CS - 344 (OS LAB)

Group 1

Members :

1. Parth Agarwal, CSE, 220101074
2. Shashwat Shankar, CSE, 220101092
3. Shobhit Gupta, CSE, 220101093
4. Shubhranshu Pandey, CSE, 220101094

Assignment - 0A-1

Exercise 1 :

Modified Program ex1.c which now includes inline assembly that increments the value of x by 1.

```c
// Simple inline assembly example
//
#include <stdio.h>
int main(int argc, char **argv)
{
    int x = 1;
    printf("Hello x = %d\n", x);
    // Inline assembly to increment the value of x by 1
    __asm__ (
        "incl %0"
        : "=r" (x)
        : "0" (x)
    );
    printf("Hello x = %d after increment\n", x);
    if(x == 2){
        printf("OK\n");
    }
    else{
        printf("ERROR\n");
    }
}
```

Output of the modified code:

```
shashwat@Shashwat:~/xv6-public$ gcc ex1.c
shashwat@Shashwat:~/xv6-public$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
shashwat@Shashwat:~/xv6-public$ |
```

Added Code:

```
// Inline assembly to increment the value of x by 1
__asm__ (
    "incl %0"
    : "=r" (x)
    : "0" (x)
);
```

The incl %0 instruction operates on the register holding x, directly modifying its value. The operands specify that x is both the input and output, ensuring the incremented value is stored back into x.

Exercise 2 :

Below is a picture of ROM BIOS traced using the si command -

```
The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0x3630,$0xf000e05b
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) si
[f000:e06a]    0xfe06a: mov    $0x7000,%sp
0x0000e06a in ?? ()
(gdb) si
```

0xffff0: ljmp $0x3630,$0xf000e05b - The processor begins execution at the (CS:IP) segment address : 0xf000:fff0. This references the physical address 0xffff0. This address is 16 bytes before the end of BIOS(0x100000). Since BIOS would not be able to accomplish much in just 16 bytes, it jumps backwards to an earlier location in the BIOS using ljmp instruction (0xf000:e05b).

0xfe05b: cmpw $0xffc8,%cs:(%esi) - Compares the word at the address pointed to by %esi in the code segment with the value 0xffc8. The w at the end of cmp means we're operating on 16-bit words. If the values compared are not equal then Zero Flag(ZF) is cleared(0).

0xfe062: jne 0xd241d0b0 - Jump if not equal instruction is used. If ZF is cleared then it would cause a jump to the address 0xd241d0b0.

It can be observed that no jump has happened, this means that the comparison has resulted in true. The above two cmpw and jne instructions together may comprise a check, maybe to ensure that everything is working as expected.

0xfe066: xor %edx,%edx - The **xor** instruction performs a logical XOR (exclusive OR) operation. This sets edx to zero, edx is a 32-bit general-purpose register.

0xfe068: mov %edx,%ss - Moves the value in %edx (which is now 0) into the stack segment register (%ss).

0xfe06a: mov $0x7000,%sp - Moves the value 0x7000 into the stack pointer (%sp). This makes the stack start at 0x00007000 since '%ss:%sp' now corresponds to 0x00007000.

Exercise 3 :

To turn on protected mode, first the zero bit of the %cr0 control register is set which can be seen below.

```
Thread 1 hit Breakpoint 1, 0x00007c1d in ?? ()
(gdb) x/5i 0x7c1d
=> 0x7c1d:      lgdtl  (%esi)
   0x7c20:      js     0x7c9e
   0x7c22:      mov    %cr0,%eax
   0x7c25:      or     $0x1,%ax
   0x7c29:      mov    %eax,%cr0
(gdb) |
```

lgdt gdtdesc is used to set GDT which would later be used to map 32 bit virtual addresses to physical ones.

```
# 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
  7c1d: 0f 01 16              lgdtl  (%esi)
  7c20: 78 7c                 js     7c9e <readsect+0x12>
movl    %cr0, %eax
  7c22: 0f 20 c0              mov    %cr0,%eax
orl     $CR0_PE, %eax
  7c25: 66 83 c8 01           or     $0x1,%ax
movl    %eax, %cr0
  7c29: 0f 22 c0              mov    %eax,%cr0
```

The last instruction executed which causes the switch from 16 bit to 32 bit code is highlighted with the arrow. It is shown below:

```
[   0:7c29] => 0x7c29:   mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   0:7c2c] => 0x7c2c:   ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb)
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
```

This corresponds to :

```
ljmp    $(SEG_KCODE<<3), $start32
```

A new value has to be loaded into a segment register to make the CPU read the GDT and change its segmentation settings. It is done by using a long jump instruction.

(SEG_KCODE<<3) is the segment selector for the code segment in the GDT.
The second operand $start32 is the offset of the 32-bit code to jump to (shown below).

```
.code32  # Tell assembler to generate 32-bit code now.
start32:
```

Tracing into Bootmain and Readsect:

```
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);
}
```

Readsect from bootmain.c

Call 0x7c8c - This calls the readsect function.
The following instructions, set up the function call by pushing necessary arguments and addresses in the stack.

```
The target architecture is set to "i386".
=> 0x7d20:      call   0x7c8c

Thread 1 hit Breakpoint 2, 0x00007d20 in ?? ()
(gdb) si
=> 0x7c8c:      push   %ebp
0x00007c8c in ?? ()
(gdb)
=> 0x7c8d:      mov    %esp,%ebp
0x00007c8d in ?? ()
(gdb)
=> 0x7c8f:      push   %edi
0x00007c8f in ?? ()
(gdb)
=> 0x7c90:      push   %ebx
0x00007c90 in ?? ()
(gdb)
=> 0x7c91:      mov    0xc(%ebp),%ebx
0x00007c91 in ?? ()
(gdb)
```

```
=> 0x7c94:      call   0x7c7e
0x00007c94 in ?? ()
(gdb)
=> 0x7c7e:      mov    $0x1f7,%edx
0x00007c7e in ?? ()
(gdb)
=> 0x7c83:      in     (%dx),%al
0x00007c83 in ?? ()
(gdb)
=> 0x7c84:      and    $0xffffffc0,%eax
0x00007c84 in ?? ()
(gdb)
=> 0x7c87:      cmp    $0x40,%al
0x00007c87 in ?? ()
(gdb)
=> 0x7c89:      jne    0x7c83
0x00007c89 in ?? ()
(gdb)
=> 0x7c8b:      ret
0x00007c8b in ?? ()
(gdb)
```

call 0x7c7e - Calls the first waitdisk() function.

The instructions between call and ret are the implementation of waitdisk().

ret - This is used to return from waitdisk().

```
=> 0x7c99:      mov    $0x1,%eax
0x00007c99 in ?? ()
(gdb)
=> 0x7c9e:      mov    $0x1f2,%edx
0x00007c9e in ?? ()
(gdb)
=> 0x7ca3:      out    %al,(%dx)
0x00007ca3 in ?? ()
(gdb)
```

Instructions correspond to : outb(0x1F2, 1);

```
=> 0x7ca4:      mov    $0x1f3,%edx
0x00007ca4 in ?? ()
(gdb)
=> 0x7ca9:      mov    %ebx,%eax
0x00007ca9 in ?? ()
(gdb)
=> 0x7cab:      out    %al,(%dx)
0x00007cab in ?? ()
(gdb)
```

Instructions corresponding to : outb(0x1F3, offset);

```
=> 0x7cac:      mov    %ebx,%eax
0x00007cac in ?? ()
(gdb)
=> 0x7cae:      shr    $0x8,%eax
0x00007cae in ?? ()
(gdb)
=> 0x7cb1:      mov    $0x1f4,%edx
0x00007cb1 in ?? ()
(gdb)
=> 0x7cb6:      out    %al,(%dx)
0x00007cb6 in ?? ()
(gdb)
```

Instructions corresponding to : outb(0x1F4, offset >> 8);

```
=> 0x7cb7:      mov    %ebx,%eax
0x00007cb7 in ?? ()
(gdb)
=> 0x7cb9:      shr    $0x10,%eax
0x00007cb9 in ?? ()
(gdb)
=> 0x7cbc:      mov    $0x1f5,%edx
0x00007cbc in ?? ()
(gdb)
=> 0x7cc1:      out    %al,(%dx)
0x00007cc1 in ?? ()
(gdb)
```

Instructions corresponding to : outb(0x1F5, offset >> 16);

```
=> 0x7cc2:      mov    %ebx,%eax
0x00007cc2 in ?? ()
(gdb)
=> 0x7cc4:      shr    $0x18,%eax
0x00007cc4 in ?? ()
(gdb)
=> 0x7cc7:      or     $0xffffffe0,%eax
0x00007cc7 in ?? ()
(gdb)
=> 0x7cca:      mov    $0x1f6,%edx
0x00007cca in ?? ()
(gdb)
=> 0x7ccf:      out    %al,(%dx)
0x00007ccf in ?? ()
(gdb)
```

Instructions corresponding to : outb(0x1F6, (offset >> 24) | 0xE0);

```
=> 0x7cd0:      mov    $0x20,%eax
0x00007cd0 in ?? ()
(gdb)
=> 0x7cd5:      mov    $0x1f7,%edx
0x00007cd5 in ?? ()
(gdb)
=> 0x7cda:      out    %al,(%dx)
0x00007cda in ?? ()
(gdb)
```

Instructions corresponding to : outb(0x1F7, 0x20);

```
=> 0x7cdb:      call   0x7c7e
0x00007cdb in ?? ()
(gdb)
=> 0x7c7e:      mov    $0x1f7,%edx
0x00007c7e in ?? ()
(gdb)
=> 0x7c83:      in     (%dx),%al
0x00007c83 in ?? ()
(gdb)
=> 0x7c84:      and    $0xffffffc0,%eax
0x00007c84 in ?? ()
(gdb)
=> 0x7c87:      cmp    $0x40,%al
0x00007c87 in ?? ()
(gdb)
=> 0x7c89:      jne    0x7c83
0x00007c89 in ?? ()
(gdb)
=> 0x7c8b:      ret
0x00007c8b in ?? ()
(gdb)
```

call 0x7c7e - Calls the first waitdisk() function.

The instructions between call and ret are the implementation of waitdisk().

ret - This is used to return from waitdisk().

```
=> 0x7ce0:      mov    0x8(%ebp),%edi
0x00007ce0 in ?? ()
(gdb)
=> 0x7ce3:      mov    $0x80,%ecx
0x00007ce3 in ?? ()
(gdb)
=> 0x7ce8:      mov    $0x1f0,%edx
0x00007ce8 in ?? ()
(gdb)
=> 0x7ced:      cld
0x00007ced in ?? ()
(gdb)
=> 0x7cee:      rep insl (%dx),%es:(%edi)
0x00007cee in ?? ()
(gdb)
=> 0x7cee:      rep insl (%dx),%es:(%edi)
0x00007cee in ?? ()
(gdb) b *0x7cf0
Breakpoint 2 at 0x7cf0
(gdb) c
Continuing.
=> 0x7cf0:      pop    %ebx

Thread 1 hit Breakpoint 2, 0x00007cf0 in ?? ()
(gdb) si
=> 0x7cf1:      pop    %edi
0x00007cf1 in ?? ()
(gdb)
=> 0x7cf2:      pop    %ebp
0x00007cf2 in ?? ()
(gdb)
=> 0x7cf3:      ret
0x00007cf3 in ?? ()
(gdb)
```

Instructions corresponding to : insl(0x1F0, dst, SECTSIZE/4);
The final **ret** here is used to return from the readsect() function.

```
The target architecture is set to "i386".
=> 0x7d57:      add    $0x10,%esp

Thread 1 hit Breakpoint 1, 0x00007d57 in ?? ()
(gdb) si
=> 0x7d5a:      cmpl   $0x464c457f,0x10000
0x00007d5a in ?? ()
(gdb)
=> 0x7d64:      jne    0x7d87
0x00007d64 in ?? ()
(gdb)
```

Instructions corresponding to :if(elf->magic != ELF_MAGIC)
    return;

```
=> 0x7d66:      mov    0x1001c,%eax
0x00007d66 in ?? ()
(gdb)
=> 0x7d6b:      lea    0x10000(%eax),%ebx
0x00007d6b in ?? ()
(gdb)
=> 0x7d71:      movzwl 0x1002c,%esi
0x00007d71 in ?? ()
(gdb)
=> 0x7d78:      shl    $0x5,%esi
0x00007d78 in ?? ()
(gdb)
=> 0x7d7b:      add    %ebx,%esi
0x00007d7b in ?? ()
```

The first 2 instructions are for setting up ph which will be the address from which boot-loader will start to load kernel in the for loop.

The last 3 instructions are for setting up eph which point to the end of the last segment of the kernel.

```
=> 0x7d7d:      cmp    %esi,%ebx
0x00007d7d in ?? ()
(gdb)
=> 0x7d7f:      jb     0x7d96
0x00007d7f in ?? ()
(gdb)
=> 0x7d96:      mov    0xc(%ebx),%edi
0x00007d96 in ?? ()
(gdb)
=> 0x7d99:      sub    $0x4,%esp
0x00007d99 in ?? ()
(gdb)
```

Start of the for loop

```
(gdb) si
=> 0x7db3:      jbe    0x7d8f
0x00007db3 in ?? ()
(gdb)
=> 0x7d8f:      add    $0x20,%ebx
0x00007d8f in ?? ()
(gdb)
=> 0x7d92:      cmp    %ebx,%esi
0x00007d92 in ?? ()
(gdb)
=> 0x7d94:      jbe    0x7d81
0x00007d94 in ?? ()
(gdb)
=> 0x7d81:      call   *0x10018
0x00007d81 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
```

For loop ends with **jbe 0x7d81**

Last instruction of the boot loader is **call *0x10018**

First instruction of the kernel is **mov %cr4, %eax**

The first page of the disk is read and loaded into memory location which is pointed at by the pointer **elf**. This page contains the elf header which has key information like the Program Header Table offset and segment details.

```
elf = (struct elfhdr*)0x10000;  // scratch space

//   1st page off disk
readseg((uchar*)elf, 4096, 0);

// Is this an ELF executable?
if(elf->magic != ELF_MAGIC)
  return;  // let bootasm.S handle error
```

Starting address of the first segment of the kernel is stored in **ph**. This is done by adding offset **phoff** to elf (which is the starting address).

eph points to location after the end of the last segment of the kernel. eph is set by adding phnum to the starting of **ph** set earlier.

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
```

It then iterates through each program segment, loading them into memory at specific addresses(**pa**) using 'readseg'. Any extra allocated memory is zeroed out (if size in memory is larger than its size in the file, happens if uninitialized static variables are present). This process continues until all segments are loaded, with the number of segments determined by the ELF header's **phnum** attribute.

```c
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);
}
```

Exercise 4:

VMA (Link Address) - Memory address from which the section expects to execute.
LMA (Load Address) - Memory address at which section should be loaded into memory.

$ objdump -h kernel

```
shobyy@LAPTOP-J1E9GRGL:~/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    00000def  00000000  00000000  00027620  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 000050b1  00000000  00000000  0002840f  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000845  00000000  00000000  0002d4c0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .debug_line_str 00000132  00000000  00000000  0002dd05  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 12 .comment      0000002b  00000000  00000000  0002de37  2**0
                  CONTENTS, READONLY
shobyy@LAPTOP-J1E9GRGL:~/xv6-public$
```

VMA and LMA of the .text section are different. This indicates that the sections in the kernel load and execute from different addresses.

$ objdump -h bootblock.o

```
shobyy@LAPTOP-J1E9GRGL:~/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      0000002b  00000000  00000000  000002e8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000318  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   00000585  00000000  00000000  00000358  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_abbrev 0000023c  00000000  00000000  000008dd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_line   00000283  00000000  00000000  00000b19  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_str    00000206  00000000  00000000  00000d9c  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line_str 00000041  00000000  00000000  00000fa2  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loclists 0000018d  00000000  00000000  00000fe3  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_rnglists 00000033  00000000  00000000  00001170  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
shobyy@LAPTOP-J1E9GRGL:~/xv6-public$
```

VMA and LMA of the .text section are the same. This indicates that the sections in the boot-loader load and execute from different addresses.

Exercise 5 :

Original(Correct) Makefile is shown :

```
Makefile ×
Ubuntu > home > shobyy > xv6-public > Makefile
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
```

Below is the changed Makefile with the altered address highlighted:

```
Makefile ×
Ubuntu > home > shobyy > xv6-public > Makefile
xv6memfs.img: bootblock kernelmemfs
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 0x7C64 -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
```

Then we ran the following commands:
make clean
make
make qemu
Below is the address the boot-loader would jump to if the link address was correct (unchanged).

```
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb)
The target architecture is set to "i386".
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
```

As we increased the link address, the same magnitude of increase is observed in the offset of ljump.

```
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb)
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c95
0x00007c2c in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b0
0x0000e062 in ?? ()
(gdb)
```

The first instruction that breaks is:

```
ljmp    $(SEG_KCODE<<3), $start32
```

This is because the previous instructions did not depend upon where they were stored in the memory. This is the first instruction that uses a specific offset to jump relative to the current address.

This instruction was responsible for the transition from 16 bit (real mode) to 32 bit (protected mode) instructions. Thus, as this instruction is affected, the desired transition will be hindered.

$ objdump -f kernel (To see the entry point)

```
shobyy@LAPTOP-J1E9GRGL:~/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:

```
(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
(gdb) x/8i 0x00100000
   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)
```

These are the values of the eight words when the boot loader is loaded. These values are all 0 because the BIOS loads the boot loader in memory locations from 0x7c00 to 0x7dff. So, the memory locations after 0x00100000 are empty.

```
(gdb) b *0x7d81
Breakpoint 2 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d81 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8i 0x00100000
   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    $0x109000,%eax
(gdb)
```

However, after when the boat loader loads the kernel, the kernel is loaded on memory location after 0x00100000 too. This led to new instructions after 0x00100000.

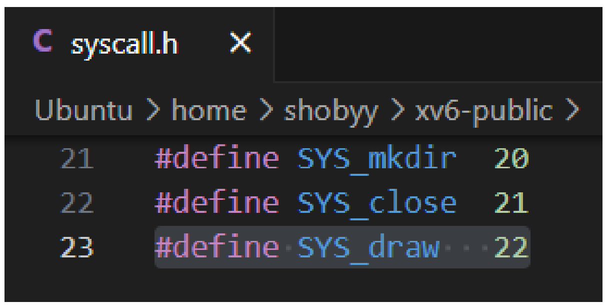
Assignment - 0B-1

An operating system operates in two modes: user mode and kernel mode, crucial for security and stability. In user mode, applications run with limited access to system resources, preventing disruptions. When an application needs access to protected resources, it makes a system call to the kernel. The CPU then switches to kernel mode, where the OS performs the necessary operations before returning to user mode. This process ensures applications run safely without compromising the system's integrity.

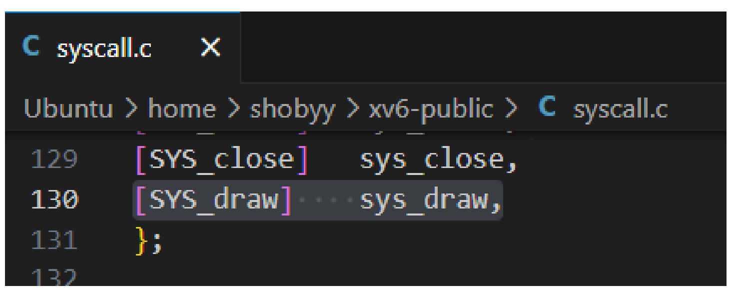
Exercise 1:

We did the following steps to add the ASCII image in the system call:

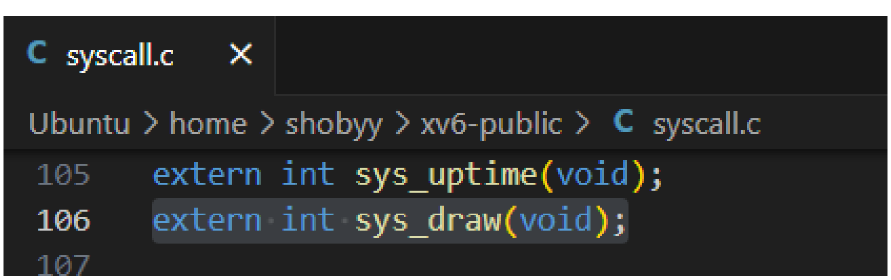
1. We added the line '#define SYS_draw 22' in **syscall.h**



2. We added the line '[SYS_draw] sys_draw' in **syscall.c**



Then we added the extern line because the actual function is in **sysproc.c**
We added the line 'extern int sys_draw(void)' in **syscall.c**



3. This is the actual sys_draw function in **sysproc.c**

sysproc.c

Ubuntu > home > shobyy > xv6-public > sysproc.c

```c
int
sys_draw(void)
{
  void* batman_buffer;
  uint sz;

  argptr(0, (void*)&batman_buffer, sizeof(batman_buffer));
  argptr(1, (void*)&sz, sizeof(sz));
  char txt[] = "\n\
                        ::                                       .-.                      \n\
                  :-+#@@=                                         =@@#+-.                 \n\
              .=#@@@@@@:                 .      .                  +@@@@@@*-.             \n\
            =#@@@@@@@@@                  +*     #=                 -@@@@@@@@@#-           \n\
         .+@@@@@@@@@@@@#.                %@=   =@%                -@@@@@@@@@@@@@+.        \n\
        +@@@@@@@@@@@@@@@@%+-:           :@@@@@@@@@:            :+%@@@@@@@@@@@@@@@@@*.     \n\
      -@@@@@@@@@@@@@@@@@@@@@@@%#*+=-:.. *@@@@@@@@@+   ..:-=+#%@@@@@@@@@@@@@@@@@@@@@@:     \n\
     =@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@+    \n\
    +@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@=   \n\
   =@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   \n\
   #*+===-----==+#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#*==-------==+*#-  \n\
                   .=%@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#-                    \n\
                      +@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@+                      \n\
                       #%*+=----=*#@@@@@@@@@@@@@@@@@@%*=-:...:-=*@@                       \n\
                                    :+%@@@@@@@@@@@@%+.             .                      \n\
                                       -%@@@@@@@@%-                                       \n\
                                         -%@@@@@=                                         \n\
                                           =@@#.                                          \n\
                                            :+   \n";

  if(sizeof(txt)>sz){
    return -1;
  }
  strncpy((char*)batman_buffer, txt, sz);
  return sizeof(txt);
}
```

4. Now we want the user to be able to use the system call. We add the line ‘SYSCALL(draw)’ in **usys.S**

usys.S

Ubuntu > home > shobyy > xv6-public > usys.S

```
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(draw)
```

Then we add the line ‘int draw(void*, uint)’ in **user.h**

user.h

Ubuntu > home > shobyy > xv6-public > user.h

```c
int sleep(int);
int uptime(void);
int draw(void*, uint);

```

Exercise 2 :

Then we created Drawtest.c that gets the image from the kernel and prints it in the terminal.

Drawtest.c

Ubuntu > home > shobyy > xv6-public > Drawtest.c

```c
#include "types.h"
#include "user.h"
#include "stat.h"

int main(void){
    static char batman_buff[2000];
    int res = draw((void*) batman_buff, sizeof(batman_buff));

    printf(1, "The draw system-call returns: %d\n%s", res, batman_buff);

    exit();
}
```

We then added **Drawtest.c** in the Makerfile in both **UPROGS** and **EXTRA**

Makefile

Ubuntu > home > shobyy > xv6-public > Makefile

```makefile
UPROGS=\
	_wc\
	_zombie\
	_Drawtest\

```

Makefile

Ubuntu > home > shobyy > xv6-public > Makefile

```
EXTRA=\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c Drawtest.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

Then we ran these commands :
make clean
make
make qemu
Then we ran **Drawtest** and got this output

Drive Link -
https://drive.google.com/drive/folders/1_XRUqc5Co5Reo1jQzHNwTydEWInbLWHR?usp=sharing