Shared memory and Atomics (proposal)

ECMA TC39 September 2015

Goal

Effective use of multiple cores in ES

2-8 cores are common now

Native apps: threaded, shared memory

ES: one core + a slow pipe between workers; no shared state

Use cases

asm.js

- pthreads in translated C/C++ code
- Support for safe threaded languages

“Plain” ES

- Shared state and multicore computation
- Fast communication through shared memory

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, the memory model must be largely compatible with C and C++

Approach

Provide low-level facilities

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

Build higher level facilities

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

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 racy, but safe

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...

In a non-browser setting (SpiderMonkey shell)

- concurrent thread, separate global environment
- mailbox mechanism for sharing memory

Implementation concerns

- Tricky to block on the main thread in browsers?
- “Subcontracting” - main browser thread acts on behalf of a worker
  - Possible to deadlock if main thread is waiting
  - But makes workers not truly concurrent
  - So where's the bug?
- Subcontracting for UI and other things are also problematic

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 writing garbage.

A race affects (at least) the union of the locations in the racing accesses. Fine points are TBD.

Memory model (4)

Complications:

- Aliased arrays and cells that are not exclusively atomic
- Weakly ordered memory (ARM, MIPS, Power)

C11, C++11 have similar issues, mildly unsound?

Java better? But complex and subtle

Other memory model issues

- No “relaxed” atomics – we probably want them but they are complicated
- No “acquire-release” atomics – ditto, though weaker use case
- Shared memory can be used to implement high-precision timers (you just count), which can be used to simplify cache sniffing attacks
- Misc minor issues, see github repo

Status

Spec is stable; memory model is being refined

In Firefox Nightly since Q1 2015; demo-level code for asm.js, plain JS is running

Google have committed publicly, at least in part

Apple, Microsoft aware of work, no public signals

Q&A

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 at any time for any reason, 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?

e.g., proper atomic cells (“Synchronic”), 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

Why not just for asm.js

Some tasks are best/only done in JS callouts

- I/O
- Thread management
- Runtime tasks in general

JS has data structures, easier to program

The callout needs at least some shmem access