# Микроядро L4 как основа ядра ОС

**DATE**: 15-06-2007

**AUTHOR**: Валерий “valerius” Седлецкий (_valerius@mail.ru),

## 0. От автора.

Данная статья прежде всего о перспективах использования L4 в качестве основы для написания ядра OS/2. А также, кроме реализации OS/2 API, цель состоит в интеграции “чужих” API. “Чужие” API, вместе с собственно API ядра OS/2, предлагается реализовать параллельно поверх того же микроядра. За основу взята схема OS/2 Warp PowerPC Edition, реализованная в начале 1990-х гг фирмой IBM.

Текст получился достаточно объемным. – В двух словах об этом рассказать невозможно; о многих обсуждаемых здесь технологиях большинство не слышало и о реальном положении дел с микроядрами многие имеют устаревшее представление; многое за последние 10 лет сильно изменилось. Акцентирую ваше внимание на главе “Обзор проектов на базе микроядра L4”. Она содержит технологии, которые содержат важные идеи и могут являться основой для реализации API. Глава “Планы на будущее” содержит общую схему, в которую входит OS/2 API вместе с API Linux и прочими API. Также полезно для понимания событий, происходящих последние 20 лет в разработке микроядерных ОС, прочитать главы “История вопроса. Понятие микроядра и проблема производительности.” и “Реализации L4”. Если вам неинтересны технические подробности и внутренности микроядра, то теоретический экскурс в механизмы и API L4 и последующие главы, можно пропустить. Надеюсь, статья будет вам полезной, заинтересует и поможет в выборе основы для будущего ядра и подхода в реализации API.

## 1. История вопроса. Понятие микроядра и проблема производительности.

Традиционные операционные системы, в том числе, OS/2 и UNIX, в их нынешнем виде, представляют собой классические монолитные ядра. То есть, управление памятью, файловая система, планировщик процессов, драйвера – все эти службы были собраны в один большой кусок исполняемого кода, исполняющийся с максимальными привилегиями. Несмотря на то, что во многих таких системах были также и подключаемые драйверы в виде отдельных файлов, это по-прежнему был монолит, в котором каждый компонент, если в нем возникает ошибка, мог испортить состояние всей системы. – Все мы знаем ситуацию, когда драйвер может “трапнуть” систему и помогает только перезагрузка.

Уже в 70-х гг. разработчики UNIX столкнулись с проблемой расширяемости монолитного ядра. Увеличивался объем кода в ядре, каждый компонент которого мог испортить другой компонент и мог напрямую менять структуры данных других компонентов. Это усложняло структуру связей между компонентами ядра ОС; взаимодействия между компонентами были очень сложными, и поэтому отлавливать ошибки было очень сложно. Также это было плохо в плане модульности – трудно отделить какой-нибудь компонент ядра и заменить его на другой компонент.

Эти недостатки были частично решены созданием загружаемых драйверов-модулей. Но все равно, оставались проблемы модульности, безопасности и устойчивости.

Все это потому, что

1. ядро очень большое, в нем велика вероятность ошибок
2. компоненты ядра исполняются в одном адресном пространстве и имеют доступ к структурам друг друга.
3. они исполняются на высшем уровне привилегий, поэтому могут сделать что угодно, в том числе, испортить систему
4. взаимоотношения между компонентами ОС не ограничены; каждый компонент может менять структуры данных других компонентов.

Все эти проблемы решаемы, если

1. вынести все ненужное из ядра
2. разнести все компоненты по разным адресным пространствам
3. вынести их на пользовательский уровень привилегий.
4. разделить систему на множество компонентов со строго заданными интерфейсами.

Ключом к этому является концепция микроядра. Микроядерный подход существует уже долгое время, начиная с 80-х гг. 20-го столетия. Одним из первых микроядер было микроядро Mach (Carnegie-Mellon University). Эти исследовательские проекты были направлены на создание надежной, производительной, безопасной, расширяемой и модульной операционной системы. Долгое время удавалось достичь почти всех этих целей, за исключением производительности, которая перечеркивала остальные преимущества. Первые микроядра были очень медленными, и на тогдашнем железе были неповоротливыми.

То есть, на микроядерные системы возлагалось очень много надежд и до последнего времени они не оправдывались. Самым популярным микроядром было микроядро Mach, разработанное в университете Карнеги-Меллона. На его основе было реализовано несколько юникс-подобных систем. В том числе, большие надежды на это микроядро возлагал проект GNU, который использовал его для разработки своей системы HURD. Также известными примерами использования Mach были Mac OS X (использует модифицированное микроядро Mach с обвязкой из BSD-сервисов и драйверов – называется это ядро Darwin); IBM Workplace OS – изначальный грандиозный план объединить на одном десктопе OS/2, AIX, Windows, DOS, AS/400 и прочие системы. Этот план был частично реализован, хотя и в общем закончился неудачей. Из всего этого набора OS Personalities были реализованы только OS/2 Personality и DOS/MVM Personality. Получившийся продукт назывался OS/2 Warp Connect (PowerPC Edition). Он был в конце концов предложен узкому кругу клиентов IBM и широко не рекламировался и не продавался. Эта разработка основывалась также на модифицированном микроядре Mach с обвязкой Personality-Neutral (не зависимых от Personality) сервисами. На основе этих сервисов были построены конкретные personalities.

Также известно, что и у компании Microsoft была попытка использовать микроядро Mach в Windows NT 3.1.

Все эти попытки в целом закончились неудачей (за исключением MacOS X). Главная из этих неудач – очень низкая производительность системы. Несколько исследовательских групп стало изучать причины низкого быстродействия микроядер. Были разработаны компромиссные подходы. Например, в Mac OS X и Windows NT драйверы были вновь включены внутрь ядра, что позволяло достичь более высокой производительности. Известна также группа из MIT (США), разработавшая в итоге этих исследований Экзоядро (Exokernel) и группа из института GMD (Германия) под руководством профессора Йохена Лидтке (Jochen Liedtke), разработавшего в итоге своих исследований микроядро L4. Известно также, что Лидтке позже сотрудничал с

IBM. Экзоядро и по сей день остается очень интересной и многообещающей концепцией, но так и осталось в качестве исследовательского проекта. Подробности опустим, но отметим, что экзоядро – более радикальный путь, чем использование микроядра. В отличие от микроядер, экзоядра еще более минимальны. Они содержат механизмы, но не создают примитивов, а вся ОС реализуется в userlevel в виде библиотек с API. Ядро только разграничивает разные “Library OS”. То есть, под ОС в экзоядерном подходе понимается набор библиотек с API. Exokernel. В отличие от экзоядра, L4 – это не просто исследовательский проект, оно очень бурно развивается. Несмотря на смерть Лидтке в 2001 году, множество его последователей продолжают исследования. На данный момент развитие L4 уже перешло в стадию коммерческих разработок (например, разработка коммерческой версии OKL4 от Open Kernel Labs (Австралия), используемой в Embedded устройствах). ok-labs

Лидтке выдвинул множество очень плодотворных идей для построения действительно эффективного микроядра.

Выяснилось, что ключом к этому является

1. минимальность самогО микроядра, необходимость сохранить в ядре только жизненно важные механизмы и примитивы.
2. зависимость самого микроядра от архитектуры машины и оптимизация под нее
3. оптимизация механизма передачи сообщений (IPC), на котором основано все взаимодействие между частями системы
4. эффективная реализация переключения контекста процессов, в том числе, эффективность переходов в/из ядра в пользовательский уровень.

То есть, производительность не является фундаментальным недостатком понятия микроядра. Оказалось, что если с умом реализовать все необходимые механизмы, то можно достичь очень высокой производительности. Я встречал в литературе несколько общих оценок производительности микроядра для случая L4. Если сравнивать микроядерную систему с монолитной на примере ядра Линукса, согласно общеизвестным фактам (я видел эти цифры в нескольких популярных статьях, в том числе, в статье Таненбаума citkit и в 15), порт Линукса на микроядро Mach под названием MkLinux, медленнее “родного” ядра Linux примерно на 20%. L4Linux – паравиртуализованая версия Linux над микроядром L4 – медленнее всего на 2-5%, что практически неощутимо. Заметим, еще что L4Linux не оптимизирован под микроядерную структуру ОС. – Это просто модифицированное для работы в окружении L4 монолитное ядро линукса. Так что, если реализовать мультисерверный аналог линукса, то производительность могла быть еще больше. Еще заметим, что в отдельных случаях L4Linux может обгонять по производительности native Linux. На заглавной странице L4HQ l4hq висит новость, что по тестам lmbench производительность переключения контекста в Wombat (паравиртуализованная версия Linux от NICTA) для процессора XScale до 30 раз выше, чем в “родном” линуксе, за счет оптимизации производительности переключения контекста в Pistachio-embedded.

Известно также, что производительность механизма межпроцессного взаимодействия (IPC – InterProcess Communication – в данном случае – передача сообщений) в L4 на 1-2 порядка превосходит производительность IPC в Mach. (Более точные и детальные цифры можно найти в статье Лидтке “Towards Real Microkernels”). Также известно, что согласно тестам производительности, L4 обгоняет даже такую заслуженную коммерческую разработку как микроядро QNX Neutrino. Графики с сравнением производительности L4, Mach, QNX и других микроядер я видел в слайдах лекций по L4 Дрезденского Технического Университета (разработчик L4/Fiasco) TU-Dresden.

Что касается минимальности. Если сравнивать Mach и L4, то микроядро Mach было очень громоздким, у него было больше сотни системных вызовов, оно содержало в себе порядка 100000 строк кода, занимало около 500 Кбайт на диске (по крайней мере, версия из OS/2 PPC, которая была в формате ELF и для RISC-процесора). L4 же на 1-2 порядка компактнее. Оригинальная реализация L4 содержала всего 7 системных вызовов, сложность его была порядка 10000 строк кода, на диске оно занимает примерно 100 K (формат ELF, платформа Intel; данные для L4Ka::Pistachio), а вот в памяти оно занимает всего 12 Кбайт! (По другим данным – 32 Кбайт, это зависит от версии ядра; неясно, о какой версии велась речь). Поэтому L4 также часто называют наноядром (nanokernel), поскольку его размер на 1-2 порядка меньше классического микроядра. Кроме того, размер исходников L4Ka::Pistachio подо все платформы вместе – около 8 Мб. Понятно, что только небольшая часть этого собирается под конкретную платформу.

Также, оригинальное микроядро L4/x86, написаное Liedtke, было оптимизировано под 486 процессор и было написано на ассемблере. Современное микроядро L4Ka::Pistachio уже написано на C++ со вставками на ассемблере, и содержит generic часть (независимую от архитектуры), части, специфичные для каждой архитектуры, части, специфичные для отдельных платформ, и склеивающего кода. То есть, присутствует оптимизация под каждую платформу и архитектуру.

Механизм передачи сообщений также был оптимизирован. Были использованы особые ухищрения типа передачи маленьких сообщений в регистрах процессора, передачи строк не копированием, а по ссылке и установки временной области разделяемой памяти для случая передачи больших объемов данных. Производительность операции IPC была достигнута настолько высокой, что время выполнения этой операции сравнимо с временем выполнения системных вызовов в Linux. Также, в L4 отсутствует понятие порта (канала для передачи сообщений, как в Mach). Оно оказалось лишней сущностью, реализация которой понижала производительность.

Ну и механизм переключения контекста тоже был оптимизирован (подробностей, к сожалению, не знаю ;)).

## 2. Реализации L4 и их эволюция.

(Этот раздел является пересказом статьи в английской Wikipedia про L4: L4-wikipedia)

L4 – это не просто название микроядра, а название API и семейства микроядер, основанных на этом API. API L4 имеет несколько различных реализаций и версий. Основными являются на данный момент L4/Fiasco, написанное в Дрезденском Техническом Университете (TUD, Германия, L4/Fiasco) и L4Ka:Pistachio, написанное System Architecture Group университета Карлсруэ – Karlsruhe University (Ka). L4Ka (тоже Германия).

История начинается с начала 1990-х гг.

### a) L3

Оригинальное микроядро, предшественник L4. Разработано Йохеном Лидтке. Доказало правильность идеи, заключающейся в том, что хорошо проработанный легковесный механизм IPC в сочетании с оптимизацией ядра для конкретной платформы дает высокий прирост

производительности. L3 было очень стабильным и производительным, и использовалось в нескольких операционных системах.

### б) L4/x86 ("Lime Pip")

Версия L4, написанная Лидтке на ассемблере. Лидтке увидел, что ядро может быть еще более минимальным, чем L3, и решил написать новое ядро, реализовав множество улучшений в базовых механизмах и в производительности. Поэтому, чтобы ядро было максимально производительным, он решил его написать на ассемблере. L4 произвело революцию в разработке микроядер. Сразу множество организаций и университетов стало изучать L4. В том числе, IBM, где Лидтке начал работать с 1996 г. В IBM Watson Research Center Лидтке и его коллеги продолжили изучение L4 и микроядерных систем в целом.

### в)L4Ka::Hazelnut

В 1999 г. Лидтке возглавил Systems Architecture Group в университете Карлсруэ, где он продолжил исследования микроядерных систем. Как доказательство возможности построения высокопроизводительного микроядра на языке высокого уровня (proof of concept), группа Ka разработала ядро L4Ka::Hazelnut. Оно было написано на C++ и работало на архитектурах IA32 и ARM. Реализация была успешной, производительность замечательной, и с выпуском Hazelnut разработка ядра на ассемблере была прекращена.

### г) L4/Fiasco

Параллельно с разработкой L4Ka::Hazelnut, в 1998 г. в Группе Операционных Систем Дрезденского Технического Университета (группа TUD:OS) была начата разработка своей реализации L4 на C++. Эта разработка была названа Fiasco. По сравнению с L4Ka::Hazelnut и его наследником L4Ka::Pistachio, который позволяет прерывать исполнение ядра только в определенных точках исполнения, Fiasco является полностью вытесняемым (fully preemptible) (за исключением очень коротких атомарных операций), что позволяет достигнуть малой задержки при обработке прерываний (to achieve a low interrupt latency). Это было признано необходимым, так как Fiasco используется в разработке ОС жесткого реального времени DROPS, также разрабатываемой в TU Dresden.

### д) L4Ka::Pistachio

До выпуска L4Ka::Pistachio и новейших версий Fiasco, все версии L4 были тесно привязаны к соответствующей архитектуре процессора. Следующим большим шагом в развитии L4 была разработка переносимого API L4, которое сохраняло высокие характеристики производительности при условии переносимости. При сохранении базовых концепций, новое API вводило много радикальных изменений, улучшающих переносимость, включало поддержку многопроцессорности, исключало тесную взаимосвязь между тредами и адресными пространствами (см. ниже). Также, были введены user-level thread control blocks (UTCB) и виртуальные регистры (см. ниже). Версия X.2 API, в реализации L4Ka::Pistachio, написанном с чистого листа в 2001 году, теперь фокусируется на высокой производительности в сочетании с

переносимостью.

#### е) Новые версии Fiasco

Также, за эти годы большие изменения претерпело ядро Fiasco. Оно теперь поддерживает большое число платформ, включая IA32, IA64, несколько ARM-платформ. Интересно, что была разработана версия Fiasco-UX, которая запускается как user-level приложение Linux. Fiasco реализует ряд расширений L4 V.2 API (см. ниже). Сообщения IPC о происходящих исключительных ситуациях (Exception IPC) позволяют ядру сообщать user-level программам об исключительных ситуациях процессора. Также, были добавлены UTCB, как в API X.2. Кроме этого, Fiasco содержит ряд механизмов для управления коммуникационными правами, а также потреблением ресурсов ядром. На основе Fiasco разработана система user-level сервисов под названием L4Env, которая среди прочего содержит паравиртуализованную версию Linux (версии 2.6.x), называемую L4Linux.

#### ж) разработки Университета Нового Южного Уэльса (UNSW) и NICTA

Также разработка шла в Университете Нового Южного Уэльса, (UNSW, Австралия), где были реализованы версии L4 для нескольких 64-битных архитектур. Так были разработаны L4/MIPS и L4/Alpha, после чего оригинальная работа Лидтке была названа L4/x86. Как и оригинальная работа Лидтке, ядра UNSW были разработаны на смеси C и ассемблера, были написаны с нуля и были непереносимыми. С релизом L4Ka::Pistachio UNSW свернул свои разработки L4 и вместо этого стал разрабатывать очень отлаженные порты L4Ka::Pistachio, включая самую быструю из всех имеющихся, разработку для Itanium (36 циклов процессора за операцию IPC). Эта группа также продемонстрировала, что драйверы в user-level могут давать такую же производительность, как драйверы в ядре, и разработала Wombat – высокомобильную версию Linux для L4, работающую на архитектурах x86, ARM и MIPS. На процессорах XScale Wombat продемонстрировал производительность переключения контекста, в 30 раз более высокую, чем в “родном” Linux. l4hq. Группа ERTOS организации NICTA, куда перебазировались разработчики UNSW, продолжает развивать свою версию Pistachio (NICTA::Pistachio-embedded), направленную прежде всего на embedded и real-time применения.

#### з) разработки Open Kernel Labs

NICTA создала коммерческую компанию под названием Open Kernel Labs (OK Labs), для продвижения на рынке решений (в основном, встраиваемых) на базе L4. OK Labs разрабатывает собственную коммерческую реализацию NICTA::Pistachio-embedded, вместе с Wombat и Iguana. Эта собственная версия L4, Wombat и Iguana носит название OKL4. “Коммерческость” OKL4 не отменяет свободности ее лицензии. OKL4 лицензируется под BSD лицензией и ее исходники открыты.

## 3. Обзор проектов на базе микроядра L4.

L4 построено таким образом, что на его базе можно построить практически любой API. Оно абстрагируется от конкретных алгоритмов управления памяти и прочих стратегий, но

предоставляет механизмы для реализации их (стратегий) в произвольном виде. Также оно поддерживает много аппаратных платформ, имеет поддержку многопроцессорности (SMP) в ядре. Поддерживаются драйверы в user space и запуск одновременно нескольких ОС на одном микроядре. Кроме того, поддерживается создание как ОС реального времени, так и обычных ОС с разделением времени; как ОС для embedded применений, так и desktop OS; можно реализовать различные стратегии планировщика процессов (и управления памятью). Поддерживаются такие возможности как orthogonal persistence (пока это еще нигде не реализовано, но базовые механизмы имеются L4-checkpointing), возможность формального доказательства правильности алгоритмов микроядра (пока не реализовано, но работы ведутся; OK Labs обещает в течение ближайших 2-ух лет выпустить первое микроядро с математически доказанным отсутствием ошибок (formally-proven bug-free microkernel) OK Labs). Все эти возможности предопредилили появление множества проектов операционных систем, основанных на L4. Здесь мы попытаемся кратко описать существующие проекты, чем они могут быть нам полезны. Прежде всего, на мой взгляд, очень важно, что все эти проекты имеют общую базу и возможен их запуск одновременно (Запустить L4Linux и WinCE на одном десктопе с OS/2 – it's cool!). Хотя сейчас и существуют расхождения в API разных реализаций микроядра, но существуют стандарты – они принимаются и разные реализации рано или поздно должны согласовать свой API со стандартом.

Теперь, собственно, о проектах и их пользе для проекта по разработке ядра операционной системы OS/2.

На данный момент существуют три крупнейших центра, разрабатывающих микроядро L4 и проекты на его основе, их мы уже упоминали. Это Университет Карлсруэ, Технический Университет Дрездена и австралийская организация NICTA (в Австралийский центр по разработке L4 входят Университет Нового Южного Уэльса (UNSW) и дочерняя фирма NICTA Open Kernel Labs). Каждый из этих центров параллельно ведут разработки своих версий микроядра, основанных на них userlevel служб, опреационных систем, а также средств разработки.

### 1) Разработки Университета Карлсруэ

Группа из Университета Карлсруэ (“группа Ka”) ведет разработки на основе своего микроядра L4Ka::Pistachio.

#### а) исследования в области persistence

Persistence переводится как “постоянство”. (К сожалению, не знаю русского аналога этого термина). Применительно к компьютерным системам это означает свойство системы сохранять свое состояние после перезагрузок и крахов. Это свойство оказывается очень важным, например, в случае очень длительных непрерывных вычислений; Обычно в этом случае при перерыве в вычислениях программа сама явным образом сохраняет на диск все промежуточные результаты, чтобы потом их можно было продолжить с того же места. Полностью ортогональная persistent система в идеале позволяет программам не заботиться о сохранении промежуточных результатов на диске, а также позволяет пользователю не бояться потери данных при случайном падении системы. Взамен этого, такая система берет заботу об этом на себя, сохраняя промежуточные состояния всех программ пользователя и всех частей системы независимо друг от друга, причем это свойство реализуется прозрачным образом для программ, позволяя им не заботиться о сохранении результатов своей работы.

Применительно к микроядру L4 оказывается, что для реализации orthogonal persistence практически не требуется модификации базовых механизмов микроядра, или же требуются только минимальные их модификации. В работе L4 checkpointing обсуждаются вопросы persistence и механизмы для ее реализации для микроядра L4. Эти механизмы основываются на системе userlevel пейджеров (про пейджеры см. раздел об управлении памятью).

Также, можно привести ссылку по теме: Проект Unununium (вот такое хитрое название) Unununium, и хотя проект этот не имеет отношения к L4, но он имеет отношение к попыткам реализации persistence.

Применение идеи persistence к обычным операционным системам, таким как OS/2 – интересный вопрос; пока неизвестно, можно ли эту идею интегрировать в систему, сохранив как было прежний облик операционной системы.

#### б) технологии виртуализации

Также, большой проект группы L4Ka занимается исследованиями в области применения L4 в качестве основы для системы виртуальных машин. Подобно применению гипервизора Xen Virtualization, L4 также может служить гипервизором, запуская несколько гостевых операционных систем в виртуальных машинах. Эти виртуальные машины не занимаются эмуляцией на уровне инструкций процессора, как виртуальные машины типа VirtualPC, а только ограничивают набор ресурсов, доступных гостевой операционной системе (объем памяти, набор доступных устройств и т. п.), а также используют технологию паравиртуализации. Паравиртуализация – это модификация гостевой операционной системы таким образом, чтобы она работала не с реальным оборудованием, а с абстракциями, предоставляемыми гипервизором. В качестве примера можно привести паравиртуализованый Linux – L4Linux. L4Linux представляет собой обычное ядро Линукса, модифицированное таким образом, что оно работает с оборудованием через абстракции L4. Изменения затрагивают лишь небольшую часть ядра. Например, драйверы Linux не зависят от архитектуры. L4 представляет собой просто еще одну архитектуру, на которую портирован Linux – в его дереве исходников появляется подкаталог l4 в каталоге arch.

То есть паравиртуализованная ОС – это не ОС, запущенная в пробирке, как Windows, запущенная под Virtual PC. Это вполне самостоятельная ОС, но для возможности запуска нескольких ОС одновременно она запускается в очень “тонкой” виртуальной машине, которая 1) абстрагирует оборудование от ОС (подобно тому, как HAL в Windows NT абстрагирует железо от ядра Windows – это тоже разновидность виртуализации) 2) отдает часть оборудования одной ОС, и часть – другой.

Страница проекта исследований виртуализации L4Ka: Virtualization

Для исследований виртуализации проект Virtualization L4Ka использует микроядро Pistachio в комбинации с менеджером ресурсов marzipan. Marzipan запускается как root сервер и позволяет создавать системы взаимодействующих виртуальных машин, разделяющих между собой ресурсы компьютера. Marzipan ведает разделением ресурсов (память, устройства) между виртуальными машинами, а также предоставляет средства для их взаимодействия. Также marzipan позволяет одним виртуальным машинам “публиковать” сервисы, а другим – находить эти сервисы через посредство marzipan.

Marzipan является основой для проектов Drivers и Afterburner.

**i) Проект Afterburner**

Проект Afterburner автоматизирует создание паравиртуализированных версий гостевых операционных систем. Это наиболее важно для тех ядер ОС, которые больше не поддерживаются и не развиваются (например, ядро OS/2, ау, IBM'еры!). Afterburner позволяет достичь нейтральности (независимости) как по отношению к гипервизору, так и по отношению к гостевой ОС. Он позволяет создавать “virtualization-friendly” ядра, которые могут работать без изменений на разных гипервизорах, на данный момент это: L4, Xen v.2, Xen v.3, а также на голом железе. Для этого используется монитор ресурсов marzipan, а также специальный модуль afterburn-wedge. Wedge зависит от гипервизора и для каждого гипервизора он свой. Проект Afterburner быстро развивается. Год назад я скачивал и собирал исходники паравиртуализованного Linux'а, основанного на Afterburner. Тогда мне пришлось применить патчи к исходникам Linux, Xen и binutils (из binutils afterburner использует ассемблер as). При этом ассемблер был запатчен так, что он при компиляции ядра ОС создавал специальный код, где “virtualization-sensitive” инструкции заменялись на особый код, где вставлялись вызовы wedge. С тех пор прошел год и многое изменилось. Я пока не пробовал новые версии afterburner, а из главной страницы Afterburner трудно понять, как сейчас обстоит дело (на странице написано очень двусмысленно и поверхностно, и я до конца так и не понял, как же оно на самом деле работает; надо будет еще раз скачать исходники и посмотреть…). Насколько я понял, дело движется к тому, что в будущем возможно применение этого проекта к бинарникам ядра (!??). Как я понял, “virtualization-sensitive” инструкции патчатся так, что место вокруг этих инструкций забивается инструкциями NOP, оставляя место для последующей замены их другими инструкциями во время выполнения (?) (The Afterburner automatically locates virtualization-sensitive instructions, pads them with scratch space for runtime instruction rewriting, and annotates them.) То есть (как я понял), возможно автоматическое патчение бинарника ядра ОС, и в результате получается ОС, без изменений работающая на Xen, L4/marzipan и на голом железе!. Если это так, то этот проект очень важен для OS/2. Это позволит нам, не имея исходников ядра, запускать его на различных гипервизорах, на данный момент – Xen и L4/marzipan. (хотя, не помешает ли тут то, что многие OS/2-программы содержат 16-битные фрагменты?… )

На Afterburner также основывается своя паравиртуализованная версия Linux от Uni. Karlsruhe. (Кроме нее есть еще две версии L4Linux). По сравнению с исходным линуксом, изменения затрагивают только ядро. Приложения обычного линукса не требуют перекомпиляции и без изменений работают на L4Linux. L4Linux – это только пример паравиртуализованной ОС, “подопытный кролик” для экспериментов в этой области.

**ii) Проект Drivers**

Проект Drivers Drivers позволяет ОС, запущенным в виртуальных машинах на основе L4 и marzipan, использовать драйвера устройств друг друга. ОС, в которой запускаются драйвера назовем Driver OS, а ОС, использующая драйвера первой ОС – Client OS. Внутри Driver OS запускается приложение-маппер, транслирующее запросы от Client OS-ов в формат запросов к ядру Driver OS. Также этот маппер позволяет “мультиплексировать” драйвера Driver OS между несколькими ОС-клиентами. Более того, путем запуска нескольких копий Driver OS, в каждой из которых запускается определенный драйвер, можно изолировать разные драйверы в разных виртуальных машинах. То есть, например, драйвер диска запускается в одной VM, а драйвер сетевой карты – в другой VM. Если первый драйвер упадет, его можно рестартовать вместе с его виртуальной машиной. При этом, падение одной виртуальной машины не затрагивает

вторую VM, то есть драйвера 1) изолированы друг от друга 2) их можно рестартовать. Драйвера запускаются в своем неизменном виде. Кроме того, они запускаются в user level, так что падения драйверов не затрагивают других операционных систем. Этот подход позволяет использовать немодифицированные драйвера legacy-операционных систем. Пока паравиртуализованные версии, для случая широкоиспользуемых ОС, имеет лишь линукс. То есть, этот проект дает возможность использовать прежде всего, драйвера Linux. Возможно создание паравиртуализированной ReactOS, и тогда можно было бы использовать немодифицированные драйвера Windows. (Также возможно, что и Microsoft повернется к нам лицом и снизойдет до обработки Windows при помощи Afterburner – все возможно, открыли же они, например, исходники WindowsCE...). Если удастся Afterburner применить к OS/2, то возможно также использование неизменных драйверов OS/2 путем запуска в виртуальной машине современного ядра OS/2 одновременно с новым, основанным на микроядре (хотя, не является ли тут препятствием их 16-битность?...).

## 2) Разработки NICTA

#### a) Kenge/Iguana/Wombat

Kenge – это набор библиотек и инструментов для разработки. В том числе, для разработки драйверов, работы с ком-портом, консолью и т. п. Реализация очередей, семафоров, работа с таймером, форматами исполняемых файлов и прочее. Инструменты включают свой IDL-компилятор Magpie, а также build environment, основанный на scons. Scons – это очень удобная и оригинальная утилита автоматизации сборки, подобная make, которая зависимости между исходниками строит на специальном объектно-ориентированном языке. (“объектно-ориентированные мейкфайлы”). Написана scons на языке python.

Kenge содержит уже готовый набор userlevel сервисов, составляющих минимальную OS Personality, необходимую для реализации минимальной инфраструктуры: Name server, root server, сервер, реализующий поддержку последовательного порта. Эта минимальная OS Personality нацелена прежде всего на встраиваемые применения, содержит минимальную инфраструктуру для управления памятью и защиты. Имеет малую нагрузку на кеш (small cache footprint). Также, позволяет работать на системах без поддержки виртуальной памяти, что часто требуется во встраиваемых приложениях. Также имеется своя подсистема драйверов. Этот набор userlevel сервисов называется Iguana Embedded OS Personality.

На основе сервисов Iguana реализован также высокомобильный Linux server под названием Wombat.

#### б) L4/Darwin aka Darbat

По аналогии с Wombat (паравиртуализованный Linux) существует проект под названием Darbat, или L4/Darwin. Это ни что иное как порт ядра Darwin, на котором основана MacOS X, на микроядро L4. Darbat пытается улучшить производительность Darwin за счет использования IPC L4, которое намного более производительное, чем IPC микроядра Mach, на котором основан Darwin. Также Darbat содержит порт IOKit на L4, который стремится получить выгоду от использования user-level драйверов (в Darwin многие драйвера работают в режиме ядра, а в Darbat они выносятся в userlevel). Одной из целей является поддержка немодифицированных

драйверов Darwin в userspace. Darbat не стремится полностью удалить Mach из Darwin, а взамен этого получить версию MacOS X, почти без изменений работающую под L4, и использующую выгоды, которые дает L4 по сравнению с Mach. Проводятся эксперименты по параллельному Запуску Darbat и Wombat, а также их взаимодействию.

Darbat – в первую очередь, надежда для макинтошников. Но и для нас было бы достаточно заманчивым исполнять приложения MacOS X на одном десктопе с нашей OS. Это еще один параллельный API, вдобавок в Linux и (возможно в будущем) ReactOS.

Разработки NICTA на основе Pistachio выпускаются на основе BSD-лицензии, так же как и само ядро Pistachio.

### 3) Разработки TU Dresden

TU Dresden на основе своего микроядра Fiasco разрабатывает ОС DROPS под GNU-лицензией. DROPS содержит большой набор библиотек и userlevel сервисов под названием L4Env (L4 Environment). L4Env также содержит средства разработки, например, IDL-компилятор DICE.

L4Env содержит следующие сервисы: 1) консоль, поддержка виртуальных консолей 2) сервер логов. Позволяет собирать в одном месте отладочные сообщения от различных сервисов. В сочетании с libc и службой имен l4vfs реализует поддержку stderr/stdout для неинтерактивных сервисов. 3) l4vfs – иерархический Name server, который интегрирует в одно дерево службы, файлы, терминалы и позволяет реализовать в libc файловые функции типа open(), read(), write() через специальные бэкенды. Поддерживаются две реализации libc – порты dietLibc и uCLibc. 4) поддержка простейшей работы с файлами без использования файловой системы, например, получение файлов с tftp-сервера или с файловой системы L4Linux. 5) Поддержка запуска исполняемых файлов формата ELF. 6) Управление памятью. Реализуются свои примитивы работы с памятью поверх примитивов микроядра: концепция Dataspaces и dataspace managers. 7) поддержка примитивов синхронизации, работа с таймерами, семафорами. 8) Менеджер задач 9) доступ к оборудованию 10) Подсистема драйверов DDE (Device Driver Environment)

DDE включает в себя поддержку драйверов Linux (DDE/Linux) и FreeBSD (DDE/FreeBSD). Эта поддержка реализует подход, альтернативный подходу в L4Ka Drivers. Драйвера погружаются в специальный слой, эмулирующий окружение ядра Linux (а не запускает реальное ядро Linux).

Все это может быть полезно для реализации своих userlevel сервисов (прежде всего Personality-neutral сервисы, не завязанные ни на какую ОС), а также поддержки оборудования.

Также, идея реализации нескольких GUI на одном десктопе содержится в проекте Nitpicker в рамках разработки DROPS. Nitpicker – это идея GUI минимальной сложности, реализующего минимальные примитивы для рисования окон. Nitpicker сделан так, что различные оконные системы (например, родной оконный менеджер DoPE системы DROPS, а также X Window из L4Linux) могут являться клиентами Nitpicker. Nitpicker работает на более низком уровне, чем GUI, рисующие окна. Примитивами для рисования являются buffers и views. Buffer – это область шаренной памяти, содержащая 2-мерные графические данные. Область памяти выдается Nitpicker'у клиентом через разделяемую память. Вместо понятия окна Nitpicker содержит более простое понятие, которое называется view. View-это прямоугольная область на экране, содержащаяся внутри буфера. Каждый view имеет определенный размер и положение на экране, определяемое клиентом. Views могут перекрывать другие view в том же буфере. То есть, views внутри буфера, предоставляемого каждым клиентом, могут иметь определенный Z-order (т.е., кто над кем находится – расположение по 3-ей оси координат). То есть, каждый

клиент nitpicker рисует внутри своего буфера, а не непосредственно на экране. Nitpicker собирает воедино все буферы и рисует все views, в них содержащиеся, на экране, собирая все в одну картинку. Кроме того, Nitpicker пытается изолировать клиенты друг от друга, следя за безопасностью работы, что позволяет затруднить работу различных key loggers и прочих троянов, записывающих нажатия клавиатуры и делающих снимки экрана. Подробнее про это см. на сайте DROPS demo CD TUD:OS Demo CD. Здесь для нас ценной является сама идея сочетания различных оконных систем в одном десктопе. Реально попробовать это в работе и поиграться с DoPE и XWindow на одном десктопе можно, загрузив DROPS Demo CD TUD:OS Demo CD.

#### 4) Прочие разработки

##### а) SawMill Linux

Разработчики: IBM Watson research center, основан на L4/x86 (aka Lime Pip). Проект больше не развивается, развитие закончено в 2001 году. Проект направлен на разработку Linux personality, построенной на мультисерверной основе. То есть, цель – создать модульный, мультисерверный, а не монолитный Linux, использовав преимущества микроядра. Что этот проект дает: лучшая Linux personality, чем L4Linux. Минусы: проект не развивается. Sawmill Linux

##### б) L4Minix

Порт Minix на L4Ka::Hazelnut. Разработчик: National Institute of Informatics (Япония). Развивается с 2001 года, последнее обновление было в 2002 году. Проект похоже, больше не развивается. Что проект дает: еще одна OS Personality.

## 4. Идеи на будущее

- Создать реализацию OS/2 API вместе с возможностью работы параллельных API других операционных систем, прежде всего – Linux и Windows (на основе ReactOS и L4Linux). Как OS/2 API, так и API других ОС реализуются как параллельные personalities.
- Разграничение различных personalities и раздача им ресурсов компьютера реализуется на основе marzipan resource monitor. На основе Marzipan реализуется менеджер аппаратных ресурсов (HRM) и root server, запускающий personality neutral (PN) сервисы. В PN сервисы входят независимые от конкретной ОС службы: 1) console server – драйвер консоли, независимый от ОС 2) сервер логов – сбор отладочных сообщений от всех серверов различных personalities. 3) Иерархический сервер имен. 4) поддержка устройств при помощи подсистем драйверов. 5) Поддержка драйверов файловых систем (IFS) 6) Поддержка драйверов исполняемых форматов IIF (Installable Image format), наподобие IFS
- Подсистемы драйверов входят в PN сервисы. Общий менеджер поддержки оборудования, в который подключаются подсистемы драйверов. “Родная” подсистема драйверов, входящая в PN сервисы. К “родной” подсистеме добавляются driver frameworks, реализованные внутри OS Personalities, прежде всего L4Linux и ReactOS. Общая схема примерно такова: Конкретная OS personality либо реализует подсистему драйверов внутри себя, как L4Linux, либо (что более желательно) обращается с запросами к

менеджеру поддержки оборудования. Менеджер оборудования перенаправляет запрос либо нативному драйверу конкретного устройства (в “родной” поддержке оборудования), либо мапперу внутри Driver OS (используется идея из проекта Drivers L4Ka). Таким образом, есть как native drivers, которые предпочтительно использовать, либо, для подстраховки на случай отсутствия native драйверов – возможность использования драйверов внутри OS personalities, прежде всего, L4linux.
- OS/2 Personality реализуется на базе примитивов, реализованных в PN services. То есть, API OS/2 – более высокоуровневый. OS/2 API server реализует семантику файловой системы OS/2 (буквы дисков, доступ к драйверам устройств через файловую систему – “файлы” в каталоге \DEV\, пайпы \PIPE\, и проч.) на основе менеджера файловых систем, содержащегося в PN сервисах. Также, реализуется VIO/KBD/MOU на основе примитивов, предоставляемых PN сервисами (например, реализация VIO поверх консоли, предоставляемой console server.)

Идея построения системы, основанной на PN-сервисах, заимствована из OS/2 PowerPC. (См. OS/2 PPC, first look)

Универсальный эмулятор x86, набазе которого строятся 1) исполнение 16-бит инструкций в LX-файлах 2) реализуется эмуляция DOS, возможно, на базе FreeDOS. (Прототип: MVM Personality из OS/2 PPC)

Единый десктоп для всех personalities. Предлагается использовать идею из Nitpicker для интеграции разных оконных систем в один десктоп. То есть, PM и XWindow работают через примитивы, предоставляемые аналогом Nitpicker.

## 5. Механизмы и API L4.

Здесь я очень кратко (подробно не получится – это выходит за рамки статьи) расскажу об идеях, на основе которых реализовано L4. Подробнее см. в L2, v. X.2, L4 user manual (MIPS case), L4 user manual (v. X.2), TU Dresden, слайды лекций, UNSW, слайды лекций. Прошу прошения за неточности, если такие встретятся. (“Я не волшебник, я еще только учусь” © ;)).

### 1) Основные понятия и абстракции

Микроядро реализует всего три абстракции (остальное реализуется в userlevel на их базе):

1. Тред – исполняемая и диспетчеризуемая единица. Имеет свой контекст – состояние регистров процессора.
2. Адресное пространство – Область виртуальной памяти, в которой исполняется несколько тредов
3. IPC (InterProcess Communication) – средство взаимодействия между тредами.

Под задачей (task) понимается адресное пространство с исполняемыми в нем тредами.

Также, к одним из основных механизмов микроядра относится планирование процессов (scheduling). Микроядро содержит базу для реализации scheduling.

### 2) Треды

Треды идентифицируются числами – Thread Id; дабы не плодить сущностей, адресное пространство идентифицируется по номеру любого принадлежащего ему треда.

Thread ID'ы могут идентифицировать обычные треды, виртуальные Interrupt треды, а также есть специальные константы типа NilThread (идентифицирует “никакой” тред), AnyThread (идентифицирует любой тред), они служат для задания значений ThreadId в API-функциях, например, в операциях IPC. Виртуальные Interrupt треды применяются для виртуализации обработки прерываний (см. ниже).

Тред называется локальным, если он принадлежит этому же адресному пространству. Локальные треды адресуются специальными значениями ThreadID – значение ThreadID имеет особую маску битов.

Треды могут взаимодействовать через IPC. Операция IPC передает сообщение от треда к треду. При этом посылающий и получающий сообщение треды в этой операции идентифицируются по Thread ID'ам.

Тред оперирует памятью и регистрами. Память может быть физической и виртуальной. Виртуальная память поделена на независимые адресные пространства. (Про конструирование адресных пространств см. ниже в разделе “управление памятью”).

Каждый тред имеет свой контекст – регистры IP, SP, FLAGS, TCRs (Thread Control Registers, см. ниже). Также, каждый тред имеет ассоциированные с ним пейджер, шедулер, а также exception handler. Контекст треда сохраняется в ядре в области, называемой TCB (Thread Control Block), причем часть TCB сохраняется в специальной области UTCB (User mode TCB), содержащейся в адресном пространстве самогО треда – она открыта для доступа из usermode.

### 3) Управление памятью (flexpages, адресные пространства и их рекурсивное конструирование)

Адресное пространство состоит из страниц памяти. Интересно, что L4 оперирует виртуальными страницами, независимыми от архитектуры – они называются flexpages. Flexpages имеют размер, больший или равный аппаратной страницы памяти процессора, и кратный степени двойки. Сама страница также должна быть выровнена по своему размеру и сам размер должен быть не меньше 1 K. Flexpages рассматриваются как неделимые объекты.

Базовое “плоское” адресное пространство, отображенное идемпотентно (то есть, один в один) на физическую память, называется sigma0, как и одноименный сервер.

Страницы памяти (flexpages) могут отображаться из одного адресного пространства в другое. После такой операции, называемой mapping, страница памяти становится доступной в обеих адресных пространствах. Так реализуется разделяемая память. Кроме операции mapping есть еще операция передачи страницы памяти (granting). Она отличается от предыдущей тем, что адресное пространство A, предоставляющее страницу памяти адресному пространству B, теряет доступ к этой странице. То есть, страница передается насовсем. Эта операция, в отличие от предыдущей, необратима.

Отображение (mapping) или передача (granting) страницы памяти осуществляется путем

посылки специального IPC-сообщения, в котором закодирован набор flexpages (MapItems и GrantItems). Ядро перехватывает посылку такого сообщения и при этом реализует операцию map/grant. То есть, операция map/grant осуществляется как побочный эффект передачи специального сообщения, а не при помощи API-функции. Напротив, для обратной операции unmap предназначен специальный системный вызов Unmap(). Операция Map (как и Grant) реализована таким образом, что процесс-получатель имеет возможность указать окно, ограничивающее область, в которую необходимо отобразить flexpages.

Операции mapping, granting и unmapping реализуются динамически, то есть, во время выполнения. Они основаны на операциях с таблицами страниц процессора, которые осуществляются динамически по запросу программ.

Адресные пространства конструируются рекурсивно. Иначе говоря, есть исходное адресное пространство sigma0. Задача sigma0 отдает области памяти серверам A и B путем операции granting. A передает (через map или grant) части полученной памяти серверам A1 и A2, B – передает B1 и B2. Te – передают дальше по цепочке. Получается иерархическая система менеджеров памяти. В результате адресные пространства могут конструироваться произвольным образом из исходного плоского адресного пространства (sigma0), и на этом основано управление памятью. Все эти операции доступны обычным непривилегированным userlevel программам. То есть, управление памяти реализуется в userlevel.

Виртуальная память, подкачка с диска и обработка page faults (страничных отказов) производится при помощи userlevel серверов, называемых пейджерами (pagers). Каждый тред имеет ассоциированный с ним пейджер. Когда тред обращается к странице памяти, которая еще не отображена в его адресное пространство, возникает page fault (иначе говоря, исключение процессора TRAP 000e). Обработчик page fault'a в ядре перехватывает управление и при этом посылает специальное IPC-сообщение пейджеру треда. Это сообщение отправляется ядром от имени сбойнувшего треда (это называется “обманывающее IPC” – “deceiving IPC” а сам тред называется virtual sender). Пейджер реализует какую-то логику обработки page fault'ов и может решить отобразить flexpages в адресное пространство сбойнувшего треда. Это, как мы уже говорили, также реализуется через IPC. – Пейджер просто отвечает на сообщение о page fault'e сообщением с отображаемыми flexpages.

Пейджеры могут обращаться к менеджерам памяти, а также часто бывает, что обе роли совмещены в одной программе.

### 4) IPC и язык IDL (использование методов распределенных систем)

Базовым методом межпроцессного взаимодействия является механизм передачи сообщений, реализованный в ядре. Также важным механизмом IPC является разделяемая память. Но собственно аббревиатура IPC обычно подразумевает именно передачу сообщений.

Передача сообщений является синхронной и не буферизованной (synchronous and unbuffered). Т.е., посылающий тред блокируется, пока получающий не войдет в фазу receive и не получит сообщение. – Системный вызов IPC() объединяет две фазы – send и receive. обе из которых опциональны (определяется параметрами вызова IPC()). Один из тредов должен находиться в фазе send, другой в фазе receive. Получающий тред должен предоставить буферы для сообщения; в ядре сообщение не буферизуется.

Синхронность и не буферизованность IPC является одной из причин высокой производительности IPC в L4. Напротив, в микроядре Mach IPC было асинхронным и

буферизованным (IBM-овское микроядро, основанное на Mach, на базе которого была построена OS/2 PPC, содержало улучшения. IPC в нем было изменено на синхронное, что дало повышение производительности OS/2 PPC, first look).

Также следует отметить, что сообщения в L4 не проверяются на правильность, в отличие от Mach.

Также, как мы уже упоминали, в L4 отсутствует понятие порта, как в Mach, что упрощает IPC и дает прирост производительности.

Операции IPC могут осуществляться вручную, но здесь для программиста очень много рутинной работы. Поэтому для автоматизации этих операций служат генераторы интерфейсов. Генераторы интерфейсов используют методы распределенных систем – язык IDL (Interface Definition Language). Использование IDL можно встретить в OS/2 Warp Toolkit, если вы программируете SOM-приложения. Использование языка IDL неслучайно. Как и в распределенных системах (Corba/DSOM, COM, DCE), микроядерная система состоит из множества объектов (серверов), взаимодействующих между собой при помощи протоколов, передающих сообщения через жестко заданные интерфейсы. И во всех этих случаях есть: 1) Фаза сборки сообщения и его посылки на стороне клиента, а также получение и разборка ответов сервера и их обработка. 2) На стороне сервера есть цикл ожидания, разборки и обработки сообщений от клиентов.

Обе эти фазы автоматизируются IDL-компилятором (IDL Compiler, Interface Generator). Для этого внешние интерфейсы сервера кодируются на языке IDL. IDL-компилятор обрабатывает исходник на языке IDL и по нему строит 1) client-side stub 2) server loop. Все это делается автоматически и на выходе компилятора IDL получаются файлы на языках C, C++, Ada и проч.

Для случая языка C на выходе IDL-компилятора получается 3 файла. 1) client-side stub 2) server loop 3) хедер, включаемый в вашу программу.

Для микроядра L4 существует несколько разных IDL-компиляторов, на данный момент известны такие компиляторы как, IDL4, DICE, Magpie а также Flick. Они генерируют высокопроизводительные и оптимизированные stub-ы, производительность которых сравнима с производительностью при ручной оптимизации. Stub code becoming important

### 5) Виртуальные регистры

Кроме памяти, треды имеют дело также с регистрами. L4 реализует виртуальные регистры, независимые от архитектуры. В зависимости от конкретного типа процессора, виртуальные регистры могут отображаться как на реальные регистры процессора, так и на области памяти. Или же часть из них отображается на память, а часть – на регистры процессора.

Существует 3 класса виртуальных регистров:

| | |
|---|---|
| TCRs (Thread Control Registers) | – служат для хранения контекста треда |
| MRs (Message Registers) | – служат для хранения IPC-сообщений в операции передачи/приема |
| BRs (Buffer Registers) | – служат для хранения буферов при передаче строк |

### 6) API

API L4 содержит очень небольшое количество функций. В оригинальной версии ядра их было всего 7. Для API версии X.2 (L4Ka::Pistachio) имеется 11 системных вызовов. Системные вызовы делятся на обычные и привилегированные. Привилегированные могут вызываться только привилегированными тредами (т.е., тредами, исполняющимися в адресном пространстве roottask)

Вот что они собой представляют:

| | | |
|---|---|---|
| 1. | KernelInterface() | позволяет получить адрес таблицы KIP |
| 2. | ExchangeRegisters() | заменяет или возвращает состояние регистров треда |
| 3. | ThreadControl() | служит для добавления или удаления тредов в адресное пространство (привилегированный системный вызов) |
| 4. | SystemClock() | получает системное время. Обычно не требует перехода в kernel mode (это удивительно, но так написано в документации!). |
| 5. | Schedule() | используется шедулерами для задания значения кванта времени, приоритета и других параметров |
| 6. | ThreadSwitch() | вызывающий тред освобождает процессор и дает возможность передачи управления другому треду |
| 7. | Unmap() | выполняет операцию unmap для указанных flexpages (виртуальных страниц памяти) |
| 8. | SpaceControl() | управляет адресными пространствами (привилегированный системный вызов) |
| 9. | IPC() | выполняет операцию приема/передачи сообщения. Существует в двух версиях просто IPC и LIPC – Lightweight IPC – специальная версия этого системного вызова, оптимизированная для передачи между тредами, исполняющимися в одном и том же адресном пространстве. В остальном обе версии идентичны. |
| 10. | ProcessorControl() | управляет параметрами процессора(-ов). Может менять внешнюю и внутреннюю частоту процессора и его вольтаж (привилегированный системный вызов) |
| 11. | MemoryControl() | управляет атрибутами страниц памяти (привилегированный системный вызов). |

Каждая из этих функций объединяет множество вариантов их использования.

### 7) Механизмы безопасности

Одним из методов разграничения различных подсистем и OS Personalities является механизм Кланов и Шефов (Clans and Chiefs). Если задача A создала задачи B, C и D, то сама задача A становится их шефом, а сами задачи B, C и D образуют клан. Каждая задача может создать свой клан, для которого она будет являться шефом. На кланах и шефах основывается разграничение подсистем – задача B может слать сообщения только членам своего клана, либо своему шефу. Если B попытается послать сообщение задаче, не входящей в ее клан, то оно (сообщение) принудительно направляется шефу. Шеф может как “дропнуть” сообщение, так и перенаправить его шефу другого клана (а он уже – членам своего клана). Кланы могут быть вложенными. Этот механизм разграничивает разные подсистемы и ОС-ы путем перенаправления потока сообщений IPC.

Также, вторым механизмом обеспечения безопасности является принцип: серверы (например, roottask и sigma0) предоставляют ресурсы первому запросившему клиенту (например, sigma0 отдает память первому запросившему серверу-менеджеру памяти), осталь- ным они отказывают в доступе. Так как первыми ресурсы запрашивают начальные серверы, список которых определяет системный администратор или разработчик дистрибутива ОС, то кто угодно не может запросить ресурс и завладеть им.

Также, различные стратегии доступа к оборудованию, портам и памяти могут быть реализованы через user-space пейджеры и менеджеры памяти.

## 8) Работа с оборудованием и драйверы в userlevel

Для случая микроядра драйвер – это обычная userlevel программа, исполняющаяся в ring3. Естественно, драйвер должен получить ресурсы – IRQ, на которые он вешается как обработчик прерывания и порты либо память, отображенную на устройство. Получает он эти ресурсы от специального сервера-менеджера ресурсов. (А сам этот менеджер ресурсов получает их у микроядра и sigma0 на правах первого запрашивающего сервера). Уже реализованы специальные менеджеры аппаратных ресурсов, и существует более-менее общепринятый интерфейс к таким менеджерам. Например, в l4env роль такого менеджера ресурсов выполняет сервер l4io; сам общепринятый интерфейс называется omega0. Omega0 также реализует интерфейс к контроллеру прерываний (например, посылка сигнала EOI) и работу с PCI configuration space и т. п.

#### а) обработка прерываний

Если приходят прерывания от устройства, ядро перехватывает их, и при этом направляет IPC-сообщение треду-обработчику прерывания. Причем, это сообщение приходит от имени несуществующего (виртуального) треда с особым значением Thread ID – Interrupt Thread. Таким образом реализована обработка прерываний в user mode – сигналы прерывания виртуализуются через посылку сообщений IPC, приходящих от виртуальных тредов.

#### б) работа с портами и memory-mapped i/o.

На большинстве архитектур регистры устройств отображаются в обычную физическую память. Поэтому сервер может запросить такую память в виде обычных flexpages у менеджера памяти, например, sigma0. На платформе PC существует отдельное адресное пространство портов ввода-вывода. В этом случае для этого адресного пространства также можно выполнять операции map/grant и unmap. Для этого служат специальные i/o flexpages. То есть, серверы (драйверы) могут запрашивать стандартным образом доступ к портам у менеджера памяти, управляющего портами, через операции map/grant. Гранулярность i/o flexpages составляет 1 байт.

Для случая доступа к i/o портам также возможно возникновение page fault'ов при доступе к неотображенной области адресного пространства портов. Обрабатывает такие page fault'ы также пейджер, ассоциированный с тредом драйвера устройства. Этот пейджер должен следовать специальному протоколу.

## 6. Общая структура, из чего состоит минимальная L4 система.

Минимальная система состоит из собственно микроядра, загрузчика kickstart, корневого менеджера памяти sigma0 и корневого сервера (rootserver). Только микроядро исполняется на высшем уровне привилегий. roottask и sigma0 исполняются в userlevel. Roottask – базовый сервер, отвечающий за инициализацию системы, а также за выполнение привилегированных операций. Sigma0 – начальный менеджер памяти, который владеет всей физической памятью системы. Sigma0 может отдавать страницы физической памяти другим серверам, реализуя для этого свой простейший протокол взаимодействия.

## 7. Инициализация системы

Для загрузки системы, основанной на L4, традиционно используется загрузчик GRUB и стандарт Multiboot. При этом GRUB загружает программу kickstart – спец. дополнительный загрузчик, выступающий в роли multiboot-ядра, и набор дополнительных модулей. Первым модулем должно быть ядро L4, вторым – sigma0, третьим – roottask, и далее следуют добавочные модули. Kickstart грузит L4, sigma0 и rootserver из их образов в памяти, полученных от GRUB'а, делает fixups, заполняет поля в области под названием Kernel Interface Page (KIP). KIP находится внутри образа микроядра L4 в памяти. Она очень важна для системы и содержит адреса начальных серверов, адреса точек входа системных вызовов микроядра, таблицы дескрипторов областей памяти и дополнительных модулей, загруженных GRUB'ом, а также прочую информацию (все эти поля заполняет kickstart и передает ядру). Затем kickstart запускает микроядро. Микроядро стартует начальные серверы – sigma0 и roottask. Roottask после этого инициализирует все userlevel сервисы системы, в том числе, различные personalities. Дополнительные модули, загруженные GRUB'ом, тоже могут стартоваться roottask'ом, причем ссылка на таблицу с адресами этих модулей находится в KIP, откуда Roottask и получает информацию о их местонахождении. Менеджеры памяти при инициализации системы получают всю память от sigma0.

Для создания более подходящего, чем GRUB, загрузчика, использующего характерные для OS/2 механизмы загрузки, например, использование microfsd, проектом osFree решено было совместить механизмы microfsd с логикой загрузки, использующей стандарт multiboot. Наш загрузчик называется FreeLdr, он находится в процессе разработки. Загрузка L4 уже почти реализована.

## 8. Ссылки:


| | | |
|---|---|---|
| [1] | Jochen Liedtke, “Towards Real Microkernels” | http://www.l4ka.org/publications/1996/towards-ukernels.pdf |
| [2] | Andrew Tanenbaum, перевод статьи на citkit: “Вторая часть Марлезонского балета” | http://citkit.ru/articles/359/ |
| [3] | Слайды лекций, TU Dresden | http://www.inf.tu-dresden.de/index.php?node_id=1317&ln=en |
| [4] | Слайды лекций, UNSW | |
| [5] | OKL4: | http://www.ok-labs.com/ |
| [6] | L4Ka: | http://l4ka.org/ |
| [7] | L4HQ: | http://l4hq.org/ |

| | | |
|---|---|---|
| [8] | TUD OS Research: | http://os.inf.tu-dresden.de/index.php?node_id=1421&ln=en |
| [8] | NICTA/ERTOS: | http://www.ertos.nicta.com.au/research/ |
| [9] | Wikipedia – L4: | http://en.wikipedia.org/wiki/L4_microkernel_family |
| [10] | Wikipedia – Exokernel: | http://en.wikipedia.org/wiki/Exokernel |
| [11] | L4 API version X.2 manual | http://l4hq.org/docs/manuals/l4-x2-20061117.pdf |
| [12] | L4 user manual (оригинальная версия, речь идет про архитектуру MIPS) | ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9801.pdf |
| [13] | L4 user manual (X.2) | http://www.ertos.nicta.com.au/publications/papers/L4UM_x.2.pdf |
| [14] | Transparent orthogonal checkpointing through using user-level pagers: Espen Skoglund, Christian Ceelen, and Jochen Liedtke | http://l4ka.org/publications/2000/l4-checkpointing.pdf |
| [15] | Производительность L4 по сравнению с монолитными ядрами – … | |
| [16] | Stub code performance is becoming important | http://www.l4ka.org/publications/2000/stubcode-performance.pdf |
| [17] | Версии API L4: | http://l4hq.org/kernels/ |
| [18] | TUD OS/DROPS demo CD: | http://demo.tudos.org/ |
| [19] | OS/2 Warp Connect (PowerPC Edition): first look | http://hobbes.nmsu.edu/pub/os2/info/os2ppc.zip (текстовая версия”) http://www.os2site.com/sw/info/redbooks/os2ppc-sg244630.zip |
| [20] | Проект Unununium: | http://unununium.org |
| [21] | L4Ka Virtualization project | http://l4ka.org/projects/virtualization/ |
| [22] | Проект Afterburner | http://l4ka.org/projects/virtualization/afterburn/ |
| [23] | Проект Drivers | http://l4ka.org/projects/virtualization/drivers.php |
| [24] | Minix3 | http://www.minix3.org/ (Главный сайт Minix3) http://www.minix3.ru/ (Русскоязычный сайт Minix3) |
| [25] | L4Minix | http://research.nii.ac.jp/H2O/index-e.html |
| [26] | Sawmill | http://www.research.ibm.com/sawmill/ |


- **Замечание**: Данная статья была впервые опубликована здесь: Микроядро L4 как основа ядра ОС