ASSIGNMENT 0A
CS344 - OS LAB

Name:- Nayanika Ghosh
Roll:- 200123036

Exercise 1 :-

Before adding the code :-

```
//
#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
// the value of x by 1 using in-line assembly
//

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

```
Hello x = 1
Hello x = 1 after increment
ERROR
```

After adding the code :-

```c
#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
// the value of x by 1 using in-line assembly
//

  __asm__ ( "addl %%ebx, %%eax;"
          : "=a" (x)
          : "a" (x), "b" (1) );

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

```
Hello x = 1
Hello x = 2 after increment
OK
```

Here, x and 1 are the input operands. The output operand is x. The code which I added later is used to add the value of x and 1 and to save the output to x. So, the value gets increased by 1.

Exercise 2 :-

```
0x0000fff0 in ?? ()
+ symbol-file kernel
warning: A handler for the OS ABI "GNU/Linux" is no
of GDB.  Attempting to continue with the default i8

(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb) si
[f000:e066]    0xfe066: xor    %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068]    0xfe068: mov    %edx,%ss
0x0000e068 in ?? ()
(gdb)
```

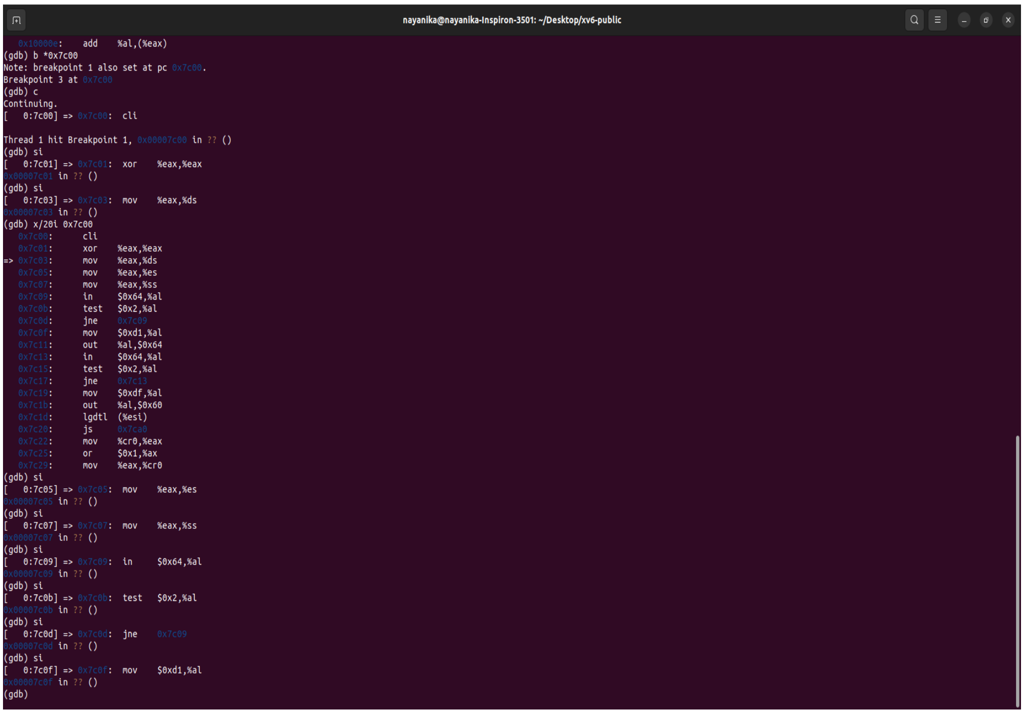
The ‘si’ instruction in gdb executes a machine instruction. On running the command ‘si’ , the first 4 instructions that appear have been shown in the image above. The observation here has been that as the BIOS runs, it immediately starts an interrupt descriptor table and initializes some devices. The "SeaBIOS" messages in the QEMU window come from this itself.

Exercise 3 :-

```

//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
    7c2c:       ea                          .byte 0xea
    7c2d:       31 7c 08 00                 xor    %edi,0x0(%eax,%ecx,1)

00007c31 <start32>:

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
    7c31:       66 b8 10 00                 mov    $0x10,%ax
  movw    %ax, %ds                # -> DS: Data Segment
    7c35:       8e d8                       mov    %eax,%ds
  movw    %ax, %es                # -> ES: Extra Segment
    7c37:       8e c0                       mov    %eax,%es
  movw    %ax, %ss                # -> SS: Stack Segment
    7c39:       8e d0                       mov    %eax,%ss
  movw    $0, %ax                 # Zero segments not ready for use
    7c3b:       66 b8 00 00                 mov    $0x0,%ax
  movw    %ax, %fs                # -> FS
    7c3f:       8e e0                       mov    %eax,%fs
```

a) From bootasm.S , the processor starts excuting the 32-bit code at - “movw $(SEG_KDATA<<3), %ax ”.
The instruction line - “ljmp $(SEG_KCODE<<3), $start32 ” marks the transition from from 16-bit to 32-bit mode.

b)



The last instruction of the bootloader is entry().

The first instruction of the kernel :-
“ 0x10000c: mov %cr4,%eax “

c)

```
  // 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);
  }

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

void
waitdisk(void)
{
```

Exercise 4 :-

Output of the “pointers.c” program :-

```c
        a[0], a[1], a[2], a[3]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
        a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",
        a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
```

```
1: a = 0x7fff76501610, b = 0x559a5fe1c2a0, c = 0x7fff76501637
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fff76501610, b = 0x7fff76501614, c = 0x7fff76501611
```

In “objdump -h kernel “ , the VMA and the LMA of “.text section” are different. So, it loads and executes from two different addresses.
I have attached the screenshot below.

```
nayanika@nayanika-Inspiron-3501:~/Desktop/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00007188  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009cb  801071a0  001071a0  000081a0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000afb0  8010a520  0010a520  0000b516  2**5
                  ALLOC
  4 .debug_line   00006aaf  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   00010e14  00000000  00000000  00011fc5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 00004496  00000000  00000000  00022dd9  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 000003b0  00000000  00000000  00027270  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_str    00000df9  00000000  00000000  00027620  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 000050b1  00000000  00000000  00028419  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002d4ca  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 0000013c  00000000  00000000  0002dd0f  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      00000026  00000000  00000000  0002de4b  2**0
                  CONTENTS, READONLY
```

In “objdump -h bootblock.o “ , the VMA and the LMA of “.text section” are the same. So, it loads and executes from the same address.
I have attached the screenshot below.

```
nayanika@nayanika-Inspiron-3501:~/Desktop/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c3  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000b0  00007dc4  00007dc4  00000238  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000026  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000310  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000350  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008d5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b11  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000210  00000000  00000000  00000d94  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 0000004b  00000000  00000000  00000fa4  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fef  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  0000117c  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Exercise 5 :-

When the boot loader’s link address is kept as 0x7C00, the commands run properly and transition from 16-bit to 32-bit occurs at 0x7C31 address. But when we change the boot loader’s link address to any other address (0x7E00 in my case) and then run “make clean” , “make” commands and then restart the gdb ,continuing from 0x7C00, the boot loader is restarting repeatedly after running some instructions in the gdb.

```
(gdb) b *0x7C00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

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

Exercise 6:-

We check the 8 words of the memory at 0x00100000 at twice,once when the BIOS enters the boot loader and the second when
the boot loader enters the kernel. Here, the command “x/8x 0x00100000” is taken after setting our breakpoints. The first breakpoint will be at 0x7c00 because this is the point where the BIOS hands control over to the boot loader. The second breakpoint will be at 0x0010000c because this is the point when the kernel is passed control by the bootloader. Different values are got at the breakpoints.

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) continue
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) continue
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) 
```