САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Шаго Павел Евгеньевич

22.Б07-ПУ, 01.03.02 Прикладная математика и информатика

Планировщики 3

Научный руководитель

к.ф.-м.н., доцент Корхов В. В.

Санкт-Петербург

16 декабря 2025 г.

Содержание

Введение

Данная работа продолжает исследования [1] и [2], связанные с планировщиками процессов в операционных системах.

В [2] были рассмотрены конкретные реализации в ядре Linux, такие как Completely Fair Scheduler (CFS) и Earliest Eligible Virtual Deadline First (EEVDF), также был кратко описан фреймворк sched_ext.

Несмотря на то, что алгоритм EEVDF считается высокопроизводительным и эффективным решением предыдущие работы продемонстрировали, что обобщенный подход не всегда является хорошим решением.

Настоящее исследование расширяет экспертизу в работе фреймворка sched_ext. Реализован алгоритм EEVDF с помощью sched_ext, проведено сравнение с эталонной реализацией в ядре Linux. Поставлена задача улучшения алгоритма планирования при условии что исполнение задач производится на процессоре с гетерогенной архитектурой (Intel Hybrid CPU, ARM big.LITTLE).

Фреймворк sched_ext построен так, что позволяет практически без косвенных затрат загружать и выгружать новые алгоритмы во время исполнения (работы) системы, то поиск и реализация повышающих эффективность использования ресурсов процессора алгоритмов планирования представляется очень перспективным.

Современные процессорные архитектуры все чаще проектируются на основе гетерогенного принципа, при котором в рамках одного кристалла объединяются вычислительные ядра, различающиеся по уровню производительности, энергопотреблению и микроархитектурным особенностям. Такой подход позволяет более гибко распределять вычислительные нагрузки между ядрами, оптимизируя использование ресурсов в зависимости от характера выполняемых задач.

Высокопроизводительные ядра ориентированы на обработку ресурсоемких и чувствительных к задержкам вычислений, тогда как энергоэффективные ядра предназначены для фоновых, периодических или менее требовательных операций.

Разработка эффективных планировщиков для гетерогенных архитектур остается актуальной задачей в системном программировании, поскольку напрямую влияет на производительность, энергопотребление и отзывчивость систем. Улучшения в этой области позволяют оптимизировать использование ресурсов в современных вычислительных платформах.

1 Фреймворк sched_ext

1.1 Введение

Инфраструктура **sched_ext** предоставляет возможности для разработки и исполнения планировщиков процессов на базе набора eBPF программ исполняемых в виртуальной машине ядра Linux.

sched_ext позволет динамически подключать альтернативные алгоритмы планирования без необходимости модификации и перекомпиляции ядра.

Более того, даже перезагружать систему нет необходимости, поскольку для загрузки нового алгоритма sched_ext просто делает системный вызов bpf(), далее как и любая eBPF программа алгоритм планирования пройдет верификацию, JIT компиляцию и сразу начнет работу в виртуальной машине.

Инфраструктура sched_ext встроена в существующую подсистему планирования Linux и использует стандартные структуры задач (struct task) и очереди runqueue.

При активации sched_ext управление выбором следующей задачи происходит из общей (для ядер процессора) очереди dsq (dispatch queue) задачи в которую помещает алгоритм планирования. При этом предоставляется возможность модификации алгоритма перехода задач из общей очереди в локальные dsq (локальные очереди исполнения dsq – очереди задач для каждого отдельного ядра процессора).

1.2 Целостность

Ядро Linux сохраняет контроль над важными аспектами выполнения, включая:

- переключение контекста
- управление состояниями задач
- обработку прерываний и таймеров
- обеспечение изоляции

Благодаря этому sched_ext реализует модель управляемого расширения, в которой пользовательский (динамически подгружаемый) код не может напрямую нарушить целостность ядра.

1.3 Архитектура

На sched_ext распостраняются архитектурные особенности eBPF, (sched_ext реализован через eBPF, sched_ext планировщик является eBPF программой).

eBPF байт-код (скомпилированная eBPF программа) всегда проходит верификацию перед тем как он будет отправлен в виртуальную машину или в JIT компилятор (предоставляется возможность не использовать JIT компилятор вовсе, скомпилированный байт-код после верификации сразу будет исполняться в виртуальной машине).

Верификация предполагает такие шаги как:

- проверка на завершимость

  (проверяется что во всех возможных путях графа потока управления (CFG, control flow graph) программа завершается и не останется в неопределенном состоянии, бесконечном цикле)

- запрет небезопасных операций с памятью

  (доступ к динамической памяти предоставляется через вспомогательные функции которые работают с памятью владение которой заранее определены за этой eBPF программой (механизм eBPF Maps)).

Эти свойства обеспечивают предсказуемость и стабильность работы реализованного через sched_ext планировщика, при этом даже если какие-то ошибки с eBPF программой все таки будут обнаружены во время исполнения произойдет откат (fallback) на стандартный планировщик EEVDF.

Основные функции программного интерфейса sched_ext:

- select_cpu() – выбор потока исполнения (физического или логического ядра) процессора
- enqueue() – вставка задачи в глобальную очередь dsq
- dispatch() – перемещение задачи в локальную dsq

Очистка очередей (постановка задач на исполнение из очередей) происходит автоматически. Каждый тик системного таймера $1/HZ$ проверяется возможность исполнять задачу дальше (HZ – частота системного таймера). Исполняемая задача может быть приостановлена прерыванием сколь угодно раз.

Задача снимается с выполнения из-за обнуления ее scx_slice, это происходит автоматически когда заканчивается выделенная задаче квота, также есть возможность в алгоритме планирования обнулить (специальным прерыванием) задаче scx_slice (таким образом получаем вытеснение планировщиком).

Как только переменная scx_slice обнулена ядро считается доступным и происходит выбор следующей задачи из dsq.

2 Обзор литературы

Данная глава представляет собой обзор новых довольно интересных статей опубликованных со времени написания [2].

2.1 Linux Kernel Scheduler Evaluation for Performance-Critical Telecom Workloads

В работе рассматривается широкое внедрение облачных технологий в телекоммуникационном секторе (telco), что требует систем, способных обеспечивать низкие задержки. Основной проблемой, с которой сталкивается Ericsson в облачных средах на базе Kubernetes, является возникновение высоких задержек, когда загрузка процессора превышает 50%. Это делает оценку производительности планировщика LAVD в диапазоне нагрузок от 50% до 95% необходимой практически (на практике) задачей, чтобы обеспечить функционирование telco-приложений в соответствии с их назначением.

Центральной целью диссертации является оценка эффективности планировщика Latency-criticality Aware Virtual Deadline (LAVD) при обработке критичных к производительности телекоммуникационных рабочих нагрузок. LAVD, первоначально разработанный для игровых рабочих нагрузок, которые имеют сходство с telco-нагрузками по требованиям к задержке и ресурсам, представляет собой перспективную попытку решения проблем с задержками. Планировщик LAVD был реализован с использованием новой инфраструктуры sched_ext в ядре Linux.

Для проведения оценки был разработан специальный симулятор. Этот настраиваемый, многопоточный симулятор, написанный на C++, генерирует синтетические рабочие нагрузки, имитирующие характеристики трафика telco-систем, основанные на статистическом анализе данных трассировки Ericsson. Для моделирования времени выполнения (burst times) реальных рабочих нагрузок трафика использовался статистический анализ, который показал, что распределение лучше всего описывается бета-распределением, что дало наибольшее значение p-value (0.084) при проверке критерием Колмогорова-Смирнова. Симулятор также поддерживает генерацию фоновых интерферирующих рабочих нагрузок, которые конкурируют за время ЦП, позволяя моделировать реалистичные сценарии конфликта.

В ходе исследования сравнивались три планировщика: LAVD, scx_simple (еще один планировщик sched_ext) и EEVDF, который является стандарт-

ным планировщиком ядра Linux и служил базовой единицей для сравнения (baseline) производительности. EEVDF подробно описан в [2]. LAVD, как планировщик, ориентированный на критичность задержки, использует сложную модель, основанную на времени выполнения, частоте пробуждения и частоте ожидания, чтобы назначать более ранние виртуальные дедлайны (Virtual Deadlines) более критичным задачам. В отличие от них, scx_simple – это минималистичный планировщик, который по умолчанию использует взвешенное виртуальное время, но не поддерживает перехват (preemption), что отличает его от EEVDF и LAVD.

Результаты показали, что LAVD имеет потенциал для значительного сокращения среднего времени оборота (mean turnaround time), особенно в сценариях с высокой загрузкой процессора и наличием интерферирующих рабочих нагрузок. В среде Kubernetes Minikube LAVD продемонстрировал улучшение среднего времени оборота по сравнению с EEVDF в диапазоне от 6% до 81%, а улучшение 99-го перцентиля (tail latency) достигало 90%. Однако, scx_simple показал удивительно хорошие результаты, превзойдя LAVD, когда количество интерферирующих рабочих нагрузок было низким. Например, в условиях, типичных для Ericsson (50% загрузки ЦП трафиком и 10% интерферирующей нагрузкой), scx_simple постоянно превосходил LAVD.

Ключевой вывод исследования состоит в том, что превосходство не принадлежит какому-то одному конкретному планировщику, а скорее демонстрируется потенциалом всей инфраструктуры sched_ext. Эта гибкость, позволяющая быстро проводить эксперименты и оптимизировать код, делает ее перспективным инструментом для адаптации планирования ЦП к требованиям, специфичным для домена telco. Кроме того, попытки оптимизировать параметры LAVD, такие как минимальный и максимальный временной срез, не привели к значительным улучшениям по сравнению с настройками по умолчанию, что позволяет предположить, что LAVD уже хорошо оптимизирован для широкого спектра сценариев.

Среди направлений для будущей работы предлагается улучшение симулятора за счет использования более широкого и разнообразного набора трасс или непосредственного использования трасс для управления выполнением вместо статистических распределений. Также рекомендуется расширить анализ, включив дополнительные метрики, такие как время ожидания и количество переключений контекста, а также провести оценку планировщиков в распределенном кластере Kubernetes, чтобы выявить потенциальные узкие места, которые могут не проявляться в локальной среде Minikube.

2.2 Rethinking Provenance Completeness with a Learning-Based Linux Scheduler

Проблема обеспечения полноты данных о происхождении (provenance) критически важна для трассируемости действий системы, что необходимо для анализа первопричин угроз безопасности и их последствий. Системы сбора данных о происхождении должны функционировать как эталонный монитор (reference monitor), гарантируя полный захват всех системных событий, чтобы ведение журнала нельзя было обойти. Однако, современные системы сбора данных сталкиваются с серьезной проблемой, известной как «угроза суперпродюсера» (super producer threat), при которой чрезмерная генерация событий может перегрузить систему, вынуждая ее отбрасывать критически важные для безопасности записи, что позволяет злоумышленнику скрыть свои действия. Эта угроза представляет значительный вызов гарантиям безопасности эталонного монитора.

Существующие решения и подходы сталкиваются с серьезными ограничениями при противодействии угрозе суперпродюсера. Традиционные планировщики Linux, такие как CFS и EEVDF, разработаны для достижения общих целей производительности, таких как пропускная способность, справедливость и задержка, но им не хватает понимания уникальных требований полноты данных о происхождении. Эта нехватка приводит к потере данных, что компрометирует надежность и безопасность системы. Некоторые современные защитные механизмы, например NoDrop, предлагают изоляцию ресурсов, но их внедрение затруднительно, поскольку требует специализированной аппаратной поддержки (например, Intel Memory Protection Keys), тесно связано с устаревшими ядрами и может вызывать ошибки, связанные с атомарными операциями, потенциально приводящие к сбоям ядра (kernel panics).

В качестве инновационного решения этих проблем представлена система Aegis – самообучающийся планировщик для ядра Linux, специально разработанный для обеспечения полноты данных о происхождении. Основная идея заключается в том, что планировщик ядра может решить проблему несвоевременного потребления генерируемых событий, которое приводит к переполнению буфера и потере данных, путем выделения достаточных ресурсов системе происхождения. Разработка Aegis преследует три основные цели: полнота (захват всех событий даже при экстремальных нагрузках), эффективность (поддержание или улучшение производительности системы) и справедливость (сбалансированное распределение ресурсов для предотвращения "голодания"задач).

Архитектура Aegis основана на двухслойном подходе, где первый слой созда-

ет основу планирования, а второй слой настраивает ее с помощью обучения для оптимизации производительности и смягчения атак. Планировщик использует основанную на очередях (queue-based) структуру, где задачи делятся на основную и несколько непервичных очередей. Непервичные очереди, предназначенные для задач, связанных с происхождением, имеют заданное время ожидания, которое выступает в качестве естественного бюджета ресурсов. Очередь становится доступной для обработки только тогда, когда время, прошедшее с момента последнего выполнения задачи в ней, превышает установленное время ожидания, что обеспечивает строгую политику выделения ресурсов и предотвращает перегрузку. Aegis также обеспечивает справедливость, потребляя наиболее «голодные» непервичные очереди (те, которые ждали дольше всего), гарантируя пропорциональное распределение ресурсов и предотвращая "голодание"задач.

Ключевым элементом Aegis является использование обучения с подкреплением (Reinforcement Learning, RL), реализуемого с помощью легковесной DeepQ-Network (DQN). DQN прогнозирует оптимальные решения о планировании, основываясь на контексте задачи, который включает как поведенческие характеристики задачи, так и признаки состояния системы происхождения (например, скорость генерации событий, доступность буфера).

Процесс обучения управляется двойным механизмом вознаграждения: вознаграждение за происхождение (r_c) наказывает за потерю событий для обеспечения полноты, а вознаграждение за утилизацию (r_p) минимизирует простой ЦП для повышения общей эффективности. Aegis реализован в пространстве ядра Linux с использованием подсистемы eBPF и фреймворка sched_ext.

В результате оценки Aegis с использованием различных макробенчмарков и двух систем происхождения (Sysdig и eAudit) была подтверждена его эффективность. Aegis последовательно демонстрировал нулевую потерю событий во всех тестовых сценариях, даже в условиях высоких нагрузок, где другие планировщики, такие как EEVDF, LAVD и Rusty, теряли от 39% до более 98% событий. Более того, Aegis не только предотвращает потерю событий, но и обеспечивает высокую производительность, в среднем улучшая время выполнения по сравнению с EEVDF, LAVD и Rusty. Общие вычислительные затраты, связанные с планированием (включая вывод нейронной сети), остаются низкими: в типичных сценариях они составляют менее 0.5%, а в самых требовательных – около 2.44%, благодаря использованию дельта-функции для пропуска ненужных решений планирования.

Таким образом, Aegis обеспечивает выполнение гарантий безопасности эталонного монитора для систем сбора данных о происхождении, эффективно предотвращая угрозу суперпродюсера путем интеграции обучения с под-

креплением в планировщик ядра. Aegis доказывает, что адаптивный, управляемый данными подход является мощным и производительным способом устранения критических проблем, с которыми не справляются традиционные и аппаратные механизмы защиты.

2.3 Mixture-of-Schedulers: An Adaptive Scheduling Agent as a Learned Router for Expert Policies

Современные планировщики операционных систем сталкиваются с серьезной проблемой: единая, статичная политика не может обеспечить оптимальную производительность в условиях разнообразия и динамичности нагрузок, характерных для современных систем. Рост гетерогенного оборудования (например, P/E-ядер) и разнообразных архитектур приложений (микросервисы, интерактивные задачи, пакетная обработка) обострил фундаментальные компромиссы между справедливостью, пропускной способностью и задержкой. В ответ на эту проблему, статья предлагает новую парадигму: динамический выбор оптимальной политики из портфеля специализированных планировщиков вместо проектирования единого, монолитного решения.

Эта философия реализована в виде Адаптивного Агента Планирования (Adaptive Scheduling Agent, ASA), который представляет собой легковесную структуру, выступающую в роли интеллектуального маршрутизатора для "экспертных"политик планирования. ASA разработан на основе расширяемого интерфейса планировщика Linux, sched_ext. Архитектура ASA декомпозирует сложную задачу планирования на две более управляемые подзадачи: распознавание шаблона рабочей нагрузки и сопоставление политики. Это позволяет ASA интеллектуально подбирать наиболее подходящую политику из своего портфеля экспертов во время выполнения.

В основе интеллекта ASA лежит цикл восприятия, решения и действия. Модуль Восприятия (Perception module) непрерывно отслеживает метрики времени выполнения, такие как использование CPU, активность I/O и сетевой трафик, собирая данные из различных источников, включая программы eBPF и файловую систему procfs. Модуль Решения (Decision module) использует обученную модель машинного обучения (классификатор на основе ансамбля, преимущественно XGBoost) для распознавания текущего шаблона нагрузки. Для обеспечения стабильности решений в динамической среде применяется алгоритм Взвешенного по Времени Вероятностного Голосования (Time-Weighted Probability Voting), который использует механизм голосования с экспоненциальным затуханием для фильтрации кратковременного системного шума. Модуль Действия (Action module) затем выполняет ди-

намическое переключение на выбранный оптимальный планировщик через фреймворк sched_ext.

Развертывание ASA включает комплексный трехэтапный автономный процесс подготовки (Offline Prepare Pipeline), который систематически создает надежного агента планирования. Он начинается с Прототипного Обучения (Prototype Learning) для создания базовой модели распознавания и первичного сопоставления планировщиков, за которым следует Динамическая Калибровка Накладных Расходов (Dynamic Overhead Calibration), имитирующая стоимость переключения планировщиков. Завершающий этап, Обучение Модели Обобщения (Generalization Model Training), позволяет ASA уточнить свои модели в реальной среде. Для оценки эффективности авторы разработали Критерий Оценки, Ориентированный на Пользовательский Опыт (User Experience Oriented Evaluation Criteria), который измеряет производительность на основе метрик, напрямую влияющих на восприятие пользователя (например, задержка взаимодействия и плавность UI), смещая цель оптимизации от системной эффективности к оптимизации пользовательского опыта. Этот критерий отражает нелинейную природу человеческого восприятия, выделяя "Зону Наилучшего Восприятия"("Sweet Spot Region").

Комплексная оценка подтвердила, что парадигма динамического выбора ASA превосходит статическую оптимизацию. ASA стабильно превосходит стандартный планировщик Linux (EEVDF), показывая лучшие результаты в 86.4% тестовых сценариев. Выбранные ASA политики являются близкими к оптимальным, входя в тройку лучших планировщиков в 78.9% всех сценариев. В некоторых сценариях со смешанными нагрузками ASA даже превосходит "Статический Оракул динамически переключая политики внутри задачи для отслеживания мгновенного оптимума.

Агент также продемонстрировал сильную способность к обобщению, поддерживая или даже улучшая производительность на новых, ранее невиденных аппаратных платформах.

Практичность ASA подтверждается минимальными накладными расходами: агент потребляет в среднем всего 1.45% одного ядра CPU и 297 МБ памяти. Хотя потолок производительности ASA определяется качеством и разнообразием портфеля экспертных планировщиков, это также является преимуществом, поскольку позволяет масштабировать улучшения путем интеграции новых, более продвинутых планировщиков, разработанных сообществом. В целом, Адаптивный Агент Планирования предлагает жизнеспособный технический путь для создания следующего поколения интеллектуальных, адаптивных планировщиков операционных систем.

3 Реализация алгоритма EEVDF через фреймворк sched_ext

3.1 Задачи

- Исследовать диссертацию [7]
- Реализовать в соответствии с описанным алгоритмом планировщик EEVDF используя sched_ext
- Провести сравнение задержки между алгоритмами

3.2 Мотивация

Выбор данной задачи обусловлен несколькими факторами и открывает важные направления для дальнейших исследований.

Реализация планировщика EEVDF в рамках инфраструктуры sched_ext создает возможность провести количественную оценку косвенных затрат, возникающих при использовании данной конфигурации, что необходимо для абсолютной оценки эффективности получившегося практического решения.

Кроме того, разработанная реализация служит воспроизводимой основой для будущего прямого сравнения альтернативной реализации планировщика (специально разработанного для гетерогенного процессора) и позволяет выявлять практические ограничения.

Наконец, работа приносит автору существенный практический опыт разработки и интеграции планировщиков в системное окружение, что важно для последующих экспериментальных и прикладных исследований.

3.3 Краткое погружение в реализацию

В основе алгоритма EEVDF лежит присвоение каждой задаче виртуального допустимого времени (VE) и виртуального крайнего срока (VD), вычисляемых следующим образом:

VE = max(VE_previous, V_now - slice), где V_now – глобальное виртуальное время, а slice – временая квота задачи (упомянутый ранее scx_slice).

VD = VE + (slice / weight, масштабируется таким образом, чтобы задачи с более высоким весом (более высоким приоритетом) получали более короткие эффективные сроки.

Задачи распределяются на основе самого раннего VD, что способствует справедливости: более ресурсоемкие задачи потребляют больше виртуального времени, но соответственно наказываются. Это предотвращает бесконечную задержку задач с низким приоритетом, что является распространенной проблемой в других алгоритмах справедливого распределения.

В моей реализации я поддерживаю глобальный контекст (eevdf_ctx), отслеживающий vtime_now (текущее виртуальное время) и total_weight (сумма весов всех задач). Общая очередь диспетчеризации dsq хранит задачи, упорядоченные по их виртуальному времени VD, используя встроенную в sched_ext функцию вставки с упорядочением по виртуальному времени (scx_bpf_dsq_insert_vtime()).

3.4 Подробности реализации

Инициализация и глобальное состояние: Мы начинаем с определения карты BPF для глобальных данных (global_data) для хранения контекста EEVDF. Карта статистики для каждого ЦП отслеживает такие метрики, как выбор незадействованных ЦП и добавление задач в очередь для отладки. Общая DSQ (ID 0) создается во время инициализации (eevdf_init), обеспечивая единую упорядоченную очередь для всех задач. Выбор ЦП (eevdf_select_cpu): Когда задача просыпается, мы используем выбор ЦП по умолчанию из sched_ext, но проверяем наличие незадействованных ядер. Если обнаруживается свободный процессор, мы увеличиваем статистику и добавляем задачу в локальную очередь задач (DSQ) с использованием среза по умолчанию — оптимизируя выполнение для немедленного запуска и уменьшая накладные расходы на миграцию. Постановка задач в очередь (eevdf_enqueue): Здесь происходит волшебство EEVDF. Мы получаем глобальный контекст, ограничиваем виртуальное допустимое время (VE) задачи, чтобы предотвратить отрицательные задержки, и вычисляем виртуальный крайний срок (VD) как VE + (срез * SCALE / вес). Задача добавляется в общую очередь задач (DSQ) с использованием семантики, упорядоченной по виртуальному времени (scx_bpf_dsq_insert_vtime), обеспечивая порядок «сначала самый ранний крайний срок». Коэффициент SCALE (100) обеспечивает точность с фиксированной запятой для дробных вычислений.

Диспетчеризация задач (eevdf_dispatch): Здесь царит простота: мы перемещаем задачи из общей очереди DSQ в локальную очередь на текущем ЦП, позволяя ядру выбирать следующую готовую к выполнению задачу на основе предварительно упорядоченного виртуального времени (VD). Хуки жизненного цикла задачи: Начало выполнения (eevdf_running): продвижение глобального виртуального времени, если виртуальное время задачи пре-

вышает его, обеспечивая монотонный прогресс. Остановка (eevdf_stopping): обновление виртуального времени задачи путем добавления потребленного времени, масштабированного по весу, с учетом фактического времени выполнения. Включение/отключение (eevdf_enable/eevdf_disable): корректировка total_weight при входе или выходе задач из планировщика, поддерживая точное глобальное состояние. Изменение веса (eevdf_set_weight): динамическая корректировка виртуального времени для сохранения задержки (разницы между глобальным виртуальным временем и виртуальным временем задачи) при изменениях веса. Это включает пропорциональное масштабирование во избежание несправедливых скачков приоритета.

Код избегает сложных структур данных, полагаясь на примитивы DSQ из sched_ext для повышения эффективности. Обработка ошибок минимальна, с ранним возвратом при сбоях поиска в карте, что обеспечивает приоритет надежности в контексте ядра. Тестирование и проверка На практике этот планировщик «просто работает» — задачам справедливо выделяется процессорное время пропорционально их весам, с гарантией низкой задержки для интерактивных рабочих нагрузок. Счетчики статистики показывают эффективное использование процессора в режиме ожидания и закономерности постановки задач в очередь. Например, при смешанных нагрузках (например, задачи, интенсивно использующие процессор, и задачи, интенсивно использующие ввод-вывод) EEVDF превосходит CFS по показателям задержки в конце очереди, поскольку виртуальные крайние сроки предотвращают чрезмерное использование ресурсов.

Будущие улучшения и соображения Хотя этот прототип отражает суть EEVDF, существует множество возможностей для его усовершенствования: виртуальное время для каждого ЦП с учетом NUMA-архитектуры, адаптивное разделение на сегменты для переменных рабочих нагрузок или интеграция с энергоэффективным планированием. Гибкость sched_ext делает эти расширения простыми. В заключение, реализация EEVDF через sched_ext демонстрирует возможности BPF в расширяемости ядра — превращая теоретические алгоритмы в развертываемые решения с минимальными накладными расходами. Эта работа не только решает исходную задачу, но и открывает двери для вклада сообщества, потенциально влияя на планирование в Linux. Полный код и инструкции по развертыванию см. в репозитории [ссылка, если доступна]. Если вы экспериментируете с пользовательскими планировщиками, EEVDF предлагает привлекательную отправную точку — справедливую, эффективную и элегантно простую.

3.5 Итоги

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] Павел Е. Шаго Отчет о научно-исследовательской работе: Планировщики 2 б. и. 2025

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

[4] Michelle Galin, Anna Hedblom Linux Kernel Scheduler Evaluation for Performance-Critical Telecom Workloads Linköping University 2025

[5] Jinsong Mao, Benjamin E. Ujcich, Shiqing Ma Rethinking Provenance Completeness with a Learning-Based Linux Scheduler arXiv 2025

[6] Xinbo Wang, Shian Jia, Ziyang Huang, Jing Cao, Mingli Song Mixture-of-Schedulers: An Adaptive Scheduling Agent as a Learned Router for Expert Policies arXiv 2025

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