CS 344 : OPERATING SYSTEMS LAB
LAB 1

GROUP NUMBER : 12

GROUP MEMBERS:

MOHAMMAD HUMAM KHAN (180123057)

KARTIKEYA SINGH (180123021)

SIDDHARTHA JAIN (180101078)

AB SATYAPRAKASH (180123062)

EXERCISE 1

Code:

```
#include <stdio.h>

int main(int argc, char **argv)
{
	int x = 1;
	printf("Hello x = %d\n", x);

	asm ("movl %1, %0;":"=r"(x):"r"(x+1):);

	printf("Hello x = %d after increment\n", x);
	if(x == 2) printf("OK\n");
	else printf("ERROR\n");
}
```

Output:

```
kartikeya@Kartikeya-PC:~/IITG/Sem 5/CS344/Lab1$ gcc ex1.c
kartikeya@Kartikeya-PC:~/IITG/Sem 5/CS344/Lab1$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
```

The explanation for inline assembly code is as follows :

1. The format of assembly inline block in C is :

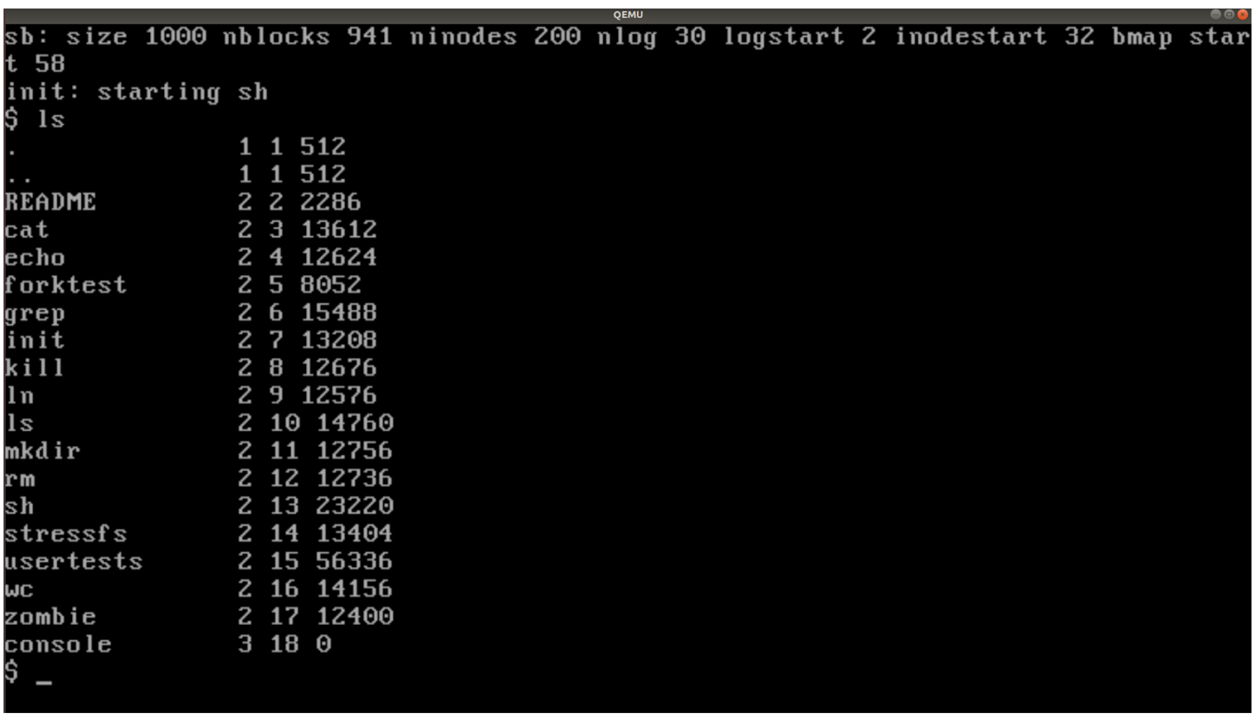
```
asm [volatile] ( AssemblerTemplate
                 : Output Operands
                 [ : Input Operands
                 [ : Clobbers ] ])
```

2. In asm, **%0** refers to the first variable (x in this case) passed as Output/Input Operand and **%1** (x+1 in this case) refers to the second variable.
3. The effective operation performed by the assembly block is **x = x + 1**.
4. **movl src, dest (movl %1, %0)** : moves the contents of the source into destination.

5. Each operand is described by an operand constraint string followed by an expression in parentheses.
6. The **“r”** in the operand constraint string indicates that the **operand must be located in a register**.
7. The **“=”** indicates that the operand is written. Each output **operand must have “=” in its constraint**.

PART-2

1. After performing an **update** and installing **build-essential**, we install qemu with the following command: *sudo apt-get install qemu.* Then we run **make** inside the xv6 directory.
2. On running the command *make qemu,* the qemu terminal opens up and typing *ls* in the terminal gives the following output.



```
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13612
echo           2 4 12624
forktest       2 5 8052
grep           2 6 15488
init           2 7 13208
kill           2 8 12676
ln             2 9 12576
ls             2 10 14760
mkdir          2 11 12756
rm             2 12 12736
sh             2 13 23220
stressfs       2 14 13404
usertests      2 15 56336
wc             2 16 14156
zombie         2 17 12400
console        3 18 0
$ _
```

EXERCISE 2

On running si command multiple times we got the following results.

On processor reset, the processor enters real mode and sets CS to 0xf000 and IP to 0xfff0,so that execution begins at that (CS:IP) segment address.Then the BIOS jumps backward to an earlier location in the BIOS.

Now, **lidtl** command sets up an Interrupt descriptor table and **lgdtl** sets up a global descriptor table. Alongwith this it initializes various devices such as the VGA display.

```
(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    0xd241d416
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    $0x2d4e,%dx
0x0000e070 in ?? ()
```

```
(gdb) si
[f000:e076]    0xfe076: jmp    0x5575ff02
0x0000e076 in ?? ()
```

```
(gdb) si
[f000:ff00]    0xfff00: cli
0x0000ff00 in ?? ()
(gdb) si
[f000:ff01]    0xfff01: cld
0x0000ff01 in ?? ()

(gdb) si
[f000:ff02]    0xfff02: mov    %ax,%cx
0x0000ff02 in ?? ()
(gdb) si
[f000:ff05]    0xfff05: mov    $0x8f,%ax
0x0000ff05 in ?? ()
(gdb) si
[f000:ff0b]    0xfff0b: out    %al,$0x70
0x0000ff0b in ?? ()
(gdb) si
[f000:ff0d]    0xfff0d: in     $0x71,%al
0x0000ff0d in ?? ()
(gdb) si
[f000:ff0f]    0xfff0f: in     $0x92,%al
0x0000ff0f in ?? ()
(gdb) si
[f000:ff11]    0xfff11: or     $0x2,%al
0x0000ff11 in ?? ()
(gdb) si
[f000:ff13]    0xfff13: out    %al,$0x92
0x0000ff13 in ?? ()
(gdb) si
[f000:ff15]    0xfff15: mov    %cx,%ax
0x0000ff15 in ?? ()
```

```
(gdb) si
[f000:ff18]    0xfff18: lidtl  %cs:(%esi)
0x0000ff18 in ?? ()
```

```
(gdb) si
[f000:ff1e]    0xfff1e: lgdtl  lgdtl  %cs:(%esi)
0x0000ff1e in ?? ()
(gdb) si
[f000:ff24]    0xfff24: mov    %cr0,%ecx
0x0000ff24 in ?? ()
(gdb) si
[f000:ff27]    0xfff27: and    $0xffff,%cx
0x0000ff27 in ?? ()
(gdb) si
[f000:ff2e]    0xfff2e: or     $0x1,%cx
0x0000ff2e in ?? ()
(gdb) si
[f000:ff32]    0xfff32: mov    %ecx,%cr0
0x0000ff32 in ?? ()
(gdb) si
[f000:ff35]    0xfff35: ljmpw  $0xf,$0xff3d
0x0000ff35 in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0xfff3d: mov    $0x10,%ecx
0x000fff3d in ?? ()
(gdb) si
=> 0xfff42: mov    %ecx,%ds
```

EXERCISE 3

The following observations can be made after performing the given tasks:

1. On tracing into **readsect()** and **bootmain()** functions of **bootmain.c** and tracing **bootblock.asm** file for assembly code, we find that the first instruction of loop is at address **0x7d8f** while the condition for entering first time in loop is checked in the instruction at address **0x7d83** and if this condition is true then **jmp** instruction at address **0x7d8d** is executed and control enters loop at address **0x7d96**. In further iterations the loop is executed from instructions at address **0x7d8f - 0x7db4**.

2. So the end of the loop is at instruction **0x7db4**. However just before exiting the loop, the last instruction executed will be **jbe** instruction at address **0x7d94** when the condition(at address **0x7d92**) for continuing in loop fails.

```
for(; ph < eph; ph++){
    7d83:	39 f3                	cmp    %esi,%ebx
    7d85:	72 0f                	jb     7d96 <bootmain+0x5b>
  entry();
    7d87:	ff 15 18 00 01 00    	call   *0x10018
    7d8d:	eb d5                	jmp    7d64 <bootmain+0x29>
  for(; ph < eph; ph++){
    7d8f:	83 c3 20             	add    $0x20,%ebx
    7d92:	39 de                	cmp    %ebx,%esi
    7d94:	76 f1                	jbe    7d87 <bootmain+0x4c>
    pa = (uchar*)ph->paddr;
    7d96:	8b 7b 0c             	mov    0xc(%ebx),%edi
    readseg(pa, ph->filesz, ph->off);
    7d99:	ff 73 04             	pushl  0x4(%ebx)
    7d9c:	ff 73 10             	pushl  0x10(%ebx)
    7d9f:	57                   	push   %edi
    7da0:	e8 53 ff ff ff       	call   7cf8 <readseg>
    if(ph->memsz > ph->filesz)
    7da5:	8b 4b 14             	mov    0x14(%ebx),%ecx
    7da8:	8b 43 10             	mov    0x10(%ebx),%eax
    7dab:	83 c4 0c             	add    $0xc,%esp
    7dae:	39 c1                	cmp    %eax,%ecx
    7db0:	76 dd                	jbe    7d8f <bootmain+0x54>
      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    7db2:	01 c7                	add    %eax,%edi
    7db4:	29 c1                	sub    %eax,%ecx
```

The code that will run when the loop is finished will be at the instruction at address **0x7d87** in which a call to **entry()** function is made. Here we have set a breakpoint at **0x7d87** and continued execution till that instruction. The last instruction executed is **call *0x10018.**

Running **si** after this instruction makes the code to enter the kernel at address **0x0010000c** as is shown in the below output image.

<u>Answer 1</u>

The point at which the processor starts executing 32-bit code is from instruction at address **0x7c31.** i.e. **mov $0x10,%ax.**

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si 18
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c31
0x00007c2c in ?? ()
(gdb) info registers
eax            0x11     17
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x6f04   0x6f04
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7c2c   0x7c2c
eflags         0x6      [ PF ]
cs             0x0      0
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) si
The target architecture is assumed to be i386
=> 0x7c31:      mov    $0x10,%ax
0x00007c31 in ?? ()
(gdb) info registers
eax            0x11     17
ecx            0x0      0
edx            0x80     128
ebx            0x0      0
esp            0x6f04   0x6f04
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x7c31   0x7c31
eflags         0x6      [ PF ]
cs             0x8      8
ss             0x0      0
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

The CR0 register is a 32 bits long control register on the 386 processor. The **PE flag** of the CR0 register decides whether the processor is in Protected mode or Real mode. If PE is enabled the processor is in protected mode.

The following steps are performed which causes the transition from 16-bit to 32-bit mode (refer to below code which is taken **from bootasm.S** line 39-56):

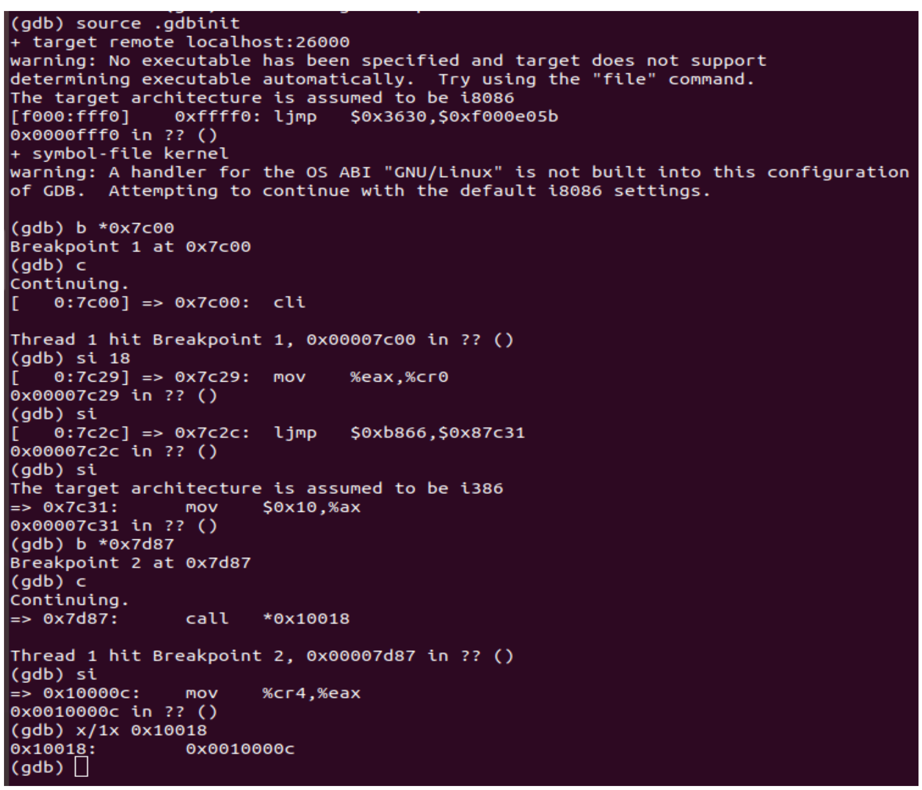
1. Setup a global descriptor table (gdt) and load it using **lgdt** command (line 42).
2. Set the **PE flag** in the CR0 register to 1 i.e. enable it (line 43-45).
3. Execute a long jump using **ljmp** instruction (line 51). After the label **start32** (line 54), the addresses are in 32-bit format. So bootloader needs a long jump to reload %cs and %eip. The segment descriptors are set up with no translation, so the mapping is identity mapping.

```
# Switch from real to protected mode.  Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# an 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
```

Answer 2

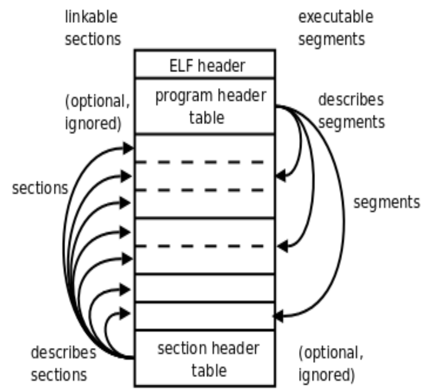
The last instruction of the bootloader that was executed is **call *0x10018** which calls the entry function and hence the bootloader transfers control to the kernel which is at address **0x0010000c**. The first instruction of the kernel it just loaded is **mov %cr4, %eax** at address **0x0010000c**. The below image shows the outcomes.



```
(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) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si 18
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(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) b *0x7d87
Breakpoint 2 at 0x7d87
(gdb) c
Continuing.
=> 0x7d87:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d87 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) x/1x 0x10018
0x10018:        0x0010000c
(gdb)
```

Answer 3

The size of the disk sector is 512 byte, as defined on line 13 of bootmain.c. When the bootloader loads the kernel, it first reads the ELF header of the kernel. The ELF header takes 4096 bytes i.e.8 sectors. This is implemented in line 28 of bootmain.c. After reading the ELF header of the kernel, the boot loader loads each segment of the kernel.



We ran command **readelf -a kernel** and we got 16 sections of the kernel. Now each section is stored on disk, aligned by 512 byte which is the actual sector size. Thus, we can find the number of sectors read by bootloader i.e. **sum(ceil(section_size)/SECTSIZE)** sectors to fetch the entire kernel. As we perform the calculation above,we got the result that 436 sectors are read to fetch the entire kernel.

The below screenshot shows the output of running the above command on the terminal from where we calculated the number of sectors read to fetch the kernel.

```
human@human-G3-3579:~/xv6-public$ readelf -a -e kernel
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x10000c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          178516 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         16
  Section header string table index: 15

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        80100000 001000 006f12 00  AX  0   0 16
  [ 2] .rodata           PROGBITS        80106f20 007f20 00101c 00   A  0   0 32
  [ 3] .data             PROGBITS        80108000 009000 002516 00  WA  0   0 4096
  [ 4] .bss              NOBITS          8010a520 00b516 00af88 00  WA  0   0 32
  [ 5] .debug_line       PROGBITS        00000000 00b516 002602 00      0   0  1
  [ 6] .debug_info       PROGBITS        00000000 00db18 0105be 00      0   0  1
  [ 7] .debug_abbrev     PROGBITS        00000000 01e0d6 00398d 00      0   0  1
  [ 8] .debug_aranges    PROGBITS        00000000 021a68 0003a8 00      0   0  8
  [ 9] .debug_str        PROGBITS        00000000 021e10 000e74 01  MS  0   0  1
  [10] .debug_loc        PROGBITS        00000000 022c84 0052fe 00      0   0  1
  [11] .debug_ranges     PROGBITS        00000000 027f82 000700 00      0   0  1
  [12] .comment          PROGBITS        00000000 028682 000029 01  MS  0   0  1
  [13] .symtab           SYMTAB          00000000 0286ac 002060 10     14  78  4
  [14] .strtab           STRTAB          00000000 02a70c 0011b0 00      0   0  1
  [15] .shstrtab         STRTAB          00000000 02b8bc 000096 00      0   0  1
```

EXERCISE 4

The output of pointer.c is as follows :-

```
kartikeya@Kartikeya-PC:~/IITG/Sem 5/CS344/Lab1$ gcc pointer.c
kartikeya@Kartikeya-PC:~/IITG/Sem 5/CS344/Lab1$ ./a.out
1: a = 0x7ffddbc466e0, b = 0x5634ceb422a0, c = 0x7ffddbc46707
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7ffddbc466e0, b = 0x7ffddbc466e4, c = 0x7ffddbc466e1
```

1. Before line 1 is printed, an array a is declared as **int a[4];**, so, **a would store the address of the first element of the array a**. The statement **int *b = malloc(16)**; would allocate a 16 byte block of memory and **b would store the address of this newly allocated memory**. The statement **int *c;** is used to declare the pointer variable c, as it is not assigned a value, the variable c **would store a garbage value**.
2. Before line 2 is printed, c is set equal to a , hence, *c would store the address of the first element of the array a*. Then the values of the array **a are set to 100,101,102 and 103.** Now, **c[0] = 200**, changes the value of **a[0] to 200** as c points to the first element of a, therefore c[0] = a[0]. So, while executing the print statement the values of the array are **200, 101, 102 and 103**.
3. Before line line 3 is printed **c[1] = 301** changes the value of **a[1] to 301**, *as c points to the first element of a*. ***(c+2)** is equivalent to **c[2]**, so ***(c+2) = 301** changes the value of **a[2] = 302**, and **3[c]** is equivalent to **c[3]**, so **c[3] = 302** changes the value of **a[3] to 302**, so finally the values in the array are **200, 300, 301, 302.**
4. Before line 4 is printed **c = c+1**, changes c to point to the second element of the array a, so ***c =400** sets **a[1] = 400.** Hence the values printed are **200, 400, 301, 302**.
5. Just after printing line 4, c points to the memory location **0x7ffddbc466e4**, the integer a[1] is stored from **0x7ffddbc466e4 - 0x7ffddbc466e7**. The statement, **c = (int *) ((char *) c + 1);** changes c to 0x7ffddbc466e5 as sizeof(char) = 1, and typecasting c into a char* and incrementing it would increase the memory location by 1 instead of 4 locations. Now *c = 500 would change contents of the memory locations from **0x7ffddbc466e5 - 0x7ffddbc466e8**, which would change the

contents of a[1] and a[2] both, as the memory locations **0x7ffddbc466e5 - 0x7ffddbc466e7** is a part of a[1] and the location **0x7ffddbc466e8** is a part of a[2]. So the values stored in a[1] and a[2] are not garbage although they seem to be corrupted.

6. As b is of type int* , and a points to the location **0x7ffddbc466e0**, b would point to **0x7ffddbc466e4** as sizeof(int) = 4. (char*)a points to the memory location **0x7ffddbc466e0** and (char*)a + 1 would be **0x7ffddbc466e1**, so finally b points to **0x7ffddbc466e1**.

PART-2: Running objdump Command

After running **objdump -h kernel** and **objdump -h bootblock.o,** the outcomes are shown below. In the second image the VMA and LMA of .text are the same i.e. load and link address of bootloader are the same.

```
humam@humam-G3-3579:~/xv6-public$ objdump -h kernel

kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00006ea2  80100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000009ec  80106ec0  00106ec0  00007ec0  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   000025e8  00000000  00000000  0000b516  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_info   0001051b  00000000  00000000  0000dafe  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_abbrev 00003946  00000000  00000000  0001e019  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_aranges 000003a8  00000000  00000000  00021960  2**3
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_str    00000e64  00000000  00000000  00021d08  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_loc    00005281  00000000  00000000  00022b6c  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_ranges 00000700  00000000  00000000  00027ded  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .comment      00000029  00000000  00000000  000284ed  2**0
                  CONTENTS, READONLY
```

```
humam@humam-G3-3579:~/xv6-public$ objdump -h bootblock.o

bootblock.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000001c0  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000bc  00007dc0  00007dc0  00000234  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000029  00000000  00000000  000002f0  2**0
                  CONTENTS, READONLY
  3 .debug_aranges 00000040  00000000  00000000  00000320  2**3
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   0000050b  00000000  00000000  00000360  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .debug_abbrev 000001e3  00000000  00000000  0000086b  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_line   0000012c  00000000  00000000  00000a4e  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_str    000001d9  00000000  00000000  00000b7a  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_loc    0000022a  00000000  00000000  00000d53  2**0
                  CONTENTS, READONLY, DEBUGGING
```

EXERCISE 5

In the Makefile, we changed the link address to **0x7c04** instead of the original link address **0x7c00**. The error which we observed was that initially the bootloader was switching from 16-bit real mode to 32-bit protected mode. However after changing the link address, the bootloader starts executing the instructions from bios part of memory again instead of entering into 32-bit protected mode as was observed in Answer 1 of Exercise 3. This happens because the last **ljmp** uses two parameters to jump to an absolute address. At this time, if the link address and load address are inconsistent, then the jump target address is also wrong.

```
(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    0xd241d416
0x0000e062 in ?? ()
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si 18
[   0:7c29] => 0x7c29:  mov    %eax,%cr0
0x00007c29 in ?? ()
(gdb) si
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87c35
0x00007c2c in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
(gdb)
```

The output of running command **objdump -f kernel** is shown below. The start address is the starting point (**0x0010000c**) at which the bootloader enters the kernel.

```
humam@humam-G3-3579:~/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) 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) 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 *0x7d87
Breakpoint 2 at 0x7d87
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d87:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d87 in ?? ()
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) 
```

After restarting the machine again and examining 8 words of memory at location 0x00100000 at the point BIOS enters bootloader and then again at point when bootloader enters kernel, we observe that:

1. At the point **BIOS enters the bootloader**, the contents at given memory location are all **zero**.
2. However at point the **bootloader enters the kernel**, the contents at given memory location are **non-zero**.
3. The reason for this is as follows. After the boot loader starts to execute, the **kernel is not yet loaded**. Thus, there are only **zero** values at the 8 words of memory in 0x00100000 . On the other hand, when the boot loader enters the kernel, **the boot loader has fully read the ELF of the kernel**. Thus, there are **hex values** in 0x00100000 and its nearby area. Hence the contents of the 8 words of memory are non-zero.

EXERCISE 7

To create the system call **int sys_wolfie(void *buf, uint size)**, we have made changes to the following files :

1. **syscall.h** :- The following flag was defined

```
#define SYS_wolfie 22
```

2. **syscall.c** :- The function prototype was declared and the function pointer was added to the array containing the system calls.

```
extern int sys_wolfie(void);

static int (*syscalls[])(void) = {
[SYS_fork]  sys_fork,
[SYS_exit]  sys_exit,
......
[SYS_wolfie]  sys_wolfie, // Here function pointer was added to array
};
```

3. **sysproc.c** :- This system call was created to copy the ASCII art of wolfie into the character buffer. if the size of the buffer is sufficient the function returns the number of bytes copied otherwise it returns -1.

```
int
sys_wolfie(char* buf,int size){
    char* wolf = "                   \t\t,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  \"\"boooooooob   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";
    argstr(0,&buf);
    argint(1,&size);
    int n = strlen(wolf);
    if(n > size)
       return -1;
    for(int i=0;i<n;++i)
       buf[i] = wolf[i];
```

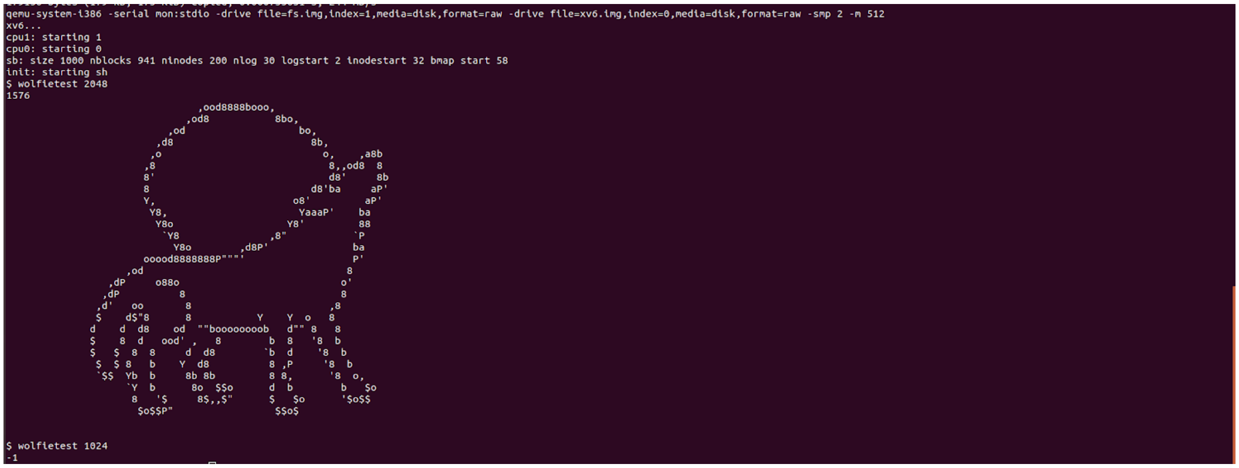
```
    return n;
}
```

EXERCISE 8

The user level application **wolfietest.c** is attached with the submission and it takes the size of the buffer as input from the command line argument. Type the following command in qemu terminal to run the file : **wolfietest 2048** where 2048 is a sample input size of buffer. If the size of the buffer provided is greater than the size of ASCII image buffer, then the number of bytes copied is printed to the qemu console along with the image of the wolf, otherwise "-1" is printed to the **qemu** console.

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

int main(int argc,char* argv[]){
    int size = atoi(argv[1]);
    char* buff = (char*)malloc(size*sizeof(char));
    int x = wolfie(buff,size);
    printf(1,"%d\n",x);

    if(x != -1){
       for(int i=0;i<x;++i){
             printf(1,"%c",buff[i]);
       }
       printf(1,"\n");
    }
    exit();
}
```

The output of running above program is shown below:



The following changes were made to the below shown files to add an interface for system call so that a user program can call it:

1. **usys.S** :- The following line was added

```
SYSCALL(wolfie)
```

2. **user.h** :- The following line was added

```
// system calls
int wolfie(char *, int);
```

3. **Makefile** :- The following changes were made to the Makefile, in order to include the program binary to fs.img.

```
UPROGS=\
    _cat\
    _echo\
    .......
    .......
    _zombie\
    _wolfietest\   # This line was added
```

```
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\
    printf.c umalloc.c wolfietest.c\   ## wolfietest.c was added here
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

The result of above steps was that the program binary was included in fs.img. The screenshot of the qemu terminal after typing **ls** command is shown below.