Extracted from:

Async JavaScript

Build More Responsive Apps with Less Code

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Express

Async JavaScript

Build More Responsive Apps with Less Code

Trevor Burnham

edited by Jacquelyn Carter

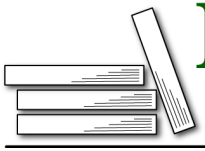
Async JavaScript

Build More Responsive Apps with Less Code

Trevor Burnham

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-27-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2012

CHAPTER 5

Multithreading with Workers

At the start of this book, I described events as an alternative to multithreading. More precisely, events replace a specific kind of multithreading, the kind where multiple parts of an application process run simultaneously (either virtually, through interrupts, or physically on multiple CPU cores). This gets to be a problem when code running in different threads has access to the same data. Even a line as simple as

```
i++;
```

can be a source of pernicious Heisenbugs[1] when it allows separate threads to modify the same i at the same time. Thankfully, this kind of multithreading is impossible in JavaScript.

On the other hand, distributing tasks across multiple CPU cores is increasingly essential because those cores are no longer making the same exponential gains in efficiency, year after year, that used to be expected. So, we *need* multithreading. Does that mean abandoning event-based programming?

Au contraire! While running on a single thread isn't ideal, naïvely distributing an app across multiple cores can be even worse. Multicore systems slow to a crawl when those cores have to constantly talk to each other to avoid stepping on each other's toes. It's much better to give each core a separate job and then sync up occasionally.

That's precisely what workers do in JavaScript. From the master thread of your application, you tell a worker, "Go run this code in a separate thread." The worker can send you messages (and vice versa), which take the form of (what else?) callbacks run from the event queue. In short, you interact with different threads the same way you do I/O in JavaScript.

1. http://en.wikipedia.org/wiki/Heisenbug

In this chapter, we'll look at workers in both their browser and Node manifestations, and we'll discuss some practical applications.

Threads vs. Processes

In this chapter, I throw around the words *thread* and *process* interchangeably. At the operating system level, there's an important distinction: threads within a process can share state, while separate processes can't. But in JavaScript, concurrent code (as run by workers) never shares state. So, workers may be implemented using lightweight OS threads, but they behave like processes.

There are Node libraries, most notably, Threads-A-GoGo,[a] that allow you to break the state-sharing rule for the sake of efficiency. Those are beyond the scope of this chapter, which is concerned only with concurrency in standard JavaScript.

a. https://github.com/xk/node-threads-a-gogo

5.1 Web Workers

Web workers are part of the living standard widely known as HTML5. To create one, you call the global Worker constructor with the URL of a script.

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(e) {
  console.log(e.data);  // echo whatever was sent by postMessage
});
```

(Usually, we want only the data property from the message event. If we were binding the same event handler to multiple workers, we could use e.target to determine which worker emitted the event.)

So, now we know how to listen to workers. Conveniently, the interface for talking to workers is symmetrical: we use worker.postMessage to send it, and the worker uses self.addEventListener('message', ...) to receive it. Here's a complete example:

```
// master script
var worker = new Worker('boknows.js');
worker.addEventListener('message', function(e) {
  console.log(e.data);
});
worker.postMessage('football');
worker.postMessage('baseball');

// boknows.js
self.addEventListener('message', function(e) {
  self.postMessage('Bo knows ' + e.data);
});
```

You can play with the message-passing interface at a little site I created, the Web Worker Sandbox.[2] Any time you create a new example, it gets a unique URL that you can share.

Restrictions on Web Workers

Web workers are primarily intended to handle complex computations without compromising DOM responsiveness. Potential uses include the following:

- Decoding video as it streams in with the Broadway implementation of the H.264 codec[3]
- Encrypting communications with the Stanford JavaScript Crypto Library[4]
- Parsing text in a web-based editor, à la Ace[5]

In fact, Ace already does this by default. When you type code into an Ace-based editor, Ace needs to perform some pretty heavy string analysis before updating the DOM with appropriate syntax highlighting. In modern browsers, that analysis is done on a separate thread, ensuring that the editor remains smooth and responsive.

Typically, the worker will send the result of its computations to the master thread, which will then update the page. Why not update the page directly? Mainly, to keep JavaScript's async abstractions intact. If a worker could alter the page's markup, we'd end up in the same place as Java, wrapping our DOM manipulation code in mutexes and semaphores to avoid race conditions.

Likewise, a worker can't see the global `window` object or any other object in the master thread (or in other worker threads). When an object is sent through `postMessage`, it's transparently serialized and unserialized; think `JSON.parse(JSON.stringify(obj))`. So, changes to the original object won't affect the copy in the other thread.

Even the trusty `console` object isn't available to workers. All a worker *can* see is its own global object, called `self`, and everything bundled with it: standard JavaScript objects like `setTimeout` and `Math`, plus the browser's Ajax methods.

Ah yes, Ajax! A worker can use `XMLHttpRequest` freely. It can even use `WebSocket` if the browser supports it. That means the worker can pull data directly from the server. And if we're dealing with a lot of data (like, say, streaming video

2. http://webworkersandbox.com/
3. https://github.com/mbebenita/Broadway
4. http://crypto.stanford.edu/sjcl/
5. http://ace.ajax.org/

that needs to be decoded), keeping it in one thread rather than serializing it with postMessage is a big win.

There's also a special importScripts function that will (synchronously) load and run the given script(s).

```
importScripts('https://raw.github.com/gist/1962739/danika.js');
```

Normally, synchronous loading is a big no-no, but remember that we're in a secondary thread here. As long as the worker has nothing else to do, blocking is A-OK.

Which Browsers Support Web Workers?

On the desktop, the Web Worker standard has been implemented in Chrome, Firefox, and Safari for a couple of years, and it's in IE10. Mobile support is spotty as well. The latest iOS Safari supports them, but the latest Android browser doesn't. At the time of this writing, that translates into 59.12 percent browser support, according to Caniuse.com.[6]

In short, you can't count on your site's users having web workers. You can, however, easily write a shim to run the target script normally if window.Worker is unavailable. Web workers are just a performance enhancement after all.

Be careful to test web workers in multiple browsers because there are some critical differences among the implementations. For instance, Firefox allows workers to spawn their own "subworkers," but Chrome currently doesn't.

5.2 Node Workers with cluster

In the early days of Node, there were many competing APIs for multithreading. Most of these solutions were clumsy, requiring users to spin up multiple instances of a server to listen on different TCP ports, which would then be hooked up to the real one via proxy. It was only in the 0.6 release that a standard was included out of the box that allowed multiple processes to bind to the same port: cluster.[7]

Typically, cluster is used to spin up one process per CPU core for optimal performance (though whether each process will actually get its own core is entirely up to the underlying OS).

6. http://caniuse.com/webworkers
7. http://nodejs.org/docs/latest/api/cluster.html

Multithreading/cluster.js

```
var cluster = require('cluster');
if (cluster.isMaster) {
  // spin up workers
  var coreCount = require('os').cpus().length;
  for (var i = 0; i < coreCount; i++) {
    cluster.fork();
  }
  // bind death event
  cluster.on('death', function(worker) {
    console.log('Worker ' + worker.pid + ' has died');
  });
} else {
  // die immediately
  process.exit();
}
```

The output will look something like

```
Worker 15330 has died
Worker 15332 has died
Worker 15329 has died
Worker 15331 has died
```

with one line for each CPU core.

The code may look baffling at first. The trick is that while web workers load a separate script, cluster.fork() causes the *same* script that it's run from to be loaded in a separate process. The only way the script knows whether it's being run as the master or a worker is by checking cluster.isMaster.

The reason for this design decision is that multithreading in Node has a very different primary use case than multithreading in the browser. While the browser can relegate any surplus threads to background tasks, Node servers need to scale up the computational resources available for their main task: handling requests.

(External scripts can be run as separate processes using child_process.fork.[8] Its capabilities are largely identical to those of cluster.fork—in fact, cluster uses child_process under the hood—except that child process can't share TCP ports.)

Talking to Node Workers

As with web workers, cluster workers can communicate with the master process by sending message events, and vice versa. The API is slightly different, though.

8. http://nodejs.org/docs/latest/api/child_process.html

Multithreading/clusterMessage.js

```
var cluster = require('cluster');
if (cluster.isMaster) {
  // spin up workers
  var coreCount = require('os').cpus().length;
  for (var i = 0; i < coreCount; i++) {
    var worker = cluster.fork();
    worker.send('Hello, Worker!');
    worker.on('message', function(message) {
      if (message._queryId) return;
      console.log(message);
    });
  }
} else {
  process.send('Hello, main process!');
  process.on('message', function(message) {
    console.log(message);
  });
}
```

The output will look something like

```
Hello, main process!
Hello, main process!
Hello, Worker!
Hello, Worker!
Hello, main process!
Hello, Worker!
Hello, main process!
Hello, Worker!
```

where the order is unpredictable, because each thread is racing to `console.log` first. (You'll have to manually terminate the process with Ctrl+C.)

As with web workers, the API is symmetric, with a `send` call on one side triggering a `'message'` event on the other side. But notice that the argument to `send` (or rather, a serialized copy) is given directly by the `'message'` event, rather than being attached as the `data` property.

Notice the line

```
if (message._queryId) return;
```

in the master message handler? Node sometimes sends its own messages from the workers, which always look something like this:

```
{ cmd: 'online', _queryId: 1, _workerId: 1 }
```

It's safe to ignore these internal messages, but be aware that they're used to perform some important magic behind the scenes. Most notably, when

workers try to listen on a TCP port, Node uses internal messages to allow the port to be shared.

Restrictions on Node Workers

For the most part, cluster obeys the same rules as web workers: there's a master, and there are workers; they communicate via events with attached strings or serializable objects. However, while workers are obviously second-class citizens in the browser, Node's workers possess all the rights and privileges of the master except, notably, the following:

- The ability to shut down the application
- The ability to spawn more workers
- The ability to communicate with each other

This gives the master the burden of being a hub for all interthread communication. Fortunately, this inconvenience can be abstracted away with a library like Roly Fentanes' Clusterhub.[9]

In this section, we've seen how workers have become an integral part of Node, allowing a server to utilize multiple cores without running multiple application instances. Node's cluster API allows the same script to run concurrently, with one master process and any number of workers. To minimize the overhead of communication, shared state should be stored in an external database, such as Redis.

5.3 What We've Learned

It's early, but I'd say the future of multicore JavaScript is bright. For any application that can be split up into largely independent processes that need to talk to each other only periodically, workers are a winning solution for leveraging maximum CPU power. Distributed computing has never been more fun.

9. https://github.com/fent/clusterhub