Compilers

INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture X
2024-12-12

Course website

burakarslan.com/inf400

Runtime Environments (cont'd)

Management of run-time resources

Correspondence between:

- static (compile-time) and
- dynamic (run-time) structures

Storage organization

Management of run-time resources

Execution of a program is initially under the control of the runtime environment.

- This is the same for machine code, WASM running on OS, WASM running in a browser, Java Bytecode interpreted by the JVM, etc.

Management of run-time resources

When a program is invoked:

- The OS allocates space for the program
- The code is loaded into part of the space
- The OS jumps to the entry point (i.e., main)

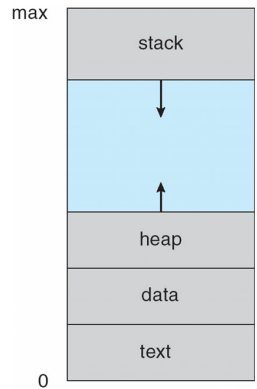
A process's view of the world

Each process has own view of machine [a]

- Its own address space
- Its own virtual CPU (through preemptive multitasking)

Simplifies programming model

- `gcc` does not care that `firefox` is running



[a] This slide is from INF-333 L03S12

Management of run-time resources

Note that these pictures are simplifications.

A WASM module can have acces to:

- multiple memory blocks
- an "infinite" number of registers

Management of run-time resources

It's the compiler's job to:

- Generating code in the format that the loader expects
- Orchestrating the use of the heap (dynamic memory) and data/text (global static memory).

Management of run-time resources

Globals

- All references to a global variable point to the same object
- Globals are assigned a fixed address once
- Variables with fixed address are statically allocated
  Depending on the language, there may be other statically allocated values

Management of run-time resources

Heap

- A value that outlives the procedure that creates (C++)

```
void foo() { new Bar }
```

- The Bar instance must survive deallocation of foo' stack frame
- Languages with dynamically allocated data use a heap to store dynamic data

Management of run-time resources

Heap

The text/code area contains object code:

- For most languages, fixed size and read only

The static area contains data (not code) with fixed addresses (e.g., global data)

- Fixed size, may be readable or writable

Heap contains all other data:

- In C, heap is managed by `malloc` and `free`

Management of run-time resources

Other Memory

Both the heap and the stack grow:

- Must take care that they don't grow into each other
- In Kiraz' case, this ensured by the WASM runtime
- If we grow beyond the memory limits, the runtime is nice enough to generate an unrecoverable trap and terminates execution

Management of run-time resources

Data Layout

Low-level details of machine architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is alignment
- WASM runtime can't protect from bad memory access patterns

Management of run-time resources

Alignment

Most machines are 32 or 64 bit

- 8 bits in a byte
- 4 bytes in a word
- Machines are either byte or word addressable

Management of run-time resources

Alignment

Data is **word aligned** if it begins at a word boundary

- Most machines have some alignment restrictions
- Or performance penalties for poor alignment

Management of run-time resources

Alignment

Example: A string:

```
"Hello"
```

Takes 5 characters (without a terminating `\0`)

- To word align next datum, add 3 "padding" characters to the string
- The padding is not part of the string, it's just ~~wasted~~ unused memory

Management of run-time resources

Alignment

WebAssembly has the `(data)` instruction to manage static memory [1]

- We need static memory to implement the `String` type

[1] ie. memory whose contents are known at compile-time

Management of run-time resources

Alignment

We have our `WasmContext` companion class for the code generation step [2]

- One of its jobs is to manage the static memory (data section).
- Any string literal encountered in the source code finds itself at a well-known location in the data section

[2] Just like the `SymbolTable` class we used during code verification,

Implementing Kiraz Primitives in WASM

Boolean

The `Boolean` type is the simplest to implement.

- What should the below statements return?
  - `let b1: Boolean;`
  - `let b2: Boolean = true;`
  - `let b3: Boolean = false;`
  - `b1 == b2;`
  - `b1 == b3;`
  - `b2 == b3;`
- Which WebAssembly type(s) to use to implement `Boolean`?

Implementing Kiraz Primitives in WASM

Integer64

The `Integer64` is next:

- What should the below statements return?
  - `let i1: Integer64;`
  - `let i2: Integer64 = 1;`
  - `let i3: Integer64 = 2;`
  - `i1 == i1;`
  - `i1 == i2;`
  - `i2 == i2;`
  - `i1 == i3;`
  - `i2 == i3;`
- Which WebAssembly type(s) to use to implement `Integer64`?

Implementing Kiraz Primitives in WASM

String

The `String` is last:

- What should the below statements return?
  - `let s1: String;`
  - `let s2: String = "a";`
  - `let s3: String = "b";`
  - `s1 == s1;`
  - `s1 == s2;`
  - `s2 == s2;`
  - `s1 == s3;`
  - `s2 == s3;`
- Which WebAssembly type(s) to use to implement `String`?

Implementing Key Operations in WASM

`io.print()`

The only way to call `io.print()` is to first implement the `import` statement.

- Note that `io.print()` is part of the Kiraz language definition
- Also note that the `io` nor the `io.print` identifiers are writable.
- $\Rightarrow$ We can hardcode the function signature(s) of the `io.print()` variants in the compiler code.
- Use the `(import)` instruction

Implementing Key Operations in WASM

`io.print()`

We only have one connection to the outside world:

- `io.print()`: Takes a single `String`, `Integer64` or `Boolean` argument. Ends up calling the `console.log` function with the given data.
- How does `io.print()` deal with different types?

Implementing Key Operations in WASM

`let`

The `let` statement:

- Is only allowed in some contexts. What are they?
- We say that code generated for the let statement is **context-dependent**.

Implementing Key Operations in WASM

`func`

The `func` statement:

- Is only allowed in some contexts.
- Can we say that the generated for the func statement is context-dependent?

Implementing Key Operations in WASM

`if`

The `if` statement:

- Is only allowed in some contexts.
- Can we say that the generated for the if statement is context-dependent?

Implementing Key Operations in WASM

OP_ADD

Integer examples:

- `let i1: Integer64;`
- `let i2: Integer64 = 1;`
- `let i3: Integer64 = 2;`
- `i1 + i1;`
- `i1 + i2;`
- `i2 + i2;`
- `i1 + i3;`
- `i2 + i3;`

Implementing Key Operations in WASM

`OP_ADD`

String examples:

- `let s1: String;`
- `let s2: String = "a";`
- `let s3: String = "b";`
- `s1 + s1;`
- `s1 + s2;`
- `s2 + s2;`
- `s1 + s3;`
- `s2 + s3;`

Implementing Key Operations in WASM

`OP_ADD`

Boolean examples:

- `let b1: Boolean;`
- `let b2: Boolean = true;`
- `let b3: Boolean = false;`
- `b1 + b1;`
- `b1 + b2;`
- `b2 + b2;`
- `b1 + b3;`
- `b2 + b3;`

Writing and Debugging WebAssembly

- The methodology is to first implement the code in WAT manually
- Then make the compiler emit the given code
- Use named `(local)` instructions for variables and temporaries
- Chromium seems to have the nicest WASM debugger – use it!

HW3 Advice

gen_wat

The `gen_wat()` function is our entrypoint to the code generation stage.

Two variants:

- `Node::Ptr gen_wat(WasmContext &);`
- `Node::Ptr gen_wat(WasmContext &, const std::string &id) const;`

HW3 Advice

`gen_wat`

- const variant: Generates code for shared nodes (types)
- non-const variant: Generates code for exclusive nodes (all the rest).
- All variants use the variable id to generate `(local)` instructions

HW3 Advice

The Testsuite

WABT integration:

- Validates the generated WASM code

SpiderMonkey integration:

- Not directly runnable by most of you
- Requires SpiderMonkey 115
- Runs the generated (and validated) WASM code
- See `wasm.cc` for details

HW3 Advice

The Testsuite

Running in the browser:

- You need to implement your own loader
- You need to use a local HTTP server (eg. `python3 -m http.server`)
- Lets you use the awesome Chrome WASM debugger
- Note that the test suite writes the generated code in `$CWD/<testname>.{wat,wasm}`

HW3 Advice

Strategy

1. Find/Write a WASM hello world module.
2. Implement and ensure that your loader works.
   Don't skip this step!
   Make sure things work before you move on
3. Start writing webassembly code that passes the tests
4. Write the compiler code that generates the desired WAT

HW3 Advice

Strategy

Compiler Strategy:

1. Start from outside to inside
2. `(module)`
3. `(func)`
4. `(if)`
5. `(local) / let`
6. `OP_ADD`

HW3 Advice

A Review of the Test Cases

Let's launch the IDE