CS-344 Assignment-0

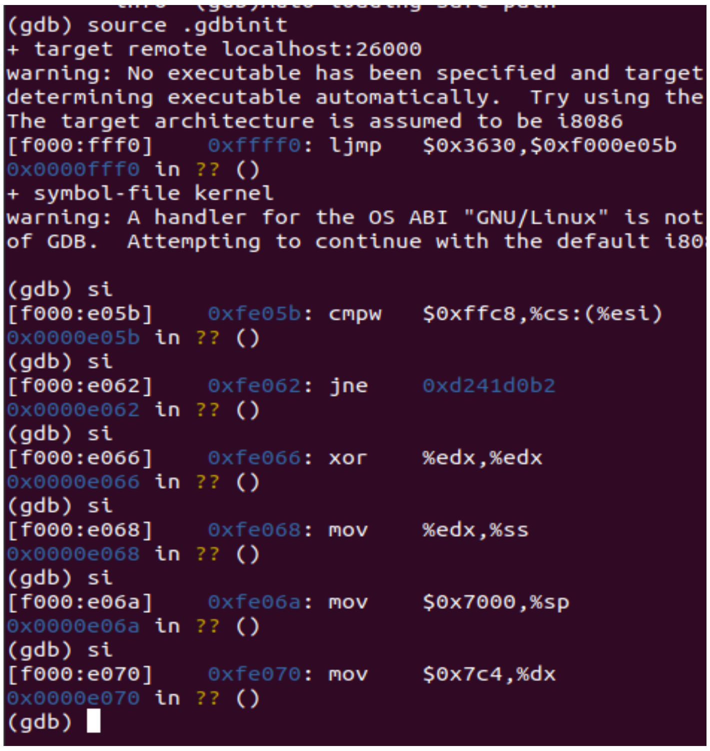
Operating Systems Laboratory
11th August, 2021

- Patnana Sai Ashrritth
- 190101061

Exercise-1:

```
#include<stdio.h>
int main(int argc, char **argv) {
        int x = 1;
        printf("Hello x = %d\n", x);
        // Put in-line assembly here to increment assembly //

        asm("movl %0, %%eax;"    // Moves value of x to the 32 bit eax register
             "addl $1, %%eax;"     // Adds 1 to eax register value
             "movl %%eax, %0;"    // moves eax value to variable x.
               : "+r" (x)     // '+' denotes that operand is both read and write enabled
               :          // no other input operands, so it is blank
               : "%eax");     // %eax is a clobbered register

        printf("Hello x = %d after increment\n", x);
        if(x == 2){
                printf("OK\n");
        }
        else{
                printf("ERROR\n");
        }
}
```

Here, in instructions there are two '%' before *eax* because it helps GCC to distinguish between the operands and registers. operands have a single '%' as prefix, whereas registers have double '%' prefix. And *eax* is a clobbered register which means, GCC won't use this register to store or modify other values to it outside of asm.



```
(gdb) source .gdbinit
+ target remote localhost:26000
warning: No executable has been specified and target
determining executable automatically.  Try using the
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is not
of GDB.  Attempting to continue with the default i80

(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) si
[f000:e070]    0xfe070: mov    $0x7c4,%dx
0x0000e070 in ?? ()
(gdb) 
```

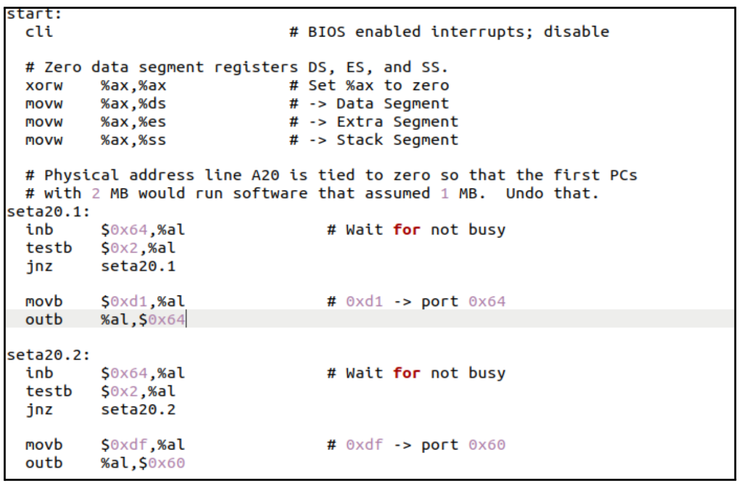
Exercise-2:

[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b
Earlier in the BIOS, jumps to the CS (0x3630). Then jumps to CS = $0xf000 & IP = 0xe05b.
[f000:e05b] 0xfe05b: cmpw $0xffc8,%cs:(%esi)
Compares the contents present at the address 0xffc8 with contents present at the segment offset address. segment = cs & offset = value at esi.
[f000:e062] 0xfe062: jne 0xd241d0b2
If the comparison of the above instruction does not set zero flag(ZF) then jumps to 0xd241d0b2
[f000:e066] 0xfe066: xor %edx,%edx
Since this is a xor operation of the same values, it sets edx to zero. Also ZF was set in the previous instruction so the jump didn't happen.
[f000:e068] 0xfe068: mov %edx,%ss
Moves the contents present at edx to ss (stack segment)
[f000:e06a] 0xfe06a: mov $0x7000,%sp
Moves the contents present at the address $0x7000 to the address pointed by the stack pointer (sp).

[f000:e070] 0xfe070: mov $0x7c4,%dx
Moves the contents present at the data register (dx) to the address $0x7c4

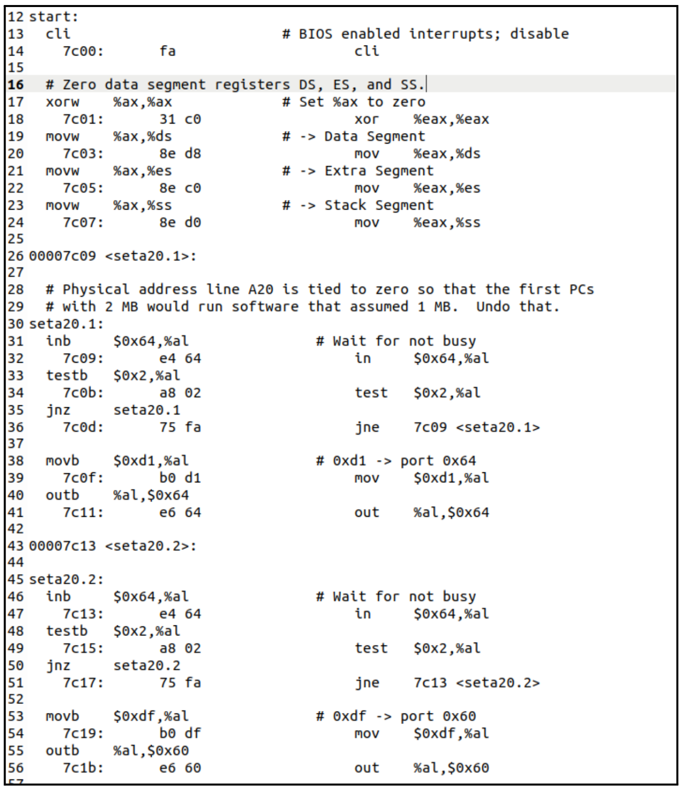
Exercise-3:

bootasm.S



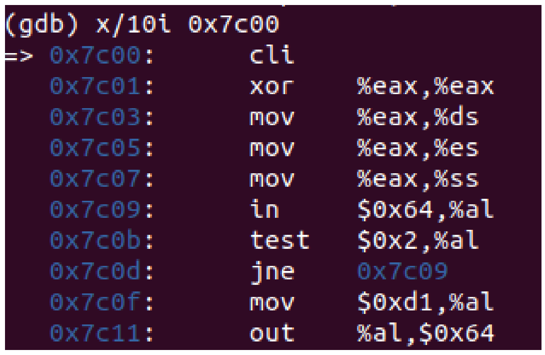
```
start:
  cli                           # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax               # Set %ax to zero
  movw    %ax,%ds               # -> Data Segment
  movw    %ax,%es               # -> Extra Segment
  movw    %ax,%ss               # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al               # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al               # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al               # 0xdf -> port 0x60
  outb    %al,$0x60
```

bootblock.asm



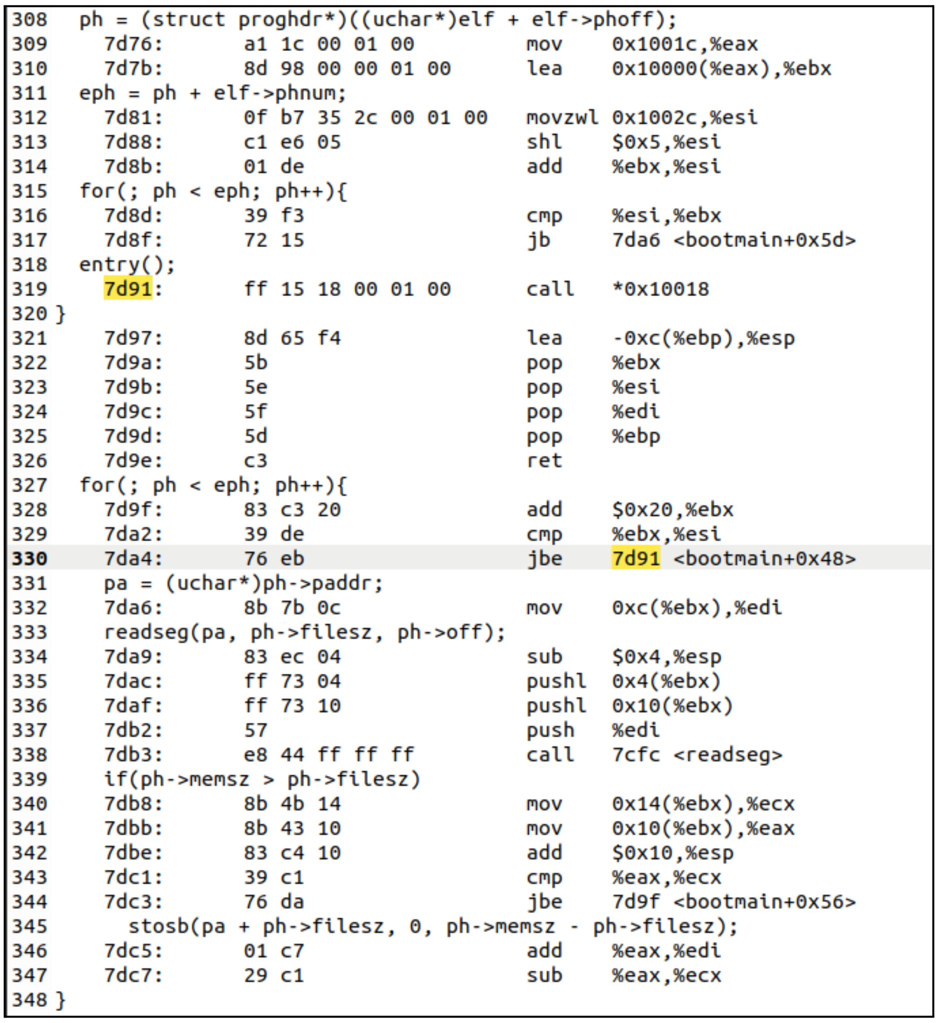
```
start:
  cli                                   # BIOS enabled interrupts; disable
    7c00:       fa                      cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax                       # Set %ax to zero
    7c01:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds                       # -> Data Segment
    7c03:       8e d8                   mov    %eax,%ds
  movw    %ax,%es                       # -> Extra Segment
    7c05:       8e c0                   mov    %eax,%es
  movw    %ax,%ss                       # -> Stack Segment
    7c07:       8e d0                   mov    %eax,%ss

00007c09 <seta20.1>:

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al                     # Wait for not busy
    7c09:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c0b:       a8 02                   test   $0x2,%al
  jnz     seta20.1
    7c0d:       75 fa                   jne    7c09 <seta20.1>

  movb    $0xd1,%al                     # 0xd1 -> port 0x64
    7c0f:       b0 d1                   mov    $0xd1,%al
  outb    %al,$0x64
    7c11:       e6 64                   out    %al,$0x64

00007c13 <seta20.2>:

seta20.2:
  inb     $0x64,%al                     # Wait for not busy
    7c13:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c15:       a8 02                   test   $0x2,%al
  jnz     seta20.2
    7c17:       75 fa                   jne    7c13 <seta20.2>

  movb    $0xdf,%al                     # 0xdf -> port 0x60
    7c19:       b0 df                   mov    $0xdf,%al
  outb    %al,$0x60
    7c1b:       e6 60                   out    %al,$0x60
```

gdb:



```
(gdb) x/10i 0x7c00
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
```

Comparing all three images above, i.e code in bootblock.asm, bootasm.S and first 15 instruction starting from *0x7c00* in gdb. We can see that first 15 are identical

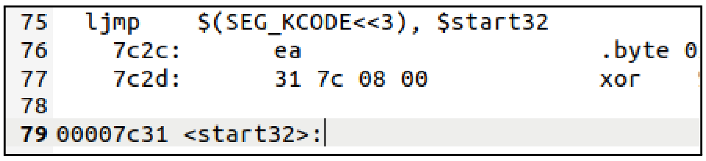


```
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    7d76:       a1 1c 00 01 00          mov    0x1001c,%eax
    7d7b:       8d 98 00 00 01 00       lea    0x10000(%eax),%ebx
  eph = ph + elf->phnum;
    7d81:       0f b7 35 2c 00 01 00    movzwl 0x1002c,%esi
    7d88:       c1 e6 05                shl    $0x5,%esi
    7d8b:       01 de                   add    %ebx,%esi
  for(; ph < eph; ph++){
    7d8d:       39 f3                   cmp    %esi,%ebx
    7d8f:       72 15                   jb     7da6 <bootmain+0x5d>
  entry();
    7d91:       ff 15 18 00 01 00       call   *0x10018
}
    7d97:       8d 65 f4                lea    -0xc(%ebp),%esp
    7d9a:       5b                      pop    %ebx
    7d9b:       5e                      pop    %esi
    7d9c:       5f                      pop    %edi
    7d9d:       5d                      pop    %ebp
    7d9e:       c3                      ret
  for(; ph < eph; ph++){
    7d9f:       83 c3 20                add    $0x20,%ebx
    7da2:       39 de                   cmp    %ebx,%esi
    7da4:       76 eb                   jbe    7d91 <bootmain+0x48>
    pa = (uchar*)ph->paddr;
    7da6:       8b 7b 0c                mov    0xc(%ebx),%edi
    readseg(pa, ph->filesz, ph->off);
    7da9:       83 ec 04                sub    $0x4,%esp
    7dac:       ff 73 04                pushl  0x4(%ebx)
    7daf:       ff 73 10                pushl  0x10(%ebx)
    7db2:       57                      push   %edi
    7db3:       e8 44 ff ff ff          call   7cfc <readseg>
    if(ph->memsz > ph->filesz)
    7db8:       8b 4b 14                mov    0x14(%ebx),%ecx
    7dbb:       8b 43 10                mov    0x10(%ebx),%eax
    7dbe:       83 c4 10                add    $0x10,%esp
    7dc1:       39 c1                   cmp    %eax,%ecx
    7dc3:       76 da                   jbe    7d9f <bootmain+0x56>
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    7dc5:       01 c7                   add    %eax,%edi
    7dc7:       29 c1                   sub    %eax,%ecx
}
```

In the above bootblock.asm, instructions from line 327 to 348 are responsible for the reading of remaining sectors of the kernel from the disk. Now when the loop is finished, instruction at 319 *call *0x10018* gets executed. We set a breakpoint there using GDB and continue till we reach that breakpoint. And this is also the last instruction executed by the boot loader as this completes reading the kernel from the disc and uploading it to the main memory.

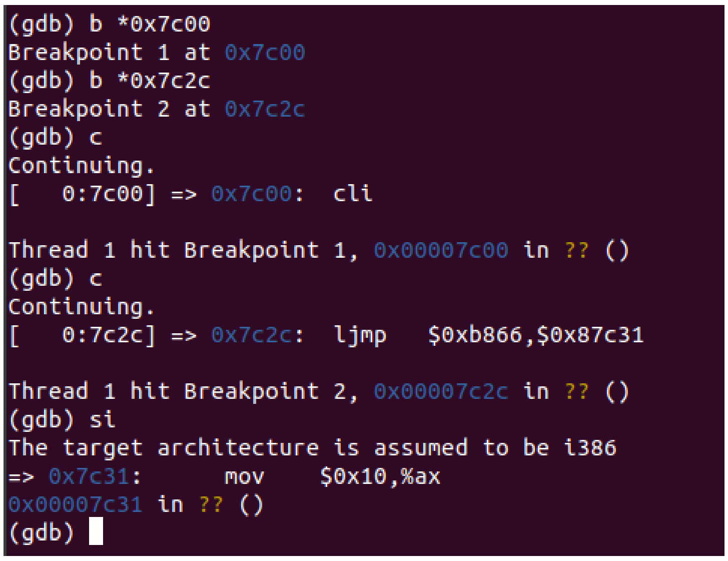
(a)

At 0x7c31 we can see that the target architecture is assumed to be i386 which is 32-bit protected mode, whereas target architecture i8086 means 16 bit real mode which can be seen at the starting of gdb. The command at 0x7c2c ljmp *$(SEG_KCODE<<3), $start32* causes the switch from 16 to 32-bit mode, and the transition occurs at 0x7c31.



```
75   ljmp    $(SEG_KCODE<<3), $start32
76     7c2c:       ea                        .byte 0
77     7c2d:       31 7c 08 00               xor
78
79 00007c31 <start32>:
```

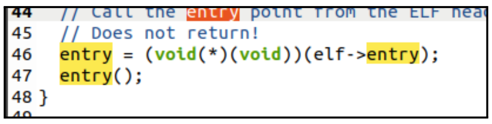
As you can see in line79 *00007c31 <start32>* ,so 0x7c31 is where the transition to 32 bit protected mode happens.



```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x7c2c
Breakpoint 2 at 0x7c2c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31

Thread 1 hit Breakpoint 2, 0x00007c2c in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
```

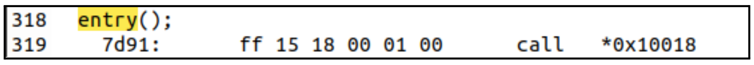
(b)

The last boot loader instruction to be executed in bootmain.c:



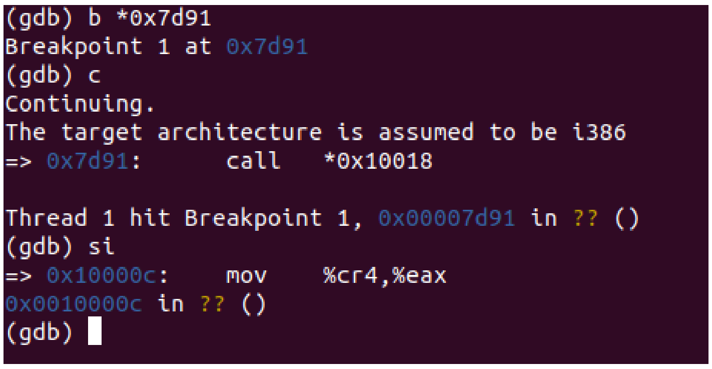
```
  // Call the entry point from the ELF hea
  // Does not return!
  entry = (void(*)(void))(elf->entry);
  entry();
}
```

bootblock.asm:



```
318   entry();
319     7d91:       ff 15 18 00 01 00       call    *0x10018
```

So the instruction at 0x7d91 (from bootblock.asm) is the last instruction in the boot loader to be executed.
Now, according to the beside image we can see that the instruction at *0x10000c* is the first instruction to run.

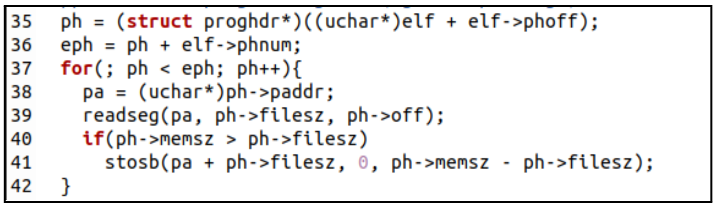


```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

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

So the **0x7d91** is the last instruction in the boot loader, and **0x10000c** is the first instruction in the kernel to be executed.

(c)

As the loop in bootmain.c shows, the bootloader runs a loop starting from *ph* to *eph* to load the kernel. The boot loader reads the number of programs in the ELF header and loads them all. Both values of ph and eph are obtained from the ELF header.
elf->phum determines the size of the loop.

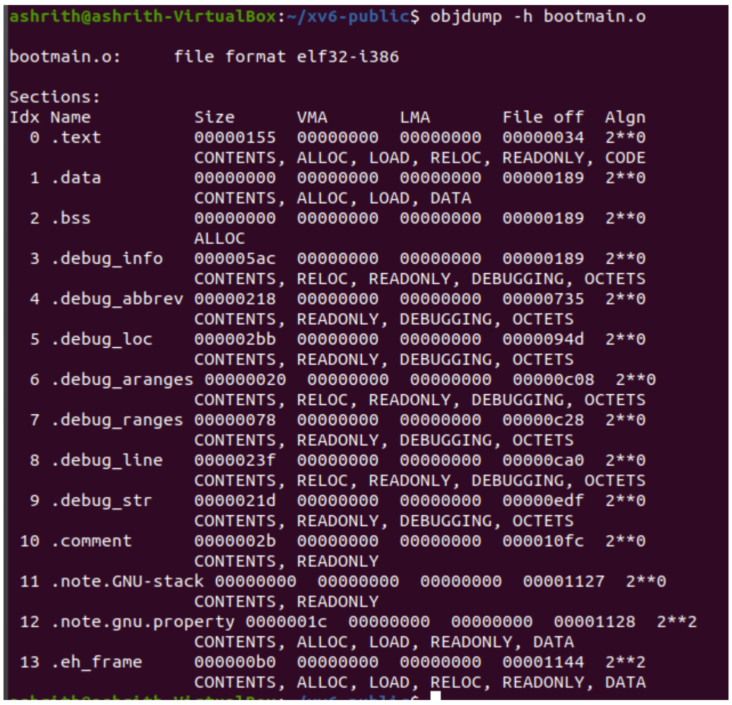


```
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
  for(; ph < eph; ph++){
    pa = (uchar*)ph->paddr;
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
  }
```

BLANK

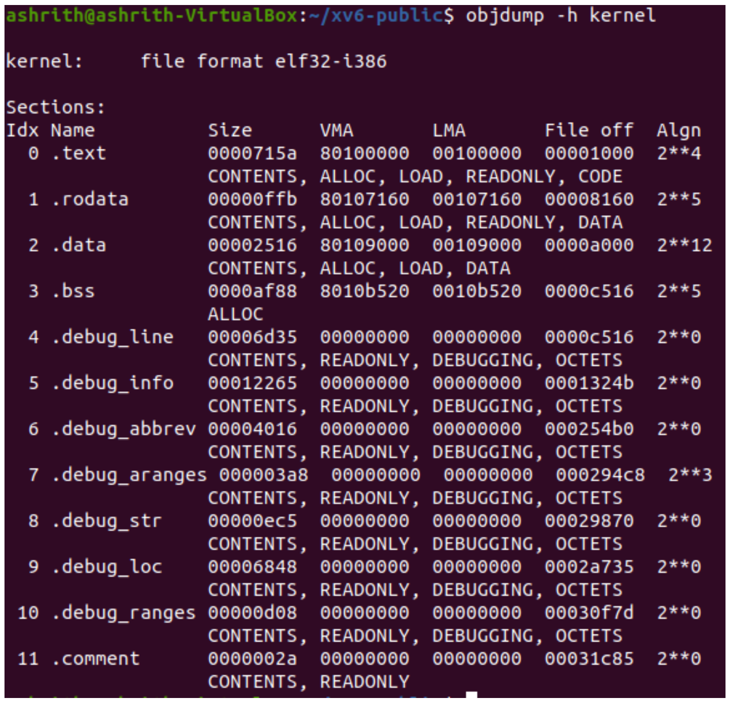
Exercise-4:

a. *objdump -h bootmain.o*



```
ashrith@ashrith-VirtualBox:~/xv6-public$ objdump -h bootmain.o

bootmain.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000155  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000189  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000189  2**0
                  ALLOC
  3 .debug_info   000005ac  00000000  00000000  00000189  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  4 .debug_abbrev 00000218  00000000  00000000  00000735  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_loc    000002bb  00000000  00000000  0000094d  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_aranges 00000020  00000000  00000000  00000c08  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  7 .debug_ranges 00000078  00000000  00000000  00000c28  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line   0000023f  00000000  00000000  00000ca0  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  9 .debug_str    0000021d  00000000  00000000  00000edf  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .comment      0000002b  00000000  00000000  000010fc  2**0
                  CONTENTS, READONLY
 11 .note.GNU-stack 00000000  00000000  00000000  00001127  2**0
                  CONTENTS, READONLY
 12 .note.gnu.property 0000001c  00000000  00000000  00001128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 13 .eh_frame     000000b0  00000000  00000000  00001144  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

b. *Objdump -h kernel*

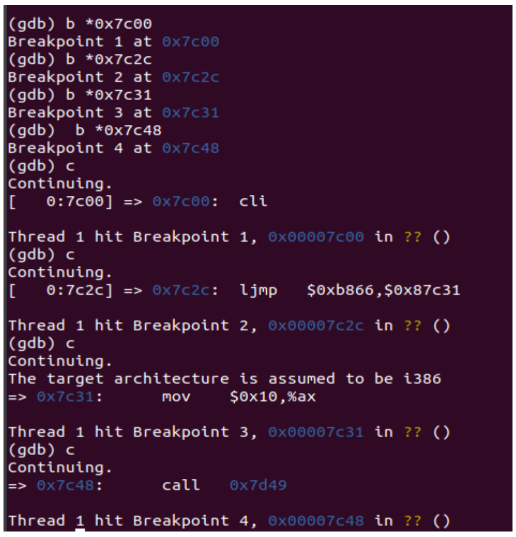


```
ashrith@ashrith-VirtualBox:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000715a  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000ffb  80107160  00107160  00008160  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80109000  00109000  0000a000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010b520  0010b520  0000c516  2**5
                  ALLOC
  4 .debug_line   00006d35  00000000  00000000  0000c516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00012265  00000000  00000000  0001324b  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00004016  00000000  00000000  000254b0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003a8  00000000  00000000  000294c8  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000ec5  00000000  00000000  00029870  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    00006848  00000000  00000000  0002a735  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  00030f7d  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00031c85  2**0
                  CONTENTS, READONLY
```

Size : Size of the section **LMA:** load address of the section **VMA :** link address of the section
Load address is the memory address where the section should be loaded. Whereas link address is the memory address where the section execution begins.
File off: denotes the offset from the beginning of the file **Algn:** this represents the alignment of the section.

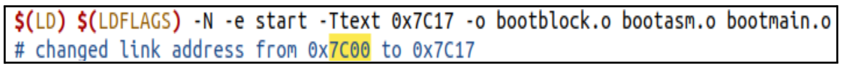
Exercise 5:

Now, before changing the link address, I added a breakpoints at *0x7c00* (address where bootloader starts executing), *0x7c2c* (where the 16bit to 32 bit protected mode transition begins), *0x7c31* (where is transition to 32 bit protected mode happens) and *0x7c48* (random instruction in bootblock.asm).
Before changing the link address all 4 of these breakpoints are hit as shown in the above image.



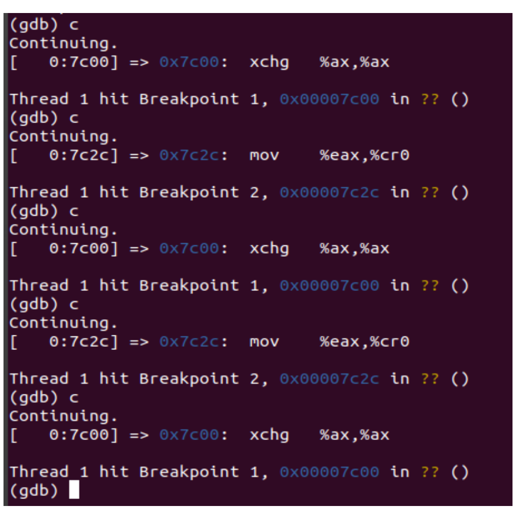
```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x7c2c
Breakpoint 2 at 0x7c2c
(gdb) b *0x7c31
Breakpoint 3 at 0x7c31
(gdb)  b *0x7c48
Breakpoint 4 at 0x7c48
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31

Thread 1 hit Breakpoint 2, 0x00007c2c in ?? ()
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax

Thread 1 hit Breakpoint 3, 0x00007c31 in ?? ()
(gdb) c
Continuing.
=> 0x7c48:      call   0x7d49

Thread 1 hit Breakpoint 4, 0x00007c48 in ?? ()
```

Now after changing the link address of the boot loader to *0x7c17* in makefile and repeating the same steps i got.



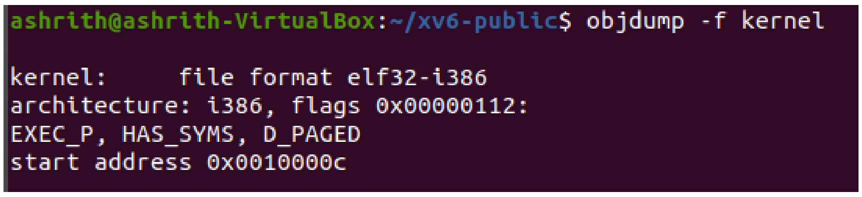
```
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C17 -o bootblock.o bootasm.o bootmain.o
# changed link address from 0x7C00 to 0x7C17
```

Now as shown in the right- below image, the breakpoint at *0x7c00* and *0x7c2c* are being hit repeatedly, but the breakpoint at *0x7c31* is not being hit, which is the very next instruction after *0x7c2c*. So we can say that instruction at *0x7c31* is the first instruction to break after changing the link address of the boot loader.



```
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  xchg   %ax,%ax

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  mov    %eax,%cr0

Thread 1 hit Breakpoint 2, 0x00007c2c in ?? ()
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  xchg   %ax,%ax

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  mov    %eax,%cr0

Thread 1 hit Breakpoint 2, 0x00007c2c in ?? ()
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  xchg   %ax,%ax

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb)
```

objdump -f kernel:

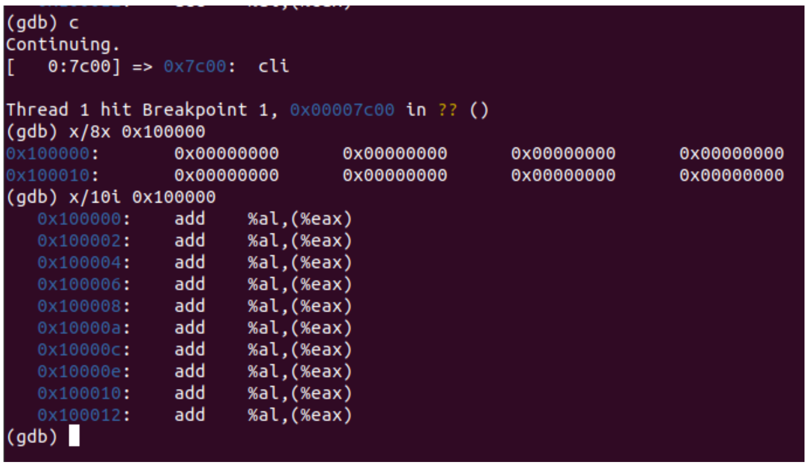


```
ashrith@ashrith-VirtualBox:~/xv6-public$ objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

We can see the link address of the entry point address *0x0010000c*

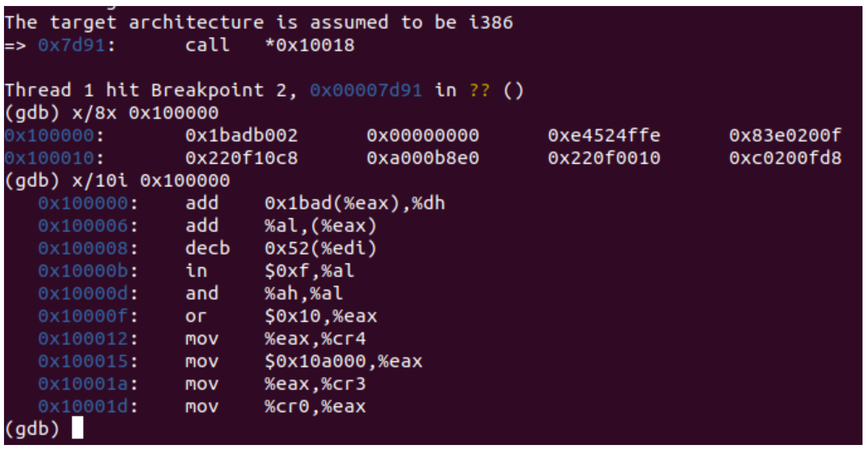
Exercise-6:

The boot loader loads the kernel and it goes into the main memory address at *0x00100000.* Before the boot loader starts running there is no useful information at this location. So the 8 words of memory at *0x00100000* are zero. As shown in the beside image.



```
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/10i 0x100000
   0x100000:    add    %al,(%eax)
   0x100002:    add    %al,(%eax)
   0x100004:    add    %al,(%eax)
   0x100006:    add    %al,(%eax)
   0x100008:    add    %al,(%eax)
   0x10000a:    add    %al,(%eax)
   0x10000c:    add    %al,(%eax)
   0x10000e:    add    %al,(%eax)
   0x100010:    add    %al,(%eax)
   0x100012:    add    %al,(%eax)
(gdb) 
```

Now, in the below image when we look into the second breakpoint at *0x7d91,* which is the last instruction to be executed. The architecture changes from 16 bit to 32 bit protected mode and setting up stack has been done. So the kernel has been fully loaded into the main memory address including at *0x100000.* So the words at *0x100000* are different as shown in the below image.



```
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0xa000b8e0      0x220f0010      0xc0200fd8
(gdb) x/10i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0xf,%al
   0x10000d:    and    %ah,%al
   0x10000f:    or     $0x10,%eax
   0x100012:    mov    %eax,%cr4
   0x100015:    mov    $0x10a000,%eax
   0x10001a:    mov    %eax,%cr3
   0x10001d:    mov    %cr0,%eax
(gdb) 
```