Spec: JavaScript Shared Memory, Atomics, and Locks

Stable draft, September 30, 2014
~~Undergoing revision, July 3, 2015~~
OBSOLETE draft, July 9, 2015

Resolution, July 1, 2015:

Note of caution to any implementers: An attempt will be made to remove SharedTypedArray and replace it with TypedArray. Judging from experiments this attempt is likely to be successful.

Undergoing discussion, May 5, 2015:

Note of caution to any implementers: In preparation for taking this spec to TC39 there's vigorous debate internally at Mozilla about whether the TypedArray type hierarchy should be split, as the proposal has it, or if SharedTypedArrays should somehow be layered onto the existing TypedArray hierarchy (likely with SAB a subtype of AB and the SharedTypedArrays just going away). Consequently this draft should not presently be considered stable.

This document is open to commenting by anyone.
Alternatively, email dev-tech-js-engine@lists.mozilla.org or lhansen@mozilla.com.

Scope

We specify a shared-memory model and shared-memory object types for JavaScript; means of transmitting shared-memory objects among workers; primitives to operate atomically on shared-memory object elements; and primitives to suspend and wake up workers.

The motivation and requirements for this work are covered in a subsection of the Rationale.

Throughout this specification, the term "worker" means any agent that may perform an operation in question, be it some kind of worker thread or a main thread. Whether a main thread should be able to access shared memory, or should be able to perform a blocking wait, is a matter for a separate discussion.

Familiarity with the ES6 ArrayBuffer and TypedArray specifications (chapters 24.1 and 22.2 of the ES6 draft [ES6]) is assumed throughout this document. Reference is occasionally made to data types and algorithms described therein, and APIs described in the present document are largely intended to mirror those of ES6. In some cases the ES6 spec is significantly more precise than the present document; assume that the present document will be updated to that level of precision eventually.

Specification

The Shared Memory Model

The model for shared memory is a superset of that of C++11. In addition to what C++11 specifies, data races do not lead to undefined behavior[1] but merely to unspecified behavior in the general case, and to specified behavior in the following cases:[2]

[1] The meaning of "undefined" in C++ is "anything could happen".
[2] See the Rationale, Discussion, and Implementation notes for background and justification.

- If there is a read-write race on a location through integer arrays of the same element size, then the value read will be either the value that was in memory before the write, or the value that was written.
- If there is a write-write race on a location through integer arrays of the same element size, then the value that ends up in memory will be one of the two values written.

The *SharedArrayBuffer* Type

There is a global constructor *SharedArrayBuffer* that constructs primitive shared-memory objects.

SharedArrayBuffer(value)

1. If *value* is a SharedArrayBuffer then return *value*.
2. Throw a **TypeError** exception.

new SharedArrayBuffer(length)

1. Let n be the result of ToLength(*length*).[3]
2. Allocate a new shared memory area M of size n.
3. Tag M with a new, address-free, globally unique identifier G.
4. Tag M with a length property whose value is n.
5. Initialize the contents of M to zero.
6. Create o, a SharedArrayBuffer object:
    a. The internal [[SharedArrayBufferData]] property of o has value M.
7. Return o.

The identifier G is used by the futex mechanism, described later: it gives an identity to the shared memory area independent of its address, size, or origin. G being address-free, M will carry the identifier G even if M is shared between processes at different virtual addresses.

get SharedArrayBuffer.prototype.byteLength

1. Let O be the **this** object.
2. If O is not a SharedArrayBuffer object then throw a **TypeError** exception.
3. Let w be the length property of O's [[SharedArrayBufferData]] property.
4. Return w.

SharedArrayBuffer.isView(value)

1. If *value* is a SharedTypedArray object then return true.
2. Return false.

Unlike an ArrayBuffer, a SharedArrayBuffer cannot be neutered, and its [[SharedArrayBufferData]] property cannot be undefined.

[3] ToLength and ToNumber are defined by ES6; the result of the conversion is a nonnegative integer, or a TypeError.

The Shared Typed Array View Types

The views on SharedArrayBuffer are similar to but distinct from the views on ArrayBuffer. The view types are *SharedInt8Array*, *SharedUint8Array*, *SharedUint8ClampedArray*, *SharedInt16Array*, *SharedUint16Array*, *SharedInt32Array*, *SharedUint32Array*, *SharedFloat32Array*, and *SharedFloat64Array*.

In the following, substitute the concrete view type name for ***SharedTypedArray***.

SharedTypedArray(value)

1. If *value* is not a ***SharedTypedArray*** object then throw a **TypeError** exception.
2. Return *value*.

new SharedTypedArray(sab, [byteOffset, [length]]) *where sab is a SharedArrayBuffer*

1. Let B be the datum type indicated by the constructor name (Int8, etc).
2. Let k be the byte size of B: 1, 2, 4, or 8.
3. Let L be the value of *sab*'s **byteLength** property.
4. If offs be ToInteger(*byteOffset)*.
5. If offs < 0 then throw a **RangeError** exception.
6. If offs % k ≠ 0 then throw a **RangeError** exception.
7. If *length* is **undefined** then:
    a. If L % k ≠ 0 then throw a **RangeError** exception.
    b. If L < offs then throw a **RangeError** exception.
    c. Let len be (L-offs)/k.
8. Else:
    a. Let len be the result of ToLength(*length*).
    b. If offs + len×k > L then throw a **RangeError** exception.
9. Construct o, a ***SharedTypedArray*** object (of base type B) where:
    a. The internal [[ViewedSharedArrayBuffer]] property has value *sab*.
    b. The internal [[ByteLength]] property has value len×k.
    c. The internal [[ByteOffset]] property has value offs.
    d. The internal [[ArrayLength]] property has value len.
10. Return o.

new SharedTypedArray(length) *where the type of length is not Object*

1. Let B be the datum type indicated by the constructor name (Int8, etc).
2. Let k be the byte size of B: 1, 2, 4, or 8.
3. Let L be the result of ToLength(*length*).
4. Construct sab, a new SharedArrayBuffer of length k×L.
5. Construct o, a ***SharedTypedArray*** (of base type B) where:
    a. The internal [[ViewedSharedArrayBuffer]] property has value *sab*.
    b. The internal [[ByteLength]] property has value k×L.
    c. The internal [[ByteOffset]] property has value 0.

        d. The internal [[ArrayLength]] property has value L.
    6. Return o.

new <u>SharedTypedArray</u>(value) *in all other cases*
    1. Throw a **TypeError** exception.

The Shared Typed Array views have built-in, privileged, non-overridable single-element getters and setters for indexed properties in the manner of the Typed Array views. In the terminology of ES6 they are "*Integer indexed exotic objects*", see section 9.4.5 of the draft ES6 specification.

<u>SharedTypedArray</u>.BYTES_PER_ELEMENT
Holds the number of bytes used to represent an element of the base type for the array.

get <u>SharedTypedArray</u>.prototype.buffer
1. Let O be the **this** object.
2. If O is not of type *<u>SharedTypedArray</u>*, then throw a **TypeError** exception.
3. Let A be the value of O's [[ViewedSharedArrayBuffer]] property.
4. Return A.

get <u>SharedTypedArray</u>.prototype.byteLength
1. Let O be the **this** object.
2. If O is not of type *<u>SharedTypedArray</u>*, then throw a **TypeError** exception.
3. Let L be the value of O's [[ByteLength]] property.
4. Return L.

get <u>SharedTypedArray</u>.prototype.byteOffset
1. Let O be the **this** object.
2. If O is not of type *<u>SharedTypedArray</u>*, then throw a **TypeError** exception.
3. Let F be the value of O's [[ByteOffset]] property.
4. Return F.

<u>SharedTypedArray</u>.prototype.copyWithin(target, start [, end])

```
1. Let O be the this object.
2. If O is not of type SharedTypedArray, then throw a TypeError exception.
3. Let len be O's [[ArrayLength]] property.
4. Let relativeTarget be ToInteger(target)
5. If relativeTarget < 0 then
    a. Let to be max(0, len+relativeTarget)
    b. Else let to be min(relativeTarget, len)
6. Let relativeStart be ToInteger(start)
7. If relativeStart < 0 then
    a. Let from be max(0, len+relativeStart)
    b. Else let from be min(relativeStart, len)
```

8. If *end* is **undefined** then let *relativeEnd* be *len*, else let *relativeEnd* be ToInteger(*end*)
9. If *relativeEnd* < 0 then
    a. Let *final* be max(0, *len*+*relativeEnd*)
    b. Else let *final* be min(*relativeEnd*, *len*)
10. Let *count* be min(*final-from*, *len-to*)
11. If *from*<*to* and *to*<*from*+*count* then
    a. Let *direction*=-1
    b. Let *from*=*from*+*count*-1
    c. Let *to*=*to*+*count*-1
12. Else
    a. Let *direction*=1
13. While *count* > 0 do
    a. Let *x* be O[*from*]
    b. Store *x* in O[*to*]
    c. Let *from*=*from*+*direction*
    d. Let *to*=*to*+*direction*
    e. Let *count*=*count*-1
14. Return O

get SharedTypedArray.prototype.length

1. Let O be the **this** object.
2. If O is not of type *SharedTypedArray*, then throw a **TypeError** exception.
3. Let L be the value of O's [[ArrayLength]] property.
4. Return L.

SharedTypedArray.prototype.set(source [, offset])

1. Let O be the **this** object.
2. If O is not of type *SharedTypedArray*, then throw a **TypeError** exception.
3. If ***source*** is a TypedArray or Array, then follow the corresponding algorithm in the ES6 draft: sections 22.2.3.22.1 and 22.2.3.22.2. In the latter case *srcBuffer* and *targetBuffer* will never be the same object however.
4. If ***source*** is a SharedTypedArray, then follow the algorithm in the ES6 draft for TypedArray, section 22.2.3.22.2, except that if *srcBuffer* and *targetBuffer* are the same then do not literally clone *srcBuffer*, but copy *srcBuffer* into a fresh TypedArray of the appropriate type. The buffers *srcBuffer* and *targetBuffer* are the same if they carry the same global identifier G, ie, they may be different SharedArrayBuffer objects but they are referencing the same memory.

SharedTypedArray.prototype.subarray([begin [, end]])

1. Let O be the **this** object.
2. If O is not of type *SharedTypedArray*, then throw a **TypeError** exception.
3. Let *buffer* be the value of O's [[ViewedSharedArrayBuffer]] property.
4. Let *srcLength* be the value of O's internal [[ArrayLength]] property.
5. Let *beginInt* be ToInteger(***begin***)

6. If *beginInt* < 0 then let *beginInt* = *srcLength* + *beginInt*.
7. Let *beginIndex* be min(*srcLength*, max(0, *beginInt*))
8. If ***end*** is **undefined** then let ***end*** be *srcLength*.
9. Let *endInt* be ToInteger(***end***)
10. If *endInt* < 0 then let *endInt* = *srcLength* + *endInt*.
11. Let *endIndex* = max(0, min(*srcLength*, *endInt*))
12. If *endIndex* < *beginIndex* then let *endIndex* = *beginIndex*.
13. Let *newLength* = *endIndex* - *beginIndex*.
14. Let *elementSize* be the size of the element type of *SharedTypedArray*.
15. Let *srcByteOffset* be the value of O's internal [[ByteOffset]] property.
16. Let *beginByteOffset* be *srcByteOffset* + *beginIndex* × *elementSize*.
17. Construct a new *SharedTypedArray* with *buffer*, *beginByteOffset*, and *newLength*.

SharedTypedArray.prototype.BYTES_PER_ELEMENT

Holds the number of bytes used to represent an element of the base type for the array.

TODO: updates to existing methods (notably on TypedArray) to handle SharedTypedArray. TypedArray.set() comes to mind.

Transmitting and Cloning SharedArrayBuffers and Views

SharedArrayBuffers and SharedTypedArrays can be transmitted from one worker to another and are handled by the emerging HTML5 structured clone algorithm [Structured clone] as follows.

When a SharedArrayBuffer object is cloned the result is a new SharedArrayBuffer object that references the same shared memory area as the input object and has the same identifier G as the input object. In this case, the input object must be present in the SharedArrayBuffer object in the transfer map for the clone.

When the SharedArrayBuffer object is cloned the input object is *not* neutered, *nor* is a copy made of the memory. The memory area remains accessible as long as at least one SharedArrayBuffer object references it.

When a SharedTypedArray object is cloned the result is a new SharedTypedArray object of the same element type. The result object references a new SharedArrayBuffer object from its internal [[ViewedSharedArrayBuffer]] field. That object is a result of cloning the input object's SharedTypedArray's SharedArrayBuffer object. In this case, the input object's "buffer" value must be present in the transfer map for the clone.

If an attempt is made to clone a SharedArrayBuffer without that object being present in the transfer list a standard clone error is thrown.

Atomic Operations on SharedTypedArray Elements

There is a global object called *Atomics*, containing methods that implement atomic operations and constants representing status values.

A step marked "Atomically do:" is known as an *atomic step*. There must be a single program-wide total order in which atomic steps occur[4], known as the *atomics order*. If an atomic step *R* reads bytes *x–y* in a SharedArrayBuffer, for each byte *B* it may return any of:

- The most recent (in the atomics order) value written by an atomic step to *B*.
- If a non-atomic step *S* writes to B, and S happens-before *R*, and there is no other step *T* that writes to *B* such that *S* happens-before *T* and *T* happens-before *R*, then the value that *S* wrote to *B*.
- If a non-atomic step *U* writes to *B* and neither *U* happens-before *R* nor *R* happens-before *U*, then the value *U* writes to *B*.

Additionally, if an atomic step *Q* occurs before *R* in the atomics order and writes exactly bytes *x–y*, then this creates an edge from *Q* to *R* in the happens-before order.

Notes:

- The fact that all Atomics accesses occur through SharedTypedArrays instead of SharedDataViews or directly to the SharedArrayBuffer guarantees that accesses will be aligned.
- If two atomic steps access the same elements of a SharedArrayBuffer through SharedTypedArrays with different element sizes, they do not induce a happens-before edge, but they do still occur in the global atomics order.
- Atomic steps do not have to be through the same type SharedTypedArray to induce a happens-before edge.

Atomics.compareExchange(sta, index, oldvalue, newvalue)[5] [6]

1. If *sta* is not an integer-valued SharedTypedArray then throw a **TypeError** exception.
2. Let B be the base type of *sta*.
3. Let x be the result of ToLength(*index*).
4. Let o be the result of coercing *oldvalue* to B.
5. Let n be the result of coercing *newvalue* to B.
6. Let A be the value of *sta*'s **length** property.
7. If x ≥ A then:
    a. Throw a RangeError exception
8. Atomically do:
    a. Let w be the result of reading *sta[x]*.
    b. If w is bitwise equal to o then store n in *sta[x]*.
9. Return w.

Atomics.load(sta, index)

1. If *sta* is not an integer-valued SharedTypedArray then throw a **TypeError** exception.

[4] Thus, data-race free programs are sequentially consistent. Other consistency models may be added in later versions of this facility, perhaps via new methods on the Atomics object that allow the model to be specified.
[5] The atomic operations are specified such that early returns for out-of-range accesses will execute a memory barrier even though the array is not updated, since the barrier has the effect of flushing other stores as well.
[6] Weak compareExchange is ignored on purpose.

2. Let x be the result of ToLength(*index*).
3. Let A be the value of *sta*'s **length** property.
4. If x ≥ A then:
    a. Throw a RangeError exception
5. Atomically do:
    a. Let w be the result of reading *sta[x]*.
6. Return w.

Atomics.store(sta, index, value)

1. If *sta* is not an integer-valued SharedTypedArray then throw a **TypeError** exception.
2. Let B be the base type of *sta*.
3. Let x be the result of ToLength(*index*).
4. Let v be the result of coercing *value* to *B*.
5. Let A be the value of *sta*'s **length** property.
6. If x ≥ A then:
    a. Throw a RangeError exception
7. Atomically do:
    a. Store v in *sta[x]*.
8. Return *value*.[7]

Atomics.add(ia, index, value)

1. If *ia* is not an integer-valued SharedTypedArray then throw a **TypeError** exception.
2. Let B be the base integer type of *ia*.
3. Let x be the result of ToLength(*index*).
4. Let v be the result of coercing *value* to *B*.
5. Let A be the value of *ia*'s **length** property.
6. If x ≥ A then:
    a. Throw a RangeError exception
7. Atomically do:
    a. Let w be the result of reading *ia[x]*.
    b. Let r be the integer sum of w and v.
    c. Let z be the result of coercing r to B.
    d. Store z in *ia[x]*.
8. Return w.

Atomics.sub(ia, index, value)

As for *Atomics.add*, but in step 7(b) the result r is the integer difference of w and v.

Atomics.and(ia, index, value)

As for *Atomics.add*, but in step 7(b) the result r is the integer bitwise "and" of w and v.

[7] Analogous to ia[index] = value, which evaluates to value, not value coerced to the element type of ia.

Atomics.or(ia, index, value)

As for *Atomics.add*, but in step 7(b) the result r is the integer bitwise "or" of w and v.

Atomics.xor(ia, index, value)

As for *Atomics.add*, but in step 7(b) the result r is the integer bitwise "xor" of w and v.

Atomics.exchange(ia, index, value)

As for *Atomics.add*, but in step 7(b) the result r is the value v.

Atomics.isLockFree(size)

If *size* is not the initial value of the BYTES_PER_ELEMENT property of some SharedTypedArray then return **false**. Otherwise: If an atomic operation (*compareExchange*, *load*, *store*, *add*, *sub*, *and*, *or*, *xor*, or *exchange*) on a datum of *size* bytes will be performed in a lock-free manner in the calling execution environment [WhatWG script env] then return **true**, otherwise return **false**.

Implementations may return **false** when they should have returned **true** but not vice versa.

If *Atomics.isLockFree(k)* returns *v* in some execution environment then every subsequent invocation of *Atomics.isLockFree(k)* in that environment must return *v*. Furthermore, any invocation of *Atomics.isLockFree(k)* in any other execution environment that can (assuming arbitrary code) currently reference or in the future come to reference any of the same shared memory that the first environment references must also return *v*.[8]

Atomic Operations for Worker Suspension and Wake-up

The *Atomics* object provides a primitive worker suspension mechanism based on the Linux futex mechanism. Futexes can be used along with the above atomic operations to create locks.

Atomics.futexWait(i32a, index, value, timeout)

1. If *i32a* is not a SharedInt32Array then throw a **TypeError** exception.
2. Let x be the result of ToLength(*index*).
3. Let v be the result of ToInt32(*value*).[9]
4. If *timeout* is not provided or is **undefined** then let t be +Infinity. Otherwise:
    a. Let q be the result of ToNumber(*timeout*)
    b. If q is NaN then let t be +Infinity, otherwise let t be max(0,q).
5. Let G be the unique ID of the value of the [[SharedArrayBufferData]] property of the SharedArrayBuffer on which *i32a* is constructed.
6. Let A be the value of *i32a*'s **length** property.

[8] The return value for a given argument is not necessarily a constant on a given piece of hardware, but also reflects implementation choices that can vary over time and across engines. The specification requires that if a SharedArrayBuffer can be communicated from one environment to another even in principle, for unrestricted environment types (web workers, service workers, ...) and unrestricted communication channels (postMessage or other mechanisms that might exist), then the two environments must agree on the values returned from *isLockFree*.

[9] ToInt32 and ToInteger are defined by ES6.

```
7. If x ≥ A then:
    a. Throw a RangeError exception
8. Atomically do:
    a. Let w be the result of reading i32a[x].
    b. If v does not equal w then return the original value of Atomics.NOTEQUAL.
    c. Record that the worker is waiting on (G,x), assigning it a wait ID that is the next available element from a monotonically increasing sequence of wait IDs for (G,x).
    d. Suspend the calling worker for up to t milliseconds.[10]
    e. (The worker wakes up. It may wake up either because the timeout expired or because it was woken explicitly, and not for other reasons. It does not wake up for any kind of event delivery. If there are system-internal ways in which the worker may spuriously awake it must suspend again, without re-checking the value of i32a[x], until the timeout expires or it is woken explicitly.)
    f. Un-record that the worker is waiting on (G,x).
9. If the suspension in 8(d) was broken by another worker with a call on Atomics.futexWake then return the original value of Atomics.OK.
10. Return the original value of Atomics.TIMEDOUT.
```

Atomics.futexWake(i32a, index, count)

```
1. If i32a is not a SharedInt32Array then throw a TypeError exception.
2. Let x be the result of ToLength(index).
3. Let c be the result of max(+0,ToInteger(count)).
4. Let G be the unique ID of the value of the [[SharedArrayBufferData]] property of the SharedArrayBuffer on which i32a is constructed.
5. Let A be the value of i32a's length property.
6. If x ≥ A then:
    a. Throw a RangeError exception
7. Let n = 0.
8. For all workers W waiting on (G,x), in increasing wait ID order while c > 0:
    a. Atomically do:
        i. Wake W
        ii. Subtract 1 from c and add 1 to n.
9. Return n.
```

Atomics.futexWakeOrRequeue(i32a, index1, count, value, index2)

```
1. If i32a is not a SharedInt32Array then throw a TypeError exception.
2. Let x1 be the result of ToLength(index1).
3. Let c be the result of max(+0,ToInteger(count)).
4. Let v be the result of ToInt32(value).
5. Let x2 be the result of ToLength(index2).
```

[10] Note t may have a fractional part; 100us is expressed as 0.1ms.

6. Let G be the unique ID of the value of the [[SharedArrayBufferData]] property of the SharedArrayBuffer on which *i32a* is constructed.
7. Let A be the value of *i32a*'s **length** property.
8. If x1 ≥ A or x2 ≥ A then:
    a. Throw a RangeError exception
9. Let n = 0.
10. Atomically do:
    a. If *ia32*[x1] is not equal to v then return the original value of Atomics.NOTEQUAL.
    b. For all workers W waiting on (G,x1), in increasing wait ID order while c > 0:
        i. Wake W
        ii. Subtract 1 from c and add 1 to n.
    c. For all remaining workers W waiting on (G,x1), in increasing wait ID order:
        i. Assign W a wait ID that is the next available element from a monotonically increasing sequence of wait IDs for (G,x2).
        ii. Unrecord that W is waiting on (G,x1)
        iii. Record that W is waiting on (G,x2)
11. Return n.

Atomics.OK
The value 0.

Atomics.NOTEQUAL
The value -1.

Atomics.TIMEDOUT
The value -2.

Additional Worker Semantics

The happens-before relationship between workers is additionally specified as follows:

- The call to the Worker constructor in the parent happens-before the execution of the main script in the worker
- The termination of a worker happens-before the parent of the worker (or any other agent) can determine that the worker has terminated.[11]
- A postMessage to a worker happens-before the event fires in the worker

If a worker is terminated by a call to its **terminate** method while it is blocked in a call to **futexWait** then the worker is first woken (removed from any wait queue) and then immediately terminated (the wakeup is not observed by the caller of **futexWait**). There are no provisions for cleaning up locked state unless the browser supports a close handler.[12]

[11] There does however not seem to be any reliable way of determining, given a worker object, whether the worker represented by that object has terminated or not.
[12] Firefox has a provision for a close handler [Worker onclose]; other browsers may as well. It is not standard.

If a worker is terminated for any other reason, such as the user agent reloading or closing the window or frame, and the worker is blocked in a call to **futexWait** when it is terminated, then the worker is first woken (removed from any wait queue) and then immediately terminated (the wakeup is not observed by the caller of **futexWait**). However, after the running script has been terminated the worker's close handler(s) will be run as if the wait had not been aborted.

futexWait may be called from a worker's close handler.

The *Synchronic* API (informative)

The Synchronic API provides somewhat higher-level and better-behaved access to the functionality already exposed on the Atomics object. There is a draft specification of / discussion document for Synchronic here: Synchronic specification.

At the moment I see Synchronic as complementary to Atomics and I do not plan to include it in version 1 of this specification. In part this is because Synchronic imposes some overheads on storage and atomic operations relative to Atomics; in part it is because Synchronic (as currently designed) is a poor fit for asm.js (it is too high level and too abstract). I expect the asm.js issues will be overcome, but it seems probable that we still will want to keep the Atomics.

Rationale

Motivation, Requirements, and Evolution

These are the main use cases:

- Support for threaded code in programs written in other languages that are translated to asm.js or plain JS or a combination of the two, notably C++ but also JVM and CIL bytecode, Halide, Clojure, and similar languages.
- Support for hand-written JS or JS+asm.js that makes use of multiprocessing facilities for select tasks, such as image processing or game AI.

The shared memory facility needs to run on hardware that runs major web browsers now and in the future, and ideally on hardware that runs other JS applications such as node.js. For now this means current and future versions of x86, ARM, Power, and MIPS, in 32-bit and 64-bit configurations.

It's strongly desirable that the shared memory facility not require garbage collected shared storage in general. Shared memory areas backing SharedArrayBuffer do need to be reference counted, but we have not provided any "objects" sitting within the shared memory, eg for locks, that will need to be managed by the JS engine.

The path we're adopting is one where we provide fairly low-level primitives, and rely on the developer community to put together abstractions that are useful in specific domains. Abstractions that seem to be of general use may later be canonized and provided natively, when that is worthwhile.

It is probable that further evolution of this API will at least incorporate SIMD data and/or pointer-free TypedObject data in some form.

It is entirely open whether we will allow for non-sequentially-consistent methods (release-acquire or relaxed). Should we choose to do so it will likely be through additional methods, eg, Atomics.loadRelaxed(), and not through optional arguments to existing methods, on the assumption that encoding the memory model in the name makes it easier for the JIT to generate appropriate code reliably. (An argument can be made for the utility of parameterizing algorithms on the memory model, so it may be worth experimenting with what the JIT can be made to do. String values are mnemonic and reliably jittable, perhaps.)

The Shared Memory Model

Outlawing races as C++ does makes no sense for a safe language. All races must have benign, if not necessarily predictable, behavior. Some programs ported from C++ to asm.js will also depend on predictable behavior for some races.

There are (at least) four assumptions being made about the hardware in order to make non-atomic accesses safe:

- It is cache coherent.
- There is no hardware race detection (or if there is it can be disabled cheaply).
- The instruction set has load and store instructions for 1, 2, and 4 bytes (no RMW).
- The hardware is *single-copy atomic* (aka *access-atomic*) for same-sized normal loads and stores of aligned data, to and from integer registers, up to the native pointer size.

Single-copy atomicity is defined as follows:

- In a read-write race (of integer data of the same size to the same address) the read will see the old data or the written data, not a mixture nor garbage.
- In a write-write race (ditto) the data that ends up in memory is either of the words being written, not a mixture nor the old value nor garbage.

Sequential Consistency vs Weaker Models

At the moment all the Atomics methods provide only sequential consistency: all the primitives act as full memory barriers. In contrast, the C++ synchronization primitives optionally provide weaker consistency along two dimensions. The C++ "atomic_store" method is sequentially consistent while the "atomic_store_explicit" takes a memory model argument that allows weaker consistency to be specified. The "atomic_compare_exchange" family additionally offers "weak" and "strong" forms, where the "weak" form may fail even if the old value in the cell was the expected value (which may be beneficial on load-linked/store-conditional types of architectures); evidently the utility of this is debated even within the C++ community.

There are several reasons why the Atomics methods provide only sequential consistency:

- SC reduces the number of machine dependencies (see the Discussion)
- SC is generally easy to use and easy to agree on

- Absent a clear performance imperative, anything but SC is premature
- Other options can be added later, usually through new methods as in C++

The Duplication of the View Hierarchy

Duplication of the view hierarchy (requiring SharedInt8Array on SharedArrayBuffer instead of Int8Array on SharedArrayBuffer, and so on) is an implementation and usage burden, but the view types are distinct for safety and interface reasons:

Safety reasons: A view on shared memory does not provide the same memory semantics as a view on unshared memory and it is inevitably a source of race condition bugs if code that receives a view that it expects to be unshared is provided with memory that is shared and updated by another agent, or if code that updates memory does not do so in a critical section or with a memory barrier to ensure the consistency of the memory. In particular, security bugs in both user code and the browser may be possible, because shared memory introduces the possibility of "time of check to time of use" bugs in unaware code.

Interface reasons: A shared memory view does not provide the same interface as an unshared view; in particular, when a shared buffer is transmitted to another worker its views in the originating worker are not neutered.

We expect that individual DOM APIs will be adapted to take shared view types when appropriate, and that those APIs' specifications will evolve to include discussion of proper synchronization when shared memory is passed.

The Atomics Object vs Methods on SharedTypedArrays

We believe that an implementation is at least as likely to be able to in-line static methods on a global object than prototype methods on arbitrary objects; engines already have logic to do this for static method calls on the Math object. Most call sites to the Atomics primitives will be monomorphic in both argument types and the return type and the type of the receiver ("Atomics") will be known, if that matters (in SpiderMonkey it does not).

An additional concern has been that the static method-call style fits in better with asm.js than does a dynamic method-call style.

Futexes

Futexes come from Linux; they make it easy to support pthreads code ported to asm.js by translating the pthreads libraries to JS. Futexes require no intricate storage management, in particular no lock data need be deallocated at any point.

For shared-memory JS programming (if applicable) it will be natural to provide higher-level abstractions built on top of futexes.

Semantic Fine Points

The **compareExchange** primitive is sometimes expressed with two return values (C++) and sometimes with one (Java and most textbooks). There are two cases where this matters: On a LL/SC architecture where it is possible to distinguish between "could not update because the cell value was not as expected" and "could not update because somebody else had updated"; and when compareExchanging floating point values, where the use of representation equality in the primitive means that a naive equality test in the caller of **compareExchange** is not sufficient to properly handle +0/-0 and NaN. Still, the form that returns the old value seems more general so that's what we've gone with.[13]

Some Rejected Ideas

Early drafts had a **futexWaitCallback** mechanism, for asynchronous waiting. The purpose of that mechanism was to make it possible to forbid blocking on the main thread (and hence blocking the browser UI), while at the same time allowing the main thread to participate in shared-memory operations. However, the mechanism was seen as premature--the status of the main thread is still very much open to discussion--and was removed. Furthermore, what is really needed is a good building block for higher-level mechanisms, and **futexWaitCallback** does not seem to have advantages over what we already have. A barrier sync that does not block on the main thread can be built from mechanisms we already have [Asymmetric barrier].

Discussion

Machine dependencies

Data races are handled in architecture- and hardware-implementation-defined ways, and differences in the handling will be visible to a JS program that is racy, in the following ways:

- On x86 the hardware is single-copy atomic for a wide variety of data sizes and alignments. On most other architectures the hardware is single-copy atomic only for aligned accesses, and sometimes not for data sizes greater than four bytes, and sometimes not if the racing accesses are of different size, and sometimes not for floating-point data. Typically there may be a problem if a program assumes that a float64 is stored atomically; on a system with only four-byte single-copy atomicity a write-write race may end up storing a value that is a mix of the two values stored.
- The reservation size for an atomic access varies and contention resolution may or may not be fair (see the next section). On systems that use load-linked/store-conditional for atomic operations contention is probably not resolved fairly, and what appears in the program to be non-contending accesses to adjacent locations may in the worst case be highly contended and attempted concurrent atomic updates to those locations may block progress.
- A program written for a strongly ordered system such as x86 or SPARC-TSO will, in the absence of synchronizing operations on an explicit memory barrier, exhibit different multiprocessor

[13] While JS in a sense has support for multiple return values - consider functions that return arrays and callers that destructure those arrays - it's not something to rely on until engines support that cliche with guaranteed low cost. There are known techniques for this that might apply to JS [Multiple values].

behavior on a weakly ordered system such as ARM or Power, since loads and stores may be reordered aggressively on the latter.

Single-copy atomicity is also architecture dependent. We only have integer arrays up to four bytes wide at the moment, which is why the spec is written the way it is. Were we to add 8-byte wide arrays the spec would have to change because not all plausible hardware is single-copy atomic for eight-byte integer data: ARMv6 and ARMv7 are not (or implementations may not be). The spec would have to state that the hardware is single-copy atomic "for up to four bytes", or we'd introduce another machine dependency.

Machine dependencies give rise to modest fingerprinting concerns, but I'm not sure they can be used to get any new information. The user-agent string of the browser already contains the name of the architecture.

Software dependencies

The ordering of unsynchronized shared memory writes of one thread as observed from other threads is not fixed but execution strategies may make it appear that the ordering is fixed: interpretation or execution by "baseline" JIT code may make the observed order (on suitable hardware) correspond to program order, for example. An optimizing JIT may create code that does not produce that observed order, as it may reorder reads and writes and remove obstacles to hardware reordering of loads and stores (CPUs have varying rules for data and control dependencies that inhibit dynamic instruction reordering). Bailouts from JITted code for type or value guard failures or even garbage collection may introduce memory barriers where optimized code has none; thus code tested with one data set may behave differently with another data set.

Implementation and Hardware Notes

General guidance

Not all hardware supports atomic operations at every granularity we need. ARMv6 and MIPS32 do not provide 8-byte atomic operations, for example, and implementing the atomic operations on SharedFloat64Array on those platforms requires the use of other techniques. Usually it means resorting to an external lock (hidden within the implementation of the atomic operations).

For an external lock is concerned, there are several approaches:

- A global lock (though not a global lock per type)
- A lock per SharedArrayBuffer
- A lock per address range, in a sparse map of some sort

In the implementation, any atomic access from C++ (from an interpreter primitive, say) must use the same mechanism as an atomic access from jitted code. This is true for atomic accesses at all sizes but tends to work out because both the C++ compiler and the JIT use the obvious atomic instructions; for 64-bit accesses on platforms with no 64-bit atomics additional coordination is required.

If the hardware supports atomic operations at a granularity *greater* than what we need for a particular access then the atomic operation on the smaller size can be implemented using a read-modify-write sequence with the wider atomic operation.

Implementing compareExchange on a LL/SC architecture is not straightforward. The primitive may only fail if the value in the cell is different from the expected value. However, in the case of LL/SC, the update will fail if the cell was written even with the old value. Thus the implementation of compareExchange must retry the update if the update failed but the cell contains the old value.

A full barrier is not always needed before and/or after an operation that contains "atomically do". Notably there is useful advice in the JSR-133 cookbook [Cookbook] and on the Cambridge Relaxed-Memory Concurrency Group's page about C++ atomics [C++11 mappings]. For example, the load in an Atomics.load() will need no barriers before it but LoadLoad and LoadStore barriers after it.

asm.js mapping

Weakly ordered C++ atomic accesses can be mapped to the strongly ordered accesses defined above, at an occasional loss in performance.

Volatile C++ accesses, which are (arguably incorrectly) used for inter-thread communication in some x86 programs, must generally be mapped to Atomics.load and Atomics.store: JITs will usually hoist or common normal loads and stores. The mapping of volatiles to sequentially consistent atomics also hides the difference between weakly and strongly ordered hardware, which is possibly desirable in programs for the web. (If we later add support for relaxed atomics they may be used to implement volatile at lower cost.) A consequence of this is that volatiles are forced to be aligned, which they are not in C++, but this seems reasonable. PNaCl makes the same choice [PNaCl volatile].

C++ atomics on integral types (integer and char types) are defined to be initializable in the same way as the integral types, eg, with zero [C++11 atomics]. Our atomics can be used to implement C++ atomics for up to int32 size that have the same size as the underlying integral, which is useful (existing code may assume this). For other data types the translator can use techniques in the "General guidance" section.

C++ has additional templates, constants, and methods that are used to determine whether an atomic access will be implemented by a lock or by a lock-free primitive. Emscripten must expose this appropriately in its headers and must make up for the rest through code generation.

Some extensions to the asm.js spec are required to support shared views and inlined Atomics operations, these are in a companion document [asm.js additions].

x86 and x86_64

Discussion based on the Intel manual set [Intel].

Single-copy atomicity

The architecture is single-copy atomic in a large number of cases (volume 3A section 8.1.1): since the 486, for naturally aligned data up to four bytes; since the Pentium, for naturally aligned data up to eight bytes; since the P6, also for up to eight unaligned bytes as long as they are all within the same cache line; and since the Core 2 Duo, for up to eight unaligned bytes in general.

Note that SSE data, being 16 bytes, are not read or written atomically (to or from SSE registers) even when properly aligned.

Memory consistency model

The memory model is not quite sequentially consistent; store-to-load consistency, where a load following a store must wait for the store to complete, must be implemented by the program with an explicit fence. (If the load is to the word being stored then the load will read the stored value without the fence). The model is known as TSO, total store order. There is a good paper by the Cambridge Relaxed Memory Concurrency Group describing the memory model [Cam cacm].

The fence is an MFENCE instruction on SSE2-equipped systems and later. On earlier systems it is a locked memory no-op; Linux uses "LOCK; ADD [esp], 0".

Synchronization

Synchronization is via LOCK-prefixed instructions such as CMPXCHG, ADD, and XADD, or via instructions where the lock is implicit, such as XCHG. These can be applied in various ways to reduce register pressure, get rid of loops, etc. On 32-bit systems these instructions work on data up to four bytes; on 64-bit systems, on data up to eight bytes. Only integer registers can be targeted. CMPXCHG8B / CMPXCHG16B can be used to implement 8-byte compare-exchange on 32-bit systems and 16-byte compare-exchange on 64-bit systems.

The x86 family has a PAUSE instruction that improves the performance of spinlocks.

ARM32 and ARM64

The discussion is based on the ARMv6-M, ARMv7-A, and ARMv8-A architecture reference manuals [ARM].

Single-copy atomicity

The ARMv6 manual states that 1, 2, and 4 byte aligned accesses are single-copy atomic. A few multi-word accesses are executed as sequences of single-copy atomic operations.

The ARMv7 manual states that 1, 2, and 4 byte aligned accesses are single-copy atomic, that LDREXD and STREXD (8 byte) aligned accesses are single-copy atomic, and aligned LDRD and STRD are single-copy atomic when the processor has support for LPAE (standard with Cortex-A15 and later). A number of multi-word accesses are executed as sequences of single-copy atomic accesses.

The ARMv7 manual also says that when two stores of different size are racing to the same location and they would otherwise be single-copy atomic, then bets are off about atomicity.

The ARMv8 manual states that 8 byte aligned accesses (in 64-bit mode) are single-copy atomic, and that LDREXD and STREXD (16 byte) aligned accesses are single-copy atomic if the store is used with the load and the store succeeds.

The ARMv8 manual also states that single-copy atomicity is guaranteed only for instructions that target general-purpose registers, ie, is not guaranteed for direct floating-point loads and stores or for SIMD data. More discussion at [JMM atomicity].

Memory consistency model
For practical purposes the architecture is weakly ordered; it is possible for some memory to be designated as "strongly ordered" but my understanding is that "normal" memory is weakly ordered.

There are two synchronization instructions, "DSB" which is a completion barrier (store has reached the coherent memory system) and "DMB" which is an ordering barrier (orders the write buffer). Each can take an "option"; the most interesting option is "ST", which so far as I understand can be used to create a cheaper StoreStore barrier. Only DMB is required for normal application software.

ARMv6 makes the DSB and DMB operations available through coprocessor instructions [ARM barrier].

Synchronization
Synchronization is via a family of load-exclusive/store-exclusive instructions (LDREX and STREX), for sizes of 1, 2, 4, and (since ARMv6-K and on all ARMv7 and ARMv8 systems) 8 bytes. These do not imply any ordering of other memory operations.

ARMv8 additionally introduces a class of load-acquire and store-release instructions (LDAEX and STLEX), for sizes of 1, 2, 4, and 8 bytes. These imply memory ordering as well as exclusivity, and reduce the need to use DMB.

POWER and PowerPC

Discussion based on the POWER5 / PowerPC 2.02 book set [PowerPC], which is quite dated. It appears to cover both 32-bit and 64-bit models.

Single-copy atomicity
The architecture is single-copy atomic for naturally aligned accesses of 1, 2, 4, and 8 bytes (vol 2 section 1.4).

Memory consistency model
Memory is weakly consistent, like ARM.

The architecture has several barrier instructions. The most comprehensive, *sync*, orders all instructions preceding the barrier before any memory accesses succeeding the barrier. A cheaper instruction, *lwsync*, creates LoadLoad, LoadStore, and StoreStore barriers. The I/O barrier, *eieio*, can be used to create a StoreStore barrier by itself.

Synchronization
There are load-linked/store-conditional instructions (here called "load word and reserve" and "store word conditional") for 4-byte and 8-byte aligned data. Clearly the 4-byte version can be used to implement 8-bit and 16-bit operations using read-modify-write; the LL/SC reservation will work just fine for that.

A particular form of the NOP instruction can be used to effect a pause for the purpose of implementing spinlocks [PPC Pause].

MIPS32

Discussion based on the MIPS32 Rev 5.03 manual set [MIPS32]. For practical purposes we can probably stick to Release 2 or later; Release 2 came in 2002, Release 3 in 2010.

Single-copy atomicity
"Architecture rule B-2", vol II-A sec B.4.1, suggests strongly that the memory model is at least single-copy atomic for naturally aligned data, because that section states that misaligned accesses are "not guaranteed to be atomic as observed from other threads, processors, and I/O devices".

It seems likely (based on prose I believe I saw elsewhere but cannot find again) that the memory model is single-copy atomic up to 64 bits, ie, floating point loads and stores are single-copy atomic.

Memory consistency model
The consistency model of MIPS32 appears to be implementation-defined, and for practical purposes we should treat MIPS as weakly ordered and similar to ARM and Power. The following is based on the "Programming notes" for the SYNC instruction.

Evidently some implementations are strongly consistent, but that's not required, and the SYNC instruction performs various kinds of memory barriers (that are no-ops on SC systems).

The SYNC instruction can be used as-is to perform all barriers, or it can be configured with flags to only provide certain kinds of barriers (LoadLoad, LoadStore, etc, and sometimes a combination). Without an option the SYNC is a "completion barrier", it ensures that a write has actually gone out to the memory system. With an option the SYNC is an "ordering barrier", it ensures that reads and/or writes before the SYNC will touch the memory system before reads and/or writes after the SYNC.

Synchronization
MIPS32 has a 32-bit LL/SC pair. Clearly this can be used to implement 8-bit and 16-bit operations using read-modify-write; the LL/SC reservation will work just fine for that.

In Release 2.5 a PAUSE instruction was introduced that improves the performance of spinlocks. This release also introduced new options for the SYNC instruction. (Volume 1, section 2.1.2.2.)

Notes on the Firefox implementation

There are no restrictions on what the main thread can do: it can access shared memory and block in a **futexWait**. The slow-script mechanism aborts the script in the standard fashion if it blocks in the **futexWait** for too long.

In Firefox, futex wakeup is handled via an existing interrupt mechanism and no additional code is required to support wait/wake on the main thread. Also, main thread blocking has been useful for experimentation. At the moment, though, there are known usability issues with waiting on the main thread. Notably, the main thread cannot immediately wait for a wakeup from a new worker; it must return to its event loop and wait for the worker to post a message that it is up and running before waiting.

Additional background material

Alglave et al, "Herding cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory" contains much information (and good tutorial material) about weak memory models. [Herding cats]

Boehm and Demsky, "Outlawing Ghosts: Avoiding Out-of-Thin-Air Results" discusses the complications around writing a language specification whose memory model properly prevents "manufactured" values while not overly constraining weak-memory-model hardware. [Boehm ghosts]

References

[ARM] ARM Architecture Reference Manual. Downloadable from http://infocenter.arm.com, free registration required.

[ARM barrier] Leif Lindholm, "Memory access ordering part 3 - memory access ordering in the ARM Architecture" http://community.arm.com/groups/processors/blog/2011/10/19/memory-access-ordering-part-3--memory-access-ordering-in-the-arm-architecture; also comments on this draft by Lindholm.

[asm.js additions] asm.js: Semantic additions for shared memory and atomics: https://docs.google.com/document/d/19X8Geo_7OMyyUICMvIqzEgSDms0rRMDODdMIkXIm9m0/edit?usp=sharing

[Asymmetric barrier] Example implementation of an asymmetric barrier sync, where the main thread gets a callback when the workers are all waiting in the barrier: https://github.com/lars-t-hansen/parlib-simple/blob/master/src/asymmetric-barrier.js.

[Boehm ghosts] http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42967.pdf

[C++11 atomics] C++11 standard, section [atomics.types.generic], paragraphs 4 and 5.

[C++11 mappings] http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

[Cam cacm] http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf.

[Cookbook] JSR-133 Cookbook for Compiler Writers. http://g.oswego.edu/dl/jmm/cookbook.html.

[ES6] http://people.mozilla.org/~jorendorff/es6-draft.html

[Herding cats] http://www0.cs.ucl.ac.uk/staff/j.alglave/papers/toplas14.pdf

[Intel] "Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 1, 2A, 2B, and 3A: System Programming Guide, Part 1", March 2012.

[JMM atomicity] http://mail.openjdk.java.net/pipermail/jmm-dev/2014-February/000011.html with subsequent and following messages in that thread.

[MIPS32] MIPS Architecture For Programmers Volume I-A, II-A, III, Revision 5.03, Sept 9, 2013. http://www.imgtec.com/mips/architectures/mips32.asp, free registration required.

[Multiple values] J. Michael Ashley and R. Kent Dybvig, "An Efficient Implementation of Multiple Return Values in Scheme", ACM LFP 1994, http://www.cs.indiana.edu/~dyb/pubs/mrvs.pdf.

[PNaCl volatile] https://developer.chrome.com/native-client/reference/pnacl-c-cpp-language-support#volatile-memory-accesses

[PowerPC] "PowerPC User Instruction Set Architecture" and "PowerPC Virtual Environment Architecture", version 2.02, January 2005. These describe the POWER5 architecture which coincided with PPC at that time. Downloaded free of charge from IBM: http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html.

[PPC pause] http://stackoverflow.com/questions/5425506/equivalent-of-x86-pause-instruction-for-ppc.

[Structured clone] http://www.w3.org/TR/html5/infrastructure.html#safe-passing-of-structured-data

[WhatWG script env] https://html.spec.whatwg.org/multipage/webappapis.html#script-execution-environment

[Worker onclose] https://developer.mozilla.org/en-US/docs/Web/API/WorkerGlobalScope/onclose

Changes since the September 12, 2014 draft

September 24:

- Added information about the load-acquire / store-release instructions on ARMv8.

September 30:

- Clarified that the intent is for the shared memory mechanism to work with shared workers as well as with dedicated workers.
- Clarified the use of a static-method style rather than a prototype-method style.
- Clarified that the memory barrier on the atomic operations is an ordering barrier, and added prose to clearly indicate what that means in terms of prohibiting reordering at various levels.
- Removed the guard against being called from the main thread on **futexWait**.
- Added logic to **futexWait** to allow a wait on the main thread to be unblocked by the script-is-hung logic.
- Created a section discussing issues introduced by jitting code.

Changes since the September 30, 2014 draft

December 10:

- Specified the meaning of **worker.terminate**.
- Specified the meaning of worker termination in general.

Feburary 3/4, 2015:

- Corrected a grammar error.
- Clarified the prose in some locations.

February 20:

- The timeout argument to **futexWait** is optional and defaults to +Infinity. It specifies the time to wait in milliseconds as a Number, not the time to wait in nanoseconds as an integer. The clamping of the value is described better.
- Noted (non-normatively) that script wakeup order may be perturbed if futexWait/futexWake interacts with interrupts in certain ways.
- Clarification in **futexWaitOrRequeue**.
- Noted that when a worker sleeps in **futexWait** it will not wake up for event delivery.
- Allowed the use of **load**, **store**, and **compareExchange** on SharedFloat32Array and SharedFloat64Array: small changes to the spec of those operations, and changes to the "General guidance" section about how to implement it and changes to the "Semantic fine points" about the single return value from **compareExchange**.
- Editorialized about the lack of multiple return value support in JS.
- Noted that machine dependencies may provide fingerprinting information.
- Corrected redundant prose: "integer-valued SharedInt32Array".
- Clarified that ARM is not single-copy atomic if the instruction does not target a general-purpose register.
- Tightened our hardware assumptions to exclude assumptions about single-copy atomicity of FPU loads and stores.
- Clarified that atomic accesses are reliable only if the accesses are to the same number of bytes at the same memory address.

- Added a footnote regarding observability of worker termination.
- Clarified the wording of what goes into the transfer map when sending a SharedTypedArray (namely, the underlying SharedArrayBuffer).

February 26:
- Reworked the wording in the introduction regarding what the main thread can or cannot do.
- Moved information about the Firefox implementation to a separate section.
- Added a note on the PPC "pause" instruction.
- Added a link to the asm.js addeda for shared memory and atomics.
- Alphabetized the reference list.
- Removed a paragraph from the introduction that was redundant with the TOC.
- Removed two obsolete references to "the main thread" in the futexWait spec and fixed a broken cross-reference in that algorithm.

February 27:
- Clarified the prose around the mapping of volatile in the asm.js section.
- Moved two references from footnotes into the References.

March 1:
- Added a reference to the C++ atomics specification.
- Added a section to the Rationale, "Some rejected ideas", and a reference to a sample asymmetric barrier.

March 5:
- Added a subsection to the Rationale, "Motivation and requirements"
- Sundry clarifications based on comments

March 11:
- Added Atomics.exchange and Atomics.isLockFree
- Corrected the cross reference from Atomics.sub et seq to Atomics.add: 6(b) should be 7(b)

March 12:
- Improved prose for Atomics.isLockFree to constrain how the result might change between execution environments and over time
- Added prose to structured clone algorithm to clarify that the identifier G is inherited when a SAB is cloned.
- Incorporated description of ARM LDRD / STRD instructions.
- Removed Atomics.fence

April 21:
- Clarification of ARMv6 coprocessor barriers.
- Clarified rationale prose around optimizations.

July 3 and later:

- All Atomic operations now throw RangeError on out-of-bounds accesses.
- Accepted various suggestions, notably around the memory model
- Obsoleted this document and moved it to github, see introductory notes.

July 9: The changelong will no longer be updated. The document is not likely to be modified much from this point on, modifications go into the github version.