A technical report on Virtual Machine assignment
Bootstrap Loader

Sourabh Kumar

30/01/2018

Contents

1 Abstract

This is a technical report for the Virtual Machine course assignment called Bootstrap Loader. The objective of this assignment is to adapt the VM instructions in dvm.c to build a new VM that can input a binary function along with its two arguments from standard input and then execute it.

2 Problem statement for the assignment

There has to be two instruction sequences, the first one (called BOOT) will load a second sequence of instructions (called KERNEL) starting from a specified location in MEM array, read two arguments into global variable A and B and then execute the KERNEL. KSTART stores the starting address of the kernel and KLEN is the length of the KERNEL instructions. The KERNEL should compute a function, output the result and save the result before giving the control back to BOOT. BOOT should suspend the VM state into an image file and the execution must resume from the suspended image. On resumption, BOOT code should re-execute the already loaded KERNEL code with a fresh set of arguments in A and B.

3 Experiments, Obervations and Results

When I first read the problem statement, I had no idea how to proceed because it was not clear to me. After reading it multiple times,things became clearer. One thing that confused me the most was the pseudo code provided. Initially I was under the impression that we have to write C code. But later I figured out, it will be done using the predefined opcodes.

3.1 BOOT code and Saving the Image

First I started with writing the problem statement on a piece of paper to get a clear picture of the flow of actions that has to be taken and how to input has to be provided to the program from terminal. I tried to understand the instructions hardcoded into the MEM array of dvm.c, executed it in debug mode to get an idea about how the instructions are written into the memory, how input is taken from the terminal, how registers are being modified, what happens when the image is saved and loaded again. On loading the image, I forgot to take note of the status of the registers. They all were cleared and this caused me a little trouble in the end.

By this time I was very sure on implementing the BOOT code. I started with reading the KSTART and KLEN into two registers R1 and R2 respectively. Since I needed the content of R1 later to jump to KSTART, I moved R1 into R7 and used R7 in the loop to increment it by one to move on to next location. At first I made some logical mistake here. Instead of adding one every time, I was adding one,two,three and so on with the help of counter. Later, on debugging I found out this and corrected. Finally all KERNEL instructions were stored and verified after executing the code in debug mode. Moving on, I wrote instructions to read two global variables and jump to kernel code followed by suspend and jump to WARMBOOT. WARMBOOT is the location from where reading of global variables start.

Now I had to figure out the SUS address to jump back from the KERNEL code. I realised that the address has to be stored before jumping to the KERNEL code. I went back in code to add KSTART to KLEN, stored the result in register R0 and jumped to R0 from KERNEL code, but that didn't work out. It made me jump forward in the memory. On making proper memory map I found that the address of SUS is fixed and I had to jump back in memory instead going forward. By executing the code in debug mode, I found the location of SUS and stored it in register R3. This time it worked and the image was saved successfully.

3.2 Loading Image

On loading the image file, instead of jumping to WARMBOOT to read new arguments, it started re-executing the BOOT code. This happened because register states are not saved on suspending and I failed to notice it earlier when I was trying to understand the dvm.c code. So I went back in the code, stored the KSTART and WARMBOOT address into data section of the memory. I used predefined variables VARS and GREET to store KSTART and WARMBOOT respectively. The jump instructions were modified accordingly. Finally everything worked as expected. I tried addition and multiplication instructions and both of them gave the expected results.

4 Conclusions/Experience

This was a great exercise which involved moving from one VM to another VM. I learned to make sense out of each piece of code, how all the pieces are connected and altogether forms a Machine that can do computations. Before this, I used to get scared seeing these many lines of code. But now I feel comfortable reading and debugging code.

References

[1] Class codes, "0vm.c avm.c bvm.c cvm.c dvm.c," .