Great, let's break down **14. Worker Threads & Child Processes** in Node.js 🚀

14. Worker Threads & Child Processes

📌 Why do we need them?

- Node.js is **single-threaded** → great for **I/O-bound tasks** (network requests, DB calls).
- But for **CPU-bound tasks** (e.g., encryption, image processing, data parsing) → the **event loop blocks**, slowing everything.
- To avoid blocking:
  - Use **Worker Threads** → parallel threads inside one process.
  - Use **Child Processes** → spawn separate Node.js processes.

1. Worker Threads

☑ What are Worker Threads?

- Introduced in **Node.js v10.5+ (stable in v12)**.
- Allow running **JavaScript in multiple threads** (inside the same process).
- Workers share **memory** via `SharedArrayBuffer`.

Example: Worker Thread

🗁 worker.js

```
const { parentPort } = require("worker_threads");

let sum = 0;
for (let i = 0; i < 1e9; i++) {
  sum += i;
}

parentPort.postMessage(sum); // send result to main thread
```

🗁 main.js

```
const { Worker } = require("worker_threads");

console.log("Main thread started");

const worker = new Worker("./worker.js");
```

```
worker.on("message", (result) => {
  console.log("Sum:", result);
});

worker.on("error", (err) => console.error(err));
worker.on("exit", (code) => console.log("Worker exited with code", code));
```

👉 Heavy computation runs **in worker thread**, not blocking the event loop.

---

2. Child Processes

☑ What are Child Processes?

- **Separate Node.js processes** created from the main process.
- Don't share memory directly (unlike workers).
- Communicate via **IPC (Inter-Process Communication)** → `stdin`, `stdout`, `stderr`, or `send()`.

Methods:

- `spawn()` → launch new process (stream-based, best for large output).
- `exec()` → launch new process (buffer-based, best for small output).
- `fork()` → special case of `spawn()` for Node.js scripts with IPC channel.

Example: `spawn()`

```
const { spawn } = require("child_process");

const ls = spawn("ls", ["-lh", "/usr"]);

ls.stdout.on("data", (data) => {
  console.log(`Output: ${data}`);
});

ls.stderr.on("data", (data) => {
  console.error(`Error: ${data}`);
});

ls.on("close", (code) => {
  console.log(`Child process exited with code ${code}`);
});
```

Example: `exec()`

```
const { exec } = require("child_process");

exec("ls -lh /usr", (error, stdout, stderr) => {
  if (error) {
    console.error(`Error: ${error.message}`);
    return;
  }
  if (stderr) {
    console.error(`Stderr: ${stderr}`);
    return;
  }
  console.log(`Stdout: ${stdout}`);
});
```

Example: `fork()`

child.js

```
process.on("message", (msg) => {
  console.log("Message from parent:", msg);
  process.send({ result: msg.num * 2 }); // send result back
});
```

main.js

```
const { fork } = require("child_process");

const child = fork("child.js");

child.send({ num: 21 }); // send data to child

child.on("message", (msg) => {
  console.log("Result from child:", msg.result);
});
```

---

3. Worker Threads vs Child Processes

Feature	Worker Threads	Child Processes
Context	Same process	Separate process
Memory Sharing	Yes (via SharedArrayBuffer)	No (separate memory)

Feature	Worker Threads	Child Processes
Communication	`postMessage`	IPC (`send()`, streams)
Use Case	CPU-bound tasks (math, ML, crypto)	Running external programs, isolating crashes
Overhead	Low	Higher (separate OS process)

4. Interview Questions

◇ Q1: When would you use Worker Threads vs Child Processes?
👉 Use **Worker Threads** for CPU-heavy tasks needing shared memory.
👉 Use **Child Processes** for running external commands or isolating failures.

◇ Q2: What's the difference between `spawn()`, `exec()`, and `fork()`?
👉 `spawn()` → stream output (large data).
👉 `exec()` → buffer output (small data).
👉 `fork()` → optimized for Node.js scripts with IPC.

◇ Q3: Why doesn't Node.js use multiple threads by default?
👉 Node.js relies on **event loop** for async I/O, which is efficient without threads. Threads only needed for **CPU-heavy tasks**.

☑ **Key Takeaway:**

- **Worker Threads** → run tasks in parallel inside same process (good for CPU-heavy tasks).
- **Child Processes** → run separate Node.js or system programs (good for isolation, large workloads).
- Choose based on **performance vs isolation needs**.

Do you want me to continue with **15. Node.js Security Best Practices** next? That's super important for production-level apps 🔒.