МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное

учреждение высшего образования

«Ярославский государственный университет им. П.Г. Демидова»

Математический факультет

Кафедра компьютерной безопасности

Заведующий кафедрой:

д. ф. - м. н., профессор

Мурин Д. М.

Направление подготовки

10.05.01 Компьютерная безопасность

Курсовая работа

Микросервисная операционная система нового времени.

Научный руководитель

д. ф. - м. н., доцент

\_\_\_\_\_\_\_\_ Д. А. Савинов

«\_\_» \_\_\_\_\_\_\_\_\_ 2024 г.

Студент группы КБ-41СО

\_\_\_\_\_\_\_\_ А.Ю.Сластухин

«\_\_» \_\_\_\_\_\_\_\_\_ 2024 г.

Ярославль 2024 г.

Оглавление

Оглавление 2

Введение 3

Микросервисная архитектура (MSA). 5

Микросервис (MS) 5

Микросервисная архитектура (MSA) 5

Сравнение монолитной и микросервисной архитектуры. 6

Операционная система MINIX 10

Простота 10

Полнота 10

Эффективность 10

Безопасность 11

Архитектура MINIX 3 12

Программные ограничения – это правильно 13

Ограничения MINIX 3 14

Модель программирования в MINIX 14

Выводы по MINIX 3 15

Сравнение путей развития MINIX, LINUX и MS-DOS 15

Сравнение путей развития MS-DOS, Linux и MINIX 16

Возрождение MINIX 17

Intel ME 18

Литература 19

Введение

В настоящее время наблюдается тенденция увеличения производительности компьютеров за счёт уменьшения вычислительных модулей. И если раньше, компьютер занимал целые помещения, то сейчас вполне умещается в маленьких коробочках. Но с каждым годом требуемые мощности для игр, развлечений, прикладных нужд растут, а возможность уменьшать, как доказано, не бесконечна.

Мы вполне можем предсказать, что будет в будущем, посмотрев на промышленное применение компьютеров – современные технологии приходят в промышленность и только потом в руки простых пользователей.

Сейчас для выполнения серьёзных промышленных вычислений приходится арендовать мощные удалённые сервера, которые уже неудобно поддерживать в рамках компании. В последние годы набирают популярность распределённые системы на базе виртуальных машин, такие как Kubernetes.

Возможно, что скоро это дойдёт и до пользователей персональных компьютеров.

В ближайшем будущем следует ожидать увеличение размеров компьютеров вплоть до размеров первых вычислительных машин и возращение облачного использование вычислительных мощностей (сейчас это используют для игр) – люди дома будут иметь терминалы, а использовать вычислительные возможности провайдеров, платя за используемые ресурсы по тарифу.

На данный момент самые популярные операционные системы Windows и Linux по своей архитектуре – монолитные, все необходимые части находятся в одном блоке, это обеспечивает высокую скорость работы, простоту ремонта, но при этом имеет массу проблем:

* Проблема устойчивости – уничтожение монолитного блока приведёт к уничтожению всей системы. Вирус, попавший в систему по сети, может получить доступ к файловой системе. Шпион, получив доступ к монолиту, может получить полный контроль над системой.
* Проблема масштабируемости – при необходимости увеличения производительности потребуется увеличивать размер монолита, но ОС зачастую трудно переносят изменения в монолите.
* Проблема распределения логики сложной системы по ядру – когда для того, чтобы полностью исключить определённую логику необходимо пересмотреть весь код ядра на наличие неявных зависимостей – микросервисная архитектура лишена подобной проблемы при правильном построении.
* Код современных ОС состоит из нескольких миллионов строк (Например, ядро ОС Windows Vista содержит около 5млн строк кода) – человек в одиночку не может его написать. Более того, важнее уместить в голове всю картину – из-за размеров это тоже невозможно.
* В коде современных ОС существуют участки так называемого легаси кода – кода, написанного десятки лет назад, его часто боятся менять и переносят в новую версию.
* Разработка ОС – большой проект, с большой командой разработчиков, с длительной историей разработки – с ростом команды, времени существования проекта копятся и ошибки, которые тяжело найти в большом коде.

Если раньше крупные приложения, были, как и ОС, монолитными, то теперь всё чаще их делают микросервисными, и в этом есть практический смысл. Разработка микросервисного приложения требует поддержания строгости проектирования.

На этом моменте возникает вопрос, почему операционные системы до сих пор монолитные?

Оказывается уже были попытки сделать микросервисные ОС.  
  
Например, операционная система Minix.

[Эндрю Таненбаум](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E) создал первую версию Minix в [1987](https://ru.wikipedia.org/wiki/1987) в качестве «иллюстрации» учебника «Операционные системы: Разработка и реализация» для обучения своих студентов.

Линус Торвальдс использовал и оценил MINIX, а также был знаком с книгой Таненбаума, однако при реализации архитектуры ядра Linux решил не использовать архитектуру MINIX, применив архитектуру монолитного ядра вместо архитектуры микроядра. Данное решение Торвальдса было не одобрено Таненбаумом и вылилось в спор Таненбаума — Торвальдса.

Я считаю, что этот момент определил развитие ОС на многие десятилетия, поэтому ОС преимущественно монолитные – этот подход проще, он появился раньше, но его проблемы начинают проявляться со временем, при росте проекта, появлении новых требований.

Более того, в последние десятилетия были уже попытки написать микросервисную ОС на базе MINIX. Самый известный проект MINIX 3, почему мы о нём ничего не знаем?

Цели моей работы:

1. Изучить микросервисный подход к разработке архитектуры
2. Провести сравнение монолитной и микросервисной архитектуры
3. Рассмотреть MINIX 3 как микросервисную операционную систему.
4. Разработать теоретическую модель современной операционной системы на базе микросервисной архитектуры.

Микросервисная архитектура (MSA).

Микросервис (MS)

MS (Micro Service) *Микросервис* – отдельная, независимая, выполняющая свою задачу, составная единица приложения.

Рассмотри основные особенности микросервисов:

* Небольшой
* Узкоспециализированный
* Независимый
* Закрытый

*Небольшой* – не даёт точной границы, какой сервис должен быть, каждый решит сам. Но советуют придерживаться такого подхода:

На полное переписывание микросервиса уйдёт не более 1 итерации Agile(1-7дней).

«На поддержку одного микросервиса не должно уходить больше двенадцати разработчиков, при этому каждый разработчик может участвовать в разработке не более двенадцати микросервисов».

*Узкоспециализированный* – сервис, в идеальном варианте, решает одну задачу, находится в таком бизнес-контексте, что целиком помещается в голове одного человека.

*Независимый* – взаимодействует с другими сервисами через заданные интерфейсы, и никак иначе. Это позволяет вести разработку независимо от разработчиков других сервисов. Более того, это избавляется от неявных зависимостей с другими модулями.

*Закрытый* – недоступна из вне. Бизнес контекст и служебная информация остаются на уровне сервиса.

Скрытая реализация сервиса и открытая часть (интерфейс), позволяют обновлять код, исправлять ошибки без влияния на другие сервисы.

Микросервисная архитектура (MSA)

Введём понятие: *Распределённой системой* называется система, использующая несколько отдельных вычислительных узлов для реализации общих задач – задач системы.

MSA (Micro Service Architecture) – реализация распределённой системы, основанная на использовании микросервисов, как отдельных вычислительных/функциональных узлов.

Считаю необходимым провести сравнение микросервисной и монолитной архитектуры с целью выявить достоинства и недостатки, оценить пригодность к задачам разработки современных ОС.

Сравнение монолитной и микросервисной архитектуры.

|  |  |  |
| --- | --- | --- |
| Параметр | Монолит | Микросервисы |
| Независимость | При малейшем изменении требуется пересборка и публикация всего приложения | Каждый сервис разрабатывается и публикуется независимо от других |
| Гибкость | Всё приложение использует единые технологии, подменить их крайне тяжело в больших проектах | За счёт скрытой реализации и подчинения внешнему интерфейсу технологии микросервиса могут быть любыми, их изменение часто не имеет больших трудностей. Однако требуется зафиксировать протокол общения модулей. От протокола будут зависеть характеристики полученной системы. |
| Масштабируемость | Необходимо масштабировать приложение целиком | Задача масштабирование решается на уровне конкретного сервиса |
| Надёжность | В случае проблем с каким-то драйвером системы, который работает в режиме ядра, может быть нарушена работа всей системы | Ошибка в работе сервиса потребует его перезапуска, но это не затронет работу других сервисов |
| Тестирование | Наблюдается экспоненциальный рост числа тестов с ростом приложения.  Выполнение всех тестов занимает значительное время.  Чем больше приложение, тем больше неявных зависимостей, которые могут не покрываться тестами – это приводит к серьёзным проблемам | Модульное тестирование в рамках конкретного сервиса не зависит от работы других сервисов – тестов нужно меньше, они более конкретны.  Необходимо интеграционное тестирование для проверки взаимодействия с другими модулями. |
| Команда | Большая команда со схожими техническими навыками | Множество маленьких команд с различными навыками |
| Управление | Команде требуется чёткая иерархия, порой трудно разобраться, кто за что отвечает | Каждая команда покрывает свой сегмент, что позволяет легко распределить ответственность |
| Коммуникация | Классическая, замедляется с ростом команды | Внутри команды - очень быстрая за счёт малой численности. Общение между командами затруднено – нужно придерживаться изолированности, но при этом иметь возможность коммуникации с другими командами – существует тонкая граница от которой напрямую зависит эффективность разработки. |
| Инфраструктура | Экономичная за счёт использования единых ресурсов и технологий | Экспоненциальная, тк каждая команда использует свои ресурсы |
| Безопасность | Сложно реализуема, потенциально невозможна из-за появления неявных зависимостей и ростом сложности анализа безопасности с ростом системы | Команды могут не иметь служебной информации о других командах, таким образом утечка в одном команде не повлияет на другую. Безопасность реализуема при грамотном построении интерфейсов |

Монолитная архитектура проста, отлично подходит для небольших систем, все её минусы раскрываются с ростом проекта.

Микросервисный подход к разработке в небольших проектах уступает монолитному почти во всём. Он, наоборот, раскрывает свои плюсы с ростом проекта, поэтому часто проект начинается как монолитный, но потом переводится на микросервисные рельсы.

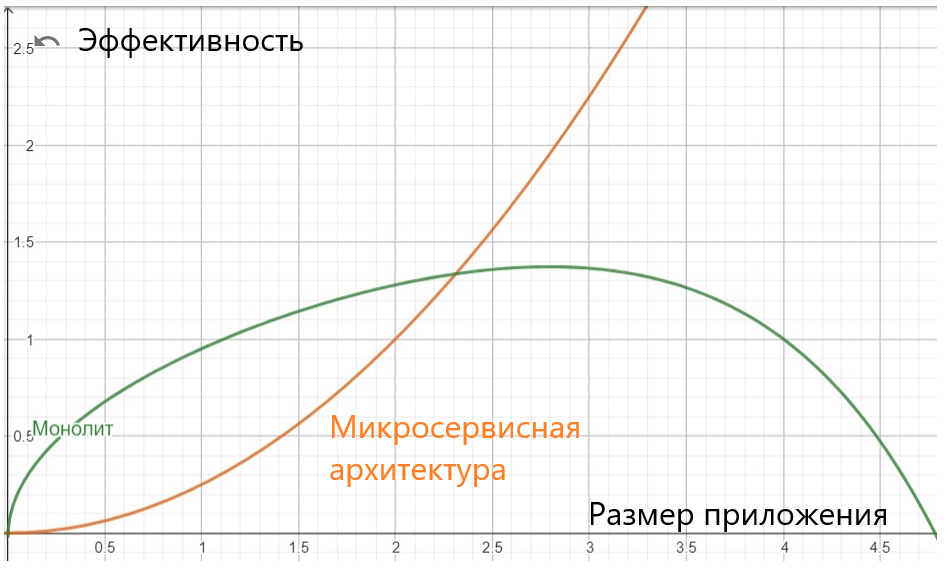
Однако, для такого перехода лучше заранее понимать, что проект будет большой и выделять команды, подразделения, писать независимые классы внутри монолитного проекта – это требует высокой профессиональной дисциплины и зачастую требует больше времени.

Хорошим примером является любая задачка программиста:

Получив маленькую задачу, программист сразу пишет весь код в одном файле Program – это быстро решает задачу, легко тестируется без применения модульных тестов. Но с ростом задачи, размер файла Program растёт, его тяжело поддерживать, тестировать, добавлять новый функционал, доходит до того, что человек не может полностью осмыслить, как это работает – появляются неявные зависимости и баги, на отлов которых уходит значительное время, тестирование затруднено.

Примерно, по такому сценарию развивается монолит.

Микросервис же, напротив, изначально требует выделения сущностей, определения зависимостей внутри приложения, но это требует на первых этапах значительных ресурсов, времени на аналитику, поскольку требуется заранее предусмотреть, что же будет дальше – при старте проекта крайне важно сформировать хороший фундамент проекта. Дальнейшие изменения идут легче, быстрее внедряются, лучше анализируются и тестируются.



Как мы видим, при разработке монолитный подход легко стартует, но с ростом его эффективность падает, а микросервисных подход требует больших затрат на старте, зато потом позволяет достигать больших результатов.

Поэтому часто используют гибридный подход – начинают с монолитной архитектуры, выделяя очевидные сущности и определяя их ответственность, но по достижении представления о приложении, проект полностью адаптируется под микросервисную архитектуру – пока приложение небольшое, это не несёт сложностей. Таким образом, на базе монолита с его плюсами в начале формируется фундамент под микросервисную архитектуру с его преимуществами в дальнейшем.

Перед разработкой собственного решения рассмотрим, на что обращал внимание Тоненбаум при разработке своей микросервисной ОС MINIX.

Операционная система MINIX

MINIX, написанная в 1987 году [Эндрю Таненбаум](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E)ом, является эталоном микросервисных ОС, поэтому рассмотрим её в качестве первоочерёдно.

[Эндрю Таненбаум](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E) рекомендует начинать разработку с формирования интерфейсов.

Операционная система состоит из наборов служб, типов сущностей, множеств операций над ними – всё это формирует пользовательский интерфейс. Пользователями выступают программисты, которые используют возможности системы для написания приложений, которые будут вызываться более многочисленными пользователями.

Помимо основного интерфейса существуют дополнительные, например, драйвера устройств, каждый из которых реализует определённую функцию, которая вызывается с помощью системного вызова.

Функции и вызовы определяют интерфейс, которые отличается от базового.

По мнению Таненбаума следует выделять особое внимание проектированию интерфейсов, если разработчики ОС рассчитывают на успех.

Рассмотрим руководящие принципы в построении интерфейсов.

Простота

Отражается цитатой известного писателя Антуана де Сента Экзюпери:

«Совершенство достигается не тогда, когда уже нечего добавить, а когда нечего убавить».

Лучше меньше, чем больше – не следует усложнять систему там, где это не нужно.

Полнота

«Всё должно быть простым, насколько это возможно, но не проще» - Альберт Эйнштейн.

Интерфейс должен предоставлять возможность выполнить требуемые задачи, при помощи минимума, который необходим, остальное лишь усложняет систему, ведёт к ошибкам.

Рассмотрим на примере MINIX:

Существует всего 3 системных вызова: send, receive и sendrec. Система состоит из набора процессов, менеджера памяти, драйвером, каждый из которых поднят в отдельном процессе.

Метод send отправляет сообщение, метод receive получает сообщение, sendrec отправляет и получает за одно системное прерывание – оптимизация. Информацию можно получать обратившись напрямую к требуемому процессу.

Эффективность

Если какая либо функция не может быть реализована эффективно, то возможно, её не стоит реализовывать. Представим, что у нас есть несколько функций, которые выполняют похожие действия – это нарушение полноты, но это приводит и к путанице: какую функцию выгоднее использовать программисту? Отвечая на этот вопрос неправильно, программист, понижает эффективность своего приложения, а с ростом ОС таких ошибок появляется слишком много и даже сам разработчик будет затруднён, отвечая на вопрос о том, что лучше использовать – не стоит вообще давать возможность выбора, всё должно быть однозначно, просто, понятно.

На этом этапе мы замечаем, что принципы проектирования интерфейсов соответствуют принципамипостроения микросервисов. За каждым интерфейсом может находиться микросервис, который выполняет свою задачу.

На этом этапе становиться интересно рассмотреть ОС, в которой «Всё есть микросервис».

Безопасность

Обычные пользователи редко обращают внимание на проблему безопасности, предпочитая говорить про скорость – так же рассуждали разработчики первых ОС. Помним, что первые ОС были узкоспециализованными машинами, основной задачей которых была быстрое решение задачи, потом безопасность добавляли уже на рабочие рельсы – большинство проблем, связанных с безопасностью кроятся в легаси коде, который был написан около 40-50 лет назад с целью сделать всё быстро. Трудно обеспечить безопасность, когда она изначально не закладывалась.  
Поэтому при разработке системы нужно уделить особое внимание проблемам безопасности.

Jorrit N. Herder, Herbert Bos, Andrew S. Tanenbaum считают, что безопасность системы достигается за счёт вынесения всех прикладных программы и драйверов в разряд процессов пользователя, оставляя лишь крошечное микроядра работать в режиме ядра.

Таким образом, достигается устойчивость системы, которая напрямую связана с её безопасностью.

Рассмотрим архитектуру ОС MINIX 3, которая соответствует описанному подходу.

Архитектура MINIX 3

MINIX 3 основа на микроядре, которое содержит не более 4 тыс. строк кода (в сравнении с 5млн в Windows Vista). Все приложения и драйверы запускаются как изолированные процессы в пространстве пользователя.

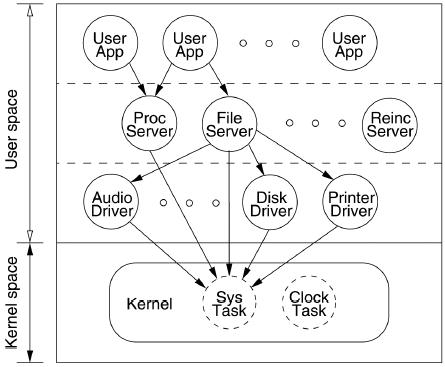


Рисунок 1 Изображение многоуровневой архитектуры MINIX 3

1. Микроядро отвечает за низкоуровневое управление памятью, аппаратные прерывания и планирование процессов. Небольшой размер ядра оперативно отслеживать ошибки.
2. Пользовательский интерфейс ОС формируется набором серверов (сервисов). Выделим основные: Файловый сервер, диспетчер процессов, сервер реинкарнации, подсистема логирования, восстановления.
3. Процессы могут общаться между собой в синхронном режиме, отправляя сообщения.

Зачастую системный вызов состоит из отправки и получения сообщения. Кажется, что это негативно сказывается на эффективности системы, но нет, на поддержания подобного взаимодействия уходит около 1% CPU, что несущественно. Более того, подобная структура (когда ядро не участвует в взаимодействии процессов) позволяет исключить проблемы переполнения буффера ядра и частично решает проблему взаимоблокировок процессов.

1. Драйверы тоже представляются изолированными процессами, за тем исключением, что им разрешено обращаться к функциям ядра, например, чтобы записать значения в аппаратные порты ввода/вывода или осуществить копирование информации из одного процесса в другой. При этом, ядро точно знает, какие порты может использовать драйвер и какие процессы ему доступны. В монолитной архитектуре драйвера имеют доступ ко всем его методам(небезопасно), а ошибка в драйверах может оказать фатальное влияние на систему. В микросервисной архитектуре драйвера используют ограниченный набор функций ядра, что повышает его безопасность, при этом драйвера запускаются, как изолированные процессы – обеспечивается стойкость системы.
2. Драйвера и микросервисы(серверы) могут общаться не со всеми серверами/драйверами, а лишь с разрешёнными. Разрешения записываются в битовую карту процесса, которую просматривает сервер реинкарнации. Вопрос реализации битовой карты важен, поскольку менеджер памяти (как и любой другой процесс) может меняться, потенциально может возникнуть ситуация нарушения целостности битовой карты, что приведёт к трудно прогнозируемым последствиям. Поэтому особое внимание уделяется реализации (списочная реализация) и работе с битовой картой.
3. Сервер реинкарнации периодически опрашивает драйвера и другие процессы в случае обнаружения ошибок перезапускает их без ущерба системе, поскольку каждый процесс запускается независимо.
4. Встроенные прерывания и сообщения – прерывания преобразуются в сообщения соответствующему драйверу. Если драйвер ожидает прерывания, то он отреагирует на сообщение, иначе получит сообщение в следующий раз. Таким образом, решается проблема вложенных прерываний, что облегчает написание драйверов.
5. Подсистема логирования – каждый сервис может быть написан таким образом, что сам логирует свои действия. Логи можно анализировать разными механизмами – детерминированным алгоритмом или машинным обучением для выявления аномального поведения в работе сервиса – подобный подход активно используется в современной разработке.
6. Пользовательские процессы(серверы) взаимодействуют c основными серверами как с WEB API, которое ориентировано на работу в условиях наличия враждебных пользователей, высокой пиковой нагрузки.  
   Таким образом, обеспечивается стабильная работа ядра, отдельных сервисов независимо от действий других элементов системы.

Программные ограничения – это правильно

По ходу развития программирования постепенно накапливались программные ограничения, которые ограничивали деятельность пользователя/программиста для увеличения устойчивости/безопасности системы.

Пример:

С появлением диспетчеров памяти (MMU) пользовательские приложения потеряли возможность напрямую обращаться к устройствам ввода-вывода, теперь это нужно делать только системные вызовы.

Многие были против подобных ограничений, ссылаясь на понижение эффективности, но в последствии пришли к соглашению, что подобные меры решают проблему аварийной остановки системы в случае ошибки в пользовательских программах.

Резюмируем ограничения, которые накладывает MINIX 3 на разработчиков.

Ограничения MINIX 3

Аналогично MINIX 3 ограничивает разработчиков больше, чем другие ОС.

1. Изоляция ядра – каждый сервис имеет свою битовую карту того, что какие системные вызовы он может использовать.
2. Изоляция памяти – каждый сервис работает в изолированном от других пространстве. В случае обнаружения попыток незаконного обращения к области памяти другого сервиса сервер реинкарнации уничтожает и переподнимает нарушителя. Законно обратиться к области памяти другого процесса можно, если он дал явное разрешение или этот процесс обладает таким правом, выданным системой (является надёжным процессом).
3. Ограниченное межпроцессорное взаимодействие – каждый процесс имеет список разрешённых сервисов, с которыми он может общаться, за контроль отвечает сервис реинкарнации.

Модель программирования в MINIX

От удобства модели программирования напрямую зависит развитие ОС.

Должна быть возможность расширять функционал ОС за счёт написание пользовательских программ внешними разработчиками, таким образом, разработчики ОС предоставляют инструменты разработки, но большую часть пользовательского ПО пишут пользователи – без этого ОС будет обладать узким функционалом, малой аудиторией, которая будет постепенно уходить в более популярные системы.

Во многом преимущества модели программирования в MINIX обусловлены преимуществами микросервисной архитектуры.

Перечислим основные преимущества:

1. Короткий цикл разработки – за счёт небольшой команды и небольшого, в сравнении с кодом ОС, кода сервиса.
2. Высокая производительность – ошибка в драйвере (или обновление) не вынуждает выполнять перезагрузку системы, логи позволяют понять, что произошло, а сервер реинкарнации перезапускает процесс.
3. Разработка упрощена в сравнении с монолитными системами за счёт того, что разработчик ограничен методами интерфейсов, он не задумывается над внутренней реализацией.
4. Высокая отслеживаемость – за счёт того, что каждый сервис изолирован и работает в среде пользователя, ошибка возникает не ядре, а в среде пользователя – её можно проанализировать средствами отладки, более того, сервисы логируют свои действия дополнительно на лог сервер. За все ошибки работы сервиса отвечает его разработчик, производитель, а не создатель ОС.
5. Гибкость конфигурации – каждый элемент системы мал и изолирован, можно сконфигурировать систему под конкретные задачи. Например: можно подменить сервис, отвечающий за файловую систему или за подсистему логирования, при этом работа остальных сервисов не нарушится.
6. Облегчение поддержки – мечта системного администратора, все модули малы, независимы и понятны.

Выводы по MINIX 3

MINIX 3 – модульная операционная система, построенная быть сверхнадёжной, это свойство обеспечивается более жёсткими ограничениями, чем у других ОС. Система гибкая, что позволяет сконфигурировать её под различные задачи, при этом, обладает широкими возможностями администрирования, что делает её обслуживание удобным. Модель программирование мало отличается от привычной модели программирования, но имеет ряд преимуществ, обусловленных особенностями микросервисной архитектуры – простотой, изолированность, эффективностью, расширяемостью. Программист для себя найдёт удобную почву для разработки, которая будет подталкивать его к написанию более строго, соответственно, более эффективного кода. Пользователь получит безопасную, конфигурируемую, высокоэффективную систему.

Я считаю, что микросервисные ОС, такие, как MINIX, должны были вытеснить монолитные ОС с рынка, но этого не произошло. Попробуем разобраться в причинах.

Сравнение путей развития MINIX, LINUX и MS-DOS

MINIX разработана профессором Эндрю [Таненбаум](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E)ом в 1987 году, как учебная UNIX подобная ОС, как альтернатива тяжёлой по тем временам системе UNIX, для работы с которой требовалось «железо» стоимостью несколько тыс долларов, что могли себе позволить не все компании, не говоря уже об обычных пользователях.

Популярный в то время Unix 7 не был предназначен для обучения, так что студентам приходилось искать альтернативы и разработка профессора была для них настоящим подарком.

В начале 90х появляется Linux – тоже UNIX подобная ОС. Linux был разработан учеником [Таненбаум](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E)а - Линусом Торвальдсом, который не разделял мнение учителя о преимуществе микросервисной архитектуры над монолитной и сделал свою ОС монолитной.

Это вылилось в спор [Таненбаум](https://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BD%D0%B5%D0%BD%D0%B1%D0%B0%D1%83%D0%BC,_%D0%AD%D0%BD%D0%B4%D1%80%D1%8E)а – Торвальдса. Профессор уже тогда (в 1992 году) считал монолитное ядро Linux устаревшим:

"Как и большинство старых операционных систем, Linux использует монолитное ядро с очень низкой переносимостью (под этим термином Таненбаум подразумевает способность адаптироваться к различным архитектурам процессора). Вся система - это один файл, работающий в режиме супервизора." - заявил Таненбаум, Линусу Торвальдсу.

В то время более доступной и быстро развивающейся альтернативой UNIX-7 был MS-DOS, который можно было запускать, имея более слабый компьютер, что делало MS-DOS доступной для широкого круга лиц.

Linux и MS-DOS - монолитноядерные ОС, как и их современные реализации в виде Windows и различных дистрибутивов на базе Linux, но почему? Мы ведь уже разобрались, что микросервисный подход потенциально лучше монолитного, особенного в таких крупных проектах, как операционная система, но про MINIX большинство пользователей даже не слышало.

Это во много обусловлено подходом к развитию и целями заинтересованных сторон:

Сравнение путей развития MS-DOS, Linux и MINIX

1. MS-DOS (1981г) была разработана для компьютера IBM PC компании IBM, которая в 1975 году выпустила первый массовый персональный компьютер (ПК) – IBM 5100, но он оказался слишком дорог для обычных пользователей. Целью компании в том время было занять часть перспективного развивающегося рынка – поэтому компания выпустила компьютер IBM PC, с операционной системой MS-DOS, которая была доработана компанией Microsoft специально под требование IBM из ОС 86-DOS, права на которую были выкуплены.

Как мы видим, MS-DOS изначально ориентирована под конкретный компьютер с конкретным процессором (Intel 8088) – это и есть «низкая переносимость».

1. Linux начал разрабатывать Линус Торвальдс для своего [персонального компьютера](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D1%81%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) с [процессором](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) [Intel 80386](https://ru.wikipedia.org/wiki/Intel_80386), который на тот момент имел очень популярную архитектуру. В последствии Linux развивался как продукт с открытым исходным кодом, открытым сообществом разработчиков. Разные мнения в последствии привели к появлению разных дистрибутивов на базе Linux.
2. Minix же, напротив, развивался не как рыночный продукт и не как открытый продукт, развиваемый сообществом – это была, в первую очередь, учебная операционная система.

Принципиально то, что MS-DOS, развиваемая большой компанией как закрытый продукт под конкретную архитектуру с целью выхода на новый широкий рынок персональных компьютеров обрела высокую популярность среди пользователей из-за своей доступности и функционала, а Linux пошёл по другому пути – по пути развития открытого продукта широким сообществом и тоже обрёл популярность, но Minix, скорее всего, начал борьбу за рынок слишком поздно.

Скорее всего, даже, если разработчики из IBM и Microsoft понимали, что микросервисный подход лучше монолитного, они бы ничего не стали менять – фактически адаптация потребовала бы

* больших вложений на переработку,
* задержку развития на рынке (не стоит забывать, что были серьёзные конкуренты, имена которых мы часто не знаем)
* отказ от поддержки уже проданных ПК повлек бы за собой потерю репутации.

На подобные риски бы не пошли главы компаний.

Последние десятилетия доля рынка постепенно переходит к Linux, во многом это обусловлено тем, что открытый подход к разработке прогрессивнее, чем закрытая политика, используемая Microsoft.

Возрождение MINIX

MINIX была (и остаётся по-прежнему) широко используемым инструментом для различный курсов по операционным системам в университетах всего мира, но она получила и дальнейшее развитие.

Современная версия MINIX - MINIX 3 разрабатывалась с 2005 по 2014 год Эндрю Таненбаум и его командой на средства, полученные в качестве гранта, который позволил развивать систему до 2014 года.

MINIX 3 реализовывает все необходимые средства для работы и действительно может считаться сверхнадёжной операционной системой, но современного графического интерфейса она не получила, имеет порядка 400 работающих приложений.

Последнее обновление вышло в 2014 году, но до сих пор поддерживается официальный сайт <http://www.minix3.org>, который ежедневно посещает около 1000 пользователей. За всё время с официального сайта скачена более 75 тысяч раз.

Существует большое сообщество (как англоязычное, так и русскоязычное), которое занимается дальнейшим развитием операционной системы.

За операционными системами на базе MINIX стоит будущее, они будут развиваться и в последствии вытеснят монолитные системы, которые подобно снежному кому скатываются с горы, набирают массу, но эта же масса им мешает маневрировать, в частности остановиться и пересмотреть свой дальнейший путь.

Intel ME

Операционная система MINIX 3 с 2017 года используется в процессорах Intel в подсистеме Intel Management Engine. MINIX 3 была выбрана из-за его устойчивости, эффективности работы с памятью и низких требований.

Подобный факт свидетельствует о том, что операционная система MINIX 3, в частности идеи, заложенные в её основу, по сей день актуальны.

Литература

1. <https://habr.com/ru/companies/raiffeisenbank/articles/346380/>
2. <https://habr.com/ru/companies/serverspace/articles/692916/>
3. Эндрю Таненбаум «Современные компьютерные системы» 1992
4. Йоррит Хердер (Jorrit N. Herder), Херберт Бос (Herbert Bos), Эндрью Таненбаум (Andrew S. Tanenbaum) «Построение надёжных операционных систем, допускающих наличие ненадёжных драйверов устройств». Перевод: Сергей Кузнецов. <http://minix3.ru/articles/reliable-os.html>
5. Йоррит Хердер (Jorrit N. Herder), Херберт Бос (Herbert Bos), Эндрью Таненбаум (Andrew S. Tanenbaum) «Модульное программирование в MINIX 3». Перевод: Сергей Кузнецов.   
   <http://minix3.ru/articles/mspinm3.html>
6. <https://ru.wikipedia.org/wiki/MS-DOS>
7. <https://dzen.ru/a/ZEuqclvPyAKVx-nV>
8. <https://ru.wikipedia.org/wiki/История_Linux>
9. <http://minix3.ru/articles/introduction_minix3.html>
10. <https://ru.wikipedia.org/wiki/MINIX_3>

Идеи для реализации

1.Обязательно учесть систему доверенного третьего лица, которое будет реализовывать соответствующий интерфейс.

2.Файловая система – это модуль системы, которых может быть много, это обеспечить высокую пропускную способность, увеличивая нагрузку на сеть

Например: в Mongo DB существуют коллекции, было бы круто равномерно распределить все файлы на несколько «узлов файловых систем», каждый из которых отвечает за свою коллекцию, чтобы можно было параллельно обрабатывать большие потоки данных.

3.Существуют различные системы для хранения файлов на серверах – рассмотреть возможность создания прослойки для работы с такими системами как с будто бы это файловая система нашей машины – это позволит использовать уже существующие решения для создания высокопроизводительной, адаптивной под задачи, системы.

4.Определиться, как рассматривать ОС – пока я представляю ОС как множество сетевых узлов, на которых может быть любая Физическая ОС, подобранная для конкретной функции – т.е. рассматривать ОС как сборку уже готовых решения, на данный момент трудно будет заставить всех перейти на новую структуру

5.В последствии рассмотреть вариант создания своего микроядра, совместимого со всеми аппаратными решениями – это очень сложно(

6.Что делать с графикой? Видеокарта – тоже модуль системы. Полагаю, нужно посылать вычислительные задачи на разные узлы, чтобы успевать обрабатывать запросы графики.

7.Пользовательский компьютер – терминал или очень слабое, в сравнении с системой, устройство для примитивных задач.

8.В данной топологии ОС, которую хочу строить будет мало уязвимостей современных ПК, но не забывать про сетевые атаки – всё взаимодействие идёт по сети, не факт, что локальной.

9.Обязательно систему хранения логов, например, ElasticSearch и системы выявления аномалий(Алгоритмическую и AI)

10.В системе скорее всего будут важные распределительные пункты, они должны быть просты, их цель распределить задачи по распределённым узлам.

11.Поговорить о безопасности(Литература 4 – интересная статья)

12.Сервер реинкарнации – от него порождаются все процессы, он отлавливает подвисшие-зомби процессы, удаляет их, перезапускает и сообщает об этом другим серверам.

13.Возможность склонировать и запустить параллельно уже работающий модуль системы для того, чтобы распараллелить нагрузку или чтобы удобно протестировать без вреда для всей системы.