Will async/await block a thread node.js

Asked 7 years, 3 months ago Modified 4 years, 2 months ago

Viewed 58k times

120

When `async/await` used in a node.js function, will it block the node.js thread untill it executes the next line of code ?

javascript node.js async-await

Share

Improve this question

Follow

edited Jul 6, 2019 at 18:35

Bergi
663k ● 158 ● 1k ● 1.5k

asked Sep 1, 2017 at 16:15

Rajesh
1,584 ● 2 ● 14 ● 15

Async/await has a synchronous behavior, so yes it will block the current respective execution flow until it is finished. – Lansana Camara Sep 1, 2017 at 16:16

16 no, it won't block the thread. other requests can continue coming in and getting handled while it is waiting. – Kevin B Sep 1, 2017 at 16:21

I would very much like clarification on @KevinB 's comment and this question : doesn't `await` '*block the current thread*' until the awaited operation is done, but allowing *other threads* to continue execution ? – Pac0 Sep 1, 2017 at 16:27

2 @Pac0 One Node.JS process has only one thread, when you `exec` or `fork`, a new process is formed and not a new thread. – Nidhin David Sep 1, 2017 at 16:29

If you think about it, if that's how it worked, why would we need `async/await`? We could just write blocking code instead. – Felix Kling Sep 1, 2017 at 16:51

6 Answers

Sorted by: Highest score (default)

240

`async/await` does not block the whole interpreter. node.js still runs all Javascript as single threaded and even though some code is waiting on an `async/await`, other events can still run their event handlers (so node.js is not blocked). The event queue is still being serviced for other events. In fact, it will be an event that resolves a promise that will allow the `await` to stop awaiting and run the following code.

Code like this:

```
await foo();            // foo is an async function th
console.log("hello");
```

is analogous to this:

```
foo().then(() => {
    console.log("hello");
});
```

So, `await` just puts the following code in that scope into an invisible `.then()` handler and everything else works pretty much the same as if it was actually written with a `.then()` handler.

So, `await` allows you to save the writing of the `.then()` handler and gives the code a synchronous look to it (though it isn't really synchronous). In the end it's a shorthand that lets you write async code with fewer lines of code. One does need to remember though that any promise that can reject must have a try/catch somewhere around it to catch and handle that rejection.

Logically, you can think of what node.js does when it encounters an `await` keyword when executing a function as the following:

1. Function call is made
2. The interpreter sees that the function is declared as `async` which means that it will always return a promise.
3. The interpreter starts executing the function.
4. When it encounters an `await` keyword, it suspends further execution of that function until the promise that is being awaited resolved.
5. The function then returns an unresolved promise.
6. At this point, the interpreter continues executing whatever comes after the function call (usually a `fn().then()` is followed by other lines of code). The

`.then()` handlers are not executed yet because the promise is not yet resolved.

7. At some point this sequence of Javascript finishes and returns control back to the interpreter.
8. The interpreter is now free to serve other events from the event queue. The original function call that ran into an `await` keyword is still suspended, but other events can be processed now.
9. At some future point, the original promise that was being awaited gets resolved. When it's time for that to get processed in the event queue, the previously suspended function continues executing on the line after the `await`. If there are any more `await` statements, then the function execution is again suspended until that promise resolves.
10. Eventually the function hits a `return` statement or reaches the end of the function body. If there is a `return xxx` statement, then the `xxx` is evaluated and its result becomes the resolved value of the promise that this `async` function has already returned. The function is now done executing and the promise it previously returned has been resolved.
11. This will cause any `.then()` handlers attached to the promise that this function previously returned to get called.
12. After those `.then()` handlers run, the job of this `async` function is finally done.

So, while the whole interpreter doesn't block (other Javascript events can still be serviced), the execution of the specific `async` function that contains the `await` statement was suspended until the promise that was being awaited resolved. What's important to understand is step 5 above. When the first `await` is hit, the function immediately returns an unresolved promise and code after this function is executed (before the promise being `awaited` is resolved). It's for this reason that the whole interpreter is not blocked. Execution continues. Only the insides of one function are suspended until a promise is resolved.

Share Improve this answer Follow

edited Jul 6, 2019 at 18:10
Michael Käfer
1,786 ● 2 ● 23 ● 40

answered Sep 1, 2017 at 16:47
jfriend00
706k ● 103 ● 1k ● 1k

6 I found v8.dev/blog/fast-async as a good reference to get the details of the actual process explained in this answer – Sasinda Rukshan Jan 29, 2019 at 20:51

1 Just to add, `await` has an underlying `yield` that is responsible for returning the flow to the caller. – testing_22 Feb 27 at 17:58

`async/await` provide an alternative way for what you would traditionally do with `then` calls on a promise. Nor Promises, nor `async` nor `await` create new threads.

30

When `await` is executed, the expression that follows it is evaluated synchronously. That expression should evaluate to a promise, but if it is not, it is wrapped into one, as if you had `await Promise.resolve(expression)`.

Once that expression is evaluated, the `async` function returns -- it returns a promise. Then code execution continues with whatever code follows that function call (same thread) until the call stack is empty.

At some point the promise -- that was evaluated for the `await` -- will resolve. This will put a microtask in a microtask queue. When the JavaScript engine has nothing more to do in the current task, it will consume the next event in the microtask queue (FIFO). As this microtask involves a resolved promise, it will restore the previous execution state of the `async` function and continue with whatever comes next after the `await`.

The function may execute other `await` statements, with similar behaviour, although the function now no longer returns to where it was originally called from (as that call was already processed with the first `await`), it merely returns leaving the call stack empty, and leaves the JavaScript engine to process the microtask and task queues.

All this happens with the same thread.

Share Improve this answer Follow

edited Oct 14, 2020 at 15:28

answered Sep 1, 2017 at 16:54

trincot
348k ● 35 ● 270 ● 320

how about this: stackoverflow.com/a/13823336/631527 `No, it is not OK to use a blocking API call in a node server` , what is the difference between async/await and synchronous versions? – Toolkit Oct 3, 2017 at 5:24

2 Code that really waits for something to happen without allowing code execution to continue elsewhere before that has happened, is synchronous, and ***blocking***. I have explained the behaviour of `async/await` which is not blocking as code continues to execute. – trincot Oct 3, 2017 at 18:00

5 FYI, `await` has some powers beyond syntactic sugar. For example, if `await` is inside a `while()` loop or a `for` loop, it will suspend that loop in a way that cannot be done without rewriting the code in an entirely different way that does not use that type of loop. So, while many garden variety uses are essentially syntactic sugar, it has capabilities beyond just that. – jfriend00 Jun 21, 2019 at 21:32

13

As long as the code contained inside the async/await is non blocking it won't block, for example db calls, network calls, filesystem calls.

But if the code contained inside async/await is blocking, then it will block the entire Node.js process, for example infinite loops, CPU intensive tasks like image processing, etc.

In essence async/await is a language level wrapper around Promises so that the code can have a synchronous 'look and feel'

Share Improve this answer

Follow

answered Sep 1, 2017 at 16:27

Nidhin David

2,474 ● 3 ● 32 ● 45

@Toolkit There's no way to know with complete certainty whether code is non-blocking, but if it doesn't return a promise or provide a callback, there is a 100% chance that it *is* blocking. – JLRishe Nov 27, 2017 at 4:05

@Toolkit you can determine this in runtime - try interact with app. Node-blocked or just simple inline test - trivial example is to execute `async` function with infinite loop inside `while(1)` and then try to log something with timeout `setTimeout( ()=>{ console.log(..) } )` - you will never see this log – fider Feb 14, 2018 at 19:48

---

1 @Nidhin David a database call is blocking isnt it? like if a query take 40 ms to execute, then event loop will be blocked for 40 ms right? – Sana.91 Jan 24, 2020 at 11:31

@Sana.91 In Node.js, no, it's asynchronous – Nidhin David Jan 25, 2020 at 15:10

---

2 Best answer I've seen in a long time on this subject. There's a lot of misunderstanding about async/await because most folks think that async. means non-blocking, but this is only true as you've specified if the code executes in an alternate thread or makes use of system calls that do. Most code that you write yourself - like a for/loop that does something, will indeed block, even in an async/await function. – Gary Aug 11, 2020 at 1:15

11

Will async/await block a thread node.js? As @Nidhin David said it depends on what code you have inside async function - db calls, network calls, filesystem calls are not blocking but blocking are for example long for/while cycles, JSON stringify/parse and evil/vulnerable regular expressions (google for ReDoS Attacks). Each of four examples below will block the main thread if `/test` request is called because of code `string.match(/^(a|a)+$/)` is synchronous and takes long time to process.

This first example will block main node thread as expected and no other requests/clients can be served.

```
var http = require('http');

// This regexp takes to long (if your PC runs it fast,
to the start of string).
// With each "a" added time to complete is always doub
// On my PC 27 times of "a" takes 2,5 seconds (when I
takes 5 seconds).
// https://en.wikipedia.org/wiki/ReDoS
function evilRegExp() {
    var string = 'aaaaaaaaaaaaaaaaaaaaaaaaaaab';
    string.match(/^(a|a)+$/);
}

// Request to http://localhost:8080/ wil be served qui
evilRegExp() but request to
// http://localhost:8080/test/ will be slow and will a
request to http://localhost:8080/
http.createServer(function (req, res) {
    console.log("request", req.url);

    if (req.url.indexOf('test') != -1) {
      console.log('runing evilRegExp()');
      evilRegExp();
    }

    res.write('Done');
    res.end();
}).listen(8080);
```

You can run many parallel requests to http://localhost:8080/ and it will be fast. Then run just one slow request http://localhost:8080/test/ and no other request (even those fast at http://localhost:8080/) will not be served until slow (blocking) request ends.

This second example uses promises but it still block main node thread and no other requests/clients can be served.

```
var http = require('http');

function evilRegExp() {
    return new Promise(resolve => {
        var string = 'aaaaaaaaaaaaaaaaaaaaaaaaaaab';
        string.match(/^(a|a)+$/);
        resolve();
    });
}

http.createServer(function (req, res) {
      console.log("request", req.url);

    if (req.url.indexOf('test') != -1) {
      console.log('runing evilRegExp()');
      evilRegExp();
    }

    res.write('Done');
    res.end();
}).listen(8080);
```

This third example uses async+await but it is also blocking (async+await is the same as native Promise).

```
var http = require('http');

async function evilRegExp() {
    var string = 'aaaaaaaaaaaaaaaaaaaaaaaaaaab';
    string.match(/^(a|a)+$/);
    resolve();
}

http.createServer(function (req, res) {
      console.log("request", req.url);

    if (req.url.indexOf('test') != -1) {
      console.log('runing evilRegExp()');
      await evilRegExp();
    }

    res.write('Done');
    res.end();

}).listen(8080);
```

Fourth example uses setTimeout() which causes slow request seems to be served immediately (browser quickly gets "Done") but it is also blocking and any other fast requests will wait until evilRegExp() ends.

```
var http = require('http');

function evilRegExp() {
    var string = 'aaaaaaaaaaaaaaaaaaaaaaaaaaab';
    string.match(/^(a|a)+$/);
}

http.createServer(function (req, res) {
      console.log("request", req.url);

    if (req.url.indexOf('test') != -1) {
      console.log('runing evilRegExp()');
      setTimeout(function() { evilRegExp(); }, 0);
    }

    res.write('Done');
    res.end();

}).listen(8080);
```

Share Improve this answer Follow

edited Jan 15, 2020 at 15:50

answered Aug 22, 2018 at 15:55

mikep
7,389 ● 2 ● 38 ● 45

---

what's the problem with that regexp string.match(/^(a|a)+$/); ? – mario ruiz Feb 19, 2019 at 23:53

---

1 @MarioRuiz it is evil regex that that takes a very long time to evaluate depending on input length. Execution time grows exponentially depending on input length.Try plying with it and you well see that it is never ending when input is long. I also explained it at the top of my post in javascript comment. In general see en.wikipedia.org/wiki/ReDoS – mikep Feb 22, 2019 at 10:46

Just not getting why acts like this? Seems pretty much simple... /^(a|a)+$/ – mario ruiz Feb 26, 2019 at 20:42

Are you saying that everyone of these examples will block the thread if /test is called, or does one of then not block the thread? – PrestonDocks Jan 15, 2020 at 10:59

@PrestonDocks yes, everyone of these 4 examples will block the main thread if `/test` request is called because of code `string.match(/^(a|a)+$/)` is synchronous and takes long time to process. I have updated my answer to clarify that. – mikep Jan 15, 2020 at 15:50

4

I just had an "aha!" moment and thought that I'd pass it on. "await" does not return control directly to JavaScript - it returns control to the caller. Let me illustrate. Here is a program using callbacks:

```
console.log("begin");
step1(() => console.log("step 1 handled"));
step2(() => console.log("step 2 handled"));
console.log("all steps started");

// ----------------------------------------------

function step1(func) {

console.log("starting step 1");
setTimeout(func, 10000);
} // step1()

// ----------------------------------------------

function step2(func) {

console.log("starting step 2");
setTimeout(func, 5000);
} // step2()
```

The behavior that we want is 1) both steps are immediately started, and 2) when a step is ready to be handled (imagine an Ajax request, but here we just wait for some period of time), the handling of each step happens immediately.

The "handling" code here is console.log("step X handled"). That code (which in a real application can be quite long and possibly include nested awaits), is in a callback, but we'd prefer for it to be top level code in a function.

Here is equivalent code using async/await. Note that we had to create a sleep() function since we need to await on a function that returns a promise:

```
let sleep = ms => new Promise((r, j)=>setTimeout(r, ms

console.log("begin");
step1();
step2();
console.log("all steps started");

// ----------------------------------------------

async function step1() {

console.log("starting step 1");
await sleep(10000);
console.log("step 1 handled");
} // step1()

// ----------------------------------------------

async function step2() {

console.log("starting step 2");
await sleep(5000);
console.log("step 2 handled");
} // step2()
```

The important takeaway for me was that the await in step1() returns control to the main body code so that step2() can be called to start that step, and the await in step2() also returns to the main body code so that "all steps started" can be printed. Some people advocate that you use "await Promise.all()" to start multiple steps, then after that, handle all steps using the results (which will appear in an array). However, when you do that, no step is handled until all steps resolve. That's not ideal, and seems to be totally unnecessary.

Share Improve this answer

Follow

answered Apr 20, 2018 at 12:53

John Deighan

4,571 ● 4 ● 21 ● 21

I believe your point is that waiting on all initial steps isn't ideal if some subset of those steps require more processing afterward. Example: You have promises A, B, and D; wait on them all; use A and B to produce promise C; wait on C; use C and D to return promise E. The problem in the example is that D might have taken longer to process than A&B, delaying C. That all said, `Promise.all()` is very useful, the example just shows a naive way of using it. Check this out for various ways of using promises: promise_variations.js – Chinoto Vokro Jul 18, 2018 at 16:36

Async functions enable us to write promise based code as if it were synchronous, but without blocking the execution thread. It operates asynchronously via the

4

event-loop. Async functions will always return a value. Using async simply implies that a promise will be returned, and if a promise is not returned, JavaScript automatically wraps it in a resolved promise with its value.

Find the article on Medium . [How to use Async Await in JavaScript.]()

Share Improve this answer Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

answered Apr 18, 2020 at 10:21
Rizwan Ansar
710 ● 1 ● 8 ● 17