PDF-TOOLS.COM
Premium PDF Technology

Dynamic Loading of Modules in WASM/Emscripten

Philipp Gloor

Table of Contents

Passing values: Reference or Value

- Passing objects created in JavaScript into Emscripten copies the objects
- Passing objects created within Emscripten via embind API obey the pass by value and pass by reference convention

Emscripten: Main Module, Side Modules I

In order to be able to dynamically load a WASM file they need to be compiled with specific options.

Following options can control how Emscripten creates the WebAssembly module:

- `-s MAIN_MODULE`
- `-s SIDE_MODULE`

The differences between the main module and the side modules:

Emscripten: Main Module, Side Modules II

- Main modules have the system libraries linked in
- Side modules are pure wasm files that contain LLVM bitcode that have no system libraries
- Side modules rely on the main module linking neede system library functions
- Compiling a main module adds a JavaScript file which sets up the Emscripten environment

Dynamic Loading I

Three different methods to dynamically link functions:

- dlopen/dlsym: Works the same as with native C code. Requires the side module to be available in the file system. Relies on C-Linkage or using mangled C++ names. Obtain dll handle with `dlopen` and link the functions with `dlsym`
- On startup: Define `Module.dynamicLibraries = [<list of wasm names>]`. Loads the wasm modules synchronously. Functions from modules need to be declared for compilation to work.
- On demand: Use `EM_ASM()` macro within C/C++.

Dynamic Loading II

Using `loadDynamicLibrary` from within C/C++ seems the best way

1. Loads WASM when really needed
2. Automatic linking (unlike dlopen)
3. Asynchronous loading of modules

loadDynamicLibrary I

- C++/C function that loads the side modules asynchronously

```
void loadLib()
{
  EM_ASM({
    loadDynamicLibrary('output/side_module.wasm', {loadAsync: true, global: true, nodelete: true}).then(
      () => {
        Module.modulePromiseResolve()
      });
    loadDynamicLibrary('output/side_module2.wasm', {loadAsync: true, global: true, nodelete: true}).then(
      () => {
        Module.module2PromiseResolve()
      });
    loadDynamicLibrary('output/dynamic.wasm', {loadAsync: true, global: true, nodelete: true}).then(
      () => {
        Module.dynamicPromiseResolve()
      });
    });
}
```

loadDynamicLibrary II

- Define a promise for each module which the promise in the EM_ASM call can resolve or reject

```
Module.modulePromise = new Promise( (resolve, reject) => {
  Module.modulePromiseResolve = resolve
  Module.modulePromiseReject = reject
});
... // same for module2Pomise and dynamicPromise

Module['onRuntimeInitialized'] = () =>
{
    console.log('onruntimeinitialized');
    Promise.all([Module.modulePromise, Module.module2Promise, Module.dynamicPromise]).then(
      () => {
        console.log("all modules loaded");
        runModuleFunctions();
    })
}
```

Performance for Calls Between Modules

- -O2 for LLVM bc and wasm/js output
- embind for JS API

`http://tofino/pgl/WebAssemblyTests`

Notes

- Development of Dynamic Linking is still ongoing and improvements are still to be expected:
  `https://github.com/emscripten-core/emscripten/projects/2`