CS344:Assignment-0

Name-Mesharya M Choudhary
Roll No.-190101053

Ans-1 The modification made in ex1.c file is:

__asm__("incl %0":"+r"(x));

+r allocates a free register to variable x and is used for both input and output ,incl is responsible for increasing the value of the operand by 1 and %0 is the register which is assigned to x.

Ans-2

```
determining executable automatically.  Try using the "file" command.
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 built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(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 ?? ()
```

The first instruction is a jump instruction.
The second instruction is a comparison of two operands at the specified addresses.
The third instruction is a conditional jump instruction based on the result of the previous instruction.
The fourth instruction is the xor of two operands, the value of edx is set to 0
The fifth instruction is a mov instruction which moves the value of edx to ss(stack segment)
The sixth instruction is a mov instruction which moves the value value 0x7000 to sp register.
The seventh instruction is a mov instruction which moves the value value 0x7c4 to dx register.
The eighth instruction is a jump instruction to the address stored at given memory location
The ninth instruction is clear interrupt flag
The tenth instruction is clear direction flag
The eleventh instruction is a mov instruction which moves the value of ax register to cx
The twelfth instruction is a move instruction which loads 0x8f to the ax register

Ans-3

```
.code16                        # Assemble for 16-bit mode
.globl start
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>:
```

Disassembly file bootblock.asm

```
.code16                        # Assemble for 16-bit mode
.globl start
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

  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
```

Source code bootasm.s

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/20i $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
   0x7c1d:      lgdtl  (%esi)
   0x7c20:      js     0x7c9e
   0x7c22:      mov    %cr0,%eax
   0x7c25:      or     $0x1,%ax
   0x7c29:      mov    %eax,%cr0
(gdb) 
```

Disassembling first 20 instructions in GDB from the breakpoint at 0x7c00 onwards

We can clearly notice that the instructions are identical in these three images.

```c

// Read a single sector at offset into dst.
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);
}
```

The readsect() function in bootmain.c

```

00007c90 <readsect>:

// Read a single sector at offset into dst.
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>
}

```

Corresponding disassembled code for readsect() in bootblock.asm

```
307     7d74:       75 21                   jne    7d97 <bootmain+0x4e>
308   ph = (struct proghdr*)((uchar*)elf + elf->phoff);
309     7d76:       a1 1c 00 01 00          mov    0x1001c,%eax
310     7d7b:       8d 98 00 00 01 00       lea    0x10000(%eax),%ebx
311   eph = ph + elf->phnum;
312     7d81:       0f b7 35 2c 00 01 00    movzwl 0x1002c,%esi
313     7d88:       c1 e6 05                shl    $0x5,%esi
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 }
```

The instructions from line 327 to 348 are responsible for reading the remaining sectors of the kernel from the disk.Once this loop is terminates the instruction at line 330 is executed and is evaluated to true and the control jumps to the instruction in line 319 corresponding to 0x7d91: **call *0x10018** which is then executed and it is also the last instruction of the bootloader

a)

```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  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

.code32  # Tell assembler to generate 32-bit code now.
start32:
  # Set up the protected-mode data segment registers
```

```
//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
```

The instruction ljmp $(SEG_KCODE<<3), $start32 is responsible for switching the processor from 16bit mode to 32 bit mode.The first instruction that is executed in the 32 bit mode is
7c31: mov $0x10,%ax
b)The last instruction of the bootloader executed is 0x7d91: call *0x10018
which corresponds to this in the bootmain.c file

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

First instruction of the kernel is 0x10000c: mov %cr4,%eax

```
(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)
```

c)The information regarding how many sectors to read to fetch the entire kernel from the disk is present in the ELF header.

```
// 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);
}
```

The bootloader runs a loop from ph and eph both of which are determined using the ELF header to load the kernel.The number of iterations of the loop is decided by elf->phnum.

Ans-4

```
ubuntu@ubuntu-VirtualBox:~/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    00000eab  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c3  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030ae9  2**0
                  CONTENTS, READONLY
```

```
ubuntu@ubuntu-VirtualBox:~/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    00000220  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  00001020  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012db  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

We use objdump -h to display information related to the program section headers in the ELF binaries
The important sections are as follows:
(i).test-the executable instructions corresponding to the program
(ii).rodata-the read only data of the program
(iii).data-initialized static and global variables of the program
(iv).bss-uninitialized static and global variables of the program.
Each section has the following information,VMA is the link address of the section,LMA is the load address of the section,Size is the size of the section,Algn is the value to which the section is aligned in the memory and the file,Offset is the offset from the beginning of the hard drive at which the section is located.Load address is the address where the section should be loaded and Link address is the address from where the section begins to execute.

Ans-5

The first instruction that will break if the provided link address is wrong is in Line 51

```
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:
55   # Set up the protected-mode data segment registers
```

When correct link address which is 0x7c00 is provided we get the following.

```
(gdb) b* 0x7c2c
Breakpoint 1 at 0x7c2c
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31

Thread 1 hit Breakpoint 1, 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
0x00007d49 in ?? ()
(gdb)
```

When wrong link address 0x7d00 is provided

```
(gdb) b* 0x7c2c
Breakpoint 1 at 0x7c2c
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87d31

Thread 1 hit Breakpoint 1, 0x00007c2c in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb) si
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
(gdb) si
[f000:d0b2]    0xfd0b2: mov    $0xdb80,%ax
0x0000d0b2 in ?? ()
(gdb) si
[f000:d0b8]    0xfd0b8: mov    %eax,%ds
0x0000d0b8 in ?? ()
(gdb) si
[f000:d0ba]    0xfd0ba: mov    %eax,%ss
0x0000d0ba in ?? ()
(gdb) si
[f000:d0bc]    0xfd0bc: mov    $0xf898,%sp
0x0000d0bc in ?? ()
(gdb) si
[f000:d0c2]    0xfd0c2: jmp    0x5476ca07
0x0000d0c2 in ?? ()
(gdb) si
[f000:ca05]    0xfca05: push   %si
0x0000ca05 in ?? ()
(gdb) si
[f000:ca07]    0xfca07: push   %bx
0x0000ca07 in ?? ()
(gdb) si
[f000:ca09]    0xfca09: push   %dx
0x0000ca09 in ?? ()
(gdb)
```

As we can see in both the versions the instructions that follow **ljmp $0xb866,$0x87d31** differ because this instruction is executed wrongly in the wrong version which causes rest of the instructions to differ.
Entry point address of the kernel is 0x0010000c

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

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

Ans-6

```
--Type <RET> for more, q to quit, c to continue without paging--c
        info "(gdb)Auto-loading safe path"
(gdb) source .gdbinit
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
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 built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

(gdb) b* 0x7c00
Breakpoint 1 at 0x7c00
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b* 0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    mov    %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) 
```

The code for kernel is stored at memory location 0x00100000, which is loaded by the bootloader to the disk.When the BIOS enters the bootloader, the kernel is yet to be loaded thus memory location is filled with zeroes from this point onwards.By the time bootloader enters the kernel, the kernel has been loaded and the memory location contains it’s instructions.