Asynchronous Programming:

The JavaScript Concurrency Model

Software Engineering - CSEN 303

A Comprehensive Study Guide

Learning Objectives:

- Understand JavaScript's single-threaded, non-blocking nature
- Differentiate between processes and threads
- Master the Event Loop mechanism
- Understand the Call Stack, Web APIs, and Callback Queues
- Learn execution flow and task prioritization

Based on lectures by Dr. Iman Awaad
German International University (GIU)

Contents

1 The Nature of JavaScript

JavaScript is fundamentally different from many other programming languages. Understanding its core nature is essential before diving into its concurrency model.

JavaScript Core Characteristics

JavaScript is a:

- **High-level** language
- **Asynchronous** language
- **Non-blocking** language
- **Single-threaded** language
- **Interpreted & JIT compiled** language
- **Concurrent** language

1.1 High-Level Language

JavaScript provides significant abstraction from underlying computer hardware:

Level	Description
Machine Language	No abstraction: `01010101000111`
Low-level Language	One level of abstraction: Assembly code
High-level Language	C, Java, JS, Python, LISP, etc.

1.2 Why Single-Threaded Yet Non-Blocking?

This seems contradictory! How can JavaScript:

- Execute only **one piece of code at a time**
- Yet **not freeze** when waiting for long operations?

The Problem

If JavaScript were synchronous and blocking, operations like:

- Fetching data from an API
- Database operations
- File I/O
- User interactions

Would **freeze the entire application** until they complete!

The Solution: Asynchronous

JavaScript achieves concurrency through an **asynchronous, event-driven model** rather than true parallelism. This provides the *illusion of simultaneous execution* while maintaining a single thread of execution.

1.3 Synchronous vs Asynchronous Execution

Synchronous execution:

```
// Each operation must complete before the next begins
console.log("log1");
console.log("log2");
console.log("log3");
// Output: log1, log2, log3 (in order)
```

Asynchronous execution:

```
console.log("log1");
setTimeout(() => {
    console.log("log2");
}, 1000);
console.log("log3");
// Output: log1, log3, log2 (log2 appears after 1 second)
```

Key Insight

In asynchronous execution:

1. Operations are **started** quickly
2. **Responses** arrive later, in any order
3. The main thread continues executing other code

2 The Architecture: Processes vs Threads

Understanding the difference between processes and threads explains why JavaScript is considered "lightweight" but why blocking the main thread is dangerous.

2.1 Process

Process

An **executing program loaded into memory**.

Characteristics:

- Has its own **distinct memory space** and resources
- Operates in **isolation** from other processes
- Considered **"heavyweight"** operations
- Requires more time and resources to create/terminate
- **Fault tolerant**: failure in one process unlikely to affect others

2.2 Thread

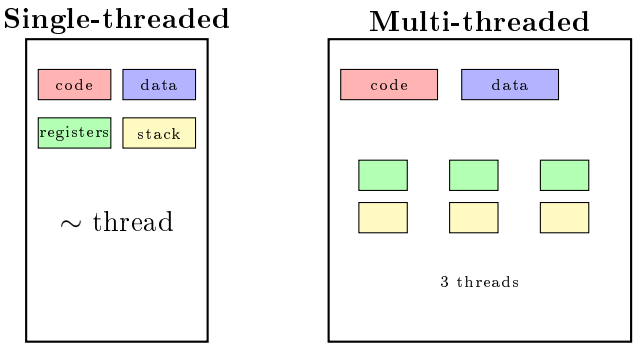
Thread

The **basic unit of execution within a process** (also called "lightweight process").

Characteristics:

- Threads in the same process **share memory** and resources
- Allows for **faster, more efficient communication**
- **Risk**: An error in one thread can crash the entire parent process

2.3 Visual Comparison



JavaScript's Approach

JavaScript executes **one piece of code at a time on a single call stack**. It does everything within a **single process**!
This means:

- Lightweight and efficient
- But blocking the main thread freezes everything
- Solution: The Event Loop mechanism

3 The Mechanism: The Event Loop

The Event Loop is JavaScript's mechanism for handling multiple tasks concurrently despite being single-threaded.

3.1 The Call Stack

Call Stack

A **LIFO (Last-In-First-Out)** data structure that tracks function execution.

3.1.1 Stack Operations

Operation	Description
`push`	Add a new piece of data to the top
`pop`	Remove data from the top
`top/peek`	Look at top data without removing
`isEmpty`	Check if stack is empty

Example:

```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    let result = square(n);
    console.log(result);
}

printSquare(4);
```

Call Stack progression:

1. `printSquare(4)` pushed
2. `square(4)` pushed
3. `multiply(4, 4)` pushed
4. `multiply` returns 16, popped
5. `square` returns 16, popped
6. `console.log(16)` pushed, executed, popped
7. `printSquare` completes, popped

3.2 Web APIs

Web APIs

Interfaces exposed by web browsers that allow JavaScript code to interact with various browser functionalities and system resources. They are **NOT part of the core JavaScript language** but are provided by the browser environment.

Web APIs:

- Guarantee **availability** of operations
- Guarantee **input and output types**

3.2.1 Examples of Web APIs

API	Description
DOM API	Interact with page structure, style, content e.g., `document.getElementById()`, `addEventListener()`
Fetch API	Make network requests (e.g., fetching data from server)
Web Storage APIs	Store data locally (LocalStorage, SessionStorage)
Multimedia APIs	Work with audio, video, media elements
Geolocation API	Request user's geographical location
Notification API	Display system notifications
Clipboard API	Access system clipboard

How Web APIs Enable Async

When a long-running operation (like `setTimeout` or `fetch`) is encountered:

1. It's handed off to the browser's Web APIs
2. JavaScript continues executing other code
3. When the operation completes, its callback goes to a queue

3.3 The Queues

JavaScript has two types of queues for handling asynchronous callbacks:

3.3.1 Callback Queue (Macrotask Queue / Task Queue)

Callback Queue

Holds callbacks from **standard asynchronous operations**.

Sources:

- `setTimeout()`, `setInterval()`
- `setImmediate()` (Node.js)
- I/O operations (network requests, file system)
- User events (mouse clicks, keyboard input)

3.3.2 Microtask Queue

Microtask Queue

Holds callbacks from **higher-priority asynchronous operations**.

Sources:

- Promise callbacks: `.then()`, `.catch()`, `.finally()`
- `MutationObserver` callbacks
- `process.nextTick()` (Node.js)

3.3.3 Key Differences

Aspect	**Callback Queue**	**Microtask Queue**
Priority	Lower	**Higher**
Contents	General async operations	Promise-related operations
Processing	One task per loop iteration	**All tasks** before next macro-task

4 Execution Flow: The Event Loop Algorithm

Event Loop Algorithm

The Event Loop follows this cycle:

1. Execute all synchronous code (until Call Stack is empty)
2. Check the **Microtask Queue**
   - Execute **ALL** microtasks until queue is empty
3. Check the **Callback Queue (Macrotask Queue)**
   - Execute **ONE** macrotask
4. Repeat from step 2

4.1 Priority in Action

```
1  console.log("Start");
2
3  setTimeout(() => {
4      console.log("Timeout callback (Macrotask)");
5  }, 0);
6
7  Promise.resolve().then(() => {
8      console.log("Promise callback (Microtask)");
9  });
10
11 console.log("End");
```

Output:

```
Start
End
Promise callback (Microtask)
Timeout callback (Macrotask)
```

Explanation:

1. `console.log("Start")` - synchronous, executes immediately
2. `setTimeout` - callback sent to Web APIs, then to Callback Queue
3. `Promise.resolve().then()` - callback goes to Microtask Queue
4. `console.log("End")` - synchronous, executes immediately
5. Call Stack empty → process Microtask Queue first
6. Promise callback executes
7. Microtask Queue empty → process one from Callback Queue
8. setTimeout callback executes

Important

Even with `setTimeout(..., 0)`, the callback goes through the Callback Queue and must wait for:

- All synchronous code to complete
- All microtasks to be processed

4.2 Callbacks: The Foundation

Callback

A function passed as an argument to another function, to be executed later when an asynchronous operation completes.

```
function fetchData(callback) {
    // Simulate async operation
    setTimeout(() => {
        const data = "Some fetched data";
        callback(data); // Execute callback with data
    }, 2000);
}

function processData(data) {
    console.log("Processing:", data);
}

fetchData(processData);
console.log("Data fetching initiated...");

// Output:
// Data fetching initiated...
// (2 seconds later)
// Processing: Some fetched data
```

4.3 Promises: Better Async Handling

Promise

An object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

Promises provide a more structured way to handle async code compared to nested callbacks (avoiding "callback hell").

```
const myPromise = new Promise((resolve, reject) => {
    // Async operation
    setTimeout(() => {
        const success = true;
        if (success) {
            resolve("Operation successful!");
        } else {
            reject("Operation failed!");
        }
    }, 1000);
});

myPromise
```

```
    .then(result => console.log(result))
    .catch(error => console.error(error))
    .finally(() => console.log("Cleanup"));
```

4.4 Async/Await: Syntactic Sugar

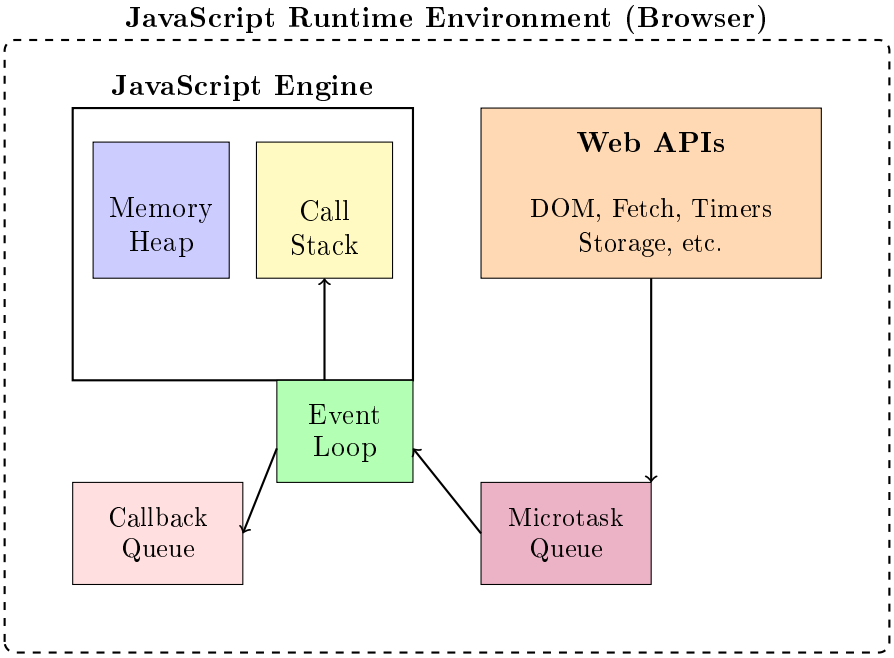
async/await

Syntactic sugar built on top of Promises that makes asynchronous code look and behave more like synchronous code.

```
async function fetchUserData() {
    try {
        const response = await fetch('/api/user');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Error:", error);
    }
}

// async functions always return a Promise
// await pauses execution until the Promise settles
```

5 JavaScript Runtime Environment



5.1 Flow Summary

1. Code enters the **Call Stack** for execution
2. When async operations are encountered, they're sent to **Web APIs**
3. Web APIs handle the operations (timers, network requests, etc.)
4. Once complete, callbacks go to appropriate queue:
   - Promises → Microtask Queue
   - Timers, I/O → Callback Queue
5. **Event Loop** checks if Call Stack is empty
6. If empty: drain Microtask Queue, then one from Callback Queue
7. Repeat

6 Web Workers: True Parallelism

Web Workers

Provide a form of **multi-threading in the browser** by allowing scripts to run in the background in separate threads.

Key Points:

- Run in separate threads (true parallelism)
- Prevent blocking the main UI thread
- Communicate with main thread via message passing
- Cannot directly access the DOM

Concurrency vs Parallelism

- **Concurrency** (async/await/event loops): Managing multiple tasks that can make progress without running simultaneously
- **Parallelism** (threads/CPU cores): Actually running multiple tasks at the same time

JavaScript's Event Loop provides **concurrency**, while Web Workers provide **parallelism**.

7 Additional Implementation Details

7.1 DOM: Document Object Model

The DOM represents HTML/XML in a tree structure that JavaScript can manipulate:

- Methods for selecting HTML elements
- Objects with methods and properties
- Changes to DOM automatically update HTML

Key mechanisms:

- Query selectors
- Event listeners
- Style properties

7.2 jQuery

A JavaScript library that simplifies:

- DOM manipulation across browsers
- AJAX requests
- UI enhancements (date pickers, sliders, etc.)
- Dynamic CSS modification

jQuery code starts with $:

```
$('#myElement').click(function() {
    $(this).hide();
});
```

7.3 AJAX

Asynchronous JavaScript and XML - technique for exchanging data with a server without reloading the page.

- Uses XMLHttpRequest object
- Can request text, JSON, HTML, etc.
- jQuery simplifies AJAX requests

7.4 Node.js

JavaScript runtime environment for the server side:

- Node.js + HTTP = Express framework
- NPM (Node Package Manager) for package management
- Same Event Loop model as browser JavaScript

8 Summary

Key Takeaways

1. **JavaScript is single-threaded but non-blocking** through its asynchronous, event-driven model
2. **The Event Loop** is the core mechanism enabling concurrency:
   - Monitors Call Stack and Queues
   - Prioritizes Microtasks over Macrotasks
3. **Call Stack** executes synchronous code (LIFO)
4. **Web APIs** handle long-running operations off the main thread
5. **Two Queues**:
   - Microtask Queue (Promises) - higher priority
   - Callback Queue (setTimeout, I/O) - lower priority
6. **Execution order**: Sync → All Microtasks → One Macrotask → Repeat
7. **Callbacks → Promises → async/await**: Evolution of async patterns
8. **Web Workers** provide true parallelism when needed

8.1 Useful Tools

- **Loupe**: `https://github.com/latentflip/loupe`
  - Visualizes Call Stack, Event Loop, Callback Queue interactions
- **JS Visualizer 9000**: `https://www.jsv9000.app/`
  - Interactive Event Loop visualization

"I mark the hours everyone
Nor have I yet outrun the sun
My use and value unto you
Are gauged by what you have to do."

- Inscription on Time Turner