**Lesson 8: Cluster and Child Processes**

[Node.js](https://nodejs.org/) is becoming more and more popular as a server-side run-time environment, especially for high traffic websites, as [statistics show](http://w3techs.com/technologies/details/ws-nodejs/all/all). Also, the availability of several frameworks make it a good environment for rapid prototyping. Node.js has an event-driven architecture, leveraging a non-blocking I/O API that allows requests being processed asynchronously.

One of the important and often less highlighted features of Node.js is its scalability. In fact, this is the main reason why [some large companies with heavy traffic are integrating Node.js in their platform](https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node) (e.g., Microsoft, Yahoo, Uber, and Walmart) or even completely moving their server-side operations to Node.js (e.g., PayPal, eBay, and Groupon).

Each Node.js process runs in a single thread and by default it has a memory limit of 512MB on 32-bit systems and 1GB on 64-bit systems. Although [the memory limit can be bumped to ~1GB on 32-bit systems and ~1.7GB on 64-bit systems](https://github.com/joyent/node/wiki/FAQ#what-is-the-memory-limit-on-a-node-process), both memory and processing power can still become bottlenecks for various processes.

The elegant solution Node.js provides for scaling up the applications is to split a single process into multiple processes or *workers*, in Node.js terminology. This can be achieved through a cluster module. The cluster module allows you to create child processes (workers), which share all the server ports with the main Node process (master).

**How to create a Node.js cluster for speeding up your applications?**

**Node.js Cluster Module: what it is and how it works**

A cluster is a pool of similar workers running under a parent Node process. Workers are spawned using the fork() method of the child\_processes module. This means workers can share server handles and use IPC (Inter-process communication) to communicate with the parent Node process.

The master process is in charge of initiating workers and controlling them. You can create an arbitrary number of workers in your master process. Moreover, remember that by default incoming connections are distributed in a round-robin approach among workers (except in Windows). Actually there is another approach to distribute incoming connections, that I won’t discuss here, which hands the assignment over to the OS (default in Windows). [Node.js documentation](https://nodejs.org/api/cluster.html#cluster_cluster_schedulingpolicy) suggests using the default round-robin style as the scheduling policy.

Although using a cluster module sounds complex in theory, it is very straightforward to implement. To start using it, you have to include it in your Node.js application:

Var cluster = require(‘cluster’);

A cluster module executes the same Node.js process multiple times. Therefore, the first thing you need to do is to identify what portion of the code is for the master process and what portion is for the workers. The cluster module allows you to identify the master process as follows:

if(cluster.isMaster) { ... }

The master process is the process you initiate, which in turn initialize the workers. To start a worker process inside a master process, we’ll use the fork() method:

cluster.fork();

This method returns a worker object that contains some methods and properties about the forked worker. We’ll see some examples in the following section.

A cluster module contains several events. Two common events related to the moments of start and termination of workers are the online and the exit events. online is emitted when the worker is forked and sends the online message. exit is emitted when a worker process dies.

**Examples**

How a Cluster Module is Used in a Node.js App

A simple server is set up that responds to all incoming requests with a message containing the worker process ID that processed the request. The master process forks four workers. In each of them, we start listening the port 8000 for incoming requests.

var cluster = require('cluster');

var http = require('http');

if(cluster.isMaster) {

    var numWorkers = require('os').cpus().length;

    console.log('Master cluster setting up ' + numWorkers + ' workers...');

    for(var i = 0; i < numWorkers; i++) {

        cluster.fork();

    }

    cluster.on('online', function(worker) {

        console.log('Worker ' + worker.process.pid + ' is online');

    });

    cluster.on('exit', function(worker, code, signal) {

        console.log('Worker ' + worker.process.pid + ' died with code: ' + code + ', and signal: ' + signal);

        console.log('Starting a new worker');

        cluster.fork();

    });

} else {

    http.createServer(function(req, res) {

        res.writeHead(200);

        res.end('process ' + process.pid + ' says hello!');

        console.log('process '+ process.pid + 'handled this request');

    }).listen(8000);

}

Test this server on by accessing the URL <http://127.0.0.1:8000/>. When requests are received, they are distributed one at a time to each worker. If a worker is available, it immediately starts processing the request; otherwise it’ll be added to a queue.

There are a few points that are not very efficient in the above example. For instance, imagine if a worker dies for some reason. In this case, you lose one of your workers and if the same happens again, you will end up with a master process with no workers to handle incoming requests. Another issue is related to the number of workers. There are different number of cores/threads in the systems that you deploy your application to. In the mentioned example, to use all of the system’s resources, you have to manually check the specifications of each deployment server, find how many threads there are available, and update it.

Node.js applications can be parallelized using cluster modules in order to use the system more efficiently. Running multiple processes at the same time can be done using few lines of code and this makes the migration relatively easy, as Node.js handles the hard part.

Worker threads and child processes, while accomplishing the same goal(parallelism), are different. Worker threads share memory whereas processes do not. worker threads are not part of the cluster module. When you create a cluster, you are creating multiple node instances, each one being a new process. You are NOT creating threads. Process != Thread. Basically, parallelism in nodejs boils down to two approaches: 1. Process based (via clusters) 2. Thread based (via worker threads).

**Child\_process Creation Methods:**

It also facilitates creation of child processes to leverage parallel processing on multi-core CPU based systems.

Child processes always have three streams **child.stdin**, **child.stdout**, and **child.stderr** which may be shared with the stdio streams of the parent process.

Node provides **child\_process** module which has the following three major ways to create a child process.

* **exec** − child\_process.exec method runs a command in a shell/console and buffers the output.
* **spawn** − child\_process.spawn launches a new process with a given command.
* **fork** − The child\_process.fork method is a special case of the spawn() to create child processes.

## The exec() method

child\_process.exec method runs a command in a shell and buffers the output. It has the following signature −

child\_process.exec(command[, options], callback)

### Parameters

Here is the description of the parameters used −

* **command** (String) The command to run, with space-separated arguments
* **options** (Object) may comprise one or more of the following options −
  + **cwd** (String) Current working directory of the child process
  + **env** (Object) Environment key-value pairs
  + **encoding** (String) (Default: 'utf8')
  + **shell** (String) Shell to execute the command with (Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows, The shell should understand the -c switch on UNIX or /s /c on Windows. On Windows, command line parsing should be compatible with cmd.exe.)
  + **timeout** (Number) (Default: 0)
  + **maxBuffer** (Number) (Default: 200\*1024)
  + **killSignal** (String) (Default: 'SIGTERM')
  + **uid** (Number) Sets the user identity of the process.
  + **gid** (Number) Sets the group identity of the process.
* **callback** The function gets three arguments **error**, **stdout,** and **stderr** which are called with the output when the process terminates.

The exec() method returns a buffer with a max size and waits for the process to end and tries to return all the buffered data at once.

## The spawn() Method

child\_process.spawn method launches a new process with a given command. It has the following signature −

child\_process.spawn(command[, args][, options])

### Parameters

Here is the description of the parameters used −

* **command** (String) The command to run
* **args** (Array) List of string arguments
* **options** (Object) may comprise one or more of the following options −
  + **cwd** (String) Current working directory of the child process.
  + **env** (Object) Environment key-value pairs.
  + **stdio** (Array) String Child's stdio configuration.
  + **customFds** (Array) Deprecated File descriptors for the child to use for stdio.
  + **detached** (Boolean) The child will be a process group leader.
  + **uid** (Number) Sets the user identity of the process.
  + **gid** (Number) Sets the group identity of the process.

The spawn() method returns streams (stdout &stderr) and it should be used when the process returns a volume amount of data. spawn() starts receiving the response as soon as the process starts executing.

## The fork() Method

child\_process.fork method is a special case of spawn() to create Node processes. It has the following signature −

child\_process.fork(modulePath[, args][, options])

### Parameters

Here is the description of the parameters used −

* **modulePath** (String) The module to run in the child.
* **args** (Array) List of string arguments
* **options** (Object) may comprise one or more of the following options −
  + **cwd** (String) Current working directory of the child process.
  + **env** (Object) Environment key-value pairs.
  + **execPath** (String) Executable used to create the child process.
  + **execArgv** (Array) List of string arguments passed to the executable (Default: process.execArgv).
  + **silent** (Boolean) If true, stdin, stdout, and stderr of the child will be piped to the parent, otherwise they will be inherited from the parent, see the "pipe" and "inherit" options for spawn()'s stdio for more details (default is false).
  + **uid** (Number) Sets the user identity of the process.
  + **gid** (Number) Sets the group identity of the process.

The fork method returns an object with a built-in communication channel in addition to having all the methods in a normal ChildProcess instance.

**Class Demonstration Code:**

/////////////////////////////////////////////////

//cluster

////////////////////////////////////////////////

var cluster = require('cluster');

var http = require('http');

if(cluster.isMaster) {

    var numWorkers = require('os').cpus().length;

    console.log('Master cluster setting up ' + numWorkers + ' workers...');

    for(var i = 0; i < numWorkers; i++) {

        cluster.fork();

    }

    cluster.on('online', function(worker) {

        console.log('Worker ' + worker.process.pid + ' is online');

    });

    cluster.on('exit', function(worker, code, signal) {

        console.log('Worker ' + worker.process.pid + ' died with code: ' + code + ', and signal: ' + signal);

        console.log('Starting a new worker');

        cluster.fork();

    });

} else {

    http.createServer(function(req, res) {

        res.writeHead(200);

        res.end('process ' + process.pid + ' says hello!');

        console.log('process '+ process.pid + 'handled this request');

    }).listen(8000);

}

////////////////////////////////////////////////////////////////////

//fork() child.js

///////////////////////////////////////////////////////////////////

process.on('message', (msg) => {

    console.log('Message from parent:', msg);

  });

  let counter = 0;

  setInterval(() => {

    process.send({ counter: counter++ });

  }, 1000);

/////////////////////////////////////////////////////////////////

//fork() parent.js

////////////////////////////////////////////////////////////////

const { fork } = require('child\_process');

const forked = fork('child.js');

forked.on('message', (msg) => {

  console.log('Message from child', msg);

});

forked.send({ hello: 'world' });

//////////////////////////////////////////////////////////////

//exec()

/////////////////////////////////////////////////////////////

const fs = require('fs');

const child\_process = require('child\_process');

var workerProcess = child\_process.exec('node eventemitter.js ',function

      (error, stdout, stderr)

      {

        console.log('worker thread created');

         console.log('stdout: ' + stdout);

        console.log('stderr: ' + stderr);

      });