SharedArrayBuffer and Atomics Stage 2.95 to Stage 3

Shu-yu Guo Lars Hansen

Mozilla

November 30, 2016

What We Have Consensus On

TC39 agreed on Stage 2.95, July 2016

- Agents
- API (frozen)

What We Have Consensus On

TC39 agreed on Stage 2.95, July 2016

- Agents
- ► API (frozen)

Memory model had fatal bug

Outline

Memory Model

- 1. Motivation
- 2. Intuition
- 3. What the Model Does

Should We Allow This Optimization?

```
 \begin{array}{lll} \textbf{let} & \texttt{x} = \texttt{U8[0]}; \\ \textbf{if} & (\texttt{x}) & \Rightarrow & \textbf{if} & (\texttt{U8[0]}) \\ & & \texttt{print(x)}; & & & \texttt{print(U8[0])}; \\ \end{array}
```

What About This One?

```
while (U8[0] == 42) ; 
 \Rightarrow \begin{array}{c} \text{let } c = U8[0] == 42; \\ \text{while } (c) ; \end{array}
```

Or This One?

```
let A = Atomics; let A = Atomics; \Rightarrow \quad \text{let } c = \text{A.load(U8,0)} == 42; while (A.load(U8,0) == 42); while (c);
```

What Can Be Printed Here?

What's a Memory Model Good For?

- Arbitrates optimization affordance
- Captures hardware reality

Memory Model Design Space

- 1. No model
- 2. Undefined behavior/values for data races
- 3. Fully defined; races have meaning

Why

Because we're the web.

- Interoperability
- Security

Why

Because we're the web.

- Interoperability
- Security
- WebAssembly

What

The model prescribes the set of values that can be read by SAB operations.

Intuition

Strong enough for programmers to reason about programs Weak enough for hardware and compiler reality

Programmers' Intuition

Sequential Consistency for Data Race Free Programs

Sequential consistency just means interleaving.

Data race freedom means no concurrent, non-atomic memory accesses where one's a write.

Implementors' Intuition: Codegen

Obvious code generation

- ▶ Non-atomics compiled to bare stores and loads
- Atomics to atomic instructions or with fences

Implementors' Intuition: Optimizations

- Atomics are carved in stone
- ▶ Reads must be stable (e.g. no read rematerialization)
- Writes must be stable (i.e. can't make observable changes to writes)
- ► Don't completely remove writes (i.e. can coalesce adjacent writes but not remove them completely)

What We Talk About When We Talk About Atomicity

Access atomicity
Indivisible action

What We Talk About When We Talk About Atomicity

Access atomicity

Indivisible action

Copy atomicity

Ordering: what memory accesses become visible to what cores when

What We Talk About When We Talk About Atomicity

The memory model orders shared memory events and prescribes what values can be read by them.

Ordering Analogies: Atomics

- ► C++ memory_order_seq_cst
- LLVM SequentiallyConsistent

Ordering Analogies: Non-Atomics

- ▶ Between C++ non-atomics and memory_order_relaxed
- Between LLVM non-atomics and Unordered

Details with all the math in the spec.

THIS SLIDE INTENTIONALLY LEFT BLANK

The rest of the presentation is not planned to be presented as it is unlikely a good use of committee time to go into the actual math. Nevertheless, they may be valuable for folks who are reading the slides and are interested in some of the math without going down the rabbit hole.

Model Overview

- Axiomatic memory model
- ▶ Interfacing with ES evaluation semantics

SAB MM

The model has two parts. The bulk of it is an axiomatic model that does the ordering of memory events as we talked about. But this model is axiomatic – it's a set of constraints, not an algorithm like the rest of ECMA262. So there's also a second part built into the evaluation semantics that interfaces with the axiomatic model.

Axiomatic Model

Ordering is done by an axiomatic model.

Input is a candidate execution—a set of memory events and a set of relations ordering them.

Output is a decision whether the candidate execution is valid.

The meaning of a program is the set of all valid executions.



SAB MM

Axiomatic Model

Ordering is done by an axiomatic model.

Axiomatic Mode

Input is a candidate execution—a set of memory events and a set of relations ordering them. Output is a decision whether the candidate execution is valid.

The meaning of a program is the set of all valid executions.

Axiomatic semantics is a big departure from the kind of semantics we do at TC39, which are all operational and algorithmic. Weak memory models allow for some weird acausal behavior that aren't capturable by a straightforward operational, algorithmic style. The state of the art in the literature of memory models is all axiomatic.

Axiomatic Model

Ordering is done by an axiomatic model.

Input is a candidate execution—a set of memory events and a set of relations ordering them.

Output is a decision whether the candidate execution is valid.

The meaning of a program is the set of all valid executions.

Not operational!

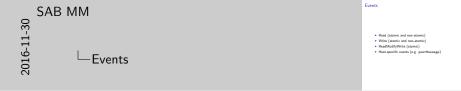
Not operational!

Axiomatic Model

Axiomatic semantics is a big departure from the kind of semantics we do at TC39, which are all operational and algorithmic. Weak memory models allow for some weird acausal behavior that aren't capturable by a straightforward operational, algorithmic style. The state of the art in the literature of memory models is all axiomatic.

Events

- Read (atomic and non-atomic)
- Write (atomic and non-atomic)
- ReadModifyWrite (atomic)
- Host-specific events (e.g. postMessage)



There are 3 kinds of shared memory events. Read events, write events, and RMW events. The host-specific events depend on the embedding.

Candidate Execution

A candidate execution is

- ► A set of events
- agent-order
- ► reads-from
- ▶ synchronizes-with
- happens-before

agent-order

The union of evaluation orders of all agents.

If E occurred before D in some agent, E is agent-order before D.

reads-from

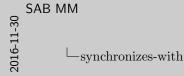
Maps Read and ReadModifyWrite events to Write and ReadModifyWrite events.

If R reads-from W, then R reads one or more bytes written by W.

synchronizes-with

A subset of $\operatorname{reads-from}$ that relates synchronizing atomic Read and ReadModifyWrite events to atomic Write and ReadModifyWrite events.

An atomic Read R synchronizes-with an atomic Write W when R reads every byte from W.





An atomic Read R synchronizes—with an atomic Write W when R reads every byte from W.

synchronizes-with

Recall that SAB API allows aliasing, so it's possible for an atomic read to read from multiple writes, atomic and non-atomic, such as in case of races.

happens-before

- agent-order relates intra-agent events
- synchronizes-with relates inter-agent events
- ▶ happens-before connects the two

 $(agent-order \cup synchronizes-with)^+$

Valid Executions

A candidate execution is valid when it has...

- ...coherent reads
- ...tear free reads
- ... sequentially consistent atomics

Coherent Reads

A read of some byte is coherent if it reads the most happens-before recent write to that byte.

R reads-from $W \Rightarrow \not\exists W'.W$ happens-before W'

SAB MM

Coherent Reads

A read of some byte is coherent if it reads the most happens-before recent write to that byte. R reads-from $W \Rightarrow 2W''.W'$ happens-before W'

Coherent Reads

Remember that not everything is related by happens-before.

Mathematically, happens-before is a strict partial order. So if there is a data race, for example, a read can read a more wall-time recent write as long as that write isn't more happens-before recent.

Tear Free Reads

Aligned accesses are well-behaved.

└─Tear Free Reads

SAB MM

The details are in the spec. The point here is that aligned accesses via integer TypedArrays have more guarantees than accesses via float TypedArrays and unaligned accesses via DataViews.

Sequentially Consistent Atomics

- All synchronizes-with atomic events exist in a strict total order consistent with happens-before.
- ▶ An atomic write becomes visible to atomic reads in finite time.



-Sequentially Consistent Atomics

guarantee. Non-atomics don't have either guarantee.

Sequentially Consistent Atomics

consistent with happens-before.

· All synchronizes-with atomic events exist in a strict total order An atomic write becomes visible to atomic reads in finite time.

This total order is the interleaving. The finite time is a liveness

Data Race Redux

E is in a data race with D iff

- ightharpoonup E and D aren't related by happens-before
- ▶ E or D is a Write or ReadModifyWrite event
- lacktriangleright E and D aren't synchronized atomics



Data Race Redux E is in a data see with D iff * E in D sen't related by langueous-before * E or D is a Write or BradehodylyWine event. * E and D sen't synchronized atomics.

A quick revisit to more precisely define data races now that we're armed with math.

Event Semantics

- ▶ A read event reads a value composed of bytes from write events it reads-from in a valid execution.
- ▶ Even racy reads have well-defined values!



SAB MM

Event Semantics

➤ A read event reads a value composed of bytes from write events it reads-from in a valid execution.

➤ Even race reads have well-defined values!

Event Semantics

This is often a set of more than one possible values. But note that this is still an axiomatic thing: we only know the value of a read event after we have the entire event graph and have ordered it according to the memory model.

Where do events come from?

└─Interface with Evaluation Semantics

To interface the axiomatic semantics with the evaluation semantics, we make the evaluation semantics nondeterministic. Read operations on SABs introduce read events, write operations write events, and Atomic RMW operations RMW events. The question is what is the value of read events during the evaluation semantics? It is nondeterministically any possible value.

Where do events come from?

Evaluation semantics introduces events

To interface the axiomatic semantics with the evaluation semantics, we make the evaluation semantics nondeterministic. Read operations on SABs introduce read events, write operations write events, and Atomic RMW operations RMW events. The question is what is the value of read events during the evaluation semantics? It is nondeterministically any possible value.

Where do events come from?

- Evaluation semantics introduces events
- Value of read events is any possible byte value

To interface the axiomatic semantics with the evaluation semantics, we make the evaluation semantics nondeterministic. Read operations on SABs introduce read events, write operations write events, and Atomic RMW operations RMW events. The question is what is the value of read events during the evaluation semantics? It is nondeterministically any possible value.

Without SAB the evaluation semantics constructs a correct execution directly.

With SAB the evaluation semantics constructs many candidate executions nondeterministically and the memory-model decides which ones are valid.

Without SAB the evaluation semantics constructs a correct execution directly.

With SAB the evaluation semantics constructs many candidate executions nondeterministically and the memory-model decides which ones are valid.

Interface with Evaluation Semantics

This makes sense intuitively – weak memory models permit many possible observed memory values, so the meaning of a program with SAB is the set of valid executions.