DOI 10.37882/2223–2966.2022.03.11

ВЗАИМОДЕЙСТВИЕ МЕЖДУ ПРОЦЕССАМИ В БЕЗОПАСНОМ ЯДРЕ

Григорян Давид Арамович
Старший разработчик, Московский государственный технический университет имени Н.Э. Баумана, г. Москва
david8lumen@gmail.com

Аннотация. Большинство современных операционных систем реализованы в виде монолитных ядер. Монолитные ядра имеют множество функций и включают в себя драйверы устройств, а программы пользовательского уровня живут в отдельных виртуальных адресных пространствах. Несмотря на хорошую производительность, монолитные ядра имеют плохую отказоустойчивость: один-единственный драйвер устройства, вызывающий панику, может вывести из строя всю систему. Хуже того, уязвимость в любой части монолитного ядра может привести к полному захвату машины. Традиционные операционные системы имеют монолитную структуру, выполняя все подсистемы ядра в общем адресном пространстве, благодаря чему достигается хорошая производительность за счет изоляции и безопасности. Микроядра улучшили эту конструкцию, разделив компоненты операционной системы на отдельные адресные пространства, но исторически они были чрезмерно дорогими из-за высокой стоимости переключения адресных пространств.

Ключевые слова: операционная система, ядро операционной системы, монолитное ядро, микроядро, гибридное ядро, производительность межпроцессного взаимодействия, многоядерные процессоры.

INTERACTION BETWEEN PROCESSES IN A SECURE KERNEL

D. Grigoryan

Summary. Most modern operating systems are implemented as monolithic cores. Monolithic kernels have many functions and include device drivers, and user-level programs live in separate virtual address spaces. Despite good performance, monolithic cores have poor fault tolerance: a single device driver that causes panic can disable the entire system. Worse, a vulnerability in any part of the monolithic core can lead to a complete takeover of the machine. Traditional operating systems have a monolithic structure, executing all kernel subsystems in a common address space, thereby achieving good performance due to isolation and security. Microkernels improved this design by dividing the operating system components into separate address spaces, but historically they have been prohibitively expensive due to the high cost of switching address spaces.

Keywords: operating system, operating system kernel, monolithic kernel, microkernel, hybrid kernel, interprocess communication performance, multicore processors.

Любой, кто использует технологии с операционной системой, работает с ядром, хотя часто и не осознает этого. Ядро организует процессы и данные на каждом компьютере. Он служит ядром операционной системы и интерфейсом между программным и аппаратным обеспечением. Это означает, что ядро постоянно используется и является ключевым компонентом операционной системы.

Ядро не только служит ядром системы, но также является программой, управляющей всем доступом к процессору и памяти. Он отвечает за наиболее важные драйверы и имеет прямой доступ к оборудованию. Это основа для взаимодействия между аппаратным и программным обеспечением и максимально эффективное управление их ресурсами.

Ядро является сердцем операционной системы и управляет всеми важными функциями оборудования — это касается Linux, macOS и Windows, смартфонов, серверов и виртуализаций, таких как KVM, а также компьютеров любого другого типа.

Ядро всегда строится одинаково и состоит из нескольких слоев.

Самый глубокий уровень — это интерфейс с оборудованием (процессоры, память и устройства), который управляет, например, сетевыми контроллерами и контроллерами PCI Express.

Кроме того, это управление памятью, которое влечет за собой распределение оперативной памяти, включая виртуальную основную память.

Затем идет управление процессами (планировщик), который отвечает за управление временем и делает возможной многозадачность.

Следующий уровень содержит управление устройствами.

Верхний уровень — это файловая система, где процессы назначаются оперативной памяти или жесткому диску.

Ядро занимает центральное место на всех уровнях, от системного оборудования до прикладного программного обеспечения. Его работа заканчивается там, где начинается доступ пользователя: в графическом пользовательском интерфейсе (GUI). Таким образом, ядро граничит с оболочкой (то есть пользовательским интерфейсом). [8]

Программа отправляет «системные вызовы» ядру, например, когда файл записывается. Ядро, хорошо разбирающееся в наборе инструкций ЦП, затем переводит системный вызов на машинный язык и перенаправляет его ЦП. Все это обычно происходит в фоновом режиме, незаметно для пользователя.

Основная задача ядра — многозадачность. Это требует не отставать от временных ограничений и оставаться открытым для других приложений и расширений. [10]

На каждое правило есть исключения в такой простой, хорошо функционирующей системе, как операционная система. Вот почему ядро служит только посредником, когда речь идет о системном программном обеспечении, библиотеках и прикладном программном обеспечении. В Linux графический интерфейс не зависит от ядра.

Когда компьютер включается, ядро загружается в оперативную память первым. Это происходит в защищенной области, загрузчике, так что ядро нельзя изменить или удалить. [9]

После этого ядро инициализирует подключенные устройства и запускает первые процессы. Загружаются системные службы, запускаются или останавливаются другие процессы, инициируются пользовательские программы и выделение памяти.

На этот вопрос лучше всего ответить, ответив: чем не является ядро? Ядро — это не ядро процессора, это ядро операционной системы. Ядро также не является API или фреймворком.

Многоядерные операционные системы могут использовать различные ядра многоядерного процессора, как сеть независимых процессоров. Как это работает? Все сводится к особой структуре ядра, которое состоит из ряда различных компонентов:

Поскольку самый нижний уровень ядра ориентирован на машины, он может напрямую взаимодействовать с оборудованием, процессором и памятью. Функции ядра варьируются в зависимости от его пяти уровней, от управления процессором до управления устройствами. Верхний уровень не имеет доступа к машинам и вместо этого отвечает за взаимодействие с программным обеспечением. [11]

Прикладные программы работают отдельно от ядра операционной системы и просто используют его функции. Без ядра связь между программами и оборудованием была бы невозможна.

Несколько процессов могут работать одновременно благодаря многозадачности ядра. Но, как правило, ЦП может одновременно обрабатывать только одно действие — если только вы не используете многоядерную систему. О быстрой смене процессов, создающей впечатление многозадачности, заботится планировщик.

Из этих компонентов следуют четыре функции ядра: [7]

1. Управление памятью: регулирует, сколько памяти используется в разных местах.
2. Управление процессами: определяет, какие процессы ЦП может использовать, а также когда и как долго они используются.
3. Драйвер устройства: посредник между оборудованием и процессами.
4. Системные вызовы и безопасность: получает запросы на обслуживание от процессов.

При правильной реализации функции ядра невидимы для пользователей. Ядро работает в своей собственной настройке, в пространстве ядра. С другой стороны, файлы, программы, игры, браузеры и все, что

видит пользователь, находится в пользовательском пространстве. Взаимодействие между этими двумя использует интерфейс системного вызова (SCI).

Чтобы понять функцию ядра в операционной системе, представьте, что компьютер разделен на три уровня:

1. Аппаратное обеспечение: основа системы, состоящая из оперативной памяти, процессора и устройств ввода и вывода. Процессор выполняет операции чтения и записи, а также вычисления для памяти.
2. Ядро: ядро операционной системы, связанное с ЦП.
3. Пользовательские процессы: все запущенные процессы, которыми управляет ядро. Ядро делает возможным взаимодействие между процессами и серверами, также известное как межпроцессное взаимодействие (IPC).

В системе существует два режима кода: режим ядра и пользовательский режим. Код в режиме ядра имеет неограниченный доступ к оборудованию, тогда как в пользовательском режиме доступ ограничен SCI. Если есть ошибка в пользовательском режиме, ничего особенного не происходит. Ядро вмешается и устранит любой потенциальный ущерб. С другой стороны, сбой ядра может привести к сбою всей системы. Однако это маловероятно из-за принятых мер безопасности.

Одним из ранее описанных типов ядра является многозадачное ядро, которое описывает несколько процессов, выполняющихся одновременно в одном ядре. Если к нему добавить управление доступом, то получится многопользовательская система, на которой одновременно могут работать несколько пользователей. Ядро отвечает за аутентификацию, так как оно может выделять или разделять вызываемые процессы.

Linux поддерживает полный архив своего ядра. Apple опубликовала типы ядер для всех своих операционных систем для открытого доступа. Microsoft также использует ядро Linux для подсистемы Windows для Linux.

Существуют различные типы ядер, которые используются в разных операционных системах и конечных устройствах. Их можно разделить на три группы: [12]

1. Монолитные ядра: Большое ядро для разных задач. Он отвечает за управление памятью и процессами, а также за связь между процессами и предлагает функции для поддержки драйверов и оборудования. Это ядро в таких операционных системах, как Linux, OS X и Windows.
2. Микроядро: микроядро намеренно маленькое, чтобы ошибки и сбои не влияли на всю операционную систему. Чтобы гарантировать, что оно по-прежнему может выполнять те же функции, что и большое ядро, оно организовано в несколько модулей. Компонент Mach для OS X служит единственным достойным примером, так как до сих пор не существует операционных систем с микроядрами. [12]
3. Гибридное ядро: сочетание микроядра и монолита. Большое ядро более компактно и может быть разбито на модули. Дополнительные части ядра могут добавляться динамически. Они частично используются Linux и OS X.

Рассмотрим некоторые примеры различных ядер.

Микроядра — это ответ на монолитные ядра с лучшей безопасностью и изоляцией ошибок, достигаемый за счет переноса как можно большей части функциональных возможностей операционной системы из ядра. Эта модульность значительно повышает отказоустойчивость и безопасность, но достигается за счет дорогостоящих пересечений адресного пространства изолированных подсистем. Современные микроядра, такие как семейство L4, улучшили производительность межпроцессного взаимодействия (IPC) и широко применяются в производстве, но по-прежнему имеют значительные накладные расходы. Современное микроядро seL4 обеспечивает минимальные накладные расходы в 1260 циклов на процессоре x86 с тактовой частотой 3,4 ГГц [1], что по-прежнему недопустимо для современных рабочих нагрузок с интенсивным вводом-выводом, требующих миллионов переключений домена в секунду. С недавним появлением недорогих безопасных для памяти языков программирования теперь есть возможность улучшить производительность коммутаторов доменов, полагаясь на гарантии безопасности памяти, а не на механизмы аппаратной изоляции.

Rust Rust — это новый язык системного программирования с упором на безопасность памяти во время компиляции, достигаемую за счет системы субструктурных типов [3]. Это означает, что обычные указатели Rust имеют статически определенный срок жизни, что предотвращает целый набор ошибок памяти, таких как использование после освобождения, висячие указатели и двойное освобождение. Кроме того, в безопасном Rust (строгое подмножество языка) приведения указателей и разыменование необработанных указателей запрещены во время компиляции. Применяя гарантии безопасности Rust, можно статически гарантировать, что программа злоумышленника не будет манипулировать внешней памятью, устраняя необходимость в виртуальных адресных пространствах.

Redleaf RedLeaf [2] — новое микроядро, написанное с нуля на Rust. RedLeaf полагается на безопасность

памяти Rust, а не на аппаратные механизмы изоляции памяти. Программы организованы в домены, которые совместно используют одно адресное пространство, но в остальном изолированы. Эта конструкция устраняет дорогостоящие пересечения адресного пространства, от которых страдают микроядра, обеспечивая невероятную производительность междоменной связи без ущерба для безопасности или изоляции.

Изоляция Хотя Rust обеспечивает безопасность памяти для каждого домена, этого недостаточно для обеспечения полной изоляции во время междоменного взаимодействия. Когда домен впадает в панику, ядро выгружает его из памяти, чтобы освободить ресурсы. Если это происходит в середине междоменного вызова, гарантии безопасности памяти Rust нарушаются, так как любые внешние указатели на мертвый домен теперь болтаются. Для междоменной связи домены должны обмениваться указателями из одной общей кучи. RRef позволяет доменам создавать объекты общей кучи и обмениваться ими без копирования, сохраняя при этом безопасность перед лицом сбоев домена.

RedLeaf — это новая операционная система, основанная на безопасности языка программирования Rust, а не на аппаратных механизмах изоляции. RedLeaf запускает все подсистемы ядра операционной системы в одном и том же аппаратном адресном пространстве и вместо этого достигает изоляции за счет сочетания языковой безопасности и специальных примитивов связи. В результате накладные расходы на коммуникацию составляют порядка десятков циклов, что сравнимо с обычным вызовом функции. Однако даже в безопасном ядре междоменная связь требует тщательного выбора конструкции для обеспечения изоляции в случае сбоя доменов. В этой диссертации описываются эти варианты проектирования и вводятся соответствующие концепции общей кучи и удаленных ссылок (RRef), которые основаны на модели безопасности Rust для обеспечения обмена данными без копирования, что обеспечивает безопасность и изоляцию даже в случае сбоя подсистем.

Легкие многоядерные процессоры обеспечивают высокую производительность и масштабируемость за счет объединения в одном кристалле сотен маломощных ядер, архитектуры с распределенной памятью и сетей на кристалле (NoC). Операционные системы (ОС) для этих процессоров имеют распределенную структуру, в которой уровень связи позволяет ядрам обмениваться информацией и взаимодействовать. В настоящее время эта коммуникационная инфраструктура основана на почтовых ящиках, которые позволяют обмениваться сообщениями фиксированного размера с малой задержкой. Однако это решение неоптимально, поскольку оно не может ни полностью использовать NoC, ни эффективно обрабатывать разнообразие протоколов связи ОС.

Легкие многоядерные процессоры были представлены для удовлетворения постоянно растущих требований к масштабируемости и низкому энергопотреблению параллельных приложений [1]. Для выполнения первого требования они полагаются на определенные архитектурные характеристики, такие как архитектура с распределенной памятью и многофункциональная сеть на кристалле (NoC) [2]. Кроме того, для повышения энергоэффективности они построены с простыми маломощными ядрами Multiple Instruction Multiple Data (MIMD) [3], они имеют систему памяти на основе Scratchpad Memories (SPM) без аппаратной поддержки когерентности и они используют неоднородность, используя ядра с разными возможностями [5]. Некоторыми успешными в отрасли примерами этих процессоров являются Kalray MPPA-256, Adapteva Epiphany и Sunway SW26010, причем последний используется в Sunway TaihuLight, который в настоящее время является четвертым по мощности суперкомпьютером в мире. по версии ТОП500.

Операционные системы (ОС) для легковесных многоядерных процессоров используют распределенную структуру, чтобы справиться с большим количеством ядер и архитектурой с распределенной памятью. Этот подход позволяет масштабировать систему до тысяч ядер [4], предоставляя при этом более богатые абстракции и интерфейсы прикладного программирования (API) для программного обеспечения пользовательского уровня. В целом, подсистемы распределенной ОС для легкого многоядерного процессора (например, диспетчер памяти, диспетчер процессов и диспетчер файловой системы) развертываются на ядрах процессора и взаимодействуют друг с другом путем исключительного обмена данными через базовый NoC [3].

Коммуникационный уровень распределенной ОС для легковесных многоядерников напоминает упрощенную версию традиционной распределенной ОС для кластеров рабочих станций. Он управляет мультиплексированием канала связи, безопасностью, маршрутизацией, перегрузкой сети, адресацией сообщений, потоком управления, переупорядочиванием и пакетированием сети. В отличие от распределенных операционных систем, предназначенных для кластеров рабочих станций, она не обрабатывает ошибки связи, поскольку базовое соединение является надежным. Чтобы подсистемы могли взаимодействовать с малой задержкой и надежно, распределенные операционные системы для легковесных многоядерных систем полагаются на абстракцию почтового ящика [3]. Эта специализированная структура системного уровня предоставляет

примитивы отправки/получения и обеспечивает обмен данными фиксированного размера.

Хотя почтовые ящики обеспечивают обмен данными на системном уровне в легковесных многоядерных системах, они могут быть неэффективными. В целом, мы рассуждаем об этом ограничении, основываясь на двух наблюдениях. Во-первых, возможности NoC используются не полностью. Например, эти межсоединения часто имеют специальные аппаратные блоки для эффективной поддержки различных типов и степеней детализации связи, таких как передача малых/больших данных и сигналов синхронизации. Поскольку правильное использование каждого из этих аппаратных ресурсов зависит от семантики связи, а эта информация недоступна на уровне связи, возможности NoC используются не полностью. Во-вторых, существующее разнообразие коммуникационных протоколов в разных подсистемах требует множественных абстракций, иначе реализация протоколов будет неэффективной. Например, службам, которые управляют данными общего назначения (например, диспетчеру памяти и диспетчеру файловой системы), требуется поток управления, чтобы избежать того, чтобы один клиентский процесс занимал всю полосу пропускания службы. Напротив, подсистемы, которые манипулируют мелкозернистыми данными (например, диспетчер процессов), обычно не нуждаются в потоке управления, поскольку они обмениваются небольшими сообщениями фиксированного размера, поэтому поток управления обрабатывается неявно. Следовательно, если для реализации обоих протоколов используется одна и та же абстракция (т.е. почтовые ящики), в последнем излишние накладные расходы.

Выводы

Замена аппаратных методов изоляции языковой безопасностью позволяет значительно повысить производительность без ущерба для гибкости междоменного взаимодействия. Поддержка Rust для разработки типов интеллектуальных указателей позволяет удаленным ссылкам вести себя как обычные указатели с дополнительным преимуществом изоляции. Использование RedLeaf IDL для генерации связующего кода в прокси-серверах делает этот подход масштабируемым для обеспечения изоляции. Rust позволяет начать новую волну разработки микроядра, которая может обеспечить мелкозернистую изоляцию без накладных расходов на аппаратное обеспечение.

Легкие многоядерные процессоры достигают высокой производительности и энергоэффективности благодаря выбранному набору архитектурных особенностей, таких как большое количество маломощных ядер, архитектура с распределенной памятью и богатые No C. И наоборот, для решения такой аппаратной задачи операционные системы для этого нового класса процессоров используют распределенную структуру для достижения масштабируемости, предоставляя при этом более богатые абстракции и API-интерфейсы программному обеспечению пользовательского уровня. При таком подходе подсистемы ОС учитываются в наборе служб, которые: развертываются на ядрах процессора; и для совместной работы друг с другом для реализации функциональных возможностей системы.

Чтобы эффективно использовать эту распределенную структуру, ОС имеет коммуникационный уровень, который предоставляет коммуникационные примитивы поверх базового No C. С этой целью современные распределенные операционные системы для облегченных многоядерных процессоров инкапсулируют эти примитивы в структуру почтового ящика: абстракцию, позволяющую передавать сообщения фиксированного размера.

ЛИТЕРАТУРА

1. Городничев М.Г. Методы проектирования и разработки клиент серверных приложений. Технологии информационного общества Международная отраслевая научно-техническая конференция: сборник трудов. 2017. С. 439–440.
2. Городничев М.Г., Кочупалов А.Е. Исследование методов межпроцессного взаимодействия в информационной системе с горизонтальным взаимодействием // Вестник Евразийской науки, 2018 № 4, https://esj.today/PDF/48ITVN418.pdf (доступ свободный)
3. Доклад о развитии цифровой (интернет) торговли ЕАЭС // ЕЭК URL: https://docs.eaeunion.org/pd/ru-ru/0123644/pd_06032019_doc.pdf (дата обращения: 12.02.2022).
4. Операционная система «РЕД ОС» Руководство пользователя URL: https://www.red-soft.ru/files/downloads/products/redos/redos_user_manual_7_3.pdf (дата обращения: 12.02.2022).
5. Операционная система QNX4 // Системная архитектура URL: https://www.kpda.ru/upload/docs/Sys_arch_QNX4.pdf (дата обращения: 12.02.2022).
6. Текущее развитие проектов в сфере цифровой экономики в Регионах России // Аналитический центр при Правительстве Российской Федерации ЕЭК URL: https://ac.gov.ru/files/publication/a/23243.pdf (дата обращения: 12.02.2022).
7. Чичев А.А. Операционные системы. Часть 1. Работа с операционной системой. Учебно-методическое пособие. / Чичев А.А., Чекал Е.Г.— Ульяновск: УлГУ, 2015.— с.

8. Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. Erim: Secure and efficient in-process isolation with memory protection keys. arXiv preprint arXiv:1801.06822, 2018.
9. Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, July 2019.
10. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), pages 973–990, 2018.
11. Pedro Henrique Penna, João Vicente Souto, João Fellipe Uller, Márcio Castro, Henrique Freitas, et al. Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores. Journal of Parallel and Distributed Computing, Elsevier, 2021, 154, pp.1–15.
12. Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany. ACM, New York, NY, USA, 15 pages.

Московский государственный технический университет имени Н.Э. Баумана