Design Document: Homework 6

Universal Virtual Machine

Spencer Schoeben (sschoe05) and Mingzhe Li (mli04)

November 4, 2014

COMP 40

We will implement a Universal Virtual Machine (UM) that is capable of executing a program written in the UM assembly language. The UM is a basic machine with 8 registers, a memory that is structured as segments of bit words, and is capable of executing 14 different kinds of instructions.

Architecture

Our program is made up of many modules which interact with each other to create the Univeral Machine abstraction.

The Controller Module is the highest level of abstraction and pulls all the modules together. To run an instance of the UM, a main file like the following would be written to create a machine, load in a program, and execute all its instructions, and then kill itself.

```
int main() {
    FILE* program = fopen(argv[1], ''r'');
    UM machine = UM_new();
    UM_run(machine, program);
    UM_kill(machine);
}
```

The virtualization module is used to share a set of truths about the abstractions of data types used in our implementation among all the modules. This module also allows registers to be set and loaded.

The program loading module is solely responsible for reading a program's data from file and putting it into the machine's memory. This module is used by the controller when UM_run is called.

Instruction execution is handled by its own module which, when given an instruction executes the instruction as expected. UM_run handles looping through the instructions, but each instruction itself is handled by the execution module.

When the machine is intructed to read or write, it uses the I/O module which allows to read and write one character at a time.

When the machine asks for memory segments, the memory management module is used to map a segment and return a segment ID that allows data to be stored and retrieved from the segments. The machine may also ask the memory manager to unmap a segment given it's segment id.

Interfaces

Controller Module

The controller module is used to create instances of a Universal Machine and run programs on the machine from a specified UM language program.

Their are only three functions in the interface. UM_new() creates a new Universal Machine instance and returns the machine in type UM. UM_run runs a program from a file on a specified instance of a UM. UM_kill() basically shuts down the UM intstance, freeing all resources used by the virtualized machine.

```
UM UM_new();
bool UM_run(UM machine, FILE* program);
bool UM_kill(UM machine);
```

The UM_new function is implemented by creating an instance of the UM_T struct defined in the virtualization module and initializing all the data structures within the struct.

UM_run calls the load() function to load the program into the machine's memory and then starts executing the program's instruction until it reaches the end of the program.

UM_kill is basically just the opposite of UM_new.

I/O Module

The I/O module facilitates printing and reading characters to and from the I/O stream.

The interface exposes two functions for reading and writing one character at a time.

```
Word read(void);
Word write(Word to_write);
```

The functions are implemented as wrappers for C's underlying stdio library. IO_read reads a single character from stdin with getchar and casts it as a Word (32-bit integer) between 0 and 255. IO_write writes a Word to stdout using putchar. IO_write only accepts values between 0 and 255. If the value is not in this range, an error will be thrown. Character is printed as ASCII.

When reaching the end of file (EOF) both IO_read and IO_write will return the EOF constant which is equal to (unsigned int) -1 or a Word with all the bits on. EOF indicates that the end of stdin has been reached or that stdout has reached some sort of capacity constraint.

Memory Management Module

The memory management module does all the heavy lifting for mapping and unmapping segments in the machine's virtualized memory.

We expose two functions, one for mapping a segment and one for unmapping a segment. We also have a intitialize_memory function which sets up the SegmentBlock for use by the machine. For access to words within mapped segments, we have put_word and get_word functions.

```
void initialize_memory(UM machine);
SegmentID map(UM machine, int size);
void unmap(UM machine, SegmentID id);
Word put_word(UM machine, SegmentID id, int offset, Word value);
Word get_word(UM machine, SegmentID id, int offset);
```

To implement the memory management module, we maintain a sequence of unmapped segment IDs in order to allow for reuse of these IDs. We also keep a next_seg counter to keep track of the next segmentID to use if there are none available in the sequence. Each segment is a basic C array of Words which we malloc and then store the pointer to the segment in the seg_map array.

We are relying on the indexes of the seg_map array as the "keys" and storing pointers to our segments as the "value."

The following structure keeps track of the data we need for our implementation of memory.

```
struct SegmentBlock_T {
    Seq_T unmapped_ids;
    SegmentID next_seg;
    Array_T seg_map;
};
```

When a segment is mapped, we create an array of Words equal to the requested size of the segment + 1. In the first position of the segment we store the size of the segment which allows us to check for calls that are out of bounds.

Instruction Execution Module

This module is used to execute a single line of UM instruction code.

The interface exposes a very simple function which executes the next instruction in the program as specified by the program counter. If the

instruction exists, it runs it and returns true. Otherwise, it returns false.

```
bool execute(UM machine);
```

When the client runs the Execute_Instruction function, the next instruction is fetched from the machine passed in as an argument. This instruction, in binary format, is fetched from memory segment 0 with offset equal to the program counter.

We decode the instruction using the Bitpack module inside of our own abstraction which stores the unpacked instruction in a struct.

We then call the function stored in the function pointer in an array of functions at index = opcode.

We have an array of pointers which point to the functions necessary to execute each type of instruction. The function pointers are stored at the index of their opcode.

When we call the functions, we pass the struct of the decoded instruction. This makes it possible for all of the functions to have the same contract and access the necessary information to execute the instruction.

Program Loading Module

The program loading module loads the UM program from a file into a specified instance of a Universal Machine.

The interface has only one exposed function. The function expects a valid and initialized UM and a valid FILE* to the program. If the loading is successful, the program will return true. Otherwise, it will raise an error and return false.

```
bool load(UM machine, FILE* program);
```

The module will check for the size of the program and will map a memory segment that is large enough to store the program. It will then read

the program in one character at a time, storing each 32-bit chunk as a word in the mapped segment.

Virtualization Module

The virtualization module defines the types used in our UM abstraction. The virtualization.h file can be included by any module to make them aware of types that are shared between modules.

```
typedef uint32_t SegmentID;
typedef uint32_t Word;
typedef Word * RegisterBank;
typedef Word * Segment;
typedef struct SegmentBlock_T * SegmentBlock;
typedef struct UM_T {
    RegisterBank registers;
    SegmentBlock memory;
    SegmentID pc;
} * UM;
```

This module also has functions for working with registers.

register_store places a value into a register specied by the id and returns the old value. register_load returns the value of the register specified by the id.

```
Word register_store(int id, Word value);
Word register_load(int id);
```

Test Cases