HARSHWARDHAN BHAKKAD

200101040

Assignment 0A

Exercise 1: The solution is in the ex1.c file in the zipped folder.

```
__asm__(
    "addl $1, %%eax\n\t"
    : "=a"(x)
    : "a"(x));
printf("Hello x = %d after increment\n", x);
```

In this function the inputs are x and 1.
It adds the values of both x and 1 and then stores the value in x itself.
Thus it increments x.

Exercise 2:

The "si" instruction in gdb is used to execute one machine instruction (follows a call). The above screenshot shows the first 4 instructions of the xv6 operating system. The first instruction is

[f000:fff0] 0xffff0: ljmp $0x3630,$0xf000e05b

Here, f000 is the Starting Code Segment, fff0 is the Starting Instruction Pointer, 0xffff0 is the Physical Address where this instruction resides, ljmp is the Instruction, 0x3630 is the Destination Code Segment, 0xf000e05b is the Destination Instruction Pointer.

After loading the BIOS jumps backward as there is only 16Bytes of space left in front. It jumps back to 0xf000:0xe05b from the initial position i.e. 0xf000:fff0 .It then set up an interrupt descriptor table and initializes various devices such as the **VGA dispay.**

The **cmp** instruction performs comparison .

The **jne** instruction is a conditional jump that follows a test.

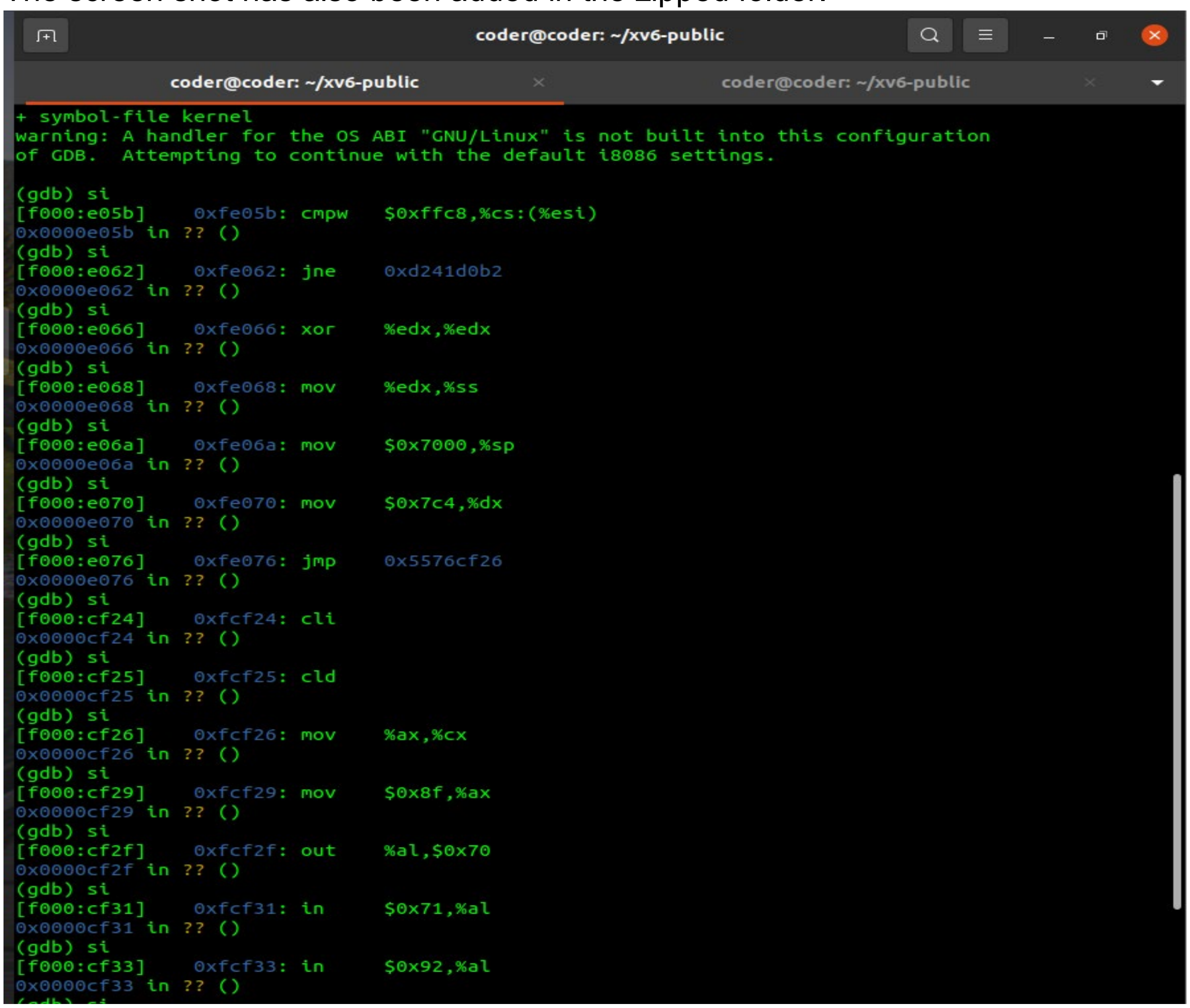
The **xor** instruction performs a logical XOR operation. It is equivalent to ‘^’ used in many languages.

The **mov** instruction move data bytes between the 2 specified locations.

The **jmp** instruction performs an unconditional jump.

The **cli** instruction clears the interrupt flag

The screen shot has also been added in the zipped folder.



Exercise 3:

```
// 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);
}
```

```
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>
}

static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
    7ca1:       b8 01 00 00 00          mov    $0x1,%eax
    7ca6:       ba f2 01 00 00          mov    $0x1f2,%edx
    7cab:       ee                      out    %al,(%dx)
    7cac:       ba f3 01 00 00          mov    $0x1f3,%edx
    7cb1:       89 d8                   mov    %ebx,%eax
    7cb3:       ee                      out    %al,(%dx)
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cb4:       89 d8                   mov    %ebx,%eax
    7cb6:       c1 e8 08                shr    $0x8,%eax
    7cb9:       ba f4 01 00 00          mov    $0x1f4,%edx
    7cbe:       ee                      out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cbf:       89 d8                   mov    %ebx,%eax
    7cc1:       c1 e8 10                shr    $0x10,%eax
    7cc4:       ba f5 01 00 00          mov    $0x1f5,%edx
    7cc9:       ee                      out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cca:       89 d8                   mov    %ebx,%eax
    7ccc:       c1 e8 18                shr    $0x18,%eax
    7ccf:       83 c8 e0                or     $0xffffffe0,%eax
    7cd2:       ba f6 01 00 00          mov    $0x1f6,%edx
    7cd7:       ee                      out    %al,(%dx)
    7cd8:       b8 20 00 00 00          mov    $0x20,%eax
    7cdd:       ba f7 01 00 00          mov    $0x1f7,%edx
    7ce2:       ee                      out    %al,(%dx)
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
    7ce3:       e8 96 ff ff ff          call   7c7e <waitdisk>
  asm volatile("cld; rep insl" :
    7ce8:       8b 7d 08                mov    0x8(%ebp),%edi
    7ceb:       b9 80 00 00 00          mov    $0x80,%ecx
    7cf0:       ba f0 01 00 00          mov    $0x1f0,%edx
    7cf5:       fc                      cld
    7cf6:       f3 6d                   rep insl (%dx),%es:(%edi)
  insl(0x1F0, dst, SECTSIZE/4);
}
    7cf8:       5b                      pop    %ebx
    7cf9:       5f                      pop    %edi
    7cfa:       5d                      pop    %ebp
    7cfb:       c3                      ret
```

```
  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);
  }
```

Beginning of the loop:

7d8d: 39 f3 cmp %esi,%ebx

End of the loop:

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

The explanation for the first instruction is that the first operation on entering the for loop will be comparison between the values of **ph** and **eph** because the loop will run only when **ph < eph**. The explanation of last instruction is that the loop ends when the values of **ph** and **eph** become equal and hence the loop jumps to the next instruction at **0x7d91**.

a)

```
(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)
```

By analysing the contents of bootasm.S, we reach the following conclusion. “**movw $(SEG_KDATA<<3), %ax**” is the first instruction to be executed in 32-bit mode. “**ljmp $(SEG_KCODE<<3), $start32**” instruction completes the transition to 32-bit protected mode.
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.

The code segment descriptor has a flag set that indicates that the code should run in 32 bit mode. Once it has loaded the **GDT** register, the boot loader enables protected mode by setting the 1 bit (CR0_PE) in register %cr0(control register) .

Set up protected mode data registers Enabling protected mode does not immediately change how the processor translates logical to physical addresses; it is only when one loads a new value into a segment register that the processor reads the GDT and changes its internal segmentation settings. One cannot directly modify %cs, so instead the code executes an **ljmp** (far jump) instruction , which allows a code segment selector to be specified. The boot loader then switches the processor from real mode to 32-bit protected mode, because it is only in this mode that software can access all the memory above 1MB in the processor's physical address space.

b) Last instruction of the boot loader executed:

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) b *0x7d91
Breakpoint 2 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018
```

7d91: ff 15 18 00 01 00   call   *0x10018
entry = (void(*)(void))(elf->entry);
entry();
The first instruction of kernel it just loaded is:

```
Thread 1 hit Breakpoint 2, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
```

0x10000c: mov   %cr4,%eax

c) First instruction of the kernel it just loaded:

The information about the the number of program segments is stored in the phnum attribute of the elf binary header .
The elf header has fixed length, it is the variable – length program segment which needs to be calculated.
The number of sectors in a particular segment is further calculated by **(ph->filesz(count)/SECTSIZE(512))** with appropriate offset settings.

```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);
}

```

The above lines of code are present in bootmain.c. This is the code that is used by xv6 to load the kernel. xv6 first loads ELF headers of kernel into a memory location pointed to by “elf”. Then it stores the starting address of the first segment of the kernel to be loaded in “ph” by adding an offset (“elf->phoff”) to the starting address (elf). It also maintains an end pointer eph which points to the memory location after the end of the last segment. It then iterates over all the segments. For every segment, pa points to the address at which this segment has to be loaded. Then it loads the current segment at that location by passing pa, ph->filesz and ph->off parameters to readseg. It then checks the memory assigned to this sector is greater than the data copied. If this is true, it initializes the extra memory with zeros.

Coming back to the question, the boot loader keeps loading segments while the condition “ph < eph” is true. The values of ph and eph are determined using attributes phoff and phnum of the ELF header. So the information stores in the ELF header helps the boot loader to decide how many sectors it has to read.

Exercise 4:

Line 1:

a, b and c are pointers to integer variables. a is allocated 16 bytes of memory on the stack. b is allocated 16 bytes of information on the heap. The pointer c is declared but is uninitialized. So it stores some junk pointer.

Line 2:

The for loop on line 15 changes the value of integers in array a to 100, 101, 102, 103. The line "c=a;" makes the pointer c point to the same integer as a. Therefore when c[0] is assigned 200 it changes the first element in array a because c is just another name for array a.

Line 3:
c[1]=300; - Changes a[1] to 300 as c is an alias for a. *(c+2)=301; - Another way of saying c[2]=301. a[2] is set to 301. 3[c]=302; -Another way of saying c[3]=302. a[3] is set to 302

Line 4:
c=c+1; - This makes c point to the location of a[1]. *c=400; - This changes a[1] to 400.

Line 5:
c = (int *) ((char *) c + 1); - The hexadecimal value of the address stored in pointer c increases by only 1 since we typecast it to a character pointer before incrementing it. This is because the size of a character type data in C is 1 byte. The pointer c is then typecast back into an integer type. At the end of this c points to a segment of 4 bytes beginning from the second byte of a[1] and ending at the first byte of a[2].
The contents of a[2] and a[3] at this point look as follows.

a[2]=400 a[3]=301

10010000 10000000 00000000 00000000 00101101 10000000 00000000 00000000

After *c=500 it changes to:

a[2]=128144 a[3]=256

10010000 11110100 00000001 00000000 00000000 10000000 00000000 00000000

Line 6:

b=(int*)a+1; - Increases hexadecimal address value by 4.
c = (int *) ((char *) a + 1); - Increases hexadecimal address value by only 1

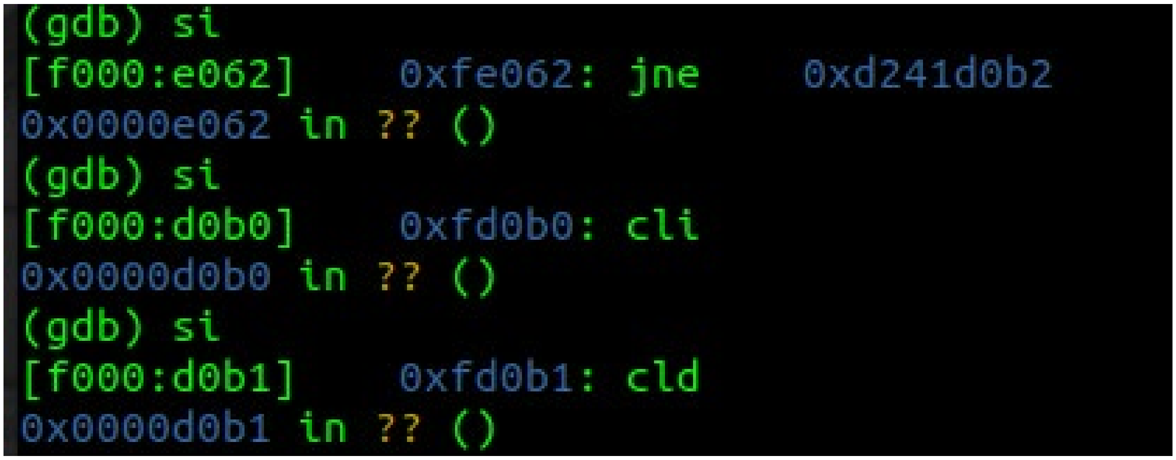
```
coder@coder:~/xv6-public$ objdump -h kernel

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
 Terminal         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    00000eaa  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c2  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde0  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002a  00000000  00000000  00030ae8  2**0
                  CONTENTS, READONLY
coder@coder:~/xv6-public$
```

```
coder@coder:~/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    0000021f  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  0000101f  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012da  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
coder@coder:~/xv6-public$
```

Exercise 5:

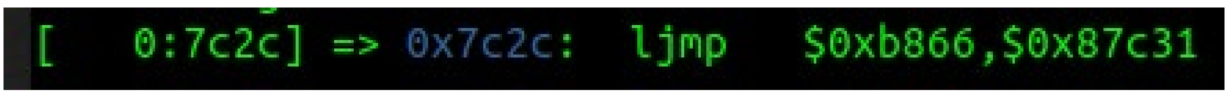
Changed address 1st point of difference

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39
```

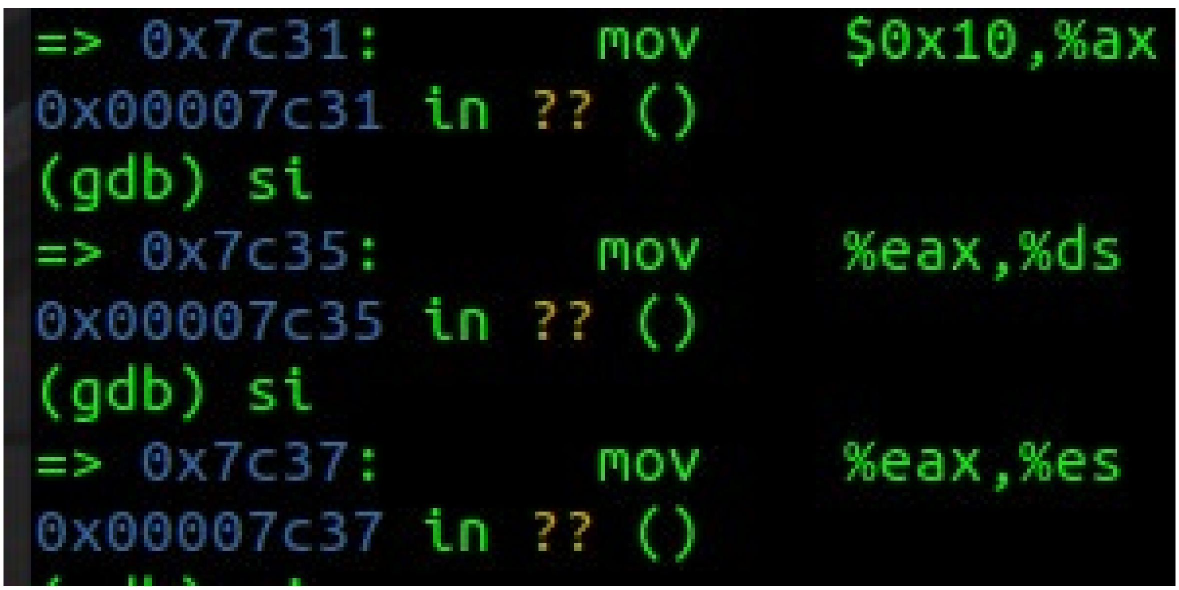


```
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb) si
[f000:d0b0]    0xfd0b0: cli
0x0000d0b0 in ?? ()
(gdb) si
[f000:d0b1]    0xfd0b1: cld
0x0000d0b1 in ?? ()
```

Original address 1st point of difference



```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
```



```
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) si
=> 0x7c35:      mov    %eax,%ds
0x00007c35 in ?? ()
(gdb) si
=> 0x7c37:      mov    %eax,%es
0x00007c37 in ?? ()
```

I changed the link address from 0x7c00 to 0x7c08. Since no change has been done to the BIOS, it will run smoothly for both of the versions and hand over the control to the boot loader. From this point onwards, we have to check for differences between the two files. I did it by using si command repeatedly to get the next 200 (approx.) instructions and then comparing the outputs of the two files. The first command where a difference was spotted is shown below along with the next 3 instructions. The first picture is when the link address was correctly set to 0x7c00 and the second picture is when it was changed to 0x7c08. I have attached the output files of gdb in my submission. I have also attached output files of “objdump -h bootmain.o” for both of the versions since the outputs differ due to the change in link address.

```
coder@coder:~/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:

For this experiment, we have to examine the 8 words of memory at 0x00100000 at two different instances of time, the first when the BIOS enters boot loader and the second when the boot loader enters the kernel. For this, we will use the command **“x/8x 0x00100000”** but before that we will have to set 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.

```
(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) b *0x0010000c
Breakpoint 2 at 0x10000c
(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
```

As we can see in the diagram, we get different values at both the breakpoints. The explanation to this is as follows. The address 0x00100000 is actually 1MB which is the address from where the kernel is loaded into the memory. Before the kernel is loaded into the memory, this address contains no data (i.e. garbage value). By default, all the uninitialized values are set to 0 in xv6. Hence, when we tried to read the 8 words of memory at 0x00100000 at the first breakpoint, we got all zeroes since no data had been loaded until that point. When we check the values at the second breakpoint, the kernel has already been loaded into the memory and thus this address now contains meaningful data instead of zeroes.

Assignment 0B

Exercise 1: An operating system supports two modes; the kernel mode and the user mode. When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that particular resource. This is done via a system call. When a program makes a system call, the mode is switched from user mode to kernel mode.

I have defined the system call as messi. And made the final c file named as drawtest.c

Adding system call :

5 files that are added :

Syscall.h: This file assigns a number to every system call in xv-6 system. Before adding messi there were 21 system calls , hence messi was assigned number 22.
Line:#define SYS_messi 22

Syscall.c: It contains an array of function pointers(syscalls[]) which uses index defined in systemcall.h to point to the respective system call function stored at a different memory location. We also put a function prototype here(but not implementation).
Line1:Extern int sys_messi(void);
Line2:[SYS_messi] sys_messi,

Sysproc.c: This is where the implementation of our system call is written.
user.h and usys.S: They act as an interface for our system to access the system call. The function prototype is added in user.h(included as header file in our program) while instruction to treat it as a system call is included in usys.S
usys.S: SYSCALL(messi)
user.h: int messi(void*,int);

Exercise 2: drawtest.c file has also been attached in the folder.

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

int
main(void)
{
        static char buf[20000];
        printf(1,"messi system call return %d\n",messi((void*) buf,20000));

        printf(1,"%s",buf);
        exit();
}
```

After this, I added this file to the Makefile under UPROGS and EXTRA.

Then I used “make clean”, “make”, “make qemu-nox” and then entered wolfietest to get the following output. Also, my program is also listed when I use the ls command. Screenshot is attached below

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ drawtest.c
exec: fail
exec drawtest.c failed
$ drawtest
messi system call return 2768
Its a goal!!!
```

Text Editor

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16268
echo           2 4 15120
forktest       2 5 9432
grep           2 6 18488
init           2 7 15708
kill           2 8 15148
ln             2 9 15004
ls             2 10 17636
mkdir          2 11 15248
rm             2 12 15228
sh             2 13 27864
stressfs       2 14 16136
usertests      2 15 67248
wc             2 16 17000
zombie         2 17 14820
drawtest       2 18 14988
console        3 19 0
```