Proposal: add thread creation to WebAssembly

Andrew Brown, Conrad Watt

Introductions

Conrad Watt

- author of Weakening WebAssembly paper
- champion of current threads proposal (phase 3 $\longrightarrow$ 4)

Andrew Brown

- co-champion of wasi-threads proposal
- other proposals: wasi-nn, Wasm SIMD

Proposal

- Add thread creation and other threads primitives to the WebAssembly specification
- Objectives today:
    - Request a vote to phase 1
    - Describe current thinking

Motivation

- Why threads in WebAssembly at all?
  - Past: it was always recognized as a post-MVP need
  - Present: it is an oft-requested feature by users:
    - to fix compilation failures
    - for algorithmic parallelism
    - for language support, e.g., async
  - Future: *the* key performance knob, expect more cores, expect more heterogeneous parallelism… (does anyone anticipate less parallelism?)

Motivation (cont.)

- Why a spec proposal?
  - Fits WebAssembly’s machine abstraction: threads are a computational resource, like memory
  - Improve compatibility: browser versus standalone incompatibility, main thread versus worker thread
  - Improve performance: thread spawning using Web Workers is slow and has memory overhead
  - Allow component model compatibility: “instance per thread” problems…
  - Solicit community feedback via prototyping (e.g., SIMD)

Current threads state

- Emscripten + browsers shipping one version of thread creation
- wasi-sdk + standalone engines shipping another version
  - several users of wasi-threads in production
  - …but wasi-threads cannot progress in the WASI subgroup
- Current thread proposal headed to phase 4
- Created thread-spawn repository for this proposal
  - 24 contributors discussed online
  - many more offline discussions!
- Browser vendors have concerns about re: web platform:
  - How does this proposal interact with JavaScript?
  - How does this proposal interact with Web Workers?

Reminder

- At this point, we *could* vote:
  - is there general interest?
  - is this in-scope?
- …but we might as well describe roughly what the proposal would encompass
- Remember:
  - the details can change during phase 1
  - many strong opinions to weigh here
  - we need prototyping to gather feedback

Proposed changes

Addition	Criticality	Notes
`shared` attributes	Required	Like shared memory but for tables, functions, globals
A spawn mechanism	Required - all platforms	e.g., a `thread.spawn` instruction
A thread-local storage mechanism	Required - all platforms	e.g., a `thread.id` instruction
Design for JS-Wasm interaction	Required - Web	How to access JS state from threads?
An early-exit mechanism	Helpful	e.g., a `thread.exit` instruction
A “how many threads?” mechanism	Helpful	e.g., a `thread.hw_concurrency` instruction

Proposed: `shared` attributes

```
(memory 0 1 shared)
(func $foo (param i32) shared ...)
(global $g1 (mut i32) (i32.const 1) shared)
(table 0 1 funcref shared)
```

From	**Can access?**	**To**	**Notes**
non-shared	✓	non-shared	This would continue to work as it does today
non-shared	✓	shared	This would also continue to work as it does today: e.g., functions can access shared memory.
shared	✗	non-shared	This is not possible, by validation.
shared	✓	shared	This is how we expect threads to access state: only shared state.

Proposed: spawn

- to decide: **instruction vs import**
  - instruction: `(thread.spawn …)`
  - import: `(import "env" "spawn" (func … shared))`
- to decide: **thread handles?**
  - no return value; join using emitted `memory.atomic.{wait*, notify}`
  - return a new “thread handle” type; e.g., cancellation?
- to decide: **multiple vs single invocation**
  - `(thread.spawn <num threads> …)`
- to decide: **fallible?**
  - return a value indicating failure
  - expect spawn to eventually succeed

Proposed: TLS

- motivation: need a way to efficiently access thread-local data
- to decide: instruction vs thread-local syntax vs ?
  - **new instruction**: `(thread.id) ⟶ i32`
    - optionally, pass the ID as an argument during spawn: e.g., `(thread.spawn … <id>)`
    - alternately, allow this to be set within the thread
    - could return the base address of the TLS region in the linear memory
  - new syntax: `(thread_local $tl1 (mut i32) i32.const 1))`
    - like a global, but “per thread per instance” instead of “per instance”
    - tempting, but difficult: TLS highly dynamic due to dynamic creation of threads and instances
  - toolchain emission: thread a TLS parameter through all shared functions
  - other options?

Proposed: JS-Wasm

- motivation: threads are not useful in browsers if one cannot call JS from the shared context
- to decide: dispatch threads vs JS context threading vs blessed callback
  - **dispatch threads**: as Emscripten does today, proxy JS calls to a main thread queue
  - **JS context forwarding**: in toolchains, thread a JS context parameter through all shared functions—issues with shared continuations?
  - **blessed callback**: one JS function is imported as `shared`; all “calls to JS” go through a thread-local dispatch table
- more to come in Conrad’s follow-on deep dive

Proposed: early exit

- **internally**: how does a thread exit itself from within the call stack? (e.g., `pthread_exit`)
  - use **exceptions**: `catch` at the top-level `shared` function, `throw` anywhere
  - new **instruction**: `(thread.exit)`
- **externally**: how does the parent thread terminate a child thread?
  - can be implemented at the toolchain or application level: atomically check a bit at every loop iteration?
  - thread cancellation not available in Rust; discouraged in Java
  - `pthread_cancel` not implemented by wasi-threads; no one complained
  - this may be easier in a web engine: e.g., `Worker.terminate()` already solves this
  - **more discussion** needed

Proposed: how many threads?

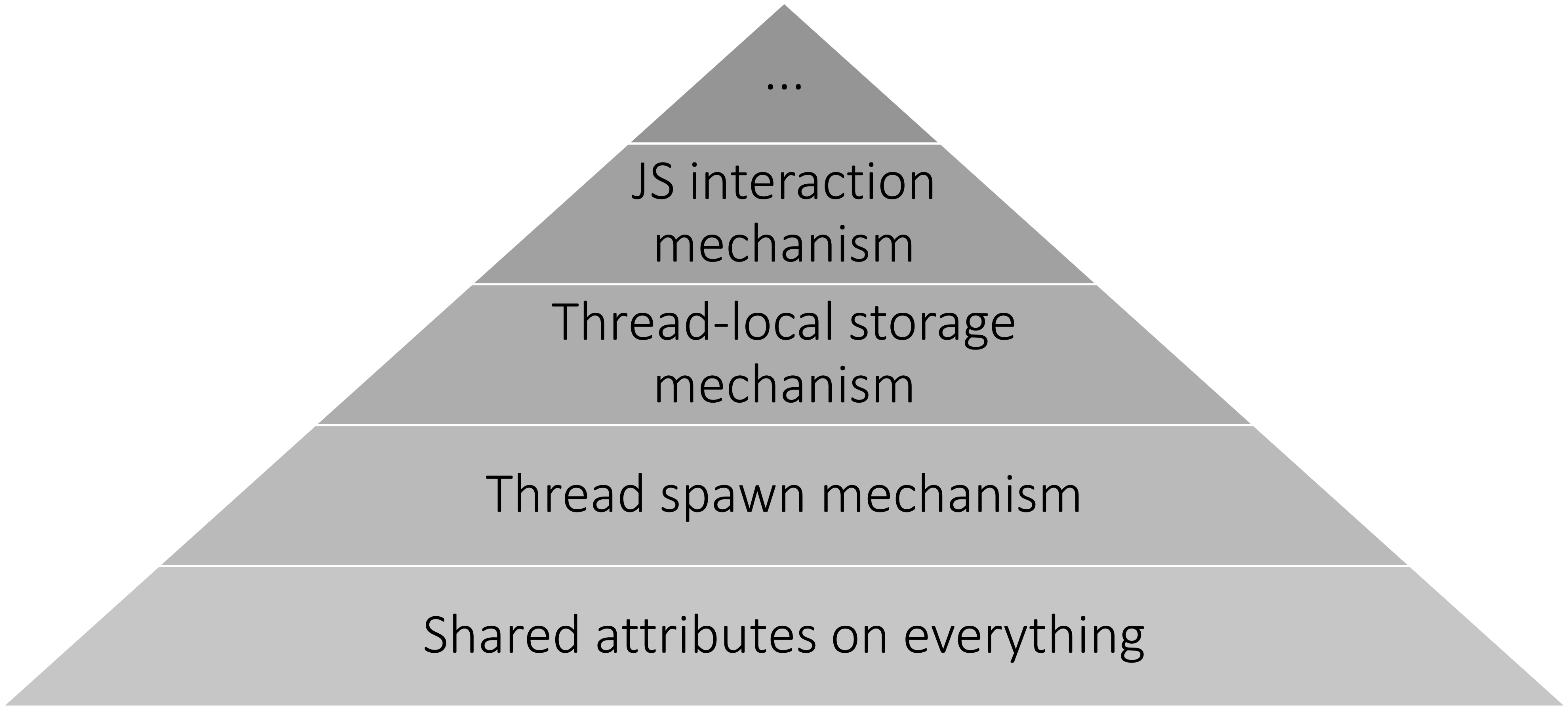
- motivation: many algorithms would like to fit to the number of threads that can execute concurrently
- to decide: **include vs remove**
  - new instruction: `(thread.hw_concurrency) ⟶ i32`
  - modeled after the browser `navigator.hardwareConcurrency` property
  - not necessarily equal to number of physical cores; could be limited by the engine, etc.

Summary

Addition	Criticality
`shared` attributes	Required
A spawn mechanism	Required on all platforms
A thread-local storage mechanism	Required on all platforms
Design for JS-Wasm interaction	Required on the Web
An early-exit mechanism	Helpful
A “how many threads?” mechanism	Helpful

- Reminder: full consensus on details is not necessary for phase 1
- Reminder: details may change in future phases

Too many details


...
JS interaction mechanism
Thread-local storage mechanism
Thread spawn mechanism
Shared attributes on everything

Shared attributes on everything

- The foundation that makes all other parts of this feasible
- Web engines need to draw a distinction between the "JS world" (all roots in the same thread, no cross-thread cycles) and the "shared world" - no "shared" -> "non-shared" refs allowed
- Two possible designs – distinguished by the question "do we believe that shared continuations will exist?"

Shared attributes on everything

- If shared continuations do not exist:

  shared functions may not close over any non-shared global, table, function declared by the module

  shared functions may still accept non-shared arguments
  (e.g. non-shared externref)

  This is because a function's execution always stays in the same thread during a call

Shared attributes on everything

- If shared continuations exist:

  shared functions may not close over any non-shared global, table, function declared by the module

  shared functions may also not accept non-shared arguments (e.g. non-shared externref)

  This is because the function could accept a non-shared argument, suspend as a shared continuation, and resume in another thread

Shared attributes on everything

My hot take:

Some people in the Stack Switching subgroup have said that shared continuations will exist, so we should go with the stricter definition of shared

Thread spawn mechamism

Two possible high-level designs:

1. "thread.spawn" is a WebAssembly instruction
2. "thread_spawn" is a host import

and then follow-on questions:

- What is the cost model for threads? Can we spawn 100?
- What control do we have over a spawned thread? Analogy – worker.terminate in JS
- Is thread creation fallible?

Thread spawn mechamism

My hot takes:

Better to have a native instruction if we can get away with it

Threads should be lightweight and not bound by hardware threads

Better to not have thread.terminate if we can get away with it

Thread creation should be fallible

Thread-local storage mechanism

One question – HOW?

Tempting design – arbitrary "thread-local globals" with some per-thread initial/default value

This would be awful to implement on the Web

- Every time a new thread is created, need to ensure it can handle TLS for every instance created everywhere
- But instances can be created full dynamically!
- Trying to do this lazily is hard as need to work out transitive reachability

Thread-local storage mechanism

One question – HOW?

More minimal design – thread.id - effectively a single i32 TLS variable that every thread gets to set (once upon creation?)

Mimic the way TLS works natively, but require that the TLS tables are implemented programmatically in Wasm shared linear memory or shared table – thread.id indexes the TLS table for the thread

Difference: allocate the single thread.id "slot" at thread creation time, no need to handle per-instance TLS state in the engine

Thread-local storage mechanism

thread.id

If threads_spawn is an import, may still want thread.id to be an instruction

Depending on how JS interaction is done (coming up), it would be hard to make thread_id an import on the Web

JS interaction mechanism

Problem: shared things can't hold references to non-shared things

All JS objects are non-shared (for now)

When compiling concurrent source code to Wasm shared functions, almost everything transitively needs to be marked as shared

So how can such code interact with JS?

JS interaction mechanism

Option 1: Proxy/dispatch threads

When a shared function wants to call into JS, it instead suspends and signals another JS thread to carry out the call and transmit the result back to the suspended thread

Signalling can be done in pure Wasm/JS through shared memories and wait/notify

Only works if the arguments and return value of the JS call can be serialised in linear memory

Some infrastructure for this already exists in Emscripten

If thread_id is the preferred TLS solution, and is a host import and not an instruction, this approach becomes incredibly messy

JS interaction mechanism

Option 2: JS context forwarding

In the "shared continuations don't exist" design for shared, can pass in a JS callback object as an argument at the JS/Wasm boundary, and propagate it through all calls in that thread

Simple to understand, but requires the unfavoured version of "shared"

A little messy and non-local for toolchains?

JS interaction mechanism

Option 3: Blessed callback

Extension to the JS API

A single JavaScript function that is importable as shared – accepts a "key" as an argument - "sharedCallback"

On the JS side, define a thread-local callback table which maps "keys" to JS functions. When sharedCallback is called, use the "key" to look up the JS function in the currently executing thread's callback table

JS interaction mechanism

Thread 1:

```
WebAssembly.callTable[0] = console.log

instantiate(m, {JS: WebAssembly.blessedCB})

…

m=

(module
  import "JS" "blessedCallback" $cb

…

(func shared $foo

  (call $cb (i32.const 0))...
```

Thread 2:

```
// get $foo through postMessage


// call $foo

// ERROR: callTable[0] not defined!
```

JS interaction mechanism

Option 3: Blessed callback

The most general and resilient solution (of the 3!)

Requires non-trivial changes to the JS API

Other orthogonal Threads post-MVP extensions

Weaker atomics such as release/acquire

Integration with the GC proposal

Integration with Stack Switching (*)

Vote

- Recall the phase 1 entry requirements (link):
  - There is general interest within the CG in this feature
  - The CG believes the feature is in-scope and will plausibly be workable
- Work to do during phase 1:
  - ☑ Create a repository (use thread-spawn)
  - ☑ Produce overview document
  - ⏳ Prototype implementations (WIP: LLVM, wasm-tools, Wasmtime)
  - ❓ Achieve broad consensus
- Poll: do you support moving this proposal to phase 1?