CS3310 - Operating Systems Lab

Lab 1

CS14B009 - G. Nikitha
CS14B046 - M. Kavya Mrudula

Exercise 1. Familiarize yourself with the assembly language materials available on the 6.828 reference page. We do recommend reading the section "The Syntax" in Brennan's Guide to Inline Assembly.

- We referred to Syntax Section in the page and noticed the difference in both the syntax.

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.

- Roughly the instructions do the following.

  The execution starts with a long jump to [0xf000:0xe05b]

  It sets up Stack and clears the interrupts and direction flags in the instructions at addresses [0xf000:0xe06a],[f000:0xd236], [f000:0xd237].

  Then it sets up the IDT, GDT in the instructions at addresses [0xf000:0xd248] and [0xf000:0xd24e]

  It sets a bit in cr0 register to 1 to enable 32−bit mode at the address [0xf000:0xd25b]

Exercise 3.

Take a look at the lab tools guide, especially the section on GDB commands.

- We took a look at GDB commands needed for this lab.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S, using the source code and the disassembly file obj/boot/boot.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 obj/boot/boot.asm and GDB.

- We observed that bootloader source code, disassembly in obj/boot/boot.asm and GDB have same instructions. The addresses of each instruction depends on the architecture of the running hardware.

Trace into bootmain() in boot/main.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.

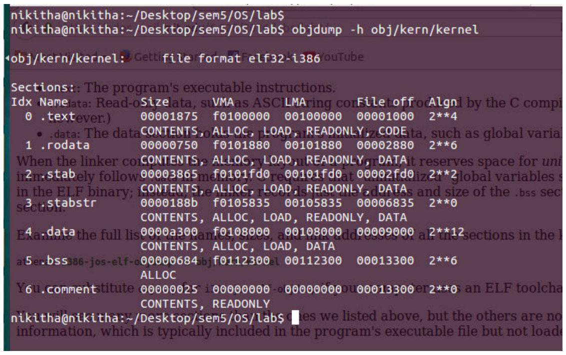
- The exact assembly instructions that correspond to each of the statements in readsect() are:

```
7c7c: 55                    push   %ebp
7c7d: 89 e5                 mov    %esp,%ebp
7c7f: 57                    push   %edi
7c80: 53                    push   %ebx
7c81: 8b 5d 0c              mov    0xc(%ebp),%ebx
7c84: e8 e1 ff ff ff        call   7c6a <waitdisk>
_asm _volatile("outb %0,%w1" : : "a" (data), "d"
    (port));
7c89: ba f2 01 00 00        mov    $0x1f2,%edx
 7c8e:  b0 01                 mov    $0x1,%al
7c90: ee                    out    %al,(%dx)
7c91: b2 f3                 mov    $0xf3,%dl
7c93: 88 d8                 mov    %bl,%al
7c95: ee                    out    %al,(%dx)
7c96: 89 d8                 mov    %ebx,%eax
7c98: b2 f4                 mov    $0xf4,%dl
7c9a: c1 e8 08              shr    $0x8,%eax
7c9d: ee                    out    %al,(%dx)
7c9e: 89 d8                 mov    %ebx,%eax
7ca0: b2 f5                 mov    $0xf5,%dl
7ca2: c1 e8 10              shr    $0x10,%eax
7ca5: ee                    out    %al,(%dx)
7ca6: 89 d8                 mov    %ebx,%eax
7ca8: b2 f6                 mov    $0xf6,%dl
7caa: c1 e8 18              shr    $0x18,%eax
7cad: 83 c8 e0              or     $0xffffffe0,%eax
7cb0: ee                    out    %al,(%dx)
7cb1: b0 20                 mov    $0x20,%al
7cb3: b2 f7                 mov    $0xf7,%dl
7cb5: ee                    out    %al,(%dx)
7cb6: e8 af ff ff ff        call   7c6a <waitdisk>
_asm _volatile("cld\n\trepne\n\tinsl"   :
7cbb: 8b 7d 08              mov    0x8(%ebp),%edi
7cbe: b9 80 00 00 00        mov    $0x80,%ecx
7cc3: ba f0 01 00 00        mov    $0x1f0,%edx
7cc8: fc                    cld
7cc9: f2 6d                 repnz insl
     (%dx),%es:(%edi)
7ccb: 5b                    pop    %ebx
7ccc: 5f                    pop    %edi
7ccd: 5d                    pop    %ebp
7cce: c3                    ret
```

- The function calls were traced in the following manner:
  bootmain() ⇒ **readseg()** ⇒ **readsect()**
  returned in the following fashion: **readsect()** ⇒ **readseg()** ⇒ **bootmain()**
- The **for loop** that reads the remaining sectors of the kernel from the disk begins at **0x7cea** and ends at **0x7cfe**
- After the for loop, the instructions from the address **0x7d00** start. *(lea -0xc(%ebp), %esp)*.
- At the address **0x7c32**, the processor starts executing 32 bit code. Setting a bit in *cr0* at **0x7c2a** &
  A long jump at the address **0x7c2d** cause the switch.
- Last instruction of the bootloader *0x7d5e:* *call *0x10018*
  First instruction of the kernel *0x10000c* *movw $0x1234,0x472*
- The count parameter sent to readseg() comes from the struct *Proghdr* attribute *p_memsz* in bootmain. The number of sectors equals this count.

Exercise 4. Read about programming with pointers in C.

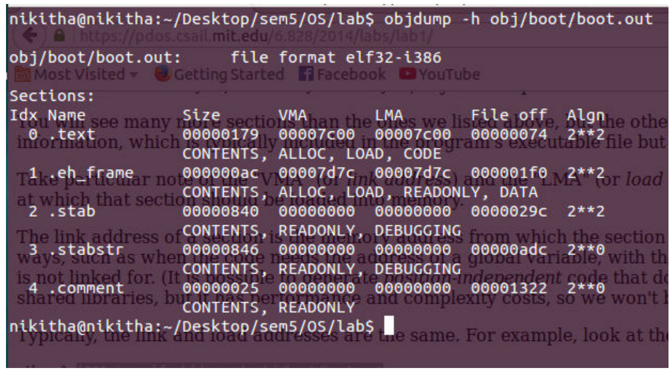
- We have gone through pointers in C.



```
nikitha@nikitha:~/Desktop/sem5/OS/lab$
nikitha@nikitha:~/Desktop/sem5/OS/lab$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001875  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000750  f0101880  00101880  00002880  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003865  f0101fd0  00101fd0  00002fd0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      0000188b  f0105835  00105835  00006835  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000684  f0112300  00112300  00013300  2**6
                  ALLOC
  6 .comment      00000025  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
nikitha@nikitha:~/Desktop/sem5/OS/lab$
```



```
nikitha@nikitha:~/Desktop/sem5/OS/lab$ objdump -h obj/boot/boot.out

obj/boot/boot.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000179  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000ac  00007d7c  00007d7c  000001f0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00000840  00000000  00000000  0000029c  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      00000846  00000000  00000000  00000adc  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      00000025  00000000  00000000  00001322  2**0
                  CONTENTS, READONLY
nikitha@nikitha:~/Desktop/sem5/OS/lab$
```

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 boot/Makefrag to something wrong, run make clean, recompile the lab with make, and trace into the boot loader again to see what happens.

- The first instruction that fails is the long jump to 32bit code.
- Suppose we give the link address as *0x7000*, it tries to long jump to *0x7032* instead of *0x7c32* and fails.

Exercise 6. Reset the machine. 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.)

- At first (when BIOS enters the boot loader), the kernel is not yet loaded at 0x100000 and thus the 8 words of memory is just 0x0.
- At the second point (when boot loader enters kernel), bootloader loads the kernel and the addresses have their values and not 0x0.

Exercise 7.

Use QEMU and GDB to trace into the JOS kernel and stop at the movl % eax,% cr0.Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the stepi GDB command. Again, examine memory at 0x00100000 and at 0xf0100000.

- movl % eax,% cr0 — This enables the paging.
- Before enabling the paging, the memory at 0x100000 is *0x1badb002* & the memory at 0xf0100000 is *0x0*.
- After enabling the paging, the memory at both 0x100000 & 0xf0100000 is *0x1badb002*.

What is the first instruction after the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the movl % eax, % cr0 in kern/entry.S, trace into it, and see if you were right.

- The first instruction after the new mapping is established that would fail to work properly if the mapping weren't in place is the jmp instruction at the address **0x10002a**. This instruction needs to jump to the address **0xf010002c**. Since paging is not enabled, this address doesn't exist and results in QEMU to dump the machine state and exit. This causes a hardware exception.

Exercise 8. The code necessary to print octal numbers using patterns of the form %o. Find and fill in this code fragment.

- Filled the octal code fragment in printfmt.c

Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

- The functions defined in printf.c are used in console.c
- console.c uses **cprintf** to log the status of execution like *Rebooting..* & *Serial port doesn't exist.*

Explain the following from console.c:

```
if (crt_pos >= CRT_SIZE) {
  int i;
  memcpy(crt_buf, crt_buf + CRT_COLS,
 (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
  for (i = CRT_SIZE - CRT_COLS; i <
 CRT_SIZE; i++){
     crt_buf[i] = 0x0700 | ' ';
  }
  crt_pos -= CRT_COLS;
}
```

CRT_SIZE ⇒ maximum number of characters in the screen.
CRT_COLS ⇒ number of columns in the screen.
crt_pos ⇒ current position in the input buffer.
If the crt_pos in the input buffer is more than the maximum number of characters in the screen, then this means the screen is completely filled, and the next character from the input buffer should be written in the next line. memmove function expands the crt_buf and the for loop initialises the extra space in crt_buf to whitespaces. Then they are decreasing the crt_pos by CRT_COLS to accommadate the fact that a new line has been created.

Trace the execution of the following code step-by-step

```
  int x = 1, y = 3, z = 4;
  cprintf("x %d, y %x, z %d\n", x, y, z);
```

- In the call to cprintf(), to what does fmt point? To what does ap point?
- List (in order of execution) each call to cons_putc, va_arg, and vcprintf. For cons_putc, list its argument as well. For va_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

fmt points to the first argument given to cprintf.
x %d, y %x, z %d\n
ap points to an array which contains the remaining arguments given to cprintf.
ap = [1, 3, 4]
Order of execution:
vcprintf(char* fmt, va_list ap) ⇒ **va_arg**(va_list ap, type) ⇒ **cons_putc**(int c)
ap is the array containing the arguments to cprintf. Before va_arg is called, ap points to **first element** in the array(Assume this is the first call). After the call, ap points to the **next element** in the array.

Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

- What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise.
- The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

Output is He110 World
Step by step execution is:
cprintf()
⇒ vcprintf()
⇒ vprintfmt() (Swtich case: 'x')
⇒ printnum()
(Switch case: 's')
⇒ putch()
ap is appropriately updated during the execution.
In Big Endian, i = **0x726c6400**

In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

```
cprintf("x=%d y=%d", 3);
```

Output is x=3 y=1124676.
ap just has one value 3 stored in it. After the first va_arg call, the program tries to call va_arg again to print the value of *y*. But this causes a random error because **ap** contains only 1 value. And prints the decimal value of the data at address after where 3 is stored.

Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

- Change the signature of cprintf to include the number of arguments. This is to ensure that we pop only the arguments of cprintf from the stack and nothing else.
  Signature of cprintf is: **int cprintf(const char *fmt, ..., int argNum);**

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

- At the address **0xf0100034**, kernel initialises the stack and **esp** is set as **0xf010fffc** and it's physical address is **0x10fffc**.
- In kern/entry.S, kernel reserved the space for stack in **.data** section as **KSTKSIZE**.
- This **KSTKSIZE** is defined in inc/memlayout.h as **8*PGSIZE**

Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32−bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

- Address of **test_backtrace** is **0xf0100040**.

- Each time **test_backtrace** is called, **ebp** is pushed onto the stack and this is done **6 times**.
- The values of ebp are tabulated in the order in which the ebp values have been pushed to the stack is as follows.

S.No.	value of ebp
1	0xf010fff8
2	0xf010ffd8
3	0xf010ffb8
4	0xf010ff98
5	0xf010ff78
6	0xf010ff58

EXERCISE 11. IMPLEMENT THE BACKTRACE FUNCTION AS SPECIFIED ABOVE

```c
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
   // Your code here.
   cprintf("Stack backtrace:\n");
   uint32_t *ebp;
   ebp = (uint32_t *)read_ebp();
   while (ebp != (uint32_t *)0x0)
   {
      cprintf("ebp %08x eip %08x args ",ebp,*(ebp+1));
      int i = 0;
      uint32_t *arg = ebp + 2;
      for (i = 0; i < 5; ++i)
      {
         cprintf("%08x ",*arg);
         arg ++;
      }
      cprintf("\n");
      ebp = (uint32_t *)*ebp;
   }
   return 0;
}
```

EXERCISE 12.

Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

```c
int
backtrace(int argc, char **argv, struct Trapframe *tf)
{
  uint32_t* base = (uint32_t*) read_ebp();
  cprintf("Stack backtrace:\n");
  while (base) {
    uint32_t eip = base[1];
    cprintf("ebp %x eip %x args", base, eip);
    int i;
    for (i = 2; i <= 6; ++i)
      cprintf(" %08.x", base[i]);
    cprintf("\n");
    struct Eipdebuginfo info;
    debuginfo_eip(eip, &info);
    cprintf("\t%s:%d: %.*s+%d\n", info.eip_file,
        info.eip_line,info.eip_fn_namelen, info.eip_fn_name,eip-info.eip_fn_addr);
    base = (uint32_t*) *base;
  }
  return 0;
}
```