

Содержание

Список сокращений и условных обозначений	3
Введение	4
Цели и задачи работы	5
Введение в предметную область	5
1 Синхронизация	5
1.1 Блокирующая синхронизация	5
1.2 Неблокирующая синхронизация	7
1.3 Общий подход к неблокирующей синхронизации	8
2 MWCAS	9
Обзор модуля широковещательной рассылки Tokio	10
3 Архитектура	10
Основная часть	10
4 Актуальность темы исследования	11
4.1 Актуальность темы исследования. Часть 1	11
4.2 Цель и Задачи	11
Заключение	12
Список использованных источников	13
Приложение	14

Список сокращений и условных обозначений

- CAS - Compare And Swap
- MWCAS - Multi Word Compare And Swap
- ОС - Операционная Система
- MVCC - Multi Version Concurrency Control

Введение

На сегодняшний день в основе большинства разрабатываемых приложений: веб-серверов, кластеров обработки данных и.т.п. лежит инфраструктурная основа в виде надёжной и производительной среды предоставления асинхронного исполнения - runtime. 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 Elixir 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 неминуемо возникает потребность в синхронизации между ними. Синхронизация по своей природе может быть нескольких типов. Каждый из них предоставит 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 ЭВМ. В большинстве случаев - это блокирующая синхронизация, несмотря на то, что существуют способы производить операции в неблокирующем режиме, который открывает возможности 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 реализация становится практически невозможной.

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

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

Введение в предметную область

1 Синхронизация

Синхронизация между потоками по своей природе может быть разных видов, в понимании того, что они предоставляют пользователю разные гарантии прогресса многопоточного исполнения. Каждый вид на сегодняшний день имеет как свои достоинства, так и недостатки.

1.1 Блокирующая синхронизация

Первый тип представляет собой блокирование прогресса всех исполняемых задач на время выполнения одной или нескольких выбранных нескольких задач. У такого типа синхронизации есть большое преимущество - он простой и не требует специального подхода к построению структуры данных над которой производятся операции.

Самый простой пример такой синхронизации - использование мьютекса:

```
let mutex = Mutex::<State> // В общей памяти
```

```
fn doCriticalSection(){  
    {  
        mutex.lock()  
        // Критическая секция  
        mutex.unlock()  
    }  
}
```

Однако данный подход крайне неэффективно масштабируется, ведь в единицу времени может выполняться только одна критическая секция. Остальные потокам необходимо ждать освобождение блокировки. Ожидание может быть сопряжено с дополнительными системными вызовами, например `futex syscall`. Или же 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 накладных расходов, накладных расходов. В худшем случае такое исполнение может показать производительность сравнительно худшую чем обычное последовательное исполнение. Существует также проблема, при которой поток или процесс захвативший блокировку и исполняющий критическую секцию, будет временно снят с исполнения планировщиком задач операционной системы. В данном случае возникает риск полной остановки прогресса исполнения системы до разблокировки.

1.2 Неблокирующая синхронизация

Для избавления от проблем присущих блокирующей синхронизации, существует альтернативный подход, при котором, операции над данными осуществляются в “неблокирующем режиме”.

Неблокирующая синхронизация предоставляет следующие преимущества по сравнению с блокирующей:

- Гарантия прогресса системы в целом - означает, что при любом исполнении, всегда есть поток или потоки успешно завершающие свои операции. Решения планировщика ОС теперь не могут привести к полной остановке системы.
- Более высокая масштабируемость - 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.
- Меньшие накладные расходы на синхронизацию. Использование атомарных переменных не требует обращения к ядру операционной системы. Операции над атомарными переменными напрямую синхронизируют L2 кэш ядер процессора, за счёт чего достигается синхронизация памяти.

1.3 Общий подход к неблокирующей синхронизации

При использовании неблокирующей синхронизации образуется общий подход при построении структуры данных и операций над ней. Выделяется общее состояние, которое становится атомарной ячейкой памяти, в том плане, что все операции над ней линеаризуемы и образуют некоторый порядок обращений.

Все операции абстрактно 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 заново с шага №1.

Такое поведение соответствует MVCC над одной переменной. Абстрактно это можно представить так:

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

fn doLockFreeOperation(){
while(true){
    let old_state = state.read()
    let modified_state = modify(old_state)
    if(state.cas(old_state,modified_state)){
        break;
    } else{
        // Операция по замене неуспешна
        // Поток повторяет цикл
        continue;
    }
}
```



```
}  
}
```

Несмотря на свои плюсы, такой подход также обладает и серьёзными недостатками. Чаще всего логика в неблокирующих структурах данных намного сложнее чем та, что представлена выше: появляется необходимость в атомарной замене сразу нескольких ячеек памяти.

2 MWCAS

Обзор модуля широковещательной рассылки Tokio

3 Архитектура

Основная часть

- обзор mcas + pseudo
- Обзор tokio + pseudo текущей
- mcas tokio + pseudo
- аллокации в mcas возможные ускорения и реализации
- benches текущие и реализованные + картинки большие)

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequi doleamus animo, cum corpore dolemus, fieri.
(рисунок 1)



Рисунок 1 — пример изображения

4 Актуальность темы исследования

4.1 Актуальность темы исследования. Часть 1

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

4.2 Цель и Задачи

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

Заключение

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

Таблица 1 — Таблица

Таблица	для	примера
item1	$\sum_{k=0}^n k = 1 + \dots + n$	description1
item2	$\sqrt{2}$	description2

Список использованных источников

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

Приложение

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