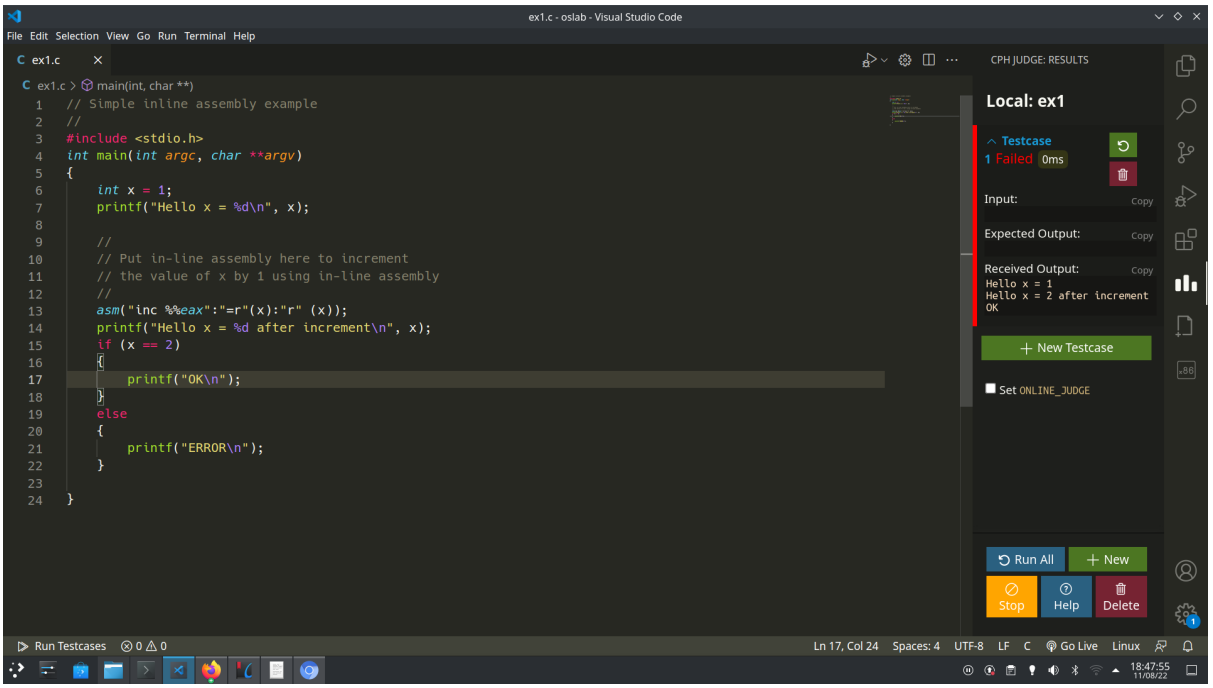
Assignment 0A

OS344 - Operating Systems Laboratory

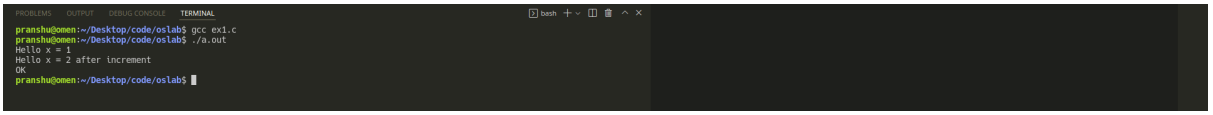
Pranshu Pandya

Assignment 0A:-

Exercise 1: Becoming familiar with inline assembly by writing a simple program



The above image shows the code with the output as below.

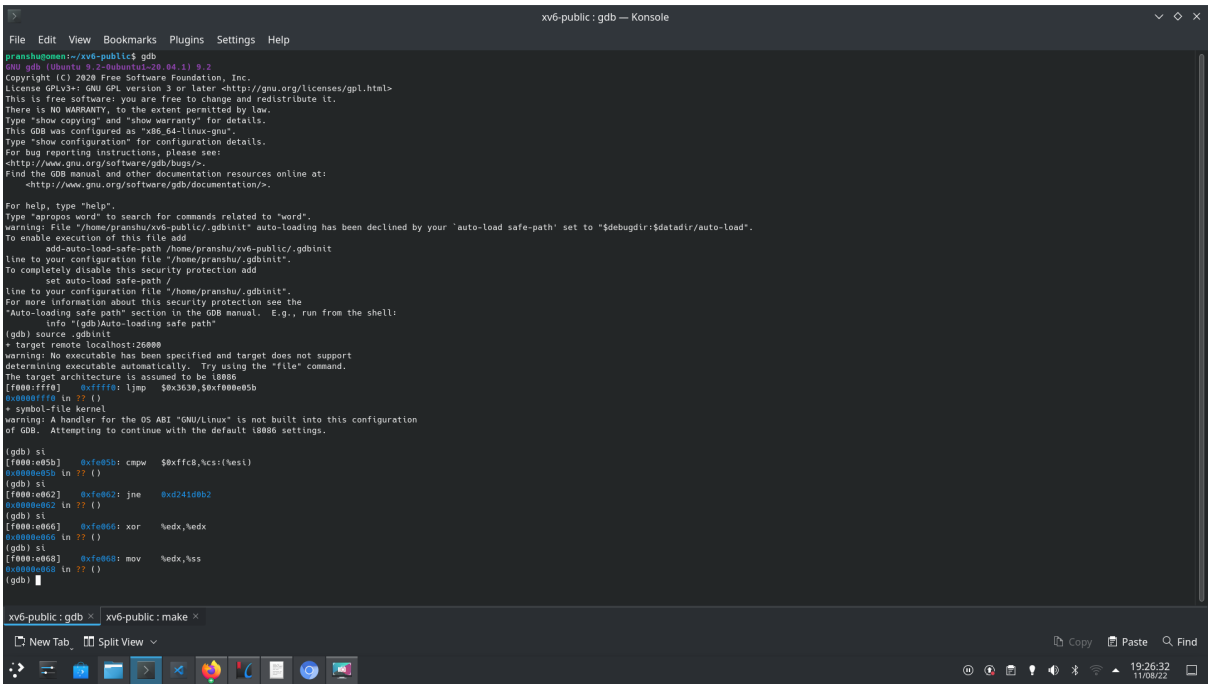


The code I have added:

```
asm("inc %%eax":"=r"(x):"r" (x));
```

In this x is both input as well as output operand. The code increases the value of x by 1.

Exercise 2: Tracing into the ROM BIOS



The si instruction in gdb is used to step through a single x86 instruction. Steps into calls. The above screenshot shows the first 5 instructions of the xv6 operating system. Following is the instruction and their explanation.

1. [fe00: fffe] 0xffff0: ljmp $0x3630, $0xf000e05b
   Here,
   The Starting Code Segment: f000
   The Starting Instruction Pointer: fff0
   The physical address of the instruction: 0xffff0
   The destination code segment: 0x3630
   The destination instructor pointer: 0xf000e05b
   While ljmp is the instruction.

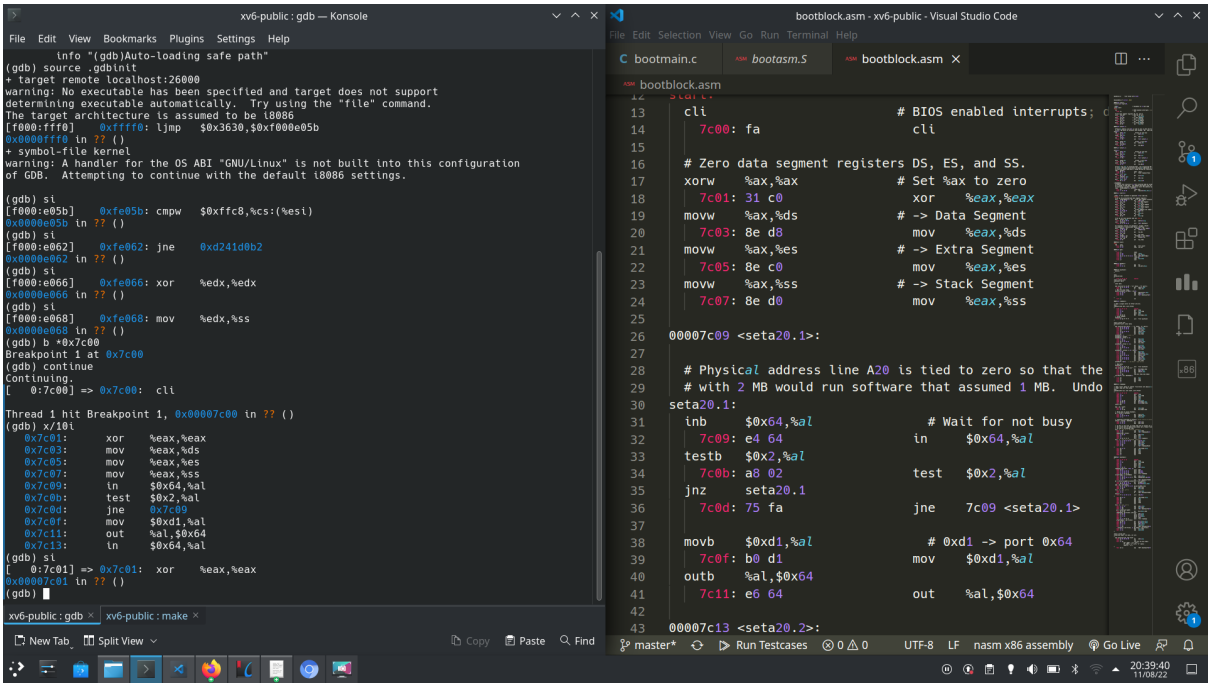
2. The cmp instruction is used to perform comparison. It sets the Zero flag to one if both the operands are equal.

3. The jnz (or jne) instruction is a conditional jump that follows a test. It jumps to the specified location if the Zero Flag (ZF) is unset.

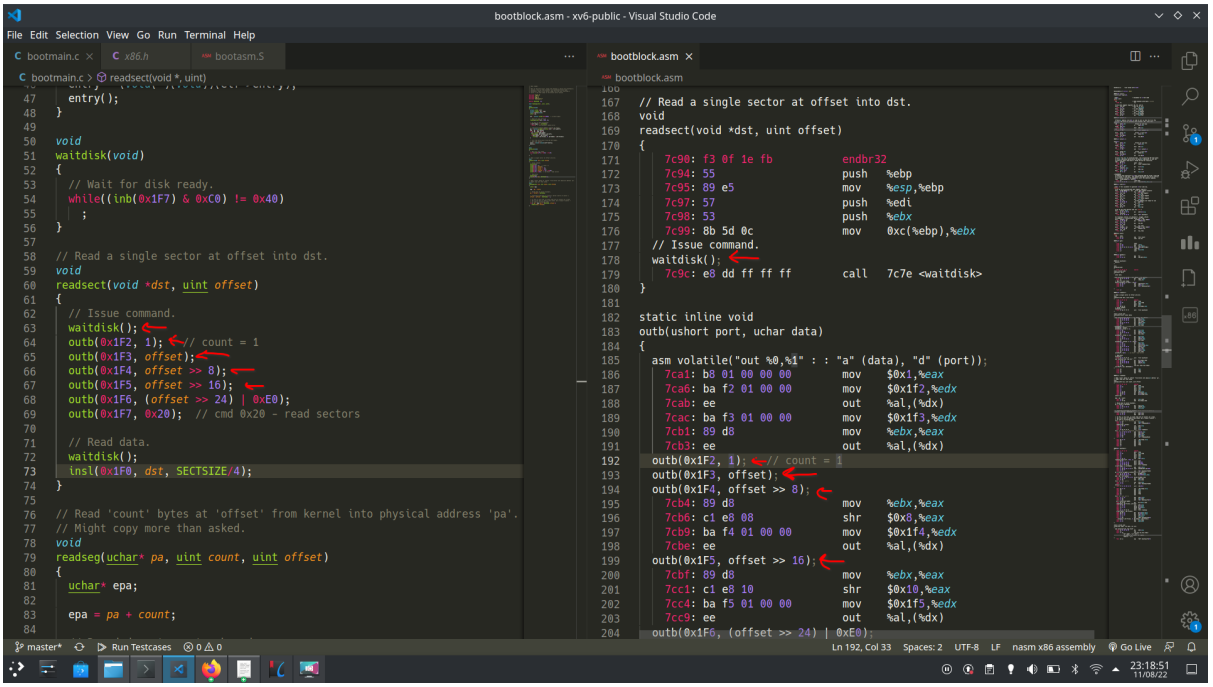
4. The xor instruction performs a logical XOR (exclusive OR) operation. This is the equivalent to the "^" operator in C++.

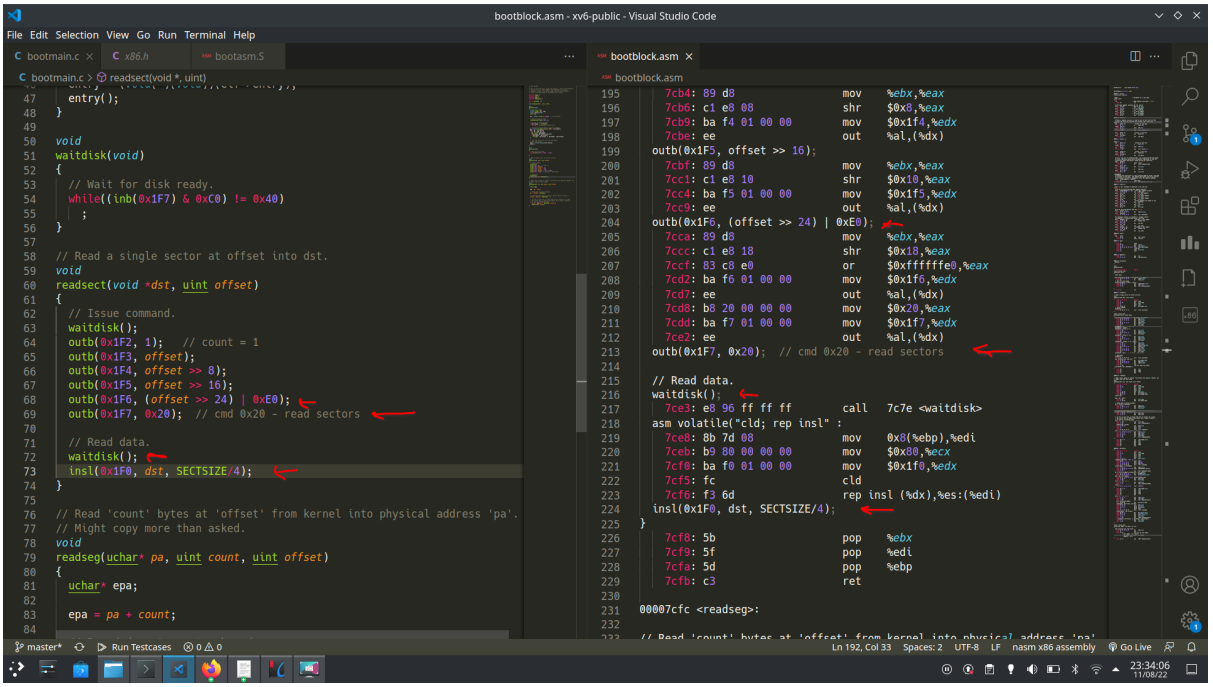
5. The mov instruction is used to move the value of edx to ss.

Part 2: The Boot Loader

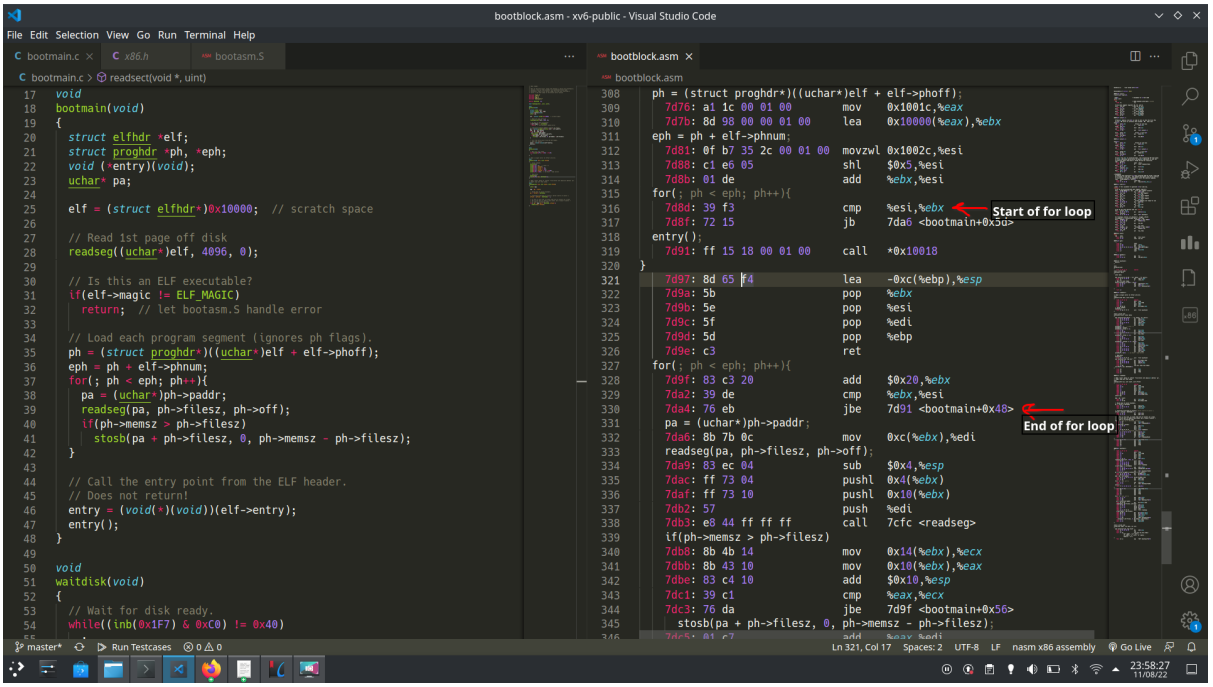


Here I have set breakpoint at address 0x7c00 using b *0x7c00. Then I have used the continue statement to continue execution till the next breakpoint which is 0x7c00. In the image you can see the right hand side window where I have shown the bootblock.asm where you can see the instruction at 7c00. I have also examined the next 10 instructions using the x/10i command.





In the above image I have traced through the readsect function in bootmain.c and pointed out the exact assembly instructions corresponding to each of the statements.



Tracing through the readsect and denoting the beginning and end of the for loop. It is marked in the image on the right side. The for loop is starting at address 0x7d8d and ending at address 0x7da4.

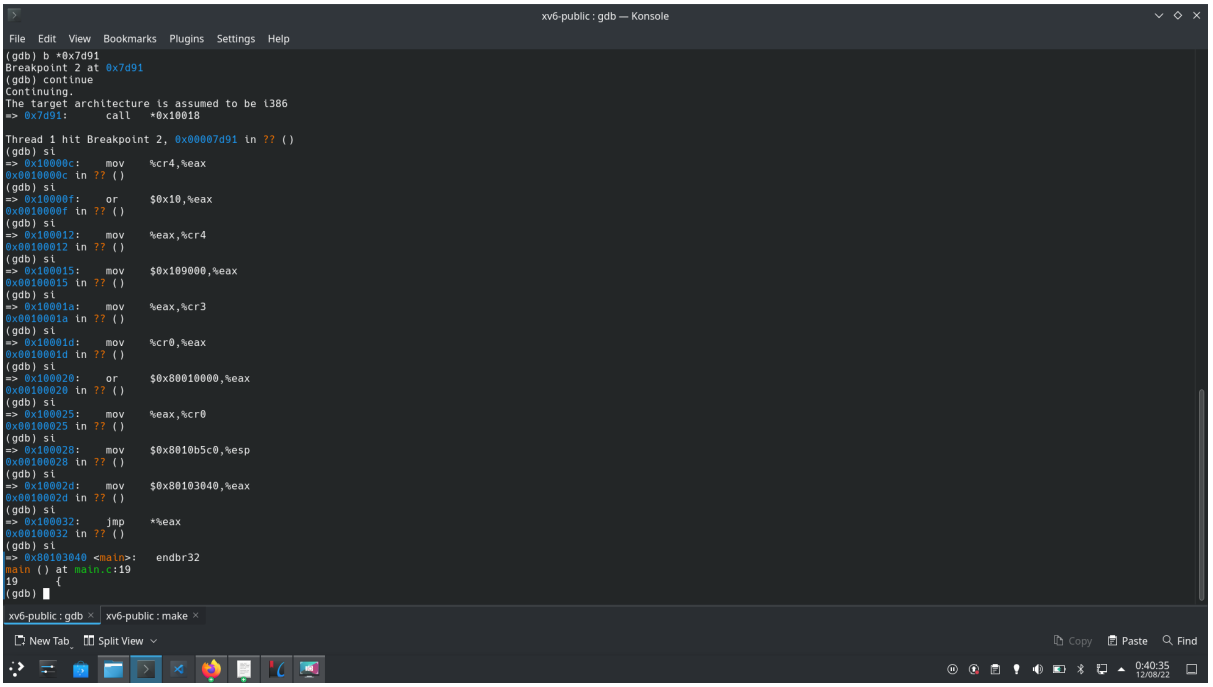
The first instruction of the for loop:

```
  7d8d: 39 f3                   cmp    %esi,%ebx
```

The Last instruction executed by the for loop:

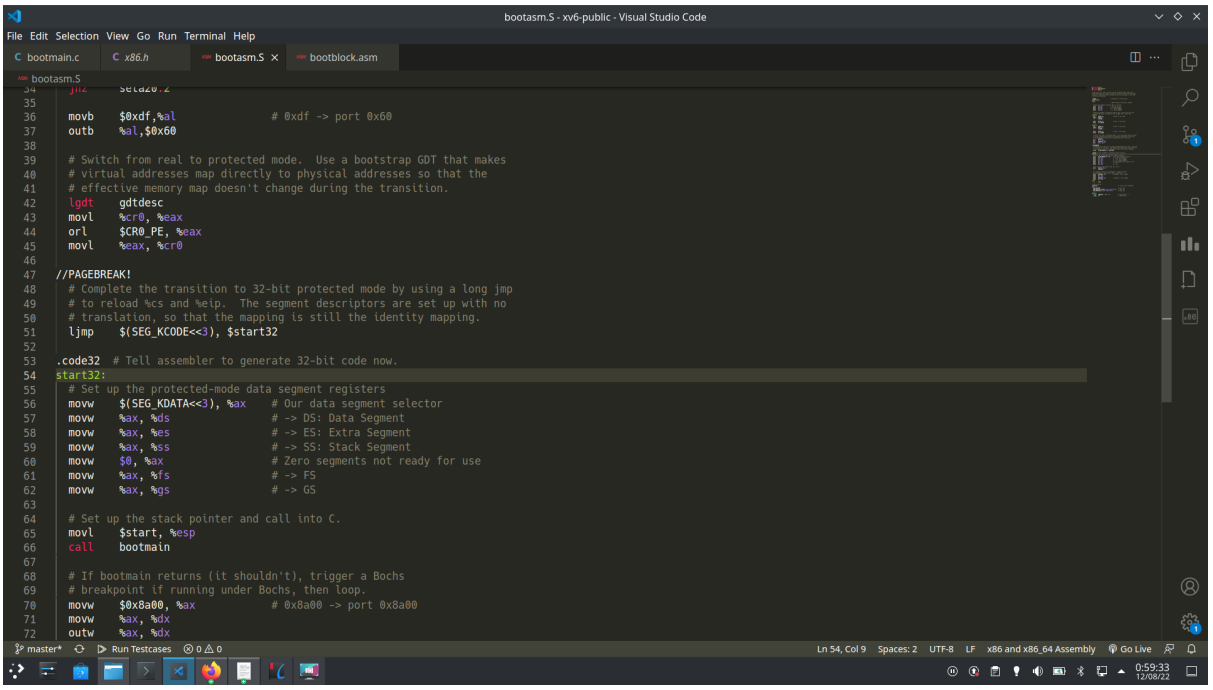
```
  7da4: 76 eb                   jbe    7d91 <bootmain+0x48>
```

The first instruction is a comparison between the values of ph and eph and that is exactly what we should do while entering the loop as the loop will run only when ph < eph.
In the last instruction, we know that the loop ends when ph and eph attain equal values and thus the loop jumps to the next instruction at 0x7d91.



In the above image I have set breakpoint at the last instruction of the for loop and then stepped into further instructions.

Exercise 3)



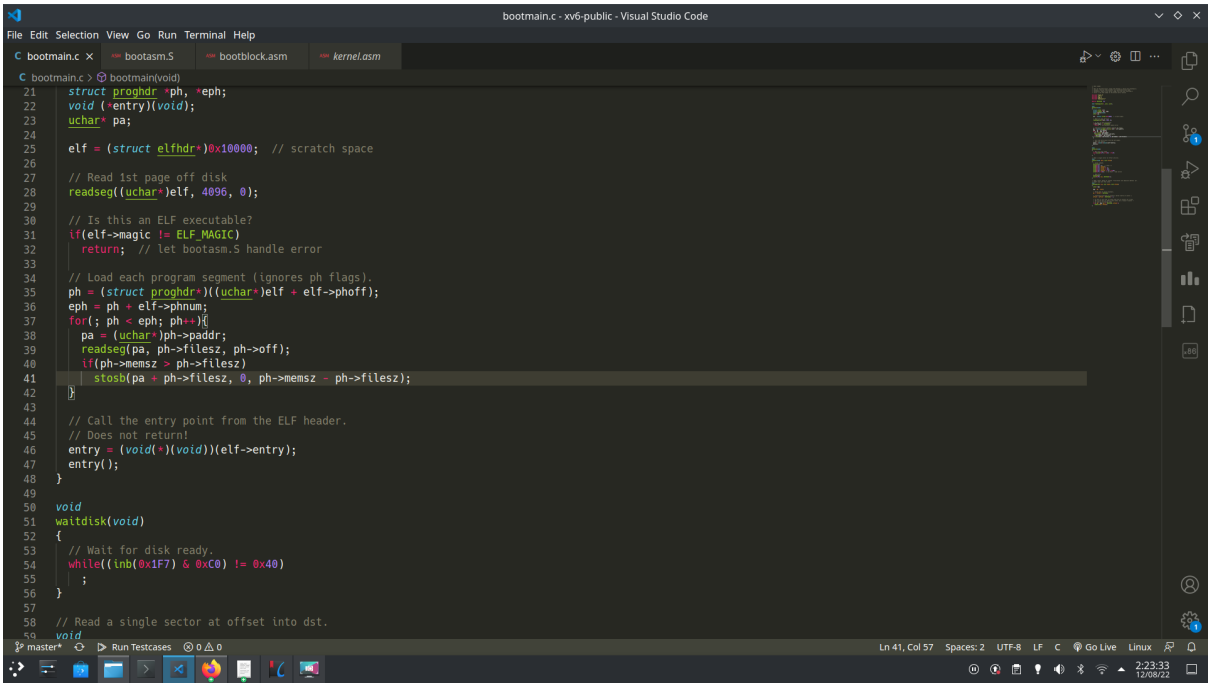
3.1) In the bootasm.S, after analysis and reading the comments we can see that “movw $(SEG_KDATA<<3), %ax ” is the first instruction to be executed in 32-bit mode. The instruction “ ljmp $(SEG_KCODE<<3), $start32 ” causes the transition to 32-bit mode.
3.2) As it is clear from the bootmain function of bootmain.c the last line of the bootloader is call to the entry function, so the last instruction to be executed is the following:

```
7d91: ff 15 18 00 01 00      call   *0x10018
```

The instruction is shifting the control to the address which is stored at 0x10018 since * operator is used for dereferencing. Now the first instruction can be found by looking into the kernel.asm file or we can see the contents of the memory location 0x10018 by using the x/1x 0x10018 command. We will get 0x0010000c as the address stored at memory location 0x10018. using x/1i 0c0010000c , we can see the instruction stored at that address.
So the first instruction of kernel is:

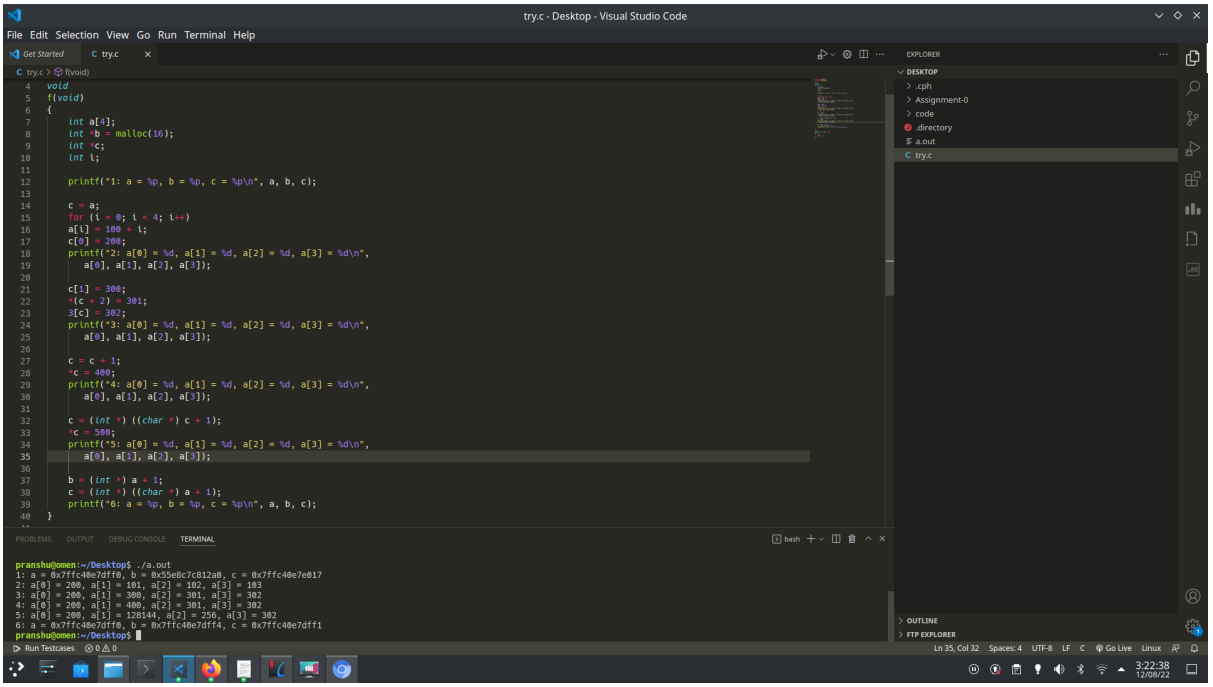
```
0x10000c: 0f 20 e0                mov    %cr4,%eax
```

3.3) The bootloader finds the information about how many sectors it must read in the information stored in the ELF header.



As we can see in the above image the code from line 35 to 42 is used to load the kernel. ELF stands for Executable and Linkable format and ELF headers are loaded by xv6 into a memory location which is pointed by an “elf” pointer. The starting address of the first segment of the kernel to be loaded in “ph”- program header by adding an offset (“elf->phoff”) to the starting address(elf). End pointer eph is also maintained which points to the memory location after the end of the last segment. Then in the for loop it iterates over all the segments. For each segment, the segment which has to be loaded is pointed by pa. Then the current segment is loaded at the location by passing pa, ph->filesz and ph->off parameters to readseg. It then checks if the file size is less than the memory which has to be read; if so then it will append zeros at the end of the extra memory. So the bootloader will keep loading segments while the condition “ph < eph” is true. The value of ph and eph are determined using attributes phoff and phnum of the ELF header.

Exercise 4)



In line 1: the address of pointers a, b , c is printed.

In the line 2: as c is equal to a and we are modifying value of c afterwards the a[0] = c[0] = 200 and a[1]=101, a[2]=102, a[3]=103 as filled by the loop

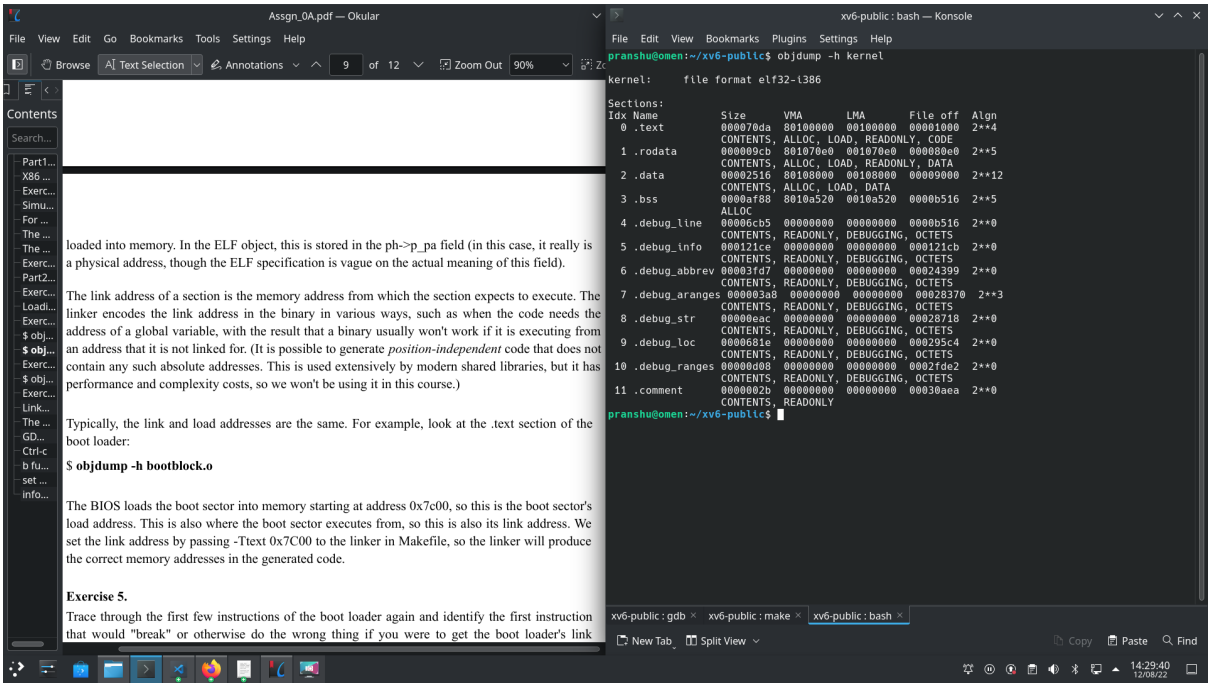
In the line 3: c[1] = a[1] = 300 , *(c+2)=c[2]=a[2]=301, 3[c]=c[3]=a[3]= 302

In the line 4: we have incremented c to c+1 so *c = c[1]=a[1]=400 which is reflected in the output

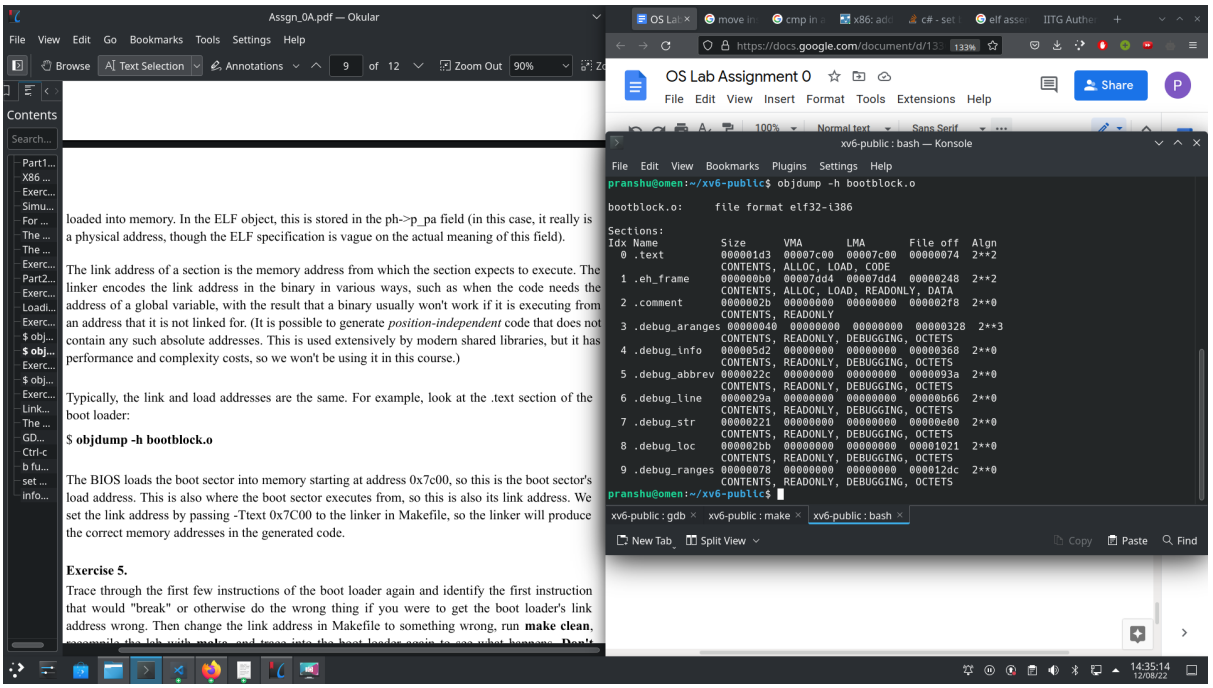
In the line 5: as we are typecasting c as character before adding 1 to it according to pointer arithmetic its address will be incremented by 1 and then again typecast as integer. So now c is pointing at the 2nd byte of 1th index of the array so it will change the last 3 byte of c[1] and first byte of c[2] thus the value of a[1] and a[2] is changed.

In the line 6: as for b we have incremented the pointer of integer pointer type by 1 the value of b will be addressof(a) + 1*sizeof(int). And for c as a is typecasted to character pointer before adding 1 value of c will be addressof(a) + 1*sizeof(char) and which are exactly the case as you can see in the image.

Displaying a full list of the names, sizes, and link addresses of all the sections in the kernel using objdump command



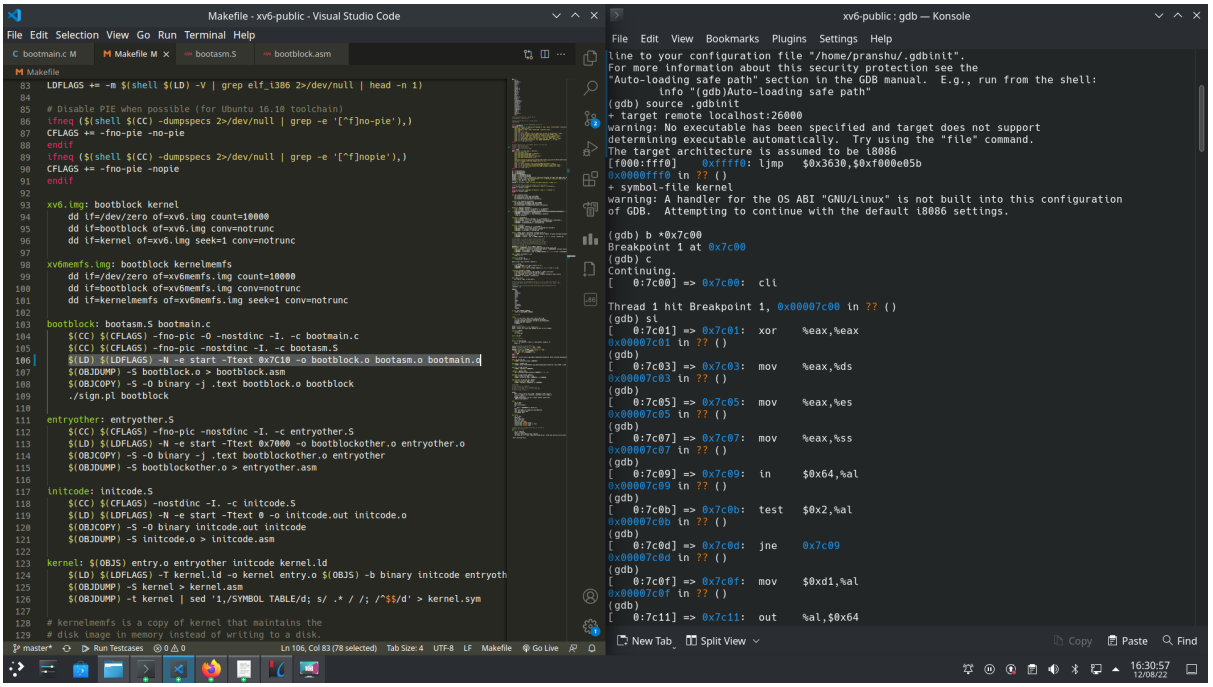
We can see the result of the command in the above image. Note that the VMA and LMA of .text of the kernel are different so it is loaded and executed from different addresses.



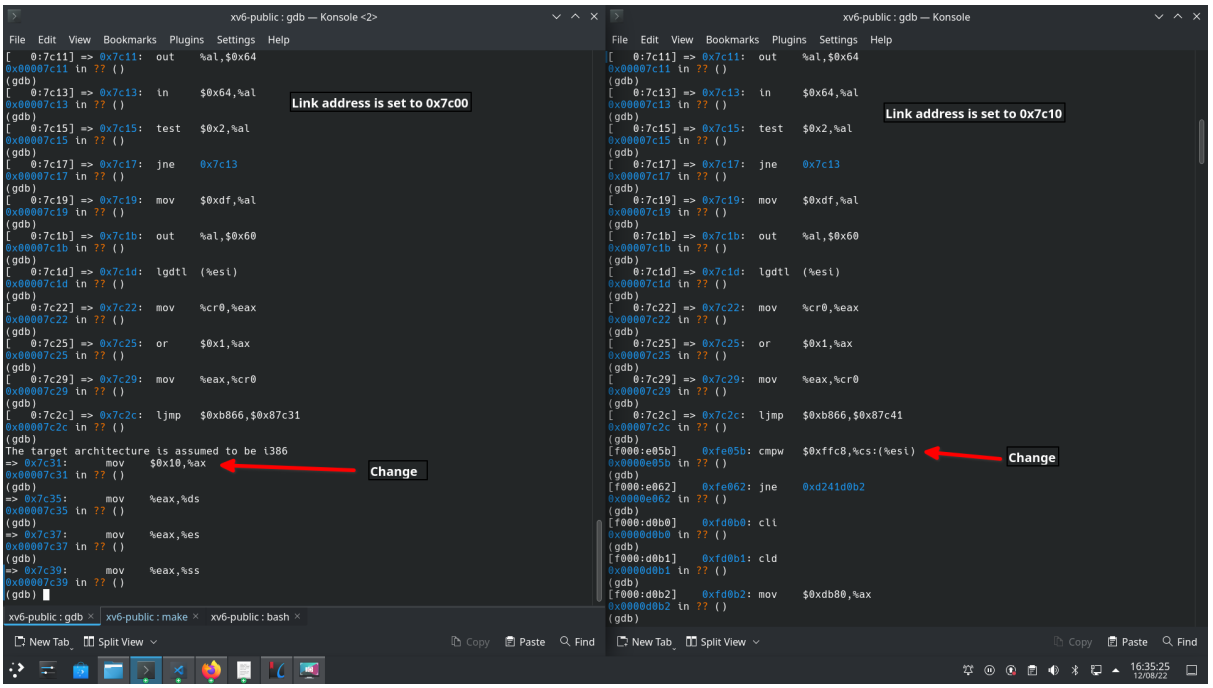
As we can see for bootblock.o the VMA and LMA are the same showing that the link and the load addresses are the same.

Exercise 5)

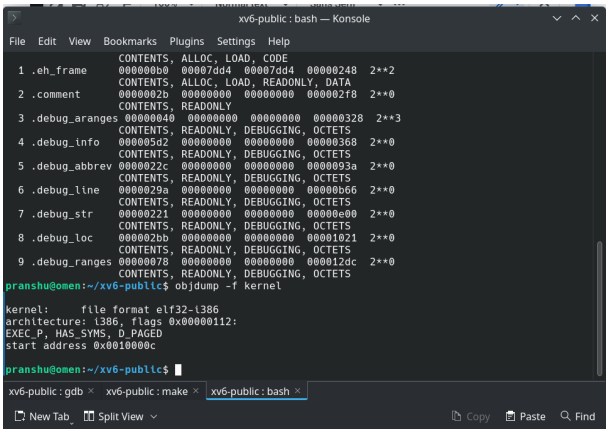
Here I have changed the bootloader's link address in Makefile from 0x7c00 to 0x7c10 and then after running make clean and make again. I started debugging it step by step.



One observation we can see is that changing the bootloader's link address will not affect BIOS and thus it will run smoothly and the control will be given to the boot loader. So I placed a breakpoint at the bootloader's previous link address that is 0x7c00 as the change is made after it so till that point both versions are guaranteed to behave identical. Now I run step into command for around 150 steps and start observing for change in both versions by keeping them side by side. The first change which I found is shown in the image below.



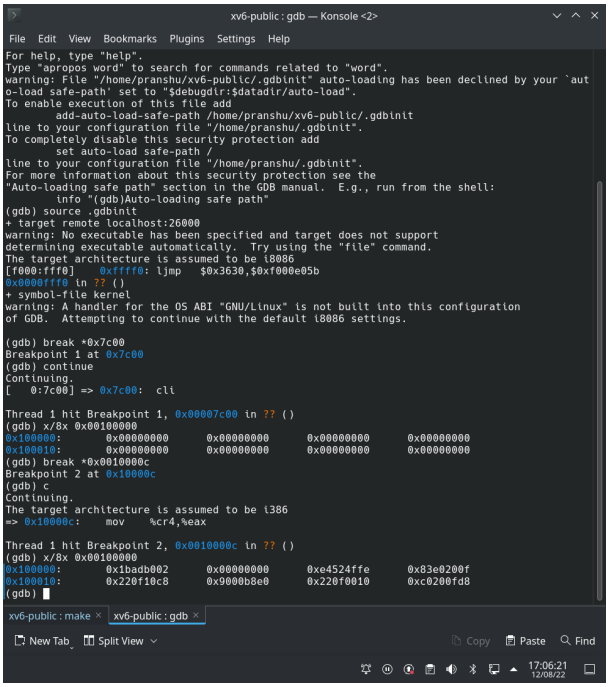
The left side is where the link address is correctly set to be 0x7c00 and the right side is where the link address is changed to be 0x7c10.



I have also tried the objdump command to see the entry point as shown in the above image.

Exercise 6)

In the below image I have restarted QEMU and GDB and first set the breakpoint at 0x7c00 where the bootloader gets the control. I continued execution till the breakpoint. Then I used the x/8x 0x00100000 command to examine the 8 words of memory at 0x00100000. Then I set the breakpoint at 0x0010000c which is the point where the bootloader enters the kernel. Then I continue the execution till the breakpoint. Then I again examine the 8 words of memory at 0x00100000.



As it can be seen in the above image at first the value at the 0x0010000 is changed after the bootloader entered the kernel. The reason behind this is that the address 0x00100000 is actually 1MB which is the address from where the kernel is loaded into the memory. Before the kernel was loaded into the memory, this address didn't have any data, i.e. garbage values. By default, all the uninitialized values are set to 0 in xv6. Hence, at the first breakpoint we got all zeroes as data hasn't been loaded at that time.
But when we check at the second breakpoint, the kernel has been loaded into the memory and thus this address now contains meaningful data instead of zeroes.