wasm_zero and developing with WASM targets

WASM targets

Each target has its own use case

- wasm32-unknown-unknown
- wasm32-unknown-ecmascripten -> similar to above, emits js glue code
- wasm32-wasi/wasm32-wasip1 -> first preview of components
  wasm32-wasip1-threads -> on top of wasip1 but threads
- wasm32-wasip2 -> http, async IO on top of wasip1
- wasm32-wali-linux-musl -> Linux syscalls in WASM!
- wasm64-unknown-unknown

Problem statement

A crypto wallet that’s entirely on the frontend browser

- Spawn a web worker on the browser
- Instantiate your crypto webassembly logic inside the web worker
- Call functions like, generate_private_key from bip39 mnemonic, keep seed in the sandboxed memory.
- Now call functions like get_balance, transfer that gives you blob tx data, ready to be sent to the node network
- Drop the worker, wasm memory is destroyed alongside with it

```rust
/// Calculate the balance info for the phoenix address at the given index for
/// the given seed.
#[unsafe(no_mangle)]
pub unsafe extern "C" fn balance(
    seed: &Seed,
    index: u8,
    notes_ptr: *const u8,
    balance_info_ptr: *mut [u8; 16],
) -> ErrorCode {
    let vk = derive_phoenix_vk(seed, index);

    let notes: Vec<NoteLeaf> = mem::from_buffer(notes_ptr)?;

    let info = balance::calculate_unchecked(&vk, notes.iter());

    ptr::copy_nonoverlapping(
        info.to_bytes().as_ptr(),
        &raw mut (*balance_info_ptr)[0],
        16,
    );

    ErrorCode::Ok
}
```

wasm32-unknown-unknown

A limited target

https://github.com/dusk-network/rusk

Task-Based Bulk Actions

To efficiently handle bulk actions, such as when multiple functions need to access the same memory, or to manage more complex tasks, it's recommended to encapsulate these operations within a defined task:

```
let task = module.task(async ({ yourFunctionA, yourFunctionB }) => {
  let result1 = await yourFunctionA(arg1);
  let result2 = await yourFunctionB(arg2);

  return { result1, result2 };
});

let results = await task();
```

Memory Management

Direct Shared Memory Access

Directly manipulate shared memory for high-performance scenarios:

```
const task = module.task(async ({ malloc, free, byte, set_byte }, { memory }) => {
  let ptr = await malloc(size);
  let buffer = new Uint8Array(memory.buffer, ptr, size);

  // Perform operations on the buffer

  await free(ptr, size);
});
```

Run WebAssembly code as parallel, isolated and abortable functions in JS engine

https://github.com/dusk-network/exu

Crossing the wasm -> language boundary

This is the FFI boundary

- Call malloc in rust -> returns ptr + len then
- use that in the rust side in the allocated memory region, serialize a string/ integer.
- Decode string/integer in javascript side

Zero copy serialisation on the rust side

- rkyv allows zero copy serialisation + alignment for types that support it only on the rust side.
- Generate the type shape at least at compile time during wasm byte code generation so javascript side has an idea of what to deserialise from raw bytes using pointer arithmetic / Text Decoder calls. This is similar to the glue wasm-bindgen emits.
- We can use FFI on javascript side with rkyv to decrease deserialisation cost (theoretical).

wasm_zero

rkyv ZC serialisation on the rust side. Only emit minimal decoding glue on js side

```rust
#[wasm_zero]
pub fn get_adult_person() -> Person { /* ... */ }
```

```javascript
import { initWasmZero } from "./pkg/bindings.js";

const wasmzero = await initWasmZero("app.wasm");
const person = wasmzero.get_adult_person(); // -> { name, age, email, scores }
```

Less wasm bundle size overall emitted
faster decoding of primitives and complex objects
support wasi target. No-std/std both supported. Use your own allocator!

Exu + wasm_zero (rayon with exu)

```
import { Module } from "./exu/src/mod.js";

const module = new Module(new URL("wasm_zero_rayon.wasm", import.meta.url));
module.defaultImports = new URL("imports.js", import.meta.url);

// Run a task: fresh sandbox + memory, dropped when this resolves (sandboxing).
const result = await module.task(async (exports, { initThreadPool }) => {
  // Spin up a rayon pool over this sandbox's shared memory.
  const threads = await initThreadPool(navigator.hardwareConcurrency);
  const primes = await exports.__wasm_zero_parallel_count_primes(1_000_000);
  return { primes, threads };
})();

console.log(result); // { primes: 78498, threads: <cores> }
```

Wasip3 Target + Tokio + Toasty ORM on Spin

- Spin SDK allows writing http components

Last frontier: Engineer your own ABI

ABI compatibility is important for WASM modules

- WASM modules are portable
- Write them in C/C++/Rust/Go, instantiate them together
- No extra code for ABI compatibility reducing wasm sizes