The JS Engine and Modern Web Browsers

CS445 Modern Asynchronous Programming

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

Maharishi University of Management - Fairfield, Iowa

JavaScript

JavaScript is the most popular programming language on the web. You can use it to create websites, servers, games and even native mobile apps.

It’s the most valuable skill in today’s job market.

ECMAScript Specs

ECMAScript is the standard JS is based on since there are many engines. ECMA is an organization. A standard that everyone agreed on.

ECMAScript 2015 specification:
http://www.ecma-international.org/ecma-262/6.0/

Basically you can write your own JS engine based on those specifications.

JS Engine: A program that convert JS code into something that computer processor understands, it should follow the ECMAScript standard and how the language should work and what features it should have.

JS Engines VMs

Google: V8

Mozilla: Spider Monkey

Microsoft: Chakra Core

Apple: JavaScript Core

Read more about JS Engines

Google V8 JavaScript Engine

Google V8 engine is an open source and you may use or change. It's used in Google Chrome. It implements ECMAScript and runs on many processors.

You may find the source code at https://github.com/v8/v8

Programming Languages

The computer is made of microprocessors and we write code to instruct this electronic chips to calculate math instructions.

But what is the language that the microprocessors understand? They don’t understand JavaScript, Python, etc., but only machine code.

Writing machine code or assembly language is not feasible, so we need High-level languages like JavaScript along with a compiler or an interpreter to convert it to machine code.

Compilers and Interpreters

Interpreter: Reads and translates the file line by line on the fly.

Compiler: A compiler works ahead of time and creates a new file which contains the machine code translation for your input file.

What are the differences in term of performance, optimization, and execution?

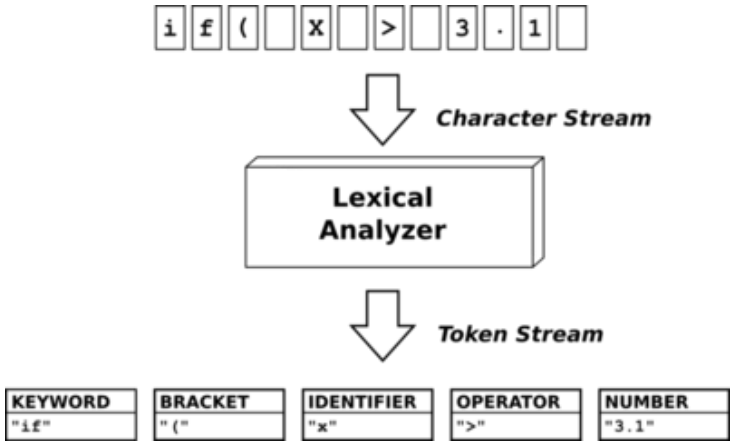
But How The JS Engine Works?

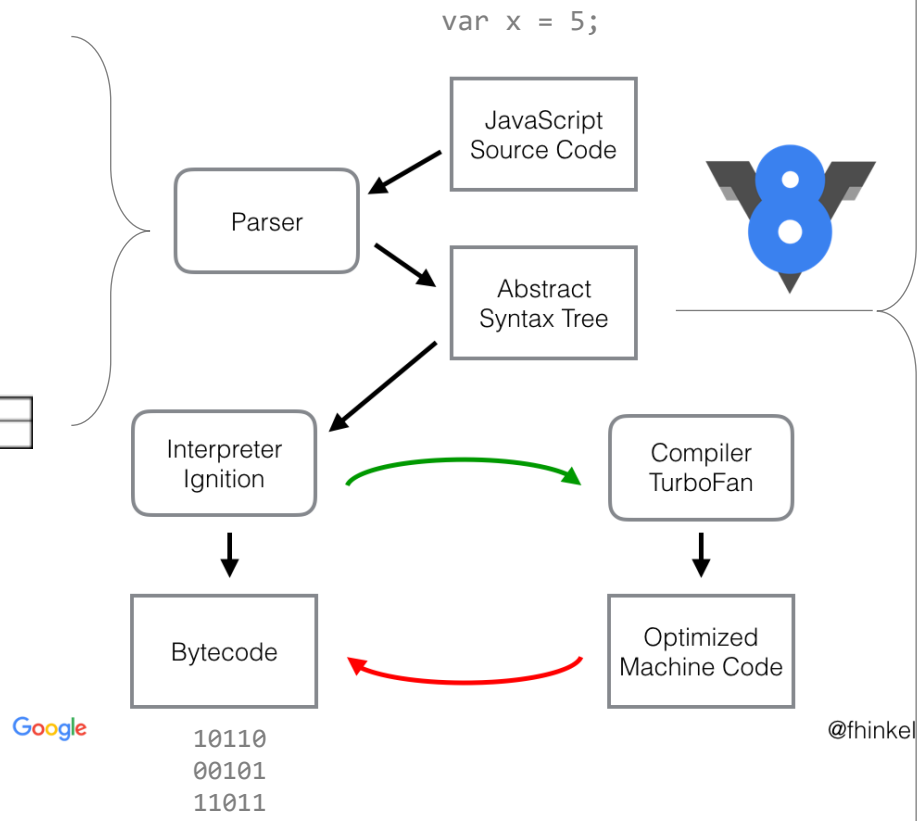
The JavaScript file enters the engine and the parser does lexical analysis which breaks the code into **tokens** to identify their meaning. These tokens make the **AST (Abstract Syntax Tree).** Later on, the ASTs are used to generate the actual bytecode or machine code.

ECMAScript engines use the best of both worlds and make a JIT(Just-in-time) compiler. JavaScript is compiled as well as interpreted but the actual implementation and order depend on the engine.

Demo: JS AST Visualizer

V8






```
{
  "type": "Program",
  "start": 0,
  "end": 10,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 10,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 9,
          "id": {
            "type": "Identifier",
            "start": 4,
            "end": 5,
            "name": "x"
          },
          "init": {
            "type": "Literal",
            "start": 8,
            "end": 9,
            "value": 5,
            "raw": "5"
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "module"
}
```

Heart of the Engine

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.

Ignition logo

Turbofan logo

JiT Profiler

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.

Example

The baseline compiler will turn the **`result += arr[i]`** line into a **Stub**, but because this instruction is **polymorphic** (there is nothing ensuring that **`i`** is going to be an integer every time or that **`arr[i]`** is going to be a string for every position on the array) it'll create a **Stub** for every possible combination.

```
function concatArray(arr) {
  var result = "";
  for(let i = 0; i < arr.length; i++) {
    result += arr[i]
  }
  return result;
}
```

Think about every step of the **`for`** loop, the interpreter is asking:

- Is **`i`** an integer?
- Is **`result`** a string?
- Is **`arr`** actually an array?
- Is **`arr[i]`** a string?

The Baseline Compiler

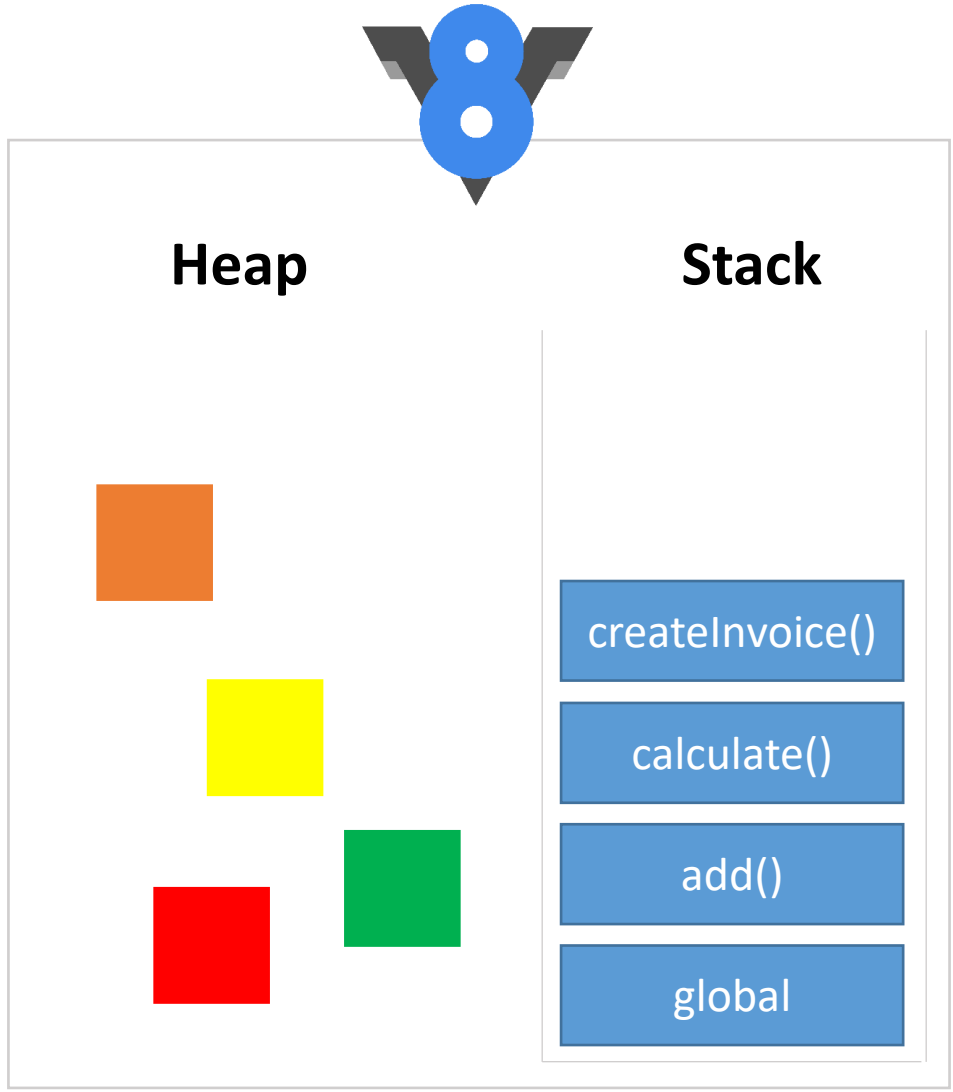
The warm sections of your code will be compiled into bytecode, which in turn, is run by an interpreter that is optimized for that type of code.

This alone will make those sections run faster, and if possible, the baseline compilation will also try to optimize the code by creating **Stubs** for every instruction being analyzed.

The Optimizing Compiler

The optimizing compiler will take charge and turn all those isolated **Stubs** into a group, this allows for the **type checks** to happen only once, before the function call, instead of on every loop.

V8 Core



Memory Allocation

Execution Context

One thread = One call stack = One thing at a time

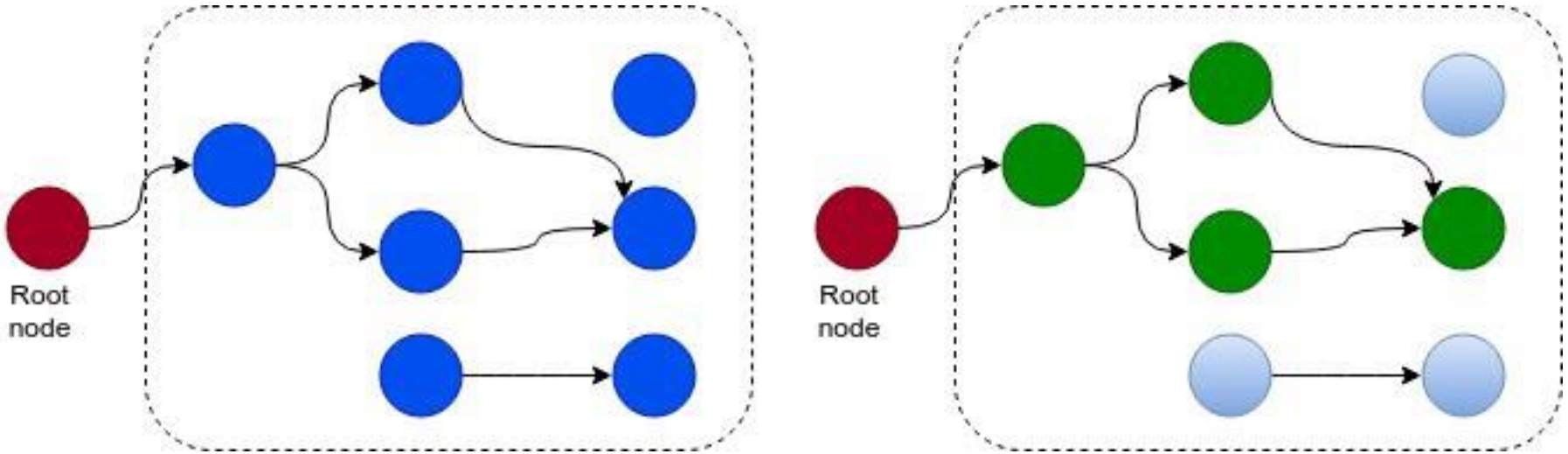
The Memory Heap

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**.

Mark and Sweep Algorithm

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

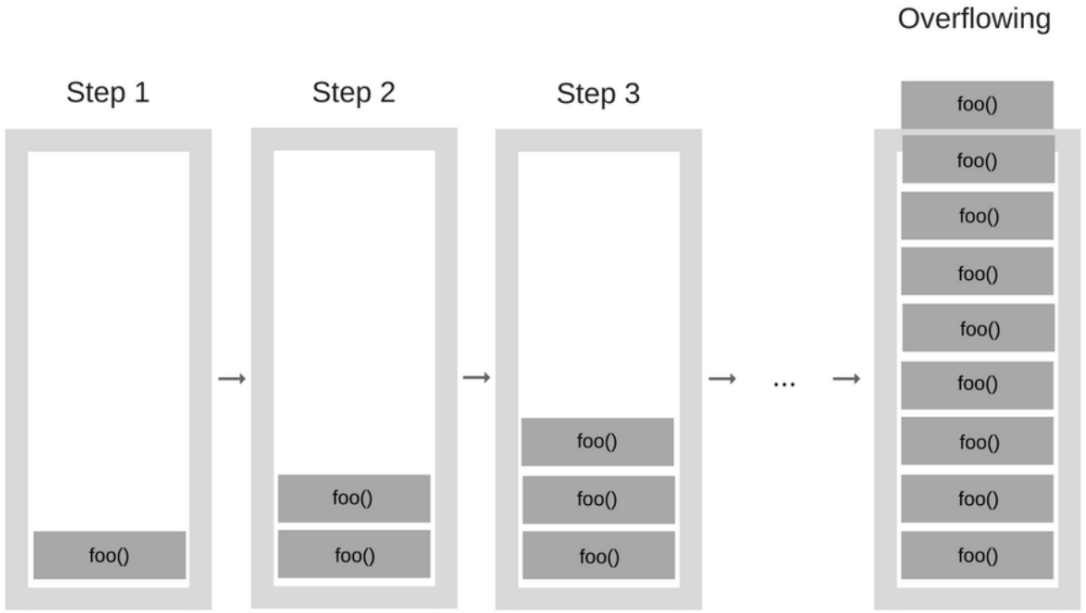
Memory leaks are parts of memory that the application needed and used in the past and it is not needed anymore but its storage is yet not returned to the memory pool.

What causes memory leaks:

- Global variables
- Event Listeners
- Intervals and Timeouts
- Removed DOM elements

The Call Stack (Execution Context)

A stack is a data structure that follows the LIFO (Last in first out) approach to store and access the data. In case of the JavaScript engine, the stack is used to remember the location of last executed command in a function.



Stack Overflow

The amount of consecutive push you can do without doing a pop on the stack depends on the size of the stack. If you run the limit and keep pushing, it will lead to something called stack overflow and chrome will throw an error along with the snapshot of the stack also called as the stack frame.

```
function callme() {
     callme();
}
```

Best Practices

Don't change object shape

The compiler will be at a loss and can no longer assume both objects belong to the same class, thus it needs to create new ones, and then it loses any possible optimization options.

Keep function arguments constant (monomorphic)
The more you change the types of the attributes you use to call your functions the more complex the function becomes in the eyes of the compiler.

10 Mins Break

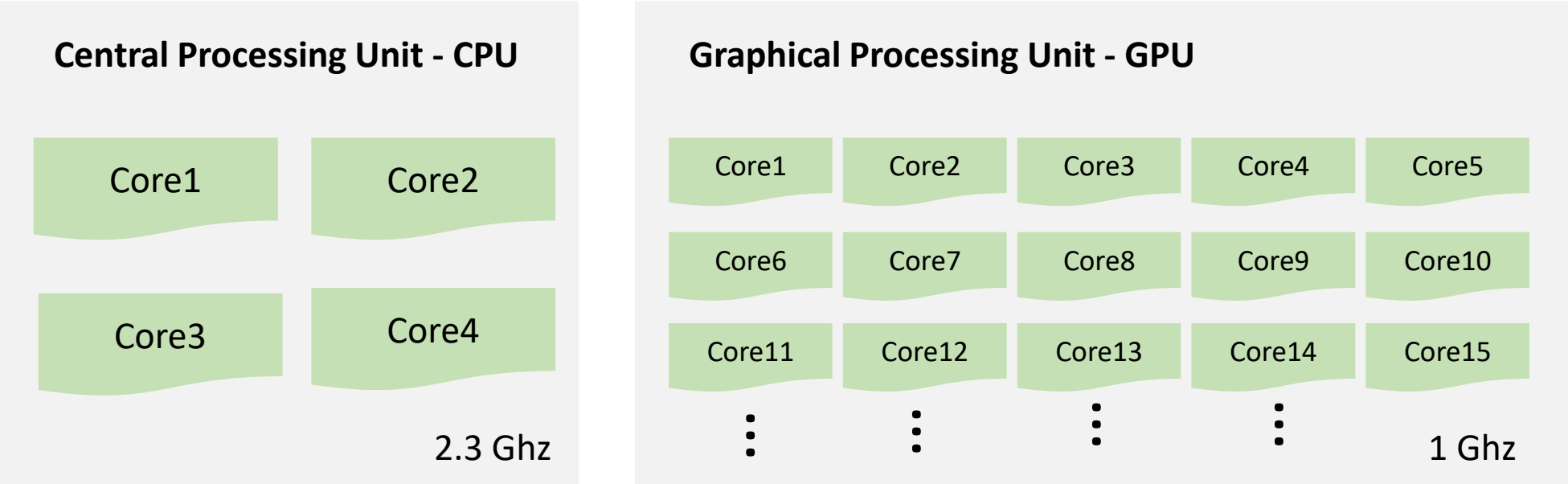
Modern Web Browsers

In order to build a Modern Web Application, it's very important to deeply understand how **Modern Web Browsers** work and what they do with the code we write.

At the core of the computer are the Central Processing Unit **CPU** and the Graphics Processing Unit **GPU**.

Applications run on the CPU and GPU using mechanisms provided by the Operating System.

CPU vs GPU



- CPU performs **simple** central works, while GPU performs **complex** calculations (render animations, images, and videos).
- CPU commonly has 4–8 fast and flexible cores clocked at 2.3Ghz whereas GPU has thousands of relatively simple cores clocked at about 1Ghz.
- CPU is good for processing **sequential** works whereas GPU is very good at processing **parallel** tasks.

Process vs Thread

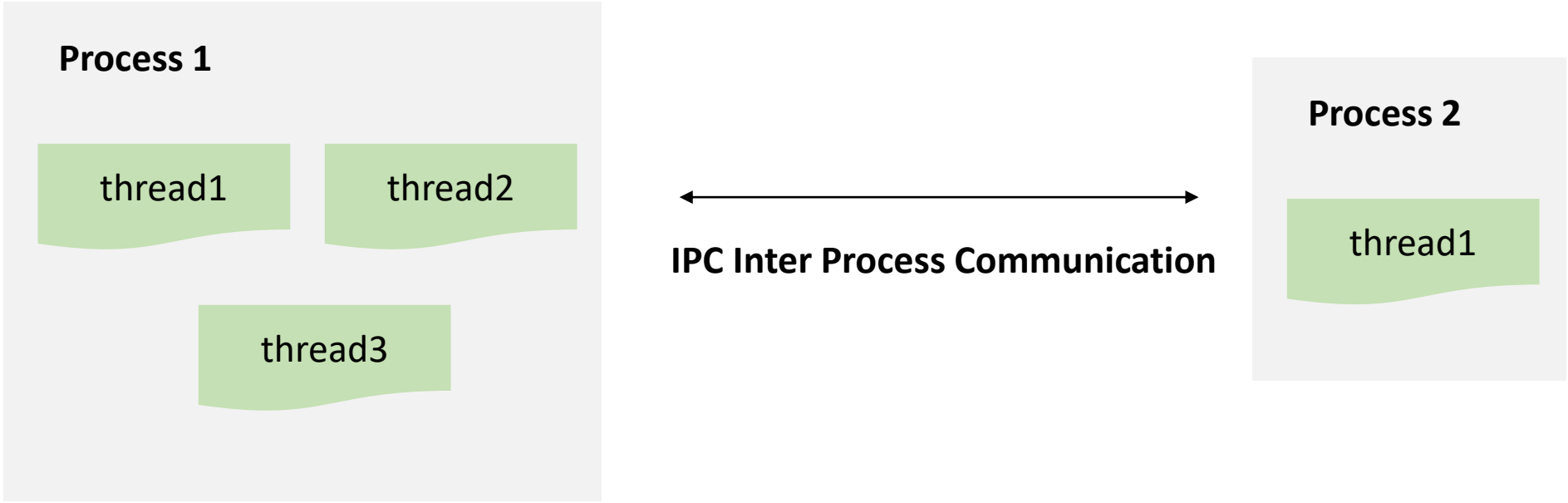
A **Process** is an application’s executing program lives in memory.

A **Thread** lives inside of process and executes any part of its process's program.

When you start an application, a process is created. The program might create thread(s) to help it do work. The OS gives the process a space of memory to work with and all application state is kept in that private memory space. When you close the application, the process goes away and the OS frees up the memory.

A process can ask the OS to fork another process to run different tasks. When this happens, different parts of the memory are allocated for the new process. If two processes need to talk, they can do so by using **Inter Process Communication (IPC).**

Application in Memory



Browser Architecture

A web browser can be built using one process with many different threads, or with many different processes with a few threads communicating over IPC.

Chrome Architecture: The main browser process coordinates with other processes that take care of different parts of the application (plugin processes, GPU process, renderer processes, utility process). For the renderer process, multiple processes are created and assigned to each tab (per domain). Chrome gives each site its own process (See Task Manager).

Chrome Processes

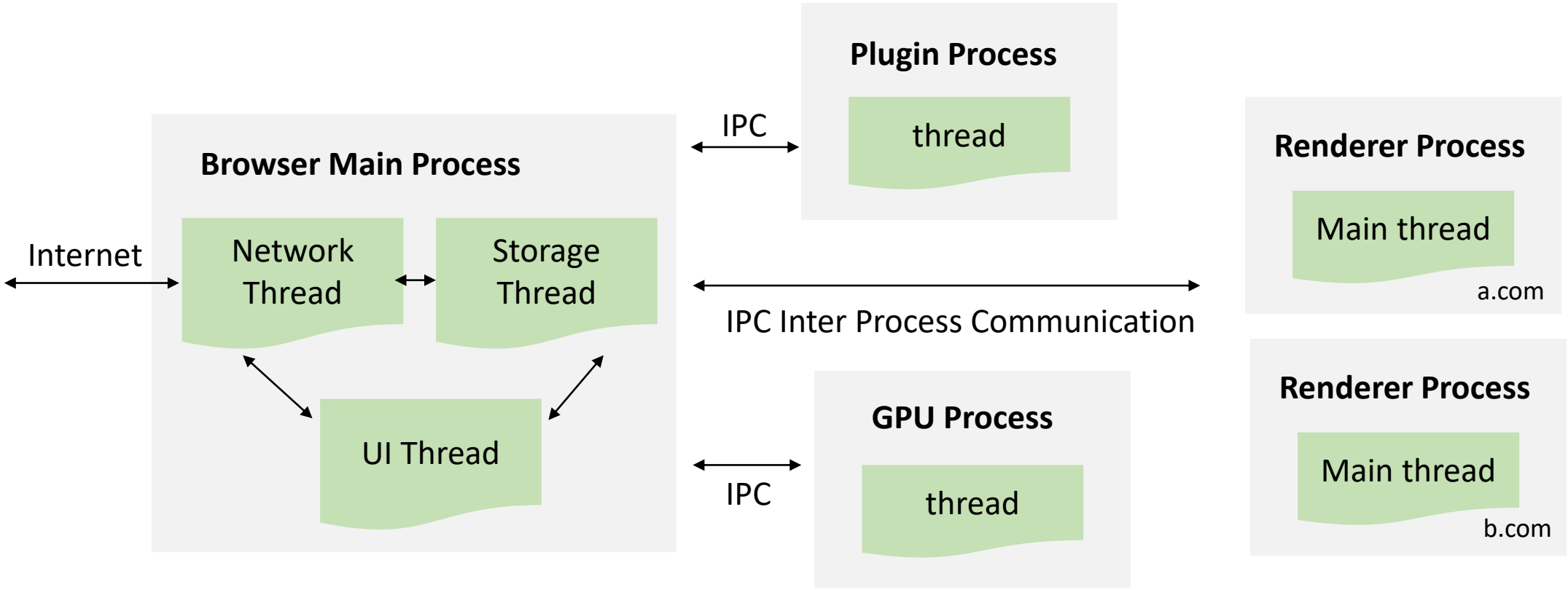
Browser Main Process Controls the address bar, bookmarks, back and forward buttons. Also handles other parts of a web browser such as network requests and file access.

Renderer Process Controls anything inside of the tab where a website is displayed (Represents the DOM for each tab).

Plugin Process Controls plugins used by the website.

GPU Process Handles GPU requests from multiple apps.

Chrome Architecture



Multi-core processes in Chrome

Responsiveness In most cases, you can imagine each tab has its own renderer process. Let’s say you have 3 tabs, and each tab is run by an independent renderer process. If one tab becomes unresponsive, then you can close it while keeping other tabs alive.

Security Because processes have their own private memory space, they often contain copies of common infrastructure (like V8 which is a Chrome's JavaScript engine, each tab has its own scope).

Site Isolation Running a separate renderer process for each cross-site iframe, making sure one site cannot access data from other sites without consent (Same Origin Policy).

Navigation Phase

You type a URL into a browser, the browser fetches data from the internet. Everything outside of a tab is handled by the **Browser process**: **UI thread** (navigation buttons and input address bar of the browser), **Storage thread** (access files, cache), **Network thread**.

UI thread needs to parse and decide whether to send you to a search engine, or to the site you requested.

When a user hits enter, the **UI thread** initiates a network call to get site content. The **network thread** goes through appropriate protocols like DNS lookup and establishing TLS Connection for the request.

Receiving the Response

Once the response starts to come in, the **network thread** looks at the first few bytes of the stream and starts parsing the header and body. The response **`Content-Type`** header should say what type of data it is.

If the response is an HTML file, then the **network thread** will need to find and pass the data to a **renderer process**, but if it is a zip file or some other file then that means it is a download request so they need to pass the data to download manager.

Cross Origin Read Blocking (CORB) check happens in order to make sure sensitive cross-site data does not make it to the renderer process.

Finding a Renderer Process

The **Network thread** tells **UI thread** that the data is ready (Address bar is updated, the security indicator and site settings UI reflects the site information of the new page). **UI thread** then finds/starts a **Renderer process** to carry on rendering of the web page.

An IPC is sent from the **Browser process** to the **Renderer process (main thread)** to commit the navigation. It also passes on the data stream so the Renderer process can keep receiving HTML data.

Once the Browser process hears confirmation that the commit has happened in the Renderer process, the navigation is complete and handed over and the Document Loading Phase begins.

Finishing Navigation Phase

Once the navigation is committed, the **Renderer process** carries on loading resources and renders the page. Once the renderer process finishes rendering, it sends an IPC back to the Browser process. At this point, the UI thread stops the loading spinner on the tab.

Note: There could be still a need to fetch additional resources and then render new views after this point.

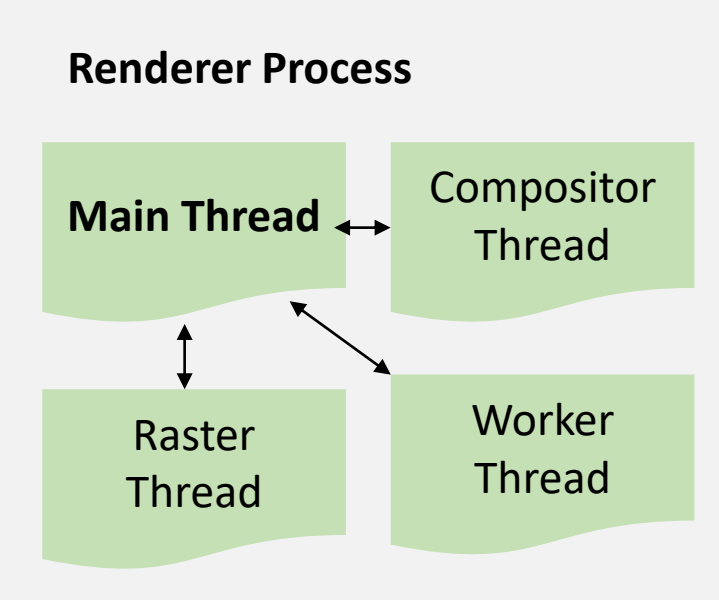
Renderer Process

The **Renderer process** is responsible for everything that happens inside of a tab.

The **Main thread** handles most of the code you send to the user.

Compositor and Raster threads are also run inside of a Renderer processes to render a page efficiently and smoothly.

The Renderer process core job is to turn HTML, CSS, and JavaScript into a web page that the user can interact with.



Parsing

Construction of a DOM

When the Renderer process receives a commit message for a navigation and starts to receive HTML data, the Main thread begins to parse the text string (HTML) and turn it into a Document Object Model (DOM).

Sub-Resource Loading

All external resources (images, CSS, and JavaScript) need to be loaded from network or cache. The Main thread could request them and sends requests to the Network thread in the Browser process.

JavaScript can block the parsing

When the HTML parser finds a `<script>` tag, it **Pauses** the parsing of the HTML document and has to load, parse, and execute the JavaScript code.

Style Calculation

Having a DOM is not enough to know what the page would look like because we can style page elements in CSS. The Main thread parses CSS and determines the **computed style** for each DOM node.

The browser has a default style sheet.

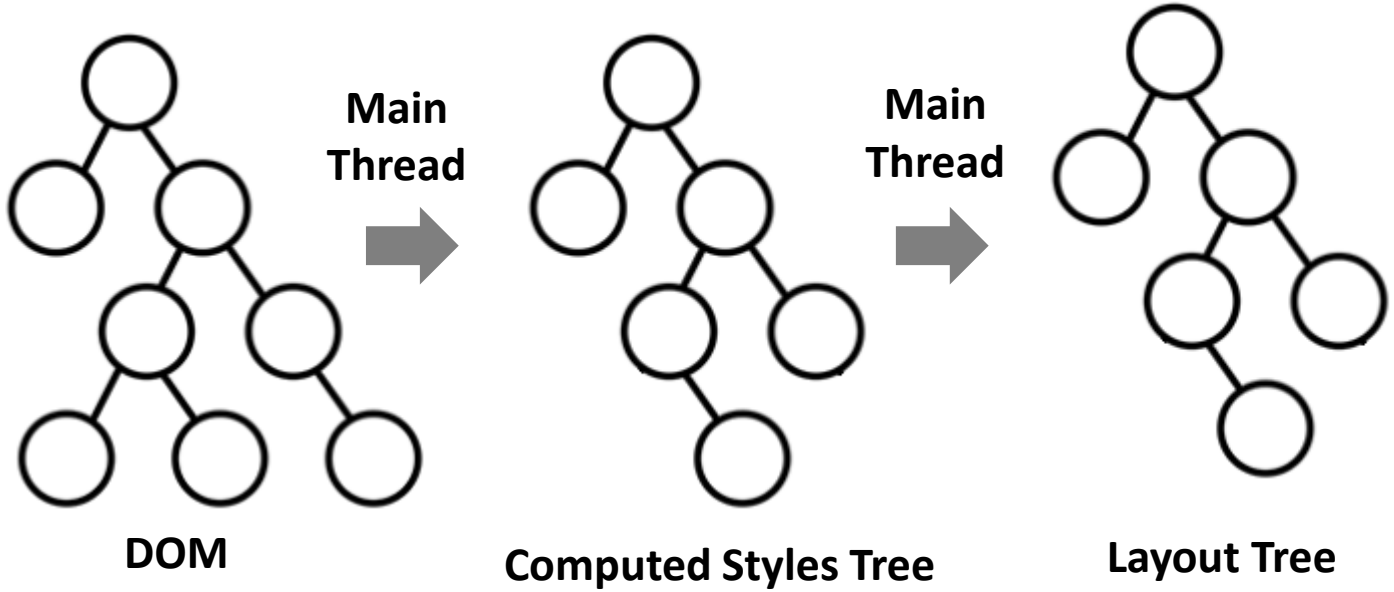
Layout

The layout is a process to find the geometry of elements. The Main thread walks through the DOM and computed styles and creates the **Layout Tree** which has information like x y coordinates and bounding box sizes. Layout tree may have similar structure to the DOM tree, but it only contains information related to **what's visible on the page**.

- If `display:none` is applied, that element is not part of the layout tree.
- An element with `visibility:hidden` is in the layout tree.
- If a pseudo class with content like `p::before{content:"CS572"}` is applied, it is included in the layout tree even though that is not in the DOM.

Paint

At this paint step, the main thread walks the layout tree to create **Paint Records**. Paint record is a note of painting process like "background first, then add text, then draw a rectangle".



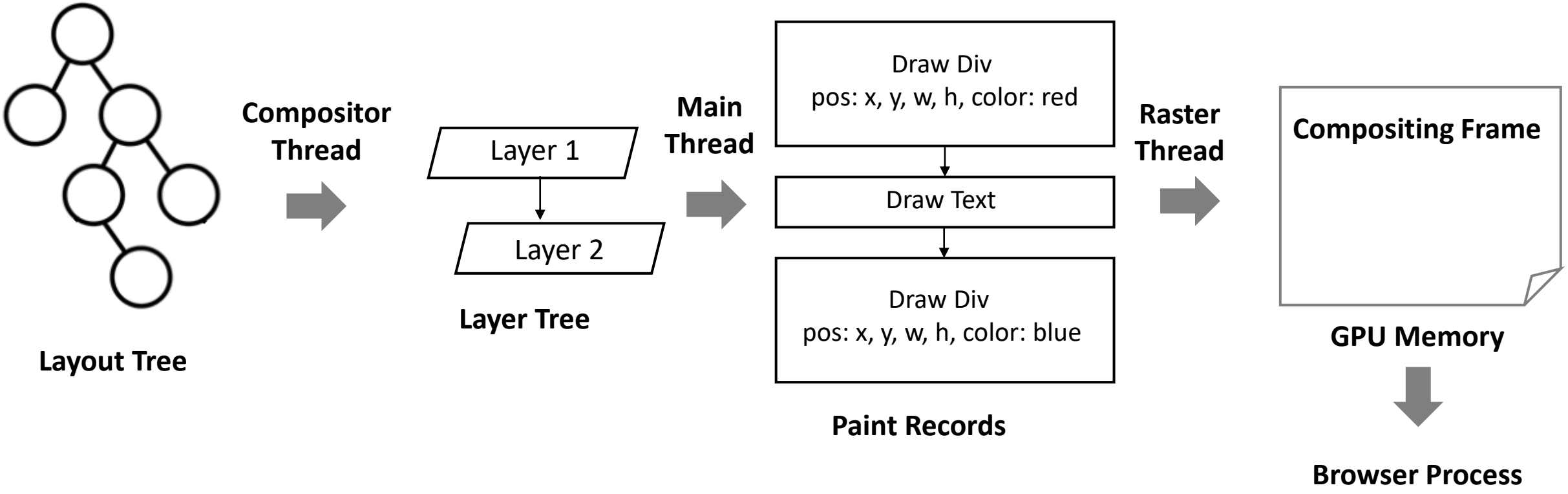
Compositing

After generating paint records, we will turn it into pixels on the screen, this is called **Rasterizing**.

We can raster only the parts inside of the viewport and when a user scrolls the page, fill in the missing parts by rastering more.

Compositing is a technique to separate parts of a page into layers, rasterize them separately. Composite for a page happens in a separate thread called **Compositor thread**. In order to find out which elements need to be in which layers, the main thread walks through the **Layout Tree** to create the **Layer Tree.**

Compositing and Rastering



Raster thread

Once the **Layer Tree** is created and **Paint records** are determined, the **Main thread** commits that information to the **Compositor thread**. The Compositor thread then **Rasterizes** each layer.

A layer could be large like the entire length of a page, so the Compositor thread divides them into tiles and sends each tile off to Raster threads. **Raster threads** rasterize each tile and store them in **GPU memory**. Once tiles are rastered, Compositor thread creates a **Frame**.

The **Frame** is then submitted to the **Browser process** via IPC. These compositor frames are sent to the **GPU** to display it on a screen. If a scroll event comes in, Compositor thread creates another compositor frame to be sent to the GPU.

Input Events

Input Events mean any gesture from the user. Mouse wheel scroll is an input event and touch or mouse over is also an input event.

When user gesture like touch on a screen occurs, the **Browser process** is the one that receives the gesture at first. Then it sends the event type (like touchstart) and its coordinates to the **Renderer process** which handles the event appropriately by finding the event target and running event listeners that are attached.

Updating the DOM is costly

The most important thing to understand is that every time you make a change to the DOM, the rendering pipeline will be executed. Many **Computed Properties** need to be re-calculated.

The Main thread is shared between your JS application and the rendering pipeline. Blocking the thread leads to creating less frames.

Frameworks like Angular and React, do not allow developers to access the DOM directly. **Instead, when changes are made to the App state, the framework patches the DOM efficiently for us.**

Be Kind to the Browsers

Remember that JavaScript is the Main thread's job and you do not want to block the main thread.