НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
январь–февраль 2026 Том 26 № 1 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
January–February 2026 Vol. 26 No 1 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2026-26-1-125-134
УДК 004.4

Оценка производительности алгоритмов синхронизации в средах исполнения с легкими потоками на языке C++

Тарас Михайлович Скаженик[1]✉, Виталий Евгеньевич Аксенов[2], Антон Александрович Малахов[3], Андрей Васильевич Чурбанов[4]

[1,2] Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация
[3] Университет Неймарк, Нижний Новгород, 603138, Российская Федерация
[4] независимый исследователь, Российская Федерация

[1] taras.skazhenik@yandex.ru✉, https://orcid.org/0009-0002-1959-2010
[2] aksenov.vitaly@gmail.com, https://orcid.org/0000-0001-9134-5490
[3] Anton.A.Malakhov@gmail.com, https://orcid.org/0009-0004-9276-2172
[4] andrey.churbanov@gmail.com, https://orcid.org/0009-0009-3165-4932

Аннотация

Введение. Традиционно, многопоточные структуры данных разрабатывались и тестировались для вызовов из потоков операционной системы. Однако в последнее время в языках программирования появилась альтернатива — легкие потоки (иначе известные, как асинхронные вызовы, и корутины). Такие потоки бывают двух видов со стеком и без него. В настоящей работе рассмотрены стековые корутины. Основное преимущество легких потоков заключается в меньшем количестве накладных расходов. В отличие от потоков операционной системы, в стандартной библиотеке языка C++ отсутствует реализация стековых легких потоков. Они предоставляются несколькими библиотеками, отличающимися: планированием исполнения потока, способом хранения данных в памяти и интерфейсом использования. Кроме того, переиспользование существующего кода невозможно из-за необходимости явной разметки кода точками переключения контекста, отсутствие которых может приводить к взаимной блокировке легких потоков. Для ограничения области исследования рассмотрен конкретный вид многопоточных примитивов — мьютекс (иначе известный, как взаимное исключение). **Метод.** Выполнен анализ нескольких библиотек для работы с легкими потоками: Argobots, Boost Fibers, Userver. На основе оценки их функциональности, разработан трехступенчатый механизм ожидания, использующий активное ожидание, переключение контекста, а также способы приостановки и возобновления легкого потока. Описание механизма включает предложение об общем способе его внедрения в известные алгоритмы мьютексов. Механизм служит дополнительным слоем абстракции между библиотеками легких потоков и мьютексами, позволяя реализовывать их в общем виде. Из-за отсутствия существующих решений, был создан инструмент для тестирования и оценки производительности модифицированных мьютексов. **Основные результаты.** С помощью предложенного метода ряд известных мьютексов был адаптирован для вызовов из легких потоков. Для тестирования получившихся примитивов разработан инструмент, позволяющий интегрировать произвольные реализации легких потоков и учитывающий специфику работы с ними. В результате была измерена пропускная способность адаптированных мьютексов с помощью нового инструмента на специальном сценарии, отражающем особенности кооперативной многозадачности. **Обсуждение.** В перспективе планируется формирование библиотеки, включающей все известные реализации мьютексов, адаптированных под вызовы из легких потоков. Это позволит провести детальное сравнение производительности этих алгоритмов в зависимости от: библиотеки легких потоков, алгоритмов планировщика и архитектуры процессора. Ожидается, что детальный анализ поведения существующих алгоритмов позволит создать новую реализацию мьютекса, работающего наиболее эффективно на вызовах из легких потоков.

Ключевые слова

методы синхронизации, мьютекс, легкие потоки, оценка производительности, NUMA-архитектура, C++

Ссылка для цитирования: Скаженик Т.М., Аксенов В.Е., Малахов А.А., Чурбанов А.В. Оценка производительности алгоритмов синхронизации в средах исполнения с легкими потоками на языке C++ // Научно-технический вестник информационных технологий, механики и оптики. 2026. Т. 26, № 1. С. 125–134. doi: 10.17586/2226-1494-2026-26-1-125-134

Performance evaluation of synchronization algorithms in lightweight thread environments in C++

Taras M. Skazhenik[1]✉, Vitaly E. Aksenov[2], Anton A. Malakhov[3], Andrey V. Churbanov[4]

[1,2] ITMO University, Saint Petersburg, 197101, Russian Federation
[3] Neimark University, Nizhny Novgorod, 603138, Russian Federation
[4] Independent Researcher, Russian Federation

[1] taras.skazhenik@yandex.ru✉, https://orcid.org/0009-0002-1959-2010
[2] aksenov.vitaly@gmail.com, https://orcid.org/0000-0001-9134-5490
[3] Anton.A.Malakhov@gmail.com, https://orcid.org/0009-0004-9276-2172
[4] andrey.churbanov@gmail.com, https://orcid.org/0009-0009-3165-4932

Abstract

Traditionally, multithreaded data structures have been designed and tested for use with Operating System (OS) threads. However, in recent years, programming languages have introduced an alternative — lightweight threads (also known as asynchronous calls or coroutines). These threads can be divided into two types: stackful and stackless; this paper focuses on stackful coroutines. The main advantage of lightweight threads lies in their reduced overhead. Unlike OS threads, the C++ Standard Library lacks a native implementation of stackful lightweight threads. They are instead provided by several third-party libraries which differ in scheduling policies, memory management approaches, and interfaces. Moreover, reusing existing code is often infeasible due to the explicit marking of context-switching points required by coroutines, their absence may lead to deadlocks between lightweight threads. To narrow the research scope, this work focuses on a specific synchronization primitive — the mutex (mutual exclusion). Several lightweight threading libraries were examined, including Argobots, Boost Fibers, and Userver. Based on an analysis of their functionality, a three-phase waiting mechanism was developed that combines active spinning, context switching, and coroutine suspension and resumption techniques. The description of the mechanism includes a general approach for integrating it into existing mutex algorithms. This mechanism acts as an additional abstraction layer between lightweight threading libraries and mutex implementations, enabling a unified design. Due to the absence of existing solutions, a dedicated tool was developed for testing and performance evaluation of the modified mutexes. Using the proposed method, several well-known mutexes were adapted for the usage from lightweight threads. To test these primitives, a tool was created that allows integration with arbitrary coroutine implementations and accounts for their specific execution models. The throughput of the adapted mutexes was measured using this tool under a specialized workload scenario, reflecting the characteristics of cooperative multitasking. In future work, the aim is to develop a comprehensive library encompassing all known mutex implementations adapted for lightweight thread environments. This will enable detailed performance comparisons across coroutine libraries, scheduling algorithms, and processor architectures. It is expected that a thorough analysis of existing algorithms behavior will lead to the development of a new mutex design optimized for lightweight thread execution.

Keywords

synchronization methods, mutex, lightweight threads, performance evaluation, NUMA architecture, C++

For citation: Skazhenik T.M., Aksenov V.E., Malakhov A.A., Churbanov A.V. Performance evaluation of synchronization algorithms in lightweight thread environments in C++. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2026, vol. 26, no. 1, pp. 125–134 (in Russian). doi: 10.17586/2226-1494-2026-26-1-125-134

Введение

Начало развития вычислительных устройств характеризовалось постоянным ростом мощности одного вычислительного ядра на процессоре [1]. Но с течением времени из-за физических ограничений вектор развития чипов сместился в сторону горизонтального масштабирования [2]. По этой причине современные решения используют многоядерную и многопроцессорную [3] структуры. Наряду с этим, поиск способов более эффективного использования аппаратных ресурсов привел к появлению процессоров с неравномерным доступом к памяти (Non-Uniform Memory Access, NUMA) [4]. Но для оптимального использования такой архитектуры требуется написание программ, способных выполнять свои задачи распределенно по нескольким ядрам или процессорам.

Традиционно, программы, распределяющие свою работу по нескольким вычислительным узлам, создавались с использованием потоков операционной системы (ОС), также называемых платформенными. Такие потоки обеспечивают изоляцию и параллелизм, однако требуют значительных затрат на переключение контекста, синхронизацию и взаимодействие с планировщиком ОС. Для снижения этих накладных расходов были введены легкие потоки, реализуемые на уровне пользовательского пространства. В то время как ряд языков программирования, таких как Java и Golang, включает легкие потоки в свои стандарты, в C++ такой стандарт отсутствует. Вместо этого существует набор библиотек, представляющих различные интерфейсы и реализации. В представленной работе рассмотрены три популярные реализации легких потоков: Argobots [5], Boost Fibers[1] и Userver[2]. Argobots выделяется как самый функциональный представитель академических исследований [6], Boost Fibers предоставляется как часть одной из наиболее известных библиотек с открытым исходным кодом для C++, а Userver, является разработ-

[1] [Электронный ресурс]. https://www.boost.org/doc/libs/1_85_0/libs/fiber/doc/html/index.html (дата обращения: 24.04.2025).

[2] [Электронный ресурс]. https://userver.tech/ (дата обращения: 24.04.2025).

кой компании Yandex и выбран по причине активного использования в отечественных компаниях.

Одной из задач многопоточного программирования является обеспечение взаимного исключения при доступе к критическим секциям кода. Решением задачи служит использование разделяемого объекта, находящегося в одном из двух состояний: заблокированном или разблокированном [7]. Такой объект принято называть мьютексом (mutex — взаимное исключение). В научных работах представлено множество реализаций таких примитивов синхронизации. Самые простые исследования используют конструкции из одной или нескольких атомарных переменных, такие как TTAS (Test-Test-And-Set) [8] или Ticket Lock [9]. Более сложные применяют многопоточную очередь для запросов, например, MCS (Mellor-Crummey and Scott) [9] или CLH (Craig-Landin-Hagersten) [10]. Последние реализации примитивов синхронизации таких как, HMCS (Hierarchical MCS) [11] или CNA (Compact NUMA-Aware) [12], используют сложную иерархическую структуру для эффективной работы на процессорах с NUMA-архитектурой [4]. В большинстве работ эксперименты проводились на процессорах с архитектурой x86 [13], но в некоторых [14] присутствуют эксперименты на процессорах с архитектурой ARM (Advanced Reduced instruction set computer Machine) [15], набирающей популярность в последнее время. Все перечисленные реализации создавались для вызовов из потоков ОС. Исходя из этого, целью настоящей работы является адаптация мьютексов к вызовам из легких потоков, а также сравнение их производительности, поскольку не было найдено аналогичных работ в этом направлении.

Для достижения поставленной цели нужно решить следующие задачи: предложить обобщенный способ адаптации примитивов синхронизации к работе с разными реализациями легких потоков; определить способ и инструмент для сравнения реализуемых алгоритмов; проанализировать процесс исполнения тестового сценария для выявления недостатков в работе примитива синхронизации. Использование существующих подходов и инструментов для решения перечисленных задач невозможно из-за отсутствия единой реализации легких потоков в языке C++. Можно предположить, что такой реализации в ближайшее время не появится, пока легкие потоки не будут включены в стандартную библиотеку языка. Для стимулирования данного процесса в настоящей работе представлена удобная система тестирования легких потоков.

Обзор реализаций легких потоков на языке C++

Легкие потоки представляют альтернативу потокам ОС и были разработаны для решения проблем, связанных с высокой стоимостью создания и переключения потоков ОС. В отличие от потоков ОС, которые требуют значительных ресурсов и взаимодействия с ядром, легкие потоки управляются в пользовательском пространстве, что позволяет создавать и использовать их с минимальными затратами. Это делает их особенно эффективными в приложениях с высокой степенью многопоточности.

Общий принцип работы. Главное отличие между легкими и платформенными потоками заключается в способе их управления. Потоки ОС управляются ядром и могут быть прерваны в любой момент времени, иначе говоря, они работают по принципу вытесняющей многозадачности. Легкие потоки, напротив, работают в пространстве пользователя и обязаны передать управление планировщику самостоятельно. Такой подход называется кооперативной многозадачностью [7, 16]. Схема работы легких потоков выглядит следующим образом [17]. Пользователь создает несколько потоков ОС, а затем исполняет легкие потоки на них. Одновременно на платформенном потоке может исполняться не более одного легкого. По этой причине, если код легкого потока делает блокирующий вызов или выполняет активное ожидание, то блокируется исполняющий поток ОС, и другие легкие потоки исполняться на нем не могут. Из этого следует, что существующие мьютексы не подходят для легких потоков, потому что они используют блокирующие вызовы и циклы активного ожидания, не передавая управление другим легким потокам. При этом библиотеки предоставляют свои собственные примитивы и абстракции для борьбы с перечисленными проблемами, но интерфейсы взаимодействия с потоками могут отличаться, что делает невозможным мгновенный перенос реализации конкурентной структуры данных для одной библиотеки на другую. Рассмотрим особенности каждой из трех выбранных библиотек.

Рассматриваемые библиотеки. Argobots — система исполнения, предлагающая наиболее полный [6], но в то же время низкоуровневый набор средств для управления жизненным циклом легких потоков. Библиотека предоставляет разделение на потоки исполнения, являющиеся обертками над платформенными, и легкие потоки. Пользователь должен определять точное число потоков обоих видов и явно обеспечивать размещение в памяти в течение всей жизни программы. Кроме того, помимо выбора из предложенных алгоритмов планировщика, присутствует возможность написания собственного алгоритма. Непосредственно из механизмов, отвечающих за управление потоком, необходимо выделить два: команда *yield*, возвращающая управление потоком обратно планировщику, и пару команд *suspend-resume*, переводящую легкий поток в приостановленное состояние (запрещающее потоку исполняться) и обратно. При этом пользователь должен следить за порядком вызовов приостановки и возобновления потока: вызов *resume* до *suspend* приведет к засыпанию, тогда как, например, на языке Java в аналогичном исполнении *suspend* проигнорируется.

Boost Fibers — библиотека для работы с легкими потоками, схожая с интерфейсом *std::thread* стандартной библиотеки языка C++ для работы с потоками ОС. Для использования надо создать необходимое число потоков ОС и назначить им алгоритм планирования. Стоит отметить, что из-за особенностей реализации, подразумевающей хранение состояния в глобальных переменных, потоки ОС должны быть зарегистрированы один раз на весь жизненный цикл программы. В отличие от Argobots, в Boost Fibers можно вернуть

управление обратно планировщику через команду *yield*, а механизм явной приостановки потока отсутствует, но им можно воспользоваться через такие абстракции, как *promise* или *conditional_variable*. В отличие от других библиотек, она предоставляет планировщик, учитывающий системы с NUMA-архитектурой.

Userver — фреймворк для создания высоконагруженных приложений, содержащий в своей основе набор абстракций для написания асинхронного кода. Ядро фреймворка построено на библиотеке Boost::Coroutine2[1] и предоставляет схожий с Boost Fibers функционал. Однако набор настроек у Userver более ограничен по сравнению с Boost Fibers, так как основным сценарием использования является построение микросервисов, а не изолированное использование в качестве платформы легких потоков.

В результате для дальнейших реализаций мьютексов будем требовать только функцию *yield* и пару функций *suspend-resume*. При этом функции *suspend-resume* могут быть пустыми. В итоге получим простой интерфейс, который надо реализовать для подключения библиотеки.

Проблема использования существующих решений в средах с легкими потоками

Для демонстрации проблемы переиспользования существующих решений рассмотрим цикл ожидания у мьютекса TTAS [8], обладающего наиболее простой структурой. Для захвата блокировки в цикле совершается попытка изменения атомарной переменной с 0 на 1. В случае неудачного изменения алгоритм может сразу повторить попытку или выполнить какие-то действия для снижения нагрузки на атомарную переменную. В случае с вытесняющей многозадачностью алгоритм не может повлиять на исполнение из потоков ОС. Это связано с тем, что алгоритм определяется планировщиком ОС, но ожидается, что планировщик со временем отдаст потоку исполнение и в работе программы будет прогресс, при этом поток закончит исполнение критической секции. Но использование того же самого кода в среде с легкими потоками потенциально ведет к взаимной блокировке. Обратимся к примеру возможного исполнения. Допустим, два легких потока исполняются на одном платформенном. Первый поток захватывает блокировку, и внутри критической секции отдает управление планировщику. Второй поток пытается захватить блокировку, но терпит неудачу, поскольку блокировкой владеет первый поток. Однако передача управления обратно первому легкому потоку не происходит, поскольку у мьютекса все циклы ожидания активные. В результате происходит взаимная блокировка потоков.

Рассмотрим причину, по которой отсутствуют аналогичные исследования. Интеграция библиотеки для работы с легкими потоками в любой существующий проект требует явной разметки кода вызовами переключения контекста. В связи с этим существует иной подход, а именно симуляция вытесняющей многозадачности с помощью различных техник [18, 19]. В работе [19] представлены две таких техники с минимальными накладными расходами и проведена их оценка на популярных системах. Отметим, что авторы [19] признают, что прямое использование легких потоков обеспечивают наивысшую эффективность, в случае переписывания больших проектов, что редко осуществимо на практике. Трудности с применением легких потоков схожи и среди других языков программирования. Например, в Golang сделали горутины асинхронно вытесняемыми в версии 1.14[2], чтобы избежать взаимной блокировки потоков [20]. Аналогичным образом, в Java сталкивались с проблемами прикрепления виртуальных потоков к потокам ОС в *synhronized* методах до версии 24[3].

Метод адаптации мьютексов к работе с легкими потоками

Рассмотрим способ решения проблемы блокировки потоков, который дополнительно позволяет уйти от специфики работы конкретной реализации легких потоков.

Определим интерфейс мьютекса. Мьютекс предоставляет два метода: *lock* и *unlock*. Каждый из методов принимает один параметр — некоторый контекст, который может быть пуст или содержать персональный объект блокировки для легкого потока. Гарантируется, что успешный вызов метода *lock* может быть выполнен единовременно только одним потоком, в то время как прочие потоки будут заблокированы до момента вызова *unlock*. Кроме того, любой мьютекс принимает шаблонный параметр *BackoffType*, содержащий в себе реализацию механизма для корректного выполнения циклов с активным ожиданием. Этот параметр позволяет разрабатывать мьютексы, независимые от реализации легких потоков. Для подключения мьютекса достаточно реализовать интерфейс *BackoffType*, состоящий из способа передачи управления планировщику (*yield*), механизма приостановки и возобновления потока (*suspend* и *resume*) и алгоритма, связывающего эти методы в единую последовательность. При этом обязательной для корректной работы является реализация *yield*. Изучим составные части данного интерфейса и их роль в корректной и эффективной адаптации мьютексов к работе с легкими потоками.

Для разрешения проблемы взаимной блокировки легких потоков активные циклы ожидания, в которых происходит синхронизация с другими потоками, должны быть обеспечены механизмом передачи управления планировщику. В качестве наиболее простого подхода можно передавать управление при каждом неудачном шаге цикла. Но такой метод может привести к высокой частоте переключений, которая вносит накладные расходы. Предложим более оптимальный способ. Будем отдавать управление планировщику только через

[1] [Электронный ресурс]. https://github.com/boostorg/coroutine2 (дата обращения: 25.06.2025).

[2] [Электронный ресурс]. https://go.dev/doc/go1.14 (дата обращения: 11.10.2025).

[3] [Электронный ресурс]. https://openjdk.org/jeps/491 (дата обращения: 11.10.2025).

фиксированное число неудачных вызовов в цикле, а между неудачными вызовами цикла будем исполнять экспоненциально возрастающее число пустых процессорных инструкций. Тем самым получим адаптированную технику экспоненциального откладывания. Если синхронизация между потоками произойдет быстро, то переключение вызвано не будет, в противном случае, будет обеспечен прогресс в системе.

Помимо использования команды *yield*, существует второй подход передачи управления — приостановка потока функцией *suspend*. Как итог, легкий поток переходит в состояние бездействия и не может быть запланирован для исполнения планировщиком. Операция приостановки потока используется в мьютексах на основе очереди. В таком алгоритме поток встает в очередь, ждет фиксированное непродолжительное время и приостанавливается. Затем, когда очередь получения блокировки доходит до приостановленного потока, его пробуждают и он берет блокировку.

В отличие от таких языков как Java, где корректная очередность применения приостановки и возобновления работы потока при любой фактической очередности их вызова гарантируется стандартом, в рассматриваемых библиотеках на C++ нет такой гарантии. Решим эту проблему, модифицировав процесс передачи блокировки. В базовой реализации владелец меняет значение атомарного флага в следующем за собой узле в очереди, а другой поток свой флаг все время проверяет. Трансформируем флаг в атомарную переменную, содержащую значение из набора состояний. Теперь приостанавливающийся поток сначала изменяет состояние этой переменной и убеждается, что блокировка ему еще не была передана. А владелец блокировки сначала проверяет, что следующий в очереди поток находится в активном состоянии и лишь потом передает владение блокировкой. Если следующий поток начал процедуру приостановки, то владелец будет обязан возобновить его работу и потом передать владение блокировкой. Таким образом, исключается вероятность оставления какого-либо потока в приостановленном состоянии навечно. Стоит отметить, что такой подход можно внедрить в любом примитиве, где поток ожидает передачу блокировки на своей локальной переменной (структура очереди здесь необязательна).

Описанные варианты передачи управления могут быть совмещены в едином механизме ожидания. Рассмотрим выбор стратегии в зависимости от длительности ожидания. Если блокировка будет передана в короткий срок, то наиболее эффективно активное ожидание. Для среднего ожидания важна передача управления планировщику для экономии процессорного времени. Наконец, при длительном ожидании, например, если поток занимает одно из последних мест в очереди, то переключение контекста будет происходить множество раз до получения блокировки и эффективнее приостановить его. Выбор из перечисленных вариантов строится на основе счетчика неуспешных проверок условия в цикле: чем выше это число, тем более тяжеловесные методы ожидания будут использоваться. Этот подход был обобщен в виде интерфейса *BackoffType*. Алгоритмическая часть вынесена на уровень библиотеки, а методы *yield* и *suspend* должны быть реализованы через вызовы интерфейсов конкретной библиотеки легких потоков.

С техническими деталями реализации описанного метода, а также с набором инструментов для запуска ряда тестовых сценариев, можно ознакомиться в публичном репозитории nasl-tools[1].

Метод тестирования производительности

Бенчмарк. Для оценки производительности алгоритмов принято выбирать общеиспользуемые инструменты. Но известные фреймворки, такие, как Google Benchmark[2] или Catch2[3], создавались с неявным предположением, что код будет запускаться на потоках ОС. Также инструменты подразумевают небольшое число таких потоков, в то время как легких может быть на порядок больше. Взаимодействие с потоками ОС не предполагает какого-то влияния на алгоритмы планирования, поэтому настройка этого аспекта не рассматривается на уровне архитектуры, единственным выходом является создание своего собственного инструмента.

Инструмент обязан удовлетворять следующим требованиям. Основная логика работы инструмента не должна зависеть от библиотеки легких потоков. Добавление новой библиотеки должно быть возможно в виде отдельного проекта Cross-Platform Make, зависящего от базового проекта. Расширение должно требовать переопределения только функционала, специфичного для библиотеки.

Несмотря на большое число технических деталей, концепцию можно обобщить до двух сущностей. Benchmark — класс-обертка над функцией, которая принимает объект состояния (State). Состояние включает в себя все вычисляемые результаты и метрики, а также владеет общим таймером. В таком случае запуск любого синтетического сценария может быть представлен в виде следующей последовательности шагов:

1) создание объекта Benchmark с функцией, инициализирующей легкие потоки и требуемый мьютекс, а также вызывающей выбранный сценарий;
2) тестирование переданного мьютекса в единой реализации сценария и заполнение результатов в объект State.

Методика тестирования. Для тестирования был выбран стандартный сценарий для потоков ОС, представленный на листинге 1. В течение ограниченного промежутка времени, на каждом шаге цикла, поток пытается захватить блокировку, при получении блокировки выполняет код критической секции, отпускает блокировку, а затем совершает действия, не требующие синхронизации, и в конце передает управление планировщику. Для выделения особенностей поведения и сравнения различных реализаций мьютексов, модифи-

[1] [Электронный ресурс]. https://github.com/tarasska/nasl-tools (дата обращения: 11.10.2025).

[2] [Электронный ресурс]. https://github.com/google/benchmark (дата обращения: 24.04.2025).

[3] [Электронный ресурс]. https://github.com/catchorg/Catch2 (дата обращение: 24.04.2025).

цируются функции критической секции *critical_section* и параллельной работы *parallel_work*, которые и являются основными параметрами сценария.

Листинг 1. Общий паттерн сценариев для теста мьютекса

```
Listing 1. General pattern of mutex tests
while (start_time + test_time < now())
{
   LOCK(mutex)
   critical_section()
   UNLOCK(mutex)
   parallel_work()
   yield()
}
```

Функции *critical_section* и *parallel_work* скрывают в себе произвольные действия, которые тестируются в сочетании с мьютексом. Для эмуляции реалистичности, в них может находиться работа с большим объемом памяти, чтение и обновление коллекций или перемножение матриц. Явный возврат управления в конце цикла служит для более равномерного распределения времени исполнения между легкими потоками.

В представленной работе выделяются две важные метрики, которые обычно принято замерять: пропускная способность (throughput) и время ожидания блокировки (latency). Пропускная способность вычисляется как отношение числа успешно захваченных блокировок ко времени работы всего теста. Число захватов учитывается в каждом потоке отдельно, затем суммируется. Для учета времени работы используется общий таймер, который всегда включается и выключается потоком с меньшим номером. Во избежание сильной рассинхронизации потоков, перед циклом и после него размещается барьер, адаптированный для легких потоков. Таким образом, начало теста едино для всех, а завершение фиксируется по последнему пришедшему потоку. Для вычисления времени ожидания блокировки замер необходимо производить до вызова функции *LOCK* и сразу после нее, разность двух величин складывать в массив, а после завершения теста вычислять набор квантилей, например, 0,9; 0,95 и 0,99. Отметим, что данный подход имеет проблемы в системах с легкими потоками.

Особенности учета времени. Изначально в качестве таймера использовался класс стандартной библиотеки *std::chrono::high_resolution_clock*. Однако полученные замеры показывали временные аномалии из-за его высоких накладных расходов. Поэтому было принято решение измерять время с помощью тиков процессора посредством выполнения легковесных ассемблерных инструкций. На листинге 2 представлены версии таких инструкций для x86 и ARM.

Листинг 2. Получение тиков процессора из регистров процессора

```
Lising 2. Getting CPU ticks from CPU registers
asm volatile(
    "rdtsc\n\tshl $32, %%rdx\n\tor %%rdx, %%rax":"=a"(ticks)::"rdx"
); // x86
asm volatile("mrs %0, cntvct_el0":"=r"(ticks)::"memory"); // ARM
```

В результате вычисление времени ожидания блокировки оказалось нетривиально, поэтому рассмотрим его более детально. Во-первых, засеченный интервал может быть настолько мал, что лишняя процессорная инструкция между таймером и вызовом захвата блокировки будет оказывать влияние на результат. В этой связи требуется использовать барьеры для компилятора при необходимости (volatile на листинге 2). Во-вторых, ассемблерные вставки полагаются на регистры, отдельные для каждого ядра. Так как во время ожидания поток может отдать управление планировщику, между начальным и конечным временными замерами легкий поток может переместиться между платформенным потоками, размещенными на разных ядрах. Поскольку таймеры на ядрах не синхронизируются в процессе работы на большинстве NUMA-устройств, будет получена некорректная разница. В итоге вместо неточных вычислений и высоких накладных расходов на подсчет latency предлагаются иные метрики, не привязанные к малым временным интервалам. Например, для оценки честности мьютекса возможно использовать число потоков, которые владели блокировкой между двумя захватами блокировки конкретным потокам. Для получения такого значения достаточно завести инкрементальный счетчик в критической секции. Затем для каждого потока вычислять разницу между текущим значением и значением, полученным при прошлом захвате блокировки. Разница показывает, сколько конкурирующих потоков смогло получить блокировку раньше, чем рассматриваемый поток, тем самым можно сделать вывод о честности блокировки.

Обнаружение ошибок и неэффективных действий. Для обнаружения ошибок в работе программы принято использовать средства отладки. Они бывают в виде инструментов пошагового исполнения, которое невозможно на высоких уровнях оптимизации компилятора, или с помощью санитайзеров, которые нацелены на поиск ошибок при работе с памятью и на обнаружение гонок данных между платформенными потоками. Использование легких потоков приносит новый вид проблем, связанных с бесконтрольной миграцией между потоками ОС. Такая миграция, с одной стороны, может негативно влиять на эвристики, основанные на работе с локальной памятью, а с другой стороны приводить к некорректному состоянию алгоритма. При этом точки миграции легких потоков в сценарии всего две: вызов команды *yield* и приостановка потока *suspend*. Следовательно, достаточно добавить к указанным вызовам отладочное событие. В связи с тем, что делается акцент на миграции легких потоков, необходимо сохранять события с меткой потока ОС, но ее вычисление требует накладных расходов, а при этом порядок создаваемых событий никак не учитывается. Для решения этой проблемы подходят локальные переменные платформенного потока. Создадим для каждого

потока ОС массив событий и поместим в него все указанные события. Таким образом, в конце тестирования для каждого платформенного потока соберется хронологическая последовательность отработавших легких потоков. Их перемещения можно отследить по времени событий или через явное сохранение переходов в виде пар номеров старого и нового потоков ОС. Но даже в таком наглядном подходе есть проблема: локальные переменные потока ОС некорректно работают в средах с легкими потоками, так как компилятор кэширует обращение к таким переменным в рамках функции и не может отследить момент смены платформенного потока. Легкий поток мигрирует, но продолжает обращаться к предыдущему значению. Обойти такое препятствие возможно, обманув компилятор и спрятав адрес переменной за абстракцию, не кэшируемую по стандарту. Используя представленную методику, появляется возможность строить граф исполнения, считать частоту миграций, определять локализацию потоков по отношению к данным и ближайшим сегментам памяти. Для визуального отображения подходит инструмент Google Trace Viewer[1], а его формат событий крайне прост в создании.

Экспериментальная оценка производительности мьютексов

Для демонстрации работоспособности метода адаптации мьютексов и возможностей разработанного инструмента тестирования был выбран произвольный сценарий, а затем проведено тестирование нескольких мьютексов на одном устройстве, но с различными реализациями легких потоков. В конце были сравнены пропускные способности мьютексов.

Техническая конфигурация. Для наглядности рассмотрим по одному мьютексу из каждой категории: простой мьютекс TTAS [6], мьютекс MCS [7] на основе очереди и иерархический мьютекс HMCS [9]. Также добавим стандартный мьютекс, предоставленный соответствующей библиотекой легких потоков.

Для теста был выбран сценарий, эмулирующий кооперативное взаимодействие легких потоков. Параллельная секция содержит цикл из 10 шагов с 1000 пустых процессорных инструкций и вызовом *yield* на каждом шаге. В критической секции создаются 12 потоков, симулирующие распараллеливание цикла обработки, каждый такой поток исполняет 1000 пустых инструкций. Эффективная реализация мьютекса для этого сценария должна заставлять ожидающие получения блокировки потоки быстро освобождать процессорные ресурсы для исполнения легких потоков в критической секции. Чем успешнее передача управления, тем выше пропускная способность.

Методика замера включает в себя последовательный запуск сценария 50 раз для каждого из представленных рисунков. Каждый запуск исполняет описанный сценарий для одного мьютекса на фиксированном числе легких потоков при фиксированном числе платформенных. Длительность одного запуска составляет до 10 с. Перед основной фазой замеров присутствует этап разогрева, включающий аналогичные операции и длящийся около одной трети от времени основной части теста.

Для экспериментов использован сервер с двумя процессорами Intel Xeon Gold 5220R CLX. Каждый из них имеет 24 физических ядра и с учетом технологии гиперпоточности (Hyper-Threading) число виртуальных ядер достигает 96. Число потоков ОС в замерах равно 96, по одному потоку на каждое виртуальное ядро.

Результаты. Рассмотрим полученные результаты эксперимента (рисунок), где по оси *X* расположено число легких потоков, а по оси *Y* пропускная способность, представленная в числе операций в миллисекунду (оп/мс). Несмотря на полную эквивалентность производимых замеров, видно, что результаты значительно зависят от реализации легких потоков.

Из рисунка, *a* видно, что наиболее простой примитив TTAS демонстрирует наибольшую пропускную способность. Далее располагаются мьютексы на основе MCS, причем при использовании потоков меньше числа платформенных потоков, результат остается сопоставимым с TTAS. Мьютекс из библиотеки Argobots работает хуже, несмотря на его сходство с алгоритмом TTAS. Предполагается, что он неэффективно выбирает момент для приостановки потока.

TTAS на рисунке, *b* также показывает лучший результат, но для остальных мьютексов результат отличается. Для объяснения низкой пропускной способности MCS и HMCS требуется дополнительное исследование, но основной гипотезой является неоптимальный выбор момента для приостановки потока во время ожидания блокировки.

На рисунке, *c* взаимное расположение мьютексов существенно изменилось. Мьютекс TTAS, показывает худший результат, что подчеркивает отсутствие явного паттерна пропускной способности одних и тех же алгоритмов для различных реализаций легких потоков. При этом показатели производительности HMCS уменьшаются на малом числе потоков и увеличиваются при появлении второго NUMA-узла, начиная с отметки в 24 потока.

[1] [Электронный ресурс]. https://chromium.googlesource.com/catapult (дата обращение: 24.05.2025).

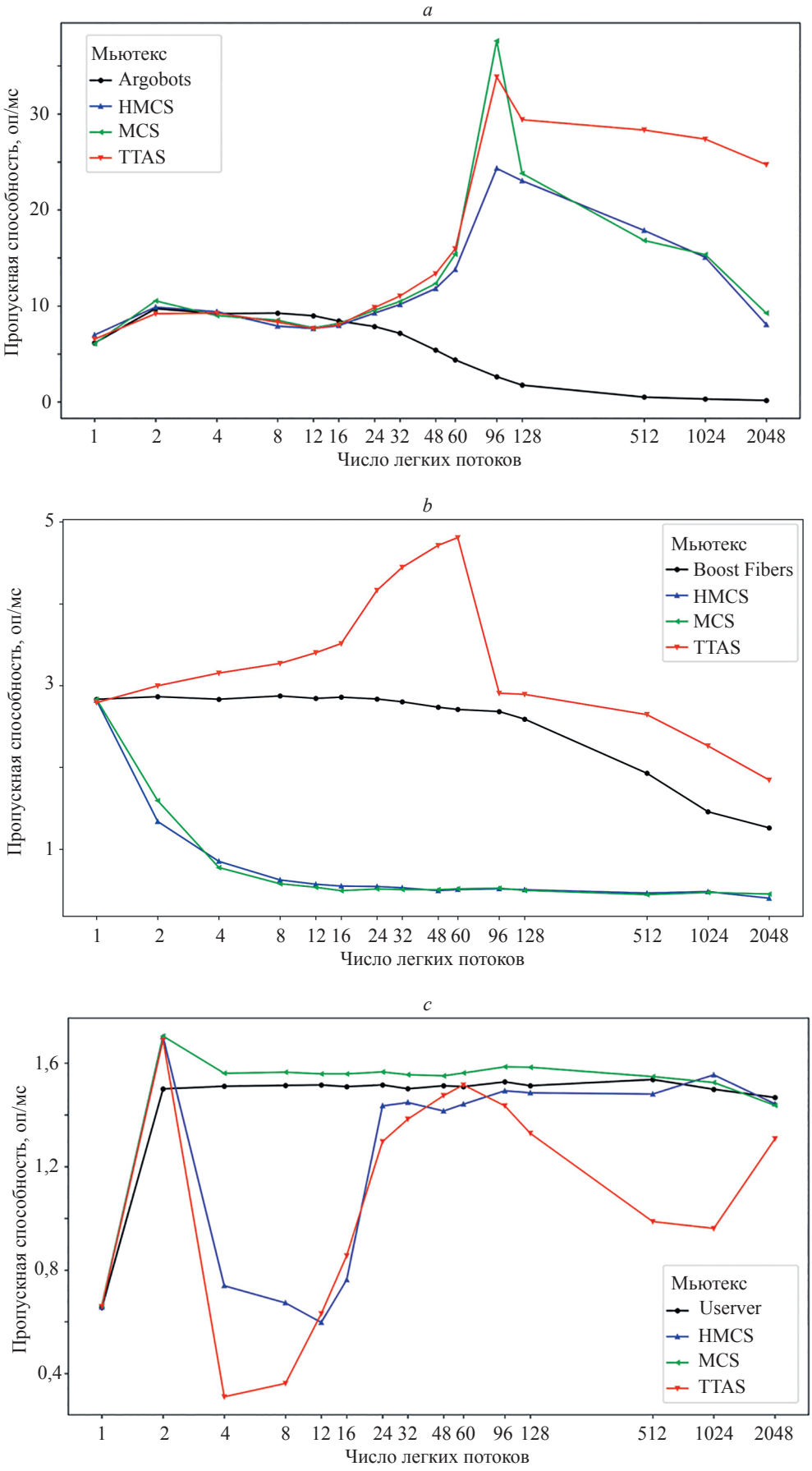

Рисунок. Пропускная способность при использовании реализации Argobots (*a*); Boost Fibers (*b*) и Userver (*c*)

Figure. Throughput when using Argobots (*a*); Boost Fibers (*b*) and Userver (*c*) implementations

Заключение

В работе показано, что реализация мьютексов для легких потоков на языке C++ и сравнение их производительности не является тривиальной задачей.

Описана проблема взаимной блокировки потоков и предложен способ борьбы с ней с помощью трехступенчатого механизма ожидания, включающего стадии активного ожидания, переключения контекста и приостановки потока. Для унификации этого подхода был сформулирован единый интерфейс, который реализуется для каждой библиотеки на основе доступных ей возможностей. Его интеграция в мьютекс производится с помощью делегирования всех точек активного ожидания в коде метода указанного интерфейса, что позволяет не привязывать алгоритм синхронизации к особенностям реализации конкретной библиотеки легких потоков.

Для изучения эффективности мьютексов для легких потоков была разработана система тестирования и оценки производительности в средах с различными реализациями легких потоков. В это решение были интегрированы три известные библиотеки: Argobots, Boost Fibers и Userver. В качестве наглядной демонстрации результата несколько наиболее известных мьютексов с различной структурой, трансформированных описанным способом, были протестированы в трех доступных средах. Кроме того, при реализации мьютексов активно использовались средства отладки. В связи с тем, что полученные замеры не имеют характерного паттерна, присущего всем результатам (рисунок), то однозначный вывод об универсальности какого-то мьютекса сделать невозможно. Но для каждого случая в отдельности можно выделить мьютекс, не уступающий, а в основном превосходящий библиотечную реализацию.

Представлена не только теоретическая идея трансформации мьютексов, но и инструмент для ее проверки на практике. Выполненная оценка производительности алгоритмов синхронизации дала содержательные на данный момент результаты и фундамент для будущих исследований в этом направлении. В предстоящих исследованиях планируется, используя разработанную систему, не только провести сравнение алгоритмов взаимного исключения на реализациях легких потоков из трех представленных библиотек, но и детально рассмотреть зависимость результатов от архитектуры процессора, алгоритмов планирования и способов ожидания. Представленная аналитическая работа поможет в разработке нового комбинированного мьютекса, работающего эффективно вне зависимости от реализации легких потоков.

Литература

1. Moore G.E. Cramming more components onto integrated circuits // Electronics. 1965. V. 38. N 8. P. 114–117.
2. Sutter H. The free lunch is over: a fundamental turn toward concurrency in software // Dr. Dobb's Journal. 2005. V. 30. N 3. P. 202–210.
3. Olukotun O.A., Hammond L., Laudon J.P. Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency. Morgan & Claypool Publishers, 2007. 145 p.
4. Lameter C. NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors // Queue. 2013. V. 11. N 7. P. 40–51. https://doi.org/10.1145/2508834.2513149
5. Seo S., Amer A., Balaji P., Bordage C., Bosilca G., Brooks A., et al. Argobots: a lightweight low-level threading and tasking framework // IEEE Transactions on Parallel and Distributed Systems. 2018. V. 29. N 3. P. 512–526. https://doi.org/10.1109/tpds.2017.2766062
6. Castello A., Gual R.M., Seo S., Balaji P., Quintana-Orti E.S., Pena A.J. Analysis of threading libraries for high performance computing // IEEE Transactions on Computers. 2020. V. 69. N 9. P. 1279–1292. https://doi.org/10.1109/tc.2020.2970706
7. Tanenbaum A.S., Bos H. Modern Operating Systems. Pearson, 2014. 1136 p.
8. Rudolph L., Segall Z. Dynamic decentralized cache schemes for mimd parallel processors // ACM SIGARCH Computer Architecture News. 1984. V. 12. N 3. P. 340–347. https://doi.org/10.1145/773453.808203
9. Mellor-Crummey J.M., Scott M.L. Algorithms for scalable synchronization on shared-memory multiprocessors // ACM Transactions on Computer Systems. 1991. V. 9. N 1. P. 21–65. https://doi.org/10.1145/103727.103729
10. Craig T.S. Building FIFO and Priorityqueuing Spin Locks from Atomic Swap. Technical Report TR 93-02-02. Department of Computer Science, University of Washington, 1993. 29 p.
11. Chabbi M., Fagan M., Mellor-Crummey J. High performance locks for multi-level NUMA systems // ACM SIGPLAN Notices. 2015. V. 50. N 8. P. 215–226. https://doi.org/10.1145/2858788.2688503
12. Dice D., Kogan A. Compact NUMA-aware locks // Proc. of the 14th EuroSys Conference. 2019. P. 1–15. https://doi.org/10.1145/3302424.3303984

References

1. Moore G.E. Cramming more components onto integrated circuits. *Electronics*, 1965, vol. 38, no. 8, pp. 114–117.
2. Sutter H. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005, vol. 30, no. 3, pp. 202–210.
3. Olukotun O.A., Hammond L., Laudon J.P. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan & Claypool Publishers, 2007. 145 p.
4. Lameter C. NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 2013, vol. 11, no. 7, pp. 40–51. https://doi.org/10.1145/2508834.2513149
5. Seo S., Amer A., Balaji P., Bordage C., Bosilca G., Brooks A., et al. Argobots: a lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 2018, vol. 29, no. 3, pp. 512–526. https://doi.org/10.1109/tpds.2017.2766062
6. Castello A., Gual R.M., Seo S., Balaji P., Quintana-Orti E.S., Pena A.J. Analysis of threading libraries for high performance computing. *IEEE Transactions on Computers*, 2020, vol. 69, no. 9, pp. 1279–1292. https://doi.org/10.1109/tc.2020.2970706
7. Tanenbaum A.S., Bos H. *Modern Operating Systems*. Pearson, 2014, 1136 p.
8. Rudolph L., Segall Z. Dynamic decentralized cache schemes for mimd parallel processors. *ACM SIGARCH Computer Architecture News*, 1984, vol. 12, no. 3, pp. 340–347. https://doi.org/10.1145/773453.808203
9. Mellor-Crummey J.M., Scott M.L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 1991, vol. 9, no. 1, pp. 21–65. https://doi.org/10.1145/103727.103729
10. Craig T.S. *Building FIFO and Priorityqueuing Spin Locks from Atomic Swap*. Technical Report TR 93-02-02. Department of Computer Science, University of Washington, 1993, 29 p.
11. Chabbi M., Fagan M., Mellor-Crummey J. High performance locks for multi-level NUMA systems. *ACM SIGPLAN Notices*, 2015, vol. 50, no. 8, pp. 215–226. https://doi.org/10.1145/2858788.2688503
12. Dice D., Kogan A. Compact NUMA-aware locks. *Proc. of the 14th EuroSys Conference*, 2019, pp. 1–15. https://doi.org/10.1145/3302424.3303984

13. Shanley T. x86 Instruction Set Architecture. MindShare Press, 2010. 1568 p.
14. Oberhauser J., Oberhauser L., Paolillo A., Behrens D., Fu M., Vafeiadis V. Verifying and optimizing the HMCS lock for Arm servers // Lecture Notes in Computer Science. 2021. V. 12754. P. 240–260. https://doi.org/10.1007/978-3-030-91014-3_17
15. Goodacre J., Sloss A.N. Parallelism and the ARM instruction set architecture // Computer. 2005. V. 38. N 7. P. 42–50. https://doi.org/10.1109/mc.2005.239
16. Bartel J. Non-preemptive multitasking // The Computer Journal. 1988. V. 30. P. 37–39.
17. Karsten M., Barghi S. User-level threading: Have your cake and eat it too // Proc. of the ACM on Measurement and Analysis of Computing Systems. 2020. V. 4. N 1. P. 1–30. https://doi.org/10.1145/3379483
18. Madsen O.L. Using coroutines for multi-core preemptive scheduling // Proc. of the 11th Workshop on Programming Languages and Operating Systems. 2021. P. 46–52. https://doi.org/10.1145/3477113.3487271
19. Shiina S., Iwasaki S., Taura K., Balaji P. Lightweight preemptive user-level threads // Proc. of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2021. P. 374–388. https://doi.org/10.1145/3437801.3441610
20. Donovan A.A., Kernighan B.W. The Go Programming Language. Addison-Wesley Professional, 2015. 400 p.

13. Shanley T. *x86 Instruction Set Architecture. MindShare Press*, 2010, 1568 p.
14. Oberhauser J., Oberhauser L., Paolillo A., Behrens D., Fu M., Vafeiadis V. Verifying and optimizing the HMCS lock for Arm servers. *Lecture Notes in Computer Science*, 2021, vol. 12754, pp. 240–260. https://doi.org/10.1007/978-3-030-91014-3_17
15. Goodacre J., Sloss A.N. Parallelism and the ARM instruction set architecture. *Computer*, 2005, vol. 38, no. 7, pp. 42–50. https://doi.org/10.1109/mc.2005.239
16. Bartel J. Non-preemptive multitasking. *The Computer Journal*, 1988, vol. 30, pp. 37–39.
17. Karsten M., Barghi S. User-level threading: Have your cake and eat it too. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 2020, vol. 4, no. 1, pp. 1–30. https://doi.org/10.1145/3379483
18. Madsen O.L. Using coroutines for multi-core preemptive scheduling. *Proc. of the 11th Workshop on Programming Languages and Operating Systems*, 2021, pp. 46–52. https://doi.org/10.1145/3477113.3487271
19. Shiina S., Iwasaki S., Taura K., Balaji P. Lightweight preemptive user-level threads. *Proc. of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 374–388. https://doi.org/10.1145/3437801.3441610
20. Donovan A.A., Kernighan B.W. *The Go Programming Language*. Addison-Wesley Professional, 2015, 400 p.

Авторы

Скаженик Тарас Михайлович — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, https://orcid.org/0009-0002-1959-2010, taras.skazhenik@yandex.ru
Аксенов Виталий Евгеньевич — PhD, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, sc 57195411657, https://orcid.org/0000-0001-9134-5490, aksenov.vitaly@gmail.com
Малахов Антон Александрович — научный сотрудник, Университет Неймарк, Нижний Новгород, 603138, Российская Федерация, https://orcid.org/0009-0004-9276-2172, Anton.A.Malakhov@gmail.com
Чурбанов Андрей Васильевич — независимый исследователь, Российская Федерация, https://orcid.org/0009-0009-3165-4932, andrey.churbanov@gmail.com

Authors

Taras M. Skazhenik — PhD Student, ITMO University, Saint Petersburg, 197101, Russian Federation, https://orcid.org/0009-0002-1959-2010, taras.skazhenik@yandex.ru
Vitaly E. Aksenov — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, sc 57195411657, https://orcid.org/0000-0001-9134-5490, aksenov.vitaly@gmail.com
Anton A. Malakhov — Scientific Researcher, Neimark University, Nizhny Novgorod, 603138, Russian Federation, https://orcid.org/0009-0004-9276-2172, Anton.A.Malakhov@gmail.com
Andrey V. Churbanov — Independent Researcher, Russian Federation, https://orcid.org/0009-0009-3165-4932,andrey.churbanov@gmail.com

Статья поступила в редакцию 25.06.2025
Одобрена после рецензирования 14.11.2025
Принята к печати 20.01.2026

Received 25.06.2025
Approved after reviewing 14.11.2025
Accepted 20.01.2026