СОДЕРЖАНИЕ

[1. Архитектура браузера 3](#_Toc157542647)

[1.1. Конкретная имплементация на примере Google Chrome 4](#_Toc157542648)

[1.2. Конкретная имплементация на примере Firefox 5](#_Toc157542649)

[2. Архитектура движка рендера WebKit 6](#_Toc157542650)

[3. Браузерный Event loop. Введение в концепцию однопоточного и многопоточного программирования. Асинхронная модель 7](#_Toc157542651)

[4. Call stack, стек вызовов. Рекурсия. Переполнение стека. 8](#_Toc157542652)

[4.1. Очередь задач. Асинхронный код 10](#_Toc157542653)

[5. Задачи JS движка (heap, call stack, выделение памяти и сбор мусора) 11](#_Toc157542654)

[6. Web API. Таймауты и слушатели событий 12](#_Toc157542655)

[7. Промисы и очереди микро и макро задач 14](#_Toc157542656)

[7.1. Что порождает микротаски, а что макротаски? 16](#_Toc157542657)

[8. Стадии рендера. DOM, CSSOM, Render tree, style calculation, layoyt, paint, composite 17](#_Toc157542658)

[8.1. Стадии рендера 17](#_Toc157542659)

[9. Что такое Node js. Из чего состоит? Устройство Node js. Движок V8 и Libuv 20](#_Toc157542660)

[10. Блокирующий и неблокирующий I/O (ввод и вывод) 22](#_Toc157542661)

[11. Планировщик потоков 25](#_Toc157542662)

[12. Демультиплексор событий и шаблон Reactor 26](#_Toc157542663)

[13. Event loop в node js. Фазы и очереди 27](#_Toc157542664)

# Архитектура браузера

* User interface - часть браузера, с которой взаимодействует пользователь через ui.
* Browser engine - соединительная часть между пользовательским интерфейсом и механизмом рендеринга.
* Rendering engine - самая важная часть, с помощью него получаем страницы для послед. Взаимодействия. (WebKit или Quantum). Строится DOM дерево, строится объектная модель css, расположение итд (стадии рендера).

------

* Networking - предназначена для работы с сетью, предназначена для истории сайтов посещения, за доменные имена, взаимодействие с DNS серверами, правила обработки, http/s, tcp итд, обмен пакетами итд.
* JS interpreter - та часть, которая обрабатывает js код (V8 например, поверх построен nodejs, chrome итд). Независимая, абстрагированная технология (heep, call stack, далее подробнее).

UI backend - подкапотная логика для интерфейса.

------

Data Persistence - хранилище данных, localStorage, session, idb, web sql. Также обрабатывает cookie, все что локально.

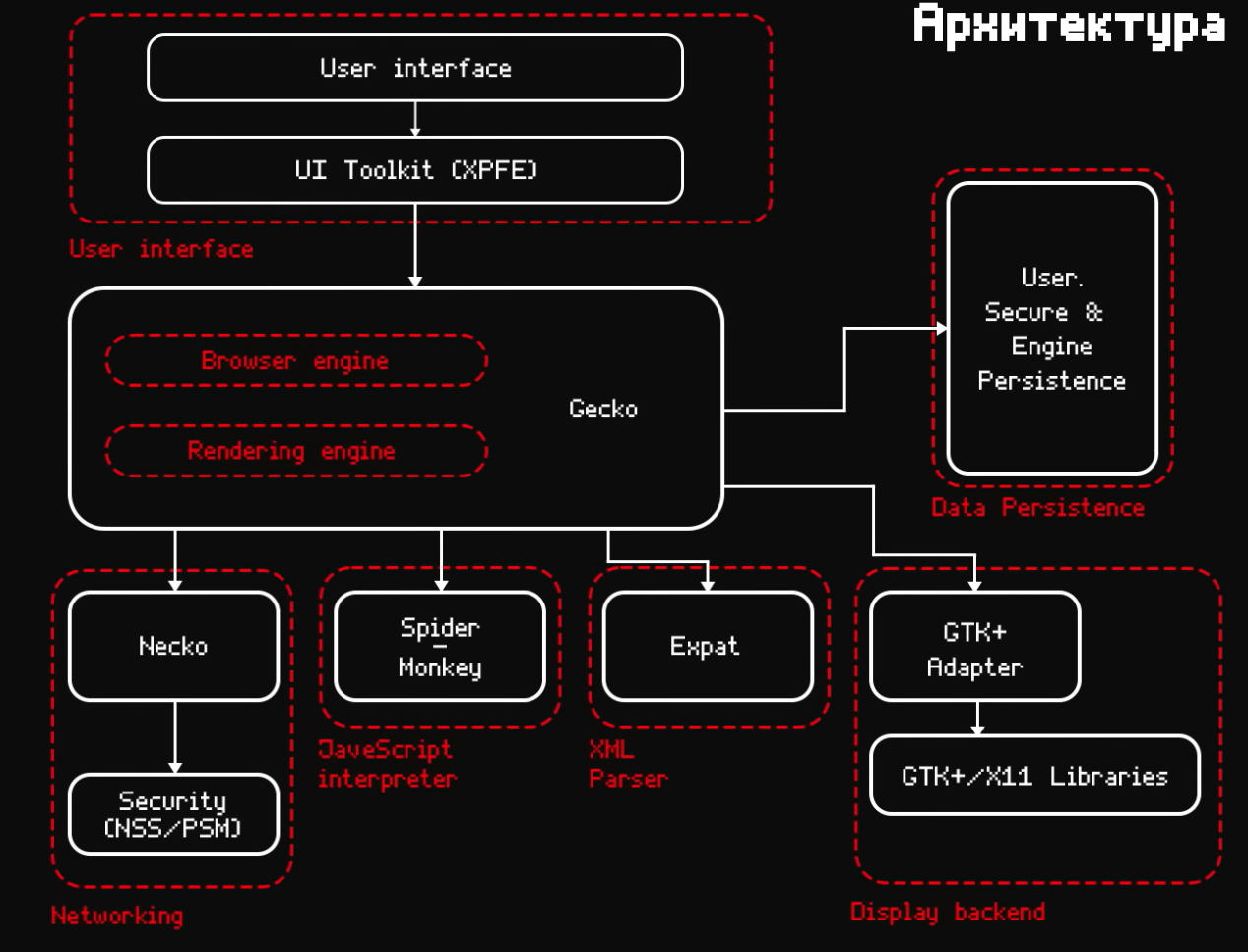
## Конкретная имплементация на примере Google Chrome



1. Архитектура браузера (Chrome)

В качестве движка WebKit, в качестве движка js V8. Также появляется parser xml, всякие плагины итд.

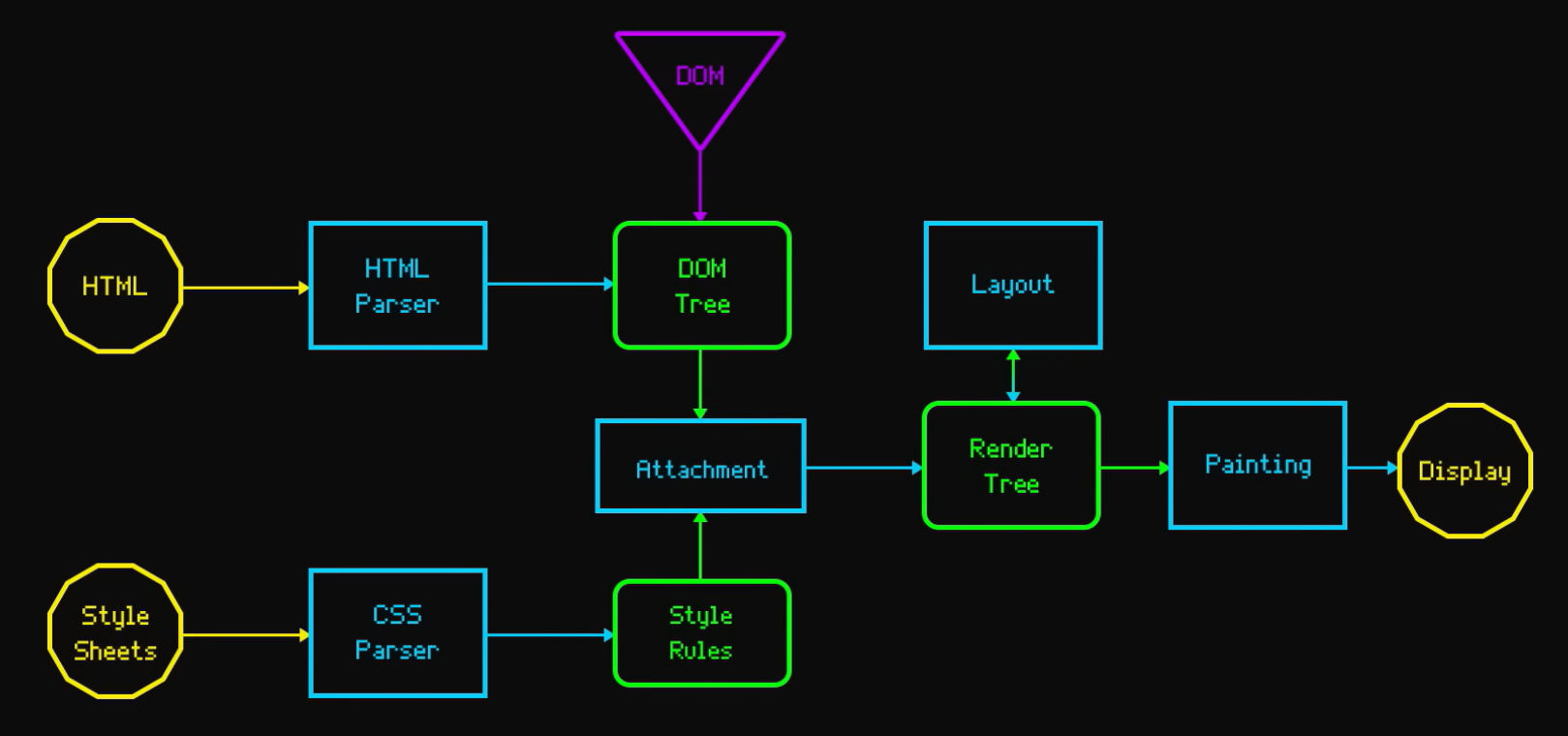
## Конкретная имплементация на примере Firefox



1. Архитектура браузера (firefox)

Здесь есть отличия, например движок js это Spider-Monkey. И также движок Gecko или Quantum.

# Архитектура движка рендера WebKit



1. Движок рендера WebKit

На вход движку приходит html, css, движок парсит их с помощью соответственных парсеров. На выходе html получается DOM Tree, с которым через JS работаем. На выходе css получается Style Rules. По итогу строится дерево рендера (Render Tree). Далее проходит определённые стадии и на выходе страница рендерится, и появляется возможность с ней взаимодействовать (разбор стадий позже).

# Браузерный Event loop. Введение в концепцию однопоточного и многопоточного программирования. Асинхронная модель

Event Loop НЕ является частью JavaScript. Например, Chrome используется V8, в NodeJS также V8, НО при этом в Chrome и NodeJS Event Loop это абсолютно разные концепции. Разные реализации.

Event Loop — это отдельный механизм, который позволяет использовать не блокирующую модель ввода и вывода. Этот механизм обрабатывает события, такие как пользовательские взаимодействия (щелчки мыши, ввод с клавиатуры), сетевые запросы, таймеры и другие асинхронные операции.

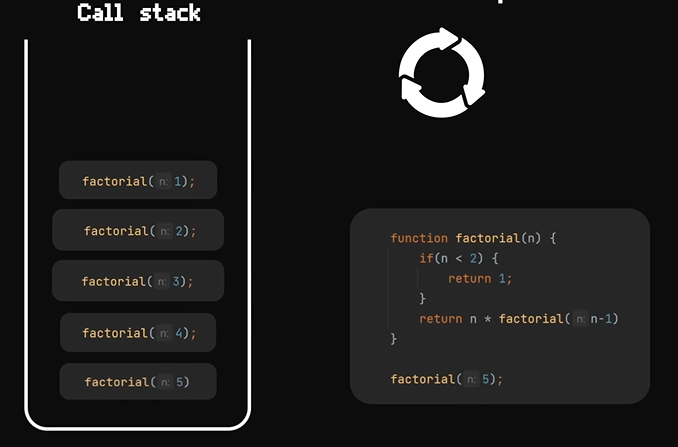
# Call stack, стек вызовов. Рекурсия. Переполнение стека.

Есть Call stack (стек вызовов), за его обработку отвечает движок JS (например V8). Стандартный стек из структур данных (LIFO (Last In, First Out)). Он складывает ф-ции в том порядке, в котором они должны быть вызваны.

* Примеры:



1. Пример 1 (sync)



1. Пример 2 (sync)

Возможно переполнение стека вызовов!!!

## Очередь задач. Асинхронный код

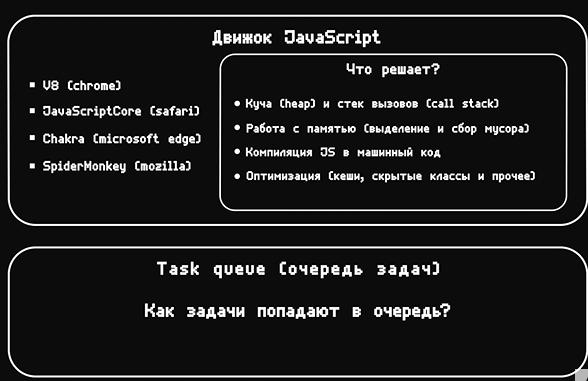


1. Пример 3 (async)

Задачи из очереди выполняются только после вызова всех функций из стека!!! (очередь задач две, разбор далее)

# Задачи JS движка (heap, call stack, выделение памяти и сбор мусора)

Как задачи попадают в очередь? За очередь отвечает Event Loop, а за Call stack движок JS.



1. Что решает движок JS

Event loop - не является частью движков, таких как V8. Цикл событий предоставляется СРЕДОЙ, например браузер или NodeJS.

# Web API. Таймауты и слушатели событий

И так, если Call Stack предоставляет Движок JS (V8), а Task queue (очередь задач) Event loop, то как они общаются между собой? У каждого браузера есть Web Api.

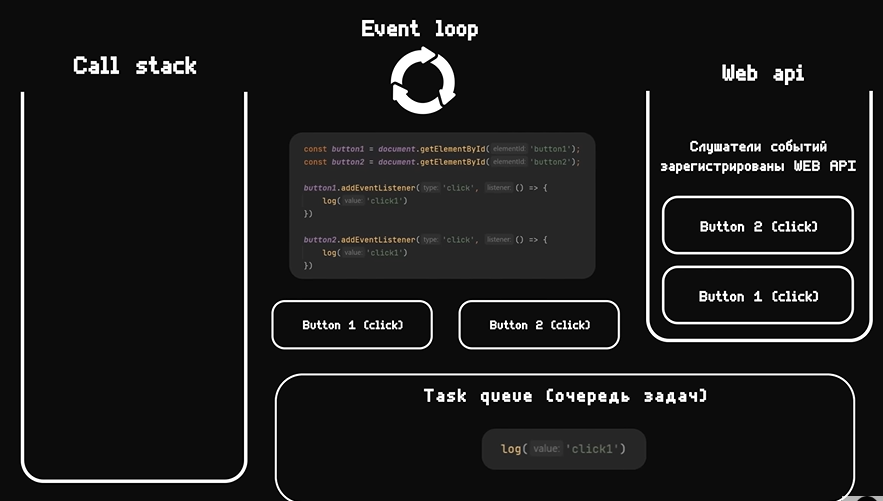
Web Api - предоставляет всякие timeout, обработку слушателей события нажатия на кнопки, различные события загрузки изображений, файлов, отправку fetch запросов. Это все браузерные штуки, не JS спецификация.

Пример



1. Код для примера

Сначала целый setTimeout попадает в Call stack. После этого он регистрируется в Web Api, на этом этапе запускается таймер. После того, как таймер иссяк, сам cb отправляется в Task queue (очередь задач). И после того как Call Stack очистился, берется задача из очереди выполняется.



1. Пример с слушателями

Также их нужно не забывать явно удалять. Все кнопки, события, это не спецификация JS, а именно Web Api, браузерная часть.

# Промисы и очереди микро и макро задач

На самом деле есть две очереди. Macrotask queue (очередь событий) и microtask queue (очередь задач). У этих очередей есть определённый приоритет, и Event loop берет таски в определённом порядке.

Важный момент: Promise всегда попадает в очередь микро тасок (microtask queue).



1. Очереди

Это и есть специфика работы Event loop, в приоритете в очереди всегда выполняются микротаски!!! Сразу все и только потом одна макротаска.

Рассмотрим пример, когда несколько макротасок и несколько микротасок:



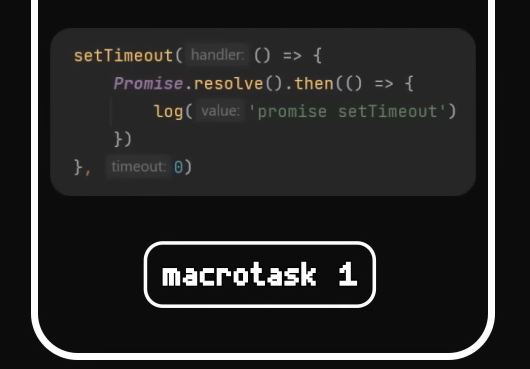
1. Этап 1



1. Этап 2

Берется ОДНА макротаска!!!

А что будет при такой ситуации?



1. Макро задача порождает микро задачу



1. Объяснение

Макротаска под номером 2 НЕ выполняется!!! Сперва выполнится микротаск, который порадился макротаском, и только потом Event loop забирает макротаску из очереди макро задач.

Сначала выполняются все синхронные задачи, потом выполняются все микротаски, потом ОДНА макротаска, если порождает другие микротаски, то выполняются они все сразу и только потом вторая макротаска.

Также важно понимать, что микро такси выполняются до рендера DOM, а макро таски после рендера.

## Что порождает микротаски, а что макротаски?

Микротаски:

* Promise
* queueMicrotask - можно создать явно с помощью данной функции
* mutationObserver - специальный инструмент, который позволяет следить за DOM нодами

Макротаски:

* Таймеры (setTimeout, setInterval)
* События (клик, загрузка Img итд)
* Браузерные нюансы (рендер, input/output, итд)

# Стадии рендера. DOM, CSSOM, Render tree, style calculation, layoyt, paint, composite

Рассмотрим, то, как браузер обрисовывает страницу, с которым пользователь будет взаимодействовать. Стадии, очередность.

## Стадии рендера

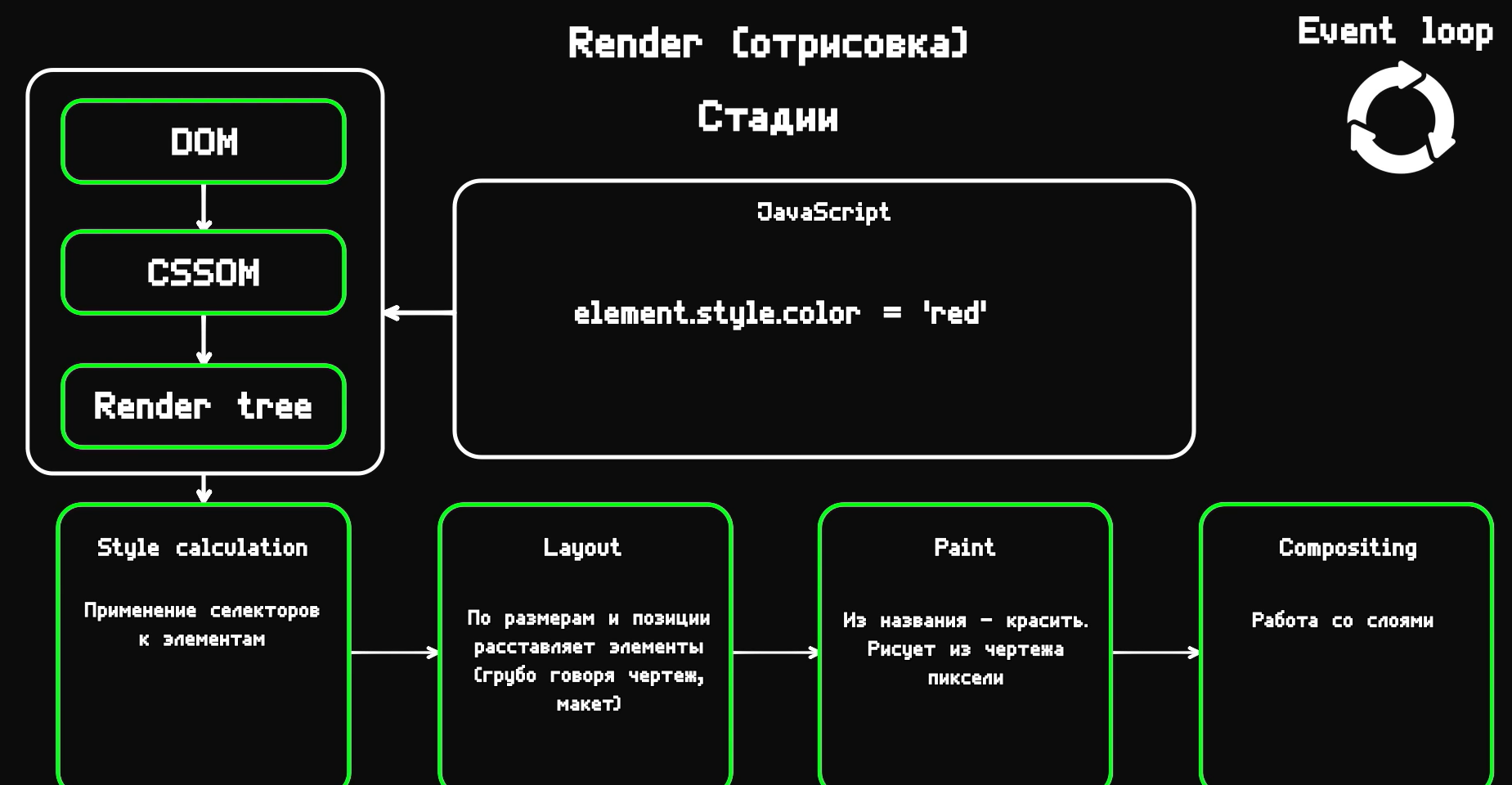
1. Стадия "Style Calculation" (калькуляция стилей) - В этом процессе браузер вычисляет и применяет стили к элементам HTML в соответствии с CSS (каскадные таблицы стилей), определенными для каждого элемента. (чем проще селектор - тем лучше ).
2. Стадия "Layout" (или "Layout Phase") - в рендеринге веб-страницы относится к процессу определения расположения элементов на странице в соответствии с их структурой и стилями. (стили из первой стадии, строит прототип). На выходе получаем Layout Tree.
3. Стадия "Paint" - относится к моменту, когда браузер начинает заполнять пиксели на экране цветами и текстурами в соответствии с рассчитанными на предыдущих этапах параметрами расположения и стилями элементов.
4. Стадия "Compositing" (компоновка) - объединяет отрисованные элементы в окончательное изображение, учитывая прозрачность, наложение и порядок отображения. Это процесс создания окончательного визуального представления страницы перед её отображением на экране. Работа со слоями.



1. Стадии

Но что будет если пользователь нажмет кнопку, и она вызовет анимацию и, например, откроется модальное окно?

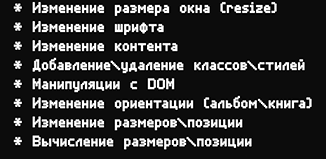
И здесь по сути ничего не меняется, стадии идут по кругу, меняется какая-то часть



1. Рендер при изменении

Рендер - дорогостоящая операция!!!

Что вызывает рендер?

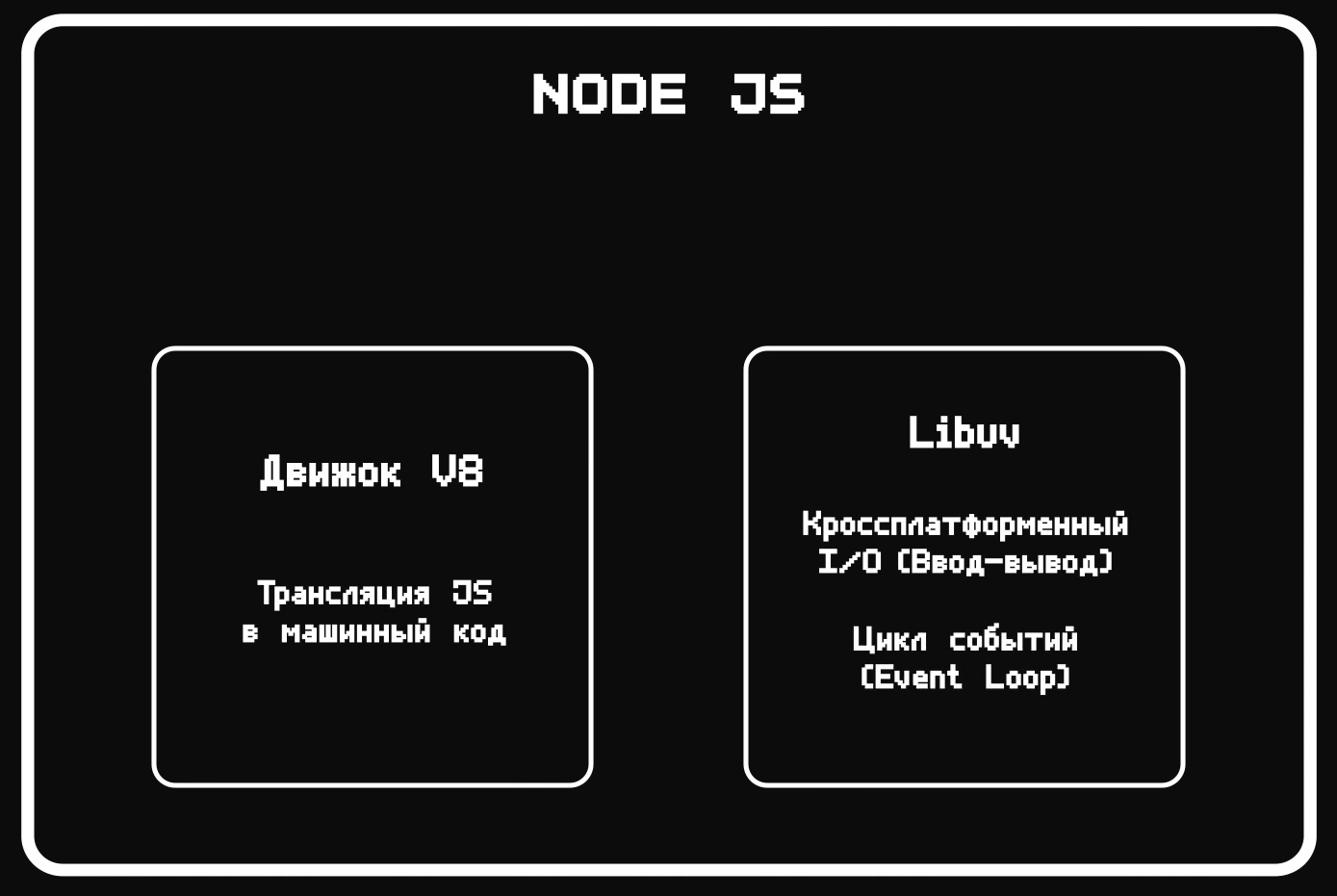


1. Причины ререндера

# Что такое Node js. Из чего состоит? Устройство Node js. Движок V8 и Libuv

NodeJS - это не ЯП, это программная платформа, которая с помощью специальных инструментов позволяет превращать JS в машинный код. Изначально JS это язык разработки браузерных приложений и некоторая функциональность там очень ограничена, например, взаимодействие с файлами, с ОС и все такое. NodeJS убирает эти ограничения.

Преобразование достигается с помощью движка V8, на котором построен NodeJS. Написан на C++



1. NodeJS

Libuv - в связке с V8 они образуют фундамент NodeJS.

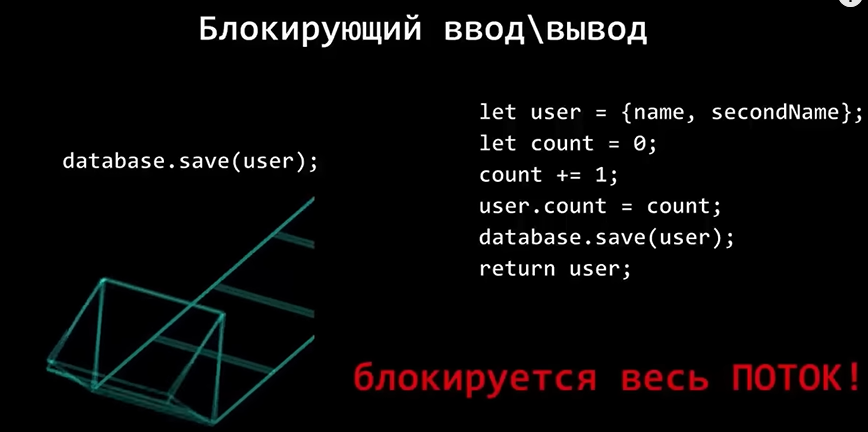
Кроссплатформенный I/O (Ввод-вывод) - относится работа с файловой системой, работа с сетью. То есть мы через JS даем команду NodeJS прочитать файл и NodeJS использует внутри себя Libuv и путем этого достигается кроссплатформенность.

Во многих ЯП (C / JAVA / PYTHON) все инструкции, все команды по умолчанию явл. блокирующими, например мы пытаемся считать файл из файловой системы, так вот пока файл не прочитается, приложение у нас заблокируется. Чтобы это предотвратить, используется многопоточное программирование. NodeJS использует цикл событий.

Преимущества Цикла событий:

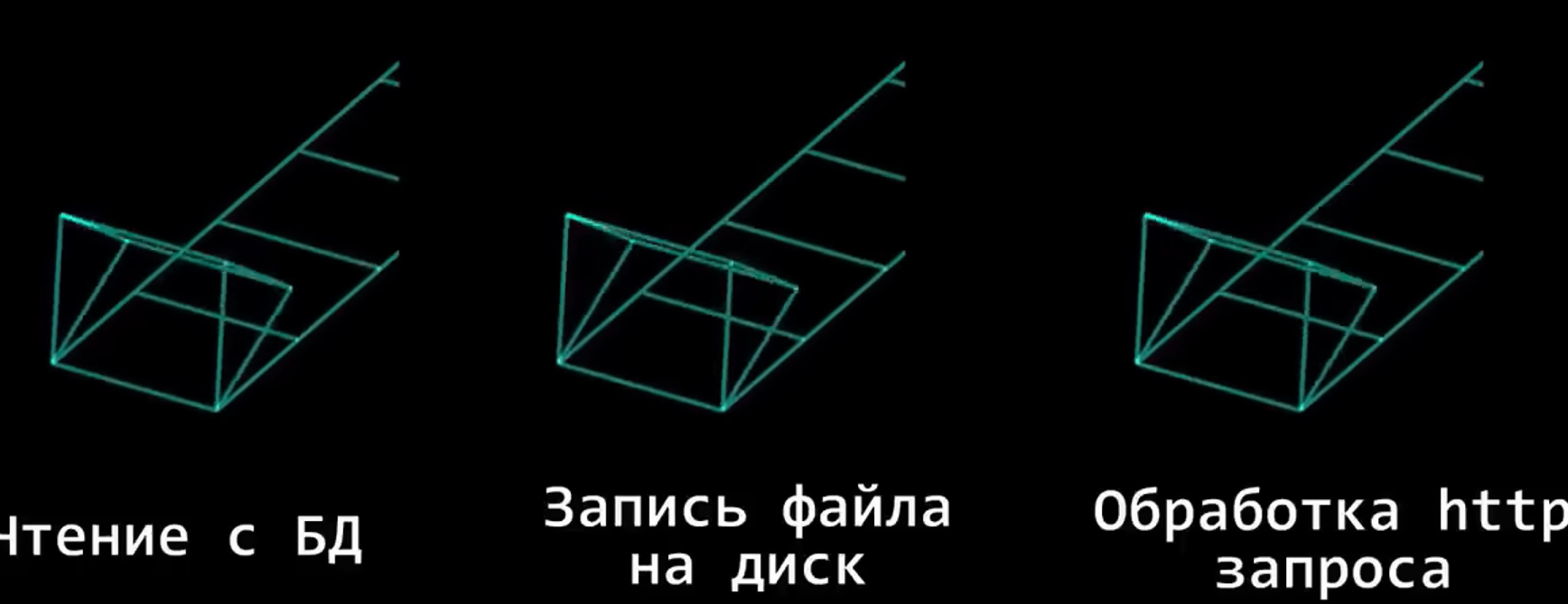
* Возможность обработки большого количества операций ввода вывода, если простые, то скорость максимальная (есть минусы, много асинхронного кода, сложные вычисления создают большую нагрузку, особенно математические).

# Блокирующий и неблокирующий I/O (ввод и вывод)



1. Блокирующий I/O

Чтобы решить эту проблему, можно создать разные потоки:



1. Разделение на потоки

Бывают ситуации, что поток создается, то какую то часть времени просто простаивает.

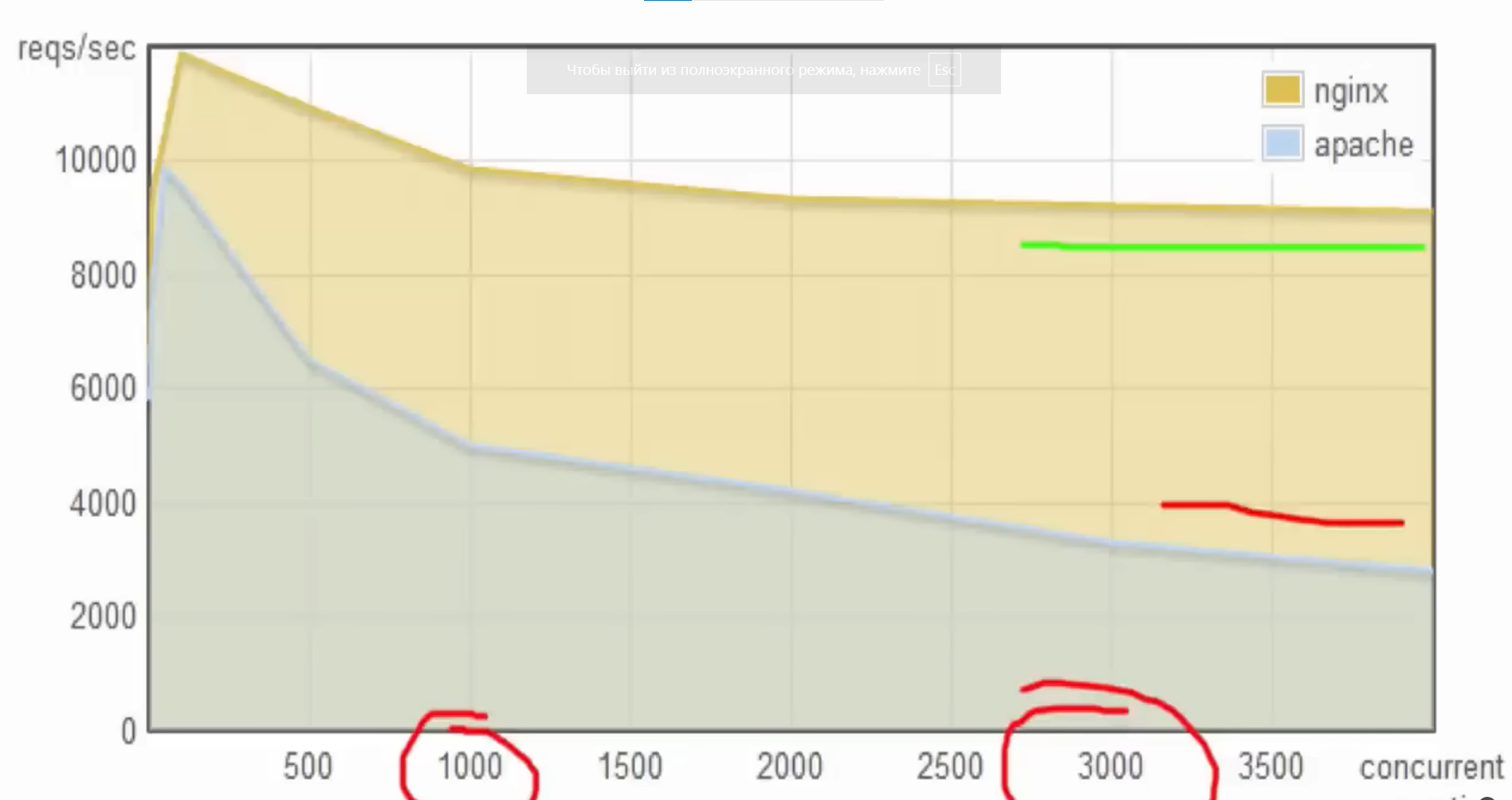


1. Простой потоков

Потоки находятся некоторое время в состоянии простоя, ожидая новых данных, получаемых из связанных с ним соединений. При этом потоки потребляют значительное количество ресурсов.

Сравнение:

* Nginx - неблокирующий ввод/вывод
* Apache - блокирующий



1. Сравнение Nginx и Apache



1. Не блокирующий поток

Сервер работает только с одним Главным потоком. Основной шаблон - активный опрос ресурсов в цикле. Шаблон цикл ожидания.

NodeJS однопоточный, вся асинхрошина достигается за счет Event loop. Однако в Libuv может управлять потока, при чем дефолтное количество равно 4. Libuv написан на C, а V8 на C++, можно писать модули и некоторые библиотеки могут быть многопоточными!!!

# Планировщик потоков

# Демультиплексор событий и шаблон Reactor

# Event loop в node js. Фазы и очереди