OS-344

Assignment-1
Doubt Clearance session

Presented by

Amit Puri

Teaching Assistant

Focus On

1. Assembly language using GNU assembler
2. Emulation of x86 hardware
3. Running toy OS xv6 on QEMU emulator
4. GNU Debugger
5. BIOS & Boot loader code Trace
6. Kernel Load Trace
7. Adding System Call

Exercise-1

- A simple modification of program is to be done using GNU assembly language.

Simulating x86 & booting xv6

For Linux users:

1. $ sudo apt-get update
2. $ sudo apt-get install build-essential
3. $ sudo apt-get install qemu
4. $ git clone git://github.com/mit-pdos/xv6-public.git

For Windows users:

1. Install Linux subsystem for Windows.
2. After successful installation, open a “bash on Ubuntu VM”.
3. Follow the same steps for Linux installation after this.

BIOS and tracing with GDB

Some Instructions prior to this exercise:

Exercise 2.

Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions and try to guess what it might be doing.

#Print screen shots or output in the report, the output of gdb at various points mentioned in instructions and exercise.

Boot Loader

Some Instructions prior to this exercise:

(Bootmain.c, Bootblock.asm) (Bootasm.S)

Exercise 3.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. **Trace through the code in bootasm.S**, using the source code and the disassembly file bootblock.asm to keep track of where you are. Also use the x/i command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in bootblock.asm and GDB.

#keep adding appropriate screenshots/ texts from terminal window (gdb/xv6) for every instruction is to be followed with commenting.

Boot Loader

Some Instructions prior to this exercise:

(Bootmain.c, Bootblock.asm) (Bootasm.S)

Exercise 3.

Trace into **bootmain()** in bootmain.c, and then into **readsect()**. **Identify the exact assembly instructions** that correspond to each of the statements in readsect(). Trace through the rest of readsect() and back out into bootmain(), and **identify the begin and end of the for loop** that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

#keep adding appropriate screenshots/ texts from terminal window (gdb/xv6) for every instruction is to be followed with commenting.

Boot Loader

Some Instructions prior to this exercise:

(Bootmain.c, Bootblock.asm) (Bootasm.S)

Exercise 3.

Answer the following questions:

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
- How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

#keep adding appropriate screenshots/ texts from terminal window (gdb/xv6) for every instruction is to be followed with commenting.

Loading the kernel

Exercise-4 (Some pointers question/answers)

Complete it for your own benefit to be able to complete further exercises and assignments. (Carry no marks)

#some instructions…

```
$ objdump -h kernel
$ objdump -h bootblock.o
```

#Take screenshots and explain in report the various fields being talked about in instructions.

Loading the kernel

#some instructions...

Exercise 5.

Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then **change the link address in Makefile to something wrong**, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens. Don't forget to change the link address back and make clean again afterwards!

Also use $ objdump -f kernel

#Perform as said, mention steps you followed, take screenshots and explain in report, the consequences.

Loading the kernel

#some instructions...

Exercise 6.

We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command x/Nx ADDR prints N words of memory at ADDR. (Note that both 'x's in the command are lowercase.) Warning: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes). Restart the machine (exit QEMU/GDB and start them again).

Examine the 8 words of memory at 0x00100000 at **the point the BIOS enters the boot loader**, and then again at **the point the boot loader enters the kernel**. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)

#Use required gdb commands to answer.

Adding a system call

#some instructions...

Exercise 7.

Create a system call **int sys_wolfie(void *buf, uint size)**, which copies an ASCII art image (**Use a buffer of a wolf picture**, google it for more information) of **Wolfie** to a user-supplied buffer, provided that the buffer is large enough. You are welcome to use an ASCII art generator, or draw your own by hand. If the buffer is too small, or not valid, return a negative value. If the call succeeds, return the number of bytes copied. You may find it helpful to review how other system calls are implemented and compiled into the kernel, such as read.

#Edit required files and add all those files in your submission **with proper comments added to your code.** (Don't forget to explain it in the report, with the required output)

Adding a system call

#some instructions...

Exercise 8.

Exercise 8.

Write a user-level application, called wolfietest.c, that gets the Wolfie image from the kernel, and prints it to the console. When the OS runs, your program's binary should be included in fs.img and listed if someone runs ls at the xv6 shell's command prompt. Study Makefile to figure out how to compile a user-mode program and add it tofs.img.

#Again, edit required files and add all those files in your submission **with proper comments added to your code.** (Don’t forget to explain it in the report, with the required output)

Questions?

Thanks….