DOI: 10.15514/ISPRAS-2026-38(3)-44

Механизмы управления доступом на основе жетонов для микроядра общего назначения

[1,2] *Е.С. Басков, ORCID: 0009-0001-3416-9790 <baskov@ispras.ru>*
[1,2,3,4] *А.В. Хорошилов, ORCID: 0000-0002-6512-4632 <khoroshilov@ispras.ru>*
[1,2,3] *А.К. Петренко, ORCID: 0000-0001-7411-3831 <petrenko@ispras.ru>*

[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.

[2] *Московский физико-технический институт,*
Россия, 141701, Московская область, г. Долгопрудный, Институтский пер., 9.

[3] *НИУ Высшая школа экономики,*
101978, Россия, г. Москва, ул. Мясницкая, д. 20.

[4] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, г. Москва, Ленинские горы, д. 1.

Аннотация. В статье рассматриваются различные подходы к реализации механизма управления доступом на основе жетонов (capability-based security) в ядрах операционных систем, обсуждается, как принятые разработчиками решения влияют на производительность и безопасность конечных систем. Описывается реализация модели доступа с помощью жетонов в микроядре общего назначения Sol, которая позволяет обеспечить высокую производительность на современных многопроцессорных системах при сохранении гибкости использования и удовлетворении всех требований безопасности, присутствующих в других реализациях. Данная реализация механизма безопасности разработана для обеспечения высокой масштабируемости на современных многоядерных микропроцессорах при сохранении функций безопасности, присущих другим современным решениям. Ключевым элементом архитектуры является хранилище жетонов, построенное на основе управляемого пользователем сжатого префиксного дерева и подсчета ссылок, которые обеспечивают быстрые и масштабируемые операции без блокировок. Архитектура поддерживает ограниченные жетоны, структуру пространства имен жетонов без ограничения набора поддерживаемых политик, отзыв доступа к конкретным объектам, а также многопоточные операции, при этом сохраняя низкие накладные расходы. Для решения проблемы управления ресурсами при потенциальном наличии циклических ссылок предлагаются два метода: первый ограничивает вложенность узлов сжатого префиксного дерева ациклическим графом, а второй вводит механизм для восстановления потерянных ссылок на потоки, который также оказывается полезным для эмуляции пространства имен идентификаторов процессов POSIX.

Ключевые слова: механизмы управления доступом; жетон доступа; микроядро; ОС общего назначения.

Для цитирования: Басков Е.С., Хорошилов А.В., Петренко А.К. Механизмы управления доступом на основе жетонов для микроядра общего назначения. Труды ИСП РАН, том 38, вып. 3, часть 4, 2026 г., стр. 7–36. DOI: 10.15514/ISPRAS–2026–38(3)–44..

An implementation of capability-based security for the general-purpose microkernel

[1,2] *E.S. Baskov, ORCID: 0009-0001-3416-9790 <baskov@ispras.ru>*
[1,2,3,4] *A.V. Khoroshilov, ORCID: 0000-0002-6512-4632 <khoroshilov@ispras.ru>*
[1,2,3] *A.K. Petrenko, ORCID: 0000-0001-7411-3831 <petrenko@ispras.ru>*

[1] *Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

[2] *Moscow Institute of Physics and Technology,*
9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.

[3] *National Research University, Higher School of Economics,*
20, Myasnitskaya st., Moscow, 101978, Russia.

[4] *Lomonosov Moscow State University, GSP-1,*
Leninskie Gory, Moscow, 119991, Russia.

Abstract. Capability-based security is a flexible access control mechanism. It is widely adopted by both mainstream operating systems and research microkernels for operating system kernel object access control. Implementation details vary significantly because kernels prioritize different trade-offs. Considerable amount of implementation details is poorly documented, only being available as a part of a source code. Moreover different developers use non-standard divergent terminology. The paper describes implementation details of capability-based security in several kernels, highlights their specifics and discusses how they affect potential system security and performance. The paper also presents a capability-based security model implemented in an experimental general-purpose microkernel Sol. The implementation is designed to deliver high scalability on modern multicore microprocessors while preserving the security features found in other state-of-the-art implementations. The key element of the design is a capabilities storage built on top of a user-managed radix tree and reference counting, featuring fast and scalable lock-free operations. It supports partial capabilities, a policy-free capability list structure, access revocation for specific objects as well as multi-threaded operations, all while maintaining a low overhead of two machine words per object and three words per revocable reference. Allowing arbitrary structures within per-thread capability namespaces complicates resource management due to potential reference cycles. To address this issue, we propose two techniques: the first restricts the nesting of radix tree nodes to an acyclic graph, while the second introduces a mechanism to retrieve lost thread references – a feature that also proves useful for emulating the POSIX process ID namespace.

Keywords: access control mechanism; capability-based security; microkernel; general purpose operating systems.

For citation: Baskov E.S., Khoroshilov A.V., Petrenko A.K. An implementation of capability-based security for the general-purpose microkernel. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 4, 2026, pp. 7-36 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-44.

1. Механизмы управления доступом в операционных системах

Управление доступом в операционных системах (ОС) – это процесс, который отвечает на вопрос: “**кто** имеет право **что** делать и **с чем**?”. В этом случае “**кто**” – субъект операции, который как правило является процессом. “**Что**” – это вид операции. “**С чем**” – объект операции, который может являться, например, файлом, директорией, сокетом или другим процессом. Субъекты также могут выступать в роли объектов. Набор видов операций зависит от объекта и, например, это могут быть операции чтения или записи для файла или отправка сигнала для процесса.

В конкретный момент времени все допустимые операции можно записать в виде матрицы управления доступом (ACM, Access Control Matrix), у которой столбцы представляют объекты, строки – субъекты, а в ячейках указаны допустимые операции для каждой пары субъект–объект. Пример такой матрицы указан на рис. 1.

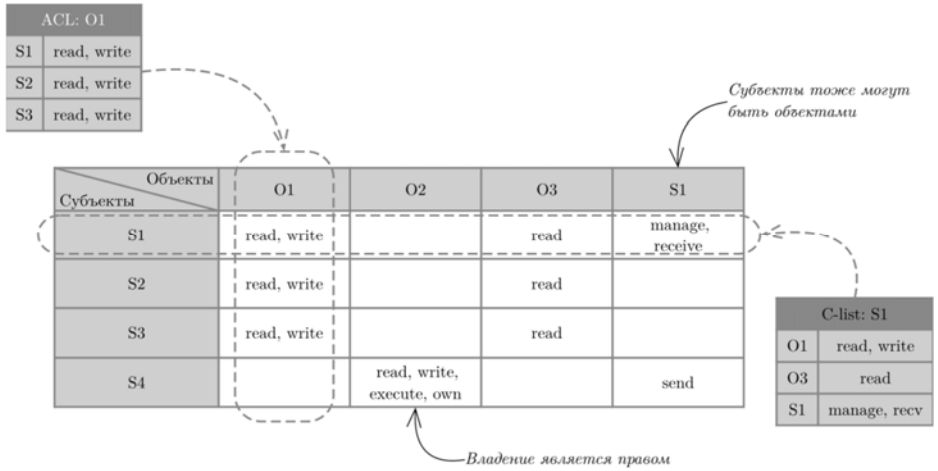

Рис. 1. Сравнение ACL и модели жетонов.
Fig. 1. Comparison of ACL and capability-based security.

Для управления доступом, как правило, используется некоторый механизм, который предоставляется ОС, и политика управления доступом, которая реализуется на основе данного механизма. Механизм определяет то, каким образом могут контролироваться права доступа. Политика определяет конкретные права доступа, и как они изменяются во времени. Используемый механизм определяет, какой набор политик возможно реализовать в системе.

Основные принципы, которые следует учитывать при реализации механизма управления доступом в ОС, обычно включают в себя [1]:

- **Минимальность механизмов** (Economy of mechanism). Реализуется наиболее простой механизм, который допускает реализацию требуемых политик.
- **Безопасные значения по умолчанию** (Fail-safe defaults). Доступ запрещается, если он не разрешен явно.
- **Полное посредничество** (Complete mediation). Контроль должен выполняться при каждом доступе к объекту.
- **Открытая архитектура** (Open design). Безопасность системы не должна основываться на предположении, что атакующему неизвестна информация о ее структуре.
- **Минимальность доступных привилегий** (Least privilege, Principle of least authority). Субъекту должны предоставляться минимальные права, требуемые для его функционирования.
- **Разделение привилегий** (Separation of privilege). Декомпозиция системы на компоненты таким образом, чтобы одному компоненту требовалось предоставлять привилегии нескольких различных видов только в случае, если это действительно оправдано.
- **Минимальность общих механизмов** (Least common mechanisms). Разные задачи не должны решаться посредством общих ресурсов или механизмов, поскольку это несет дополнительные риски утечки информации, непреднамеренного взаимовлияния, а также усложняет интерфейсы.
- **Психологическая приемлемость** (Psychological acceptability). Механизмы должны быть интуитивно понятными и достаточно простыми для применения пользователем.

1.1 Понятие жетона

Механизм управления доступом, для которого в англоязычной литературе используется термин *capability-based security,* основывается на использовании **жетонов** (**capabilities**) – неподделываемых идентификаторов объектов с прикрепленными к ним правами доступа. Владение *жетоном* отождествляется с правом совершения операций с ним, возможно некоторого подмножества операций. *Жетоны* можно передавать другим субъектам, таким образом передавая им права на совершение операций. Неподделываемость означает, что существующий *жетон* нельзя подобрать самому, возможно только получить *жетон* от другого субъекта. Эта неподделываемость может обеспечиваться либо механизмами защиты, либо криптографически, за счет использования случайных идентификаторов с длиной, достаточной для того, чтобы сделать подбор трудозатратным. Первый подход широко используется ядрами ОС или аппаратными механизмами защиты [2]. Второй подход, например, применяется для токенов доступа (*API access tokens)*, которые являются частным случаем *жетонов*.

Наиболее гибкой вариацией *жетонов* является вариация, называемая **жетонами объектов (object capabilities)**. В этой модели *жетон* идентифицирует некоторый объект неотрывно от прав доступа к нему. *Токены доступа* не является *жетонами объектов*, поскольку, как правило, не связаны с конкретными объектами. В дальнейшем в данной статье под *жетонами* будут подразумеваться именно жетоны объектов.

Англоязычный термин ‘*capability*’ не имеет хорошего перевода на русский язык[1], поэтому в данной статье вводится термин ‘*жетон*’. Прямым английским эквивалентом данного термина является слово ‘*token*’, однако данный перевод не используется в русском языке. Также термин ‘*token*’ в контексте управления доступом широко используется с более узким значением криптографически защищаемых жетонов, не привязанных к объектам, что отличающееся от значения термина ‘*object capability*’, как уже было упомянуто.

1.2 Проблема «запутанного посредника» (*confused deputy)*

Данная проблема возникает в ситуациях, когда субъект (программа), предоставляет интерфейс для использования другими субъектами и обладает при этом неявными привилегиями (**ambient authority**), недоступными напрямую для пользователей интерфейса. Если такая программа-посредник обрабатывает объекты, имена которых получает от пользователей интерфейса, то возникает угроза, что пользователи могут эксплуатировать программу-посредника для совершения операций, которые им недоступны.

Например, рассмотрим ситуацию, в которой непривилегированный процесс использует компилятор. При этом допустим, что компилятор может писать в привилегированный журнал аудита, к которому не имеет доступа пользователь. В этом случае, если пользователь передаст в качестве пути выходного файла путь к этому журналу, компилятор, если он не проверяет это явно, может ошибочно перезаписать его. При этом явные проверки являются сложными, поскольку такие привилегированные файлы могут использоваться не только напрямую самим приложением, но и косвенно библиотеками, от которых он зависит.

2. Жетоны в ядрах ОС

В рамках данной статьи **жетон** – неподделываемый локальный идентификатор объекта ядра с прикрепленными к нему правами доступа и возможностью передачи через IPC (Inter-Process Communication, межпроцессное взаимодействие). **Имя** жетона является адресом жетона в ядре ОС и, как правило, представлено целым числом, возможно, с некоторой внутренней структурой. Поскольку жетоны являются локальными идентификаторами, подбор

[1] Существует перевод **мандатная ссылка**, однако он звучит близко к мандатному контролю доступа, к которому доступ на основе жетонов не относится.

численного значения имени от другого субъекта не предоставляет доступа к объекту, что обеспечивает неподделываемость, поэтому разделение возможно только средствами ядра, а именно IPC. При этом пространство имен жетонов называется списком жетонов (*capability list*) или **C-list**.

Не следует путать понятие жетона с используемым в ядре Linux понятием ‘capability’ [3], которые являются флагами, предоставляющими подмножество привилегий суперпользователя, поскольку они не привязаны к конкретным объектам, а также не являются идентификаторами.

Существует ряд ОС, которые реализуют сущности, подходящие под понятие жетона. Файловые дескрипторы (file descriptors, fd) UNIX-подобных ОС являются жетонами, поскольку они являются локальными идентификаторами, и владение ими дает возможность работать с объектами, на которые ссылаются файловые дескрипторы. Также файловые дескрипторы возможно пересылать с помощью механизма межпроцессного взаимодействия (Interprocess Communication, IPC) – доменного сокета UNIX (UNIX domain socket, UDS) с использованием вспомогательных сообщений (*ancillary messages*) с типом SCM_RIGHTS [4]. Фреймворк Capsicum [5-6] расширяет модель файловых дескрипторов, добавляя возможность указания разрешенных операций для каждого файлового дескриптора, а также запрещая доступ к глобальным пространствам имен.

Помимо этого, многие микроядра используют жетоны для управления доступом к объектам ядра, так как данная модель позволяет реализовать широкий набор политик в пространстве пользователя без усложнения ядра. К таким ядрам относятся, например, Mach [7] (а также ядро macOS XNU, которое частично основано на микроядре Mach[2]), EROS [8], Zircon [9], Managarm [10], Coyotos [11] и seL4 [12]. Также жетоны используются во фреймворке Genode [13] для конструирования ОС поверх ряда микроядер. Полезный обзор микроядерных ОС, помимо использующих *модель жетонов*, можно найти в статьях [14-15].

2.1 Сравнение модели ACL и модели жетонов

Модель Access Control List (ACL) является более классической моделью задания прав доступа, которая используется, например, для указания прав доступа к файлам в UNIX. Если рассматривать систему в фиксированный момент времени, модель жетонов будет эквивалентна модели доступа с помощью ACL. При этом в первом случае *C-list* представляет собой строку матрицы контроля доступа, а ACL – столбец. Однако в динамике существуют значительные отличия в предоставляемом наборе прав и подходе к их передаче для этих моделей.

2.2 Пространства имен

Модель ACL использует глобальное пространство имен для объектов, например, пути файловой системы или пространство имен идентификаторов процессов UNIX. Права доступа задаются в виде некоторого идентификатора субъекта, который предоставляет неявные права. Примерами могут служить уникальные идентификаторы пользователей (User Identifier, UID) и идентификаторы групп (Group Identifier, GID) в UNIX-подобных системах. При этом для объектов указывается список идентификаторов субъектов и соответствующие им права. Имена объектов при этом не связаны с правами доступа к ним, из-за чего возникает проблема «запутанного посредника» (confused deputy).

В модели жетонов применяются только локальные имена, связанные с конкретным объектом, что решает эту проблему. Также жесткая связь лучше отражает принцип минимальности привилегий, поскольку при каждом обращении используются только права, имеющиеся для

[2] Однако представляет **жетоны** отдельными от файловых дескрипторов сущностями, которые называются **port right**.

конкретного объекта, и субъекты не объединяются общим идентификатором на группы, имеющие одинаковые права.

2.3 Изменение прав

В модели ACL модификация прав является неявным правом либо владельца объекта, либо монитора ссылок, если применяется мандатное управление доступом (Mandatory Access Control, MAC). При этом изменение владельца – привилегированная операция, поэтому владение достаточно статично.

В модели жетонов изменение прав выполняется передачей жетона через механизм IPC и отзывом (*revocation*) жетонов. Передача субъекту возможна, только если текущий субъект владеет жетоном канала связи с ним. При этом существуют несколько расширений механизмов управления передачей жетонов:

- Для возможности передачи требуется, чтобы у передаваемого жетона было установлено право **delegate**.
- Для возможности передачи требуется, чтобы у жетона, посредством которого происходит передача сообщения, содержащего другие жетоны, было установлено право **grant**.

2.4 Наследование прав

При создании дочернего процесса в модели ACL, он по умолчанию наследует идентификаторы родителя, что дает ему те же права. Это нарушает принцип минимальности привилегий и безопасных значений по умолчанию.

При запуске процесса в модели жетонов, жетоны родителя не наследуются и должны передаваться явно. Файловые дескрипторы нарушают это правило, поскольку для них наследуются все дескрипторы, для которых не установлен флаг O_CLOEXEC [16]. Этот флаг сброшен по умолчанию для обратной совместимости.

2.5 Интерпозиция

Жетоны, представляющие механизмы IPC (например, порты IPC в микроядерных системах или доменные сокеты UNIX, отчасти каналы pipe), предоставляют интерфейс для отправки и получения сообщений. Этот интерфейс не зависит от субъекта, с которым происходит взаимодействие, что позволяет реализовать интерпозицию – прозрачную подмену этих жетонов на некоторые промежуточные. Субъект, который владеет подмененным жетоном, может предоставить альтернативную реализацию интерфейса оригинального жетона или дополнить его функциональность. Во втором случае сообщение после обработки перехватывающим субъектом передается оригинальному жетону.

В качестве дополнительной функциональности возможно реализовать, например:

- ведение журнала операций;
- отладку;
- отложенный запуск сервисов;
- контроль доступа.

2.6 Общий интерфейс системных вызовов

В микроядерных системах все или некоторые системные вызовы могут также быть реализованы через интерфейс отправки сообщений как жетоны, представляющие объекты ядра. Такая реализация также может ускорить механизмы межпроцессного взаимодействия,

поскольку оно будет находиться на «горячем» пути. Это расширяет применимость интерпозиции на такие системные вызовы.

Часто в системах, использующих такой подход, системные вызовы называют методами, из-за чего возникают утверждения о том, что такие системы имеют значительно меньше системных вызовов, чем системы, использующие монолитные ядра. Например, seL4 имеет около 80 системных вызовов, реализованных как методы, но только 7 или 10 классических системных вызовов[3]. Также такой подход используют микроядра EROS, Coyotos и Mach.

2.7 Реализация мандатного управления доступом поверх модели жетонов

Если все системные вызовы реализованы посредством передачи сообщений, возможно реализовать мандатный контроль доступа на уровне пользователя. В этом случае используется дополнительный компонент, **монитор ссылок**, который при помощи интерпозиции реализует управление доступом для всех сообщений. При этом монитор ссылок становится частью доверенной кодовой базы (Trusted Computing Base, TCB), поскольку сам не подлежит мандатному контролю. Например, такой подход используется в системах KeySAFE [17] и EROS [8].

Такой подход будет медленнее, чем классические, поскольку увеличивает количество сообщений в два раза. Более практическим подходом является интерпозиция только на границах доменов защиты. В этом случае мандатный контроль производится только для сообщений между доменами, а операции внутри одного домена используют только модель жетонов, которая является дискреционной.

Отметим, что это вполне соответствует реализациям мандатного управления доступом в классических монолитных ОС, поскольку в таких системах как ядро, так и привилегированные процессы представляют собой крупные компоненты, содержащие в себе множество разнородной функциональности, на которые распространяются единые правила управление доступом. В микроядерных ОС домены как раз соответствуют таким крупным компонентам, а наличие декомпозиции доменов на изолированные подкомпоненты позволяет реализовать дополнительную защиту, недоступную в случае монолитного подхода.

2.8 Ограниченные жетоны и уникальные имена

Есть два аспекта реализации модели жетонов, которые являются взаимоисключающими. Оба аспекта имеют плюсы и минусы, поэтому существуют реализации, которые выбирают каждый из них.

Ограниченные жетоны предоставляют доступ только к части операций, поддерживаемых объектом, на который ссылается жетон. Это обеспечивается за счет поддержки набора флагов прав доступа для жетона, каждый из которых разрешает выполнять подмножество всех операций, реализуемых конкретным объектом.

Эти флаги могут быть как грубыми, например, до 4 флагов в seL4: отправка сообщений, отправка жетонов в сообщениях, отправка с ответом и получение сообщений; так и мелкогранулярными, например, 64 флага, дополненных списками разрешенных операций fcntl() и ioctl() в Capsicum [5]. Помимо этого, такие флаги поддерживаются, например, Fuchsia [18], EROS [8] и Coyotos [11], а также в рудиментарном виде для файловых дескрипторов UNIX.

Наличие флагов позволяет улучшить поддержку принципа минимальности привилегий и разрешить некоторые проблемы ранних реализаций.

[3] 10 – для версии ядра с поддержкой систем со смешанной критичностью.

Минусом такой модели является то, что для нее необходимо поддерживать несколько имен для одного жетона. Это может потребовать явных системных вызовов для сравнения жетонов в некоторых ситуациях, хотя набор таких ситуаций минимизируется при корректном подходе к реализации передачи через IPC, а именно, при поддержке автоматической замены жетонов на их метки при передаче, если этот жетон ссылается на тот же примитив IPC, через который происходит передача.

Уникальные имена гарантируют, что, если субъект получил по IPC жетон, который уже присутствует в его пространстве имен, то этот жетон получит то же имя. Плюсом такого подхода является возможность сравнения жетонов по имени, что не требует вмешательства ядра.

К минусам подхода относится сложность реализации. Поскольку имена транслируются при IPC, требуется поддержка выделения имен ядром, что усложняет реализацию. Помимо этого, требуется поддержка подсчета ссылок на жетоны, поскольку компоненты, работающие с одним именем жетона, полученным несколько раз, могут не знать друг о друге, и один компонент может удалить имя, пока другой все еще использует его. Данный подход применяется в Genode [19] и Mach [20].

Также к минусам выделения имен жетонов ядром относится проблема их повторного использования. Если объект был удален кем-то извне, то при повторном использовании этого имени возможно *использование после освобождения* (*Use After Free, UAF*) имени жетона, что приводит к проблемам с безопасностью и корректностью. Mach исправляет данную проблему, заменяя жетон на специальную заглушку – *dead name*, которая не позволяет выполнять никакие операции, но должна быть освобождена явным образом, чтобы ядро могло повторно использовать данное имя жетона [20].

3. Проблемы ранних реализаций

Ранние реализации модели жетонов имели два дефекта, влияющих на безопасность системы: отсутствие контроля распространения и невозможность возврата доступа. Особенно они были характерны для аппаратных реализаций данной модели, которые не могли гарантировать выполнение ***-property**[4] из модели безопасности Белла-Лападулы [21-22]. В этих реализациях жетоны были аппаратными и отмечались одним битом метки в памяти. При этом, если в объекте с меньшим уровнем привилегии, который доступен только через жетон с правом только на чтение из текущего, более привилегированного субъекта, содержатся жетоны, у которых есть право на запись, то субъект может их извлечь, таким образом получив право на запись в менее привилегированный объект. Более современные аппаратные реализации [2] и программные реализации, как правило, не имеют данного дефекта.

3.1 Контроль распространения

Если субъект, владеющий жетоном, передает жетон другому субъекту, он не может ограничить каким субъектам жетон может быть передан в дальнейшем. Потенциально жетон может стать доступен всем субъектам, которые достижимы по IPC из получателя, то есть находятся в транзитивном замыкании множества достижимости по IPC.

Пусть o – объект, R={recv,send,grant} – множество прав, (o,r) – жетон $r \subseteq R$, $C(o)$={c_i=(o_i,r_i)} – пространство имен жетонов субъекта o.

$$R_s(o)=\{s \vee (p,r_1) \in C(o) \wedge \text{send} \in r_1 \wedge (p,r_2) \in C(s) \wedge \text{recv} \in r_2\}$$

$$R_r(o)=\{s \vee (p,r_1) \in C(o) \wedge \text{recv} \in r_1 \wedge (p,r_2) \in C(s) \wedge \text{send} \in r_2\}$$

$$R^0(o)=R_s(o) \cup R_r(o)$$

[4] Субъект с большим уровнем привилегий не может писать в объект с меньшим.

$$R^n(o)=R^{n-1}(o) \cup \left(\bigcup_{o_i \in R^{n-1}(o)} \left(R_s(o_i) \cup R_r(o_i) \right) \right)$$

$$R^*(o)=R^\infty(o)$$

$R^*(o)$ – множество субъектов, которые могут получить доступ к жетонам субъекта o. $G^*(o)$ ограничено, поскольку количество объектов в системе конечно.

Более современные модели позволяют ограничить это множество двумя способами:

- Поддержка права **grant**, которое позволяет пересылать жетоны через данный жетон, предоставляющий доступ к примитиву IPC. В этом случае отсекаются части графа достижимости. Этот механизм поддерживается, например, в seL4.
- Поддержка права **delegate**, которое разрешает передавать данный жетон посредством IPC. Это позволяет сократить подмножество субъектов, которым может быть доступен данный жетон, только до прямого получателя. Этот механизм поддерживается, например, для Fuchsia как **ZX_RIGHT_TRANSFER** [22].

В модели, поддерживающей **grant**, множество субъектов $G^*(o)$, которые могут получить доступ к жетонам субъекта o, определяется следующим образом:

$$G_s(o)=\{s \vee (p,r_1) \in C(o) \wedge \{\text{send,grant}\} \subseteq r_1 \wedge (p,r_2) \in C(s) \wedge \text{recv} \in r_2\}$$

$$G_r(o)=\{s \vee (p,r_1) \in C(o) \wedge \text{recv} \in r_1 \wedge (p,r_1) \in C(s) \wedge \{\text{send,grant}\} \subseteq r_2\}$$

$$G^0(o)=G_s(o) \cup G_r(o)$$

$$G^n(o)=G^{n-1}(o) \cup \left(\bigcup_{o_i \in G^{n-1}(o)} \left(G_s(o_i) \cup G_r(o_i) \right) \right)$$

$$G^*(o)=G^\infty(o)$$

При этом, как можно видеть, $G^*(o) \subseteq R^*(o)$. Однако данные модели, хотя и являются корректными, не позволяют получить практически полезных результатов в динамических системах.

Также существуют механизмы, которые позволяют ограничить права у распространяемых объектов.

EROS и Coyotos поддерживают атрибут **weak**, который позволяет обеспечить *-property для жетонов-контейнеров[5]. Если объект помечен как **weak**, то все жетоны, прочитанные из него, помечаются как **weak** и **read-only**. Это является важным для данных реализаций, потому что в них виртуальное адресное пространство процесса описывается как дерево, в котором внутренними узлами являются жетоны объектов **node**, а листьями – жетоны, описывающие страницы памяти. При этом некоторые узлы могут быть разделены между доменами защиты. Включение атрибута **weak** в capability-based security в EROS позволило доказать возможность реализации на данной системе политик безопасности, не нарушающих *-property [23].

Это не является проблемой для seL4, которое также использует иерархию таблиц страниц[6], поскольку seL4 не поддерживает разделение промежуточных таблиц страниц. При этом для таблиц жетонов **CNode**, которые в отличие от EROS являются отдельной от таблиц страниц сущностью, проблема контроля распространения сохраняется, и в разделяемых таблицах возможно ограничить права у жетонов-листьев.

3.2 Отзыв прав доступа

После того как жетон был передан, может потребоваться отозвать права доступа. Например, таким сценарием может быть намерение повторно использовать ресурс или изменение

[5] **node** – объект, который сам является жетоном и может содержать в себе 32 других жетона.

[6] В отличие от EROS, в seL4 такая иерархия является прямым представлением аппаратных таблиц страниц.

политики безопасности. В оригинальной модели жетонов это не представлялось возможным. Более новые модели поддерживают два механизма отзыва прав доступа.

3.2.1 Строгое владение

При использовании данного подхода существует единственный жетон, который владеет описываемым объектом, и при его удалении все остальные ссылки автоматически удаляются.

К реализациям со строгим владением относятся Mach и Genode. В случае с последним владение привязывается к субъекту, создавшему жетон и не может быть передано. В Mach может существовать только один жетон *receive right*, который является жетоном, позволяющим получать сообщения. При его уничтожении удаляются все соответствующие жетоны *send right*, которые позволяют отправлять сообщения.

3.2.2 Возможность отзыва производных жетонов

В этом случае поддерживается информация о всех жетонах, производных (derived) от данного, то есть скопированных или переданных по IPC. Благодаря этому поддерживается возможность аннулировать (revoke, rescind) доступ ко всем производным жетонам, тем самым прекратив доступ к ресурсу без его удаления. Это делает данный механизм более гибкой альтернативой.

В существующих микроядрах используются два подхода к реализации данного механизма:

- Поддержка дерева производных жетонов и возможности удаления поддерева всех жетонов производных от данного. Данный подход применяется в seL4.
- Версионирование ссылок и возможность отозвать доступ путем увеличения номера версии в самом объекте. При этом для обеспечения гибкости поддерживаются объекты-посредники (proxy objects), предоставляющие прозрачный доступ к родительскому объекту. Такой подход реализован в EROS и Coyotos.

3.3 Отзыв жетонов в seL4

Дерево производных жетонов в seL4 (рис. 2) представляется в виде двусвязного списка (рис. 3, 4) с дополнительной информацией о глубине каждого элемента. Это позволяет сократить метаданные об иерархии до двух машинных слов. Однако эта реализация подразумевает использование одной большой блокировки для синхронизации, что делает ее неприменимой в масштабируемых системах с большим числом процессоров. Также отзыв доступа к объекту требует удаления всех производных жетонов, что требует $O(n)$ операций, где n – их число, которое контролируется потенциально недоверенными субъектами. Глубина дерева ограничена на уровне архитектуры ядра, поэтому не может переполниться.

Также поддерживаются помеченные объекты-посредники (badged capabilities), которые прозрачно предоставляют доступ к родителю. Они заменяются значением своей метки при отправке сообщения через них или при их отправке в качестве аргумента через жетон, ссылающийся на родительский объект или другой объект-посредник для того же родительского объекта. Размер метки – одно слово. Это позволяет различать отправителей и прикреплять указатель на контекст сервера, связанный с данным отправителем. Объекты-посредники позволяют отозвать только часть ссылок.

Метаданные жетонов хранятся по значению и занимают 4 машинных слова, два из которых содержат ссылки на иерархию. Размер жетона больше машинного слова не позволяет реализовать чтение, модификацию и перемещение объектов без захвата блокировок.

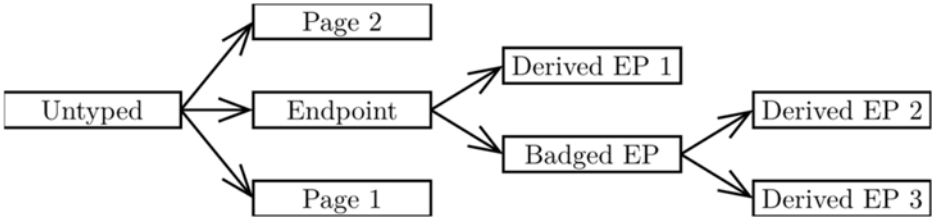

Рис. 2. Дерево производных из seL4.
Fig. 2. Capability derivation tree in seL4.

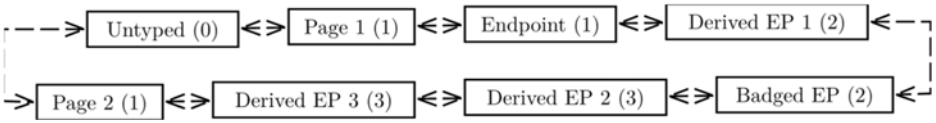

Рис. 3. Представление дерева в виде списка.
Fig. 3. Tree represented as a list.

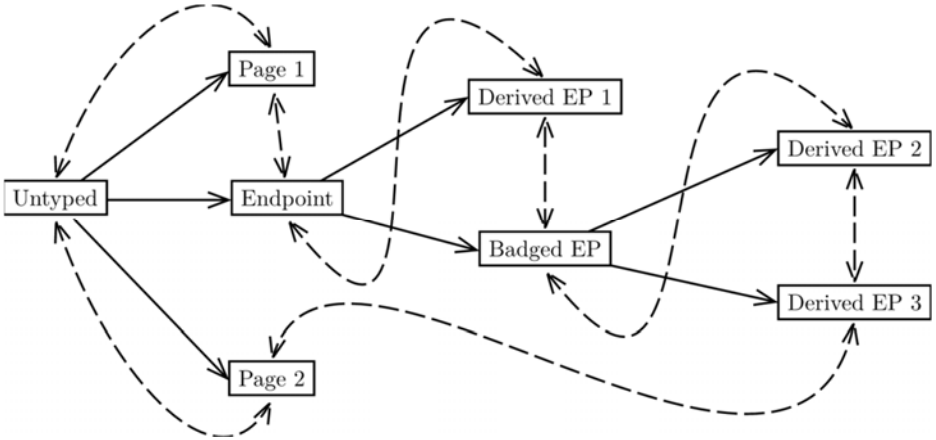

Рис. 4. Совмещенное представление в виде списка и дерева.
Fig. 4. Combined representation as a list and a tree.

3.4 Отзыв жетонов в EROS

В EROS используется глобальная таблица объектов для реализации свойства сохранности состояния или персистентности[7] [24]. Жетоны ссылаются на таблицу, а элемент таблицы ссылается на сам объект. Жетоны и таблица содержат номера версии объекта. Сами жетоны занимают 16 байтов на 32-битных системах (4 машинных слова) и содержат тип и права доступа помимо вышеуказанных полей.

При этом жетон действителен, только если оба номера версии совпадают. Благодаря этому возможно отозвать все ссылки на объект, увеличив номер версии в таблице, то есть за $O(1)$. Объекты-посредники прозрачно предоставляют доступ к родителю, что позволяет отозвать доступ только к части ссылок.

[7] Способности сохранения состояния системы после завершения ее работы. В таких системах вся память программ является только кешем страниц некоторого энергонезависимого накопителя.

Эта реализация не имеет недостатков подхода seL4. Однако к недостаткам этой модели относятся возможное переполнение номера версии и ограничение количества объектов размерами таблицы.

Помимо этого, размер жетона больше машинного слова не позволяет реализовать чтение и модификацию объектов без захвата блокировок. При этом наличие двух уровней косвенности – объекта-посредника и таблицы объектов – добавляет потенциальные промахи кеша процессора, количество которых значительно влияет на производительность программ на современных системах [25].

4. Структура пространства имен

Как было указано выше, как правило, имена жетонов являются численными. Однако структура и реализация самого пространства имен могут разниться от системы к системе.

Пространство имен может быть локальным для потока (seL4, EROS, Coyotos) или процесса (Genode, Mach, Fuchsia, UNIX fd, Capsicum). Имена могут выделяться пользователем (seL4, EROS, Coyotos), системой (Genode, Fuchsia) или обоими (Mach, UNIX fd, Capsicum).

Если имена выделяются пользователем, части пространства имен также могут выделяться пользователем (seL4, Coyotos). Поскольку принято, что микроядерные системы должны минимизировать политику в ядре, это является предпочтительным вариантом, который, однако, усложняет пространство пользователя.

EROS имеет фиксированный размер пространства имен (32 capability registers), и при необходимости использования большего числа жетонов пользователь должен их извлекать из объектов-контейнеров **node** явным образом. Остальные описанные системы имеют динамический размер пространства имен.

4.1 Структуры данных

Для представления пространства имен в ядре могут использоваться различные структуры данных. Ниже перечислены те из них, которые применяются в описываемых системах.

4.1.1 Вектор

Это наиболее простая и быстрая реализация. Она используется для UNIX fd и, соответственно, Capsicum. Также она применяется в EROS с фиксированным размером пространства имен. Возможна реализация неблокирующего поиска при использовании механизмов отложенного освобождения памяти[8].

Для этой структуры данных изменение размера является сложной операцией, однако возможно распределить цену изменения размера во времени посредством использования двух массивов и применения инкрементального копирования из старого вектора в новый при выполнении каждой операции с пространством имен.

Также разреженные имена будут требовать большого количества памяти. Память может выделяться как физически непрерывно, так и отображаться виртуально в ядре. Второй вариант является более сложным и вызывает больше расходов на изменения отображений памяти в ядре, но снимает требования по выделению непрерывных диапазонов физических адресов.

4.1.2 Управляемое ядром префиксное дерево

Эта структура исправляет недостатки вектора, поскольку она может изменять свой размер без необходимости использования больших непрерывных выделений памяти, настройки виртуальных отображений и копирования содержимого. Помимо этого, пространство имен

[8] Например, RCU [24] или Hazard pointers [25].

может быть разреженным, но имеет лучшую локальность памяти по сравнению с хеш-таблицей, поэтому может быть быстрее. Для этой структуры данных также возможен неблокирующий поиск.

Как правило, для более быстрого поиска используется дерево с основанием равным некоторой степени двойки (обычно 16, 32 или 64).

В Mach используется пара префиксных деревьев с основанием узла 64: одно для прямого отображения имени в объект, и одно – для обратного – из объекта в имя [28].

4.1.3 Управляемое пользователем префиксное дерево

Этот механизм аналогичен предыдущему, но требует выделения узлов дерева пользователем. Это позволяет точно управлять выделением памяти, а также позволяет использовать узлы большего размера в пользовательских сценариях, где это может привести к повышению производительности.

4.1.4 Сжатое префиксное дерево или Guarded Page Table

GPT расширяет предыдущий механизм, добавляя в узлы **guard** – битовую строку, которая должна быть префиксом ключа, чтобы этот узел был применим. После сопоставления префикс отбрасывается. Это позволяет пропускать промежуточные узлы в разреженном пространстве имен. Это наиболее гибкая реализация из трех.

Данный механизм используется в Coyotos [11] и seL4 [29].

4.1.5 Хеш-таблица

Managarm использует хеш-таблицу в качестве пространства имен [30], что также решает проблему разреженности, но сохраняет проблему выделения, если не использовать HAMT (Hash Array Mapped Trie [31]), которые построены на префиксных деревьях. Также хеш-таблицы имеют меньшую локальность по сравнению с предыдущими реализациями.

4.1.6 Глобальная таблица имен

Fuchsia использует глобальную таблицу имен, индексы в которой хешируются и возвращаются пользователю как имена. Чтобы контролировать доступ, каждая запись содержит в себе ссылку на пространство имен, которому оно принадлежит.

Данный подход имеет те же минусы, что и хеш-таблица, но имеет меньшее потребление памяти. Также использование глобальной структуры данных проблемно с точки зрения масштабируемости и поддержки NUMA-систем[9].

Среди плюсов данной реализации – возможность простой реализации интроспекции всех жетонов, используемых в системе.

4.1.7 Двоичное дерево поиска

Эта структура данных применима для очень разреженных пространств имен, если имена выделяются системой. Двоичные деревья имеют наименее эффективную реализацию поиска из приведенных структур данных[10] и не поддерживают неблокирующий поиск.

Genode использует AVL-дерево для пространства имен жетонов [32].

[9] Non-Uniform Memory Access, система со временем доступа к ресурсам, таким как ядра процессора, системная память или устройства, зависящим от субъекта, производящего доступ.

[10] Поскольку имеют наибольшую глубину, то есть могут вызвать наибольшее количество доступов к памяти, и, соответственно, количество промахов кеша.

5. Общий взгляд на существующие реализации

5.1 Файловые дескрипторы

Файловые дескрипторы в UNIX-подобных системах [33] (file descriptors, fd) подходят под определение жетонов. В этом механизме нет различных дополнений модели, которые позволяют ограничить распространение жетонов, отозвать существующий доступ[11].

Помимо этого, ограниченные жетоны также существуют только в очень рудиментарном виде. Файл можно открыть как доступный только на чтение, но состояние открытого файла (file description), на которое ссылается дескриптор, является изменяемым и включает в себя текущее смещение и флаги доступа. Изменение смещения или флагов, таких как O_NONBLOCK, может помешать работе других приложений, использующих тот же дескриптор.

В приложениях остается доступ ко многим глобальным пространствам имен, присутствующим в UNIX-подобных системах. Также файловые дескрипторы директорий не ограничивают достижимые пути в файловой системе из-за наличия ссылки на родительскую директорию «..» [34].

Часть этих ограничений исправляются для Capsicum.

5.2 Фреймворк Capsicum

Capsicum [5] – фреймворк для FreeBSD, использующий модель жетонов. В отличие от ядра macOS XNU [35], которое использует компоненты микроядра Mach, включая жетоны, Capsicum использует пространство имен файловых дескрипторов для жетонов, а не добавляет еще одно независимое.

Добавляется возможность ограничить системные вызовы, которые можно выполнить для жетонов с использованием 64 флагов доступа. Помимо этого возможно указать список разрешенных операций для **fcntl()** и **ioctl()**, которые являются точками расширения и не имеют фиксированного набора операций.

Также существует возможность включить режим **Capability mode**, который запрещает использование глобальных пространств имен[12] и неявных прав доступа. Некоторые глобальные пространства имен все еще доступны, но только в очень ограниченном виде, например, из 3000 параметров **sysctl()**, доступны 30, которые не нарушают изоляцию. В этом режиме нельзя использовать ссылку на родительскую директорию “..”, чтобы выйти за пределы дерева директорий, которое описывает дескриптор.

Глобальное пространство имен PID процессов заменяется на дескрипторы процессов (**process descriptors**), которые также являются файловыми дескрипторами и создаются с помощью системного вызова **pdfork()**.

Добавление этих ограничений позволяет поддержать точный контроль доступа для поддержания принципа минимальных привилегий. Внедрение этого механизма требует некоторых изменений в программе, например использование **openat()** вместо **open()**, однако зачастую они тривиальны. Утверждается, что интеграция Capsicum является более простой при соблюдении того же уровня изоляции [5], чем использование более мощного фреймворка SELinux [36-37], который используется для реализации мандатного контроля доступа с использованием модели FLASK [38].

Отзыв доступа все еще не поддерживается, а также не поддерживается запрет на передачу через UNIX domain socket.

[11] Для ОС семейства BSD существует системный вызов **revoke()**, который, однако, работает по пути файла и только для некоторых типов файлов устройств.

[12] Глобальные пространства имен в ОС FreeBSD: Process ID (PID), File paths, NFS file handles, File system ID, Protocol addresses, Sysctl MIB, System V IPC, POSIX IPC, System clocks, Jails, CPU sets.

5.3 Система Fuchsia

Fuchsia [39] – микроядерная операционная система от Google, которая использует микроядро Zircon [9], изначально основанное на микроядре Little Kernel [40], но сильно отклонившееся от него. Это микроядро отходит от принципов минимальности и реализует механизмы более близкие к микроядрам первого поколения ради практичности. Zircon реализован на C++ и насчитывает порядка 500 тысяч строк кода.

В Fuchsia для жетонов используют термин «handle». Пространство имен использует глобальную таблицу в реализации, но привязано к процессу [41]. Поддерживаются ограниченные жетоны с 25 различными правами. Жетоны можно дублировать, сокращать права и отправлять с помощью каналов. Имена выделяются ядром.

Существует возможность запретить передачу жетона другим субъектам, однако средства отзыва ссылок отсутствуют. Существует глобальное пространство имен KOID, которое добавляет уникальные идентификаторы объектам ядра. Модель в целом близка к файловым дескрипторам в Capsicum.

5.4 Система Managarm

В Managarm [10] используется микроядро, состоящее из двух частей: Eir, которая является частью, зависящей от платформы, и выполняет инициализацию системы и Thor, который является частью ядра, работающей при инициализированной системе [42].

Данная система также отказывается от минимальности в пользу более прагматичного подхода к реализации ядра. Ядро также написано на C++, однако насчитывает только порядка 50 тысяч строк кода и является более минимальным по сравнению с Zircon.

Пространства имен являются объектами первого класса («universe»). Жетоны называют дескрипторами. Дескрипторы возможно передавать напрямую и через IPC. Механизмов ограничения распространения и возврата владения нет. Имена выделяются ядром.

5.5 Микроядро Mach

Mach [7, 19] является микроядром первого поколения, поэтому имеет значительный размер, механизмы более близкие к монолитным ядрам и сложный механизм IPC. IPC сообщения буферизуются, типизированы и имеют сложную структуру, которая проверяется ядром.

Пространство имен жетонов локально для процесса (task) и задается префиксным деревом.

Жетоны в Mach называются *port right*, поскольку все объекты представляются как порты IPC и взаимодействие происходит посредством IPC. Существует 5 типов портов: **receive**, **send**, **send once**, **port set** и **dead name**. *Port set* используется для мультиплексирования портов, send once используется для ответов на сообщения и удаляется после использования. Можно иметь только один жетон *receive right* и множество жетонов *send right*. При удалении жетона receive right все жетоны *send right* становятся заглушками (жетонами *dead name*), над которыми нельзя производить операции, что является грубой формой прекращения доступа. Имена уникальны и управляются ядром.

Через порты можно отправлять жетоны. При отправке можно указывать широкий спектр вариантов поведения, таких как передача, копирование жетона, создание нового *send-once* жетона. Декодирование всех жетонов в заголовке сообщения происходит атомарно, что позволяет использовать операции, которые уничтожают старое имя жетона (например, MACH_MSG_TYPE_MOVE_SEND) одновременно с другими.

5.6 Фреймворк Genode

Genode [13, 18] является фреймворком для построения операционных систем общего назначения поверх ряда микроядер. Среди таких микроядер seL4, Nova, okL4, Pistacio, Fiasco.OS, а также поддерживается работа поверх Linux и может использоваться собственное

микроядро base-hw, которое является наиболее эффективной реализацией данного фреймворка. Genode использует компонентный подход и модель жетонов.

Компонент состоит из домена защиты, образа исполняемого файла и одного или нескольких потоков. В большинстве случаев это соответствует понятию процесса UNIX-like систем. Пространство имен жетонов локально для компонента и представляется AVL-деревом.

Компоненты являются иерархическими и образуют дерево. Родительский компонент владеет дочерним и может осуществлять контроль над управлением и предоставлением ресурсов дочернему, реализуя некоторую локальную политику безопасности. При уничтожении родительского компонента уничтожается каждый дочерний. Получение доступа ко всем начальным ресурсам и жетонам осуществляется через родителя. Предполагается, что дочерний компонент доверяет родительскому, но не наоборот.

Взаимодействие между компонентами выполняется по модели клиент-сервер с использованием соединений, описываемых сессиями, представленных жетонами и некоторым состоянием в клиенте и сервере. Сервер не доверяет клиенту. Клиент не знает сущность сервера, но изначальный жетон сервера получается от родителя, поэтому вынужден ему частично доверять.

Core – корневой компонент системы, реализующий низкоуровневые и привилегированные функции, таких как управления памятью поверх интерфейсов используемого ядра.

Genode написан на языке C++ и также содержит порядка 50 тысяч строк кода [32], из которых 20 тысяч занимает ядро, совмещенное с Core со всеми аппаратными платформами.

Используются уникальные имена жетонов, выделяемые библиотекой или ядром. Отзыв владения не поддерживается, но все ссылки на жетон становятся недоступными при удалении объекта. Объект может удалить только владелец, которым является компонент, создавший объект. Передача владения невозможна, но поддерживается передача жетонов по IPC. Средств ограничения распространения нет.

5.7 Микроядро seL4

seL4 [12] является формально верифицированным микроядром второго поколения, которое считается одним из самых продвинутых на данный момент. Оно насчитывает около 23 тысяч строк кода на языке C для **x86-64**. Данное микроядро использует модель жетонов для представления всех ресурсов, в том числе физической памяти и процессорного времени[13].

В качестве пространства имен используется управляемое пользователем сжатое префиксное дерево (Guarded Page Table в терминологии seL4). Узлы дерева **CNode** имеют размер произвольной степени двойки и являются объектами первого класса, также описываемыми жетонами [43]. Благодаря этому пространство имен жетонов может иметь произвольную структуру. При поиске жетона используется полная глубина слова, поэтому для модификации жетонов требуется явно указывать глубину поиска в битах, что усложняет интерфейсы ядра.

Поддерживаются ограниченные жетоны, но набор прав зависит от типа объекта. Права поддерживаются только для жетонов, ссылающихся на объекты IPC (**Endpoint**, **Notification**, **Reply**) и памяти (**Page**). Нет возможности ограничить отображение страницы как исполняемой. Поддерживается отзыв жетонов, а также право **grant**. Этот механизм реализует модель **take-grant** [44-45]

Для реализации отзыва, как было указано выше, поддерживается CDT (Capability Derivation Tree), представленное в виде двусвязного списка с прикрепленной глубиной. Жетоны хранятся по значению и занимают 4 машинных слова, два из которых используются для поддержания CDT и два – для данных жетона. Минусы данного подхода были описаны выше.

[13] Только для MCS (Mixed Criticality Systems) варианта.

Глубина CDT ограничена. Физическая память описывается жетонами типа **Untyped**, которые могут использовать для создания других типов объектов (операцией **retype**). В том числе других **Untyped** меньшего размера. От каждого типа объекта можно создать один уровень производных (**derived**) жетонов. Дополнительно для портов IPC возможно создать помеченные объекты-посредники (**badged** жетоны), для которых можно также создать один уровень производных. В результате глубина ограничена количеством битов в физическом адресе плюс 3.

5.8 Система EROS

EROS [8] (Extremely Reliable Operating System) – микроядерная операционная система с поддержкой персистентности. Она наследует структуру более старой системы KeyKOS, которая имеет отличную терминологию, но практически такую же структуру.

Поддерживаются ограниченные имена, в том числе право **weak**, которое было описано выше. Также поддерживается отзыв всех ссылок на жетон и части ссылок с использованием объектов-посредников (**indirection object capability**).

Пространство имен фиксированное и состоит из 32 слотов (**capability registers**). Чтобы хранить большее число жетонов используются объекты-контейнеры **node**, из которых возможно извлекать объекты явным образом. Это является достаточно ограниченным поведением, и работа с большим числом жетонов требует большого числа дополнительных системных вызовов. Помимо этого, **node** используются для описания адресного пространства субъекта.

5.9 Система Coyotos

Coyotos [11] является наследником EROS, но добавляет ряд черт, которые заимствуются из seL4, поэтому итоговая структура ядра выглядит как гибрид между этими двумя ядрами. Часть кода сериализации объектов для реализации персистентности была перенесена в пространство пользователя, что уменьшает объем кода ядра.

Coyotos использует Guarded Page Table для пространства имен. Также добавляется поддержка **Endpoints**, которые представляют порты IPC, как в seL4. В EROS IPC-адресатами выступали потоки. Также поддерживается механизм асинхронных уведомлений (**events**), аналогичный **notification objects** из seL4. Также добавились **receive queues**, которые аналогичны **port set** из **Mach**.

Для таблиц GPT поддерживается атрибут **Opaque**, который не позволяет адресовать жетоны в них. Это применяется для реализации управляемых извне отображений памяти.

6. Реализация в микроядре Sol

Sol (лат. Солнце, также *Supervisor Of Lux*, где **Lux** – лат. Свет, также *Lightweight Userspace for POSIX* – набор пользовательских сервисов и библиотек данной системы) является микроядром общего назначения, направленным на лучшую поддержку современных многоядерных систем. В связи с этим, Sol отказывается от минимальности для части механизмов, которые плохо подходят для систем общего назначения, потому что они могут быть более динамичными, чем окружения, применяющие микроядра второго и третьего поколения. В частности, для подсистемы памяти был выбран высокоуровневый интерфейс, подобный интерфейсам Mach, Fuchsia или Managarm.

Часть механизмов, таких как IPC и реализация модели жетонов расширяются под требования динамичности, масштабируемости на многоядерных системах, а также с учетом повышения

цены переключения контекста на современных системах, вызванным наличием защиты от аппаратных уязвимостей [47-48] и увеличенным размером контекста[14].

Хотя утверждается, что однопоточное ядро не является проблемой в микроядерных системах [50], но, в отличие от встраиваемых применений, в системах общего назначения гораздо чаще выполняются IPC и системные вызовы [51], поэтому для микроядра общего назначения является важной хорошая масштабируемость. Также использование высокоуровневых механизмов управления памятью приводит к появлению операций в ядре, которые могут выполняться длительное время.

Реализация модели жетонов, в частности, адаптирована таким образом, чтобы поддерживать неблокирующий поиск объекта по имени без барьеров памяти, перемещение жетонов, а также создание и удаление объектов без использования блокировок. Также поддерживается отзыв доступа к группам объектов с минимальным количеством дополнительных метаданных.

При этом поддерживаются существующие расширения модели, увеличивающие безопасность, а также дополнительные механизмы, которые позволяют реализовать более строгие политики безопасности в пространстве пользователя и механизмы POSIX.

6.1 Пространство имен

Пространство имен жетонов использует управляемое пользователем дерево отрезков. Узлы дерева имеют размер, равный произвольной степени двойки и являются объектами первого класса – таблицы жетонов **ktable**, которые должны выделяться пользователем.

Для упрощения доступа к пространству имен длина имени кодируется явным образом как младшие 5 или 6 битов битового представления имени, в зависимости от разрядности машинного слова. В длине пути учитывается сама эта длина, чтобы 0 и -1 не были допустимыми именами жетонов, что защищает от распространенных ошибок, поскольку эти два значения часто используются как значения по умолчанию или для обозначения отсутствующего значения. Более того, использование явной длины снижает вероятность того, что какие-либо случайные значения будут действующими именами, поскольку младшие биты должны быть равны глубине дерева в точности и наличие ненулевых битов в незначащей части пути является ошибкой.

Компоненты после длины хранятся от старших битов значащей части к младшим. То есть для имени длины L, поиск начинается с корневой таблицы размером в 2^n, для индексации в которой используются биты $(L, L\text{-}n]$, как это указано на рис. 5.

Поскольку таблицы являются объектами первого класса, их можно отображать для нескольких потоков, что позволяет разделять все пространство имен (для потоков внутри одного процесса), так и его части. Помимо этого, таблицы можно использовать как контейнеры для жетонов, чтобы эффективно передавать по IPC группы жетонов.

Чтобы обеспечить безопасное разделение пространства имен, для таблиц жетонов поддерживаются 3 вида прав доступа:

- **SOL_KPROT_READ** позволяет чтение жетонов из данной таблицы.
- **SOL_KPROT_WRITE** позволяет модификацию жетонов.
- **SOL_KPROT_EXECUTE** разрешает адресовать элементы данной таблицы как промежуточные, как показано на рис. 5.

[14] Например, расширение AVX-512 использует 32 512-битных регистра и 8 регистров маски [49], то есть состояние требует сохранения и загрузки более 2 КиБ при переключении контекста между потоками, если оба используют это расширение.

Права таблиц при поиске комбинируются логическим «И». Например, чтобы элемент таблицы был доступен на запись, во всех таблицах, которые были пройдены при поиске данного слота, должен быть установлен флаг **SOL_KPROT_WRITE**.

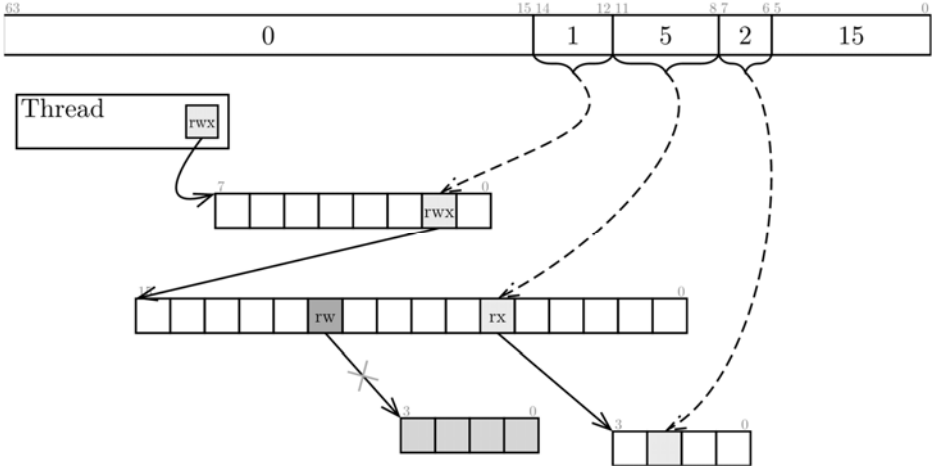

Рис. 5. Поиск жетона по имени.
Fig. 5. Capability lookup by name.

6.2 Жетоны

Для реализации неблокирующего поиска по таблицам жетоны кодируются как указатель на объект, который выделяется и хранится отдельно, и права доступа к этому объекту. Эти два поля упаковываются в одно машинное слово, чтобы поддержать выполнение атомарных операций над ними на всех популярных процессорных архитектурах.

Права жетона кодируются в 7 младших битах машинного слова. При этом указатель сдвигается на 7 вправо при кодировании. Это является более эффективным представлением, поскольку, во-первых, большая часть архитектур может использовать непосредственные значения, как минимум шириной в 7 битов, чтобы эффективно проверять права.

Во-вторых, виртуальные адреса имеют ширину не более 57 битов на большинстве архитектур [49, 52-54] и достигает 57, например, на **x86-64** при использовании пятиуровневых таблиц страниц [49]. Верхние биты указателя при этом расширяются либо знаковым битом, либо нулем.

Благодаря такому представлению указателей возможно декодировать указатели с использованием одного логического или арифметического сдвига на константу.

Поддерживается до 6 флагов, 3 из которых зависят от типа объекта. **SOL_KPROT_SEND** позволяет передавать жетон с помощью механизмов IPC, **SOL_KPROT_RESEND** предотвращает сброс **SOL_KPROT_SEND** после передачи, **SOL_KPROT_INFO** позволяет получать информацию об объекте через **kquery**, что позволяет запрещать интроспекцию.

Для поддержки неблокирующего поиска по дереву при удалении объектов используется механизм отложенного освобождения памяти RCU (**read-copy-update**) [27], который реализован с использованием *состояния спокойствия (quiescence state)* и поддерживает вытеснение потоков ядра, находящихся в критической секции, для лучшей поддержки задач реального времени. Также в реализации для эффективной синхронизации состояний используется дерево состояний (**tree RCU**) [55]. Поскольку используется RCU, поиск имени не потребует барьеров памяти ни на одной архитектуре, кроме DEC Alpha [56-57]. Накладные расходы на синхронизацию сводятся к 4 неатомарным операциям: два увеличения счетчика

на единицу, два уменьшения счетчика на единицу. Один из счетчиков локален для процессора, другой – для потока.

Для того чтобы избежать выделения памяти при каждом копировании жетона, используется подсчет ссылок. Также это позволяет создавать дочерние объекты без захвата локальных или глобальных блокировок – достаточно атомарно увеличить счетчик ссылок родителя на единицу, если он не равен нулю. Если счетчик равен нулю, объект удален и находится в очереди RCU, поэтому увеличивать счетчик в этом случае не допускается. Количество ссылок увеличивается на время выполнения операций, а также при добавлении дочерних объектов и жетонов.

В заголовке каждого объекта хранится тип, ссылка на родителя и счетчик ссылок. Заголовок объекта занимает 2 слова на все жетоны, ссылающиеся на данный объект, что является более эффективным, чем seL4 и EROS, которые требуют 4 слова на каждую дополнительную ссылку. Для Sol количество метаданных жетона не является критичным, потому что страницы памяти и таблицы страниц не описываются отдельными жетонами. Вместо этого используются высокоуровневые объекты адресного пространства **kdomain**, кеша страниц **kbuffer** и **kpmr** для описания квот выделения памяти, и системе требуется гораздо меньшее количество таких объектов для работы.

На рис. 6 приведен эквивалент графа объектов из раздела про seL4[15]

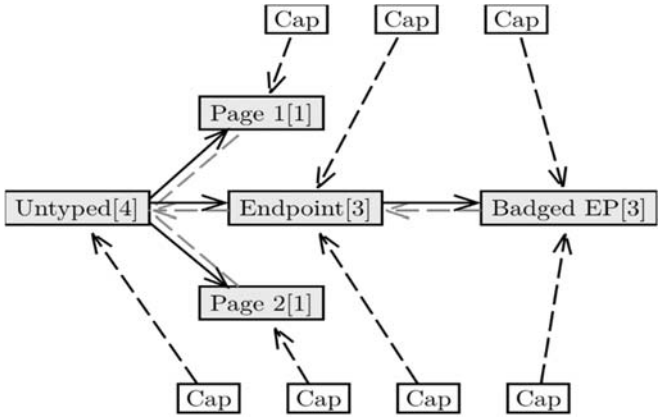

Рис. 6. Структура графа объектов.
Fig. 6. The structure of an object graph.

6.3 Отзыв доступа

Как можно заметить, из-за использования счетчика ссылок нельзя отзывать доступ к произвольным объектам. При проектировании ядра было замечено, что, как правило, требуется отзывать доступ только к определенным объектам, которые заранее известны. К таким объектам относятся примитивы IPC, которые используются на стороне клиента, а также объекты, которые могут отображаться в память.

Для поддержки отзыва таких объектов используются три типа объектов-посредников: помеченные объекты **kevent**, **kendpoint** и объекты-потоки **kstream**. Первые два из них прозрачно предоставляют доступ к родительским объектам (**knote** и **kport**, соответственно), но также содержат метку размером в машинное слово, которую нельзя изменять и которая

[15] Данное ядро использует другие типы объектов, типы объектов из seL4 сохранены для наглядности.

передается ядром как часть сообщения, отправляемого с участием данного объекта-посредника в качестве адресата сообщения или аргумента.

Третий используется для эффективного представления кешей страниц, которые находятся в ядре для эффективности и описываются объектами **kbuffer**. Через эти объекты можно читать и писать в кеш страниц, а также отображать кеш страниц в память. При отзыве доступа все отображения, которые были созданы напрямую с использованием этого объекта или косвенно с использованием первых, также делаются недоступными. Однако для безопасности отображения не удаляются, а замещаются специальными заглушками, которые резервируют виртуальное адресное пространство до тех пор, пока не будут явно удалены или замещены. Данная мера безопасности предотвращает возможность возникновения уязвимостей, вызванных созданием отображений на тех же адресах, что использовались асинхронно отозванными отображениями.

Отзыв происходит посредством удаления указателя на объект-родитель с использованием необходимых блокировок у родителя, чтобы гарантировать отсутствие новых операций с данным объектом после отзыва. Во время отзыва доступа ссылки на объект-посредник не изменяются и удаляются лениво при попытке обращения через них. Освобождение объекта-посредника выполняется только после удаления всех ссылок на него, поэтому такие объекты могут оставаться в памяти длительное время. Однако каждый из них занимает только 24 байта на 64-битных архитектурах, поэтому это не вызывает значительных накладных расходов памяти.

Поскольку при копировании жетона только увеличивается счетчик ссылок, пользователи жетона не могут напрямую влиять на время отзыва жетонов объектов-посредников. Однако они все еще могут увеличивать время отзыва для объектов-потоков **kstream**, поскольку время их отзыва пропорционально количеству и размеру отображений.

6.3.1 Предыдущая реализация

Одна из предыдущих реализаций данного механизма в Sol использовала подход, близкий к seL4. В этом случае структуры данных нужно было модифицировать, чтобы иметь возможность использовать мелкогранулярные блокировки – для создания, удаления или отзыва ребенка требовался захват только блокировки, которая относится к родителю. Для этого потребовалось хранить все ссылки в дереве производных, то есть 4 указателя: указатель на родителя, первого ребенка, следующий и предыдущий объекты, которые являются дочерними того же родителя.

Дополнительно, чтобы поддерживать неблокирующий поиск, таблицы жетонов хранили только указатель на эти данные, а сами данные выделялись отдельно, что требовало выделения памяти при каждом копировании жетона.

Чтобы иметь возможность отозвать объект, потребовалось хранить ссылку на элемент таблицы жетонов, который содержит данный жетон. В результате требовалось 5 дополнительных слов метаданных на каждый жетон вместо 2 в seL4.

Также, когда использовался низкоуровневый механизм выделения памяти, метаданные о состоянии владения памяти были частью CDT. Это делало невозможным многопоточное освобождение памяти после прохождения *состояния спокойствия* (**quiescence state**), поскольку в одной очереди могли содержаться как ссылка на объект, так и на его дочерний объект, и в этом случае появлялись требования к очередности удаления.

В новой системе этого не требуется, потому что ядро может выделять память и использует жетоны, описывающие квоты выделения, вместо описания диапазонов физических адресов. Поэтому память родителя и ребенка может быть освобождена независимо и параллельно.

Третий недостаток поддержки отзыва произвольных объектов – усложнение механизмов ядра. Поскольку объекты ядра, в частности потоки, содержат в себе ссылки на другие объекты, когда поддерживалась возможность отозвать произвольный объект асинхронно,

ядро должно было обрабатывать ситуации, когда части состояния, например адресное пространство или контекст планирования, становились недоступными асинхронно. Это не является проблемой в однопоточном ядре, таком как seL4, однако приводит к значительному усложнению кода в ядрах, которые являются более масштабируемыми.

Из-за этих недостатков в текущей реализации был применен принцип минимальности механизма, и была убрана возможность отзывать произвольные жетоны.

6.3 Удаление объектов

Использование подсчета ссылок может вызвать каскадное удаление объектов. Эта проблема была исправлена благодаря использованию асинхронных очередей удаления, которые являются частью реализации механизмов RCU. Для поддержания очереди не требуется больше памяти, поскольку для ссылки на следующий элемент очереди используется слово, содержавшее счетчик ссылок, которое в данном случае будет численно отрицательным.

Данные очереди реализованы без использования блокировок. При этом у каждого процессора есть MPSC (multiple producer, single consumer) неблокирующая очередь переменного размера, в которую добавляются объекты, ожидающие освобождения после прохождения *состояния спокойствия*, финализацию, а также произвольные функции обратного вызова, также ожидающие вызов по прошествии *состояния спокойствия*.

Сами очереди добавляются в неблокирующую MPMC (multiple producer, multiple consumer) очередь фиксированного размера, когда они становятся непустыми. Процессоры, которые начинают бездействовать или пользовательские потоки, которые выполняют специальный системный вызов, извлекают MPSC-очередь из MPMC-очереди и обрабатывают ее до возникновения какого-либо события или опустошения. При этом пользователь ответственен за добавление процесса, который будет периодически вызывать соответствующий системный вызов, чтобы поддерживать пропускную способность системы.

6.4 Реализация системных вызовов

Все системные вызовы выполнены в виде сообщений IPC, что позволяет выполнять интерпозицию всех жетонов. При этом, существуют некоторые глобальные жетоны, которые предоставляют доступ к группам связанных системных вызовов. Например, **kquery** позволяет получать информацию об объектах.

IPC реализуется как синхронный, блокирующий, с прямым копированием между буфером получателя и буфером отправителя и миграцией потоков. Используются мелкогранулярные блокировки на каждый объект порта IPC, поэтому данный механизм обладает практически идеальной масштабируемостью. Реализация является эффективной для однопоточного выполнения и использует только один MCS spin-lock для извлечения получателя из очереди и быстрый путь IPC, поэтому задержки отправки сообщений сравнимы с таковыми из seL4. Также поддерживается отправка нескольких сообщений за один системный вызов для дальнейшего уменьшения количества переключений контекста, однако подробное описание механизмов IPC выходит за рамки данной статьи.

6.5 Ограничение распространения

В качестве механизма ограничения распространения поддерживается флаг **grant**, отсутствие которого позволяет запретить передачу жетонов через IPC-порт (**kport**), на который ссылается данный жетон, как это было указано в разделе 2.3. Также поддерживается флаг **delegate**, который разрешает передачу объекта и реализован через **SOL_KPROT_SEND** и **SOL_KPROT_RESEND**. Помимо этого, поддерживаются описанные выше права для разделяемых таблиц жетонов.

Помимо этого, добавляется способ идентификации отправителей: каждому домену защиты (объекту **kdomain**, который описывает виртуальное адресное пространство) назначается

уникальный идентификатор. При этом ядро прикрепляет данный идентификатор как часть сообщения IPC. Это позволяет серверу поддерживать более строгие политики безопасности по сравнению с обычной моделью жетонов, например, возвращать ошибку при несовпадении ожидаемого идентификатора, или запрашивать подтверждение прав доступа, которое выполняется с использованием специального сервера аутентификации, который входит в TCB. Этот механизм является полезным на практике и расширяет множество реализуемых с использованием ядра Sol политик безопасности.

6.6 Циклические ссылки

Поскольку в системе используются счетчики ссылок и объекты могут ссылаться на другие объекты, в базовой реализации могут возникать циклические ссылки, вызывающие утечки памяти.

Однако из-за структуры системы циклы могут возникать только для двух типов объектов: потоков и таблиц. Для этих типов проблема разрешается различными способами.

6.6.1 Строгая иерархия таблиц

При их создании каждой таблице назначается численный идентификатор, который называется уровнем. Для запрета возникновения циклов между таблицами система поддерживает инвариант, который говорит о том, что таблицы могут хранить только таблицы с численно большим уровнем. Таким образом, граф таблиц становится ациклическим.

Однако при этом появляется другая проблема: создание таблиц верхнего уровня становится невозможным, поскольку они не могут храниться в других таблицах верхнего уровня.

6.6.2. Локальные для потока слоты

Для решения этой проблемы используется механизм локальных для потока слотов жетонов. При этом для каждого потока создается небольшая (как правило, не больше 32 слотов) таблица жетонов, размер которой указывается при создании объекта потока. Локальные слоты имеют наименьший уровень, поэтому в них возможно хранить любые таблицы.

Помимо этого, данный механизм используется для быстрого выделения и освобождения временных слотов, которые могут потребоваться, например, для получения жетонов по IPC или создания жетонов с меньшими правами перед отправкой.

Также в локальных слотах можно кешировать часто используемые жетоны. Доступ к локальным слотам является самым быстрым, поскольку они находятся на верхнем уровне префиксного дерева.

6.6.3 Domain sets

Потоки также описываются жетонами. Между ними нежелательно запрещать циклические ссылки, в том числе потому, что поток должен владеть жетонами, ссылающимися на потоки того же процесса. Поэтому вместо запрета создания циклов в графе ссылок, описываемых жетонами, в Sol добавляется механизм для получения потерянных ссылок на потоки и домены, подобно **zx_object_get_child()** из Zircon [58], но не требующая получения информации о существовании дочернего объекта извне[16].

[16] В **zx_object_get_child()** используются глобальные идентификаторы объектов ядра – KOID. Эти идентификаторы можно получить через **zx_object_get_info()** с аргументами **ZX_INFO_PROCESS_THREADS**, **ZX_INFO_JOB_PROCESSES**, **ZX_INFO_JOB_CHILDREN** для объектов, которые поддерживаются в **zx_object_get_child()** [57]. В ядре Sol нет глобальных пространств имен объектов, за исключением уникальных идентификаторов доменов, однако они используются только для того, чтобы различать объекты, а не для выполнения каких-либо операций.

Таким механизмом являются объекты типа **kdomainset,** которые предоставляют интерфейс итератора без состояния (протокол[17] приведен на рис. 7) и сами объекты **kdomain**, позволяющие получить список потоков данного домена.

```
protocol domainset {
   next!68(domain iter) -> (domain dst, u64 domain_id)
   create_domainset!69(pmr pmr) -> (domainset dst)
   swap_domain!70(thread thread, domain new_domain,
                  configure_action action) -> (domain old_domain)
   set_id!71(domain domain, u64 new_id)
}
protocol domain : mappable {
   # … прочие методы опущены для краткости
   next_thread!27(protocol thread iter)
                 ->(thread dst, user_thread_state state, u64 thread_id)
}
```

Рис. 7. Определение протокола domain set.
Fig. 7. Domain set protocol definition.

Итератор получает на вход жетон домена и возвращает следующий за ним жетон домены или код ошибки **KERROR_INVALID_RANGE**, если этот домен был последним. Новые домены добавляются в конец списка, а удаленные домены могут удаляться из любой части списка. Чтобы получить указатель на первый домен в списке, нужно передать в качестве предыдущего нулевой жетон.

Объекты **kdomainset** логически являются группами доменов. Каждый домен должен принадлежать **kdomainset**. Каждый поток должен принадлежать домену. Первоначальный **kdomainset** домена и домен потока указываются при их создании.

Имея ссылку на **kdomainset** с достаточными правами, возможно создать дочерний **kdomainset**. При удалении дочернего **kdomainset** домены, входящие в него, переносятся в родительский. Удаление корневого **kdomainset**, создаваемого при старте системы, приводит к панике ядра.

Поскольку **kdomainset** позволяет получать жетоны доменов, которые его используют, он является привилегированным объектом и должен быть доступен с правами на чтение только пользовательскому менеджеру процессов **proc**.

Сами объекты **kdomain** также предоставляют аналогичный интерфейс для получения потоков, принадлежащих данному домену, для чего требуется право **SOL_KPROT_WRITE**. Для обеспечения корректности замена домена для потока требует соответствующего **kdomainset** с установленным правом на чтение, то есть является привилегированной операцией. При создании нового потока требуется указание домена, а при создании домена требуется указание **kdomainset** с правом **SOL_KPROT_WRITE**.

Для удаления потока, менеджер процессов должен удалить все элементы встроенной таблицы жетонов, чтобы разорвать возможные циклические ссылки. Ядро это делает автоматически, когда поток отмечается как завершенный (**killed**).

Помимо получения утерянных ссылок данный механизм используется для реализации пространства имен процессов POSIX [34] в пространстве пользователя. Данное пространство имен является глобальным и для Lux реализуется в пространстве пользователя менеджером процессов **proc**, который также обеспечивает контроль доступа к процессам, иерархию процессов и сигналы POSIX.

[17] Sol использует собственный генератор функций-прослоек для IPC, который специфичен для данного ядра, например, он поддерживает передачу жетонов и отправку группы сообщений.

7. Заключение

Ранние реализации механизма управления доступом на основе жетонов (capability-based security) имели ряд проблем, рассмотренных в разделе 3, таких как невыполнимость ***property**, отсутствие поддержки отзыва права доступа, отсутствие поддержки контроля распространения прав доступа. Это приводило к невозможности реализации ряда политик безопасности, а также к сложностям при применения данного механизма на практике. Данные проблемы со временем были решены благодаря внедрению ряду расширений базового механизма, что позволило повысить его гибкость и упростить применение на практике, в том числе, как основного механизм, обеспечивающего управление доступом в ядре ОС.

Несмотря на наличие общего механизма управления доступом на основе жетонов, практически каждая реализация обладает рядом особенностей, отличающих её от других и влияющих на накладные расходы по памяти и времени выполнения. В контексте применения механизма управления доступом на основе жетонов для микроядер общего назначения наиболее важными свойствами реализации являются:

- возможность реализовать желаемые политики безопасности;
- масштабируемость применяемых алгоритмов при работе на многоядерных микропроцессорах;
- эффективность реализации, в частности, размер дополнительных метаданных.

Sol использует реализацию модели жетонов, которая улучшает многие из этих свойств по сравнению с большинством микроядер третьего поколения, более подходящих для применения во встраиваемых системах. При этом реализуются все современные расширения базового механизма, а также некоторые дополнительные средства, улучшающие безопасность и гибкость системы. Сводная информация по основным свойствам рассмотренных реализаций механизмов управления доступом на основе жетонов представлена в табл. 1.

Поскольку ОС общего назначения без программного обеспечения не имеет большого количества практических применений и адаптация всего требуемого ПО не является практически выполнимой задачей, ОС должна поддерживать интерфейсы, которые уже используются. Среди таких – POSIX [34] и Win32 [60]. В Lux предполагается поддержка первого как более простого и стандартизированного. Однако POSIX предоставляет не все абстракции, которые требуются современным приложениям [59], поэтому планируется поддержка популярных расширений из Linux, вроде **inotify**, **epoll** [62] или **futex** [63], чтобы покрыть функциональность, которая не входит в POSIX, либо имеет меньшую производительность.

Из-за вопросов обеспечения совместимости с существующим программным обеспечением требуется поддерживать классические механизмы управления доступом, такие как права доступа к файлам на основе ACL и идентификаторов пользователей и групп, которые используются в UNIX-подобных ОС, реализующих POSIX. В продолжение данной работы предполагается проверить гипотезу о том, что при использовании эффективной реализации механизма capability-based security возможно поддержать совместимые механизмы в пространстве пользователя с производительностью, сопоставимой с классическими монолитными ядрами.

Эта гипотеза будет проверяться на основе реализации сервисов пользовательского пространства Lux для микроядра Sol, поскольку они включают поддержку интерфейсов POSIX. Поэтому проектные решения должны учитывать возможность эффективной реализации требуемых высокоуровневых программных интерфейсов. В данном случае, например, **kdomainset** позволяет эффективно поддерживать пространство имен процессов – PID из стандарта POSIX.

Табл. 1. Сравнение рассмотренных реализаций механизмов управления доступом на основе жетонов.
Table 1. Comparison of discussed implementations of capability-based security.

	IPC-интерфейс системных вызовов	Уникальные имена	Ограниченные жетоны	Отзыв доступа	Атрибуты	Пространство имен
Файловые дескрипторы UNIX	–	–	–/+	–		Динамический массив
Capsicum	–	–	+	–	64 атрибута	Динамический массив
Managarm	–	–	–	–		Хеш-таблица
Mach	+/–	+	–	–/+ (строгое владение)		2 сжатых префиксных дерева
XNU	–/+	+	–	–/+ (строгое владение)		Массив и хеш-таблица
Genode	н/п или +[18]	+	–	–/+ (строгое владение)		AVL-дерево
Fuchsia	–	–	+	–	25 атрибутов, в т.ч. delegate	Глобальный разреженный массив
seL4	+	–	+	+ (дерево производных)	read, write, grant, grant-reply	Управляемое пользователем сжатое префиксное дерево (GPT)
EROS	+	–	+	+ (посредники и версионирование)	read-only, no-execute, weak	Фиксированный массив и объекты-контейнеры
Coyotos	+	–	+	+ (посредники и версионирование)	read-only, no-execute, weak, opaque	Управляемое пользователем сжатое префиксное дерево (GPT)
Sol	+	–	+	+ (посредники)	read, write, execute, send, resend, info	Управляемое пользователем сжатое префиксное дерево

Список литературы / References

[1]. Saltier J. H., Schroeder M. P. Protection of information in computer systems, IEEE CSIT Newsletter, 1975, vol. 3, issue 12, p. 19, DOI: 10.1109/CSIT.1975.6498831.

[2]. Woodruff J., Watson R. N., Chisnall D., Moore S. W., Anderson J., Davis B., Laurie B., Neumann P. G., Norton R., Roe M. The CHERI capability model: revisiting RISC in an age of risk, SIGARCH Comput. Archit. News, 2014, vol. 42, issue 3, pp. 457–468, DOI: 10.1145/2678373.2665740.

[3]. capabilities(7) – Linux manual page. Available at: https://man7.org/linux/man-pages/man7/capabilities.7.html, accessed 19.10.2025.

[4]. unix(7) – Linux manual page. Available at: https://man7.org/linux/man-pages/man7/unix.7.html, accessed 19.10.2025.

[18] Не применимо для вариантов фреймворка, которые используют микроядра кроме **base-hw**.

[5]. Watson R., Anderson J., Laurie B., Kennaway K. Capsicum: practical capabilities for UNIX. Available at: https://papers.freebsd.org/2010/rwatson-capsicum.files/rwatson-capsicum-paper.pdf, accessed 19.10.2025.
[6]. Watson R. N. M., Anderson J., Laurie B., Kennaway K. A taste of Capsicum: practical capabilities for UNIX, Commun. ACM, 2012, vol. 55, issue 3, pp. 97–104, DOI: 10.1145/2093548.2093572.
[7]. The Mach Project Home Page. Available at: https://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html, accessed 06.05.2025.
[8]. Shapiro J. S., Smith J. M., Farber D. J. EROS: a fast capability system, SIGOPS Oper. Syst. Rev., 1999, vol. 33, issue 5, pp. 170–185, DOI: 10.1145/319344.319163.
[9]. Learn about Fuchsia: Zircon. Available at: https://fuchsia.dev/fuchsia-src/concepts/kernel, accessed 06.05.2025.
[10]. Grinten A. van der The Managarm Project. Available at: https://managarm.org/, accessed 19.10.2025.
[11]. Shapiro J. S., Adams J. W. Coyotos Microkernel Specification. Available at: https://archive.fo/EkWWI, accessed 19.10.2025.
[12]. The seL4 Microkernel. Available at: https://sel4.systems/, accessed 06.05.2025
[13]. GENODE – Operating System Framework. Available at: https://genode.org/, accessed 06.05.2025.
[14]. И.Б. Бурдонов, А.С. Косачев, В.Н. Пономаренко. Операционные системы реального времени. Препринт ИСП РАН № 14, 2006, с. 98.
[15]. Бурдонов И.Б., Косачев А.С., Петренко А.К., Хорошилов А.В., Чепцов В.Ю. Семейство операционных систем КЛЮС. Труды ИСП РАН, том 37, вып. 6, часть 4, 2025 г., стр. 11-26. DOI: 10.15514/ISPRAS–2025–37(6)-47. / Burdonov I.B., Kossatchev A.S., Petrenko A.K., Khoroshilov A.V., Cheptsov V.Yu. CLOS Operating System Family. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 4, 2025., pp. 11-26 (in Russian). DOI: 10.15514/ISPRAS–2025–37(6)-47.
[16]. <fcntl.h>. Available at: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/fcntl.h. html, accessed 19.10.2025.
[17]. Introduction to KeySAFE. Available at: https://web.archive.org/web/20070607185423/http://www.cis.upenn.edu/~KeyKOS/agorics/KeyKos/keysafe/Keysafe.html, accessed 31.10.2025.
[18]. Learn about Fuchsia: Rights. Available at: https://fuchsia.dev/fuchsia-src/concepts/kernel/rights, accessed 06.05.2025.
[19]. Feske N. Genode Operating System Framework Foundations. Available at: https://genode.org/documentation/genode-foundations-25-05.pdf, accessed 06.05.2025.
[20]. Brinkmann M., Matzigkeit G., Hasnaoui G., Baron R. V., Draves R. P., Thompson M. R., Barrera J. S. The GNU Mach Reference Manual. Available at: https://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html, accessed 06.05.2025.
[21]. E. Boebert W. On the inability of an unmodified capability machine to enforce the *-property, Proceedings of the 7th DOD/NBS Computer Security Conference, 1984, pp. 457-468. Available at: http://zesty.ca/capmyths/boebert.html.
[22]. Miller M. S., Yee K.-P., Shapiro J. Capability Myths Demolished, technical report, 2003. Available at: https://classpages.cselabs.umn.edu/Spring-2019/csci5271/papers/SRL2003-02.pdf, accessed 06.05.2025.
[23]. Learn about Fuchsia: Zircon Handles. Available at: https://fuchsia.dev/fuchsia-src/concepts/kernel/handles, accessed 06.05.2025.
[24]. Shapiro J. S., Weber S. Verifying the EROS Confinement Mechanism, Proceedings of the 2000 IEEE Symposium on Security and Privacy, SP '00. IEEE Computer Society, 2000, p. 166.
[25]. Shapiro J. S. Dependency Tracking in the EROS Kernel. Available at: https://archive.fo/IOv2j, accessed 19.10.2025.
[26]. Drepper U. What Every Programmer Should Know About Memory, technical report, 2007. Available at: https://people.freebsd.org/~lstewart/articles/cpumemory.pdf, accessed 06.05.2025.
[27]. McKenney P. E., Appavoo J., Kleen A., Krieger O., Russell R., Sarma D., Soni M. Read-Copy Update, Ottawa Linux Symposium, 2001. Available at: https://kernel.org/doc/ols/2001/read-copy.pdf, accessed 06.05.2025.
[28]. Michael M. M. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, IEEE Trans. Parallel Distrib. Syst., 2004, vol. 15, issue 6, pp. 491–504, DOI: 10.1109/TPDS.2004.8.
[29]. gnumach/ipc/ipc_space.h - Github. Available at: https://github.com/flavioc/gnumach/blob/master/ipc/ipc_space.h, accessed 06.05.2025.
[30]. seL4 Reference Manual Version 13.0.0. Available at: https://sel4.systems/Info/Docs/seL4-manual-latest.pdf, accessed 06.05.2025.

[31]. Grinten A. van der managarm/managarm: Pragmatic microkernel-based OS with fully asynchronous I/O. Available at: https://github.com/managarm/managarm, accessed 06.05.2025.
[32]. Bagwell P. Ideal Hash Trees, technical report, 2001. Available at: https://infoscience.epfl.ch/server/api/core/bitstreams/f66a3023-2cd0-4b26-af6e-91a9a6ae7450/content, accessed 06.05.2025.
[33]. genodelabs/genode: Genode OS Framework. Available at: https://github.com/genodelabs/genode, accessed 06.05.2025.
[34]. The Open Group Base Specifications Issue 7, 2018 edition. Available at: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/, accessed 19.10.2025.
[35]. Walfield N. H., Brinkmann M. A critique of the GNU hurd multi-server operating system, SIGOPS Oper. Syst. Rev., 2007, vol. 41, issue 4, pp. 30–39, DOI: 10.1145/1278901.1278907.
[36]. Kernel Architecture Overview. Available at: https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html, accessed 01.12.2023.
[37]. Loscocco P., Smalley S. Integrating Flexible Support for Security Policies into the Linux Operating System, 2001 USENIX Annual Technical Conference (USENIX ATC 01), USENIX Association, 2001. Available at: https://www.usenix.org/conference/2001-usenix-annual-technical-conference/integrating-flexible-support-security-policies, accessed 07.06.2026.
[38]. SELinux User's and Administrator's Guide. Available at: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/index, accessed 19.10.2025.
[39]. Spencer R., Smalley S., Loscocco P., Hibler M., Andersen D., Lepreau J. The flask security architecture: system support for diverse security policies, Proceedings of the 8th Conference on USENIX Security Symposium, SSYM'99, vol. 8. USENIX Association, 1999, p. 11.
[40]. Learn about Fuchsia. Available at: https://fuchsia.dev/fuchsia-src/get-started/learn-fuchsia, accessed 06.05.2025.
[41]. The Little Kernel Embedded Operating System. Available at: https://github.com/littlekernel/lk, accessed 06.05.2025.
[42]. zircon - fuchsia - Git at Google. Available at: https://fuchsia.googlesource.com/fuchsia/+/master/zircon/, accessed 06.05.2025.
[43]. Grinten A. van der thor and eir - Managarm Handbook. Available at: https://docs.managarm.org/handbook/sys-arch/thoreir/index.html, accessed 19.10.2025.
[44]. Elkaduwe D., Derrin P., Elphinstone K. Kernel design for isolation and assurance of physical memory, Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems, IIES '08. Association for Computing Machinery, 2008, pp. 35–40. DOI: 10.1145/1435458.1435465.
[45]. Lipton R. J., Snyder L. A Linear Time Algorithm for Deciding Subject Security, J. ACM, 1977, vol. 24, issue 3, pp. 455–464, DOI: 10.1145/322017.322025.
[46]. Elkaduwe D., Klein G., Elphinstone K. Verified Protection Model of the seL4 Microkernel, Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments, VSTTE '08. Springer-Verlag, 2008, pp. 99–114. DOI: 10.1007/978-3-540-87873-5_11.
[47]. Kocher P., Horn J., Fogh A., , Genkin D., Gruss D., Haas W., Hamburg M., Lipp M., Mangard S., Prescher T., Schwarz M., Yarom Y. Spectre Attacks: Exploiting Speculative Execution. Available at: https://arxiv.org/abs/1801.01203, accessed 01.12.2023.
[48]. Lipp M., Schwarz M., Gruss D., Prescher T., Haas W., Fogh A., Horn J., Mangard S., Kocher P., Genkin D., Yarom Y., Hamburg M. Meltdown: Reading Kernel Memory from User Space. Available at: https://arxiv.org/abs/1801.01207, accessed 01.12.2023.
[49]. Intel 64 and IA-32 Architectures Software Developer's Manual. Available at: https://web.archive.org/web/20240505001054/https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, accessed 06.05.2025.
[50]. Peters S., Danis A., Elphinstone K., Heiser G. For a Microkernel, a Big Lock Is Fine, Proceedings of the 6th Asia-Pacific Workshop on Systems, APSys '15. Association for Computing Machinery, 2015. DOI: 10.1145/2797022.2797042.
[51]. Chen H., Miao X., Jia N., Wang N., Li Y., Liu N., Liu Y., Wang F., Huang Q., Li K., Yang H., Wang H., Yin J., Peng Y., Xu F. Microkernel goes general: performance and compatibility in the HongMeng production microkernel, Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation, OSDI'24. USENIX Association, 2024.

[52]. The RISC-V Instruction Set Manual Volume II: Privileged ISA. Available at: https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf, accessed 06.05.2025.
[53]. Learn the architecture - aarch64 memory management guide - Size of virtual addresses. Available at: https://developer.arm.com/documentation/101811/0105/Address-spaces/Size-of-virtual-addresses, accessed 06.05.2025.
[54]. Power ISA™ Version 3.1. Available at: https://files.openpower.foundation/s/bo728kgiWfgMHAr, accessed 06.05.2025.
[55]. Hierarchical RCU. Available at: https://lwn.net/Articles/305782/, accessed 06.05.2025.
[56]. McKenney P. E. Memory Barriers: a Hardware View for Software Hackers, technical report, 2010. Available at: http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf, accessed 06.05.2025.
[57]. Alpha Architecture Reference Manual. Available at: https://download.majix.org/dec/alpha_arch_ref.pdf, accessed 06.05.2024.
[58]. Learn about Fuchsia: zx_object_get_child(). Available at: https://fuchsia.dev/reference/syscalls/object_get_child, accessed 06.05.2025.
[59]. Learn about Fuchsia: zx_object_get_info(). Available at: https://fuchsia.dev/reference/syscalls/object_get_info, accessed 06.05.2025.
[60]. Programming reference for the win32 API - win32 apps. Available at: https://learn.microsoft.com/en-us/windows/win32/api/, accessed 19.10.2025.
[61]. Atlidakis V., Andrus J., Geambasu R., Mitropoulos D., Nieh J. POSIX abstractions in modern operating systems: the old, the new, and the missing, Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16. Association for Computing Machinery, 2016. DOI: 10.1145/2901318.2901350.
[62]. Kerrisk M. The Linux Programming Interface. No Starch Press, 2010, 1552 p.
[63]. Drepper U. Futexes Are Tricky. Available at: https://www.akkadia.org/drepper/futex.pdf, accessed 06.05.2025.

Информация об авторах / Information about authors

Евгений Сергеевич БАСКОВ – аспирант ИСП РАН, выпускник магистратуры ВМК МГУ. Сфера научных интересов: операционные системы, микроядерная архитектура, безопасность и производительность операционных систем.

Evgeniy Sergeevich BASKOV – postgraduate student at ISP RAS, CS MSU master's graduate. Research interests: operating systems, microkernel architecture, security and performance of operating systems.

Алексей Владимирович ХОРОШИЛОВ – кандидат физико-математических наук, ведущий научный сотрудник, руководитель Центра исследований безопасности системного программного обеспечения ИСП РАН, доцент кафедр системного программирования МГУ, ВШЭ и МФТИ. Основные научные интересы: методы проектирования и разработки ответственных систем, формальные методы программной инженерии, методы верификации и валидации, тестирование на основе моделей, методы анализа требований, операционная система Linux.

Alexey Vladimirovich KHOROSHILOV – Cand. Sci. (Phys.-Math.), Leading Researcher, Director of the Linux Verification Center at ISP RAS, Associate Professor of System Programming Departments at MSU, NRU HSE, and MIPT. Main research interests: design and development methods for critical systems, formal methods of software engineering, verification and validation methods, model-based testing, requirements analysis methods, Linux operating system.

Александр Константинович ПЕТРЕНКО – доктор физико-математических наук, профессор, заведующий отделом Технологий программирования ИСП РАН, профессор кафедр Системного программирования ВМК МГУ и ФКН НИУ ВШЭ. Научные интересы: формальные методы программной инженерии, операционные системы, языки спецификаций и моделирования, верификация.

Alexander Konstantinovich PETRENKO – Dr. Sci. (Phys.-Math.), Prof., Head of the Software Engineering Department at the Ivannikov Institute for System Programming, Russian Academy of Sciences, Professor of MSU and the Faculty of Computer Science, NRU HSE. Research interests: formal methods of software engineering, operating systems, specification and modeling languages, verification.