CS 344 Assignment 0 Report

Aditya Trivedi
190101005

Exercise 1 : Inline Assembly in C

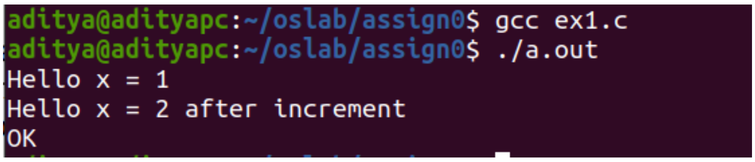
Inline assembly code in C has following format

```
__asm__ ( "assembly code"
        : output operands             /* optional */
        : input operands              /* optional */
        : list of clobbered registers     /* optional */
);
```

So we use following inline assembly code to increment value of x in ex1.c
__asm__("incl %%eax;" : "=a" (x) : "a" (x));

* incl instruction increments the operand by 1.
* %eax The register %eax is used as both the input and the output variable.
* x input is read to %eax and updated %eax is stored in x again after increment.

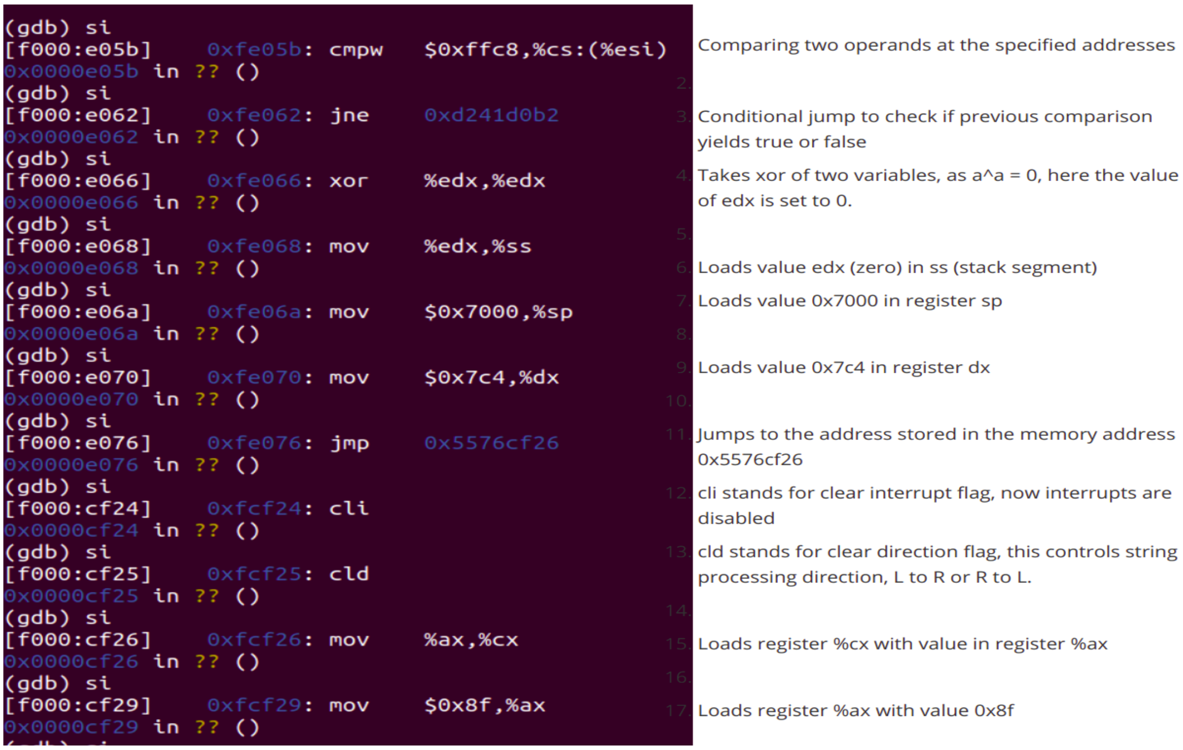
Successful compilation and output



```
aditya@adityapc:~/oslab/assign0$ gcc ex1.c
aditya@adityapc:~/oslab/assign0$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
```

Exercise 2 : GDB

ROM BIOS instructions traced using the si command (Refer Image)



```
(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) si
[f000:e076]    0xfe076: jmp    0x5576cf26
0x0000e076 in ?? ()
(gdb) si
[f000:cf24]    0xfcf24: cli
0x0000cf24 in ?? ()
(gdb) si
[f000:cf25]    0xfcf25: cld
0x0000cf25 in ?? ()
(gdb) si
[f000:cf26]    0xfcf26: mov    %ax,%cx
0x0000cf26 in ?? ()
(gdb) si
[f000:cf29]    0xfcf29: mov    $0x8f,%ax
0x0000cf29 in ?? ()
```

Comparing two operands at the specified addresses

Conditional jump to check if previous comparison yields true or false

Takes xor of two variables, as a^a = 0, here the value of edx is set to 0.

Loads value edx (zero) in ss (stack segment)

Loads value 0x7000 in register sp

Loads value 0x7c4 in register dx

Jumps to the address stored in the memory address 0x5576cf26

cli stands for clear interrupt flag, now interrupts are disabled

cld stands for clear direction flag, this controls string processing direction, L to R or R to L.

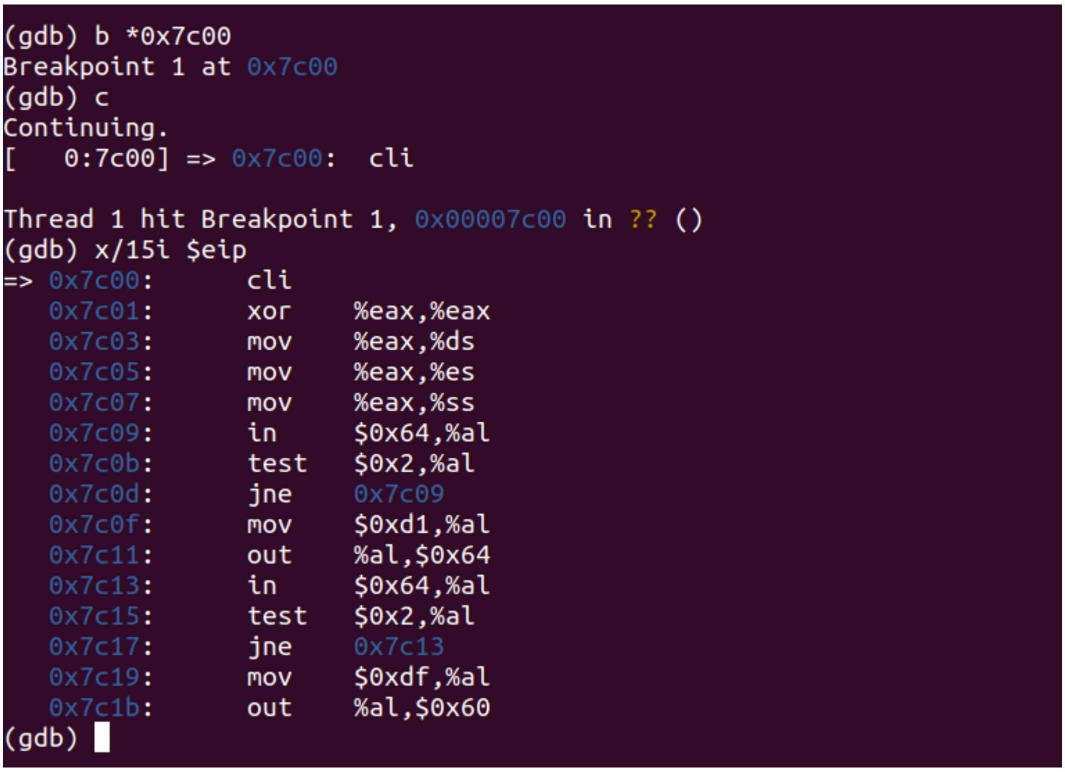
Loads register %cx with value in register %ax

Loads register %ax with value 0x8f

So the bootloader firstly initialises the PCI bus and other devices and then searches for the bootable drive. In the above tracing, we can see that lots of memory locations are being initialised and flags are being set.

Exercise 3 : Bootloader

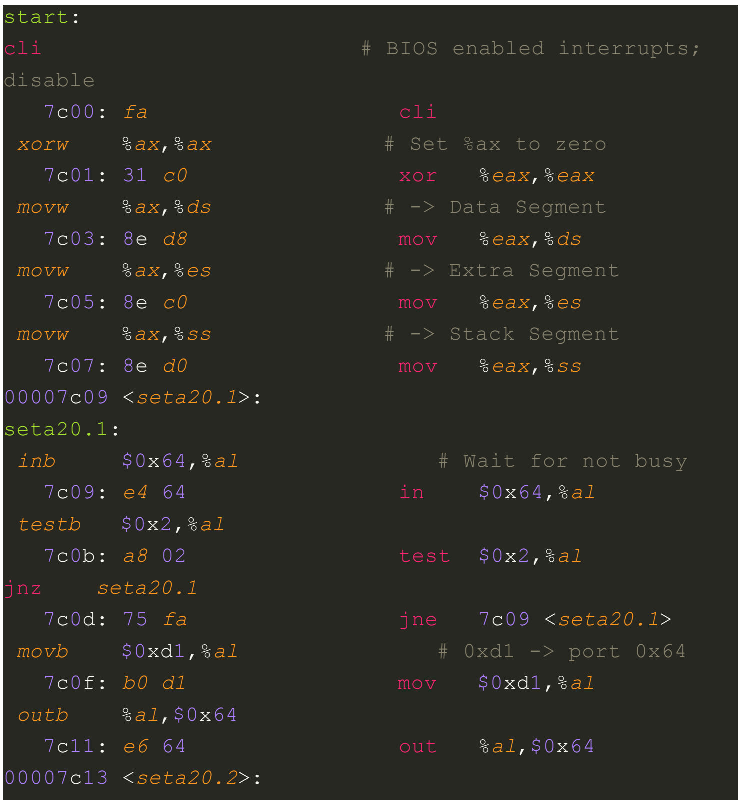
Firstly we set breakpoint at address 0x7c00 using b* command.
Then using x/15i $eip we display the 15 assembly instructions starting at the current instruction pointer.



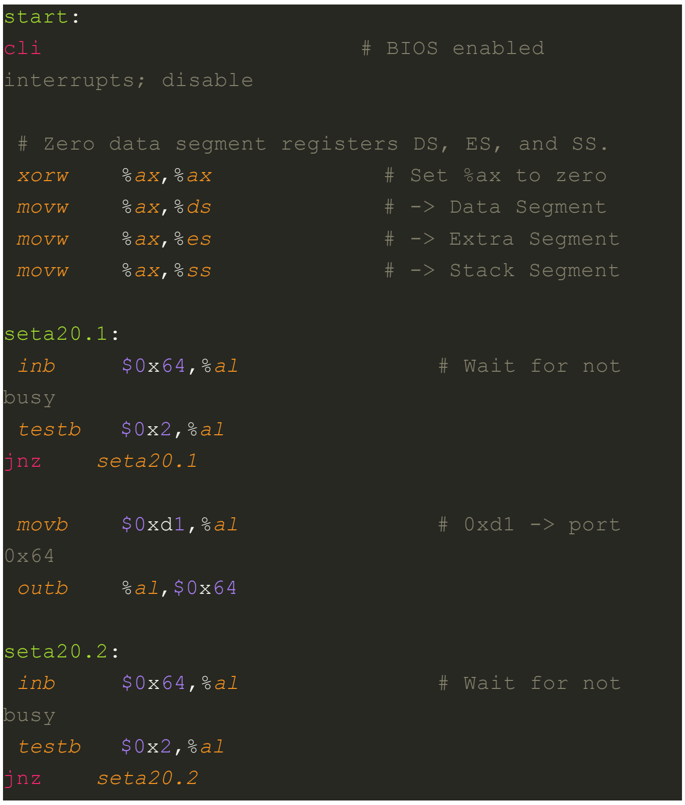
```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/15i $eip
=> 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
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
(gdb) 
```

Upon comparing GDB, with disassembly in bootblock.asm and bootasm.S we find that the first 15 instructions are identical in all. Only, there are a few differences in how some keywords are written.

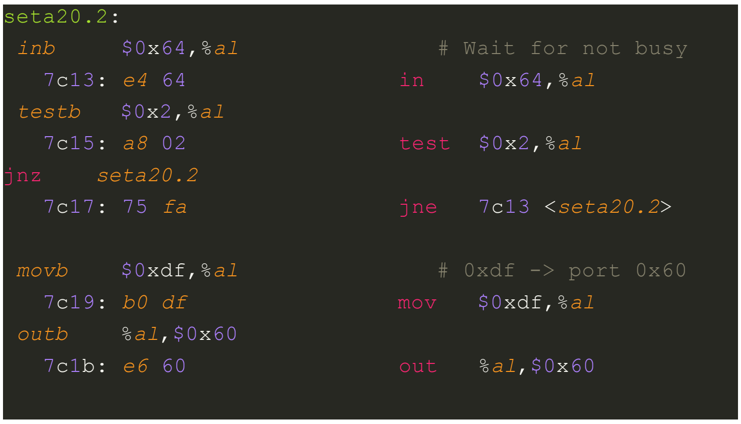
Corresponding code in bootblock.asm and bootasm.S, which is identical to GDB trace



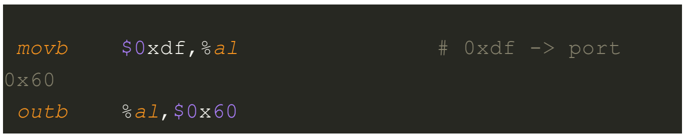
```
start:
cli                           # BIOS enabled interrupts;
disable
   7c00: fa                    cli
 xorw    %ax,%ax              # Set %ax to zero
   7c01: 31 c0                 xor    %eax,%eax
 movw    %ax,%ds              # -> Data Segment
   7c03: 8e d8                 mov    %eax,%ds
 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>:
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>:
```



```
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

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
```

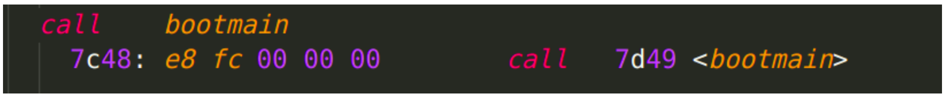


```
seta20.2:
 inb     $0x64,%al                    # Wait for not busy
   7c13: e4 64                  in     $0x64,%al
 testb   $0x2,%al
   7c15: a8 02                  test   $0x2,%al
jnz     seta20.2
   7c17: 75 fa                  jne    7c13 <seta20.2>

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



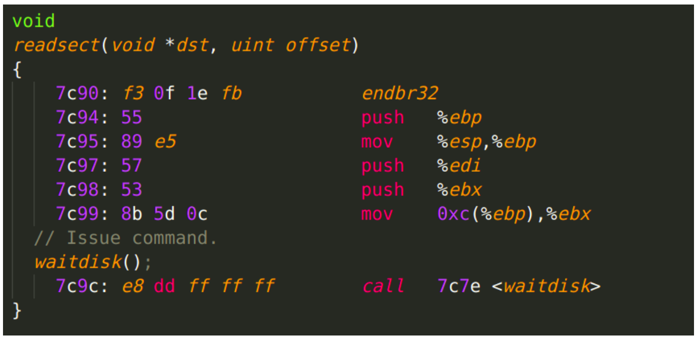
```
 movb    $0xdf,%al                    # 0xdf -> port
0x60
 outb    %al,$0x60
```

bootmain.c is called



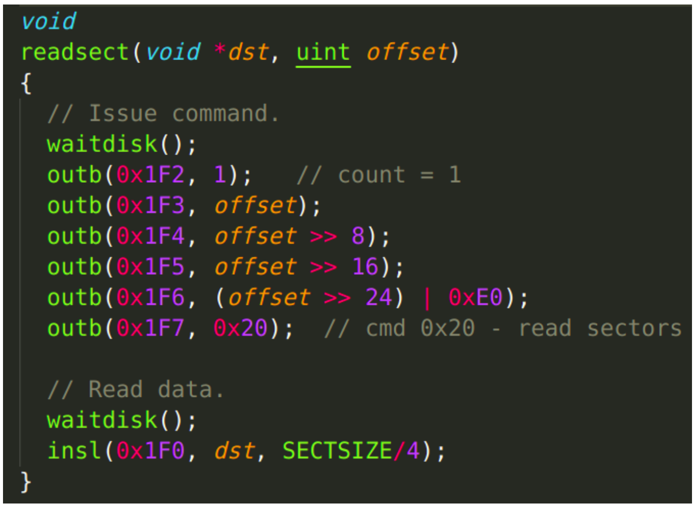
```
  call    bootmain
    7c48: e8 fc 00 00 00          call   7d49 <bootmain>
```

readsect is called at 7c90 in **bootblock.asm**



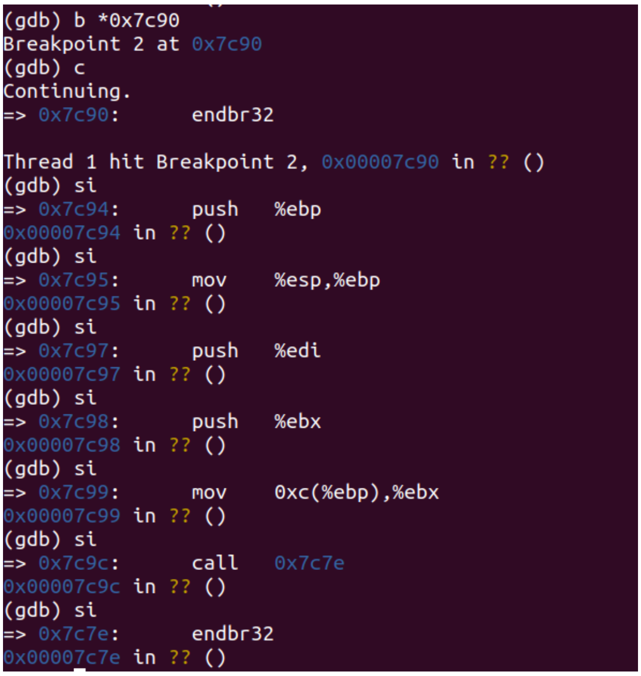
```
void
readsect(void *dst, uint offset)
{
    7c90: f3 0f 1e fb             endbr32
    7c94: 55                      push   %ebp
    7c95: 89 e5                   mov    %esp,%ebp
    7c97: 57                      push   %edi
    7c98: 53                      push   %ebx
    7c99: 8b 5d 0c                mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c9c: e8 dd ff ff ff          call   7c7e <waitdisk>
}
```

readsect in **bootmain.c**



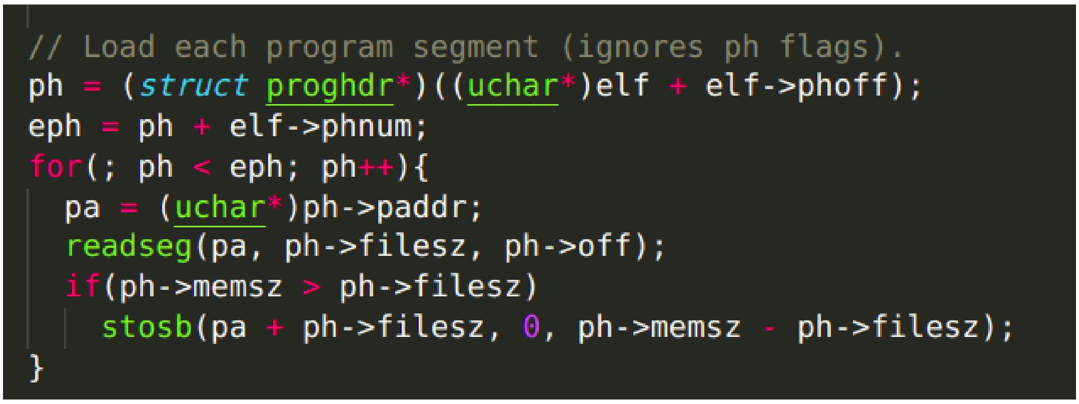
```
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
  insl(0x1F0, dst, SECTSIZE/4);
}
```

Corresponding readsect traced in gdb



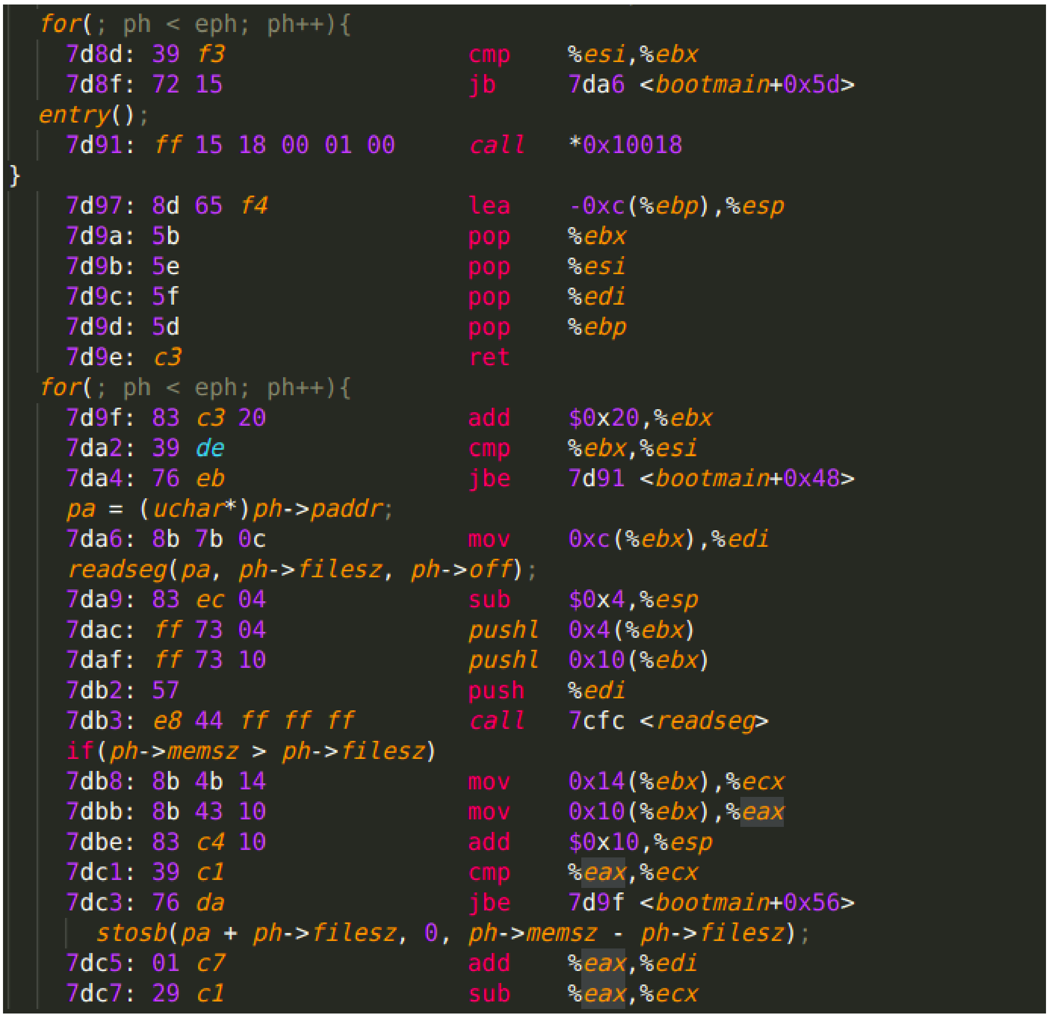
```
(gdb) b *0x7c90
Breakpoint 2 at 0x7c90
(gdb) c
Continuing.
=> 0x7c90:      endbr32

Thread 1 hit Breakpoint 2, 0x00007c90 in ?? ()
(gdb) si
=> 0x7c94:      push   %ebp
0x00007c94 in ?? ()
(gdb) si
=> 0x7c95:      mov    %esp,%ebp
0x00007c95 in ?? ()
(gdb) si
=> 0x7c97:      push   %edi
0x00007c97 in ?? ()
(gdb) si
=> 0x7c98:      push   %ebx
0x00007c98 in ?? ()
(gdb) si
=> 0x7c99:      mov    0xc(%ebp),%ebx
0x00007c99 in ?? ()
(gdb) si
=> 0x7c9c:      call   0x7c7e
0x00007c9c in ?? ()
(gdb) si
=> 0x7c7e:      endbr32
0x00007c7e in ?? ()
```

Following code in bootmain.c is responsible for **reading the remaining sectors** of the kernel from the disk



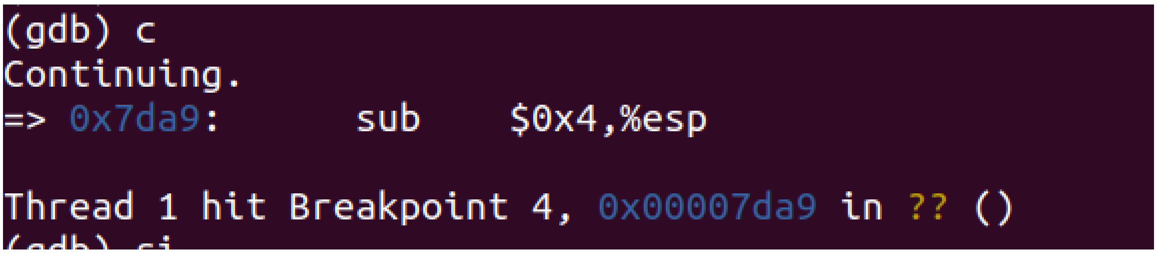
```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
for(; ph < eph; ph++){
  pa = (uchar*)ph->paddr;
  readseg(pa, ph->filesz, ph->off);
  if(ph->memsz > ph->filesz)
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
}
```

Corresponding disassembled



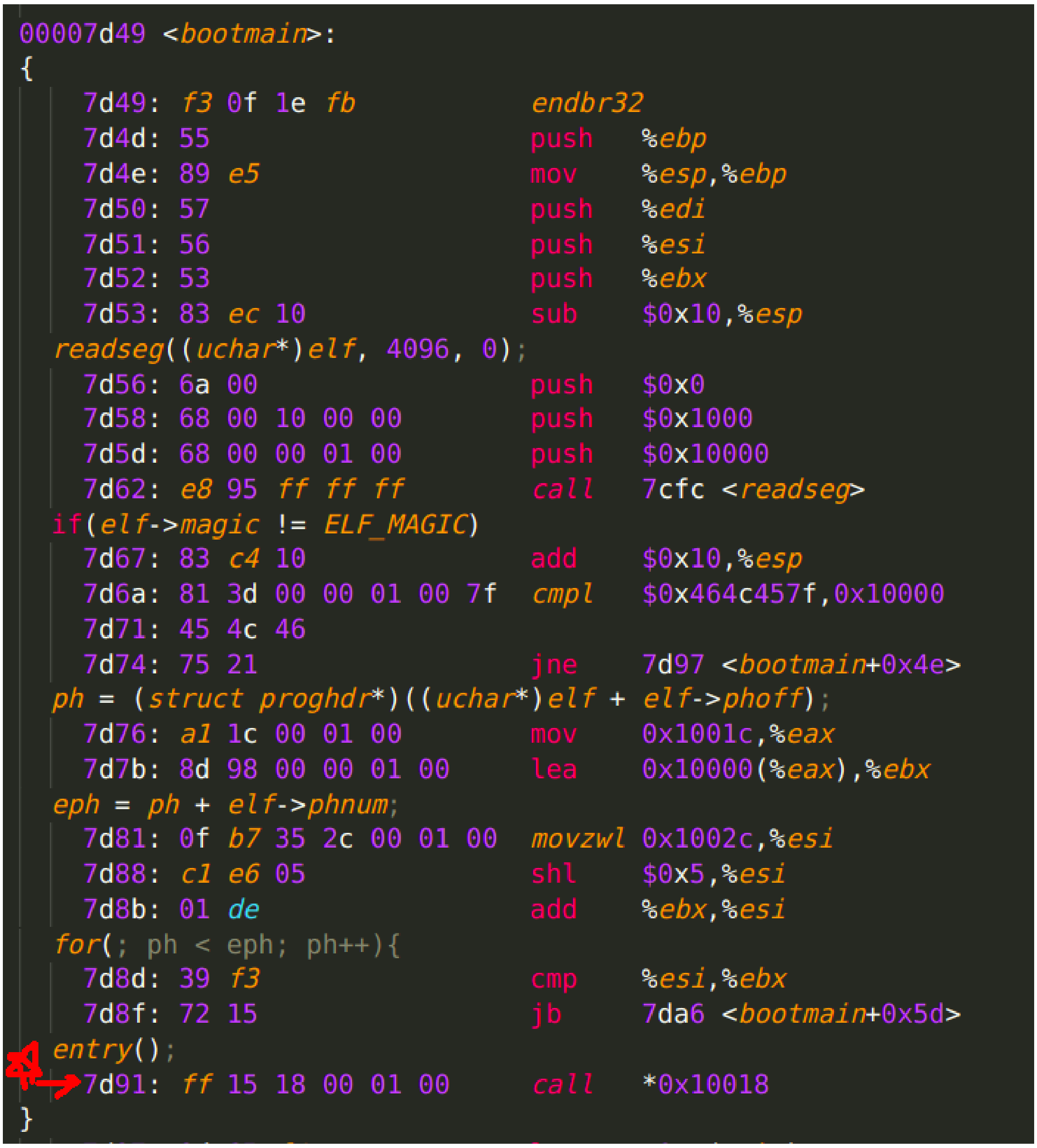
```
  for(; ph < eph; ph++){
    7d8d: 39 f3                cmp    %esi,%ebx
    7d8f: 72 15                jb     7da6 <bootmain+0x5d>
  entry();
    7d91: ff 15 18 00 01 00    call   *0x10018
}
    7d97: 8d 65 f4             lea    -0xc(%ebp),%esp
    7d9a: 5b                   pop    %ebx
    7d9b: 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
```

Marking 0x7da9 as breakpoint

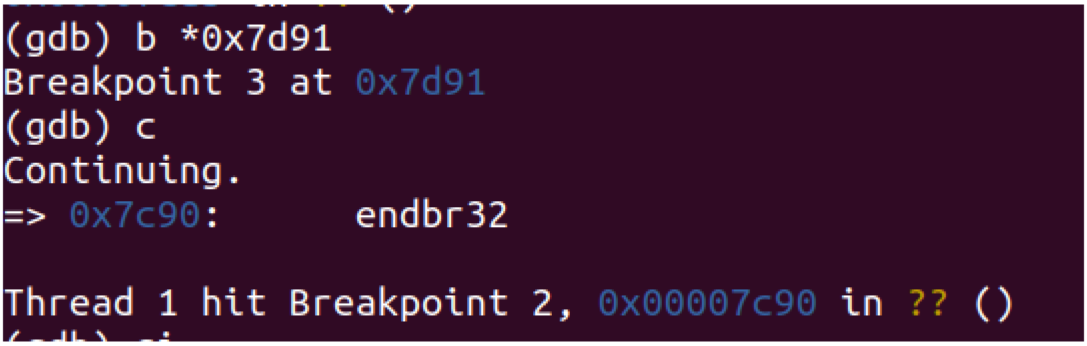


```
(gdb) c
Continuing.
=> 0x7da9:      sub    $0x4,%esp

Thread 1 hit Breakpoint 4, 0x00007da9 in ?? ()
(gdb) si
```

Now bootmain.c ends at 0x7d91, where we put an endpoint, this marks the end of the bootloader and it is the **last instruction executed**. (marked in image)

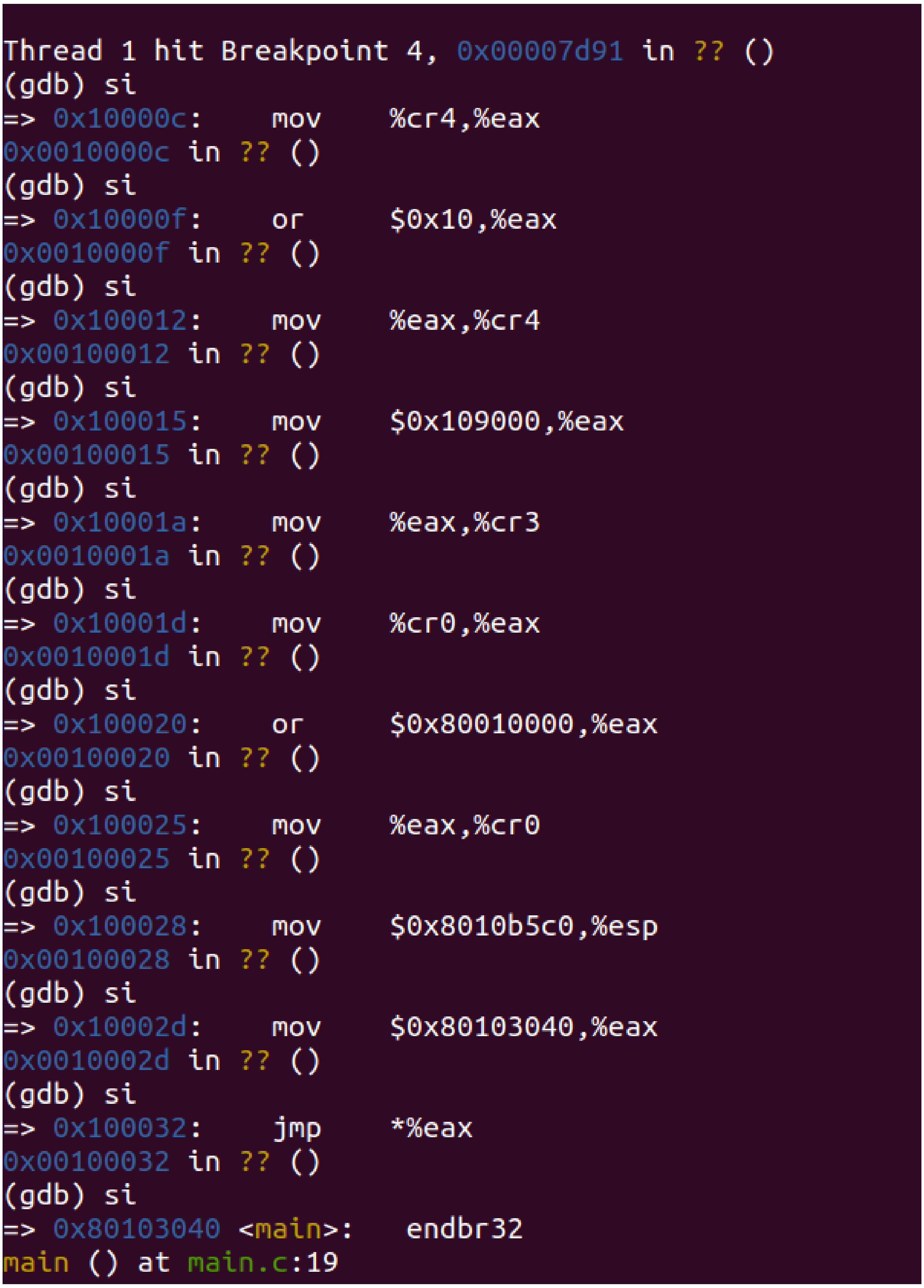


```
00007d49 <bootmain>:
{
    7d49: f3 0f 1e fb           endbr32
    7d4d: 55                    push   %ebp
    7d4e: 89 e5                 mov    %esp,%ebp
    7d50: 57                    push   %edi
    7d51: 56                    push   %esi
    7d52: 53                    push   %ebx
    7d53: 83 ec 10              sub    $0x10,%esp
  readseg((uchar*)elf, 4096, 0);
    7d56: 6a 00                 push   $0x0
    7d58: 68 00 10 00 00        push   $0x1000
    7d5d: 68 00 00 01 00        push   $0x10000
    7d62: e8 95 ff ff ff        call   7cfc <readseg>
  if(elf->magic != ELF_MAGIC)
    7d67: 83 c4 10              add    $0x10,%esp
    7d6a: 81 3d 00 00 01 00 7f  cmpl   $0x464c457f,0x10000
    7d71: 45 4c 46
    7d74: 75 21                 jne    7d97 <bootmain+0x4e>
  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
}
```



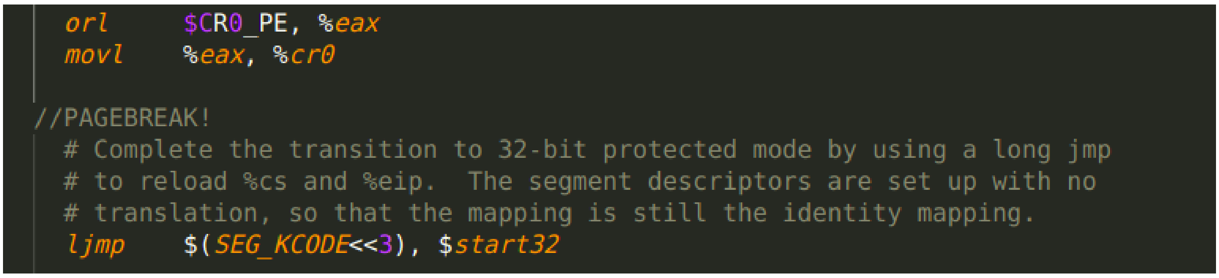
```
(gdb) b *0x7d91
Breakpoint 3 at 0x7d91
(gdb) c
Continuing.
=> 0x7c90:      endbr32

Thread 1 hit Breakpoint 2, 0x00007c90 in ?? ()
```

Now running step by step



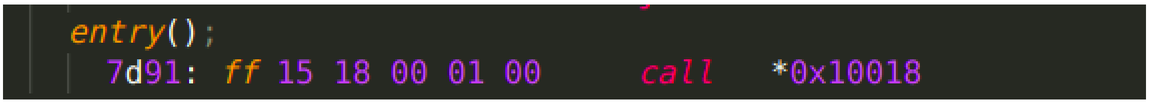
```
Thread 1 hit Breakpoint 4, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x109000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010000,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0x8010b5c0,%esp
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    mov    $0x80103040,%eax
0x0010002d in ?? ()
(gdb) si
=> 0x100032:    jmp    *%eax
0x00100032 in ?? ()
(gdb) si
=> 0x80103040 <main>:   endbr32
main () at main.c:19
```

(a) Instruction given in line 51 causes the processor to start executing 32-bit code. Up Till this point all instructions were executed in 16 bit mode. The exact code responsible is :



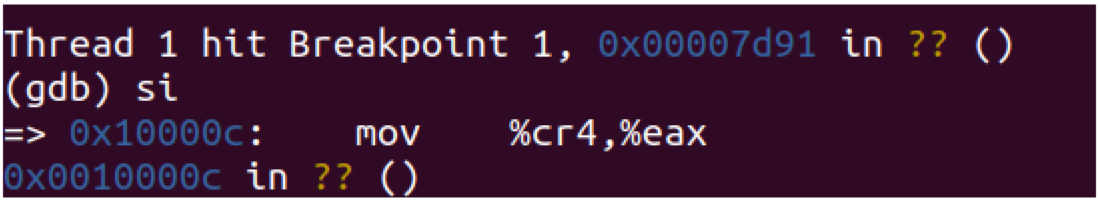
```
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32
```

(b) **Last instruction** of bootloader is at entry() :
call *0x10018



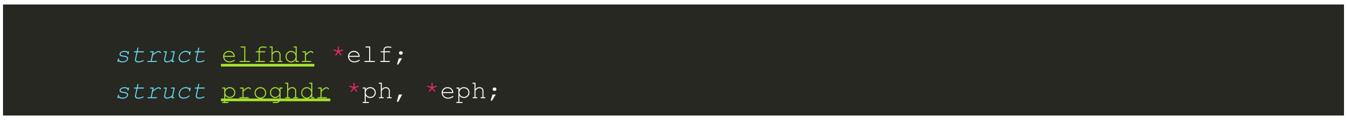
```
entry();
  7d91:  ff 15 18 00 01 00       call   *0x10018
```

First instruction of kernel it loaded is
mov %cr4, %eax



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

(c) Bootloader extracts starting and ending pointers and iterates using a for-loop. It gets this information completely from the program header. Bootmain.c uses the following struct pointers to decide how many sectors to read & fetch the entire kernel from disk



```
        struct elfhdr *elf;
        struct proghdr *ph, *eph;
```

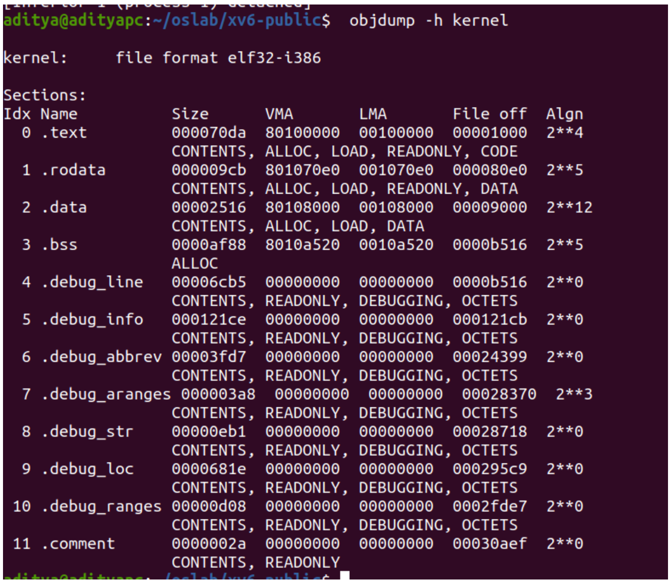
Exercise 4 : objdump

VMA : link address
LMA : load address

Memory address from where the section should begins to execute is called link address and address where the section should be loaded is called load address.

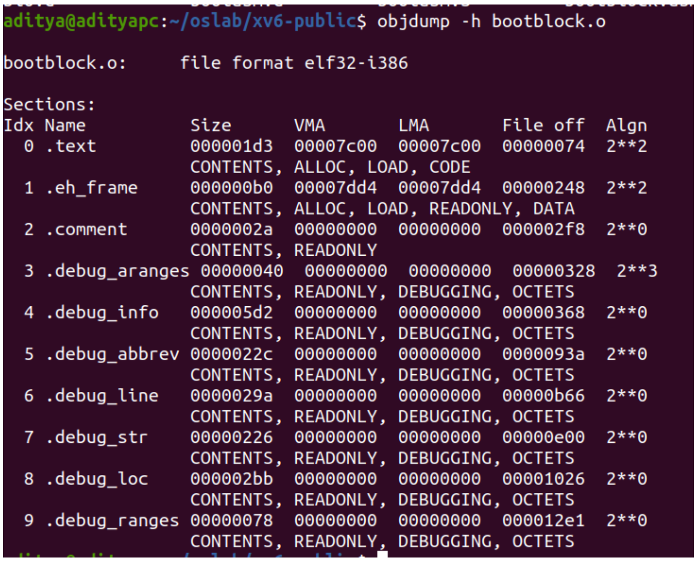
Screenshots of output :

objdump -h kernel



```
aditya@adityapc:~/oslab/xv6-public$  objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000070da  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801070e0  001070e0  000080e0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006cb5  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   000121ce  00000000  00000000  000121cb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00003fd7  00000000  00000000  00024399  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003a8  00000000  00000000  00028370  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000eb1  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde7  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030aef  2**0
                  CONTENTS, READONLY
```

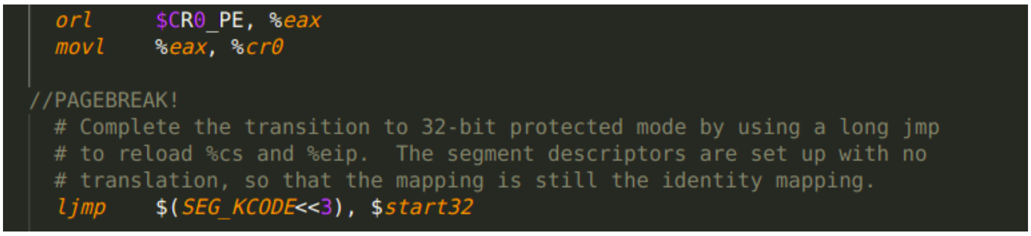
objdump -h bootblock.o



```
aditya@adityapc:~/oslab/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001d3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dd4  00007dd4  00000248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      0000002a  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000328  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000368  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000022c  00000000  00000000  0000093a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   0000029a  00000000  00000000  00000b66  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000226  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001026  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012e1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

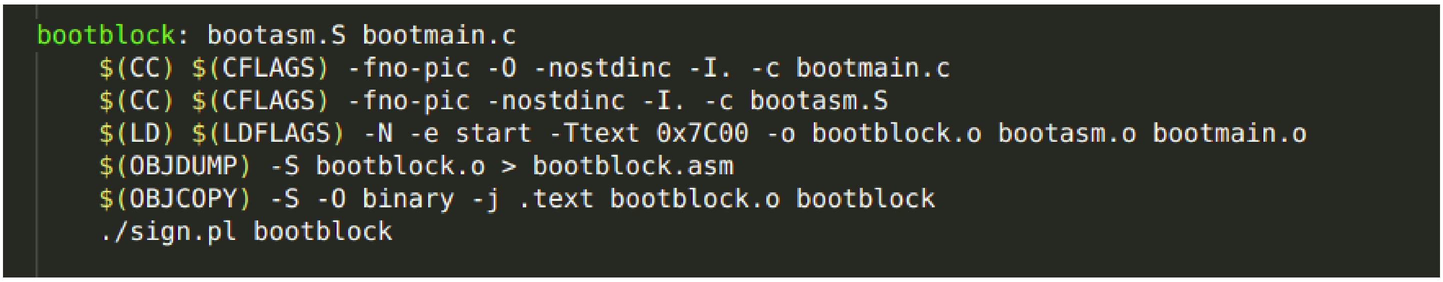
Exercise 5 : Modifying link address

The following code will break if the boot loader's link address was not provided correctly.



```
    orl     $CR0_PE, %eax
    movl    %eax, %cr0

//PAGEBREAK!
  # Complete the transition to 32-bit protected mode by using a long jmp
  # to reload %cs and %eip.  The segment descriptors are set up with no
  # translation, so that the mapping is still the identity mapping.
  ljmp    $(SEG_KCODE<<3), $start32
```

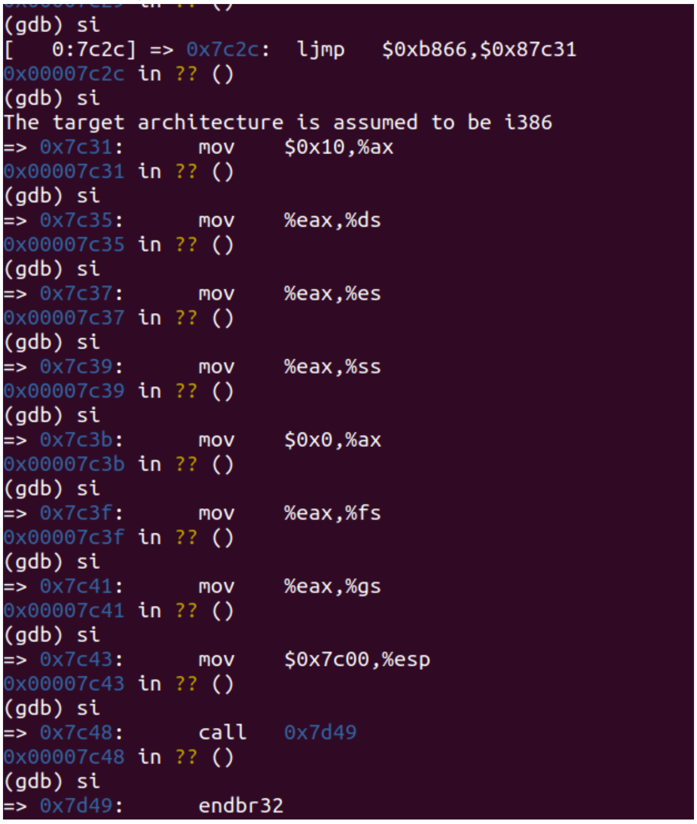
Changing the link address in the Makefile from 0x7c00 to 0x7c42



```
bootblock: bootasm.S bootmain.c
	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
	$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
	$(OBJDUMP) -S bootblock.o > bootblock.asm
	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
	./sign.pl bootblock
```

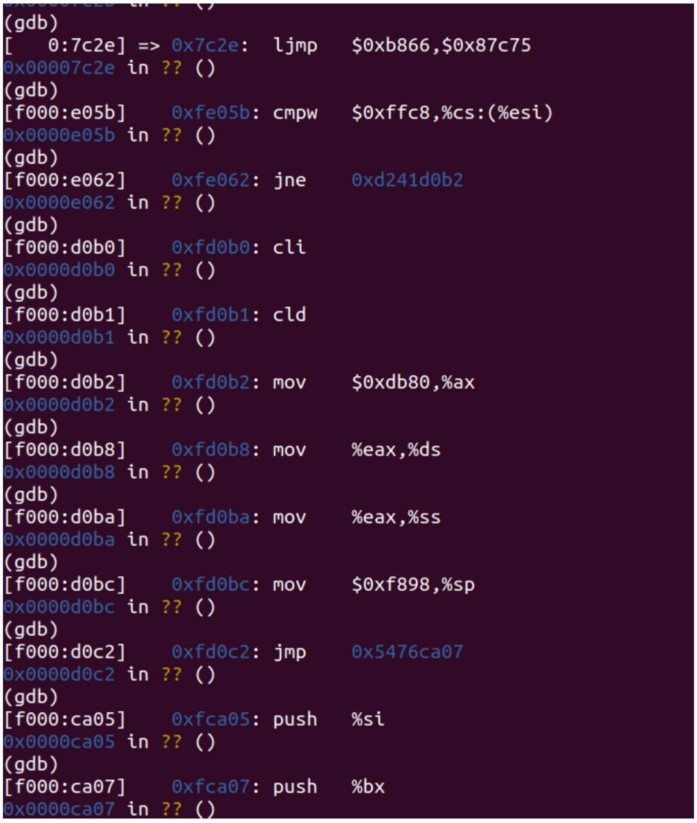
Upon changing the link address in the Makefile and setting it to a junk value, we can observe using gdb that post the ljmp instruction, all instructions are different, and executed incorrectly.

Correct ouput at 0x7c00



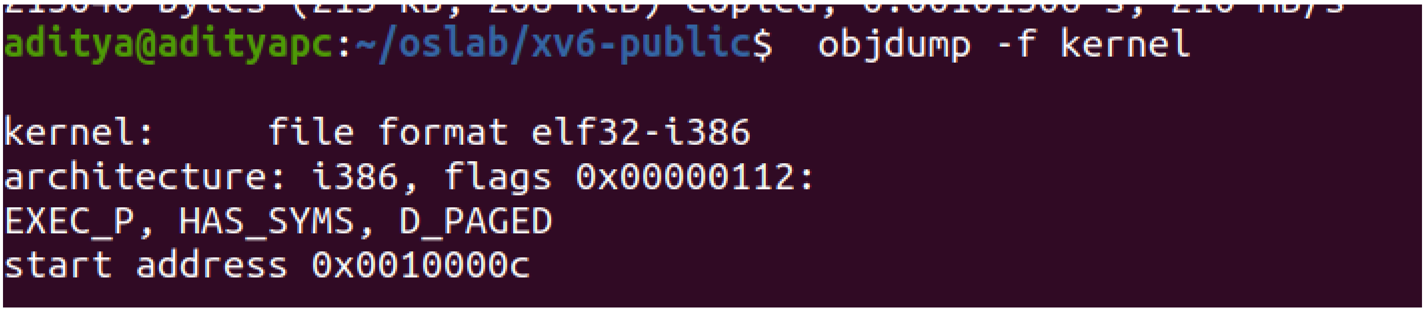
```
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb) si
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
(gdb) si
=> 0x7c39:      mov    %eax,%ss
0x00007c39 in ?? ()
(gdb) si
=> 0x7c3b:      mov    $0x0,%ax
0x00007c3b in ?? ()
(gdb) si
=> 0x7c3f:      mov    %eax,%fs
0x00007c3f in ?? ()
(gdb) si
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb) si
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb) si
=> 0x7c48:      call   0x7d49
0x00007c48 in ?? ()
(gdb) si
=> 0x7d49:      endbr32
```

Incorrect output at junk value 0x7c42



```
(gdb)
[   0:7c2e] => 0x7c2e:  ljmp   $0xb866,$0x87c75
0x00007c2e in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb)
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb)
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
(gdb)
[f000:d0b2]    0xfd0b2: mov    $0xdb80,%ax
0x0000d0b2 in ?? ()
(gdb)
[f000:d0b8]    0xfd0b8: mov    %eax,%ds
0x0000d0b8 in ?? ()
(gdb)
[f000:d0ba]    0xfd0ba: mov    %eax,%ss
0x0000d0ba in ?? ()
(gdb)
[f000:d0bc]    0xfd0bc: mov    $0xf898,%sp
0x0000d0bc in ?? ()
(gdb)
[f000:d0c2]    0xfd0c2: jmp    0x5476ca07
0x0000d0c2 in ?? ()
(gdb)
[f000:ca05]    0xfca05: push   %si
0x0000ca05 in ?? ()
(gdb)
[f000:ca07]    0xfca07: push   %bx
0x0000ca07 in ?? ()
```

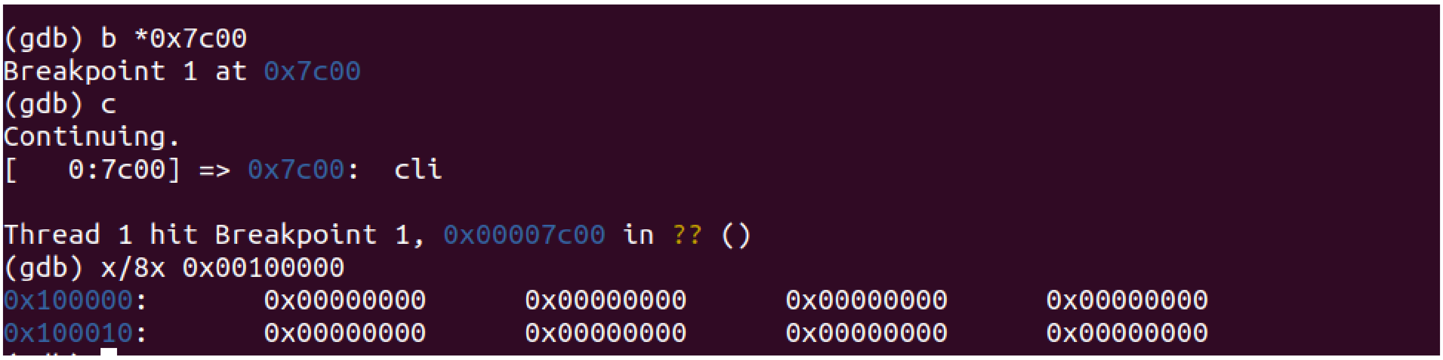
Running objdump -f kernel we find the address of the entry point : 0x0010000c



```
aditya@adityapc:~/oslab/xv6-public$  objdump -f kernel

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

Exercise 6 : Examining memory

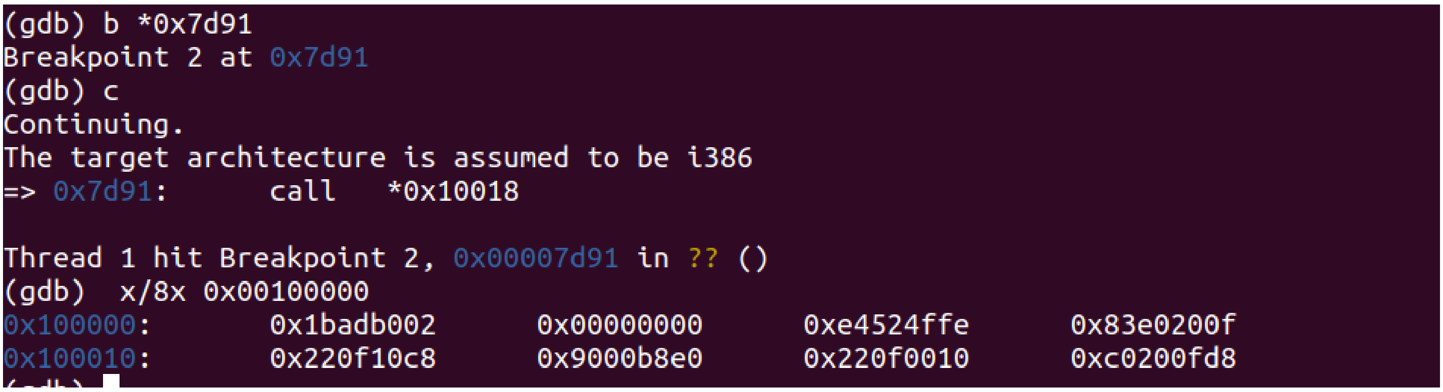
Now using gdb we set a breakpoint at 0x7c00, then using x/Nx ADDR command, we can print N words at address location ADDR.
Output of x/8x 0x00100000



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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

Here, all the words are 0s because there is no data loaded yet, because the bootloader has not started running yet. The breakpoint here is at the beginning of the boot loader.

Now setting the breakpoint at 0x7d91, we get :



```
(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)  x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
```

Now all the kernel has been fully loaded and hence we see that the words are not zeros now.
The **importance of breakpoint** is that it marks the end of the boot loader and hence everything is loaded correctly.

References

1. https://en.cppreference.com/w/c/language/asm
2. https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C