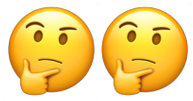
Node.js Threads



Essehemy

Threads vs Processes

TL;DR

A. Both processes and threads are independent sequences of execution.

B. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

Threads vs Processes

Details

A. *Threads are easier to create than processes since they don't require a separate address space.*
B. *Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.*
C. *Threads are considered lightweight because they use far less resources than processes.*
D. *Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other.*
E. *A process can consist of multiple threads.*

Why is Node Scalable

- The secret to Node's scalability is that it uses a small number of threads to handle many clients. If Node can make do with fewer threads, then it can spend more of your system's time and memory working on clients rather than on paying space and time overheads for threads (memory, context-switching).

- Node is fast when the work associated with each client at any given time is “small” (Be aware of REDOS, JSON DOS).

- In a one-thread-per-client system like Apache, each pending client is assigned its own thread. If a thread handling one client blocks, the operating system will interrupt it and give another client a turn. The operating system thus ensures that clients that require a small amount of work are not penalized by clients that require more work.

- Because Node handles many clients with few threads, if a thread blocks handling one client's request, then pending client requests may not get a turn until the thread finishes its callback or task. *The fair treatment of clients is thus the responsibility of your application*. This means that you shouldn't do too much work for any client in any single callback or task.

Node Threads Types

1. Event Loop (aka the main loop, main thread, event thread, etc.)

2. pool of `n` Workers in a Worker Pool (aka the threadpool)

1- 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.

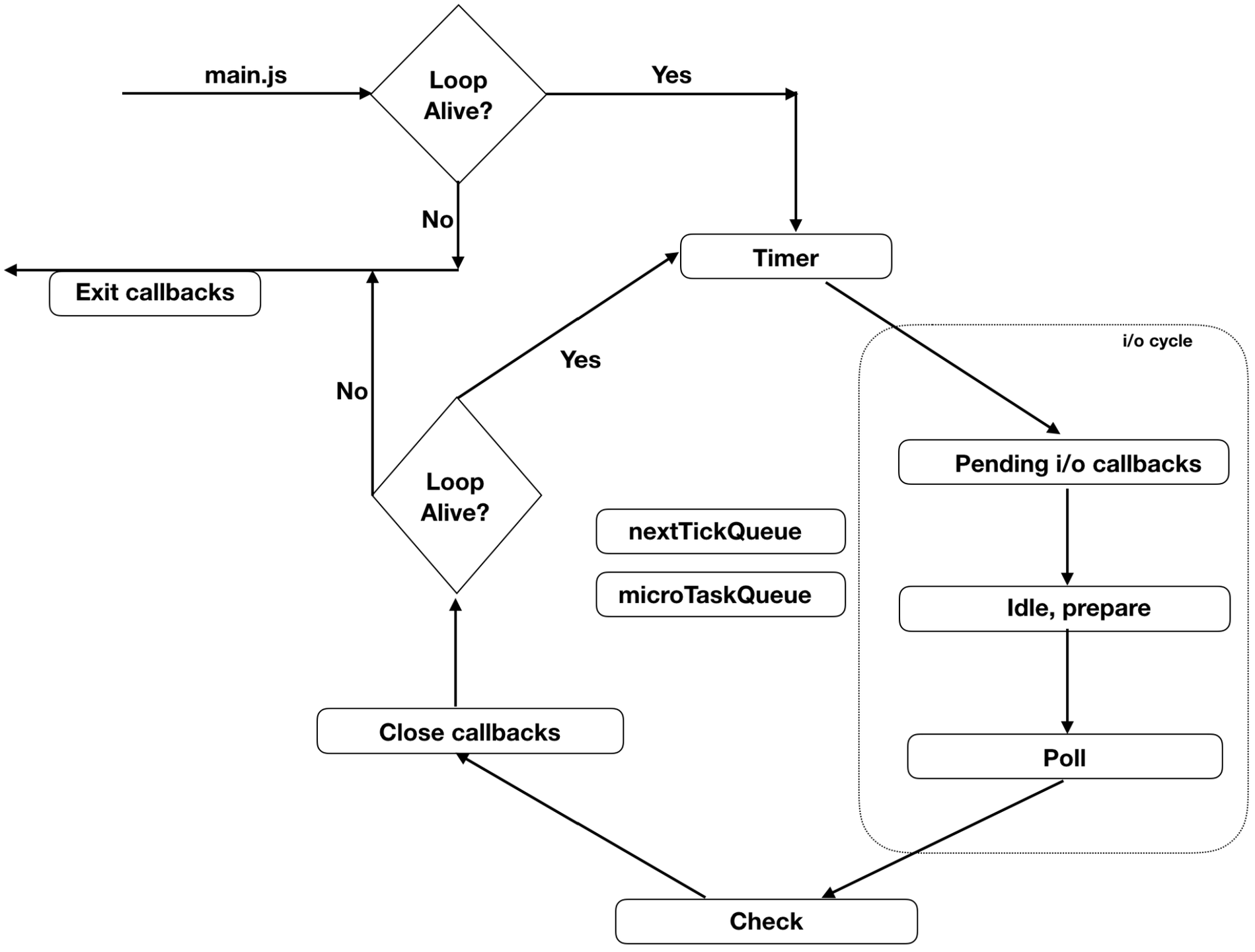
- Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the **poll** queue to eventually be executed.

- 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.

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.


main.js
Loop Alive?
Yes
No
Exit callbacks
Timer
i/o cycle
Yes
No
Pending i/o callbacks
Loop Alive?
nextTickQueue
microTaskQueue
Idle, prepare
Poll
Close callbacks
Check

2- Worker Threads

- Node uses the Worker Pool to handle "expensive" tasks. This includes I/O for which an operating system does not provide a non-blocking version, as well as particularly CPU-intensive tasks.

These are the Node module APIs that make use of this Worker Pool:

1. I/O-intensive
    1. DNS: dns.lookup(), dns.lookupService().
    2. File System: All file system APIs except `fs.FSWatcher()` and those that are explicitly synchronous use libuv's threadpool.
2. CPU-intensive
    1. Crypto: crypto.pbkdf2(), crypto.randomBytes(), crypto.randomFill().
    2. Zlib: All zlib APIs except those that are explicitly synchronous use libuv's threadpool.

But, How To Handle CPU Intensive Tasks?🤔

Remember, the Event Loop should orchestrate client requests, not fulfill them itself. For a complicated task, move the work off of the Event Loop onto a Worker Pool.

2 Approaches

1. Partitioning

2. offloading

    1. You can use the built-in Node Worker Pool by developing a C++ addon. On older versions of Node, build your C++ addon using NAN, and on newer versions use N-API. npm module node-webworker-threads offers a JavaScript-only way to access Node's Worker Pool.

    2. You can create and manage your own Worker Pool dedicated to computation rather than Node's I/O-themed Worker Pool.

        A. processes: The most straightforward ways to do this is using Child Process or Cluster.

        B. Threads: experimental built in worker_threads module.

Resources

- https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
- https://nodejs.org/en/docs/guides/dont-block-the-event-loop/
- http://voidcanvas.com/nodejs-event-loop/
- https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS
- https://medium.com/dailyjs/threads-in-node-10-5-0-a-practical-intro-3b85a0a3c953
- https://nodejs.org/api/worker_threads.html
- https://github.com/audreyt/node-webworker-threads