asynchronous execution using fs (promise --> asynx and await)

- async and await both are keywords which are used together.
- async is used in function decalaration.
- await is used inside async function. (It stops the execution till get the data)
- async function always return a promise
- await keyword suspend the function execution in call stack till the promises resolves.

Example :-

```
async function greet() {
  return "HELLO";
}

let data = greet();
console.log(data);
```

streaming

- sending a data continuously from sourse to destination in chunks is said to be sreaming.

Buffer

- It is an array like object which holds binary data, which is used to store data in memory (RAM).
- Buffers are used to store data in chunks.
- Buffer size can not be controlled or set by user.
- Size of buffer cannot be modified also throughout the operation.
- Once the operation is done, buffer is destroyed.

Complete Node.js File System (fs) Module Notes

Table of Contents

Introduction to fs Module {#introduction}

The File System (fs) module in Node.js provides an API for interacting with the file system. It allows you to:

- Create, read, update, and delete files
- Create, rename, and delete directories
- Work with file permissions and metadata
- Stream large files efficiently

```
import fs from "node:fs"; // Callback-based
import fsSync from "node:fs"; // Synchronous
import fsPromise from "node:fs/promises"; // Promise-based
```

Synchronous vs Asynchronous Operations {#sync-vs-async}

Synchronous Operations

- Block the main thread until operation completes
- Use `Sync` suffix in method names
- Return values directly or throw errors

```
import fs from "node:fs";

try {
  let data = fs.readFileSync("./file.txt", "utf-8");
  console.log(data);
} catch (error) {
  console.error("Error:", error.message);
}
```

Asynchronous Operations (Callback-based)

- Non-blocking operations
- Use callbacks for handling results/errors

```
import fs from "node:fs";

fs.readFile("./file.txt", "utf-8", (err, data) => {
  if (err) {
    console.error("Error:", err.message);
    return;
  }
  console.log(data);
});
```

Promise-based Operations {#promises}

Using `fs/promises` for cleaner asynchronous code:

```
import fsPromise from "node:fs/promises";

// Using .then() and .catch()
fsPromise
  .readFile("./file.txt", "utf-8")
  .then((data) => {
    console.log("File content:", data);
  })
  .catch((err) => {
    console.error("Error reading file:", err.message);
  });

// Deleting a file
fsPromise
  .unlink("./file.txt")
  .then(() => {
    console.log("File deleted successfully");
  })
  .catch((err) => {
    console.error("Error deleting file:", err.message);
  });
```

Async/Await Implementation {#async-await}

Key Concepts

- **async**: Keyword used in function declaration
- **await**: Keyword used inside async function to wait for promises
- **async function**: Always returns a promise
- **await**: Suspends function execution until the promise resolves

```
import fsPromise from "node:fs/promises";

async function greet() {
  return "HELLO";
}

let data = greet();
console.log(data); // Returns: Promise { 'HELLO' }

// File Operations with async/await
async function fileOperations() {
  try {
    // Create file
    await fsPromise.writeFile("./demo.txt", "Hello World!");
    console.log("File created");

    // Read file
    let content = await fsPromise.readFile("./demo.txt", "utf-8");
    console.log("File content:", content);

    // Update file
    await fsPromise.appendFile("./demo.txt", "\nNew line added");
    console.log("File updated");
  } catch (error) {
    console.error("Error:", error.message);
  }
}

fileOperations();
```

File Operations {#file-operations}

1. Creating Files

```
// writeFile() - Creates new file or overwrites existing
async function createFile() {
  await fsPromise.writeFile("./demo.txt", "Initial content");
  console.log("File created");
}

// appendFile() - Adds content to existing file
async function appendToFile() {
  await fsPromise.appendFile("./demo.txt", "\nAppended content");
  console.log("Content appended");
}
```

2. Reading Files

```
async function readFile() {
  try {
    let data = await fsPromise.readFile("./demo.txt", "utf-8");
    console.log("File content:", data);
  } catch (error) {
    console.error("File not found or cannot be read");
  }
}
```

3. Updating Files

```
async function updateFile() {
  try {
    // Read current content
    let currentContent = await fsPromise.readFile("./demo.txt", "utf-8");

    // Modify content
    let updatedContent = currentContent + "\nUpdated content";

    // Write back to file
    await fsPromise.writeFile("./demo.txt", updatedContent);
    console.log("File updated");
  } catch (error) {
    console.error("Error updating file:", error.message);
  }
}
```

4. Deleting Files

```
async function deleteFile() {
  try {
    await fsPromise.unlink("./demo.txt");
    console.log("File deleted");
  } catch (error) {
    console.error("Error deleting file:", error.message);
  }
}
```

5. Renaming Files

```
async function renameFile() {
  try {
    await fsPromise.rename("./oldname.txt", "./newname.txt");
    console.log("File renamed");
  } catch (error) {
    console.error("Error renaming file:", error.message);
  }
}
```

6. File Stats and Information

```
async function getFileStats() {
  try {
    let stats = await fsPromise.stat("./demo.txt");
    console.log("File size:", stats.size, "bytes");
    console.log("Is file:", stats.isFile());
    console.log("Is directory:", stats.isDirectory());
    console.log("Created:", stats.birthtime);
    console.log("Modified:", stats.mtime);
  } catch (error) {
    console.error("Error getting file stats:", error.message);
  }
}
```

Directory Operations {#directory-operations}

1. Creating Directories

```
async function createDirectory() {
  try {
    // Create single directory
    await fsPromise.mkdir("./newFolder");

    // Create nested directories
    await fsPromise.mkdir("./Project/backend/controllers", { recursive: true });
    console.log("Directories created");
  } catch (error) {
    console.error("Error creating directory:", error.message);
  }
}
```

2. Reading Directory Contents

```
async function readDirectory() {
  try {
    let files = await fsPromise.readdir("./");
    console.log("Directory contents:", files);

    // Get detailed information
    let filesWithStats = await Promise.all(
      files.map(async (file) => {
        let stats = await fsPromise.stat(file);
        return {
          name: file,
          isDirectory: stats.isDirectory(),
          size: stats.size,
        };
      })
    );
    console.log("Detailed info:", filesWithStats);
  } catch (error) {
    console.error("Error reading directory:", error.message);
  }
}
```

3. Deleting Directories

```javascript
async function deleteDirectory() {
  try {
    // Delete empty directory
    await fsPromise.rmdir("./emptyFolder");

    // Delete directory with contents (Node.js 14+)
    await fsPromise.rm("./folderWithContents", {
      recursive: true,
      force: true,
    });
    console.log("Directory deleted");
  } catch (error) {
    console.error("Error deleting directory:", error.message);
  }
}
```

4. Project Structure Creation

```javascript
async function createProjectStructure() {
  try {
    // Create project structure
    await fsPromise.mkdir("./Project/backend/controllers", { recursive: true });
    await fsPromise.mkdir("./Project/backend/models", { recursive: true });
    await fsPromise.mkdir("./Project/frontend/components", { recursive: true });

    // Create files
    await fsPromise.writeFile(
      "./Project/backend/app.js",
      `const express = require('express');\nconst app = express();\n\nmodule.exports = app;`
    );
    await fsPromise.writeFile(
      "./Project/backend/package.json",
      `{\n  "name": "backend",\n  "version": "1.0.0"\n}`
    );

    console.log("Project structure created");
  } catch (error) {
    console.error("Error creating project structure:", error.message);
  }
}
```

Streaming {#streaming}

Concept

Streaming is the process of sending data continuously from source to destination in chunks, rather than loading the entire file into memory at once.

Benefits of Streaming

- Memory efficient for large files

- Faster processing
- Real-time data processing
- Better user experience

Read Streams

```
import fs from "node:fs";

// Create read stream
const readStream = fs.createReadStream("./largefile.txt", {
  encoding: "utf-8",
  highWaterMark: 1024, // Buffer size in bytes
});

readStream.on("data", (chunk) => {
  console.log("Received chunk:", chunk.length, "bytes");
});

readStream.on("end", () => {
  console.log("File reading completed");
});

readStream.on("error", (error) => {
  console.error("Error reading file:", error.message);
});
```

Write Streams

```
const writeStream = fs.createWriteStream("./output.txt");

writeStream.write("First chunk of data\n");
writeStream.write("Second chunk of data\n");
writeStream.end("Final chunk of data\n");

writeStream.on("finish", () => {
  console.log("File writing completed");
});
```

Pipe Operations

```
// Copy file using streams
const readStream = fs.createReadStream("./input.txt");
const writeStream = fs.createWriteStream("./output.txt");

readStream.pipe(writeStream);

writeStream.on("finish", () => {
  console.log("File copied successfully");
});
```

Buffer Concept {#buffer}

What is Buffer?

- Array-like object that holds binary data
- Used to store data in memory (RAM)
- Represents raw memory allocation outside V8 heap
- Fixed-size sequence of bytes

Key Characteristics

- Buffer size cannot be controlled by user during creation
- Size cannot be modified after creation
- Automatically destroyed after operation completes
- More efficient for binary data operations

Buffer Examples

```
// Creating buffers
const buf1 = Buffer.from("Hello World", "utf-8");
const buf2 = Buffer.alloc(10); // Creates 10-byte buffer filled with zeros
const buf3 = Buffer.allocUnsafe(10); // Faster but contains arbitrary data

console.log(buf1); // <Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64>
console.log(buf1.toString()); // Hello World
console.log(buf1.length); // 11

// Buffer operations
buf2.write("Hello");
console.log(buf2.toString()); // Hello\u0000\u0000\u0000\u0000\u0000

// Working with streams and buffers
const readStream = fs.createReadStream("./file.txt");
readStream.on("data", (chunk) => {
  console.log("Buffer chunk:", chunk);
  console.log("String representation:", chunk.toString());
});
```

Error Handling {#error-handling}

Common Error Types

- **ENOENT**: File or directory not found
- **EACCES**: Permission denied
- **EISDIR**: Expected file but found directory
- **ENOTDIR**: Expected directory but found file

Error Handling Patterns

```
async function robustFileOperation() {
  try {
    // Check if file exists before reading
    await fsPromise.access("./file.txt");
    let data = await fsPromise.readFile("./file.txt", "utf-8");
    console.log(data);
  } catch (error) {
    switch (error.code) {
      case "ENOENT":
        console.error("File does not exist");
        break;
      case "EACCES":
        console.error("Permission denied");
        break;
      default:
        console.error("Unexpected error:", error.message);
    }
  }
}

// Utility function to check file existence
async function fileExists(filePath) {
  try {
    await fsPromise.access(filePath);
    return true;
  } catch {
    return false;
  }
}
```

Best Practices {#best-practices}

1. Always Handle Errors

```
async function safeFileOperation() {
  try {
    await fsPromise.readFile("./file.txt", "utf-8");
  } catch (error) {
    console.error("Operation failed:", error.message);
    // Implement fallback logic
  }
}
```

2. Use Appropriate Method for Task

- Use **streams** for large files
- Use **async/await** for better readability
- Use **synchronous methods** only when necessary

3. Clean Up Resources

```
async function processLargeFile() {
  const stream = fs.createReadStream("./largefile.txt");

  try {
    // Process stream
  } catch (error) {
    console.error(error);
  } finally {
    stream.destroy(); // Clean up
  }
}
```

4. Use Path Module for File Paths

```
import path from "node:path";

const filePath = path.join(__dirname, "data", "file.txt");
const absolutePath = path.resolve("./relative/path/file.txt");
```

5. Validate File Operations

```
async function safeFileCreation(fileName, content) {
  // Validate input
  if (!fileName || !content) {
    throw new Error("Filename and content are required");
  }

  // Check if file already exists
  if (await fileExists(fileName)) {
    throw new Error("File already exists");
  }

  // Create file
  await fsPromise.writeFile(fileName, content);
}
```

Global Object in Node.js

The global object in Node.js is `global` (not `window` like in browsers).

```
console.log(global); // Global object
console.log(global.process); // Process object
console.log(global.Buffer); // Buffer constructor
```

Summary

The fs module is essential for file system operations in Node.js. Key takeaways:

- Use **promise-based methods** with **async/await** for clean, readable code
- Handle errors appropriately for robust applications
- Use **streams** for large file operations to optimize memory usage
- Understand **buffers** for efficient binary data handling
- Follow best practices for maintainable and reliable code

Remember: Always consider the size of files you're working with and choose the appropriate method (direct read/write vs streaming) accordingly.