Davis, Brandon
10/1/2024
45045888

Report – Bootloader Programming Tools

Introduction

This lab tasked students with writing 2 bootloaders: ARM and x86. Both programs loaded a bootloader from a device using qemu and read some log data sending it to UART. VsCode and WSL were used to test and write the boot loaders.

Screen Cast

https://youtu.be/6nD4BlE9qxg

Design

Code Structure

The code programmatically for UART works very similar to each other:

1. Point to the start of the logged message in RAM (x86) or log dev (ARM)
2. Check if the character is the null terminating character. If '\0' end code, else continue
3. Put Data out into the UART transmit register (Tx)
4. Check the Status Register (LSR) if the message has been transmitted; check if bit 0x20 is one for status of the Tx buffer (THR) is cleared.
5. If THR bit is not clear poll till the buffer cleared
6. Else increment pointer to the next character.
7. Repeat process till '\0'

x86

x86 has a standardized memory map that has a predefined boot section led by BIOS, so on the developer side we just had to write the assembly to start at 0x7C00 with data from standardized COM port addresses for UART at beginning of conventional memory, which is where the code reads the logged message (0x500). Also, the standardization of x86 makes it very compatible with other systems, which is why the x86_64 emulator works successfully with the implementation.

ARM

ARM however, is not standardized on memory mapping on boot, so every machine is different. For this lab, the Akita machine was used. Since there are no standards, the ARM processor code we made did not work for the raspi2. On Boot, ARM processor needs a Trap table initialized to what subroutines should be run in order like system reset which is done first. In the Lab, all the trap tables point to the start of code since we aren't worried about a full ARM boot process only logging a message, which for Akita is at the log dev address (0xA3F00000). Also, the developers need to set up the supervisor, turn off the TUMB instruction, and disable regular IRQs (This is done before the main UART code).

Testing and debugging

To test, a make file was created with directives to generate an object file with the assembler (as), an executable from the object file (lb), and a binary file from the executable. This was done for separate ARM and x86 targets, and the "all" target built both files. Then qemu targets ran the .bin of each respective file depending on the processor and machine. Also, the gdb debugger's "layout regs and asm" helped with stepping through the code to find bugs in testing. This helps developers se the registers they are changing on what line of code.

Conclusion

Overall, this lab was a good introduction to x86 and refresher on ARM. Also, bootloaders were introduced, and how to modify them. There are no known bugs in my code.