

Assignment-2

1. Explain the difference between single-threaded and multi-threaded execution.

Why is Node.js considered single-threaded?

Single-threaded execution means only one sequence of instructions runs at a time, while multi-threaded execution allows multiple sequences to run simultaneously. NODE.JS is considered single-threaded because it has only one main thread that executes JAVASCRIPT code. It uses an event loop on this single thread to handle asynchronous operations. However, for I/O operations, Node.js uses LIBUV which manages a thread pool behind the scenes, but the JAVASCRIPT execution context remains single-threaded. This design makes Node.js efficient for I/O-heavy applications without the complexity of managing multiple threads.

2. What is the main purpose of libuv in Node.js? Explain in your own words with an analogy.

Libuv is a C library that provides NODE.JS with asynchronous I/O capabilities. Its main purpose is to abstract away the complexities of different operating systems' I/O operations.

As an analogy, libuv is like a restaurant manager who delegates tasks to different waiters (threads). When a customer (your code) places an order, the manager doesn't prepare the food himself but assigns waiters to handle it. Once the food is ready, the manager notifies the customer. Similarly, LIBUV takes I/O requests from JavaScript, assigns them to its thread pool, and notifies the main thread when operations complete, allowing NODE.JS to remain non-blocking.

3. Differentiate between blocking and non-blocking I/O with examples.

Blocking I/O: Operations that block the execution thread until completed. The program cannot perform other tasks while waiting.

Example: In traditional synchronous file reading

- `fs.readFileSync()` - the thread waits until file reading is complete before moving to the next line.

Non-blocking I/O: Operations that don't block the execution thread. The program continues running while I/O operations happen in background.

Example: With `fs.readFile()` with a callback, the program continues executing other code while the file is being read, and the callback is called once the operation completes. This is why Node.js can handle many actually concurrent connections with a single thread.

4. What are the roles of package.json in a Node.js project? Name at least three key fields.

The package.json file is the manifest of a Node.js project.

- Its roles include:
1. Describing the project metadata (name, version, description)
 2. Managing dependencies and their versions
 3. Defining scripts for running, testing, and deploying the application
 4. Setting configuration for npm/yarn and other tools
- Three key fields are:
- dependencies: Lists production dependencies required by the application
 - scripts: Contains command shortcuts (start, test, build, etc.)
 - devDependencies: Lists dependencies needed only during development

5. Describe how the Event Loop works in Node.js. Include the role of the call stack and queues.

The Event Loop in NODE.JS is the mechanism that allows for non-blocking I/O operations. It works like this:

1. When Node.js starts, it initializes the Event Loop and processes the main script.
2. The call stack executes code in order, pushing function calls onto the stack and popping them when they return.
3. When an asynchronous operation (like a file read) is encountered, Node registers a callback and continues execution.
4. When the async operation completes, its callback is placed in the appropriate queue.
5. When the call stack is empty, the Event Loop checks the queues and pushes callbacks to the call stack for execution. The call stack keeps track of where we are in the program. Queues (like the timer queue, I/O queue, check queue, and microtask queue) hold callbacks waiting to be executed in a specific order determined by the Event Loop phases.

6. List the types of task queues in Node.js and their execution order (e.g., microtask vs macro task).

Node.js has several task queues that are processed in a specific order by the Event Loop:

1. Microtask Queue (highest priority): - process.nextTick queue (runs before other microtasks) - Promise queue (resolved promises)
2. Macrotask Queues (in order of execution): - Timer Queue (setTimeout, setInterval) - I/O Queue (callbacks for completed I/O operations) - Check Queue (setImmediate callbacks) - Close Queue (close events) After each macrotask phase, NODE.JS checks and empties the microtask queues before moving to the next phase. This ensures that microtasks always have priority over macrotasks.

7. What is the difference between process.nextTick() and a Promise? When do they execute?

process.nextTick() and Promises are both microtasks but differ in their execution timing:

process.nextTick():

- Executes callbacks before any other microtask or macrotask
- Has the highest priority in the event loop
- Runs immediately after the current operation completes, before returning to the event loop
- Can potentially block the event loop if used recursively

Promises:

- Execute after all nextTick callbacks have been processed
- Are part of the microtask queue but with lower priority than nextTick
- Run before the next macrotask (like setTimeout or I/O callbacks) Both execute before returning to the event loop for the next phase, but nextTick always runs before promise callbacks.

8. Name and explain three major components of the V8 engine used in Node.js.

Three major components of the V8 engine are:

- 1. JIT (Just-In-Time) Compiler:** Converts JAVASCRIPT code to optimized machine code instead of interpreting it. This makes execution much faster. It includes Ignition (bytecode interpreter) and TurboFan (optimizing compiler).
- 2. Garbage Collector:** Automatically manages memory allocation and release. V8 uses a generational garbage collector with different strategies for new and old objects, minimizing application pauses.
- 3. Hidden Classes:** V8's internal mechanism to optimize property access on JAVASCRIPT objects. It creates hidden classes behind the scenes to make property lookup faster by converting dynamic lookups to fixed offsets.

9. Mention two use cases where Node.js is NOT a suitable choice. Explain why.

Two use cases where NODE.JS is not suitable:

- 1. CPU-intensive Applications:** Applications requiring heavy computation like video encoding, scientific simulations, or machine learning algorithms are not ideal for Node.js. Since NODE.JS is single-threaded for JavaScript execution, CPU-bound tasks will block the event loop, preventing it from handling other requests and negating its concurrency advantages.
- 2. CRUD Applications with Relational Databases:** For simple CRUD applications heavily dependent on relational database operations, frameworks like Django, Ruby on Rails, or Laravel might be more efficient. These frameworks offer better built-in database integration, ORM capabilities, and admin interfaces which would require additional setup and configuration in Node.js.

10. What are the differences between npm and yarn? List pros and cons of each.

Differences between NPM and yarn:

NPM:

Pros: - Default package manager for NODE.JS

- Vast ecosystem and community support –

Improved performance in recent versions (npm 7+)

- No additional installation required

Cons:

- Historically slower than YARN
- Less deterministic in older versions
- Security vulnerabilities in the past

YARN:

Pros:

- Faster installation (especially YARN 2+)
- Better security with checksums
- Highly deterministic installations via lockfiles
- Offline cache for packages

Cons:

- Requires separate installation
- Some compatibility issues with certain packages
- Learning curve when switching from npm
- Less seamless integration with NODE.JS ecosystem YARN was created to address npm's shortcomings, but npm has improved significantly in recent versions, narrowing the gap between the two.

11. Using Promises, write a function `getData()` that resolves after 1 second with the message "Data fetched". Then, call it and log the result.

```

21 function getData() {
22   return new Promise((resolve, reject) => {
23     setTimeout(() => {
24       resolve("Data fetched");
25     }, 1000);
26   });
27 } // Calling the function
28 getData().then(result => { console.log(result); }) .catch(error => { console.error(error); });
29

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE **TERMINAL** PORTS COMMENTS

```

at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:170:5)
at node:internal/main/run_main_module:36:49

Node.js v22.13.0
[nodemon] app crashed - waiting for file changes before starting...
[nodemon] restarting due to changes...
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`
Data fetched

```

12. Create a script that sets two timers: one using `setTimeout` and another using `setImmediate`. Log which one executes first and explain why.

```

29 console.log("Start");
30 setTimeout(() => {
31   console.log("setTimeout executed");
32 }, 0);
33 setImmediate(() => {
34   console.log("setImmediate executed");
35 });
36 console.log("End");
37
38

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE **TERMINAL** PORTS COMMENTS

```

Node.js v22.13.0
[nodemon] app crashed - waiting for file changes before starting...
[nodemon] restarting due to changes...
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`
Start
End
setImmediate executed
setTimeout executed
[nodemon] clean exit - waiting for changes before restart

```

The execution order can vary between runs if this code is run directly in the main module. If run inside an I/O cycle (like a file read callback), `setImmediate` will consistently execute before `setTimeout`. This happens because `setImmediate` is designed to execute in the check phase of the event loop, while `setTimeout(fn, 0)` schedules execution in the timers phase. When the delay is 0ms, Node can't guarantee

exact timing due to process scheduling, so the order becomes unpredictable in the main module. Inside I/O callbacks, the event loop is already in the I/O phase and will next proceed to the check phase, making `setImmediate` execute first.

13. Write a function using `process.nextTick()` that logs "Tick callback executed". Then, call it along with a `Promise.resolve()` and a `setTimeout()`. Observe and explain the output order.

```
39 function tickFunction() {
40   process.nextTick(() => {
41     console.log("Tick callback executed");
42   });
43 }
44 console.log("Start");
45 setTimeout(() => {
46   console.log("Timeout callback executed");
47 }, 0);
48 Promise.resolve().then(() => {
49   console.log("Promise callback executed");
50 });
51 tickFunction();
52 console.log("End");
53
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`
Start
End
Tick callback executed
Promise callback executed
Timeout callback executed
[nodemon] clean exit - waiting for changes before restart
```

Start End Tick callback executed Promise callback executed Timeout callback executed
Explanation: The order follows the event loop priority.

First, synchronous code runs ('Start' and 'End'). After that, the microtasks queue is processed in order: `process.nextTick` callbacks have the highest priority, followed by Promise callbacks. Finally, the macrotasks from the timer queue (`setTimeout`) are executed.

14. Create a timer that logs "Done" every 2 seconds and stops after 3 times using setInterval.

```

54 let count = 0;
55 const maxExecutions = 3;
56 const intervalId = setInterval(() => {
57   console.log("Done");
58   count++;
59   if (count >= maxExecutions) {
60     clearInterval(intervalId);
61     console.log("Timer stopped after 3 executions");
62   }
63 }, 2000);
64 console.log("Timer started");
65

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```

[nodemon] restarting due to changes...
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`
Timer started
Done
[nodemon] restarting due to changes...
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`

```

This code creates an interval that executes every 2 seconds (2000ms). Each time it executes, it logs 'Done' and increments the counter. Once the counter reaches 3, the interval is cleared using `clearInterval()`, preventing further executions.

15. Predict the output of this code:

```

console.log('Start');
setTimeout(() => {
  console.log('Timeout');
}, 0);
Promise.resolve().then(() => {
  console.log('Promise');
});
console.log('End');

```

Output: `` Start End Promise Timeout ``

Explanation:

First, the synchronous code executes: 'Start' and 'End' are logged.

Then, the event loop processes the microtask queue, which contains the Promise callback, so 'Promise' is logged.

Finally, the macrotask queue (timer queue) is processed, and 'Timeout' is logged even though the timeout is set to 0ms. This demonstrates the priority of execution in the event loop, where microtasks always execute before macrotasks.

16. Predict the output of this code:

```
process.nextTick(() => console.log('Tick'));
Promise.resolve().then(() => console.log('Promise'));
console.log('End');
```

Output: `` End Tick Promise ``

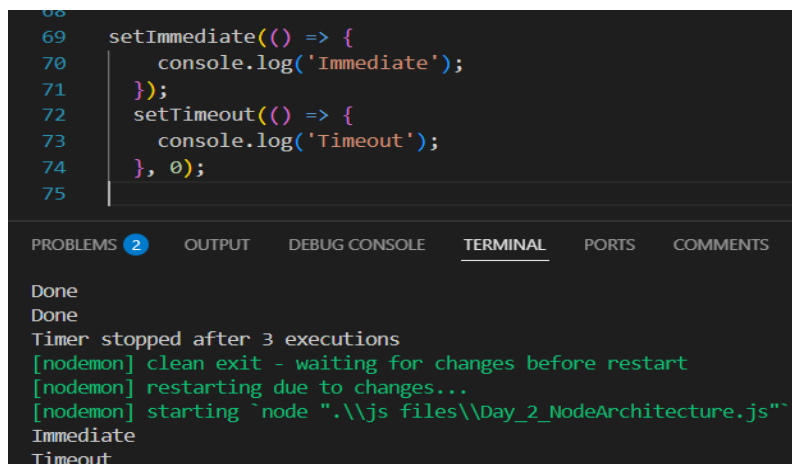
Explanation: First, synchronous code runs, so 'End' is logged.

Then microtasks are processed, process.nextTick callbacks have priority over Promise callbacks in the microtask queue. So 'Tick' is logged before 'Promise'.

This shows the execution order within the microtask queue, where nextTick callbacks are always processed before Promise callbacks. I have understood this topic very well.

17. Predict the output of this code:

```
setImmediate(() => {
  console.log('Immediate');
});
setTimeout(() => {
  console.log('Timeout');
}, 0);
```



```
68
69 setImmediate(() => {
70   console.log('Immediate');
71 });
72 setTimeout(() => {
73   console.log('Timeout');
74 }, 0);
75
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

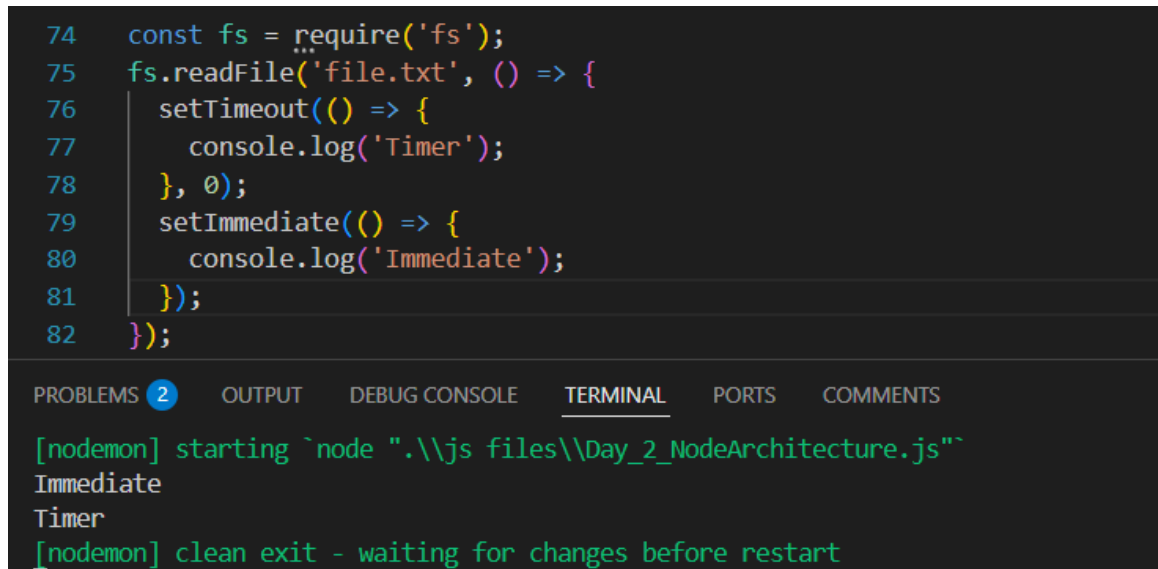
Done
Done
Timer stopped after 3 executions
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`
Immediate
Timeout

Explanation: When running this code directly in the main module, the execution order is non-deterministic. `setTimeout` with 0ms delay and `setImmediate` operate in different phases of the event loop (timers and check phases).

The actual timing depends on system load, process scheduling, and how quickly the event loop cycles through its phases. If this code were run inside an I/O callback, `setImmediate` would consistently execute before `setTimeout`.

18. Predict the output of this code:

```
const fs = require('fs');
fs.readFile('file.txt', () => {
  setTimeout(() => {
    console.log('Timer');
  }, 0);
  setImmediate(() => {
    console.log('Immediate');
  });
});
```



```
74  const fs = require('fs');
75  fs.readFile('file.txt', () => {
76    setTimeout(() => {
77      console.log('Timer');
78    }, 0);
79    setImmediate(() => {
80      console.log('Immediate');
81    });
82  });
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
[nodemon] starting `node ".\js files\Day_2_NodeArchitecture.js"`
Immediate
Timer
[nodemon] clean exit - waiting for changes before restart
```

Explanation: When both `setTimeout(0)` and `setImmediate` are scheduled within an I/O callback (like `fs.readFile`), `setImmediate` will always execute first.

This happens because after the I/O callback completes, the event loop is already in the I/O phase. The next phase is the check phase (where `setImmediate` callbacks run), followed by a return to the timers phase (where `setTimeout` callbacks run) in the next iteration.