Wasm RPC thoughts

https://brionv.com/log/2019/05/10/wasm-rpc-thoughts/

Foreign functions attached as imports or table entries can be called with any signature, combining scalar values such as int and floats:

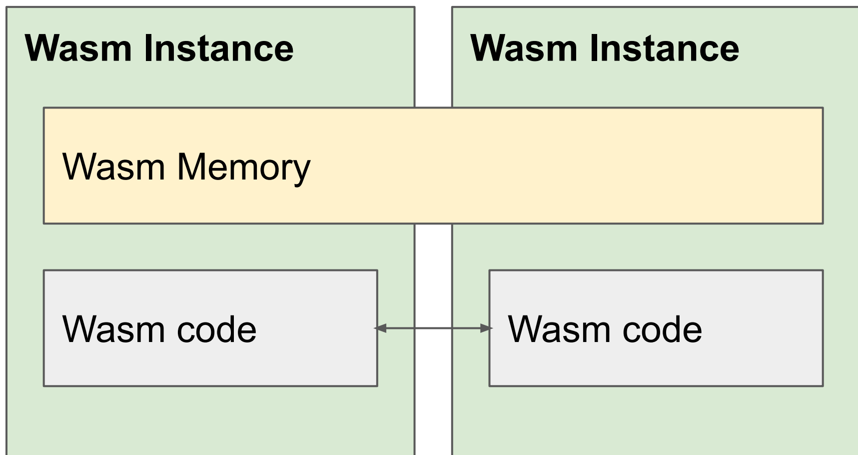
Plugin Wasm

```
canvas_set_size(640, 480);
```

Host Wasm

```
void canvas_set_size(int w, int h) {
    global_state.w = w;
    global_state.h = h;
}
```

This "just works" until you discover "pointers" and "structs"...

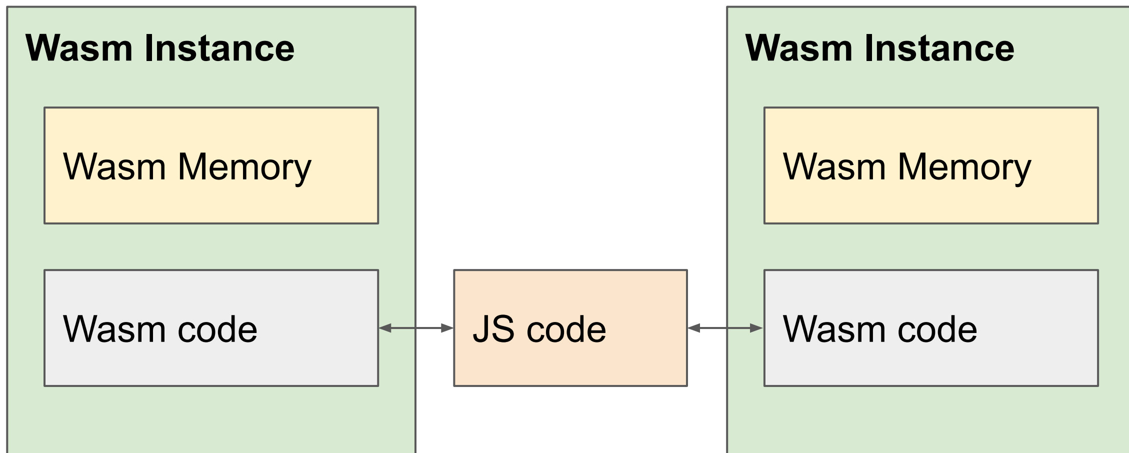


When **dynamic linking**, instances can share a memory and send pointers directly to each others' functions. But, they can access beyond the bounds of provided pointers or violate lifetime constraints.

This is unsuitable for isolation of untrusted plugins.

Instead we use instances with separate memories, providing **address space separation**.

A common JS "kernel" serves both Wasm instances with function imports as privileged "syscalls".



If you use pointers only as opaque **handles** accessed through functions, you could keep directly sending them:

Plugin Wasm

```
canvas_t* c = canvas_new();
canvas_set_size(c, 640, 480);
```

Host Wasm

```
canvas_t* canvas_new(void) {
    return malloc(sizeof(canvas_t));
}

void canvas_set_size(canvas_t* c, int w, int h)
{
    c->w = w;
    c->h = h;
}
```

But this sidesteps the memory protections -- the other module may pass any value for the pointer, accessing arbitrary memory in your module.

If we abstract the handles to **capability** references, then things get interesting.

Instead of directly passing around pointers, we use a level of indirection to map between capability handles and local pointers.

This requires an intermediary call on the import to translate values -- a capability must be unforgeable to be secure, so `cap_t` values must be translated between each module's namespace.

Plugin Wasm

```
cap_t handle = canvas_new();
```

Kernel JS

```
function canvas_new() {
    return mapping.recvCap(
        mapping.original()
    );
}
```

Host Wasm

```
cap_t canvas_new(void) {
    canvas_t *c = malloc(sizeof(canvas_t));
    return handle_create(c, canvas_class);
}
```

Each function mapping that includes caps transfer requires a signature-specific translator function in the kernel.

This can be created dynamically optimized for each signature, or a generic function using rest/spread args and an array traversal.

Signatures must be specified out-of-band for link-time imports, which requires metadata that's in sync with your code's ABI.

Note that when receiving a locally-owned cap, we must validate it through a syscall to get the original, unforgeable pointer.

Plugin Wasm

```
canvas_set_size(handle, 640, 480);
```

Kernel JS

```
function canvas_set_size(a, b, c) {
    a = mapping.sendCap(a);
    mapping.original(a, b, c);
}
```

Host Wasm

```
void canvas_set_size(cap_t handle, int w, int h)
{
    canvas_t* c = handle_badge(handle,
                               canvas_class);
    if (c) {
        c->w = w;
        c->h = h;
    }
    cap_release(handle);
}
```

Or, we can simplify translation functions by sending parameters, both caps and arguments, out of band, itself through a cap.

This feels weird but doesn't require telling the kernel about your argument list structure. You probably had to write nice C/Rust wrapper APIs anyway so… not so bad probably?

Downside is every arg requires transfer through linear memory, and you have to play with a lot of structs.

Plugin Wasm

```
canvas_set_size_args_t args = {
   .width = 640, .height = 480
};
cap_t msg = msg_create(
    &args, sizeof(args), // send data
    &handle, 1,          // send caps
    NULL, 0,             // recv data
    NULL, 0              // recv caps
);
canvas_set_size(msg);
msg_free(msg);
```

Host Wasm

```
void canvas_set_size(cap_t msg)
{
    canvas_set_size_args_t args;
    cap_t handle;
    msg_recv(msg, &args, sizeof(args), &handle, 1);
    canvas_t* c = handle_badge(handle,
                               canvas_class);
    if (c) {
        c->w = args.width;
        c->h = args.height;
        cap_release(handle);
    }
}
```

A limited number of fixed integer args and an integer retval could be added for efficiency on common cases with buffers passed explicitly when needed.

Only one translation function would be required, accepting that fixed number of arguments and translating only the message cap.

This is similar to how messages on L4 are optimized to use register-passing for the first few integer arguments.

It feels inelegant though, as floating point and caps still have to go through the message.

Plugin Wasm

```
cap_t msg = msg_create(
    &handle, 1,          // send caps
    NULL, 0              // recv caps
);
canvas_set_size(msg, 640, 480, 0, 0);
msg_free(msg);
```

Host Wasm

```
int canvas_set_size(cap_t msg, int a, int b, int c, int d)
{
    cap_t handle;
    msg_recv_caps(msg, &handle, 1);
    canvas_t* c = handle_badge(handle,
                               canvas_class);
    if (c) {
        c->w = a;
        c->h = b;
        cap_release(handle);
    }
    return 0;
}
```

Capability namespacing requires some thought.

Use a JS array of object references (or multiple JS arrays with synchronized pointers to minimize object allocations) indexed by the cap_t value as an expandable stack.

Index 0 is reserved for CAP_NULL.

Released caps are replaced with a dead-value indicator in the JS arrays, and their index is added to a second stack.

When allocating or receiving a new cap, the most recently freed cap index is popped from the released stack and the dead-value indicator is replaced with the new object.

This prevents the caps namespace from growing without end when sending/receiving many messages, and keeps the re-allocation search constant-time at the expense of some memory.

Caps stack

```
0 = null
1 = handle<0x12345678>
2 = sendbuf<0x23456789, 32>
3 = released
4 = released
5 = handle<0x34567891>
```

Released stack

```
0 = 4
1 = 3

// next index to be reused
// will be 3
```

Isolation guarantees

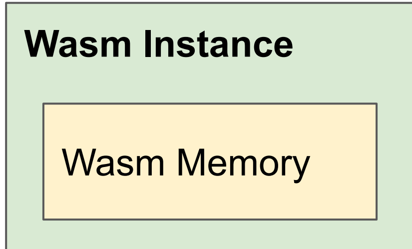
Yay:

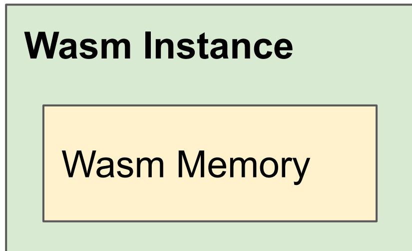
- Memory usage can be bounded.
- Modules can only alter each other's memory with a grant.
- Modules can only call each other via imports granted to them at setup or dynamically added by the kernel.
- Traps/exceptions in a sync call can be caught at the syscall boundary.

But:

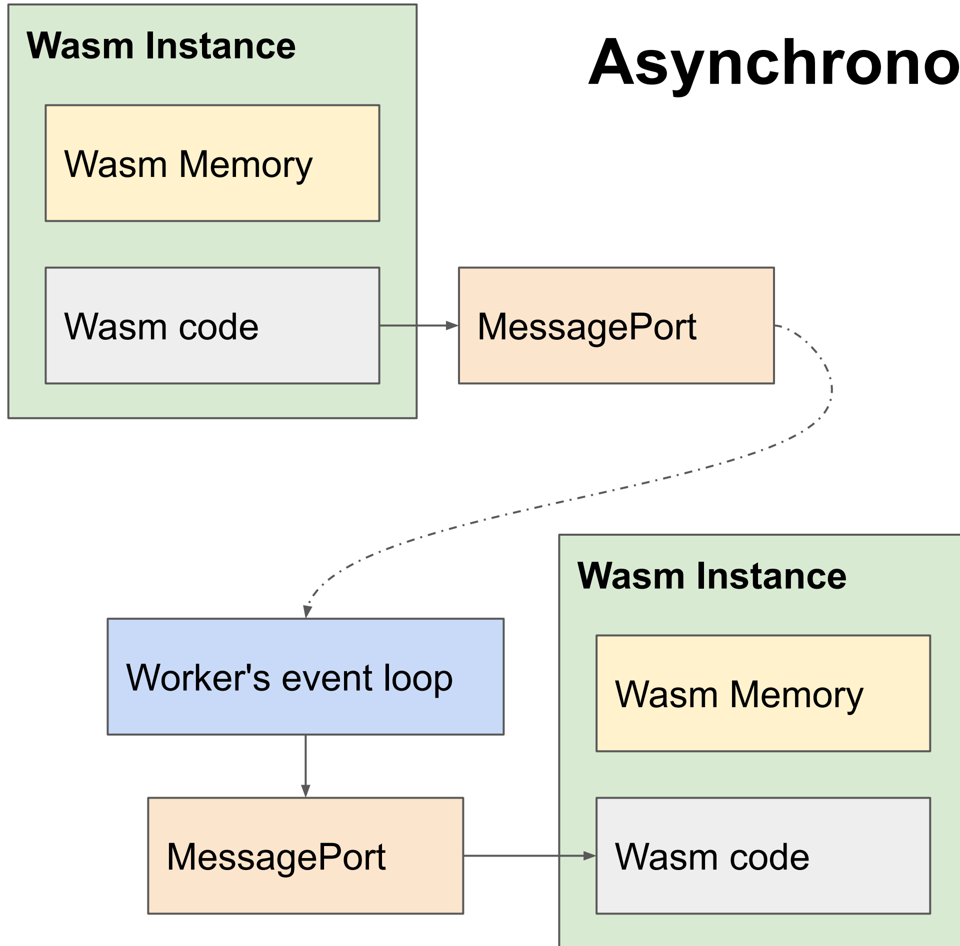
- Sync calls may never yield control of CPU.
- Incoming sync calls may be close to the stack limit, which may cause you to trap when innocently recursing.
- Wasm traps will cause the execution stack to unwind to the syscall boundary, but will not run destructors, finally clauses, or other cleanup code in your C/C++/Rust -- not even fixing the linear-memory stack pointer!

Traps might be handled by killing the offending module, or maybe killing the entire process.





Asynchronous worker extensions



Browsers support asynchronous event-loop messaging between separate CPU threads running in **Web Workers**.

Sync calls are not possible with this model, but other capabilities such as **async message ports** and buffer transfers with an intermediate copy can be done.

Note that some browser APIs such as network access and compiling new WebAssembly modules require a run through the event loop as well.

Multithreading extensions

Some browsers have re-enabled JavaScript's **SharedArrayBuffer** and the WebAssembly extension to use them as memories.

This allows **multithreaded** WebAssembly modules by creating the same module graph in multiple Workers, each instance sharing a memory with its co-instances in other threads but still separate from other modules' instances.

Sync calls across modules in separate threads/processes would also be possible, using suitable locks.

But beware! You must still exit to the event loop to obtain new WebAssembly module instances, which can lead to thread synchronization issues when linking new modules.

Next steps?

Get it down on paper:

- Decide on metadata format for signature translation, or whether to stick with message buffers.
- Code up C and Rust APIs for syscall interface and a couple sample APIs using it
- Proof-of-concept JS kernel for running in browser

Possible examples?

- API to the JS kernel to dynamically load new Wasm modules and populate them with imports
- Photo/paint program with filter plugins or file format translators
- Fractal viewer with calculation plugins

End.. for now!