Episode-10 | Thread pool in libuv

Whenever there's an asynchronous task, V8 offloads it to libuv. For example, when reading a file, libuv uses one of the threads in its thread pool. The file system (fs) call is assigned to a thread in the pool, and that thread makes a request to the OS. While the file is being read, the thread in the pool is fully occupied and cannot perform any other tasks. Once the file reading is complete, the engaged thread is freed up and becomes available for other operations. For instance, if you're performing a cryptographic operation like hashing, it will be

assigned to another thread. There are certain functions for which libuv uses the thread pool.

In Node.js, the default size of the thread pool is 4 threads:

```
UV_THREADPOOL_SIZE=4
```

Now, suppose you make 5 simultaneous file reading calls. What happens is that 4 file calls will occupy 4 threads, and the 5th one will wait until one of the threads is free.

Q: When does libuv use the thread pool?

Whenever you perform tasks like file system (fs) operations, DNS lookups (Domain Name System), or cryptographic methods, libuv uses the thread pool.

Now that you have enough knowledge, answer this question: Is Node.js single-threaded or multi-threaded?

If you're dealing with synchronous code, Node.js is single-threaded. But if you're dealing with asynchronous tasks, it utilizes libuv's thread pool, making it multi-threaded.

the order of execution is not gauranteed over here which thread executes first will win

---

Q: Can you change the size of the thread pool?

A: Yes, you can change the size of the thread pool by setting the `UV_THREADPOOL_SIZE` environment variable. For example, you can set it to 8 like this:

```
process.env.UV_THREADPOOL_SIZE = 8;
```

If your production system involves heavy file handling or other tasks that benefit from additional threads, you can adjust the thread pool size accordingly to better suit your needs.

Q: Suppose you have a server with many incoming requests, and users are hitting APIs. Do these APIs use the thread pool?

A: No.

In the libuv library, when it interacts with the OS for networking tasks, it uses sockets. Networking operations occur through these sockets. Each socket has a socket descriptor, also known as a file descriptor (although this has nothing to do with the file system).

When an incoming request arrives on a socket, and you want to write data to this connection, it involves blocking operations. To handle this, a thread is created for each request. However, creating a separate thread for each connection is not practical, especially when dealing with thousands of requests.

Instead, the system uses efficient mechanisms provided by the OS, such as `epoll` (on Linux) or `kqueue` (on macOS). These mechanisms handle multiple file descriptors (sockets) without needing a thread per connection.

Here's how it works:

- `epoll` **(Linux)** and `kqueue` **(macOS)** are notification mechanisms used to manage many connections efficiently.
- When you create an `epoll` or `kqueue` descriptor, it monitors multiple file descriptors (sockets) for activity.
- The OS kernel manages these mechanisms and notifies libuv of any changes or activity on the sockets.
- This approach allows the server to handle a large number of connections efficiently without creating a thread for each one.

The kernel-level mechanisms, like `epoll` and `kqueue`, provide a scalable way to manage multiple connections, significantly improving performance and resource utilization in a high-concurrency environment.


CONNECTION
THERE'S ACTIVITY ON A CONNECTION!
THANK YOU, EPOLL!
EPOLL

File Descriptors (fds) and Socket Descriptors

File Descriptors (FDs) are integral to Unix-like operating systems, including Linux and macOS. They are used by the operating system to manage open files, sockets, and other I/O resources.

Socket descriptors are a special type of file descriptor used to manage network connections. They are essential for network programming, allowing processes to communicate over a network.

Event Emitters

Event Emitters are a core concept in Node.js, used to handle asynchronous events. They allow objects to emit named events that can be listened to by other

parts of the application. The `EventEmitter` class is provided by the Node.js `events` module. Here's a brief overview:

- **Creating an EventEmitter**: You create an instance of `EventEmitter` and use the `on` method to register event listeners.
- **Emitting Events**: Use the `emit` method to trigger events and pass data to listeners.
- **Handling Events**: Listeners (functions) handle the emitted events and perform actions based on the event data.

Streams

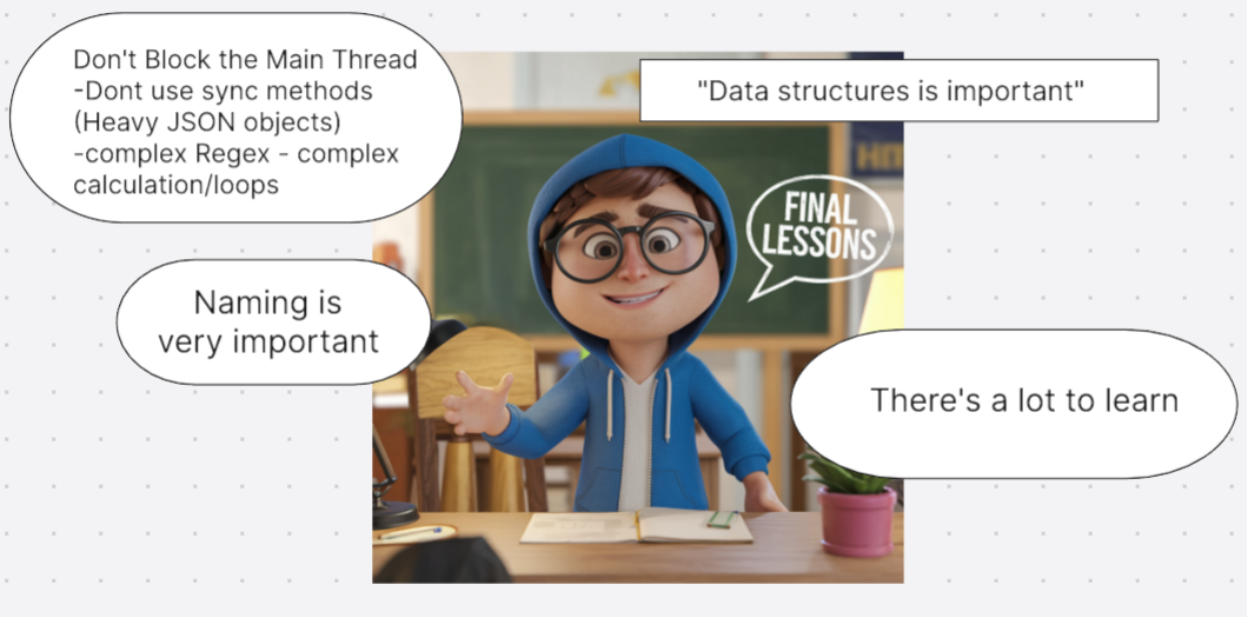
Streams in Node.js are objects that facilitate reading from or writing to a data source in a continuous fashion. Streams are particularly useful for handling large amounts of data efficiently.

Buffers

Buffers are used to handle binary data in Node.js. They provide a way to work with raw memory allocations and are useful for operations involving binary data, such as reading files or network communications.

Pipes in Node.js

Pipes in Node.js are a powerful feature for managing the flow of data between streams. They simplify the process of reading from a readable stream and writing to a writable stream, facilitating efficient and seamless data processing.


Don't Block the Main Thread
-Dont use sync methods
(Heavy JSON objects)
-complex Regex - complex
calculation/loops
"Data structures is important"
FINAL LESSONS
Naming is
very important
There's a lot to learn