Shared memory and Atomics (proposal)

Ecma TC39 January 2016

lars-t-hansen.github.io/ecmascript_sharedmem

github.com/lars-t-hansen/ecmascript_sharedmem

Outline

Motivation for proposal

Technical recap -- if desired

Concerns raised at September '15 meeting

Discussion of proposal's prospects & status

Motivation

Goal: Performance

Effective use of multiple cores in ES

2-8 cores are common now

Native apps: threaded + shared memory

ES: one core/worker + slow events

Use cases (1)

asm.js and wasm

- “ES as a compiler target”
- Threads and shared memory in translated code
- Compatible with C / C++ / pthreads
- Fast interoperation with ES

In truth, the primary use case.

Use cases (2)

“Plain” ES

- Multicore computation with <u>shared state</u>
- Fast communication through shared memory
- Decoupling computation and rendering
- Management of shared assets
- Examples: pdf.js, game AI

Compromises and constraints

Use cases conflict:

- asm.js has flat memory, no GC, strong types
- “Plain” ES is object-based, GCd, weak types

Can a compromise solution serve both?

For asm.js and wasm, the memory model must be largely compatible with C and C++

Approach (1)

Provide low-level facilities

- SharedArrayBuffer + TypedArray
- Atomic operations
- Concurrent agents (workers)
- Agent sleep/wakeup operations

Approach (2)

Build higher level facilities for ES

- Locks, barriers, synchronic objects
- Communication channels
- Parallel computation abstractions

Extensible Web approach works well here

Why not only for wasm? (1)

Some tasks are best / only done in ES callouts

- Input and output, DOM access
- “Runtime” tasks

Copy-in / copy-out at boundary too slow (WebGL)

Thus ES wants <u>some</u> shmem access

Also asm.js (= ES) can't polyfill for wasm

Why not only for wasm? (2)

"wasm-only" does not remove complexity

- ES can call wasm
- wasm will export accessor functions to its shared memory

Now we have threads-as-library: not better

Why not only for wasm? (3)

Multicore computation in ES is valuable

ES is the better language (GC, rich types)

Divide responsibilities:

- ES code and objects for logic
- Shared memory for shared state
- Shared memory for fast communication
- TypedObjects to structure shared state

Why not something completely different?

“Compiler target” use case is constraining

Requires shared, racy memory + atomics

“Something different” would be ES-only, not useful as compiler target

Not what this proposal is about

Status

Spec is stable, change is incremental

Memory model “ok” but not quite done

In Firefox (Nightly) and Chrome (release) now

Apple, Microsoft aware of work, no public signals

API

API: Shared memory

A new data type:

```
var sab = new SharedArrayBuffer(size)
```

Like ArrayBuffer but:

- cannot be neutered
- memory becomes shared when it is transfered to another agent
- implementations must GC these across agents

API: Views on shared memory

Existing TypedArrays can be applied to SAB:

```
var sab = new SharedArrayBuffer(size)
var ia = new Int32Array(sab)
var fa = new Float64Array(sab, 8, 10)
```

Complicates TA spec a little

Memory access through views is <u>racy</u>, but <u>safe</u>

API: Atomic operations

Suite of atomic operations on integer TypedArray

```
Atomics.load(ia, n)
Atomics.store(ia, n, value)
Atomics.compareExchange(ia, n, expectVal,
                        replaceVal)
Atomics.exchange(ia, n, value)
Atomics.add(ia, n, value)
...
```

(All are sequentially consistent; details later)

API: Agent sleep and wakeup

Modeled on Linux “futex” (fast user-space mutex)

```
Atomics.futexWait(i32a, loc, expect, timeout)
Atomics.futexWake(i32a, loc, count)
```

Minimal assumptions, very flexible

Fairly hard to use directly

Different use cases will wrap futex differently

Example: mutex lock()

```
// State 0 = unlocked, 1 = uncontended, 2 = maybe contended

Lock.prototype.lock = function () {
  const iab = this.iab;               // Int32Array (shared)
  const idx = this.stateIdx;          // Index in iab
  var c;
  if((c = Atomics.compareExchange(iab, idx, 0, 1)) != 0) {
    do {
      if(c == 2 ||
         Atomics.compareExchange(iab, idx, 1, 2) != 0)
      {
         Atomics.futexWait(iab, idx, 2);
      }
    } while((c=Atomics.compareExchange(iab,idx, 0, 2))!=0);
  }
}
```

Example: mutex unlock()

```
// State 0 = unlocked, 1 = uncontended, 2 = maybe contended

Lock.prototype.unlock = function () {
  const iab = this.iab;            // Int32Array (shared)
  const idx = this.stateIdx;       // Index in iab
  var v0 = Atomics.sub(iab, idx, 1);
  if (v0 != 1) {
    Atomics.store(iab, idx, 0);
    Atomics.futexWake(iab, idx, 1);
  }
}
```

Agent model

Need a model for concurrency in ECMAScript

Define concurrency in terms of agents

Define agents in terms of ES6 jobs

Give jobs a forward progress guarantee

Creating agents, sharing a SAB among agents: mapping-specific.

Agent mapping

In a browser, an agent could be a web worker

- SAB sharing is by postMessage
- Web worker semantics need work...
- No futexWait on the browser's main thread

Memory model (1)

Atomics in the program are totally ordered.

Conventional happens-before relation on events:

- Program order (intra-agent)
- Atomic-write → atomic-read (inter-agent)
- futexWake called → futexWait returns (ditto)
- postMessage → event callback (ditto)
- transitivity, irreflexivity

Memory model (2)

Reads only see writes that happened before them (and only the last of those writes)

Unordered accesses, where at least one is a write, create a data race.

Data-race-free programs are sequentially consistent.

Memory model (3)

Races are <u>safe</u>: Programs don't blow up.

But races are <u>unpredictable</u>: A race poisons memory by reading and writing garbage.

Type-safe values are read (no pointers).

A race affects the union of the locations in the racing accesses.

Memory model (4)

Complications:

- Aliased arrays and cells that are not exclusively atomic
- Weakly ordered memory (ARM, MIPS, Power)
- Races have “a little” meaning

Non-complications:

- No out-of-thin-air values

Other memory model issues

Only sequentially consistent atomics, for now

- No “relaxed” atomics – we want them, but they are complicated
- No “acquire-release” atomics – ditto

Concerns Raised

Concerns

futexWait on the main thread:

- can we avoid it, do we need more features?

Side-channel attacks

- we can build a really good clock

Memory model leaks into the language

- races and optimizations become visible

futexWait on the main thead

No longer available in Firefox

We get along without it

Have built “asymmetric” data structures to cope:

- futexWait in workers
- Message event on the main thread

Barriers, synchronics, latches, queues

futexWaitCallback?

Using a Message event is non-compositional

What about this as a better abstraction?

```
// Main thread
futexWaitCallback(mem, idx, callback)

// Worker
futexWake(mem, idx)
```

futexWaitCallback premature

Better to experiment for a while IMO

Might be useful for “synchronic” primitive

To be investigated further

Side-channel attack

Can build a good clock in shared memory:

```
// Clock producer thread
while (true) Atomics.add(mem, idx, 1)

// Clock consumer thread
let t = Atomics.load(mem, idx)
```

4ns resolution, fairly reliable

Cache attack

Clock can be used to launch cache attacks

- cache sniffing (user interactions, secret data)
- row hammering (unclear impact in js)

A clock is needed to distinguish ops that hit or miss the cache

<u>This</u> clock is not needed, but it is better

Not a new attack in browsers

The cache attack can be launched in other ways

- Flash Player
- Java
- PNaCl/NaCl
- Any native browser extension
  - Chrome native messaging
  - Firefox js-ctypes
  - ActiveX
- (WebAssembly)

Status quo?

A new capability for JS, <u>not</u> really a new capability in most browsers.

For example, Flash Player:

- Runs automatically for most people
- The rest of the people click “yes” when prompted (says telemetry)

(And then, WebAssembly)

Mitigations for side-channel?

Mitigation options are so-so:

- Thread affinity solutions are plausible
- Opt-in for shared memory seems difficult

But again, not a new attack in browsers

Complexity

Races create indeterminacy

ES implementation must not create new races

Atomic operators order operations

ES implementations must not undo that order

Affects semantics and compilers in subtle ways

Compiler can't introduce races

Must assume unknown function has an atomic op

- Can't move or duplicate TA access across call unless access is known not to be shared

Must honor datum size

- Can't RMW larger datum

Must avoid speculative accesses

- `if (cond) x++;` vs `x++; if (!cond) x--;`

“Shared memory”

“Shared memory” is independent of the SAB

Atomic ops on one SAB orders <u>all</u> atomic ops

JIT must assume there are unknown SABs

JIT must assume a TA is shared unless proven otherwise

User affected “only” by races

Data races expose compiler optimizations

```
// One thread (racy load)
while (!mem[flagLoc]) {}

// Another thread (racy store)
mem[flagLoc] = 1
```

Some JITs hoist the load...

Loops forever if jitted, terminates if interpreted

Proposal's prospects

Where do we stand? (1)

Does TC39 want this proposal?

Racy memory, true concurrency <u>not negotiable</u>

- high-res timer issue
- spec complexity

Anything else might be negotiated

Where do we stand? (2)

If TC39 does not take it -- then what?

- competitive issues (native apps more capable)
- web ecosystem (wasm needs something)

Wasm will do its own thing

Wasm programs will just export peek(), poke()

Where do we stand? (3)

Ready for stage 2? ("Do we think we want it?")

- More guidance needed
- Committee involvement (reviewers) desired
- Wasm involvement

(Sundry other slides)

What's wrong with workers?

- No way to detect if a worker was
  - Started (resource constraints)
  - Terminated
- No forward progress guarantee
- Browser may kill a worker <u>at any time for any reason</u>, clearly overbroad

Weaker memory models

Acquire/release desirable for performance

```
Atomics.storeRelease(iab, loc, val)
Atomics.loadAcquire(iab, loc)
```

Relaxed desirable for special cases

```
Atomics.loadRelaxed(iab, loc)
Atomics.storeRelaxed(iab, loc, val)
```

Excluded to control (initial) complexity

“Racy-jQuery” (1)

Fear that the web will become racy when a horde of casual scripters start using shared memory

Not unwarranted but overblown IMO

Experiments are probably needed to settle this (ie, deployment and PR)

“Racy-jQuery” (2)

Restrictions have been suggested:

- Main thread can access shared memory when workers are stopped (callback, probe)
- Main thread has no access to shared memory at all (proxy through workers)
- Workers must opt in to use shared memory (pragma, constructor flag, special module)

But how effective?

Why not PJS?

PJS subset is hard to pin down

- hard to know what's effective
- impediment to portability, standardization

"Casual" API does not provide speedup

- overhead (recompile, gc, scheduling)
- granularity issues, lack of control

Fixable? Maybe, but not obvious

Alternate designs (1)

Proper atomic cells or mutexes?

Hard to reconcile with asm.js needs

- Atomic objects require some type of storage management and initialization
- Problematic for pthreads, C++11 atomic<>, as well as C11 atomics
- “Rich” features may serve fewer use cases

Alternate designs (2)

Avoid TypedArray, do something else:

```
let s = new Shared(byteSize)
let v = Shared.int32.load(s, 37)
Shared.int32.store(s, 37, x)
```

Somewhat verbose, user-hostile?

Does not get rid of essential complexity: races