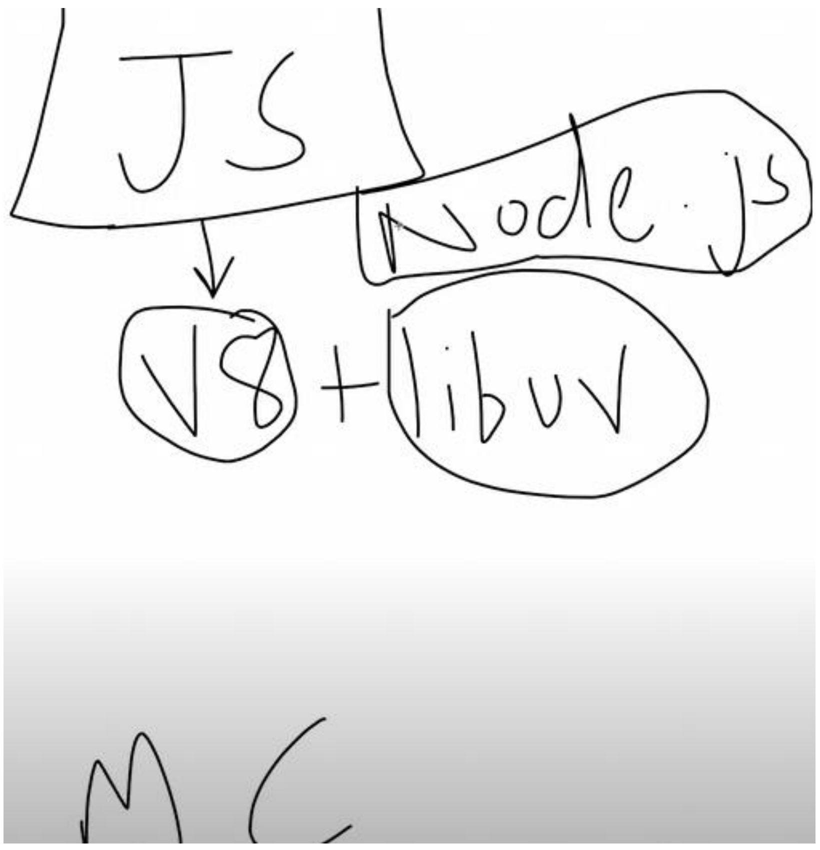
Qu: What is Nodejs Javascript Runtime exactly is?

We as developers we typically write code in javascript , does not matter if we are on browser or on nodejs server. This javascript at the end of the day has to reach our processor it means finally it has to be executed as machine code. So we have to reach at this machine level so we can write a piece of software which can convert javascript code directly into machine code and that is what an interpreter does. This is the job of an interpreter well this is the bigger picture for us and this interpreter here is know as V8 engine. This V8 engine interprets the javascript code and then converts it into the machine code which is then executed by our processor and this execution is independent of browser based execution or nodejs based execution but the things are not so simple in the real world. What happens for browser is we have a lot of helper functions available like document.getElementById etc which we can use directly in javascript. V8 however does not contains those functions so we can think of this as our browser is injecting certain commands in your javascript runtime so things like these can be customized on the basis of where you are executing your piece of javascript. For example if you are running native script which is another framework for executing javascript as mobile applications you're gonna see you can access something like java apis on android from native scripts. So native script injects certain bindings in V8 which helps you to invoke certain native calls directly from javascript.

In nodejs there are two main component of executing javascript which is V8 and linuv.

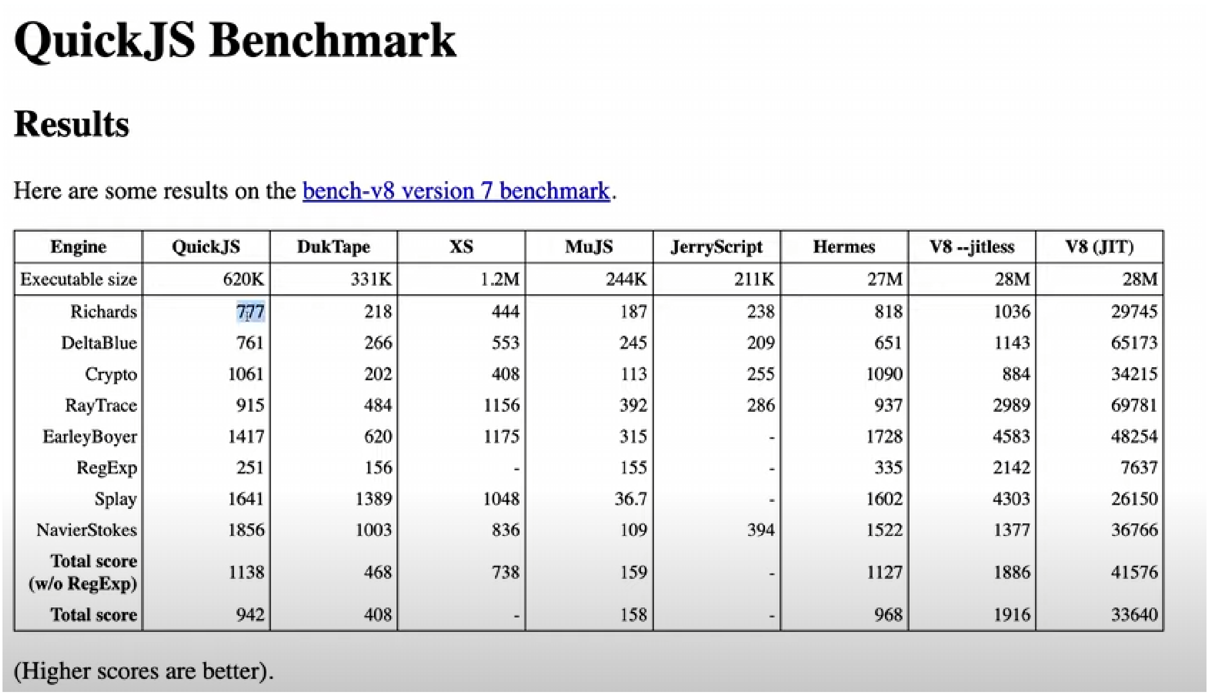


So What is the role of nodejs exactly here, well similarly to the browser which acts as a medium to connect the webpages to V8 and libuv which is the actual underlying technology which converts our javascript to machine code. Similarly the nodejs is a bridge which fills the gap of certain APIs, which makes certain APIs consistent across your javascript codes by this we means that our V8 software which is written mostly in C++ is responsible for low level execution and understanding of our javascript code. However if you want to run an HTTP server that probably is not present as a direct thing in V8. That's a direct function so first of all we need to get sockets out of V8 then you need to open a PORT on a particular socket then you need to probably listen to the traffic and when you want to tear down the server you have to close the server manually so you'll have to do all that stuff manually now this can be done either in javascript or you might need C or C++ if nodejs is not there. SO if nodejs is not there and you are directly interacting with V8 then you might need to write a bunch of C++ codes and a bunch of javascript code. Nodejs however does this automatically for you so it sort of bridges the gap so that you don't have to interact with V8 and libuv because at the end we just want to write javascript we probably don't want to interact with C and C++ interfaces of V8 so nodejs bridges all those gaps , all the things which you possibly might need as a javascript developer obviously you can not access every low level concept for example memory management in terms of pointers which you can do in terms of C and C++ and the point is we don't need to manage all these because nodejs is managing all these things.

So if we don't have nodejs in between we probably would not have the http package with you. http package is not the job of V8 it is the job of nodejs, we would not have the fs package which is the file system package because fs is not the job of V8 it is the job of V8, similarly we would not have the crypto package. The job of V8 is to understand what you have written in javascript and convert it into machine code, the job of nodejs is to extend the functionality and fill the gap between our javascript world and V8 and libuv world to provide you better libraries , less verbose solutions of the most commonly used things and this are just libraries built in javascripts.

Nodejs however consists of both javascript code, the source code of nodejs and the C++ code.

Note: Nodejs is something which helps you fill the gap of running javascript on the server. How? By introducing C++ code for bindings libraries like file system , http, crypto etc because the job of V8 is only and only to understand your javascript and convert it into an optimized code which our computer can understand and can run and libuv on the other hand manages the event loop and concurrency tasks and asynchronous parts of javascript.



QuickJS Benchmark

Results

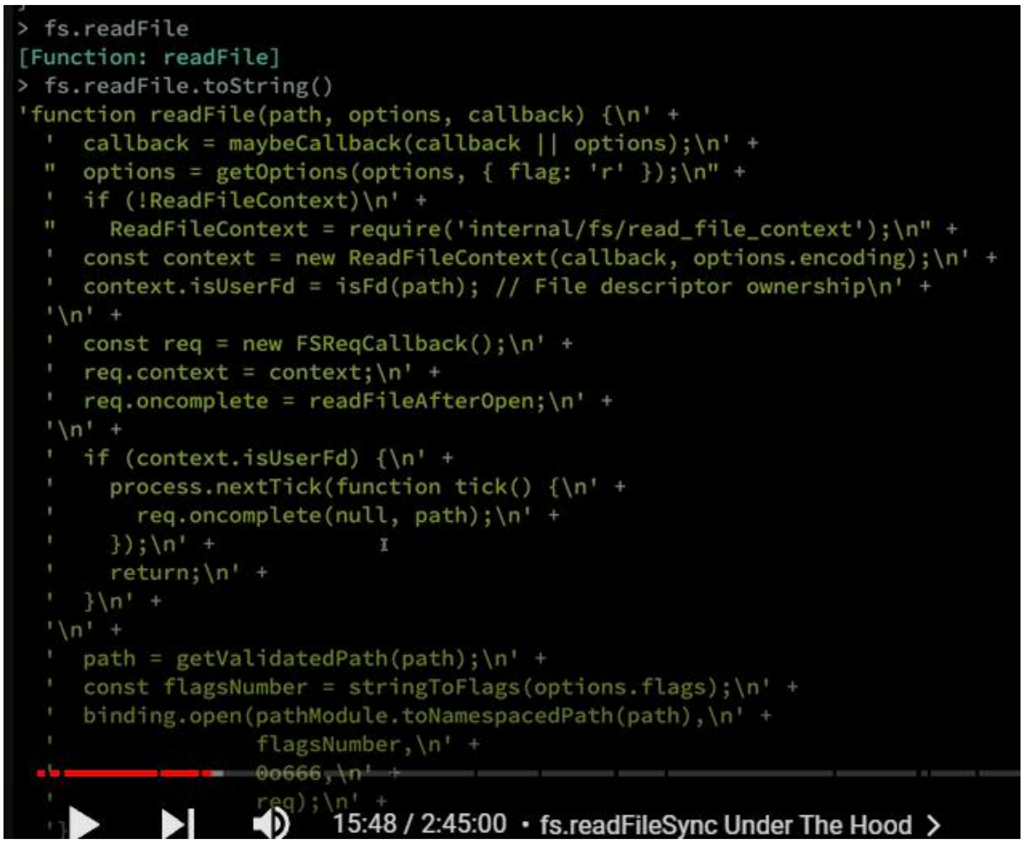
Here are some results on the bench-v8 version 7 benchmark.

Engine	QuickJS	DukTape	XS	MuJS	JerryScript	Hermes	V8 --jitless	V8 (JIT)
Executable size	620K	331K	1.2M	244K	211K	27M	28M	28M
Richards	777	218	444	187	238	818	1036	29745
DeltaBlue	761	266	553	245	209	651	1143	65173
Crypto	1061	202	408	113	255	1090	884	34215
RayTrace	915	484	1156	392	286	937	2989	69781
EarleyBoyer	1417	620	1175	315	-	1728	4583	48254
RegExp	251	156	-	155	-	335	2142	7637
Splay	1641	1389	1048	36.7	-	1602	4303	26150
NavierStokes	1856	1003	836	109	394	1522	1377	36766
Total score (w/o RegExp)	1138	468	738	159	-	1127	1886	41576
Total score	942	408	-	158	-	968	1916	33640

(Higher scores are better).

Qu: Underlying implementations of javascript functions in nodejs? fs.readFileSync under the hood?

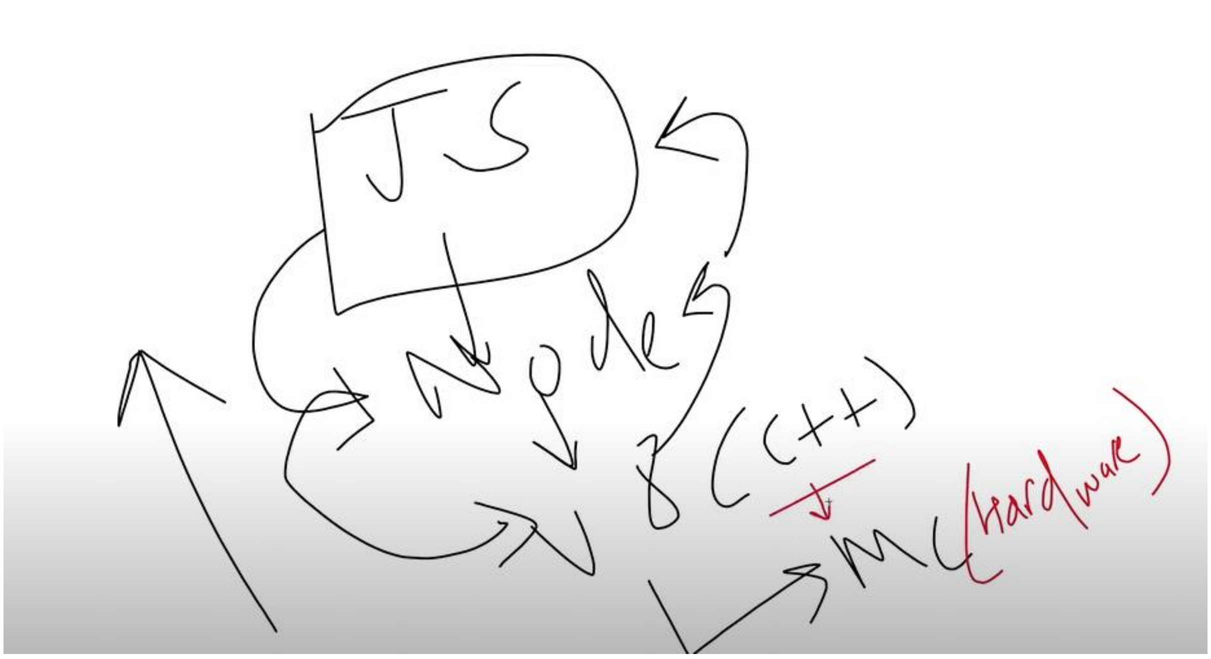
If you take a look at a typical nodejs program , you have libraries which comes built in like fs which is a module and we have a bunch of functions available in fs. So where exactly is the fs.readFile function.



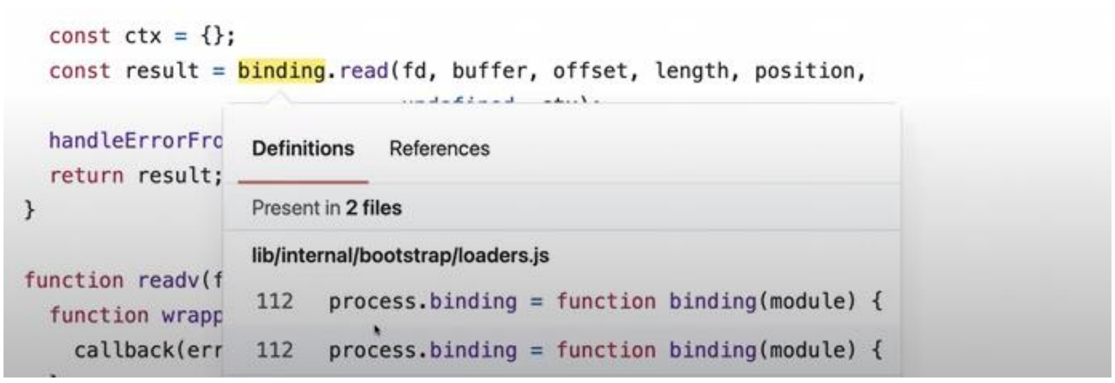
```
> fs.readFile
[Function: readFile]
> fs.readFile.toString()
'function readFile(path, options, callback) {\n' +
  '  callback = maybeCallback(callback || options);\n' +
  "  options = getOptions(options, { flag: 'r' });\n" +
  '  if (!ReadFileContext)\n' +
  "    ReadFileContext = require('internal/fs/read_file_context');\n" +
  '  const context = new ReadFileContext(callback, options.encoding);\n' +
  '  context.isUserFd = isFd(path); // File descriptor ownership\n' +
  '\n' +
  '  const req = new FSReqCallback();\n' +
  '  req.context = context;\n' +
  '  req.oncomplete = readFileAfterOpen;\n' +
  '\n' +
  '  if (context.isUserFd) {\n' +
  '    process.nextTick(function tick() {\n' +
  '      req.oncomplete(null, path);\n' +
  '    });\n' +
  '    return;\n' +
  '  }\n' +
  '\n' +
  '  path = getValidatedPath(path);\n' +
  '  const flagsNumber = stringToFlags(options.flags);\n' +
  '  binding.open(pathModule.toNamespacedPath(path),\n' +
  '               flagsNumber,\n' +
  '               0o666,\n' +
  '               req);\n' +
```

But where is it? Where can I see it? Even if I look at it , it looks like a regular javascript function so where is the C++ code of this function? We can see this in the nodejs package which is available in github.

We can do most of the stuffs in javascripts but the javascript is not good in computations because the javascript do the input output in the most possible way due to how the infastructure of it is laid down.



The above shows how javascript works. Now what happens is when you want to compute something in javascript that has to perform this hops over and over again then V8 will return the result to nodejs and then nodejs will return it to your code and then you'll use it then the things would happen again and again. So the farther away the hardware the more likely your code is slower for CPU operations. You see that the C and C++ just sits above the bare metal of your computer because C and C++ gets converted to assembly code and then it converts into byte code or opcode or machine code so its very very close to the hardware javascript on the other hand can involve a lot of validation checks for example if you access an array which is out of bounds then instead of crashing the program it returns undefined so the stuff like this will actually eat up the execution time of things. So the reason we are discussing this stuffs is because a lot of stuff instead of going through javascript world to the actual machine code and then all the way back, we just code it up in the C++ so reading a file or crypto algorithm is one such thing that is it is coded up in C++ or in some level format of language and then we bridge the javascript world with the C++ world with this binding which comes from process.binding.



```
  const ctx = {};
  const result = binding.read(fd, buffer, offset, length, position,

  handleErrorFro
  return result;
}

function readv(f
  function wrapp
    callback(err
```

Definitions References

Present in 2 files

lib/internal/bootstrap/loaders.js

```
112   process.binding = function binding(module) {
112   process.binding = function binding(module) {
```

So this is the bridge between javascript and C++ world that means this read function exists in C++ world.

Note: In the translation of javascript to node.js of the CPP , we obviously need V8 to kick in and help node understand what's happening so its not so fair to say that javascript sits on top of node.js which internally sits on top of V8 because sometimes what happens is that your javascript or maybe like your code could actually request V8 to convert a sort of a code first and then node.js just interprets it in a native way. Node.js uses V8 engine and V8 data types and promises to make sense of the actual javascript code.

Qu: Threads and Processes ?

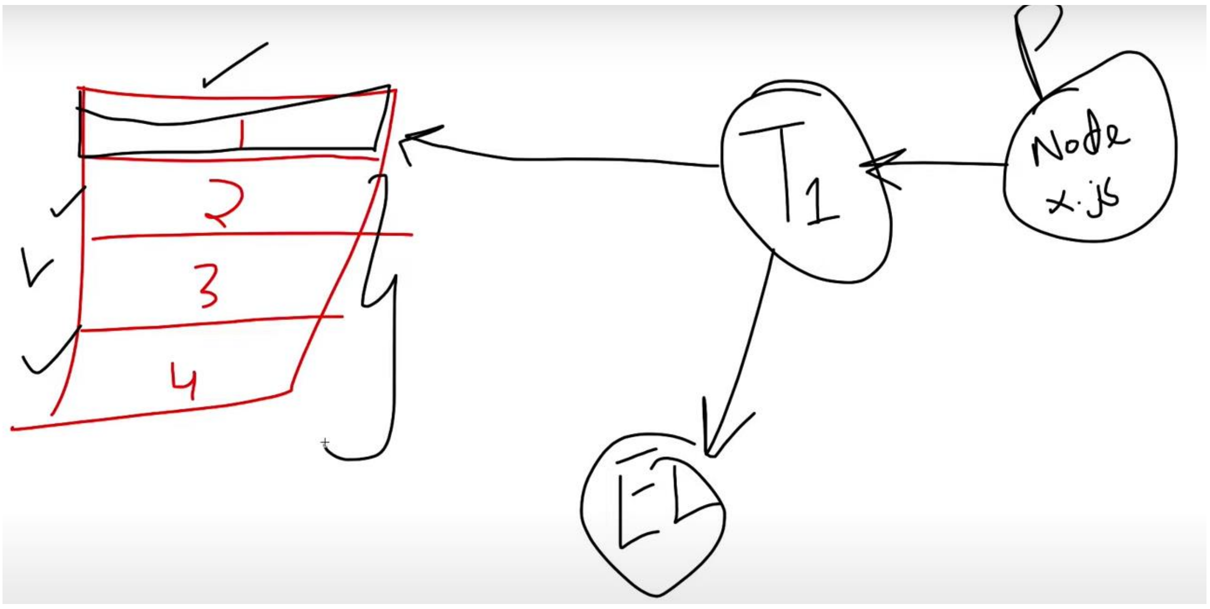
Every Single Executable if it is running in foreground or background does not matter has to have a process. The process is a fundamental unit of execution in terms of a standalone application you can not just have a single thread of something a process should be there which can spawn multiple threads. This thread is pretty much what does the work. So just think of process as a manager or something which spawns the threads which does the work. So the process can spawn single thread or it can spawn multiple threads in which case we can think that this process is doing multiple things parallelly.

The CPU which we have has got multiple Cores. Lets say we have 4 Core Processor it means that our CPU is capable of executing four threads or four execution units parallelly because thread is the fundamental unit of work and process is a sort of parent which manages these threads, the memory access , communication and all that stuff can go through the process but the actual CPU work the calculation , the computation is going to happen in these threads. So if your computer is a four core processor every thread if assuming that there are four threads actively running , every thread gets a complete core. This is fine but how about if you have 100 threads running so what happens is operating systems comes with something known as schedulers.

The Schedulers have the job of making sure that this total of 100 threads for example of what we have and limited four cores we have for execution, the scheduler’s job is to allocate this threads based on certain algorithms so that all the treads get their fair amount of time and by fair it does not mean equal amount of time is spread on every thread why? Because some threads are of higher priority for example a thread of uploading a file can be delayed due to some other thread of cancelling a nuclear launch so this is the reason schedulers exists. Here what we have to understand is that the modern computers and operating systems are capable of running threads more than their logical or physical cores in CPU and that is how context switching happens and that is how you are multiple applications.

Note: Node.js is sort of single threaded but not exactly. People say that Node.js is single threaded by this it means that node.js will use only a single core out of all the available cores which is kind of true but it is not 100% true but also it is not 100% false as well and vice versa.

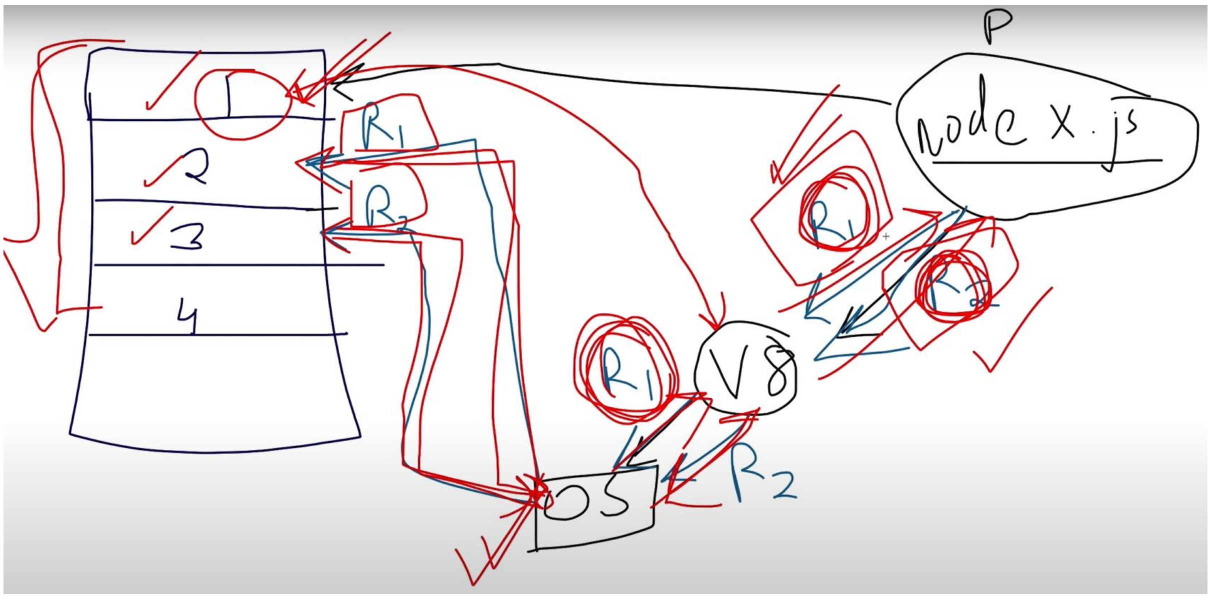
Note: Node.js is fundamentally single threaded language it means when you start node.js what happens is that this process launches a thread T1 and that’s it and this thread right here consists of event loop and it just keeps on executing on a single core , it does not executes on other cores it just executes on a single core , so are your rest of three cores wasted ? kind of it just not a 100% true and false.



Qu: Node.js is Multithreaded by default ?

As we already know that Node.js is single threaded in one way but its not in another way.

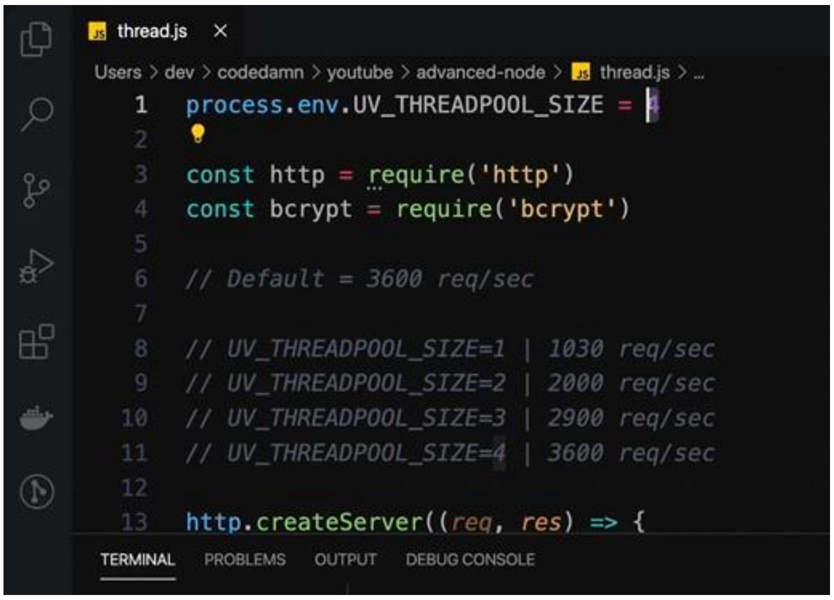
Let’s say we have a CPU with 4 cores and once you execute a script known as node x.js now we know that the process which is created for this script will run on the first core this is fine but you saw that when we were discussing the Fs.readFile function which is asynchronous, we actually delegate those certain tasks and certain things to operating system and how do we do that node.js makes a call to CPP world and CPP world makes a call to V8 world and V8 talks directly to the bit guys that is the operating system and an operating system is not restricted to a single core or a single process or a single thread now what do we mean by that consider an example in which your node.js code makes two HTTP requests asynchronously to some website so lets say first request is R1 and second request is R2 so this two requests are made asynchronously that means that node.js did not wait for this request to complete in order to give instruction for the next request so both of these requests are forwarded to V8 , V8 tells operating system that hey boss we need to visit R1 and R2 and operating system says ok that's not a problem although this is a network request but you can just think of this as an operating system stuff as well so what OS will do, it will say that Core 2 you take care of this R1 thing which we have and hey Core 3 you take care of this R2 thing which we have now you can sort of replace this network request analogy to something which is computationally heavy for example encryption of some sort so you can replace this with encryption and it will still hold true. So now what you are doing is that although you are executing the instructions that is this core 1 is responsible for running V8 but the actual execution which the operating system does in this case that is performing an encryption of whatever it is which V8 delegates further to OS which is performed you know the OS is now free to perform on any core because OS is not bounded by just a single core, the OS has the access to complete memory and complete hardware basically, so this way if you think about this then the node.js does not appear to be single threaded because if you can schedule tasks in a better way in an efficient way which node.js does with the help of event loop if you can schedule the task such that you are able to you know once the response has come back you are able to join the responses correctly (the response gets sent to V8 and then V8 tells node.js about them) so if node.js is able to join these responses correctly for whatever call it made then it makes complete sense to say that this is not a single threaded model because you just performed two computations R1 and R2 on two different cores although your node.js was a single threaded in a actual single threaded environment (single processor) you would probably wait for R1 first and then dispatch a request for R2 but because node.js is asynchronous because node.js supports asynchronous behaviour with the help of how event loop works, you can have sort of multithreaded behaviour with the help of operating system even without the language being multi-threaded so its like that node.js is kind of single threaded in itself but if you take a look at a bigger picture the stuff which you can do with node.js is not exactly single threaded.



Qu: Bcrypt ?

Bcrypt is a hashing library and the most common use case of bcrypt is for hashing the passwords for storing it in the database. You can think of bcrypt as a function which takes in a value (buffer or string) and converts it into something which is garbage and this garbage is a special type of garbage because this is only predictable one way that means if you have the plain text and then you do the bcrypt of this text you get some garbage value now you can not really go from this garbage value to the text back without massive brute forcing and computing which makes it almost unfeasible to do it however if you have this text again you can verify this against the garbage value quickly and this is how bcrypt or any other hashing algorithm works.

Qu: What is Thread Pool ?

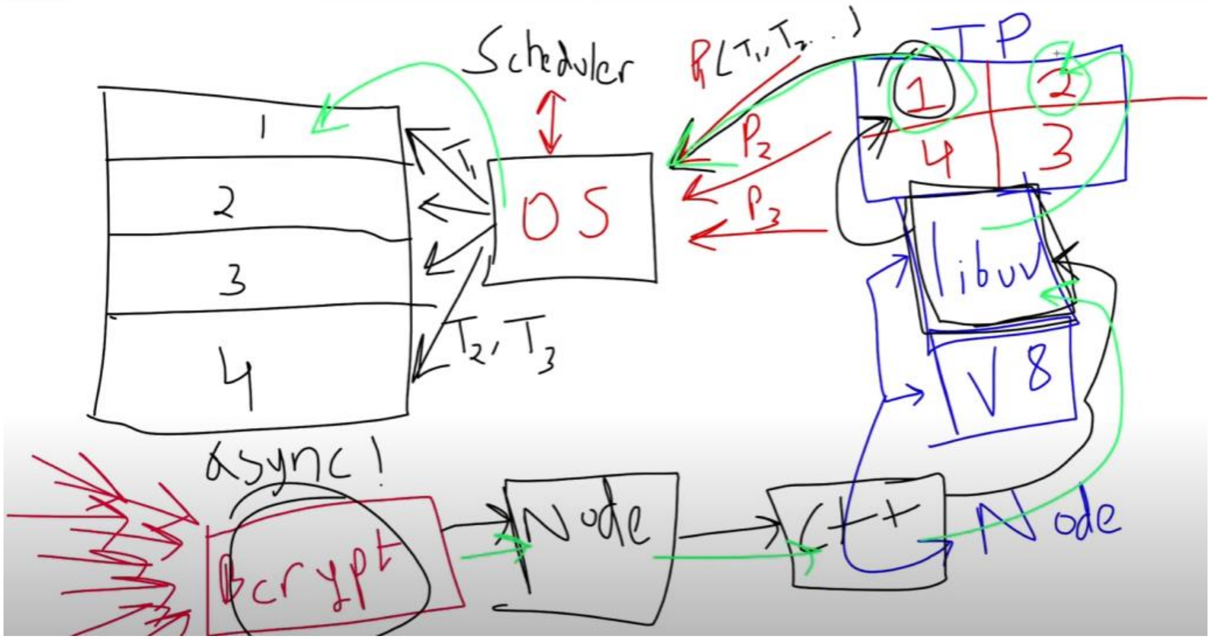


Users > dev > codedamn > youtube > advanced-node > thread.js > ...

```
process.env.UV_THREADPOOL_SIZE = 4

const http = require('http')
const bcrypt = require('bcrypt')

// Default = 3600 req/sec

// UV_THREADPOOL_SIZE=1 | 1030 req/sec
// UV_THREADPOOL_SIZE=2 | 2000 req/sec
// UV_THREADPOOL_SIZE=3 | 2900 req/sec
// UV_THREADPOOL_SIZE=4 | 3600 req/sec

http.createServer((req, res) => {
```

The default value of UV_THREADPOOL_SIZE IS 4.

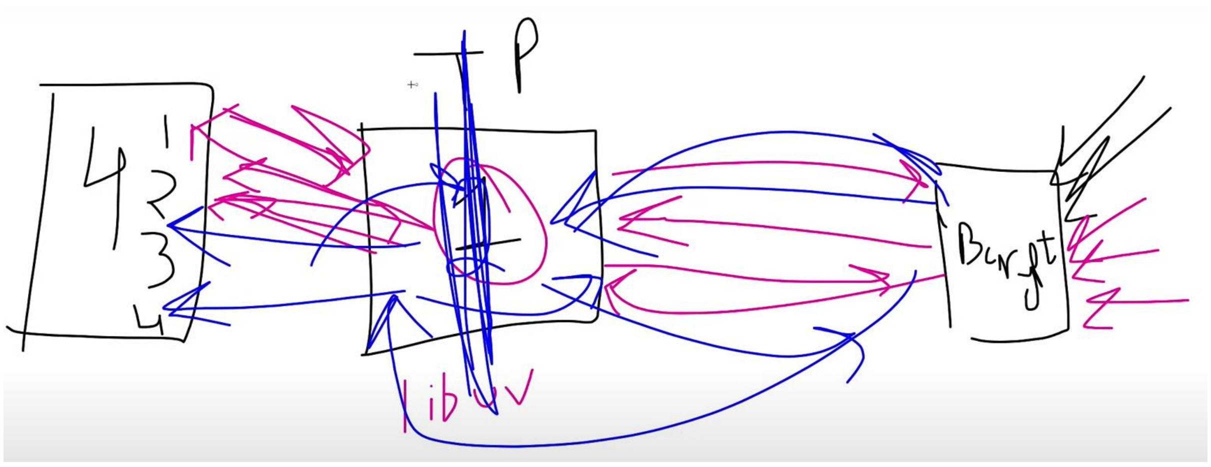
Lets say we have a 4 core system now we already know a little bit about scheduler, now you can think of scheduler as a special program by OS so the OS owns this scheduler. So schedular gets every other process or every other thread (P1,P2,P3) and then schedules their time on different different processors for execution and these processes could consists of t1,t2 and t3 and then the OS could divide and send this processes on different cores and if the OS wants its can juggle them around any Core as well if the OS wants to. Now with node.js what happens is that we know that node.js right here sits on V8 and libuv so the libuv is the component of the node.js execution on the whole thing which helps node.js to manage concurrent execution in basically this whole execution of things. So there is no relation as such between V8 and libuv, these are independent components but they can work together with node.js that is node.js , V8 and libuv works together to provide an asynchronous behaviour and asynchronous capabilities to the javascript which you write so libuv has something known as a thread pool and this thread pool actually consists of certain blocks of spaces so you might have the size of the thread pool be 4 which is exactly what we said with this UV_THREADPOOL_SIZE=4, Now once you set this number what happens is that whatever node.js has a certain asynchronous task which can be delegated to this particular thread pool which might require a thread pool , it gets that particular thread or it gets that particular slot in the thread pool. Now lets consider this bcrypt library so you see what we are doing is that we are bombarding this HTTP server with a lot of requests like 100 request 100 concurrent request for a thousand total request and this bcrypt is async which is the best part because what this bcrypt.hash can do is that it can say that hey I don't know what to do with this, javascript world does not know what to do with this let the CPP guys handle it. So bcrypt.hash actually delegates this hashing responsibility to the CPP world which is obviously fast but before actually delegating that (that is basically done by node.js) so bcrypt says to node.js hey I don't know anything at all , what to do, node.js says hey I don't even know in the javascript world but in the CPP world I do know so it contacts with its CPP world and the CPP world says hey I can do this but I can do this asynchronously as well so let me just go ahead and ask libuv to get a thread for me, so it goes to libuv and it says that hey libuv I need a thread for execution and libuv give a thread (thread number 1) from a thread pool now that this information that thread number 1 is now available for this particular process is conveyed to the operating system scheduler and then scheduler sends this particular thread to OS and then this OS assigns a processor to this particular thread now remember that this is happening very very fast because you are getting a lot of request parallely and when another request comes this cycle happens again.



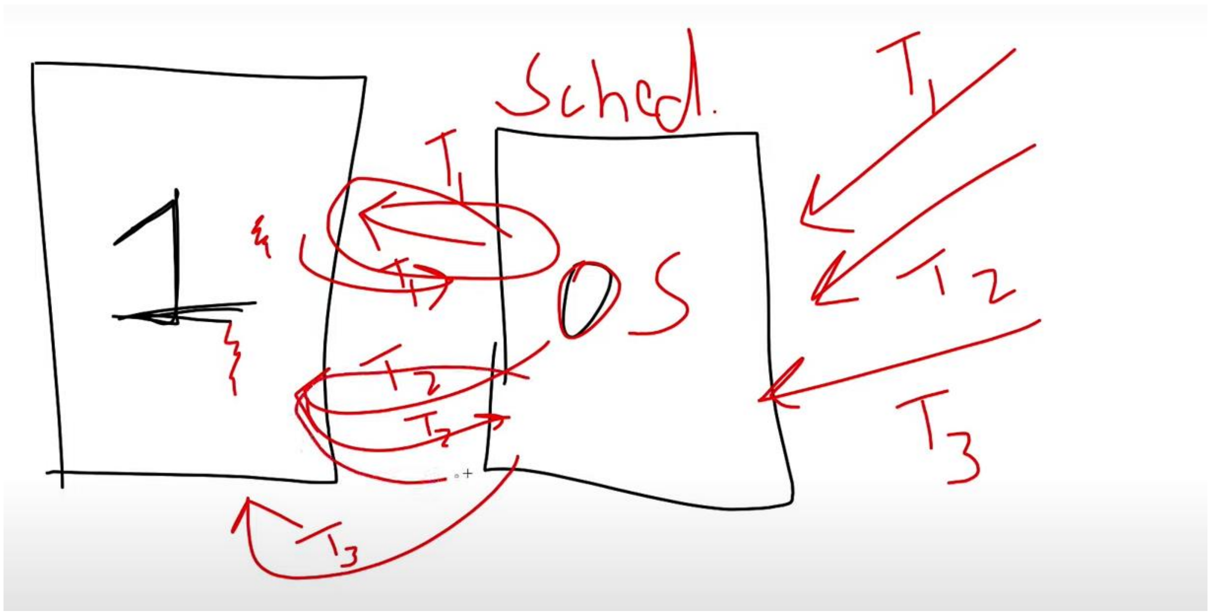
But Now what happens when we are not sending just 4 requests, we are sending a lot of requests because the thread pool is completely occupied so right now what happens is that libuv queues your execution so it says to t5 to wait because it does not have any empty space here all 4 are occupied and all 4 are executing at the moment so the t5 that is the 5th request has to wait for the thread pool to become empty now remember there are two components here , the thread pool from libuv and the scheduler from operating system, so the thread pool limits how many threads libuv is going to forward for execution to the operating system now it might seem a bad idea to have a restriction on the thread pool but it is actually a good thing , the t5 to lets say t50 request are waiting at the moment so you know lets say processor number 2 is done with its thread so process 2 becomes awakened and the t5 gets the next slot on process on this thread pool 2 and then this is forwarded to scheduler to gets assigned a new thread and the thing continues so this is how libuv is able to execute parallely a lot of computation with the help of multiple cores on your system.

Another Example:

Lets say we have a 4 core CPU and we have the thread pool from libuv which consists of only one thread at the moment and you have your node server for bcrypt which is getting bombarded with the request so now if the thread pool size of libuv is one that means only single execution can go on a single core it has to come back to bcrypt and then the next one can go so we can see that it is not a parallel execution anymore if the thread pool size is one that is why when you switch to a libuv thread pool size 1 and when you restart the server and when you try to run the apache benchmark test you see it has such a bad performance because now your libuv is actually always waiting for the execution to complete on a single processor even though our system has multiple cores and now when you make it two then obviously you have this two areas of processing so now you can process two requests at a time ideally if you have a lot of requests pending and you can distribute them on two cores.



Qu: Optimum threadpool size ?



Lets understand why choosing the uv thread pool size greater than your logical cores is probably a bad idea so lets just start with the analogy of just a single core processor now you can technically run multiple applications on a single core processor as well because this OS scheduler sits in front of your hardware and the processes we have lets say p1,p2,p3 and the scheduler gets t1,t2,t3 so what its going to do its going to divide the time for this processes , first its going to send t1 to execute for some time then the scheduler is going to bring it back the scheduler then its going to send t2 to execute its going to execute on this processor for some time then its going to bring back t2 then its going to send t3 so what is happening here in the switching of the threads there is certain time which is getting wasted, when you send t1 , t1 gets executed then t1 is brought back by the scheduler now the sending of t1 information to processor and bringing it back and then sending of t2 is complete waste of time in terms of execution because that time your processor is idle when your scheduler is sending the t1 for the processor to execute when the scheduler is bringing back t1 and then sending t2 again that time is wasted no matter how small it is in nano seconds or anything but that time is wasted and if you have a lot of threads to execute you're going to see that if those threads have a similar sort of priority you're going to see that a lot of time will just be wasted because of this to and fro of threads and this is what is called as context switching in operating system so you switch the context , you switch the threads execution and this time which is wasted is the context switch time so what happens now assume that you have one processor, a scheduler and a thread pool with a very large size lets say a four size for a single core processor so whats happening here is that although your thread pool has empty slots which are bcrypt server (it is receiving lot of requests per seconds) a server will try to allocate these all 4 threads again and again whats just gonna happen is that libuv's thread pool is going to send all these four threads for execution to the scheduler and at the end scheduler decides which thread is going to execute because obviously if you just have a single processor you can not execute four thread even though if you have four slots in your thread pool because thread pool is a higher level abstraction in terms of hardware , scheduler is a lower level abstraction and the lowest is obviously the processor so the scheduler would not allow four threads of execution because it can not be done that's the whole point , you just have a single processor so how can you execute four threads on that so the scheduler will sant only a single thread and because it sees that well I have four threads of execution pending here, its going to take away this thread back after some time, its going to send a second thread and then its going to take away this thread back and then its going to send a third thread and so on and it will try to execute all four threads concurrently not parallely there's a difference. So the point is if we increase lot of threads inside your thread pool it would not help you to execute faster if the count of threads is more than the logical or the physical cores of your system because at the end of the day the processor can not execute those many threads anyways so the scheduler has to send a single thread and take away that thread after some time so eventually you lose that time of context switching if you did not do that , that means you keep on executing a single thread that way you will only lose the time in the context switch whenever the thread work is actually done. If you have three threads , you just lose that much time but in the case when you send all the threads in the terms of thread pools you lose so much time like this. We should optimize the thread pool size in such a way that it should not exceed the logical cores because if it exceeds your logical cores, your scheduler will kick in and it will obviously not allow you to execute those many threads parallelly so at a given time always some of the threads given that all of the threads are executing some of the threads will be sitting idle for context switch to happen and when the context switch happens your CPU wastes time so that's why it is a good idea to restrict it to at max the amount of logical cores.

Qu: What are Event Loops ?

The event loop in the nodejs as well as the javascript present in the browser are the same but there are few exceptions.

We need Event loop to actually schedule the events , we don't want to run everything at once and block everything.

Qu: What is Call Stack ?

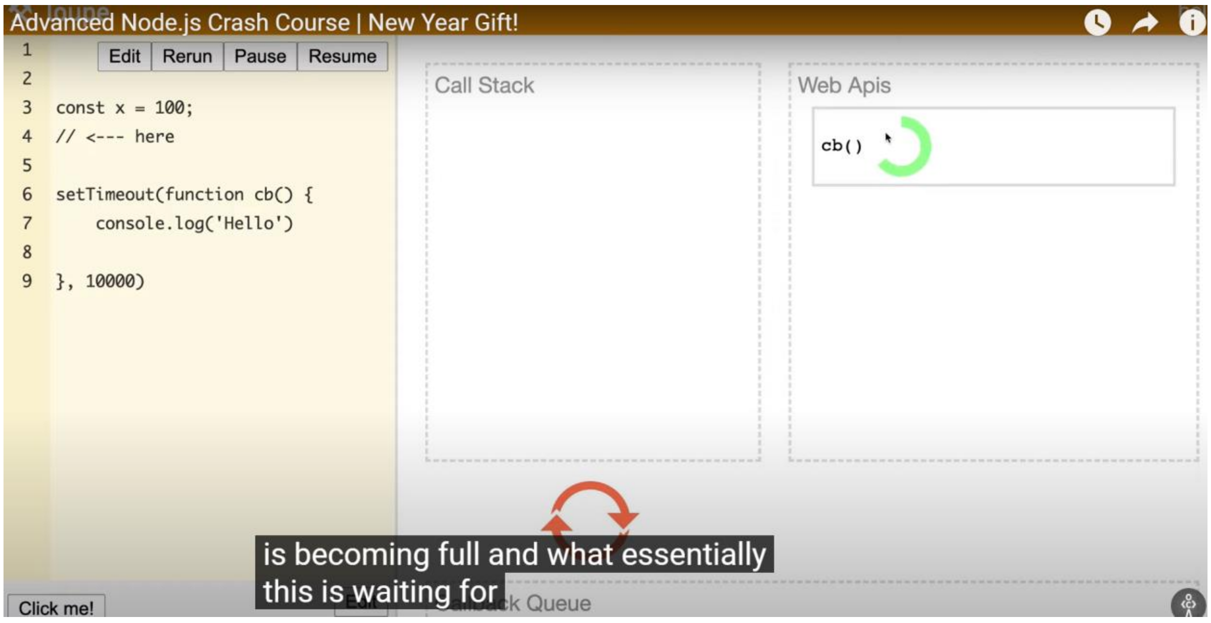
The Call Stack basically means that whenever you call a function it goes on to the call stack and as long as that function is executing that means as long as that function has not returned anything that function would be on the call stack.

Qu: What is internal web APIs?

There is nothing anywhere any such section in the code instead this is just used for categorizing web apis which are external in this tool. The call stack is actually a real thing. The internal web APIs are the APIs which are provided by the browsers and not by the V8 like setTimeout, setInterval etc.

Qu: What is Task Queues ?

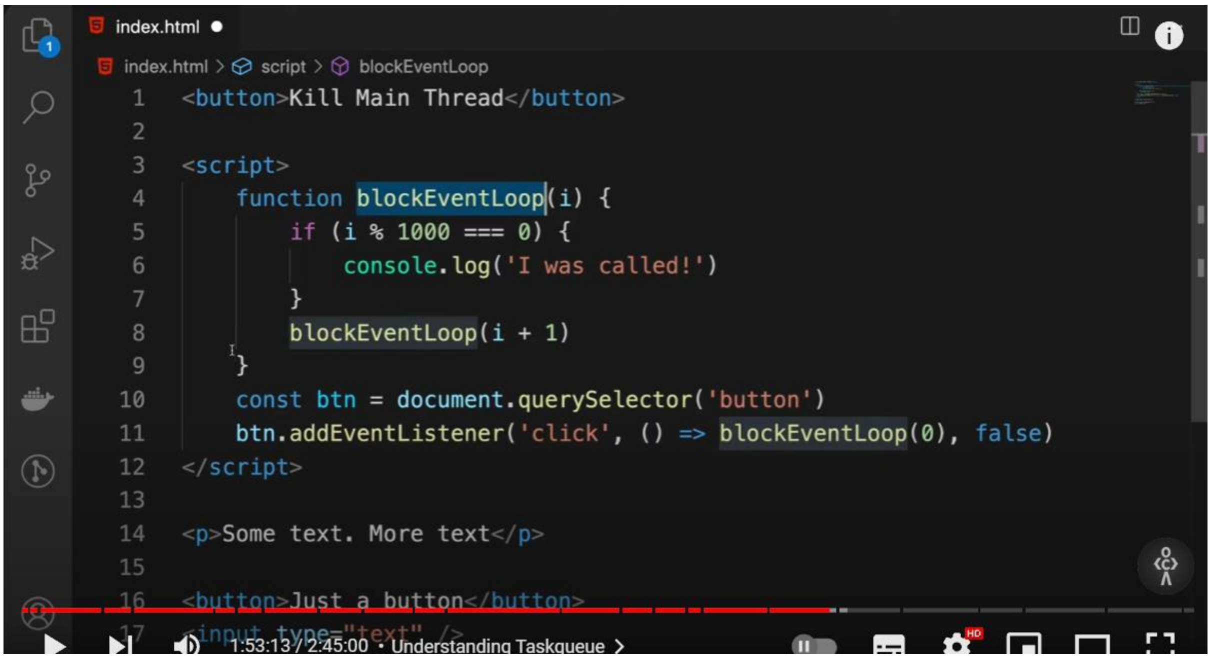
When we use some web api like setTimeout or setInterval, what's going to happen, we may think that its going to wait 10 seconds and then its going to run the callback function well not exactly to be honest. When javascript starts executing this function, the javascript will realise that setTimeout is something which v8 can not handle internally so web APIs will instruct the call stack to mark the setTimeout as done even though this function is not executed, never executed and the time limit never expired so the setTimeout function will get cleared from the call stack at this point the call back is not executed but the timer has also not expired so at this point the callback function in web APIs will wait for the timer to expire So when you called setTimeout, it took this callback function out of the stack and it placed it probably under a timer in some other place now its waiting for this timer to expire and right now the call stack is empty that means if any other task is coming in javascript would be running just as usual so this waiting of timer and the task which might come up in call stack would be happening simultaneously and once you have that happening parallelly that means you're working with some sort of multi-threaded environment but still it is not multi-threaded exactly because you as a developer never gets access to the different threads in which you know browser can play different games. And once this timer expires what this web api that is the browser or node would do is instead of pushing this callback function directly on the call stack , it queues that function inside the task queue, by queuing what is gives javascript is the ability to pick a function when it warns it is very much possible that when that timer expired the call stack, the main thread of javascript is busy with some other things so then at that point this web apis and what this internal structure supposed to do , it can just go ahead and push it all the way at the top which would interrupt the javascript execution and this would start the callback thing and would create a disaster of race condition or you can let the V8 that is the call stack in this case just free up a little and then decide what it has to do so it does exactly that, in javascript we get a task queue in which all the stuff like timer based or not really fixed or asynchronous like network request and file reading stuff like that is placed directly inside the task queue when its done and then its left onto V8 when it wants to pick up these functions from task queue and then general rule of thumb is V8 would always pick the first item from the task queue as soon as its call stack is empty. V8 is not going to pick up the functions if it already has something running on the call stack i.e even if a single function is present in call stack, V8 is not going to pick up that task queue function.



```

const x = 100;
// <--- here

setTimeout(function cb() {
    console.log('Hello')

}, 10000)
```

Note: The javascript which you can work with is always single threaded so you as a developer never gets explicit access to the multi-threaded world of javascript that is the underlying C and C++ architecture on which the V8 and browsers work but that does not mean that browsers and operating systems themselves can not make use of multi-threading in order to improve support for the single threaded architecture of javascript.

Example of task queue in use:

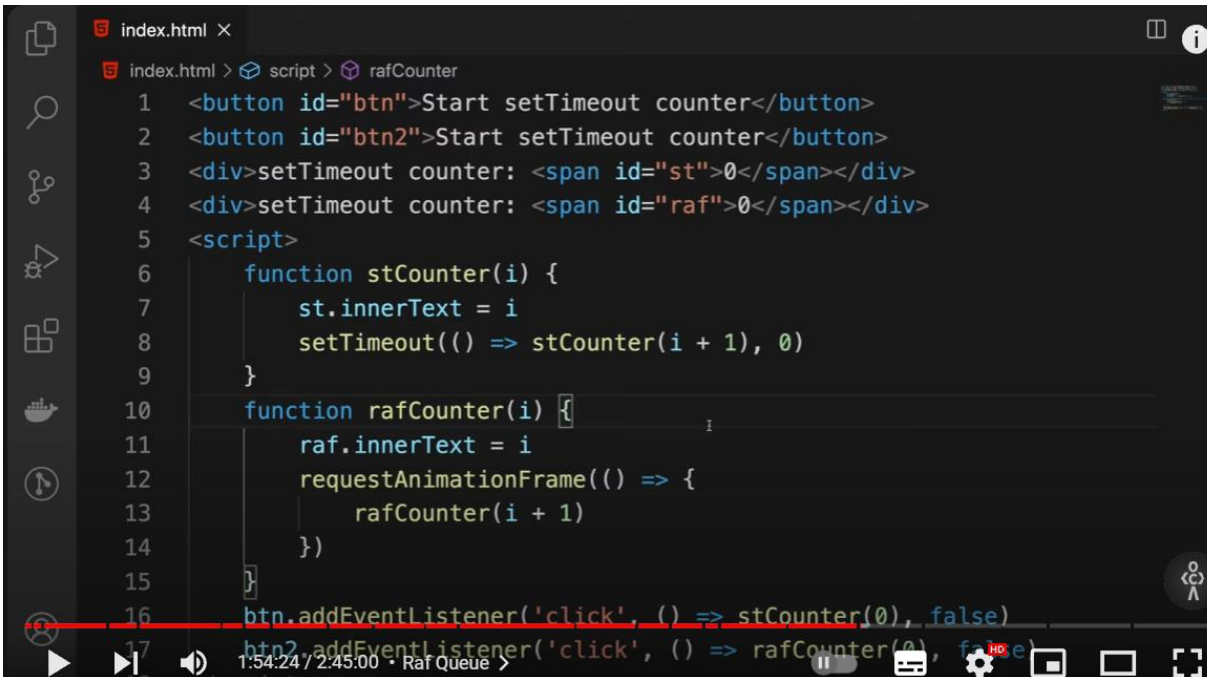


index.html > script > blockEventLoop

```
<button>Kill Main Thread</button>

<script>
    function blockEventLoop(i) {
        if (i % 1000 === 0) {
            console.log('I was called!')
        }
        blockEventLoop(i + 1)
    }
    const btn = document.querySelector('button')
    btn.addEventListener('click', () => blockEventLoop(0), false)
</script>

<p>Some text. More text</p>

<button>Just a button</button>
<input type="text" />
```

Qu: What is Raf queue ?

Raf is know as request animation frame which is closely related to task queue but its in a different place than the actual task queue where the functions happens. Raf queue is also known as animation queue.



index.html > script > rafCounter

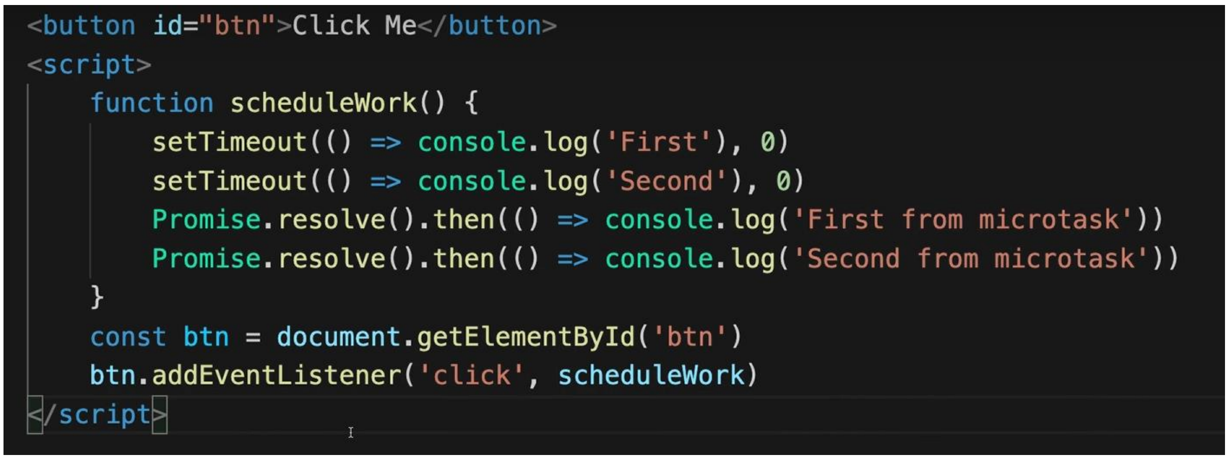
```
<button id="btn">Start setTimeout counter</button>
<button id="btn2">Start setTimeout counter</button>
<div>setTimeout counter: <span id="st">0</span></div>
<div>setTimeout counter: <span id="raf">0</span></div>
<script>
    function stCounter(i) {
        st.innerText = i
        setTimeout(() => stCounter(i + 1), 0)
    }
    function rafCounter(i) {
        raf.innerText = i
        requestAnimationFrame(() => {
            rafCounter(i + 1)
        })
    }
    btn.addEventListener('click', () => stCounter(0), false)
    btn2.addEventListener('click', () => rafCounter(0), false)
```

The difference between task queue and animation queue is the point where the browser runs the particular piece of callback. The callback function from the raf queue (requestAnimationFrame) is slightly slower then the task queue's callback function. The requestAnimationFrame uses another queue which is not the task queue instead it uses an animation queue. The callback function inside the requestAnimationFrame would not execute on every time the call stack gets empty instead it is a function which executes just before the browser is rendering , just before the browser is updating your screen, so What happens is that browsers do not update continuously on your screen there is a thing called 60 frames per second it means that your browser in this case is refreshing your display at that rate i.e 60 times a second and whats happening with the setTimeout however is that you are refreshing it as soon as the call stack becomes empty and the computers under the hood actually operates at a very fast speed i.e millions of operations per second however your display does not operates at millions of operations , millions of times per second and we don't want to waste the computations because the human eyes can not perceive the refresh rate more that 140 frames per seconds. So if you are trying to animate some sort of object then you're going to make use of request animation frame because this would allow you to spend less time on the parts where the user can not really see anything on the screen because it will just call this function not more than the refresh rate of your system but it does not guarantee that the callback function inside the requestAnimationFrame would run 60 times a second but if the system is not under load and everything is fine then it will just run 60 time a second only, the requestAnimationFrame can slow down if you are doing a lot of work on javascript's other threads.

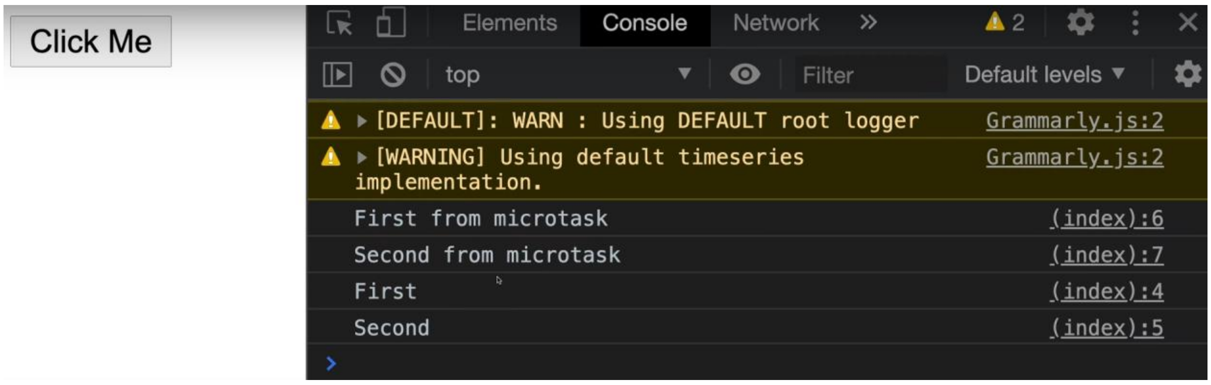
Qu: Introduction to Microtask Queue ?

We know that when we use setTimeout or setInterval those things are scheduled as tasks in the task queue. There is another queue Microtask queue which is very similar to task queue but it differs in a couple of ways which is going to make a lot of difference.

The promises work on the microtask queue. When you call .then() at any promise than at that particular time whenever the promise is resolved its going to place that function on the microtask queue and its going to execute whenever the event loop gets some time to execute the microtask queues.



```
<button id="btn">Click Me</button>
<script>
    function scheduleWork() {
        setTimeout(() => console.log('First'), 0)
        setTimeout(() => console.log('Second'), 0)
        Promise.resolve().then(() => console.log('First from microtask'))
        Promise.resolve().then(() => console.log('Second from microtask'))
    }
    const btn = document.getElementById('btn')
    btn.addEventListener('click', scheduleWork)
</script>
```

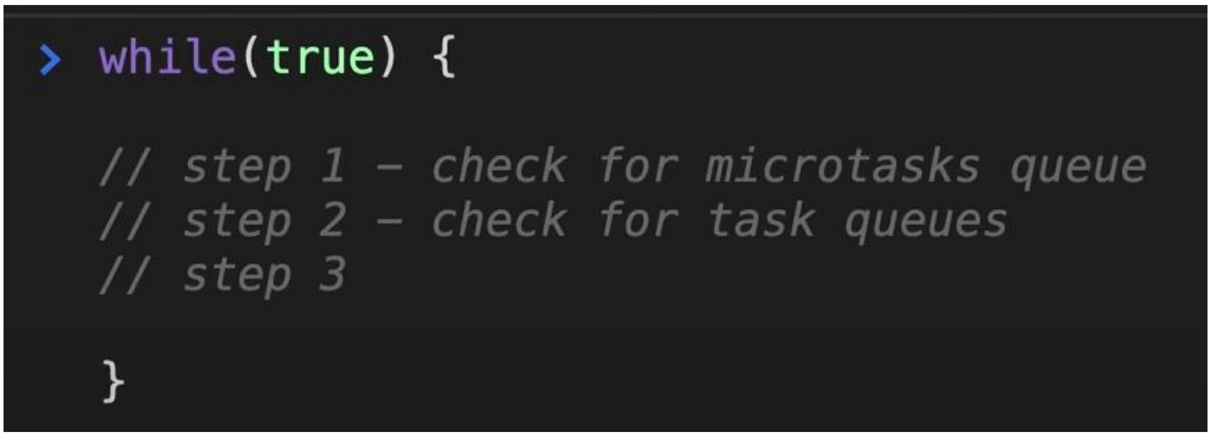


```
[DEFAULT]: WARN : Using DEFAULT root logger          Grammarly.js:2
[WARNING] Using default timeseries                   Grammarly.js:2
implementation.
First from microtask                                     (index):6
Second from microtask                                    (index):7
First                                                    (index):4
Second                                                   (index):5
```

From the above example, the first thing is very clear that you are going to see that you always get the microtask queue output before the task queue output and it is because when the event loop is running first is checks for the microtask queue and then on step 2 is checks for task queue. Point to note here is that microtask queues are always evaluated in the single event loop take it means that when this above code was being called what happened was setTimeout was called on task queue and then the promises are called on the microtask Queue and now what happens is when the event loop is running what's going to happen is that it would try to completely exhaust the micro task queue array so in the first step it will keep checking the microtask queues again and again just to see if there's any pending tasks or not and it will execute the microtask one by one but with the case of task queues, what really happens is that it runs only once per event loop cycle that means the next item in task queue will execute in the next cycle of event loop.

taskQueue = ['First', 'Second'];

microTaskQueue = ['First', 'Second'];



```
> while(true) {

    // step 1 - check for microtasks queue
    // step 2 - check for task queues
    // step 3

  }
```

Note: The microtask queue and the task queue are only be going to checked when the call stack is empty.