The Node.js Event Loop, Timers, and `process.nextTick()`

What is the Event Loop?

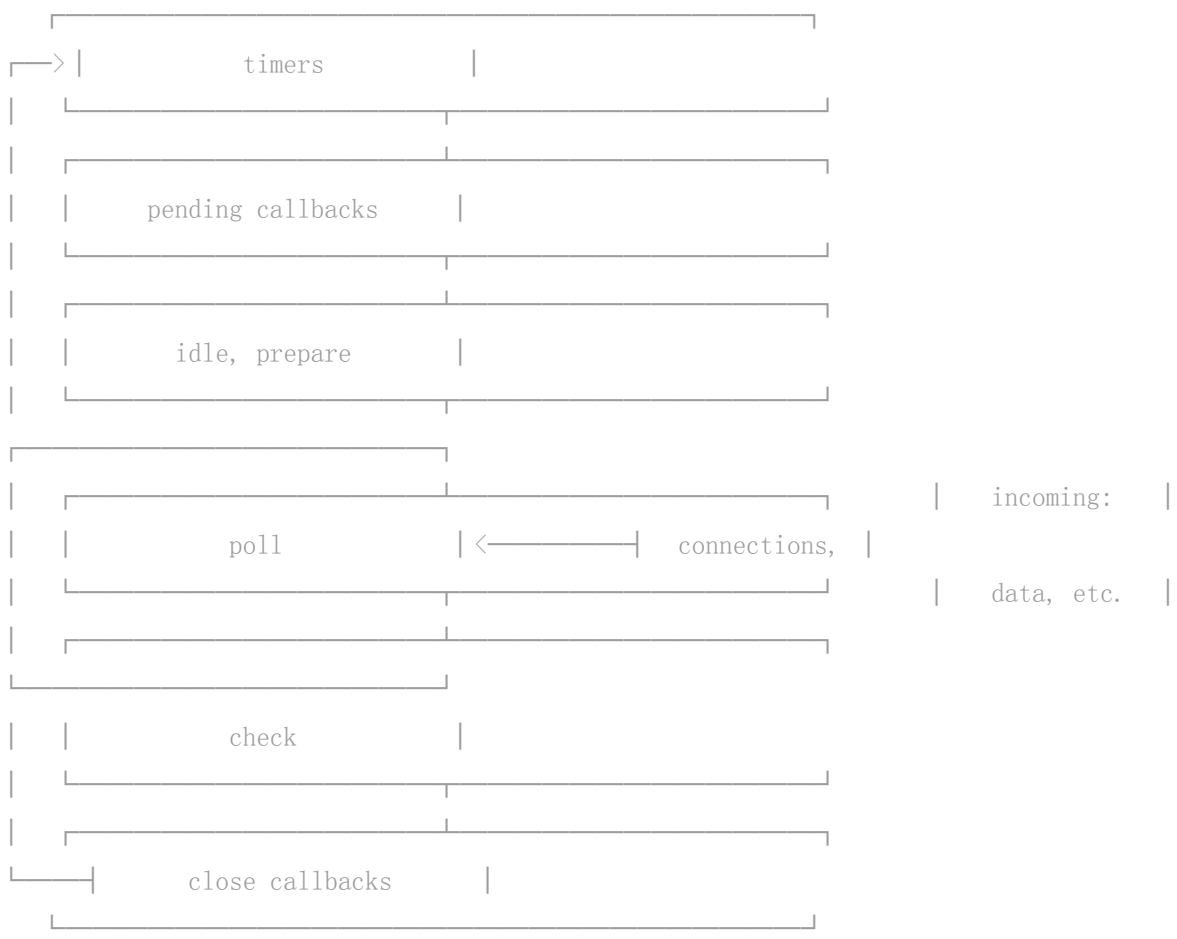
The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the **poll** queue to eventually be executed. We'll explain this in further detail later in this topic.

Event Loop Explained

When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL, which is not covered in this document) which may make async API calls, schedule timers, or call `process.nextTick()`, then begins processing the event loop.

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


timers
pending callbacks
idle, prepare
poll
connections,
incoming:
data, etc.
check
close callbacks

note: each box will be referred to as a "phase" of the event loop.

Each phase has a FIFO queue of callbacks to execute. While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

Since any of these operations may schedule *more* operations and new events processed in the **poll** phase are queued by the kernel, poll events can be queued while polling events are being processed. As a result, long running callbacks can allow the poll phase to run much longer than a timer's threshold. See the **timers** and **poll** sections for more details.

NOTE: *There is a slight discrepancy between the Windows and the Unix/Linux implementation, but that's not important for this demonstration. The most important parts are here. There are actually seven or eight steps, but the ones we care about — ones that Node.js actually uses - are those above.*

Phases Overview

- **timers**: this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`.
- **pending callbacks**: executes I/O callbacks deferred to the next loop iteration.
- **idle, prepare**: only used internally.
- **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.
- **check**: `setImmediate()` callbacks are invoked here.
- **close callbacks**: some close callbacks, e.g. `socket.on('close', ...)`.

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

Phases in Detail

timers

A timer specifies the **threshold** *after which* a provided callback *may be executed* rather than the **exact** time a person *wants it to be executed*. Timers callbacks will run as early as they can be scheduled after the specified amount of time has passed; however, Operating System scheduling or the running of other callbacks may delay them.

***Note**: Technically, the **poll** phase controls when timers are executed.*

For example, say you schedule a timeout to execute after a 100 ms threshold, then your script starts asynchronously reading a file which takes 95 ms:

```
const fs = require('fs');

function someAsyncOperation(callback) {
  // Assume this takes 95ms to complete
  fs.readFile('/path/to/file', callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;

  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);


// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();

  // do something that will take 10ms...
  while (Date.now() - startCallback < 10) {
    // do nothing
  }
});
```

When the event loop enters the **poll** phase, it has an empty queue (`fs.readFile()` has not completed), so it will wait for the number of ms remaining until the soonest timer's threshold is reached. While it is waiting 95 ms pass, `fs.readFile()` finishes reading the file and its callback which takes 10 ms to complete is added to the **poll** queue and executed. When the callback finishes, there are no more callbacks in the queue, so the event loop will see that the threshold of the soonest timer has been reached then wrap back to the **timers** phase to execute the timer's callback. In this example, you will see that the total delay between the timer being scheduled and its callback being executed will be 105ms.

Note: To prevent the **poll** phase from starving the event loop, libuv (the C library that implements the Node.js event loop and all of the asynchronous behaviors of the platform) also has a hard maximum (system dependent) before it stops polling for more events.

pending callbacks

This phase executes callbacks for some system operations such as types of TCP errors. For example if a TCP socket receives `ECONNREFUSED` when attempting to connect, some *nix systems want to wait to report the error. This will be queued to execute in the **pending callbacks** phase.

poll

The **poll** phase has two main functions:

1. Calculating how long it should block and poll for I/O, then
2. Processing events in the **poll** queue.

When the event loop enters the **poll** phase *and there are no timers scheduled*, one of two things will happen:

- *If the **poll** queue **is not empty***, the event loop will iterate through its queue of callbacks executing them synchronously until either the queue has been exhausted, or the system-dependent hard limit is reached.

- *If the **poll** queue **is empty***, one of two more things will happen:
    - If scripts have been scheduled by `setImmediate()`, the event loop will end the **poll** phase and continue to the **check** phase to execute those scheduled scripts.
    - If scripts **have not** been scheduled by `setImmediate()`, the event loop will wait for callbacks to be added to the queue, then execute them immediately.

Once the **poll** queue is empty the event loop will check for timers *whose time thresholds have been reached*. If one or more timers are ready, the event loop will wrap back to the **timers** phase to execute those timers' callbacks.

check

This phase allows a person to execute callbacks immediately after the **poll** phase has completed. If the **poll** phase becomes idle and scripts have been queued with `setImmediate()`, the event loop may continue to the **check** phase rather than waiting.

`setImmediate()` is actually a special timer that runs in a separate phase of the event loop. It uses a libuv API that schedules callbacks to execute after the **poll** phase has completed.

Generally, as the code is executed, the event loop will eventually hit the **poll** phase where it will wait for an incoming connection, request, etc. However, if a callback has been scheduled with `setImmediate()` and the **poll** phase becomes idle, it will end and continue to the **check** phase rather than waiting for **poll** events.

close callbacks

If a socket or handle is closed abruptly (e.g. `socket.destroy()`), the `'close'` event will be emitted in this phase. Otherwise it will be emitted via `process.nextTick()`.

`setImmediate()` vs `setTimeout()`

`setImmediate` and `setTimeout()` are similar, but behave in different ways depending on when they are called.

- `setImmediate()` is designed to execute a script once the current **poll** phase completes.
- `setTimeout()` schedules a script to be run after a minimum threshold in ms has elapsed.

The order in which the timers are executed will vary depending on the context in which they are called. If both are called from within the main module, then timing will be bound by the performance of the process (which can be impacted by other applications running on the machine).

For example, if we run the following script which is not within an I/O cycle (i.e. the main module), the order in which the two timers are executed is non-deterministic, as it is bound by the performance of the process:

```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);

setImmediate(() => {
  console.log('immediate');
});
```

```
$ node timeout_vs_immediate.js
timeout
immediate

$ node timeout_vs_immediate.js
immediate
timeout
```

However, if you move the two calls within an I/O cycle, the immediate callback is always executed first:

```
// timeout_vs_immediate.js
const fs = require('fs');

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
```

```
$ node timeout_vs_immediate.js
immediate
timeout

$ node timeout_vs_immediate.js
immediate
timeout
```

The main advantage to using `setImmediate()` over `setTimeout()` is `setImmediate()` will always be executed before any timers if scheduled within an I/O cycle, independently of how many timers are present.

`process.nextTick()`

Understanding `process.nextTick()`

You may have noticed that `process.nextTick()` was not displayed in the diagram, even though it's a part of the asynchronous API. This is because `process.nextTick()` is not technically part of the event loop. Instead, the `nextTickQueue` will be processed after the current operation completes, regardless of the current phase of the event loop.

Looking back at our diagram, any time you call `process.nextTick()` in a given phase, all callbacks passed to `process.nextTick()` will be resolved before the event loop continues. This can create some bad situations because **it allows you to "starve" your I/O by making recursive `process.nextTick()` calls**, which prevents the event loop from reaching the **poll** phase.

Why would that be allowed?

Why would something like this be included in Node.js? Part of it is a design philosophy where an API should always be asynchronous even where it doesn't have to be. Take this code snippet for example:

```
function apiCall(arg, callback) {
  if (typeof arg !== 'string')
    return process.nextTick(callback,
                            new TypeError('argument should be string'));
}
```

The snippet does an argument check and if it's not correct, it will pass the error to the callback. The API updated fairly recently to allow passing arguments to `process.nextTick()` allowing it to take any arguments passed after the callback to be propagated as the arguments to the callback so you don't have to nest functions.

What we're doing is passing an error back to the user but only *after* we have allowed the rest of the user's code to execute. By using `process.nextTick()` we guarantee that `apiCall()` always runs its callback *after* the rest of the user's code and *before* the event loop is allowed to proceed. To achieve this, the JS call stack is allowed to unwind then immediately execute the provided callback which allows a person to make recursive calls to `process.nextTick()` without reaching a `RangeError: Maximum call stack size exceeded from v8.`

This philosophy can lead to some potentially problematic situations. Take this snippet for example:

```
let bar;

// this has an asynchronous signature, but calls callback synchronously
function someAsyncApiCall(callback) { callback(); }

// the callback is called before `someAsyncApiCall` completes.
someAsyncApiCall(() => {
  // since someAsyncApiCall has completed, bar hasn't been assigned any value
  console.log('bar', bar); // undefined
});

bar = 1;
```

The user defines `someAsyncApiCall()` to have an asynchronous signature, but it actually operates synchronously. When it is called, the callback provided to `someAsyncApiCall()` is called in the same phase of the event loop because `someAsyncApiCall()` doesn't actually do anything asynchronously. As a result, the callback tries to reference `bar` even though it may not have that variable in scope yet, because the script has not been able to run to completion.

By placing the callback in a `process.nextTick()`, the script still has the ability to run to completion, allowing all the variables, functions, etc., to be initialized prior to the callback being called. It also has the advantage of not allowing the event loop to continue. It may be useful for the user to be alerted to an error before the event loop is allowed to continue. Here is the previous example using `process.nextTick()`:

```
let bar;

function someAsyncApiCall(callback) {
  process.nextTick(callback);
}

someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});

bar = 1;
```

Here's another real world example:

```
const server = net.createServer(() => {}).listen(8080);

server.on('listening', () => {});
```

When only a port is passed, the port is bound immediately. So, the `'listening'` callback could be called immediately. The problem is that the `.on('listening')` callback will not have been set by that time.

To get around this, the `'listening'` event is queued in a `nextTick()` to allow the script to run to completion. This allows the user to set any event handlers they want.

`process.nextTick()` vs `setImmediate()`

We have two calls that are similar as far as users are concerned, but their names are confusing.

- `process.nextTick()` fires immediately on the same phase
- `setImmediate()` fires on the following iteration or 'tick' of the event loop

In essence, the names should be swapped. `process.nextTick()` fires more immediately than `setImmediate()`, but this is an artifact of the past which is unlikely to change. Making this switch would break a large percentage of the packages on npm. Every day more new modules are being added, which means every day we wait, more potential breakages occur. While they are confusing, the names themselves won't change.

We recommend developers use `setImmediate()` in all cases because it's easier to reason about (and it leads to code that's compatible with a wider variety of environments, like browser JS.)

Why use `process.nextTick()`?

There are two main reasons:

1. Allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues.

2. At times it's necessary to allow a callback to run after the call stack has unwound but before the event loop continues.

One example is to match the user's expectations. Simple example:

```
const server = net.createServer();
server.on('connection', (conn) => { });

server.listen(8080);
server.on('listening', () => { });
```

Say that `listen()` is run at the beginning of the event loop, but the listening callback is placed in a `setImmediate()`. Unless a hostname is passed, binding to the port will happen immediately. For the event loop to proceed, it must hit the **poll** phase, which means there is a non-zero chance that a connection could have been received allowing the connection event to be fired before the listening event.

Another example is running a function constructor that was to, say, inherit from `EventEmitter` and it wanted to call an event within the constructor:

```
const EventEmitter = require('events');
const util = require('util');

function MyEmitter() {
  EventEmitter.call(this);
  this.emit('event');
}
util.inherits(MyEmitter, EventEmitter);

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```

You can't emit an event from the constructor immediately because the script will not have processed to the point where the user assigns a callback to that event. So, within the constructor itself, you can use `process.nextTick()` to set a callback to emit the event after the constructor has finished, which provides the expected results:

```
const EventEmitter = require('events');
const util = require('util');

function MyEmitter() {
  EventEmitter.call(this);

  // use nextTick to emit the event once a handler is assigned
  process.nextTick(() => {
    this.emit('event');
  });
}
util.inherits(MyEmitter, EventEmitter);

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
```