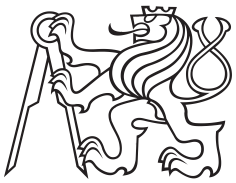

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická

Využití WebAssembly pro webové aplikace

Bc. Bedřich Schindler

Vedoucí práce: RNDr. Ondřej Žára
Leden 2022

Poděkování

Rád bych poděkoval vedoucímu práce RNDr. Ondřeji Žárovi za možnost vypracovávání tohoto tématu a za pomoc při sestavování specifikace semestrální práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu a zdroje, a to v souladu s Metodickým pokynem č. 1/2009 o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 10. ledna 2022

Abstrakt

Cílem tohoto semestrálního projektu je seznámení se s datovým formátem WebAssembly, popis problémů, které řeší a jaké k němu existují alternativy. Dále demonstruje všechny kroky nezbytné ke kompilaci knihovního kódu v jazyce C/C++ do WebAssembly a následného využití ze strany klientského JavaScriptu a porovnává toto řešení s alternativními přístupy. Na dvou existujících knihovnách poté ukazuje výše uvedený postup a popisuje problémy, které mohou při takové konverzi nastat.

Klíčová slova: WebAssembly, webové aplikace

Vedoucí práce: RNDr. Ondřej Žára

Abstract

The aim of this term thesis is to get acquainted with the data format of WebAssembly, a description of the problems it solves and what alternatives there are. It also demonstrates all the steps necessary to compile C/C++ library code into WebAssembly and subsequent use by JavaScript client and compares this solution with alternative approaches. The two existing libraries then show and describe the problems that can occur with such a conversion.

Keywords: WebAssembly, web applications

Title translation: Use of WebAssembly for web applications

Obsah

1 Úvod **1**
1.1 Motivace 1
1.2 Možnosti řešení 1
1.3 Cíl 2

2 WebAssembly **3**
2.1 Definice 3
2.2 Vlastnosti 3
2.3 Formát 4
2.4 Kompilace 4
2.5 Koncepce použití ve webovém prohlížeči 5
2.6 Alternativy 6

3 Využití ve webových aplikacích **7**
3.1 Prostředí 7
3.2 Základní použití 7
3.2.1 Část C++ 7
3.2.2 Část webové aplikace 9
3.2.3 Shrnutí 11
3.3 Pokročilé použití s knihovnou třetí strany 12
3.3.1 Část C++ 12
3.3.2 Část webové aplikace 15
3.3.3 Shrnutí 18
3.4 Další pokročilé ukázky použití . . 19

4 Závěr **21**

Literatura **23**

Obrázky

Kapitola 1
Úvod

1.1 Motivace

V rámci vývoje webových rozhraní pro síťové prvky, kterému se posledních několik let věnuji, jsem se postupně začal setkávat s požadavky na implementaci specifických standardů, jejichž použití by na klientské straně vyžadovalo buď přispívání do knihoven s otevřeným zdrojovým kódem, omezení implementace pouze na dostupné funkcionality zvolené knihovny nebo implementaci vlastního řešení.

Vzhledem k tomu, že backend tyto knihovny typicky buď používá nebo implementuje vlastní řešení, se zdá nejlepším řešením využít tyto knihovny i na klientské straně v rámci webového rozhraní a ušetřit tím jak čas, tak finanční prostředky na implementaci stejné funkcionality v dalším jazyce.

Pro podobné případy, kdy by bylo nezbytné implementovat složitou funkcionalitu, která již existuje v jiném jazyce, je možné použít WebAssembly, a proto jsem se rozhodl demonstrovat použití C/C++ knihoven v rámci webových aplikací pomocí WebAssembly.

1.2 Možnosti řešení

Vzhledem k tomu, že se jedná o pokročilou problematiku, se předpokládá znalost webových technologií jako je HTML a JavaScript. Pro potřeby této práce je dále nezbytná základní znalost jazyků C a C++, jež budou použity pro demonstraci kompilace knihoven z C/C++ do WebAssembly.

Ačkoliv je možné do WebAssembly kompilovat i jiné jazyky, kterými jsou např. Python, Go, nebo Rust, jsem pro účely této práce zvolil jazyky C a C++ (označovány společně jako C/C++). Ty jsou pro kompilaci do WebAssembly využívány nejčastěji. Kompilátor pro tyto jazyky má nejpokročilejší implementaci a co především, tak je v C/C++ napsána řada knihoven, které se můžou hodit (např. pro práci se síťovými standardy, zpracování obrazu a zvuku apod.).

1.3 Cíl

Cílem práce je seznámit vás s technologií WebAssembly, popsat problémy, které řeší a jaké k němu existují alternativy. Nejprve budou představeny nezbytné kroky ke kompilaci kódu v jazyce C/C++ do WebAssembly a jeho následné použití ze strany webové aplikace pomocí Javascriptu.

Vše bude demonstrováno na jednoduché webové aplikaci a C/C++ knihovnách, na níž budou popsány kroky implementace a problémy, které mohou nastat.

Kapitola 2
WebAssembly

2.1 Definice

WebAsssembly, zkráceně WASM, je nízkoúrovňový jazyk s kompaktním binárním formátem navržený pro spuštění v moderních webových prohlížečích. Jeho hlavním cílem je umožnit provádění vysoce výkonného kódu ve webovém prohlížeči s výkonem podobným nativnímu strojovému kódu. WebAssembly se využívá především jako cíl kompilace z vysokoúrovňových programovacích jazyků jako jsou C/C++, Python, Go nebo Rust.

2.2 Vlastnosti

WebAsssembly je navrženo s důrazem na následující vlastnosti [1]:

- **Rychlost** - Kód se spouští s výkonem blízkým nativnímu kódu.
- **Bezpečnost** - Kód se ověřuje a spouští v izolovaném prostředí, které zabraňuje narušení bezpečnosti.
- **Hardwarovová nezávislost** - Může být zkompilován pro všechny moderní architektury, a to jak pro desktop, mobilní zařízení nebo vestavěné systémy.
- **Jazyková nezávislost** - Neupřednostňuje žádný konkrétní programovací jazyk či model. Díky kompilátorům z vysokoúrovňových jazyků může být do WebAssembly zkompilována celá řada jazyků.
- **Platformní nezávislost** - Ačkoliv je WebAssembly spouštěn převážně ve webových prohlížečích, tak je platformně nezávislý a může být spouštěn i v dalších prostředích podporující WebAssembly (např. NodeJS).
- **Kompaktnost** - Má binární formát, který umožňuje rychlejší přenos díky jeho menší velikosti v porovnání s textovými formáty.
- **Modularita** - Může být rozdělen na menších části, které mohou být přenášeny, cachovány a zpracovány nezávisle.

- **Streamovatelnost** - Umožňuje dekódování, validaci a kompilaci ještě před tím než jsou k dispozici všechna data.
- **Paralelizace** - Umožňuje dekódování, validaci a kompilaci provádět paralelně.

2.3 Formát

Ačkoliv bylo v definici uvedeno, že WebAssembly je binárním formátem, tak nelze opomenout, že existuje i jeho textová varianta [2], jež je čitelná člověkem.

Základem binárního i textového formátu WebAssembly je modul. Modul je v textovém formátu zapisován pomocí symbolických výrazů. Symbolické výrazy[3], tzv. s-výrazy, jsou speciálním zápisem strukturovaných dat, konkrétně stromové struktury. V rámci modulu lze definovat jeho části jako např. importy, exporty, funkce, lokální a globální proměnné atd.

WebAssembly mimo jiné definuje čtyři číselné datové typy, které lze v zápisu používat. Datovými typy reprezentující celá čísla jsou `i32` a `i64`, dále pak typy reprezentující reálná čísla `f32` a `f64`.

Funkce jsou ve WebAssembly zapisovány v takovém formátu, aby odpovídaly zásobníkovému stroji.

Pro představu si ukážeme, jak by mohl vypadat jednoduchý WebAssembly modul zapsaný v textovém formátu, který exportuje funkci `addNumbers`, která sčítá dvě celá čísla:

```
(module
  (func $addNumbers (param $a i32) (param $b i32) (result i32)
    local.get $a
    local.get $b
    i32.add)
  (export "addNumbers" (func $addNumbers))
)
```

Soubory s WebAssembly kódem v textovém formátu využívají přípony `.wat`. Takovýto soubor lze převést do binárního formátu pomocí nástroje `wat2wasm` [4], který vytvoří binární soubor s příponou `.wasm`, jehož lze spustit např. ve webovém prohlížeči.

I když je možné využít textového formátu, díky kterému lze sestavit libovolný modul, se však tento zápis příliš nepoužívá a do binárního formátu WebAssembly se kompilují kódy z vysokoúrovňových programovacích jazyků.

2.4 Kompilace

Jak již bylo uvedeno v předchozí kapitole, pro překlad textového formátu WebAssembly (zkráceně WAT) do jeho binárního formátu (zkráceně WASM) se využívá nástroje `wat2wasm`.

Ve většině případů je WebAssembly využíváno vyššími programovacími jazyky, které do WASM kompilujeme pomocí příslušných kompilátorů, které jsou uvedené v následujícím seznamu podle jazyků [5]:

- **C/C++** – Pro kompilaci C/C++ se používá kompilátoru Emscripten, který je založení na LLVM/Clang. Jedná se nejpoužívanější kompilátor do WASM.
- **Rust** – Pro kompilaci Rustu stačí využít balíčkovacího nástroje Cargo, pomocí něhož je nezbytné nainstalovat balíček wasm-pack.
- **Go** – Go má na výběr hned dva nástroje, a to standardní Go nebo minimalistickou verzi TinyGo. Oba nástroje mají již vestavěnou podporu pro WebAssembly a je nezbytné nastavit pouze správný cíl kompilace na WASM.
- **Python** – Python není na oficiálním seznamu podporovaných jazyků, ale Pyodide umožňuje psán kód v Pythonu a pomocí portu na CPython využívá ke kompilaci do WASM již dříve zmíněný Emscripten.
- **Java** – Java též není na oficiálním seznamu podporovaných jazyků, ale existují zde projekty jako TeamVM nebo JWebAssembly, které lze pro kompilaci do WASM využít.

Existuje celá řada oficiálních i neoficiálních kompilátorů do WASM, ale vývoj v tomto odvětví je v současné době poměrně rychlý, a tak jsou zde uvedeny pouze kompilátory pro základní rozšířené jazyky. V rámci této práce se budeme později zajímat pouze o kompilátor Emscripten pro C/C++, který je nejpoužívanějším a nejoblíbenějším kompilátorem do WASM vůbec.

2.5 Koncepce použití ve webovém prohlížeči

V současné době se při tvorbě webových aplikací využívá JavaScript. JavaScript je vysokoúrovňový dynamicky typovaný jazyk, který umožňuje tvořit dynamický web, používá webová API prohlížečů (např. DOM, History API, Web Storage API, Service Workers API atd.) a díky rozsáhle celosvětové komunitě vývojářů nabízí širokou škálu frameworků a knihoven, kterou vývojáři používají při tvorbě webových aplikací. WebAssembly je vedle toho nízkoúrovňový jazyk, který využívá dříve uvedených vlastností a neslouží jako náhrada JavaScriptu, nýbrž jako jeho doplněk, aby mohla aplikace využívat z každé technologie to nejvhodnější.

Webový prohlížeč načítá a provádí dva nezávislé kódy, JavaScript a WebAssembly. WebAssembly kód je načten pomocí JavaScriptové funkce `fetch`. Pomocí funkce `WebAssembly.instantiate` či `WebAssembly.instantiateStreaming` z WebAssembly JavaScript API [6] je vytvořena instance WebAssembly kódu. Na instanci je možné volat exportované funkce WebAssembly kódu stejně, jako by se jednalo o kód napsaný v JavaScriptu. Pokud při vytváření instance

WebAssembly kódu definujeme funkce pro import do WebAssembly kódu, tak můžeme potom tyto JavaScriptové funkce volat z WebAssembly kódu.

Pro detailnější pochopení se podívejme na následující pojmy a objekty z WebAssembly JavaScript API:

- **Module** - Reprezentuje kód WebAssembly, který byl zkompilován do strojového kódu spustitelného ve webovém prohlížeči. Modul deklaruje exporty a importy. Dále poskytuje API pro základní práci s WebAssembly kódem, a to například funkce pro validování, kompilování či instanciování WebAssembly kódu.
- **Instance** - Reprezentuje instanci kódu WebAssembly, jež se vytváří na základě modulu.
- **Memory** - Je pole proměnné délky využívající JavaScriptový typ ArrayBuffer, obsahující lineární pole bajtů, které reprezentuje paměť instance WebAssembly kódu a které je dostupné pro čtení a zápis jak z JavaScriptu, tak z dané instance WebAssebmly kódu.
- **Table** - Je pole proměnné délky obsahující reference na funkce, které jsou dostupné pro čtení a zápis jak z JavaScriptu, tak z dané instance WebAssebmly kódu.
- **Global** - Představuje objekt globální proměnné, který je možný číst i zapisovat jak z JavaScriptu, tak z dané instance WebAssebmly kódu.

Výše uvedené je základní koncepce použití WebAssembly kódu ve webovém prohlížeči. Samotná implementace a komunikace mezi JavaScriptem a WebAssembly se může lišit podle zvoleného přístupu, zvoleného kompilátoru a softwarového pozadí aplikace. Jeden z možných přístupů bude popsán v dalších kapitolách.

2.6 Alternativy

Jedinou významnou alternativou k WebAssembly je asm.js [7]. Asm.js je striktní podmnožina JavaScriptu, do které je překládán kód napsaný staticky typovanými jazyky s manuální správou paměti, jako jsou např. jazyky C/C++. Asm.js umožňuje provádění kódu výrazně rychleji než je tomu u běžného JavaScriptu.

Pro překlad kódu se opět používá již dříve uvedený Emscripten, který umožňuje vedle WASM překládat kód v C/C++ i do asm.js. Mezi dalšími jazyky, které lze přeložit do asm.js je např. Rust, Python či Ruby.

Asm.js má oproti WebAssembly sice podporu u starších verzí prohlížečů Google Chrome a Mozilla Firefox, ale např. není vůbec podporován v prohlížeči Safari. Implementace a podpora se navíc ještě poměrně liší napříč prohlížeči. WebAssembly je oproti tomu plně podporován všemi moderními prohlížeči. Internet Explorer není podporován ani jednou z těchto technologií.

Kapitola 3

Využití ve webových aplikacích

3.1 Prostředí

Jak bylo již dříve uvedeno, v rámci této práce se budeme věnovat pouze kompilaci kódu z C/C++ do WebAssembly a jeho následnému použití ve webových aplikacích. Pro kompilaci C/C++ bude nezbytné nainstalovat nástroj Emscripten[1].

Emscripten je možné nainstalovat dvěma způsoby. Prvním způsobem je stažení zdrojového kódu a jeho instalace pomocí instalačních skriptů, které jsou dostupné u zdrojových kódu. Druhým způsobem je využití balíčkovacích systémů, které se liší na základě operačních systémů.

Vzhledem ke komplexnosti instalačního procesu navštivte oficiální stránky nástroje Emscripten a nainstalujte jej pomocí instrukcí uvedených na jejich webu.

Ačkoliv pro webové aplikace běžně není zapotřebí žádné speciální prostředí, tak v rámci testovací aplikace, kde demonstruji použití WebAssembly v praxi, však využívám NodeJS[2] a jeho balíčkovací systém NPM. Bez tohoto prostředí se sice můžete obejít, ale většina moderních aplikací tohoto prostředí využívá, a proto je vhodnější představit použití WebAssembly v praxi s pomocí současných nástrojů a trendů.

3.2 Základní použití

V této kapitole bude popsána kompilace jednoduchého kódu z C++ do WebAssembly a jeho využití ve webové aplikaci.

3.2.1 Část C++

Pro pochopení začneme s velmi primitivní ukázkou. V C++ implementujeme několik základních funkcí, přidáme direktivy specifické pro Emscripten a provedeme kompilaci do WebAssembly.

[1] Instalace Emscripten: https://emscripten.org/docs/getting_started/downloads.html
[2] NodeJS: https://nodejs.org/en/about/

V souboru `library.cpp` si implementujeme dvě funkce. Jedna funkce bude přijímat ukazatel na pole celých čísel a počet prvků daného pole, provede součet těchto prvků a vrátí výsledek. Druhá funkce funguje totožně, ale přijímá a vrací reálná čísla.

```
int arrayIntSum(int * inputArray, int inputArrayLength) {
    int sum = 0;
    for (int i = 0; i < inputArrayLength; i++) {
        sum += inputArray[i];
    }
    return sum;
}

float arrayFloatSum(float * inputArray, int inputArrayLength) {
    float sum = 0;
    for (int i = 0; i < inputArrayLength; i++) {
        sum += inputArray[i];
    }
    return sum;
}
```

Lze vidět, že jedná se o zcela obyčejné funkce napsané v C++. Pokud chceme tyto funkce vystavit pro WASM, tak máme dvě možnosti. První variantou je využití direktivy `EMSCRIPTEN_KEEPALIVE` [8], která zajistí, že dané funkce Emscriptem vystaví v rámci výsledného WASM. Druhá možnost, kterou si ukážeme v další ukázce, spočívá ve využití direktivy `EMSCRIPTEN_BINDINGS`, pomocí které můžeme vystavovat existující funkce a třídy.

Následující kód je třeba vložit do již zmiňovaného souboru `library.cpp`, a to před výše definované funkce.

```
#include <emscripten.h>

extern "C" {
    EMSCRIPTEN_KEEPALIVE
    int arrayIntSum(int * inputArray, int inputArrayLength);
    EMSCRIPTEN_KEEPALIVE
    float arrayFloatSum(float * inputArray, int inputArrayLength);
}
```

Výše uvedený kód musíme zkompilovat pomocí Emscripten, konkrétně pomocí příkazu `em++` [9]. Tomuto příkazu předáme nejprve název souboru, který chceme zkompilovat, tedy `./src/library.cpp`. Dále musíme uvést cestu a příponu výstupních souborů, v našem případě `-o ./build/wasm.js`.

V tomto případě je třeba upřesnit, že máme na výběr tři možnosti výstupu. Pokud zvolíme soubor s příponou `.wasm`, bude nám vygenerován samotný WASM soubor. Pokud zvolíme soubor s příponou `.js`, bude nám vygenerován jak WASM soubor, tak JavaScriptový soubor, který bude obsahovat řadu

pomocných funkcí pro obsluhu WASM souboru. V případě přípony `.html` nám bude ještě vygenerován HTML soubor, který bude načítat jak WASM, tak JavaScriptový soubor. Protože si chceme většinou však implementovat vlastní HTML soubor, volíme soubor s příponou `.js`. Parametr `-no-entry` určuje, že nemáme v rámci kódu implementovanou funkci `main`.

Dalším nezbytným parametrem je `NO_DISABLE_EXCEPTION_CATCHING`, tím aktivujeme zachytávání výjimek. Parametrem `ENVIRONMENT=web` určíme, že je cílovým prostředím web, čímž bude výstup optimalizován pro webové prostředí a nebude obsahovat konstrukty potřebné pro další prostředí. Parametrem `MODULARIZE=1` určíme, že nechceme instanci WebAssembly modulu exportovat globálně, ale že chceme vytvořit tovární funkci, která bude vytvářet instanci modulu. Parametrem `EXPORT_ES6=1` určíme, že chceme JavaScriptový kód vygenerovat ve standardu ECMAScript 2015, neboli ES6. A v neposlední řadě nastavíme parametr `ALLOW_MEMORY_GROWTH=1`, díky kterému umožníme dynamicky měnit velikost paměti, pokud její přesnou velikost v čase kompilace neznáme.

Parametry, jež nám umožňují exportovat řadu funkcí, které nám Emscripten nabízí, jsou `EXPORTED_FUNCTIONS` a `EXPORTED_RUNTIME_METHODS`.

Konkrétní příkaz pro kompilaci ukázkového souboru vypadá následovně:

```
em++ ./src/library.cpp \
  -o ./build/wasm.js \
  --no-entry \
  -s NO_DISABLE_EXCEPTION_CATCHING \
  -s ENVIRONMENT=web \
  -s MODULARIZE=1 \
  -s EXPORT_ES6=1 \
  -s EXPORTED_FUNCTIONS="['_malloc', '_free']" \
  -s EXPORTED_RUNTIME_METHODS="['FS']" \
  -s ALLOW_MEMORY_GROWTH=1
```

Po kompilaci budeme mít dostupné soubory `wasm.wasm` a `wasm.js`, které použijeme v druhé části ukázky ve webové aplikaci.

Nutno podotknout, že příkaz `em++` nabízí celou řadu parametrů, která zde není uvedena. V závislosti na implementaci WebAssembly do vaši aplikace se mohou lišit i výše popsané a uvedené parametry. Avšak z praktického pohledu se zdá podobná sada parametrů velmi užitečnou.

3.2.2 Část webové aplikace

V této části se podíváme na integraci WebAssembly z pohledu webové aplikace. Jak bylo v úvodní kapitole avizováno, tak se od čtenáře předpokládá znalost webových technologií, a proto věci jako je vytváření HTML souboru budou vynechány a budou často používány pokročilé konstrukty a technologie, které JavaScript, resp. poslední verze ECMAScript, nabízí. V ukázkách se navíc předpokládá využití nástroje Webpack[3], který sdružuje JavaScriptové moduly

[3]Webpack: https://webpack.js.org/concepts/

a vytváří finální balík JavaScriptu pro prohlížeč.

Importování souboru `wasm.js` můžeme provést dvě způsoby. Prvním způsobem je do HTML souboru vložit `<script/>` tag s odkazem na uvedený soubor. Druhá možnost, které využijeme, je koncept modulů, který je u moderních JavaScriptových aplikací již standardem. Přes <script/> tag si importujeme JavaScriptový soubor `main.js`, který si sám importuje soubor `wasm.js` přes direktivu `import`.

V hlavním JavaScriptovém souboru naší aplikace si importujeme `wasm.js`, jehož defaultním exportem je tovární metoda, která vytvoří instanci WebAssembly modulu.

```
// main.js
import createWasmModule from './wasm';
self.WasmModule = await createWasmModule();
```

V globální proměnné `WasmModule` máme uloženou instanci, která má k dispozici řadu funkcí, které jsme exportovali. Obsahuje např. námi implementované funkce `WasmModule._arrayIntSum` a `WasmModule._arrayFloatSum`. Dále obsahuje funkce `WasmModule._malloc`, `WasmModule._free` a objekt `WasmModule.FS`, jež jsme exportovali pomocí parametrů kompilátoru `em++`.

Pokud bychom měli implementovanou funkci, která přijímá primitivní hodnotu, např. funkci `min(a,b)` vracející menší z uvedených čísel, tak takovou funkci můžeme zavolat přímo pomocí `WasmModule._min(10,20)`. Protože jsme však implementovali funkce, které přijímají jako první parametr ukazatel na pole a jako druhý parametr velikost tohoto pole, bude celá záležitost složitější.

Nejprve si budeme muset vytvořit pole celých čísel (pro druhou funkci pole reálných čísel). Pomocí exportované funkce `WasmModule._malloc` si alokujeme místo v paměti ve WebAssembly modulu a předáme paměti WebAssembly modulu naše pole čísel. Ukazatel na paměť spolu s velikostí pole předáme funkci `WasmModule._arrayIntSum` (resp. `WasmModule._arrayFloatSum`). Po získání výsledku musíme dealokovat paměť pomocí funkce `WasmModule._free`.

Pro přehlednější ukázku výše uvedené zapíšeme do dvou funkcí. Funkce `arrayIntSum` bude obalovat funkci `WasmModule._arrayIntSum` a bude řešit alokaci a dealokaci paměti. Funkce `createInt32Array` bude pomocná funkce pro práci s polem a jeho pamětí.

```
// functions/arrayIntSum.js
import createInt32Array from '../helpers/createInt32Array';
export default (integerArray) => {
  const arrayObject = createInt32Array(integerArray);
  const result = WasmModule._arrayIntSum(
    arrayObject.getPointer(),
    arrayObject.getArray().length,
  );
  arrayObject.freeMemory();
  return result;
};
```

```
// helpers/createInt32Array.js
export default (integerArray) => {
  const int32Array = new Int32Array(integerArray);
  const int32ArrayPointer = WasmModule._malloc(
    int32Array.length * int32Array.BYTES_PER_ELEMENT
  );
  WasmModule.HEAP32.set(
    int32Array,
    int32ArrayPointer / int32Array.BYTES_PER_ELEMENT
  );

  return {
    freeMemory: () => {
      WasmModule._free(int32ArrayPointer);
    },
    getArray: () => int32Array,
    getPointer: () => int32ArrayPointer,
  };
};
```

Obdobný postup bychom zvolili pro funkci `WasmModule._arrayFloatSum`. Pokud bychom v rámci aplikace chtěli volat jednu z těchto funkcí, tak zavoláme námi obalené funkce následujícím způsobem.

```
// main.js
import arrayIntSum from './functions/arrayIntSum.js;
import arrayFloatSum from './functions/arrayFloatSum.js';

const iSum = arrayIntSum([10, 20, 30]);
const iFloat = arrayFloatSum([2.5, 5, 12.5]);
```

3.2.3 Shrnutí

Výše uvedená ukázka je jeden z mnoha přístupu, který lze aplikovat při používání WebAssembly. Jak C++ část, tak část webové aplikace, mohou být implementovány celou řadou způsobů.

C++ část, která byla uvedena výše, je jedna z nejzákladnějších implementací a je možné ji použít pouze pro implementaci vlastního kódu. Její výhodou je jednoduchá implementace. Pro implementaci větší vlastní knihovny nebo pro portování knihovny třetí strany, je však tento postup nedostatečný a je nezbytné zvolit vhodnější techniku, kterou si představíme v další kapitole.

Webová část, která byla uvedena výše, má taktéž řadu možných implementací, ale výše uvedená by mohla být dostačující pro řadu použití. Její nevýhodou je synchronní provádění kódu, takže v případě náročných operací může dojít k zamrznutí hlavního vykreslovacího vlákna. Pokročilejší variantu si představíme taktéž v další kapitole.

3.3 Pokročilé použití s knihovnou třetí strany

V této kapitole navážeme na znalosti získané v předchozí kapitole a představíme si pokročilejší a praktičtější implementaci jak na straně C++, tak na straně webové aplikace. Primárním cílem této kapitoly bude představit možnost, jak převést již hotovou C++ knihovnu třetí strany (případně vlastní napsanou knihovnu) do WebAssembly, a jak vylepšit implementaci webové aplikace, aby při složitějších operacích nedocházelo k blokování hlavního vykreslovacího vlákna.

3.3.1 Část C++

Předpokládejme, že máme C/C++ knihovnu, kterou chceme zkompilovat do WebAssembly. Touto knihovnou může být libovolná knihovna třetí strany, případně vámi napsaná knihovna.

Pro naši ukázku předpokládejme, že chceme převést knihovnu `word_counter`, která přijímá jméno souboru, tento soubor čte a spočítá výskyty slov a znaků v tomto souboru. Mimo jiné obsahuje funkce na nastavování a získávání konfiguračních parametrů. Její hlavičkový soubor vypadá následovně:

```
class word_counter {
public:
    word_counter(
        const bool is_case_sensitive,
        const bool use_alfanumeric_characters_only
    );
    ~word_counter();

    const bool get_flag_is_case_sensitive();
    const bool get_flag_use_alfanumeric_characters_only();
    void set_flag_is_case_sensitive(const bool is_case_sensitive);
    void set_flag_use_alfanumeric_characters_only(
        const bool use_alfanumeric_characters_only
    );
    const unsigned int get_min_word_length();
    const unsigned int get_max_word_length();
    void set_min_word_length(const unsigned int min_word_length);
    void set_max_word_length(const unsigned int max_word_length);

    const std::map<char, unsigned int>& get_character_occurrences();
    const std::map<std::string, unsigned int>& get_word_occurrences();
    const unsigned int get_total_word_count();

    void process_file(const std::string& filename);
    void clear();
    void merge_into(word_counter* word_counter);
}
```

Vytvoříme si opět soubor `library.cpp`. Ten bude obsahovat import knihovny, kterou chceme exportovat, a poté kód, kterým budeme mapovat C++ třídu do WebAssembly. Jednotlivé konstrukty si rozebereme následovně.

```
#include <emscripten.h>
#include <emscripten/bind.h>
#include "../external/word-counter/word_counter.hpp"

using namespace emscripten;

EMSCRIPTEN_BINDINGS(word_counter_library) {
    class_<word_counter>("word_counter")
        .constructor<const bool, const bool>()
        .function("get_flag_is_case_sensitive",
            &word_counter::get_flag_is_case_sensitive)
        .function("get_flag_use_alfanumeric_characters_only",
            &word_counter::get_flag_use_alfanumeric_characters_only)
        .function("set_flag_is_case_sensitive",
            &word_counter::set_flag_is_case_sensitive)
        .function("set_flag_use_alfanumeric_characters_only",
            &word_counter::set_flag_use_alfanumeric_characters_only)
        .function("get_min_word_length",
            &word_counter::get_min_word_length)
        .function("get_max_word_length",
            &word_counter::get_max_word_length)
        .function("set_min_word_length",
            &word_counter::set_min_word_length)
        .function("set_max_word_length",
            &word_counter::set_max_word_length)
        .function("get_total_word_count",
            &word_counter::get_total_word_count)
        .function("get_character_occurrences",
            &word_counter::get_character_occurrences)
        .function("get_word_occurrences",
            &word_counter::get_word_occurrences)
        .function("process_file", &word_counter::process_file)
        .function("clear", &word_counter::clear)
    ;
    register_map<std::string, unsigned int>("map<string, int>");
    register_map<char, unsigned int>("map<char, unsigned int>");
    register_vector<std::string>("vector<string>");
    register_vector<char>("vector<char>");
}
```

Oproti první ukázkové implementaci zde již používáme importovanou knihovnu, do které nijak nezasahujeme. Vystavování C++ kódu tentokrát řešíme pomocí `EMSCRIPTEN_BINDINGS` [10].

`EMSCRIPTEN_BINDINGS` nám dovoluje vystavovat jak funkce, tak třídy. Podporuje i další konstrukty jazyka jako jsou např. konstanty, enumy, ukazatele, dědičnost, abstraktní a virtuální metody a třídy či přetěžování funkcí. Vzhledem k široké škále problémů, které jsou řešitelní pomocí `EMSCRIPTEN_BINDINGS`, není možné v této ukázce obsáhnout všechny možné případy, a proto doporučuji pro další informace navštívit dokumentaci Emscripten.

Pokud se však vrátíme k našemu příkladu, tak je zde několik konstruktů stojících za zmínku.

Pro exportování třídy slouží `class_<T>(const char* name)`. Do šablony uvádíme třídu, kterou chceme exportovat a jako parametr funkce uvádíme název třídy, pod kterým danou třídu exportujeme. Pokud třída disponuje konstruktorem, tak ho exportujeme pomocí `constructor<ConstructorArgs...>()` a datové typy parametrů konstruktoru definujeme v šabloně.

Jednotlivé metody registrujeme pomocí funkce `function(const char* name, ReturnType (*fn)(Args...))`, kde prvním parametrem je název funkce, pod kterou funkci exportujeme a druhým parametrem je ukazatel na funkci. Tímto zápisem lze exportovat i obyčejnou funkci mimo třídu. Statické funkce lze registrovat pomocí funkce `class_function`, která má shodnou signaturu.

Ačkoliv to v ukázce není uvedené, tak lze registrovat i třídní proměnou, a to pomocí funkce `property(const char* fieldName, FieldType ClassType::*field)`, případně `property(const char* fieldName, Getter getter, Setter setter)`.

Pokud některý z parametrů funkce je typu `std::vector` nebo `std::map`, tak je nezbytné tyto typy zaregistrovat zvlášť. `std::vector` se registruje pomocí funkce `register_vector<T>(const char* name)` a `std::map` pomocí `register_map<T key, T value>(const char* name)`. Datové typy se uvádějí v šabloně, jméno datového typu pro export může být libovolné, ale mělo by být unikátní.

Pokud máme připravenou definici pro export knihovny, můžeme ji zkompilovat pomoci následujícího příkazu.

```
em++ ./src/library.cpp \
  ./external/word-counter/word_counter.cpp \
  -o ./build/wasm.js \
  --bind \
  --no-entry \
  -s NO_DISABLE_EXCEPTION_CATCHING \
  -s ENVIRONMENT=web \
  -s MODULARIZE=1 \
  -s EXPORT_ES6=1 \
  -s EXPORTED_FUNCTIONS="['_malloc', '_free']" \
  -s EXPORTED_RUNTIME_METHODS="['FS']" \
  -s ALLOW_MEMORY_GROWTH=1
```

Oproti parametrům `em++` z první ukázky zde máme navíc uvedenou cestu k souborům knihovny a parametr `-bind`, který definujeme, že chceme vyu-

žít k exportování knihovny výše využívaný `EMSCRIPTEN_BINDINGS`. Ostatní parametry byly popsány již v předchozí ukázce.

3.3.2 Část webové aplikace

Přístup k webové aplikaci se oproti předchozí ukázce taktéž změní. Hlavní změnou bude využití technologie Web Worker, která nám umožní provádění kódu v paralelním vlákně.

Pro lepší práci s Web Workerem si implementujeme funkci `executeWorker`, která bude vytvářet instanci Web Workeru na základě uvedených parametrů a bude ihned odesílat data Web Workeru ke zpracování. Celá funkce je implementována jako Promise, zachytává události a zprávy z Web Workeru a vrací je takovou formou, že je možné s funkcí pracovat jako by byla asynchronní.

```
// services/workerService/executeWorker.js
export const executeWorker = (
    Worker,
    workerArguments,
) => new Promise((resolve, reject) => {
  const worker = new Worker();

  worker.addEventListener('message', ({ data }) => {
    resolve(data);
    worker.terminate();
  });

  worker.addEventListener('error', () => {
    reject();
    worker.terminate();
  });
  worker.addEventListener('messageerror', () => {
    reject();
    worker.terminate();
  });

  worker.postMessage(workerArguments);

  return worker;
});
```

Výše uvedený krok by bylo samozřejmě možné vynechat a komunikovat s Web Workerem pomocí událostí. Implementace pomocí Promise je však modernější, lépe uchopitelná a integrovatelná do již existující aplikace. Implementace pro případ, kdy bychom potřebovali volat funkce nad instancí opakovaně, by se samozřejmě lišila a bylo by nezbytné rozlišovat jaké funkce voláme, ideálně pomocí dalšího zasílaného parametru.

Dále si vytvoříme soubor, který bude implementovat samotný Web Worker. Nejprve bych rád uvedl, že ukázkový kód využívá balíčkovací nástroj Webpack a modul pro integraci Web Workerů s názvem `worker-loader`[4]. Výhoda `worker-loader` je v tom, že soubor s Web Workerem je možné do aplikace importovat pomocí direktivy `import`, kdy se daný soubor tváří jako třída, z níž je možné vytvořit instanci. Webpack toto řeší při sestavení balíčku.

Pokud nemáte k dispozici Webpack, tak je možné Web Worker implementovat standardní cestou jako zvláštní soubor a instanci vytvořit pomocí konstruktoru `new Worker('executeWordCounter.worker.js')`. V potaz je nutné vzít ale to, že nelze jednoduše importovat a používat jiné servisní funkce, které v této ukázce budeme používat.

Teď již k samotnému souboru Web Workeru. Ten bude implementovat funkci `addEventListener('message', (data) => {})`, jež bude zavolána v momentě, kdy z hlavního vlákna pošleme Web Workeru zprávu, kterou obdržíme v parametru funkce. V rámci těla výše uvedené funkce budeme volat funkci `postMessage`, která pošle zprávu zpět do hlavního vlákna

```
// workers/executeWordCounter.worker.js
import createWasmModule from './wasm';
addEventListener('message', async ({ data }) => {
  const { fileContent, isAlphaNumericalOnly,
    isCaseSensitive } = data;
  const fileName = 'wordCounterFata.txt';

  const WasmModule = await createWasmModule();
  WasmModule.FS.writeFile(fileName, fileContent);
  const wordCounter = new WasmModule.word_counter(
    isCaseSensitive, isAlphaNumericalOnly
  );

  let isFailed = false;
  try {
    wordCounter.process_file(fileName);
  } catch (e) {
    isFailed = true;
  } finally {
    WasmModule.FS.unlink(fileName);
  }

  if (isFailed) {
    return postMessage({
      characterOccurrences: [],
      isEmpty: false,
      isFailed: true,
      totalWords: 0,
```

[4]https://www.npmjs.com/package/worker-loader

```
      wordOccurrences: [],
    });
  }

  const totalWords = wordCounter.get_total_word_count();
  if (totalWords === 0) {
    return postMessage({
      characterOccurrences: [],
      isEmpty: true,
      isFailed: false,
      totalWords: 0,
      wordOccurrences: [],
    });
  }

  const wordOccurrencesRaw = wordCounter.get_word_occurrences();
  const wordOccurrences = [];
  for (let i = 0; i < wordOccurrencesRaw.keys().size(); i += 1) {
    const key = wordOccurrencesRaw.keys().get(i);
    wordOccurrences.push({
      id: key,
      value: wordOccurrencesRaw.get(key),
    });
  }
  const characterOccurrencesRaw = wordCounter.get_character_occurrences();
  const characterOccurrences = [];
  for (let i = 0; i < characterOccurrencesRaw.keys().size(); i += 1) {
    const key = characterOccurrencesRaw.keys().get(i);
    characterOccurrences.push({
      id: String.fromCharCode(key),
      value: characterOccurrencesRaw.get(key),
    });
  }

  wordCounter.clear();
  wordCounter.delete();

  postMessage({
    characterOccurrences,
    isEmpty: false,
    isFailed: false,
    totalWords,
    wordOccurrences,
  });
});
```

Teď si rozebereme jednotlivé části výše uvedené funkce. V rámci parametru `data` obdržíme obsah souboru a parametry pro C++ knihovnu `word_counter`.

Pomocí již známe tovární funkce `createWasmModule` vytvoříme instanci modulu. Instance obsahuje námi exportovanou třídu `word_counter`, jež je dostupná pomocí `WasmModule.word_counter`. Jejímu konstruktoru předáme parametry a vytvoříme její instanci.

V minulé ukázce jsme si ukázali práci s paměti pomocí funkcí `WasmModule._malloc` a `WasmModule._free`. Protože třída `word_counter` nepřijímá obsah souboru, ale přijímá pouze název souboru, který čte přímo, představíme práci se souborovým systémem.

Pro práci se souborovým systémem WebAssembly modulu využijeme exportovaný objekt `WasmModule.FS`. Ten obsahuje všechny nezbytné funkce pro práci se souborovým systémem. Pomocí `WasmModule.FS.writeFile(fileName, fileContent)` vytvoříme ve WebAssembly modulu soubor ze zadaným jménem a obsahem.

S totožným jménem souboru zavoláme funkci `process_file` z instance třídy `word_counter`, který si načte daný soubor a provede výpočet počtu slov a znaků v daném souboru. Po dokončení této rutiny vymažeme soubor pomocí `WasmModule.FS.unlink(fileName)`.

Pokud by bylo nezbytné, tak můžeme vytvářet další a soubory složky, procházet adresářovou strukturu a provádět další nezbytné úkony, které můžeme dělat s každým souborovým systémem.

Za zmínku ještě stojí práce s datovými typy `std::vector` a `std::map`. V rámci JavaScriptu nejsou tyto datové typy reprezentovány polem či objektem, ale pomocí funkce `keys()` je nezbytné zjistit název dostupných klíčů a pomocí funkce `get(key)` získat příslušnou hodnotu.

Na konci je nezbytné zavolat nad instancí třídy `word_counter` funkci `delete()`, která uvolní paměť, jinak by došlo k jejímu přetečení.

V hlavním JavaScriptovém souboru poté pouze importujeme výše uvedenou pomocnou funkci pro práci s Web Workery a daný Web Worker, který mu i s parametry předáme.

```
// main.js
import { executeWorker } from './services/workerService/executeWorker';
import WordCounterWorker from './workers/executeWordCounter.worker';
// ...
const result = await executeWorker(WordCounterWorker, {
    fileContent,
    isAlphaNumericalOnly,
    isCaseSensitive,
}
```

3.3.3 Shrnutí

Výše uvedená ukázka představuje pokročilejší implementaci C++ kódu pomocí WebAssembly do webové aplikace.

C++ část umožňuje pomocí `EMSCRIPTEN_BINDINGS` exportovat téměř libovolnou část jak vlastního kódu, tak kódu třetí strany.

Část webové aplikace byla oproti předchozí ukázce rozšířena o implementaci s technologií Web Worker, jež umožňuje provádět WebAssembly kód paralelně bez blokace hlavního vlákna webového prohlížeče.

Implementace podobného rázu je dostatečná pro použití v reálných aplikacích. Emscripten nabízí samozřejmě řadu dalších konstruktů, aby bylo možné exportovat např. třídy podporující dědičnost, enumy, konstanty. Dále poskytuje možnosti jak pracovat např. se standardním vstupem a výstupem, vlákny nebo jak volat JavaScriptový kód z C++.

Takové techniky jsou však pokročilé a jsou silně závislé na konkrétních potřebách, a tudíž zde není možné obsáhnout všechny tyto techniky. Pro další možnosti doporučuji navštívit dokumentaci Emscripten.

3.4 Další pokročilé ukázky použití

V příloze této práce jsou dostupné ukázky jak kódu, který byl představen v textu této práce, tak obsahuje i další ukázky a implementace, které byly vytvořeny v rámci této práce. Ukázky kódu byly v rámci tohoto dokumentu zjednodušeny, ve zdrojových souborech se mohou nacházet v pozměněné variantě.

Kapitola 4

Závěr

V rámci této práce jsme si představili WebAssembly, jeho vlastnosti, datový formát, kompilátory a koncepci propojení WebAssembly s webovou aplikací.

Dále jsme si představili jednu základní a jednu pokročilou ukázku, jak kód napsaný v C/C++ zkompilovat do WebAssembly, a jak jej lze následně využít ve webové aplikaci. V rámci této ukázky jsme si představili jak exportování funkcí, tak celých tříd s neprimitivními datovými typ, práci s pamětí, práci se souborovým systémem a dalšími konstrukty, které Emscripten nabízí.

Po přečtení by čtenář znalý webových technologií a C/C++ měl mít dostatečné informace k tomu, aby dokázal vytvořit takovou aplikaci, která dokáže pomocí WebAssembly využívat C/C++ kód, který bude obsluhovat z webové aplikace.

V rámci práce byla vytvořena i praktická část práce, která slouží jako doplněk této práce, kde se čtenář může podívat na reálné zapojení WebAssembly do webové aplikaci odpovídající soudobým standardům.

Literatura

[1] W. C. Group. (2021) Webassembly specification. [Online]. Available: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[2] M. Contributors. (2021) Understanding webassembly text format. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

[3] W. Contributors. (2021) Understanding webassembly text format. [Online]. Available: https://en.wikipedia.org/wiki/S-expression

[4] M. Contributors. (2021) Converting webassembly text format to wasm. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Text_format_to_wasm

[5] G. Contributors. (2021) Awesome webassembly languages. [Online]. Available: https://github.com/appcypher/awesome-wasm-langs#java

[6] M. Contributors. (2021) Using the webassembly javascript api. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API

[7] L. W. a. A. Z. David Herman. (2014) asm.js – working draft. [Online]. Available: http://asmjs.org/spec/latest/

[8] M. Contributors. (2021) Compiling an existing c module to webassembly. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/existing_C_to_wasm

[9] emscripten.org. (2021) Emscripten compiler frontend. [Online]. Available: https://emscripten.org/docs/tools_reference/emcc.html

[10] ——. (2021) Emscripten – bind.h. [Online]. Available: https://emscripten.org/docs/api_reference/bind.h.html?highlight=emscripten_bindings