

Санкт-Петербургский государственный университет

***ЕРИН Игорь Антонович***

Выпускная квалификационная работа

# Оптимизация алгоритмов injected queue в рантайме Tokio

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование  
информационных систем»*

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

Научный руководитель:  
Доцент кафедры системного программирования, к.ф.-м.н., Гориховский В. И.

Рецензент:  
Старший разработчик ПО, ООО “Ядро Центр Программных Разработок”,  
Черепанов А. П.

Санкт-Петербург  
2025

Saint Petersburg State University

*Igor Erin*

Bachelor's Thesis

# Optimization of injected queue algorithms in Tokio runtime

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2021 "Programming Technologies"*

Scientific supervisor:  
Docent, C.Sc. V. I. Gorikhovskii

Reviewer:  
Senior Software Developer, OOO "Yadro Centr Programmnykh Razrabotok" A. P. Cherepanov

Saint Petersburg  
2025

# Оглавление

|                                                              |           |
|--------------------------------------------------------------|-----------|
| <b>Введение</b>                                              | <b>4</b>  |
| <b>1. Постановка задачи</b>                                  | <b>5</b>  |
| <b>2. Асинхронный интерфейс языка Rust</b>                   | <b>6</b>  |
| 2.1. Waker . . . . .                                         | 7         |
| 2.2. Ключевое слово await . . . . .                          | 8         |
| 2.3. Вывод . . . . .                                         | 8         |
| <b>3. Обзор tokio</b>                                        | <b>10</b> |
| 3.1. Планировщик многопоточного рантайма tokio . . . . .     | 10        |
| 3.2. Интерфейс tokio . . . . .                               | 12        |
| 3.3. Жизненный цикл асинхронного замыкания . . . . .         | 13        |
| 3.4. Вывод . . . . .                                         | 15        |
| <b>4. Система бенчмарков</b>                                 | <b>16</b> |
| 4.1. Реализация бенчмарков . . . . .                         | 17        |
| <b>5. Исследование производительности tokio</b>              | <b>19</b> |
| 5.1. Условия экспериментов . . . . .                         | 19        |
| 5.2. Ход исследования . . . . .                              | 20        |
| 5.3. Вывод . . . . .                                         | 21        |
| <b>6. Проектирование решения основанного на шардировании</b> | <b>22</b> |
| 6.1. Шардирование планировщика . . . . .                     | 22        |
| <b>7. Производительность решения</b>                         | <b>24</b> |
| 7.1. Ход исследования . . . . .                              | 24        |
| 7.2. Выводы . . . . .                                        | 27        |
| <b>8. Заключение</b>                                         | <b>28</b> |
| <b>Список литературы</b>                                     | <b>30</b> |

# Введение

TATLIN.BACKUP<sup>1</sup> — проект группы компаний YADRO<sup>2</sup>, посвященный созданию системы бекапов с использованием алгоритмов дедупликации, написанный на языке Rust [2]. Rust предоставляет `async / await`[16] интерфейс для обработки асинхронных событий. Язык не фиксирует реализацию, позволяя пользователю выбирать асинхронный рантайм.

`tokio`<sup>3</sup> — проект, предоставляющий две реализации асинхронного рантайма: так называемые `current_thread` и `multi_thread` рантаймы, частично реализует стандартную библиотеку языка с асинхронным интерфейсом, примитивы синхронизации, коллекции, таймеры, профилировщики — целую экосистему для написания асинхронных программ.

Многопоточный рантайм из `tokio` используется в TATLIN.BACKUP, где был замечен недостаток текущей реализации: общая очередь асинхронных событий, разделяемая всеми потоками `tokio`, защищена мьютексом и становится бутылочным горлышком по предположению инженеров из YADRO.

Шардирование [13] — это известный метод разделения системы на независимые составляющие, способные выполнять все функции исходной системы. Данная работа посвящена исследованию увеличения пропускной способности многопоточного рантайма `tokio` в типичных для TATLIN.BACKUP сценариях с помощью шардирования планировщика.

---

<sup>1</sup>TATLIN.BACKUP — система хранения данных резервных копий (Дата обращения: 25.5.2025)

<sup>2</sup>YADRO — официальный сайт компании (Дата обращения: 25.5.2025)

<sup>3</sup>tokio — официальный сайт проекта (Дата обращения: 25.5.2025)

# 1. Постановка задачи

Целью данной работы является увеличение пропускной способности tokio в типичных для TATLIN.BACKUP сценариях. Для чего были поставлены следующие задачи:

1. Анализ архитектуры tokio для выявления потенциальных критических областей.
2. Разработка системы бенчмарков, специфичной для TATLIN.BACKUP, позволяющей получать воспроизводимый результат.
3. Анализ собранных метрик и локализация проблемы.
4. Проектирование решения, основанного на шардировании и его реализация.
5. Анализ производительности решения.

## 2. Асинхронный интерфейс языка Rust

В данной главе будет рассмотрен асинхронный интерфейс языка Rust<sup>4</sup>. Это необходимо для объяснения реализации асинхронных рантаймов.

На листинге 1 представлена синтаксическая конструкция, описывающая асинхронное замыкание.

```
let closure = async { }
```

**Листинг 1: Асинхронное замыкание.**

Асинхронное замыкание — это значение, представляющее конечный автомат, сгенерированное компилятором на основании пользовательского кода, использованного в фигурных скобках после ключевого слова `async`. На листинге это значение связано с именем `closure` и представляет собой тривиальный конечный автомат. Каждое сгенерированное компилятором асинхронное замыкание автоматически реализует интерфейс, представленный на листинге 2.

```
trait Future {  
    type Output;  
    fn poll(self: Pin<&mut Self>, cx: &mut Context)  
        -> Poll<Self::Output>;  
}
```

**Листинг 2: Интерфейс асинхронных замыканий в языке Rust.**

Где метод `poll()` выражает попытку совершить переход от состояния к состоянию и сигнализирует о завершении выполнения замыкания с помощью возвращаемого значения типа, представленного на листинге 3.

---

<sup>4</sup>[Документация](#) асинхронного интерфейса (Дата обращения: 25.5.2025)

```
enum Poll<T> { Ready(T), Pending }
```

### Листинг 3: Асинхронное замыкание.

Способного представить результирующее значение асинхронного вычисления в варианте `Ready` или сигнализировать об отсутствии готового результата с вариантом `Pending`.

Метод `poll()` в качестве первого аргумента принимает само асинхронное замыкание, окруженное структурой `Pin`, — это необходимо для статической гарантии безопасности. В качестве второго аргумента — контекст, служащий обёрткой для значения `Waker`.

## 2.1. Waker

Типичным сценарием использования значения `Waker` является:

- Создание с помощью специфичной для рантайма таблицы виртуальных методов.
- Передача в метод `poll()` внутри структуры `Context`.
- Если метод `poll()` вернул `Pending` — замыкание должно зависать от реализации способом сохранить `Waker` для вызова его метода `wake()` при готовности совершать прогресс.

То есть `Waker` является связующим звеном между асинхронным ресурсом и исполнителем асинхронных замыканий, сообщаящим первому о готовности второго.

- Асинхронный ресурс — листовое асинхронное замыкание, реализованное библиотекой `tokio`, абстрагирующее архитектурно зависимые интерфейсы, например: `epoll` [17], `kqueue` [7].
- Исполнитель асинхронных замыканий — в случае `tokio` планировщик.

## 2.2. Ключевое слово `await`

Ключевое слово `await` позволяет пользователю описать состояние конечного автомата, путем применения его к асинхронному замыканию, как представлено на листинге 4.

```
async { sleep(10).await }
```

### Листинг 4: Использование ключевого слова `await`.

Применение ключевого слова `await` к вызову функции `sleep()`, которая является листовым асинхронным замыканием и представляет архитектурно зависимый асинхронный ресурс, генерирует состояние конечного автомата. Исполнение этого замыкания предполагает следующие шаги:

1. Первый вызов метода `poll()` совершит переход от начального состояния к состоянию вызова функции `sleep()`, зарегистрирует структуру `Waker` переданную в вызов метода `poll()` с помощью архитектурно зависимого интерфейса для ожидания истечения десяти секунд.
2. По истечении десяти секунд, архитектурно зависимый интерфейс сигнализирует о готовности замыкания совершать прогресс путем вызова метода `wake()` на структуре типа `Waker`.
3. Получив уведомление о готовности асинхронного замыкания, исполнитель продолжит его исполнение с помощью очередного вызова метода `poll()`.

## 2.3. Вывод

Таким образом асинхронные замыкания описанные пользователем с помощью синтаксической конструкции `async`, ключевого слова `await` и библиотечных методов абстрагирующих машинно зависимые детали реализации: таймеры, файловые и сетевые интерфейсы. Асинхронные

замыкания исполняются рантаймом путем ожидания вызова метода `wake()` и последующего исполнения метода `poll`.

### 3. Обзор tokio

Настоящая глава содержит описание реализации многопоточного рантайма tokio. К сожалению, tokio не имеет архитектурной документации, а работы [12], связанные с исследованием tokio, не многочисленны и не содержательны, поэтому данная глава призвана создать представление об устройстве планировщика многопоточного рантайма для последующего исследования его производительности.

#### 3.1. Планировщик многопоточного рантайма tokio

При создании рантайм создаёт определённое количество системных потоков для так называемого блокирующего пулла, призванного исполнять ресурсоёмкие задачи. Всего создаётся `worker_threads + max_blocking_threads` потоков, где

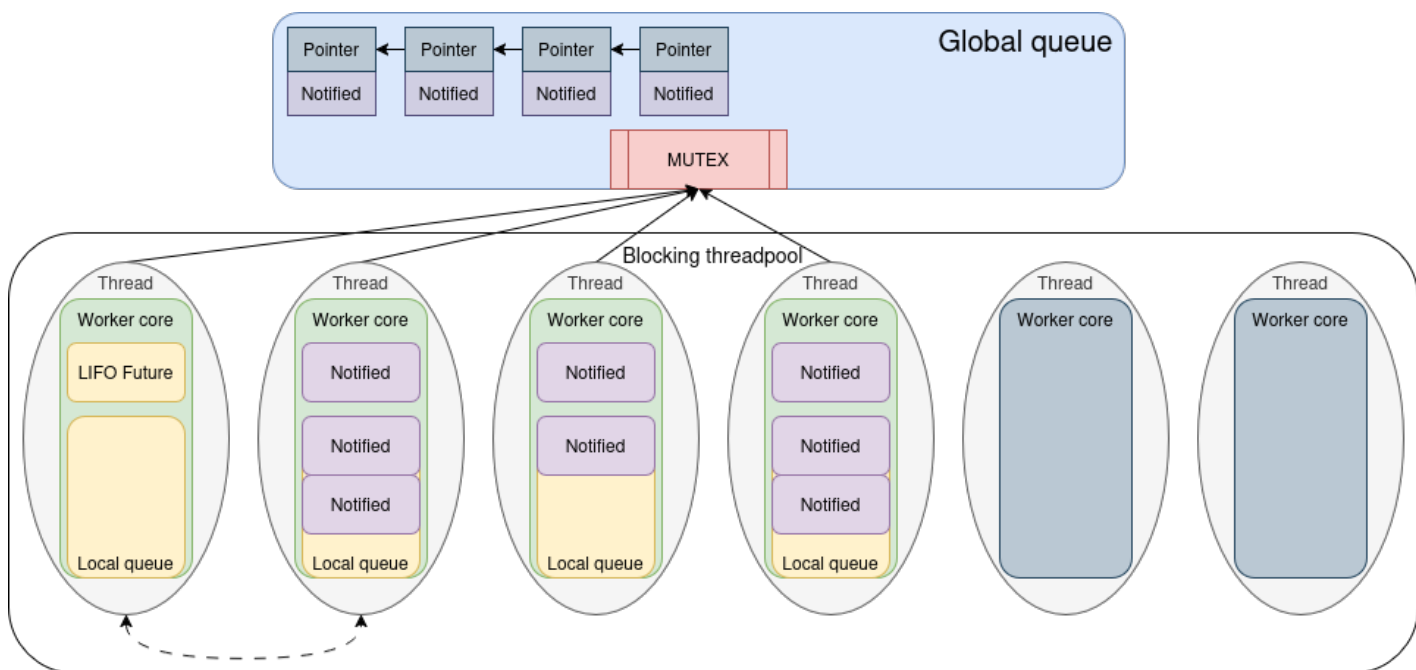


Рис. 1: Упрощенное представление многопоточного рантайма.

- `worker_threads` — количество потоков, предназначенных для исполнения асинхронных задач

- `max_blocking_threads` — максимальное количество блокирующих потоков

`Notified` — тип единицы планирования в `tokio`. Далее в тексте они также будут называться задачами.

Исполнитель, так называется сущность, исполняющая задачи. Она ассоциирована с каждым из `worker_threads` потоков с помощью размещения в локальной для этих потоков переменной так называемого ядра исполнителя — структуры, необходимой для исполнения асинхронных задач, включающей локальную очередь, указатель глобальной очереди и тому подобное.

Локальная очередь исполнителя выступает в качестве кэша задач, имеет фиксированный размер и предполагает добавление элементов исключительно из потока владельца. Однако, изъятие из нее может быть осуществлено потоками других исполнителей при нехватке задач — процесс, называемый похищением (stealing) [12]. Все это: фиксированный размер и единственность производителя — позволяет ей иметь lock-free реализацию. В случае переполнения часть элементов перемещается в глобальную очередь.

Глобальная очередь или `InjectedQueue` представляет собой очередь произвольного размера для агрегации задач. Она реализована с помощью интрузивного связного списка [6], защищенного мьютексом. Именно такая реализация по мнению разработчиков из YADRO ограничивает пропускную способность рантайма.

В рамках данной работы будет рассмотрен единственный метод интерфейса планировщика — `schedule_task()`. Именно он используется для отправления на исполнение в планировщик задач. Большое значение имеет контекст вызова этого метода:

- В потоке исполнителя задача будет помещена в его локальную очередь.
- В ином случае, задача будет помещена в `InjectedQueue`.

## 3.2. Интерфейс tokio

Вследствие выше сказанного любое асинхронное замыкание можно исполнить достаточно долго, вызывая на нем метод `poll()`, передавая в него `Waker` — пустышку. Однако, для удобства и эффективности использования асинхронного интерфейса, предоставляемого языком Rust, проект `tokio` предлагает собственный интерфейс для исполнения асинхронных замыканий.

### 3.2.1. tokio::spawn()

Метод `tokio::spawn()`, регистрирующий асинхронное замыкание на исполнение в рантайме `tokio`, возвращает значение `JoinHandle`, позволяющее ожидать исполнения исходного замыкания, получать результирующее значение или отменять исполнение вовсе. Например, исполнить тривиальное замыкание можно способом, приведённым на листинге 5:

```
let join: JoinHandle<()> = tokio::spawn(async {});
```

**Листинг 5: Пример исполнения замыкания с помощью tokio.**

Где пользователь аллоцирует замыкание в `tokio` рантайме передавая его в метод `tokio::spawn()`.

### 3.2.2. tokio::yield\_now()

Метод `tokio::task::yield_now()` предоставляет возможность вернуть поток управления планировщику, тем самым создав еще одно состояние в конечном автомате асинхронного замыкания. Его использование приведено на листинге 6.

```
async { task::yield_now().await }
```

**Листинг 6: Пример возвращение управления планировщику в tokio.**

При исполнении замыкания, дойдя до вызова `yield_now()`, управление вернётся исполнителю замыкания.

`yield_now()` является листовым асинхронным замыканием, однако, никакого настоящего ресурса под собой не содержит. То есть отсутствует внешняя сущность, способная уведомить рантайм о готовности замыкания продолжать вычисления — некому хранить `Waker`. В подобных случаях для сохранения инстансов типа `Waker` используется, локальная для потока исполнителя, коллекция, куда помещается структура `Waker`.

### 3.3. Жизненный цикл асинхронного замыкания

В данном разделе будет описан процесс аллокации асинхронного замыкания и создания первой единицы планирования, ассоциированной с ним.

#### 3.3.1. Аллокация асинхронного замыкания

Обработка асинхронного замыкания, переданного пользователем в метод `tokio::spawn()` содержит следующие шаги:

- Создание уникального идентификатора. Происходит это с помощью статической атомарной переменной, над которой выполняется FAA (Fetch and Atomic Addition) с `relaxed` семантикой.
- Аллокация структуры, содержащей замыкание, счётчик указателей и слот для результирующего значения, — так называемой `Cell` на куче.
- Создание трёх указателей на эту область памяти: `Task`, `Notified` и `JoinHandle`.
  - `Task` помещается в `OwnedTasks` — очередь, предназначенную для хранения указателей на все замыкания, исполняемые рантаймом.
  - `Notified` отправляется в планировщик, с помощью метода `schedule_task()`.

– `JoinHandle` возвращается пользователю.

### 3.3.2. Исполнение замыкания

Далее, происходит исполнение замыкания: после вызова метода `wake()` у соответствующего инстанса типа `Waker`, новый указатель `Notified` отправляется в планировщик `tokio` рантайма с помощью метода `schedule_task()`. Далее у `Notified` будет вызван метод `poll()`, что повлечет исполнение метода `poll()` на исходном замыкании.

Таким образом, по готовности каждого асинхронного замыкания, уже аллоцированного `tokio` рантаймом, создается указатель `Notified`, отправляемый на исполнение в планировщик с `schedule_task()`.

### 3.3.3. Удаление замыкания

При достижении конечного состояния `Task` удаляется из `OwnedTasks`, `Cell` деаллоцируется — вызывается его деструктор, запись об этом замыкании удаляется из `OwnedTasks`.

### 3.3.4. Цикл работы исполнителя

Исполнитель пытается найти задачу в локальной очереди, затем в глобальной очереди. Не найдя задач в очередях, он предпримет попытку похищения задач из локальных очереди других исполнителей. Локальная очередь имеет `lock-free` реализацию, поэтому, для уменьшения количества синхронизаций, рантайм `tokio` пытается ограничить количество исполнителей одновременно предпринимающих попытки похитить задачи. Если похитить задачи не удалось, исполнитель проверяет локальную для своего потока коллекцию структур типа `Waker` и вызывает метод `wake()` на каждом из таковых. Вызов метода `wake()` повлечет планирование структуры типа `Notified` с помощью метода `schedule_task()`, тем самым насыщая локальную очередь исполнителя. Если после этого очереди остались пустыми, исполнитель уведомляет операционную систему о необходимости прекратить исполнение

данного потока для экономии ресурсов. Исполнение потока будет возобновлено при появлении задач, доступных исполнителю. Заполучив задачу, исполнитель один раз вызывает ее метод `poll()`.

### 3.4. Вывод

Таким образом взаимодействие с глобальной очередью происходит исключительно при готовности асинхронных замыканий, аллоцированных `tokio` рантаймом, совершать прогресс: структура типа `Notified` помещается в локальную очередь, что может повлечь переполнение и вызвать необходимость перемещения части задач в глобальную очередь, задача, помещенная в глобальную очередь, изымается исполнителем.

Эту проблему можно решить двумя подходами.

- При достаточном уровне параллелизма и количестве исполнителей дополнительные синхронизации, вызванные похищением задач и взаимодействием исполнителей с глобальной очередью, могут ограничивать производительность системы.
- Другой причиной может являться реализация глобальной очереди: интрузивный связный список требует дополнительных действий при изъятии из глобальной очереди, что увеличивает критическую секцию и может негативно сказываться на производительности.

В представленной работе рассматривается только первый подход, тогда как второй рассматривается в работе ВКР Матвея Калашникова: “Оптимизация системы управления очередью задач в библиотеке `Tokio` для асинхронного рантайма”. Поскольку оба подхода рассматривают разные аспекты оптимизации, они могут давать независимый прирост производительности. Поэтому, возможно, что их совместное применение даст наилучший эффект.

## 4. Система бенчмарков

Бенчмарки в репозитории проекта `tokio` не пригодны для анализа производительности в рамках данной работы, так как используют методы синхронизации имеющие нетривиальный эффект на результатах экспериментов:

- Атомарное чтение и запись из множества исполняемых разными потоками задач скорее позволяет измерять производительность системы памяти физической машины, нежели взаимодействия структур рантайма [5].
- Ожидание исполнения в методе `block_on` `tokio` рантайма чревато неточным измерением из-за текущей реализации<sup>5</sup>.

Поэтому в качестве системы бенчмарков был создан проект `tokiobench`<sup>6</sup>, где предполагалось реализовать сценарии использования проекта `tokio` в `TATLIN.BACKUP`. При общении с командой `TATLIN.BACKUP` был выделен основной сценарий, представленный на рисунке 2:

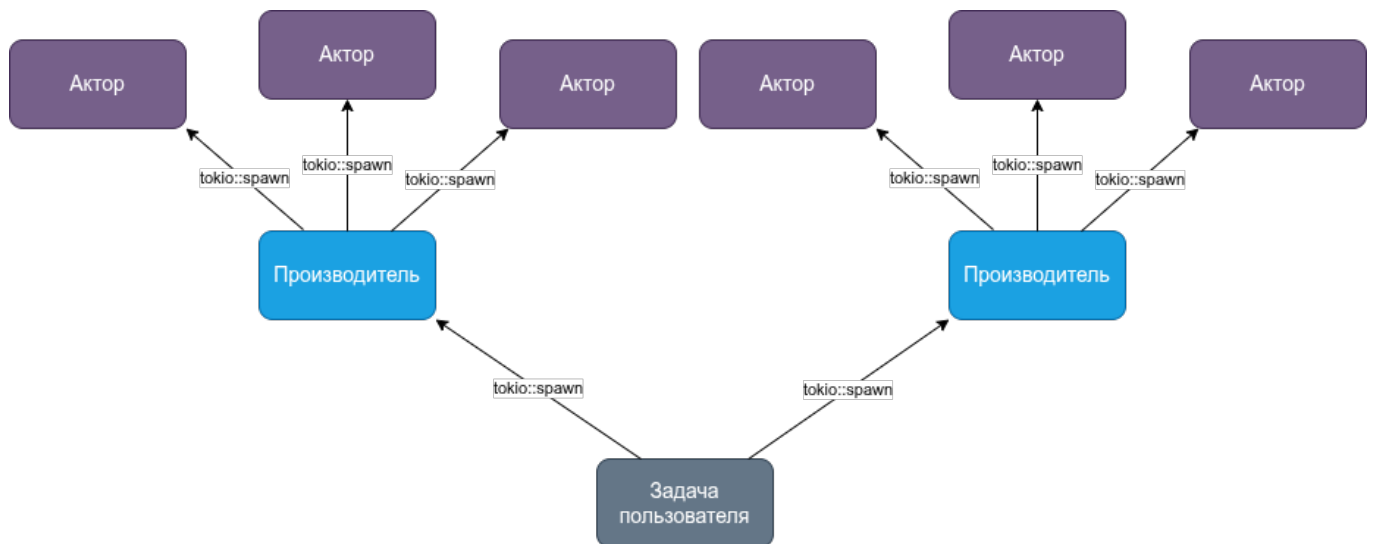


Рис. 2: Основной сценарий использования `tokio` в `TATLIN.BACKUP`.

<sup>5</sup>Проблема измерения производительности асинхронных функций (Дата обращения: 25.5.2025)

<sup>6</sup>Репозиторий проект `tokiobench` (Дата обращения: 25.5.2025)

- Актор — замыкания, в цикле ожидающие асинхронные события.
- Производитель — замыкания, отправляющее на исполнение множество замыканий акторов.

Необходимо было исследовать производительность такого сценария при следующих параметрах:

- Тысяча замыканий акторов.
- Тысяча итераций исполнения акторов.
- Сто замыканий производителей.

## 4.1. Реализация бенчмарков

Бенчмарки `tokio` не предполагают измерения производительности, а призваны лишь отслеживать регрессии при изменениях исходного кода. Поэтому в `tokiobench` реализуются бенчмарки со следующими изменениями:

- атомарная переменная была заменена иерархией преаллоцированных буферов, переиспользуемых от итерации к итерации и коллекционирующих структуры типа `JoinHandle`;
- ожидание структур типа `JoinHandle` перенесено из текущего потока с методом `block_on` в отдельную задачу, которая сообщает о завершении обработки с помощью блокирующего канала.

Все это призвано сократить наблюдаемые эффекты от реализации системы бенчмаркинга.

Использование блокирующего канала приводит к появлению дополнительных накладных расходов. Для их учета была создана реализация использующая спинлок: фиксирующий исполнение и сигнализирующий поток пишет значение в атомарную переменную, поток бенчмарка ожидает этого в цикле.

Для эмуляции операций ввода вывода был использован метод `yield_now()`. А именно, каждый актор тысячу раз выполняет `yield_now()`, что заставляет исполнителя сохранять структуру типа `Waker` в локальной коллекции, для последующего вызова метода `wake()`. После чего вызов метода `wake()` приводит к переполнению локальной очереди исполнителя, вынуждая последнего перемещать задачи из локальной очереди в глобальную. После исполнения задач из локальных очередей исполнители начинают обращаться к глобальной очереди, затем похищают задачи друг у друга. Такая реализация приводит к повышению числа взаимодействий у исполнителей, что способствует более объективной оценке накладных расходов этого взаимодействия.

## 5. Исследование производительности tokio

Целями данной главы является изучение изменения производительности при разделении константного количества исполнителей между различным количеством инстансов асинхронных рантаймов: если синхронизации исполнителей накладывают ограничение на производительность системы, использующей один асинхронный рантайм, использование нескольких асинхронных рантаймов с меньшим количеством исполнителей должно продемонстрировать большую пропускную способность.

### 5.1. Условия экспериментов

Для измерения производительности была использована библиотека `criterion`<sup>7</sup>. Так как она популярна, имеет обширную документацию и используется в `tokio`.

Здесь и далее эксперименты производились при следующих условиях:

- Исследования проводились на системе YADRO VEGMAN Rx20 G2<sup>8</sup>.
- Бенчмарк был запущен со значением `nice` равным минус двадцати.
- Исполнение было рекомендовано на одной NUMA единице с помощью `taskset`.

Машина для измерения производительности была предоставлена командой TATLIN.BACKUP и использовалась удаленно, в связи с чем на ней было невозможно отключение сети.

Библиотека `criterion` имела следующую конфигурацию:

- Пять секунд прогрева.

---

<sup>7</sup>[Репозиторий](#) проекта `criterion` (Дата обращения: 25.5.2025).

<sup>8</sup>[Описание](#) системы YADRO VEGMAN Rx20 G2 (Дата обращения: 25.5.2025).

- Сорок семплов для каждого измерения.
- Линеаризация в качестве способа семплирования.

## 5.2. Ход исследования

Для проверки гипотезы об ограничении производительности ресурсами одного рантайма, были произведены измерения пропускной способности системы, состоящей из нескольких рантаймов, приведенные на графике 3. Каждое измерение использовало одинаковое количество потоков исполнителей и задач производителей, равномерно разделенных между рантаймами.

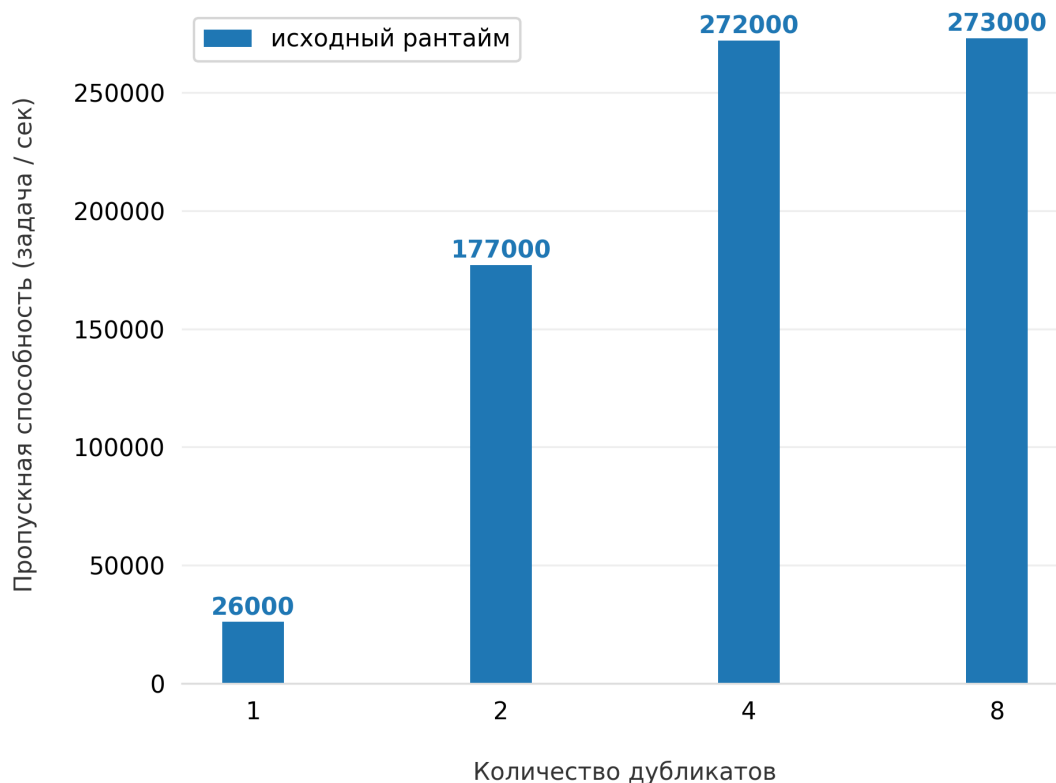


Рис. 3: Производительность системы при использовании нескольких рантаймов.

Из изображения заметным становится увеличение производительности при увеличении количества рантаймов, то есть разделение задач

между меньшими изолированными группами исполнителей позволяет увеличить пропускную способность.

### 5.3. Вывод

При изоляции меньших групп исполнителей с отдельными глобальными очередями в рамках отдельных инстансов рантаймов наблюдается увеличение пропускной способности. Это может происходить из-за уменьшения количества синхронизаций между исполнителями.

- Меньшая борьба за глобальную очередь, за счет деления исполнителей между несколькими очередями.
- Меньшее количество исполнителей пытаются украсть задачи у соседей из-за изоляции исполнителей в различных инстансах рантайма.

Таким образом, шардирование рантайма позволяет увеличить пропускную способность системы.

## 6. Проектирование решения основанного на шардировании

После полученных результатов команда `TATLIN.BACKUP` провела эксперименты, где так же стало очевидно преимущество использования нескольких рантаймов с точки зрения пропускной способности.

Однако, неясным осталась надежность такой системы: разработчики `tokio` не дают никаких гарантий, а наоборот утверждают, что при некотором обращении с такой конструкцией может быть наблюдаемо падение производительности или достижимо неопределенное поведение. `TATLIN.BACKUP` позиционирует себя как высоконадежное производительное решение, а потому использовать в нем конструкцию без гарантий не возможно.

### 6.1. Шардирование планировщика

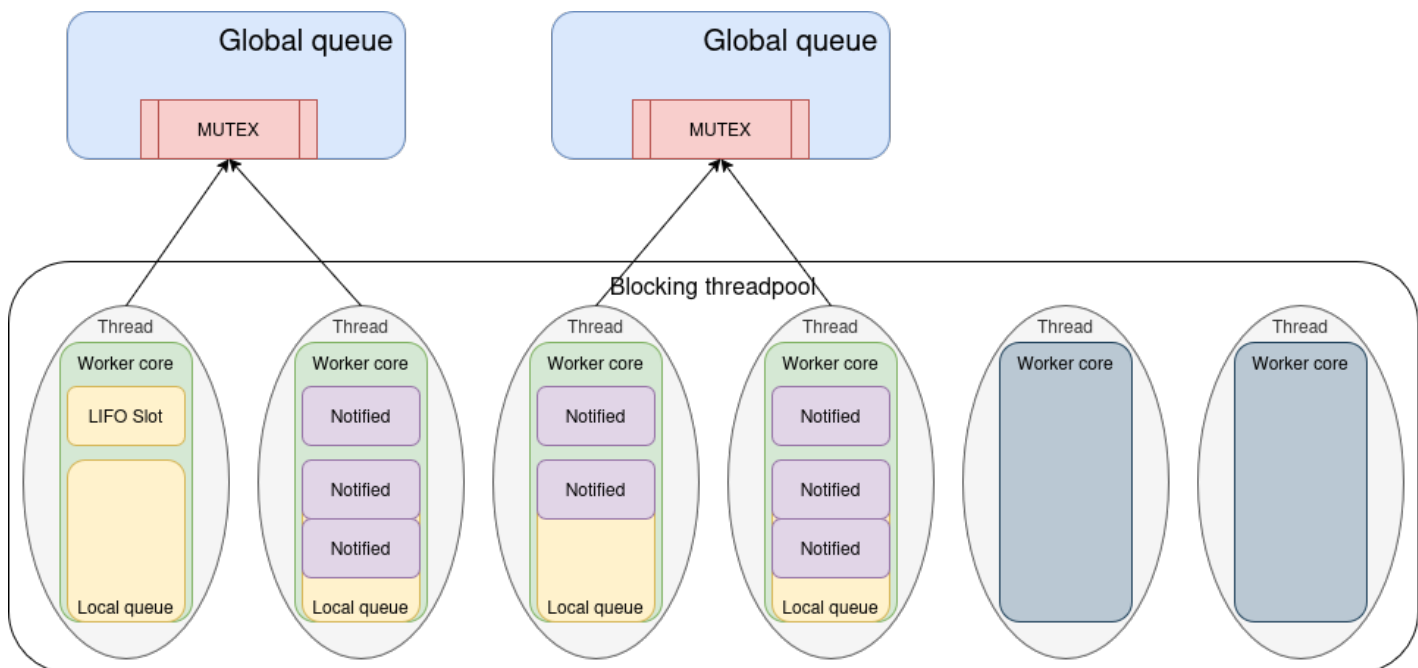


Рис. 4: Упрощенное представление шардированного планировщика в рантайме `tokio`.

Если дублировать весь рантайм нельзя, вероятно, уменьшить ограничение накладываемое взаимодействием исполнителей получится при использовании нескольких изолированных глобальных очередей и исполнителей в одном рантайме, как представлено на изображении 4. Здесь и далее под **рабочей группой** будет пониматься глобальная очередь с фиксированным количеством исполнителей.

Такое шардирование планировщика было реализовано, так как предполагалось, что эта модификация позволит увеличить пропускную способность: каждая рабочая группа изолирована от остальных, то есть исполнители не могут похищать задачи из глобальных или локальных очередей других групп, это вынуждает их останавливать исполнение собственных потоков — все это позволяет избежать конфликтов между исполнителями из различных групп и экономить системные ресурсы при нехватке задач.

При аллокации асинхронного замыкания ранее использовался метод `tokio::spawn`. Теперь при аллокации асинхронного замыкания необходимо выбрать рабочую группу для его выполнения, для чего используется локальное для потока состояние [10]. Для локализации задач акторов, созданных одной задачей производителем интерфейс был дополнен методом `tokio::spawn_into()`, позволяющим пользователю сообщать желаемый номер рабочей группы для исполнения замыкания. Это позволяет пользователю распределять задачи производителя между рабочими группами.

Для подтверждения гипотезы об улучшении пропускной способности с помощью шардирования планировщика необходимо произвести повторные эксперименты и сравнить использование рабочих групп с дублированием всего рантайма.

## 7. Производительность решения

Целями следующих экспериментов является выявление эффекта, произведенного использованием нескольких глобальных очередей, и сопоставлением этих результатов с результатами использования нескольких рантаймов, для чего были поставлены следующие вопросы:

- RQ1.** Будет ли пропускная способность рантайма с одной рабочей группой сопоставима с пропускной способностью исходного рантайма?
- RQ2.** Будет ли наблюдаться увеличение производительности при использовании большего количества рабочих групп?
- RQ3.** Будет ли пропускная способность рантайма с использованием нескольких рабочих групп сопоставима с пропускной способностью системы состоящей из такого же количества рантаймов?
- RQ4.** Будет ли наблюдаться изменение в производительности при изменении количества рабочих групп, если задачи акторы будут тривиальными конечными автоматами?

Эксперимент производился в окружении описанном в разделе 5.1. Метрикой для ответа на вопрос является пропускная способность рантайма или совокупности рантаймов.

### 7.1. Ход исследования

Количество потоков исполнителей и задач производителей было фиксировано и равномерно распределено между рантаймами или рабочими группами. Под количеством дубликатов имеется в виду количество исходных рантаймов или количество рабочих групп в составе одного модифицированного рантайма.

На изображении 5 представлены результаты измерения основного сценария.

На изображении 6 представлены результаты измерения сценария с тривиальными акторами.

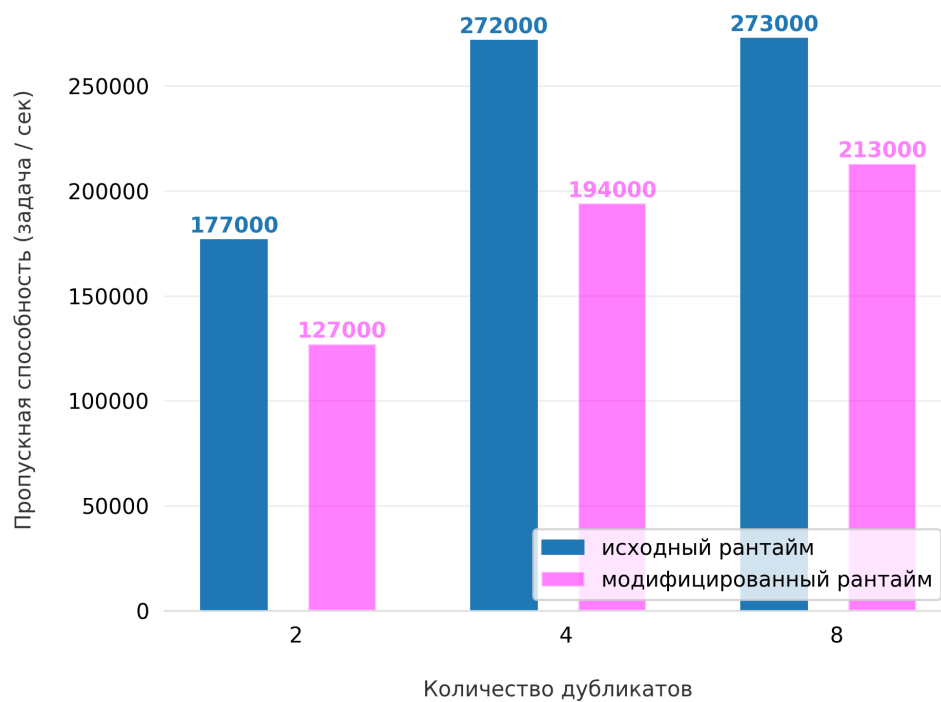


Рис. 5: Сравнение производительности исходного и модифицированного рантаймов в основном сценарии.

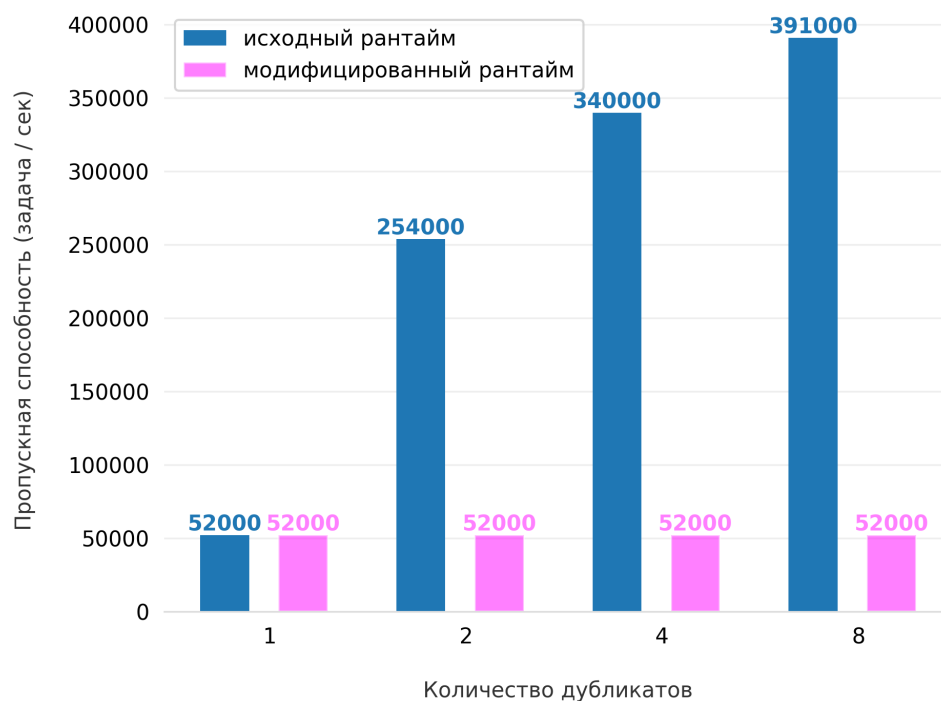


Рис. 6: Сравнение производительности исходного и модифицированного рантаймов в сценарии с тривиальными акторами.

Рис. 5 и Рис. 6 демонстрирует ответы на поставленные выше вопросы 7:

- Пропускная способность системы из одного рантайма и модифицированного рантайма, содержащего одну рабочую группу, демонстрирует одинаковую пропускную способность в обоих случаях.
- При увеличении количества глобальных очередей в системе, состоящей из одного рантайма, наблюдается увеличение пропускной способности в случае нетривиальных акторов. Это ожидаемый результат — меньшее количество исполнителей конкурирует за глобальную и локальные очереди.
- Пропускная способность системы, состоящей из одного модифицированного рантайма, содержащего несколько очередей, меньше пропускной способности системы, состоящей из нескольких рантаймов. Это ожидаемый результат — в системе, состоящей из отдельных рантаймов, требуется меньшее количество синхронизаций:
  - Различные метрики, собираемые рантаймом реализованы с помощью атомарных счетчиков, разделяемых в случае одного рантайма.
  - Многие структуры данных находятся близко в памяти, что может так же требовать дополнительной синхронизации между различными потоками.
- В случае использования тривиальных акторов не наблюдается изменение пропускной способности при изменении количества групп в модифицированном рантайме, однако наблюдается при дубликации экземпляров рантайма. Это ожидаемый результат — замыкания акторы создают большое количество единиц планирования, тем самым активно эксплуатируют планировщик, напротив, в процессе обработки тривиальных замыканий аллокация, регистрация в `OwnedQueue` и деаллокация составляют большую часть действий.

Дублирование инстансов рантаймов позволяет избежать конкуренции за эти ресурсы, чего нельзя сказать об использовании одного рантайма с шардированным планировщиком, которые почти не участвуют в процессе аллокации асинхронных замыканий.

## 7.2. Выводы

Шардирование планировщика многопоточного рантайма `tokio` позволило увеличить пропускную способность системы, создающей большое количество замыканий акторов, однако при аллокации тривиальных асинхронных задач пропускная способность не изменилась.

## 8. Заключение

В ходе работы были выполнены следующие задачи:

- Получено первое описание реализации tokio с точки зрения асинхронного интерфейса языка Rust. В результате обзора выявлены две возможные уязвимости с точки зрения производительности: взаимодействие исполнителей и реализация глобальной очереди. Первая уязвимость была устранена в результате текущей работы путем шардирования планировщика многопоточного рантайма. Изменение многопоточного рантайма с точки зрения алгоритмов глобальной очереди рассмотрены в ВКР Матвея Калашникова: “Оптимизация системы управления очередью задач в библиотеке Tokio для асинхронного рантайма”.
- Создан проект, содержащий бенчмарки, позволяющий автоматически собирать метрики и визуализировать результаты<sup>9</sup>.
- Произведен эксперимент, основанный на ключевом сценарии использования tokio в TATLIN.BACKUP. Обнаружена возможность увеличить пропускную способность TATLIN.BACKUP, что может свидетельствовать о существовании ограничения, накладываемого глобальной очередью. Однако, такой подход остается небезопасным.
- Предложена и реализована безопасная модификация рантайма, позволяющая использовать несколько глобальных очередей в одном инстансе рантайма<sup>10</sup>.
- Проведены эксперименты на целевом сценарии TATLIN.BACKUP, показавшие увеличение производительности предложенного решения.

---

<sup>9</sup>[Репозиторий](#) проект tokiobench (Дата обращения: 25.5.2025)

<sup>10</sup>[Реализация](#) шардирования многопоточного рантайма. Имя пользователя: IgorErin (Дата обращения: 25.5.2025)

Языковые асинхронные интерфейсы трудны в создании вследствие необходимости наличия высокопроизводительной реализации для различных сред: будь то приложения для встроенных систем [1, 14] или пользовательские программные продукты, способные эффективно исполняться на различных архитектурах и операционных системах [9].

Известно множество исследований, направленных на изучение создания и оптимизации пуллов для исполнения вычислительных задач [8, 15, 11, 20]. Однако автору не удалось найти академические работы, посвященные реализации языковым асинхронных рантаймов. Вероятно, это связано с относительно недавним их появлением в языках с повышенным контролем над ресурсами [9].

Стоит отметить, что асинхронные движки таких популярных языков, как Go, Kotlin, tokio в экосистеме языка Rust представляют собой вариации дизайна, предложенного Дмитрием Вьюковым для языка Go [19, 3]. Данная работа показывает, что незначительные изменения в этом популярном подходе могут увеличить пропускную способность отдельных бенчмарков на порядок, что может свидетельствовать о необходимости дальнейших исследований использования современных асинхронных интерфейсов [18] и подходов к многопоточности [4].

## Список литературы

- [1] C++20 Coroutines on Microcontrollers -What We Learned / Bruce Belson, Wei Xiang, Jason Holdsworth, Bronson Philippa // [IEEE Embedded Systems Letters](#). — 2020. — 02. — Vol. PP. — P. 1–1.
- [2] Dandamudi Rohit, Adaji Ifeoma, Rodríguez-Pérez Gema. Reflection on Code Contributor Demographics and Collaboration Patterns in the Rust Communit. — 2025. — 03.
- [3] Deshpande Neil Ashish, Weiss Nathaniel. Analysis of the Go runtime scheduler. — 2012. — URL: <https://api.semanticscholar.org/CorpusID:6999621>.
- [4] Enberg Pekka, Rao Ashwin, Tarkoma Sasu. [The Impact of Thread-Per-Core Architecture on Application Tail Latency](#). — 2019. — 09. — P. 1–8.
- [5] Goncharenko Evgeniy, Paznikov Alexey, Tabakov Andrey. Evaluating the performance of atomic operations on modern multicore systems // [Journal of Physics: Conference Series](#). — 2019. — 12. — Vol. 1399. — P. 033107.
- [6] Khan Koffka. Lecture notes on queues in Computer Science. — 2025. — 01.
- [7] Lemon Jonathan. Kqueue: A generic and scalable event notification facility. — 2002. — 04.
- [8] Ling Yibei, Mullen Tracy, Lin Xiaola. Analysis of Optimal Thread Pool Size. // [Operating Systems Review](#). — 2000. — 04. — Vol. 34. — P. 42–55.
- [9] Lupşa Radu, Lupşa Dana. Coroutines Communications. Design and Implementation Issues in C++20 // [Studia Universitatis Babeş-Bolyai Informatica](#). — 2023. — 06. — Vol. 67. — P. 37–48.

- [10] Marsaglia George. Xorshift RNGs // [Journal of Statistical Software](#). — 2003. — 01. — Vol. 08.
- [11] Mizuno Masaaki, Chen Liubo, Wallentine Virgil. Synchronization in a Thread-Pool Model and its Application in Parallel Computing. — 2003. — 01. — P. 1879–1885.
- [12] Sal Moslehian Arash. [An Experimental Integration of io\\_uring and Tokio: An Asynchronous Runtime for Rust](#) : Ph.D. thesis / Arash Sal Moslehian. — 2022. — 09.
- [13] Solat Siamak. Sharding Distributed Databases: A Critical Review. — 2024. — 04.
- [14] A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems / Bruce Belson, Jason Holdsworth, Wei Xiang, Bronson Philippa // [ACM Transactions on Embedded Computing Systems](#). — 2019. — 06. — Vol. 18. — P. 1–21.
- [15] Syer Mark, Adams Bram, Hassan Ahmed E. [Identifying performance deviations in thread pools](#). — 2011. — 09. — P. 83–92.
- [16] Syme Don, Petricek Tomas, Lomov Dmitry. [The F# Asynchronous Programming Model](#). — Vol. 6539. — 2011. — 01. — P. 175–189.
- [17] Tolaram Nanik. [Epoll Library](#). — 2022. — 12. — P. 223–235. — ISBN: [978-1-4842-8730-9](#).
- [18] [Understanding modern storage APIs: a systematic study of libaio, SPDK, and io\\_uring](#) / Diego Didona, Jonas Pfefferle, Nikolas Ioannou et al. — 2022. — 06. — P. 120–127.
- [19] Vyukov Dmitry. Scalable go scheduler design doc. — 2012. — URL: [https://docs.google.com/document/d/1TTj4T2J042uD5ID9e89oa0sLKhJYD0Y\\_kqxv3I3XMw/edit?tab=t.0](https://docs.google.com/document/d/1TTj4T2J042uD5ID9e89oa0sLKhJYD0Y_kqxv3I3XMw/edit?tab=t.0).

- [20] Wang Liangzhou, Wang Chaobin. Producer-consumer Model Based Thread Pool Design // [Journal of Physics: Conference Series](#). — 2020. — 08. — Vol. 1616. — P. 012073.