

 **Node.js is single-threaded, but it can run tasks in parallel using Child Processes!**



index.js

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

const child = spawn('ls', ['-l']);

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

#Javascript #Programming #Tips



ViniciussMelo

# 💡 What Are Child Processes?

Child Processes in Node.js allow us to run separate processes (sub-processes) independently from the main process. This is useful for parallel execution of CPU-intensive tasks.

Node.js is single-threaded, meaning it can only process one operation at a time in the main thread. However, with Child Processes, we can run multiple tasks in parallel, each in its own process.



# Why Do Child Processes Exist?

- ✓ Node.js is single-threaded, which means that heavy CPU operations (e.g., file compression, cryptography, or large calculations) can block the Event Loop and slow everything down.
- ✓ Child Processes allow true parallel execution by running separate Node.js instances outside the main thread.
- ✓ Unlike Worker Threads, which share memory, Child Processes have their own memory space, making them useful for running completely independent applications.



# **When Should You Use Child Processes?**

Best for CPU-heavy tasks like:

- ✓ Large data processing (e.g., reading/writing large files);
- ✓ Compression & Encryption (e.g., zipping files, hashing passwords);
- ✓ Running Shell Commands (e.g., executing system commands);
- ✓ Multiple Independent Processes (e.g., running a separate service or script).



## ✓ **How to Use Child Processes?**

Node.js provides a built-in `child_process` module to create and manage child processes. There are four ways to use it:

- ✓ `exec()` → Run shell commands.
- ✓ `spawn()` → Stream real-time output.
- ✓ `fork()` → Run separate Node.js scripts.
- ✓ `execFile()` → Execute external programs.



# 1 Running a Command Using exec

The exec function is used to run shell commands in a new process.

✓ Use Case: Running shell commands like ls, mkdir, or rm.

⚠ Warning: exec has a buffer size limit, so it's not ideal for large outputs.

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

exec('ls -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`Error: ${error.message}`);
    return;
  }
  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
});
```



## 2 Running a Command Using spawn (Streaming Output)

The spawn function is similar to exec, but instead of returning all data at once, it streams output in real time.

✅ Use Case: Handling large data streams (e.g., real-time logging).

⚡ Difference from exec? spawn is better for continuous data streams and has no buffer limit.

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

const child = spawn('ls', ['-l']);

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

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

child.on('close', (code) => {
  console.log(`Process exited with code ${code}`);
});
```



### 3 Creating a Separate Node.js Script Using fork

The fork function is specifically designed to run another Node.js script as a separate process.

- ✓ Use Case: Running another Node.js script as a separate process and communicating via messages.
- ◆ Forked processes have a separate memory space, so they don't share variables with the main process.

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

// child.js is a separate file
const child = fork('./child.js'); // Start the child process

// Send a message to the child
child.send('Hello from main process');

child.on('message', (message) => {
  console.log(`Main received message: ${message}`);
});
```






## 4 Running a Background Task Using execFile

If you want to run an external program or script, `execFile` is the best choice.

✓ Use Case: Running external executables like Python scripts or system binaries.



```
const { execFile } = require('child_process');

execFile('node', ['-v'], (error, stdout, stderr) => {
  if (error) {
    console.error(`Error: ${error.message}`);
    return;
  }
  console.log(`Node.js version: ${stdout}`);
});
```



## Conclusion

Child Processes allow Node.js to handle CPU-heavy tasks in parallel by creating separate processes. This prevents the main thread from being blocked, making applications faster and more responsive.

Have you ever used Child Processes in Node.js? What's your favorite way to handle CPU-heavy operations? Let's discuss in the comments! ↓

