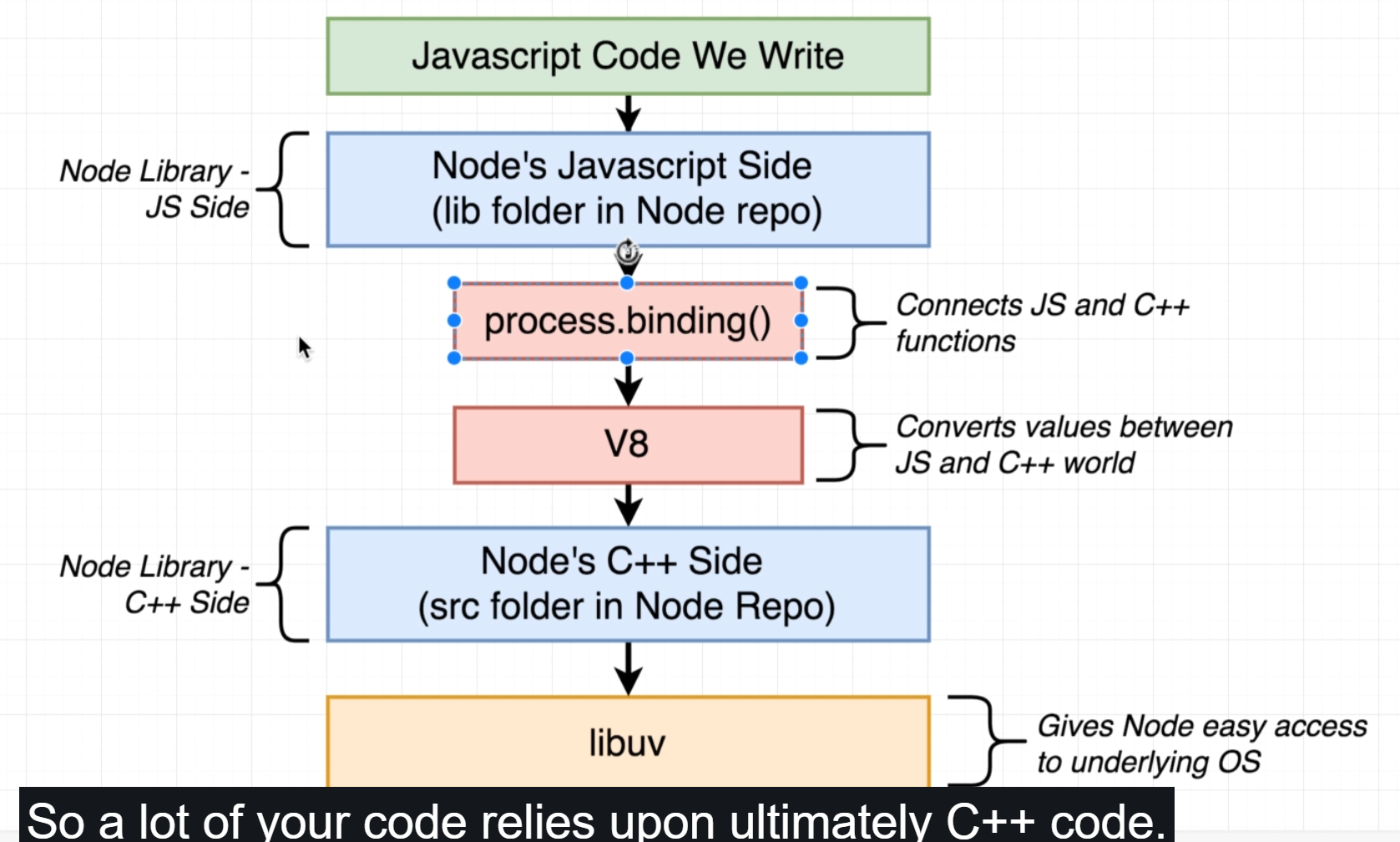


**NodeJS** là **JS runtime** dc xây dựng dựa trên **V8 JS Engine**(**Open Source**) của **Chrome** dc phát triển bởi GG và **LibUV** (C++ Lib)

**NodeJS** cung cấp những API thích hợp để truy cập vào chức năng dc thực hiện trong V8 và LibUV.



Source code NodeJS có 2 folder:

* **Lib** : chứa code JS của NodeJS.
* **Src**: chứa code C++ của NodeJS.

Các hàm hỗ trợ của Core module NodeJS viết bằng JS dc chứa trong **lib** khi ta gọi hàm **NHƯNG** trong này chỉ **check các error input** và **CUỐI CÙNG** gọi đến 1 hàm khác xử lý ở phía C++ thông qua **process.binding()** – là cầu nối giữa code JS và code C++.

Những gì ta gọi hàm ở JS dc xử lý chính thức ở code C++.

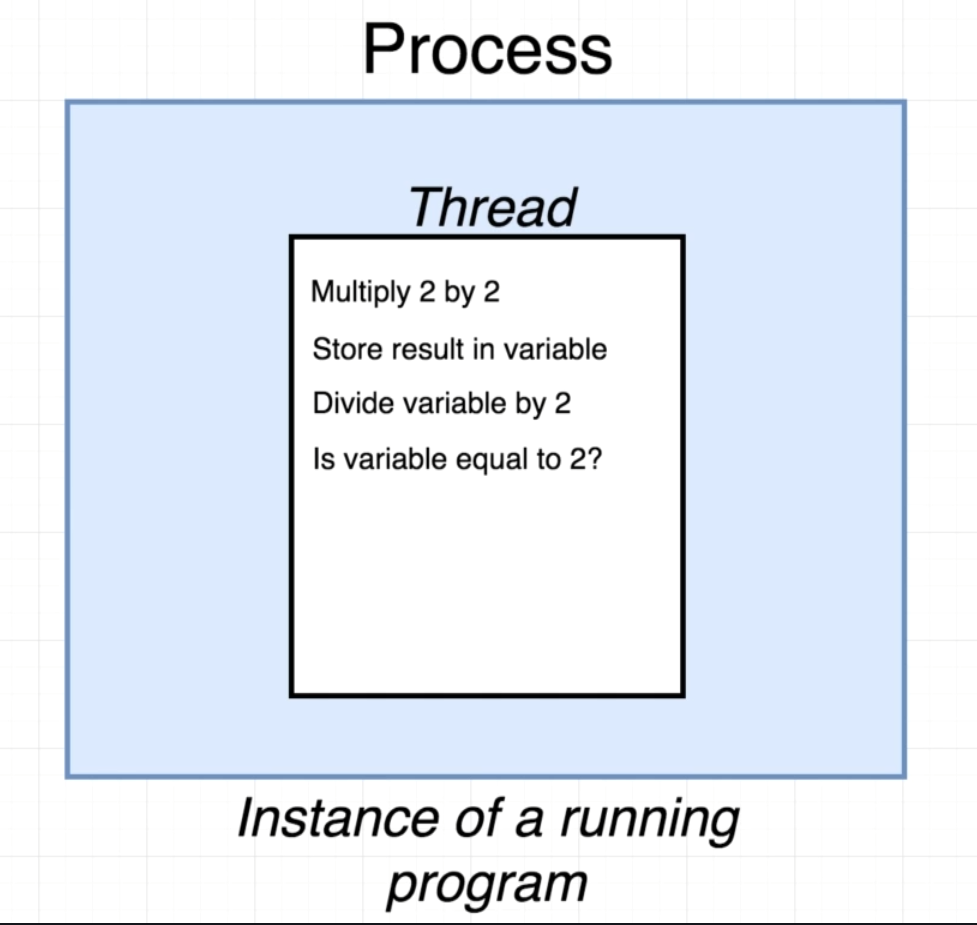
* **Khi viết code JS cuối cùng cũng thực thi dựa trên code C++.**

**V8** đóng vai trò trung gian để convert các giá trị của JS qua các giá trị tương đương bên C++. Ví dụ như: Object, String, Array,...

**LibUV** dc sử dụng cho rất nhiều cấu trúc xử lý trên phía C++.

**Event Loop** dc Node sử dụng để xử lý **Async code**.

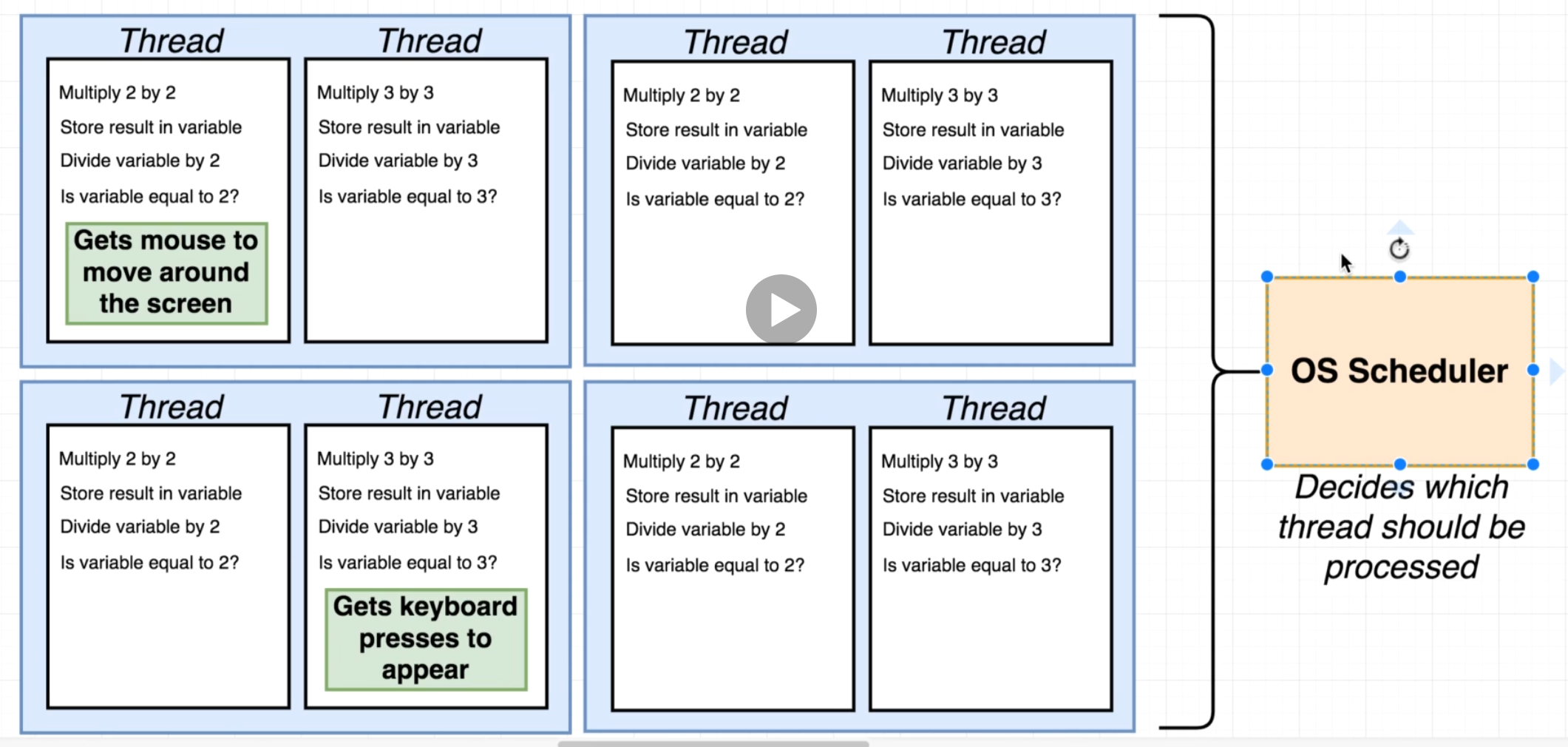
Thread



**PROCESS** sẽ dc tạo ra khi ta run 1 program nào đó.

Trong 1 **PROCESS** có thể có 1 hay nhiều **THREAD**.

**THREAD** giống như 1 todo-list chứa các chỉ dẫn cần **CPU** thực hiện từng bước một từ trên xuống.

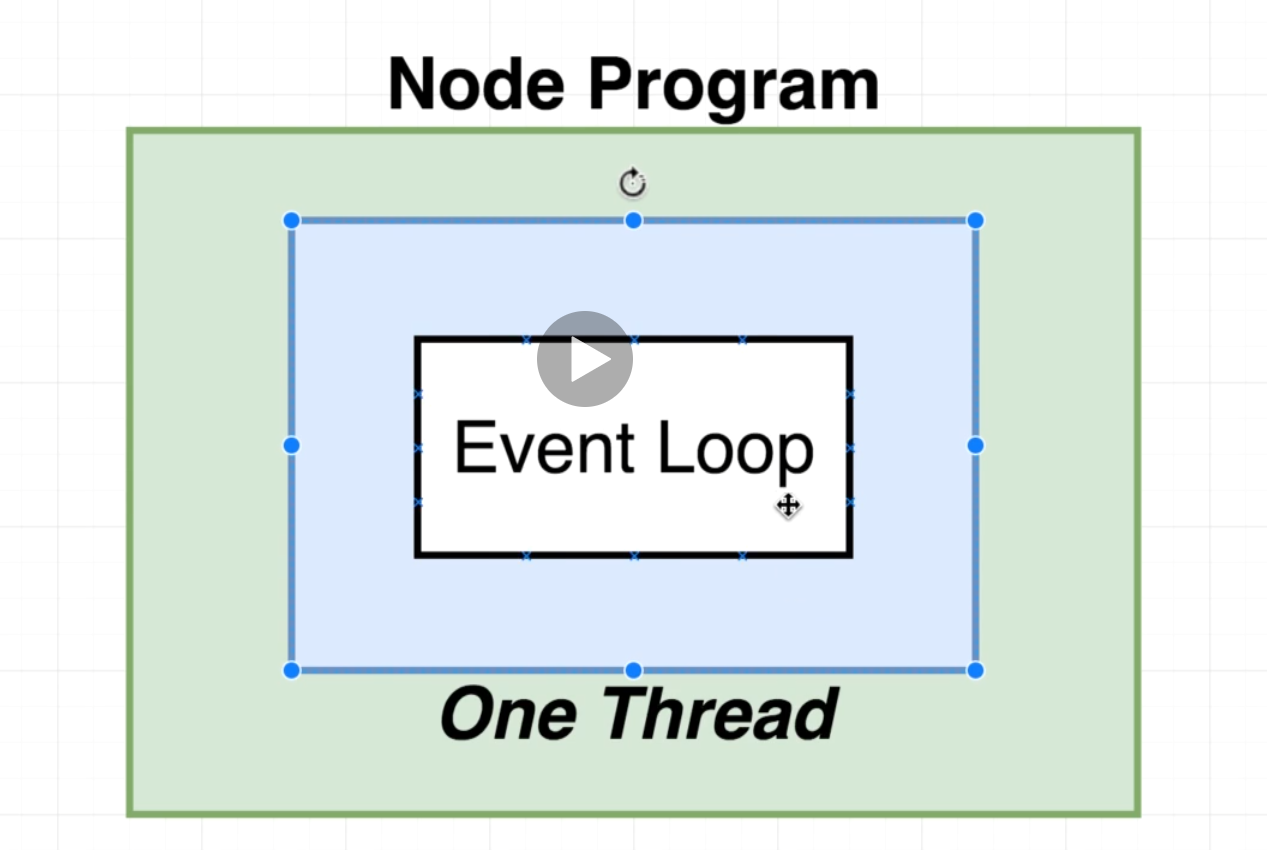


**SCHEDULING** của **OS** có khả năng quyết định **THREAD** nào sẽ được thực thi tại bất kỳ thời điểm.

**OS SCHEDULER** sẽ xem **THREAD** nào dag yêu cầu để thực thi và tìm ra cách xử lý 1 số lượng công việc trên từng **THREAD** đó để đảm bảo các **THREAD** đó ko phải đợi quá lâu.

Để thực thi các **THREAD** nhanh hơn hay thực thi nhiều hơn trong 1 khoảng tgian:

* Bổ sung thêm CPU core



Khi run chương trình **Node** trên máy thì **Node** sẽ tự động tạo ra 1 **THREAD** để xử lý các code bên trong.

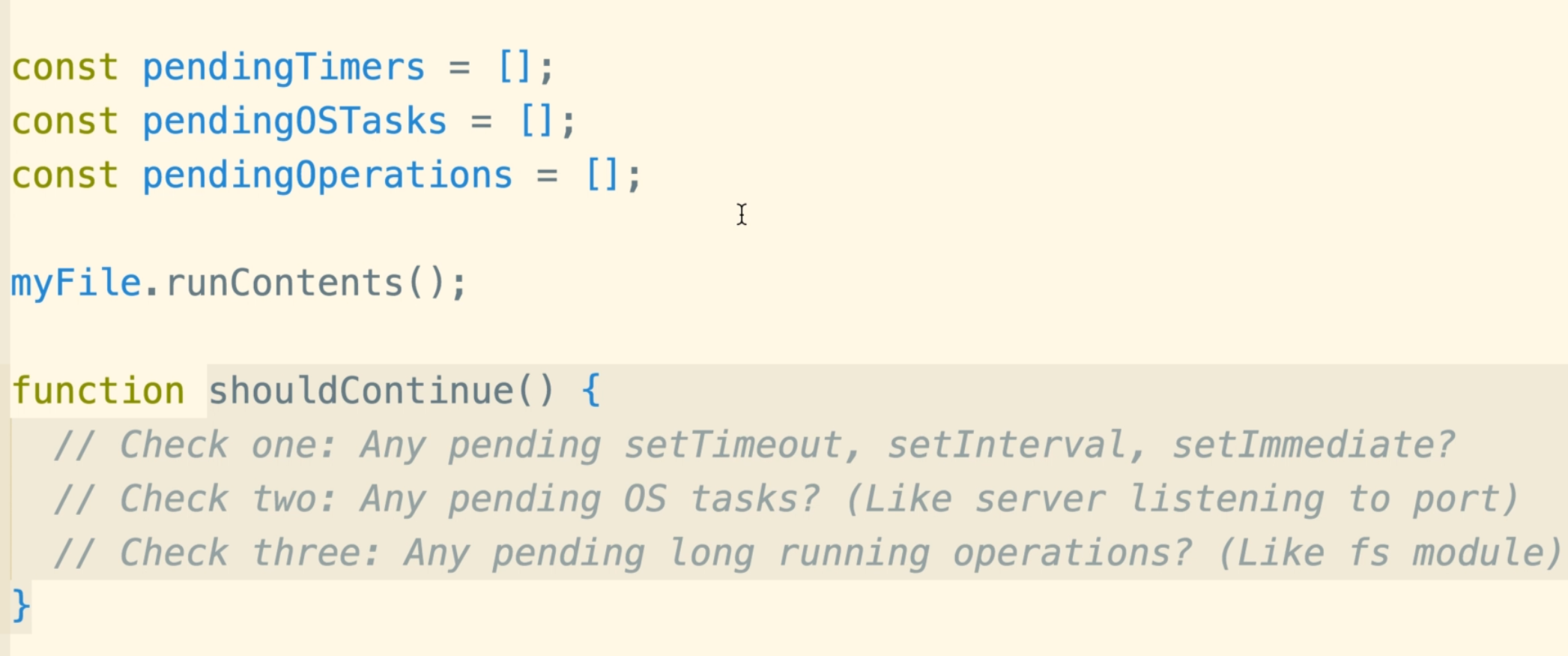
Bên trong **THREAD** đó tồn tại **EVENT LOOP**.

Có thể xem **EVENT LOOP** như là 1 cấu trúc điều khiển quyết định **THREAD** sẽ làm gì trong 1 thời điểm.

**EVENT LOOP** rất quan trọng trong vấn đề cải thiện **PERFORMANCE** của Node.

Hiểu dc cách thức hoạt động của **EVENT LOOP** thì sẽ sẵn sàng cho việc giải quyết vấn đề **PERFORMANCE**.

Khi chạy lệnh \***node app.js**\* thì **EVENT LOOP CHƯA** dc thực thi ngay mà Node sẽ đọc và thực thi code trong file **app.js** trước.

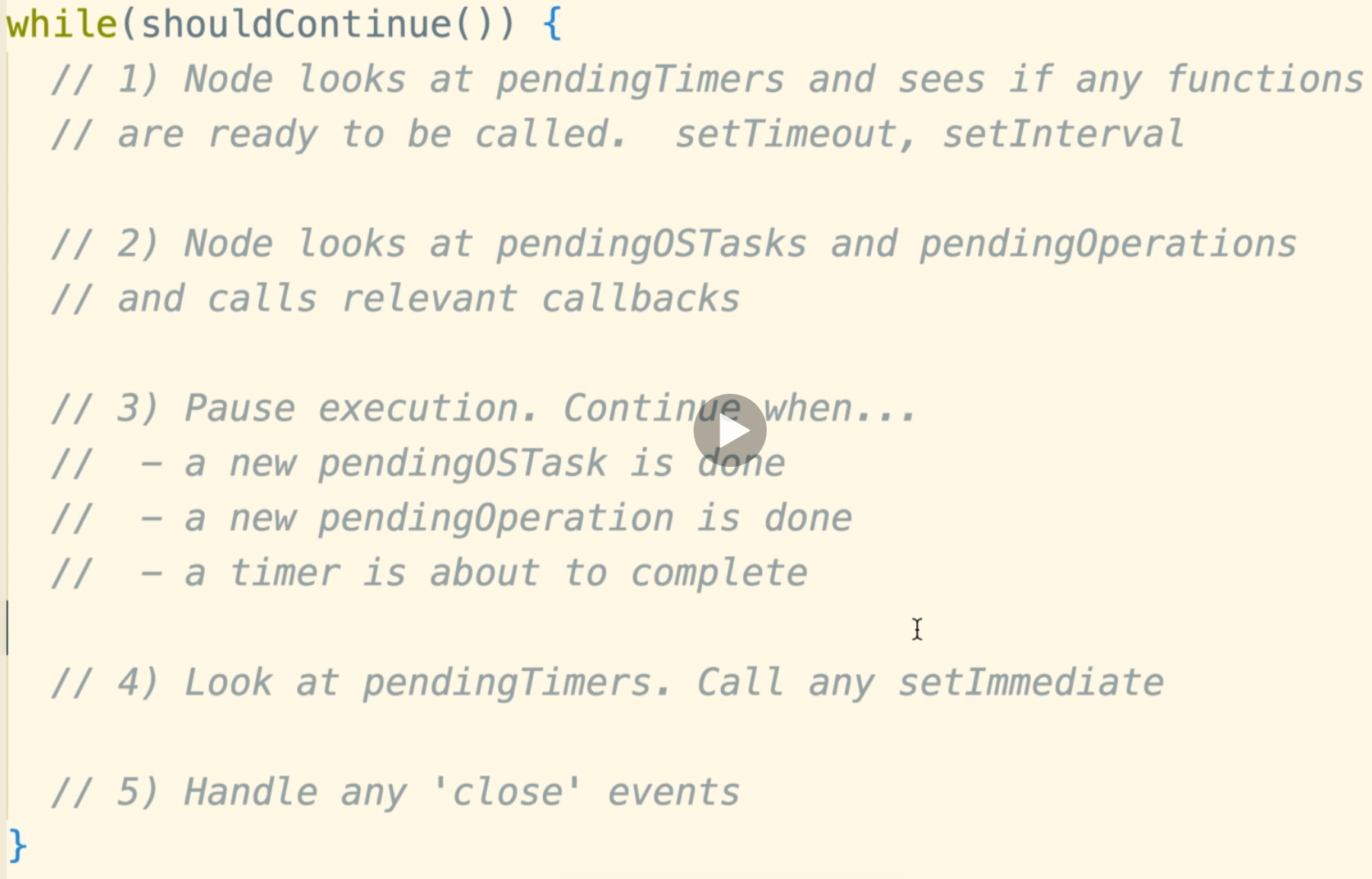


Để **EVENT LOOP** hoạt động thì **Node** phải thực hiện check **3** điều kiện sau:

* Tồn tại **setTimeout, setInterval, setImmediate**.
* Tồn tại **OS task** (VD: Server đang lắng nghe request ở PORT nào đó)
* Tồn tại **pending operations** (FS module)

**3** điều kiện này sẽ quyết định **EVENT LOOP** có tiếp tục chạy hay ko.

**Node** sẽ tự động **DETECT** **3** điều kiện khi chương trình chạy.

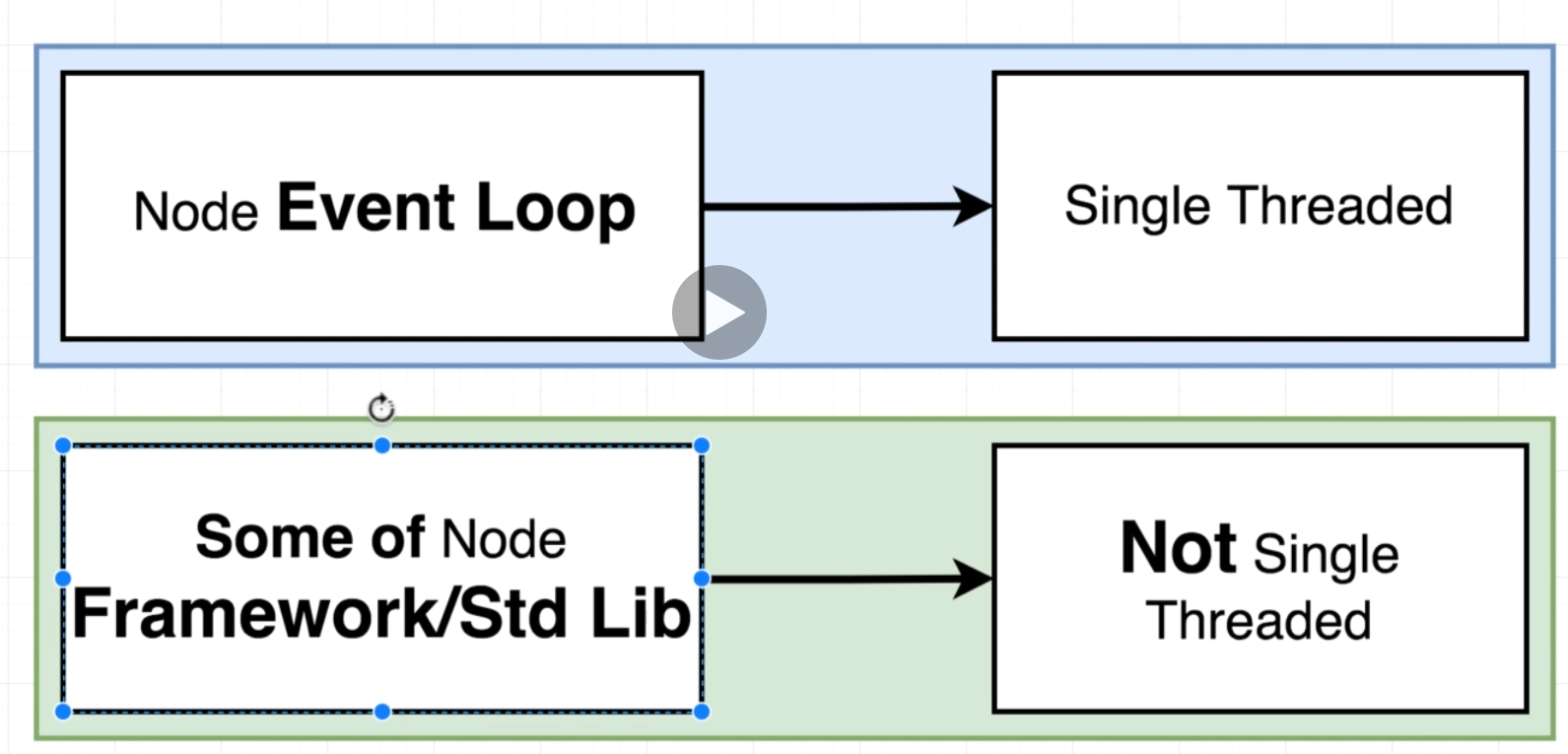


Trong **EVENT LOOP** sẽ thực thi **5** bước:

1. **Node** sẽ tìm trong các **pendingTimers** xem có Timer callback nào chuẩn bị dc thực thi hay ko. Nếu có **Node** sẽ gọi **callback** tương ứng với từng **Timers**. (**setTimeout**, **setInterval**).
2. **Node** sẽ tìm trong **pendingOSTask** và **pendingOperations (Blocking Operation)** và gọi các callback tương ứng.
3. **Tạm Dừng** thực thi và chờ cho các **Pendings** hoàn tất.

Nếu có quá nhiều **callbacks** cần xử lý thì **Node** chỉ thực thi các callbacks trong 1 tgian nhất định và sau đó tiếp tục **LOOP**. Node sẽ hoãn lại để xử lý các callbacks ở **LOOP** tiếp theo.

1. Node sẽ tìm ở **pendingTimers** nhưng lần này là **setImmediate** và gọi các callback khi hoàn tất thời gian countdown.
2. Xử lý các ‘close’ events callback.



Một số **HÀM** trong **LIBRARY** của **Node** **KO** phải **Single Thread**. (Một số Hàm ở trong **Node** sẽ chạy bên ngoài **EVENT LOOP** và ở ngoài **Single Thread** đó).

Nên nói **Node** là **Single Thread** **KO HOÀN TOÀN** đúng.