Gunjan Dhanuka   Roll - 200101038
Dept - Computer Science and Engineering

CS344 - Assignment 0A

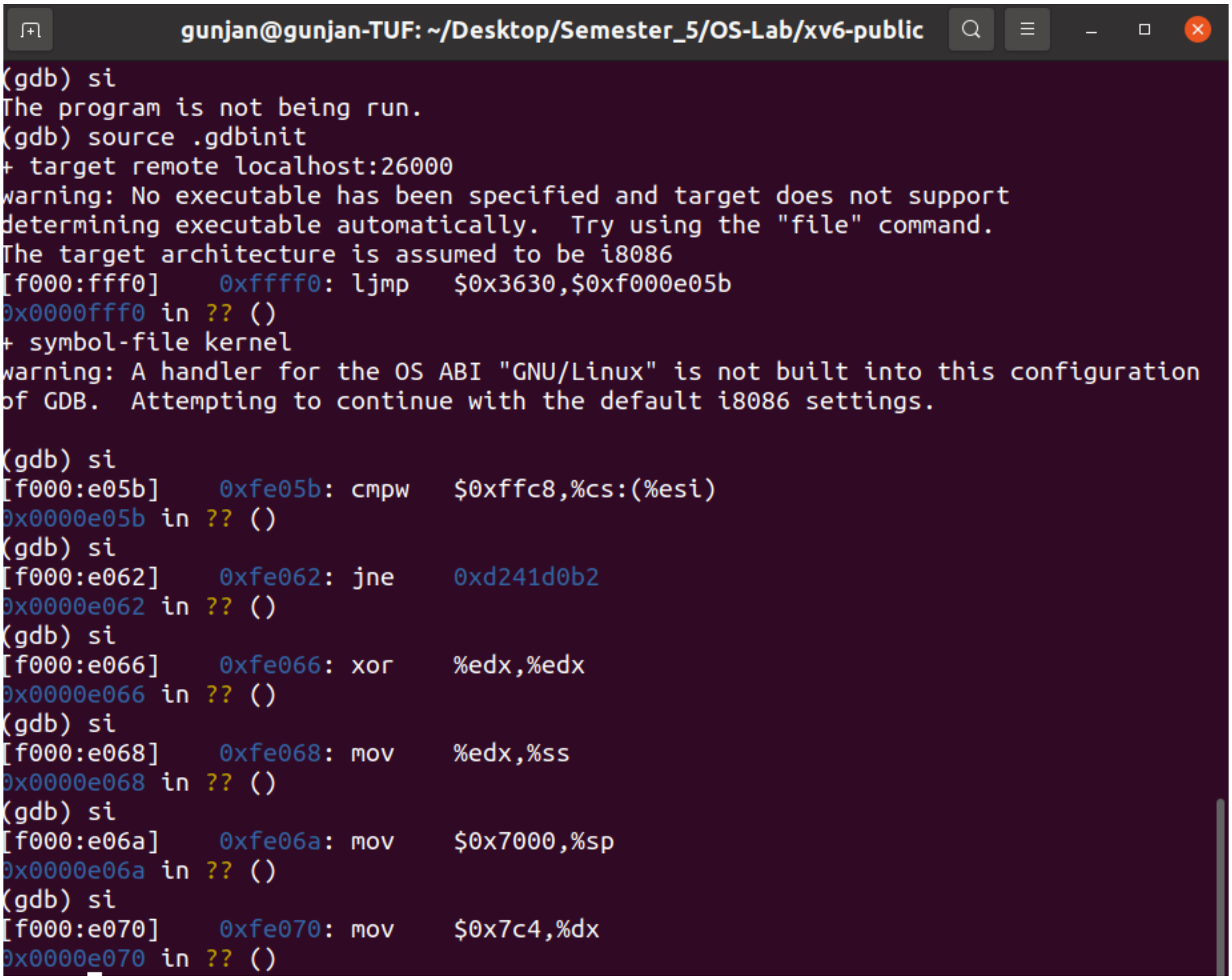
Exercise 1

```
● (base) gunjan@gunjan-TUF:~/Desktop/Semester_5/OS-Lab/Assignment-0$ gcc ex1.c
● (base) gunjan@gunjan-TUF:~/Desktop/Semester_5/OS-Lab/Assignment-0$ ./a.out
  Hello x = 1
  Hello x = 2 after increment
  OK
```

```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__("inc %0" : "+r" (x));
    printf("Hello x = %d after increment\n",
            x);
    if (x == 2)
    {
        printf("OK\n");
    }
    else
    {
        printf("ERROR\n");
    }
}
```

The new line added is __asm__("inc %0" : "+r" (x));
Which is the inline assembly code for incrementing the value of x using the INC instruction.

Exercise 2



gunjan@gunjan-TUF: ~/Desktop/Semester_5/OS-Lab/xv6-public

```
(gdb) si
The program is not being run.
(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) 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 ?? ()
```

1. [f000:fff0]   0xffff0:   ljmp $0x3630,$0xf000e05b
   a. It is a jump instruction, which jumps to the segmented address CS = 0xf000 and IP = 0xe05b
2. [f000:e05b]   0xfe05b: cmpw $0xffc8,%cs:(%esi)
   a. It is a compare instruction, which compares the 16-bit content in the address 0xffc8, with the CS value and the output is stored in the ESI (Source index register).
3. [f000:e062]   0xfe062:   jne   0xd241d0b2
   a. If the zero flag is cleared (0), instruction jumps to the location 0xd241d0b2.
4. [f000:e066]   0xfe066:   xor   %edx,%edx
   a. Set the value in the EDX register to 0 by XORing it with itself.
5. [f000:e068]   0xfe068:   mov   %edx,%ss
   a. Moves the value from Stack Segment (SS) to the EDX Register.
6. [f000:e06a]   0xfe06a: mov $0x7000,%sp
   a. Moves the value from stack pointer to the address 0x7000.

7. [f000:e070]   0xfe070:   mov $0x7c4,%dx
    a. Moves the value from the data register to the address 0x7c4.

```
[f000:e076]    0xfe076: jmp    0x5576cf26
0x0000e076 in ?? ()
(gdb) si
[f000:cf24]    0xfcf24: cli
0x0000cf24 in ?? ()
(gdb) si 5
[f000:cf31]    0xfcf31: in     $0x71,%al
0x0000cf31 in ?? ()
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli
```

The above diagram shows using breakpoints in GDB.

Exercise 3

The code for readsect() is given below

```c
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 corresponding assembly code for readsect() is:

```
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

  // Read data.
  waitdisk();
    7ce3:   e8 96 ff ff ff          call   7c7e <waitdisk>
```

The for loop that reads the sectors of the kernel from the disk 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 loop starts from this assembly command:

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

The last instruction for this loop is:

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

The first instruction is a compare operation between the values of ph and eph on the entering the for loop to check if the value of ph < eph or not. Only then will the loop continue further.

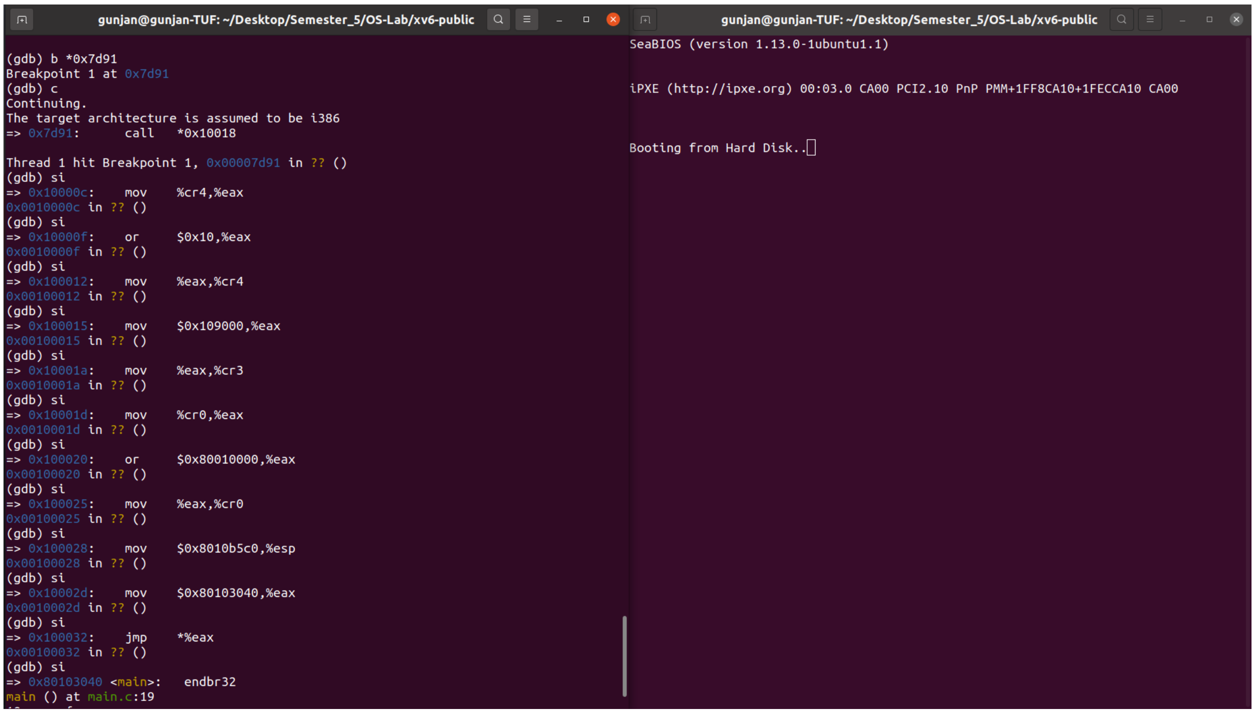
The last instruction shows that the loop should finish when the values of ph and eph are equal and hence the loop condition of ph < eph is false. Then the control jumps using the jump statement to 0x7d91.

The next instruction after the for loop finishes is:

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

So we set a breakpoint at 0x7d91, and then continue execution till the breakpoint. Then we step through the rest of the bootloader one instruction at a time.

Here is the terminal output:



```
  # 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
  movw    %ax, %ds                # -> DS: Data Segment
  movw    %ax, %es                # -> ES: Extra Segment
  movw    %ax, %ss                # -> SS: Stack Segment
  movw    $0, %ax                 # Zero segments not ready for use
  movw    %ax, %fs                # -> FS
  movw    %ax, %gs                # -> GS
```

a. *At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?*

   By analyzing the code in Bootasm.S, we can say that the first instruction where the processor starts executing 32-bit code is from the line “movw $(SEG_KDATA<<3), %ax # Our data segment selector”.
   The “ljmp $(SEG_KCODE<<3), $start32” instruction completes the transition into the 32-bit protected mode.

b. *What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?*

   We can say that bootasm.S switches the OS into 32-bit mode and then calls bootmain.c. bootmain.c first loads the kernel using the ELF header and then enters the kernel using entry(). Hence the last instruction of the bootloader is entry().
   Its corresponding instruction in bootblock.asm is:

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

Which is a call instruction, thus shifting the control to the address stored the location 0x10018. A dereferencing operator (*) has been used to get the address stored at the location 0x10018. Now to get the starting address of the kernel, we can look at the first word of memory stored at 0x10018 (using **x/1x 0x10018** or by using "**objdump -f kernel**").

After getting the starting address of the kernel, to get the instruction stored at that address, we can use **x/1i 0x0010000c** or look into the kernel.asm file.

```
gunjan@gunjan-TUF: ~/Desktop/Semester_5/OS-Lab/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) x/1x 0x10018
0x10018:        0x0010000c
(gdb) x/1i 0x0010000c
   0x10000c:    mov    %cr4,%eax
(gdb) 
```

c. *How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?*

```
// 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 snippet is taken from the bootmain.c file. It is used by the xv6 to load the kernel in a stepwise manner:

1. xv6 loads the ELF headers of the kernel into a memory location pointed at by “elf”.
2. It then stores the starting address of the first segment of kernel to be loaded in “ph” by adding an offset (elf->phoff) to the starting address (elf).
3. An end pointer (eph) is also maintained which points to the location after the end of last segment.
4. Now the loop iterates over all the segments between ph and eph. For every segment, “pa” points to the address to which the current segment is to be loaded. Then it loads the segment by using the “readseg” function with the parameters as pa, ph->filesz, and ph->off. Then it checks if the memory assigned to this sector is greater than the copied data size, and if it holds true, the extra memory is initialized by zeros.

Thus, the bootloader keeps loading segments while “ph < eph” holds. The values of ph and eph are obtained from the phoff and phnum attributes of the ELF header. Thus the information stored in the ELF header helps the boot loader decide the number of sectors it has to read.

Exercise 4

As we can see in the screenshots below, VMA (link address) and LMA (load address) of the .text are different indicating that it loads and executes from different addresses.

```
                 CONTENTS, READONLY
(base) gunjan@gunjan-TUF:~/Desktop/Semester_5/OS-Lab/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    00000ec5  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295dd  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fdfb  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      0000002b  00000000  00000000  00030b03  2**0
                  CONTENTS, READONLY
```

As we can see in the below screenshot, the VMA and LMA of .text section are the same, showing that it loads and executes from the same address. The BIOS loads the boot sector into memory starting at 0x7c00, so this is the boot sector's load address. This is also where the boot sector executes from, so this is also its link address.

```
                 CONTENTS, READONLY
(base) gunjan@gunjan-TUF:~/Desktop/Semester_5/OS-Lab/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      0000002b  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    0000023a  00000000  00000000  00000e00  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_loc    000002bb  00000000  00000000  0000103a  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_ranges 00000078  00000000  00000000  000012f5  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
```

Exercise 5

Here we trace through few instructions of the bootloader and try to identify the first instruction that would “break” if we did something wrong with the bootloader’s link address.

Then we change the linking address in the Makefile to something else (0x7c08). Then we use make clean and recompile with make, and then check what and where it goes wrong.

Case1: The link address is set to the default value (0x7c00)

```
(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 ?? ()
```

Case 2: After changing to 0x7c08, we see that the instructions differ from the screenshot above.

```
(gdb) b *0x7c2c
Breakpoint 2 at 0x7c2c
(gdb) c
Continuing.
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c39

Thread 1 hit Breakpoint 2, 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 ?? ()
```

```
(base) gunjan@gunjan-TUF:~/Desktop/Semester_5/OS-Lab/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

We use the command “x/8x 0x00100000” to examine the 8 words in the memory location 0x0010000 at two instances. So our breakpoints will be:

a. 0x7c00 : When the bootloader gets control from the BIOS
b. 0x7d91: When the bootloader hands control to the kernel, and the first command of the kernel is executed at the location 0x0010000c.

The contents in the memory address 0x00100000 when the bootloader starts executing.

```
(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
```

The contents in the memory address 0x00100000 when the bootloader enters the kernel.

```
(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) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
```

From the above snippets, we can clearly see that there is a difference in the values at both the breakpoints. Actually, the address 0x00100000 is 1 MB which is the address that the kernel uses to load into the memory. Before loading of kernel, this location contains no data. And all uninitialized values are set to 0 in the xv6 paradigm by default.

Then at the second breakpoint the kernel has now been loaded into the memory and thus this address contains useful data instead of garbage value (zeros).

CS-344 Assignment 0B

Exercise 1

In Order to define a custom system call in xv6, changes have to be made to 5 files. These files are:

a. syscall.h
b. Syscall.c (kernel side of the system call table)
c. Sysproc.c
d. usys.S (user level system definitions)
e. user.h (user level header for system calls)

Firstly we edit the syscall.h file, by adding a new entry for our new system call, with the number 22 being assigned to it.

```
#define SYS_draw   22
```

Secondly, we define a pointer to the system call in the syscall.c file. In this file, we see an array of function pointers using the numbers defined in syscall.h file as indexes to system calls defined in a different location.

```
[SYS_draw]    sys_draw,
```

Now when the user program calls the system call with number 22, the function pointer sys_draw which has index SYS_draw will call the system call function. We add a prototype of the system call function to the syscall.c file.

```
extern int sys_draw(void);
```

Then I implemented the sys_draw function in the sysproc.c file. According to the question, the program copies an ASCII art to a user supplied buffer, with the provision to output a negative number (-1) incase the buffer is smaller than the size of the ASCII string. If the calls succeeds, we will return the number of bytes copied.

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

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

  char text[] = R"(

    Say Hi to Squirtle!



               _,........__
            ,-'            "`-.
          ,'                   `-.
        ,'                        \
      ,'                           .
      .'\               ,"".       `
     ._.'|             / |  `       \
     |   |            `-.'  ||       `.
     |   |            '-._,'||       | \
     .`.,'             `..,'.'       , |`-.
     l                       .'`.  _/  |   `.
     `-.._'-   ,          _ _'   -" \  .     `
`."""""'-.`-...,---------','         `. `....__.
.'        `"-..___      __,'\          \  \     \
\_ .          |   `""""'    `.           . \     \
  `.          |              `.          |  .     L
    `.        |`--...________.'.        j   |     |
      `._    .'      |          `.     .|   ,     |
         `--,\       .            `7""' |  ,      |
            ` `      `            /     |  |      |    _,-'"""`-.
             \ `.     .          /      |  '      |  ,'          `.
              \  v.__  .        '       .   \    /| /              \
               \/    `""\"""""""`.       \   \  /.''                |
                `        .        `._ ___,j.  `/ .-       ,---.     |
                ,`-.      \         ."     `.  |/        j     `    |
               /    `.     \       /         \ /         |     /    j
              |       `-.   7-.._ .          |"          '         /
              |          `./_    `|          |            .     _,'
              `.           / `----|          |-............`---'
                \          \      |          |
               ,'           )     `.         |
                7____,,..--'      /          |
                                  `---.__,--.'gunjan


    )";


    if(sizeof(text)>size){
      return -1;
    }
    strncpy((char *)buf, text, size);
    return sizeof(text);
}
```

Then I added an interface to call the system call for the user program in usys.S.

```
SYSCALL(draw)
```

Then I added this prototype to user.h for calling the function from the user side.

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

Exercise 2

Now we have to add a user program to call the system call that we have created above. I made a file called drawtest.c inside the xv6 directory, and included the following code. This file gets the image from the kernel, and prints it to the console.

```
//drawtest.c

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

int main(void){
    static char buf[5000];
    printf(1, "draw sys call returns %d\n", draw((void*) buf, 5000));

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

After creating this user program, I made changes to the Makefile under the UPROGS and EXTRA sections. The changes are as follows:

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_drawtest\
```

```
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\
```

After making these changes, on the terminal run the following commands:

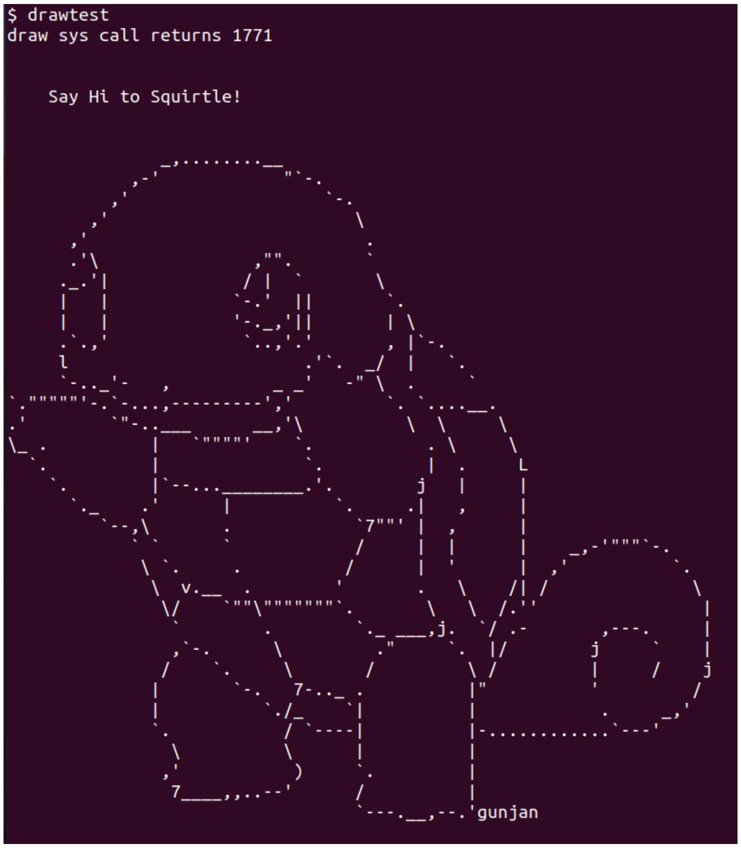
a. make clean
b. make
c. make qemu-nox

Now if we run ls, the output is:

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16296
echo           2 4 15148
forktest       2 5 9460
grep           2 6 18512
init           2 7 15732
kill           2 8 15176
ln             2 9 15028
ls             2 10 17660
mkdir          2 11 15272
rm             2 12 15252
sh             2 13 27892
stressfs       2 14 16164
usertests      2 15 67268
wc             2 16 17028
zombie         2 17 14840
drawtest       2 18 15004
console        3 19 0
```

The drawtest command is now visible here!

Now run ‘drawtest’ in the command prompt.



Hence the task is completed!

I have also supplied a patch file called “file.patch” which you can apply to get all the changes. Also, I have provided the codes of all the files where I made the changes, so just copying and overwriting them on the existing files would also work.

For any questions in code execution/report,
please contact me at -
Gunjan Dhanuka,
Email: d.gunjan@iitg.ac.in
Phone: +91-7240227672