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 9

Литература 10

Введение

«Мир, который мы знали, близок к концу, и, как и прежде, ключ к будущему скрывается в прошлом.» - The Witcher II

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 цельного блока приведёт к уничтожению всей системы. Вирус, попавший в систему по сети, легко получит 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, 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, об этом погорим чуть позже.

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.

Цели моей работы:

1. Изучить микросервисный подход к разработке архитектуры
2. Вернуться к спору спору Таненбаума — Торвальдса
3. Составить требования к ОС нового времени
4. Вспомнить про существующие ОС на базе MINIX(их было немало)
5. Разработать и обосновать теоретическую модель такой ОС на основе 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* – реализация скрыта, недоступна из вне. Не нужно задумываться, как сервис работает внутри, это должны решать разработчики этого 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) – организация 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 |
| Масштабируемость | Необходимо масштабировать приложение целиком | Задача масштабирование решается на уровне конкретного сервиса |
| Надёжность | В случае проблем с каким-то компонентом системы, требуется перепубликовывать всё приложение | Остановка и перепубликация сервиса с ошибкой не влияет на другие сервисы |
| 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 | Всей командой, требуется чёткая иерархия, порой трудно разобраться, кто за что отвечает. | Каждая команда покрывает свой сегмент, что позволяет легко распределить ответственность |
| Коммуникация | Привычная, но медленная. | Внутри команды - очень быстрая за счёт малой численности. Межкомандная требуется не часто – это могут быть совещания руководителей команд. |
| Инфраструктура | Экономичная за счёт использования единых ресурсов, репозиториев и технологий. | Экспоненциальная, тк каждая команда использует свои ресурсы, репозитории, технологии. |
| Безопасность | Сложно реализуема, потенциально невозможна. | Команды могут не иметь служебной информации о других командах, таким образом утечка в одном команде не повлияет на другую. |

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

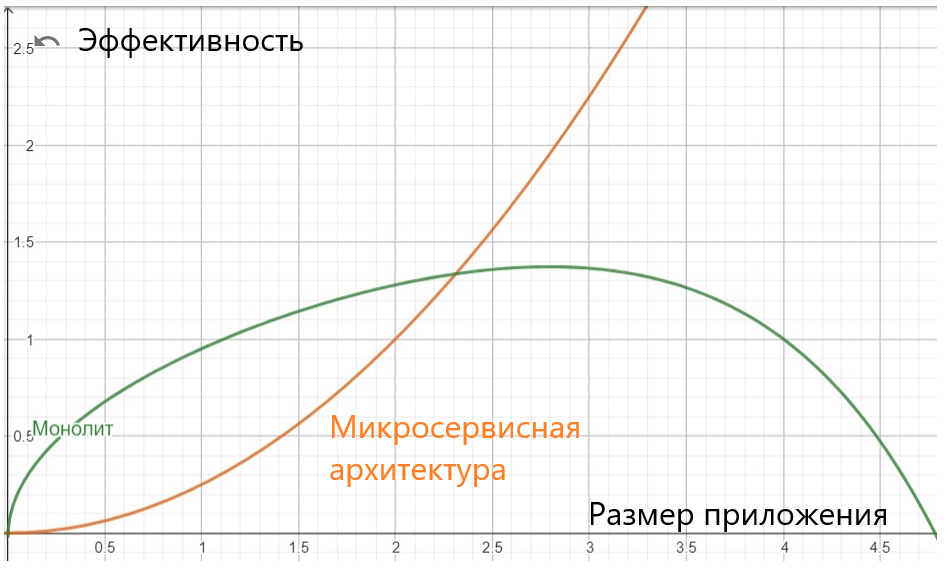
Микросервисный подход к разработке в маленьких проектах уступает монолитному почти во всём, он, наоборот, раскрывает свои плюсы с ростом проекта, поэтому часто проект начинают как монолитный, но потом переводят его на микросервисные рельсы, однако для такого перехода лучше заранее понимать, что проект 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, которые не отлавливаются, тк 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, поскольку требуется заранее предусмотреть, а что же будет дальше – при старте проекта, это сложно, поэтому часто требуется переписывать начало, для формирования хорошего фундамента проекта. Дальнейшие изменения идут легче, быстрее внедряются, лучше анализируются и тестируются.



Как мы видим, даже 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 функций 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 пересекаются с принципами построения микросервисов. За каждым интерфейсом может находиться микросервис, который выполняет свою задачу.

На этом этапе становиться интересно рассмотреть ОС, в которой «Всё есть микросервис», но вернёмся к этому позже.

Познакомимся с основными парадигмами.

Парадигмы

Парадигма – набор идей, шаблонов мышления, теорий, методов, стандартов, в соответствии с которыми осуществляется деятельность. В данном случае деятельность – проектирование ОС.

Парадигмы важно обсудить, поскольку они обеспечивают архитектурную согласованность элементов системы.

Парадигмы пользовательского интерфейса

Определяют способ взаимодействия пользователя с системой. На данный момент самой популярной парадигмой пользовательского интерфейса является WIMP (Окна, иконки, меню, указатели), которая реализована как в Windows, так и во многих Unix системах.

Как бы ни было построено взаимодействие, важно, чтобы оно было единообразно, это облегчает адаптацию пользователя к новым приложениям.

Разработчики ОС должны предоставить программистам инструменты для реализации интерфейса: библиотеки и системные вызовы.

Парадигмы исполнения

Определяют принцип работы программ.

Самые популярные парадигмы исполнения:

-Алгоритмическая: приложение реализует чёткий алгоритм действий, выполняет заранее заложенную функцию, что позволяет обеспечить безопасность пользователя и системы.

-Событийная: ключевым понятием является «событие» - действие, выполняемое при выполнении определённых условий (например, нажатие кнопки, изменение положения элемента на экране и т.д.). При старте приложение ожидает возникновения событий, и только потом на них реагирует. Событийная парадигма ориентирована на взаимодействие с пользователем.

Какие ещё бывают парадигмы исполнения?

Парадигмы данных

Интерфейс системных вызовов

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