Synchronic

Rough draft
13 May 2015 / lhansen@mozilla.com

The idea of Synchronic is to lift the API slightly from Atomics + futexes [4] to the level of simple event (scalar value) inter-thread communication. There is a C++ draft proposal [1] as well as C++ code that implements a larger API than the draft proposal [2]. Comments from the C++ community on the Atomics spec indicate that a Synchronic proposal is regarded as a good idea.

A synchronic<T> holds one value of primitive type T, for Javascript that would currently be one of the shared-memory types Int8, Uint8, ..., Float64. A write into a synchronic updates the value and sends a signal. Threads can listen for that signal by waiting on the synchronic to hold a given value or to hold something other than a given value. To take the simplest example, consider *sync* to be a synchronic<Int32> and assume a C-like API. In the following example, thread 2 is sending a flag to thread 1, which is waiting for it:

```
Thread 1                                        Thread 2

// sync has the value 0
synchronic_load_when_not_equal(sync, 0)    ...
// thread 1 waits
                                                synchronic_store(sync, 1)
// thread 1 wakes
```

There's a suite of operations on a synchronic that closely mimics the suite of operations on Atomics (load, store, compareExchange, exchange, add, sub, and, or, xor) in addition to several operations for waiting on a synchronic (loadWhenEqual, loadWhenNotEqual, expectUpdate). Operations that update the synchronic will wake threads that are waiting on the synchronic, as appropriate.

Discussion

Synchronics have two advantages over Atomics + futexes:

(1) The API is easier to use correctly for sending and receiving events.
(2) A native implementation of wait/wake can perform better than a portable implementation based on Atomics + futexes. Chiefly this is because low-level platform-dependent details (such as relaxing the memory system inside a spinloop, how to spin, how long to spin, and what kinds of memory ordering directives to use internally in the operations) can be hidden inside the native implementation. See below for more.

Synchronics *by themselves* also have some disadvantages over Atomics + futexes:

(1) Some advanced optimization opportunities are lost, such as futexWakeOrRequeue.

(2) Individual cells in shared arrays can no longer be updated atomically, instead they must be incorporated into Synchronic objects/values/locations, and the update of a Synchronic always entails a check to see whether a waiter should be woken.
(3) Initialization of a Synchronic is conceptually complex, and thus a poor fit for translating eg Pthreads code from C++, as emscripten would do.

Thus there are several important questions:

(1) Is the synchronic API complementary to, or does it replace, the Atomics API?
(2) If complementary to the Atomics API, does it need to be usable from asm.js?
(3) If complementary to the Atomics API, does it need to be in v1.0 of the Atomics spec, or indeed, does it need to be in that spec at all? Or is it a separate spec?
(4) Are there alternatives to Synchronic that add significant value without the complexity?

Below are two strawman APIs. The first is JS-style, using worker-local JS objects to provide methods and state for manipulating the underlying shared storage. The second is Atomics-style, exposing the storage as shared array objects and indices into those. The latter is (largely) usable from asm.js, the former is not.

It's pretty clear that the first API can be provided as a layer on top of the second. It's also pretty clear that the first API is the cleaner API for JS programmers. And though it may not be obvious, either library can be provided at the user level, on top of Atomics+futexes [4].

I think that the answers to my first three questions are:

(1) Synchronic and Atomics are complementary.
(2) Synchronic should be usable from asm.js (because asm.js can benefit from the performance gains and because we can wrap a simple API around an asm.js compatible API and expose it to JS programmers)
(3) I think we should draft a separate spec for Synchronic and that we should provide a polyfill/library on top of Atomics + futexes for the time being. I've written such a library [3].

In regard to alternatives to Synchronic: Reference [1] goes into some detail about why the Synchronic abstraction is good and makes the point that strategy and tuning of the implementation are platform dependent and beyond the capability of most programmers. I believe that is true. However, the Web platform is more restrictive (uniform) than the C++ platform, and could it be that there are intermediate-level solutions that provide much of the value of Synchronic without providing the complexity? What would they look like?

On architectures I'm familiar with it's probably true that we could expose a primitive that provides a wait-for-update-with-timeout function that has a fixed small upper bound on the timeout and another primitive that provides a signal-a-new-value function. At the same time, it seems that these would be awkward to handle, requiring parameters for working storage and perhaps using different strategies for

equality and atomicity depending on the data type. (These remarks are based on experience with a prototype.) It's not obvious to me that these primitives would have significantly lower complexity than the full Synchronic interface, which is really fairly simple.

So I think that the answer to my fourth question is:

(4) There's no obvious, substantially simpler mechanism than Synchronic for providing the functionality that Synchronic provides.

Strawman API 1: JS style

The main thing that is difficult about this API is storage management. The style I've adopted here has served me well in other contexts where I've been building abstractions on top of SAB+Atomics.

new SynchronicInt32(sab, offset) => *sync* object
(ditto for all other shared-memory types)

Return a new local JS object that represents a Synchronic Int32 mapped onto some consecutive bytes in the SharedArrayBuffer *sab*, starting at byte offset *offset*. (Int32 is known as the *base type* of the Synchronic.) The number of consecutive bytes reserved is given by SynchronicInt32.BYTES_PER_ELEMENT. Offset must be divisible by SynchronicInt32.BYTE_ALIGNMENT.

Multiple Synchronic objects of the same base type may be constructed on the same memory range (be it multiple objects within a single worker or scattered across several workers). Those constructor calls may proceed concurrently in separate agents.

The memory on which the Synchronic is constructed must be zero-filled (and those zero values must be visible to all the constructors as they are being called).

Apart from mapping multiple Synchronic objects of the same base type onto the same memory range, the bytes allotted to a Synchronic must not be used for any other purpose during the lifetime of any Synchronic object mapped onto that memory. The bytes allotted to a Synchronic must not be updated except through methods exposed by the Synchronic API.

Note: This could be defined as **new Synchronic(ta, index) => object**, where ta is some SharedTYPEArray. The BYTES_PER_ELEMENT and ALIGNMENT might then be exposed as TYPE_ELEMENTS and TYPE_ALIGNMENT on the Synchronic constructor (eg, INT32_ELEMENTS and INT32_ALIGNMENT). However, that that only makes sense so long as the space required for a synchronic<T> is a multiple of the size of T. For data up to float64 that's reasonable. For SIMD or struct data it probably is not reasonable.

Note: It should be straightforward to extend this type of constructor to SIMD data ("SynchronicFloat32x4", "SynchronicInt32x4") and even to TypedObject data (where a constructor for a synchronic for a struct type would be hanging off the object representing the struct type).

Note: The style could also be **new Synchronic.Int32(sab, offset)** -- to follow SIMD. I'm agnostic on this point. It might be a better fit, if we were to provide both API 1 and API 2.

***sync*.isLockFree() => value**
Return **true** if access to the cell is lock-free, otherwise **false**.

***sync*.load() => value**
Atomically load the value from the cell.

***sync*.store(v) => value**
Atomically update the cell with the value *v* and notify waiters. Return nothing.

***sync*.compareExchange(oldv, newv) => value**
Atomically compare-and-swap the cell's value: if the value is equal to *oldv* then store *newv*. Notify waiters if the value was updated. Return the old value in the cell. See below for notes on floating point equality.

***sync*.exchange(v) => value**
Atomically exchange the cell's value with *v* and notify waiters. Return the old value in the cell.

***sync*.add(v) => value**
***sync*.sub(v) => value**
Atomically add/subtract *v* into the cell and notify waiters. Return the old value in the cell.

***sync*.and(v) => value**
***sync*.or(v) => value**
***sync*.xor(v) => value**
Integer cells only. Atomically bitwise-operate v into the cell and notify waiters. Return the old value in the cell.

***sync*.loadWhenEqual(v) => value**
Wait until the value of the cell is observed to be *v*, then return the value in the cell (which may once again no longer be *v*, if there are concurrent writers). See below for notes on floating point equality.

***sync*.loadWhenNotEqual(v) => value**
Wait until the value of the cell is observed to be something other than *v*, then return the value in the cell (which may once again be *v*, if there are concurrent writers). See below for notes on floating point equality.

***sync*.expectUpdate(v, timeout) => void**
Wait until the value of the cell is observed to be something other than *v*, or *timeout* milliseconds have passed. See below for notes on floating point equality.

***sync*.notify() => void**

Wake all waiters and ask them to reevaluate their conditions.

Note: The utility of this method is slightly unclear to me, in this form. Working on figuring it out.

Floating point equality: For **compareExchange**, **loadWhenEqual**, **loadWhenNotEqual**, and **expectUpdate**, two floating point values a and b are equal if a===b or if both a and b are NaN and both NaNs have canonical representation. (This contrasts with Atomics.compareExchange, where +0 and -0 are considered different.)

It is unspecified what happens if the NaN in the cell has a noncanonical representation. Such a NaN can only be placed there through the use of aliasing and byte operations; such updating is expressly prohibited above.

Note: In principle I would like to strengthen this definition to give a definite meaning even to NaNs that have noncanonical representation, but such code is not expressible in JS using the Atomics API, and it does not seem important.

Strawman API 2: Atomics style

This API is modeled on the SIMD and Atomics APIs, but with the additional complication (discussed below) that the "view" argument in all cases is a SharedArrayBuffer and not actually a view, since Synchronic objects are not necessarily naturally aligned. (A Synchronic.float32x4 would not be a multiple of float32x4 in size.) I'm actually not sure how that plays with asm.js. Possibly the use of a byte array could be allowed in all cases.

It may be that I'm overthinking it, since for TypedObject structs asm.js would do its own layout, and we should stop here after SIMD, or even before SIMD.

Synchronic.Int8
Synchronic.Uint8
Synchronic.Int16
Synchronic.Uint16
Synchronic.Int32
Synchronic.Uint32
Synchronic.Float32
Synchronic.Float64

These are constant values that represent base types.

Note: There could easily be SIMD types too (Synchronic.float32x4, etc).

Note: How to fit TypedObject types into this is less clear, but one could imagine that a constructor T for a TypedObject has a "synchronic" object hanging off it, so the style would be

T.synchronic.load(...), and if one has a global name N for T then Synchronic.N=T.synchronic would polyfill the methods suitably.

Synchronic.Type.BYTES_PER_ELEMENT
Synchronic.Type.BYTE_ALIGNMENT

The number of bytes required for a synchronic cell for the Type, and the required byte alignment.

Note: The reason to use *bytes* as the unit is that for larger, future types (think SIMD data, or TypedObject structs) the number of bytes required for a synchronic<T> will be the size of T plus a small constant, not a multiple of the size of T.

Note: The alignment requirement is the size of the type for smaller types, but once we get to SIMD types (ie for TypedObject) it's probably more like malloc alignment in C.

Note: asm.js really needs these values to be exposed, to allow them to be baked into translated code. Unfortunately the best values are probably platform-dependent. Establishing a platform independent upper bound on the space and alignment would be a good first step towards fixing that.

Synchronic.Type.init(sab, offset)

Initialize a synchronic for a given Type. *Sab* is a SharedArrayBuffer, *offset* is a valid index into *sab*, and *offset*+Synchronic.Type.BYTES_PER_ELEMENT-1 is also a valid index into *sab*. *Offset* must be divisible by Synchronic.Type.BYTE_ALIGNMENT.

Constraints on zero-fill and alias-only-with-compatible-type are as for API 1.

Note: Though asm.js may in principle be able to use this kind of an API, it's not clear that code translated from C++ (say) can be translated to make use of it, since there may not be any obvious place to insert the init call -- initialization can be static.

Synchronic.Type.load(sab, offset)
Synchronic.Type.store(sab, offset, v)
Synchronic.Type.compareExchange(sab, offset, oldv, newv)
Synchronic.Type.exchange(sab, offset, v)
Synchronic.Type.add(sab, offset, v)
Synchronic.Type.sub(sab, offset, v)
Synchronic.Type.and(sab, offset, v)
Synchronic.Type.or(sab, offset, v)
Synchronic.Type.xor(sab, offset, v)
Synchronic.Type.loadWhenEqual(sab, offset, v)
Synchronic.Type.loadWhenNotEqual(sab, offset, v)
Synchronic.Type.expectUpdate(sab, offset, v, timeout)
Synchronic.Type.notify(sab, offset)

Functionally identical to the operations in API 1, with the constraint that the memory must have been initialized and the same *sab* and *offset* are used as were used during initialization.

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4195.pdf
[2] https://code.google.com/p/synchronic/
[3] https://github.com/lars-t-hansen/parlib-simple/tree/synchronic, look in src/synchronic.js and test/synchronic-shelltest.js
[4] Spec: JavaScript Shared Memory, Atomics, and Locks https://docs.google.com/document/d/1NDGA_gZJ7M7w1Bh8S0AoDyEqwDdRh4uSoTPSNn77PFk/edit?usp=sharing