# CS344: Operating Systems Lab
# Assignment 0

Ritish Bansal
190101076

August 20, 2021

## Exercise 1

`__asm__ ( "incl %%eax;" : "=a" (x) : "a" (x) );`
The exact code is provided in ex1.c.

## Exercise 2

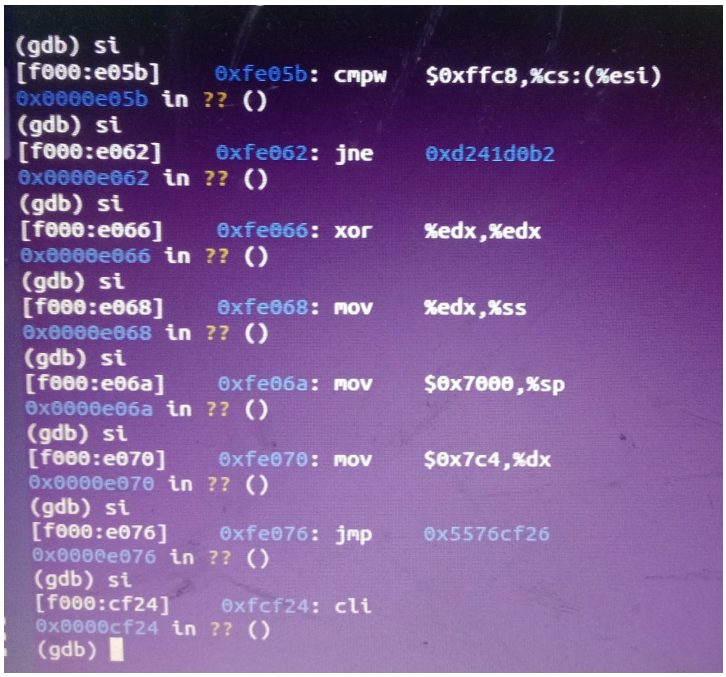



Image. 1: ROM BIOS instructions with si command

compare two operands at specific address.

conditional jump, to check if result of previous cmp is true or false.

take xor of two operands, In this case set value in edx to 0 as a(xor)a=0.

loads value in register ss(stack segment) with value of edx which is 0.
loads value 0x7000 in register sp.

loads value of 0x7c4 in register dx.

jumps to address stored in memory address given.

clear interrupt flag. interrupts disabled when interrupt flag is cleared.

## Exercise 3

Image 2: GDB

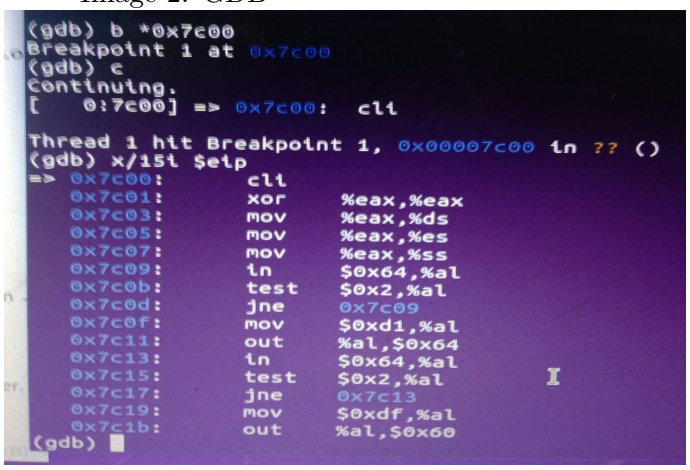



Image 3: Source code in **bootasm.s**

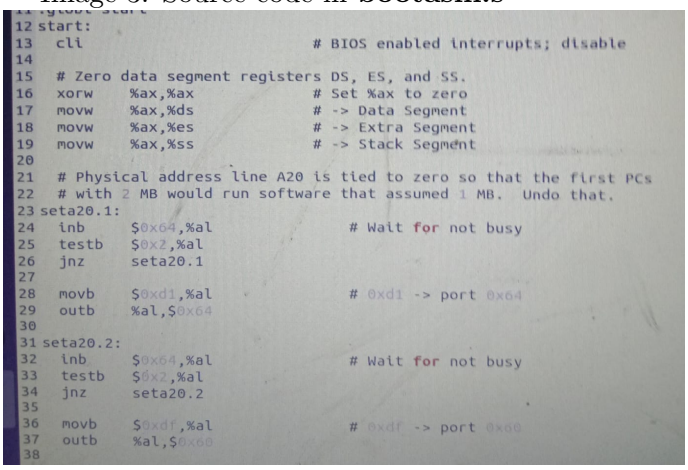


Image 4: Disassembly file,bootblock.asm

```
11 .globl start
12 start:
13   cli                         # BIOS enabled interrupts; disable
14     7c00:       fa                      cli
15
16   # Zero data segment registers DS, ES, and SS.
17   xorw    %ax,%ax             # Set %ax to zero
18     7c01:       31 c0                   xor    %eax,%eax
19   movw    %ax,%ds             # -> Data Segment
20     7c03:       8e d8                   mov    %eax,%ds
21   movw    %ax,%es             # -> Extra Segment
22     7c05:       8e c0                   mov    %eax,%es
23   movw    %ax,%ss             # -> Stack Segment
24     7c07:       8e d0                   mov    %eax,%ss
25
26 00007c09 <seta20.1>:
27
28   # Physical address line A20 is tied to zero so that the first PCs
29   # with 2 MB would run software that assumed 1 MB.  Undo that.
30 seta20.1:
31   inb     $0x64,%al               # Wait for not busy
32     7c09:       e4 64                   in     $0x64,%al
33   testb   $0x2,%al
34     7c0b:       a8 02                   test   $0x2,%al
35   jnz     seta20.1
36     7c0d:       75 fa                   jne    7c09 <seta20.1>
37
38   movb    $0xd1,%al               # 0xd1 -> port 0x64
39     7c0f:       b0 d1                   mov    $0xd1,%al
40   outb    %al,$0x64
41     7c11:       e6 64                   out    %al,$0x64
42
43 00007c13 <seta20.2>:
44
45 seta20.2:
46   inb     $0x64,%al               # Wait for not busy
47     7c13:       e4 64                   in     $0x64,%al
48   testb   $0x2,%al
49     7c15:       a8 02                   test   $0x2,%al
50   jnz     seta20.2
51     7c17:       75 fa                   jne    7c13 <seta20.2>
52
53   movb    $0xdf,%al               # 0xdf -> port 0x60
54     7c19:       b0 df                   mov    $0xdf,%al
55   outb    %al,$0x60
56     7c1b:       e6 60                   out    %al,$0x60
57
58   # Switch from real to protected mode.  Use a bootstrap GDT that makes
```

Image 5: Remaining sectors of kernel, bootblock.asm

```
314     7d8b:       01 de                   add    %ebx,%esi
315   for(; ph < eph; ph++){
316     7d8d:       39 f3                   cmp    %esi,%ebx
317     7d8f:       72 15                   jb     7da6 <bootmain+0x5d>
318   entry();
319     7d91:       ff 15 18 00 01 00       call   *0x10018
320 }
321     7d97:       8d 65 f4                lea    -0xc(%ebp),%esp
322     7d9a:       5b                      pop    %ebx
323     7d9b:       5e                      pop    %esi
324     7d9c:       5f                      pop    %edi
325     7d9d:       5d                      pop    %ebp
326     7d9e:       c3                      ret
327   for(; ph < eph; ph++){
328     7d9f:       83 c3 20                add    $0x20,%ebx
329     7da2:       39 de                   cmp    %ebx,%esi
330     7da4:       76 eb                   jbe    7d91 <bootmain+0x48>
331     pa = (uchar*)ph->paddr;
332     7da6:       8b 7b 0c                mov    0xc(%ebx),%edi
333     readseg(pa, ph->filesz, ph->off);
334     7da9:       83 ec 04                sub    $0x4,%esp
335     7dac:       ff 73 04                pushl  0x4(%ebx)
336     7daf:       ff 73 10                pushl  0x10(%ebx)
337     7db2:       57                      push   %edi
338     7db3:       e8 44 ff ff ff          call   7cfc <readseg>
339     if(ph->memsz > ph->filesz)
340     7db8:       8b 4b 14                mov    0x14(%ebx),%ecx
341     7dbb:       8b 43 10                mov    0x10(%ebx),%eax
342     7dbe:       83 c4 10                add    $0x10,%esp
343     7dc1:       39 c1                   cmp    %eax,%ecx
344     7dc3:       76 da                   jbe    7d9f <bootmain+0x56>
345       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
346     7dc5:       01 c7                   add    %eax,%edi
347     7dc7:       29 c1                   sub    %eax,%ecx
348 }
349
350 static inline void
351 stosb(void *addr, int data, int cnt)
```

Image 6: readsect() in bootmain.c

```
165 00007c90 <readsect>:
166
167 // Read a single sector at offset into dst.
168 void
169 readsect(void *dst, uint offset)
170 {
171     7c90:       f3 0f 1e fb             endbr32
172     7c94:       55                      push   %ebp
173     7c95:       89 e5                   mov    %esp,%ebp
174     7c97:       57                      push   %edi
175     7c98:       53                      push   %ebx
176     7c99:       8b 5d 0c                mov    0xc(%ebp),%ebx
177   // Issue command.
178   waitdisk();
179     7c9c:       e8 dd ff ff ff          call   7c7e <waitdisk>
180 }
181
```

Image 7: readsect() in bootblock.asm

```
58 // Read a single sector at offset into dst.
59 void
60 readsect(void *dst, uint offset)
61 {
62   // Issue command.
63   waitdisk();
64   outb(0x1F2, 1);   // count = 1
65   outb(0x1F3, offset);
66   outb(0x1F4, offset >> 8);
67   outb(0x1F5, offset >> 16);
68   outb(0x1F6, (offset >> 24) | 0xE0);
69   outb(0x1F7, 0x20);  // cmd 0x20 - read sectors
70
71   // Read data.
72   waitdisk();
73   insl(0x1F0, dst, SECTSIZE/4);
74 }
75
```

Image 8: Transition from 16-bit mode to 32-bit mode in bootasm.s

```
38
39   # Switch from real to protected mode.  Use a bootstrap GDT that makes
40   # virtual addresses map directly to physical addresses so that the
41   # effective memory map doesn't change during the transition.
42   lgdt    gdtdesc
43   movl    %cr0, %eax
44   orl     $CR0_PE, %eax
45   movl    %eax, %cr0
46
47 //PAGEBREAK!
48   # Complete the transition to 32-bit protected mode by using a long jmp
49   # to reload %cs and %eip.  The segment descriptors are set up with no
50   # translation, so that the mapping is still the identity mapping.
51   ljmp    $(SEG_KCODE<<3), $start32
52
53 .code32  # Tell assembler to generate 32-bit code now.
54 start32:
```

Image 9: Final boot loader and first kernel instruction

```
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb)
```

Image 10: Remaining sectors of kernel, **bootblock.asm**

```
34  // Load each program segment (ignores ph flags).
35  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
36  eph = ph + elf->phnum;
37  for(; ph < eph; ph++){
38    pa = (uchar*)ph->paddr;
39    readseg(pa, ph->filesz, ph->off);
40    if(ph->memsz > ph->filesz)
41      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
42  }
43
```

Image 2 shows the breakpoint is set at address *0x7c00* where boot sector was loaded. Using x/15i command I have executed next 15 instruction. Image 3 and Image 4 show the source code in bootasm.s and disassembly code in bootblock.asm respectively. Comparing three images we can see that all the 15 instructions are identical except there is some difference in between some keywords of same instruction.
Image 5 shows that instructions from line 327 to 348 read the remaining sectors of kernel from the disk. When the starting loop is finished, it *call *0x10018*(line 319) is executed. We set a breakpoint at the address 0x7d91 using GDB and continue until we reach that that breakpoint as shown in Image 9.
Image 6 and 7 shows code of readsect() function in bootmain.c and bootblock.asm respectively.

- The section shown in Image 8 switches the processor from 16-bit mode to 32-bit mode. The instruction in line 51 causes this switch. All the instructions until this part were executed as 16-bit mode and after this all instructions will be executed as 32-bit mode.

- 0x7d91: *call *0x10018* – last boot loader instruction executed.
  0x10000c: *mov %cr4, %eax* – first kernel instruction.

- The section in Image 10 shows boot loader runs from ph to eph to load kernel. Both the values are obtained from the ELF header. *elf* → *phnum* provides size of loop.

## Exercise 4

Image 11: *objdump -h bootmain.o*

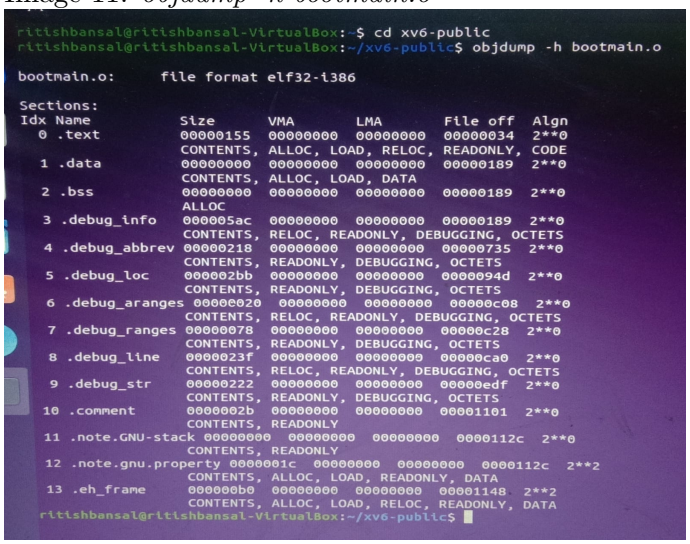
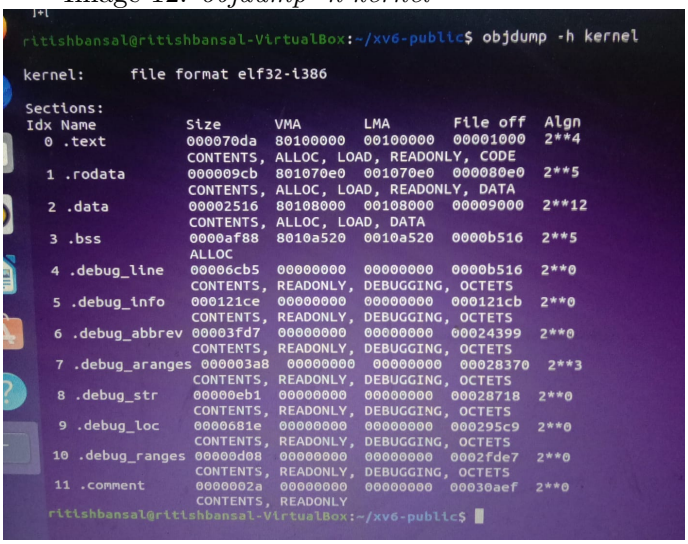



Image 12: *objdump -h kernel*




Here the results in Image 11 and 12 are provided as columns:

- **Size**: This denotes size of sector.

- **VMA**: Link address of section. This is memory address from where the section begins to execute.

- **LMA**: Load address of section. This is memory address from where section should be loaded.

## Exercise 5

The instruction in line 51 of Image 8 in bootasm.s is first line to break if provided address is wrong.

Image 13: Correct link address

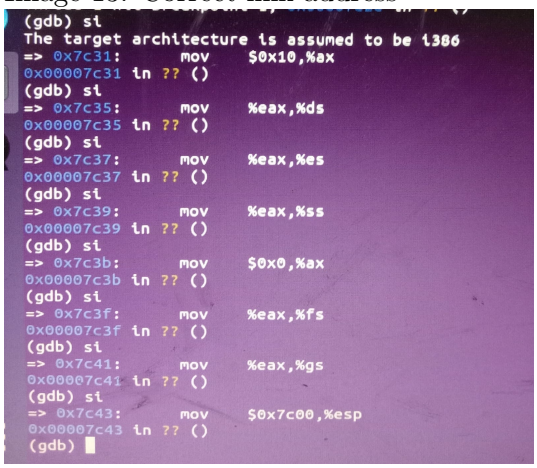
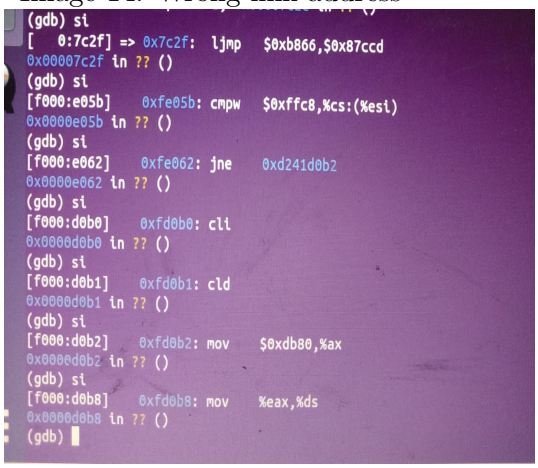
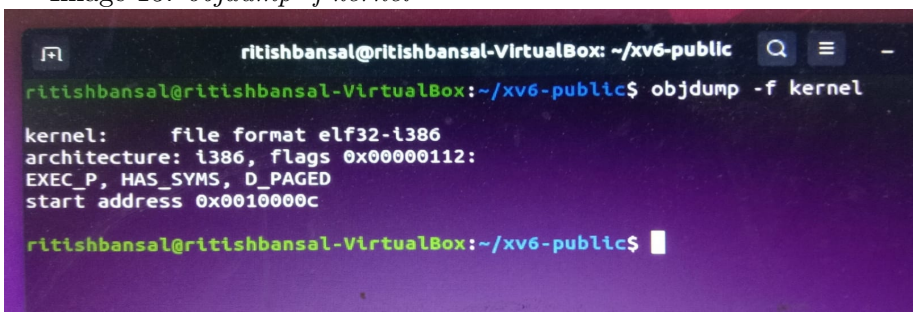



Image 14: Wrong link address



The correct link address given in make file is *0x7c00*. I have changed it to *0x7c99* and then again used make clean and make qemu to reload the boot loader. Here by using gdb we can see that until instruction 51 in bootasm.s the same instructions were executed but after this command there was some different instructions executed in wrong address part.

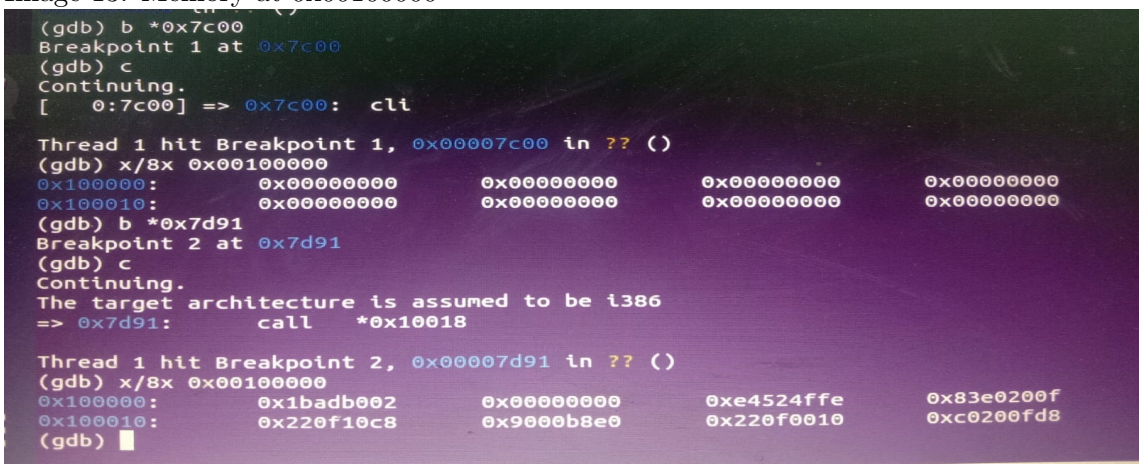
Image 15: *objdump -f kernel*




Entry point address by this command is *0x0010000c*.

## Exercise 6

Image 13: Memory at 0x00100000




Here firstly I have put a breakpoint at address *0x7c00* and continue till this address. As this is starting address and until this no process has been started so there is no useful data in given memory addresses. Also the boot loader loads the kernel into main memory starting from address *0x00100000*.
Here we will get the useful data when all the process by boot loader is complete. So I have put another break point at address *0x7d91* where all process of boot loader are complete and continuing till this address. As we see the memory addresses after this breakpoint is hit, we get some useful information about process.
The second breakpoint should be at end of boot loader process and kernel has been fully loaded into main memory starting from this address.