ASSIGNMENT 1

OS344 - Operating Systems Laboratory

Part 1: PC Bootstrap

1.

```c
// Simple inline assembly example
//
#include <stdio.h>
int
main(int argc, char **argv)
{
    int x = 1;
    printf("Hello x = %d\n", x);

    // in-line assembly code to increment
    // the value of x by 1
    __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");
    }
}
```

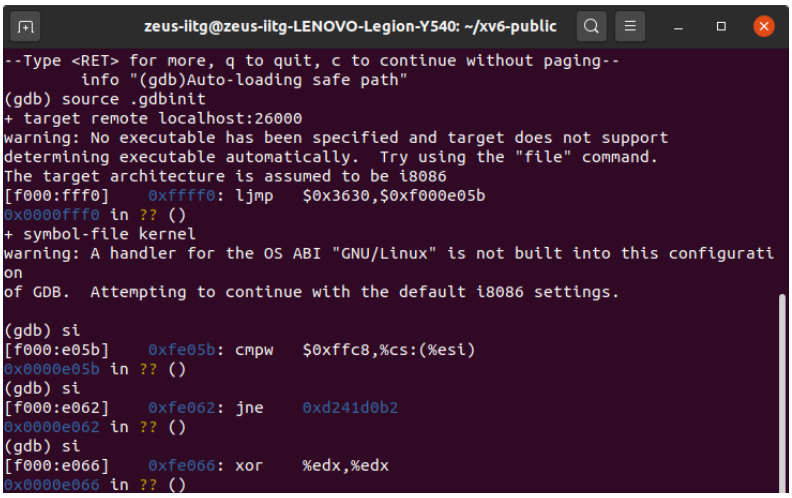
```
PS C:\Users\zeus_iitg\Desktop\os report> gcc ex1.c
PS C:\Users\zeus_iitg\Desktop\os report> ./a.exe
Hello x = 1
Hello x = 2 after increment
OK
```

Added code:

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

In this, x and 1 are the input operands while x is the output operand. This code adds the value of x and 1 and saves the output to x. Hence, incrementing the value by 1.

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**.

The **cmp** instruction is used to perform comparison. It's identical to the sub instruction except it does not affect operands.

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 cleared (0). jnz is commonly used to explicitly test for something not being equal to zero whereas jne is commonly found after a cmp instruction.

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

Part 2: The Boot Loader

3. The code for readsect() is given below

```
// 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);
}
```

```
waitdisk();
  7c9c:e8 dd ff ff ff          call   7c7e <waitdisk>
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
waitdisk();
  7ce3:       e8 96 ff ff ff          call   7c7e <waitdisk>
insl(0x1F0, dst, SECTSIZE/4);
```

The assembly code for readsect() is given above. Now we will discuss about the for loop that reads the sectors of kernel from the disk. The code is given below:

```
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 first instruction of this for loop is

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

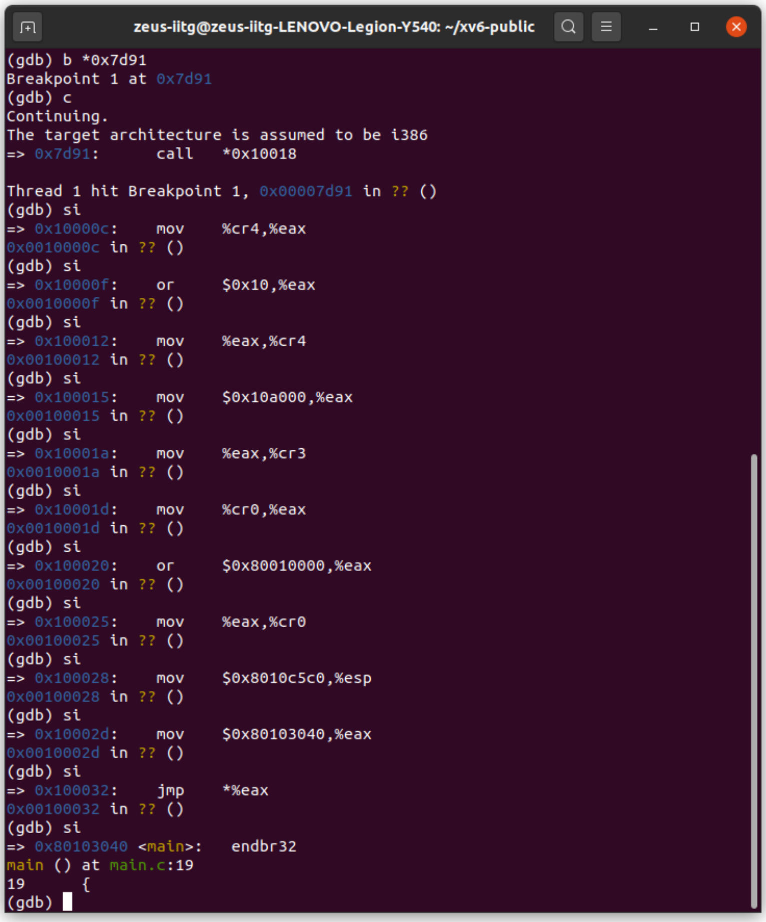
The last instruction of this for loop is

```
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. Hence the jump instruction will be the last instruction of the for loop. The next instruction after the for loop is

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

Making a breakpoint at that address and then stepping into further instructions gives the following output.



```
zeus-iitg@zeus-iitg-LENOVO-Legion-Y540: ~/xv6-public
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:    or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:    mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:    mov    $0x10a000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:    mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:    or     $0x80010000,%eax
0x00100020 in ?? ()
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) si
=> 0x100028:    mov    $0x8010c5c0,%esp
0x00100028 in ?? ()
(gdb) si
=> 0x10002d:    mov    $0x80103040,%eax
0x0010002d in ?? ()
(gdb) si
=> 0x100032:    jmp    *%eax
0x00100032 in ?? ()
(gdb) si
=> 0x80103040 <main>:   endbr32
main () at main.c:19
19      {
(gdb)
```

```
  # 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
  movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
```

(a) 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.

(b) By analysing the contents of bootasm.S, bootmain.c and bootblock.asm, we conclude that bootasm.S switches the OS into 32-bit mode and then calls bootmain.c which first loads the kernel using ELF header and the enters the kernel using `entry()`. Hence the last instruction of bootloader is `entry()`. Looking for the same in bootblock.asm, we find out the instruction to be

```
1. 7d91:     ff 15 18 00 01 00       call    *0x10018
```

which is a call instruction which shifts control to the address stored at 0x10018 since **dereferencing operator** (*) has been used. Now we need to know the starting address of the kernel. We can find this by two methods:

(i) By looking at the first word of memory stored at 0x10018 (by using the command **"`x/1x 0x10018`"**)
(ii) By looking at the contents of **"`objdump -f kernel`"**

After getting the starting address of kernel, we need to see what is the instruction stored at that address to get the first instruction of kernel. We can do this by two methods:

(i) By using **"`x/1i 0x0010000c`"**
(ii) By looking into kernel.asm

```
(gdb) b *0x7d91
Breakpoint 1 at 0x7d91
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d91:      call   *0x10018

Thread 1 hit Breakpoint 1, 0x00007d91 in ?? ()
(gdb) x/1x 0x10018
0x10018:        0x0010000c
(gdb) x/1i 0x0010000c
   0x10000c:    mov    %cr4,%eax
```

Hence, the first instruction of kernel is

```
1. 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);
}
```

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.

4.

```
zeus-iitg@zeus-iitg-LENOVO-Legion-Y540:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000717a  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       0000101f  80107180  00107180  00008180  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00002516  80109000  00109000  0000a000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          0000af88  8010b520  0010b520  0000c516  2**5
                  ALLOC
  4 .debug_line   00006cfd  00000000  00000000  0000c516  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_info   0001225d  00000000  00000000  00013213  2**0
```

As we can see in the above screenshot, VMA and LMA of .text section is different indicating that it loads and executes from different addresses.

```
zeus-iitg@zeus-iitg-LENOVO-Legion-Y540:~/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      00000024  00000000  00000000  000002f8  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000320  2**3
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  4 .debug_info   000005d2  00000000  00000000  00000360  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

As we can see in the above screenshot, VMA and LMA of .text section is same indicating that it loads and executes from the same address.

5. 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.

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c31:  mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb)
=> 0x7c35:  mov    %eax,%ds
0x00007c35 in ?? ()
(gdb)
=> 0x7c37:  mov    %eax,%es
```

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39
0x00007c2c in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb)
[f000:e062]    0xfe062: jne    0xd241d0b2
0x0000e062 in ?? ()
(gdb)
[f000:d0b0]    0xfd0b0: cli
```

```
PS C:\Users\zeus_iitg\Desktop\xv6-public> objdump -f kernel

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

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      0xa000b8e0      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.

Part 3: Adding a System Call

7. 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.

In order to define our own system call in xv6, changes need to be made to 5 files. Namely, these files are as follows.

(i) syscall.h
(ii) syscall.c
(iii) sysproc.c
(iv) usys.S
(v) user.h

We would start the procedure by editing syscall.h in which a number is given to every system call. This file already contains 21 system calls. In order to add the custom system call, the following line needs to be added to this file.

```
#define SYS_wolfie 22
```

Next, we need to add a pointer to the system call in the syscall.c file. This file contains an array of function pointers which uses the above-defined numbers (indexes) as pointers to system calls which are defined in a different location. In order to add our custom system call, add the following line to this file.

```
[SYS_wolfie] sys_wolfie,
```

When the system call with number 22 is called by a user program, the function pointer sys_wolfie which has the index SYS_wolfie or 22 will call the system call function. Hence, our next objective is to implement a system call function. But we do not implement the system call function in syscall.c. We only add a prototype as shown below.

```
extern int sys_wolfie(void);
```

We implement the system call function in sysproc.c.

```
int
sys_wolfie(void)
{
  void* buf;
  uint size;

  argptr(0, (void*)&buf, sizeof(buf));
  argptr(1, (void*)&size, sizeof(size));

  char text[] = "Say Hi to wolfie: \n\
                                 ,ood8888booo,\n\
                              ,od8           8bo,\n\
                           ,od                   bo,\n\
                         ,d8                       8b,\n\
                        ,o                           o,    ,a8b\n\
                       ,8                             8,,od8  8\n\
                       8'                             d8'     8b\n\
                       8                           d8'ba     aP'\n\
                       Y,                       o8'         aP'\n\
                        Y8,                      YaaaP'    ba\n\
                         Y8o                   Y8'         88\n\
                          `Y8               ,8\"           `P\n\
                            Y8o        ,d8P'              ba\n\
                       ooood8888888P\"\"\"'                   P'\n\
                    ,od                                  8\n\
                 ,dP     o88o                           o'\n\
                ,dP          8                          8\n\
               ,d'   oo       8                       ,8\n\
               $    d$\"8      8           Y    Y  o   8\n\
              d    d  d8    od  \"\"zeus<3nezu  d\"\" 8   8\n\
              $    8  d   ood' ,   8        b  8   '8  b\n\
              $   $  8  8     d  d8        `b  d    '8  b\n\
               $  $ 8   b    Y  d8          8 ,P      '8  b\n\
               `$$  Yb  b     8b 8b         8 8,      '8  o,\n\
                    `Y  b      8o  $$o      d  b        b   $o\n\
                     8   '$     8$,,$\"      $   $o      '$o$$\n\
                      $o$$P\"                 $$o$\n\n";
  if(sizeof(text)>size)
    return -1;

  strncpy((char *)buf, text, size);
  return sizeof(text);
}
```

Now, to add an interface for a user program to call the system call, we add

```
SYSCALL(wolfie)
```

to usys.S and

```
int wolfie(void*, uint);
```

to user.h

8. Now the only task left is to add a user program to call the system call that we just made above. For this, I made a file wolfietest.c inside xv6 folder and wrote the following code in it.

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

int
main(void)
{
    static char buf[2000];
    printf(1,"wolfie sys call returns %d\n",wolfie((void*) buf,2000));

    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.