Проектирование примитивов синхронизации Mutex и Event для многопоточной управляемой среды исполнения со stackful-корутинами

И. А. Панферов

Аннотация — Примитивы синхронизации операционной системы неприменимы в управляемых средах исполнения с корутинами: захват мьютекса ОС блокирует системный поток целиком, лишая исполнения все привязанные к нему корутины, что в условиях кооперативной многозадачности приводит к взаимоблокировкам. Существующие корутино-ориентированные решения тесно привязаны к внутренним структурам конкретной среды (глобальная таблица семафоров в Go, каркас AQS в JVM), рассчитаны на нативное окружение без сборщика мусора (Boost.Fiber) или допускают ожидание только в асинхронном контексте (C#, Kotlin). В работе предложена самостоятельная архитектура примитивов синхронизации Mutex и Event для многопоточной управляемой среды исполнения ArkTS/OpenHarmony со stackful-корутинами. Примитивы являются объектами управляемой кучи со встроенными очередями ожидающих корутин, что допускает перенос архитектуры на другие управляемые среды при наличии в планировщике операций приостановки и возобновления корутины. Одноразовая семантика Event замещает условную переменную, устраняя её известные сложности. Для мьютекса предложена адаптация под короткие критические секции: спин-фаза с экспоненциальной задержкой и адаптивный режим голодания, гарантирующий захват за ограниченное число пробуждений. На основе примитивов построены высокоуровневые механизмы Promise<T> и AsyncLock с обнаружением взаимоблокировок по графу ожидания. Экспериментальная оценка на 8-ядерном устройстве arm64 показала: среднее время захвата адаптивного мьютекса не превышает сотен наносекунд при числе корутин до 1000, накладные расходы в большинстве конфигураций не превышают 1.5x — против 10–14x у sync.Mutex языка Go; ценой выигрыша является рост максимальных пауз при длинных критических секциях и высокой конкуренции.

Ключевые слова — ArkTS, OpenHarmony, взаимоблокировка, корутины, мьютекс, примитивы синхронизации, управляемая среда исполнения.

I. ВВЕДЕНИЕ

Ошибки синхронизации относятся к числу наиболее опасных и труднообнаруживаемых дефектов программных систем. Классическим примером их цены стал аппарат лучевой терапии Therac-25: состояние гонки, проявлявшееся при быстром вводе команд оператором, привело как минимум к шести случаям передозировки радиации со смертельными исходами [1]. Проблема носит систематический характер: анализ 105 конкурентных дефектов в проектах MySQL, Apache, Mozilla и OpenOffice показал, что 31 из них (около 30 %) — взаимоблокировки, а среди 74 остальных дефектов, не связанных с взаимоблокировкой, практически все (97 %) составляют нарушения атомарности составных операций и их порядка [2].

Одновременно традиционная многопоточность с блокирующими системными вызовами уступает место легковесным конкурентным задачам на основе корутин: приостановка и возобновление исполнения выполняются планировщиком среды в пользовательском пространстве, что существенно снижает стоимость переключения контекста и позволяет оперировать миллионами задач в рамках одного процесса. О масштабе перехода свидетельствует хронология внедрения конструкции async/await в массовых языках программирования: C# (2012), Python (2015), JavaScript (2017), Kotlin (2018), Rust (2019), Swift (2021). Дело здесь не только в производительности: управляемые среды развиваются в сторону уменьшения числа ошибок и упрощения разработки, в том числе многопоточной. Та же тенденция видна и в интерфейсах примитивов синхронизации: система примитивов и сам подход к написанию конкурентного кода проектируются безопасными для пользователя, так чтобы типичные ошибки исключались на уровне интерфейса, а не выявлялись при отладке.

Для координации конкурентных задач сложились два подхода: разделяемая память с синхронизацией обращений и передача сообщений (акторная модель). В большинстве управляемых сред память программы разделяется между корутинами, поэтому при их исполнении на нескольких системных потоках необходимы примитивы синхронизации. Классические примитивы операционной системы (например, на основе механизма futex в Linux) в такой среде неприменимы напрямую: захват мьютекса ОС блокирует поток-носитель целиком, что нарушает инвариант планировщика и способно приводить к взаимоблокировке (подробнее — раздел III-B). Требуются примитивы, взаимодействующие с планировщиком корутин: при невозможности захвата ресурса приостанавливается лишь текущая корутина, а поток-носитель продолжает исполнение других задач.

Настоящая работа выполнена в контексте управляемой среды исполнения языка ArkTS операционной системы OpenHarmony [3], в которой применяется многопоточное исполнение stackful-

Статья получена 13.07.2026. Иван Алексеевич Панферов — Московский физико-технический институт (национальный исследовательский университет) (email: panferov.ia@phystech.edu).

корутин — модель, аналогичная используемой в языке Go. Цель работы — спроектировать и реализовать систему надёжных, эффективных и удобных в использовании примитивов синхронизации, адаптированных к корутиновой модели исполнения. В отличие от существующих решений, тесно привязанных к внутренним структурам конкретных сред исполнения, предлагается самостоятельная архитектура примитивов, переносимая на широкий класс управляемых сред с аналогичной моделью исполнения. Вклад работы состоит в следующем: 1) сформулирован набор требований к корутино-ориентированным примитивам синхронизации — корректность в терминах отношения happens-before [4], отсутствие блокировки потока-носителя, производительность, справедливость (fairness) и совместимость со сборщиком мусора; 2) предложена архитектура примитивов Mutex и Event со встроенными очередями ожидающих корутин, интегрированных с планировщиком и удовлетворяющих сформулированным требованиям, включая адаптацию мьютекса к коротким критическим секциям (спин-фазу) и алгоритм пробуждения, исключающий голодание; 3) на их основе построены высокоуровневые механизмы Promise<T> и AsyncLock, включая обнаружение взаимоблокировок; 4) выполнена экспериментальная оценка на CPU-bound- и contention-нагрузках, включая сравнение с sync.Mutex языка Go.

Оставшаяся часть статьи организована следующим образом. В разделе II рассматриваются существующие решения; в разделе III — модели корутин и семантика блокировок в управляемых средах; в разделе IV описываются предлагаемые примитивы; раздел V содержит экспериментальную оценку; раздел VI обсуждает ограничения и направления дальнейшей работы; раздел VII завершает статью.

II. Существующие решения

Задача неблокирующего ожидания в корутиновых средах решена в ряде промышленных систем, различающихся моделью корутин, способом организации очередей ожидания и степенью интеграции с рантаймом. Ниже разобраны четыре характерных решения — примитивы среды Go, виртуальные потоки JVM (Project Loom), библиотека Boost.Fiber и stackless-модель async/await в C# и Kotlin. Для каждого рассматриваются механизм приостановки задачи и устройство очередей ожидания; в конце раздела подходы сопоставляются с требованиями целевого рантайма ArkTS.

A. Go: sync.Mutex и планировщик горутин

Планировщик Go реализует модель M:N: горутины (G) исполняются пулом системных потоков (M), каждому из которых сопоставлен логический процессор (P) с локальной очередью готовых задач и перехватом работы (work stealing) [6]. Приостановка и возобновление горутин выполняются функциями рантайма gopark/goready; на них опираются все блокирующие операции — от каналов до примитивов пакета sync [5].

Объект sync.Mutex состоит из двух 32-битных полей: в поле state закодированы бит захвата, бит наличия пробуждённой горутины, бит режима голодания и счётчик ожидающих; поле sema служит ключом очереди ожидания. Быстрый путь захвата — одна операция CAS по нулевому состоянию. Медленный путь начинается со спин-фазы: ограниченного активного ожидания в расчёте на то, что владелец покинет короткую критическую секцию раньше, чем окупится стоимость парковки. Если спин-фаза не привела к захвату, горутина атомарно увеличивает счётчик ожидающих и паркуется. Очередь ожидания при этом хранится не в самом мьютексе: по ключу sema рантайм находит корзину глобальной таблицы семафоров, внутри которой ожидающие организованы в сбалансированное дерево (treap), защищённое блокировкой корзины. Для исключения голодания мьютекс переходит в режим starvation, если горутина ожидает дольше 1 мс: владение передаётся головной горутине очереди напрямую, а новые претенденты не входят в спин-фазу и встают в хвост очереди.

Слабое место решения — глобальная таблица семафоров. При большом числе одновременно заблокированных независимых мьютексов их очереди отображаются в общие корзины: стоимость операций растёт с O(1) до O(log n) на поиске и балансировке дерева, а горутины, ожидающие разных мьютексов, конкурируют за блокировку одной корзины. Кроме того, sync.Mutex не предоставляет средств обнаружения взаимоблокировок — среда диагностирует лишь глобальную остановку всех горутин.

B. Project Loom: виртуальные потоки JVM

Проект Loom (JEP 444 [7]) вводит в JVM виртуальные потоки — по сути stackful-корутины: виртуальный поток представлен continuation, кадры стека которого хранятся в куче JVM и монтируются на поток-носитель (carrier thread) из пула ForkJoinPool на время исполнения. При блокирующей операции continuation размонтируется, освобождая носитель. Примитивы java.util.concurrent при этом не потребовали переписывания. Мьютекс (ReentrantLock) построен поверх синхронизаторного каркаса AQS (AbstractQueuedSynchronizer) [16]: объект содержит атомарное поле state и встроенную FIFO-очередь ожидающих — вариант CLH-очереди [19], [20]. Быстрый путь захвата — операция CAS по полю state; при неудаче претендент добавляет узел в очередь и паркуется через LockSupport.park, который для виртуального потока приостанавливает continuation вместо потока-носителя. Освобождение сбрасывает state и разблокирует преемника — головной узел очереди. По умолчанию ReentrantLock работает в nonfair-режиме: новый претендент может захватить блокировку в обход очереди (barging), что повышает пропускную способность ценой справедливости; опциональный fair-режим запрещает обгон проверкой наличия предшественников в очереди. Длительное время ограничением был pinning: при ожидании внутри synchronized-секции или нативного вызова виртуальный поток закреплялся за носителем и блокировал его; для synchronized проблема устранена только в JDK 24 (JEP 491 [8]). Решение Loom подтверждает жизнеспособность

прозрачного неблокирующего ожидания в stackful-модели, однако глубоко интегрировано с JVM: continuation, планировщик и примитивы связаны внутренними API рантайма и не переносимы на другие среды.

C. Boost.Fiber

Библиотека Boost.Fiber [9] переносит модель stackful-корутин в нативный C++: файберы переключаются в пространстве пользователя средствами Boost.Context, каждый системный поток исполняет собственный планировщик с подключаемым алгоритмом (round_robin, work_stealing и др.). Реализация fiber::mutex устроена просто: объект хранит указатель на файбер-владелец и интрузивную очередь ожидающих, защищённые внутренним спинлоком. Операция lock под спинлоком проверяет владельца: если мьютекс свободен, текущий файбер записывается владельцем; иначе он добавляется в очередь и приостанавливается, атомарно с этим освобождая спинлок. Операция unlock сбрасывает владельца и пробуждает головной файбер очереди; владение при этом не передаётся — пробуждённый файбер повторяет попытку захвата и может уступить новому претенденту (barging), поэтому FIFO-порядок пробуждения не гарантирует FIFO-порядка владения. Спин-фаза и механизм защиты от голодания, аналогичные применяемым в Go, отсутствуют. Отметим деталь: lock распознаёт тривиальную взаимоблокировку (повторный захват мьютекса собственным владельцем) и возбуждает исключение resource_deadlock_would_occur, однако циклические зависимости между несколькими мьютексами не отслеживаются. Такая архитектура — самодостаточный примитив со встроенной очередью — близка к развиваемой в настоящей работе, однако Boost.Fiber ориентирован на нативную среду: управляемая куча и сборщик мусора отсутствуют, а примитивы не переносимы в managed-рантайм без пересмотра модели памяти и жизненного цикла объектов.

D. Stackless-модель: async/await в C# и Kotlin

В stackless-подходе (машина состояний, приостановка только в размеченных точках await; подробно — раздел III-A) устройство мьютексов сходно с Boost.Fiber: очередь ожидающих continuation встроена в объект примитива и либо реализована неблокируемой (lock-free), либо защищена коротким локом и гарантирует FIFO-порядок пробуждения. Так, в Kotlin примитив kotlinx.coroutines.sync.Mutex [11] — семафор с одним разрешением поверх фреймворка CQS (CancellableQueueSynchronizer) [15]: continuation хранятся в lock-free сегментной очереди с честным FIFO-порядком и поддержкой отмены ожидания, а контракт памяти зафиксирован в документации — unlock находится в отношении happens-before с последующим успешным lock; алгоритмы CQS формально верифицированы в Iris/Coq [15]. В C# асинхронного мьютекса как такового нет — его роль играет SemaphoreSlim с одним разрешением [10]: двусвязный список узлов-задач TaskNode защищён монитором, Release завершает головной узел, передавая continuation в пул потоков; асинхронные ожидающие обслуживаются в FIFO-порядке, однако при смешанной нагрузке приоритет отдаётся синхронным.

Общий недостаток stackless-решений известен как «раскраска функций» (function coloring) [12]: асинхронное ожидание допустимо только внутри асинхронного контекста, поэтому синхронный код не может прозрачно захватить примитив, а библиотечные интерфейсы разделяются на две несовместимые версии. Для рантайма со stackful-корутинами это ограничение принципиально: примитив должен допускать захват из любой точки кода без синтаксических аннотаций.

E. Условные переменные

Единообразие наблюдается не только в мьютексах, но и во втором примитиве — механизме ожидания условия. Классические системы предоставляют для этого условную переменную (pthread_cond_t, sync.Cond в Go, fiber::condition_variable, Monitor.Wait/Pulse в C#), семантика которой сложна для корректного применения: состояние ожидания логически распределено между условной переменной и внешним мьютексом, после каждого пробуждения требуется перепроверка условия в цикле, стандарты допускают ложные пробуждения (spurious wakeup), а нарушение порядка операций ведёт к потере уведомления. В kotlinx.coroutines условной переменной нет вовсе — вместо неё предлагаются специализированные механизмы (Deferred, каналы). В настоящей работе сделан похожий выбор: в базис вместо условной переменной входит одноразовое событие Event с самодостаточной семантикой, свободной от перечисленных сложностей (раздел IV).

F. Сопоставление с требованиями рантайма ArkTS

Целевая среда — управляемый рантайм ArkTS [3]: stackful-корутины, исполняемые несколькими системными потоками над общей управляемой кучей со сборщиком мусора.

Табл. 1. Сопоставление существующих решений с требованиями рантайма ArkTS

Свойство	**Go sync.Mutex**	**Loom / j.u.c.**	**Boost.Fiber**	**C# / Kotlin**
Модель корутин	stackful	stackful	stackful	stackless
Очередь в объекте примитива	нет (глобальная таблица)	да (AQS)	да	да
Защита от голодания	адаптивная (режим starvation)	статическая (fair-режим)	нет (возможен barging)	FIFO (Kotlin), частичная (C#)
Совместимость с GC	да	да	— (нативная среда)	да
Захват из синхронного кода	да	да	да	нет
Обнаружение взаимоблокировок	нет	нет	только самозахват	нет

Отсюда вытекает набор требований к примитиву: объект управляемой кучи под контролем сборщика мусора; приостановка корутины через планировщик без блокировки потока-носителя; захват из любого кода без

раскраски функций; операции с очередью за O(1) независимо от числа примитивов; отсутствие голодания. Ниже они формализованы как T1–T5 (раздел III-D); сопоставление с ними рассмотренных решений приведено в табл. 1.

Ни одно из рассмотренных решений не удовлетворяет всем требованиям сразу. Реализация Go жёстко завязана на внутренние структуры рантайма: очереди ожидающих хранятся в глобальной таблице семафоров с деревьями treap, что не только исключает перенос архитектуры на другую среду, но и оборачивается деградацией операций до O(log n) и конкуренцией за корзины таблицы. Boost.Fiber предлагает нативную реализацию: примитивы не рассчитаны на управляемую кучу, сборщик мусора и его stop-the-world-паузы. В C# и Kotlin использование примитивов возможно только из async-контекста — для среды со stackful-корутинами, где захват должен быть допустим из любой точки кода, это ограничение принципиально. Ближе всего к требованиям подходит Project Loom: stackful-модель, неблокируемые очереди внутри объекта примитива, прозрачный захват из синхронного кода. Но и у него есть слабые места: реализация неотделима от внутренних API JVM (механизм continuation, LockSupport, каркас AQS), а защита от голодания сводится к статическому выбору при создании примитива — nonfair-режим по умолчанию допускает неограниченный обгон, честный же режим сопряжён с существенной потерей пропускной способности; адаптивного переключения, аналогичного режиму starvation в Go, каркас не предоставляет. Наконец, средства обнаружения взаимоблокировок во всех системах ограничиваются распознаванием самозахвата в Boost.Fiber — циклические зависимости между примитивами не отслеживает ни одна из них.

Настоящая работа направлена на закрытие этого пробела: предлагается самостоятельная архитектура самодостаточных примитивов Mutex и Event со встроенными очередями ожидающих — объектов управляемой кучи, интегрированных с планировщиком и сборщиком мусора, — сочетающая прозрачный захват stackful-модели с адаптивной защитой от голодания. Event с упрощённой одноразовой семантикой замещает условную переменную, а на основе примитивов строятся высокоуровневые механизмы, включая диагностику взаимоблокировок. Ключевой элемент архитектуры — GC-совместимая очередь ожидания с нативными узлами на стеках корутин; её устройство и отличие от очередей-локов MCS и CLH описаны в разделе IV-B.

III. Теоретические основы

A. Модели корутин и планирование

Корутина — легковесная конкурентная задача, управляемая планировщиком среды исполнения: приостановка и возобновление выполняются в пространстве пользователя без системного вызова, что делает переключение контекста на один–два порядка дешевле потокового и позволяет оперировать миллионами задач в одном процессе. По способу сохранения состояния при приостановке выделяются два класса реализаций. Stackful-корутины обладают собственным стеком вызовов: приостановка возможна в произвольной точке и прозрачна для вызывающего кода — любая функция может выполнить блокирующую операцию без изменения сигнатур промежуточных вызовов (горутины Go, виртуальные потоки JVM). Stackless-корутины собственного стека не имеют: компилятор преобразует асинхронную функцию в машину состояний, и при приостановке сохраняется лишь объект-continuation текущей функции; приостановка допустима только в размеченных точках await, а асинхронность распространяется вверх по цепочке вызовов (C#, Kotlin, Swift). Stackful-модель расходует больше памяти на задачу, но освобождает пользователя от «раскраски функций» [12].

По способу передачи управления различают вытесняющее планирование, при котором задача снимается с исполнения принудительно по истечении кванта, и кооперативное, при котором корутина сама передаёт управление планировщику. Управляемые среды с корутинами, как правило, реализуют кооперативную модель M:N: корутины исполняются пулом системных потоков, и действует инвариант планировщика — поток-носитель не должен блокироваться, он либо исполняет корутину, либо выбирает следующую готовую задачу. Базовые механизмы передачи управления аналогичны системным вызовам ОС: yield приостанавливает текущую корутину с возвратом в очередь готовых (аналог sched_yield), suspend блокирует корутину до внешнего пробуждения (аналог futex_wait [13]), resume переводит заблокированную корутину в состояние готовности (аналог futex_wake). Именно эти три механизма служат строительными блоками корутино-ориентированных примитивов синхронизации.

B. Семантика блокировок в управляемой среде

Корректный примитив взаимного исключения обязан обеспечивать не только эксклюзивность критической секции, но и упорядоченность наблюдаемых эффектов памяти. Компилятор вправе переставлять независимые операции, а процессор — буферизовать записи в локальном store-buffer, поэтому без явных гарантий корутина, захватившая мьютекс, может прочитать устаревшие значения, записанные предыдущим владельцем. Требуемое свойство формулируется как отношение happens-before [4]: все операции с памятью, выполненные до освобождения примитива, должны быть наблюдаемы корутиной, впоследствии захватившей тот же примитив. Реализуется оно парой барьеров с acquire/release-семантикой [14]: захват содержит acquire-барьер, запрещающий перенос последующих операций выше точки захвата, освобождение — release-барьер, запрещающий перенос предшествующих операций ниже точки освобождения. В управляемой среде эти гарантии формулируются в терминах модели памяти самой среды, а не конкретной аппаратной архитектуры.

Второе отличие управляемой корутиновой среды — недопустимость блокировки потока-носителя. Примитивы ОС (мьютекс на основе futex) при невозможности захвата блокируют системный поток целиком. В корутиновой среде это нарушает инвариант

планировщика: заблокированный поток лишает исполнения все привязанные к нему корутины, включая не конкурирующие за примитив, а в условиях кооперативной многозадачности — в том числе и текущего владельца мьютекса, что приводит к взаимоблокировке (рис. 1). Корутино-ориентированный примитив обязан вместо этого приостанавливать через suspend лишь текущую корутину: поток-носитель остаётся свободным, а при освобождении примитива владелец вызывает resume для одной из ожидающих.

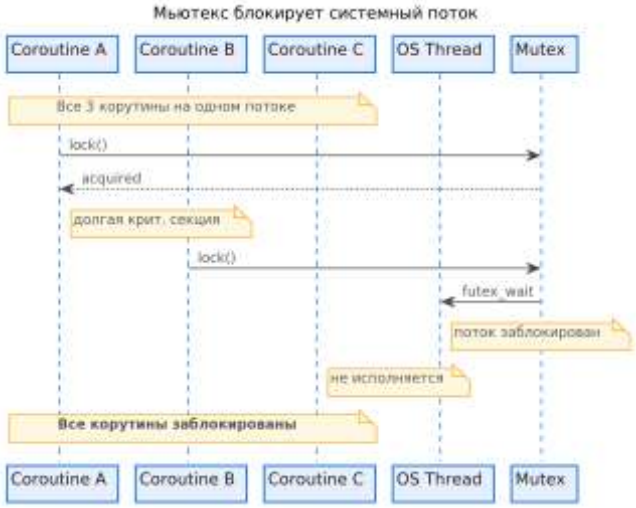

Рис. 1. Проблема блокировки потока-носителя при использовании примитива ОС в корутиновой среде.

Третья особенность — взаимодействие со сборщиком мусора. Примитив синхронизации в управляемой среде является объектом управляемой кучи: его время жизни контролирует сборщик мусора, и реализация не должна приводить к утечкам нативной памяти. Кроме того, реализация должна быть безопасна относительно stop-the-world-пауз: примитив не должен удерживать нативные блокировки, препятствующие остановке мутаторов и запуску сборщика.

C. Общая схема корутино-ориентированного примитива

Сопоставление реализаций, рассмотренных в разделе II, обнаруживает общую трёхшаговую схему, которой следуют все системы: 1) атомарная проверка и модификация компактного состояния примитива — быстрый путь захвата; 2) при конкуренции — постановка текущей задачи (горутины, виртуального потока, файбера, continuation) в очередь ожидающих; 3) приостановка через suspend-механизм планировщика и симметричное пробуждение через resume при освобождении. Однако единой переносимой архитектуры за этой схемой не стоит: реализации расходятся во всех ключевых решениях — где хранится очередь (глобальная таблица семафоров в Go против встроенной в объект у остальных), как организован доступ к ней (неблокируемые CLH-очередь AQS в JVM и сегментная очередь CQS в Kotlin — против очередей под коротким локом: спинлок корзины таблицы в Go, спинлок в Boost.Fiber, монитор в SemaphoreSlim) и как достигается справедливость (двухрежимная схема Go, опциональный fair-режим j.u.c., FIFO в Kotlin, barging в Boost.Fiber, приоритет синхронных ожидающих в C#). Каждая реализация при этом опирается на внутренние механизмы своей среды исполнения и не переносима на другие. Эта трёхшаговая схема принимается за основу и в настоящей работе; требования следующего подраздела фиксируют, какими свойствами должна обладать её реализация в целевой среде.

D. Требования к примитивам Mutex и Event

На основании изложенного сформулируем требования к проектируемым примитивам; каждому придана проверяемая формулировка, на которую опираются разделы IV и V.

T1. Корректность. Мьютекс обеспечивает взаимное исключение: при произвольном числе корутин и системных потоков в критической секции находится не более одной корутины. Освобождение примитива состоит в отношении happens-before с последующим его захватом; операции над внутренним состоянием примитива — изменение флага захвата, постановка в очередь, извлечение из очереди — атомарны и свободны от гонок. Проверка: обоснование по модели памяти среды и конкурентные стресс-тесты.

T2. Отсутствие блокировки потока-носителя. При невозможности захвата корутина приостанавливается через интерфейс планировщика, а поток-носитель продолжает исполнение других корутин. Проверка: при ожидании примитива корутинами одного потока прогресс остальных корутин этого потока сохраняется.

T3. Производительность. Захват и освобождение в отсутствие конкуренции выполняются за несколько атомарных операций без системных вызовов; операции с очередью ожидания выполняются за O(1) независимо от числа примитивов в системе — прямое следствие анализа глобальной таблицы семафоров Go (раздел II-A). Проверка: микробенчмарки в сравнении с нативными аналогами (раздел V).

T4. Справедливость. Порядок пробуждения исключает неограниченное откладывание: корутина, поставленная в очередь ожидания, получает примитив за конечное число шагов. Минимально достаточное условие — обслуживание очереди в порядке FIFO. Проверка: ограниченность числа обгонов ожидающей корутины новыми претендентами.

T5. Совместимость со сборщиком мусора. Примитив и его очередь ожидания являются объектами управляемой кучи; реализация не создаёт утечек нативной памяти и не удерживает блокировок, препятствующих stop-the-world-паузам. Проверка: тесты с принудительными циклами сборки мусора под конкурентной нагрузкой.

Требования T1–T5 предъявляются к двум примитивам; семантическое различие между ними — взаимное исключение у Mutex и одноразовое сигнальное состояние у Event — фиксируется интерфейсами, описанными в разделе IV.

IV. Предлагаемый метод

A. Общая схема и интерфейсы планировщика

Предлагаемые примитивы следуют трёхшаговой схеме из раздела III-C; отличие от существующих

решений — в организации очереди ожидающих и способе её встраивания в объект управляемой среды.

Взаимодействие с планировщиком сведено к минимальному интерфейсу. Ожидающую корутину со свершением события связывает структура BlockingEvent, содержащая флаг наступления события и указатель на корутину. Планировщик предоставляет две операции: Await атомарно проверяет состояние события и, если оно не наступило, переводит корутину в заблокированное состояние, сохраняет её контекст на собственном стеке и передаёт управление следующей готовой корутине — поток-носитель при этом не блокируется (требование T2); Unblock отмечает событие наступившим и возвращает корутину в очередь готовых к исполнению. Примитивы опираются только на эти две операции. Таким образом, от планировщика среды требуется лишь два элемента интерфейса: операции приостановки и возобновления корутины (suspend/resume, роль которых исполняют Await и Unblock) и доступ к указателю на текущую исполняемую корутину. Эквивалентные механизмы присутствуют в планировщике любой кооперативной корутиновой среды (раздел III-A); при наличии этих двух примитивов интерфейса предложенная архитектура допускает перенос на другую управляемую среду без изменения самих примитивов синхронизации.

B. Очередь заблокированных корутин

В отличие от Go, очередь заблокированных корутин хранится непосредственно в примитиве, что исключает взаимное влияние независимых примитивов и даёт O(1) на операциях с очередью (T3). Поскольку примитив — объект управляемой кучи, очередь также должна быть управляемым объектом. Прямолинейное решение — защитить очередь системным мьютексом, сохранив нативный указатель на него в поле типа long, — ведёт к утечке: сборщик мусора не осведомлён, что поле содержит указатель на нативную память, и при удалении объекта память мьютекса не освобождается (нарушение T5). Отсюда следуют два проектных ограничения: синхронизация очереди должна быть неблокирующей, а время жизни нативных данных — явно ограниченным.

Предложенная очередь (рис. 2) — неограниченная интрузивная MPSC-очередь (несколько производителей — конкурентные вызовы lock и wait; один потребитель — владелец примитива, вызывающий unlock или fire), построенная на идее стека Трайбера [17], [23] и следующая известной конструкции интрузивной MPSC-очереди Д. Вьюкова [21]. Вставка выполняется CAS-подменой атомарного указателя на хвост в цикле до первой удачной попытки. Извлечение выполняется потребителем без атомарных операций: за голову списка никто не конкурирует; когда голова пуста, хвост атомарно обменивается на пустой указатель, и изъятый список разворачивается (операция restock). Если потребитель застал очередь пустой при ненулевом счётчике ожидающих — окно между атомарным увеличением счётчика и фактической вставкой узла, — он дожидается появления узла в коротком активном цикле. Операции с очередью выполняются с порядком памяти acquire-release, что устанавливает happens-before между инициализацией узла до вставки и обращением к его полям после извлечения (T1).

Центральное место в конструкции занимает размещение узлов: узел очереди аллоцируется на стеке засыпающей корутины и содержит нативные указатели — структуру BlockingEvent и указатель на следующий узел. Время жизни узла ограничено вызовом блокирующего интерфейса: корутина удаляет узел после собственного пробуждения, поэтому утечки нативной памяти исключены, а аллокаций в управляемой куче на пути блокировки не возникает. Приём размещения узла ожидания на стеке самой ожидающей задачи восходит к масштабируемым очередям-локам MCS [18] и CLH [19], [20]; в предлагаемой конструкции он адаптирован к управляемой среде: на стеке корутины размещается интрузивный узел с нативными указателями, что сохраняет совместимость со сборщиком мусора. Сам управляемый объект очереди хранит лишь атомарный указатель на хвост. Такая комбинация — управляемая обёртка плюс интрузивные нативные узлы на стеках корутин — сохраняет независимость от внутренних структур конкретной среды и, как следствие, переносимость (раздел IV-A).

Рис. 2. Очередь заблокированных корутин WaitersList: FIFO-список с узлами на стеках корутин и атомарным указателем на хвост (managed-поле long поверх native std::atomic<Node*>).

C. Примитивы Mutex и Event

Mutex и Event являются объектами управляемой кучи: их время жизни контролирует сборщик мусора наравне с любым другим объектом (T5). Реализация алгоритмов при этом нативная: управляемому классу соответствует нативный класс-зеркало, поля которого объявлены как std::atomic. Корректность такого отображения обеспечивается совпадением расположения памяти: для типов размером не более машинного слова std::atomic<T> имеет тот же layout, что и T. За счёт этого все операции над состоянием примитива атомарны, и примитив как разделяемый объект свободен от гонок данных (T1).

Состояние Event кодируется 32-битным полем: бит 0 — наступило ли событие, биты 1–31 — счётчик ожидающих корутин. Метод wait операцией fetch_add увеличивает счётчик и анализирует предыдущее значение: если событие уже наступило, управление возвращается немедленно с acquire-семантикой; иначе корутина добавляет узел в очередь и приостанавливается. Метод fire операцией exchange устанавливает бит события: если оно уже было взведено, метод завершается; иначе из предыдущего значения извлекается число ожидающих, и все они разблокируются. При конкурентных wait и fire либо засыпающая корутина увидит наступившее событие, либо пробуждающая увидит увеличенный счётчик — потерянное пробуждение исключено атомарностью fetch_add и exchange относительно друг друга. Семантика Event одноразовая: после fire объект навсегда переходит в сигнальное состояние, повторные fire эффекта не имеют, а последующие wait завершаются немедленно, сохраняя happens-before с вызовом fire. В отличие от условной переменной (раздел II-E) состояние ожидания не распределено между двумя объектами, а перепроверка условия в цикле и ложные пробуждения исключены самой семантикой примитива.

Строгая реализация Mutex использует поле-счётчик waiters: lock операцией fetch_add увеличивает счётчик, и переход от нуля к единице означает захват по быстрому пути; при ненулевом предыдущем значении корутина встаёт в очередь и приостанавливается. unlock операцией fetch_sub уменьшает счётчик; если очередь не пуста, головная корутина разблокируется и получает владение мьютексом напрямую — после пробуждения она не пытается захватить его снова, поскольку ненулевой счётчик закрывает быстрый путь остальным. Это даёт строгий FIFO-порядок и полностью устраняет голодание: каждая корутина получает мьютекс за один шаг (T4). Формальное обоснование корректности алгоритмов в терминах модели памяти среды выходит за рамки статьи; ключевые аргументы — взаимная атомарность RMW-операций и acquire-release-порядки операций с очередью — приведены по ходу изложения.

D. Баланс справедливости и производительности

Строгая реализация удовлетворяет требованиям T1, T2, T4, T5 и имеет быстрый путь в одну атомарную операцию, однако на коротких критических секциях при высокой конкуренции деградирует. Причина в самой передаче владения: корутина, только что освободившая мьютекс, не может немедленно перезахватить его по быстрому пути — она вынуждена вставать в очередь, которая при высокой конкуренции стремительно растёт, нагружая планировщик. В измерениях среднее время захвата вырастало до единиц микросекунд, а замедление программы достигало ~68 раз (раздел V, рис. 5, а).

Для коротких критических секций от строгого FIFO пришлось отступить. Адаптивная реализация отказывается от отдельного поля счётчика: всё состояние упаковано в единственное машинное слово state — бит 0 (LOCKED), бит 1 (STARVATION) и биты 2–63, хранящие указатель на хвост списка ожидающих, что возможно благодаря выравниванию адресов минимум на 4 байта. Захват начинается с CAS-перехода UNLOCKED→LOCKED; при неудаче корутина входит в спин-фазу — несколько раундов попыток захвата с экспоненциально возрастающей задержкой между ними [22] (функция BackOff, на части архитектур — инструкция pause). Активное ожидание здесь оправдано: если критическая секция коротка, владелец освободит мьютекс за десятки–сотни тактов, и спин обойдётся дешевле приостановки — без обращения к планировщику, переключения контекста и сохранения регистров; неограниченный спин, напротив, недопустим, поскольку отнимает время у корутин того же потока-носителя. Общее время спин-фазы должно быть сопоставимо с характерным временем критической секции: при слишком короткой спин-фазе корутина всё равно приостановится, добавив холостые попытки к стоимости парковки, при слишком длинной — процессорное время тратится впустую после того, как быстрый захват уже стал невозможен. Если спин-фаза не принесла успеха, корутина вставляет узел в список одним CAS, одновременно устанавливающим бит LOCKED и подменяющим указатель на хвост, и приостанавливается.

Освобождение симметрично: быстрый путь — CAS-переход LOCKED→UNLOCKED при пустом списке; при непустом списке ожидающие корутины разблокируются, однако владение мьютексом им не передаётся — в этом главное отличие от строгой реализации. Пробуждённая корутина возобновляет спин-фазу и конкурирует за мьютекс на общих основаниях с новоприбывшими (barging). Это позволяет владельцу немедленно перезахватывать мьютекс в обход очереди, существенно повышая пропускную способность, но создаёт предпосылки для голодания.

Защита от голодания реализована адаптивно. Узел списка дополнен счётчиком пробуждений: если корутина проиграла конкуренцию 16 пробуждений подряд, она помечает свой узел флагом голодания, переводит мьютекс в состояние STARVATION и блокируется; прочие корутины, обнаружив этот режим, немедленно покидают спин-фазу. unlock, встретив узел с флагом голодания, действует по строгой схеме — передаёт владение голодающей корутине напрямую, после чего примитив возвращается в обычный режим. В итоге корутина гарантированно получает мьютекс не позднее чем через 16 собственных пробуждений (T4). Ограничения эвристики: константы раундов и задержек спин-фазы и порог пробуждений подобраны

эмпирически и зависят от числа ядер и характера нагрузки, а режим голодания обслуживает одну корутину за раз. Алгоритмы захвата, освобождения и спин-фазы TrySpinLockFor приведены на рис. 3.

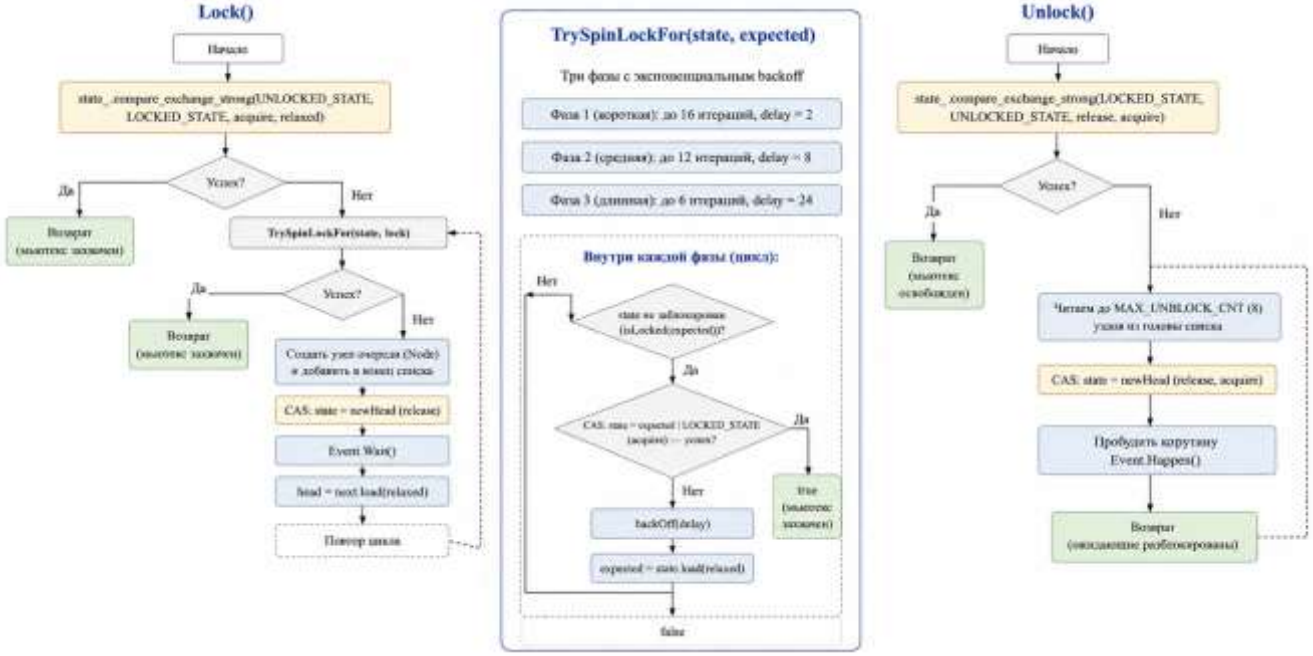

Рис. 3. Алгоритмы захвата и освобождения адаптивного мьютекса.

E. Высокоуровневые механизмы Promise<T>, AsyncLock

Mutex и Event — низкоуровневый базис, с которым пользователь напрямую работает редко; их роль — служить строительными блоками высокоуровневых абстракций, как Task в C# или каналы в Go. В рассматриваемой среде на их основе реализованы два механизма.

Promise<T> представляет результат асинхронного вычисления, доступный в будущем, — прямой аналог Task в C#: цепочки then/catch/finally образуют граф продолжений, а оператор await приостанавливает корутину до заполнения или отклонения промиса. Вся синхронизация сведена к одному Event, на котором приостанавливается await, и одному Mutex, защищающему очередь колбэков от конкурентной регистрации, — дополнительных примитивов реализация не требует.

AsyncLock — блокировка для асинхронного кода: метод lockAsync захватывает блокировку, асинхронно выполняет переданный колбэк, освобождает блокировку и возвращает Promise с результатом. Поддерживаются режимы exclusive и shared, превращающие AsyncLock в блокировку «писатель–читатели», аналогичную sync.RWMutex. Кроме того, AsyncLock обнаруживает взаимоблокировки: все живые экземпляры регистрируются в общем реестре, который по информации о захваченных и ожидаемых блокировках строит граф ожидания (wait-for graph) [24] и ищет в нём циклы; если захват не завершается за отведённое время и цикл найден, промис вложенного lockAsync отклоняется с описанием цикла A→B→A. Подобная диагностика отсутствует у примитивов рассмотренных в разделе II сред. Оба механизма используют один шаблон: Mutex защищает служебное состояние объекта, а Event — общий либо по одному на каждого ожидающего — отвечает за адресное пробуждение корутин. Реализация двух качественно разных механизмов исключительно поверх Mutex и Event подтверждает, что выбранный базис достаточен для построения библиотеки синхронизации корутиновой среды.

V. ЭКСПЕРИМЕНТАЛЬНАЯ ОЦЕНКА

A. Методика измерений

Производительность двух предложенных примитивов оценивается по-разному. Все операции Event выполняются за O(1) при любой нагрузке: wait и fire сводятся к одной атомарной RMW-операции на быстром пути и константному числу операций с очередью на медленном, поэтому производительность события ограничивается исключительно пропускной способностью планировщика, осуществляющего приостановку и возобновление корутин, и отдельного исследования не требует. Основным объектом оценки является Mutex.

Измерение производительности мьютекса представляет собой самостоятельную исследовательскую задачу: единой метрики не существует, поскольку любой показатель зависит от характера нагрузки — длительности критической секции, числа конкурирующих корутин, количества потоков-воркеров и загруженности планировщика. По этой причине в работе используются три взаимодополняющие метрики. Среднее время захвата мьютекса характеризует общий прогресс системы — типичные накладные расходы на синхронизацию в установившемся режиме. Максимальное время захвата мьютекса конкретной корутиной показывает, насколько эффективно решается проблема голодания и насколько долго прогресс отдельной задачи может быть приостановлен: если максимум растёт существенно быстрее среднего, отдельные корутины систематически

вытесняются конкурентами (требование T4). Наконец, поскольку синхронизация сама по себе не ускоряет программу, а лишь добавляет издержки в обмен на защиту от гонок, накладные расходы измеряются как отношение времени работы программы, защищённой мьютексом, ко времени той же программы без синхронизации (с пустой блокировкой); эта метрика отражает относительное замедление полезной работы, то есть пропускную способность (T3).

Методика едина для исследуемой среды и Go: запускаются N конкурентных задач, каждая из которых выполняет фиксированное число итераций «захват — критическая секция длительностью delay — освобождение». Конфигурации покрывают диапазон от CPU-bound-нагрузки (длинные критические секции) до чистой contention-нагрузки (пустая критическая секция при тысячах конкурирующих корутин); варьируются длительность критической секции (0–100 мкс) и число корутин (до 7000). Для каждой конфигурации тот же бенчмарк дополнительно прогоняется с DummyLock — пустой блокировкой без синхронизации. Бенчмарк портирован на Go без изменения структуры; сравнение проводится только с sync.Mutex, поскольку из рассмотренных систем именно рантайм Go ближе всего к исследуемой среде по модели конкурентности — многопоточное исполнение stackful-корутин над разделяемой памятью. Все измерения выполнялись на смартфоне Huawei Mate 70 Pro (SoC HiSilicon Kirin 9020: два больших ядра Taishan с частотой до 2.5 ГГц, шесть средних до 2.1 ГГц и четыре малых до 1.6 ГГц. Для устранения влияния гетерогенного планирования и динамического управления частотой бенчмарк привязывался средствами cpuset к восьми производительным ядрам — двум большим и шести средним, — а их тактовая частота фиксировалась на 2 ГГц; четыре малых ядра из-под нагрузки исключались.

Каждая конфигурация измерялась в десяти прогонах. Каждому замеру предшествовал прогрев — проход с DummyLock и два прохода с Mutex, — доводящий JIT-компиляцию горячих методов до установившегося состояния. На пути блокировки аллокаций в управляемой куче не происходит: узлы очереди размещаются на стеках корутин (раздел IV-B), поэтому сборщик мусора в измеряемых участках не срабатывает и на результаты не влияет (T5). Базовой линией служит DummyLock: общий счётчик итераций в обеих версиях инкрементируется атомарно, поэтому базовая линия свободна от гонок данных и измеряет чистую стоимость каркаса и критической секции без синхронизации, относительно которой вычисляются накладные расходы.

Стоит упомянуть и отрицательный результат. Попытка выполнить тот же бенчмарк в среде ArkTS с нативным мьютексом ОС вместо предложенного примитива завершилась взаимоблокировкой: заблокированные потоки-носители лишили исполнения корутины, способные освободить мьютекс. Сценарий из раздела III-B (рис. 1) воспроизвёлся на практике — требование T2 оказывается не оптимизацией, а условием работоспособности синхронизации в корутиновой среде.

B. Результаты

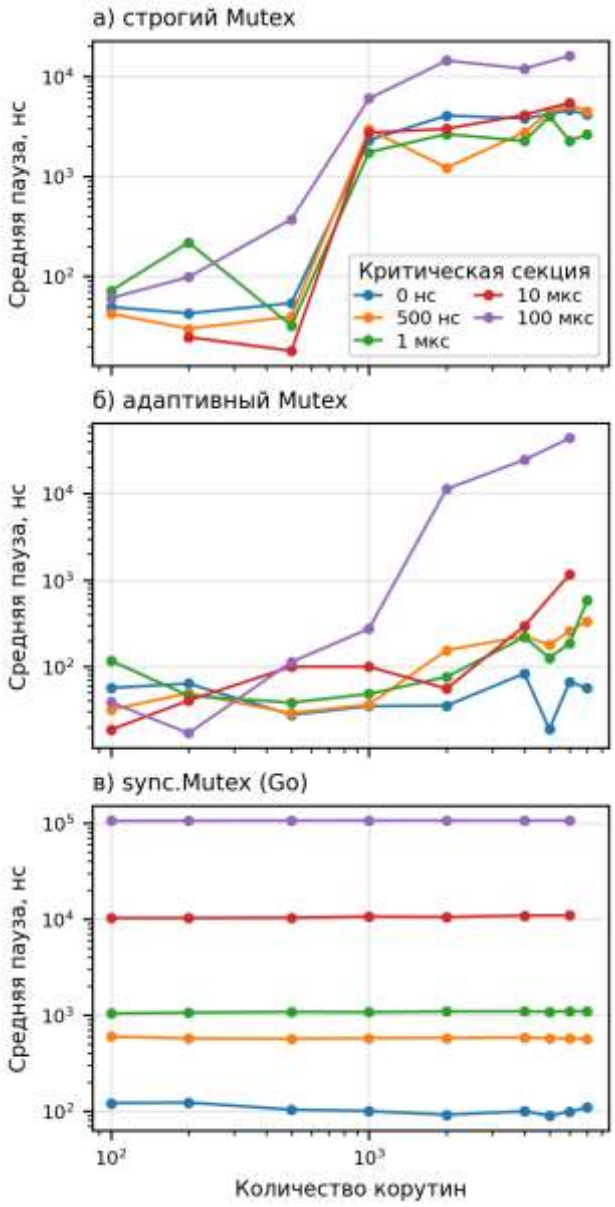

Рис. 4. Средняя пауза захвата мьютекса: а) строгий Mutex; б) адаптивный Mutex; в) sync.Mutex (Go).

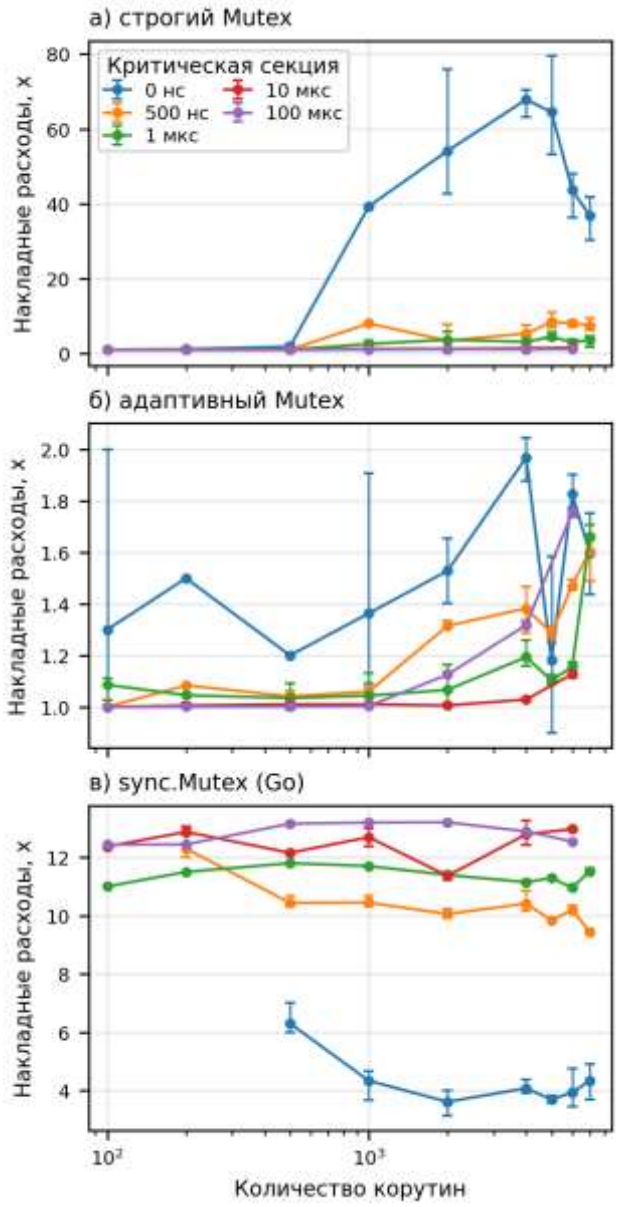

Рис. 5. Накладные расходы на синхронизацию: а) строгий Mutex; б) адаптивный Mutex; в) sync.Mutex (Go). Планки погрешностей — разброс min/max по трём прогонам.

При числе корутин до 1000 среднее время захвата адаптивного мьютекса не превышает нескольких сотен наносекунд независимо от длительности критической секции — большинство корутин захватывают мьютекс в спин-фазе (рис. 4, б). Для самой длинной критической секции (100 мкс) с ростом конкуренции среднее время захвата растёт — мьютекс всё чаще работает в режиме голодания — и составляет ~11 мкс при 2000 и ~44 мкс при 6000 корутин, оставаясь меньше длительности самой секции. Накладные расходы в большинстве конфигураций не превышают 1.5x и лишь в отдельных точках (около 4000 корутин при пустой критической секции) достигают пика ~2x (рис. 5, б). Для сравнения на рис. 4, а и 5, а приведены показатели строгой реализации: с ростом конкуренции средняя пауза выходит на единицы микросекунд при любой длительности секции, а замедление на чистой contention-нагрузке достигает ~68 раз.

Для sync.Mutex картина иная: среднее время захвата практически совпадает с длительностью критической секции при любом ненулевом delay (порядка 10 мкс при секции 10 мкс, порядка 110 мкс при секции 100 мкс) — в среднем вызывающая горутина простаивает всё время удержания мьютекса (рис. 4, в). Это прямое следствие честной FIFO-передачи владения в sync.Mutex: под конкуренцией среднее ожидание закономерно растёт с позицией в очереди, тогда как адаптивный мьютекс с barging жертвует справедливостью ради среднего времени. Поэтому сравнение средних времён захвата отражает в том числе различие политик, а не только эффективность реализации. Накладные расходы для критических секций от 1 мкс стабильно составляют 10–14x практически независимо от числа горутин; даже для пустой критической секции они достигают 4–6x (рис. 5, в). Сводные результаты приведены в табл. 2.

Табл. 2. Сравнение адаптивного Mutex с sync.Mutex (Go)

Метрика (конфигурация)	**Адаптивный Mutex**	**sync.Mutex**
Среднее время захвата (до 1000 корутин)	130–170 нс (не растёт с числом корутин)	≈ длительность КС (при delay>0)
Среднее время захвата (КС 100 мкс, 6000 корутин)	≈ 44 мкс	≈ 110 мкс
Накладные расходы (типичные)	≤ 1.5x	10–14x (КС ≥ 1 мкс)
Накладные расходы (пик, пустая КС)	≈ 2x	4–6x
Макс. пауза (КС 0 нс — 1 мкс)	сопоставимы (в пределах порядка)	сопоставимы
Макс. пауза (КС 100 мкс, тысячи корутин)	до десятков секунд	до ≈ 1.4 с

C. Максимальные паузы и интерпретация

Сопоставление максимальных пауз даёт более сложную картину (рис. 6). У адаптивного мьютекса при числе корутин до 500 максимальная пауза остаётся в диапазоне сотен наносекунд — десятков микросекунд; начиная с 1000 корутин она скачкообразно возрастает на три и более порядка, достигая десятков миллисекунд при пустой критической секции и десятков секунд при секции 100 мкс. У sync.Mutex резкий рост максимальной паузы (до единиц–десятков миллисекунд) присутствует уже при 100 горутинах, однако с ростом конкуренции она увеличивается слабо, не превышая ≈1.4 с даже в самой тяжёлой конфигурации. Таким образом, при секциях до 1 мкс максимальные паузы обеих реализаций сопоставимы — различие в пределах порядка и не всегда в пользу Go; при секции 10 мкс адаптивный мьютекс уступает примерно на порядок, при 100 мкс — примерно на полтора порядка (до 30–40 раз). Это согласуется с отмеченным в разделе IV-D ограничением эвристики: режим голодания обслуживает одну корутину за раз и не масштабируется на множество одновременно голодающих корутин при длинных секциях и высокой конкуренции.

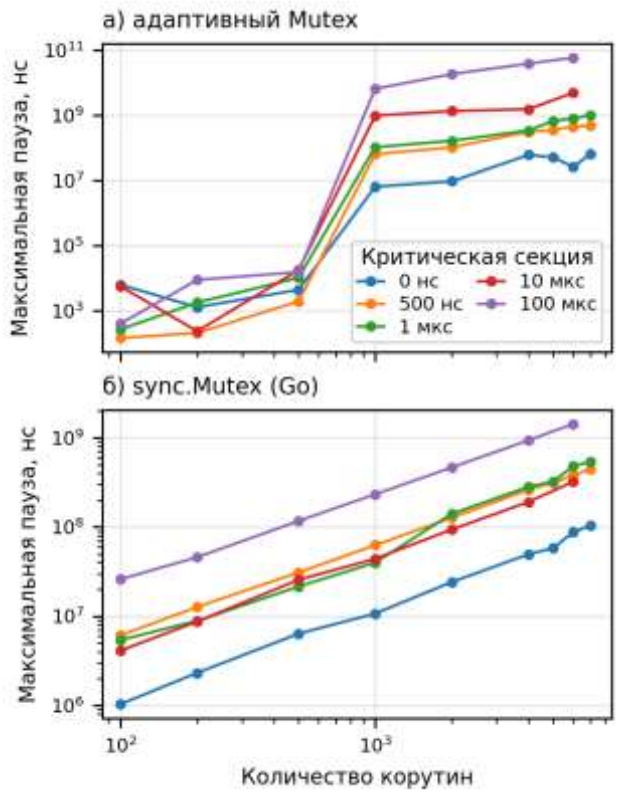

Рис. 6. Максимальная пауза захвата мьютекса: а) адаптивный Mutex; б) sync.Mutex (Go).

В целом предложенная реализация выигрывает по среднему времени захвата и пропускной способности во всём рассмотренном диапазоне нагрузок, расплачиваясь худшим поведением в наихудшем случае — при сочетании длинных критических секций с высокой конкуренцией. Требование T3 выполнено: накладные расходы на порядок ниже, чем у ближайшего аналога, а быстрый путь не содержит системных вызовов и аллокаций. Требование T4 также выполняется — голодающая корутина получает мьютекс не позднее чем через 16 собственных пробуждений, — но ценой роста хвостовых задержек при экстремальных нагрузках.

VI. Ограничения и направления дальнейшей работы

Предложенная архитектура даёт работающий и переносимый базис примитивов синхронизации, однако ряд аспектов остался за рамками настоящей работы. Ниже они систематизированы как ограничения текущего результата и соответствующий им план дальнейших исследований — ориентир для тех, кто будет развивать это направление. Часть из перечисленного (эмпирический подбор параметров, оценка на одном устройстве) ограничивает именно инженерную реализацию адаптивного мьютекса; ядро вклада работы — переносимая архитектура примитивов и сформулированные требования к её миграции (достаточность минимального интерфейса suspend/resume и указателя на текущую корутину, раздел IV-A) — от этих ограничений не зависит.

A. Эмпирический подбор и чувствительность параметров

Адаптивная эвристика мьютекса (раздел IV-D) опирается на три подбираемых параметра: суммарную длительность спин-фазы, расписание экспоненциальной задержки (число раундов и величина шага) и порог пробуждений для перехода в режим голодания. В текущей реализации они подобраны по микробенчмаркам и анализу типичной длительности критических секций на целевом устройстве: характерные секции укладываются в диапазон 1–10 мкс, а суммарная длительность спин-фазы установлена около 8 мкс.

Чувствительность к этим параметрам носит следующий качественный характер. Длительность спин-фазы привязана к среднему времени захвата и освобождения: её увеличение сверх характерной длины критической секции повышает среднее время, поскольку корутины дольше остаются в активном ожидании уже после того, как быстрый захват стал невозможен; слишком короткая спин-фаза, напротив, добавляет холостые попытки к стоимости приостановки (раздел IV-D). Порог пробуждений управляет балансом между средней и хвостовой задержкой: при его стремлении к единице алгоритм вырождается в строгий FIFO (средние задержки растут — строгая реализация из разделов IV-C и IV-D), а при стремлении к сколь угодно большим значениям (порядка UINT_MAX) проблема голодания не решается вовсе и неограниченно растут максимальные задержки. Выбранное значение 16 балансирует эти крайности для рассмотренных нагрузок.

B. Переносимость и настройка под целевое устройство

Экспериментальная оценка проведена на одном устройстве (SoC Kirin 9020) — целевой платформе, под которую оптимизированы параметры спин-фазы; переносимость самих числовых констант (в отличие от переносимости архитектуры, обоснованной в разделе IV-A) на другое оборудование не установлена. В общем случае оптимизированный алгоритм должен поддерживать либо адаптивный режим (онлайн-настройка, см. подраздел A), либо конфигурацию параметров на этапе компиляции под конкретное устройство — по аналогии с реализацией pthread_mutex, где поведение спина задаётся параметром pthread_mutexattr_setspinlock_np.

C. Энергопотребление

Активное ожидание в спин-фазе разменивает процессорные такты на снижение стоимости приостановки; на мобильных платформах у этого размена есть энергетическая составляющая, которую настоящая работа не измеряет. Для целевого окружения OpenHarmony энергопотребление — критическая метрика, поэтому добавление энергоориентированной оценки как дополнительного критерия при выборе длительности спин-фазы, — естественное расширение работы.

D. Формальная верификация

Корректность алгоритмов в настоящей работе обоснована неформально: ключевые аргументы — взаимная атомарность RMW-операций и acquire-release-порядок операций с очередью — приведены по ходу изложения (разделы IV-B, IV-C), однако формального

доказательства нет. Формальная верификация примитивов средствами проверки моделей или в системе интерактивного доказательства (как это сделано для каркаса CQS в Iris/Coq [15]) повысила бы уверенность в корректности относительно модели памяти, особенно с учётом тонкого окна между атомарным увеличением счётчика ожидающих и фактической вставкой узла в очередь (раздел IV-B).

E. Расширение сравнительной оценки

Сравнение ограничено мьютексом sync.Mutex языка Go, выбранным в качестве ближайшего аналога по модели конкурентности (многопоточное исполнение stackful-корутин над разделяемой памятью). Картину уточнили бы ещё сравнение с реализацией мьютекса в JVM Loom (ReentrantLock поверх каркаса AQS), который лишён спин-фазы и предоставляет лишь статический режим справедливости, фиксируемый при создании примитива; тем не менее сравнение позволило бы количественно оценить цену такого статического выбора относительно адаптивного переключения. Расширение оценки на эти системы оставлено для дальнейшей работы.

VII. Заключение

В работе решена задача проектирования и реализации примитивов синхронизации для многопоточной управляемой среды исполнения со stackful-корутинами. Показано, что примитивы операционной системы неприменимы в такой среде в исходном виде: блокировка потока-носителя нарушает инвариант планировщика и способна приводить к взаимоблокировкам. Существующие корутино-ориентированные решения, в свою очередь, привязаны к внутренним структурам конкретной среды, не рассчитаны на управляемую память или ограничивают ожидание асинхронным контекстом.

На основе анализа существующих систем сформулированы пять проверяемых требований к корутино-ориентированным примитивам и предложена самостоятельная архитектура, в которой базисом служат самодостаточные Mutex и Event со встроенными очередями ожидающих. Основу архитектуры составляет неблокируемая интрузивная MPSC-очередь: её управляемая обёртка хранит лишь атомарный указатель, а нативные узлы размещаются на стеках приостанавливаемых корутин, что исключает утечки нативной памяти и допускает перенос архитектуры на другие управляемые среды при наличии в их планировщике операций приостановки/возобновления корутины и доступа к текущей корутине. Роль условной переменной взяло на себя одноразовое событие Event с более простой семантикой. Новым является именно сочетание управляемой обёртки с интрузивными узлами на стеках корутин, обеспечивающее совместимость очереди ожидания со сборщиком мусора.

Для мьютекса предложена адаптация под короткие критические секции, сочетающая спин-фазу с экспоненциальной задержкой и адаптивный режим голодания с гарантией захвата не позднее чем через 16 пробуждений. Экспериментальное сравнение с sync.Mutex языка Go показало: среднее время захвата адаптивного мьютекса не превышает сотен наносекунд при числе корутин до 1000; накладные расходы в большинстве конфигураций не превышают 1.5x против 10–14x у sync.Mutex; при критической секции 100 мкс и 6000 корутин среднее время захвата составляет ~44 мкс против ~110 мкс у sync.Mutex, у которого оно практически совпадает с длительностью самой секции. Обратная сторона — рост максимальных пауз при сочетании длинных критических секций с высокой конкуренцией, до полутора порядков относительно sync.Mutex.

Достаточность выбранного базиса подтверждена реализацией высокоуровневых механизмов Promise<T> и AsyncLock, включая обнаружение взаимоблокировок по графу ожидания, — исключительно поверх Mutex и Event, без дополнительных низкоуровневых примитивов.

Библиография

[1] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," Computer, vol. 26, no. 7, pp. 18–41, Jul. 1993.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in Proc. 13th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), Seattle, WA, USA, 2008, pp. 329–339.

[3] OpenHarmony Project Documentation. [Online]. Available: https://gitee.com/openharmony/docs [дата обращения: 12.07.2026]

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[5] The Go Programming Language, "Package sync." [Online]. Available: https://pkg.go.dev/sync [дата обращения: 12.07.2026]

[6] D. Vyukov, "Scalable Go Scheduler Design Doc," 2012. [Online]. Available: https://golang.org/s/go11sched [дата обращения: 12.07.2026]

[7] R. Pressler and A. Bateman, "JEP 444: Virtual Threads," OpenJDK. [Online]. Available: https://openjdk.org/jeps/444 [дата обращения: 12.07.2026]

[8] "JEP 491: Synchronize Virtual Threads without Pinning," OpenJDK. [Online]. Available: https://openjdk.org/jeps/491 [дата обращения: 12.07.2026]

[9] O. Kowalke, "Boost.Fiber documentation." [Online]. Available: https://www.boost.org/doc/libs/release/libs/fiber/ [дата обращения: 12.07.2026]

[10] Microsoft, "Task-based asynchronous pattern (TAP) in .NET." [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap [дата обращения: 12.07.2026]

[11] R. Elizarov, M. Belyaev, M. Akhin, and I. Usmanov, "Kotlin coroutines: design and implementation," in Proc. 2021 ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2021), 2021, pp. 68–84.

[12] R. Nystrom, "What Color is Your Function?," 2015. [Online]. Available: https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/ [дата обращения: 12.07.2026]

[13] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in Linux," in Proc. Ottawa Linux Symposium, Ottawa, Canada, 2002, pp. 479–495.

[14] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in Proc. 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08), Tucson, AZ, USA, 2008, pp. 68–78.

[15] N. Koval, D. Khalanskiy, and D. Alistarh, "CQS: A formally-verified framework for fair and abortable synchronization," Proceedings of the ACM on Programming Languages, vol. 7, no. PLDI, art. 116, Jun. 2023.

[16] D. Lea, "The java.util.concurrent synchronizer framework," Science of Computer Programming, vol. 58, no. 3, pp. 293–309, Dec. 2005.

[17] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, San Jose, CA, USA, Tech. Rep. RJ 5118, 1986.

[18] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Transactions on Computer Systems, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[19] T. S. Craig, "Building FIFO and priority-queuing spin locks from atomic swap," Dept. of Computer Science, University of Washington, Seattle, WA, USA, Tech. Rep. UW-CSE-93-02-02, Feb. 1993.

[20] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in Proc. 8th Int. Symp. on Parallel Processing (IPPS), Cancún, Mexico, 1994, pp. 165–171.

[21] D. Vyukov, "Intrusive MPSC node-based queue," 1024cores. [Online]. Available: https://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue [дата обращения: 12.07.2026].

[22] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[23] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming. San Francisco, CA, USA: Morgan Kaufmann, 2008.

[24] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," ACM Computing Surveys, vol. 3, no. 2, pp. 67–78, Jun. 1971.

Design of Mutex and Event Synchronization Primitives for a Multithreaded Managed Runtime with Stackful Coroutines

I. A. Panferov

Abstract* — *Operating system synchronization primitives are unsuitable for managed runtimes with coroutines: acquiring an OS mutex blocks the entire carrier thread, which leads to deadlocks under cooperative multitasking. Existing coroutine-aware solutions are either tightly coupled to the internals of a specific runtime (Go, JVM), or target native environments without a garbage collector (Boost.Fiber), or allow waiting only in an asynchronous context (C#, Kotlin). This paper proposes a self-contained architecture of Mutex and Event synchronization primitives for the multithreaded managed ArkTS/OpenHarmony runtime with stackful coroutines. The primitives are managed-heap objects with built-in queues of waiting coroutines allowing the architecture to be ported to other managed runtimes whose scheduler provides coroutine suspend/resume operations. The one-shot Event replaces the condition variable with simpler semantics. The mutex is adapted to short critical sections with an exponential-backoff spin phase and an adaptive starvation mode guaranteeing acquisition within a bounded number of wakeups. Promise<T> and AsyncLock with wait-for-graph deadlock detection are built on top. On an 8-core arm64 device, the average acquisition time stays within hundreds of nanoseconds for up to 1000 coroutines, with overhead below 1.5x in most configurations versus 10–14x for Go's sync.Mutex. The gain comes at the cost of higher maximum (tail) latencies under long critical sections and high contention.

Keywords* — *ArkTS, coroutines, deadlock, managed runtime, mutex, OpenHarmony, synchronization primitives.

REFERENCES

[1] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," Computer, vol. 26, no. 7, pp. 18–41, Jul. 1993.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in Proc. 13th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), Seattle, WA, USA, 2008, pp. 329–339.

[3] OpenHarmony Project Documentation. [Online]. Available: https://gitee.com/openharmony/docs [Accessed: 12.07.2026]

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[5] The Go Programming Language, "Package sync." [Online]. Available: https://pkg.go.dev/sync [Accessed: 12.07.2026]

[6] D. Vyukov, "Scalable Go Scheduler Design Doc," 2012. [Online]. Available: https://golang.org/s/go11sched [Accessed: 12.07.2026]

[7] R. Pressler and A. Bateman, "JEP 444: Virtual Threads," OpenJDK. [Online]. Available: https://openjdk.org/jeps/444 [Accessed: 12.07.2026]

[8] "JEP 491: Synchronize Virtual Threads without Pinning," OpenJDK. [Online]. Available: https://openjdk.org/jeps/491 [Accessed: 12.07.2026]

[9] O. Kowalke, "Boost.Fiber documentation." [Online]. Available: https://www.boost.org/doc/libs/release/libs/fiber/ [Accessed: 12.07.2026]

[10] Microsoft, "Task-based asynchronous pattern (TAP) in .NET." [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap [Accessed: 12.07.2026]

[11] R. Elizarov, M. Belyaev, M. Akhin, and I. Usmanov, "Kotlin coroutines: design and implementation," in Proc. 2021 ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2021), 2021, pp. 68–84.

[12] R. Nystrom, "What Color is Your Function?," 2015. [Online]. Available: https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/ [Accessed: 12.07.2026]

[13] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in Linux," in Proc. Ottawa Linux Symposium, Ottawa, Canada, 2002, pp. 479–495.

[14] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in Proc. 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08), Tucson, AZ, USA, 2008, pp. 68–78.

[15] N. Koval, D. Khalanskiy, and D. Alistarh, "CQS: A formally-verified framework for fair and abortable synchronization," Proceedings of the ACM on Programming Languages, vol. 7, no. PLDI, art. 116, Jun. 2023.

[16] D. Lea, "The java.util.concurrent synchronizer framework," Science of Computer Programming, vol. 58, no. 3, pp. 293–309, Dec. 2005.

[17] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, San Jose, CA, USA, Tech. Rep. RJ 5118, 1986.

[18] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Transactions on Computer Systems, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[19] T. S. Craig, "Building FIFO and priority-queuing spin locks from atomic swap," Dept. of Computer Science, University of Washington, Seattle, WA, USA, Tech. Rep. UW-CSE-93-02-02, Feb. 1993.

[20] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in Proc. 8th Int. Symp. on Parallel Processing (IPPS), Cancún, Mexico, 1994, pp. 165–171.

[21] D. Vyukov, "Intrusive MPSC node-based queue," 1024cores. [Online]. Available: https://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue [Accessed: 12.07.2026].

[22] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[23] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming. San Francisco, CA, USA: Morgan Kaufmann, 2008.

[24] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," ACM Computing Surveys, vol. 3, no. 2, pp. 67–78, Jun. 1971.