8 Child processes: Integrating external applications with Node

This chapter covers

- Executing external applications
- Detaching a child process
- Interprocess communication between Node processes
- Making Node programs executable
- Creating job pools
- Synchronous child processes

No platform is an island. Although it would be fun to write everything in JavaScript, we'd miss out on valuable applications that already exist in other platforms. Take GraphicsMagick, for instance (http://www.graphicsmagick.org/): a full-featured image manipulation tool, great for resizing that massively large profile photo that was just uploaded. Or take wkhtmltopdf (http://wkhtmltopdf.org/), a headless webkit PDF generator, perfect for turning that HTML report into a PDF download.

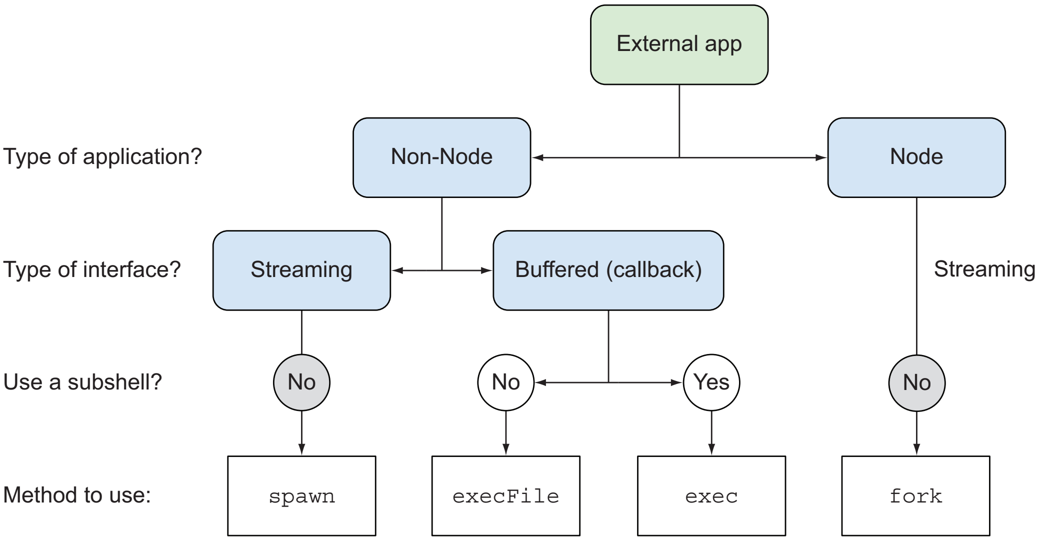

Figure 8.1 Choosing the right method

In Node, the `child_process` module allows us to execute these applications and others (including Node applications) to use with our programs. Thankfully, we don't have to re-invent the wheel.

The `child_process` module provides four different methods for executing external applications. All methods are asynchronous. The right method will depend on what you need, as shown in figure 8.1.

- *`execFile`*—Execute an external application, given a set of arguments, and callback with the buffered output after the process exits.
- *`spawn`*—Execute an external application, given a set of arguments, and provide a streaming interface for I/O and events for when the process exits.
- *`exec`*—Execute one or more commands inside a shell and callback with the buffered output after the process exits.
- *`fork`*—Execute a Node module as a separate process, given a set of arguments, provide a streaming and evented interface like `spawn`, and also set up an interprocess communication (IPC) channel between the parent and child process.

Throughout this chapter we'll dive into how to get the most out of these methods, giving practical examples of where you'd want to use each. Later on, we'll look into some other techniques to use when working with child processes: detaching processes, interprocess communication, file descriptors, and pooling.

8.1 Executing external applications

In this first section, we will look at all the ways you can work asynchronously with an external program.

TECHNIQUE 56 Executing external applications

Wouldn't it be great to run some image processing on a user's uploaded photo with ImageMagick, or validate an XML file with xmllint? Node makes it easy to execute external applications.

Problem

You want to execute an external application and get the output.

Solution

Use `execFile` (see figure 8.2).

Discussion

If you want to run an external application and get the result, using `execFile` makes it simple and straightforward. It'll buffer the output for you and provide the results and any errors in a callback. Let's say we want to run the `echo` program given the parameters `hello world`. With `execFile`, we would do the following:

```
var cp = require('child_process');

cp.execFile('echo', ['hello', 'world'],
   function (err, stdout, stderr) {
     if (err) console.error(err);
     console.log('stdout', stdout);
     console.log('stderr', stderr);
   });
```

Provide command as first parameter and any command arguments as an array for second parameter

Callback includes any error executing the command and buffered output from stdout and stderr

How does Node know where to find the external application? To answer that, we need to look at how paths work in the underlying operating system.

8.1.1 Paths and the PATH environment variable

Windows/UNIX has a `PATH` environment variable (envvar: http://en.wikipedia.org/wiki/PATH_(variable)). `PATH` contains a list of directories where executable programs exist. If a program exists in one of the listed directories, it can be located without needing an absolute or relative path to the application.

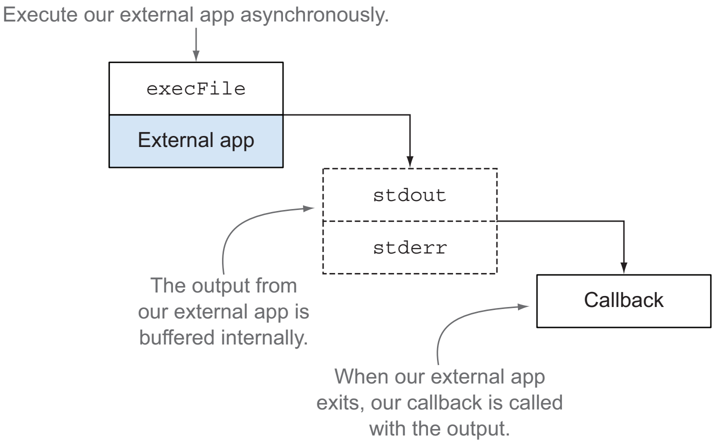
Execute our external app asynchronously.
execFile
External app
stdout
stderr
The output from our external app is buffered internally.
Callback
When our external app exits, our callback is called with the output.

Figure 8.2 The `execFile` method buffers the result and provides a callback interface.

Node, using execvp behind the scenes, will search for applications using PATH when no absolute or relative location is provided. We can see this in our earlier example, since directories to common system applications like echo usually exist in PATH already.

If the directory containing the application isn't in PATH, you'll need to provide the location explicitly like you would on the command line:

```
cp.execFile('./app-in-this-directory' ...
cp.execFile('/absolute/path/to/app' ...
cp.execFile('../relative/path/to/app' ...
```

To see what directories are listed in PATH, you can run a simple one-liner in the Node REPL:

```
$ node
> console.log(process.env.PATH.split(':').join('\n'))
/usr/local/bin
/usr/bin/bin
…
```

If you want to avoid including the location to external applications not in PATH, one option is to add any new directories to PATH inside your Node application. Just add this line before any execFile calls:

```
process.env.PATH += ':/a/new/path/to/executables';
```

Now any applications in that new directory will be accessible without providing a path to execFile.

8.1.2 Errors when executing external applications

If your external application doesn't exist, you'll get an ENOENT error. Often this is due to a typo in the application name or path with the result that Node can't find the application, as shown in figure 8.3.

If the external application does exist but Node can't access it (typically due to insufficient permissions), you'll get an EACCES or EPERM error. This can often be mitigated by either running your Node program as a user with sufficient permissions or changing the external application permissions themselves to allow access.

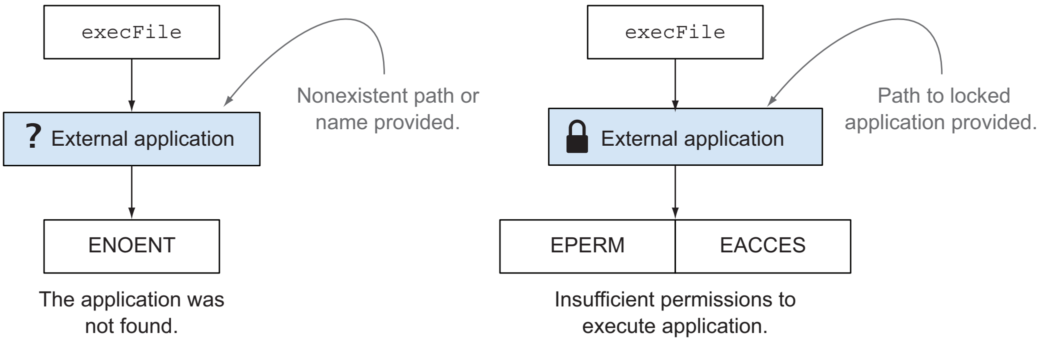
execFile
? External application
ENOENT
Nonexistent path or name provided.
The application was not found.
execFile
External application
EPERM
EACCES
Path to locked application provided.
Insufficient permissions to execute application.

Figure 8.3 **Common child process errors**

You'll also get an error if the external application has a non-zero exit status (http://mng.bz/MLXP), which is used to indicate that an application couldn't perform the task it was given (on both UNIX and Windows). Node will provide the exit status as part of the error object and will also provide any data that was written to stdout or stderr:

```
var cp = require('child_process');
cp.execFile('ls', ['non-existent-directory-to-list'],
  function (err, stdout, stderr) {
    console.log(err.code);
     console.log(stderr);
 });
```

Output exit code, which is 1 in this case, indicating command failed

Output error details stored in stderr

Having `execFile` is great for when you want to just execute an application and get the output (or discard it), for example, if you want to run an image-processing command with ImageMagick and only care if it succeeds or not. But if an application has a lot of output or you want to do more real-time analysis of the data returned, using streams is a better approach.

TECHNIQUE 57 Streaming and external applications

Imagine a web application that uses the output from an external application. As that data is being made available, you can at the same time be pushing it out to the client. Streaming enables you to tap into the data from a child process as it's being outputted, versus having the data buffered and then provided. This is good if you expect the external application to output large amounts of data. Why? Buffering a large set of data can take up a lot of memory. Also, this enables data to be consumed as it's being made available, which improves responsiveness.

■ Problem

You want to execute an external application and stream the output.

■ Solution

Use `spawn` (see figure 8.4).

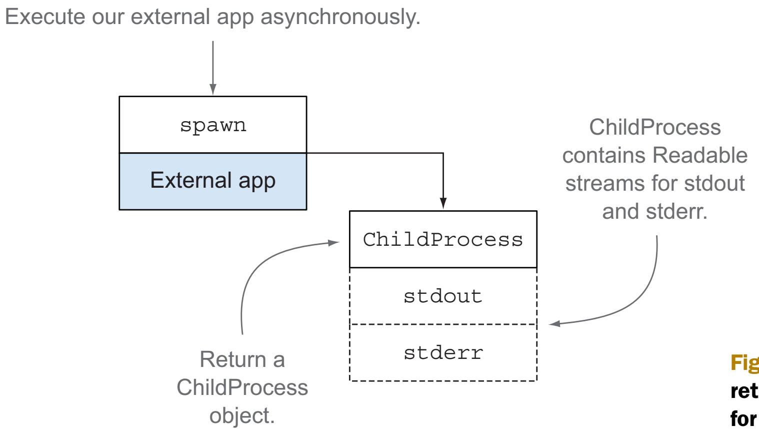
Execute our external app asynchronously.
spawn
External app
ChildProcess
stdout
stderr
ChildProcess contains Readable streams for stdout and stderr.
Return a ChildProcess object.

Figure 8.4 The `spawn` method returns a streaming interface for I/O.

Discussion

The spawn method has a function signature similar to execFile:

```
cp.execFile('echo', ['hello', 'world'], ...);
cp.spawn('echo', ['hello', 'world'], ...);
```

The application is the first argument, and an array of parameters/flags for the application is the second. But instead of taking a callback providing the output already buffered, spawn relies on streams:

```
var cp = require('child_process');

var child = cp.spawn('echo', ['hello', 'world']);
child.on('error', console.error);
child.stdout.pipe(process.stdout);
child.stderr.pipe(process.stderr);
```

Spawn method returns a ChildProcess object containing stdin, stdout, and stderr stream objects

Errors are emitted on error event

Output from stdout and stderr can be read as it's available

Since spawn is stream-based, it's great for handling large outputs or working with data as it's read in. All other benefits of streams apply as well. For example, child.stdin is a Writeable stream, so you can hook that up to any Readable stream to get data. The reverse is true for child.stdout and child.stderr, which are Readable streams that can be hooked into any Writeable stream.

API SYMMETRY The ChildProcess API (child.stdin, child.stdout, child.stderr) share a nice symmetry with the parent process streams (process.stdin, process.stdout, process.stderr).

8.1.3 Stringing external applications together

A large part of UNIX philosophy is building applications that do one thing and do it well, and then communicating between those applications with a common interface (that being plain text).

Let's make a Node program that exemplifies this by taking three simple applications that deal with text streams and sticking them together using spawn. The cat application will read a file and output its contents. The sort application will take in the file as input and provide the lines sorted as output. The uniq application will take the sorted file as input, and output the sorted file with all the duplicate lines removed. This is illustrated in figure 8.5.

Let's look at how we can do this with spawn and streams:

```
var cp = require('child_process');
var cat = cp.spawn('cat', ['messy.txt']);
var sort = cp.spawn('sort');
var uniq = cp.spawn('uniq');

cat.stdout.pipe(sort.stdin);
sort.stdout.pipe(uniq.stdin);
uniq.stdout.pipe(process.stdout);
```

Call spawn for each command we want to chain together

Stream result to the console with process.stdout

Output of each command becomes input for next command

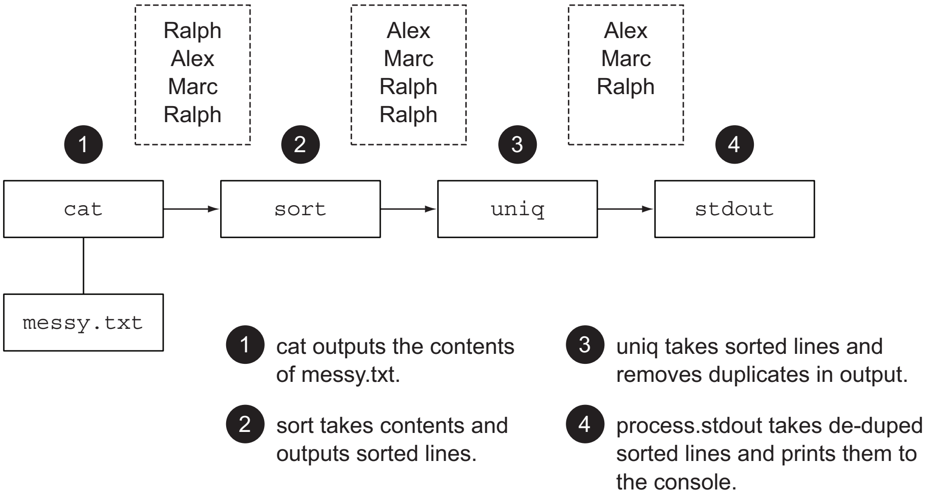
Ralph
Alex
Marc
Ralph
Alex
Marc
Ralph
Ralph
Alex
Marc
Ralph
1
2
3
4
cat
sort
uniq
stdout
messy.txt
1 cat outputs the contents of messy.txt.
2 sort takes contents and outputs sorted lines.
3 uniq takes sorted lines and removes duplicates in output.
4 process.stdout takes de-duped sorted lines and prints them to the console.

Figure 8.5 Stringing external applications together with `spawn`

Using `spawn`'s streaming interfaces allows a seamless way to work with any stream objects in Node, including stringing external applications together. But sometimes we need the facilities of our underlying shell to do powerful composition of external applications. For that, we can use `exec`.

> **APPLYING WHAT YOU'VE LEARNED** Can you think of a way to avoid using the `cat` program based on what you learned with the `fs` module and streaming in chapter 6?

TECHNIQUE 58 Executing commands in a shell

Shell programming is a common way to build utility scripts or command-line applications. You could whip up a Bash or Python script, but with Node, you can use JavaScript. Although you could execute a subshell manually using `execFile` or `spawn`, Node provides a convenient, cross-platform method for you.

■ Problem

You need to use the underlying shell facilities (like pipes, redirects, file blobs) to execute commands and get the output.

■ Solution

Use `exec` (see figure 8.6).

■ Discussion

If you need to execute commands in a shell, you can use `exec`. The `exec` method runs the commands with `/bin/sh` or `cmd.exe` (on Windows). Running commands in a shell means you have access to all the functionality provided by your particular shell (like pipes, redirects, and backgrounding).

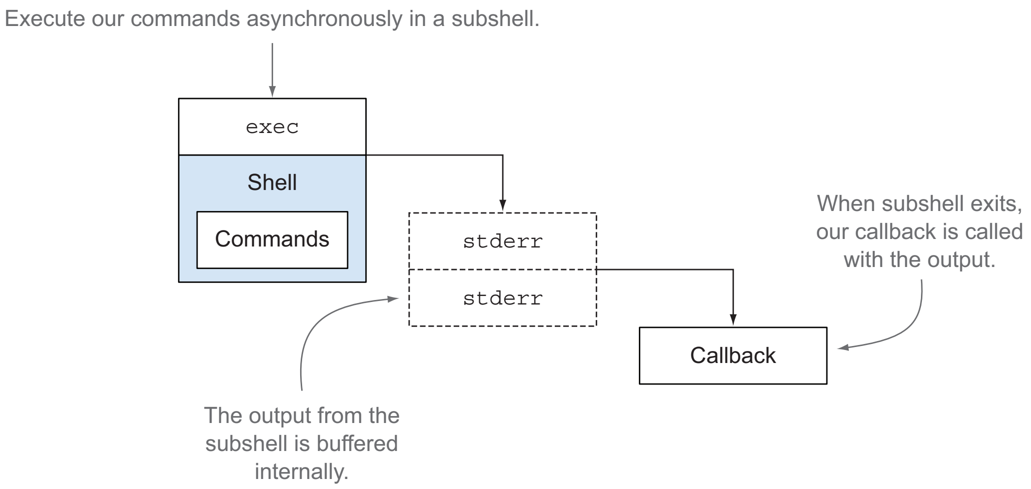
Execute our commands asynchronously in a subshell.
exec
Shell
Commands
stderr
stderr
When subshell exits, our callback is called with the output.
Callback
The output from the subshell is buffered internally.

Figure 8.6 The `exec` method runs our commands in a subshell.

> **A SINGLE COMMAND ARGUMENT** Unlike `execFile` and `spawn`, the `exec` method doesn't have a separate argument for command parameters/flags, since you can run more than one command on a shell.

As an example, let's pipe together the same three applications we did in the last technique to generate a sorted, unique list of names. But this time, we'll use common UNIX shell facilities rather than streams:

```
cp.exec('cat messy.txt | sort | uniq',
  function (err, stdout, stderr) {
    console.log(stdout);
  });
```

Pipe cat, sort, and uniq together like we would on command line

If successful, stdout will contain sorted, de-duped version of messy.txt

> **ABOUT SHELLS** UNIX users should keep in mind that Node uses whatever is mapped to `/bin/sh` for execution. This typically will be Bash on most modern operating systems, but you have the option to remap it to another shell of your liking. Windows users who need a piping facility can use streams and spawn as discussed in technique 57.

8.1.4 Security and shell command execution

Having access to a shell is powerful and convenient, but it should be used cautiously, especially with a user's input.

Let's say we're using xmllint (http://xmlsoft.org/xmllint.html) to parse and detect errors in a user's uploaded XML file where the user provides a schema to validate against:

```
cp.exec('xmllint --schema '+req.query.schema+' the.xml');
```

If a user provided "http://site.com/schema.xsd," it would be replaced and the following command would run:

```
xmllint --schema http://site.com/schema.xsd the.xml
```

But since the argument has user input, it can easily fall prey to command (or shell) injection attacks (https://golemtechnologies.com/articles/shell-injection)—for example, a malicious user provides "; rm -rf / ;" causing the following comment to run (*please don't run this in your terminal!*):

```
xmllint --schema ; rm -rf / ; the.xml
```

If you haven't guessed already, this says, "Start new command (`;`), remove forcibly and recursively all files/directories at root of the file system (`rm -rf /`), and end the command (`;`) in case something follows it."

In other words, this injection could potentially delete all the files the Node process has permission to access on the entire operating system! And that's just one of the commands that can be run. Anything your process user has access to (files, commands, and so on) can be exploited.

If you need to run an application and don't need shell facilities, it's safer (and slightly faster) to use `execFile` instead:

```
cp.execFile('xmllint', ['--schema', req.query.schema, 'the.xml']);
```

Here this malicious injection attack would fail since it's not run in a shell and the external application likely wouldn't understand the argument and would raise an error.

TECHNIQUE 59 Detaching a child process

Node can be used to kick off external applications and then allow them to run on their own. For example, let's say you have an administrative web application in Node that allows you to kick off a long-running synchronization process with your cloud provider. If that Node application were to crash, your synchronization process would be halted. To avoid this, you detach your external application so it'll be unaffected.

■ Problem

You have a long-running external application that you want Node to start but then be able to exit with the child process still running.

■ Solution

Detach a spawned child process (see figure 8.7).

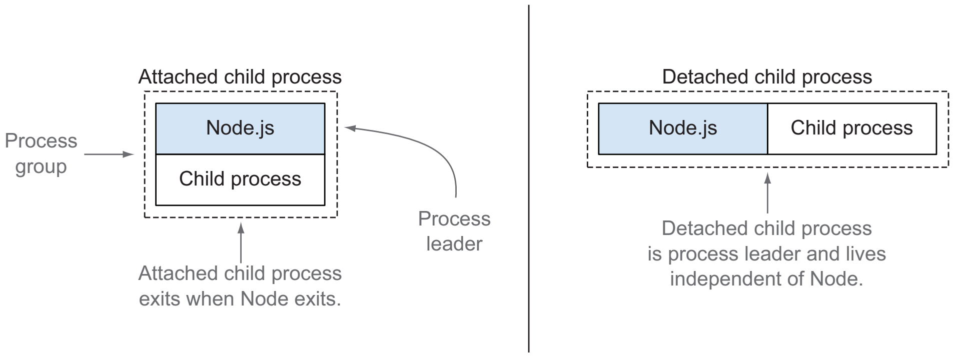
Attached child process
Process group
Node.js
Child process
Process leader
Attached child process exits when Node exits.
Detached child process
Node.js
Child process
Detached child process is process leader and lives independent of Node.

Figure 8.7 Detached child process exists independent of the Node process

Discussion

Normally, any child process will be terminated when the parent Node process is terminated. Child processes are said to be *attached* to the parent process. But the `spawn` method includes the ability to *detach* a child process and promote it to be a process group leader. In this scenario, if the parent is terminated, the child process will continue until finished.

This scenario is useful when you want Node to set up the execution of a long-running external process and you don't need Node to babysit it after it starts.

This is the `detached` option, configurable as part of a third options parameter to `spawn`:

```
var child = cp.spawn('./longrun', [], { detached: true });
```

In this example, `longrun` will be promoted to a process group leader. If you were to run this Node program and forcibly terminate it (Ctrl-C), `longrun` would continue executing until finished.

If you didn't forcibly terminate, you'd notice that the parent stays alive until the child has completed. This is because I/O of the child process is connected to the parent. In order to disconnect the I/O, you have to configure the `stdio` option.

8.1.5 Handing I/O between the child and parent processes

The `stdio` option defines where the I/O from a child process will be redirected. It takes either an array or a string as a value. The string values are simply shorthands that will expand to common array configurations.

The array is structured such that the *indexes* correspond to file descriptors in the child process and the *values* indicate where the I/O for the particular file descriptor (FD) should be redirected.

> **WHAT ARE FILE DESCRIPTORS?** If you're confused about file descriptors, check out technique 40 in chapter 6 for an introduction.

By default, `stdio` is configured as

```
stdio: 'pipe'
```

which is a shorthand for the following array values:

```
stdio: [ 'pipe', 'pipe', 'pipe' ]
```

This means that file descriptors 0-2 will be made accessible on the `ChildProcess` object as streams (`child.stdio[0]`, `child.stdio[1]`, `child.stdio[2]`). But since FDs 0-2 often refer to `stdin`, `stdout`, and `stderr`, they're also made available as the now familiar `child.stdin`, `child.stdout`, and `child.stderr` streams.

The `pipe` value connects the parent and child processes because these streams stay open, waiting to write or read data. But for this technique, we want to disconnect the two in order to exit the Node process. A brute-force approach would be to simply destroy all the streams created:

```
child.stdin.destroy();
child.stdout.destroy();
child.stderr.destroy();
```

Although this would work, given our intent to not use them, it's better to not create the streams in the first place. Instead, we can assign a file descriptor if we want to direct the I/O elsewhere or use `ignore` to discard it completely.

Let's look at a solution that uses both options. We want to `ignore` FD 0 (`stdin`) since we won't be providing any input to the child process. But let's capture any output from FDs 1 and 2 (`stdout`, `stderr`) just in case we need to do some debugging later on. Here's how we can accomplish that:

```
var fs = require('fs');
var cp = require('child_process');

var outFd = fs.openSync('./longrun.out', 'a');
var errFd = fs.openSync('./longrun.err', 'a');

var child = cp.spawn('./longrun', [], {
  detached: true,
  stdio: [ 'ignore', outFd, errFd ]
});
```

Open two log files, one for stdout and one for stderr

Ignore FD 0; redirect output from FDs 1 and 2 to the log files

This will disconnect the I/O between the child and parent processes. If we run this application, the output from the child process will end up in the log files.

8.1.6 Reference counting and child processes

We're almost there. The child process will live on because it's detached and the I/O is disconnected from the parent. But the parent still has an internal reference to the child process and won't exit until the child process has finished and the reference has been removed.

You can use the `child.unref()` method to tell Node not to include this child process reference in its count. The following complete application will now exit after spawning the child process:

```
var fs = require('fs');
var cp = require('child_process');

var outFd = fs.openSync('./longrun.out', 'a');
var errFd = fs.openSync('./longrun.err', 'a');

var child = cp.spawn('./longrun', [], {
  detached: true,
  stdio: [ 'ignore', outFd, errFd ]
});

child.unref();
```

Remove reference of child in the parent process

To review, detaching a process requires three things:

- The `detached` option must be set to `true` so the child becomes its own process leader.
- The `stdio` option must be configured so the parent and child are disconnected.
- The reference to the child must be severed in the parent using `child.unref()`.

8.2 Executing Node programs

Any of the prior techniques can be used to execute Node applications. However, in the techniques to follow, we will focus on making the most out of Node child processes.

TECHNIQUE 60 Executing Node programs

When writing shell scripts, utilities, or other command-line applications in Node, it's often handy to make executables out of them for ease of use and portability. If you publish command-line applications to npm, this also comes in handy.

Problem

You want to make a Node program an executable script.

Solution

Set up the file to be executable by your underlying platform.

Discussion

A Node program can be run as a child process with any of the means we've already described by simply using the `node` executable:

```
var cp = require('child_process');
cp.execFile('node', ['myapp.js', 'myarg1', 'myarg2' ], ...
```

But there are many cases where having a standalone executable is more convenient, where you can instead use your app like this:

```
myapp myarg1 myarg2
```

The process for making an executable will vary depending on whether you're on Windows or UNIX.

Executables on Windows

Let's say we have a simple one-liner `hello.js` program that echoes the first argument passed:

```
console.log('hello', process.argv[2]);
```

To run this program, we type

```
$ node hello.js marty
hello marty
```

To make a Windows executable, we can make a simple batch script calling the Node program. For consistency, let's call it `hello.bat`:

```
@echo off
node "hello.js" %*
```

Don't echo commands to stdout

Call node executable, passing in any additional parameters (%*)

Now we can execute our hello.js program by simply running the following:

```
$ hello tom
hello tom
```

Running it as a child process requires the `.bat` extension:

```
var cp = require('child_process');
cp.execFile('hello.bat', ['billy'], function (err, stdout) {
  console.log(stdout); // hello billy
});
```

Executables on UNIX

To turn a Node program into an executable script on most UNIX systems, we don't need a separate batch file like in Windows; we simply modify `hello.js` itself by adding the following to the top of the file:

```
#!/usr/bin/env node
console.log('hello', process.argv[2]);
```

Execute node command wherever it's found in user's environment

Then to actually make the file executable, we run the following command:

```
$ chmod +x hello.js
```

We can then run the command like this:

```
$ ./hello.js jim
hello jim
```

The file can be renamed as well to look more like a standalone program:

```
$ mv hello.js hello
$ ./hello jane
hello jane
```

Executing this program as a child process will look the same as its command-line counterpart:

```
var cp = require('child_process');
cp.execFile('./hello', ['bono'], function (err, stdout) {
  console.log(stdout); // hello bono
});
```

> **PUBLISHING EXECUTABLE FILES IN NPM** For publishing packages that contain executable files, use the UNIX conventions, and npm will make the proper adjustments for Windows.

TECHNIQUE 61 Forking Node modules

Web workers (http://mng.bz/UG63) provide the browser and JavaScript an elegant way to run computationally intense tasks off the main thread with a built-in communication stream between the parent and worker. This removes the painful work of breaking up computation into pieces in order to not upset the user experience. In Node, we have the same concept, with a slightly different API with `fork`. This helps us break out any heavy lifting into a separate process, keeping our event loop running smoothly.

Problem

You want to manage separate Node processes.

Solution

Use `fork` (see figure 8.8).

Discussion

Sometimes it's useful to have separate Node processes. One such case is computation. Since Node is single-threaded, computational tasks directly affect the performance of the whole process. This may be acceptable for certain jobs, but when it comes to network programming, it'll severely affect performance since requests can't be serviced when the process is tied up. Running these types of tasks in a forked process allows the main application to stay responsive. Another use of forking is for sharing file descriptors, where a child can accept an incoming connection received by the parent process.

Node provides a nice way to communicate between other Node programs. Under the hood, it sets up the following `stdio` configuration:

```
stdio: [ 0, 1, 2, 'ipc' ]
```

This means that, by default, all output and input are directly inherited from the parent; there's no `child.stdin`, `child.stdout`, or `child.stderr`:

```
var cp = require('child_process');
var child = cp.fork('./myChild');
```

If you want to provide an I/O configuration that behaves like the `spawn` defaults (meaning you get a `child.stdin`, and so on), you can use the `silent` option:

```
var cp = require('child_process');
var child = cp.fork('./myChild', { silent: true });
```

> **INTERNALS OF INTERPROCESS COMMUNICATION** Although a number of mechanisms exist to provide interprocess communication (IPC; see http://mng.bz/LGKD), Node IPC channels will use either a UNIX domain socket (http://mng.bz/1189) or a Windows named pipe (http://mng.bz/262Q).

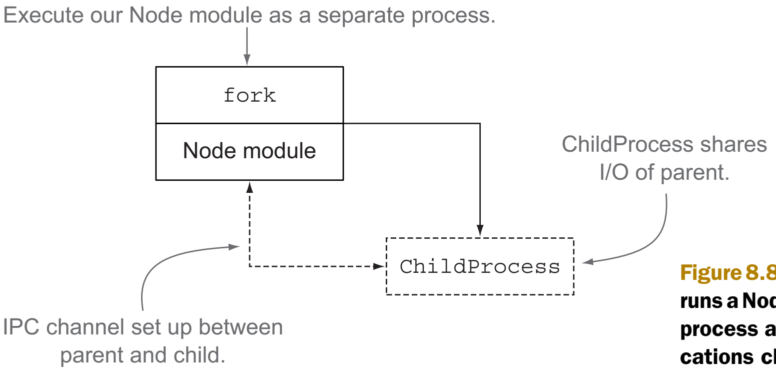
Execute our Node module as a separate process.
fork
Node module
ChildProcess shares I/O of parent.
ChildProcess
IPC channel set up between parent and child.

Figure 8.8 The `fork` command runs a Node module in a separate process and sets up a communications channel.

Communicating with forked Node modules

The `fork` method opens up an IPC channel that allows message passing between Node processes. On the child side, it exposes `process.on('message')` and `process.send()` as mechanisms for receiving and sending messages. On the parent side, it provides `child.on('message')` and `child.send()`.

Let's make a simple echo module that sends back any message received from the parent:

```
process.on('message', function (msg) {
  process.send(msg);
});
```

When child receives a message, this handler will be called

Send message back to parent

An application can now consume this module using `fork`:

```
var cp = require('child_process');
var child = cp.fork('./child');
child.on('message', function (msg) {
  console.log('got a message from child', msg);
});
child.send('sending a string');
```

When parent receives a message, this handler will be called

Log out echoed message

Send message to child process

Sending data between the processes maintains the type information, which means you can send any valid JSON value over the wire and it retains the type:

```
child.send(230);
child.send('a string');
child.send(true);
child.send(null);
child.send({ an: 'object' });
```

Disconnecting from forked Node modules

Since we're opening an IPC channel between the parent and child, both stay alive until the child is disconnected (or exits some other way). If you need to disconnect the IPC channel, you can do that explicitly from the parent process:

```
child.disconnect();
```

TECHNIQUE 62 Running jobs

When you need to run *routine* computational jobs, forking processes on demand will quickly eat up your CPU resources. It's better to keep a job pool of available Node processes ready for work. This technique takes a look at that.

■ Problem

You have routine jobs that you don't want to run on the main event loop.

■ Solution

Use `fork` and manage a pool of workers.

■ Discussion

We can use the IPC channel built into `fork` to create a pattern for handling computationally intensive tasks (or jobs). It builds upon our last technique, but adds an important constraint: when the parent sends a task to the child, it expects to receive exactly one result. Here's how this works in the parent process:

```
function doWork (job, cb) {
  var child = cp.fork('./worker');
  child.send(job);
  child.once('message', function (result) {
     cb(null, result);
  });
}
```

Send job to the child process

Expect child to respond with exactly one message providing the result

But receiving a result is only one of the possible outcomes. To build resilience into our `doWork` function, we'll account for

- The child exiting for any reason
- Unexpected errors (like a closed IPC channel or failure to fork)

Handling those in code will involve a couple more listeners:

```
child.once('error', function (err) {
  cb(err);
  child.kill();
 });
child.once('exit', function (code, signal) {
  cb(new Error('Child exited with code: ' + code));
});
```

Unexpected error; kill the process as it's likely unusable

This is a good start, but we run the risk of calling our callback more than once in the case where the worker finished the job but then later exited or had an error. Let's add some state and clean things up a bit:

```
function doWork (job, cb) {
  var child = cp.fork('./worker');
  var cbTriggered = false;

  child
    .once('error', function (err) {
      if (!cbTriggered) {
        cb(err);
        cbTriggered = true;
      }
      child.kill();
    })
    .once('exit', function (code, signal) {
      if (!cbTriggered)
        cb(new Error('Child exited with code: ' + code));
    })
    .once('message', function (result) {
      cb(null, result);
      cbTriggered = true;
    })
    .send(job);
}
```

Track if callback was called

Message will never be triggered if an error or exit happens, so we don't need cbTriggered check

So far we've only looked at the parent process. The child worker takes in a job, and sends exactly one message back to the parent when completed:

```
process.on('message', function (job) {
  // do work
  process.send(result);
});
```

8.2.1 Job pooling

Currently, our doWork function will spin up a new child process every time we need to do some work. This isn't free, as the Node documentation states:

> *These child Nodes are still whole new instances of V8. Assume at least 30ms startup and 10mb memory for each new Node. That is, you cannot create many thousands of them.*

A performant way to work around this is not to spin off a new process whenever you want to do something computationally expensive, but rather to maintain a pool of long-running processes that can handle the load.

Let's expand our doWork function, creating a module for handling a worker pool. Here are some additional constraints we'll add:

- Only fork up to as many worker processes as CPUs on the machine.
- Ensure new work gets an available worker process and not one that's currently in-process.
- When no worker processes are available, maintain a queue of tasks to execute as processes become available.
- Fork processes on demand.

Let's take a look at the code to implement this:

```
var cp = require('child_process');
var cpus = require('os').cpus().length; )

module.exports = function (workModule) {
  var awaiting = [];
  var readyPool = [];
  var poolSize = 0;

  return function doWork (job, cb) {
    if (!readyPool.length && poolSize > cpus)
      return awaiting.push([ doWork, job, cb ]);

    var child = readyPool.length
      ? readyPool.shift()
      : (poolSize++, cp.fork(workModule));
    var cbTriggered = false;

    child
```

Grab number of CPUs

Keep list of tasks that are queued to run when all processes are in use

Keep list of worker processes that are ready for work

Keep track of how many worker processes exist

If no worker processes are available and we've reached our limit, queue work to be run later

Grab next available child, or fork a new process (incrementing the poolSize)

```
      .removeAllListeners()
      .once('error', function (err) {
        if (!cbTriggered) {
          cb(err);
          cbTriggered = true;
        }
        child.kill();
      })
      .once('exit', function () {
        if (!cbTriggered)
          cb(new Error('Child exited with code: ' + code));
        poolSize--;
        var childIdx = readyPool.indexOf(child);
        if (childIdx > -1) readyPool.splice(childIdx, 1);
      })
      .once('message', function (msg) {
        cb(null, msg);
        cbTriggered = true;
        readyPool.push(child);
        if (awaiting.length) setImmediate.apply(null, awaiting.shift());
      })
      .send(job);
  }
}
```

Remove any listeners that exist on child, ensuring that a child process will always have only one listener attached for each event at a time

If child exits for any reason, ensure it's removed from the readyPool

Child is ready again; add back to readyPool and run next awaiting task (if any)

APPLYING WHAT YOU'VE LEARNED Other constraints may apply depending on the needs of the pool, for example, retrying jobs on failure or killing long-running jobs. How would you implement a retry or timeout using the preceding example?

8.2.2 Using the pooler module

Let's say we want to run a computationally intensive task based on a user's request to our server. First, let's expand our child worker process to simulate an intensive task:

Actual work happens here; in our case, we'll simply generate a CPU load on the child

Receive task from the parent

```
process.on('message', function (job) {
  for (var i = 0; i < 1000000000; i++);
  process.send('finished: ' + job);
});
```

Send result of task back to the parent

Now that we have a sample child process to run, let's put this all together with a simple application that uses the pooler module and worker modules:

Create job pool around the worker module

Include pooler module to make job pools

```
var http = require('http');
var makePool = require('./pooler');
var runJob = makePool('./worker');

http.createServer(function (req, res) {
  runJob('some dummy job', function (er, data) {
```

Run job on every request to the server, responding with the result

```
        if (er) return res.end('got an error:' + er.message);
        res.end(data);
      });
    }).listen(3000);
```

Pooling saves the overhead of spinning up and destroying child processes. It makes use of the communications channels built into `fork` and allows Node to be used effectively for managing jobs across a set of child processes.

> **GOING FURTHER** To further investigate job pools, check out the third-party `compute-cluster` module (https://github.com/lloyd/node-compute-cluster).

We've discussed asynchronous child process execution, which is when you need to juggle multiple points of I/O, like servers. But sometimes you just want to execute commands one after another without the overhead. Let's look at that next.

8.3 Working synchronously

Non-blocking I/O is important for keeping the event loop humming along without having to wait for an unwieldy child process to finish. However, it has extra coding overhead that isn't pleasant when you want things to block. A good example of this is writing shell scripts. Thankfully, synchronous child processes are also available.

TECHNIQUE 63 Synchronous child processes

Synchronous child process methods are recent additions to the Node scene. They were first introduced in Node 0.12 to address a very real problem in a performant and familiar manner: shell scripting. Before Node 0.12, clever but nonperformant hacks were used to get synchronous-like behavior. Now, synchronous methods are a first-class citizen.

In this technique we'll cover all the synchronous methods available in the child process modules.

■ **Problem**

You want to execute commands synchronously.

■ **Solution**

Use `execFileSync`, `spawnSync`, and `execFile`.

■ **Discussion**

By now, we hope these synchronous methods look extremely familiar. In fact, they're the same in their function signatures and purpose as we've discussed previously in this chapter, with one important distinction—*they block and run to completion* when called.

If you just want to execute a single command and get output synchronously, use `execFileSync`:

```
var ex = require('child_process').execFileSync;
var stdout = ex('echo', ['hello']).toString();
console.log(stdout);
```

Extract execFileSync method as ex for a shorthand way to refer to it

execFileSync returns a Buffer of the output, which is then converted to a UTF-8 string and assigned to stdout

Outputs "hello"

If you want to execute multiple commands synchronously and programmatically where the input of one depends on the output of another, use spawnSync:

```
var sp = require('child_process').spawnSync;
var ps = sp('ps', ['aux']);
var grep = sp('grep', ['node'], {
  input: ps.stdout,
  encoding: 'utf8'
});
console.log(grep);
```

Extract spawnSync method as sp for a shorthand way to refer to it

Run ps aux and grep node synchronously

Pass stdout Buffer from ps aux as input to grep node

Indicate all resulting stdio should be in UTF-8

The resulting synchronous child process contains a lot of detail of what happened, which is another advantage of using spawnSync:

```
{ status: 0,
  signal: null,
  output:
     [ null,
      'wavded 4376 ... 9:03PM 0:00.00 (node)\n
       wavded 4400 ... 9:11PM 0:00.10 node spawnSync.js\n',
      '' ],
  pid: 4403,
  stdout: 'wavded ... 9:03PM 0:00.00 (node)\n
    wavded 4400 ... 9:11PM 0:00.10 node spawnSync.js\n',
  stderr: '',
  envPairs:
     [ 'USER=wavded',
      'EDITOR=vim',
      'NODE_PATH=/Users/wavded/.nvm/v0.11.12/lib/node_modules:',
      ... ],
  options:
     { input: <Buffer 55 53 45 52 20 20 20 ... >,
       encoding: 'utf8',
       file: 'grep',
       args: [ 'grep', 'node' ],
       stdio: [ [Object], [Object], [Object] ] },
       args: [ 'grep', 'node' ],
       file: 'grep' } [
```

Exit status of the process

Signal used to end the process

Output of all the stdio streams used in the child process; we can see that index I (stdout) has data

PID of the process

stdout output for the process

stderr output for the process

Environment variables present when the process ran

Options used to create the process; we can see our input buffer from ps aux here

Arguments used to execute the process

Executable file

Lastly, there's execSync, which executes a subshell synchronously and runs the commands given. This can be handy when writing shell scripts in JavaScript:

```
var ex = require('child_process').execSync;
var stdout = ex('ps aux | grep').toString();
console.log(stdout);
```

Extract execSync method as ex for a shorthand way to refer to it

Execute shell command synchronously and return the output as a string

This will output the following:

```
wavded 4425 29.7 0.2 ... 0:00.10 node execSync.js
wavded 4427 1.5 0.0 ... /bin/sh -c ps aux | grep node
wavded 4429 0.5 0.0 ... grep node
wavded 4376 0.0 0.0 ... (node)
```

Output shows we're executing a subshell with execSync

Error handing with synchronous child process methods

If a non-zero exit status is returned in `execSync` or `execFileSync`, an exception will be thrown. The error object will include everything we saw returned using `spawnExec`. We'll have access to important things like the status code and `stderr` stream:

```
var ex = require('child-process').execFileSync;
try {
  ex('cd', ['non-existent-dir'], {
     encoding: 'utf8'
  });
} catch (err) {
  console.error('exit status was', err.status);
  console.error('stderr', err.stderr);
}
```

Executing cd on nonexistent directory gives non-zero exit status

Although more verbose than toString(), setting encoding to UTF-8 here will set it for all our stdio streams when we handle the error

This program yields the following output:

```
exit status was 1
stderr /usr/bin/cd: line 4:cd:
  non-existent-dir: No such file or directory
```

We talked errors in `execFile` and `execFileSync`. What about `spawnSync`? Since `spawnSync` returns everything that happens when running the process, it doesn't throw an exception. Therefore, you're responsible to check the success or failure.

8.4 Summary

In this chapter you learned to integrate different uses of external applications in Node by using the `child_process` module. Here are some tips in summary:

- Use `execFile` in cases where you just need to execute an external application. It's fast, simple, and safer when dealing with user input.
- Use `spawn` when you want to do something more with the I/O of the child process, or when you expect the process to have a large amount of output. It provides a nice streamable interface, and is also safer when dealing with user input.
- Use `exec` when you want to access your shell's facilities (pipes, redirects, blobs). Many shells allow running multiple applications in one go. Be careful with user input though, as it's never a good idea to put untrusted input into an `exec` call.
- Use `fork` when you want to run a Node module as a separate process. This enables computation and file descriptor handling (like an incoming socket) to be handled off the main Node process.

- Detach `spawned` processes you want to survive after a Node process dies. This allows Node to be used to set up long-running processes and let them live on their own.
- Pool a cluster of Node processes and use the built-in IPC channel to save the overhead of starting and destroying processes on every `fork`. This is useful for building computational clusters of Node processes.

This concludes our dive into Node fundamentals. We focused on specific core module functionality, focusing on idiomatic Node principals. In the next section, our focus will expand beyond core concepts into real-world development recipes.