

# Оптимизация системы управления очередью задач в Tokio runtime

Калашников М.М., СПбГУ, Санкт-Петербург [kalashnikov.matvey.m@gmail.com](mailto:kalashnikov.matvey.m@gmail.com),  
Гориховский В.И., доцент кафедры системного программирования СПбГУ, к.ф.-м.н.,  
Санкт-Петербург [v.gorikhovskii@spbu.ru](mailto:v.gorikhovskii@spbu.ru)

## Аннотация

Система поддержки периода исполнения Tokio для языка Rust предназначена для эффективного выполнения асинхронных задач в многопоточных приложениях. Однако при высокой нагрузке она сталкивается с ограничением масштабируемости, связанным с конкуренцией при распределении задач между потоками.

В этой работе рассматривается модификация Tokio, включающая дополнительную очередь задач, что позволяет снизить конкуренцию за общие ресурсы планировщика и сократить задержки при распределении задач между потоками.

## Введение

Многопоточное программирование стало неотъемлемой частью разработки высокопроизводительных систем, позволяя выполнять множество операций одновременно, эффективно используя вычислительные ресурсы процессора. Это особенно важно для серверных приложений и систем хранения данных, которые подвергаются высокой нагрузке. Однако при работе с разделяемыми ресурсами, такими как память, файлы или сетевые соединения, возникает необходимость синхронизации доступа. Использование для этих целей блокирующей синхронизации, хотя и упрощает реализацию, может приводить к простоем процессора и снижению производительности.

Для решения этой проблемы применяются асинхронные системы поддержки периода исполнения (runtime), которые позволяют эффективно управлять выполнением задач в многопоточных приложениях, минимизируя задержки, связанные с блокирующей синхронизацией. Они распределяют задачи между потоками и обеспечивают кооперативное переключение между ними, уменьшая накладные расходы на создание новых потоков и синхронизацию доступа к общим ресурсам. В языке Rust наиболее часто используемой системой поддержки периода исполнения является Tokio<sup>1</sup>.

---

<sup>1</sup>Tokio: Asynchronous runtime for Rust — <https://tokio.rs/tokio>.

Основой эффективной работы асинхронной системы поддержки периода исполнения является механизм распределения задач между потоками-работниками. Планировщик в Tokio реализует множество оптимизаций, одной из которых является использование двух типов очередей для хранения задач. Помимо общей глобальной очереди, каждый поток-работник имеет свою локальную очередь для хранения задач<sup>2</sup>. Такая архитектура позволяет снизить конкуренцию за доступ к глобальной очереди и повысить степень параллелизма.

Тем не менее, в определенных сценариях исполнения многопоточных программ под управлением Tokio наблюдается снижение производительности при обработке большого количества коротких задач. Эта проблема была выявлена сотрудниками компании ООО «Ядро Центр Программных Разработок». При этом конкуренция за доступ к глобальной очереди задач становится узким местом, снижая производительность многопоточного приложения. Эта работа направлена на проектирование и внедрение улучшенного алгоритма управления очередями задач в Tokio, направленного на повышение производительности системы поддержки периода исполнения.

## Устройство Tokio

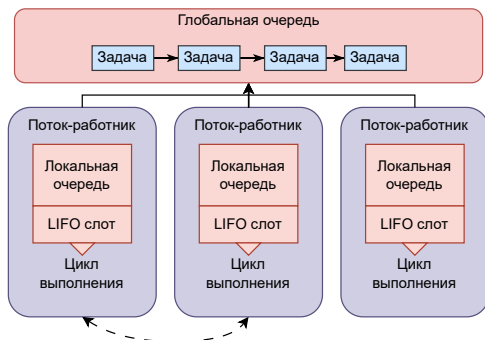


Рис. 1: Архитектура многопоточной системы поддержки периода исполнения

Tokio использует фиксированный пул потоков-работников. Количество потоков в пуле обычно определяется количеством ядер процессора, но может быть настроено пользователем. Планировщик задач Tokio организован

<sup>2</sup>Making the Tokio scheduler 10x faster — <https://tokio.rs/blog/2019-10-scheduler>.

следующим образом:

- каждый поток-работник имеет собственную локальную очередь задач;
- каждый поток-работник может получить блокирующий доступ к глобальной очереди;
- для балансировки нагрузки между потоками используется механизм кражи задач (work stealing).

Потоки-работники сначала обрабатывают задачи из своей локальной очереди (структура LIFO). При ее опустошении поток обращается к глобальной очереди, либо пытается украсть задачи у соседей. Задачи, создаваемые самим потоком-работником в процессе исполнения, сначала помещаются в его локальную очередь, а при ее переполнении вытесняются в глобальную.

Использование блокирующего доступа к глобальной очереди снижает производительность Tokio в сценариях с высокой нагрузкой, особенно при большом количестве короткоживущих задач. В таких условиях потоки пытаются одновременно получить доступ к глобальной очереди, что приводит к увеличению времени ожидания.

## Архитектура решения

Для решения описанной проблемы была предложена модификация системы очередей Tokio, основанная на использовании дополнительной очереди задач. Такой подход позволяет снизить количество обращений к глобальной очереди и уменьшить расходы на синхронизацию. Исходный код модификации Tokio доступен в репозитории<sup>3</sup>.

Дополнительная очередь выполняет роль буфера между потоками-работниками и глобальной очередью. Работа с дополнительной очередью состоит из следующих этапов.

1. **Вытеснение задач из локальной очереди.** Когда локальная очередь потока-работника переполняется, половина задач из нее переносится в глобальную очередь.
2. **Перенос задач в дополнительную очередь.** При достижении определенного порога задач в глобальной очереди группа задач из начала очереди переносится в дополнительную очередь.

---

<sup>3</sup><https://github.com/foreverjun/tokio/pull/2>

3. **Приоритет дополнительной очереди.** При поиске следующей задачи для выполнения поток-работник сначала обращается к дополнительной очереди. Это позволяет снизить нагрузку на глобальную очередь.
4. **Обращение к глобальной очереди.** Если дополнительная очередь пуста, поток-работник обращается к глобальной очереди для получения задач.

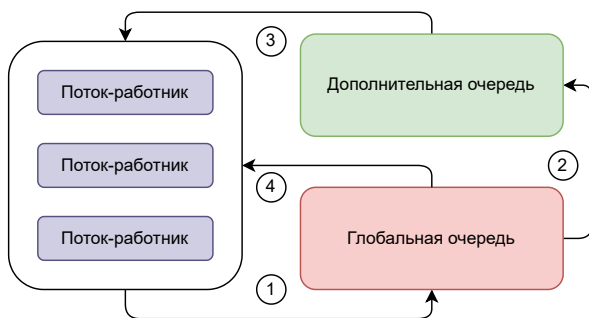


Рис. 2: Схема работы системы с дополнительной очередью

### ***Выбор и реализация дополнительной очереди***

Для реализации дополнительной очереди были рассмотрены структуры данных, использующие атомарные операции. Такой подход эффективнее при высокой конкуренции и снижает расходы на синхронизацию.

- **SegQueue** — очередь, состоящая из блоков, каждый из которых представляет собой массив из 31 элемента<sup>4</sup>. Использует CAS операции. Используется реализация из библиотеки `crossbeam`, активно применяющейся в Rust.
- **FAAArrayQueue** — неблокирующая очередь, состоящая из блоков фиксированного размера, где доступ к ячейкам синхронизируется с помощью атомарной инструкции FAA [1]. Были реализованы две версии с различными механизмами управления памятью: одна с использованием Epoch-Based Reclamation (EBR)<sup>5</sup>, другая с использованием Hazard

<sup>4</sup>SegQueue. An unbounded multi-producer multi-consumer queue — <https://docs.rs/crossbeam/latest/crossbeam/queue/struct.SegQueue.html>.

<sup>5</sup>Crossbeam: Epoch-Based Reclamation — <https://docs.rs/crossbeam-epoch>.

Pointers<sup>6</sup>.

- **BatchQueue** — упрощенная реализация неблокирующей очереди, поддерживающая пакетную обработку задач [2]. Основана на модифицированной версии классической очереди Michael–Scott [3].

## Эксперимент

### Сравнение очередей

Производительность очередей сравнивалась в сценарии «производитель–потребитель». Элементы передавались в очереди пакетами фиксированного размера (64 элемента). Пропускная способность измерялась для разных соотношений числа производителей и потребителей (производитель:потребителей). Эксперимент основан на подходе, описанном в статье [4]. Исходный код очередей и сценария тестирования доступен в репозитории<sup>7</sup>.

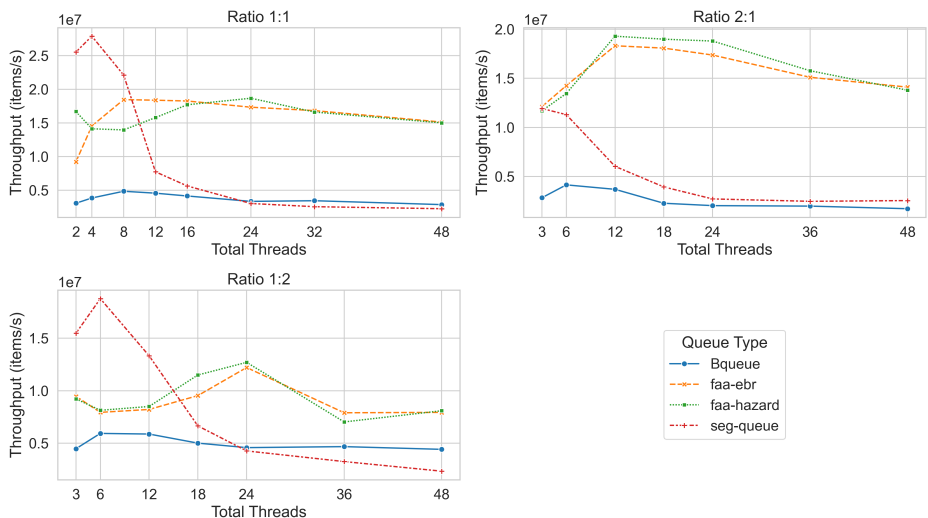


Рис. 3: Сравнение пропускной способности реализованных очередей и SeqQueue. Больше — лучше

<sup>6</sup>Haphazard: Hazard Pointers for Rust — <https://docs.rs/haphazard/0.1.8/haphazard/index.html>.

<sup>7</sup><https://github.com/foreverjun/queue-bench>

По итогам сравнения наилучшую производительность при большом количестве потоков показала `FAAArrayQueue`. Заметим, что использование различных способов освобождения памяти (EBR и Hazard Pointers) практически не влияет на пропускную способность. Для интеграции в `Tokio` в качестве дополнительной очереди была выбрана `FAAArrayQueue` с реализацией на основе EBR, так как библиотека `crossbeam` широко используется в сообществе Rust, благодаря своей активной поддержке, длительной истории разработки и высокой надежности.

### *Сравнение модифицированного и оригинального Tokio*

Для оценки производительности модификации использовался тестовый сценарий, основанный на реализации Игоря Ерина<sup>8</sup>. Его цель — имитировать ситуацию, при которой глобальная очередь заполняется большим количеством коротких процессорозависимых задач. Нагрузка создается задачами-производителями, каждая из которых генерирует по 7000 пустых задач. Эти задачи добавляются в локальные очереди потоков-работников и при переполнении вытесняются в глобальную очередь, моделируя высокую конкуренцию за доступ к ней. Такой подход позволяет нагружать как локальные, так и глобальную очереди.

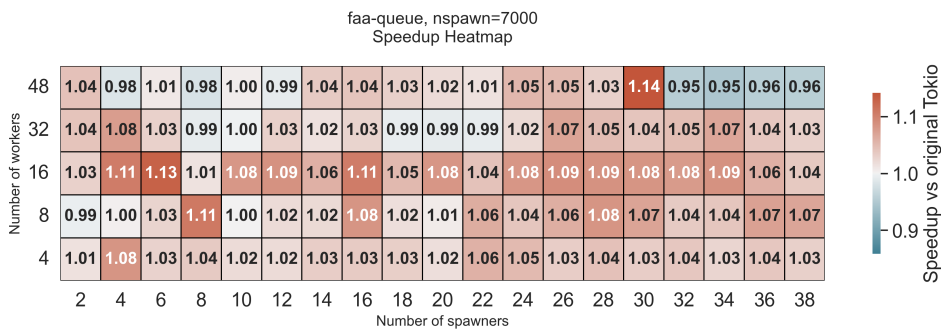


Рис. 4: Пропускная способность Tokio с `FAAArrayQueue`, относительно оригинала

На тепловой карте представлена относительная пропускная способность модифицированного Tokio по сравнению с оригинальной реализацией. Модифицированная версия показывает устойчивое увеличение пропускной способности практически для всех конфигураций, особенно для 16 потоков-работников, где достигается прирост до 13%. Даже в сценариях с высоким

<sup>8</sup>[https://github.com/foreverjun/tokiobench/tree/additional\\_q\\_bench](https://github.com/foreverjun/tokiobench/tree/additional_q_bench)

числом потребителей модифицированная версия не уступает оригинальной в производительности.

Тестирование выполнялось на виртуальной машине Yandex Cloud с процессором Intel Xeon (Ice Lake) с 24 физическими и 48 виртуальными ядрами. Для стабильности приоритет выполнения был повышен до максимального (nice -20).

## Заключение

В работе была спроектирована и реализована модификация системы поддержки периода исполнения Tokio с использованием дополнительной очереди задач. Проведен сравнительный анализ неблокирующих очередей. Лучшая реализация интегрирована в Tokio, в результате чего удалось добиться увеличения пропускной способности системы до 13% для целевых сценариев использования.

## Список литературы

- [1] P. Ramalhete, A. Correia. FAAArrayQueue — MPMC lock-free queue. Concurrency Freaks Blog. <https://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html>.
- [2] Gal Milman-Sela, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2022. BQ: A Lock-Free Queue with Batching. ACM Trans. Parallel Comput. 9, 1, Article 5 (March 2022), 49 pages. <https://doi.org/10.1145/3512757>
- [3] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96). Association for Computing Machinery, New York, NY, USA, 267–275. <https://doi.org/10.1145/248052.248106>
- [4] Raed Romanov and Nikita Koval. 2023. The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23). Association for Computing Machinery, New York, NY, USA, 14–26. <https://doi.org/10.1145/3572848.3577485>