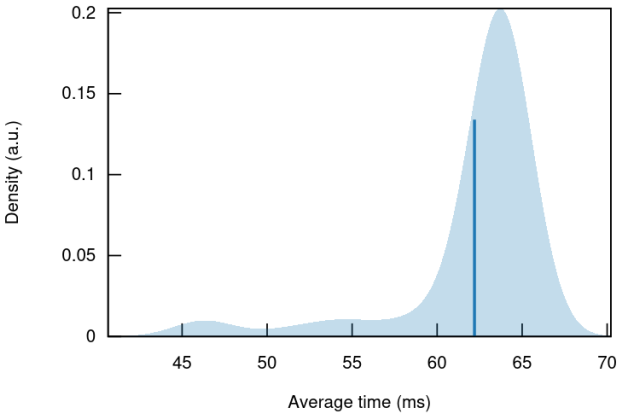

Рисунок 2 — График времени выполнения с блокирующим каналом в режиме 32/32

В режиме доступа 32/32, медианные значения составили:

- Блокирующая версия - 62.197 миллисекунд
- Неблокирующая версия - 21.774 миллисекунд

Прирост производительности - 185,64%

7 Заключение

В рамках выполнения работы был спроектирован новый алгоритм для блокирующего примитива широковещательной рассылки фреймворка Tokio, который позволяет более эффективным способом использовать многоядерное масштабирование инструмента. При этом были выполнены все поставленные задачи. Рассмотрение и анализ существующих возможностей программной реализации вместе с особенностями алгоритма рассматриваемого примитива синхронизации помогли определить оптимальный инструмент для реализации нового алгоритма. Анализ показал, что у каждого способа есть свои достоинства и недостатки.

В итоге, в качестве основы для нового алгоритма была взята реализация программной транзакционной памяти. Выбор был обусловлен невозможностью реализации алгоритма стандартными методами неблокирующей синхронизации. Данный способ предоставил все достоинтсва неблокирующей синхронизации вместе с заметной простотой, чего нельзя сказать о классическом подходе.

При сравнительном тестировании новое решение продемонстрировало более эффетивное использование процессорного ресурса системы по сравнению со своим предшественником в виде блокирующей синхронизации.

8 Список использованных источников

1. Требования к ВКР [Электронный ресурс] – URL: https://student.itmo.ru/files/1314 (дата обращения: 07.03.2024).
2. Документация фреймворка Tokio [Электронный ресурс] – URL: https://docs.rs/tokio/latest/tokio/index.html (дата обращения: 31.03.2024).
3. Документация по работе с атомарными переменными в Rust [Электронный ресурс] – URL: https://doc.rust-lang.org/std/sync/atomic (дата обращения: 31.03.2024).
4. The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit - 2008
5. Timothy L. Harris, Keir Fraser and Ian A. Pratt, A Practical Multi-Word Compare-and-Swap Operation [Электронный ресурс] – URL: https://www.cl.cam.ac.uk/research/srg/netos/papers/2002-casn.pdf (дата обращения: 13.04.2024).
6. Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, Igor Zablotchi, Efficient Multi-word Compare and Swap [Электронный ресурс] – URL: https://arxiv.org/pdf/2008.02527 (дата обращения: 13.04.2024).
7. Tianzheng Wang, Justin Levandoski, Per-Ake Larson, Easy Lock-Free Indexing in Non-Volatile Memory [Электронный ресурс] – URL: https://www2.cs.sfu.ca/~tzwang/pmwcas.pdf (дата обращения: 21.04.2024).
8. Nikita Koval, Dan Alistarh, Roman Elizarov, FAST AND SCALABLE CHANNELS IN KOTLIN COROUTINES [Электронный ресурс] – URL: https://arxiv.org/pdf/2211.04986 (дата обращения: 15.03.2024).
9. Nikita Koval — Synchronization primitives can be faster with SegmentQueueSynchronizer [Электронный ресурс] – URL: https://2020.hydraconf.com/2020/msk/talks/5vknrc3qqnuplsd9mi9baq/ (дата обращения: 20.03.2024).
10. Maurice Herlihy, J. Eliot B. Moss, Transactional Memory: Architectural Support for Lock-Free Data Structures [Электронный ресурс]

– URL: https://cs.brown.edu/~mph/HerlihyM93/herlihy93transactional.pdf (дата обращения: 21.04.2024).

11. Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy, Composable Memory Transactions [Электронный ресурс] – URL: https://cs.brown.edu/people/mph/HarrisMPJH05/stm.pdf (дата обращения: 20.04.2024).
12. Lock-free структуры данных [Электронный ресурс] – URL: https://habr.com/en/post/195770/ (дата обращения: 14.04.2024).

Приложение

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.