BCDV 1022

Node Scalability & Cluster

2023 Fall

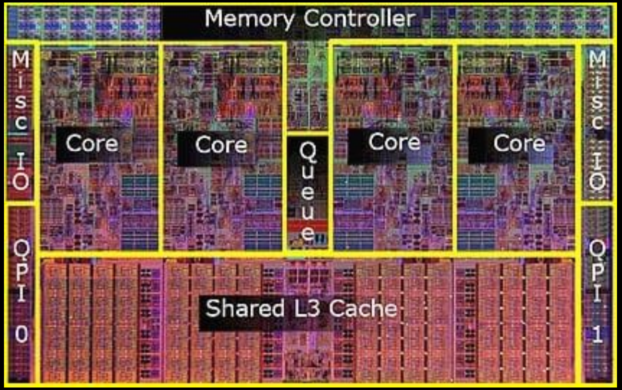
week 03 - class 07

Topics

- Scalability & Child Processes
- Clusters

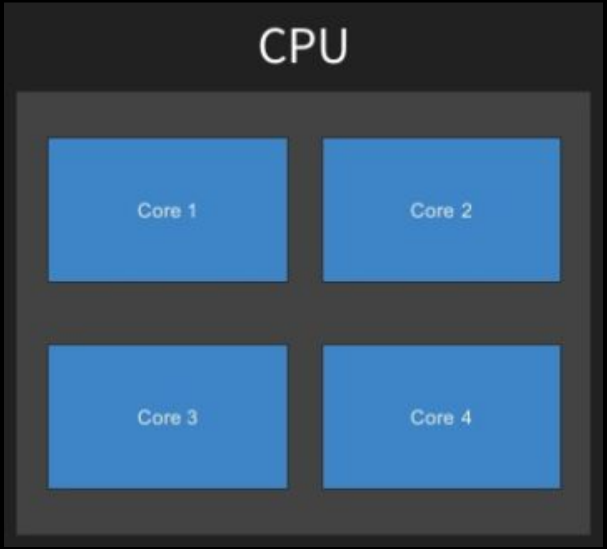
Scalability

CPU Cores



- Our typical OS has different processes running in the background
- Each process is being managed by a single-core of our CPU
- In order to take full advantage of our CPU, we would need a number of processes that equals the number of cores in our CPU
- Previously, we have been limiting ourselves to a single Node process and single CPU core.

Single Process Limitations

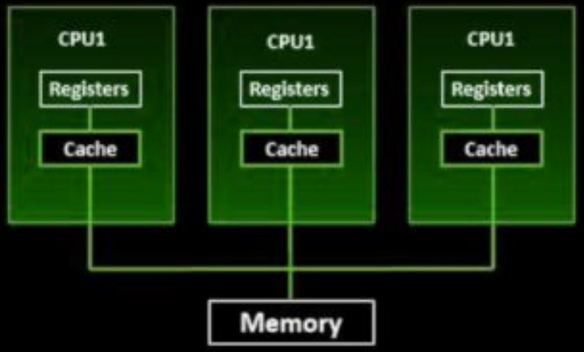


- Single-threaded, non-blocking performance in Node works great for a single process.
- Eventually, one process in one CPU is not going to be enough to handle the increased workload of your application
- Node.js runs in a single thread, but it doesn't mean we can’t have multiple processes and multiple machines.

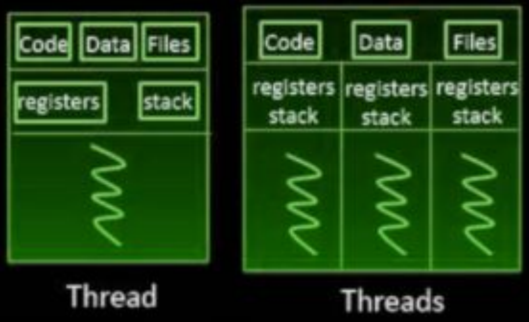
Multiprocessing vs Multithreading

- Multiprocessing is adding more number of or CPUs/processors to the system which increases the computing speed of the system.

- Multithreading is allowing a single process to create more threads which increase the responsiveness of the system.

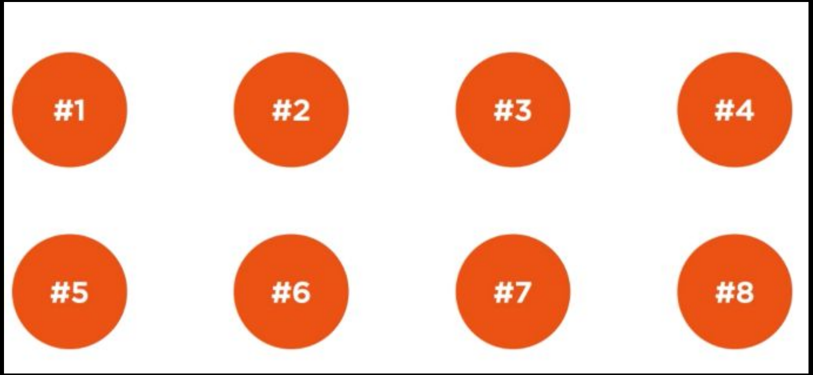


Multiprocessing



Multithreading

Node Scalability



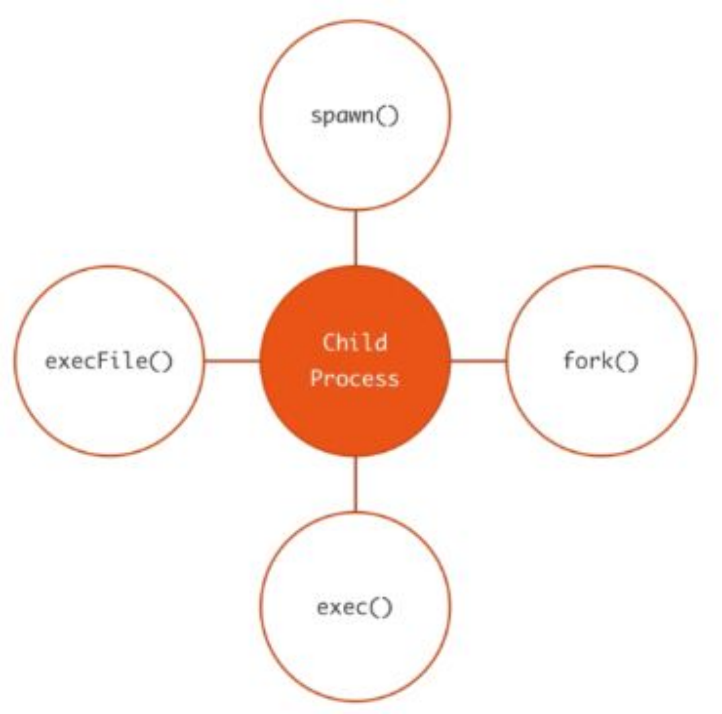
- Scalability is built into the core of the Node runtime.

- Node is named *Node* to emphasize the idea that a Node application should comprise multiple distributed nodes that communicate with each other.

- Node has built-in module to help with:
  - Running multiple nodes for your Node application
  - Running a Node process on every CPU core of your server
  - Load balancing all requests on all distributed servers

Child Processes

Child Process



- The child_process module provides the ability to spawn child processes

- The child_process module enables us to access Operating System functionalities by running any system command inside a child process

- We can control that child process input stream, and listen to its output stream.

- There are four different ways to create a child process in Node: spawn(), fork(), exec() and execFile()

Child Process - Events and Streams

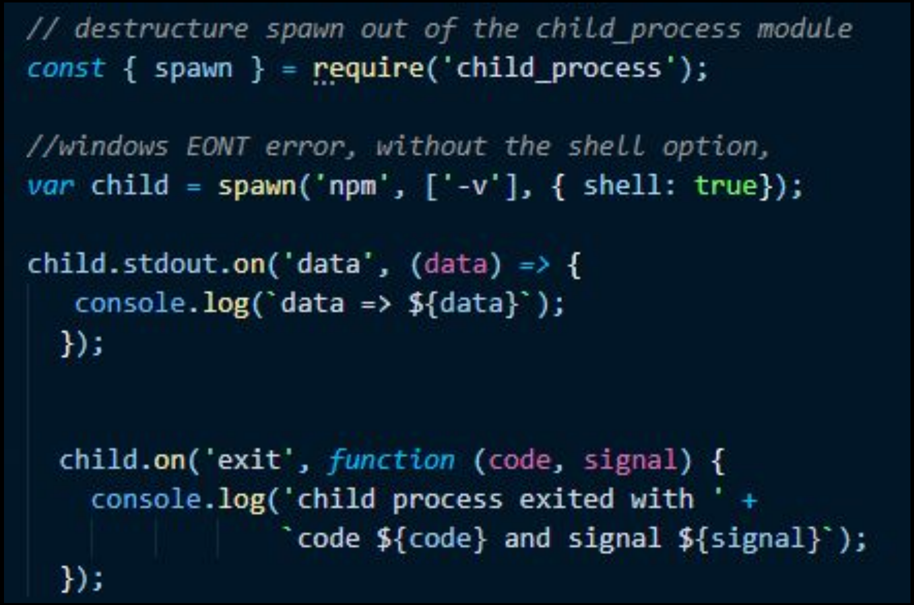
- The events that we can register handler for with childProcess instances are exit, disconnect, error, close and message

- The message event is the most important. It's emitted when the child process uses the process.send() to send messages. This how parent/child process can communicate with each other

- Every child process also gets the three standard stdio streams ie. child.stdin, child.stdout, child.stderr

- Since all streams are event emitters we can listen to different events attached to child process

spawn



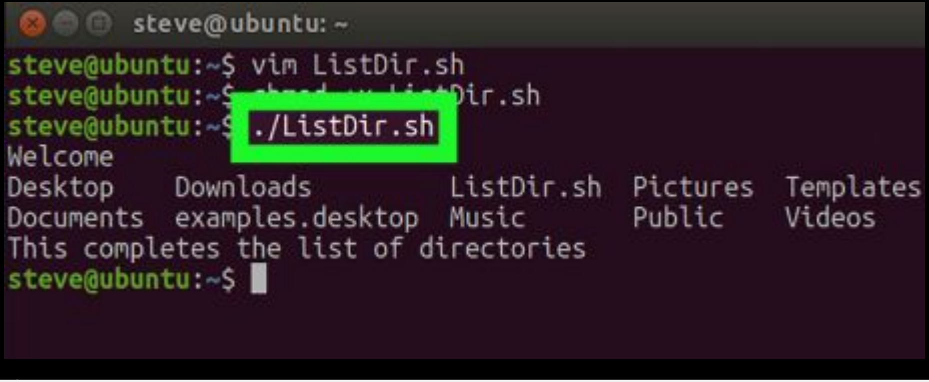
```
// destructure spawn out of the child_process module
const { spawn } = require('child_process');

//windows EONT error, without the shell option,
var child = spawn('npm', ['-v'], { shell: true});

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


  child.on('exit', function (code, signal) {
    console.log('child process exited with ' +
                `code ${code} and signal ${signal}`);
  });
```

spawn(command[, args][, options])

- The spawn function launches a command in a new process and we can use it to pass that command any arguments
- The result of spawn is a childProcess instance, which implements the EventEmitter API
- We can register event handlers for events on child directly i.e Exit event

shell

- A shell is a command-line interpreter or shell that provides a command line interface
- Bash is the command line shell, we use to run our Node commands and shell scripts
- Node provides us with a function that will span an instance of bash and execute the given command
- This function is called exec() and returns the stdout as a string, just like execFile() does.



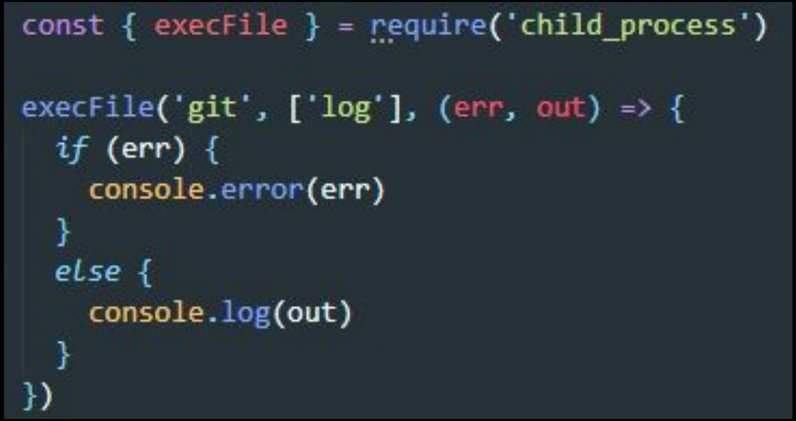
```
steve@ubuntu:~$ vim ListDir.sh
steve@ubuntu:~$ chmod +x ListDir.sh
steve@ubuntu:~$ ./ListDir.sh
Welcome
Desktop     Downloads          ListDir.sh  Pictures  Templates
Documents   examples.desktop   Music       Public    Videos
This completes the list of directories
steve@ubuntu:~$ 
```

exec

exec(command[options][callback])

- This function runs the provided command in a shell.
- Spawn does not create a shell, so it more efficient than the exec
- the exec function buffers the output and passes it to a callback function in the stout (instead of streams, which spawn does)
- * The spawn function is a much better choice when the data is large..

execFile

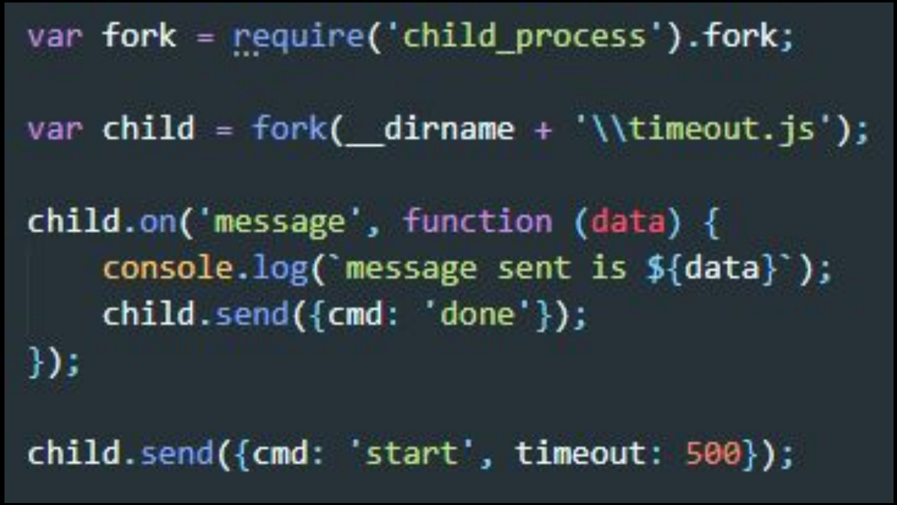


```
const { execFile } = require('child_process')

execFile('git', ['log'], (err, out) => {
  if (err) {
    console.error(err)
  }
  else {
    console.log(out)
  }
})
```

execFile(file[args],[options],[callback])

- ExecFile is similar to exec but instead of launching a process and executing the command, the file parameter is executed directly.

fork

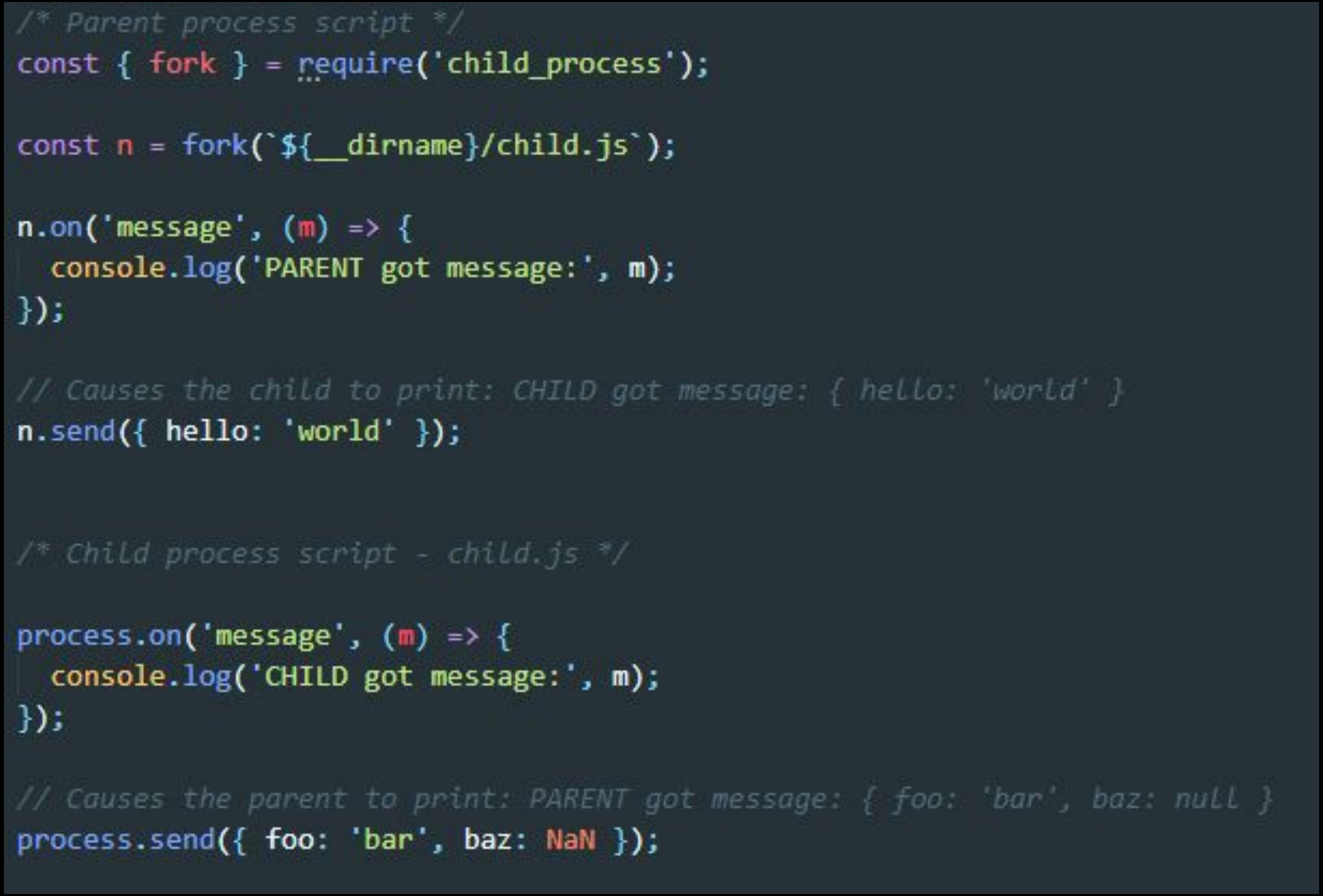


```
var fork = require('child_process').fork;

var child = fork(__dirname + '\\timeout.js');

child.on('message', function (data) {
    console.log(`message sent is ${data}`);
    child.send({cmd: 'done'});
});

child.send({cmd: 'start', timeout: 500});
```

- fork() is a specialized version of the spawn function especially for creating Node processes.

- Similar to spawn(), but it also adds an additional send function and message event to facilitate message passing between the parent and child processes.

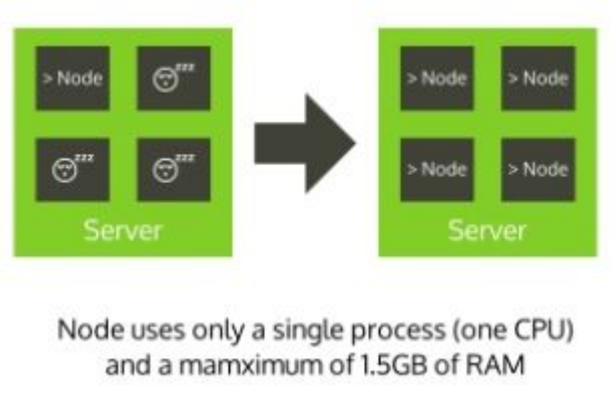
- The communication channel between the main process and the child process (known as ipc - Inter Process Communication)



```
/* Parent process script */
const { fork } = require('child_process');

const n = fork(`${__dirname}/child.js`);

n.on('message', (m) => {
  console.log('PARENT got message:', m);
});

// Causes the child to print: CHILD got message: { hello: 'world' }
n.send({ hello: 'world' });


/* Child process script - child.js */

process.on('message', (m) => {
  console.log('CHILD got message:', m);
});

// Causes the parent to print: PARENT got message: { foo: 'bar', baz: null }
process.send({ foo: 'bar', baz: NaN });
```

Cluster Module

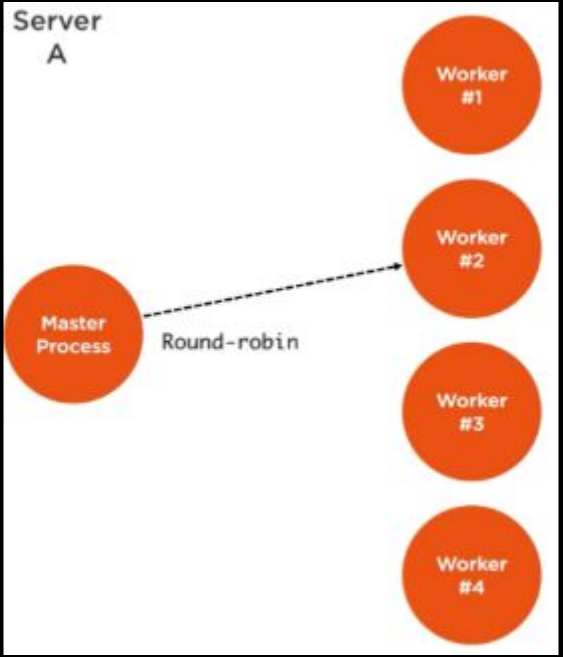
Cluster Module



Node uses only a single process (one CPU) and a mamximum of 1.5GB of RAM

- NodeJs single-threaded nature is by default to use a single core of a processor for code execution.
- Cluster scales an application execution on multiple processor cores by creating worker processes.
- Cluster module uses forking processes (similar to old fork() in Unix) to maximize the CPU usage

Master..Workers

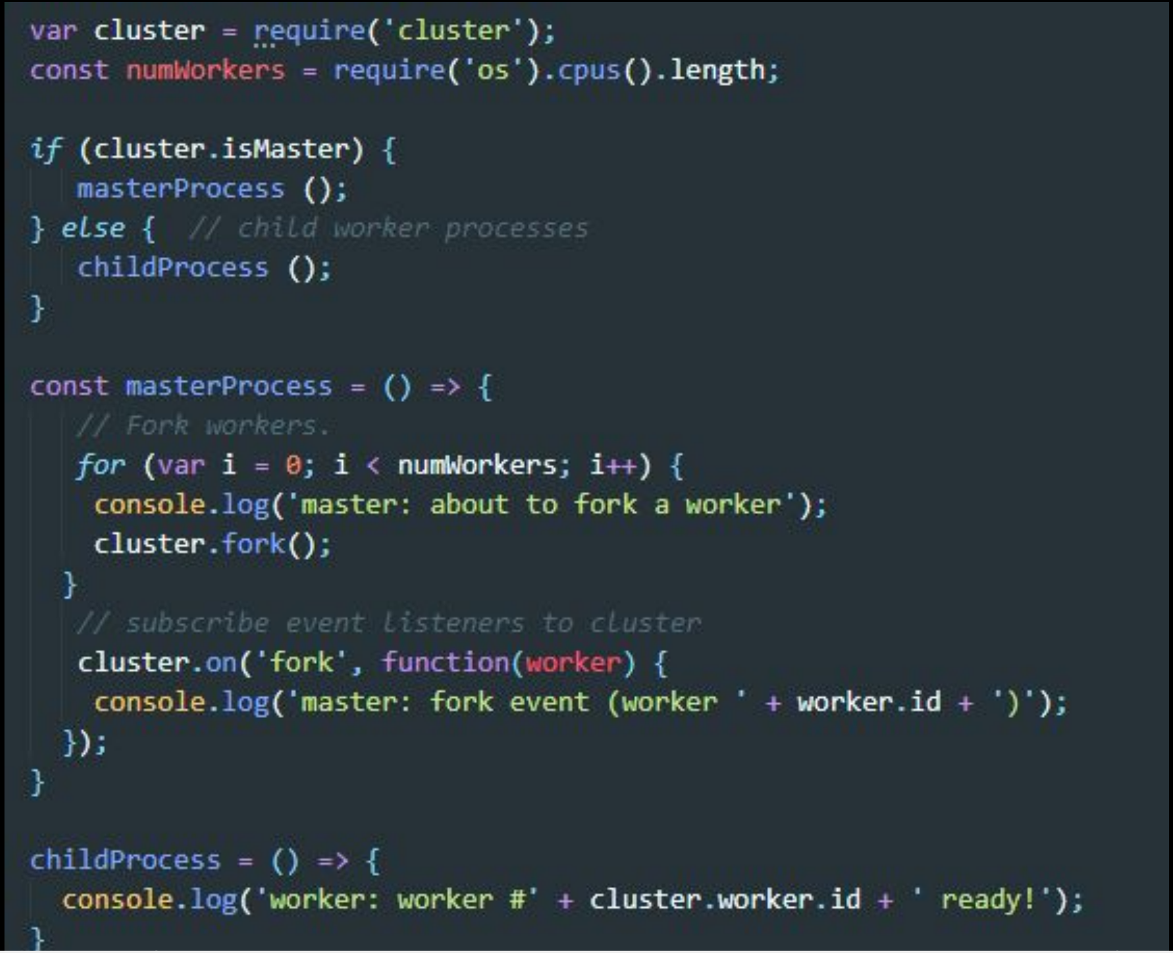


- With the cluster module a parent/master process can be forked in any number of child/worker processes

- Communication between worker processes and master happens through the IPC (Inter-process communication)

- Worker processes share a single port, therefore requests are routed through a shared port

- ** Remember there is no shared memory among processes



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

if (cluster.isMaster) {
   masterProcess ();
} else {  // child worker processes
   childProcess ();
}

const masterProcess = () => {
   // Fork workers.
   for (var i = 0; i < numWorkers; i++) {
    console.log('master: about to fork a worker');
    cluster.fork();
  }
   // subscribe event listeners to cluster
   cluster.on('fork', function(worker) {
    console.log('master: fork event (worker ' + worker.id + ')');
  });
}

childProcess = () => {
  console.log('worker: worker #' + cluster.worker.id + ' ready!');
}
```