Inside look at modern web browser (part 2)

Published on Friday, September 7, 2018 • Updated on Friday, September 21, 2018

Mariko Kosaka
Mariko is a drawsplainer

Table of contents

What happens in navigation

This is part 2 of a 4 part blog series looking at the inner workings of Chrome. In the previous post, we looked at how different processes and threads handle different parts of a browser. In this post, we dig deeper into how each process and thread communicate in order to display a website.

Let's look at a simple use case of web browsing: you type a URL into a browser, then the browser fetches data from the internet and displays a page. In this post, we'll focus on the part where a user requests a site and the browser prepares to render a page - also known as a navigation.

It starts with a browser process

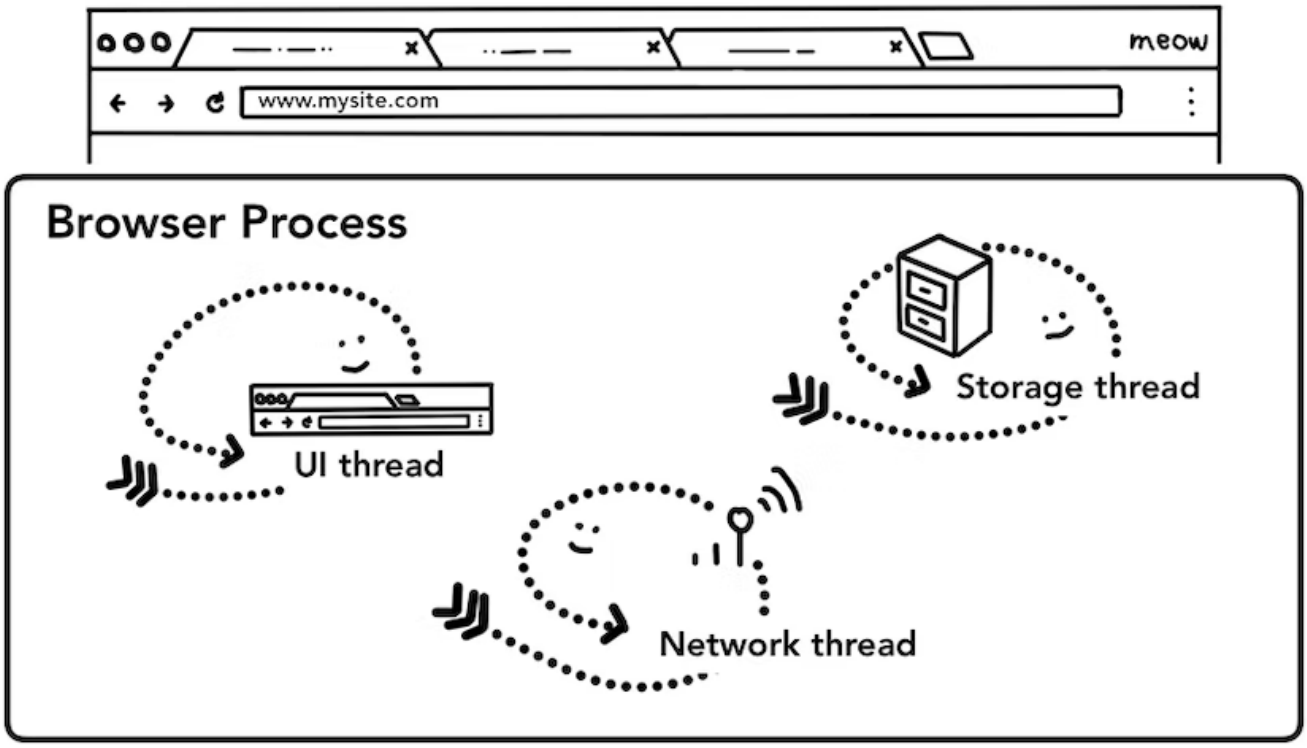

Figure 1: Browser UI at the top, diagram of the browser process with UI, network, and storage thread inside at the bottom

As we covered in part 1: CPU, GPU, Memory, and multi-process architecture, everything outside of a tab is handled by the browser process. The browser process has threads like the UI thread which draws buttons and input fields of the browser, the network thread which deals with network stack to receive data from the internet, the storage thread that controls access to the files and more. When you type a URL into the address bar, your input is handled by browser process's UI thread.

A simple navigation

Step 1: Handling input

When a user starts to type into the address bar, the first thing UI thread asks is "Is this a search query or URL?". In Chrome, the address bar is also a search input field, so the UI thread needs to parse and decide whether to send you to a search engine, or to the site you requested.

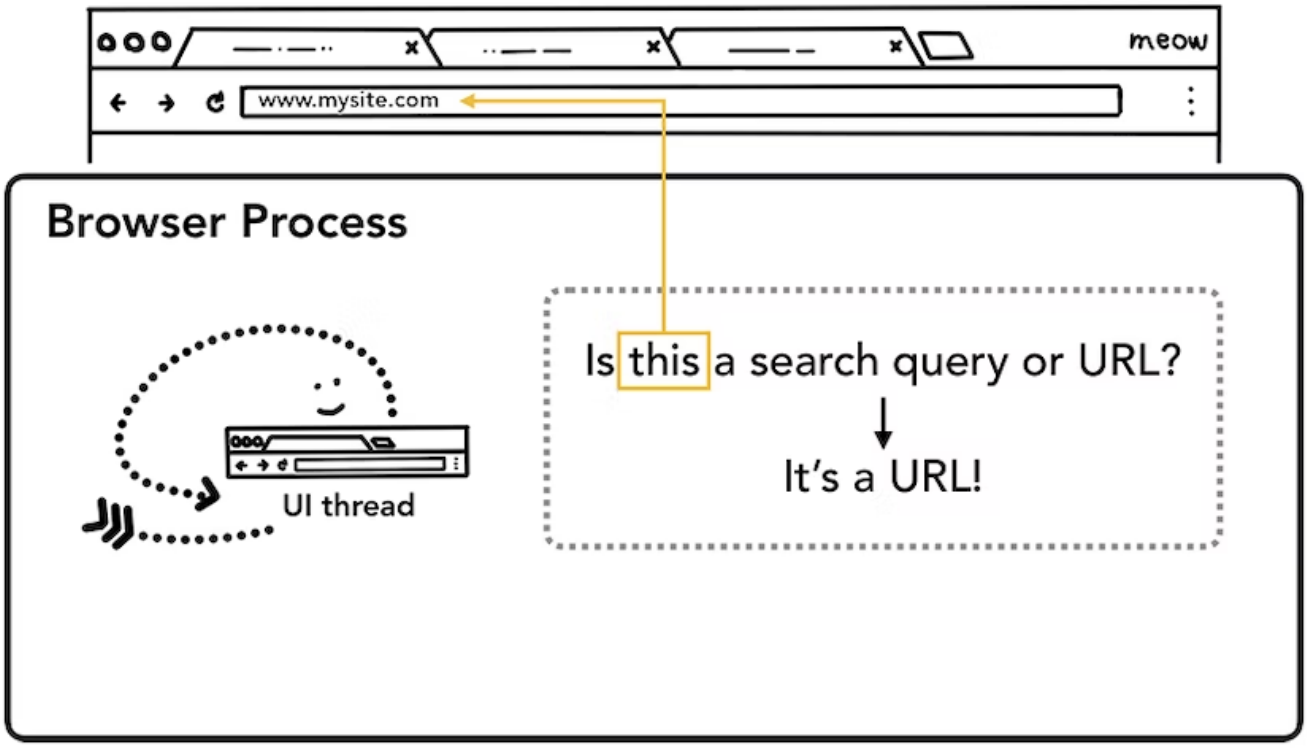

Figure 1: UI Thread asking if the input is a search query or a URL

Step 2: Start navigation

When a user hits enter, the UI thread initiates a network call to get site content. Loading spinner is displayed on the corner of a tab, and the network thread goes through appropriate protocols like DNS lookup and establishing TLS Connection for the request.

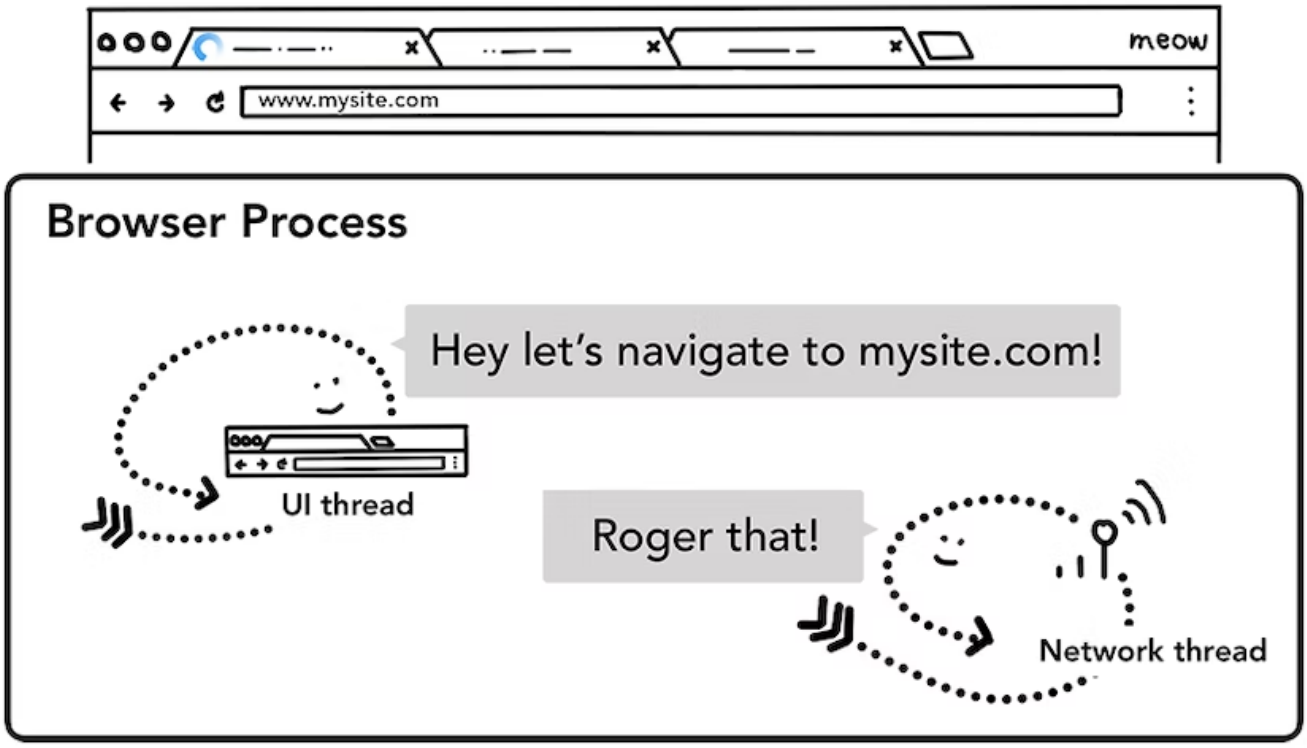

Figure 2: the UI thread talking to the network thread to navigate to mysite.com

At this point, the network thread may receive a server redirect header like HTTP 301. In that case, the network thread communicates with UI thread that the server is requesting redirect. Then, another URL request will be initiated.

Step 3: Read response

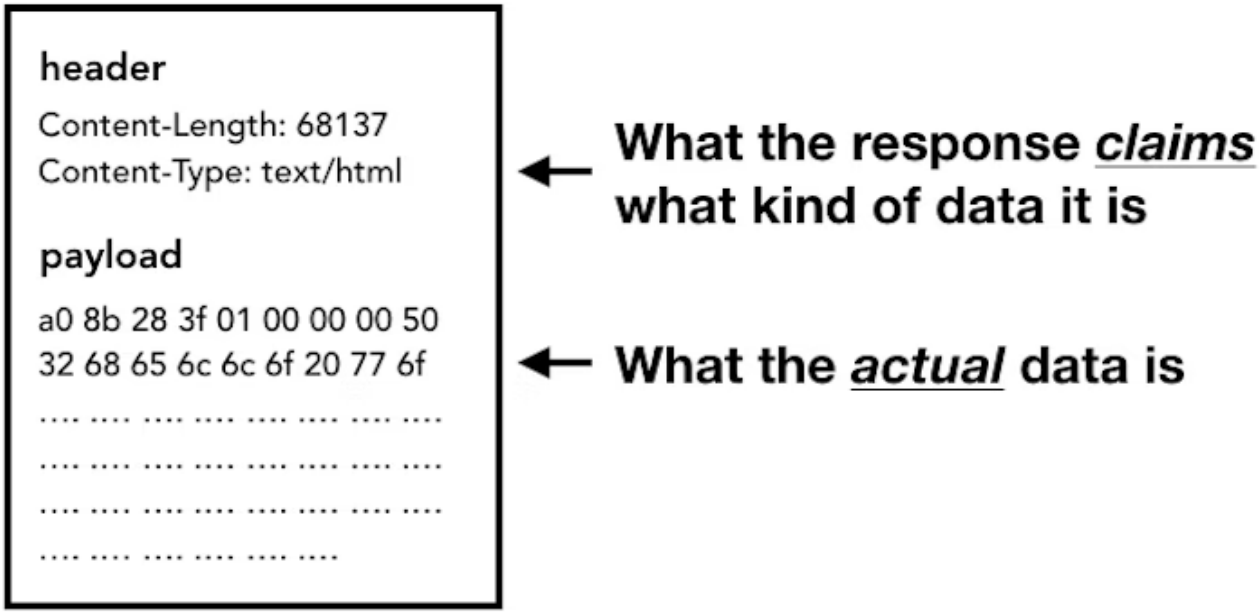

Figure 3: response header which contains Content-Type and payload which is the actual data

Once the response body (payload) starts to come in, the network thread looks at the first few bytes of the stream if necessary. The response's Content-Type header should say what type of data it is, but since it may be missing or wrong, MIME Type sniffing is done here. This is a "tricky business" as commented in the source code. You can read the comment to see how different browsers treat content-type/payload pairs.

If the response is an HTML file, then the next step would be to pass the data to the 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.

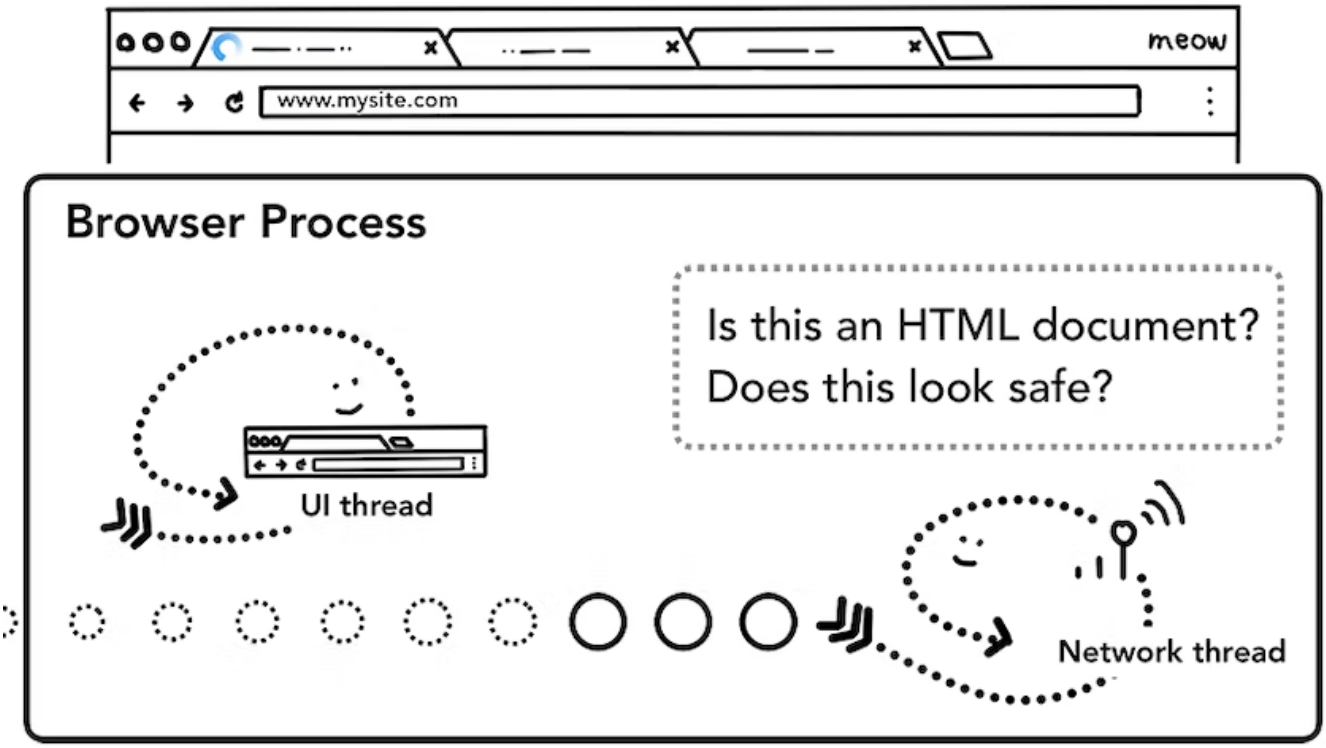

Figure 4: Network thread asking if response data is HTML from a safe site

This is also where the SafeBrowsing check happens. If the domain and the response data seems to match a known malicious site, then the network thread alerts to display a warning page. Additionally, **C**ross **O**rigin **R**ead **B**locking (**CORB**) check happens in order to make sure sensitive cross-site data does not make it to the renderer process.

Step 4: Find a renderer process

Once all of the checks are done and Network thread is confident that browser should navigate to the requested site, the Network thread tells UI thread that the data is ready. UI

thread then finds a renderer process to carry on rendering of the web page.

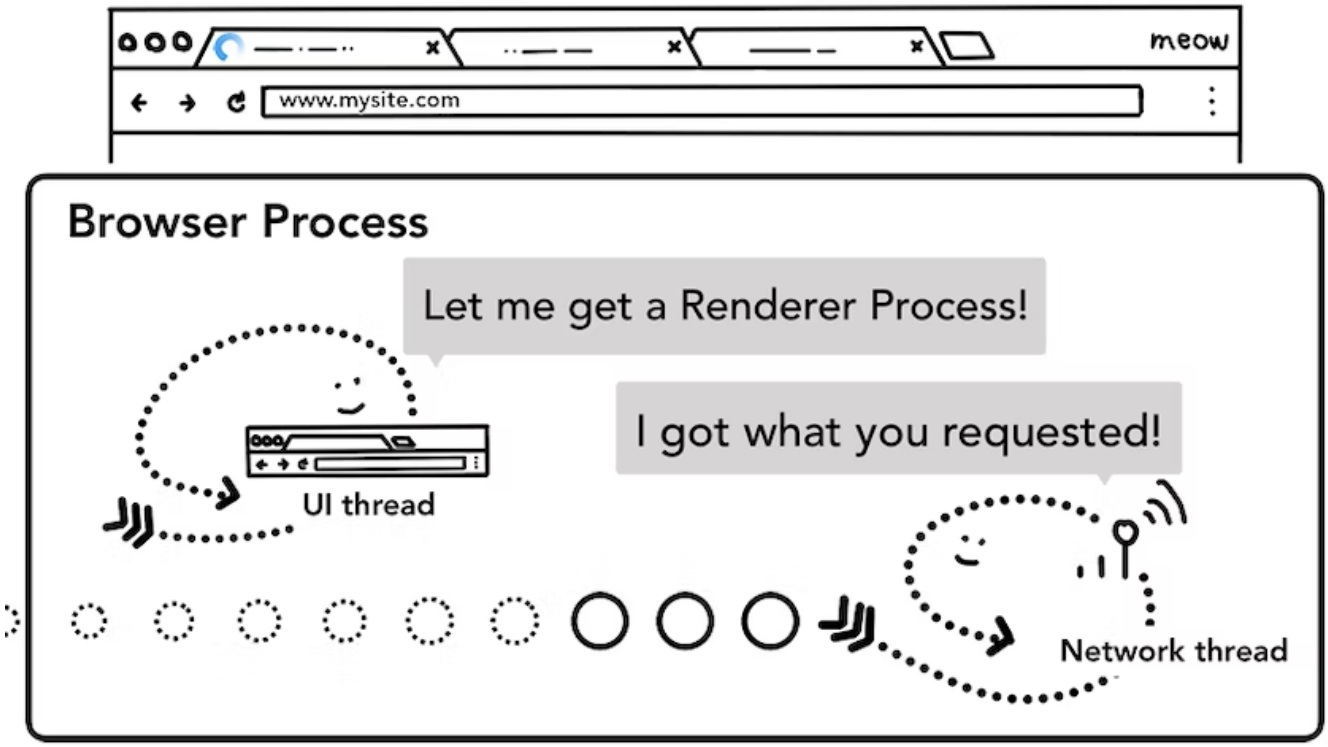

Figure 5: Network thread telling UI thread to find Renderer Process

Since the network request could take several hundred milliseconds to get a response back, an optimization to speed up this process is applied. When the UI thread is sending a URL request to the network thread at step 2, it already knows which site they are navigating to. The UI thread tries to proactively find or start a renderer process in parallel to the network request. This way, if all goes as expected, a renderer process is already in standby position when the network thread received data. This standby process might not get used if the navigation redirects cross-site, in which case a different process might be needed.

Step 5: Commit navigation

Now that the data and the renderer process is ready, an IPC is sent from the browser process to the renderer process 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 the document loading phase begins.

At this point, address bar is updated and the security indicator and site settings UI reflects the site information of the new page. The session history for the tab will be updated so back/forward buttons will step through the site that was just navigated to. To facilitate tab/session restore when you close a tab or window, the session history is stored on disk.

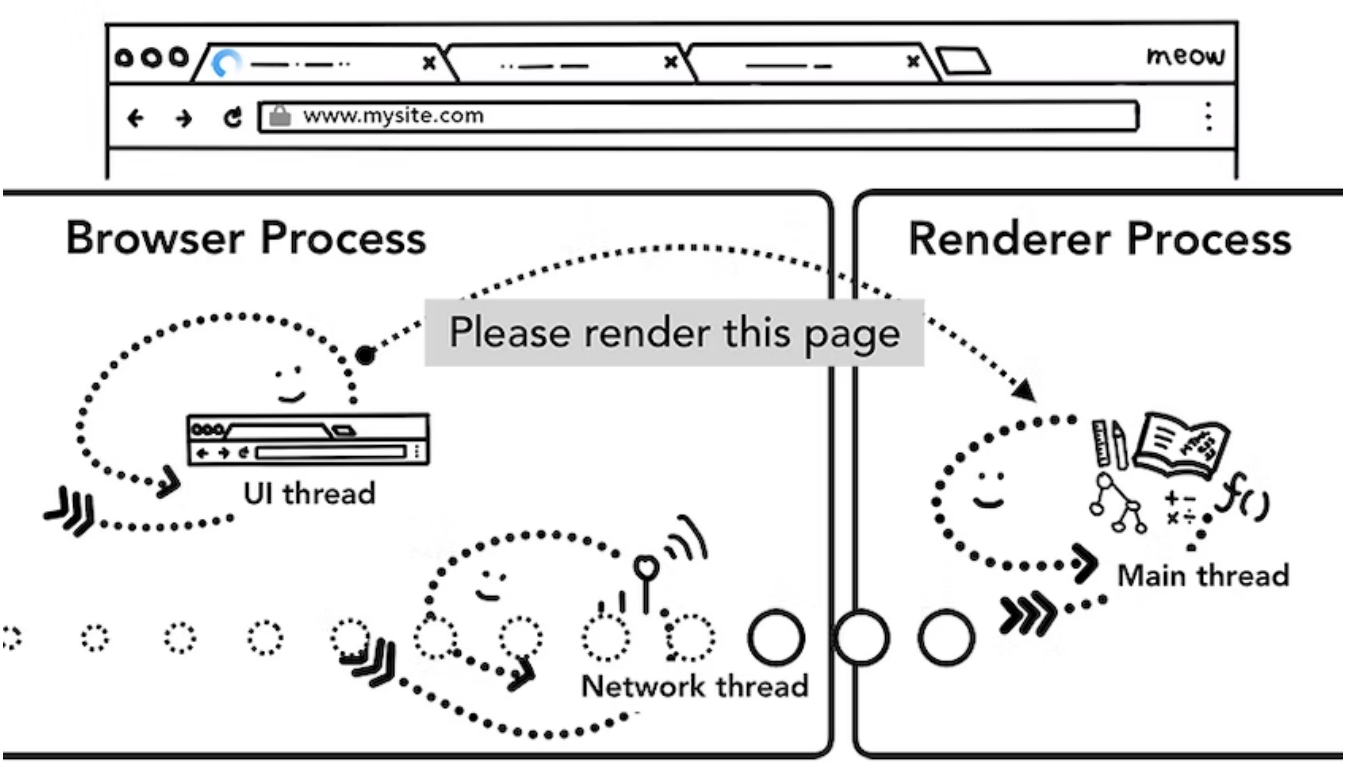

Figure 6: IPC between the browser and the renderer processes, requesting to render the page

Extra Step: Initial load complete

Once the navigation is committed, the renderer process carries on loading resources and renders the page. We will go over the details of what happens at this stage in the next post. Once the renderer process "finishes" rendering, it sends an IPC back to the browser process (this is after all the `onload` events have fired on all frames in the page and have finished executing). At this point, the UI thread stops the loading spinner on the tab.

I say "finishes", because client side JavaScript could still load additional resources and render new views after this point.

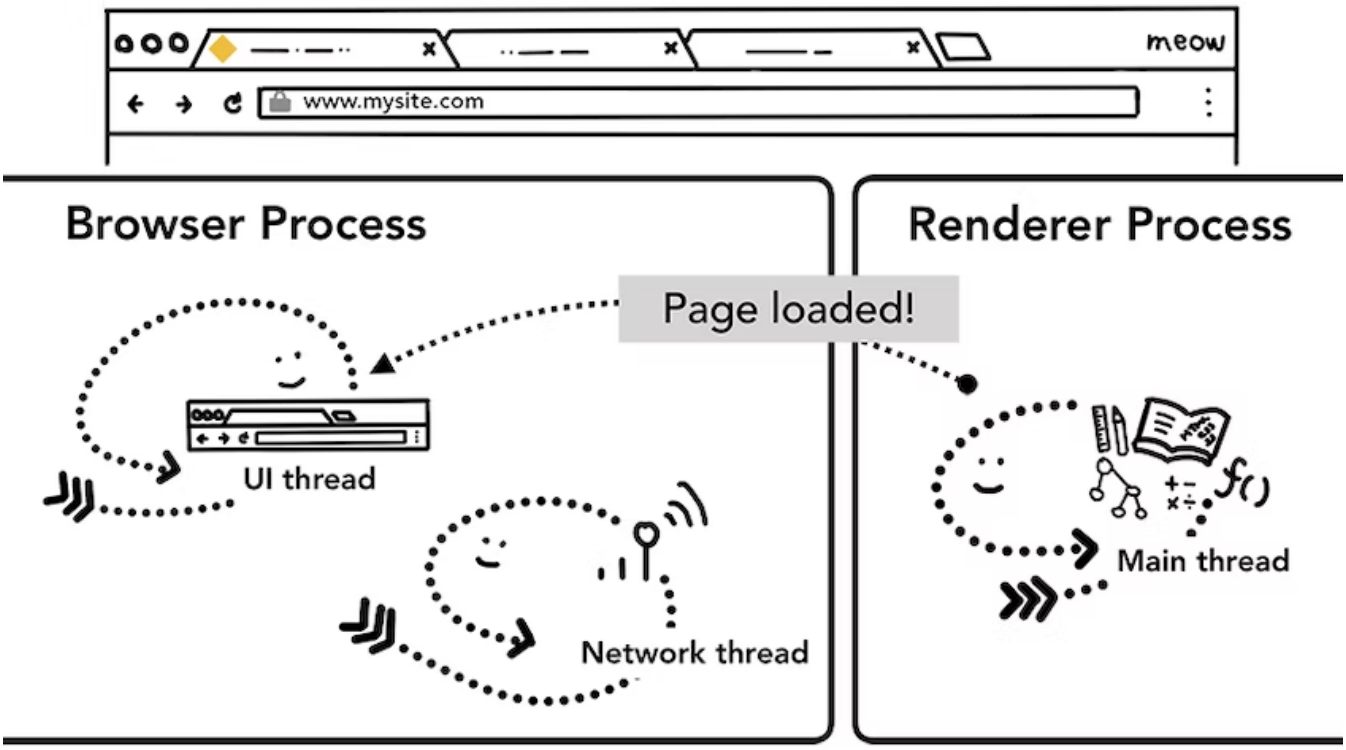

Figure 7: IPC from the renderer to the browser process to notify the page has "loaded"

Navigating to a different site

The simple navigation was complete! But what happens if a user puts different URL to address bar again? Well, the browser process goes through the same steps to navigate to the different site. But before it can do that, it needs to check with the currently rendered site if they care about `beforeunload` event.

`beforeunload` can create "Leave this site?" alert when you try to navigate away or close the tab. Everything inside of a tab including your JavaScript code is handled by the renderer process, so the browser process has to check with current renderer process when new navigation request comes in.

Caution

Do not add unconditional `beforeunload` handlers. It creates more latency because the handler needs to be executed before the navigation can even be started. This event handler should be added only when needed, for example if users need to be warned that they might lose data they've entered on the page.

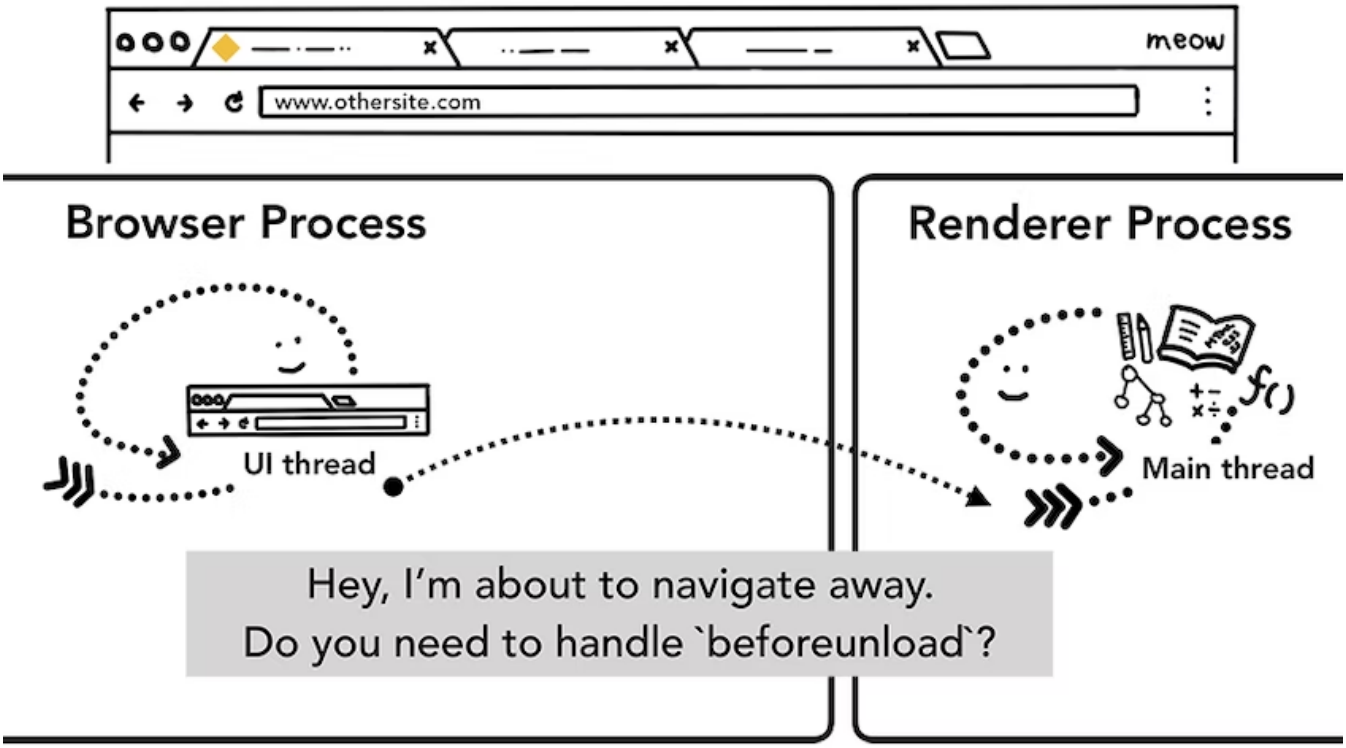

Figure 8: IPC from the browser process to a renderer process telling it that it's about to navigate to a different site

If the navigation was initiated from the renderer process (like user clicked on a link or client-side JavaScript has run `window.location = "https://newsite.com"`) the renderer process first checks `beforeunload` handlers. Then, it goes through the same process as browser process initiated navigation. The only difference is that navigation request is kicked off from the renderer process to the browser process.

When the new navigation is made to a different site than currently rendered one, a separate render process is called in to handle the new navigation while current render process is kept around to handle events like `unload`. For more, see an overview of page lifecycle states and how you can hook into events with the Page Lifecycle API.

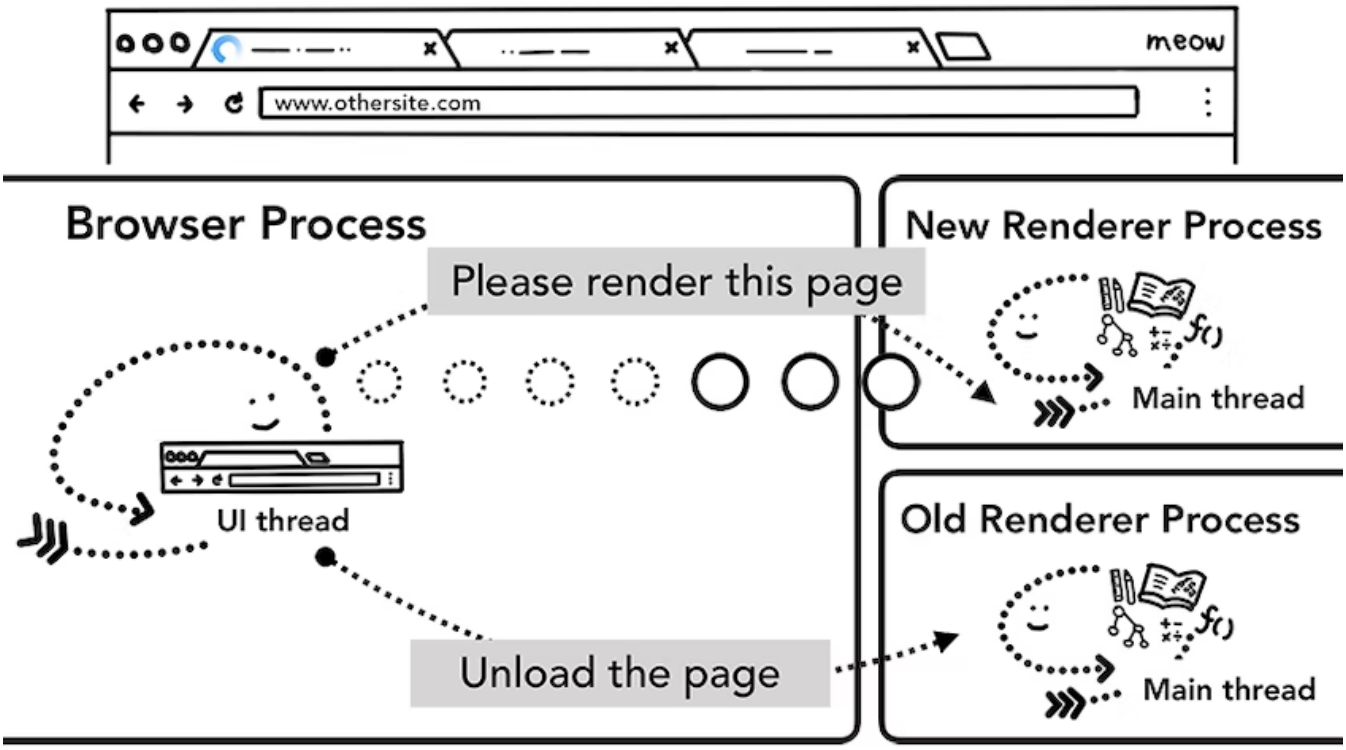

Figure 9: 2 IPCs from a browser process to a new renderer process telling to render the page and telling old renderer process to unload

In case of Service Worker

One recent change to this navigation process is the introduction of service worker. Service worker is a way to write network proxy in your application code; allowing web developers to have more control over what to cache locally and when to get new data from the network. If service worker is set to load the page from the cache, there is no need to request the data from the network.

The important part to remember is that service worker is JavaScript code that runs in a renderer process. But when the navigation request comes in, how does a browser process know the site has a service worker?

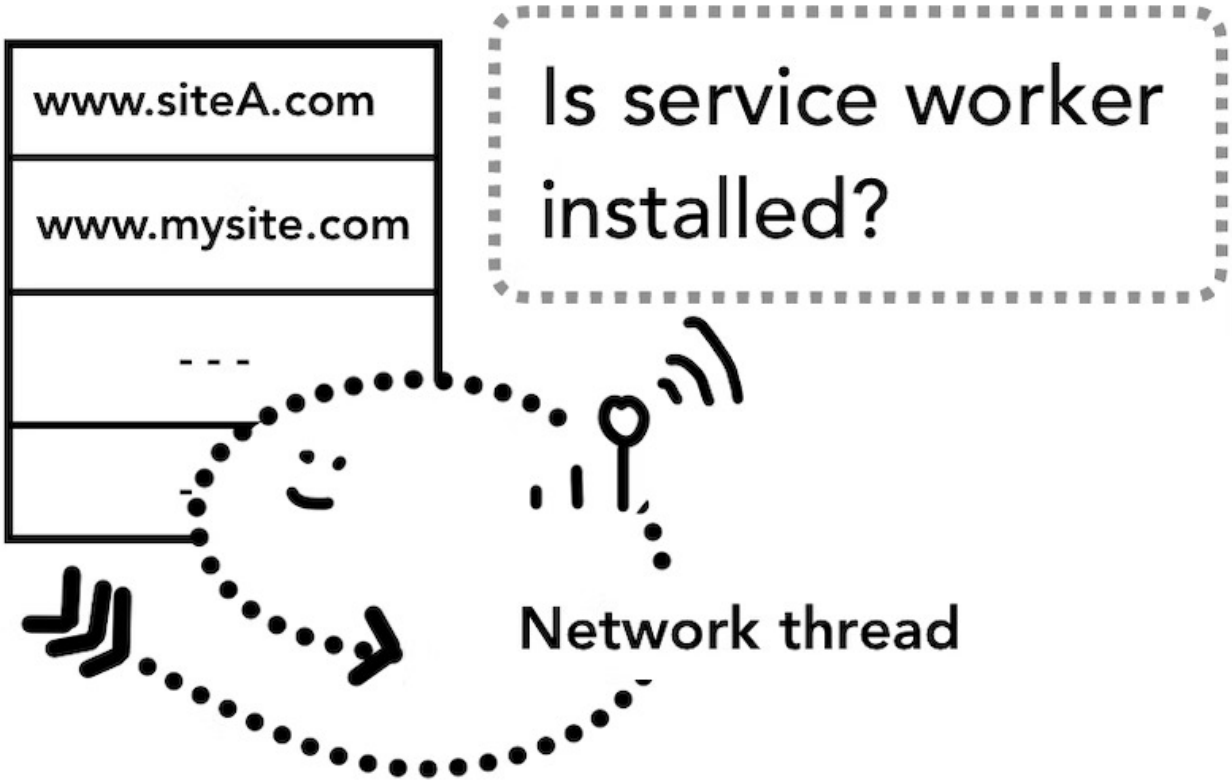

Figure 10: the network thread in the browser process looking up service worker scope

When a service worker is registered, the scope of the service worker is kept as a reference (you can read more about scope in this The Service Worker Lifecycle article). When a navigation happens, network thread checks the domain against registered service worker scopes, if a service worker is registered for that URL, the UI thread finds a renderer process in order to execute the service worker code. The service worker may load data from cache, eliminating the need to request data from the network, or it may request new resources from the network.

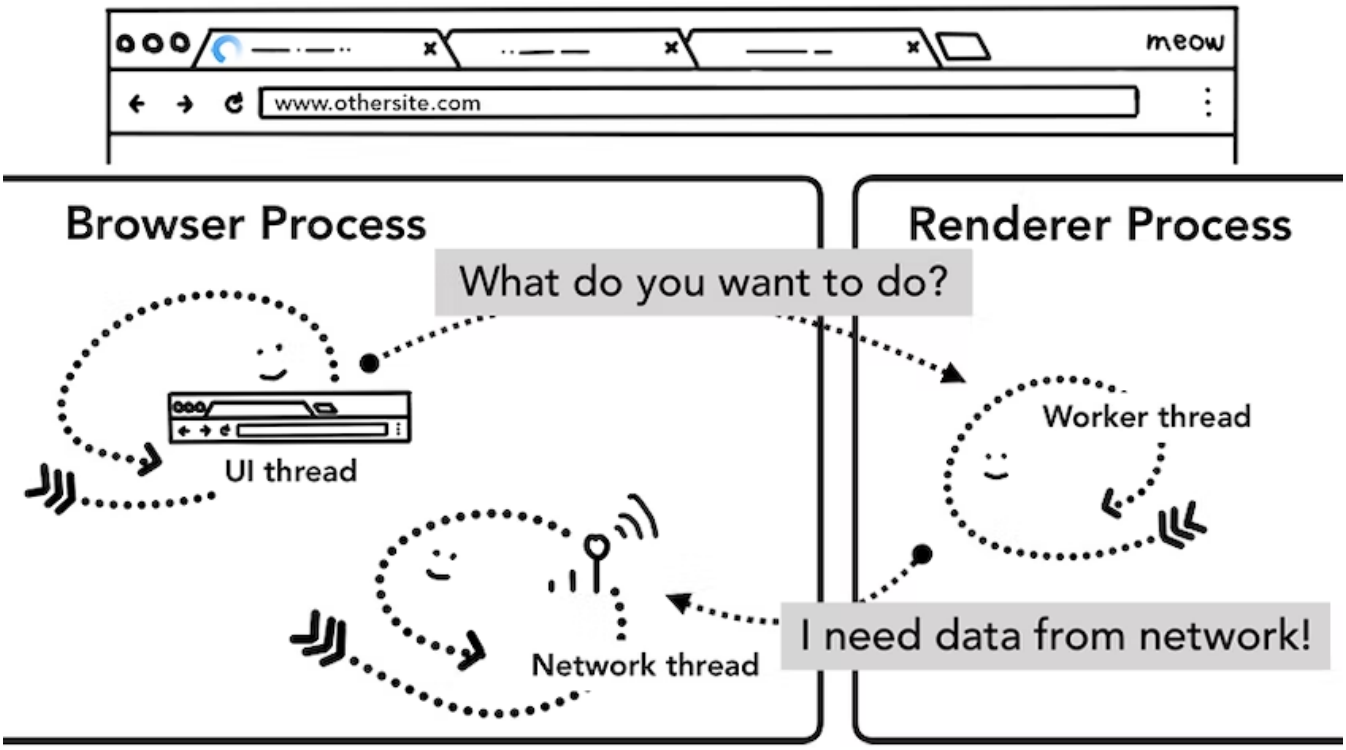

Figure 11: the UI thread in a browser process starting up a renderer process to handle service workers; a worker thread in a renderer process then requests data from the network

Navigation Preload

You can see this round trip between the browser process and renderer process could result in delays if service worker eventually decides to request data from the network. Navigation Preload is a mechanism to speed up this process by loading resources in parallel to service worker startup. It marks these requests with a header, allowing servers to decide to send different content for these requests; for example, just updated data instead of a full document.

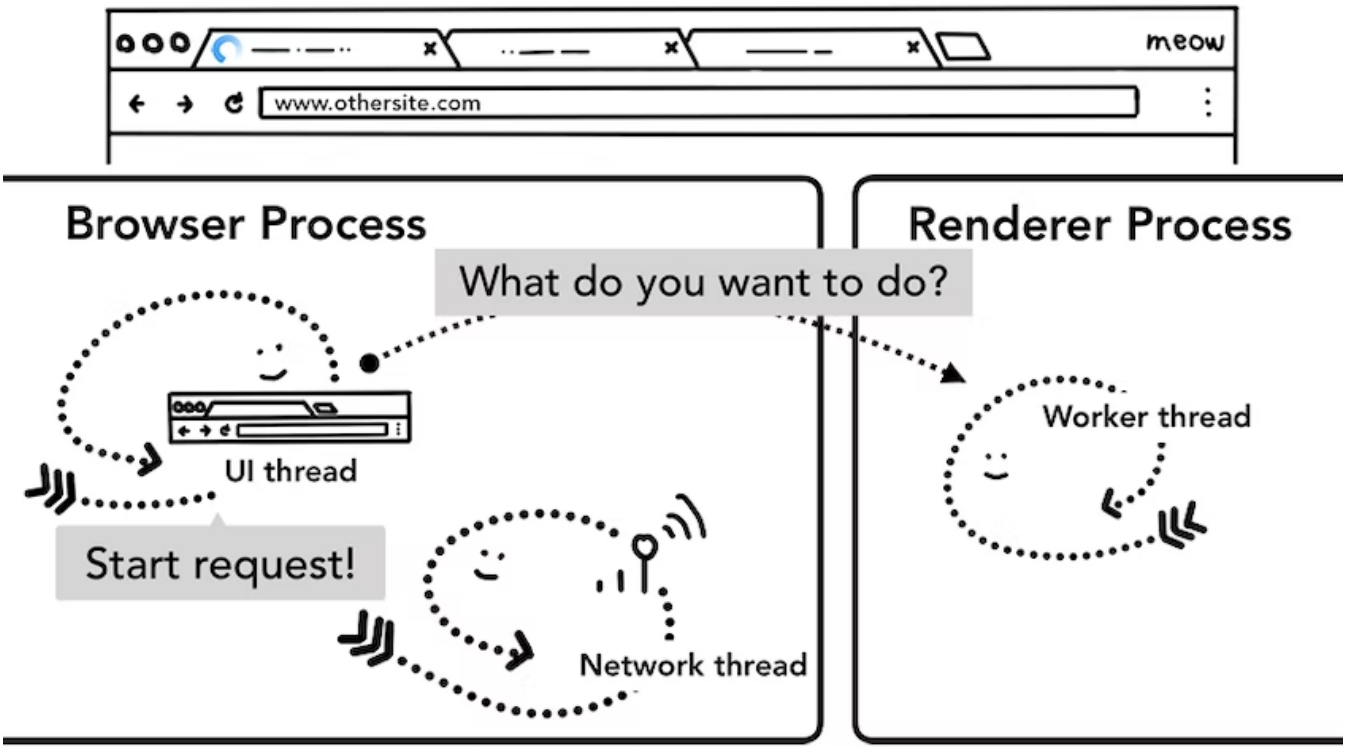

Figure 12: the UI thread in a browser process starting up a renderer process to handle service worker while kicking off network request in parallel

Wrap-up

In this post, we looked at what happens during a navigation and how your web application code such as response headers and client-side JavaScript interact with the browser. Knowing the steps browser goes through to get data from the network makes it easier to understand why APIs like navigation preload were developed. In the next post, we'll dive into how the browser evaluates our HTML/CSS/JavaScript to render pages.

Did you enjoy the post? If you have any questions or suggestions for the future post, I'd love to hear from you in the comment section below or @kosamari on Twitter.

Next: Inner workings of a Renderer Process

Last updated: Friday, September 21, 2018 Improve article

Follow us

Follow us

Contribute

File a bug

View source

Related content

web.dev

Case studies

Podcasts

Connect

Twitter

YouTube

GitHub

Google Developers

Chrome Firebase All products Privacy Terms

Choose language

ENGLISH (en)