You Don't Know JS Yet: Sync & Async - 2nd Edition

Chapter 5: Program Performance

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. **Please ignore this stuff.** |

This book so far has been all about how to leverage asynchrony patterns more effectively. But we haven't directly addressed why asynchrony really matters to JS. The most obvious explicit reason is **performance**.

For example, if you have two Ajax requests to make, and they're independent, but you need to wait on them both to finish before doing the next task, you have two options for modeling that interaction: serial and concurrent.

You could make the first request and wait to start the second request until the first finishes. Or, as we've seen both with promises and generators, you could make both requests "in parallel," and express the "gate" to wait on both of them before moving on.

Clearly, the latter is usually going to be more performant than the former. And better performance generally leads to better user experience.

It's even possible that asynchrony (interleaved concurrency) can improve just the perception of performance, even if the overall program still takes the same amount of time to complete. User perception of performance is every bit – if not more! – as important as actual measurable performance.

We want to now move beyond localized asynchrony patterns to talk about some bigger picture performance details at the program level.

Note: You may be wondering about micro-performance issues like if `a++` or `++a` is faster. We'll look at those sorts of performance details in the next chapter on "Benchmarking & Tuning."

Web Workers

If you have processing-intensive tasks but you don't want them to run on the main thread (which may slow down the browser/UI), you might have wished that JavaScript could operate in a multithreaded manner.

In Chapter 1, we talked in detail about how JavaScript is single threaded. And that's still true. But a single thread isn't the only way to organize the execution of your program.

Imagine splitting your program into two pieces, and running one of those pieces on the main UI thread, and running the other piece on an entirely separate thread.

What kinds of concerns would such an architecture bring up?

For one, you'd want to know if running on a separate thread meant that it ran in parallel (on systems with multiple CPUs/cores) such that a long-running process on that second thread would **not** block the main program thread. Otherwise, "virtual threading" wouldn't be of much benefit over what we already have in JS with async concurrency.

And you'd want to know if these two pieces of the program have access to the same shared scope/resources. If they do, then you have all the questions that multithreaded languages (Java, C++, etc.) deal with, such as needing cooperative or preemptive locking (mutexes, etc.). That's a lot of extra work, and shouldn't be undertaken lightly.

Alternatively, you'd want to know how these two pieces could "communicate" if they couldn't share scope/resources.

All these are great questions to consider as we explore a feature added to the web platform circa HTML5 called "Web Workers." This is a feature of the browser (aka host environment) and actually has almost nothing to do with the JS language itself. That is, JavaScript does not *currently* have any features that support threaded execution.

But an environment like your browser can easily provide multiple instances of the JavaScript engine, each on its own thread, and let you run a different program in each thread. Each of those separate threaded pieces of your program is called a "(Web) Worker." This type of parallelism is called "task parallelism," as the emphasis is on splitting up chunks of your program to run in parallel.

From your main JS program (or another Worker), you instantiate a Worker like so:

```
var w1 = new Worker( "http://some.url.1/mycoolworker.js" );
```

The URL should point to the location of a JS file (not an HTML page!) which is intended to be loaded into a Worker. The browser will then spin up a separate thread and let that file run as an independent program in that thread.

Note: The kind of Worker created with such a URL is called a "Dedicated Worker." But instead of providing a URL to an external file, you can also create an "Inline Worker" by providing a Blob URL (another HTML5 feature); essentially it's an inline file stored in a single (binary) value. However, Blobs are beyond the scope of what we'll discuss here.

Workers do not share any scope or resources with each other or the main program – that would bring all the nightmares of threaded programming to the forefront – but instead have a basic event messaging mechanism connecting them.

The `w1` Worker object is an event listener and trigger, which lets you subscribe to events sent by the Worker as well as send events to the Worker.

Here's how to listen for events (actually, the fixed `"message"` event):

```
w1.addEventListener( "message", function(evt){
    // evt.data
} );
```

And you can send the `"message"` event to the Worker:

```
w1.postMessage( "something cool to say" );
```

Inside the Worker, the messaging is totally symmetrical:

```
// "mycoolworker.js"

addEventListener( "message", function(evt){
    // evt.data
} );

postMessage( "a really cool reply" );
```

Notice that a dedicated Worker is in a one-to-one relationship with the program that created it. That is, the `"message"` event doesn't need any disambiguation here, because we're sure that it could only have come from this one-to-one relationship – either it came from the Worker or the main page.

Usually the main page application creates the Workers, but a Worker can instantiate its own child Worker(s) – known as subworkers – as necessary. Sometimes this is useful to delegate such details to a sort of "master" Worker that spawns other Workers to process parts of a task. Unfortunately, at the time of this writing, Chrome still does not support subworkers, while Firefox does.

To kill a Worker immediately from the program that created it, call `terminate()` on the Worker object (like `w1` in the previous snippets). Abruptly terminating a Worker thread does not give it any chance to finish up its work or clean up any resources. It's akin to you closing a browser tab to kill a page.

If you have two or more pages (or multiple tabs with the same page!) in the browser that try to create a Worker from the same file URL, those will actually end up as completely separate Workers. Shortly, we'll discuss a way to "share" a Worker.

Note: It may seem like a malicious or ignorant JS program could easily perform a denial-of-service attack on a system by spawning hundreds of Workers, seemingly each with their own thread. While it's true that it's somewhat of a guarantee that a Worker will end up on a separate thread, this guarantee is not unlimited. The system is free to decide how many actual threads/CPUs/cores it really wants to create. There's no way to predict or guarantee how many you'll have access to, though many people assume it's at least as many as the number of CPUs/cores available. I think the safest assumption is that there's at least one other thread besides the main UI thread, but that's about it.

Worker Environment

Inside the Worker, you do not have access to any of the main program's resources. That means you cannot access any of its global variables, nor can you access the page's DOM or other resources. Remember: it's a totally separate thread.

You can, however, perform network operations (Ajax, WebSockets) and set timers. Also, the Worker has access to its own copy of several important global variables/features, including `navigator`, `location`, `JSON`, and `applicationCache`.

You can also load extra JS scripts into your Worker, using `importScripts(..)`:

```
// inside the Worker
importScripts( "foo.js", "bar.js" );
```

These scripts are loaded synchronously, which means the `importScripts(..)` call will block the rest of the Worker's execution until the file(s) are finished loading and executing.

Note: There have also been some discussions about exposing the `<canvas>` API to Workers, which combined with having canvases be Transferables (see the "Data Transfer" section), would allow Workers to perform more sophisticated

off-thread graphics processing, which can be useful for high-performance gaming (WebGL) and other similar applications. Although this doesn't exist yet in any browsers, it's likely to happen in the near future.

What are some common uses for Web Workers?

- Processing intensive math calculations
- Sorting large data sets
- Data operations (compression, audio analysis, image pixel manipulations, etc.)
- High-traffic network communications

Data Transfer

You may notice a common characteristic of most of those uses, which is that they require a large amount of information to be transferred across the barrier between threads using the event mechanism, perhaps in both directions.

In the early days of Workers, serializing all data to a string value was the only option. In addition to the speed penalty of the two-way serializations, the other major negative was that the data was being copied, which meant a doubling of memory usage (and the subsequent churn of garbage collection).

Thankfully, we now have a few better options.

If you pass an object, a so-called "Structured Cloning Algorithm" (https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm) is used to copy/duplicate the object on the other side. This algorithm is fairly sophisticated and can even handle duplicating objects with circular references. The to-string/from-string performance penalty is not paid, but we still have duplication of memory using this approach. There is support for this in IE10 and above, as well as all the other major browsers.

An even better option, especially for larger data sets, is "Transferable Objects" (http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast). What happens is that the object's "ownership" is transferred, but the data itself is not moved. Once you transfer away an object to a Worker, it's empty or inaccessible in the originating location – that eliminates the hazards of threaded programming over a shared scope. Of course, transfer of ownership can go in both directions.

There really isn't much you need to do to opt into a Transferable Object; any data structure that implements the Transferable interface (https://developer.mozilla.org/en-US/docs/Web/API/Transferable) will automatically be transferred this way (support Firefox & Chrome).

For example, typed arrays like `Uint8Array` (see the *ES6 & Beyond* title of this series) are "Transferables." This is how you'd send a Transferable Object using `postMessage(..)`:

```
// ‘foo‘ is a ‘Uint8Array‘ for instance

postMessage( foo.buffer, [ foo.buffer ] );
```

The first parameter is the raw buffer and the second parameter is a list of what to transfer.

Browsers that don't support Transferable Objects simply degrade to structured cloning, which means performance reduction rather than outright feature breakage.

Shared Workers

If your site or app allows for loading multiple tabs of the same page (a common feature), you may very well want to reduce the resource usage of their system by preventing duplicate dedicated Workers; the most common limited resource in this respect is a socket network connection, as browsers limit the number of simultaneous connections to a single host. Of course, limiting multiple connections from a client also eases your server resource requirements.

In this case, creating a single centralized Worker that all the page instances of your site or app can *share* is quite useful.

That's called a `SharedWorker`, which you create like so (support for this is limited to Firefox and Chrome):

```
var w1 = new SharedWorker( "http://some.url.1/mycoolworker.js" );
```

Because a shared Worker can be connected to or from more than one program instance or page on your site, the Worker needs a way to know which program a message comes from. This unique identification is called a "port" – think network socket ports. So the calling program must use the `port` object of the Worker for communication:

```
w1.port.addEventListener( "message", handleMessages );

// ..

w1.port.postMessage( "something cool" );
```

Also, the port connection must be initialized, as:

```
w1.port.start();
```

Inside the shared Worker, an extra event must be handled: `"connect"`. This event provides the port `object` for that particular connection. The most convenient way to keep multiple connections separate is to use closure (see *Scope & Closures* title of this series) over the `port`, as shown next, with the event listening and transmitting for that connection defined inside the handler for the `"connect"` event:

```
// inside the shared Worker
addEventListener( "connect", function(evt){
    // the assigned port for this connection
    var port = evt.ports[0];

    port.addEventListener( "message", function(evt){
        // ..

        port.postMessage( .. );

        // ..
    } );

    // initialize the port connection
    port.start();
} );
```

Other than that difference, shared and dedicated Workers have the same capabilities and semantics.

Note: Shared Workers survive the termination of a port connection if other port connections are still alive, whereas dedicated Workers are terminated whenever the connection to their initiating program is terminated.

Polyfilling Web Workers

Web Workers are very attractive performance-wise for running JS programs in parallel. However, you may be in a position where your code needs to run in older browsers that lack support. Because Workers are an API and not a syntax, they can be polyfilled, to an extent.

If a browser doesn't support Workers, there's simply no way to fake multithreading from the performance perspective. Iframes are commonly thought of to provide a parallel environment, but in all modern browsers they actually run on the same thread as the main page, so they're not sufficient for faking parallelism.

As we detailed in Chapter 1, JS's asynchronicity (not parallelism) comes from the event loop queue, so you can force faked Workers to be asynchronous using timers (`setTimeout(..)`, etc.). Then you just need

to provide a polyfill for the Worker API. There are some listed here (https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#web-workers), but frankly none of them look great.

I've written a sketch of a polyfill for `Worker` here (https://gist.github.com/getify/1b26accb1a09aa53ad25). It's basic, but it should get the job done for simple `Worker` support, given that the two-way messaging works correctly as well as `"onerror"` handling. You could probably also extend it with more features, such as `terminate()` or faked Shared Workers, as you see fit.

Note: You can't fake synchronous blocking, so this polyfill just disallows use of `importScripts(..)`. Another option might have been to parse and transform the Worker's code (once Ajax loaded) to handle rewriting to some asynchronous form of an `importScripts(..)` polyfill, perhaps with a promise-aware interface.

SIMD

Single instruction, multiple data (SIMD) is a form of "data parallelism," as contrasted to "task parallelism" with Web Workers, because the emphasis is not really on program logic chunks being parallelized, but rather multiple bits of data being processed in parallel.

With SIMD, threads don't provide the parallelism. Instead, modern CPUs provide SIMD capability with "vectors" of numbers – think: type specialized arrays – as well as instructions that can operate in parallel across all the numbers; these are low-level operations leveraging instruction-level parallelism.

The effort to expose SIMD capability to JavaScript is primarily spearheaded by Intel (https://01.org/node/1495), namely by Mohammad Haghighat (at the time of this writing), in cooperation with Firefox and Chrome teams. SIMD is on an early standards track with a good chance of making it into a future revision of JavaScript, likely in the ES7 timeframe.

SIMD JavaScript proposes to expose short vector types and APIs to JS code, which on those SIMD-enabled systems would map the operations directly through to the CPU equivalents, with fallback to non-parallelized operation "shims" on non-SIMD systems.

The performance benefits for data-intensive applications (signal analysis, matrix operations on graphics, etc.) with such parallel math processing are quite obvious!

Early proposal forms of the SIMD API at the time of this writing look like this:

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

var v3 = SIMD.int32x4( 10, 101, 1001, 10001 );
```

```
var v4 = SIMD.int32x4( 10, 20, 30, 40 );

SIMD.float32x4.mul( v1, v2 );   // [ 6.597339, 67.2, 138.89, 299.97 ]
SIMD.int32x4.add( v3, v4 );     // [ 20, 121, 1031, 10041 ]
```

Shown here are two different vector data types, 32-bit floating-point numbers and 32-bit integer numbers. You can see that these vectors are sized exactly to four 32-bit elements, as this matches the SIMD vector sizes (128-bit) available in most modern CPUs. It's also possible we may see an `x8` (or larger!) version of these APIs in the future.

Besides `mul()` and `add()`, many other operations are likely to be included, such as `sub()`, `div()`, `abs()`, `neg()`, `sqrt()`, `reciprocal()`, `reciprocalSqrt()` (arithmetic), `shuffle()` (rearrange vector elements), `and()`, `or()`, `xor()`, `not()` (logical), `equal()`, `greaterThan()`, `lessThan()` (comparison), `shiftLeft()`, `shiftRightLogical()`, `shiftRightArithmetic()` (shifts), `fromFloat32x4()`, and `fromInt32x4()` (conversions).

Note: There's an official "prollyfill" (hopeful, expectant, future-leaning polyfill) for the SIMD functionality available (https://github.com/johnmccutchan/ecmascript_simd), which illustrates a lot more of the planned SIMD capability than we've illustrated in this section.

asm.js

"asm.js" (http://asmjs.org/) is a label for a highly optimizable subset of the JavaScript language. By carefully avoiding certain mechanisms and patterns that are *hard* to optimize (garbage collection, coercion, etc.), asm.js-styled code can be recognized by the JS engine and given special attention with aggressive low-level optimizations.

Distinct from other program performance mechanisms discussed in this chapter, asm.js isn't necessarily something that needs to be adopted into the JS language specification. There *is* an asm.js specification (http://asmjs.org/spec/latest/), but it's mostly for tracking an agreed upon set of candidate inferences for optimization rather than a set of requirements of JS engines.

There's not currently any new syntax being proposed. Instead, asm.js suggests ways to recognize existing standard JS syntax that conforms to the rules of asm.js and let engines implement their own optimizations accordingly.

There's been some disagreement between browser vendors over exactly how asm.js should be activated in a program. Early versions of the asm.js experiment required a `"use asm";` pragma (similar to strict mode's `"use strict";`) to help clue the JS engine to be looking for asm.js optimization opportunities and hints. Others have asserted that asm.js should just be a set of heuristics that engines automatically recognize without the author having to do anything extra,

meaning that existing programs could theoretically benefit from asm.js-style optimizations without doing anything special.

How to Optimize with asm.js

The first thing to understand about asm.js optimizations is around types and coercion (see the *Types & Grammar* title of this series). If the JS engine has to track multiple different types of values in a variable through various operations, so that it can handle coercions between types as necessary, that's a lot of extra work that keeps the program optimization suboptimal.

Note: We're going to use asm.js-style code here for illustration purposes, but be aware that it's not commonly expected that you'll author such code by hand. asm.js is more intended to a compilation target from other tools, such as Emscripten (https://github.com/kripken/emscripten/wiki). It's of course possible to write your own asm.js code, but that's usually a bad idea because the code is very low level and managing it can be very time consuming and error prone. Nevertheless, there may be cases where you'd want to hand tweak your code for asm.js optimization purposes.

There are some "tricks" you can use to hint to an asm.js-aware JS engine what the intended type is for variables/operations, so that it can skip these coercion tracking steps.

For example:

```
var a = 42;

// ..

var b = a;
```

In that program, the `b = a` assignment leaves the door open for type divergence in variables. However, it could instead be written as:

```
var a = 42;

// ..

var b = a | 0;
```

Here, we've used the `|` ("binary OR") with value `0`, which has no effect on the value other than to make sure it's a 32-bit integer. That code run in a normal JS engine works just fine, but when run in an asm.js-aware JS engine it *can* signal that `b` should always be treated as a 32-bit integer, so the coercion tracking can be skipped.

Similarly, the addition operation between two variables can be restricted to a more performant integer addition (instead of floating point):

```
(a + b) | 0
```

Again, the asm.js-aware JS engine can see that hint and infer that the `+` operation should be 32-bit integer addition because the end result of the whole expression would automatically be 32-bit integer conformed anyway.

asm.js Modules

One of the biggest detractors to performance in JS is around memory allocation, garbage collection, and scope access. asm.js suggests one of the ways around these issues is to declare a more formalized asm.js "module" – do not confuse these with ES6 modules; see the *ES6 & Beyond* title of this series.

For an asm.js module, you need to explicitly pass in a tightly conformed namespace – this is referred to in the spec as `stdlib`, as it should represent standard libraries needed – to import necessary symbols, rather than just using globals via lexical scope. In the base case, the `window` object is an acceptable `stdlib` object for asm.js module purposes, but you could and perhaps should construct an even more restricted one.

You also must declare a "heap" – which is just a fancy term for a reserved spot in memory where variables can already be used without asking for more memory or releasing previously used memory – and pass that in, so that the asm.js module won't need to do anything that would cause memory churn; it can just use the pre-reserved space.

A "heap" is likely a typed `ArrayBuffer`, such as:

```
var heap = new ArrayBuffer( 0x10000 );  // 64k heap
```

Using that pre-reserved 64k of binary space, an asm.js module can store and retrieve values in that buffer without any memory allocation or garbage collection penalties. For example, the `heap` buffer could be used inside the module to back an array of 64-bit float values like this:

```
var arr = new Float64Array( heap );
```

OK, so let's make a quick, silly example of an asm.js-styled module to illustrate how these pieces fit together. We'll define a `foo(..)` that takes a start (`x`) and end (`y`) integer for a range, and calculates all the inner adjacent multiplications of the values in the range, and then finally averages those values together:

```
function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;

        // calculate all the inner adjacent multiplications
        for (i = x | 0;
            (i | 0) < (y | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            // store result
            arr[ p >> 3 ] = (i * (i + 1)) | 0;
        }

        // calculate average of all intermediate values
        for (i = 0, p = 0;
            (i | 0) < (count | 0);
            p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            sum = (sum + arr[ p >> 3 ]) | 0;
        }

        return +(sum / count);
    }

    return {
        foo: foo
    };
}

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 );      // 233
```

Note: This asm.js example is hand authored for illustration purposes, so it doesn't represent the same code that would be produced from a compilation tool

targeting asm.js. But it does show the typical nature of asm.js code, especially the type hinting and use of the `heap` buffer for temporary variable storage.

The first call to `fooASM(..)` is what sets up our asm.js module with its `heap` allocation. The result is a `foo(..)` function we can call as many times as necessary. Those `foo(..)` calls should be specially optimized by an asm.js-aware JS engine. Importantly, the preceding code is completely standard JS and would run just fine (without special optimization) in a non-asm.js engine.

Obviously, the nature of restrictions that make asm.js code so optimizable reduces the possible uses for such code significantly. asm.js won't necessarily be a general optimization set for any given JS program. Instead, it's intended to provide an optimized way of handling specialized tasks such as intensive math operations (e.g., those used in graphics processing for games).

Review

The first four chapters of this book are based on the premise that async coding patterns give you the ability to write more performant code, which is generally a very important improvement. But async behavior only gets you so far, because it's still fundamentally bound to a single event loop thread.

So in this chapter we've covered several program-level mechanisms for improving performance even further.

Web Workers let you run a JS file (aka program) in a separate thread using async events to message between the threads. They're wonderful for offloading long-running or resource-intensive tasks to a different thread, leaving the main UI thread more responsive.

SIMD proposes to map CPU-level parallel math operations to JavaScript APIs for high-performance data-parallel operations, like number processing on large data sets.

Finally, asm.js describes a small subset of JavaScript that avoids the hard-to-optimize parts of JS (like garbage collection and coercion) and lets the JS engine recognize and run such code through aggressive optimizations. asm.js could be hand authored, but that's extremely tedious and error prone, akin to hand authoring assembly language (hence the name). Instead, the main intent is that asm.js would be a good target for cross-compilation from other highly optimized program languages – for example, Emscripten (https://github.com/kripken/emscripten/wiki) transpiling C/C++ to JavaScript.

While not covered explicitly in this chapter, there are even more radical ideas under very early discussion for JavaScript, including approximations of direct threaded functionality (not just hidden behind data structure APIs). Whether that happens explicitly, or we just see more parallelism creep into JS behind the scenes, the future of more optimized program-level performance in JS looks really *promising*.