Shared Memory in Wasmtime

Andrew Brown, Alex Crichton

May 2022

Motivation

- The threads proposal has three main parts:
  - Atomic instructions (e.g., `i32.atomic.rmw.add`)
  - `wait`/`notify` instructions
  - Shared linear memory
- This is enough for implementing threads with Web Workers in the browser
- Wasmtime is missing the shared memory part and implementations for `wait`/`notify`—shared memory is needed first
- Current discussions about “WASI threads” rely on shared memory
- Future plans about “pure Wasm threads” rely not only on shared memory, but also on shared tables, globals, functions

Two Sides to Shared Memory

- Created externally

```
let wat = r#"
    (module (import "env" "memory" (memory 1 5 shared)))
"#;
let mut config = Config::new();
config.wasm_threads(true);
let engine = Engine::new(&config)?;
let module = Module::new(&engine, wat)?;
let mut store = Store::new(&engine, ());
let shared_memory = SharedMemory::new(&engine,
    MemoryType::shared(1, 5))?;
let memory = Memory::from_shared_memory(&mut store,
    &shared_memory)?;
let instance = Instance::new(&mut store, &module,
    &[memory.into()])?;
```

- Created internally

```
let wat = r#“
    (module (memory (export "memory") 1 5 shared))
"#;
let mut config = Config::new();
config.wasm_threads(true);
let engine = Engine::new(&config)?;
let module = Module::new(&engine, wat)?;
let mut store = Store::new(&engine, ());
let instance = Instance::new(&mut store, &module, &[])?;
let shared_memory = instance
    .get_memory(&mut store, "memory")
    .unwrap()
    .into_shared_memory()?;
```

The allocated memory must be share-able both internally and externally

The Problem

- Internally, both the Wasmtime runtime and Cranelift codegen depend on a specific representation of memory metadata
  - shared memory's `base` will never change (allocated to maximum)
  - but the `current_length` must be shared between all uses of the shared memory
  - `current_length` must be atomically incremented

```
struct VMContext {
    …
    memories: [struct VMMemoryDefinition {
        base: *mut u8,
        current_length: usize,
    }]
}
```

New design needed: we can't have a separate `VMMemoryDefinition` for each shared memory use

Solution #1

- Use a `union` to sometimes insert the owned structure (non-shared memory) and other times insert a pointer to the structure (shared memory)—many code changes
- The shared `VMMemoryDefinition` would be owned in the runtime `SharedMemory` structure

```
struct VMContext {
    …
    memories: [union VMMemoryUnion {
        shared: *mut VMMemoryDefinition,
        owned: VMMemoryDefinition,
    }]
}
```

```
struct SharedMemory (Arc<struct Inner {
  mem: RwLock<Box<dyn …>>,
  ty: …,
  def: VMMemoryDefinition,
}>)
```

Solution #2

- Create a new table for owned `VMMemoryDefinitions`; convert the existing table to `*mut VMMemoryDefinition`—many code changes
- As before, the shared `VMMemoryDefinition` would be owned in the runtime `SharedMemory` structure

```
struct VMContext {
    …
    memories: [*mut VMMemoryDefinition],
    owned_memories: [VMMemoryDefinition]
}
```

```
struct SharedMemory (Arc<struct Inner {
  mem: RwLock<Box<dyn …>>,
  ty: …,
  def: VMMemoryDefinition,
}>)
```

Solution #3

- Move memory definitions out of the instance (`VMContext`) entirely

Other considerations

- `VMMemoryImport` as a `*mut VMMemoryDefinition`—is this even possible?
- Switch `memory.size` to a host call to use `SharedMemory`’s locks

Questions