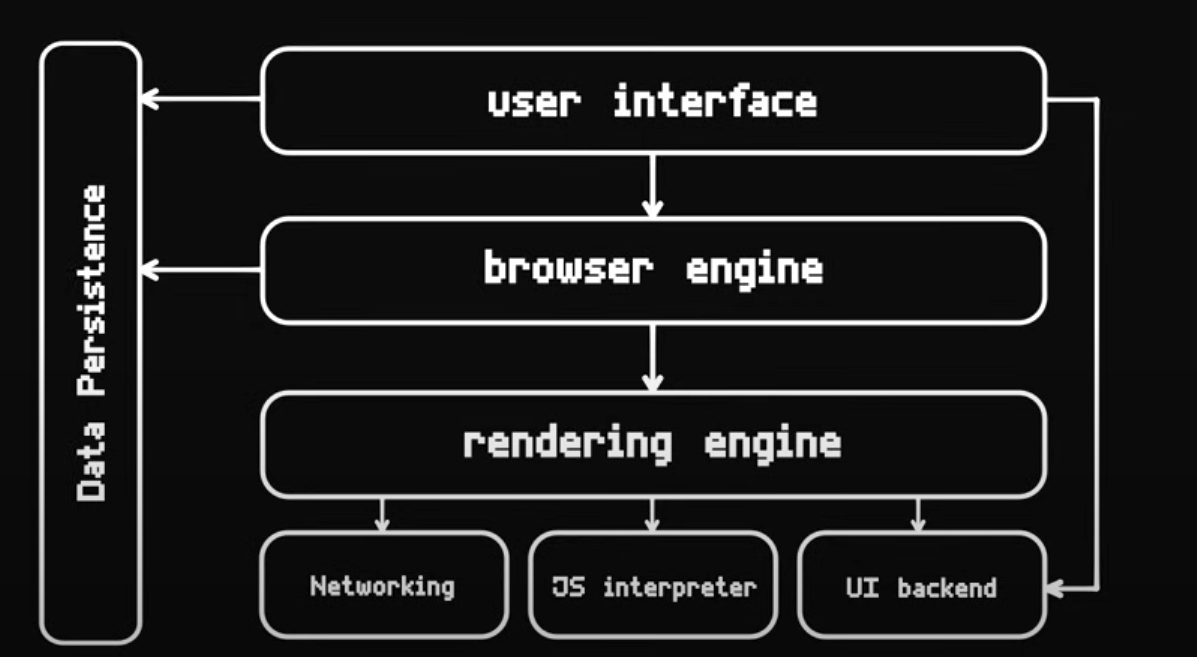


Начнем с того, из чего каждый браузер состоит, скажем так базовая схема архитектуры любого браузера.



User Interface. Это часть браузера, с которой мы взаимодействуем. Как правило, это верхняя или какая-то боковая панель. Это пространство, где мы вводим URL, нажимаем на кнопки вперед-назад, где различные настройки и все, с чем мы в принципе взаимодействуем через UI.

Browser engine - соединительная часть между пользовательским интерфейсом и механизмом рендеринга. На основе входных данных от пользователя он взаимодействует с этим механизмом рендеринга и управляет им.

Rendering engine – получаем страницы, приложения, с которыми мы в браузере уже можем взаимодействовать. Два самых популярных таких движка - WebKit, который используется в Chrome и Gecko, который используется в Firefox. И основная задача этого движка, как раз обработка того кода, который вы написали, HTML, CSS и  JavaScript. Строится дом-дерево, строится объектная модель, CSS-дерево, определяется расположение элементов, в общем проходят определенные стадии рендера. И после всех этих стадий мы уже видим ту самую заветную страницу в браузере.

Networking предназначена для работы с сетью. Она отвечает за историю сайтов, которые вы посещали, за доменные имена, за взаимодействие с DNS сервером, за правила обработки различных доменов, HTTP, HTTPS, открытие TCP соединения, взаимодействие по UDP, в общем, все, что связано с сетью, обмен пакетами и т.д.

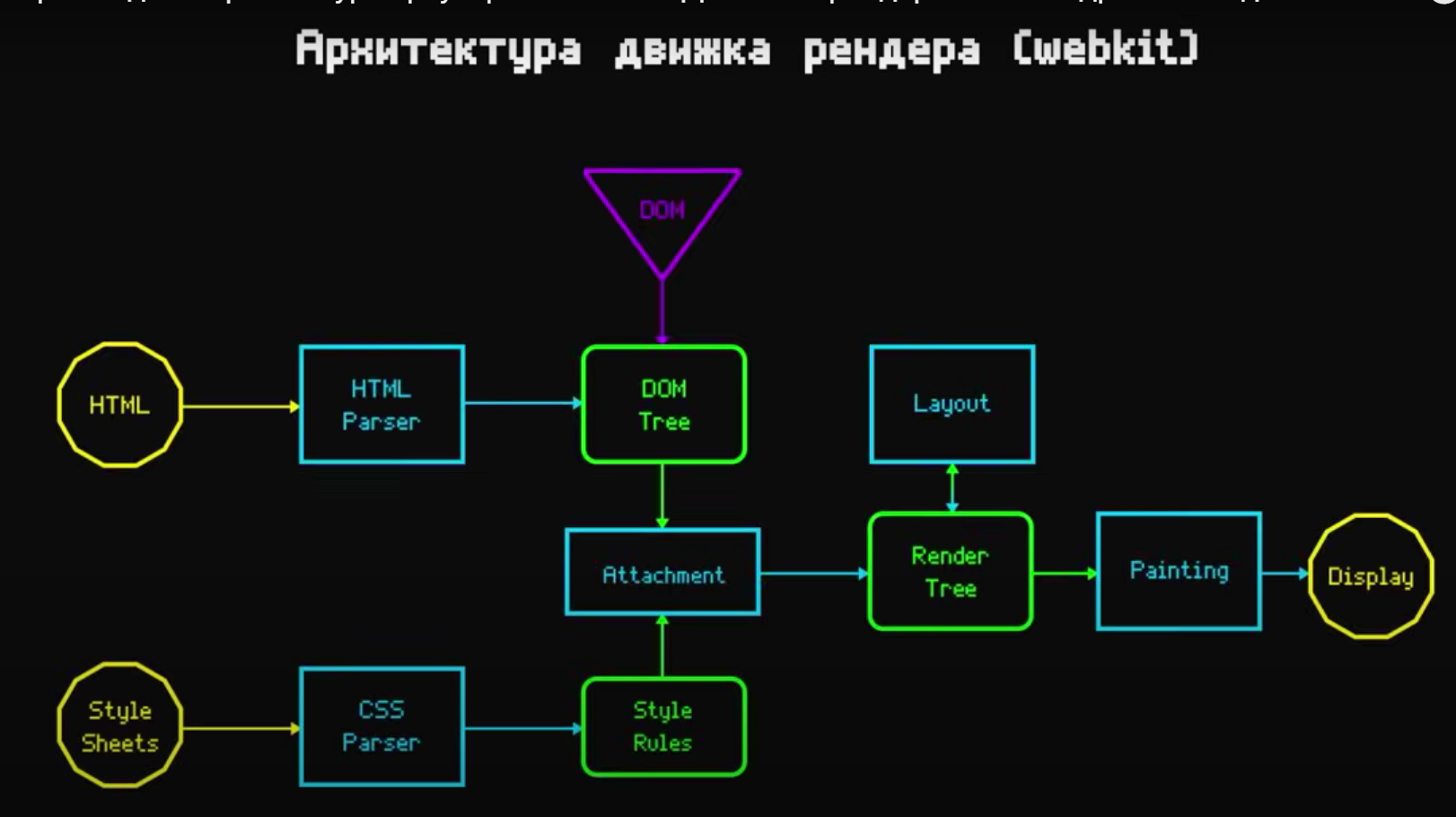
JS interpreter - движок JavaScript, это часть, которая обрабатывает JavaScript код, самый популярный движок - V8. Поверх него построен node.js, google chrome. т.е. у нас движок, который обрабатывает JavaScript используется и на платформе Node. Js, которая позволяет нам писать серверный или десктопные приложения, и также этот же движок используется в браузере, т.е. это независимая, абстрагированная технология, которая позволяет всего лишь обрабатывать JavaScript код. Про движок JavaScript мы еще отдельно будем говорить, но если кратко, он предоставляет heap, где хранятся всякие объекты, массивы, функции…. Стек вызовов (call stack) предоставляет работу с памятью, выделение, сборка мусора, компиляция JS в машинный код.

UI backend - внутренняя часть, которая предоставляет логику для интерфейса самого браузера, то есть какая-то логика подкапотная.

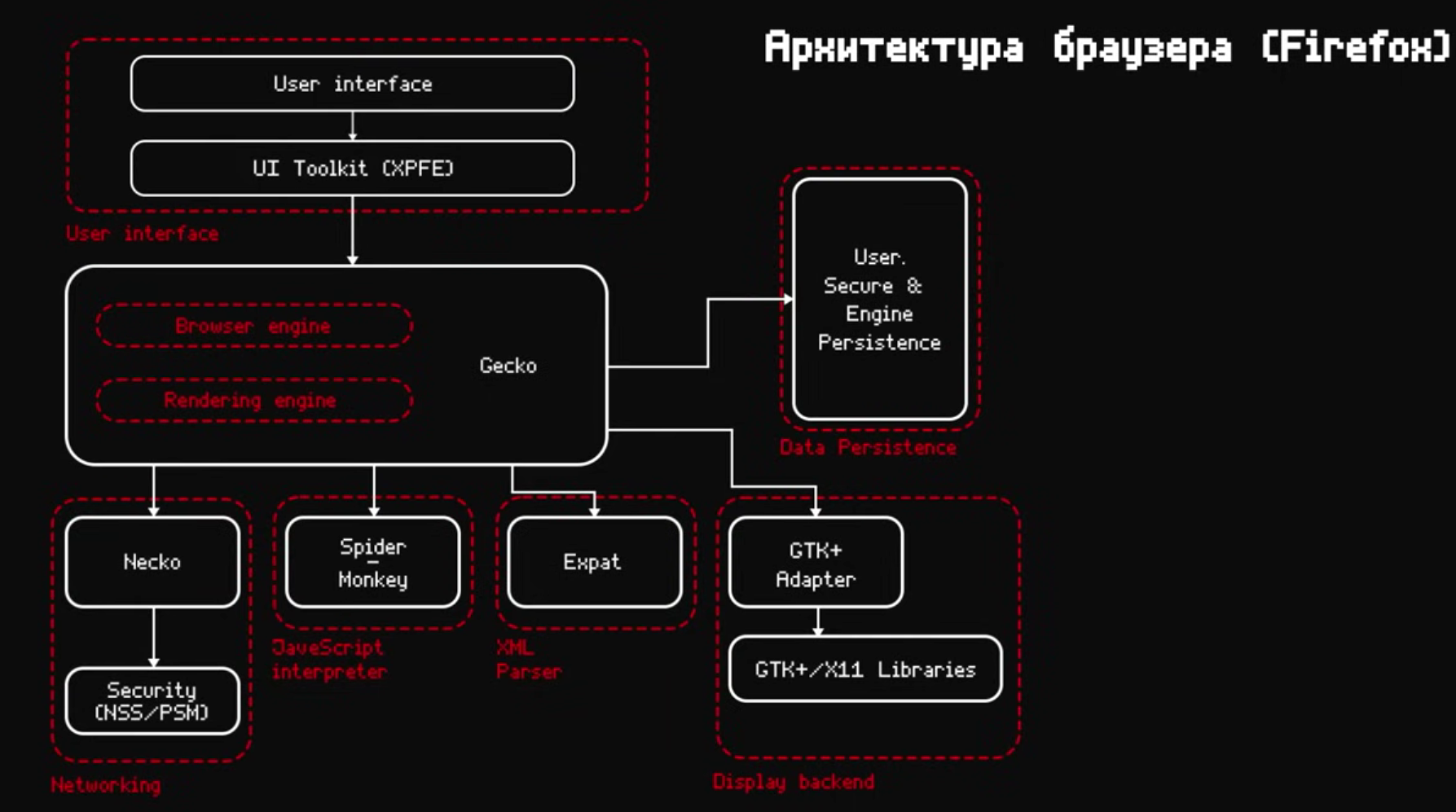
Data Persistence - хранилище данных. Это Local Storage, Session Storage, это IDB, WebSQL, файловая система, если нам, например, там картинку какую-то в браузер загрузить надо. В общем, это такое хранилище, которое использует браузер. Также он обрабатывает куки, работу со вкладками, хранит их в памяти. Это все локальная база данных.



В качестве движка рендера здесь используется Webkit, в качестве движка JS используется V8, тот же самый, что используется под капотом в node.js. Здесь у нас еще появляется parser XML, появляются Plugins, Network Stack.



То есть на вход в него приходит html, css, он парсит html с помощью специального парсера, парсит css с помощью другого специального парсера. На выходе первого у нас получается дом-дерево, с которым через javascript мы уже работаем. На выходе второго получаются определенные правила стилей. По итогу строится дерево рендера, это ключевое дерево, за счет которого мы видим элементы на странице, и потом проходят определенные стадии, layout, painting, composite и страница рендерится и мы можем с ней взаимодействовать.

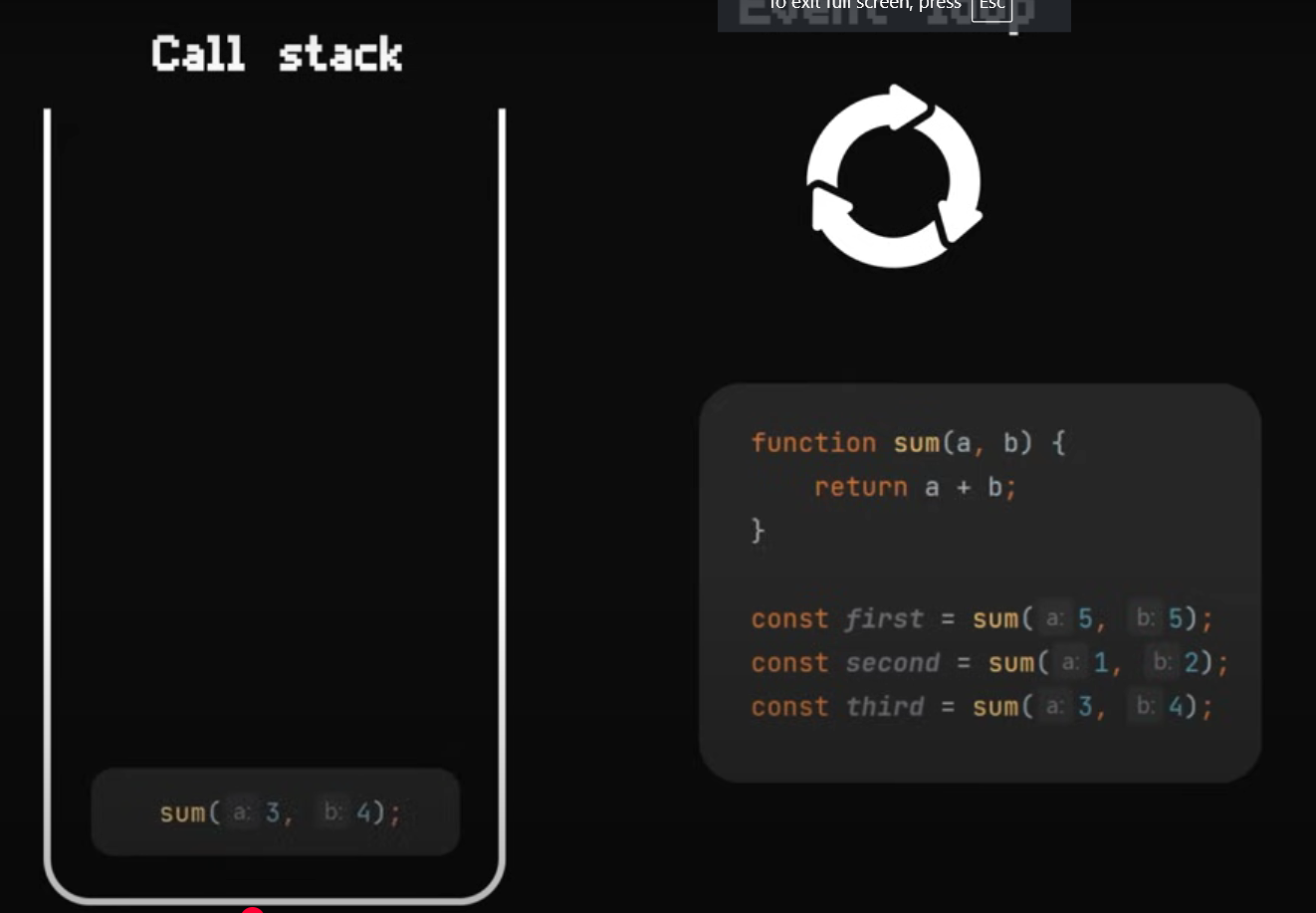


Браузер-энджин, рендеринг-энджин, джаваскрипт-интерпретатор, инструменты для работы с сетью, но, как вы можете тоже заметить, что здесь используются другие имплементации. То есть, если для парсинга джаваскрипта в хроме использовался V8, то здесь используется совсем другой движок, он называется SpiderMonkey. Если говорить про движок рендера, то здесь используется Gecko.

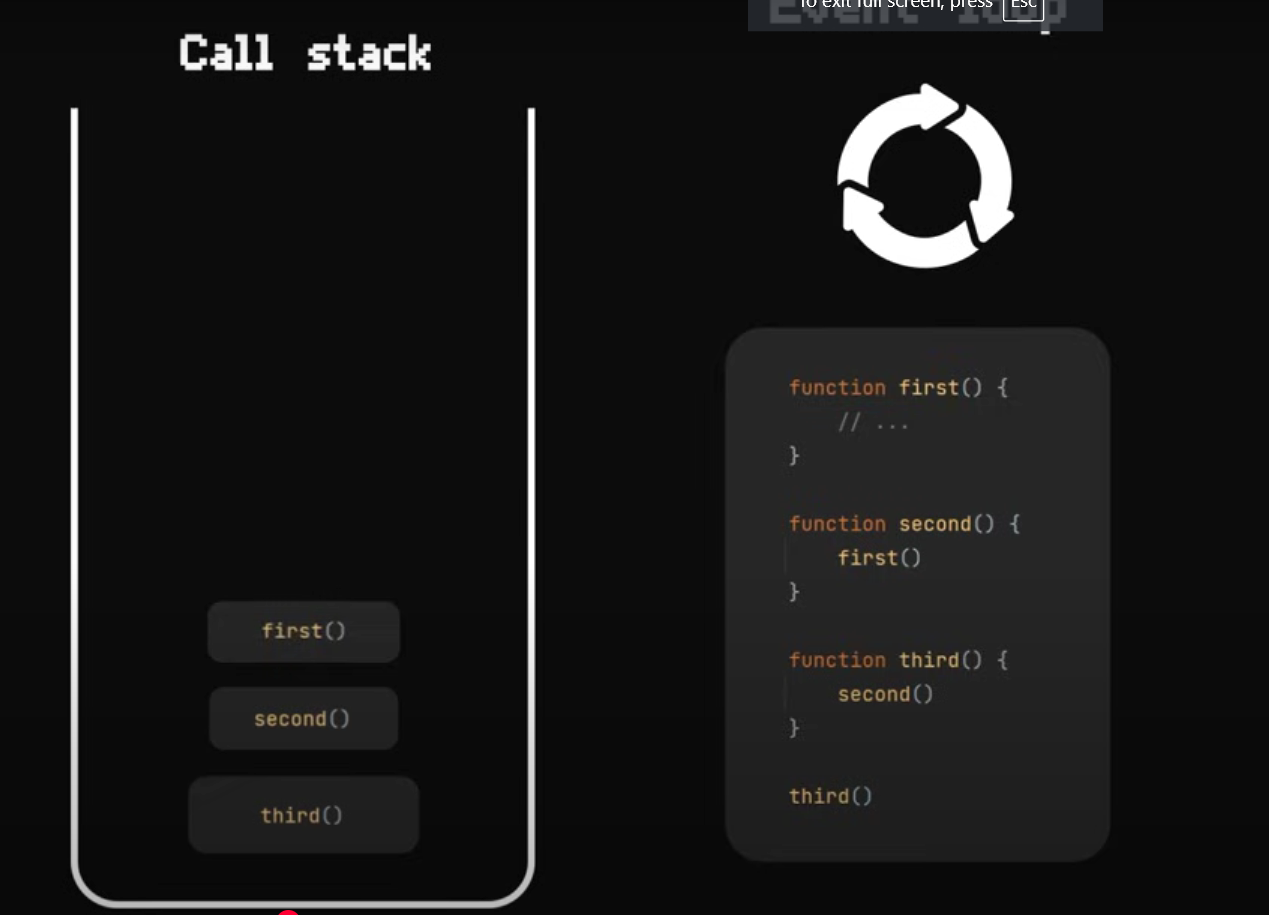
Сейчас самое важное понимать, что Event Loop не является частью JavaScript. И в доказательство этого можно сказать, что в Chrome используется V8, движок для обработки JavaScript, в Node. Js используется также V8, но при этом EventLoop в браузере и EventLoop в Node. Js — это абсолютно разные концепции, в целом решают они одну задачу, но реализованы они по-разному.

EventLoop — это отдельный механизм, который позволяет использовать неблокирующую модель ввода и вывода. Представим себе ситуацию, что у нас нет никакой асинхронщины, нет никакого event loop, а мы вот так шаг за шагом выполняем строчку за строчкой нашего JavaScript кода. Но возникает вопрос, что делать, если мы хотим вот это выполнение кода как-то распараллелить. Например, одновременно отправить запрос, нажимать на кнопку, вводить что-то в input, и при этом, чтобы у нас анимация на странице работала. И чтобы не получилось так, что мы отправили запрос на сервер, и у нас весь код заблокировался. Потому что нам важно дождаться ответа от этого сервера. То есть еще раз задача как-то чтение и выполнение этого кода распараллелить, чтобы отправка запроса, нажатие на кнопку, ввод чего-то в input не блокировало весь остальной код, чтобы анимация могла там работать где-то независимо, обработка нажатия кнопки была независима, но думаю основную идею вы поняли. И вот для этого как как раз event loop, цикл событий и предназначен.

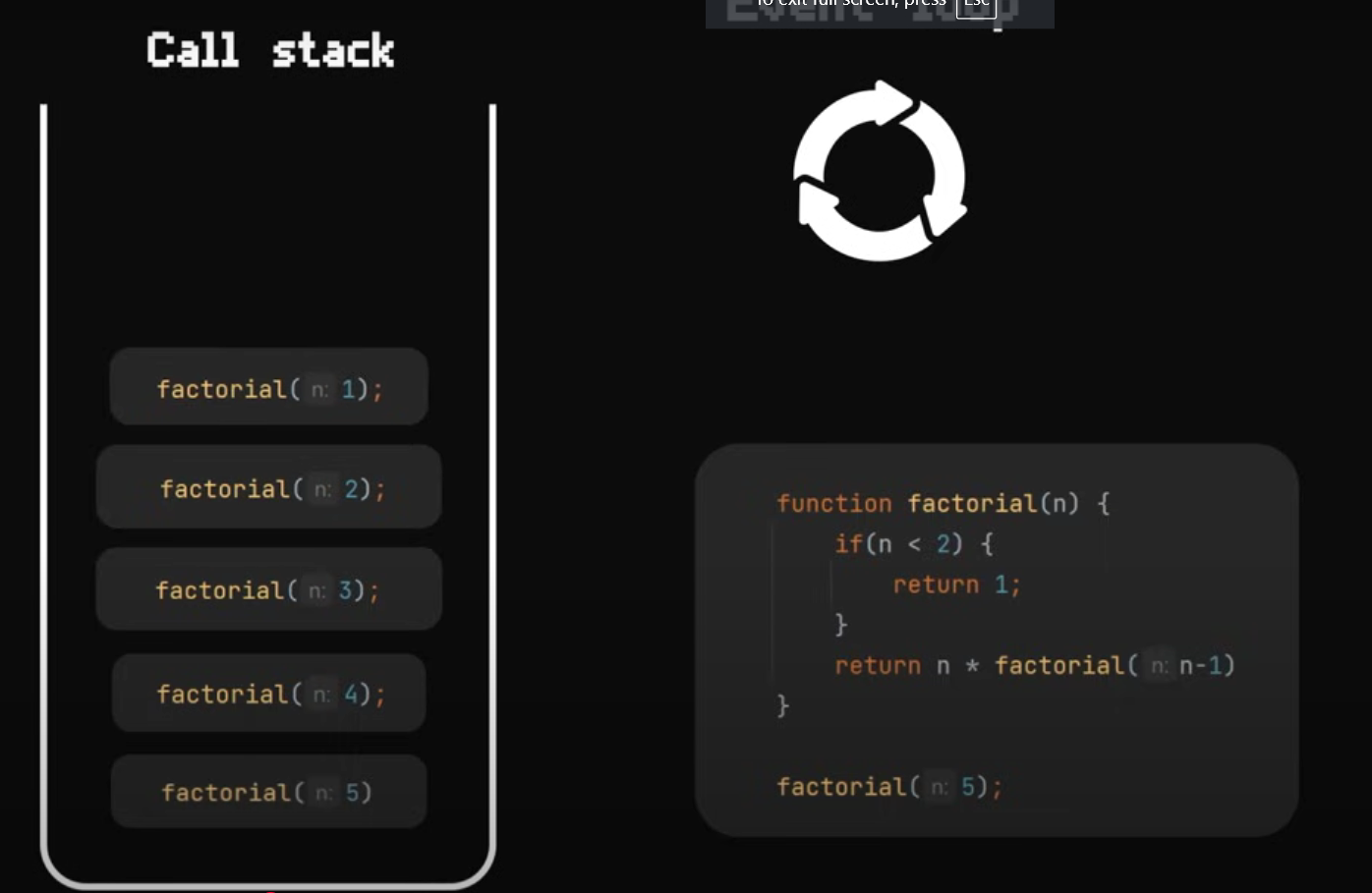
Есть стек вызовов, за его обработку отвечает движок JS (представим что v8). Стек - определенная структура данных, хранилище, которое можно представить в виде стопки бумаг, если мы хотим бумагу взять, мы ее берем сверху стопки, а если мы хотим бумагу в эту стопку добавить, то мы складываем ее так же наверх. И вот call stack как бумагу складывает функции в определенном порядке, в том, в котором они должны быть вызваны.



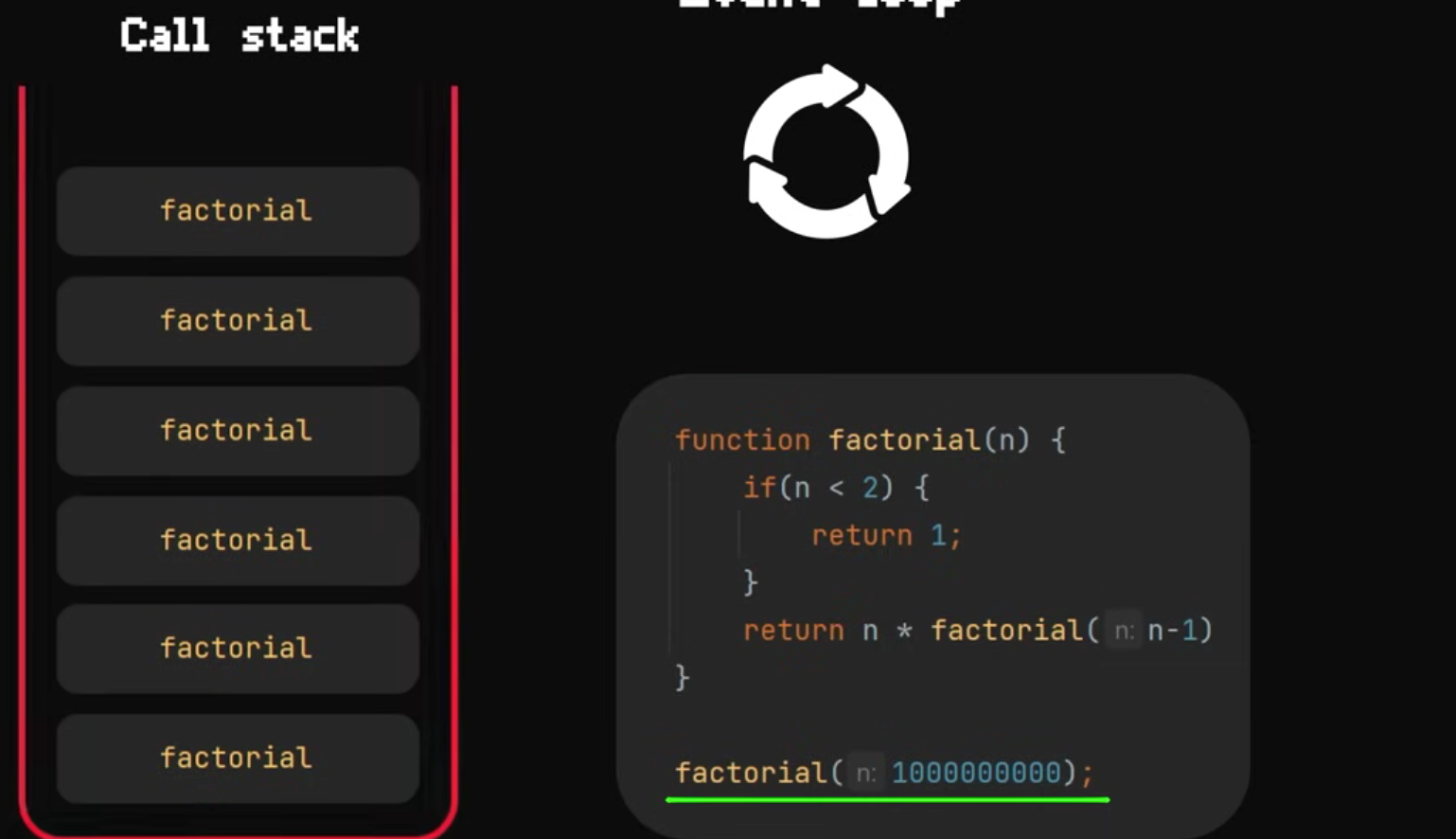
Мы выполняем одну функцию, выполняем вторую функцию, то есть она попадает в stack и сразу же из него выходит. И третья функция также попадает в стэк и сразу же из него выходит и выполняется. То есть здесь функции выполняются одна за другой.



Здесь функции вложены в друг друга. Мы выполняем третью функцию. Внутри неё содержится вызов второй функции, а внутрь неё содержится вызов первой функции. И вот теперь они поочерёдно, начиная сверху начинают вот так выходить. То есть выполняется первая функция, потом вторая и только потом третья, это как раз аналогия с той самой стопкой бумаг, так работает стэк.

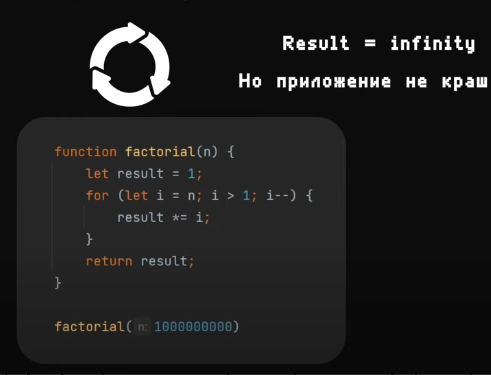


Здесь у нас рекурсия, функция факториал внутри себя вызывает другую функцию факториал, при этом есть базовый случай, который позволяет вот эту вот петлю рекурсивную закрыть. Мы вызываем факториал с пятеркой, затем вызываем с 4, с 3, с 2 и с 1. При этом здесь у нас на базовом случае рекурсия завершается. Мы начинаем в таком порядке эти функции выполнять, начиная с единички и так до пятерки. В итоге мы получаем нужный нам результат, но есть один очень важный нюанс.

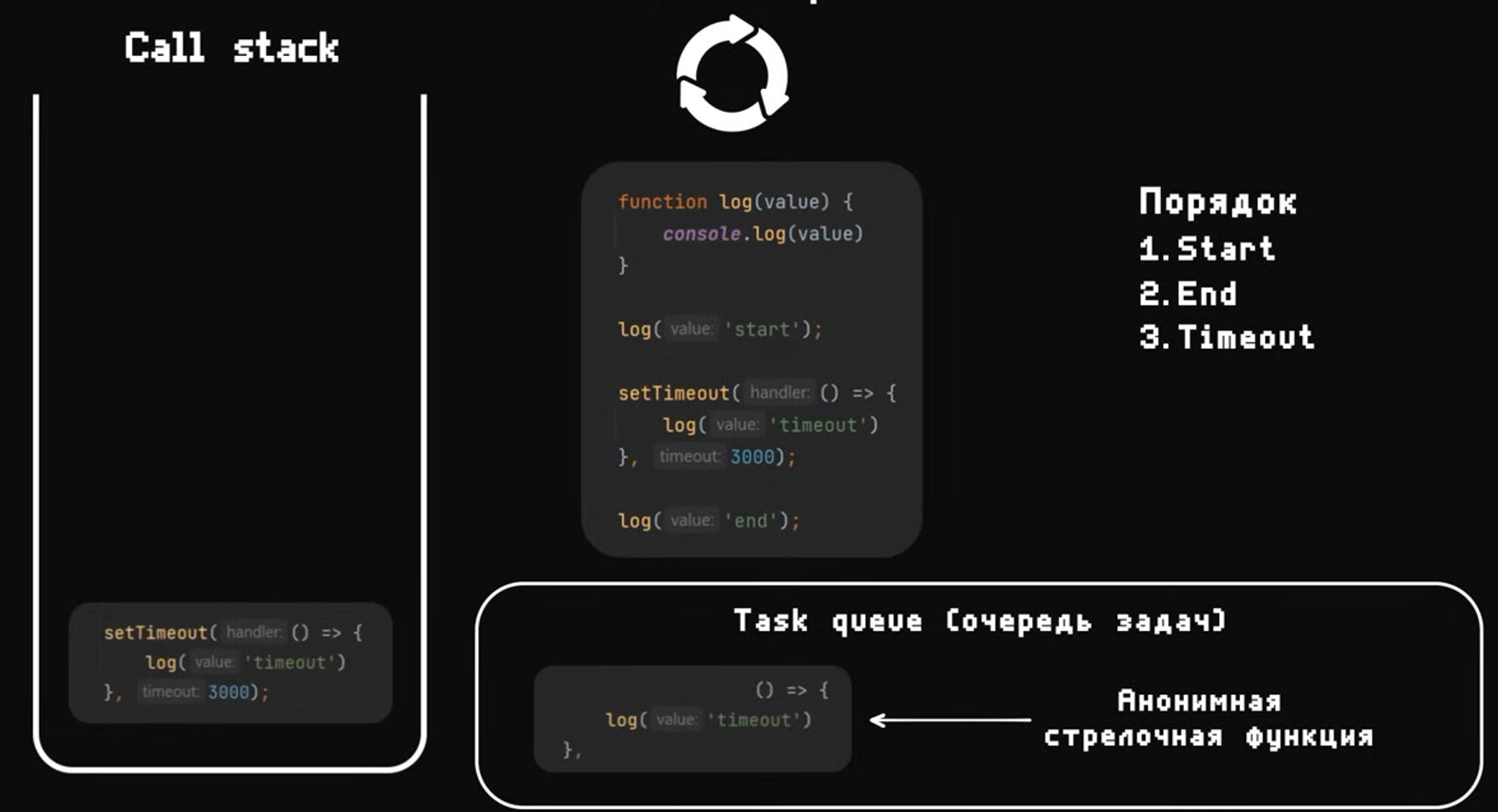


Call stack не бесконечный, то есть при определенных условиях его можно переполнить и приложение у нас крашнется, Например, если мы вызовем факториал с очень большим числом, у нас Call stack переполнится и приложение выстрелит ошибкой.

Чтобы здесь не было переполнения стэка вызовов, мы могли решить эту задачу с помощью вот такого вот обычного цикла, перебором.



Да, мы бы в результате получили бесконечность, потому что джава скрипт с такими большими числами работать не умеет, но приложение при этом у нас не крашнулось бы.



Примеры, которые мы рассматривали ранее, являются синхронными. То есть код выполняется строго в определенном порядке шаг за шагом. Но что, если мы хотим отложено, через 3 секунды выполнить какое-нибудь действие, например, показать пользователю сообщение о том, что он провел на нашем сайте 5 минут. Давайте посмотрим, например, кода. То есть у нас есть lock start, lock end и посередине у нас set timeout, который выполнится через 3 секунды. И давайте рассмотрим, как с точки зрения event loop этот код должен выполняться. EventLoop содержит внутри себя очередь. Эта очередь называется очередь задач. То есть порядок выполнения кода здесь будет такой: lock start, lock end и только потом timeout. У нас попадает lock в call stack, затем у нас туда попадает set timeout, но callback, который мы передаем внутрь, тот самый handler, в котором выводится timeout, он не выполняется сразу, то есть он не попадает в этот call stack, а попадает в очередь задач. То есть это вот в нашем случае анонимная стрелочная функция, которую мы передали внутрь setTimeout. Она остается в этой очереди и находится там. То есть мы ее запомнили и зарегистрировали. Потом у нас выполняется lockEnd, и только после того, как у нас прошли 3 секунды, у нас выполняется вот эта стрелочная функция, и внутри нее выполняется lockTimeout. Задачи из очереди могут попасть в стэк только после того, когда этот стэк полностью очистится, то есть все функции должны выполниться. с

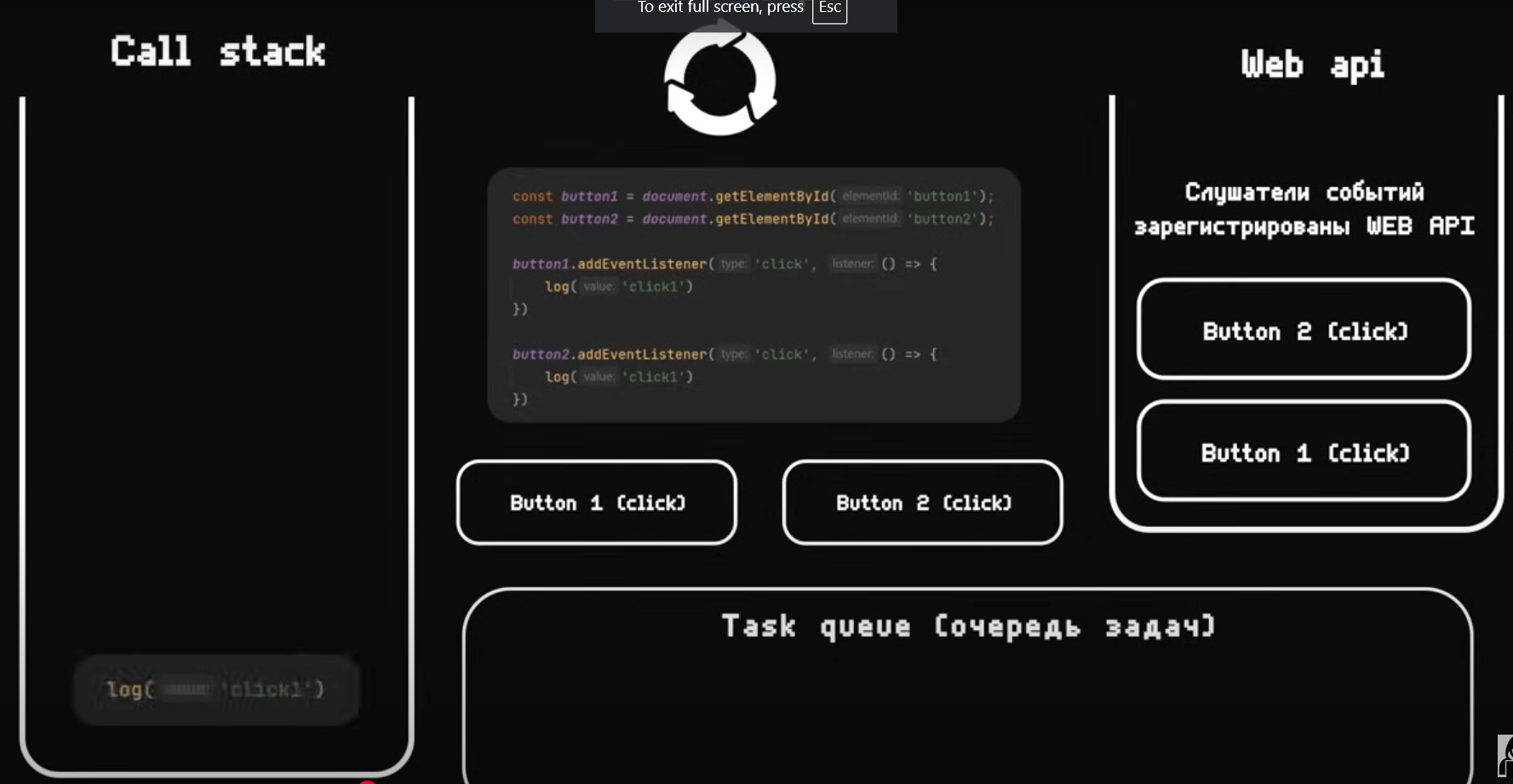
Как вот эти вот задачи попадают в очередь? За очередь отвечает event loop, а за call stack у нас отвечает движок javascript, это может быть v8 (chrome), chakra (Microsoft edge), spider monkey (mozilla) или javascript core (safari).

Движок javascript решает следующие 4 основные задачи. Во-первых, это работа с кучей, так называемых heap и стеком вызовов. Работа с памятью, выделение памяти и сбор мусора. Компиляция JS в машинный код. Отвечает за различные компиляции скрытые - кэши, скрытые классы…

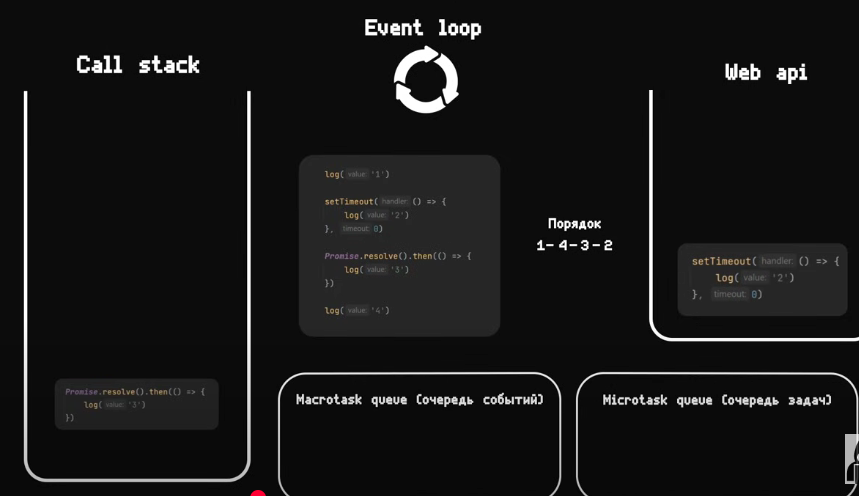
Event Loop предоставляется средой, это может быть браузер, Node.js или какая-либо другая среда разработки, в которой этот event loop требуется.

Если у нас движок JavaScript предоставляет call stack, а очередь задач предоставляет event loop, как они общаются между собой? У каждого браузера есть web API. Это API предоставляет тайм-ауты, обработку слушателей, события нажатия на кнопки, различные события, загрузки изображений, файлов, отправку каких-нибудь fetch-запросов. То есть это все не спецификация JS, это все исключительно браузерные штуки. И вот давайте тот же самый пример рассмотрим, но теперь с участием web API.

Точно так же setTimeout у нас попадает в call stack, после этого он регистрируется в web API и вот на этом этапе запускается таймер. После того, как таймер иссяк, мы отправляем callback в очередь задач и он после того, как call stack отчистился, когда настало время взять задачу из очереди, он выполняется. То есть порядок такой, выполнились все синхронные задачи, после истечения таймера задача попадает в очередь, и после того, как все синхронные задачи выполнены, мы берем задачу из этой очереди и выполняем ее.



Точно так же addEventListener попадает в call stack, выполняется, после чего web API регистрирует слушатель события на кнопку. То же самое происходит с второй кнопкой. И мы знаем, что эти события у нас зарегистрированы. При этом интерфейс у нас не блокируется. И после того, как мы нажмем на эти кнопки на интерфейсе, у нас также генерируется task в очередь задач. И после того, как выполнились все синхронные задачи с call stack, у нас task выполняется. То есть за счет вот этой асинхронной модели мы можем иметь одновременно в нашем приложении тысячи слушателей события на кнопки, input, select. При этом эти слушатели события будут зарегистрированы пока мы их явно не удалим через removeEventListener(). Все кнопки, события, это не спецификация JS, это браузерная часть — браузерная web API.



В коде появляется еще Promise. С помощью Promise мы также можем запускать какой-то асинхронный код, который не будет блокировать основной поток. Браузерный Fetch для получения каких-то данных работает именно на Promise. Но как вы можете заметить в setTimeout у нас в качестве таймаута передано 0 миллисекунд. Что запустится быстрее Promise или setTimeout? Порядок: 1-4-3-2.

Как event loop понимает, какие задачи брать ему быстрее? Есть очередь событий (макрозадач) и очередь задач (микрозадач). У этих очередей есть определенный приоритет, и Event Loop берет task в определенном порядке. Web API зарегистрировалось setTimeout, а Promise на текущий момент находится у нас в call stack. Promise всегда попадают в очередь microtask. Promise у нас попал в очередь microtask, затем у нас выполняется лог 4, и на текущем этапе у нас все синхронные задачи выполнены, то есть call stack у нас пустой. Начинает работать Event Loop и забирать task из очередей. + в какой-то момент нулевой тайм-аут у нас выполняется, WebAPI его обрабатывает и перемещает callback, который находится внутри тайм-аута в marcotask очередь.

В приоритете, в первую очередь, всегда выполняются микротаски, причем сразу все, и только потом выполняется одна макротаска. У нас сейчас выполняется лог 3 из promise, и только потом у нас выполняется лог 2, который находится в setTimeout. Порядок: синхронные, микротаски и макротаска.

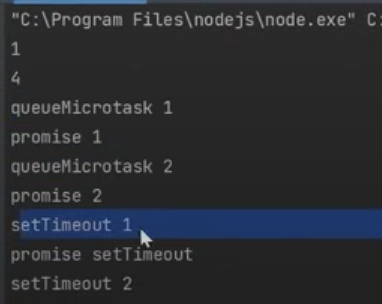
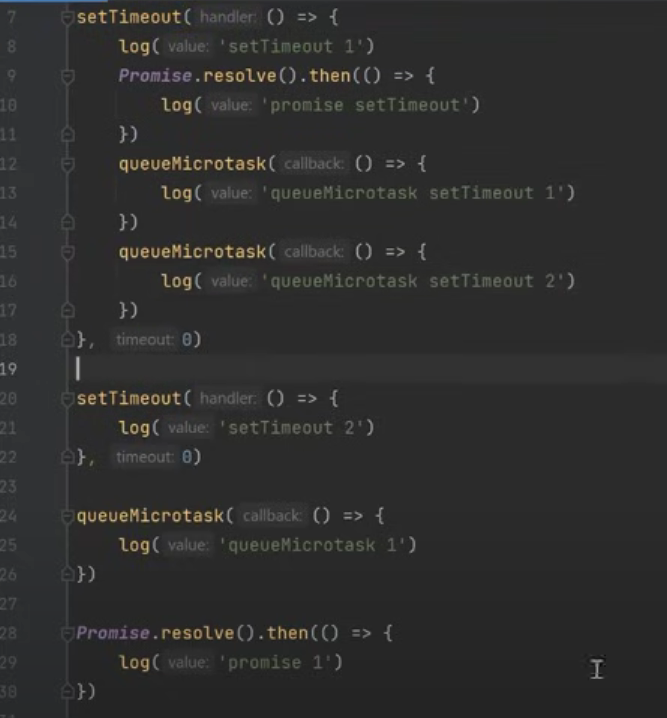
Несколько их. По очереди выполнит все микротаски одну за одной, и после того, как очередь микрозадач освободилась, берется одна задача из очереди макротасок. Давайте представим, что у нас там какой-нибудь se tTimeout, который внутри себя выполняет фетч-запрос. Это тоже promise. Макрозадача порождает микрозадачу, promise внутри себя. И вот здесь обратите внимание, макротаск под номером 2 у нас не выполняется. У нас выполнится микротаск, который породился макротаском, и только потом у нас event loop забирает задачу из очереди макрозадач, вторую задачу.

Сначала выполняются все синхронные задачи, потом выполняются все микротаски, выполняется одна макротаска, если она порождает другие микротаски, то выполняются опять они, все сразу, потом выполняется вторая макротаска. И так по кругу, одна макротаска, все микротаски, одна макротаска, все микротаски.

Микротаски на 99% это всегда Promise. Но если возникла необходимость, микротаску можно создать явно с помощью функции queueMicrotask. И также микротаски порождают mutationObserver —специальный инструмент, который позволяет следить за DOM-нодами.

Макротаски: таймеры (setTimeout, setInterval), события (загрузка изображения, клик, ввод в input), браузерные нюансы (рендер, input/output и все что браузерная подкапотная.

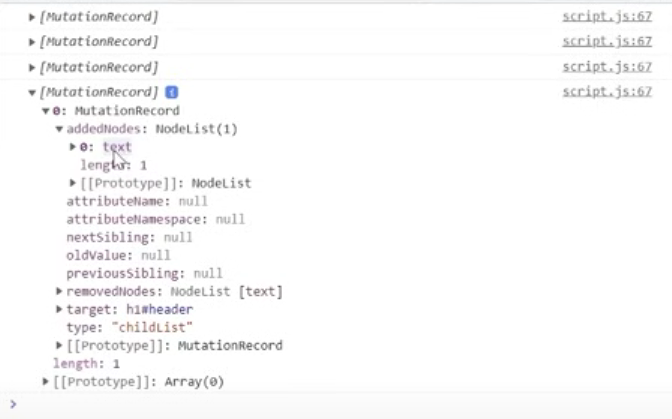




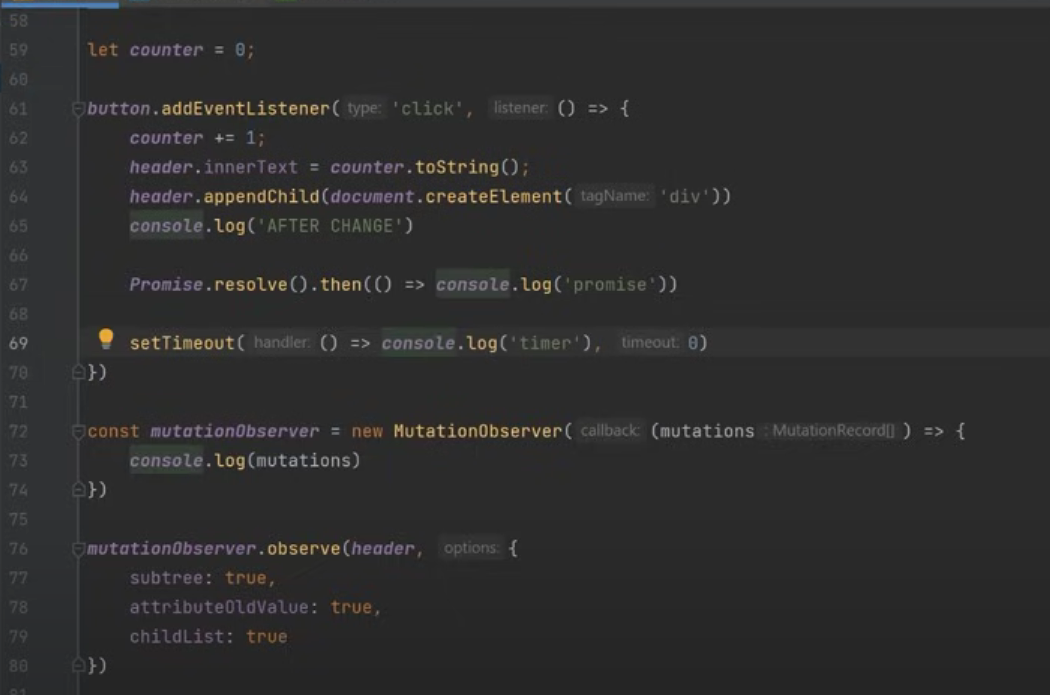
mutationObserver. Для общего развития давайте посмотрим, что он из себя представляет. Для лучшего понимания сразу рассмотрим на примере. У нас есть вот такая простенькая HTML, в которой есть заголовок и кнопка, которая в этом заголовки увеличивает на единичку счетчик. 0, 1, 2, 3, в общем обычная инкрементация. Так вот теперь давайте попробуем воспользоваться mutationObserver. Если кратко, то это специальный механизм, который позволяет следить за изменениями в DOM. Как видите, в него передается вот такой callback и в нем есть вот такой вот MutationRecord. Здесь можно следить за добавленными нодами, за какими-то предыдущими значениями, за удаленными нодами, то есть можно следить за манипуляциями, которые происходили с нодами.



Чтобы получше понять, как эти мутации выглядят, давайте выведем их в логи. И также, помимо создания самого mutationObserver, ему необходимо сказать, зачем мы будем следить. И указываем, что следить мы будем за header. Также можно передать опции, в которых можно указать атрибуты, за которыми мы хотим следить. Можно указать, хотим ли мы следить за поддеревом, хотим ли мы следить за предыдущим значением.

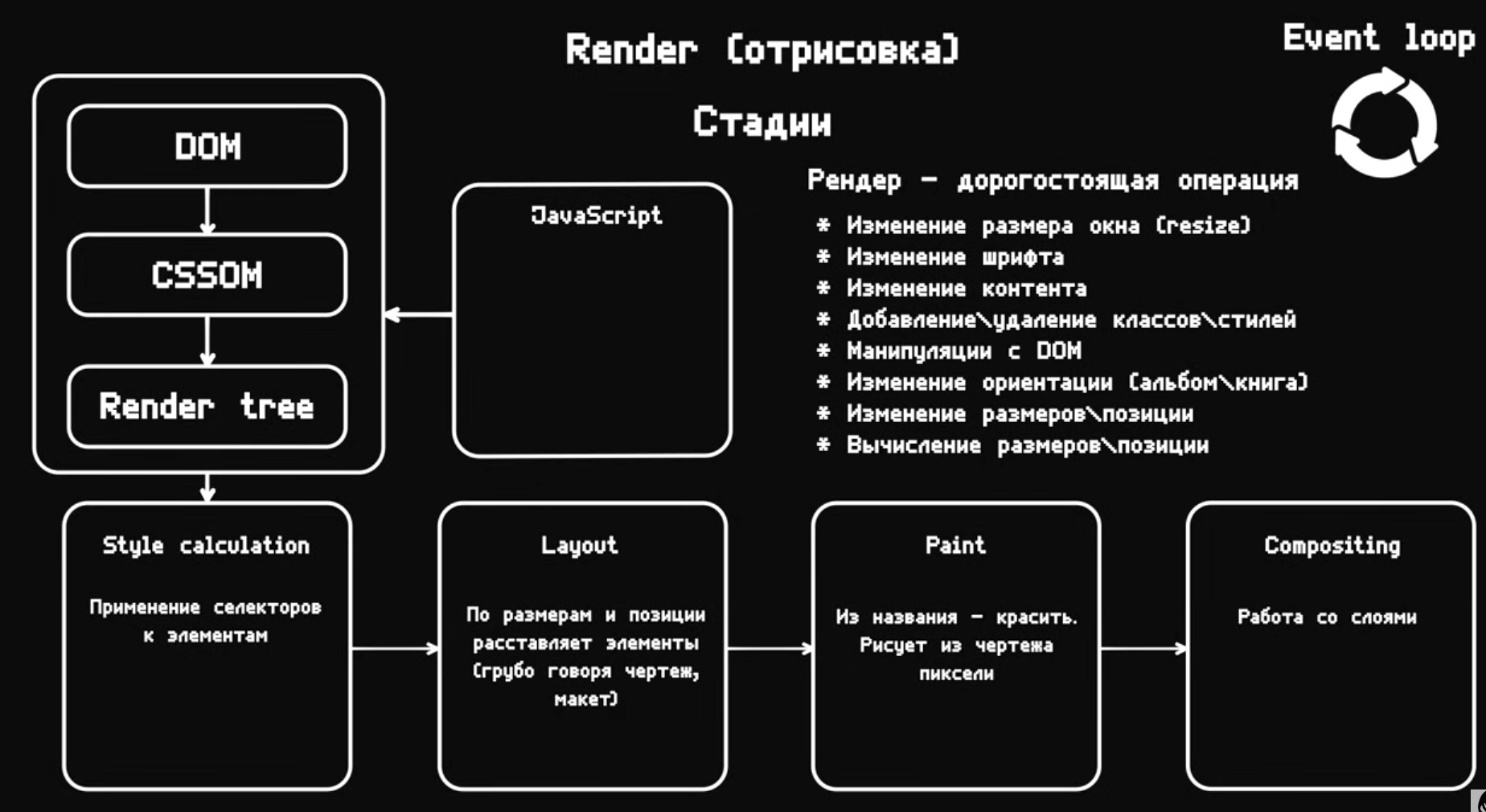
Открываем консольку, нажимаем на кнопку и видим как у нас в логах появляются вот эти самые MutationRecord. Если мы зайдем, то мы увидим, что здесь у нас есть как раз все необходимые данные, можем проследить какие ноды были добавлены, видите вот эта нода типа текст, то здесь у нас в качестве дата передана была 4. И на основании этих данных мы можем производить тоже какие-то манипуляции, следить за изменением того или иного элемента.



Давайте попробуем теперь не заменять HTML внутри самого counter, а попробуем добавить у него какую-нибудь ноду с помощью appendChild(). С помощью document.createElement() создадим какой-нибудь div, ну и в принципе наполнять его ничем не будем, как есть передадим. Обновим страницу, понажимаем на кнопку, как видим теперь в MutationRecord у нас две записи, то есть дважды объект был изменен, ну и мы видим, что у нас добавилась новая нода. Это у нас уже не просто text, а это уже непосредственно div.

Прямо внутри eventListener() давайте выведем логи. Сразу после того, как мы поменяли дом ноду, напишем after change, потом сделаем Promise.resolve(). Здесь тоже выведем что-нибудь в консоль логи. И добавим setTimeout. И если mutationObserver действительно генерирует микротаску, то мы увидим ее перед выполнением Promise.resolve(), т.е. после after change мы в логах должны получить список наших мутаций, и только потом мы должны получить set timeout. -- Нажимаем на кнопку, получаем сначала синхронный консоль лог, потом получаем записи MutationRecord, потом Promise и только потом таймер. Из этого можно сделать вывод, что действительно после изменения какой-то дом ноды, у нас mutationObserver стреляет микротасками.

И сейчас, думаю, подходящий момент рассмотреть то, как браузер отрисовывает непосредственно ту страницу, с которой мы взаимодействуем в браузере.



«Калькуляция стилей». На этом этапе браузер применяет селекторы, которые мы пишем к элементам, то есть он по этим селекторам проходится и определяет, какие стили и к чему необходимо применить.

Тут важно понимать, что для одного и того же элемента на странице могут быть применены разные селекторы в разных файлах, с разными стилями и вот это все надо подсчитать, поэтому этапы называются калькуляция стилей и уже понять, к чему и что мы применяем. И вот здесь тоже важно понимать, чем сложнее у вас селектор, если у вас там есть какие-то вложенности, подсчет каких-то чилдронов. И вот чем больше вот таких вот селекторов, тем конечно же подсчитать эти стили сложнее и тем больше нагрузка на ваше железо.

«Лайаут». Этот этап достаточно ресурсоемкий. Составление макета, чертежа нашей страницы. Он понимает, на какую часть, с какими отступами и куда необходимо разместить тот или иной элемент. Это когда без реальных цветов, без реальных каких-то иконок, просто позиционирование, расположение каких-то элементов на странице. Строит прототип, и на выходе после этого этапа мы получаем так называемый layout tree, дерево расположения, дерево позиционирования, дерево макета. И если какой-нибудь дом-дерево содержит полную информацию о каждой ноде, из которой состоит наша страница, то вот это layout tree содержит в себе информацию о том, где элемент должен располагаться, виден ли он, не виден и так далее.

Paint. На выходе из предыдущего этапа мы имели вот этот каркас, прототип и его остается в соответствии со стилями, которые мы получили на первом этапе разукрасить. Строится paint records определенной записи самой отрисовки.

Сompositing. Браузер делит то, что мы получили на определенные слои и создает дерево слоев. Если на этапе layout у нас было layout tree, то на этом этапе у нас получается layer tree. Браузер работает со слоями, делает он это так, чтобы у нас правильно все это обрабатывало видеокарта, чтобы у нас правильно располагались элементы на странице в зависимости от Z-индекса и так далее.

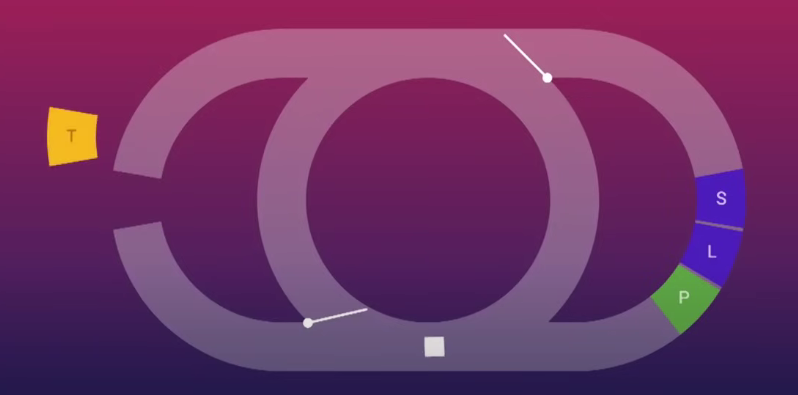
Когда у нас строится дом-дерево, когда у нас строится дерево стилей? На самом деле происходит это все чуточку раньше, как только мы получили непосредственно html и css, в этот момент строится дом, css-om, css-object-model и непосредственно Render tree, а потом уже идет калькуляция стилей, layout, paint и compositing.

При первой отрисовке работает это все именно так, но что будет если мы нажмем на какую-нибудь кнопку, она вызовет какую-нибудь анимацию и при этом у нас еще откроется модальное окно, то есть по сути у нас и стили меняются и отображение элементов на странице меняется и при этом дом дерева тоже может меняться, потому что у нас появляется в нем еще и модалка. И здесь по сути ничего не меняется. Мы нажимаем на кнопку, у нас JS триггерит изменение стиля шрифта и у нас вот эти стадии идут по кругу, у нас генерится новое дом дерева, ну какая-то часть его меняется, меняется какая-то часть CSS Object Model, строится новый рендер tree и проходят все вот эти стадии по кругу.

И вот здесь важно понимать, что рендер - очень дорогостоящая операция. И поэтому важно стараться работать с дом-нодами как можно меньше. И если вы с ними работаете, то надо стараться затрагивать как можно, скажем так, нижние части этого дерева, то есть не перерисовывать большой кусок под дерево, а перерисовывать как можно меньше, это вызовет как бы меньшую нагрузку. Если вы работаете с каким-нибудь React, об этом по большей части думать не нужно, потому что под капотом React он работает с виртуальным дом-деревом и к манипуляциям с реальным дом-деревом он приступает, когда что-то действительно изменилось. А вот если вы пишете на нативном JavaScript, об этом всегда стоит помнить, потому что иногда бывает так, что изменяется, например, какой-нибудь один элемент в списке, а код написан таким образом, что там весь список перезатирается ради того, чтобы изменить одну дом ноду. И вот такого, конечно же, стоит избегать.

Теперь давайте разберемся, что вообще вызывает рендер. Это изменение размера окна, когда вы его уменьшаете или наоборот увеличиваете. Изменение шрифта, изменение контента, что по сути одно одно и то же, добавление или удаление новых классов или стилей, манипуляции с дом-деревом напрямую, изменение ориентации, когда вы телефон разворачиваете в горизонтальное или вертикальное положение, когда вы меняете размер или позицию, в том числе через transform, это тоже меняет и тоже надо пересчитывать, хотя это не затрагивает соседние элементы, а только элемент, которому вы эту трансформацию применяете.

Стадии в флоу синхронно-асинхронного кода тоже имеют свое определенное место. И браузер понимает, когда эти операции необходимо делать и в каком порядке. То есть там какие-то свои внутренние оптимизации тоже есть.

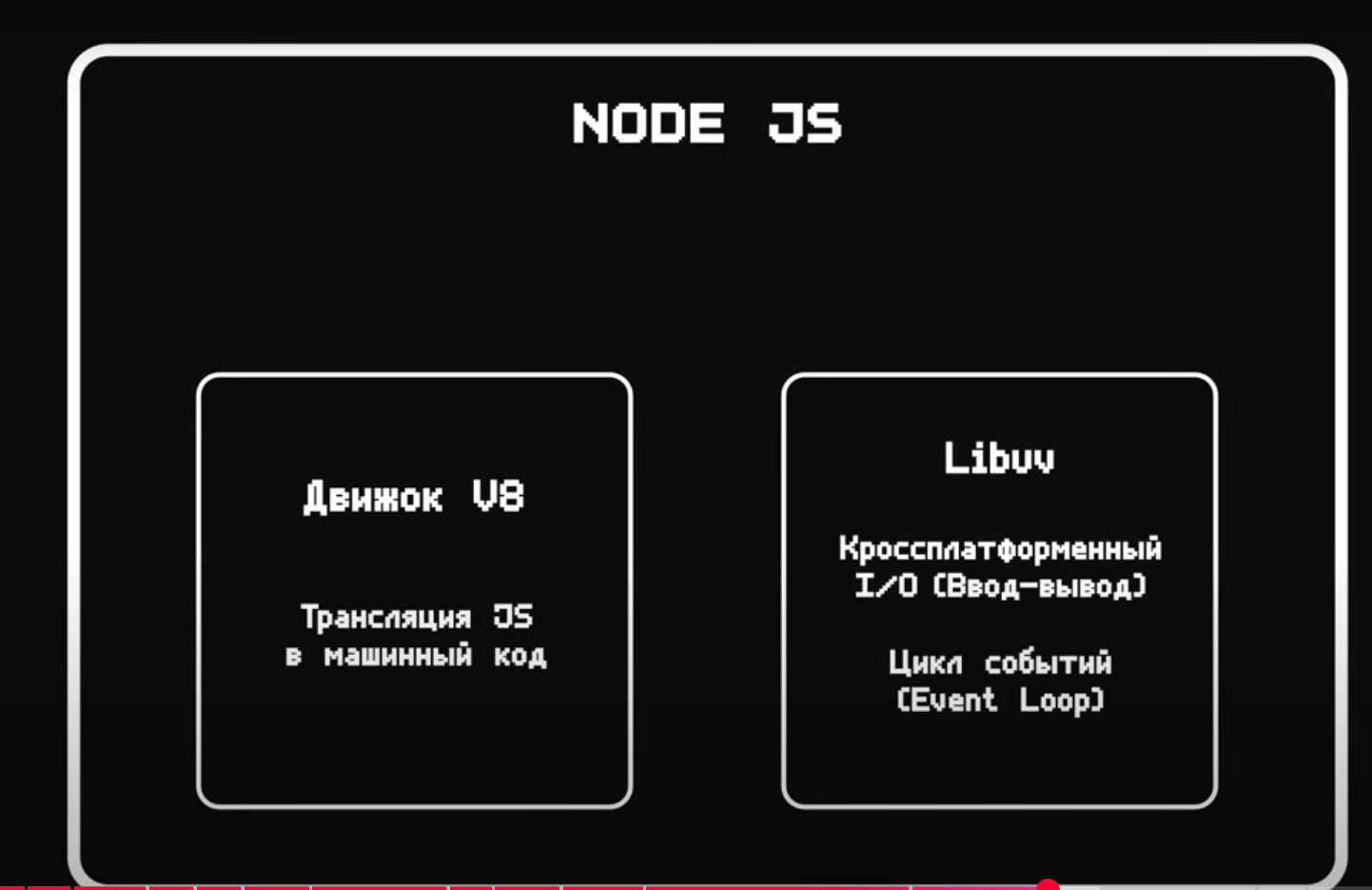


Это все вот так по кругу крутится, выполняются таски, микротаски, задачи из call stack'а, производится рендер и у этого всего есть определенный порядок. Итак, вот она эта визуализация, центральная часть это у нас синхронные задачи.

Слева у нас расположены какие-то таски, а справа непосредственно рендер. Первая стадия калькуляция стиля S. Вторая стадия у нас была layout L. И третья стадия у нас была paint P. И вот, соответственно, Event Loop наш крутится, крутится, выполняет какие-то задачи, потом в определенный момент браузер понимает, что ему необходимо интерфейс обновить. Он открывает вот эту вот заслонку и у нас проходят все стадии рендера. То есть перерисовка при изменении каких-то визуальных частей происходит немгновенно. Браузер достаточно умный, там встроены оптимизации внутри и он понимает, что накопилась какая-то порция изменений и эту порцию он перерисовывает сразу для того, чтобы вот эти стадии не проходить несколько раз. Когда у нас появляется какая-то асинхронная задача, у нас открывается левая заслонка, и тогда браузер выполняет асинхронную задачу, то есть вот в таком вот формате он крутится, крутится, крутится, выполняет асинхронные задачи, выполняет асинхронные задачи, делает перерендер.

Call stack — это часть движка JavaScript и отношения к Event Loop Call stack не имеет. Event loop это в случае браузера по сути две очереди, макро очередь и микро очередь.

Очередность выполнения: сначала выполняются все синхронные задачи с call Stack, затем выполняются все микротаски, затем выполняется одна макротаска. Если эта макротаска поражает микротаски, то тогда выполняются опять все порожденные микротаски и затем выполняется одна макротаска. И так до тех пор, пока все задачи не будут выполнены.



Нода — это программная платформа. С помощью специальных инструментов нода позволяет превращать JavaScript в машинный код. Изначально JavaScript — это язык разработки браузерных приложений, и некоторая функциональность там очень ограничена. Например, взаимодействовать с операционной системой мы не можем. Работа с файловой системой крайне ограничена. Node. Js добавляет возможность для JavaScript взаимодействовать с устройствами ввода-вывода через свой специальный API, написанный на C++. Задача Node. Js - преобразовать исходный код на JavaScript в машинный код. Подобное преобразование достигается за счет движка V8, на котором построен Node. Js. V8 — это движок, который разрабатывает компания Google, в него вливаются огромные деньги, и он также используется во всеми используемом браузере Google Chrome. V8 написан на C++ и работает он как на Винде, на Маке, так и на Линуксе. Его задача скомпилировать и выполнить исходный код на JavaScript. Также он занимается выделением памяти для объектов и занимается сборкой мусора, то есть удаляет из памяти те объекты, которые ему больше не нужны.

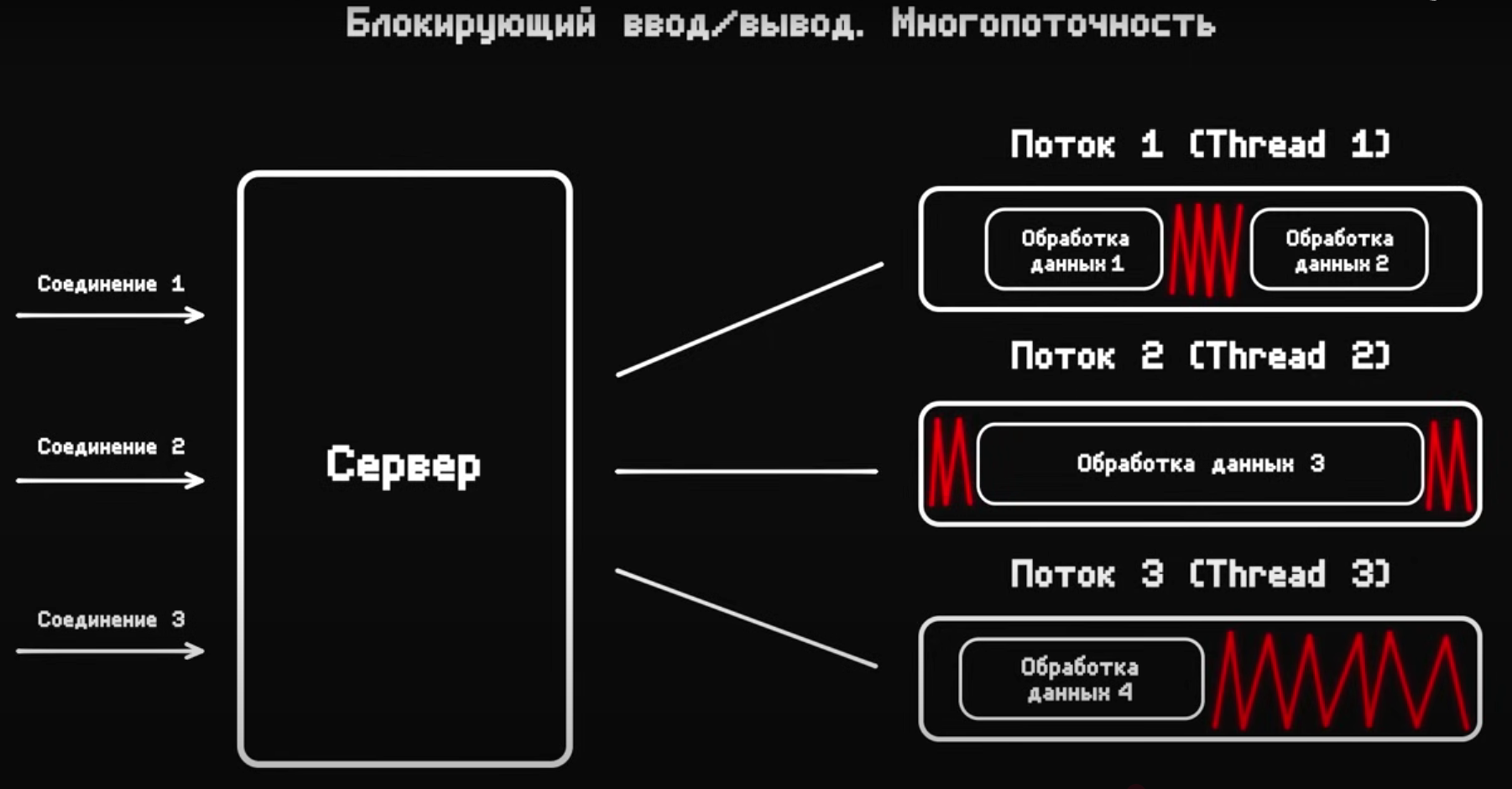
Второй столб, на котором основан Node. Js, это libuv. В связке с V8 они образуют фундамент Node. Js и по сути являются основой. LibuVee отвечает за два принципиально важных момента. Первый — это кросс-платформенный input-output. И второй момент — это event loop.

Кросс-платформенные операции ввода и вывода. К ним относятся работа с файловой системой, работа с сетью. И на разных ОС эта работа происходит по-разному. Но для того, чтобы мы могли установить Node. Js на любой ОС и имели возможность работать, например, с файловой системой, нужна некоторая оболочка, некий программный интерфейс, позволяющий нам это взаимодействие организовать. Этим как раз и занимается LibuVee. То есть мы через JavaScript даем команду Node. Js прочитать какой-то файл или отправь какие-то данные на клиент. А Node. Js, чтобы это сделать, внутри себя использует эту библиотеку Libuvi. Таким образом, вот эта вот кросс-платформенность как раз достигается средствами Libuvi.

Цикл событий. Во многих языках программирования, в таких как C, Java, Python, все инструкции, команды по умолчанию являются блокирующими, то есть разработчик, который не позаботился об асинхронном выполнении кода, заблокирует все приложение. Например, мы пытаемся считать файл с файловой системы или же обработать какой-то сетевой запрос, и так вот пока этот файл не считается полностью из файловой системы, приложение у нас заблокируется и другие инструкции выполняться не будут. Для того, чтобы это предотвратить, используется так называемая многопоточное программирование, управление потоками или трэдами. С помощью цикла событий разработка асинхронных приложений становится гораздо проще. Нам не нужно заморачиваться над созданием физических потоков. Когда Node. Js необходимо выполнить операцию ввода-вывода, опять же, загрузки каких-то данных из сети, доступа к базе данных, к файловой системе. Вместо того, чтобы на время выполнения этой операции заблокировать главный поток, Node.Js просто продолжает заниматься другими делами, как раз до тех пор, пока результаты выполнения этой операции не будут получены. Подобные махинации происходят в цикле событий.

Итак, давайте подведем итоги и выявим преимущества и недостатки цикла событий. Плюсы: возможность обрабатывать большое количество операций ввода-вывода, и если они простые, то скорость достигается максимальная. Таких скоростей нельзя добиться, используя потоки и блокирующую модель поведения. Минусы: получается достаточно много асинхронного кода, и также сложные вычисления создают большую нагрузку. То есть обратиться к базе данных, обработать сетевой запрос — это все будет работать очень быстро. А провести какие-то сложные математические операции в цикле — это уже задача не из простых и она будет достаточно ресурсоемкой.

Начнем с классической модели, если ее так можно назвать, с блокирующего ввода-вывода. Так работают классические веб-сервера, например, на Java. Концепция достаточно простая, есть некоторый поток выполнения, поведения, который строчка за строчкой выполняет какие-то команды, какие-то инструкции. Пока команда у нас не выполнилась, к следующей перейти мы не можем, то есть весь поток блокируется. Если инструкции просты, все хорошо, строчка за строчкой мы их выполняем, никаких трудностей не возникает. Но не все инструкции бывают просты, иногда нам надо записать файл, получить файл, поработать с базой данных или с сетью, как в данном примере. При этом при блокирующей модели поведения весь поток у нас блокируется, то есть поток занят какой-то трудоемкой операции ввода или вывода, например, он считывает файл с диска или же этот файл записывает. При этом приложение не способно обрабатывать другие операции. Для того, чтобы решить эту проблему, подобные операции выполняют в разных потоках. То есть на уровне ОС создаются потоки, которые обрабатывают разные операции. Какие-то потоки обрабатывают HTTP-соединения, какие-то потоки взаимодействуют с базой данных, какие-то потоки взаимодействуют с файловой системой. Такой подход является отличным решением, но, но он имеет определенные минусы. Первый и один из самых важных минусов, что поток потребляет большое количество ресурсов. При этом зачастую возникают ситуации, когда поток создается, но какую-то часть времени он просто простаивает. Но ресурсы для этого потока ОС уже были выделены. И второй важный недостаток — это сложность управления этими потоками. То есть программист, который пишет код и пытается с ними взаимодействовать, должен обладать хорошей квалификацией. Потому что потоки могут обращаться к одним и тем же участком памяти, возникают так называемые дэдлоки, т.е. взаимная блокировка.

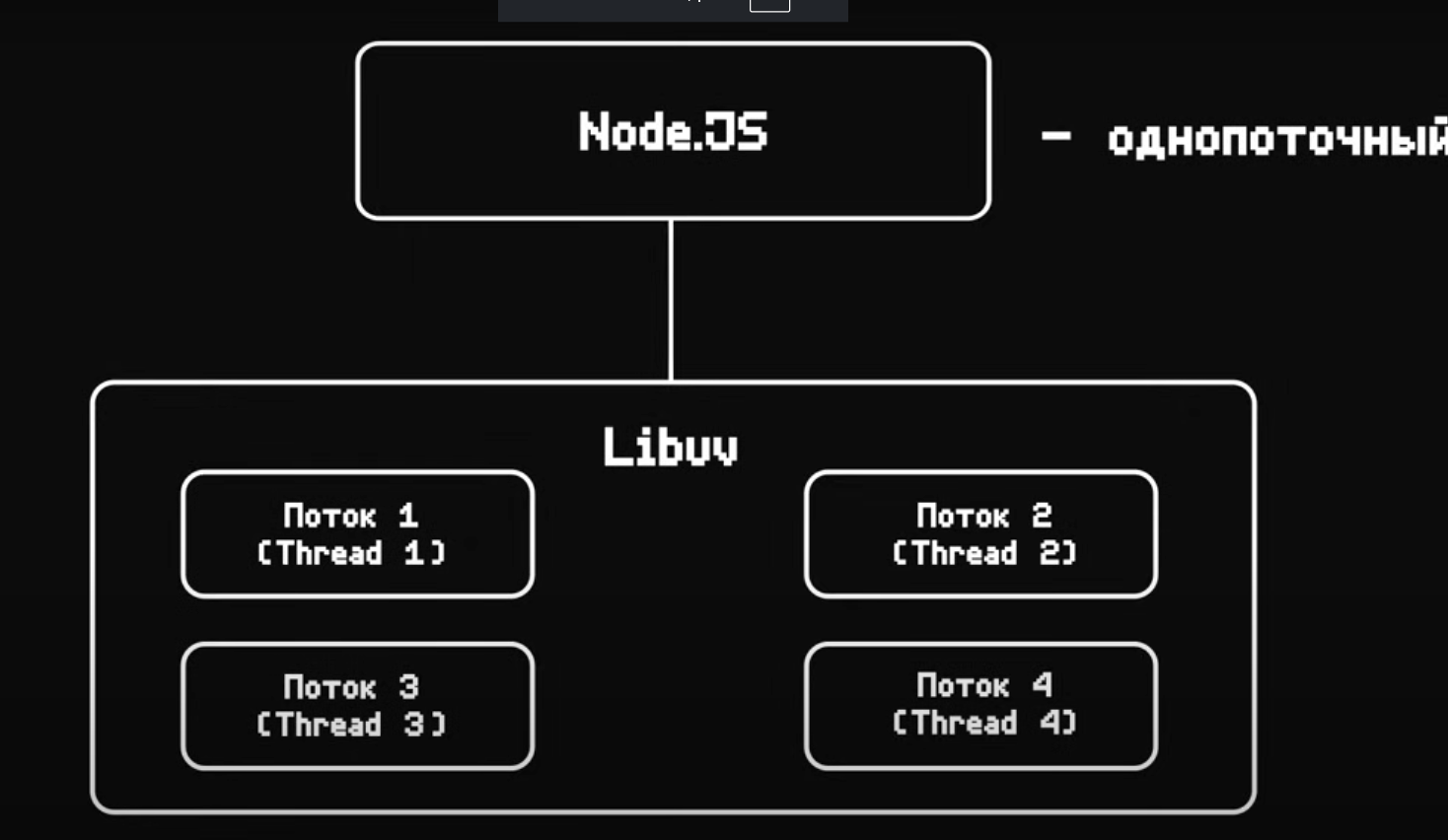


Есть некоторый веб-сервер, который прослушивает некоторые запросы. С веб-сервером устанавливается первое соединение и для него создается отдельный поток. Внутри потока выполняется какая-то операция по обработке данных и затем спустя какое-то время выполняется вторая операция. Параллельно с первым соединением устанавливается второе, для него также создается поток с какой-то обработкой данных. И также параллельно устанавливается третье соединение, для него выделяется новый поток. Итак, как видно на схеме, потоки некоторое время находятся в состоянии простоя, ожидая новых данных, получаемых из связанных с ним соединений. При этом потоки потребляют значительное количество ресурсов. Они расходуют память и вызывают переключение контекста, поэтому иметь достаточно долго выполняющийся поток для каждого подключения и при этом не использовать его длительную часть времени — это не лучшее решение с точки зрения эффективности.

Ярчайший пример для сравнения блокирующей и не блокирующей модели поведения — это два веб-сервера Apache и Nginx. Apache на каждое входное соединение создавал отдельный поток и работал по блокирующей схеме. К Nginx мы вернемся чуть позже, когда обсудим неблокирующую модель поведения.



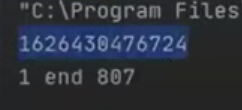
Теперь поговорим про неблокирующий ввод и вывод. Опять же, есть некоторый веб-сервер, работающий по этой модели, и извне установлено несколько сетевых подключений. И сервер работает только с одним потоком, с так называемым главным потоком или же мейн-тред. При использовании неблокирующего ввода и вывода, системные вызовы немедленно возвращают управление, при этом не ожидая выполнения чтения или записи каких-либо данных. Подобный механизм доступа к ресурсам поддерживает большинство ОС и основным шаблоном реализации неблокирующего ввода и вывода является активный опрос ресурсов в цикле. Поскольку мало создать операцию и забыть про нее, нам необходимо получить еще и результат ее работы. Подобный шаблон, подобный паттерн называется «цикл ожидания». Это простейший паттерн, простейший шаблон, который определенно не является идеальным способом, который предназначен для неблокирующей работы с ресурсами. Но в большинстве ОС есть более эффективный механизм для неблокирующей работы и он называется demultiplexer событий.



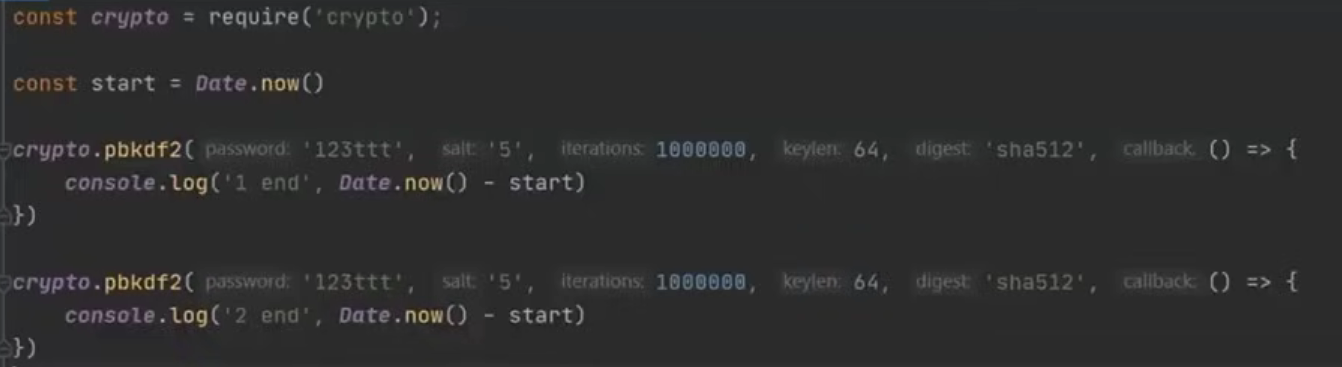
Apache работал по блокирующей модели поведения, где на каждое соединение открывался новый поток. В свою очередь Nginx работал по неблокирующему принципу ввода и вывода. При этом на графике вы можете заметить, что при росте одновременных подключений количество запросов, которые мог обрабатывать Apache, значительно ниже, чем количество запросов, который обрабатывал Nginx. Таким образом, Nginx работал примерно в 2,5 раза быстрее, чем Apache. Nginx использовал неблокирующую модель ввода-вывода, а Apache был классическим многопоточным веб-сервером. Итак, на данный момент мы разобрались с тем, что представляет из себя неблокирующий ввод-вывод, и поняли то, что Node. Js работает именно по этому принципу. Исходя из этой схемы, может возникнуть вполне резонный вопрос. Получается, что у нас есть один главный поток, и получается, что Node. Js однопоточный. Это и так, и не так одновременно. Сам по себе JavaScript является однопоточным и вся асинхронность достигается за счет так называемого event loop. Что же касается Node. Js? Сам по себе Node. Js однопоточный. То есть разработчики, которые создают приложения на Node. Js, пишут асинхронный код без использования потоков. Но при этом чуть ранее я упоминал, что в основе Node. Js лежит libuv, который занимается как раз операциями ввода и вывода. В своей основе Libuv может управлять потоками, причем дефолтное их количество равняется 4, это количество можно изменять. Из минусов неблокирующего ввода и вывода можно выделить то, что операции, подобные чтению файла или записи его на диск, являются очень тяжелыми, поскольку выполняются в одном потоке, хоть и с использованием неблокирующего ввода и вывода. Поэтому некое распараллеливание этих процессов все же необходимо.

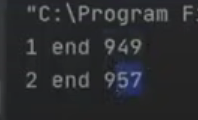
Итак, Node. Js однопоточный, но при этом LibuVee использует потоки. LibuVee написан на C, а движок v8, на котором основан Node. Js, написан на C++. И как вы, наверное, уже могли догадаться, и на C и на C++ можно писать какие-то модули для Node. Js. И это говорит нам о том, что некоторые библиотеки могут использовать потоки.





Итак, импортируем стандартный модуль Node. Js Crypto, который предназначен для криптографических операций, для шифрования, дешифрования и хеширования соответственно. Вызываем функцию, которая шифрует какой-то пароль, некоторый ключ за определенное количество итераций. Первым параметром укажем любой строковый пароль, затем укажем соль, которая необходима для хеширования, затем укажем количество итераций, сделаем их побольше, чтобы этот процесс занимал какое-то время и мы наглядно могли увидеть. Затем указываем длину ключа и алгоритм шифрования. Последним аргументом мы передали callback. Он отработает именно в тот момент, когда функция хеширования будет закончена. Эта функция как раз является асинхронной, именно не блокирующей, то есть поток при ее вызове заблокирован не будет. И давайте теперь проверим, сколько эта функция по времени будет выполняться. Из текущей даты отнимем дату, когда скрипт у нас запустился, то есть создадим переменную и туда поместим текущее время, а затем из текущего времени вот это стартовое время мы просто вычтем. Итак, запускаем. Вот это время в формате timestamp в миллисекундах, это стартовое время. А сама функция хеширования отработала за 800 миллисекунд, плюс-минус.





Теперь давайте скопируем вызов этой функции и вызовем ее еще раз. Единицу в логах поменяем на 2, чтобы видеть разницу, и запустим еще раз. Как видите, и первая, и вторая функция результат вернули примерно в одно и то же время, 950 +- миллисекунд. То есть они вызвались как-то параллельно. Неважно как, но параллельно. Теперь продублируем вызов функции еще несколько раз и запустим. Немного подождем и видим, что 4 раза вызвалась функция и при этом время выполнения примерно 1200. Опять же все логично, все выполняется параллельно, но попробуем добавить еще пятый вызов и посмотрим, как отреагирует программа. Как видите, первые четыре отработали +- в одно время, но пятый запуск выбился и отработал гораздо позже. Давайте запустим еще раз. И ситуация здесь примерно такая же. Первые четыре запуска закончили примерно в одно время, а пятый закончил выполнение на 500 миллисекунд позже. Почему же так происходит? Просмотрим еще одну схему.



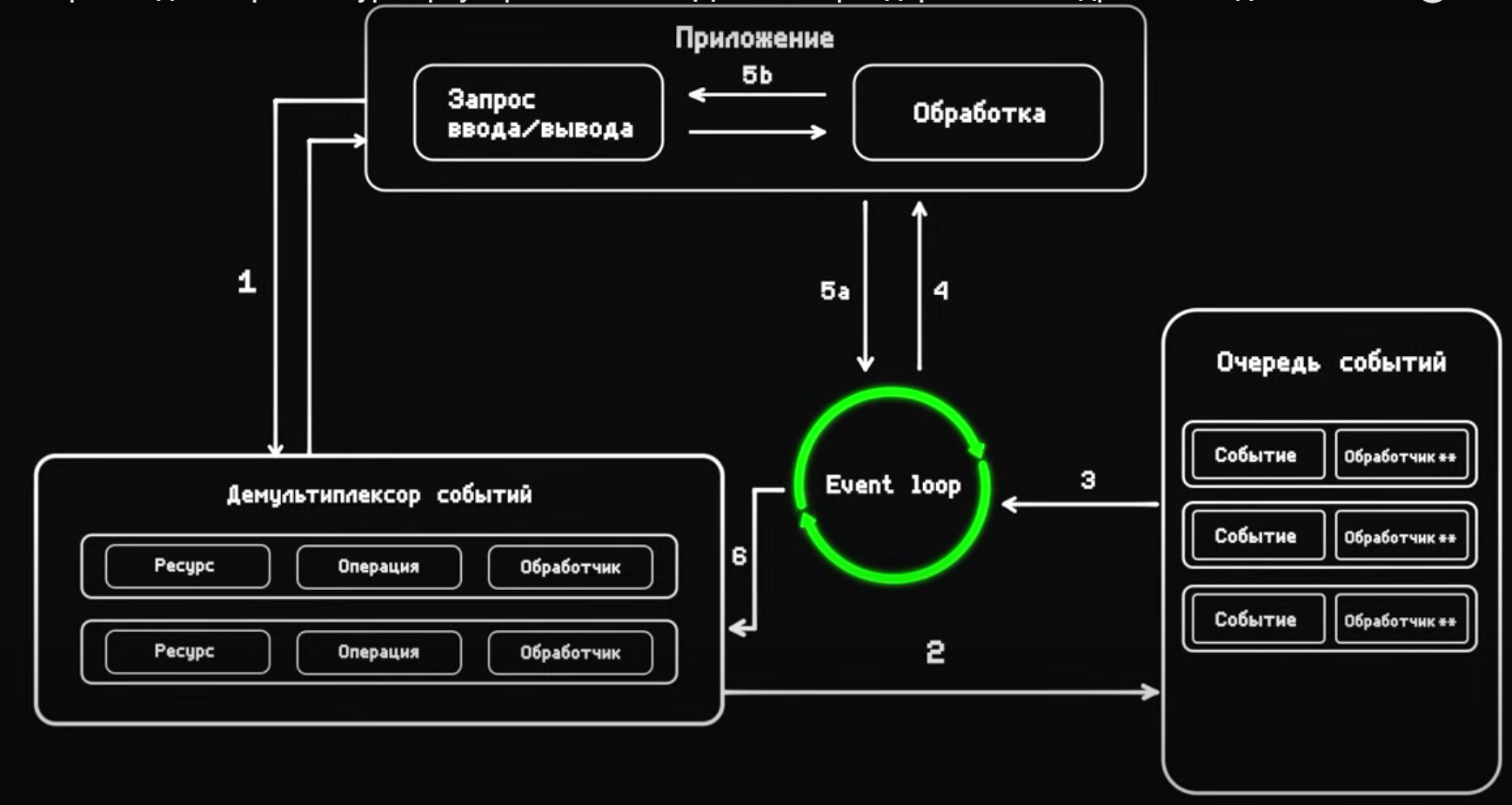
Есть некоторый планировщик потоков. Планировщик — это часть ОС, которая отвечает за параллельное выполнение задач, потоков, процессов. И вот как раз планировщик выделяет этим потоком некоторое процессорное время, память, стэк и прочие ресурсы. Также есть некоторый пул потоков, и в нем находятся в данном случае 4 потока. Мы 5 раз пытаемся запустить криптографическую функцию, и давайте посмотрим, как потоки будут выполнять эти функции. Первый поток забирает запуск первой криптографической функции, второй, второй, третий, третий и четвертый, четвертый. По умолчанию 4 потока, поэтому каждый из этих потоков взял на себя выполнение одной из криптографических функций. Пятое при этом осталось ожидать. Четыре функции выполнились примерно в одно время, при этом все потоки освободились. И тот из потоков, который освободился раньше, забирает на себя пятую функцию. Именно этим поведением объясняется то, что пятая функция была выполнена примерно на 500−600 миллисекунд позже, чем предыдущие четыре.

С версии 11.7.0 в Node.Js был введен модуль, который называется worker\_threads и с помощью него можно управлять потоками. Современные ОС предоставляют некоторый механизм, который называется демультиплексор событий. Именно благодаря демультиплексору событий неблокирующий ввод и вывод становится доступен. Рассмотрим, как работает демультиплексор событий, а также шаблон-реактор, на котором устроен Node. Js.

Начнем с демультиплексора событий. Он представляет из себя некоторый интерфейс уведомления о событиях. Его задача осуществляется в сборке и постановке в очередь событий ввода и вывода, которые поступают из набора наблюдаемых ресурсов, а также блокировка появления новых, доступных для обработки событий. Следующая не менее важная запчасть всего механизма работы шаблона реактор — это Event Loop. Есть некоторая очередь событий и есть некоторый бесконечный цикл, тот самый Event Loop, который синхронно выполняет задачи из этой очереди и распределяет их дальше. Очередь содержит в себе некоторые события, например было отправлено какое-то сообщение, например закончилось время у таймера и для каждого события устанавливается некий обработчик. В Node. Js он представлен функцией обратного вызова, так называемым callback.

Итак, есть demultiplexer событий, некоторый бесконечный цикл и очередь событий. Логично, что должно быть какое-то приложение, которое будет ожидать, запрос ввода и вывода, в нашем случае это приложение на Node. Js, запросом ввода и вывода может быть, опять же, любой сетевой запрос, запрос на чтение или запись файла, обращение к базе данных и так далее. И после того, как операция ввода и вывода была закончена, ее необходимо обработать, этим также занимается приложение. Итак, на данном этапе у нас есть Demultiplexer событий, который представляет из себя некоторый интерфейс уведомления о событиях, некоторый бесконечный цикл EventLoop и очередь, которая содержит в себе события и их обработчики. Также есть приложение, которое ожидает запрос ввода и вывода и обрабатывает как раз эти события.

Теперь рассмотрим последовательность взаимодействия этих компонентов в шаблоне реактор.



1: приложение создает новую операцию ввода и вывода, передав запрос демультиплексору событий. Также приложение должно определить обработчик (функция обратного вызова). Обработчик будет вызван тогда, когда операция будет завершена. Самый важный момент, что отправка нового запроса для демультиплексора событий не приводит к блокировке приложения. Управление немедленно возвращается обратно к приложению.

2.После завершения обработки набора операций ввода вывода, демультиплексор событий добавляет эти новые события в очередь.

3.Цикл событий выполняет обход элементов в очереди событий, после чего для каждого события вызывается соответствующий обработчик, то есть функция обратного вызова, которая была указана для того или иного события.

5. В случае 5а обработчик, который является частью кода приложения возвращает управление циклу событий, то есть какое-то событие мы обработали и опять вернули управление циклу событий EventLoop, но во время выполнения этого обработчика вполне могут запрашиваться какие-то новые асинхронные операции, например, мы прочитали какую-то информацию из базы данных и теперь хотим записать ее в файл, это приводит к добавлению новых операций в demultiplexer событий и вся вот эта схема повторяется по новой. После того, как EventLoop обработал все элементы из очереди, цикл вновь заблокируется демультиплексором событий и вся эта процедура начнется по новой, когда появится новый запрос на операцию ввода и вывода.



Каждая ОС имеет свой интерфейс для демультиплексора событий.

Поскольку в каждой ОС интерфейсы разные, и например в Unix обычные файлы не поддерживают неблокирующих операций, поэтому для имитации неблокирующей модели поведения необходимо использовать отдельный поток вне цикла событий, т.е. нужна некоторая абстракция, которая для всех ОС будет делать неблокирующую модель поведения одинаковой. Именно по этой причине разработчики ядра платформы Node. Js разработали библиотеку на С LibUV. За счет нее как раз и обеспечивается совместимость Node. Js со всеми основными платформами. Библиотека LibUV реализует шаблон реактор, о котором мы говорили несколько минут назад, и обеспечивает программный интерфейс для создания циклов событий, управления очередью событий, выполнения асинхронных операций ввода-вывода и организации очереди задач разных типов.

Когда Node. Js запускается, она инициализирует Event Loop. Node. Js обрабатывает предоставленный на вход код, который может выполнять вызовы асинхронного API, настраивать различные таймеры и т.д. После чего начинается обработка цикла событий. Порядок выполнения операций в цикле событий:



Каждый из представленных сегментов на слайде представляет из себя некоторую фазу цикла событий. Каждая из этих фаз имеет очередь коллбеков для выполнения. Когда цикл событий входит в фазу, он будет выполнять любые операции, относящиеся к этой фазе. Затем он будет выполнять коллбеки из очереди, пока эта очередь не будет исчерпана. После того, как он выполнит все коллбеки, он переходит на следующую фазу и проделывает подобную процедуру.

1 - таймеры. В этой фазе выполняются каллбеки, которые запланированы двумя функциями — setTimeout и setInterval.

2 - каллбеки ввода и вывода. Здесь выполняются почти все каллбеки, за исключением событий close, таймеров и событий, которые были определены с помощью функции setImmediate в Node. Js.

3 - ожидание и подготовка, и она используется только для внутренних целей.

4 - опрос. В ней происходит получение новых событий ввода и вывода. При этом Node. Js может блокироваться на этом этапе.

5 - проверка. Вызываются каллбеки, которые были определены с помощью функции setImmediate.

6 - все каллбеки события close. Например, закрытие WebSocket соединения, то есть события close. Закрытие, например, стрима, считывающего какие-то данные.

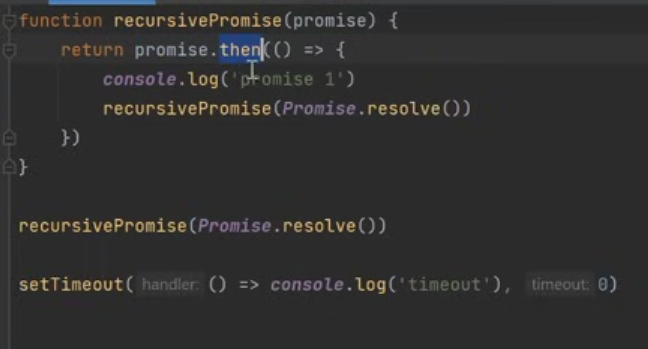
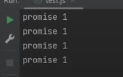
И вот так вот цикл проходит все фазы, выполняет колбеки из очереди и запускается по новой.

То есть, если на текущем этапе сравнить Node. Js Event Loop с браузерным, то уже можно заметить, что в браузерном у нас было всего две очереди, это очередь микро- и макрозадач.

То здесь у нас появляется 6 очередей, которые последовательно выполняются. Таймеры, колбеки, ожидания, опрос, проверка, колбеки клосс. То есть, задачи из этих очередей выполняются одна за одной. Здесь нет никаких микротасок, макротасок, сам Event Loop этого не подразумевает. Здесь просто есть очереди, которые внутри себя содержат колбеки определенного вида.

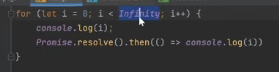
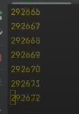
Но на самом деле, если быть более справедливым, очередей по факту будет семь, потому что почему-то вот в документации, когда вот этот Event Loop рисуют, здесь не подразумевают промисы. Промисы в любом случае будут выполняться перед таймерами. Промисы в любом случае всегда будут иметь больше приоритет, чем какие-то другие колбеки, таймеры…

Совсем недавно в собеседовании меня спросили про реальный кейс, который случился у людей в продакшене. У них работала нода и в каком-то из кейсов случалось так, что промис зацикливался сам на себя. Это происходило не явно, где-то там обрабатывался запрос, было какое-то чтение, например, из базы данных и получалось так, что промис зацикливался. Из-за этого, естественно, приложение крашилось, долго не могли понять почему, потому что это не всегда бывает очевидно.

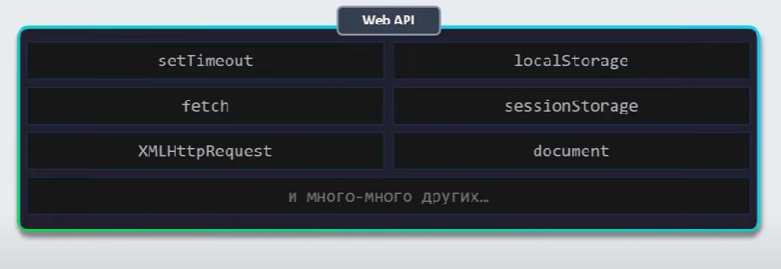
Есть вот такая вот функция рекурсивная, аргументом она принимает promise, возвращает она вот такую вот конструкцию promise.then(). Внутри мы в логе выводим просто Promise 1 и опять вызываем рекурсивно эту функцию и внутрь передаем Promise.resolve().

Ну, первый раз мы эту функцию вызываем вот здесь ниже. Ну, и, соответственно, вот так вот оно в цикле все будет крутиться. Promise.resolve() возвращает Promise. И чуть ниже есть SetTimeout, который никогда в жизни у нас не вызовется за счет того, что очередь микротасок, ну в нашем случае это event loop, там другие очереди, она всегда будет переполняться. И обратите внимание, что вот так вот рекурсивно зациклившись на Promise у нас вот этот бесконечный поток, который черпает из одной очереди эти Promise, крутится бесконечно. При этом setTimeout у нас вообще никогда вызван не будет.

Похожая ситуация здесь. Здесь у нас бесконечный цикл, обычный for, в нем мы внутри выводим логи, и вот здесь у нас есть promise resolve, в котором мы просто в логи тоже выводим текущий элемент итерации. И в таком случае у нас Promise также выполнен никогда не будет. У нас бесконечно будет инкрементироваться цикл, это синхронные задачи, они попадают в call stack, и очередь Promise никогда выполнена не будет.

Вот всякие вот циклы, петли, которые могут образовываться, которые не позволят выполнить задачи из соседних очередей, которые должны идти в очереди как бы следующими по выполнению.



Части веб браузера