САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Шаго Павел Евгеньевич

22.Б07-ПУ, 01.03.02 Прикладная математика и информатика

Планировщики 2

Научный руководитель

к.ф.-м.н., доцент Корхов В. В.

Санкт-Петербург

11 мая 2025 г.

Содержание

Введение

Данная работа продолжает исследования, связанные с планировщиками, и фокусируется на конкретных реализациях, архитектурных решениях и сравнительном анализе.

Несмотря на то что в предыдущей работе Планировщики [1] было показано, что программные планировщики позволяют решать широкий спектр задач самого разного характера, в настоящем исследовании основное внимание сосредоточено на планировщике операционной системы Linux — Completely Fair Scheduler (CFS)/Earliest Eligible Virtual Deadline First (EEVDF).

Выбор CFS/EEVDF обусловлен его высокой степенью развития по сравнению с альтернативами, а также эффективностью в решении задач планирования как отдельных процессов, так и их групп в рамках операционных систем (ОС) на базе ядра Linux. В контексте CFS/EEVDF подробно проанализированы его архитектура и принцип работы. Особое внимание уделено причинам, по которым в Linux произошла замена алгоритма выбора следующей задачи: вместо ранее использовавшегося, хорошо зарекомендовавшего себя подхода в CFS, описанного в [1], была внедрена модель виртуальных дедлайнов, реализованная и описанная в EEVDF.

Дополнительно была рассмотрена новейшая подсистема sched_ext (scheduler extensions), которая предоставляет интерфейсы для более точной адаптации ОС к специализированным сценариям использования – например, планирование с учётом перспективы интерфейса на экране пользователя. Такой механизм позволяет повысить эффективность решения задачи планирования за счёт использования дополнительной информации о контексте исполнения.

Исследование алгоритмов планирования остаётся критически важным направлением в области системного программирования, поскольку именно планировщик определяет эффективность использования ресурсов, отзывчивость и предсказуемость поведения операционной системы. Совершенствование планировщиков напрямую влияет на производительность вычислительных систем и на качество взаимодействия пользователя с программным обеспечением.

1 Earliest eligible virtual deadline first (EEVDF)

1.1 Введение

EEVDF — планировщик потоков в ядре Linux, который в 2023 году (с версии ядра Linux 6.6) заменил CFS в качестве планировщика по умолчанию [3].

CFS на тот момент использовался в ядре 16 лет, то есть ровно половину всего времени существования ядра Linux. Это решение поражает, казалось бы CFS за такой срок уже доведен до идеала, поэтому причина замены достаточно интересна.

На самом деле все не так драматично, потому что замены не произошло, случилось эволюционное улучшение CFS. EEVDF не является полностью новым планировщиком, а представляет собой модификацию существующего CFS: теперь именно EEVDF используется в качестве алгоритма выбора следующей задачи к выполнению внутри CFS.

Теоритическая основа планировщика EEVDF описана в статье 1996 года Earliest Eligible Virtual Deadline First : A Flexibleand Accurate Mechanism for Proportional Share Resource Allocation [4], написанной Ионом Стоикой и Хусейном Абдель-Вахабом. Сама же реализация EEVDF в Linux выполнена Инго Молнаром (автор CFS) и его коллегой по компании Red Hat Питером Зейлстрой. В основном реализация совпадает со статьей, хотя и несколько отличается, по теме отличий предлагается раздел 4 этой главы.

1.2 Причины отказа от механизма выбора CFS

Можно сформировать такие проблемы которые решает EEVDF:

1. CFS не имеет явного механизма задания требований к задержке. Как отмечает Джонатан Корбет [20], CFS не даёт процессам возможности выразить требования к задержке; его единственный механизм приоритизации – изменение распределения процессорного времени, а real-time классы, обеспечивающие низкую задержку, являются привилегированными и могут блокировать систему.

2. На практике у CFS может возникать старвация: если одна задача постоянно догоняет по *vruntime* других, она может долго не попадать на CPU. Особенно это заметно при экстремальных нагрузках: как указано в исследо-

вании [21], CFS при обслуживании огромного потока запросов показывает подвержен сильной старвации.

3. Непредсказуемость поведения. С точки зрения обеспечения гарантированной задержки или масштабирования, CFS ведёт себя сложно для анализа. Из-за описанных свойств время ожидания задач может сильно варьироваться в зависимости от числа параллельных задач и их веса. Так, как показано в [21], максимальное время голодания ограничено $O(n)$. Это означает, что в больших системах время отклика может расти без жёсткого контроля. Кроме того, из-за архитектуры и эмпирических параметров (granularity, sched_latency и др.) CFS фактически использует эвристики для повышения интерактивности, и их настройка сложна. Всё это делает поведение планировщика менее определённым: система плохо масштабируется по времени отклика и сложно предсказать худшие случаи задержки для критичных задач.

1.3 Алгоритм работы EEVDF

Данный раздел является кратким, но достаточно полным для задач этой работы описанием статьи [4] и устройства EEVDF.

Ключевое понятие EEVDF — виртуальное время $V(t)$, скорость роста которого обратно пропорциональна суммарному весу всех активных задач. Формально оно определяется интегралом $V(t) = \int_0^t \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau$, где w_j – вес (приоритет) задачи j, а $A(\tau)$ – множество активных (готовых) к выполнению задач в момент τ. Таким образом, при увеличении конкуренции (большей $\sum w_j$) виртуальное время замедляется.

В идеальной модели каждая задача i за отрезок $[t_1, t_2]$ должна получить $S_i = w_i\big(V(t_2) - V(t_1)\big)$ реального процессорного времени. На практике EEVDF отслеживает lag каждой задачи – насколько она отстала или перегнала свою долю: положительный lag означает, что задача должна ещё получить процессорное время, отрицательный – что она его получила сверх доли. При планировании каждой задаче выдается виртуальное время готовности (VE) и виртуальный дедлайн (VD). В момент поступления запроса на выполнение задаче присваивают $VE = V(t) - V(t_c)$, где t_c – текущее виртуальное время, а затем вычисляют виртуальный дедлайн по формуле $VD = VE + \frac{C}{w}$, где C – объём запрошенного процессорного времени, а w – вес задачи. Иными словами, $VD_i = VE_i + \frac{C_i}{w_i}$. Такая схема даёт более поздние дедлайны тяжёлым/высокоприоритетным задачам и более ранние – лёгким, что улучшает интерактивность.

Задача считается eligible к исполнению, если её $lag \geq 0$ (то есть она не опередила свою долю). Затем планировщик выбирает среди всех eligible задач ту, у которой виртуальный дедлайн минимален (правило earliest virtual deadline). Это гарантирует, что при постоянной нагрузке каждое приложение получит свою справедливую долю процессорного времени (с ограниченным lag) и не будет ожидать больше одного кванта сверх положенного. Выбор раннего дедлайна дополнительно придаёт приоритет коротким или приоритетным задачам, повышая отзывчивость системы.

Стоит отметить что EEVDF сам не управляет квантованием времени и не предлагается в качестве полноценного справедливого алгоритма планирования (англ. fair-scheduler). Вместо этого он чаще всего использует планировщик на уровень выше для этих целей (англ. supervisor).

Таким образом, EEVDF может использоваться как fair-scheduler в среде где существует supervisor fair-scheduler, а техника с виртуальными дедлайнами будет обеспечивать ограниченное отставание задач, улучшенную отзывчивость и лучшую обработку задач возникающих в системе с периодом. Алгоритм определяет порядок запуска задач на основе их виртуальных параметров, а непосредственное выполнение и переключение контекста поручается внешнему механизму.

1.4 Отличие реализации от статьи

Раздел основан на сообщениях [18]. В статье [4] виртуальное время считается по формуле $V(t) = \int_0^t \frac{1}{\sum_{j \in A(\tau)} w_j} d\tau$. EEVDF в Linux берет инвариант $\sum lag_i = 0$ и определение $lag_i = S - s_i = w_i \cdot (V - v_i)$, потом это уравнение решается относительно виртуального времени V. Это означает, что виртуальное время фактически рассчитывается как взвешенное среднее виртуальных времен задач.

В реализации интегрального варианта есть проблема: определение времени $t = 0$ в этом контексте довольно неоднозначно. Является ли это моментом загрузки системы, инициализации очереди задач или чем-то еще? Может ли оно быть установлено как любая произвольная постоянная, если только она будет до первого расчета виртуального времени? К сожалению, статья не отвечает на эти вопросы и опускает реализацию функции get_current_vt.

Фред Рохан в [18] предлагает реализовать версию с интегралом следующим образом: мы храним виртуальное время в переменной $virtual\_time$. Когда планировщик периодически запускается по прерываниям, мы обновляем виртуальное время по формуле: $virtual\_time \mathrel{+}= \frac{1}{\sum w_j} \cdot (t - t')$, где t – текущее

время, а t' – время последнего обновления переменной $virtual\_time$. Когда задача покидает/вступает в систему, мы обновляем виртуальное время по формуле: $virtual\_time = virtual\_time \pm \frac{lag_i(t)}{\sum w_j}$, но сумма весов уже новая (после покидания/вступления).

Для периодического обновления $virtual\_time$ (не для случая выхода и вступления) мы фактически разбиваем интеграл на две части: одну для интервала от 0 до t', и другую – от t' до t. Так как интеграл для интервала от 0 до t' уже вычислен, нам не нужно беспокоиться о изменениях суммы весов в этом интервале. Поскольку мы обновляем $virtual\_time$ и сумму весов каждый раз, когда задача покидает/вступает в систему, то t' – это последнее время, когда веса могли измениться до t. Таким образом, мы знаем, что сумма весов остается постоянной между t' и t.

Данное предложение было сформировано Фредом Роханом в виде электронного письма и отправлено разработчикам и мейнтейнерам планировщика Linux – Инго Молнару и Петеру Зейлстре, которые уже упоминались ранее в работе [1], а также в настоящем исследовании. Однако по неясным причинам оно осталось без ответа. Возможными причинами могли стать высокая загрузка получателей или чрезмерная сложность самого сообщения.

2 sched_ext

2.1 Введение

sched_ext (scheduler extensions) — это инфраструктура ядра Linux, позволяющая описывать планировщик с помощью набора BPF-программ (поэтому sched_ext также называют BPF scheduler). Также можно встретить альтернативное название — ECX (Extensible Scheduler Class).

sched_ext предоставляет возможность реализовывать собственные алгоритмы планирования, которые могут опираться на дополнительную информацию о планируемых сущностях. С помощью BPF-событий такие планировщики передают ядру необходимые состояния для управления задачами.

2.2 Основа sched_ext — BPF

BPF (Berkeley Packet Filter) — это специальный байт-код, исполняемый встроенной в ядро виртуальной машиной, который позволяет добавлять функциональность в ядро из пользовательского пространства. Изначально он был разработан для динамического добавления правил низкоуровневой фильтрации пакетов сетевыми картами внутри ядра Linux.

Со временем BPF значительно эволюционировал. Современная версия BPF является синонимом к **eBPF** (extended BPF), изначальный же вариант BPF сейчас называется cBPF (classic BPF) и практически не используется. В наименованиях существует некоторая асимметрия: в сообществе программистов чаще используется обозначение BPF [16, 17, 18], тогда как в научной литературе предпочитается термин eBPF [3, 5, 7, 8, 11, 12, 13, 14, 15].

BPF предоставляет ограниченный набор инструкций и спроектирован так, чтобы его можно было легко верфицировать, то есть чтобы до выполнения можно было быстро узнать нет ли в данном коде небезопасных последовательностей инструкций. BPF зародился в сообществе Linux, но сейчас также был портирован на Windows [19].

BPF может быть использован (помимо планировщика) для доработки различных систем ядра, некоторые из которых: tc, XDP, kprobes, uprobes, tracepoints, LSM, perf events, seccomp-bpf.

BPF считается в сообществе революционной технологией [3, 5, 8, 12], но есть также и критика [7].

2.3 Ускорение известных сценариев

Как уже упоминалось ранее, использование sched_ext позволяет вынести алгоритм планирования в пользовательское пространство. Это даёт возможность задействовать дополнительную информацию, недоступную на уровне ядра операционной системы. При этом реализовать простейший планировщик по принципу FIFO можно всего в 23 строках BPF-C кода, что подчёркивает доступность и гибкость данной инфраструктуры. Построение планировщика в sched_ext осуществляется через структуру sched_ext_ops, в которой описан необходимый интерфейс взаимодействия. Например, добавление задачи в очередь описывается с помощью переопределения функции enqueue, удаление – через dequeue. Для создания простейшего планировщика по принципу FIFO достаточно реализовать минимальный набор функций: enqueue, которая просто помещает задачу в очередь без дополнительной логики; dispatch, которая извлекает задачу из очереди и назначает её на выполнение на CPU; а также init, выполняющую инициализацию структуры и необходимых ресурсов.

В проекте **scx** [17] ведётся развитие подобных контекстно-оптимизирующих планировщиков. В документации представлены многочисленные сценарии, в которых такой подход оказывается полезным. Например, NUMA-aware планировщики, использующие знания о топологии вычислительной системы для более эффективного распределения задач; планировщики, ориентированные на пользовательскую перспективу, повышающие приоритет задач, отображаемых на экране; а также серверные решения, в которых, по метрике скорости ответа сервера на запрос, оказывается эффективнее распределять задачи между разными ядрами, а не потоками внутри одного ядра.

Преимуществом подхода sched_ext является также воспроизводимость: результаты, представленные в [17], достаточно просто смоделировать и верифицировать на собственной системе. Именно это и было выполнено в рамках подготовки данной научно-исследовательской работы.

Отдельно следует подчеркнуть, что одним из фундаментальных принципов sched_ext является обеспечение целостности системы: в случае критической ошибки расширяемый планировщик автоматически отключается, передавая управление обратно стандартному планировщику – CFS/EEVDF. Из этого принципа выводится и следующий принцип: sched_ext-планировщик может быть включён или отключён в любой момент времени.

Ещё одним важным преимуществом sched_ext выступает гибкость в реализации: алгоритмы, структуры данных и внутреннюю логику планировщика можно описывать на любом подходящем языке программирования – будь то Rust, Go или C++. В рамках проекта scx [17] уже создано значительное

количество примеров sched_ext-планировщиков. Примечательно, что большая часть из них реализована на языке Rust. Хотя BPF-часть по-прежнему должна быть реализована на bpf-c, она играет роль интерфейса, а не ядра логики. В коммерческих компаниях это может позволить упростить процесс разработки и передать задачу создания планировщика непосредственно команде, разрабатывающей основной продукт.

Далее следует обзор статьи Optimizing Linux scheduling based on global runqueue with SCX [3].

Авторы выделяют ключевую проблему, связанную с высокими накладными вычислительными расходами, возникающими при использовании планировщика CFS/EEVDF в процессе выбора следующей задачи (то есть в прямо в EEVDF). Это особенно критично в ситуациях, когда размер задачи крайне мал. В таких случаях возможно, что время, затрачиваемое на планирование, превышает время, необходимое для выполнения задачи. Также вышеописанная проблема возникает в ситуации с черезмерно частыми переключениями контекста.

Для решения этой проблемы была предложена облегчённая стратегия планирования SRAND, реализованная через sched_ext. Стратегия базируется на концепции глобальных и локальных очередей исполнения. Основой является пятиуровневая очередь (каждый уровень FIFO), отображаемая в BPF-карте, которая распределяет задачи по виртуальному времени исполнения. Задачи с меньшим виртуальным временем помещаются в глобальную очередь (также FIFO), их выполнение более приоритетно.

Если задача является потоковой задачей ядра (ОС), она напрямую назначается в локальную очередь ядра процессора. Во время извлечения обычной задачи из глобальной очереди локальной очередью происходит выбор ядра. В случае, если это ядро занято, выбирается другое свободное ядро, а задаче назначается временной квант в 20 мс, ядро активируется и задача передаётся на исполнение. Если во время последнего выполнения задача была вытеснена задачей с более высоким приоритетом, она возвращается в глобальную очередь, где повторно определяется подходящее ядро для её выполнения.

Дополнительно отслеживается состояние простаивающих ядер, что позволяет своевременно перераспределять задачи, повышая отзывчивость и снижая сложность планирования.

Реализация стратегии SRAND показала значительные улучшения по сравнению с планировщиком EEVDF: сокращение времени переключения контекста в среднем на 11,83% и улучшение общей производительности в стресс-тестах на 7,02%, при этом обеспечивается приемлемый уровень балансировки нагрузки.

2.4 За гранью оптимизации

Данный раздел представляет собой обзор статьи Concurrency Testing in the Linux Kernel via eBPF [5].

Планировщик, построенный с использованием sched_ext, может быть полезен не только для повышения производительности систем с заранее известной спецификой. В [5] рассматривается применение специализированных алгоритмов планирования типа Controlled Concurrency Testing (CCT) для проведения fuzzing-тестирования параллельного кода в ядре Linux.

Fuzzing-тестирование — это распространённый подход к тестированию программного решения путем отправления в него случайно сгенерированного входа. Такой подход позволяет значительно увеличить одну из главных метрик в тестировании программного обеспечения – покрытие (англ. coverage). Повышенное покрытие, в свою очередь, снижает вероятность ошибок в сложных и редко возникающих сценариях исполнения.

Подобные сценарии действительно существуют и нередко становятся причиной трудноуловимых сбоев. Их природа заключается в том, что некоторая последовательность команд исполняется в параллельной системе и лишь при определённом (достаточно маловероятном) порядке межпоточного взаимодействия возникает состояние, не предусмотренное разработчиком.

Авторы в [5] предлагают программное решение Lightweight Automated Concurrency-tester via EBPF (LACE) для проведения такого рода тестирования. Они выделяют четыре основные проблемы, характерные для существующих решений, которые призван устранить LACE: ограниченный контроль над процессом планирования, высокие накладные расходы, отсутствие совместимости с будущими версиями, слабая расширяемость.

Ограниченность контроля проявляется в том, что во многих тестовых пакетах точки планирования могут быть реализованы лишь в определённых местах, а управление исполнением часто сводится к вставке искусственных задержек. Такой подход не всегда обеспечивает достаточную точность для выявления всех потенциальных ошибок в параллельном исполнении.

Высокие накладные расходы связаны с тем, что некоторые существующие пакеты тестирования требуют запуска в режиме гипервизора, что значительно усложняет и утяжеляет процесс тестирования.

Отсутствие совместимости с будущими версиями ядра у существующих решений обусловлено их изначальной архитектурой, плохо адаптированной к быстрому темпу развития Linux. Эта проблема выражается в необходимости внесения масштабных изменений в инструменты тестирования при обновлении ядра. Например, для адаптации инструмента krace к современным

версиям требуется применение патчей объёмом более 10000 строк, затрагивающих 105 файлов. Подобные объёмы изменений значительно усложняют сопровождение, усложняют автоматизацию и снижают практическую применимость таких решений в долгосрочной перспективе.

Ограниченная расширяемость объясняется тем, что алгоритмы планирования в существующих системах часто являются уникальными и жёстко встроены в ключевые компоненты своей реализации. Попытки расширить или привнести новые алгоритмы в кодовую базу оказываются крайне сложными и требуют глубокой экспертной подготовки в области низкоуровневого планирования. В результате на практике до сих пор широко применяются традиционные фреймворки, полагающиеся на поведение планировщика по умолчанию, несмотря на их ограниченную эффективность в выявлении ошибок исполнения.

LACE значительно смягчает или полностью решает эти проблемы, он предлагает собственный CCT планировщик написанный на eBPF в инфраструктуре sched_ext, а также точечную стратегию внедрения прерываний в код ядра. Кроме того, в LACE реализован двухфазный подход, при котором сначала происходит тестирование набором последовательных команд, а затем производится анализ межпоточных ошибок.

Если оценить решение на актуальных версиях ядра Linux, то можно получить следующие результаты: LACE обеспечивает на 38% больше охвата ветвлений по сравнению с SegFuzz (фреймворк с наибольшим покрытием), демонстрирует высокую эффективность в исследовании вариантов планирования, обеспечивая ускорение в 11,4 раза. Кроме того, с его помощью было обнаружено восемь новых ошибок в ядре, четыре из которых были исправлены, а ещё две подтверждены разработчиками.

Значительным преимуществом использования sched_ext в таком сценарии является то, что планировщик может быть загружен однократно во время подготовки к тестированию всего за несколько секунд, без необходимости перезагрузки системы или перекомпиляции ядра.

В последующих разделах статьи рассматриваются используемые алгоритмы, а также приводится детальное сравнение LACE с альтернативными решениями. Однако это выходит за рамки общего обзора и в данной работе описываться не будет. Цель этого раздела — подчеркнуть, что подход, основанный на sched_ext, применим не только для оптимизации планирования под конкретные сценарии, но и способен решать более широкие задачи.

Заключение

В данной работе были рассмотрены продвинутые алгортмы планирования. Были описаны наиболее актуальные решения, используемые для задач планирования в современных операционных системах, такие как CFS/EEVDF и sched_ext. Продолжающееся развитие данной области показывает, что решаемые проблемы актуальны и востребованны.

В работе преднамеренно было опущено обсуждение важности разработки более эффективных планировщиков в тех местах, где это допустимо в рамках формата отчета о научно-исследовательской работе. Это сделано по следующим причинам: во-первых, с момента публикации [1] не произошло качественно новых изменений в оценке значимости рассматриваемой темы – она остаётся исключительно актуальной, и повторение ранее изложенного не представляется целесообразным; во-вторых, данную работу лучше всего рассматривать как логичное продолжение ранее проведённого исследования, поэтому внимание к важности целесообразно уделить именно в заключении.

Разработка более эффективных алгоритмов планирования имеет ключевое значение в условиях постоянно растущих требований к производительности вычислительных систем. Современные задачи, такие как обеспечение предсказуемой задержки, адаптация к разнородным нагрузкам, энергоэффективность, а также гетерогенное устройство наиболее современных вычислительных машин требуют всё более точных и адаптивных решений. Поэтому исследования в этой области способствуют не только улучшению существующих реализаций, но и открывают путь к созданию новых, более универсальных и надёжных планировщиков.

Таким образом, проведённое исследование дополняет и расширяет картину, описанную в [1]. Оно укрепляет базу, необходимую для перехода на прикладной уровень: разработки более эффективных алгоритмов и создания программных решений, лучше соответствующих специфике решаемых задач. В будущем планируется сосредоточиться на подготовке образа тестовой системы с ядром Linux и написании пакета для сбора метрик качества обработки задач планирования. В дальнейшем работа в рамках этой темы может пойти в различных направлениях: от внесения инкрементальных патчей в CFS/EEVDF на основе показаний тест-пакета до создания специализированных sched_ext планировщиков, построенных на продвинутых алгоритмах для разнообразных прикладных задач

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Павел Е. Шаго Отчет о научно-исследовательской работе: Планировщики б. и. 2024

[2] Joseph Y-T. Leung Handbook of Scheduling: Algorithms, Models, and Performance Analysis CRC Press 2004

[3] Tang Qinan, Gao Xing, Li Guilin, Lin Juncong Optimizing Linux scheduling based on global runqueue with SCX IEEE International Conference SMC 2024

[4] Ion Stoica, Hussein Abdel-Wahab Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation Old Dominion University 1996

[5] Jiacheng Xu, Dylan Wolff, Han Xing Yi, Jialin Li, Abhik Roychoudhury Concurrency Testing in the Linux Kernel via eBPF arXiv 2025

[6] Andrew S. Tanenbaum, Herbert Bos MODERN OPERATING SYSTEMS Pearson Education 2023

[7] Zhong, Shawn Wanxiang, Jing Liu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau Revealing the Unstable Foundations of eBPF-Based Kernel Extensions Twentieth European Conference on Computer Systems 2025

[8] Liz Rice Learning eBPF O'Reilly Media, Inc 2023

[9] William Stallings Operating Systems: Internals and Design Principles Pearson Education 2020

[10] Michael J. Morrison Resource Management and Scheduling in Multitasking Operating Systems CRC Press 2017

[11] Daniel Hodges Scheduling at Scale : eBPF Schedulers with Sched_ext USENIX Association 2024

[12] Mores Konstantinos User-space guided memory management with eBPF NTUA 2025

[13] Xiaobo Zheng, Zihao Duan, Shiyi Li, Haojun Hu, Wen Xia F4: Fast, Flexible and stable-Fortified User-Space Thread Scheduling Framework International Conference on Networking, Architecture and Storage IEEE 2024

[14] Saito Shogo, Kei Fujimoto Port contention aware task scheduling for SIMD applications IEEE Access 2024

[15] Feitong Qiao, Yiming Fang, Asaf Cidon Energy-Aware Process Scheduling in Linux ACM SIGENERGY Energy Informatics 2024

[16] Torvalds L. Linux Kernel [Электронный ресурс] / L. Torvalds. – Режим доступа: https://github.com/torvalds/linux – Дата обращения 15.04.2025.

[17] sched-ext/scx [Электронный ресурс]. – Режим доступа: https://github.com/sched-ext/scx – Дата обращения 27.04.2025.

[18] lkml [Электронный ресурс]. – Режим доступа: https://lkml.org – Дата обращения 20.04.2025.

[19] eBPF for windows [Электронный ресурс]. – Режим доступа: https://github.com/microsoft/ebpf-for-windows – Дата обращения 01.05.2025.

[20] lwn article 925371 [Электронный ресурс]. – Режим доступа: https://lwn.net/Articles/925371 – Дата обращения 06.05.2025.

[21] Sungju Huh, Jonghun Yoo, Seongsoo Hong Analytical Evaluation of Linux CFS Scheduler under Extreme Workload ITC-CSCC 2011