Modern Web Browsers and JS Engine

Rujuan Xing

Maharishi International University - Fairfield, Iowa

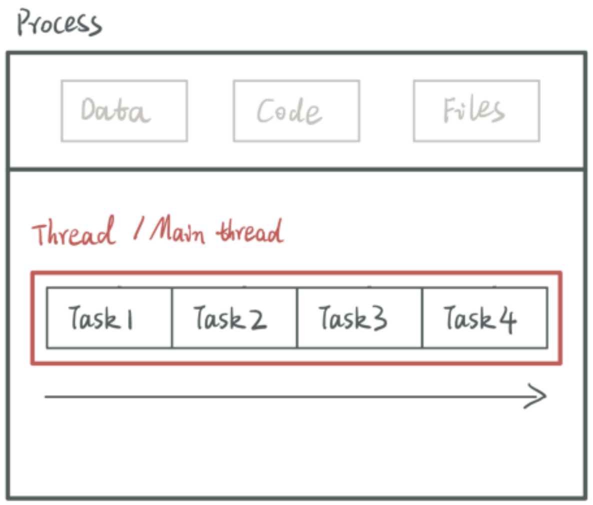
Modern Web Browser

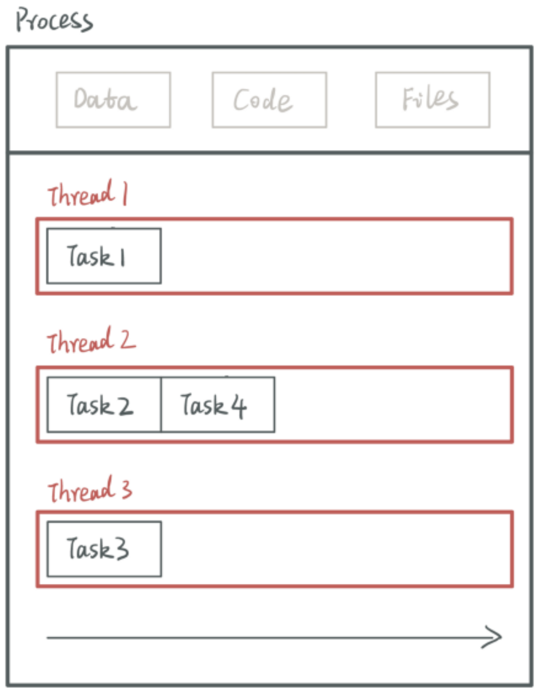
Processes and Threads

- A browser was a black box. It eats my HTML, CSS, and JavaScript files, and displays a page.
- The essential part of browsers are all based on processes and threads.
- Threads are living in a process, along with data, codes, and files.
- A thread is a queue, processing tasks one by one. The next task has to wait until the earlier ones finish.

- Single-threaded process
  - Only one thread
  - The main thread of the process.

- Multi-threaded process
  - The tasks can process faster in multiple threads simultaneously
  - All threads share the data in the memory.

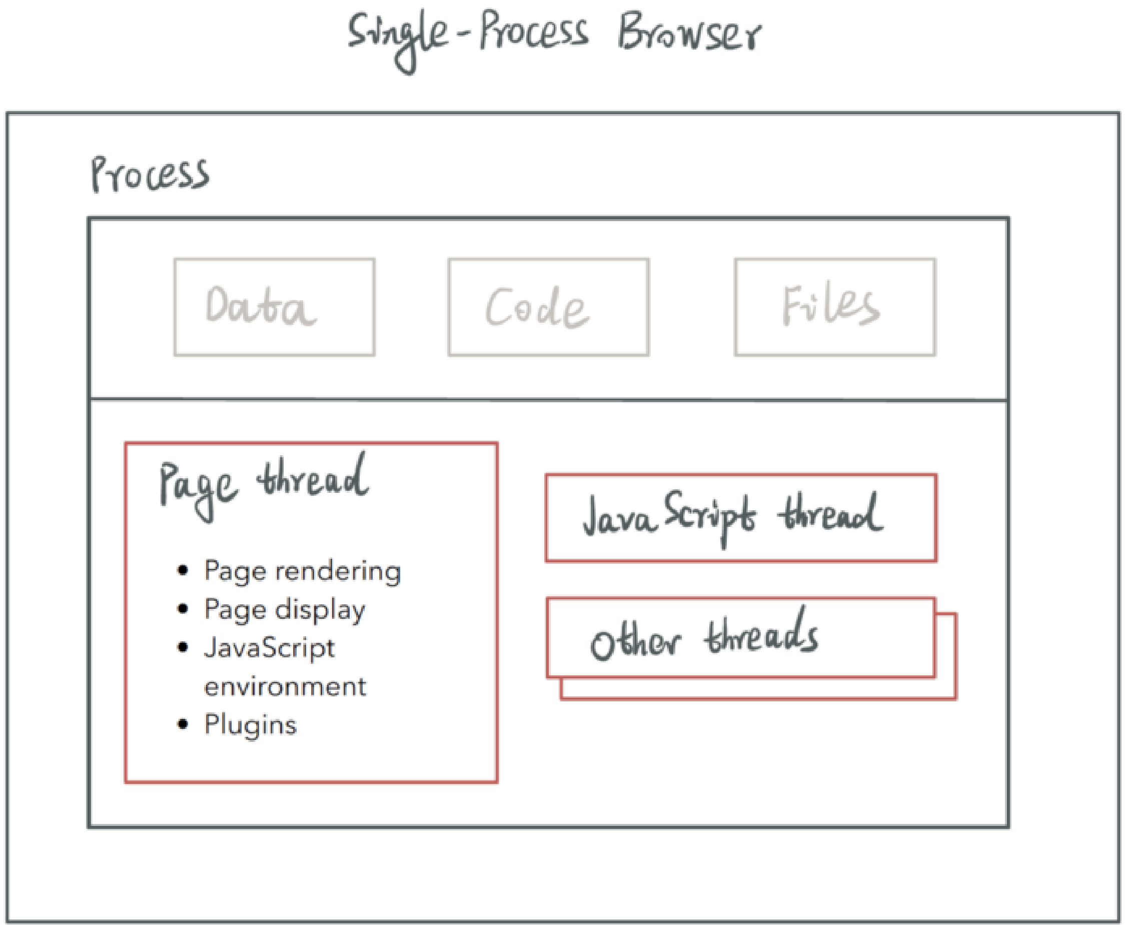




Browser in early age - single-process browser

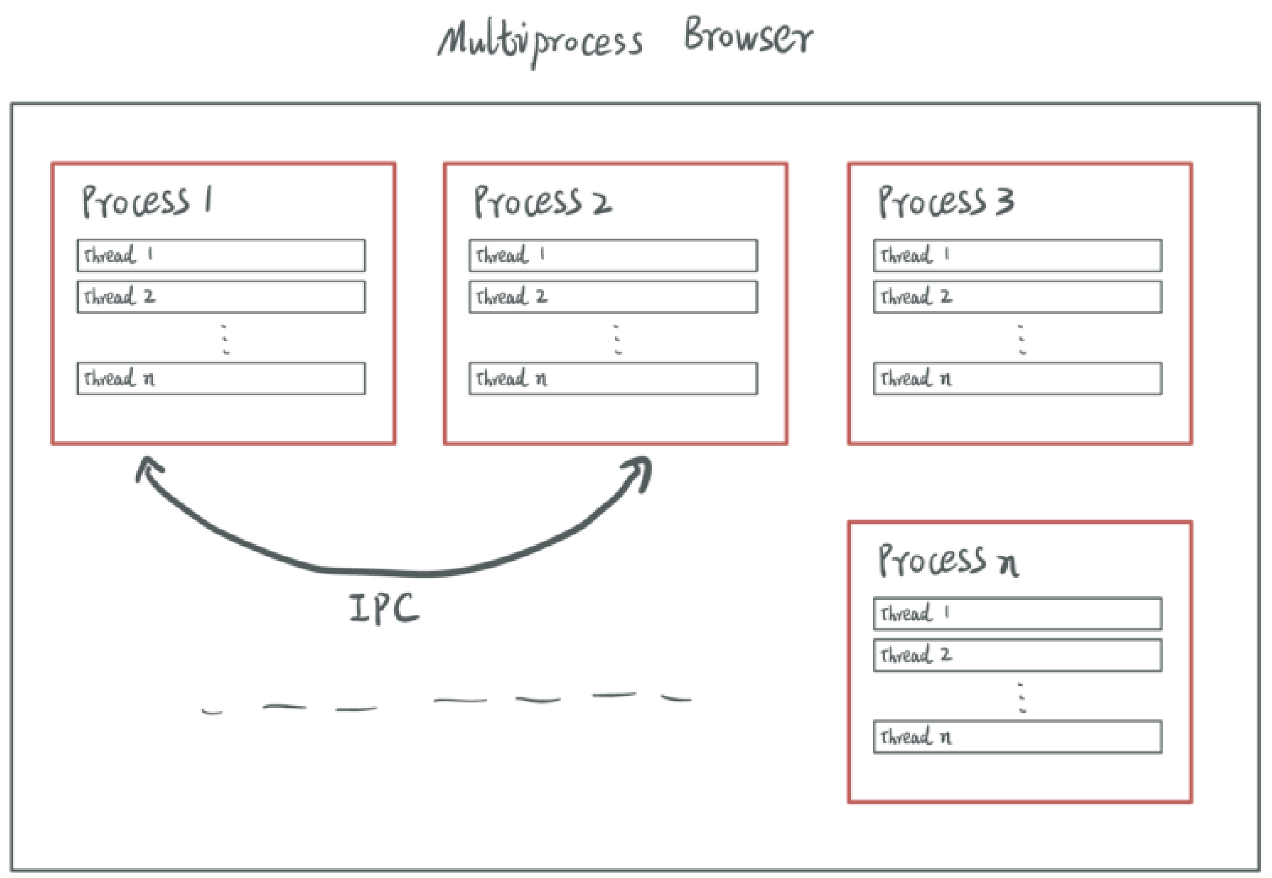
- No tabs
- Each web page was allocated with a window
- The process needs to take care of everything, including page rendering, JavaScript execution, and more.

- **Instability**
  - When a thread crashes, the process stops, and the whole browser freezes.
- **Low performance**
  - Some JavaScript takes a while to finish.
- **Insecurity**
  - Evil plugins can easily gain access to your operating system with admin permission through the single-process browser.



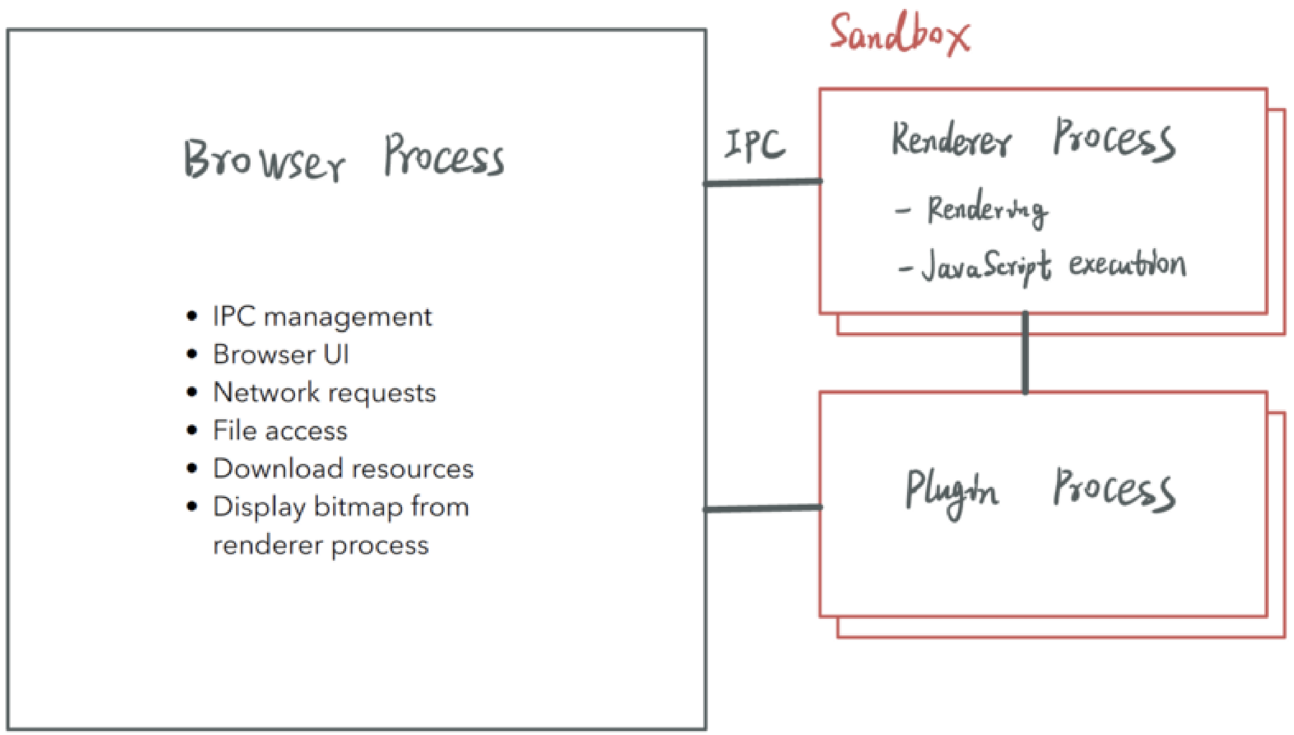
Modern Web Browser - Multiprocess browser

- Multiple processes
- Each process runs various threads
- Between processes, they communicate with Inter-process communication (IPC)



Multiprocess in Modern Web Browser

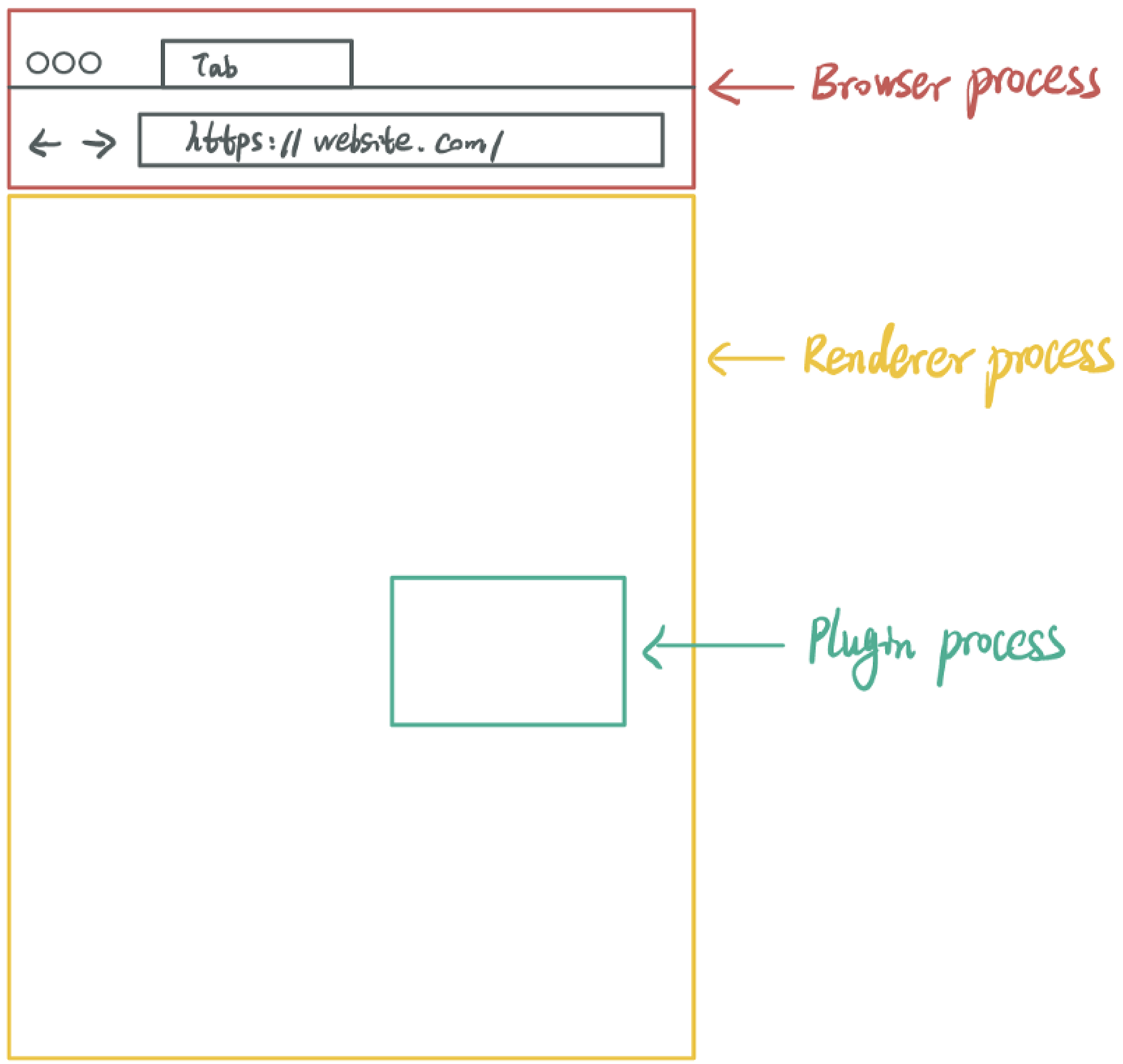
- Many processes running in browser:
    - Browser process
    - Renderer process
    - Plugin process
    - GPU process
    - Network process
    - Device process
    - Video process (one of the utility processes)
    - Audio process (one of the utility processes)



- IPC management
- Browser UI
- Network requests
- File access
- Download resources
- Display bitmap from renderer process

Multiprocess Architecture

- Two significant changes
- Separating the renderer and plugin process from the browser process
- Placing renderer and plugin processes in sandboxes

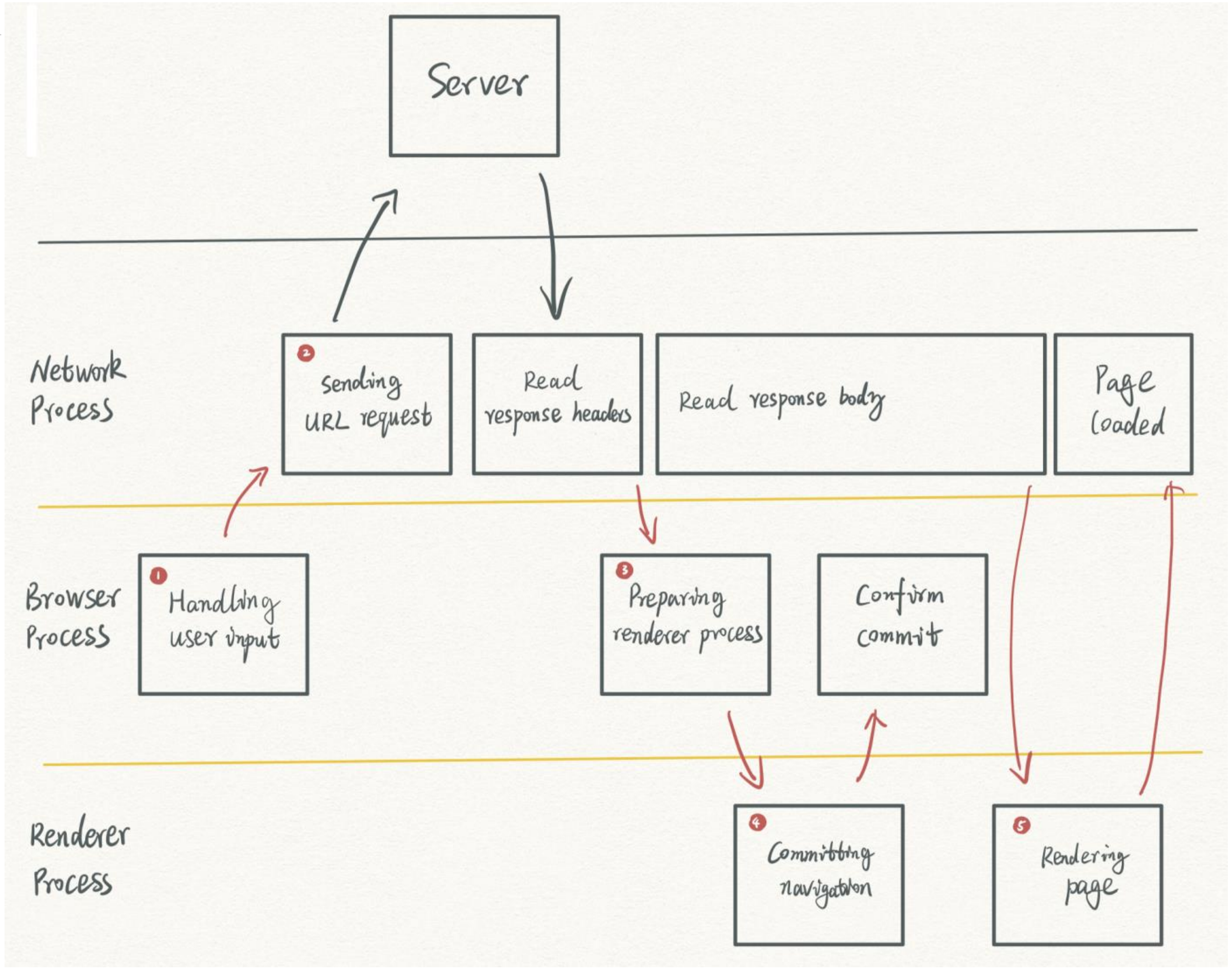


Changes solve the problem in Single Process Browser

- **Fight against the instability**
    - The renderer process takes care of the page rendering and JavaScript execution. When it crashes, only the corresponding browser tab stops. You can still browse other tabs because the browser process remains intact.
    - Each plugin has a separate process. For example, if the Flash plugin crashes, it doesn’t affect your browsing experience anymore.
- **Improve the low performance**
    - Each tab has a separate renderer process. When we open multiple tabs, the browser creates multiple renderer processes.
    - If one of the pages has a JavaScript issue blocking its process, the problem does block page rendering in other tabs. Multiple plugin processes help in the same way.
- **Security with sandboxing**
    - All high-risk components are moved to the renderer process and running in a sandbox, including HTML parsing, JavaScript virtual machine, and Document Object Model (DOM).
    - System functionality is placed in the browser process, including persistence storage, user interaction, and networking.
    - Vicious codes in renderer and plugin processes are difficult to touch your operating system.
- **Site isolation**
    - all cross-site iframes get a stand-along renderer process.

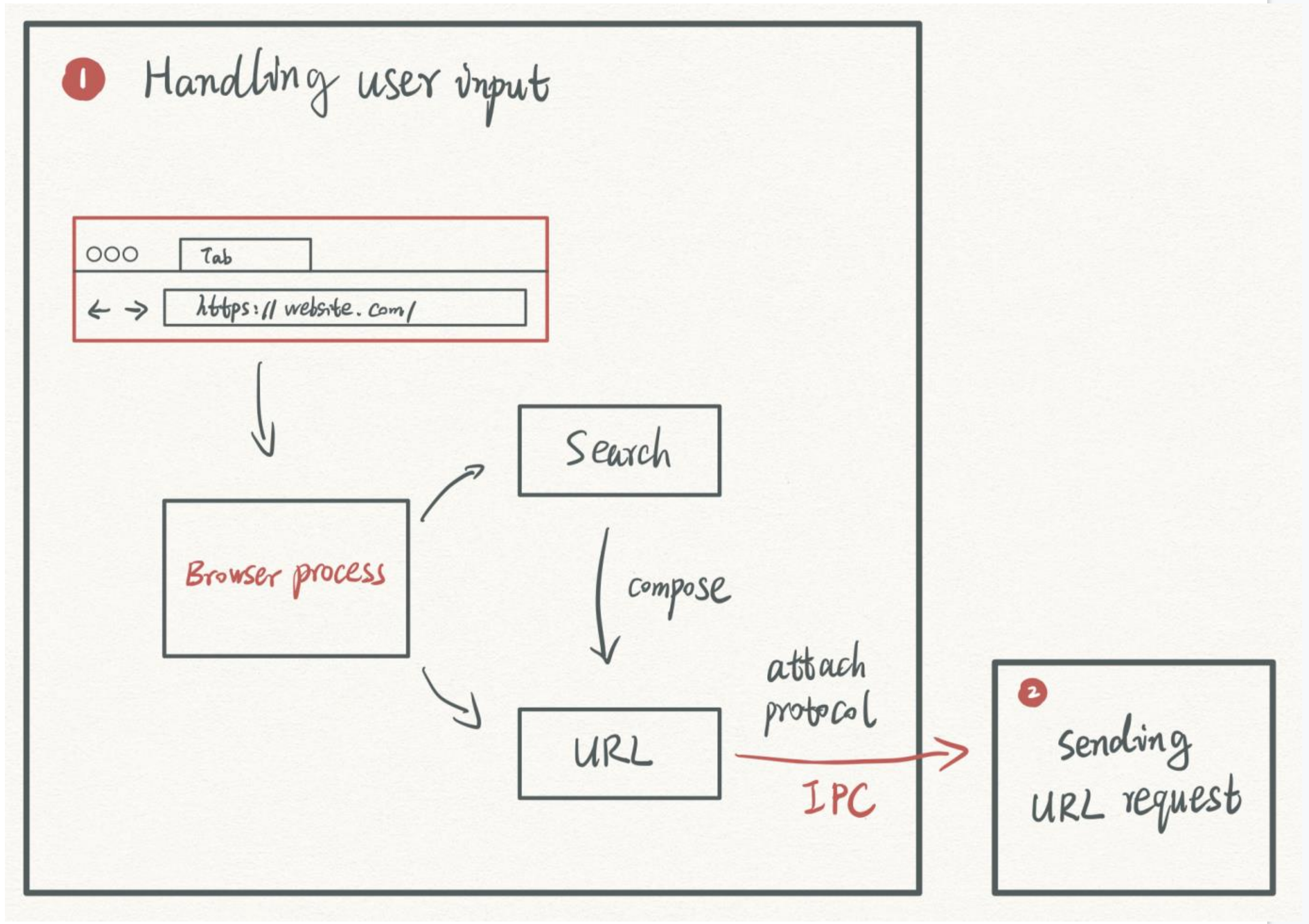
Navigation Phase

- **A high-level view of 5 steps**
  1. Handling user input
  2. Sending a URL request
  3. Preparing a renderer process
  4. Committing navigation
  5. Rendering page



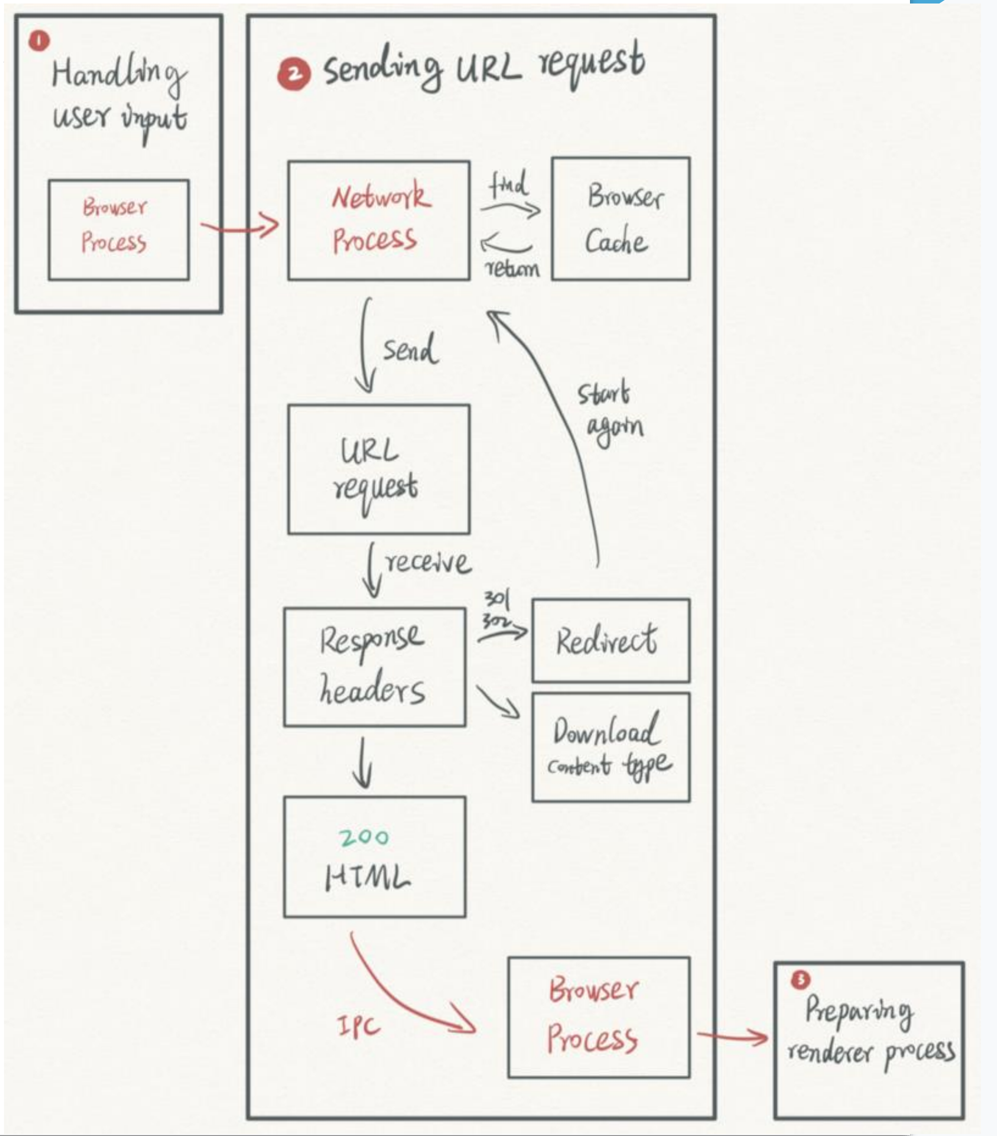
Step 1: Handling user input

- Before navigation starts, a browser process (UI thread) parses the user input.
- Two possibilities exist when a user enters something in the address bar:
    - A search query
    - A URL



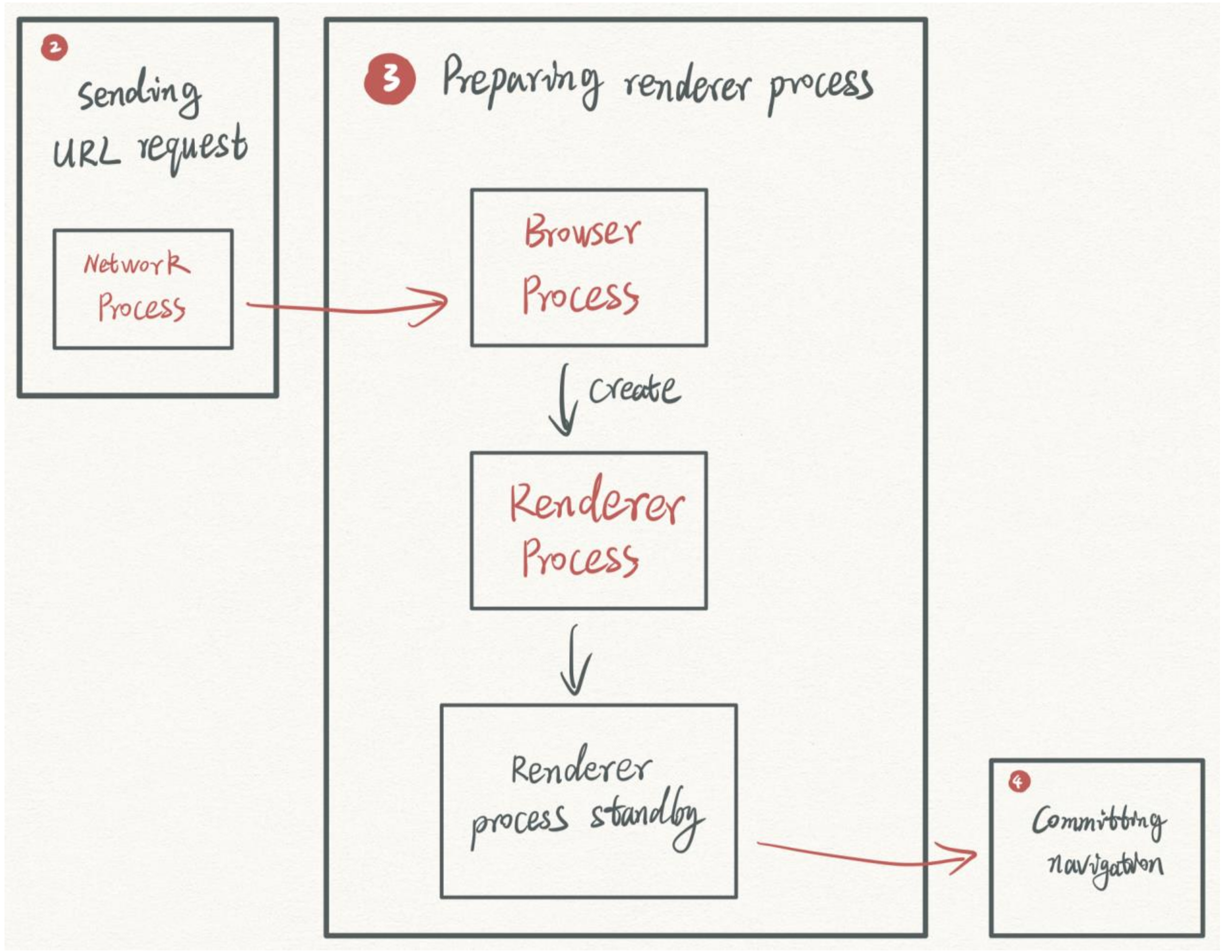
Step 2: Sending a URL request

- A network process receives the command from the browser process to handle the URL request.
- The server sends its response to the network process. The network process starts parsing the response headers.
  - **Safe Browsing and CORB**
    - Browser checks the domain and response data, trying to match them with its malicious site database.
    - a Cross Origin Read Blocking (CORB) check happens to prevent side-channel attacks.
  - **Case 1 — Redirection**
  - **Case 2 — Content-Type is not text/html**
    - If the Content-Type is application/octet-stream, it says the data is byte stream. Usually, the browser treats it as a download request, submits it to its download manager. The navigation ends.
  - **Case 3 — Status code is 200 and the Content-Type is text/html**



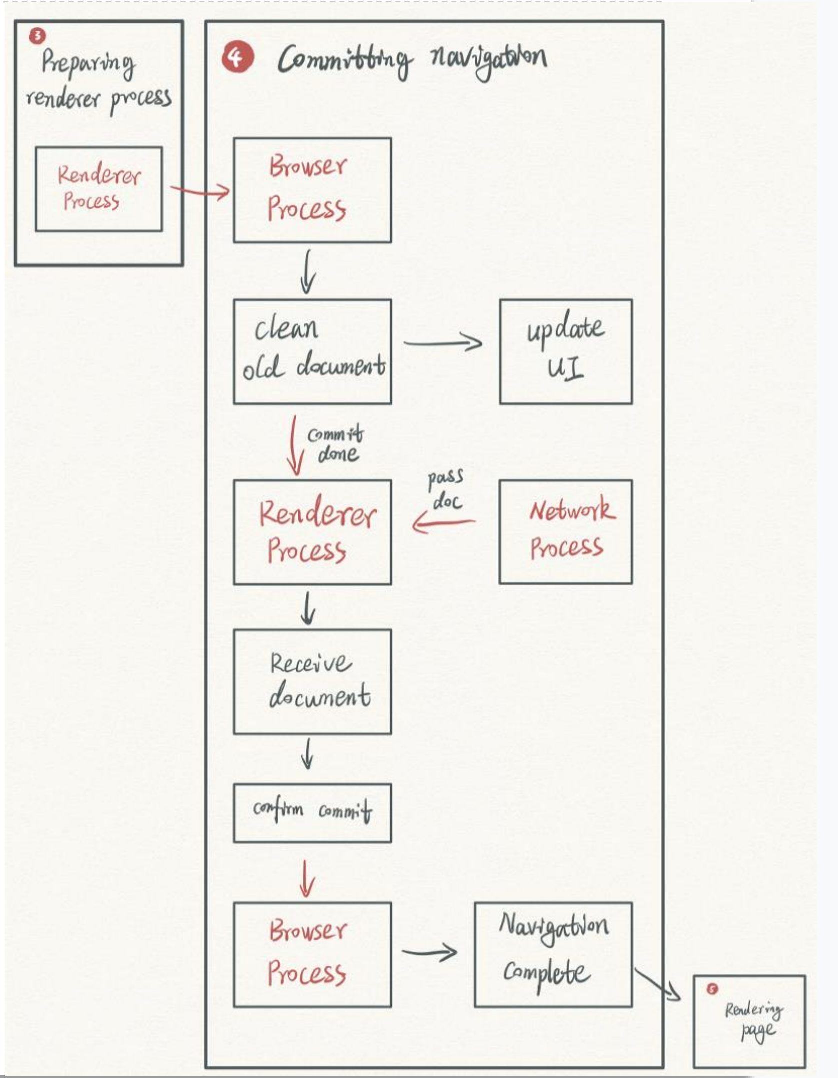
Step 3: Preparing a renderer process

- The browser process creates a renderer process, waiting for the ready signal from the renderer process.
- Meanwhile, the network process is continuously downloading the document for the renderer process.
- When the renderer process is in the standby position, the next step starts.



Step 4: Committing navigation

- The browser process and the renderer process starts working together to replacing the old document with the new one.
- Detailed Steps:
  1. When the render process is ready, it sends a "committing navigation" message to the browser process. The message notifies the browser process that the renderer process is prepared to parse document data.
  2. After receiving the message, the browser process starts clearing the old document.
  3. Then, a confirmation is sent from the browser process to the renderer process, letting it know that the commit has been completed.
  4. Meanwhile, the browser process updates its UI status.
  5. The renderer process receives confirmation and starts receiving the document from the network process.
  6. The renderer process confirms with the browser process that the document is committed.
  7. The navigation completes.



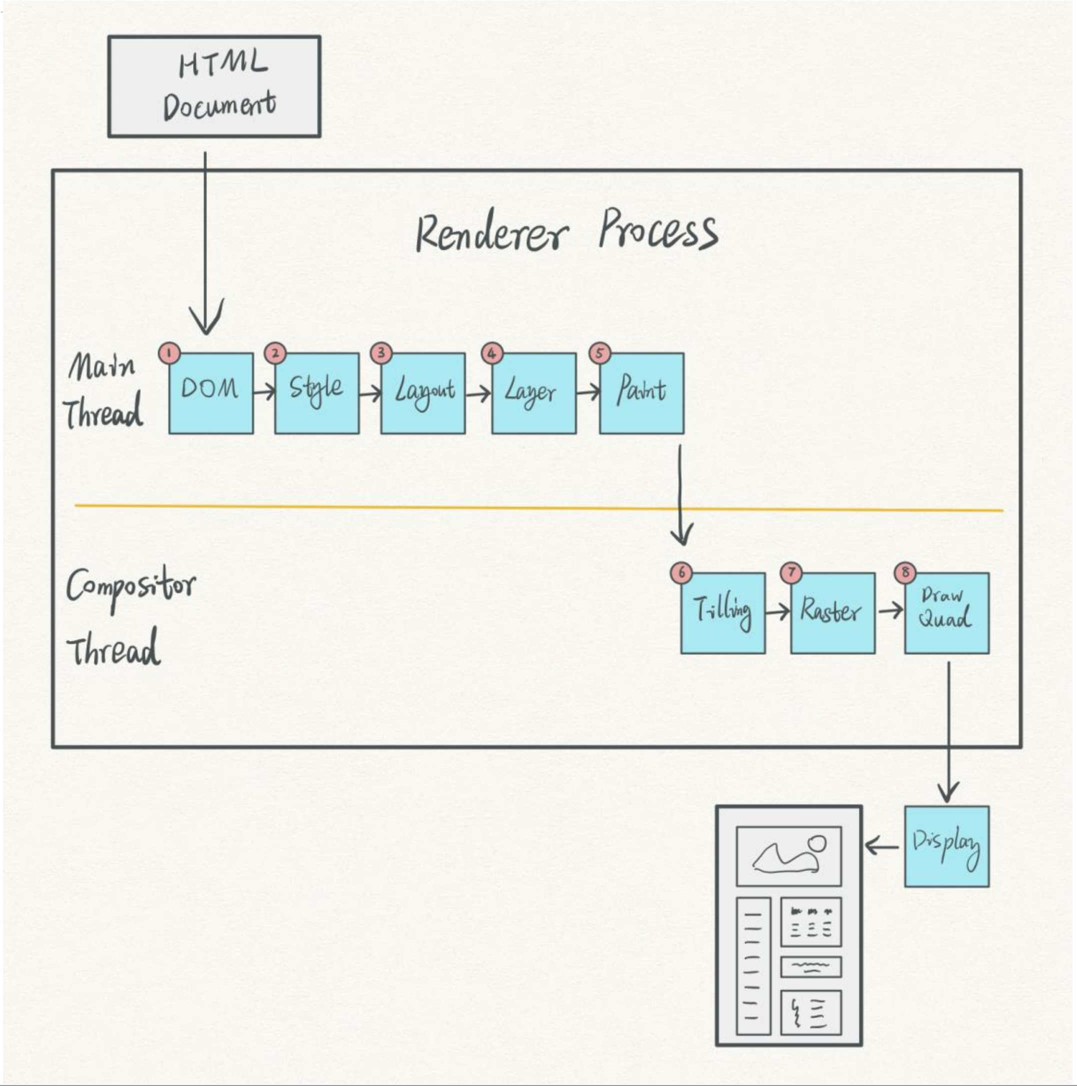
Step 5: Rendering page

- The renderer process receives confirmation. It carries on parsing the document data streamed from the network process and displays a page in front of you.
- At the end of the rendering, the renderer process tells the browser process that the rendering is done.
- The UI thread replaces the loading spinner in its tab with the favicon of the website. A `unload` event is fired.

Rendering Phase - From document to web page

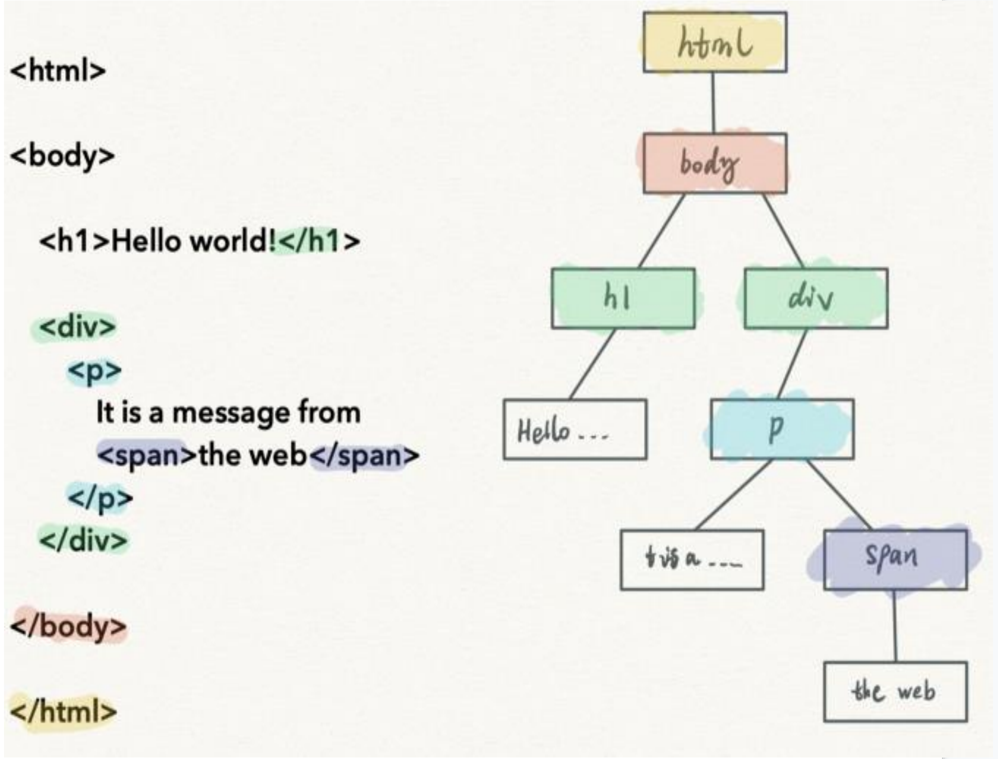
- 8 sub-phases:
    1. DOM construction
    2. Style computation
    3. Layout
    4. Layer
    5. Paint
    6. Tilling
    7. Raster
    8. Draw Quad and display

- Two threads in the renderer process are involved in:
    - The **main thread** takes care of the 1–5 phases
    - The **compositor thread** manages the 6–8 phases

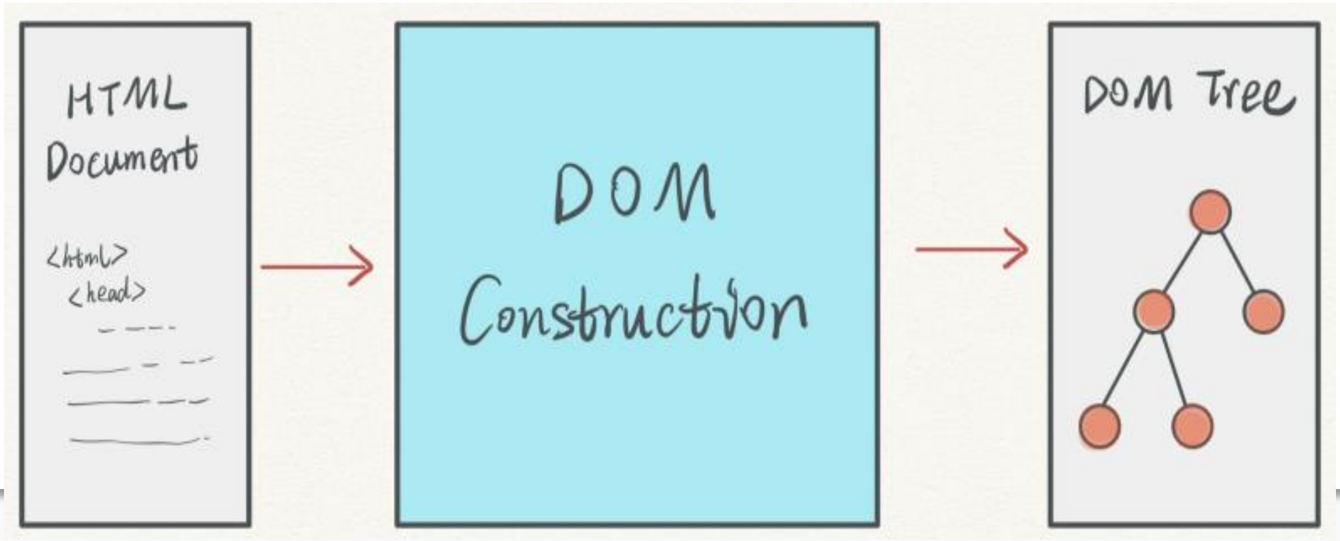


Phase 1: DOM construction

- An **HTML document** converts to a **DOM tree**.
- Why DOM tree? A browser doesn’t speak HTML.
- A DOM tree is a representation of HTML codes. We can visit it by entering the “*document*” in the Chrome console.
- The DOM tree exists in memory. Therefore, JavaScript can visit and edit it.

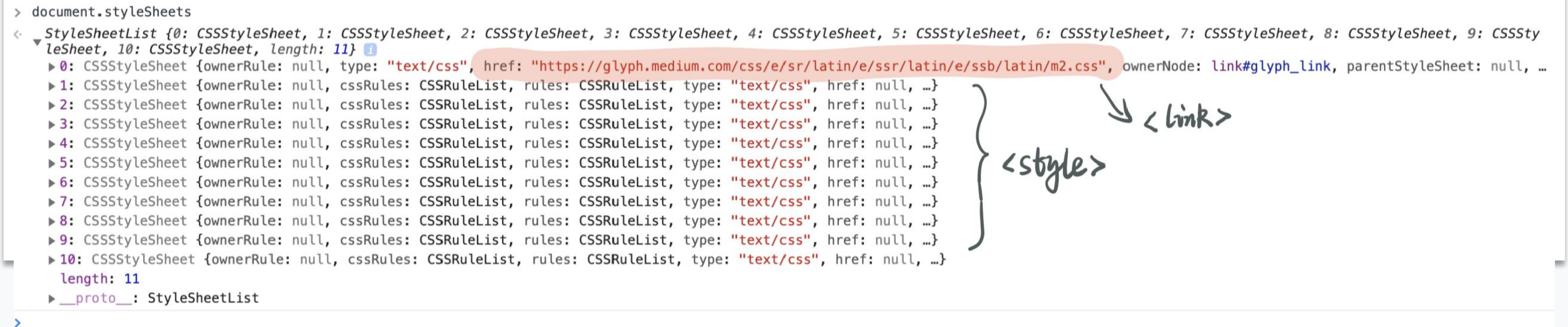


```
<html>

<body>

  <h1>Hello world!</h1>

  <div>
    <p>
      It is a message from
      <span>the web</span>
    </p>
  </div>

</body>

</html>
```



Phase 2: Style computation

- The input is **CSS styles**, and the output is a **computed style structure**.
- **Step 1: “Translating” CSS**
  - Similar to HTML, the browser doesn’t speak CSS. The translated result is **style sheets**.
  - Entering *“document.styleSheets”* in the Chrome browser console, we can see all parsed style sheets.
  - These style sheets come from
    - a source linked in the *<link>* tag,
    - styles inside of a *<style>*, and
    - inline styles
  - Same as a DOM tree, the style sheets look like a JavaScript object structure and can be visited and edited in the memory.



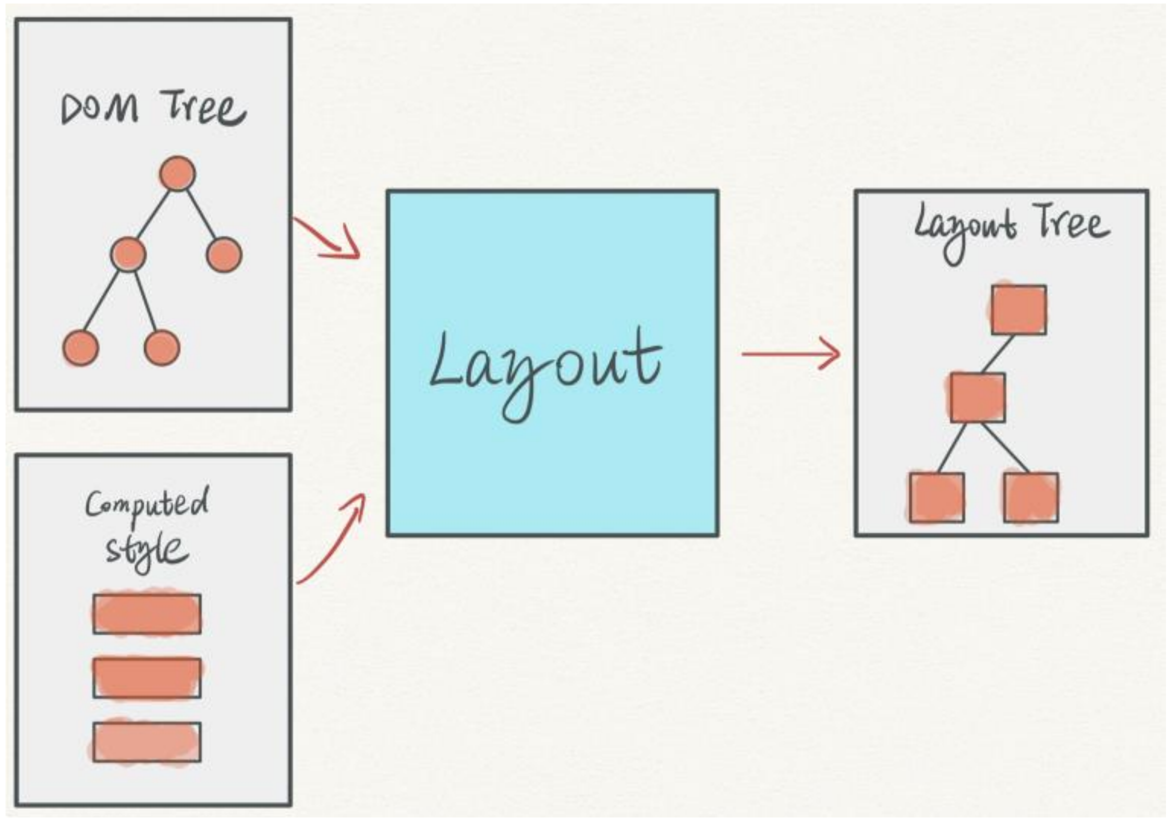
```
> document.styleSheets
<· ▼StyleSheetList {0: CSSStyleSheet, 1: CSSStyleSheet, 2: CSSStyleSheet, 3: CSSStyleSheet, 4: CSSStyleSheet, 5: CSSStyleSheet, 6: CSSStyleSheet, 7: CSSStyleSheet, 8: CSSStyleSheet, 9: CSSSty
    leSheet, 10: CSSStyleSheet, length: 11}
   ▶0: CSSStyleSheet {ownerRule: null, type: "text/css", href: "https://glyph.medium.com/css/e/sr/latin/e/ssr/latin/e/ssb/latin/m2.css", ownerNode: link#glyph_link, parentStyleSheet: null, …
   ▶1: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶2: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶3: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶4: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶5: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶6: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶7: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶8: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶9: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
   ▶10: CSSStyleSheet {ownerRule: null, cssRules: CSSRuleList, rules: CSSRuleList, type: "text/css", href: null, …}
    length: 11
   ▶__proto__: StyleSheetList
>
```

Phase 2: Style computation (Cont.)

- **Step 2: standardizing values and unites**
  - All relative values are converted to the absolute ones, pixels. Why pixels? At the end of the rendering phase, a browser displays bitmaps on the screen. Bitmaps are made of pixels.
    - `width: 50%;` => `width: 500px` (Assuming the width of its parent element is `1000px`.)
    - `padding: 2em 0` => `padding: 32px 0` (Assuming the font size of the element is `16px`.)
    - `font-size: 1em` => `font-size: 16px` (Assuming the font size of the root element is `16px`.)
- **Step 3: style computation**
  - Finally, the renderer process calculates the computed styles and attaches them to each element.
  - The renderer process computes all styles based on the cascading rules. Then it generates a list of final computed styles for each element.
  - For example, an element inherits the `font-size: 16px` from its ancestors. Itself has a `font-size: 32px`. The computed `font-size` style is `font-size: 32px`.

Phase 3: Layout

- In this phase, the inputs are the **DOM tree** and **computed styles**. The output is a **layout tree** with computed layout information.



Phase 3: Layout (Cont.)

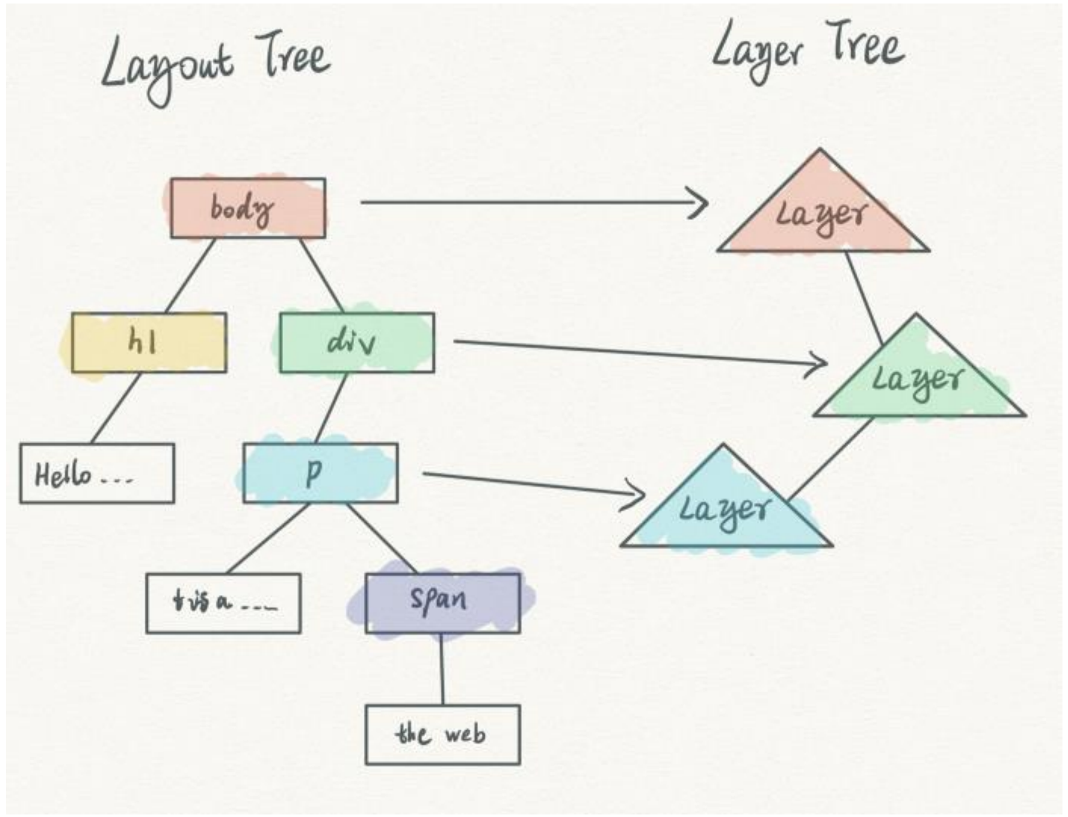
- **Step 1: constructing the layout tree**
  - The layout tree looks like a duplication of the DOM tree.
  - Differences
    - All “invisible” elements in the DOM tree won’t be included in the layout tree.
      - an element with `display: none`
      - all elements inside of the <`header`> tag
    - Some CSS properties add content to the layout.
      - `div::after {content: ‘I’m here’;}` creates a content included in the layout tree though it is not existing in the DOM tree

- **Step 2: calculating the geometry information**
  - Each element in the layout tree is a box.
  - To paint a box on a blank canvas, we need to know:
    - the starting `x`, `y` coordinates, and
    - the size of the box

Phase 4: Layer

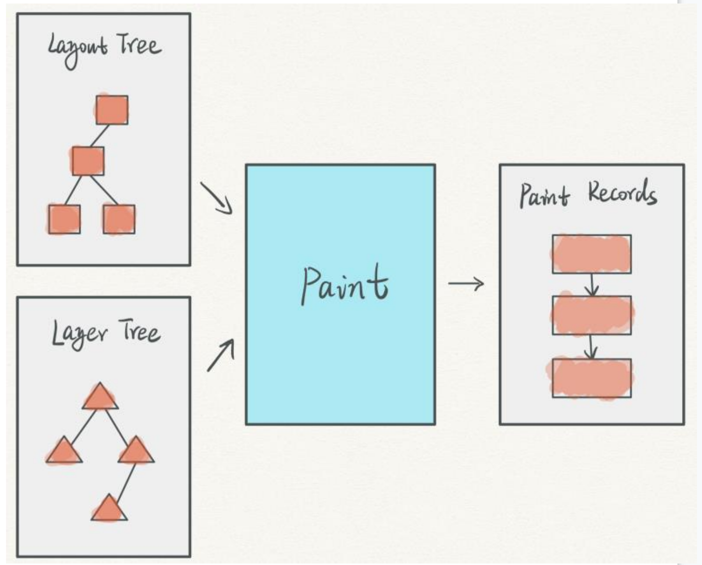
- The input is the **layout tree**, and the output is a **layer tree**.
- A layer tree is about the painting order
  - 3d transforming effect
  - *z-index* property

- Two kinds of elements are considered by the renderer process.
  - **Elements with stacking context**
    - z-index
    - position: absolute
    - transform
  - **Clipped elements**
    - A typical case of clipped elements is an overflowed text element.



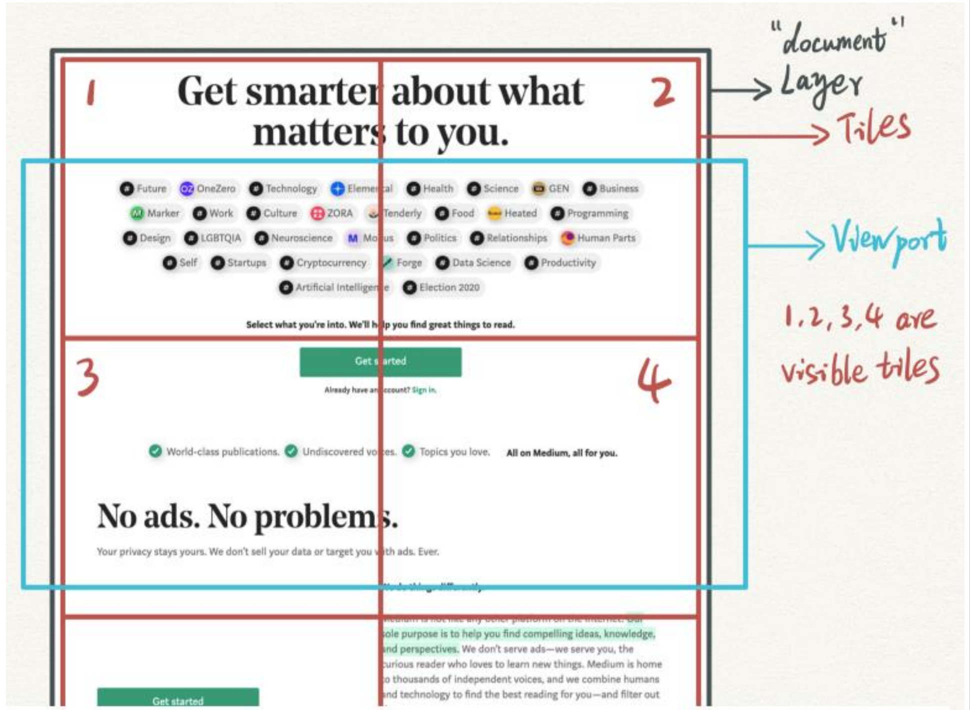
Phase 5: Paint

- In this phase, the inputs are the **layout tree** and **layer tree**, and the output is **paint records**.
- A paint record has 3 parts:
  - Action
  - Position, including the coordinates (x, y) and the size (width, height).
  - Styles
- For example, a paint record looks like the following:
  - `Action: Draw Rect`
  - `Pos: 0, 0, 300, 300`
  - `backgroundColor: red`
  - It is a paint record to draw a `300px * 300px red rectangular` at `(0, 0)`.
- A paint record is like a note for the browser to execute the painting.
- The paint records are a list of these notes in an order we confirmed in the layer tree, like “background first, then rectangle, then text”.



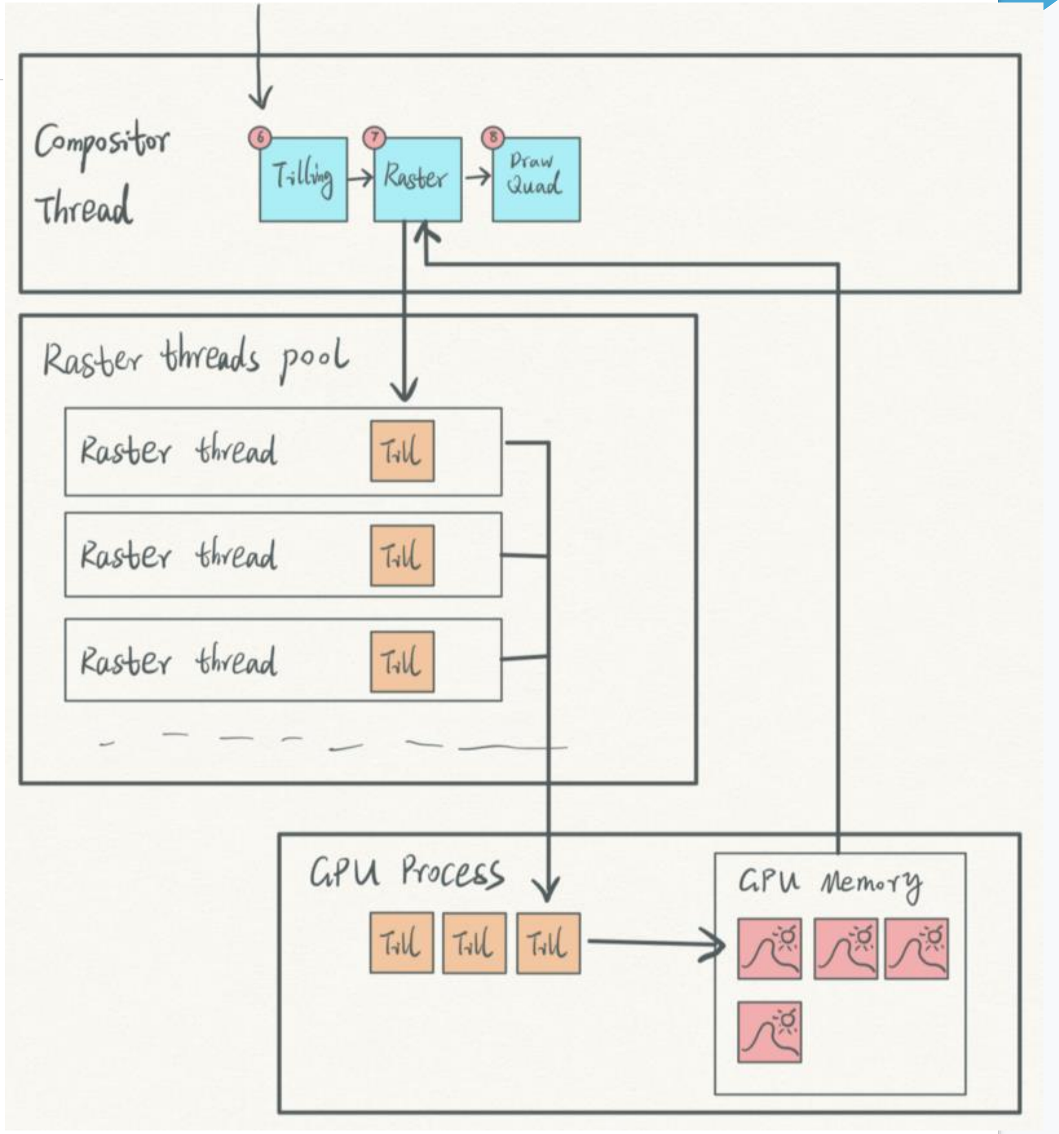
Phase 6: Tilling

- The input are the **paint records** and **layers**, and the output is **tiles**.
- When tiling, the layers are separated into tiles, and the browser priorities the rendering based on the viewport position.
- There are three keywords here: layer, tile, and viewport.
- Rendering all tiles are expensive. The visible tiles, which are currently in the viewport, have the priority.


"document"
Layer
Tiles
Viewport
1, 2, 3, 4 are visible tiles
1
2
3
4
Get smarter about what matters to you.
Select what you're into. We'll help you find great things to read.
Get started
No ads. No problems.
Your privacy stays yours. We don't sell your data or target you with ads. Ever.

Phase 7: Raster

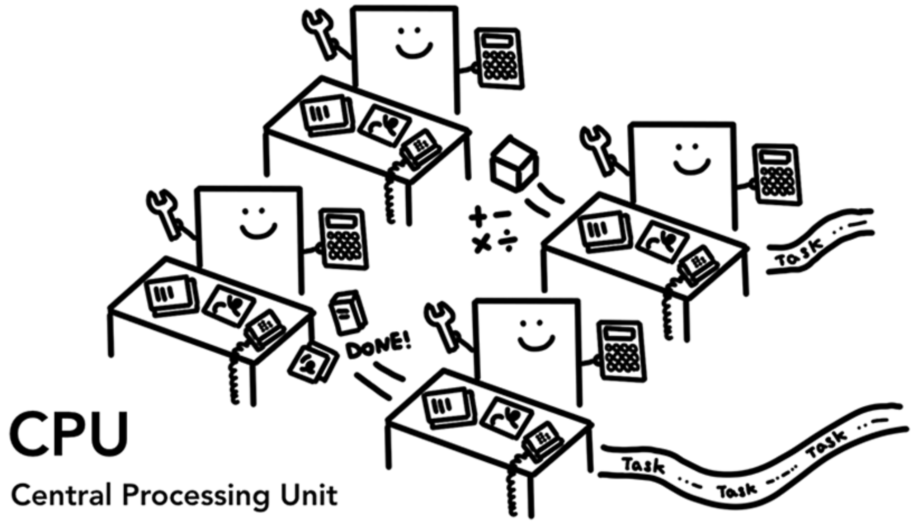
- The input is **tiles**, and the output is **bitmaps**.
- After knowing the tiles, the compositor thread creates a raster threads pool.
- Multiple raster threads carry on rastering tiles simultaneously.
- To accelerate the process, the raster threads send tiles to the GPU process through IPC. Then the GPU process generates bitmaps from tiles and saving bitmaps in GPU memory.
- When the bitmaps are ready, GPU process delivers them back to the compositor thread in the render process for the next step.

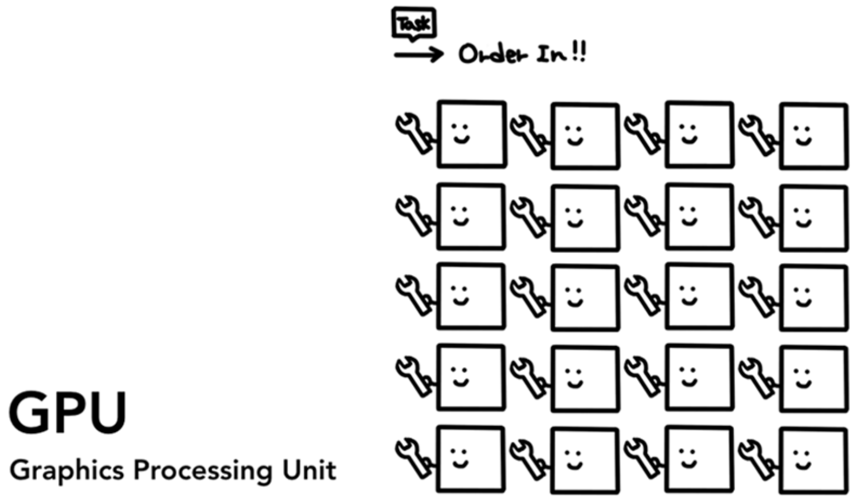


CPU(Central Processing Unit) VS GPU (Graphics Processing Unit)

- CPU
  - computer's brain
  - A CPU core, pictured here as an office worker, can handle many different tasks one by one as they come in.
  - It can handle everything from math to art while knowing how to reply a customer call.
  - In the past most CPUs were a single chip. A core is like another CPU living in the same chip.
  - In modern hardware, you often get more than one core, giving more computing power to your phones and laptops.

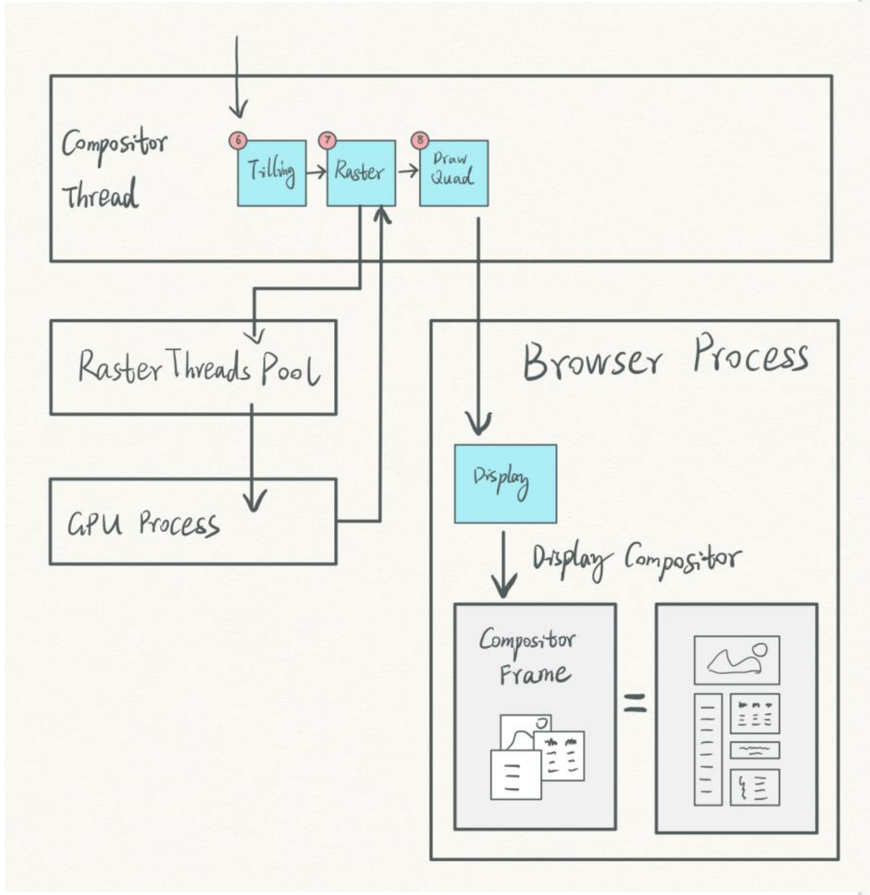
- GPU
  - GPU is good at handling simple tasks but across multiple cores at the same time.
  - As the name suggests, it was first developed to handle graphics. This is why in the context of graphics "using GPU" or "GPU-backed" is associated with fast rendering and smooth interaction.
  - In recent years, with GPU-accelerated computing, more and more computation is becoming possible on GPU alone.





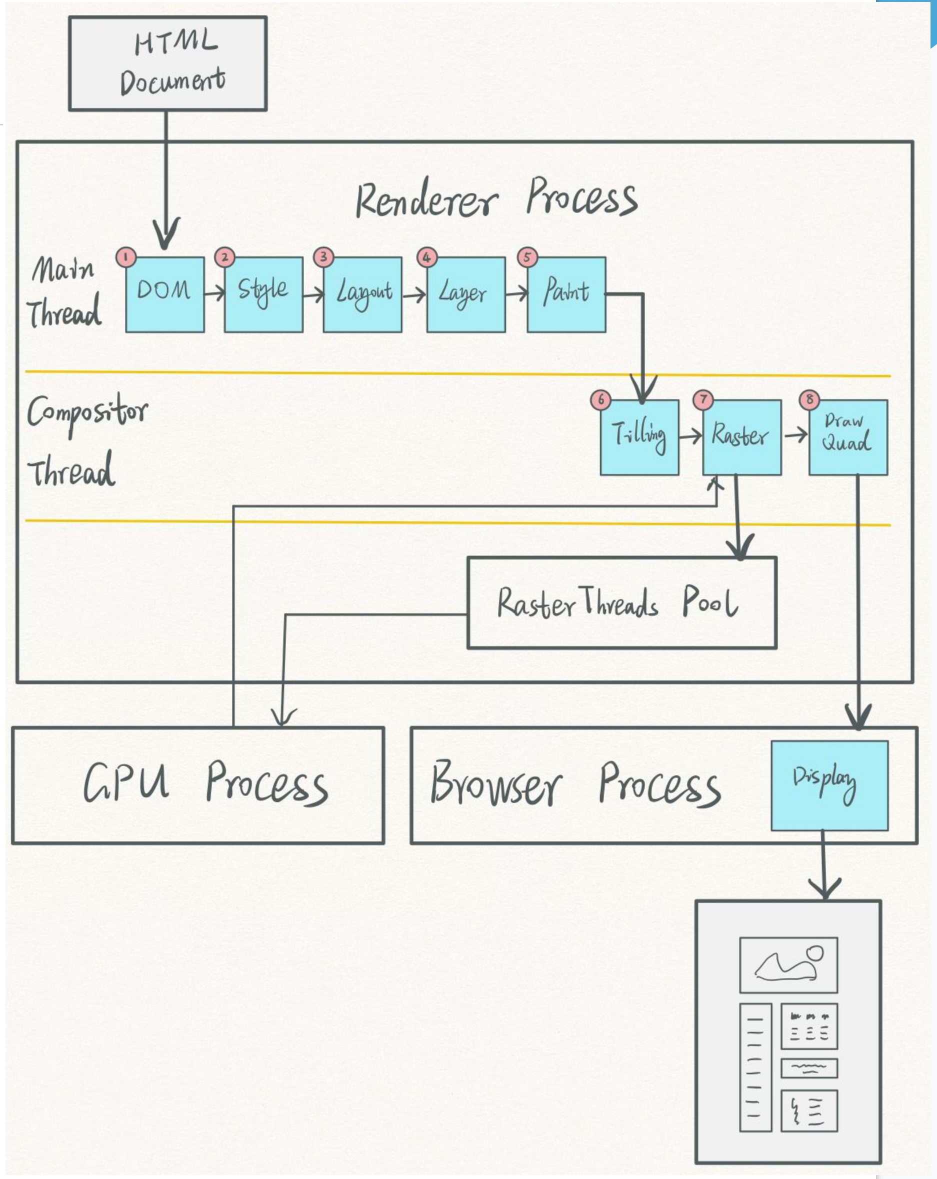
Phase 8: Draw Quad and Display

- The input is bitmaps, and the output is a compositor frame.
- When all needed tiles are processed, the compositor thread sends a command called Draw Quad to the browser process.
- Inside of the browser process, a “viz” component receives the Draw Quad command, executes Display Compositor command, and “draws” a compositor frame into our computer memory.
- Finally, the browser process displays the compositor frame in the browser.
- It is impressive that the entire 8 phases are happening within 1/60 second in a modern browser. That is 16.67 milliseconds.



Summary of Rendering Phase

1. The main thread in the renderer process “translates” the HTML document to **DOM tree**
2. It conveys CSS into the **computed style**
3. It creates a **layout tree** based on the DOM tree and computed style
4. From a layout tree, it generates a **layer tree**
5. Lastly, it establishes **paint records**
6. The compositor thread carries over the paint records, starts **tilling** based on the current viewport
7. Multiple raster threads **raster** tiles into bitmaps with the help of the GPU process.
8. The browser process receives the **Draw Quad** command from the render process, then it displays a frame of the page in front of us.



JS Engine

ECMAScript

- **ECMAScript** is the scripting language that forms the basis of JavaScript. ECMAScript standardized by the ECMA International standards organization in the **ECMA-262 and ECMA-402 specifications**.
- ECMAScript Specification
  - ES5: https://262.ecma-international.org/5.1/
  - The latest version ES2020: https://262.ecma-international.org/11.0/
- Based on the specification, you can create your own JavaScript Engine.

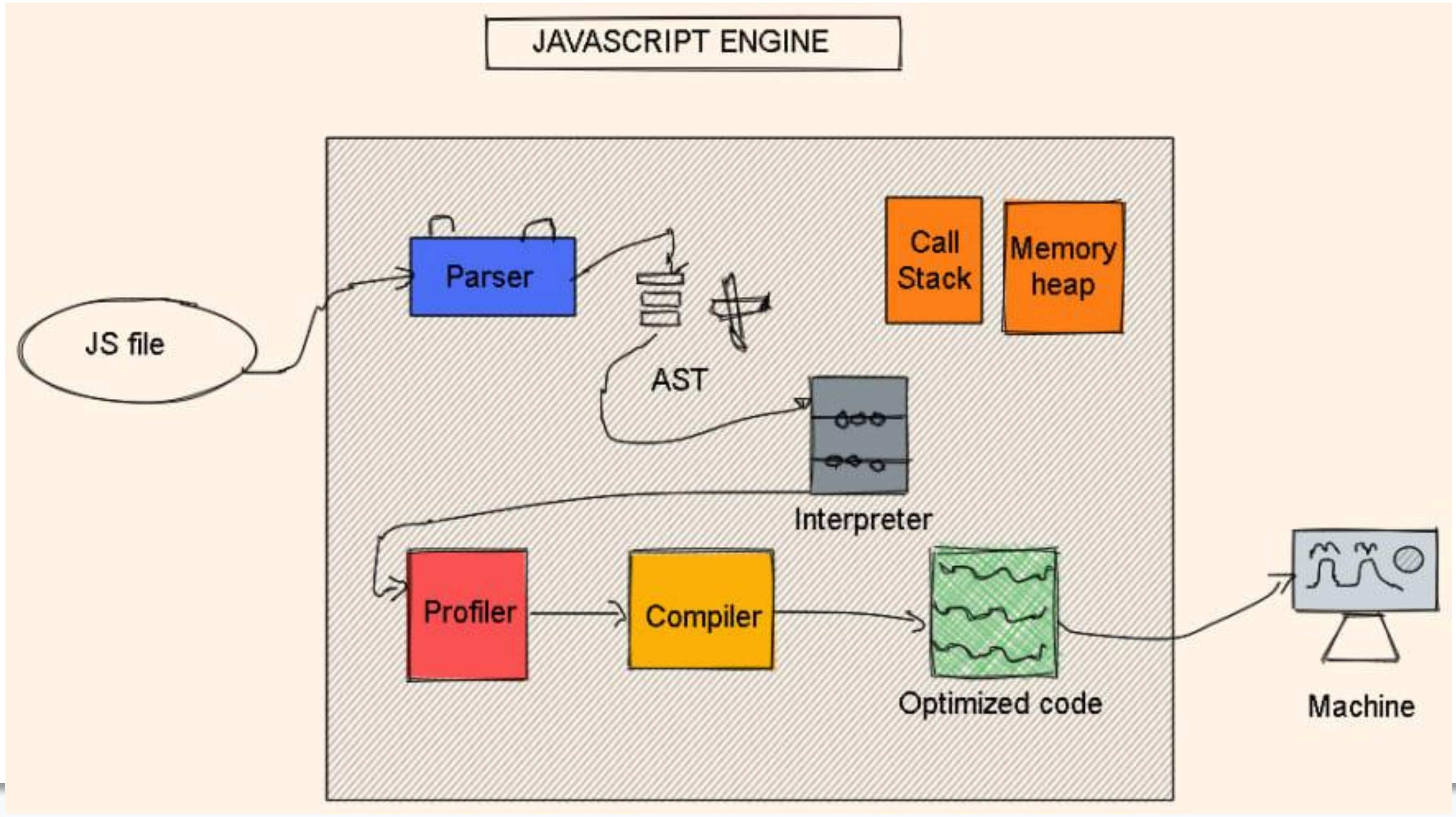
JavaScript Engine

- A **JavaScript engine** is a computer program that executes JavaScript code.
- It should follow the ECMAScript standard and how the language should work and what features it should have.

- Notable Engines:
  - V8 from Google is the most used JavaScript engine. Google Chrome and the many other Chromium-based browsers use it, or any other framework that embeds Chromium. Other uses include the Node.js and Deno runtime systems.
  - SpiderMonkey is developed by Mozilla for use in Firefox and its forks.
  - JavaScriptCore is Apple's engine for its Safari browser. Other WebKit-based browsers also use it.
  - Chakra is the engine of the Internet Explorer browser. It was also forked by Microsoft for the original Edge browser, but Edge was later rebuilt as a Chromium-based browser and thus now uses V8.

The JavaScript Engine Pipeline

- JS is a higher level dynamic language and it has no way to directly interact with our machines lower level logic. So JavaScript engine can be implemented as a standard interpreter, or just-in-time compiler that compiles JavaScript to bytecode in some form.



Details about JS Engine Pipeline

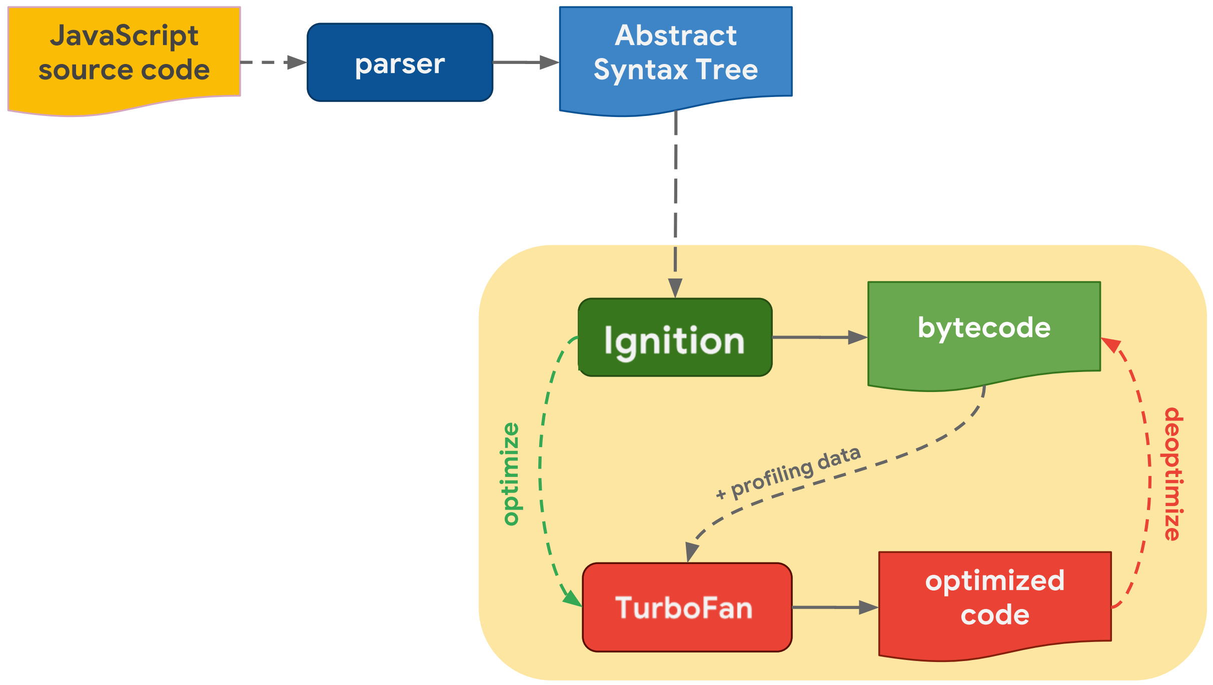
- Parser
  - The Html Parser will fetch all scripts loaded via <script> tag.
  - The source code inside this script gets loaded as a UTF-16 byte stream to a byte stream decoder.
  - This byte stream decoder then decodes the bytes into token and then its sent to parser.
- AST
  - The parser creates nodes based on the tokens it receives. With these nodes, it creates an Abstract Syntax Tree (AST).
  - https://astexplorer.net/
- Interpreter
  - Walks through the AST and generates byte code.
  - Reads the code line by line. When the byte code has been generated, the AST is deleted for clearing up memory space.
  - The problem with interpreters is that running the same code multiple times can get really slow.

Details about JS Engine Pipeline (Cont.)

- Profiler (JiT)
  - The Profiler monitors and watches code to optimize it.
  - While the code is executed by the interpreter, a profiler will keep track of how many times the different statements get hit. The moment that number starts growing, it'll mark it as **Warm** and if it grows enough, it'll mark it as **Hot**.
  - In other words, it'll detect which parts of your code are being used the most, and then it'll send them over to be compiled and stored.
- Compiler
  - The compiler works ahead of time and creates a translation of the code that has been written and compiles down to a lower level language that machines can read.
- Optimized code
  - highly-optimized machine code which can run faster

How V8 looks like?

- JavaScript is interpreted by an **interpreter** named **Ignition** as well as compiled by a **JIT optimizing compiler** named **TurboFan**.
- The AST generated in the previous step is given to the interpreter which generates non-optimized machine code quickly and the execution can start with no delay.
- Profiler watches the code as it runs and identifies areas where optimizations can be performed. Any unoptimized code is passed to the compiler to perform optimizations and generate machine code which eventually replaces its counterpart in the previously generated non-optimized code by the interpreter.



Example of JS Optimize Code

- Function Inlining
- Browser will sometimes essentially rewrite your JavaScript.
- As you can see from the right, V8 is essentially removing the step where we call `func` and instead inlining the body of `square`. This is very useful as it will improve the performance of our code.

```
const square = (x) => { return x * x };

const callFunction100Times = (func) => {
  for(let i = 0; i < 100; i++) {
    // the func param will be called 100 times
    func(2);
  }
}

callFunction100Times(square);
```

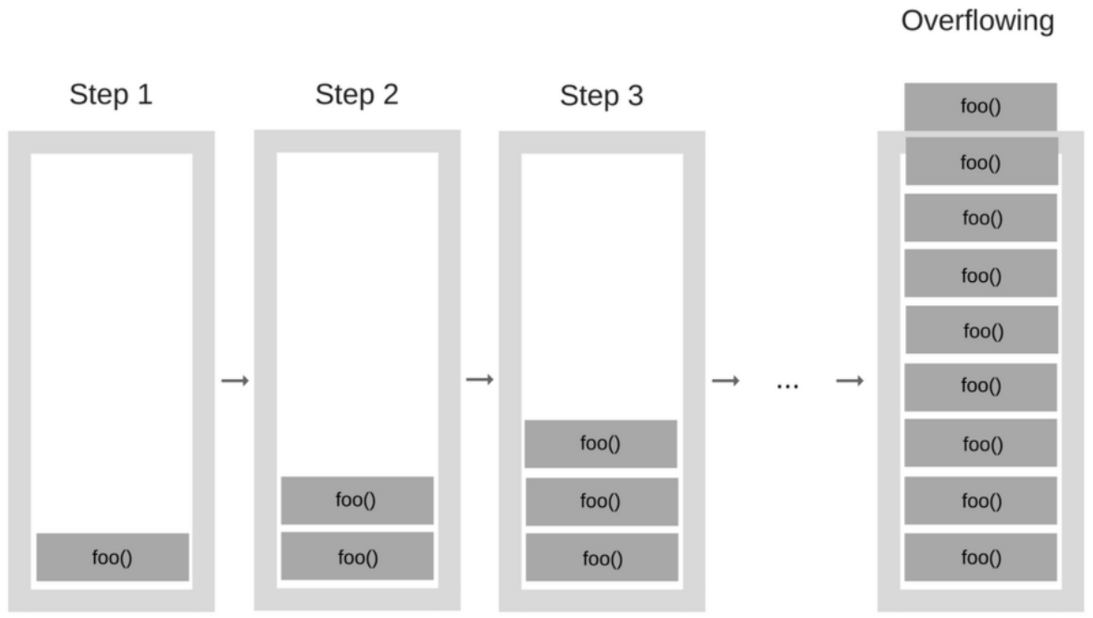
```
const square = (x) => { return x * x };

const callFunction100Times = (func) => {
    for (let i = 100; i < 100; i++) {
        // the function is inlined so we don't
 have to keep calling func
        return x * x;
    }
}

callFunction100Times(square);
```

The Memory Heap

- The memory heap is a place to store and write information, where we allocate, use and remove memory.
- Whenever you define a variable, constant, object, etc in your JavaScript program, it is stored in the memory heap. This memory is limited, and it is essential to make wise use of the available memory.
- Unlike languages like C, where we need to explicitly allocate and free memory, JavaScript provides the feature of automatic garbage collection. Once an object/variable is not used by any execution context, its memory is reclaimed and returned to the free memory pool. In V8 the garbage collector is named as **Orinoco**.

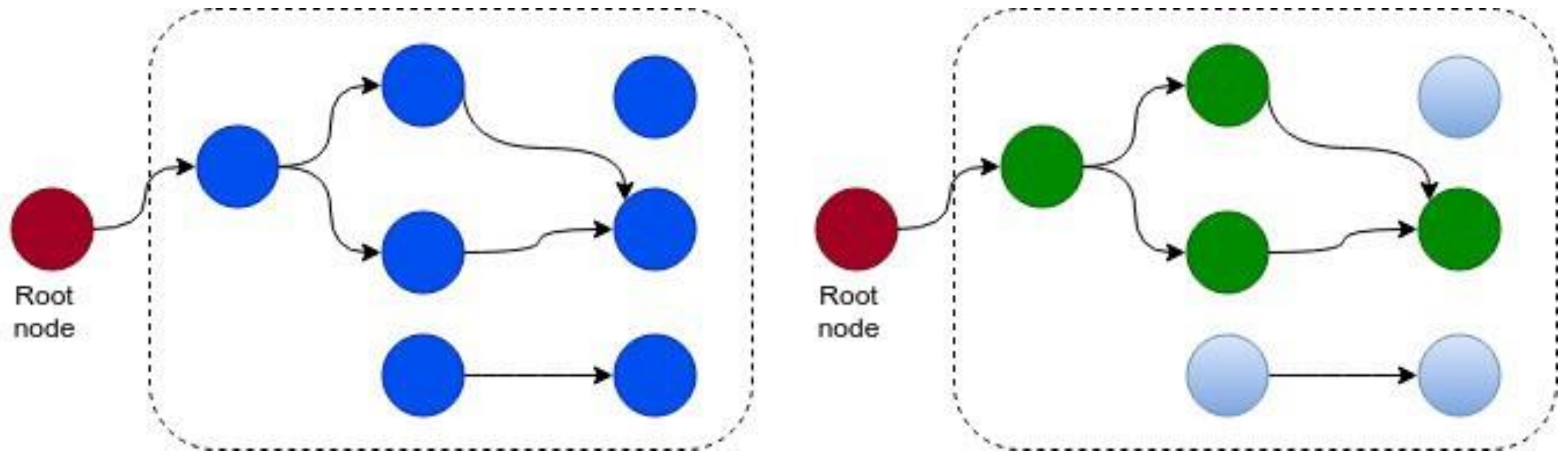
Call stack

- The call stack keeps track of where we are in the code. It uses first in and last out and stacks for example functions.
- The callstack calls a function from the memory heap and after executing removes it from the stack. When the maximum call stack has been reached, e.g. with an infinite loop, we call it a **stack overflow**.



Garbage Collection

- JS automatically frees up memory that no longer is used. It marks and sweeps it from the memory.
    - E.g. when a value gets reassigned and the original value is not used anymore.
- The garbage collector starts with the root or global objects periodically and moves to the objects referenced by them, and then to the objects referenced by these references and so on. All the unreachable objects are then cleared.



Memory leaks

- Memory leaks happens when a piece of memory is no longer being used or is required by an application but still is taking memory.
- This happens for example when you:
  - Accidentally set global variables
  - Don’t close eventListeners or setIntervals
  - Reference out of the DOM

References

- https://developers.google.com/web/updates/2018/09/inside-browser-part1
- https://cabulous.medium.com/how-browser-works-part-i-process-and-thread-f63a9111bae9
- https://cabulous.medium.com/how-does-browser-work-in-2019-part-ii-navigation-342b27e56d7b
- https://cabulous.medium.com/how-does-browser-work-in-2019-part-iii-rendering-phase-i-850c8935958f
- https://medium.com/@sanderdebr/a-brief-explanation-of-the-javascript-engine-and-runtime-a0c27cb1a397