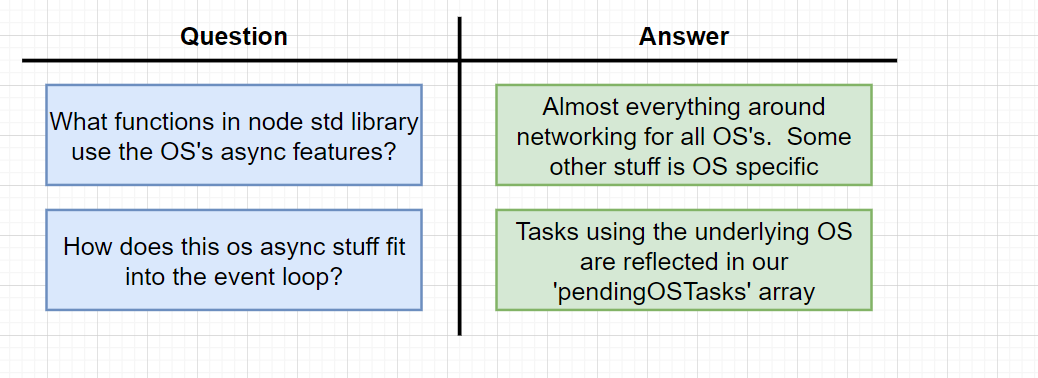
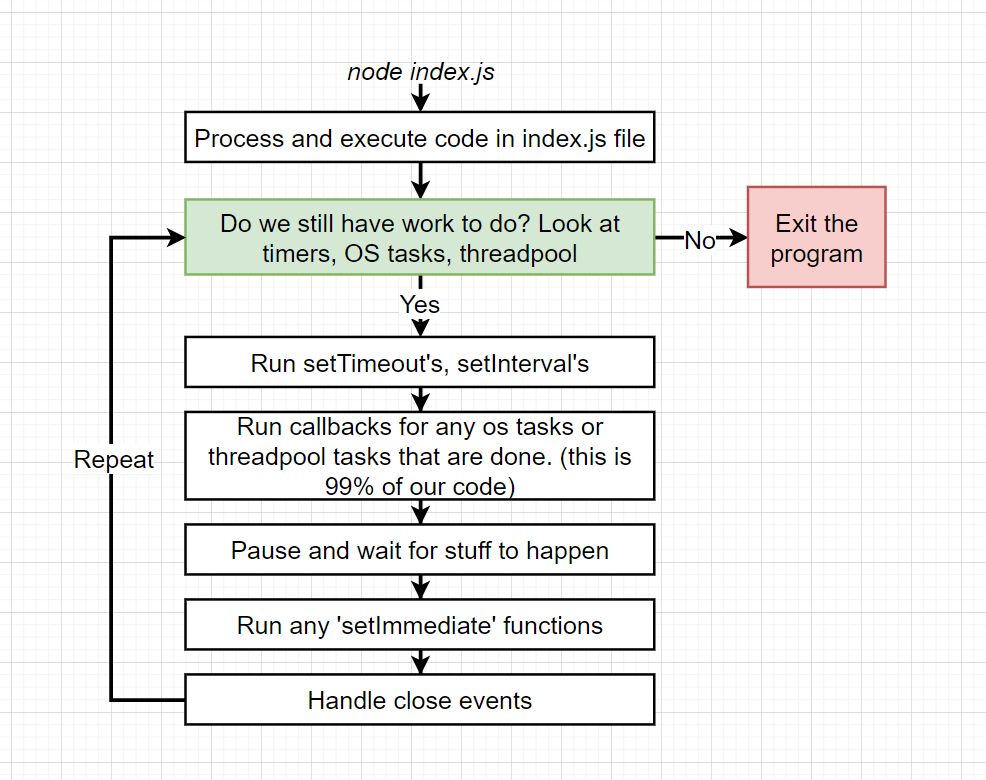


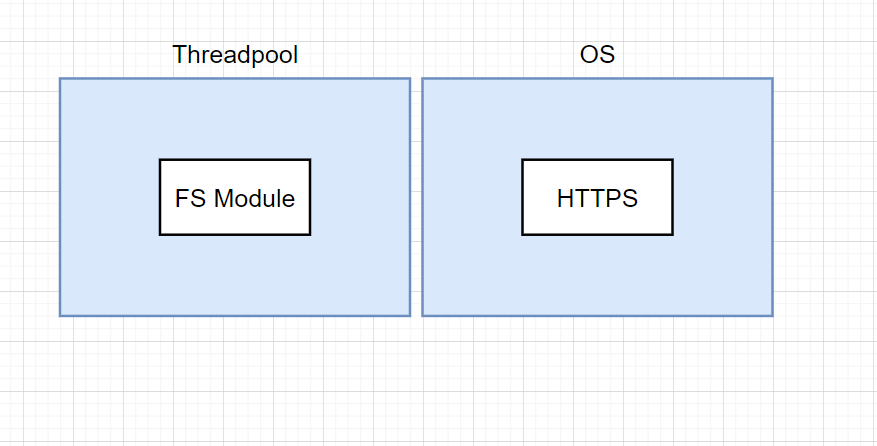
This network request task does not touch the libuv thread pool, instead it is delegated to OS async helpers.

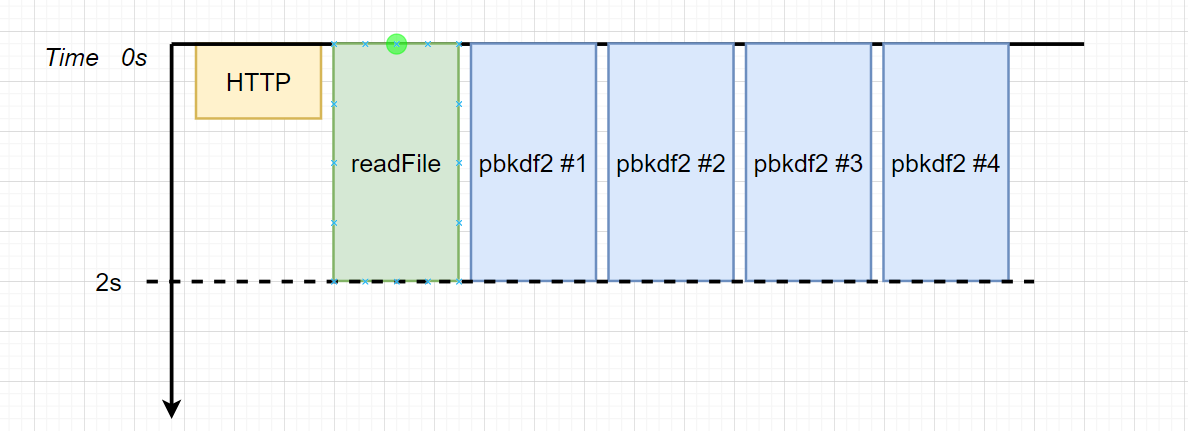


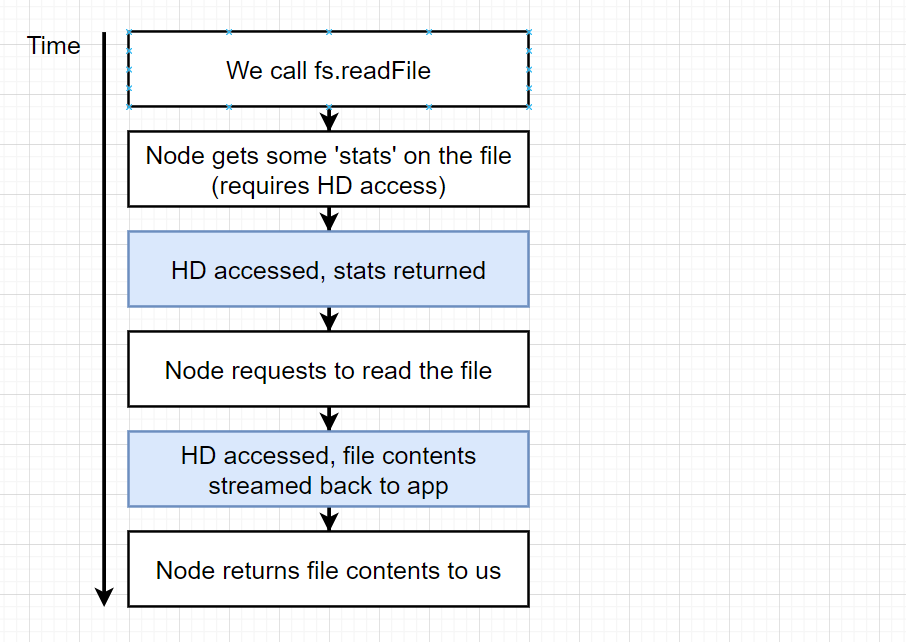


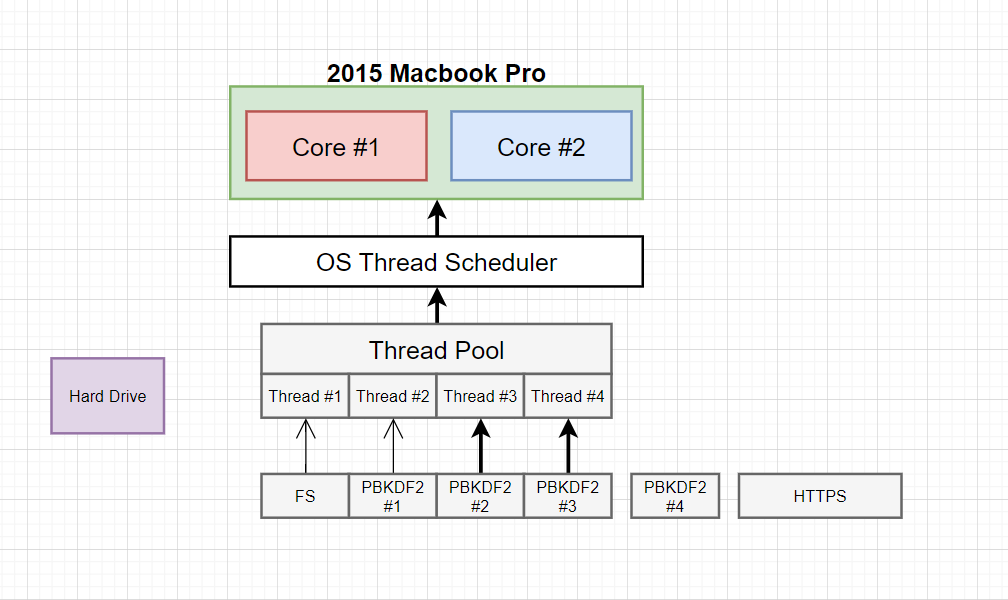
Can you guess the order of console logs?

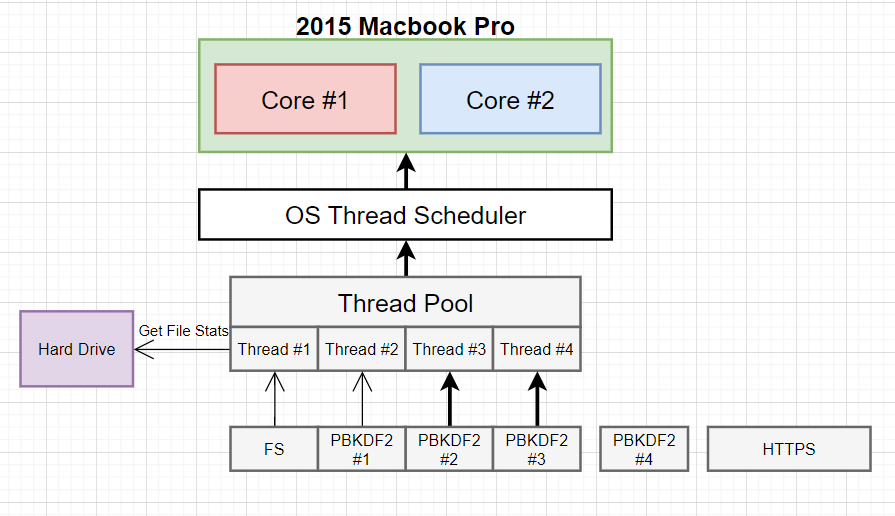


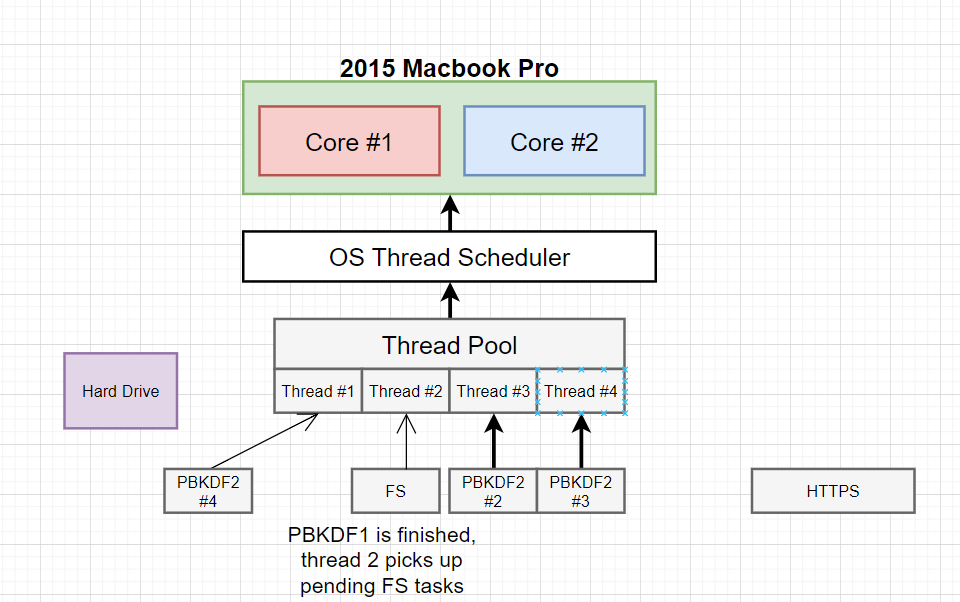
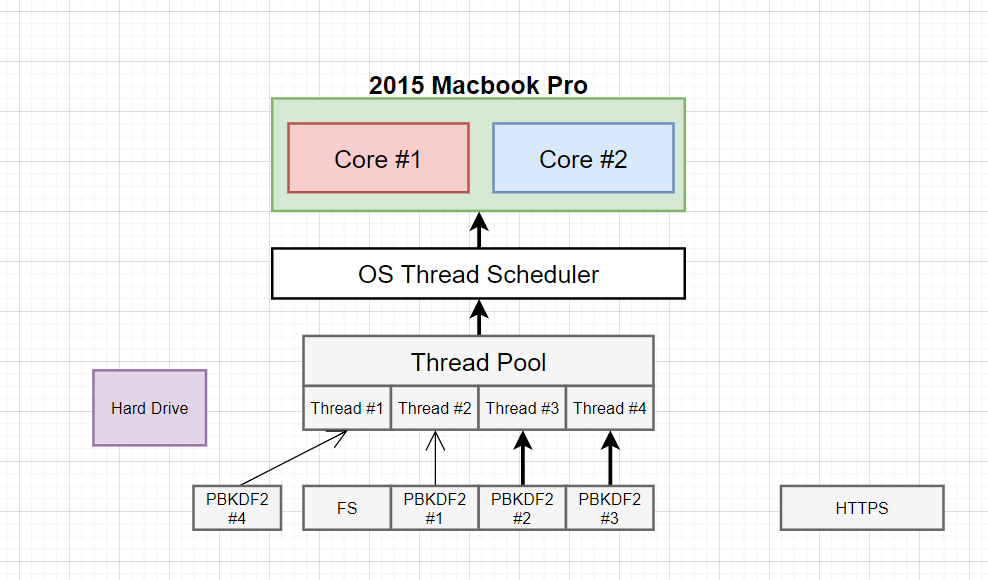












**Reactor Pattern**

NodeJS works in an event-driven model that involves an **Event Demultiplexer** and an **Event Queue**. All I/O requests will eventually generate an event of completion/failure or any other trigger, which is called an **Event**. These events are processed according to the following algorithm.

1. Event demultiplexer receives I/O requests and delegates these requests to the appropriate hardware.
2. Once the I/O request is processed (e.g, data from a file is available to be read, data from a socket is available to be read, etc.), event demultiplexer will then add the registered callback handler for the particular action in a queue to be processed. These callbacks are called events and the queue where events are added is called the **Event Queue**.
3. When events are available to be processed in the event queue, they are executed sequentially in the order they were received until the queue is empty.
4. If there are no events in the event queue or the Event Demultiplexer has no pending requests, the program will complete. Otherwise, the process will continue from the first step.

The program which orchestrates this entire mechanism is called the **Event Loop**.

Event Loop is a single-threaded and semi-infinite loop. The reason why this is called a semi-infinite loop is that this actually quits at some point when there is no more work to be done. In the developer’s perspective, this is where the program exits.

# Event Demultiplexer

Event Demultiplexer is not a component which exists in the real world, but an abstract concept in the reactor pattern. In the real world, event demultiplexer has been implemented in different systems in different names such as **epoll** on Linux, **kqueue**on BSD systems (macOS), **event ports**in Solaris, **IOCP (Input Output Completion Port)** in Windows, etc. NodeJS consumes the low-level non-blocking, asynchronous hardware I/O functionalities provided by these implementations.

## Complexities in File I/O

But the confusing fact is, not all the types of I/O can be performed using these implementations. Even on the same OS platform, there are complexities in supporting different types of I/O. Typically, network I/O can be performed in a non-blocking way using these epoll, kqueue, event ports and IOCP, but the file I/O is much more complex. Certain systems, such as Linux does not support complete asynchrony for file system access. And there are limitations in file system event notifications/signalling with kqueue in macOS systems (you can read more about these complications [here](http://blog.libtorrent.org/2012/10/asynchronous-disk-io/)). It is very complex/nearly impossible to address all these file system complexities in order to provide complete asynchrony.

## Complexities in DNS

Similar to the file I/O, certain DNS functions provided by Node API also have certain [complexities](https://nodejs.org/api/dns.html#dns_implementation_considerations). Since NodeJS DNS functions such as dns.lookup accesses system configuration files such as nsswitch.conf,resolv.conf and /etc/hosts , file system complexities described above are also applicable to dns.lookup function.

## The solution?

Therefore, a **thread pool** has been introduced to support I/O functions which cannot be directly addressed by hardware asynchronous I/O utils such as epoll/kqueue/event ports or IOCP. Now we know that not all the I/O functions happen in the thread pool. NodeJS has done its best to do most of the I/O using non-blocking and asynchronous hardware I/O, but for the I/O types which blocks or are complex to address, it uses the thread pool.

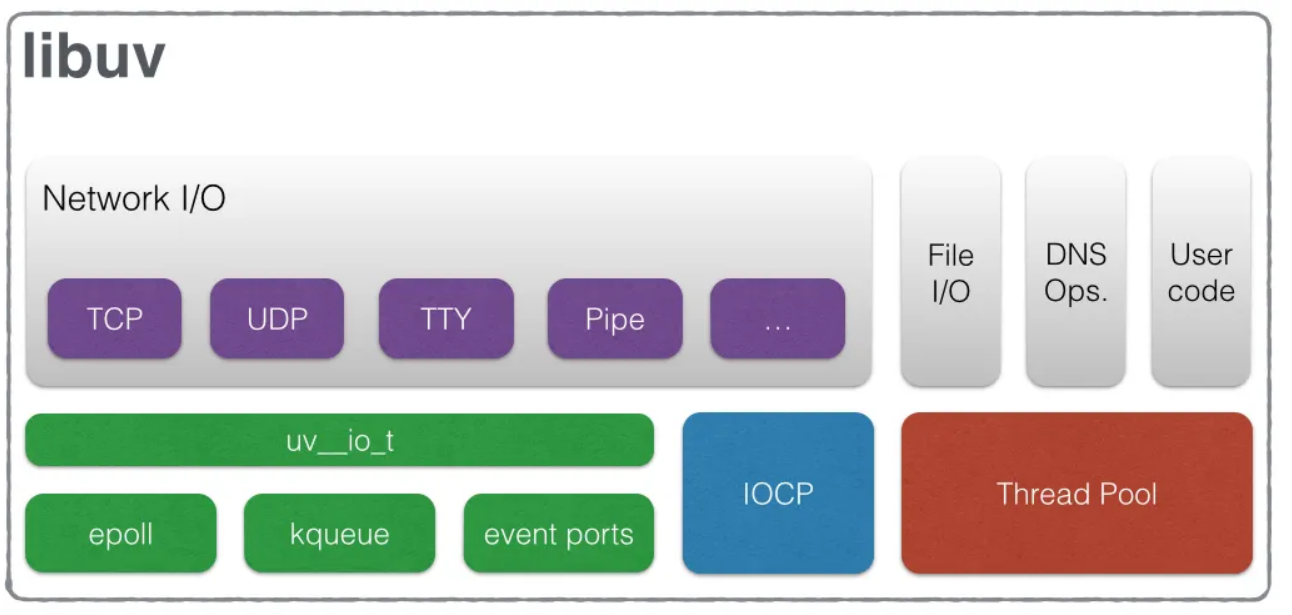
*However, I/O is not the only type of tasks performed on the thread pool. There are some Node.js crypto functions such as crypto.pbkdf2, async versions of crypto.randomBytes,*crypto.randomFill and async versions of zlib functions *which run on the libuv thread pool because they are highly CPU intensive. Running them on the threadpool prevents blocking of the event loop.*

## Gathering All Together

As we saw, in the real world it is really difficult to support all the different types of I/O (file I/O, network I/O, DNS, etc.) in all the different types of OS platforms. Some I/O can be performed using native hardware implementations while preserving complete asynchrony, and there are certain I/O types which should be performed in the thread pool so that the asynchronous nature can be guaranteed.

To govern this entire process while supporting cross-platform I/O, there should be an abstraction layer that encapsulates these inter-platform and intra-platform complexities and expose a generalized API for the upper layers of Node.

Now let’s see how libuv is composed. The following diagram is from the official libuv docs and describes how different types of I/O have been handled while exposing a generalized API.



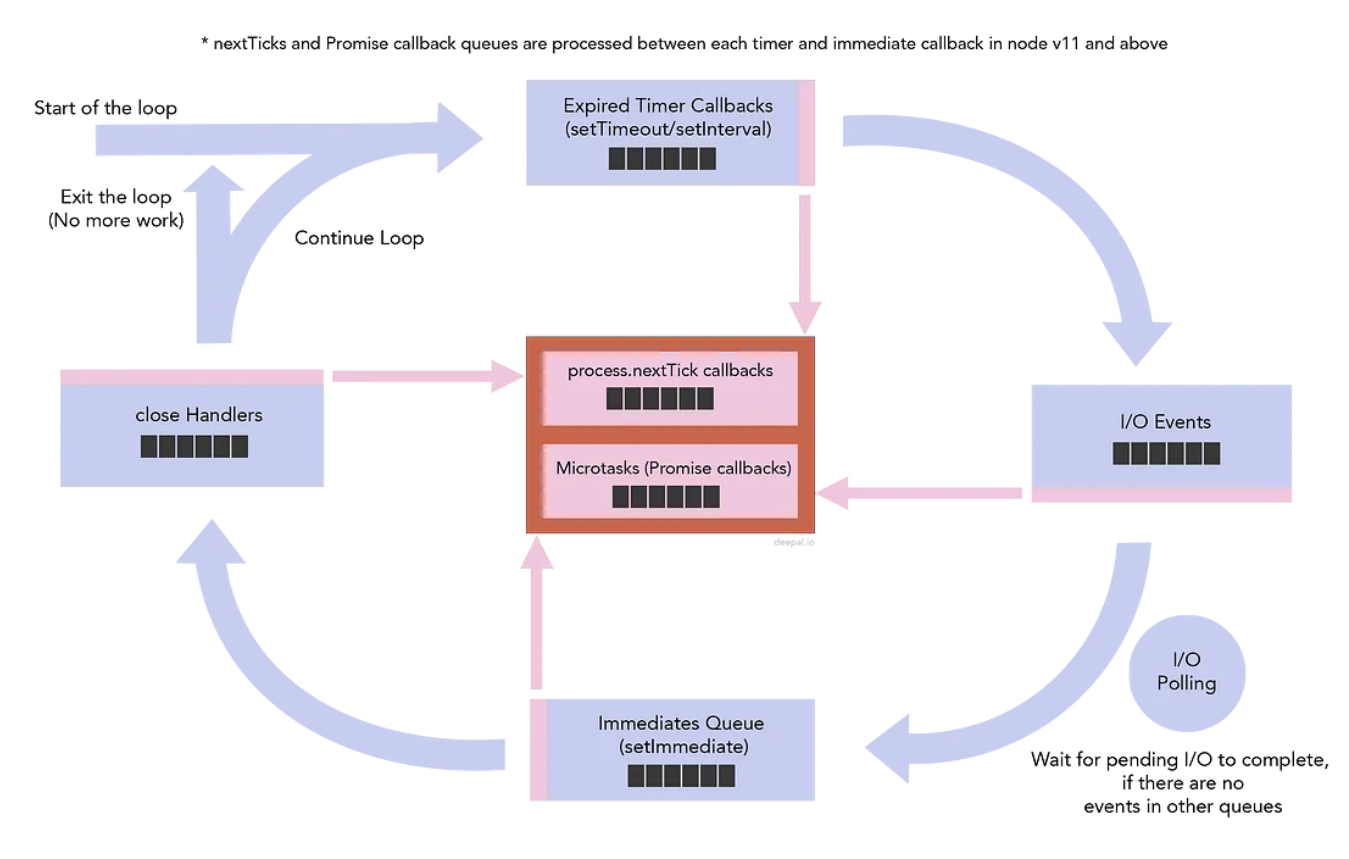
Now we know that the **Event Demultiplexer**, is not an atomic entity, but a collection of an I/O processing APIs abstracted by the Libuv and exposed to the upper layers of NodeJS. It’s not only the event demultiplexer that libuv provides for Node. Libuv provides the entire event loop functionality to NodeJS including the event queuing mechanism.

There are 4 main types of queues that are processed by the native libuv event loop.

* **Expired timers and intervals queue** — consists of callbacks of expired timers added using setTimeout or interval functions added using setInterval.
* **IO Events Queue** — Completed IO events
* **Immediates Queue** — Callbacks added using setImmediate function
* **Close Handlers Queue**— Any close event handlers.

Besides these 4 main queues, there are additionally 2 interesting queues which I previously mentioned as ‘intermediate queues’ and are processed by Node. Although these queues are not part of libuv itself but are parts NodeJS. They are,

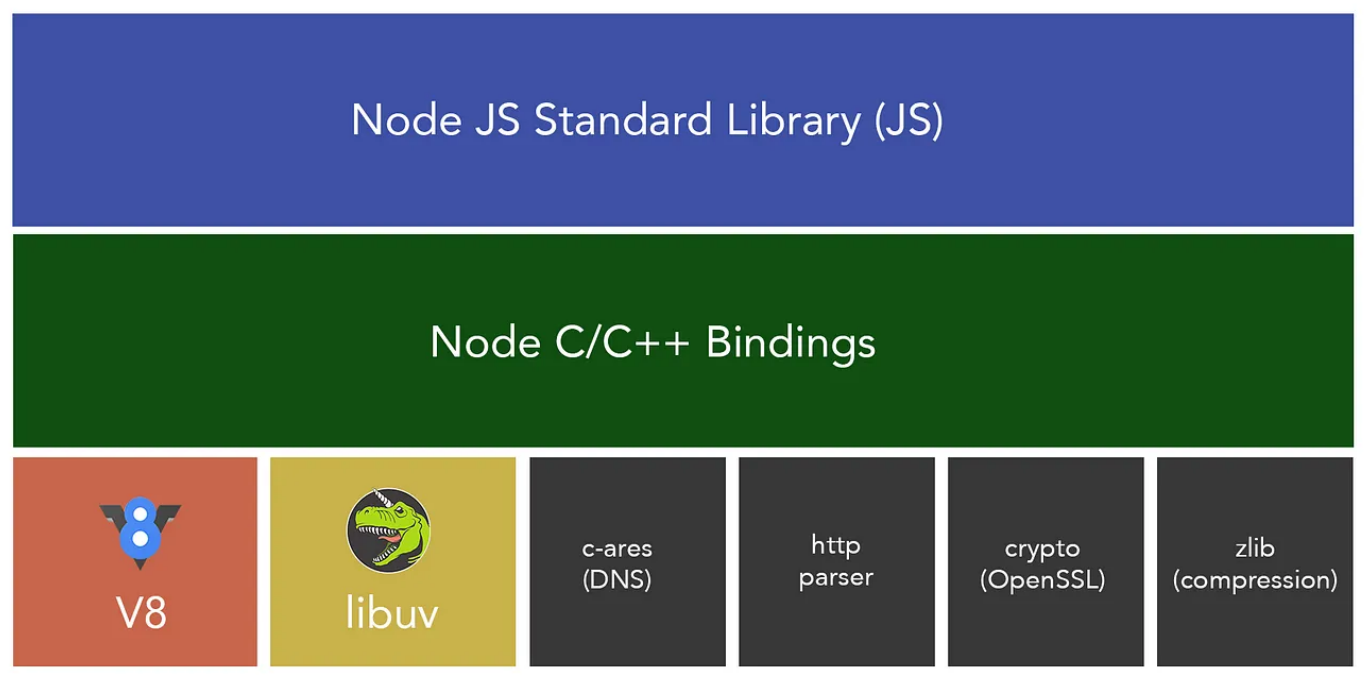
* **Next Ticks Queue** — Callbacks added using process.nextTick function
* **Other Microtasks Queue** — Includes other microtasks such as resolved promise callbacks



## Next tick queue vs Other Microtasks

Next tick queue has even higher priority over the Other Micro tasks queue. Although, they both are processed in between two phases of the event loop when libuv communicates back to higher layers of Node at the end of a phase. You’ll notice that I have shown the next tick queue in dark red which implies that the next tick queue is emptied before starting to process resolved promises in the microtasks queue.

Finally, now you know what event loop is, how it is implemented and how Node handles asynchronous I/O. Let’s now look at where Libuv is in the NodeJS architecture.



# Timers queue

When you add a timer using setTimeout or an interval using setInterval, Node will add the timer to the timers heap, which is a data structure accessed through libuv. At the timers phase of the event loop, Node will check the timers heap for expired timers/intervals and will call their callbacks respectively. If there are more than one timer which were expired (set with the same expiration period), they will be executed in the order they were set.

# Immediates Queue

Although the immediates queue is somewhat similar to timeouts on how it behaves, it has some of its own unique characteristics. Unlike timers which we cannot guarantee when its callback gets executed even though the timer expiration period is zero, immediates queue is guaranteed to be processed immediately after the I/O phase of the event loop. Adding an event(function) to the immediates queue can be done using setImmediate function

***What is I/O?***

*Generally, any work which involves external devices except the CPU is called I/O. The most common abstract I/O types are File Operations and TCP/UDP network operations.*

## I/O Polling

Now, you may be wondering what I/O polling is. Although I merged I/O callbacks queue and I/O polling into a single phase in the event loop diagram (diagram1), I/O Polling happens after consuming the completed/errored I/O callbacks.

But, the most important fact in I/O Polling is, **it’s optional**. I/O poling will or will not happen due to certain situations. To understand this thoroughly, let’s have a look at how this is implemented in libuv.

## Some words about Threadpool

So far, we didn’t talk about the thread pool in this article, the thread pool is mostly used to perform all File I/O operations, getaddrinfo and getnameinfo calls during DNS operations merely due to the complexities of File I/O in different platforms. Since the size of the thread pool is limited (default size is 4), multiple requests to file system operations can still be blocked until a thread becomes available to work. However, the size of the thread pool can be increased up to **128** (at the time of this writing) using the environment variable UV\_THREADPOOL\_SIZE, to increase the performance of the application.

**dns.lookup() vs dns.resolve\*()**

If you have gone through NodeJS docs for [dns](https://nodejs.org/api/dns.html) module, you might have seen that there are two ways to resolve a host name to an IP address using dns module. They are either using dns.lookup or using one of the dns resolve functions such as dns.resolve4, dns.resolve6 etc. While these two approaches seem to be the same, there is a clear distinction between them on how they work internally.

dns.lookup function behaves similarly to how ping command resolves a hostname. It calls the getaddrinfo function in operating system’s network API. Unfortunately, this call is not an asynchronous call. Therefore to mimic the async behavior, this call is run on libuv’s threadpool using theuv\_getaddrinfo function. This could increase the contention for threads among other tasks which run on the threadpool and could result in a negative impact to the application’s performance. It is also important to revise that libuv threadpool contains only 4 threads by default. Therefore, four parallel dns.lookup calls can entirely occupy the threadpool starving other requests (file I/O, certain crypto functions, possibly more dns lookups).

In contrast, dns.resolve() and other dns.resolve\*() behave in a different way. Here is how dns.resolve\* is described in [official docs](https://nodejs.org/api/dns.html#dns_implementation_considerations).

These functions are implemented quite differently than [dns.lookup()](https://nodejs.org/api/dns.html#dns_dns_lookup_hostname_options_callback). They do not use [getaddrinfo(3)](http://man7.org/linux/man-pages/man3/getaddrinfo.3.html) and they *always* perform a DNS query on the network. This network communication is always done asynchronously, and does not use libuv's threadpool.

NodeJS provides the DNS resolving capabilities using a popular dependency called [c-ares](https://github.com/c-ares/c-ares). This library does not depend on libuv’s threadpool and runs entirely on the network.

dns.resolve does not overload the libuv threadpool. Therefore, it is desirable to use dns.resolve instead of dns.lookup unless there’s a requirement to adhere to configuration files such as /etc/nsswitch.conf, /etc/hosts which are considered during getaddrinfo.

But there’s an even bigger problem!

Let’s say you are using NodeJS to make an HTTP request to www.example.com. First, it will resolve [www.example.com](http://www.example.com/) into an IP address. Then it will use the resolved IP to set up the TCP connection **asynchronously**. So, sending an HTTP request is a **two-step** process.

Currently, Both Node http and https modules internally use dns.lookup to resolve hostname to IP. During a failure of the DNS provider or a due to a higher network/dns latency, multiple http requests can easily keep the thread pool out-of-service for other requests. This has been a [raised concern](https://github.com/request/request/issues/2491) about http and https, but is still left as-is at the time of this writing, in order to stick to the native OS behavior. Making things worse, many userland http client modules such as request also use http and https under the hood and are affected by this issue.

If you notice drastic performance drop in your application in terms of file I/O, crypto or any other threadpool-dependent task, there are few things you can do to improve your application’s performance.

* You can increase the capacity of the threadpool up-to 128 threads by setting UV\_THREADPOOL\_SIZE environment variable.
* Resolve hostname to IP address using dns.resolve\* function and use IP address directly.

Since “the event loop” is nothing but a programming pattern, V8 allows the ability to plug-in an external event loop implementation to work with its JavaScript runtime. Using this flexibility, the Chrome browser uses [**libevent**](https://libevent.org/) as its event loop implementation, and NodeJS uses [**libuv**](https://libuv.org/)to implement the event loop. Therefore, chrome’s event loop and NodeJS’s event loop are based on two different libraries and which have differences, but they also share the similarities of the common “Event Loop” programming pattern.

### **poll phase**

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.

fs.readFile(\_\_filename, () => {

setTimeout(() => { console.log('timeout'); }, 0);

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

});

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.

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.

