Rust Universal Machine Design Document

Connor Gray

Overview

This program seeks to simulate computer hardware. It will be able to run binary code in the same way a real CPU could and correctly execute programs that it is given.

The main problem of this program will be figuring out how to manage memory segments. The binary programs that will be run on the universal machine will make requests for memory allocation and deallocation and it is up to our implementation to efficiently manage that memory. It is important to maintain that all allocated memory remains where it was allocated as well as reusing memory segments that have been deallocated lest it grow without bounds.

Architecture

The structure of the universal machine and the use of the tools provided within this structure will solve this problem. My implementation will maintain a virtual machine struct. This struct will contain the following fields.

```
pub struct vm {
	pub regs: Vec<u32>,
	pub memory: Vec<Vec<u32>>,
	pub unmapped_segments: Vec<usize>,
	pub max_mapped_segment: usize,
	pub program_counter: u32
}
```

The 8 registers of the machine will be stored in a single vector of u32 values. Each u32 value represents a register and can be accessed and altered by accessing the machine using a mutable reference. This quality holds true for all the fields of the struct. The program memory will be contained within a 2D vector. Each internal vector represents a segment of memory. The program counter variable exists to keep track of precisely what instruction the program is executing. When the virtual machine is loaded all of the instructions are stored and are accessible through the first memory segment to which the program counter refers.

The remaining fields are required for the specific implementation of memory management I have chosen. Unmapped segments are cleared of their values and remain in place in memory. The indices of unmapped segments are kept track of using a vector, unmapped_segments. When a segment is requested to be mapped, the program will first check for any unmapped segments and use one of those if available. Otherwise, a new segment will be pushed into memory. A single u32 value, max_mapped_segment, will keep track of the index of any newly pushed segment of memory and notify the program of that index. This value is incremented whenever a new segment is pushed.

My implementation of this program will be split up into the main file, a file for the virtual machine struct and its methods, and lastly a file for the instruction set. The following methods will be implemented for the virtual machine struct.

```
pub fn new() -> Self

// Constructor; will return a bare bones virtual machine struct.

pub fn boot(&mut self)

// Reads in bytes from a file, stores them into a vector of u32's and pushes that vector as the first segment in memory.

pub fn run(&mut self)

// Loop that runs the following fetch and execute methods for each instruction.
// Loop halts when halt instruction is called (calls std::process::exit(0))

fn fetch(&mut self) -> u32

// Returns the next instruction from the first segment in memory using the program counter and increments that counter.

fn execute(&mut self, word: u32)

// Unpacks and matches on the operation code of current instruction and calls respective function from the instruction set file.
```

The instruction set file will contain all 14 necessary functions that will be called by the binary programs. Each function (excluding halt) accepts a mutable reference to the

virtual machine along with the binary instruction code. The virtual machine is directly manipulated within these functions so no return is necessary.

The bitpack module is utilized for unpacking the values from the binary instruction code.

The main function of this program will instantiate a virtual machine struct using its constructor method. It will then call the boot method and then the run method on that struct. That is all the main function will be responsible for.

Testing Plan

Unit tests will be written to stress test the virtual machine directly. These unit tests will instantiate a vm struct and a designated set of instructions can be pushed into memory. This will replace calling the boot method. This will allow direct testing of how the virtual machine is affected by specific instructions. Edge cases for each field of the struct should be tested and the program should properly panic upon an invalid instruction. There should be at least one unit test per instruction and more for complex instructions with finicky edge cases such as map segment and load program. Much of the universal machine's behavior is undefined and thus it is important not to waste time testing for those cases.

In order to debug the virtual machine the RUMDump lab provides the ability to read the binary instructions being executed as readable assembly code. My implementation of RUMDump also includes a mode to print each instruction as it is being executed which allows for an easier time seeing the sequence of instructions that directly caused the program failure.

Lastly, programs were provided alongside the assignment handout. The universal machine should execute these programs completely, correctly, and within a reasonable timeframe. These programs are exhaustive and only a properly designed virtual machine will be able to fully execute the heavier programs codex and advent. This will especially test if the virtual machine can withstand a large amount of memory allocation and deallocation without running out of system memory.