A cartoon intro to ArrayBuffers and SharedArrayBuffers

By Lin Clark

Posted on June 14, 2017 in A cartoon intro to SharedArrayBuffers, Code Cartoons, and JavaScript

This is the 2nd article in a 3-part series:

1. *A crash course in memory management*
2. *A cartoon intro to ArrayBuffers and SharedArrayBuffers*
3. *Avoiding race conditions in SharedArrayBuffers with Atomics*

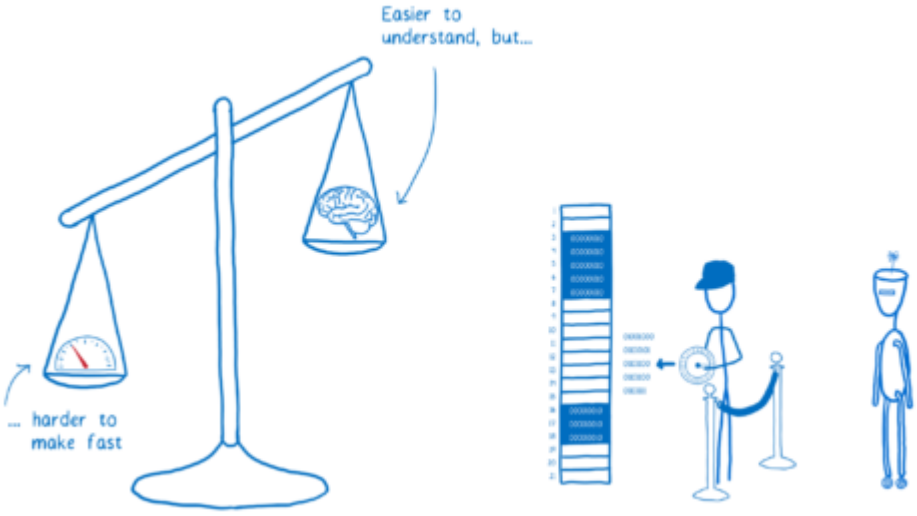
In the last article, I explained how memory-managed languages like JavaScript work with memory. I also explained how manual memory management works in languages like C.

Why is this important when we're talking about ArrayBuffers and SharedArrayBuffers?

It's because ArrayBuffers give you a way to handle some of your data manually, even though you're working in JavaScript, which has automatic memory management.

Why is this something that you would want to do?

As we talked about in the last article, there's a trade-off with automatic memory management. It is easier for the developer, but it adds some overhead. In some cases, this overhead can lead to performance problems.

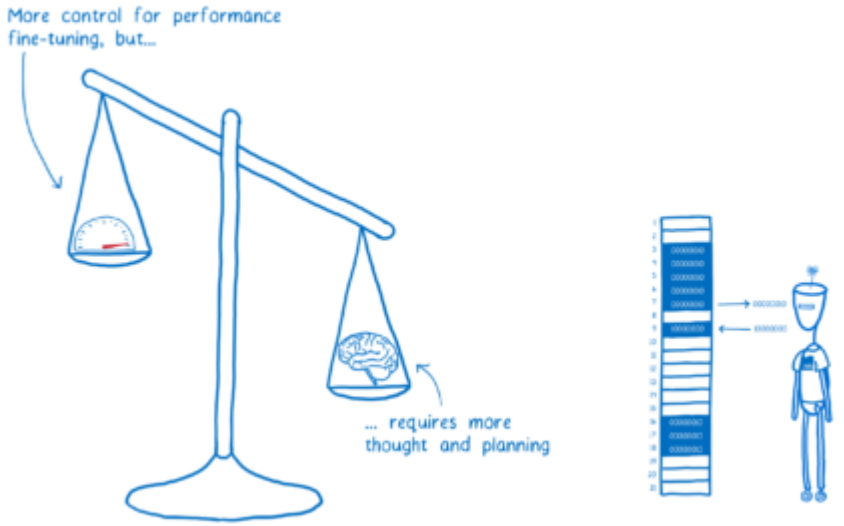


For example, when you create a variable in JS, the engine has to guess what kind of variable this is and how it should be represented in memory. Because it's guessing, the JS engine will usually reserve more space than it really needs for a variable. Depending on the variable, the memory slot may be 2–8 times larger than it needs to be, which can lead to lots of wasted memory.

Additionally, certain patterns of creating and using JS objects can make it harder to collect garbage. If you're doing manual memory management, you can choose an allocation and de-allocation strategy that's right for the use case that you're working on.

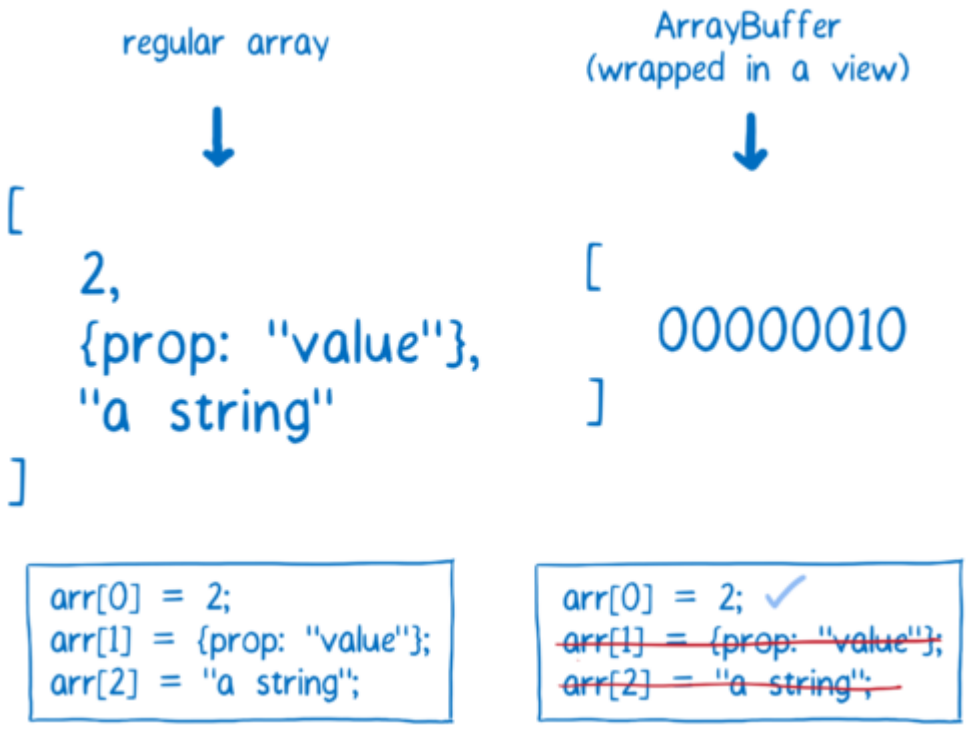
Most of the time, this isn't worth the trouble. Most use cases aren't so performance sensitive that you need to worry about manual memory management. And for common use cases, manual memory management may even be slower.

But for those times when you need to work at a low-level to make your code as fast as possible, ArrayBuffers and SharedArrayBuffers give you an option.



So how does an ArrayBuffer work?

It's basically like working with any other JavaScript array. Except, when using an ArrayBuffer, you can't put any JavaScript types into it, like objects or strings. The only thing that you can put into it are bytes (which you can represent using numbers).

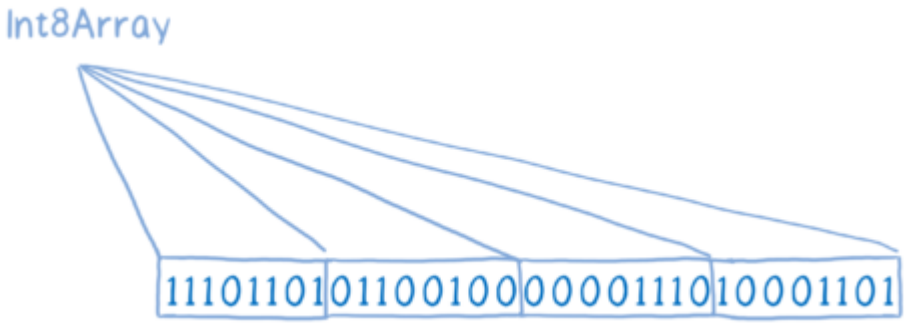


One thing I should make clear here is that you aren't actually adding this byte directly to the ArrayBuffer. By itself, this ArrayBuffer doesn't know how big the byte should be, or how different kinds of numbers should be converted to bytes.

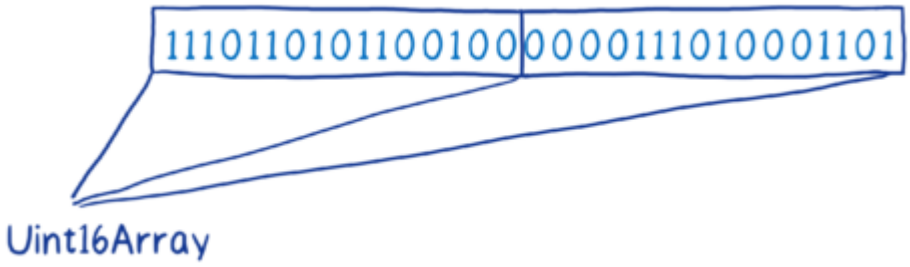
The ArrayBuffer itself is just a bunch of zeros and ones all in a line. The ArrayBuffer doesn't know where the division should be between the first element and the second element in this array.

To provide context, to actually break this up into boxes, we need to wrap it in what's called a view. These views on the data can be added with typed arrays, and there are lots of different kinds of typed arrays they can work with.

For example, you could have an Int8 typed array which would break this up into 8-bit bytes.

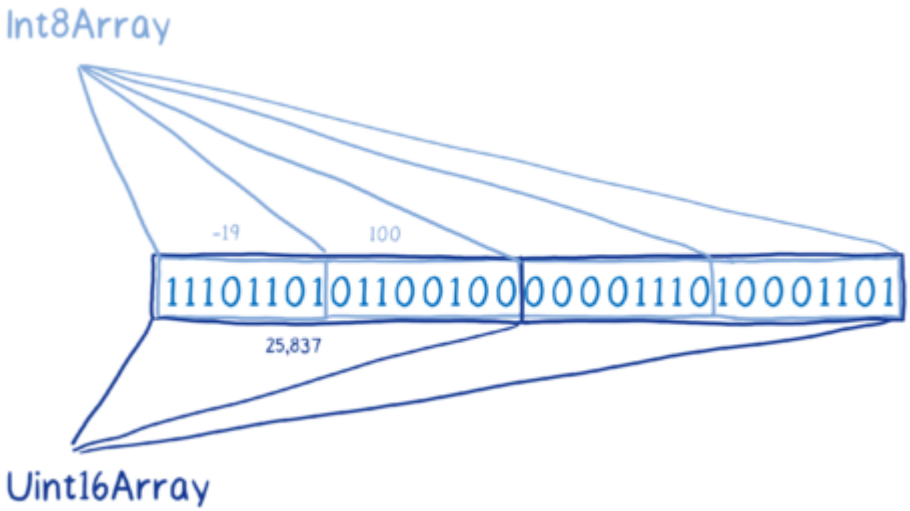


Or you could have an unsigned Int16 array, which would break it up into 16-bit bites, and also handle this as if it were an unsigned integer.



You can even have multiple views on the same base buffer. Different views will give you different results for the same operations.

For example, if we get elements 0 & 1 from the Int8 view on this ArrayBuffer, it will give us different values than element 0 in the Uint16 view, even though they contain exactly the same bits.



In this way, the ArrayBuffer basically acts like raw memory. It emulates the kind of direct memory access that you would have in a language like C.

You may be wondering why don't we just give programmers direct access to memory instead of adding this layer of abstraction. Giving direct access to memory would open up some security holes. I will explain more

about this in a future article.

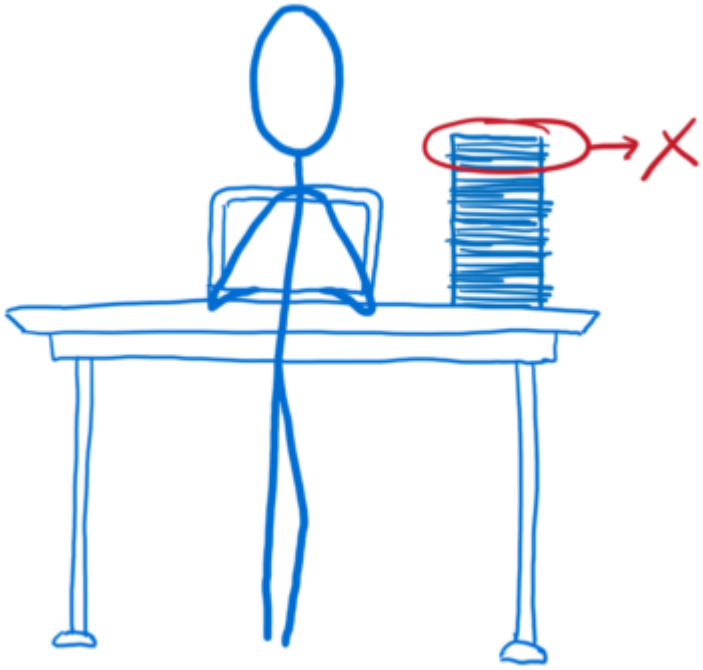
So, what is a SharedArrayBuffer?

To explain SharedArrayBuffers, I need to explain a little bit about running code in parallel and JavaScript.

You would run code in parallel to make your code run faster, or to make it respond faster to user events. To do this, you need to split up the work.

In a typical app, the work is all taken care of by a single individual—the main thread. I've talked about this before... the main thread is like a full-stack developer. It's in charge of JavaScript, the DOM, and layout.

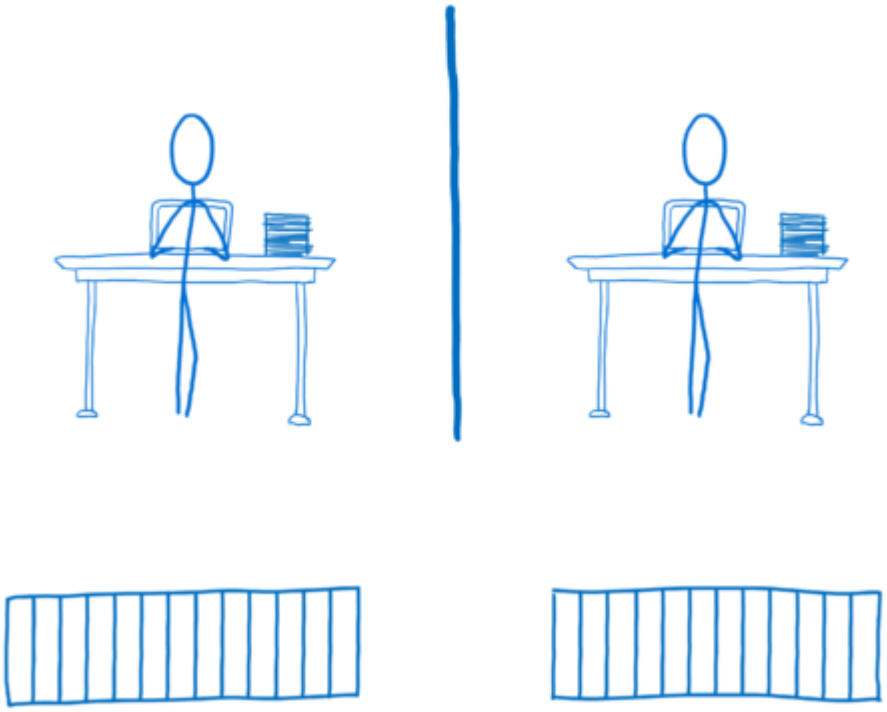
Anything you can do to remove work from the main thread's workload helps. And under certain circumstances, ArrayBuffers can reduce the amount of work that the main thread has to do.



But there are times when reducing the main thread's workload isn't enough. Sometimes you need to bring in reinforcements... you need to split up the work.

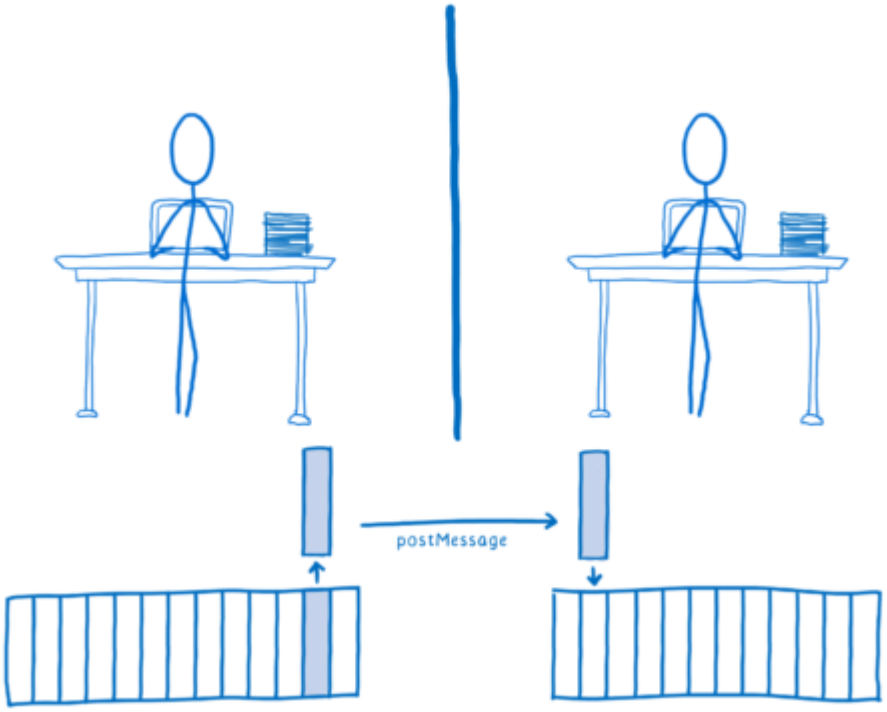
In most programming languages, the way you usually split up the work is by using something called a thread. This is basically like having multiple people working on a project. If you have tasks that are pretty independent of each other, you can give them to different threads. Then, both those threads can be working on their separate tasks at the same time.

In JavaScript, the way you do this is using something called a web worker. These web workers are slightly different than the threads you use in other languages. By default they don't share memory.



This means if you want to share some data with the other thread, you have to copy it over. This is done with the function postMessage.

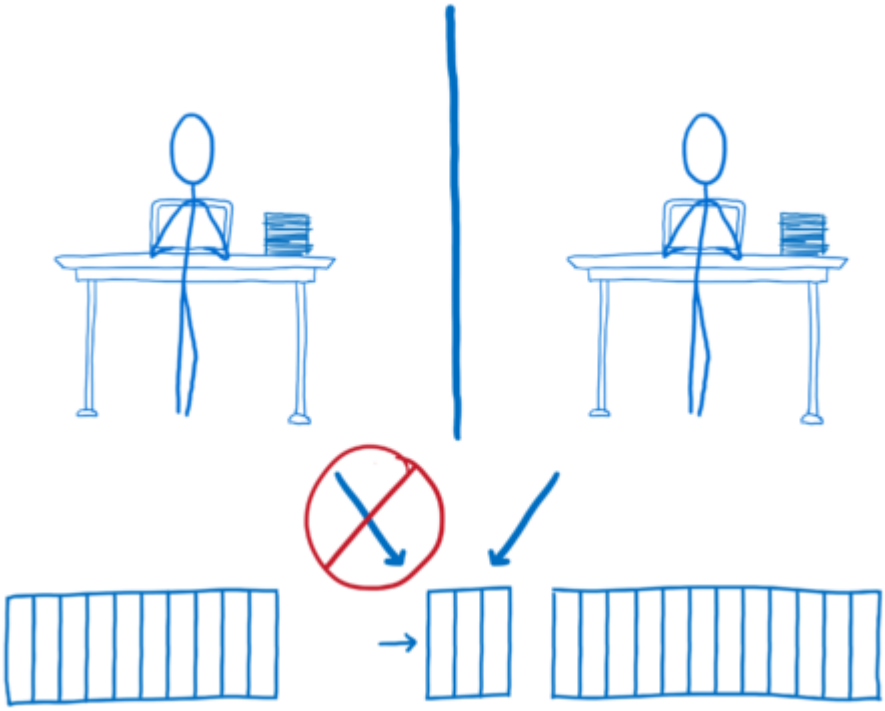
postMessage takes whatever object you put into it, serializes it, sends it over to the other web worker, where it's deserialized and put in memory.



That's a pretty slow process.

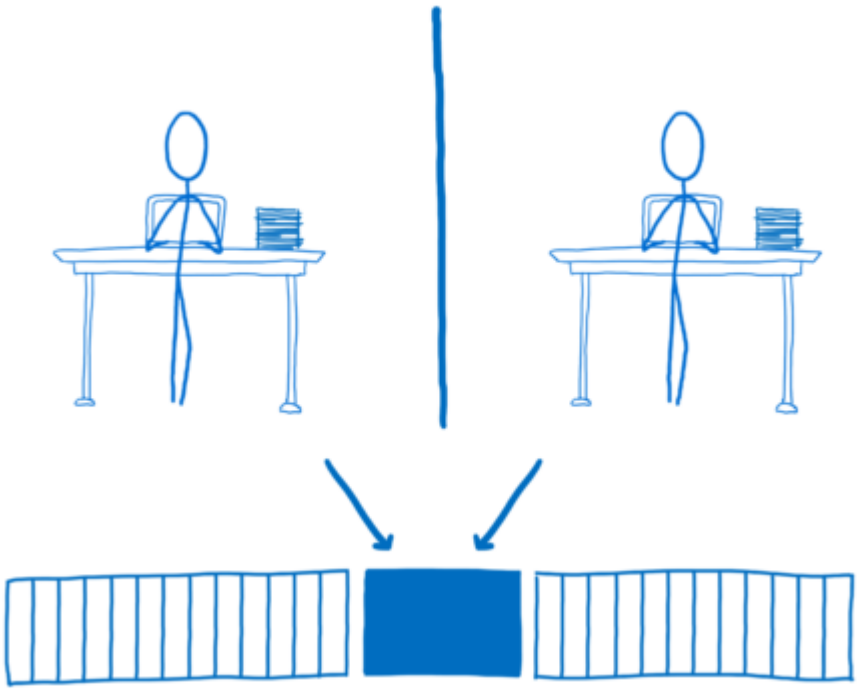
For some kinds of data, like ArrayBuffers, you can do what is called transferring memory. That means moving that specific block of memory over so that the other web worker has access to it.

But then the first web worker doesn't have access to it anymore.



That works for some use cases, but for many use cases where you want to have this kind of high performance parallelism, what you really need is to have shared memory.

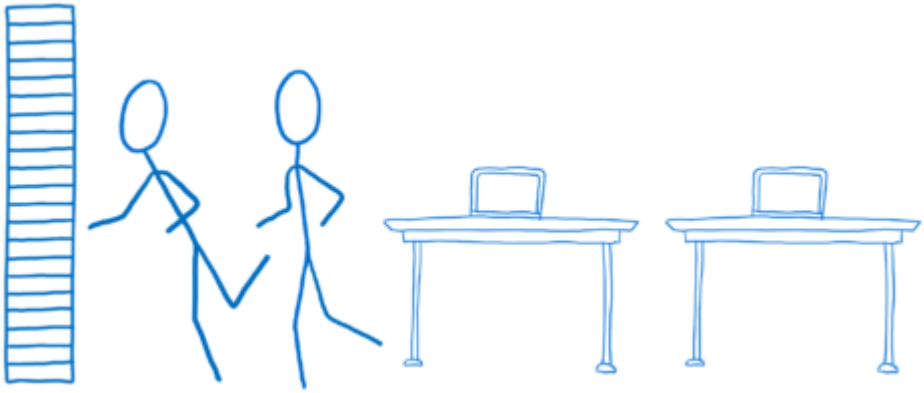
This is what SharedArrayBuffers give you.



With the SharedArrayBuffer, both web workers, both threads, can be writing data and reading data from the same chunk of memory.

This means they don't have the communication overhead and delays that you would have with postMessage. Both web workers have immediate access to the data.

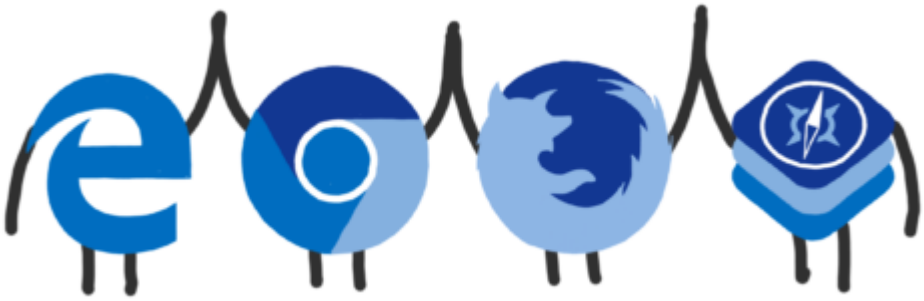
There is some danger in having this immediate access from both threads at the same time though. It can cause what are called race conditions.



I'll explain more about those in the next article.

What's the current status of SharedArrayBuffers?

SharedArrayBuffers will be in all of the major browsers soon.



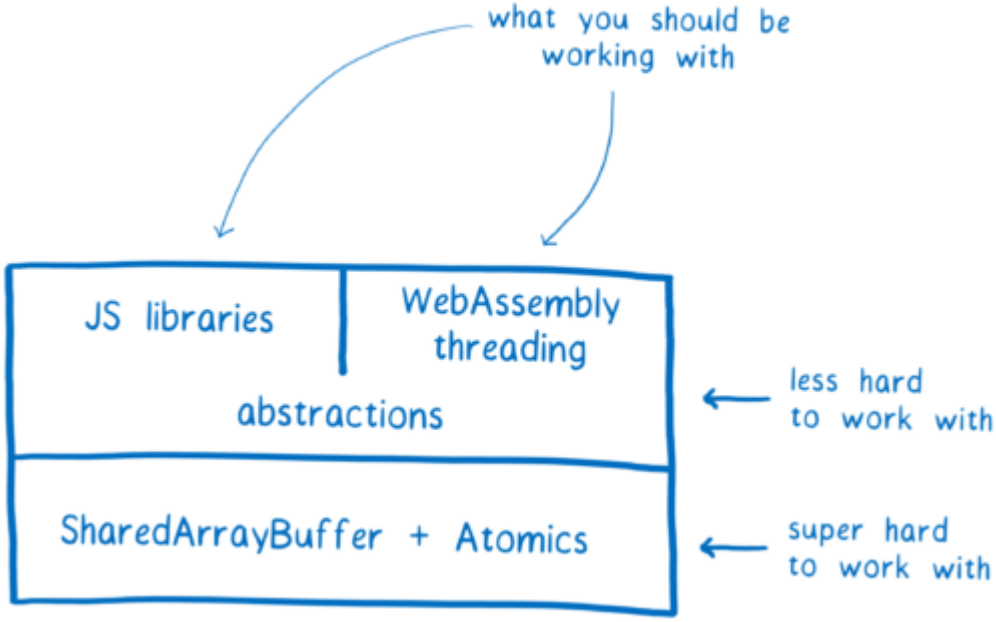
They've already shipped in Safari (in Safari 10.1). Both Firefox and Chrome will be shipping them in their July/August releases. And Edge plans to ship them in their fall Windows update.

Even once they are available in all major browsers, we don't expect application developers to be using them directly. In fact, we recommend against it. You should be using the highest level of abstraction available to you.

What we do expect is that JavaScript library developers will create libraries that give you easier and safer ways to work with SharedArrayBuffers.

In addition, once SharedArrayBuffers are built into the platform, WebAssembly can use them to implement support for threads. Once that's in place, you'd be able to use the concurrency abstractions of a language like Rust, which has fearless concurrency as one of its main goals.

In the next article, we'll look at the tools (Atomics) that these library authors would use to build up these abstractions while avoiding race conditions.



About Lin Clark

Lin is an engineer on the Mozilla Developer Relations team. She tinkers with JavaScript, WebAssembly, Rust, and Servo, and also draws code cartoons.

code-cartoons.com
@linclark

More articles by Lin Clark...

3 comments

Julen

Thanks a lot for such amazing explanation Lin.
Looking forward to be able to use it in modern Web Apps

June 16th, 2017 at 17:09

alican krlr

thanks for the articles! i just came across these on the main page of mozilla hacks just after looking up multithreaded node.js discussions (e.g. why is node.js single threaded etc) and webassembly on stack overflow so this article made a lot of sense.
https://softwareengineering.stackexchange.com/questions/315454/what-are-the-drawbacks-of-making-a-multi-threaded-javascript-runtime-implementat
https://softwareengineeringdaily.com/2015/08/02/how-does-node-js-work-asynchronously-without-multithreading/
https://stackoverflow.com/questions/40028377/is-it-possible-to-achieve-multithreading-in-nodejs
https://stackoverflow.com/questions/17959663/why-is-node-js-single-threaded

June 19th, 2017 at 06:00

Andrey Melikhov

Hi! Russian version is done :)
https://medium.com/devschacht/a-cartoon-intro-to-arraybuffers-and-sharedarraybuffers-952198b0a1c9

June 26th, 2017 at 03:23

Comments are closed for this article.