Shared Memory and Atomics (Stage 2 Proposal -- Update)

Ecma TC39 March 2016

tc39.github.io/ecmascript_sharedmem

github.com/tc39/ecmascript_sharedmem

Outline

Recent progress

Concerns raised at/since January '16 meeting

Plans going forward

Recent progress

What has happened?

Spec is smaller!

- “Agents” is now a (small) spec of its own
- Structured clone adjustments moved to HTML
- Most browser notes moved with Agents and SC

Clarifications throughout

- Memory model
- Bug fixes and adjustments

Resolutions (1)

futexWaitOrRequeue() disappears

- Unclear value, premature optimization

futexWait() return codes become strings (probably)

- Integer values not canonical, not needed

int32 slots are always lock-free

- Simplifies programs, seems unproblematic

Resolutions (2)

Futexes are in, synchronics are out

- Synchronic proposal a poor fit for jS

Bug fixes

Range checking aligned with SIMD, DataView

futexWake() now wakes all waiters by default

Reference the Agents spec where appropriate

Memory model

Sundry fixes:

- Define the program region affected by a race
- Constrain the values that can result from a race
- Verify that there is no risk of “quantum garbage”
- Editorial fixes

Memory model work is ongoing.

Concerns raised at the January '16 meeting (and since)

Run-down of issues

These issues were raised:

- Alignment with WebAssembly
- Disable shared memory on main thread
- “Quantum garbage” in data races
- Concern about races “leaking” into language
- Revealing secrets through races
- Termination and partial failure

Alignment with WebAssembly

Discussions with WebAssembly authors:

- Alignment is fine, so far

Overlap between SAB group and wasm group helps maintain alignment

Discussion ongoing re aligning formal memory models

No shared memory on main thread?

Issue #54 tracks this (long writeup)

“Not blocking” on main thread != “always racy”

- synchronization possible & affordable
- use messages when necessary

No (new) investigations performed

Plenty of demonstration code exists

Quantum Garbage

Can this print “0” if mem is shared memory?

```
var x = mem[n]     |  // optimization
if (x)             |  if (mem[n])
  print(x)         |    print(mem[n])
```

Answer: No. Requires optimization that is arguably forbidden by ES semantics when combined with data race semantics.

Optimization (1)

Data races do not cause “undefined behavior” but are constrained:

- Memory region affected by race
- Observed values

The engine must assume a TA access is racy (or prove otherwise)

- JITs are fine with this

What happens in a race

Nondeterministic values in races

- Model too loose now (“any value whatsoever”)
- Can constrain it to “combination of bits written”
- Maybe even more
- Modern hardware probably helps here
- Ideally back this up with a formal model

Optimization (2)

Some optimizations not legal in shared memory:

- Introducing reads, eg
  - Rematerialization (quantum garbage)
- Introducing writes, eg
  - Using shared memory for compiler's temps
  - Making conditional writes unconditional
  - Read-modify-write on a larger datum

Arguably forbidden already because races do not cause UB.

Optimization (3)

Common optimizations do not introduce races in DRF programs (Sevcik PLDI 2011):

- reordering reads and writes
- removing redundant reads and writes

How best to characterize what's legal / not?

- Ideally fall out of the memory model, not ad-hoc rules

Partial failure & termination

Two agents share memory and one crashes (eg separate processes, one is gunned down)

What happens to the survivor? What if the victim was “holding a lock”?

Agent spec tries to constrain this (tentative):

- Agents within a cluster die together, or
- There exists a notification mechanism

Plans and open issues

Open issues

Memory model still not complete

- fine points around races remain
- awkward / fuzzy wording in many places
- desirable to formalize it

Optimization primitives for lock implementation?

- Atomics.pause() functionality
- orthogonal to everything else

Memory model formalization

Desirable but perhaps not a hard requirement

Formalization an emerging art + science

Our memory model is low-level

- Existing results suggest proofs may be tricky
- Mixing atomics + non-atomics, different widths
- No objects, only cells

Implementation status

In Firefox (beta) and Chrome (release) now

Behind flags in both browsers

Test suite affirms “reasonable” compatibility

Progress plan

Still aiming for Stage 3 at the May meeting

- Memory model is main outstanding issue

Formal review start ca 1 May?

- Rolling reviews very welcome

Agents spec evolves in parallel, not a blocker

Formal reviewers

Formal reviewers for Stage 3

At large: Dan Ehrenberg, Brian Terlson, Filip Pizlo

Memory model: Waldemar Horwat

Agents / HTML interaction: Domenic Denicola

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 safe: Programs don't blow up.

But races are unpredictable: 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

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 all 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

Where do we stand? (1)

Does TC39 want this proposal?

Racy memory, true concurrency not negotiable

- high-res timer issue
- spec complexity

Anything else might be negotiated