The Emulator Project

Ian Duncan

September 29, 2015

Chapter 1

An Overview

1.1 What is the Emulator Project?

It is not a secret that systems programming and operating system development are not easy things to learn and do. The inherent complexities of these subjects are made worse by the current platform of choice: the personal computer with an x86 processor. There are many parts of this platform which make it difficult for beginners. This is where the emulator project comes in. The goal of the emulator project (which is still in need of a decent name) is to provide an easier environment for learning the concepts of systems programming and OS development. The emulator project provides a set of tools for this task: an emulator with a built-in interactive debugger, an assembler, disassembler, and a memory dump analyzer. While not the best or most-polished set of tools ever developed, they are enough to get the job done.

The bulk of the simplifications of the emulator are included in the "firmware", referred to as the BIOS. Tasks such as printing strings and numbers can be accomplished by simply providing the pointer to a null-terminated string in the emulator's memory or the number itself. The BIOS also contains what is referred to as the *interop module*. This module allows for easy access of files on the actual disk. In this way the emulator project can also be viewed as a tool to teach assembly-language programming. BIOS methods which provide finer control and emulate real hardware (e.g. disks and displays) are planned.

Chapter 2

User's Guide

2.1 Getting and Compiling the Emulator Project Code

The emulator project is open source, and licensed under the MIT license. It is distributed in source code form only in a GitHub repository. This book makes the assumption you are using a UNIX-like environment (e.g. Linux, Free-BSD, Cygwin). Out-of-the-box Windows support is not planned.

With that said we can move on to downloading and compiling the code. In order to download the source code from the GitHub repository you will need to ensure Git is installed. Other than this, ensure that you have a C compiler and a **modern** C++ compiler (i.e. one that really supports C++11), otherwise you will not be able to make use of the assembler. I have had problems with this on MinGW in the past, so I recommend just using Cygwin if you use Windows.

Once you are sure everything is in order, simply run the following to get and compile the emulator project code:

```
$ git clone https://github.com/jansky/JanskyProcessor.git
$ cd JanskyProcessor
$ make
$ sudo make install
```

As a final note, if you are unfamiliar with Git, the command to update the repository to the most recent version is `git pull origin master`.

2.2 Using the Emulator

The emulator is invoked as `jemulator`, and takes a variety of command-line options, only a few of which are required to actually use it. Many options only change default values. If you are satisfied with these values, it is not necessary to specify them on the command line.

The only required switch is the `-p` or `--program` switch which specifies the location of the program you wish to execute. Thus in order, to launch the emulator with default options with the program `program.bin`, you would run `jemulator -p program.bin` at the command line.

Of course, sometimes the default settings are not what you want. The table below lists the options and flags that can be set at the command line. Note that all numbers must be specified in hexadecimal.

Option	**Purpose**
`--no-reg-dump`	Disables the register dump that normally takes place after execution is finished.
`--no-mem-dump`	Tells the emulator not to produce the default memory dump that takes place after execution is finished.
`--version`	Prints version and licensing information.
`--debug`	Enables the interactive debugger
`--program, -p [Program file]`	Specifies the program file for the emulator to load
`--memsize [RAM size]`	Specifies the size of the emulator's RAM in bytes. This value must at least `0x100000` (1 MB). If this value is not specified the size will be `0xA00000` (10 MB).
`--stacksize [Stack size]`	Specifies the number of stack elements. This value must be at least `0x96`. By default it is `0x100`.
`--loadat [Address]`	Specifies the memory address to load the program into. By default it is `0x400`.

`--dumpfile [filename]`	Specifies the file that the memory dump will be written to. If this value is not specified it will be written out to `mem.dmp`.
`--rootdir [directory]`	Specifies the root directory for all interop disk operations. This value must be specified to enable interop for disk operations. Do not append a trailing slash.

2.3 Using the Assembler

The assembler is much simpler in terms of usage. It is invoked with the command `jassembler` and always takes two arguments, the assembly file and the output file. For example, if you wanted to assemble `program.s` to `program.bin`, you would run `jassembler program.s program.bin`.

The assembler will output any errors, as well as the location of all labels for debugging purposes. For example, running the assembler on the `shell.s` file in the `jos` directory produces the following output:

```
shellloop 0x25
end 0xee
error_input 0x116
error_exec 0x141
res1 0x1a2
res2 0x1ae
res3 0x1b2
res4 0x27c
res5 0x2a3
res6 0x2b8
```

It is important to note that at the moment the assembler does not support the linking of binary files or the inclusion of other files through the use of an include-style directive. If you wish to do the latter, the `sasm.sh` for ***s****mart* ***as****se****m****bler* script that comes with jOS (look in the `jos` directory) may be helpful. It takes one argument, the assembly code file, and runs this through the C preprocessor and finally the assembler, producing a `.o` file of the same

name. For example, running `./sasm.sh program.s` would produce a binary file called `program.o`. Despite the extension, the `.o` file produced does not contain relocatable code like a normal object file. Rather, it is simply output from the assembler. No tools at the moment support the linking of binary files.

2.4 Using the Disassembler

The disassembler is invoked in a manner similar to the assembler. To run the disassembler on a binary file, run `jdisassembler program.bin disassembled.s`. If you would rather the disassembler output to standard output than a file, run `jdisassembler program.bin STDOUT` (case is important).

At the moment, the disassembler has not been updated for all of the new memory and register location types, but is still helpful as it outputs the location of each instruction as a comment for debugging purposes.

2.5 Using the Memory Dump Analyzer

While not as important any more due to the presence of an interactive debugger in the emulator itself, the memory dump analyzer is still a useful tool. It operates on the memory dumps produced by the emulator after execution has finished. To run the memory dump analyze, run `jmemanalyze dumpfile.dmp`. The memory dump analyzer is an interactive program. After starting up, it should greet you with a prompt.

```
Allocated 10485760 bytes of RAM.
%
```

At the prompt, enter the type and address of the value you want to view. The types are as follows.

Type	Value
b	8 bit integer
w	16 bit integer
d	32 bit integer

After the type, you must specify the memory address in hexadecimal. Thus, an example session looks like

```
% b 400
BYTE 0x400: 0xf, 15
% b 401
BYTE 0x401: 0x2, 2
% d 500
DWORD 0x500: 0x12000000, 301989888
```

To quit enter `q 0` at the prompt.