LAB-1 REPORT OS-344

Group ID- G35

Group Members:

SAURABH BARANWAL Roll number- 180101072 Branch- CSE
MOHAN KUMAR Roll number- 180101042 Branch- CSE
RAHUL KRISHNA Roll number- 180123035 Branch- MNC
PANKAJ KUMAR Roll number- 180123031 Branch- MNC

EX-1

```
int add_1(int x){
   int res = 0;
   int inc = 1;
   __asm(
   "movl %[input],%%eax\n"              //copying value of input1 in eax register
   "add %[input2],%%eax\n"              //add value of input2 in eax register
   "movl %%eax,%[output]\n"             //copy value of eax register in output register
   :[output] "=r" (res)                   //linking res to output register
   :[input] "r" (x),   [input2] "r" (inc)     //linking x to input register and inc
                                          //value to input2 register
   );
   return res;
}
```

Ex-2

Running multiple si commands in gdb give us the following instructions:

[f000:e05b] 0xfe05b: cmpw $0xffc8,%cs:(%esi) # compare 0xf6ac8 with the effective address (addressed by the esi plus an offset of value in register cs). Set ZF=1 if equal else ZF=0 i.e. not equal
[f000:e062] 0xfe062: jne 0xd241d416 # if != 0 i.e. ZF=0, jump to 0xfd2e1
[f000:e066] 0xfe066: xor %edx,%edx # set %edx = 0 (by taking xor with itself)
[f000:e068] 0xfe068: mov %edx,%ss # set stack segment = 0
[f000:e06a] 0xfe06a: mov $0x7000,%sp # set stack pointer = 0x7000
[f000:e070] 0xfe070: mov $0x2d4e,%dx # set %dx = 0x2d4e
[f000:e076] 0xfe076: jmp 0x5575ff02 #jump to address 0x5575ff02
[f000:ff00] 0xffff00: cli # clear interrupt flag
[f000:ff01] 0xffff01: cld # clear direction flag

Ex-3

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

The point where processor starts executing 32-bit code is at memory location 0x7c2c
After setting the breakpoint at 0x7c00, we ran si commands multiple times and at location 0x7c2c we got output:
[0:7c2c] => 0x7c2c: ljmp $0xb866,$0x87c31
The **ljmp $0xb866,$0x87c31** at 0x7c2c makes the processor start executing 32-bit code..

#In the file bootasm.S line number 51 to 54
ljmp $(SEG_KCODE<<3) , $start32
.code32 #this line tells the compiler to generate 32-bit code now
.start32

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

In the file bootmain.c, the last instruction is to call the entry point from ELF header. It is:

entry = (void(*)(void))(elf->entry);

The corresponding disassembly of this line in file bootblock.asm is:

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

This is the last instruction of the bootloader executed.

Now, on entering the kernel the first instruction loaded is the instruction stored at memory address given by string stored at 0x10018. We run the following two commands in gdb to get the instruction:

(gdb) x/1w 0x10018 #FInd the word stored at 0x10018

0x10018: 0x0010000c

(gdb) x/1i 0x0010000c **#**Find the instruction stored at 0x10000c

0x10000c: add %al,(%eax)

First instruction of the Kernel is thus: **add %al,(%eax)**

3. 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?

The bootloader calculates the number of sectors to be read by parsing the Kernel, which is in form of an ELF file. In file bootmain.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);
```

The argument pa tells the physical address for the segment and ph->filesz tells the size of the program segment. Using the above 2 fields, the bootloader can decide how many sectors to load since a sector is 512B and the kernel starts at sector 1.

EX-4:-

After running the pointer.c file we get following results;

1: a = 0x7ffd44923f70, b = 0x55f79b23c260, c = 0xf0b5ff

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 = 0x7ffd44923f70, b = 0x7ffd44923f74, c = 0x7ffd44923f71

In line 1 a is the address of array a (a == &a[0]) , b is the address of memory allocated by the malloc in heap while c is address of uninitialized pointer.

For line 2: a and c shares the same memory address, changing c[0] = 200 will change the memory location addressed by &a[0], thus changing c[0] will also reflect change in a[0]

Line 3:- In the pointer arithmetic a[i] is same as *(a + i) and that is also same as i(a) thus code is changing the memory location a[1],a[2],a[3]

Line 4:-c = c+1 => this make c pointing to c[1] thus changing *c = 400 is same as c[1] = 400 and since c and a shares same memory address hence a[1] = 400

Line 5:- &c -> 0x7ffd44923f74 and &c[1]-> 0x7ffd44923f78

Casting c to (char *) will make the address of c to 0x76 (1 byte for char) because x86 architecture is little endian.

(int*)((char *)c + 1) will increase the address by 1 Byte and parsing back to int will make c = 0x7ffd44923f5

So setting *c = 500 will set 500 in the memory location 0x7ffd44923f5 to 0x7ffd44923f9 (this memory is shared by both c and (c+1)

Thus it will change the value in both the address.

Line 6:- b is set to address of pointer a + 4 bytes for integer. C = (int*)((char*)a + 1) this will first convert a to char pointer and increment it by 1

Byte and than again converting the address to int* will set c = 0x7ffd44923f71

EX-5:-

objdump -h kernel

```
(base) mohan@mohan-Lenovo-ideapad-520-15IKB:~/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
```

As per the result of above command shown in the image the LMA(load address) of .text file is 00100000 and VMA(link address) is 80100000

objdump -d kernel | head -n 20

```
(base) mohan@mohan-Lenovo-ideapad-520-15IKB:~/xv6-public$ objdump -d kernel|head -n 20

kernel:     file format elf32-i386


Disassembly of section .text:

80100000 <multiboot_header>:
80100000:       02 b0 ad 1b 00 00       add    0x1bad(%eax),%dh
80100006:       00 00                   add    %al,(%eax)
80100008:       fe 4f 52                decb   0x52(%edi)
8010000b:       e4                      .byte 0xe4

8010000c <entry>:
8010000c:       0f 20 e0                mov    %cr4,%eax
8010000f:       83 c8 10                or     $0x10,%eax
80100012:       0f 22 e0                mov    %eax,%cr4
80100015:       b8 00 90 10 00          mov    $0x109000,%eax
8010001a:       0f 22 d8                mov    %eax,%cr3
8010001d:       0f 20 c0                mov    %cr0,%eax
80100020:       0d 00 00 01 80          or     $0x80010000,%eax
```

After tracing the file bootasm.S (boot loader) we observed that line number 51 which is below command to (transition to 32 bit protected mode)

ljmp $(SEG_KCODE<<3), $start32   #in bootasm.S

(gdb)

[0:7c2c] => 0x7c2c: ljmp $0xb866,$0x87c31   **#**in the terminal while running gdb.

Is using the absolute address and hence produces unexpected results on providing the wrong link address to the linker in the Makefile.

The program goes in infinite loop. At address [f000:e437] it jumps to [f000:e517] and again at
[f000:e520] it jumps to [f000:e423] and the process repeats.

```
[f000:e437]    0xfe437: jmp    0x6676e519
0x0000e437 in ?? ()
(gdb)
[f000:e517]    0xfe517: out    %al,(%dx)
0x0000e517 in ?? ()
(gdb)
[f000:e518]    0xfe518: mov    %bp,%si
0x0000e518 in ?? ()
(gdb)
[f000:e51b]    0xfe51b: lea    0x1(%bp),%bp
0x0000e51b in ?? ()
(gdb)
[f000:e520]    0xfe520: jmp    0x6677e425
0x0000e520 in ?? ()
(gdb)
[f000:e423]    0xfe423: mov    %cs:0x0(%di),%al
0x0000e423 in ?? ()
(gdb)
```

I changed the link address in the linker in Makefile from 0x7c00 -> 0x7000

bootblock: bootasm.S bootmain.c

	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c

	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S

	$(LD) $(LDFLAGS) -N -e start -Ttext **0x7000** -o bootblock.o bootasm.o bootmain.o

	$(OBJDUMP) -S bootblock.o > bootblock.asm

	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock

	./sign.pl bootblock

Changing this command produces the unexpected result in the QEMU terminal.

EX-6

The bootloader will load the Kernel at the LMA (Load address) 0x100000. So before the loading of the Kernel, 8 words of the memory can be anything. Once the Kernel is loaded, they will be the first 8 words of the Kernel.

The point at which BIOS enters bootloader is **0x7c00.** We set a breakpoint here and try to look the 8 words here. Output is that QEMU sets all memories to zero here.

```
Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

Second break point is set at 0x1000c (first instruction of Kernel). At this point, we again view content at 0x100000 and we find it different. The 8 instructions found, is same as found in the given disassembly in file **kernel.asm.**

```
Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x83e0200f
0x100010:       0x220f10c8      0x9000b8e0      0x220f0010      0xc0200fd8
(gdb) x/8i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0xf,%al
   0x10000d:    and    %ah,%al
   0x10000f:    or     $0x10,%eax
   0x100012:    mov    %eax,%cr4
   0x100015:    mov    $0x109000,%eax
```

EX-7

To add the system call **wolfie**, the following changes were made in each of the following files:
Syscall.c Line `extern int sys_wolfie(void);` is added and line `[SYS_wolfie]  sys_wolfie,` is appended in the array/list of syscalls.

Syscall.h: LIne: `#define SYS_wolfie 22` was added

User.h : The declaration `int wolfie(void* buf, uint size);` was added

Usys.S: Line `SYSCALL(wolfie)` was added

Sysproc.c : We defined a character string **wolfie_sample** as the ASCII art image globally. Besides that, the following function definition was written for system call **wolfie: (Description of code can be understood through comments)**

```
int
sys_wolfie(void){
  char* buf;
  int size;
//Check if arguments are taken correctly for the function call or not
  if(argint(1, &size) < 0 || argptr(0, &buf, size) < 0){
        cprintf("Arguments not taken correctly\n");
        return -2;
  }
  //Calculate size of sample wolfie image (wolfie_sample is a character string containing the ASCII art
image text) using strlen function
  int check= strlen(wolfie_sample) + 1;

  //if size of buffer is less than or equal to zero, if buffer is NULL or size of buffer is less than size of
sample wolfie image, then return a negative value, say -2
  if(size <= 0 || !buf || size < (check)){
        cprintf("Buffer size comes out to be small\n");
        return -2;
  }
  //Copy wolfie_sample into buffer using strncpy and finally return the length of the respective string in
        buffer
  int pp= strlen(strncpy(buf, wolfie_sample, size));
  return pp;
}
```

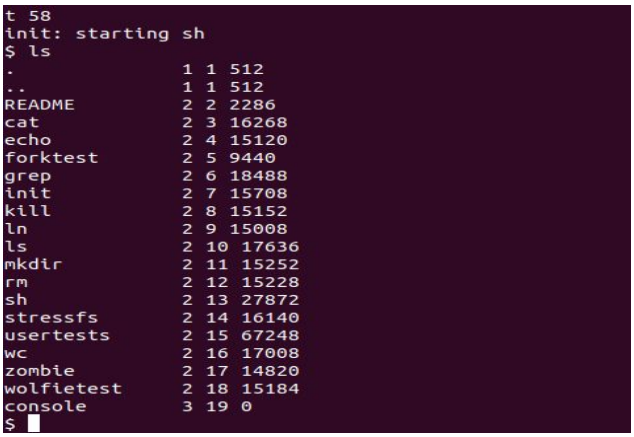
EX-8

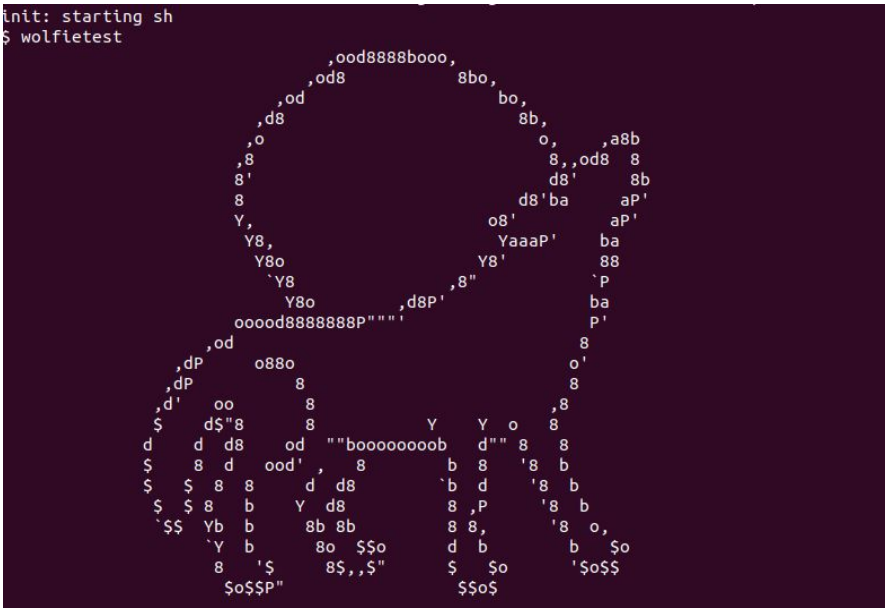
The following is the code snippet for file **wolfietest.c :**

```
#include "types.h"
#include "user.h"
#include "stat.h"
#include "syscall.h"
#define SIZE_OF_BUFFER 6001
int main(int argc, char *argv[])
```

```c
{
    //Allocate SIZE_OF_BUFFER size to buffer
    void* buffer = malloc(SIZE_OF_BUFFER);
    //Check if wolfie system call return value greater than 0, then print the content stored in buffer
    if(wolfie (buffer, SIZE_OF_BUFFER) > 0)  printf(1,"%s\n", buffer);
    //free buffer
    free(buffer);
    exit();
}
```

To add the user program to the QEMU terminal so that it displays on doing ls and runs, we do certain changes in the Makefile. The text: **_wolfietest** ls added in the UPPROGS section and text: **wolfietest.c** is added in EXTRA section. After doing this, we run **make qemu** on terminal and now QEMU terminal opens. We can type **ls** to see **wolfietest** is visible. If we type wolfietest in the QEMU terminal, it runs the code wolfietest.c and displays the necessary output





We can see that the entire ASCII art image of wolf could be copied to the user defined buffer, hence it displays the entire image on the QEMU terminal on typing **wolfietest**
