**NodeJS**

**NodeJS** = > V8 (handles JS code) + **libuv** (handles fs + network)

**LibUV:**

LibUV is the core engine that powers Nodejs. LibUV provides support for asynchronous I/O operations.  It also supports non-blocking network support, asynchronous file system access, etc.

To understand LibUV better, we will have to understand the following three concepts first and then about more on LibUV by keeping these three in mind.

1. **Handles:** Handles represent long-lived objects capable of performing certain operations while active. When it completes the job, handles will invoke the corresponding callbacks. As long as a handle is active, the event loop will continue running. Some examples of handles are TCP servers that get their connection callback called every time there is a new connection, timers, signals, and child processes.
2. **Requests:** Abstractions for short-lived operations. In contrast to handles that are considered as objects, requests can be thought of as functions or methods. Requests are used to write data on handles. Like handles, active requests will also keep the event loop alive.Another essential concept in LibUV is thread pool. LibUV delegates all the heavy work to a pool of worker threads.
3. **Thread pool:** The thread pool takes care of the file I/O and DNS lookup. All the callbacks, however, are executed on the main thread. Since Node 10.5, worker threads can also be used by the programmer to run Javascript in parallel.

For some standard library function calls, the node C++ side and libuv decide to do expensive calculations outside of the event loop entirely. LibUV creates something called a thread pool. This **thread pool** **consists** of **four threads** by default but can be changed using the **UV\_THREADPOOL\_SIZE**.

The I/O (or event) loop is the central part of libuv. It establishes the content for all I/O operations, and it’s meant to be tied to a single thread. One can run multiple event loops as long as each runs in a different thread.

**Features of libuv:**

* Full-featured event loop backed by epoll (Linux), kqueue (OSX), IOCP (Windows), event ports (SunOS).
* Asynchronous TCP (net module) and UDP (dgram module)
* Asynchronous DNS resolution (used partly for the dns module)
* Asynchronous file, file system operations & events (fs module)
* ANSI escape code controlled TTY
* Thread pool and Signal handling
* Child processes
* High-resolution clock
* Threading and synchronization primitives.
* Inter-Process Communication using sockets and Unix domain sockets (Windows)

**Event or I/O loop:** Event or I/O loop uses a **single threaded asynchronous I/O approach**, hence it is tied to a single thread. In order to run multiple event loops, **each of these event loops** must be run on a **different thread**. It is **not thread-safe by default**with some exceptions.

Libuv maintains an **Event queue**and **event demultiplexer**. The loop listens for incoming I/O and emits **event** for each request. The requests are then assigned to specific handler (OS dependent). After successful execution, **registered callback is enqueued**in event queue which are continuously executed one by one.   
**Note:** The current time required during entire process is cached by libuv at beginning of each iteration of loop to minimize frequent system calls.

**Ex:** If a network request is made, a callback is registered for that request, and the task is assigned to the handler. Until it is performed other operations carry on. On successful execution/termination, the registered callback is en-queued in the event queue which is then executed by the main thread after the execution of previous callbacks already present in the queue.

It uses platform-specific mechanisms as mentioned earlier to achieve the best compatibility and performance **epoll (Linux), kqueue (OSX), IOCP (Windows), event ports (SunOS).**

**File I/O:** File I/O is implemented in libuv using a **global thread pool** on which all loops can queue work. It allows disk to be used in an abstracted asynchronous fashion. It breaks down complex operations into simpler operations to facilitate **async-like** behavior.

**Example:** If the program instructs to write a buffer to a specific file, in normal situations, the I/O will be blocked until the operation is successful/terminated. However, libuv abstracts this into an async manner by putting an **event notification** which would notify about operations success/failure after it is finished, until then the other I/O operations can be performed hassle-free.   
**Note:** Thread-safety is not assured by libuv (with few exceptions)

Unlike event loop, File I/O uses platform-independent mechanisms. There are 3 kinds of **async disk APIs** that are handled by File I/O:

1. linux AIO (supported in kernel)
2. posix AIO (supported by linux, BSD, Mac OS X, solaris, AIX, etc)
3. Windows’ overlapped I/O

**Benefits:**

* Disk operations are performed asynchronously.
* High level operations can be broken down to simpler disk operations which facilitate rectifying the information.
* Disk thread can use vector operations like readv & writev allowing more buffers to be passed.