Async JS

Created	@May 4, 2023 8:25 PM
Class	
Type	
Materials	
Reviewed	☐

What is a thread ?

All of us know that if we make a program like `test.js` and run it in our computer it becomes a process i.e. program in a running state is a process.

Let’s take a simple example:

If a man takes `X` days to complete a job, how many days will it take for `Y` men to complete the same job ?

`Ans`: X / Y days because we are assuming all of them will work parallel and divide the task among themselves.

Similar concepts is applicable for computers also. Computer have got different mechanisms to achieve parallel computations.

```
Threads are execution entity, very much like process, but they are
extensively very light weight process.
```

Let’s write a very simple program

```
i = 0;
sum = 0;
while(i <= 100000000) {
  sum += i;
  i++;
}
```

Now if we run the above code on a machine by let's say saving it as a C++ code or JS code, then what will happen is, this program will be loaded in the memory and become a process and will be running on any one of the cores of our processor. Assuming we have an Octa core processor, we will be having 7 other cores sitting idle.

So what the above program is actually doing ? It is simply adding the first 100000000 naturals numbers.

How about we try to do some parallelisation, i.e. how about we try to divide our task of summing up first 100000000 natural numbers between all of our 8 processor cores.

Then there will be 8 process running in different cores which will sum up the things in parallel fashion.

This kind of a model is called as `Processor model`.

Now there is something called as `Thread Mode`.

In thread mode what we will do is we create 1 process with 8 threads, and each thread does 1/8th of the given work. It will be equivalently fast as running it on 8 different processors.

One interesting fact about a thread is that, all the thread maintains their own call stack (and it's obvious right ? They are just light weight process).

You might think that, won't it be slow to run 8 threads as compared to running process on 8 different cores ?

It won't be that slow, as threads are very light weight on memory, the context switching is very fast compared to normal process, and threads can also communicate within themselves.

Real life example:

On earlier mobile phone like android, the main application that you used to see generally used to run on something called as Main thread. Now the app has to download some data, then we should not download the data on the main thread. For this we can make another thread and download data over that thread leaving the main thread unblocked.

Characteristics Of JS:

- Javascript is single threaded (then how does it manage time consuming tasks, like downloading some data ????)
- JS supports Synchronous execution of code (JS will executing everything one after another i.e. if you have some time consuming task, JS will wait for it to complete and then only move forward)

```
function blockingTimeConsumingCode() {
    for(let i = 0; i < 10000000000; i++) {}
}
console.log("Start");
blockingTimeConsumingCode();
console.log("End");
```

In this piece of code, JS is running everything one after another, the for loop os a blocking piece of code, so JS has to wait for it to complete before it can move forward. That means this time consuming for loop is not getting executed in parallel fashion, it is blocking our main thread.

`Note`: The above statement of synchronous execution is only applicable for those piece of code which is native and known to JS for ex: while loop, or for loop etc.

Now you must be thinking, is there anything that JS doesn't know natively ? Yes, for example, `setTimeout` function you can execute in your browser's JS console, but this function is no where mentioned in the official JS docs. That means, official JS is not shipped with this function. JS is somehow able to execute it, but it is not native to JS.

So the above blocking behaviour or synchronous behaviours will not be shown for non native features.

```
console.log("Start");
setTimeout(function() {
    console.log("Completed timer");
}, 10000);
console.log("End");
```

In this piece of code, JS doesn't wait for the timer to get completed in 10 sec, and instead moves forward, giving you an output like

```
Start
End
Completed timer
```

Now you should be getting two questions in your mind i.e.

- if setTimeout is not native and known to JS, how come it is able to execute it ?
- And how does JS handles execution of these non native features? Doesn't it become unpredictable ?

Runtime Environment

In order to understand how exactly JS is running non native functions, we need to first of all understand from where these are coming up ?

```
A runtime environment, is where your code/program will be executed and
while execution this environment is going to provide multiple different aids
in order to add more functionalities to our code. This runtime environment
understands how the code should be run and what capabilities might be useful
for the program during runtime.
```

For example: We can run our JS code in a browser, here browser is a Runtime environment. Browser provides additional capabilities to JS in order to run in a efficient way and be more productive. For ex: Browser provides JS functionalities like

- How to run a timer ? So functionalities like `setTimeout` or `setInterval` these are actually provided to JS through the browser runtime.
- Browser also gives JS a functionality to download some data over the network, using the `fetch` or `XMLHttpRequest` functions.
- Browser also provides capabilities using which JS can interact with the DOM or HTML.

Similar to browser we have other runtime environments as well where we can run JS, for example: NodeJS (NodeJS is neither a language nor a framework, instead it is a Runtime environment).

What capabilities NodeJS provide ?

Using NodeJS We can execute JS out of browser and inside our terminal. So whenever you are running a JS code inside VS-Code, then you are actually running it in the NodeJS runtime.

So here when we actually run the JS file in VS-Code it runs it with the following command:

```
node file.js
```

So here we are invoking the NodeJS env to run our file.

Now as NodeJS is a different runtime, it will be having different behaviour than browser. Because now we are not running JS in browser, so there is no point of giving browser based functionalities like reading html. Instead we give it other functionalities like

- Inside NodeJS , Javascript will be able to read files from our File system.
- It gives access to process level details.

Apart from some new functionalities there can some similar functionalities also.

- NodeJS also provides access to timers. (Read more about timers in Node)

  Note: As I said that functionalities can be similar , they are not necessarily same. Thats why if you run setTimeout in browser it returns you and ID of the timer which is a Number, where as in NodeJS, it returns an object.

Note: Even two different browsers can act as two different runtimes. Although now days most of the functionalities are consistent in major browsers.

So this technically answers that how JS gets these functions like `setTimeout`

How JS handles the execution of Runtime features?

So, till now you must be already aware that when you run a JS code, it becomes a process. For a process we have access to a memory area in which there are multiple components like `Call Stack`

But there are few other components as well that will actually unravel this mystery for us. These components are

- Macrotask Queue (Also called as Callback Queue)
- MicroTask Queue
- Event Loop

Part 1 → Signalling the runtime

As we know that any feature which is native to JS, is executed in a synchronous fashion. At any point of time, when JS encounters any particular feature that is something non native or belongs to the runtime, then JS does one very simple task, it just notifies the runtime that your feature has been called, and then comes back to its normal execution state. JS will immediately comeback to its normal execution code, and behind the scenes the runtime actually starts executing the feature.

```
let x = 0;
setTimeout(function f() {
    console.log("Timer done");
}, 3000);
x += 10;
```

Let's say this is a piece of code, we are running in JS, line number 1 is native to JS, so it will execute it in a sync way and create a variable x. In line number 2, JS detects that we are calling a Runtime feature, so all it will do is go to the runtime, and signal it that it needs to start running a timer of 3s and post that execute the callback `f`

After this JS will not wait there for even a millisecond and immediately comeback to line number 3 and increment the value of x.

Now you might be thinking what if the timer is of 0 millisecond ? Then also JS will not wait for it ?

Answer is no, JS never waits for runtime features, even if there is a timer of 0ms, runtime will take some moment to setup the timer run it and so on, this will take some ms or micro seconds right, so JS never even waits for that much time quantum.

Part 2 → What happens when runtime completes the task ?

When runtime, completes it's task what will happen ? You might be thinking that let's say JS is executing some piece of code, and in between that, runtime completed its execution so will the JS code flow stop in between to execute the runtime's callback ? OR not wait for it again and continue its normal execution ? What will happen ?

Let's take an example:

```
function blockingLoop() {
    for(let i = 0; i < 10000000000; i++) {
        // whenever there is multiple of 500000000, print it
        if(i % 500000000 == 0) console.log(i);
    }
}
let x = 0;
setTimeout(function f() {
    console.log("Timer done");
}, 1000); // start a timer of 1s
blockingLoop();
x += 10;
```

In the above piece of code, I have a blocking loop implemented that will surely take more than 1s to execute, probably it will 5-6 sec. Then we are also triggering a timer of 1s, which will of course run in the runtime environment, and post triggering the timer, we initiate a blocking loop of more than 1s. What will happen when the timer will be done ? Are we going to pause the loop in between and execute the callback `f` or we will not pause ?

```
x += 10;
0
500000000
1000000000
1500000000
2000000000
2500000000
3000000000
3500000000
4000000000
4500000000
5000000000
5500000000
6000000000
6500000000
7000000000
7500000000
8000000000
8500000000
9000000000
9500000000
10
Timer done
```

This will be the output, I was doing the loggings so that we can see even if timer (which is a runtime feature is done) we will never ever halt or pause the execution going on in the main thread.

When the runtime completes it's task, then it cannot halt the current execution of JS code, so what it will do is that whatever is the callback function to be executed after the task is done, it sends that callback function to wait for sometime in `Macrotask Queue or Callback Queue`.

This callback will wait in the queue and never halt the current execution flow. If there are more runtime features, as soon as they are done, their callbacks will also wait in the callback queue.

So when will these waiting callbacks are gonna get executed ?

To handle the execution of these waiting callbacks we have something called as an `Event Loop`.

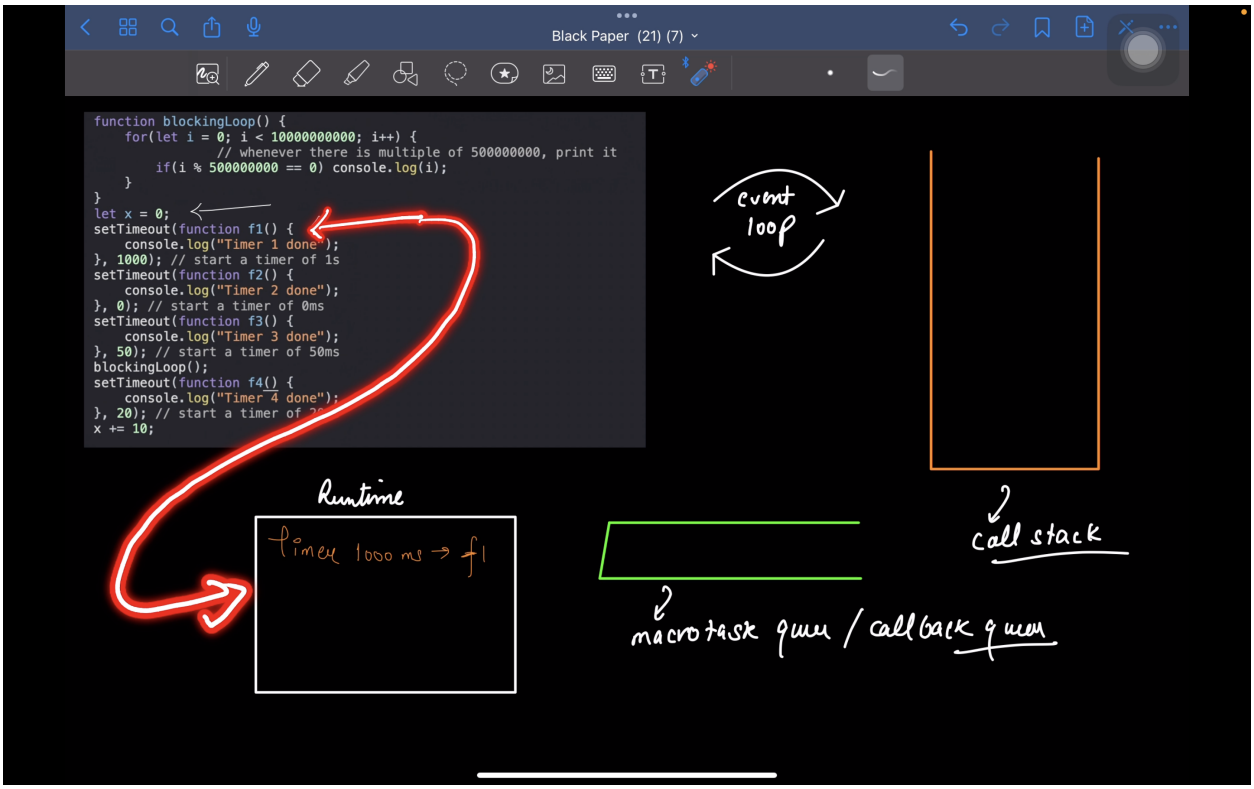
What happens is this event loop is a continuous infinitely running loop (till the time program is executing) which will constantly checks whether there is a function present in the `call stack` which is still executing ? Or if there is some piece of JS code, left in the global scope / area.

Till the time there is something in the call stack or the global area, Event loop will never allow any callback function waiting in the Callback queue to start execution. Once call stack and global area is empty, then Event loop will take the front element waiting in the Callback queue, and push it inside the call stack and start it's execution. Till the time this callback is getting executed, other callbacks have to wait in the queue, because now this function went into the call stack, so it is a JS code getting executed in the main thread.

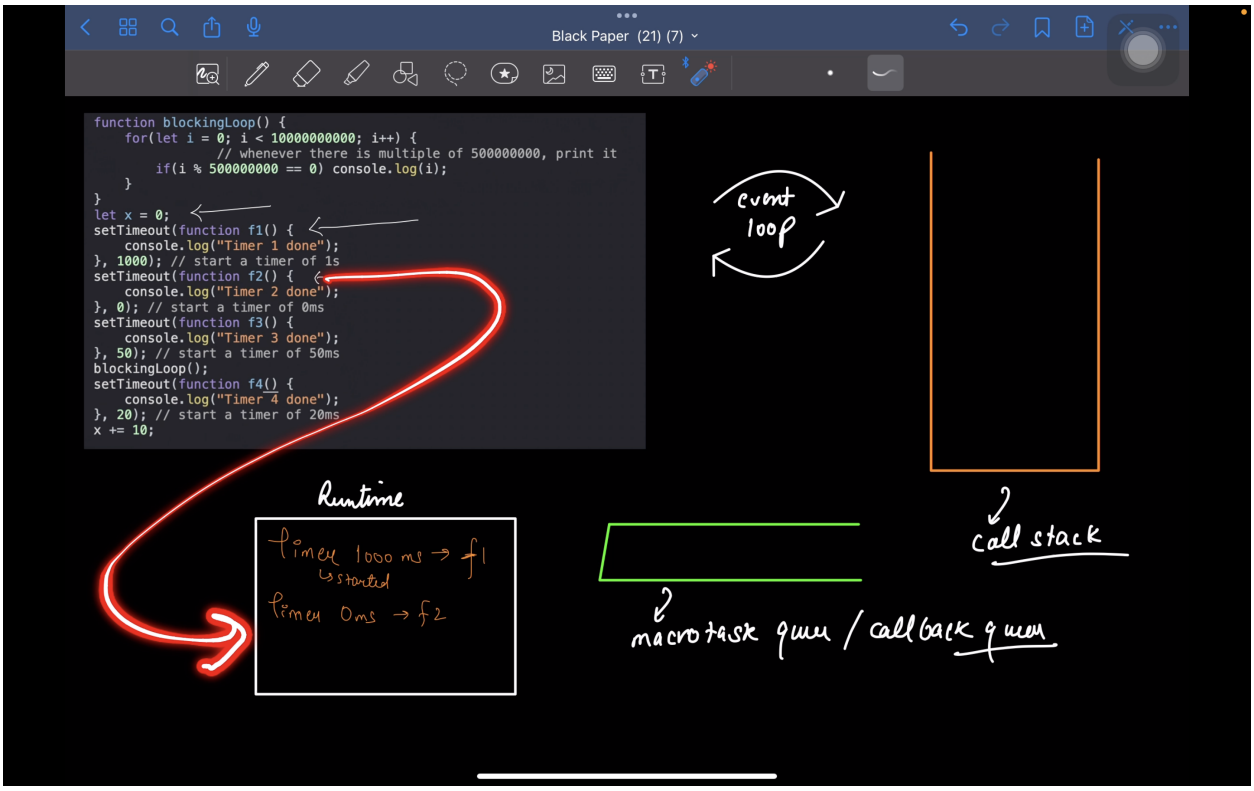
```
function blockingLoop() {
    for(let i = 0; i < 10000000000; i++) {
        // whenever there is multiple of 500000000, print it
        if(i % 500000000 == 0) console.log(i);
    }
}
let x = 0;
setTimeout(function f() {
    console.log("Timer 1 done");
}, 1000); // start a timer of 1s
setTimeout(function f() {
    console.log("Timer 2 done");
```

```
}, 0); // start a timer of 0ms
setTimeout(function f() {
    console.log("Timer 3 done");
}, 50); // start a timer of 50ms
blockingLoop();
setTimeout(function f() {
    console.log("Timer 4 done");
}, 20); // start a timer of 20ms
x += 10;
```

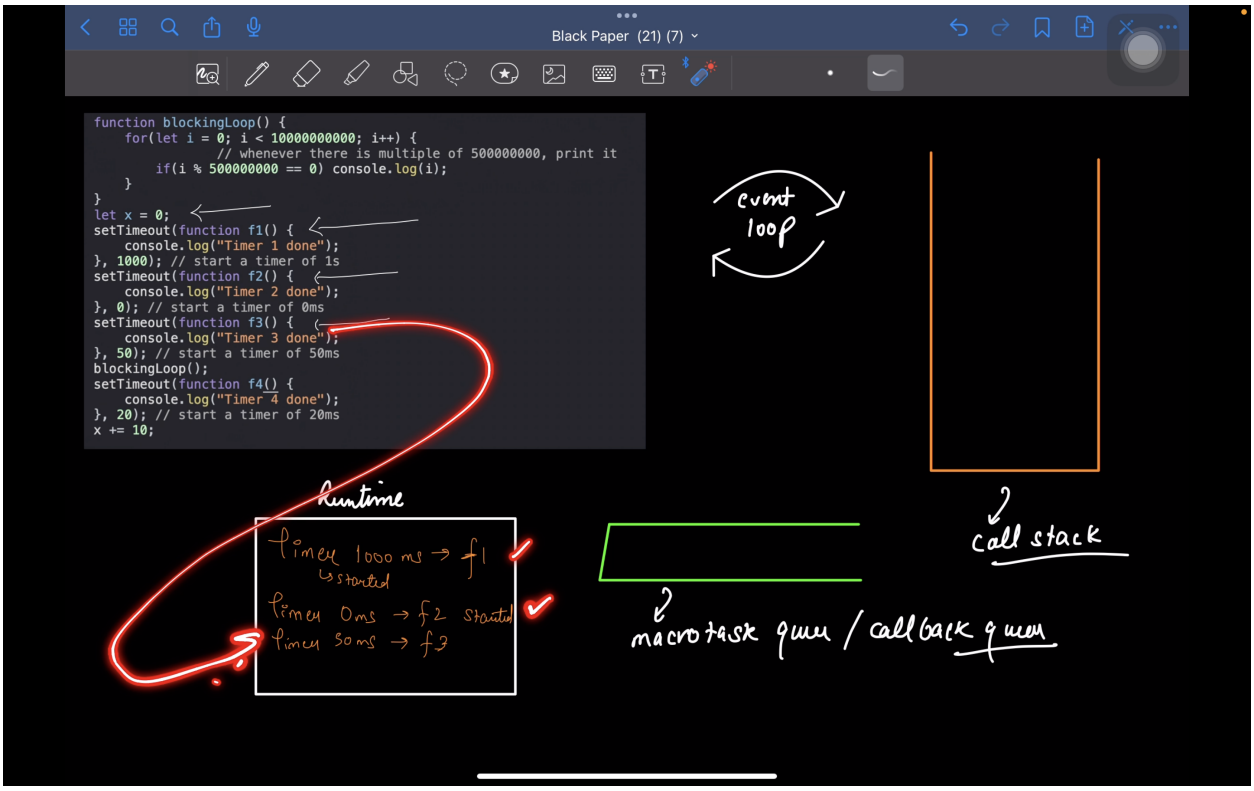
Let’s try to execute this piece of code one by one:



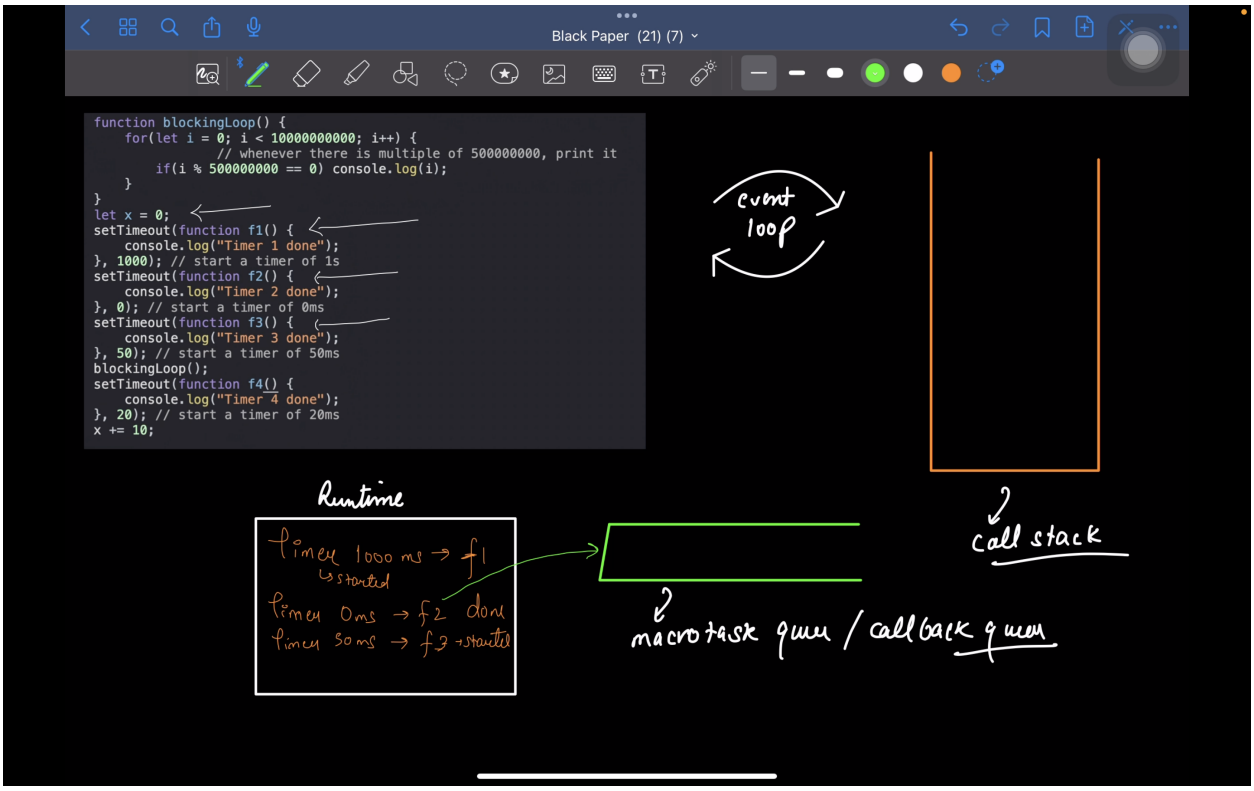
Firstly we initialise the variable x to 0, which is sync call. Then we see a timer, so we signal the runtime to start a timer of 1 sec, and post that execute the callback `f1`. And then JS will go back, not wait for anything and behind the scenes in a moment this timer will also start.



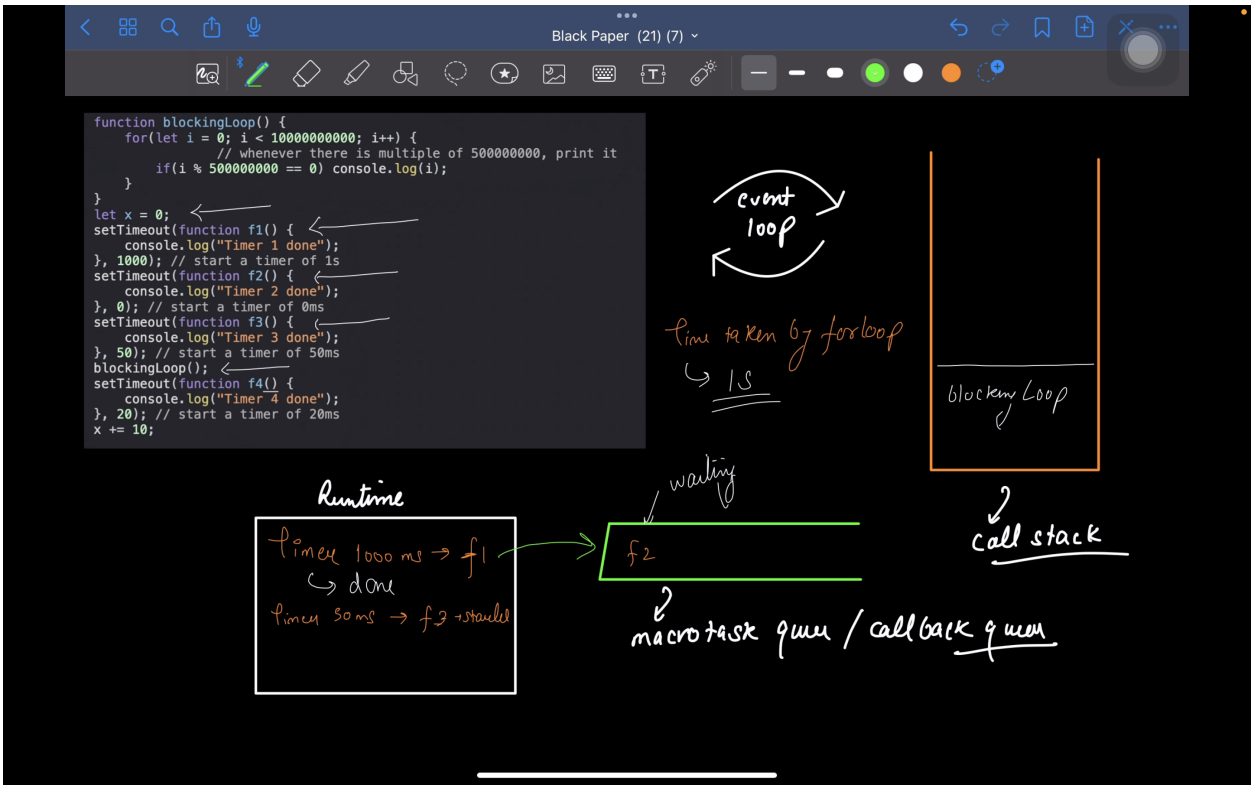
Then JS sees another timer, so it again goes and signals the runtime to start a timer of 0ms and post that execute the callback `f2`. Also behind the scenes timer 1 is running. And now JS will go back and not wait for anyone.



Then JS sees another timer, so it again goes and signals the runtime to start a timer of 50ms and post that execute the callback `f3`. Also behind the scenes timer 1 is running, and timer 2 must be completed almost. And now JS will go back and not wait for anyone.

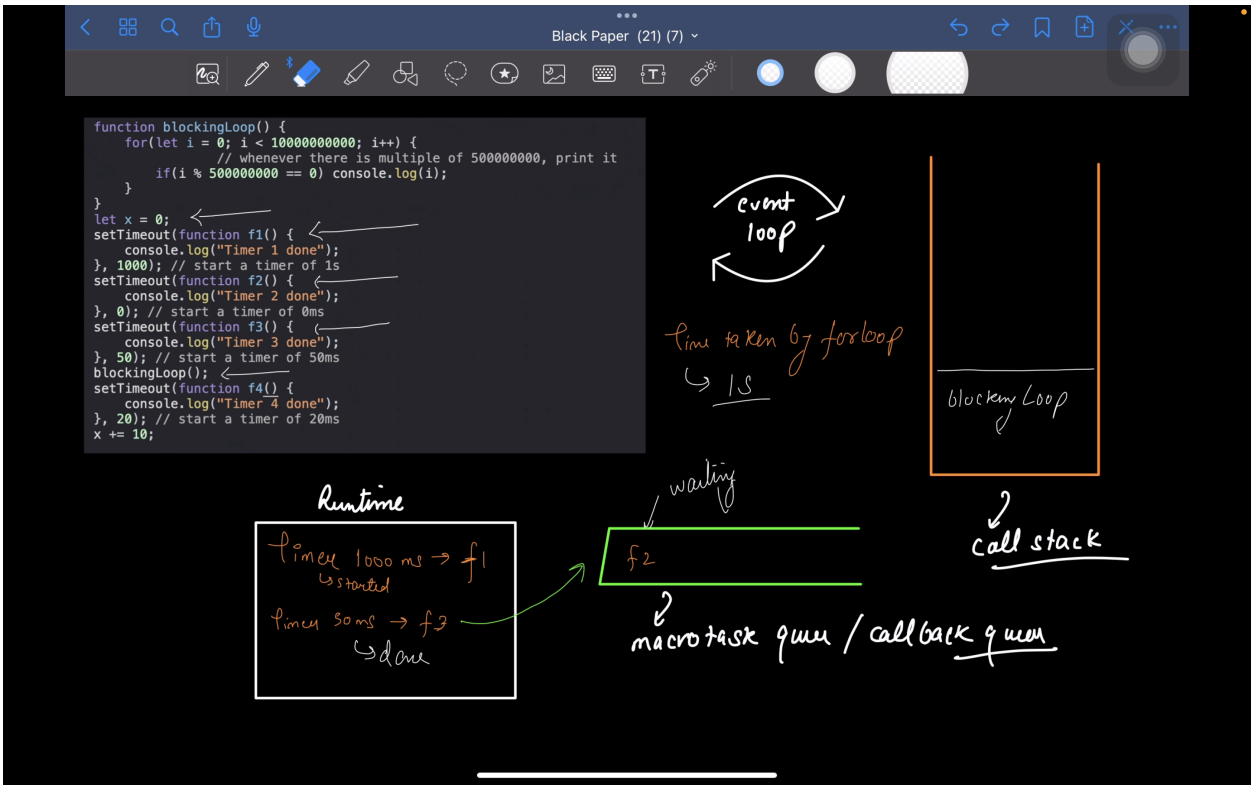


Now as behind the scene while JS was signalling about the third timer, timer 2 is done. Meanwhile Event loop is constantly checking that is the call stack empty and global piece of code is done? But global code is still left, so `f2` has to wait in the callback queue.

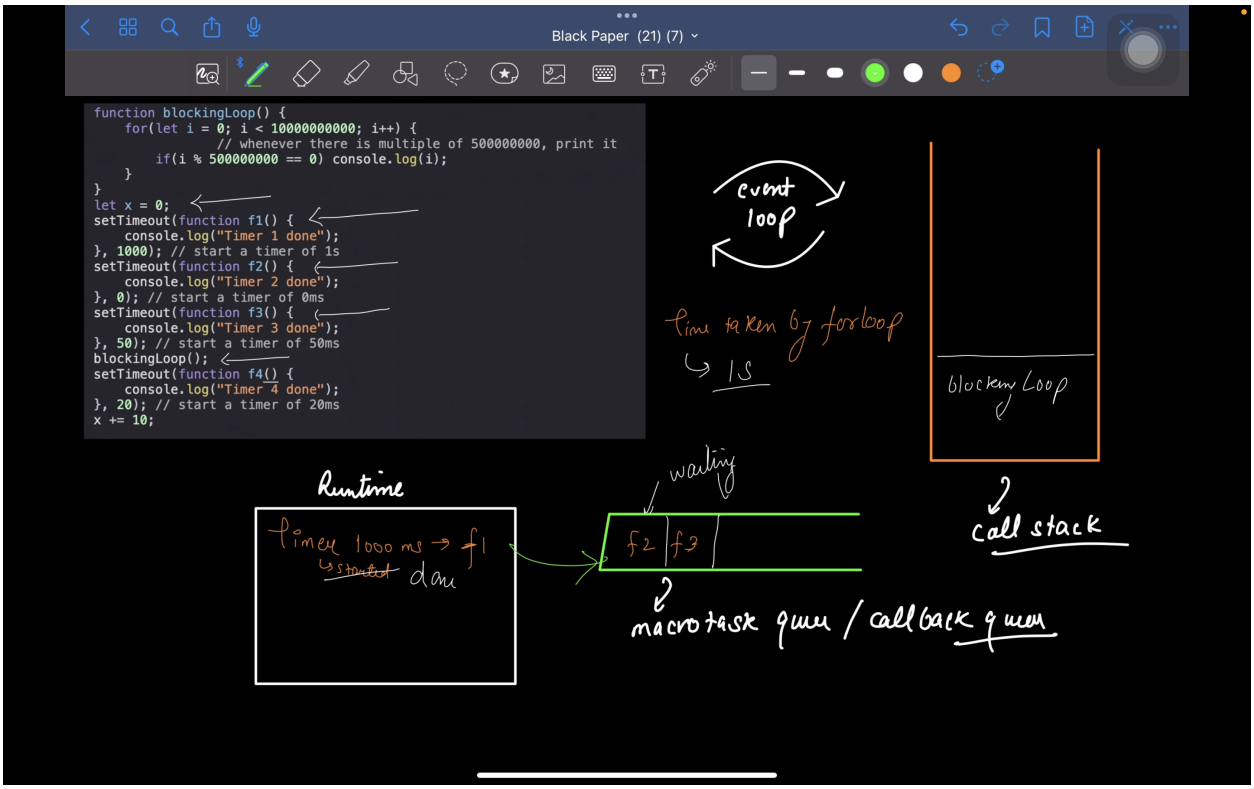


So f2 goes in the callback queue, and JS meanwhile goes back and triggers the function `blockingLoop`. As a new function call is made, it goes in the `call stack` and starts execution.

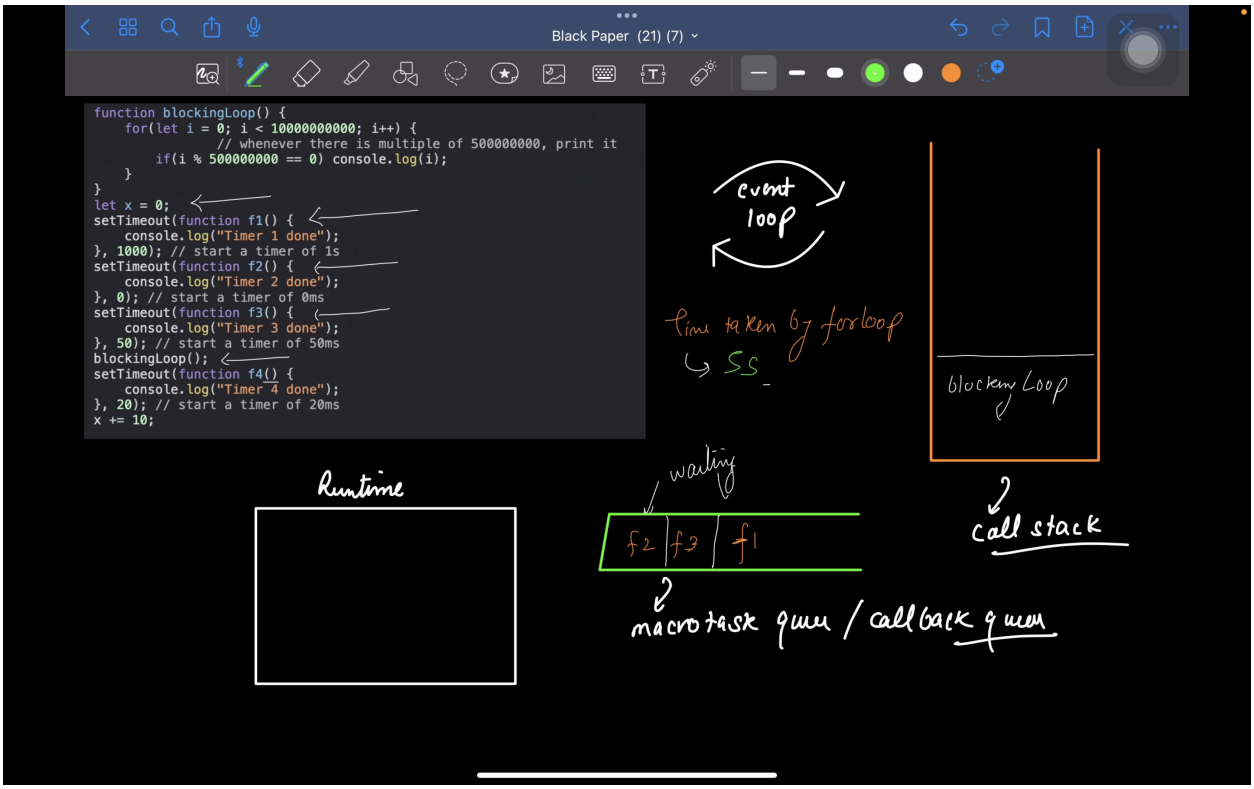
This is a pretty long and time consuming loop. So definitely by the time this loop will end, approximately after 10-12 s, the timers must be done. But let's take a close look, of what will happen after 1s of execution of this loop.



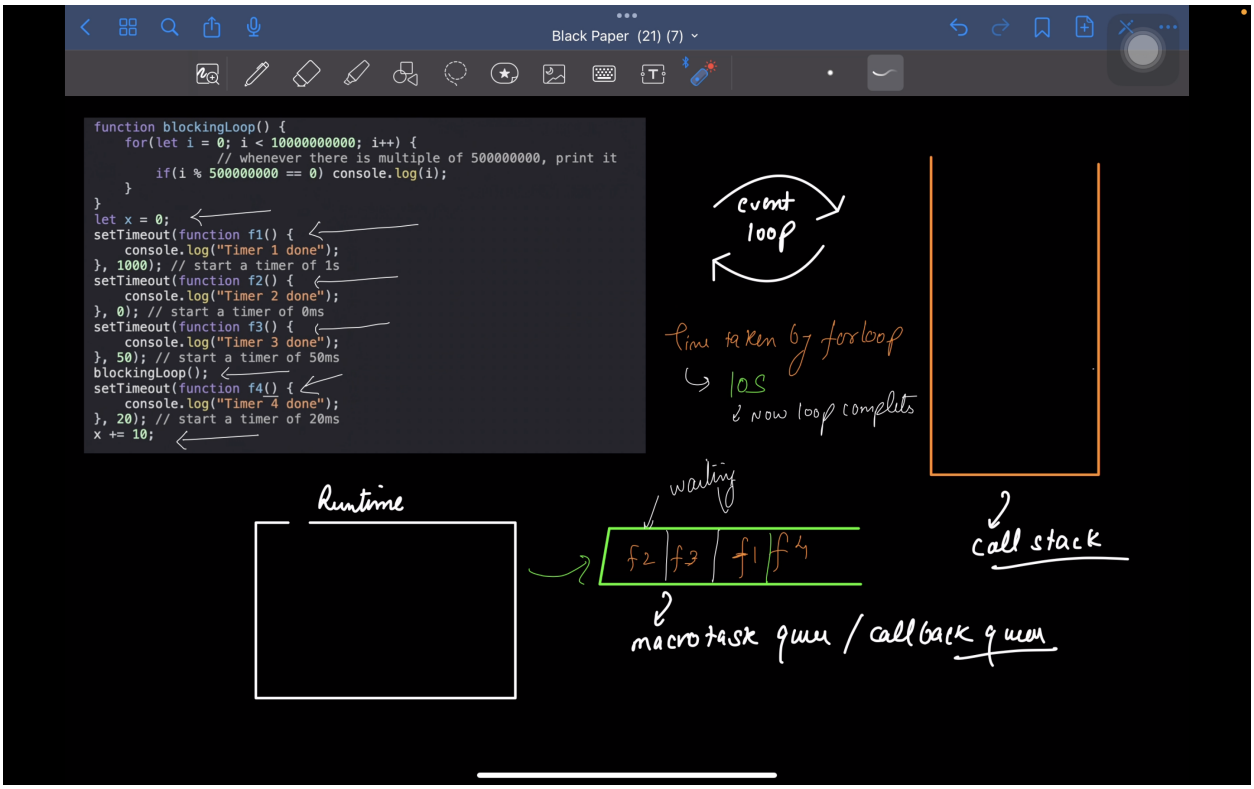
So even before we complete 1s of the loop, the 50ms timer which is the timer 3 is done. Event loop meanwhile checks is the call stack empty ? No, so `f3` callback has to wait in the callback queue, so we will transfer it.



Now as 1s of execution of for loop is done, definitely the Timer 1 is also done. Event loop checks is the call stack empty ? no, so `f1` has to wait in the callback queue. ()And of course f1 has to wait because f2 and f3 are already in the waitlist, so f1 will get chance after them only).

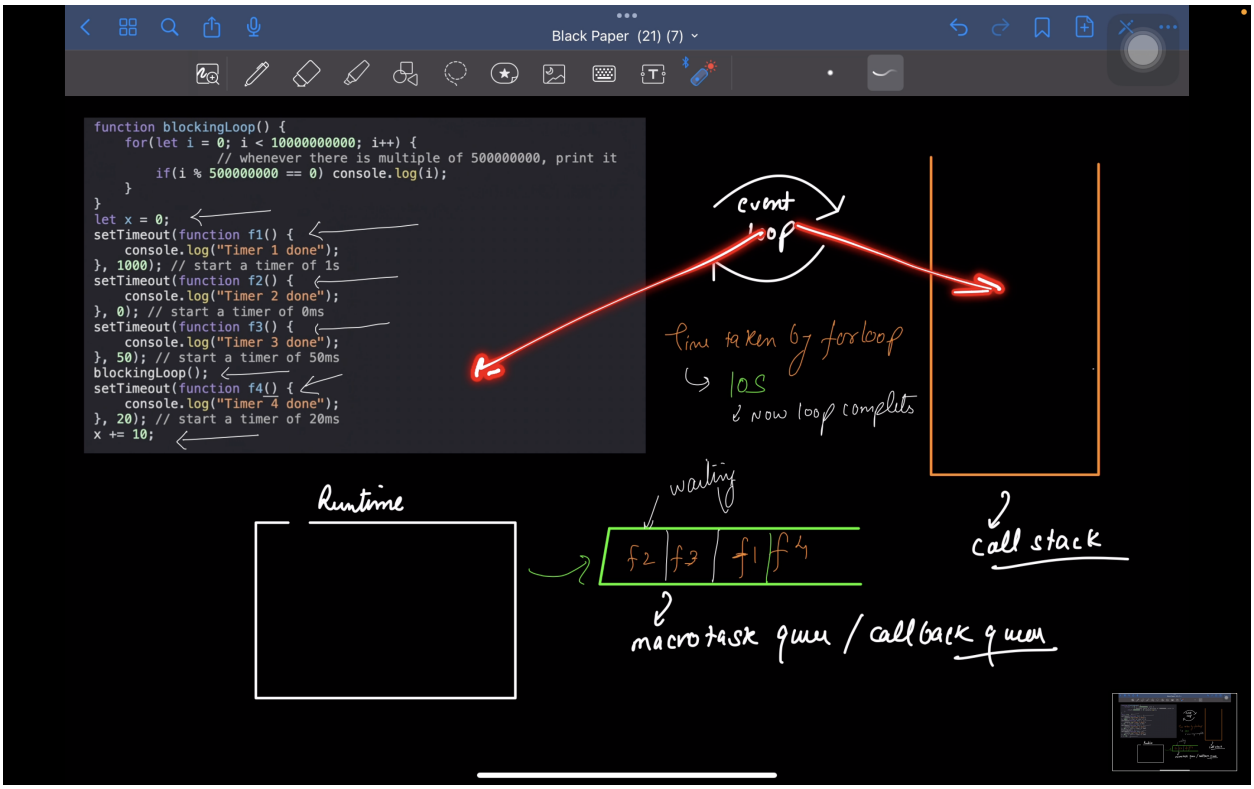


No all the timers are done, and let’s say 5 more sec are completed , but still callbacks won’t be executed as the Event loop will detect that call stack is not empty.

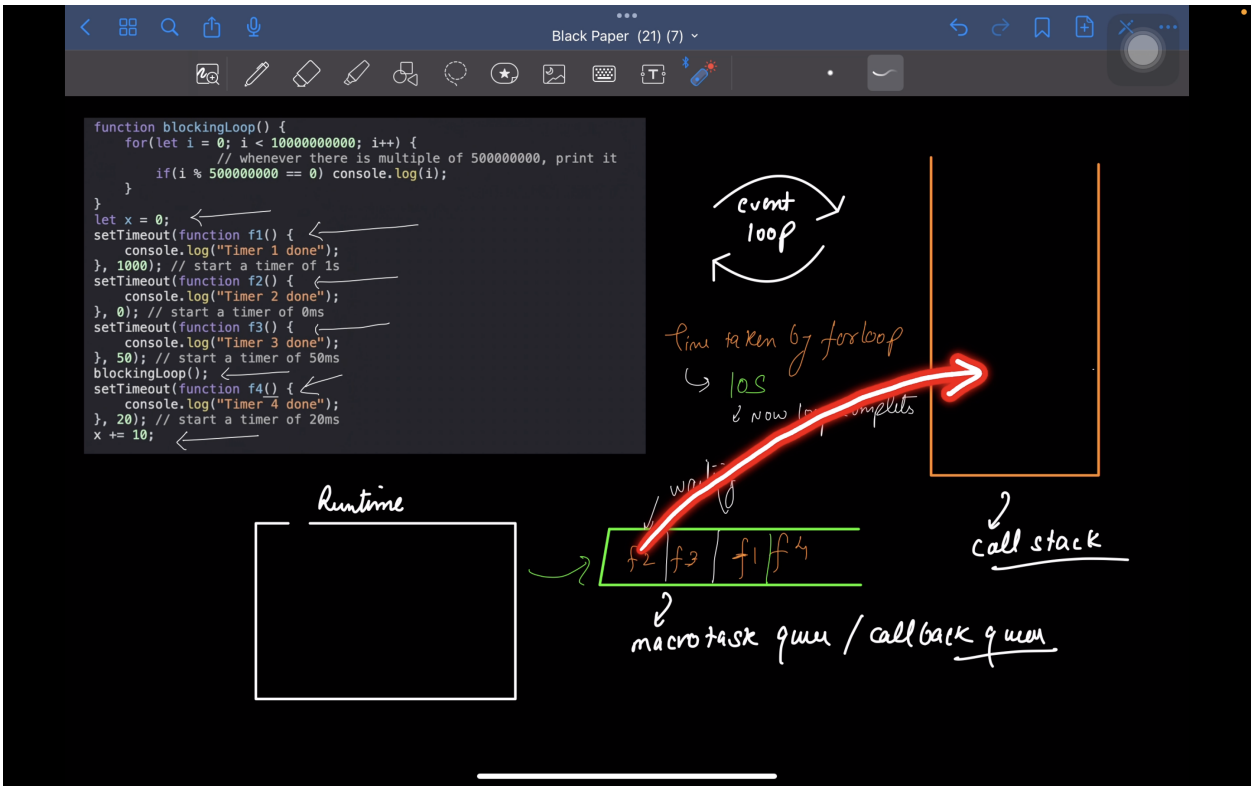


Now let's say, 10s are gone, and the function `blockingLoop` is done, so it will be removed from the call stack, Event loop meanwhile again and again check is the call stack empty or not. But this time call stack is empty, now it will check is the global piece of code done ? No. So callbacks waiting in the callback queue has to wait more. So now we will trigger timer 4 also, and ask the runtime to start a timer of 20ms. And then not wait there, and comeback to do `x+=10`

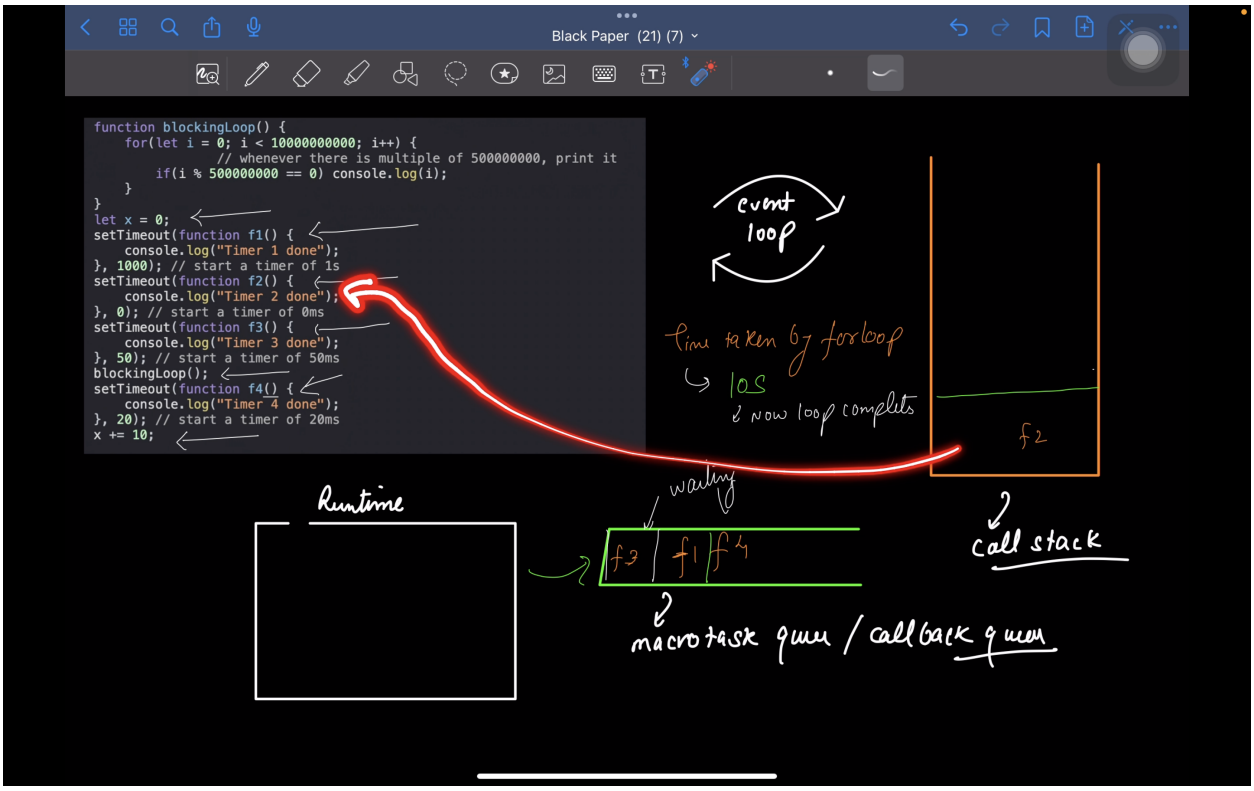
After this also within a moment timer 4 will be also be done and `f4` will go in the callback queue.



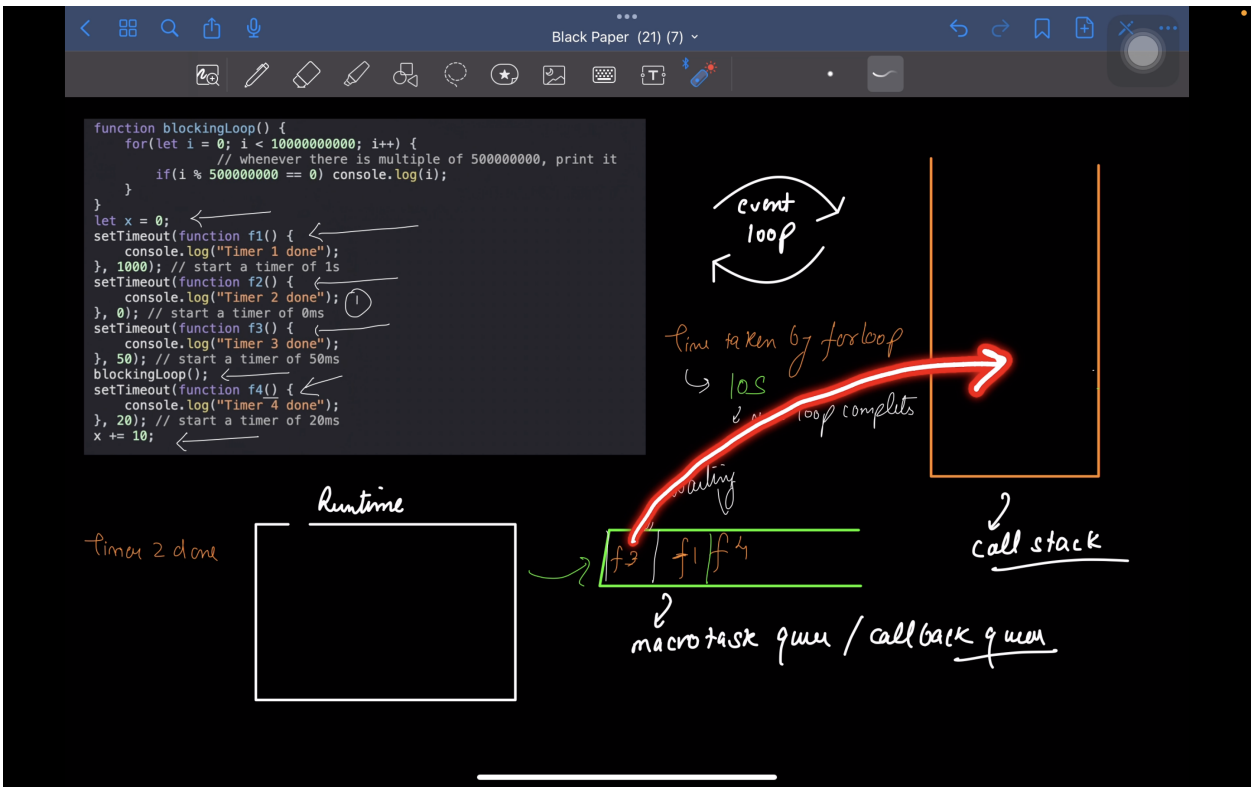
Now here, event loop detects that call stack is empty, and even the global piece of code is done. So now, Event loop will signal the callback queue, that dequeue a callback and push it in the call stack. Here f2 will be dequeued as it came first.



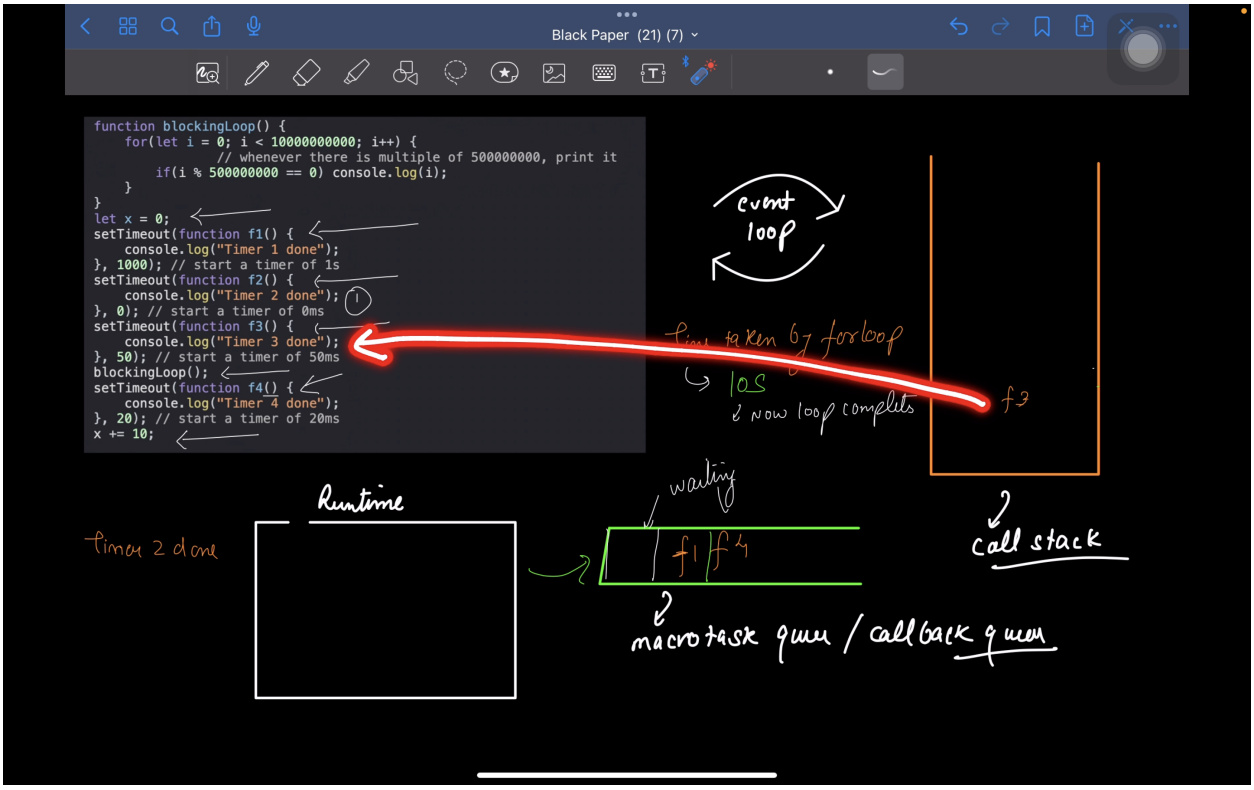
So now f2 will start the execution, and as f2 is a function which is going in the call stack, the remaining call backs f3, f1, and f4 has to wait more.



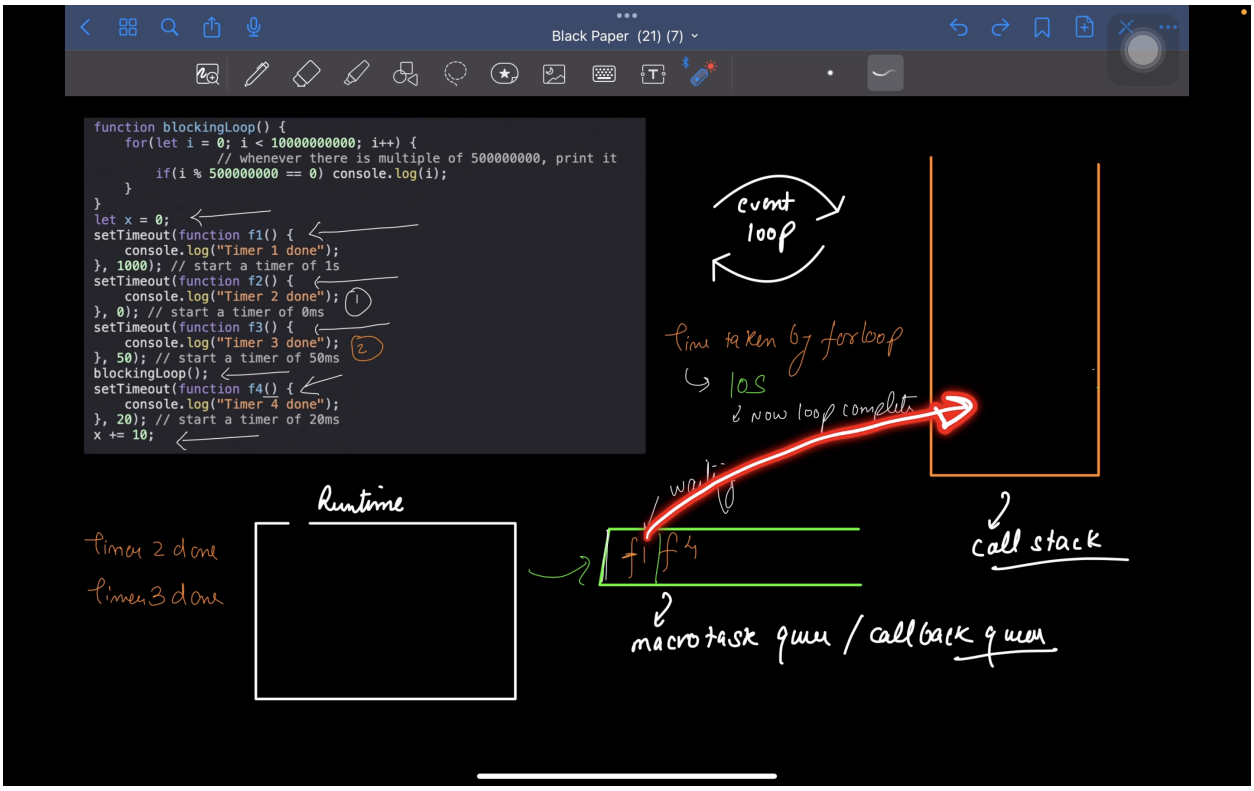
Now f2 completes the execution, and is going to be popped out of the call stack.



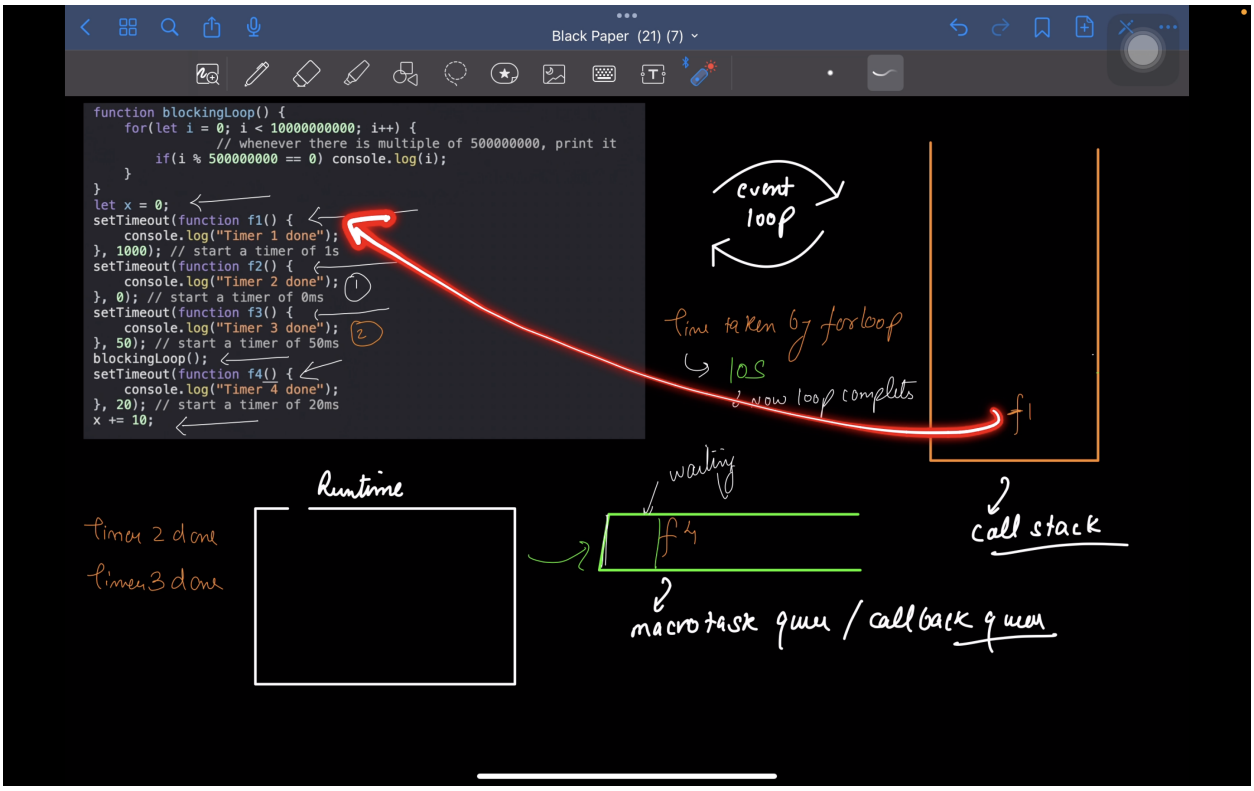
Now Event loop will detect that call stack is done as `f2` is completed, and no global code is left, so it will signal the callback queue to dequeue one element and push in the call stack, this time `f3` it is.



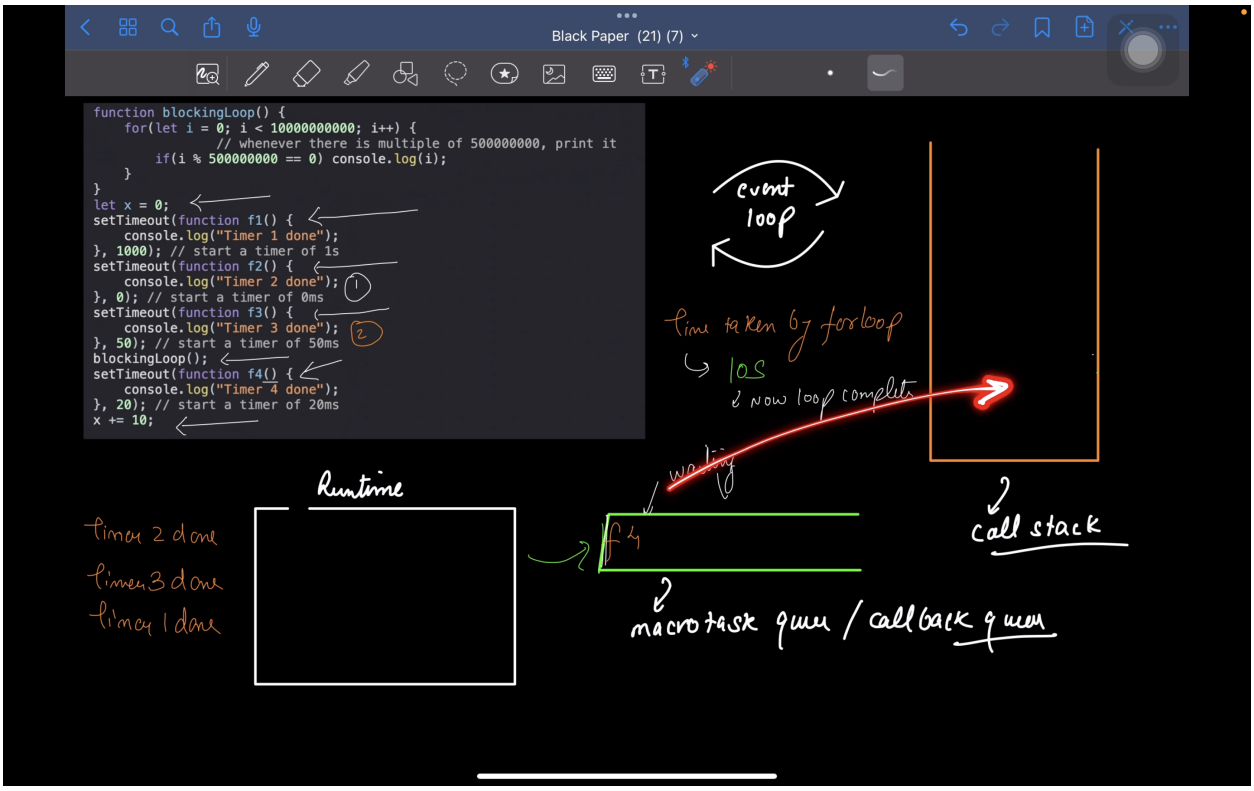
Now f3 completes the execution, and is going to be popped out of the call stack.



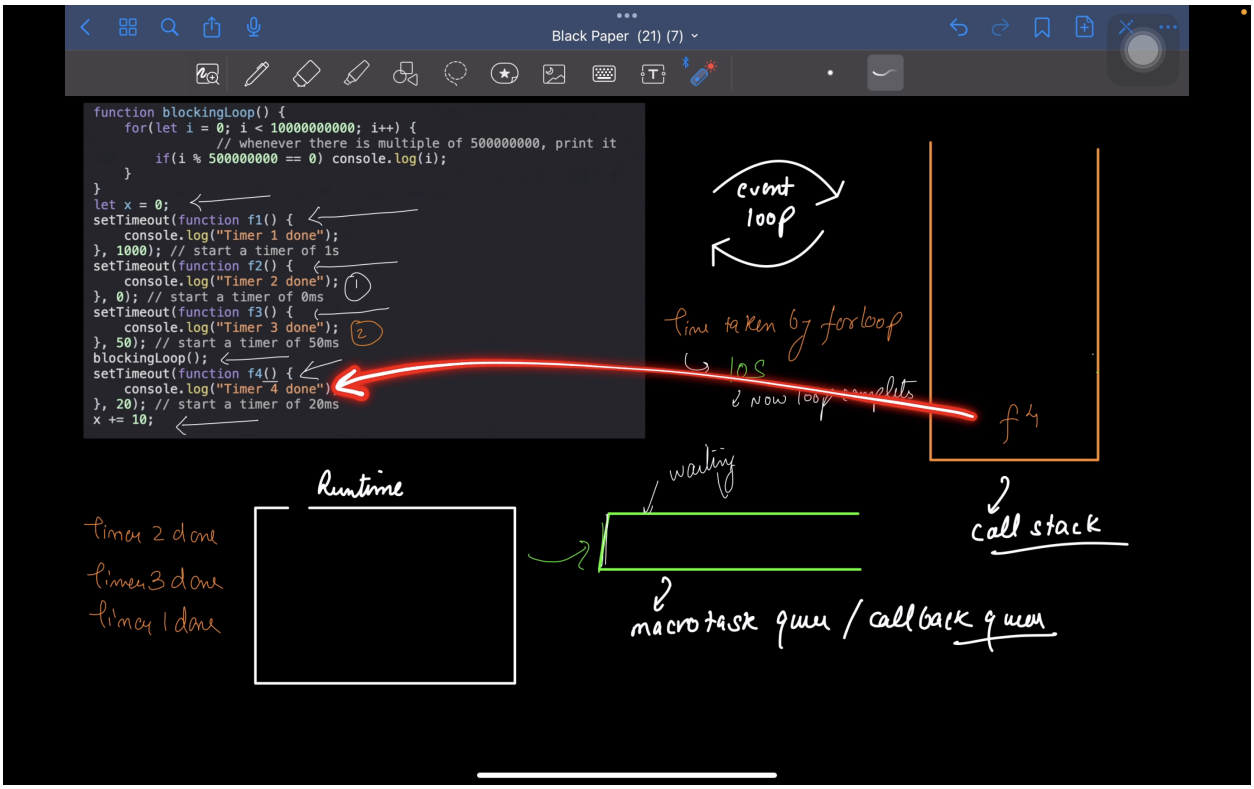
Now Event loop will detect that call stack is done as `f3` is completed, and no global code is left, so it will signal the callback queue to dequeue one element and push in the call stack, this time `f1` it is.



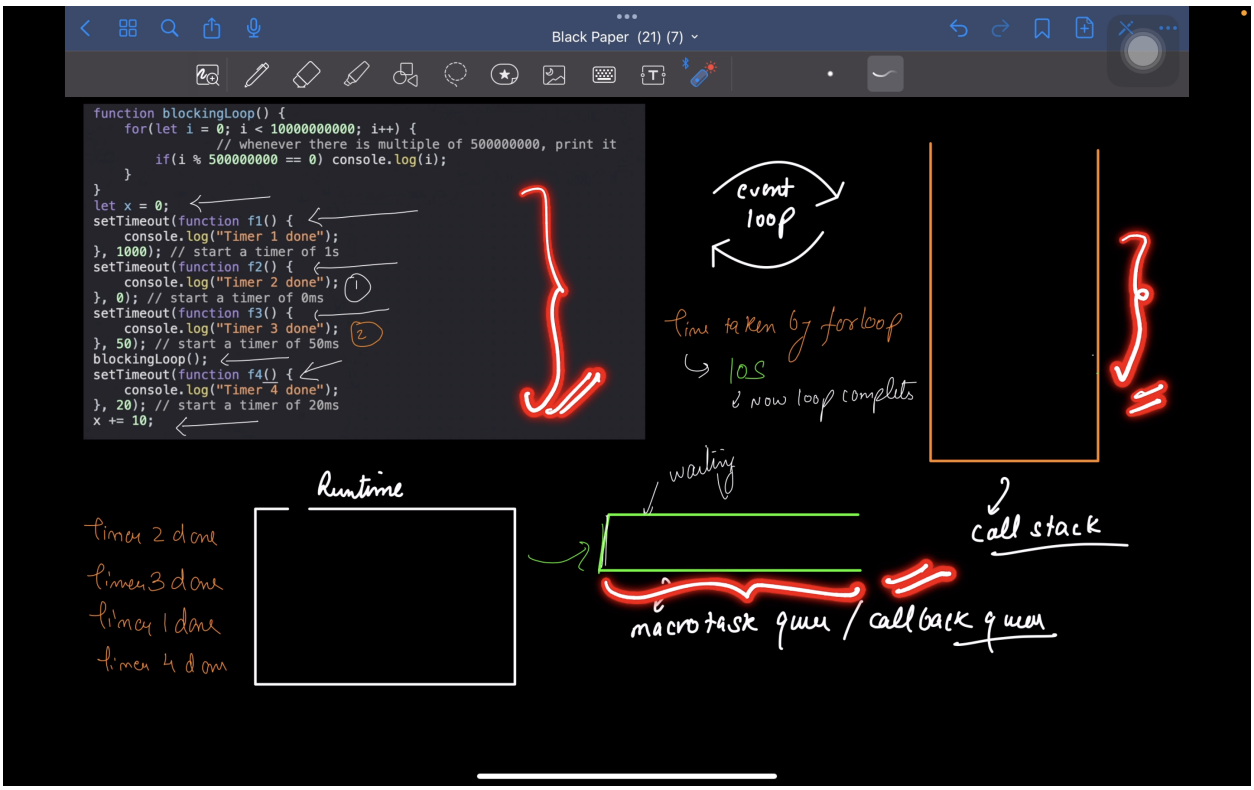
Now f1 completes the execution, and is going to be popped out of the call stack. Also on the left side, you can the what is getting printed.



Now Event loop will detect that call stack is done as `f1` is completed, and no global code is left, so it will signal the callback queue to dequeue one element and push in the call stack, this time `f4` it is.

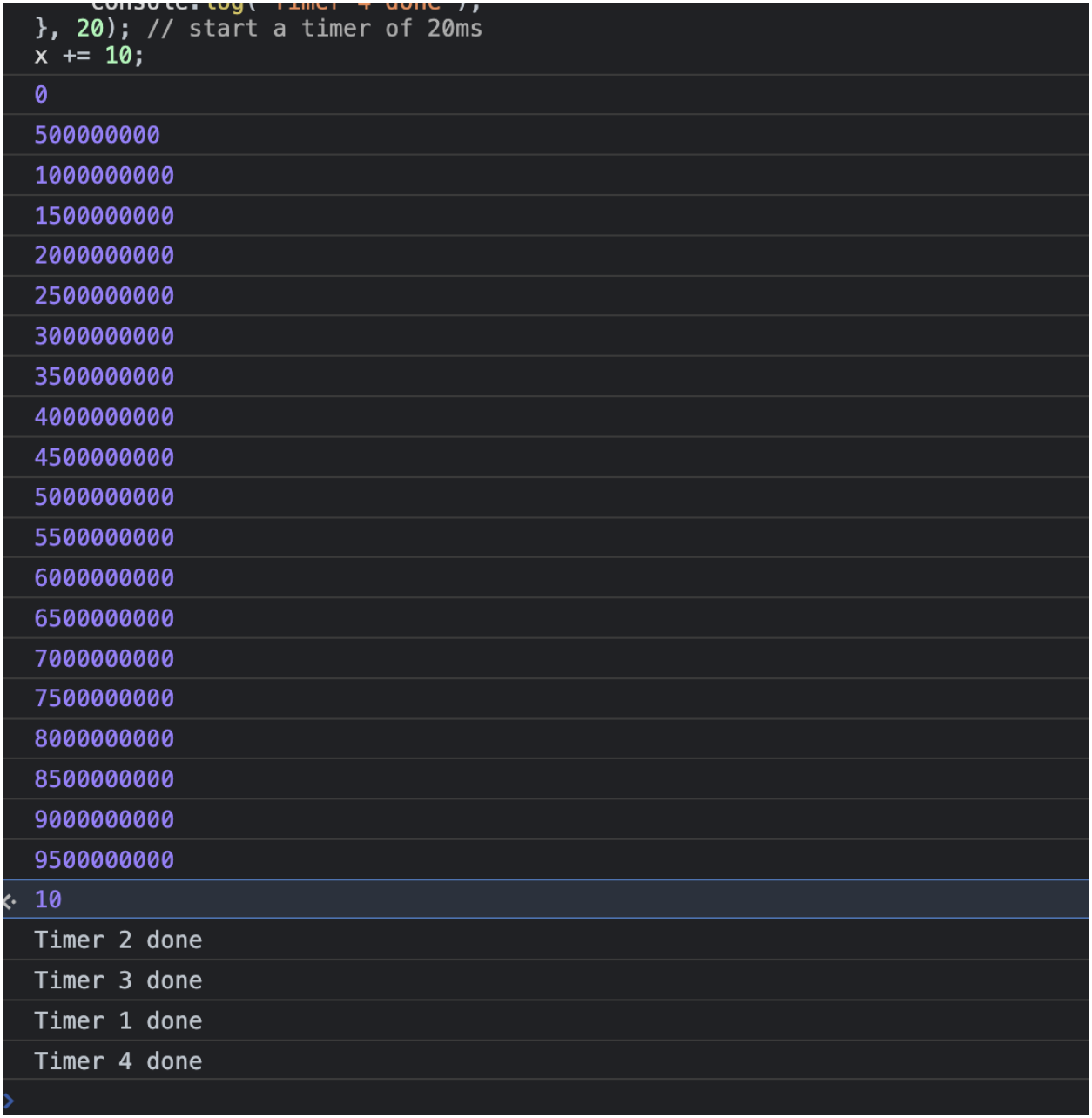


Now f4 completes the execution, and is going to be popped out of the call stack.



And now the call stack is empty, there is no one waiting in the macro task queue, nothing left in the runtime, all the global code done, so the program ends.

Output of this code will be



Time 2 , timer 3 and timer 1 all were running parallel but not in JS, but inside the runtime. How runtime handles their parallel execution is of no use to JS. JS will just signal the runtime ti start a task, and then move ahead.

Output will also depend on which timer / task got completed first and pushed it's call back in the queue.

Note: You must be thinking what is the use of the `Microtask queue` that we mentioned earlier. So don't worry we will discuss it later.

Knowledge test: Famous interview question

What will be the output if this code?

```
let x = 0;
console.log("start");
setTimeout(function f() {
    console.log("done");
}, 0);
console.log("end");
```

Here output will be

```
start
end
done
```

Why ? because even if there is a 0ms timer, JS will not wait for it as the runtime might need to set it up and then do some computation, so JS will come back and print end, and then call back will be meanwhile transferred in the call back queue, which will be picked once call stack is empty and global code is done.