Shared memory and atomics
(Stage 2 proposal)

Update + Petition for Stage 2.95 (API freeze)

July, 2016

Outline

API recap

Memory model

Accumulated committee concerns

Petition for API freeze (Stage 2.95?)

Agents and agent clusters

An agent just packages an ES execution env

An agent cluster is a maximal set of agents that can communicate through shared memory

We need these ideas to talk about:

- blocking, forward progress, termination
- shared invariants (endianness, lock-freedom)

Agent semantics

<u>Blocking</u>: An agent can block waiting to be woken without returning to its event loop

<u>Web awareness</u>: An embedding can deny some agents the ability to block (eg browser's main thread)

<u>Forward progress</u>: Agents must eventually advance if they are not blocked

<u>Cluster unity</u>: If the embedding terminates an agent, it must terminate all; if it suspends one, it must suspend all

SharedArrayBuffer

SharedArrayBuffer is a new data type

Like ArrayBuffer:

- Map TypedArray and DataView onto it

Unlike ArrayBuffer:

- not detachable
- buffer memory is shared between agents
- different memory access semantics

Sharing memory

The agents are the main thread + web workers

Memory is shared with postMessage

```
var w=new Worker(...)
var sab=new SharedArrayBuffer(...)
w.postMessage(sab)

var arr=new Float64Array(sab)
arr[10]=3.5**7     // unsynchronized
```

Atomics

The global Atomics namespace has static methods that operate on integer TypedArrays

Atomic access: load, store, add, sub, and, or, xor, exchange, compareExchange

Suspend and wakeup: wait, wake

Optimization advice: isLockFree

Synchronization

Agent A busy-waits for flag:

```
while (Atomics.load(arr, flagLoc) == 0) {}
x = Atomics.load(arr, flagLoc)
assert(x == 0xC0DE)
```

Agent B writes:

```
Atomics.store(arr, flagLoc, 0xC0DE)
```

Efficient synchronization

Agent A blocks to wait for flag:

```
Atomics.wait(arr, flagLoc, 0)    // blocks
x = Atomics.load(arr, flagLoc)
assert(x == 0xC0DE)
```

Agent B writes the flag and wakes up waiters:

```
Atomics.store(arr, flagLoc, 0xC0DE)
Atomics.wake(arr, flagLoc)
```

And that's it for the API!

Any questions about the API?

Memory model

What's it for, anyway?

Challenges and opportunities

Two-level memory model

Synchronization order and viability

Data races and reordering

Write buffering and speculation

Why a memory model?

A concise and precise model for <u>which writes can be observed by which reads</u>

Allows programmers and compilers to reason about programs and optimizations

Allows programmers to establish general properties, eg, sequentially consistent behavior for race-free programs

Challenges

ES shared memory is low level

- No separate “atomic cell” idea
- Aliased memory, no real type discipline

ES programs must remain safe and portable

- “Undefined behavior” is not acceptable
- Widely varying implementation behavior is bad

... and opportunities

ES is “easier” than C/C++

- Performance demands slightly lower
- Shared memory separated from non-shared
- Only <u>flat</u> shared memory; no pointers or objects

ES can make hardware assumptions (if it must)

- Non-tearing non-atomic accesses
- Mature atomic operations

Two-level memory model (1)

Conventional axiomatic high level model

- Defines synchronizing accesses
- ... and synchronization order
- ... and the happens-before relationship
- ... and a notion of data race freedom (DRF)
- ... and sequential consistency for DRF programs

Two-level memory model (2)

Mostly operational low level model

Two aspects:

- Limits on atomic operations to establish synchronization order
- Restrictions on program reorderings to limit the effects of data races

Synchronization order

Viable atomics are executed in a total order, the synchronization order

Synchronization order gives rise to

- “happens-before”
- “data race”
- other conventional definitions

Familiar from other languages

Viability (1)

“Viable” atomic accesses behave as if we had strongly (dynamically) typed atomic cells

- a viable atomic read <u>sees</u> only atomic writes to the exact same bytes
- a viable atomic write isn't <u>interfered with:</u> any concurrent write to overlapping byte ranges is always an atomic write to the exact same bytes

(Need to account for initializating stores, too)

Viability (2)

```
a32 = new Int32Array(sab)
a8 = new Int8Array(sab)

Agent 1                      Agent 2

store(a32, 0, 1)   --> load(a32, 0) // ok
store(a8, 0, 1)    --> load(a32, 0) // not
a32[0] = 1         --> load(a32, 0) // not
```

Viability (3)

```
a32 = new Int32Array(sab)
a8 = new Int8Array(sab)

Agent 1                Agent 2

store(a32, 0, 1)       store(a32, 0, 2) // ok
store(a32, 0, 1)       store(a8, 0, 1)  // not
a32[0] = 1             store(a32, 0, 1) // not
```

Viability (4)

Oops! Can't use “happens-before” or “data race” to define “viable”

Viability can almost certainly be specified operationally to avoid circularity

This is ongoing work (because the previous non-operational attempt did not avoid circularity)

Use a simple abstract memory system that keeps track of de-facto atomic cells and handles conflicts

Data races and reordering (1)

Data races expose optimizations, notably reordering, but also introductions of reads and writes

Most optimizations remain allowed. Spec bans:

- reordering data accesses wrt atomic accesses
- duplicated reads where more than one value can be observed
- inserted writes that cause observable changes in memory that don't follow from standard semantics
- value speculation

Data races and reordering (2)

Data races have bounded effect on the values that are read or written

- values are at least some interleaving of the bytes of the values going into the race
- in some defined cases, values don't “tear”

Non-tearing adds a little spec complexity but is valuable, and is supported on all hardware

Write buffering and speculation

All shared memory access goes through ReadSharedMemory and WriteSharedMemory

Memory semantics can be different from GetValueFromBuffer and PutValueInBuffer

For example, CopyArrayBufferData on shared memory get the variant semantics

Reordering, in the form of write buffering and speculation, can happen in ReadSharedMemory and WriteSharedMemory

Stabilizing the memory model

Viability needs to be pinned down properly

Problem is better understood than before

Undertaking work in August/September to have it ready for September meeting

Accumulated committee concerns

- Memory model correctness
- Out-of-thin-air values
- Races “leaking” into the memory model
- Races revealing secrets at run-time
- Quantum garbage
- Introducing new data races
- Termination
- Security
- Compatibility with WebAssembly

Petition for Stage 2.95 (API freeze)

Formal requirements are mostly met:

- Complete solution; requires user/impl feedback
- Complete spec text with most semantics
  - Memory model: viability not yet well-defined
- No major outstanding issues; API very stable
- Most reviewers signed off
  - Waldemar holding off because of MM
- Editor signed off
- Firefox and Chrome implement most of the spec interoperably