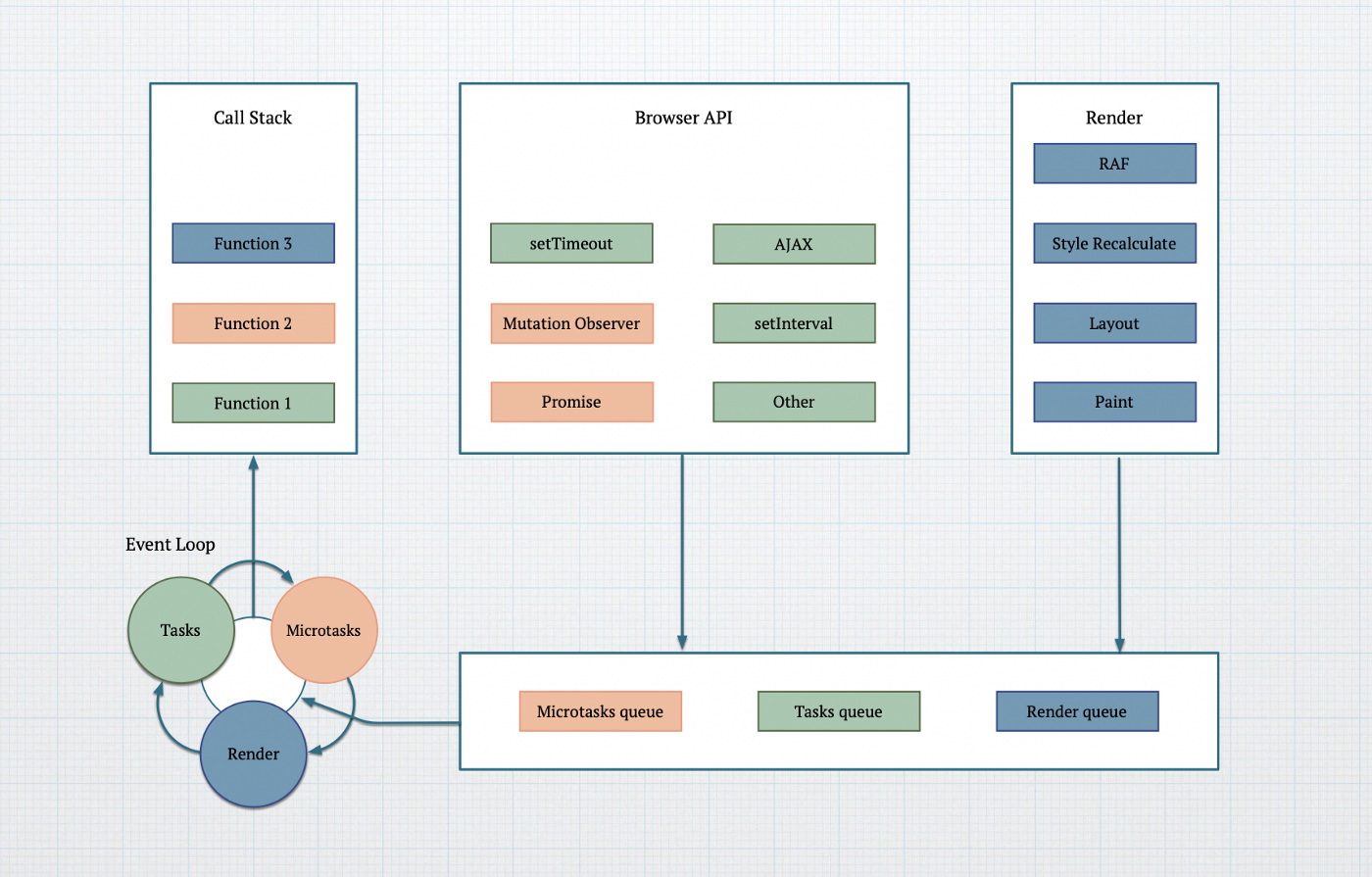
**Main Thread Loop**



From <https://docs.google.com/document/d/1GSHp-KoP4M1oq-vQaw8W8g93lm_k1zsykP_GAzRrbYo/edit#heading=h.fkrfg3qeppr8>

V8 is a main thread task.

## **Executor, Microtask**

Executor (From <https://javascript.info/promise-basics>:

“The function passed to the new Promise is called the executor. When a new Promise is created, the executor runs automatically. ”

Executor 1:

let promise = new Promise(function(resolve, reject) {

// **Executor**

});

const ret1 = promise.then((value) => {

// **Microtask**

console.log("Microtask1 # posted by Resolve: "+value);

return true;

});

Executor 2:

async function asyncCall() {

// **Executor**

await something();

// **Microtask**

}

{

await asyncCall();

}

## **Queue a microtask, execute a microtask**

**Who queues a microTask?** Other thread(Network), Main Thread(Call Promise.resolve.)

Main thread queues a microtask: time\_promisethen.js

|  |
| --- |
| const promise1 = new Promise((resolve, reject) => {  console.log("Main # Post MicroTask1 # In Promise new ");  // Resolve posts a microtask  resolve('Success From Promise!'); }); console.log('Main # after new promise'); const ret1 = promise1.then((value) => {  // Microtask  console.log("Microtask1 # posted by Resolve: "+value);  return true; }); |

Other thread queues a microTask: time\_promisethen2await\_fetch.js

|  |
| --- |
| async function fetchData() {  const response = await fetch('./index.html');  // 1. Main thread post task to network (Done by fetch API);   // 2. When the network thread is done, post microtask into the main thread.  // Microtask  console.log("Microtask1 # posted by Chromium network");  return true; } |

**Who executes a microTask?** Main Thread (We don't talk about worker thread here).

Demo:

### **Demo # promise then**

[Promise Then](http://127.0.0.1:5500/time_promisethen.html)

[Promise Then(fetch)](http://127.0.0.1:5500/time_promisethen_fetch.html)

[Promise Then(fetch) to await](http://127.0.0.1:5500/time_promisethen2await_fetch.html)

## **Three RULES**

From <https://chromium.googlesource.com/chromium/src/+/HEAD/third_party/blink/renderer/platform/scheduler/TaskSchedulingInBlink.md> :

**#RULE 1** “At the moment Blink Scheduler treats tasks as an atomic unit — **if a task has started, it can’t be interrupted until it completes**. The scheduler can only choose a new task to run from the eligible tasks or can elect not to run any task at all.

”

“A [microtask](https://html.spec.whatwg.org/C#microtask) (which is a synonym of a task that happens to belong to a microtask queue) is run when

**#RULE 2**  (1) JavaScript is finished to run (JavaScript is also a Task, so JavaScript finished run equals “running task comes to an end”)

**#RULE 3** (2) the currently running task comes to an end. You can think of it as a hook that's invoked on a script or task completion.

## **Top level await or not**

|  |
| --- |
| async function asyncCall(tag) {  console.log(tag + ' asyncCall start');  const start = performance.now();  const result = await foo();  const end = performance.now();  console.log(tag + 'asyncCall time = ' + (end - start)); }  // Option 1: Top level with await  // time\_topawait.js (async function() {  // Question: when will the below console.log run?  // console.log();  await asyncCall();  loop(); })();  // Option 2: Top level without await (e2e works in this method)  // runs in the V8 task.  // time\_notask.js asyncCall(); // Without await!  loop(); |

According to RULE 1: Once a task is scheduled, it will run until it ends.

Toplevel async workflow:

|  |
| --- |
| async function asyncCall(tag) {  **// #1 V8Task: Get start time**  const start = performance.now();  // #2 V8Task: Insert microtask.  // #2 V8Task comes to an end. Schedule in microtask1.   const result = await foo();  const end = performance.now();  **// #3 Microtask1 execute. Get end time**  // #3 Microtask1 end, insert microtask 2. Schedule in microtask 2.  }  // Option 1: Top level async  // time\_topawait.js (async function() {  // Question: when will the below console.log run?  // console.log();  await asyncCall();  // #4 Microtask execute 2.  loop(); })(); |

Toplevel async without await workflow:

|  |
| --- |
| async function asyncCall(tag) {  **// #1 V8Task: Get start time**  const start = performance.now();  // #2 V8Task: Insert microtask1.   const result = await foo();  const end = performance.now();  **// #4 Microtask1 executes. Get end time**  console.log(tag + 'asyncCall time = ' + (end - start)); }  // Option 2: Top level async without await (e2e works in this method) // runs in the V8 task.  // time\_notask.js asyncCall(); // Without await!  **// #3 V8Task loop**  loop();  // #3 V8Task comes to end, scheduled in microtask1. goto #4 |

Demo:

### **Demo # no task**

[no Task](http://127.0.0.1:5500/time_notask.html) 1st is incorrect. 2nd(Comment out default) is correct

[top await](http://127.0.0.1:5500/time_topawait.html) 1st is correct

## **Measure execution time of settimeout**

### **Demo # settimeout**

When the loop runs too long, the measured time of settimeout is incorrect.

[Loop 1000000000.](http://10.239.47.14:5501/time_settimeout.html?loop=1) Loop time < timeout time, asyncCall is correct

[Loop 1000000000\*2.](http://10.239.47.14:5501/time_settimeout.html?loop=2) Loop time > timeout time, asyncCall is incorrect

[Loop 1000000000\*2, topawait.](http://10.239.47.14:5501/time_settimeout_topawait.html?loop=2) Loop time > timeout time, asyncCall is correct in topawait

[Loop 1000000000, no loop after call async timeout](http://10.239.47.14:5501/time_settimeout.html?loop=1&control=1)

## **Series or parallel**

Two goal for async:

1. Improve main thread(Renderer + V8) responessive.
2. **Async doesn’t guarantee parallel, but it will benefit from parallel.**

Demo:

### **Demo # series or parallel**

Async doesn't guarantee thread parallel. But it will benefit from parallel.

[series await settimeout](http://127.0.0.1:5500/time_seriesparallel.html?control=0)

[parallel await settimeout](http://127.0.0.1:5500/time_seriesparallel.html?control=1) async doesn't always mean real Parallel! This is fake parallel

[series await nop loop](http://127.0.0.1:5500/time_seriesparallel3.html?control=0)

[parallel await nop loops](http://127.0.0.1:5500/time_seriesparallel3.html?control=1) async doesn't always mean real Parallel! This is not parallel

[series await fetch](http://127.0.0.1:5500/time_seriesparallel_fetch.html?control=0)

[parallel await fetch](http://127.0.0.1:5500/time_seriesparallel_fetch.html?control=1) async doesn't always mean real Parallel! This is possible parallel