

3. Node.js Event Loop

Friday, May 24, 2024

11:52 AM

Heart of Node.js

Event Loop:

1. All Callback functions (non top-level code) run in Event Loop
2. Node.js is built around Callbacks functions (some operations that finish & return something in the future)
3. Event-driven architecture:
 - Events are emitted
 - Event loops pick them up
 - Callbacks are called

Event Loop receives events each time something important happens:

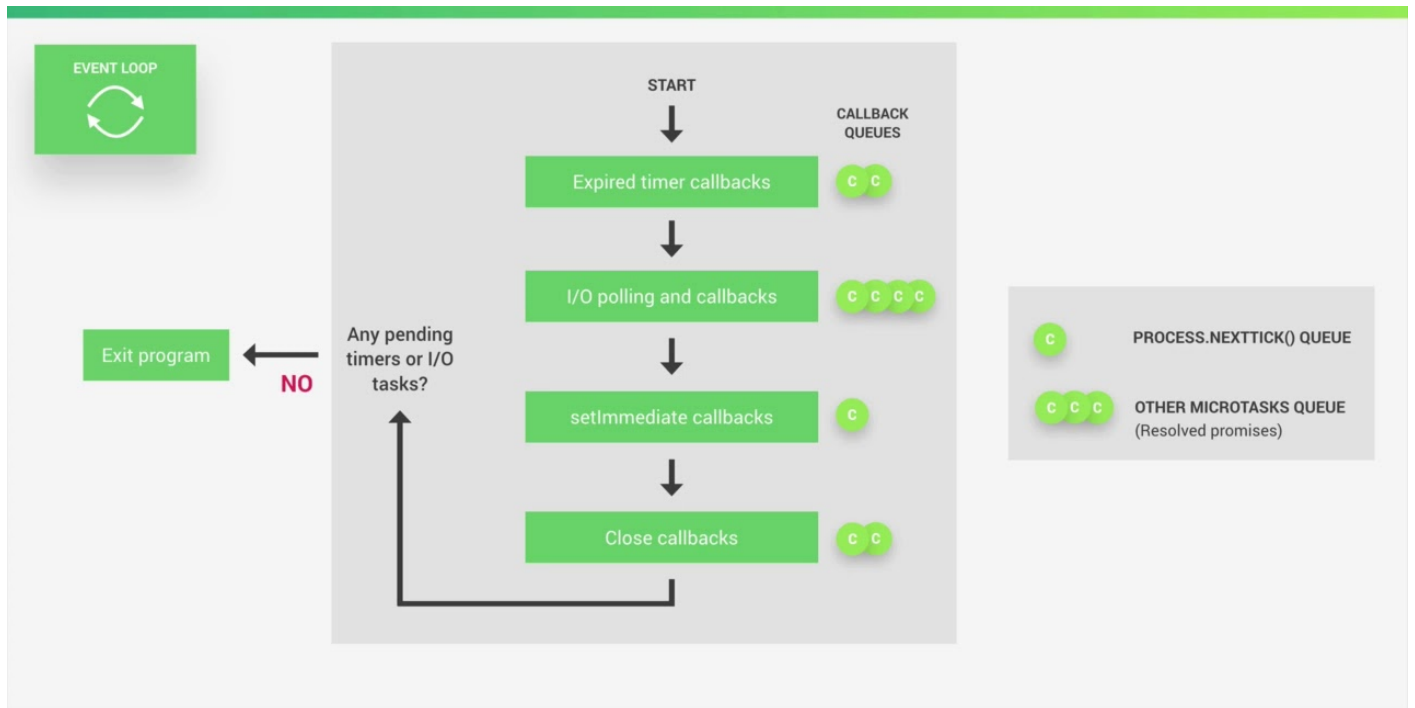
-Orchestration:

1. Receives Events
2. Call their callback funcs
3. Off-load resource intensive tasks to Thread Pool

Node runtime starts

Callback Queues:

Callbacks coming from events that Event Loop receives, sometimes, there's only 1 event queue, each phase has its own callback queues.

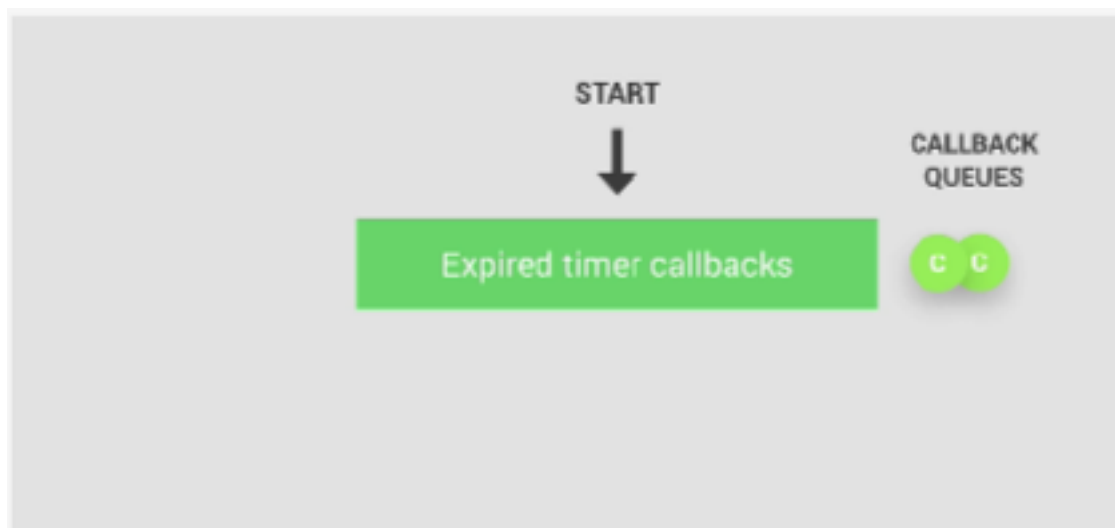


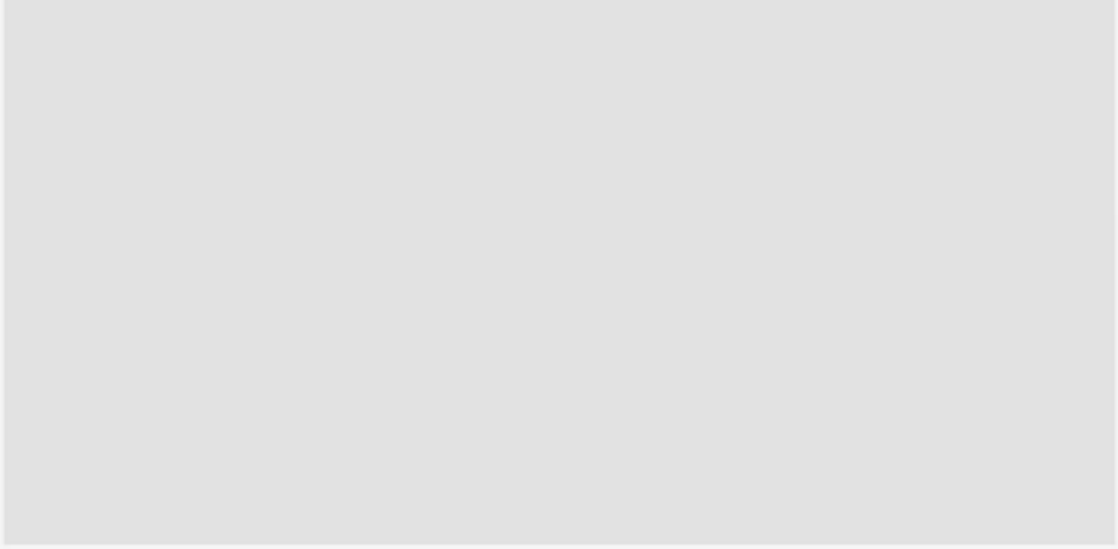
Event Loop does **Orchestration**,
by receiving events ->
calling Callback functions ->
offloading resource intensive tasks to Thread Pool

4 Most important Event Loop phases:

0. Start

1. Call 'Expired timer callbacks'





```
setTimeout(() => {  
  console.log('Timer expired!');  
});
```

Example:

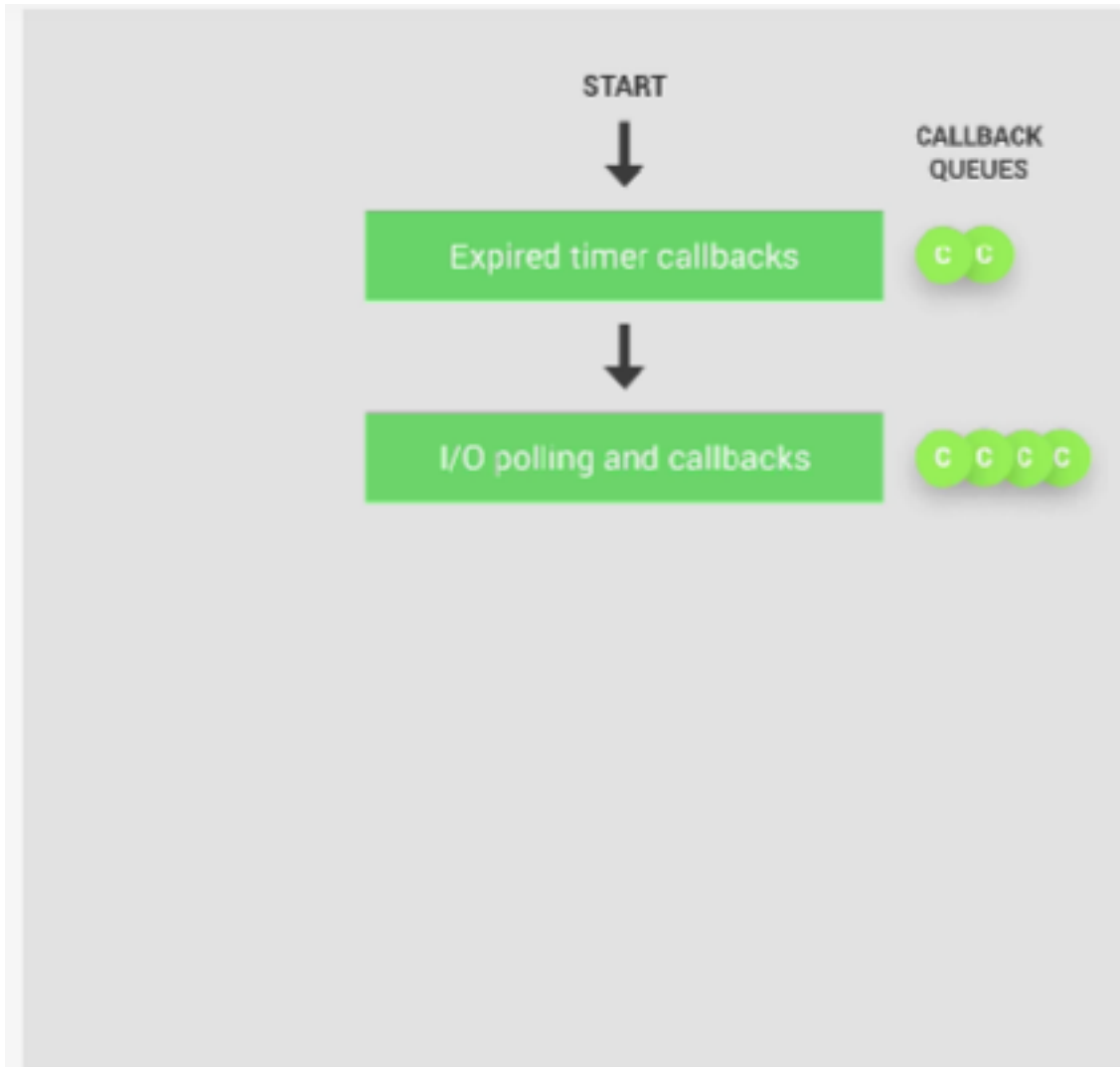
```
setTimeout(() => {  
  console.log('Timer expired!');  
}, ms)
```

If there're Callback funcs from timers that just expired, these are the 1st ones to be proceeded by the Event Loop.

****If timer expires later, during the time, when 1 of the other phases are being processed, then the Callback of that timer will only be called when Event Loop comes back to this 1st phase.**

***Callbacks in each queue are processed 1 by 1, until there're no one left in the queue, only then Event Loop will enter next phase ***

2. I/O polling & callbacks (looking for new I/O events to callback queue)



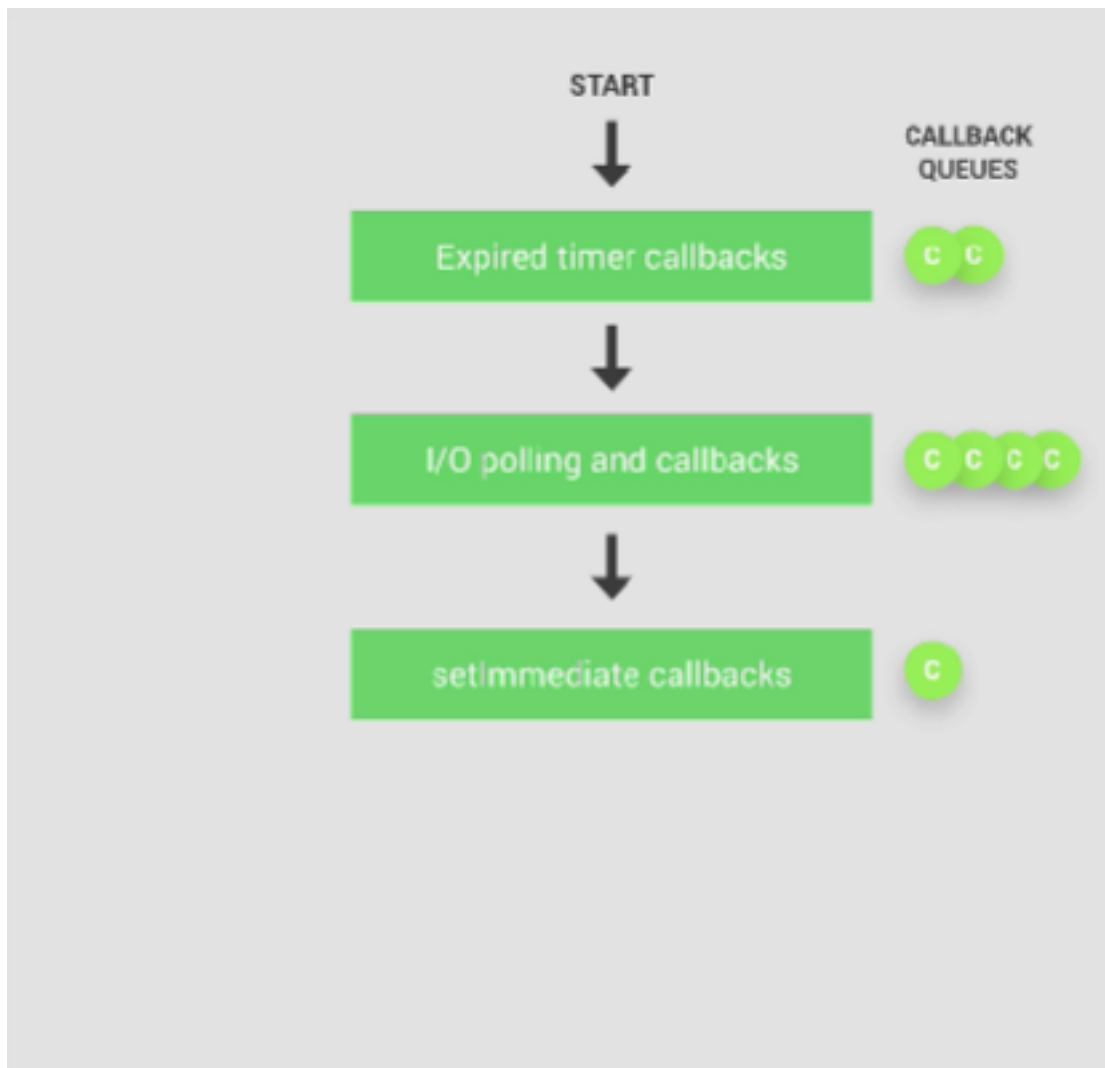
Example:

**** I/O = Networking & File Access**, 99% of our code gets executed here

```
fs.readFile('file.txt', (err, data) => {  
  if (err) console.log(err);  
  console.log('File read!\n', 'data: \n', data);  
  return data;  
});
```

2 setImmediate callbacks

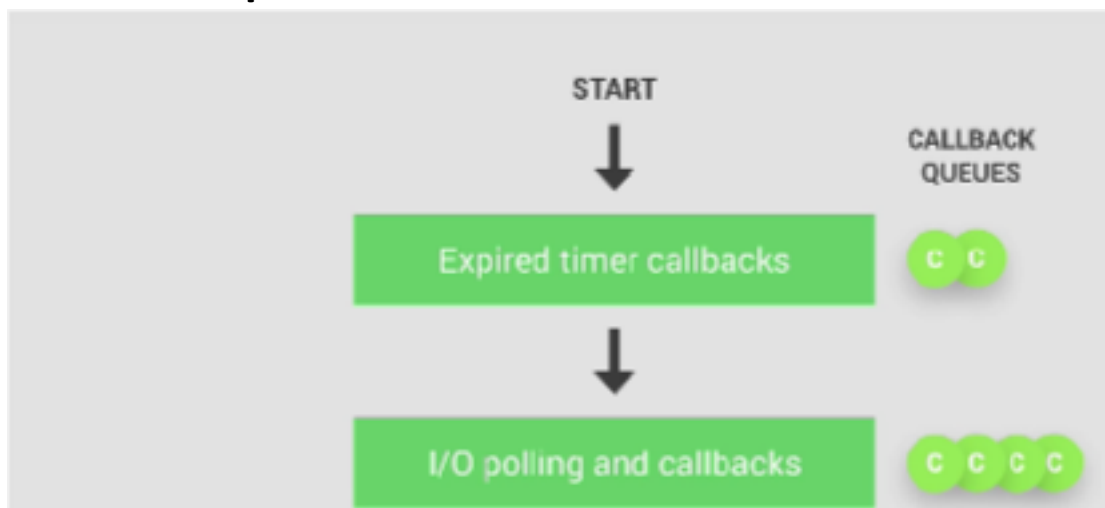
3. setImmediate callbacks

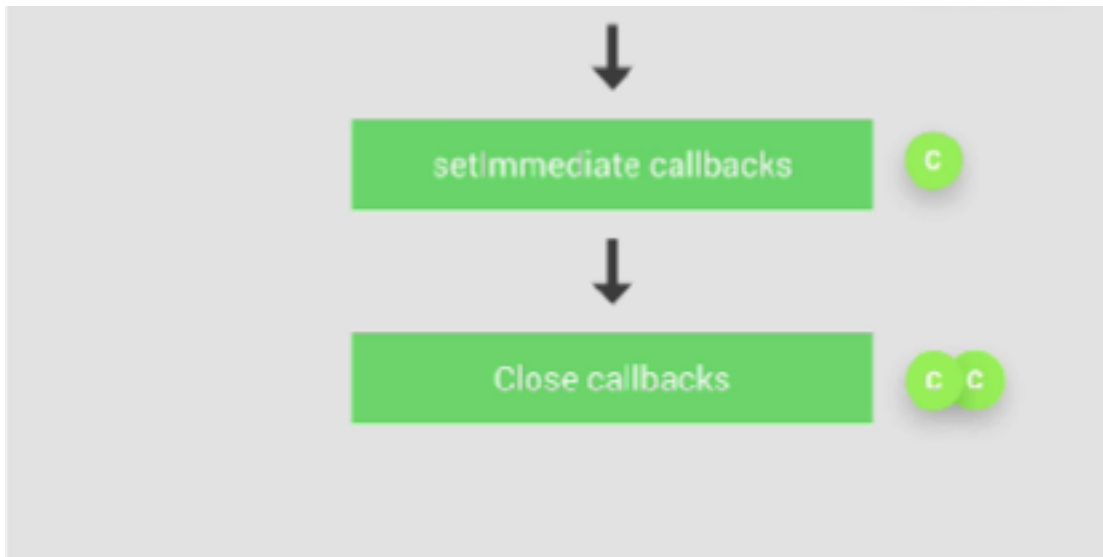


A special kind of timer if we want to process callbacks immediately after the I/O polling & callbacks phase.
This is only used for some really advanced use-cases.

4. Close callbacks

Not that important





All close events are processed
When a web server / web socket shuts down

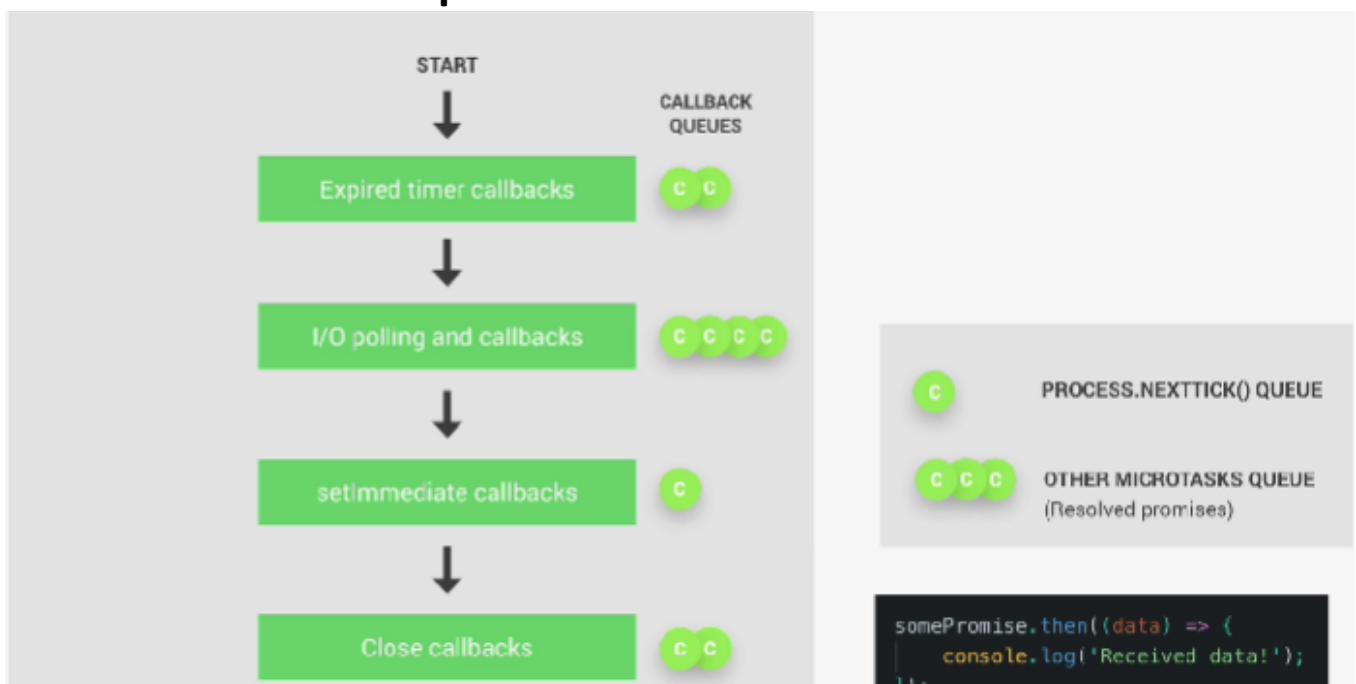
These are all phases in the Event Loop

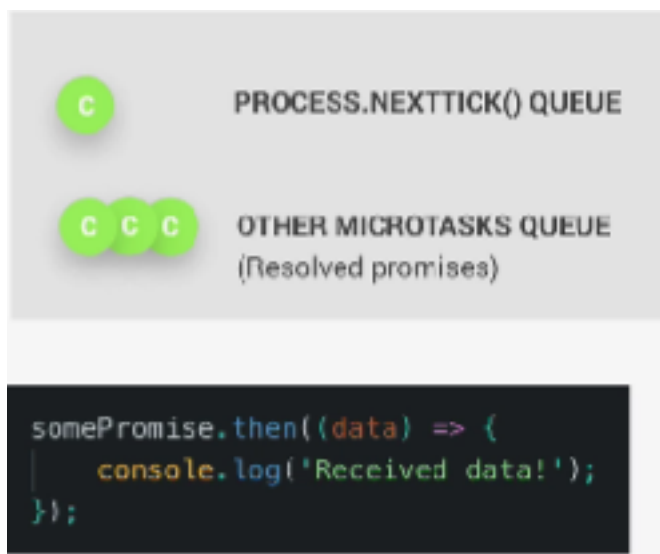
1. Expired timer callbacks
2. I/O polling and callbacks
3. setImmediate callbacks
4. Close callbacks

There're also 2 other Queues:

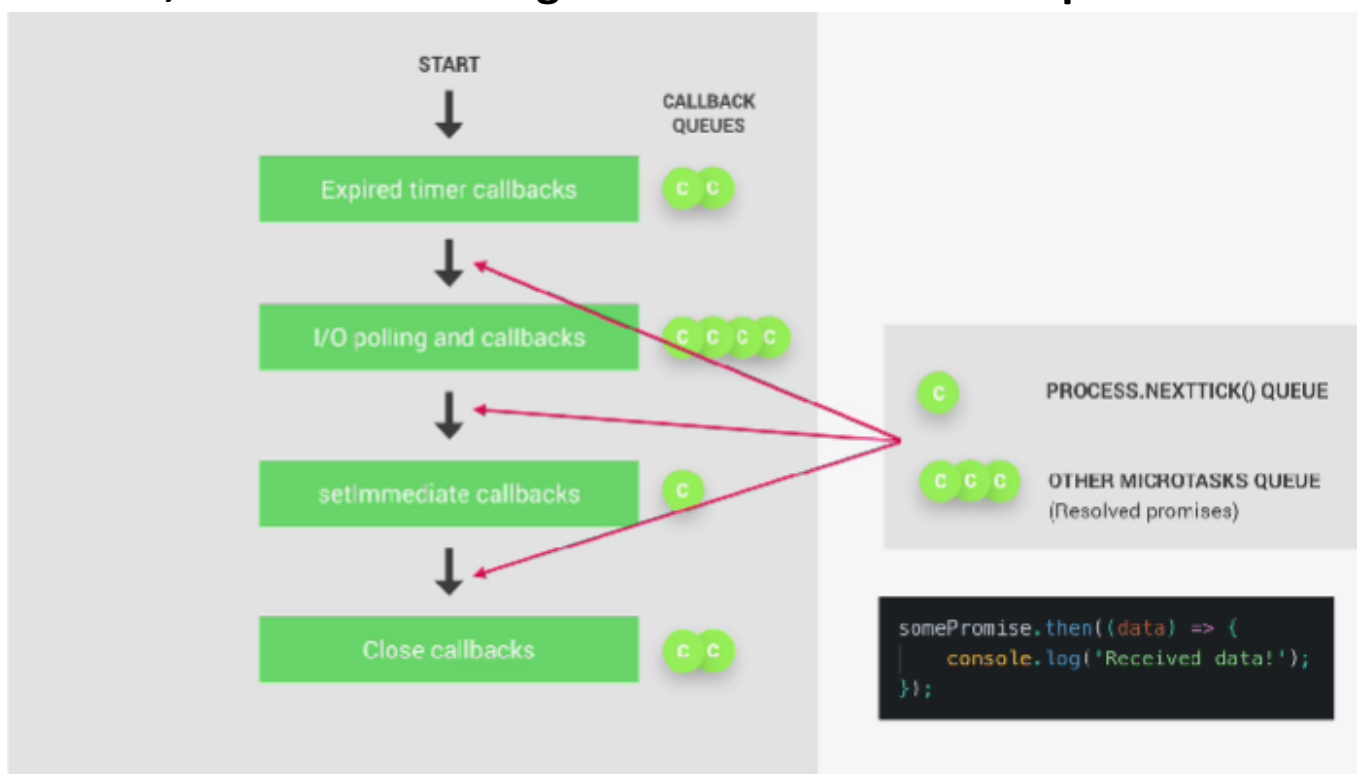
i. `Process.nextTick()` queue:

ii. Other Microtasks queue: Resolved Promises





If there are any Callbacks in 1 of these special queues to be processed, they'll be executed right after the current phase finishes, instead of waiting for the entire Event Loop to finish.



Execute Callbacks right after current Event Loop phase

Other Microtasks queue

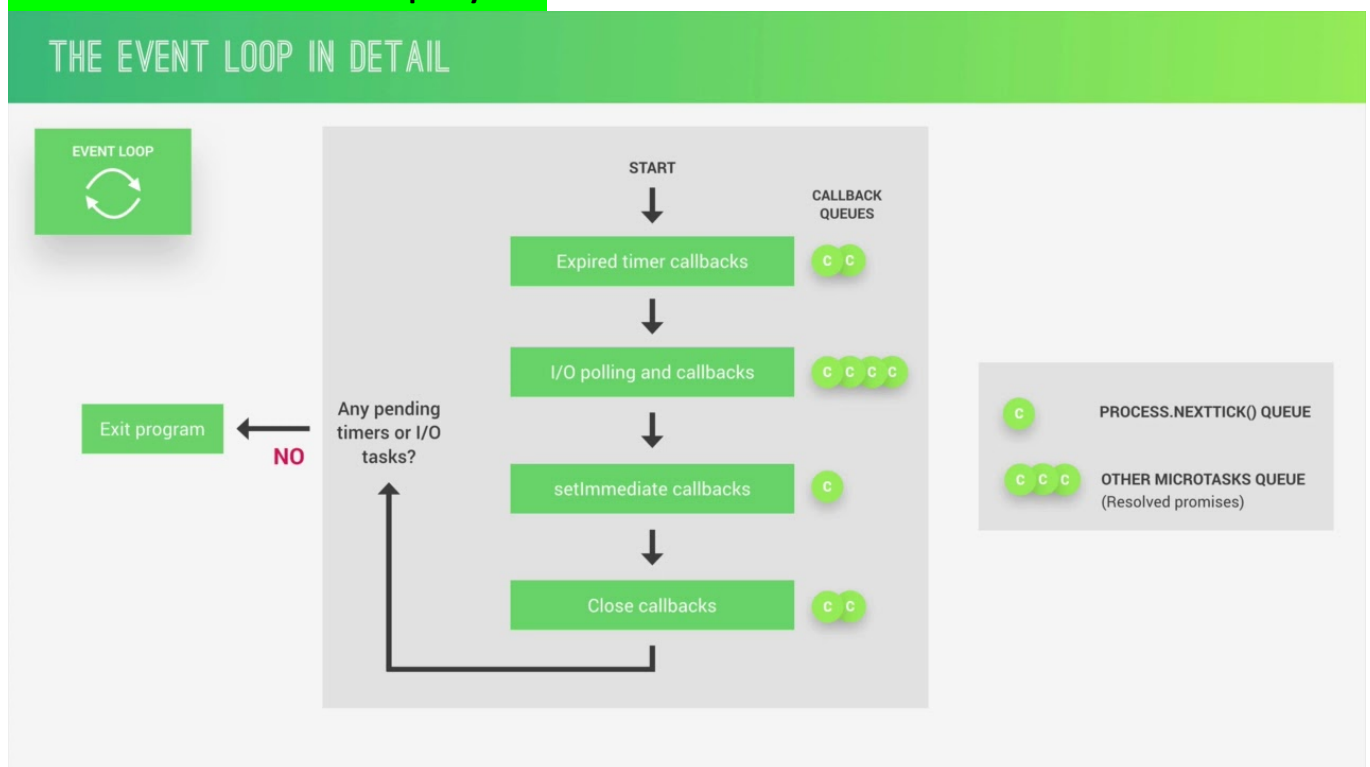
```
somePromise.then((data) => {  
  console.log('Received data!');  
});
```

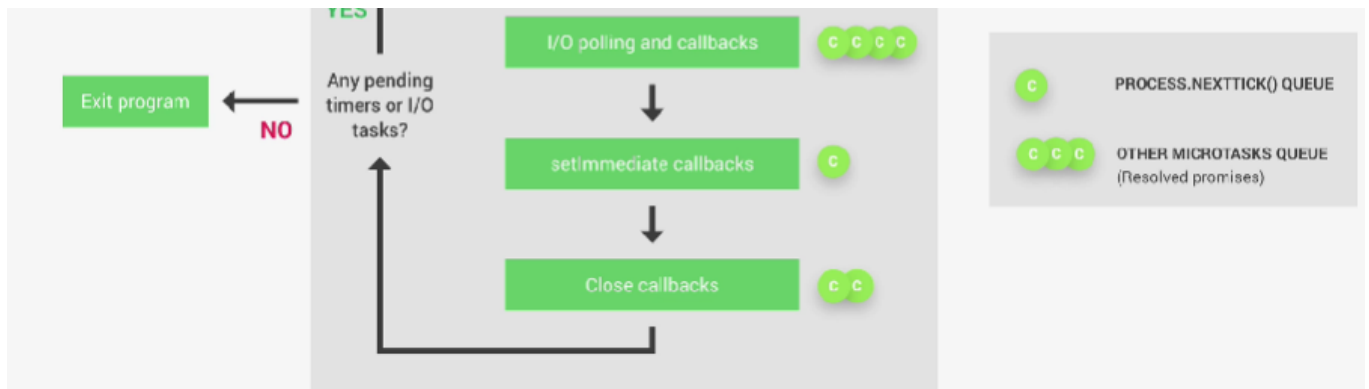
i. `Process.nextTick()` queue:

`Process.nextTick()` queue is only used, when we **really need to execute a certain Callback right after the current Event Loop phase**.

Similar to `setImmediate`, yet **`setImmediate` only runs right after I/O callback phase finishes**
[really advanced use-cases]

1 Tick = 1 Event Loop cycle





Event Loop => Any pending timers or I/O tasks ?
 (Proceed NextTick) : (Exit Program)

i.e. Node-farm project

-When we're listening for incoming HTTP requests, we're basically running an I/O tasks, thus Node.js keeps running & listening to HTTP requests coming in, rather than Exiting the app

When `fs.readFile` or `fs.writeFile`, there's also an I/O task, Node won't Exit

It's really important to grasp the concepts of Node.js Event Loop to debug



It's the **Event Loop** makes **Asynchronous programming**

possible in Node.js,
making it completely different from other platforms

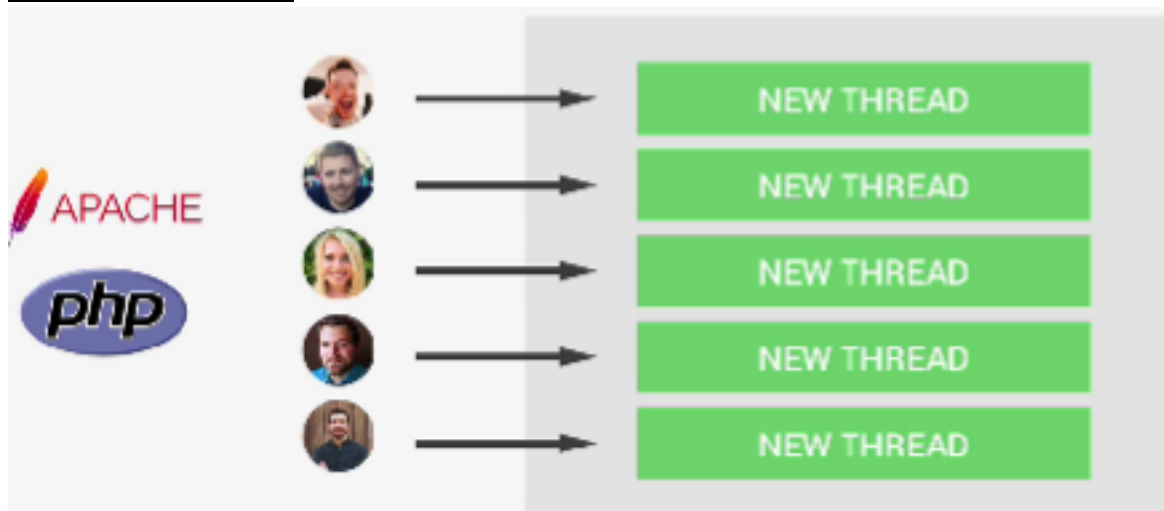
Event Loop does the **Orchestration** to
off-load heavy tasks such as File Access,
Networking to the Thread Pool & doing the simpler works itself.

Why we need the Node.js Event Loop?

Cuz, in Node.js, everything works in 1 single thread,
thus you can have thousands of
users using the same thread at the same time

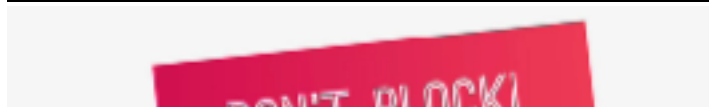
However, there's a danger of blocking the single thread,
making our entire app slow or even stopping
all users from accessing the entire app

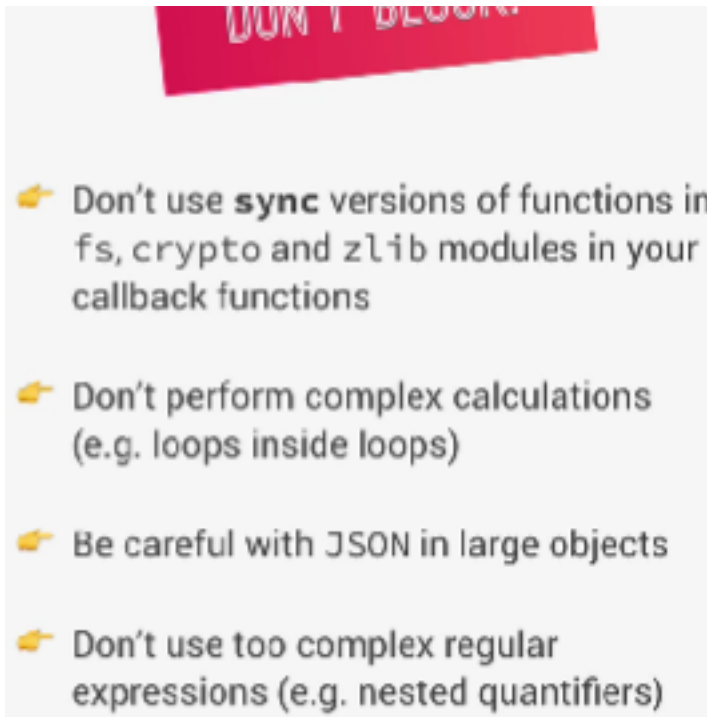
Apache + PHP:



A new thread is created for each new user.
Way more resource-intensive,
there's no danger of blocking Event Loop.

**How NOT to block Event Loop in Node.js:





1. Do NOT use SYNC functions in fs, crypto and zlib modules in Callback functions

2. Do NOT perform complex calculations in Event Loop (e.g. Loops inside Loops)

**3. Be careful with JSON in very large objects
It takes long-time to JSON.parse() || JSON.stringify()**

4. Do NOT use too complex Regular Expressions (e.g. Nested Quantifiers)

Potential Solutions:

- Manually off-loading Heavy Tasks to Thread Pool**
- Using Child processors**