

# How, in general, does Node.js handle 10,000 concurrent requests?

Asked 8 years, 11 months ago   Modified 2 years ago   Viewed 254k times



726



I understand that Node.js uses a single-thread and an event loop to process requests only processing one at a time (which is non-blocking). But still, how does that work, lets say 10,000 concurrent requests. The event loop will process all the requests? Would not that take too long?

I can not understand (yet) how it can be faster than a multi-threaded web server. I understand that multi-threaded web server will be more expensive in resources (memory, CPU), but would not it still be faster? I am probably wrong; please explain how this single-thread is faster in lots of requests, and what it typically does (in high level) when servicing lots of requests like 10,000.

And also, will that single-thread scale well with that large amount? Please bear in mind that I am just starting to learn Node.js.

node.js

Share

Improve this question

Follow

edited Jun 17, 2019 at 11:05



Yilmaz

48.7k ● 18 ● 210 ● 264

asked Jan 18, 2016 at 12:56



g\_b

12.4k ● 12 ● 46 ● 86

16 Because most of the work (moving data around) doesn't involve the CPU. – OrangeDog Jan 18, 2016 at 12:58

16 Note also that just because there's only one thread executing Javascript, doesn't mean there aren't lots of other threads doing work. – OrangeDog Jan 18, 2016 at 12:58

This question is either too broad, or a duplicate of various other questions. – OrangeDog Jan 18, 2016 at 13:04

[stackoverflow.com/questions/3629784/...](https://stackoverflow.com/questions/3629784/...) – Shanoor Jan 18, 2016 at 13:12

Along with single threading, Node.js does something called as "non blocking I/O". Here is where all the magic is done – Anand N Nov 26, 2019 at 10:31

9 Answers

Sorted by: Highest score (default)



1311

If you have to ask this question then you're probably unfamiliar with what most web applications/services do. You're probably thinking that all software do this:



```
user do an action
  |
  v
application start processing action
  └─> loop ...
        └─> busy processing
end loop
  └─> send result to user
```

However, this is not how web applications, or indeed any application with a database as the back-end, work. Web apps do this:

```
user do an action
  |
  v
application start processing action
  └─> make database request
        └─> do nothing until request completes
request complete
  └─> send result to user
```

In this scenario, the software spend most of its running time using 0% CPU time waiting for the database to return.

## Multithreaded network app:

Multithreaded network apps handle the above workload like this:

```
request ─> spawn thread
           └─> wait for database request
                 └─> answer request
request ─> spawn thread
           └─> wait for database request
                 └─> answer request
request ─> spawn thread
           └─> wait for database request
                 └─> answer request
```

So the thread spend most of their time using 0% CPU waiting for the database to return data. While doing so

they have had to allocate the memory required for a thread which includes a completely separate program stack for each thread etc. Also, they would have to start a thread which while is not as expensive as starting a full process is still not exactly cheap.

## Singlethreaded event loop

Since we spend most of our time using 0% CPU, why not run some code when we're not using CPU? That way, each request will still get the same amount of CPU time as multithreaded applications but we don't need to start a thread. So we do this:

```
request —> make database request  
request —> make database request  
request —> make database request  
database request complete —> send response  
database request complete —> send response  
database request complete —> send response
```

In practice both approaches return data with roughly the same latency since it's the database response time that dominates the processing.

The main advantage here is that we don't need to spawn a new thread so we don't need to do lots and lots of malloc which would slow us down.

## Magic, invisible threading

The seemingly mysterious thing is how both the approaches above manage to run workload in "parallel"? The answer is that the database is threaded. So our single-threaded app is actually leveraging the multi-threaded behaviour of another process: the database.

## **Where singlethreaded approach fails**

A singlethreaded app fails big if you need to do lots of CPU calculations before returning the data. Now, I don't mean a for loop processing the database result. That's still mostly  $O(n)$ . What I mean is things like doing Fourier transform (mp3 encoding for example), ray tracing (3D rendering) etc.

Another pitfall of singlethreaded apps is that it will only utilise a single CPU core. So if you have a quad-core server (not uncommon nowadays) you're not using the other 3 cores.

## **Where multithreaded approach fails**

A multithreaded app fails big if you need to allocate lots of RAM per thread. First, the RAM usage itself means you can't handle as many requests as a singlethreaded app. Worse, malloc is slow. Allocating lots and lots of objects (which is common for modern web frameworks) means we can potentially end up being slower than singlethreaded apps. This is where node.js usually win.

One use-case that end up making multithreaded worse is when you need to run another scripting language in your thread. First you usually need to malloc the entire runtime for that language, then you need to malloc the variables used by your script.

So if you're writing network apps in C or go or java then the overhead of threading will usually not be too bad. If you're writing a C web server to serve PHP or Ruby then it's very easy to write a faster server in javascript or Ruby or Python.

## Hybrid approach

Some web servers use a hybrid approach. Nginx and Apache2 for example implement their network processing code as a thread pool of event loops. Each thread runs an event loop simultaneously processing requests single-threaded but requests are load-balanced among multiple threads.

Some single-threaded architectures also use a hybrid approach. Instead of launching multiple threads from a single process you can launch multiple applications - for example, 4 node.js servers on a quad-core machine. Then you use a load balancer to spread the workload amongst the processes. The [cluster](#) module in node.js does exactly this.

In effect the two approaches are technically identical mirror-images of each other.

answered Jan 18, 2016 at 14:37

[slebetman](#)

114k ● 19 ● 146 ● 176

---

206 This is by far the best explanation for node I have read so far. That "single-threaded app is actually leveraging the multi-threaded behaviour of another process: the database." did the work – [yashpandey8055](#) Feb 26, 2017 at 18:20

---

6 @CaspainCaldion It depends on what you mean by very fast and lots of clients. As is, node.js can process upwards of 1000 requests per second and speed limited only to the speed of your network card. Note that it's 1000 requests per second not clients connected simultaneously. It can handle the 10000 simultaneous clients without issue. The real bottleneck is the network card. – [slebetman](#) Apr 4, 2017 at 5:40

---

3 @slebetman , best explanation ever. one thing though, if i have a Machine Learning algorithm which processes some info and delivers results accordingly , should i use Multi threaded approach or single threaded – [Ganesh Karewad](#) Aug 25, 2017 at 7:02

---

11 @GaneshKarewad Algorithms use CPU, services (database, REST API etc.) use I/O. If the AI is an algorithm written in js then you should run it in another thread or process. If the AI is a service running on another computer (like Amazon or Google or IBM AI services) then use a single threaded architecture. – [slebetman](#) Aug 25, 2017 at 7:07

---

5 @VikasDubey Note that video streams are not processed by the node server at all. 99% of the processing happens in the browser/media player. In this case node.js serves as nothing more than a file server which means it leverages on the OS's parallel disk/network I/O capabilities. For network I/O most OSes are equally capable but for disk I/O Linux tends to trump everyone else if used with fast filesystems like Btrfs or ext4 (of course, RAID makes almost everything fast) – [slebetman](#) Sep 4, 2019 at 8:00

---



112



What you seem to be thinking is that most of the processing is handled in the node event loop. Node actually farms off the I/O work to threads. I/O operations typically take orders of magnitude longer than CPU operations so why have the CPU wait for that? Besides, the OS can handle I/O tasks very well already. In fact, because Node does not wait around it achieves much higher CPU utilisation.

By way of analogy, think of NodeJS as a waiter taking the customer orders while the I/O chefs prepare them in the kitchen. Other systems have multiple chefs, who take a customers order, prepare the meal, clear the table and only then attend to the next customer.

Share Improve this answer

[edited Nov 21, 2016 at 16:52](#)

Follow

answered Jan 18, 2016 at 13:51



[chriskelly](#)

7,736 ● 3 ● 34 ● 52




---

21 Thanks for the restaurant analogy! I find analogies and real-world examples so much easier to learn from. – [LaVache](#) Oct 7, 2017 at 0:23

---

js part of node is single threaded while the c++ part underlying is using a thread pool .

[stackoverflow.com/a/70161215/4034825](https://stackoverflow.com/a/70161215/4034825) – [mercury](#) Nov 30, 2021 at 5:15 

---

1 In restaurant example: what if we have 10000 customer with one waiter? isn't it slow to handle these customers with one waiter(one thread)? – [Ali Sherafat](#) Feb 1, 2022 at 12:48

---

2 @AliSherafat a bit late but if you're asking if the waiter can be overworked, the answer is yes. Generally, I/O operations are hundreds of time slower than in memory CPU managed operations. i.e. the real threads (the chefs) are doing a lot more work per single request taken by the waiter. Of course in real-life, 10000 customers is too much for a waiter but managing the forwarding and returning 10000 requests might not be. Out of the box this is how node works with the event loop but there are ways to use more waiter threads (e.g. worker threads). – [chriskelly](#) Feb 1, 2023 at 6:02

---

@chriskelly in the restaurant example the waiter is also responsible for cashier activities(simulating CPU work) so when more customers come in response time will increase or serving time will increase – [Lau](#) Mar 21, 2023 at 5:00

---



## Single Threaded Event Loop Model Processing Steps:

52



- Clients Send request to Web Server.
- Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client



Requests.



- Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.
- Node JS Web Server internally has a Component, known as “Event Loop”. Why it got this name is that it uses indefinite loop to receive requests and process them.
- Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.
- Event Loop checks any Client Request is placed in Event Queue. If not then wait for incoming requests for indefinitely.
- If yes, then pick up one Client Request from Event Queue
  1. Starts process that Client Request
  2. If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.
  3. If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
- Checks Threads availability from Internal Thread Pool

- Picks up one Thread and assign this Client Request to that thread.
- That Thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop

very nicely explained by @Rambabu Posa for more explanation go throw this [Link](#)

Share Improve this answer

Follow

edited Oct 21, 2019 at 5:51



Piyush Balapure

1,131 ● 9 ● 14

answered May 7, 2019 at 6:11



sudheer nunna

1,719 ● 15 ● 17

---

2 the diagram given in that blog post seems to be wrong, what they have mentioned in that article is not completely correct.  
– [rranj](#) Feb 9, 2020 at 8:50

---

3 AFAIK, there's no blocking I/O in node (unless you use sync APIs), the thread pool is only used momentarily to handle I/O responses and deliver them to the main thread. But, while waiting the I/O request [there's no thread]([blog.stephencleary.com/2013/11/there-is-no-thread.html](http://blog.stephencleary.com/2013/11/there-is-no-thread.html)) , otherwise the thread pool would get clogged pretty quickly.  
– [pmoleri](#) Feb 13, 2021 at 2:21

---

i cleared the confusion

[stackoverflow.com/a/70161215/4034825](https://stackoverflow.com/a/70161215/4034825) – [mercury](#) Nov 30, 2021 at 5:19

---

I think the last three steps are only true for non blocking i/o. It doesnt check thread pool for threads when you have



22



I understand that Node.js uses a single-thread and an event loop to process requests only processing one at a time (which is non-blocking).

I could be misunderstanding what you've said here, but "one at a time" sounds like you may not be fully understanding the event-based architecture.

In a "conventional" (non event-driven) application architecture, the process spends a lot of time sitting around waiting for something to happen. In an event-based architecture such as Node.js the process doesn't just wait, it can get on with other work.

For example: you get a connection from a client, you accept it, you read the request headers (in the case of http), then you start to act on the request. You might read the request body, you will generally end up sending some data back to the client (this is a deliberate simplification of the procedure, just to demonstrate the point).

At each of these stages, most of the time is spent waiting for some data to arrive from the other end - the actual time spent processing in the main JS thread is usually fairly minimal.

When the state of an I/O object (such as a network connection) changes such that it needs processing (e.g. data is received on a socket, a socket becomes writable, etc) the main Node.js JS thread is woken with a list of items needing to be processed.

It finds the relevant data structure and emits some event on that structure which causes callbacks to be run, which process the incoming data, or write more data to a socket, etc. Once all of the I/O objects in need of processing have been processed, the main Node.js JS thread will wait again until it's told that more data is available (or some other operation has completed or timed out).

The next time that it is woken, it could well be due to a different I/O object needing to be processed - for example a different network connection. Each time, the relevant callbacks are run and then it goes back to sleep waiting for something else to happen.

The important point is that the processing of different requests is interleaved, it doesn't process one request from start to end and then move onto the next.

To my mind, the main advantage of this is that a slow request (e.g. you're trying to send 1MB of response data to a mobile phone device over a 2G data connection, or you're doing a really slow database query) won't block faster ones.

In a conventional multi-threaded web server, you will typically have a thread for each request being handled, and it will process ONLY that request until it's finished. What happens if you have a lot of slow requests? You end up with a lot of your threads hanging around processing these requests, and other requests (which might be very simple requests that could be handled very quickly) get queued behind them.

There are plenty of others event-based systems apart from Node.js, and they tend to have similar advantages and disadvantages compared with the conventional model.

I wouldn't claim that event-based systems are faster in every situation or with every workload - they tend to work well for I/O-bound workloads, not so well for CPU-bound ones.

Share Improve this answer

answered Jan 18, 2016 at 14:54

Follow



sheltond

1,937 ● 12 ● 16

---

Good explanation , for understanding that event loop works for multiple requests simultaneously. – [mercury](#) Nov 11, 2021 at 16:34

---



20

Adding to slebetman answer: When you say `Node.js` can handle 10,000 concurrent requests they are essentially non-blocking requests i.e. these requests are majorly pertaining to database query.



Internally, `event loop` of `Node.js` is handling a `thread pool`, where each thread handles a `non-blocking request` and event loop continues to listen to more request after delegating work to one of the thread of the `thread pool`. When one of the thread completes the work, it send a signal to the `event loop` that it has finished aka `callback`. `Event loop` then process this callback and send the response back.

As you are new to NodeJS, do read more about `nextTick` to understand how event loop works internally. Read blogs on <http://javascriptissexy.com>, they were really helpful for me when I started with JavaScript/NodeJS.

Share Improve this answer

answered Jan 18, 2016 at 14:56

Follow



Aman Gupta

3,797 ● 4 ● 46 ● 67



19

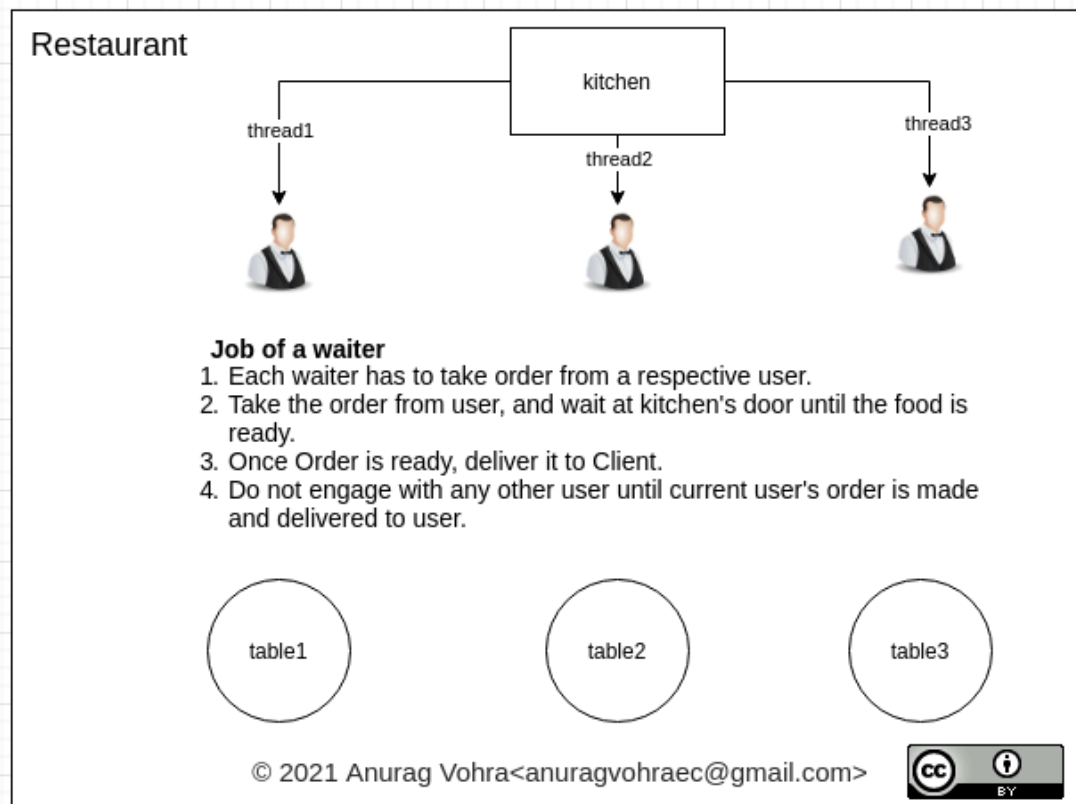


The **blocking part** of the multithreaded-blocking system makes it less efficient. The thread which is blocked cannot be used for anything else, while it is waiting for a response.

While a non-blocking single-threaded system makes the best use of its single-thread system.

See diagram below:

## How a multi-threaded blocking system work

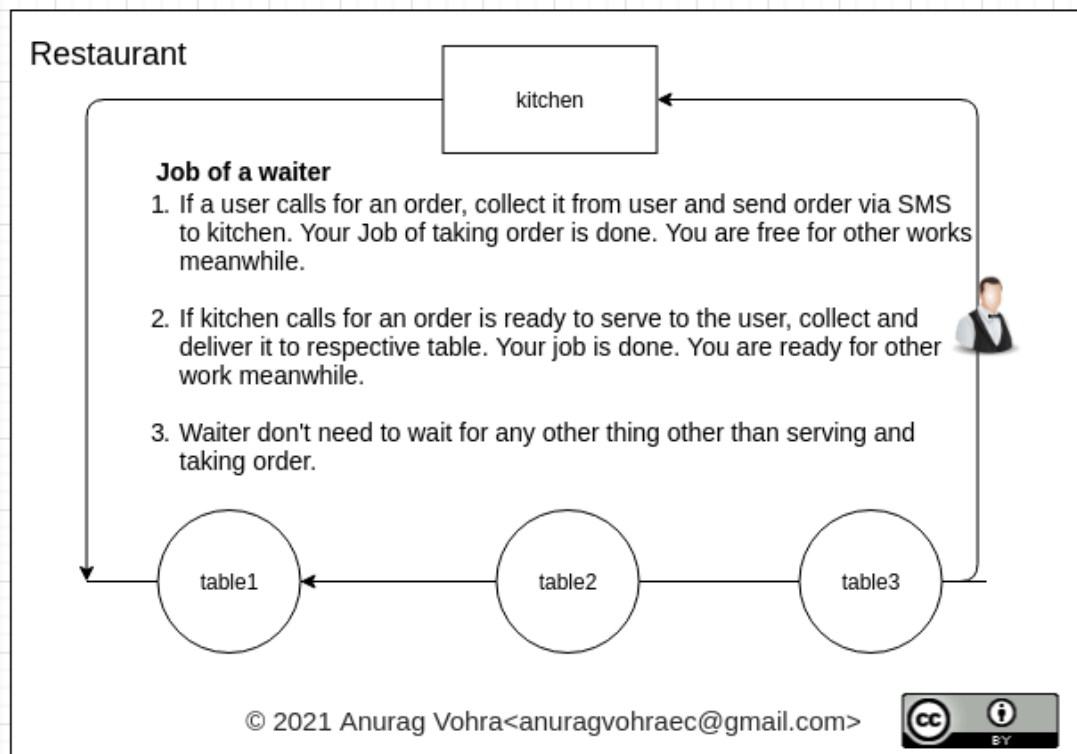


Here waiting at kitchen door or waiting while customer is selecting food items, is "Blocking" the full capacity of the waiter. In sense of Compute systems, it could be waiting for IO, or DB response or anything which blocks whole thread, even though the thread is capable of other works while waiting.

Let see how non blocking works:



## How a non blocking single thread system works



In a non blocking system, waiter only takes order and serve order, do not waits at anywhere. He shares his mobile number, to give a **call back** when they have finalzed the order. Similarly he shares his number with Kitchen to callback when order is ready to serve.

This is how Event loop works in NodeJS, and performs better than blocking multithreaded system.

Share Improve this answer

edited Mar 16, 2022 at 3:42

Follow

answered Oct 13, 2021 at 15:01



Anurag Vohra

1,973 ● 1 ● 17 ● 34

How does the client not timeout if it doesn't have it's own thread though? Is there some sort of queue that the callbacks

are added to? And then this queue is worked through quickly if the code is designed well and doesn't block the event loop.  
– [MattieG4](#) Mar 9, 2022 at 17:10

---

@MattieG4 Client after passing callback continues to do other things. Now its job of callack reciver to call the callback once jobs is done. And yes event loop has a queue strutcture, where in task are added and served in FIFO basis. If any blocking code is in the queue, it will delay every item/task in the queue. – [Anurag Vohra](#) Mar 10, 2022 at 3:57

---



Adding to **slebetman**'s answer for more clarity on what happens while executing the code.

14



The internal thread pool in nodeJs just has 4 threads by default. and its not like the whole request is attached to a new thread from the thread pool the whole execution of request happens just like any normal request (without any blocking task) , just that whenever a request has any long running or a heavy operation like db call ,a file operation or a http request the task is queued to the internal thread pool which is provided by libuv. And as nodeJs provides 4 threads in internal thread pool by default every 5th or next concurrent request waits until a thread is free and once these operations are over the callback is pushed to the callback queue. and is picked up by event loop and sends back the response.

Now here comes another information that its not once single callback queue, there are many queues.

## 1. NextTick queue

2. Micro task queue
3. Timers Queue
4. IO callback queue (Requests, File ops, db ops)
5. IO Poll queue
6. Check Phase queue or SetImmediate
7. close handlers queue

Whenever a request comes the code gets executing in this order of callbacks queued.

It is not like when there is a blocking request it is attached to a new thread. There are only 4 threads by default. So there is another queueing happening there.

Whenever in a code a blocking process like file read occurs , then calls a function which utilises thread from thread pool and then once the operation is done , the callback is passed to the respective queue and then executed in the order.

Everything gets queued based on the the type of callback and processed in the order mentioned above.

Share Improve this answer

Follow

edited Jun 8, 2021 at 18:48



OfirD

10.4k ● 7 ● 55 ● 97

answered Feb 6, 2020 at 9:08



rranj

288 ● 1 ● 5 ● 17

---

The internal thread pool in nodeJs just has 4 threads by default ...Could you elaborate? How is node use 4 threads ?

– [mercury](#) Nov 11, 2021 at 16:44 ✎

---

- 1 NodeJs internally uses a few threads (4 by default) for the purpose of connecting with DB, reading file from disk, Interacting with OS etc. You can increase this count of threads by using the environment variable 'UV\_THREADPOOL\_SIZE' by setting it to a different number while running your nodejs app. – [rranj](#) Nov 16, 2021 at 6:13
- 



8



Here is a good explanation from this [medium article](#):

Given a NodeJS application, since Node is single threaded, say if processing involves a Promise.all that takes 8 seconds, does this mean that the client request that comes after this request would need to wait for eight seconds? No. NodeJS event loop is single threaded. The entire server architecture for NodeJS is not single threaded.

Before getting into the Node server architecture, to take a look at typical multithreaded request response model, the web server would have multiple threads and when concurrent requests get to the webserver, the webserver picks threadOne from the threadPool and threadOne processes requestOne and responds to clientOne and when the second request comes in, the web server picks up the second thread from the threadPool and picks up requestTwo and processes it and responds to clientTwo. threadOne is responsible for all kinds of operations that

requestOne demanded including doing any blocking IO operations.

The fact that the thread needs to wait for blocking IO operations is what makes it inefficient. With this kind of a model, the webserver is only able to serve as much requests as there are threads in the thread pool.

NodeJS Web Server maintains a limited Thread Pool to provide services to client requests. Multiple clients make multiple requests to the NodeJS server. NodeJS receives these requests and places them into the EventQueue . NodeJS server has an internal component referred to as the EventLoop which is an infinite loop that receives requests and processes them. This EventLoop is single threaded. In other words, EventLoop is the listener for the EventQueue. So, we have an event queue where the requests are being placed and we have an event loop listening to these requests in the event queue. What happens next? The listener(the event loop) processes the request and if it is able to process the request without needing any blocking IO operations, then the event loop would itself process the request and sends the response back to the client by itself. If the current request uses blocking IO operations, the event loop sees whether there are threads available in the thread pool, picks up one thread from the thread pool and assigns the particular request to the picked thread. That thread does the blocking IO operations and sends the response back to the event loop and once the response gets to the event

loop, the event loop sends the response back to the client.

How is NodeJS better than traditional multithreaded request response model? With traditional multithreaded request/response model, every client gets a different thread where as with NodeJS, the simpler request are all handled directly by the EventLoop. This is an optimization of thread pool resources and there is no overhead of creating the threads for every client request.

Share Improve this answer

answered May 19, 2021 at 14:31

Follow



[asif.ibtihaj](#)

391 ● 1 ● 5 ● 15

---

'then the event loop would itself process the request and sends the response back to the client by itself.' <- In this case, the cpu processes the request without the I/O initiation correct? And when using I/O the cpu initiates I/O to do the work then when the I/O is done, sends the cpu an update to send back to the client? – [Ralph Dingus](#) May 25, 2022 at 15:50

---

great explanation with correct statements. So the proof is: if you handle a while loop in an endpoint that blocks the "code" but does not block the main thread(event loop) it can't handle any requests. because the loop is not blocking the thread. Therefore, the main thread thinks that I can handle it without picking up a thread from the thread pool. If you call a non-blocking io it passes the task to a thread. – [Ali Berat Çetin](#) Aug 10, 2022 at 21:28

---



3



In node.js request should be IO bound not CPU bound. It means that each request should not force node.js to do a lot of computations. If there are a lot of computations involved in solving a request then node.js is not a good choice. IO bound has little computation required. most of the time requests are spent in either making a call to a DB or a service.

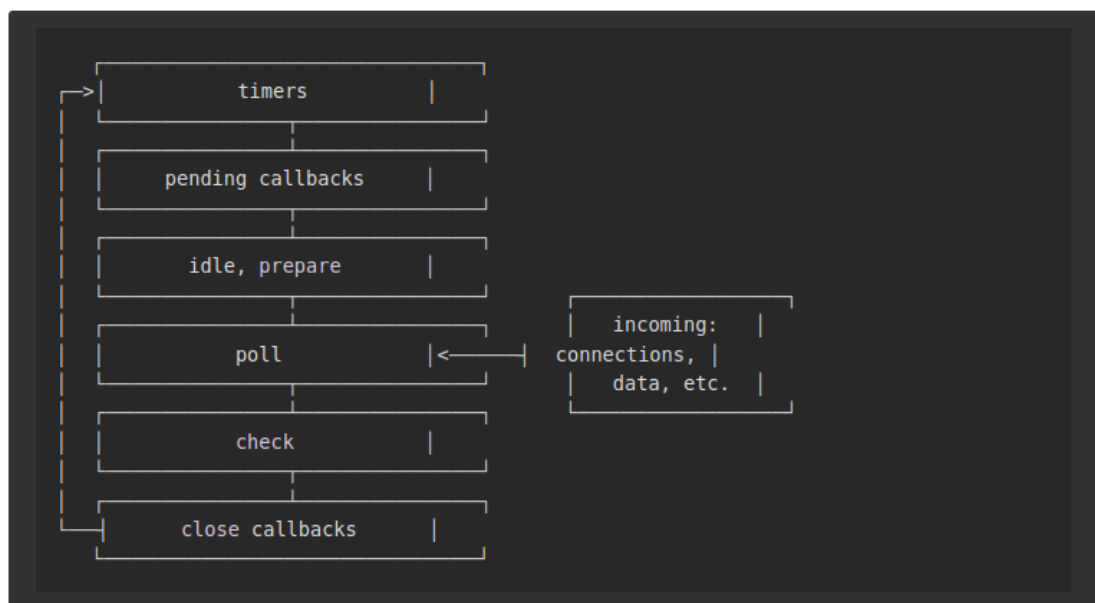
Node.js has single-threaded event loop but it is just a chef. Behind the scene most of the work is done by the operating system and Libuv ensures the communication from the OS. From the Libuv docs:

In event-driven programming, an application expresses interest in certain events and respond to them when they occur. The responsibility of gathering events from the operating system or monitoring other sources of events is handled by libuv, and the user can register callbacks to be invoked when an event occurs.

The incoming requests are handled by the Operating system. This is pretty much correct for almost all servers based on request-response model. Incoming network calls are queued in OS Non-blocking IO queue. 'Event Loop' constantly polls OS IO queue that is how it gets to know about the incoming client request. "Polling" means checking the status of some resource at a regular interval. If there are any incoming requests, evnet loop will take that request, it will execute that synchronously .

while executing if there is any async call (i.e `setTimeout`), it will be put into the callback queue. After the event loop finishes executing sync calls, it can poll the callbacks, if it finds a callback that needs to be executed, it will execute that callback. then it will poll for any incoming request. If you check the [node.js docs](https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#timers) there is this image:

The following diagram shows a simplified overview of the event loop's order of operations.



Each box will be referred to as a "phase" of the event loop.

## From docs phase-overview

**poll:** retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`); node will block here when appropriate.

So event loop is constantly polling from different queues. If any request needs to an external call or disk access, this is passed to OS and OS also has 2 different queues



for those. As soon as `event loop` detects that something has to be done async, it puts them in a queue. Once it is put in a queue, event-loop will process to the next task.

One thing that to mention here, event loop continuously runs. Only Cpu can move this thread out of CPU, event loop itself will not do it.

From the docs:

The secret to the scalability of Node.js is that it uses a small number of threads to handle many clients. If Node.js can make do with fewer threads, then it can spend more of your system's time and memory working on clients rather than on paying space and time overheads for threads (memory, context-switching). But because Node.js has only a few threads, you must structure your application to use them wisely.

Here's a good rule of thumb for keeping your Node.js server speedy: Node.js is fast when the work associated with each client at any given time is "small".

Note that small tasks mean IO bound tasks not CPU. Single `event loop` will handle the client load only if the work for each request is mostly IO work.

`Context switch` basically means CPU is out of resources so It needs to stop the execution of one process to allow

another process to execute. OS first has to evict process1 so it will take this process from CPU and it will save this process in the main memory. Next, OS will restore process2 by loading process control block from memory and it will put it on the CPU for execution. Then process2 will start its execution. Between process1 ended and the process2 started, we have lost some time. Large number of threads can cause a heavily loaded system to spend precious cycles on thread scheduling and context switching, which adds latency and imposes limits on scalability and throughput.

Share Improve this answer

edited Sep 27, 2022 at 1:16

Follow

answered Sep 27, 2022 at 0:57



Yilmaz

48.7k ● 18 ● 210 ● 264

---