CS344 : OS Lab, Assignment - 1

Group 2

180101081 : Tejas Khairnar
180101056 : Parth Bakare
180101057 : Pooja Bhagat
180101055 : Param Aryan Singh

PART I

Exercise 1.

In extended inline assembly, we can also specify the operands. It allows us to specify the input registers, output registers and a list of clobbered registers.

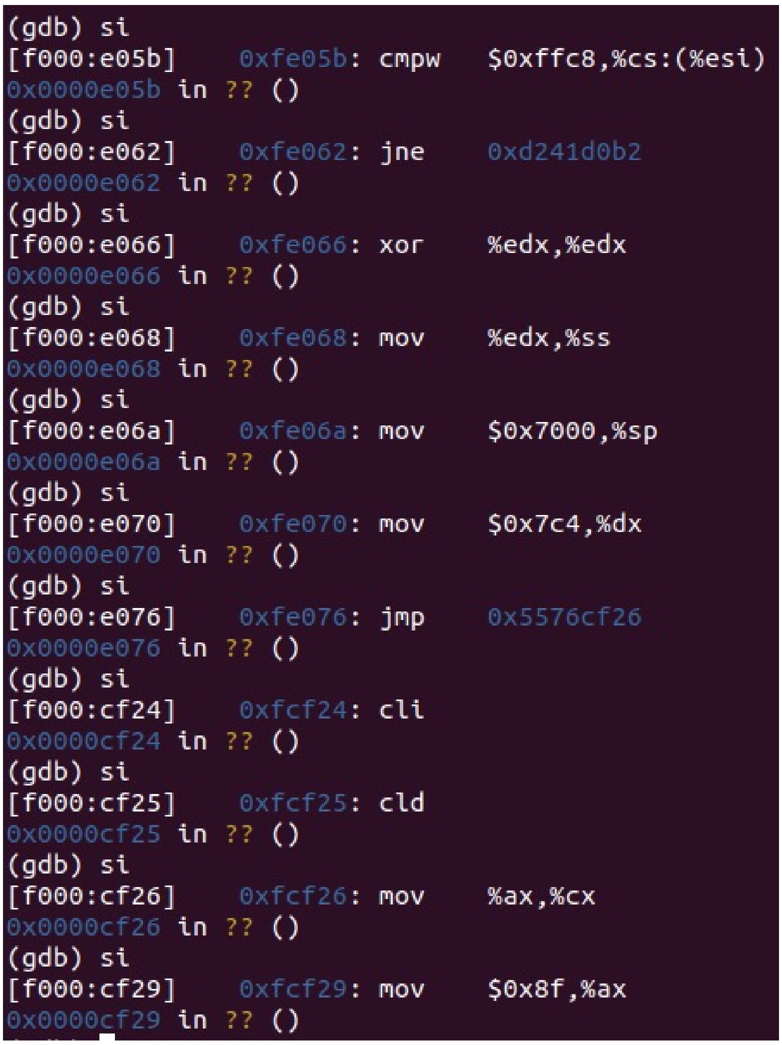
Syntax :

asm("assembly code"
: output operands *// optional*
: input operands *// optional*
: list of clobbered registers); *// optional*

__asm__("incl %%eax;" : "=a" (x) : "a" (x));

Complete edited code attached in the folder.

Exercise 2.



```
(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 ?? ()
(gdb) si
[f000:e076]    0xfe076: jmp    0x5576cf26
0x0000e076 in ?? ()
(gdb) si
[f000:cf24]    0xfcf24: cli
0x0000cf24 in ?? ()
(gdb) si
[f000:cf25]    0xfcf25: cld
0x0000cf25 in ?? ()
(gdb) si
[f000:cf26]    0xfcf26: mov    %ax,%cx
0x0000cf26 in ?? ()
(gdb) si
[f000:cf29]    0xfcf29: mov    $0x8f,%ax
0x0000cf29 in ?? ()
```

compare two operands at specified address

conditional jump, to check if result of previous cmp function is true or false

take xor of the two operands, in this case set value in edx to zero

loads value in register ss (stack segment) with value in edx i.e. 0

loads value 0x7000 in register sp

loads value 0x7c4 in register dx

jump to the address stored in the memory address given

clears interrupt flag; interrupts disabled when interrupt flag cleared.

clear direction flag (controls the left-to-right or right-to-left direction of string processing)

loads value in register ax with value in cx

loads value 0x8f in register ax

Image 2.1 : ROM BIOS instructions traced using the *si* command

PART II

Exercise 3.

Image 3.1 : Disassembly file, **bootblock.asm**

```
.code16                           # Assemble for 16-bit mode
.globl start
start:
  cli                             # BIOS enabled interrupts; disable
    7c00:       fa                      cli

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax                 # Set %ax to zero
    7c01:       31 c0                   xor    %eax,%eax
  movw    %ax,%ds                 # -> Data Segment
    7c03:       8e d8                   mov    %eax,%ds
  movw    %ax,%es                 # -> Extra Segment
    7c05:       8e c0                   mov    %eax,%es
  movw    %ax,%ss                 # -> Stack Segment
    7c07:       8e d0                   mov    %eax,%ss

00007c09 <seta20.1>:

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al                   # Wait for not busy
    7c09:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c0b:       a8 02                   test   $0x2,%al
  jnz     seta20.1
    7c0d:       75 fa                   jne    7c09 <seta20.1>

  movb    $0xd1,%al                   # 0xd1 -> port 0x64
    7c0f:       b0 d1                   mov    $0xd1,%al
  outb    %al,$0x64
    7c11:       e6 64                   out    %al,$0x64

00007c13 <seta20.2>:

seta20.2:
  inb     $0x64,%al                   # Wait for not busy
    7c13:       e4 64                   in     $0x64,%al
  testb   $0x2,%al
    7c15:       a8 02                   test   $0x2,%al
  jnz     seta20.2
    7c17:       75 fa                   jne    7c13 <seta20.2>

  movb    $0xdf,%al                   # 0xdf -> port 0x60
    7c19:       b0 df                   mov    $0xdf,%al
  outb    %al,$0x60
    7c1b:       e6 60                   out    %al,$0x60
```

Image 3.2 : Source code, **bootasm.s**

```
  cli                             # BIOS enabled interrupts; disable

  # Zero data segment registers DS, ES, and SS.
  xorw    %ax,%ax                 # Set %ax to zero
  movw    %ax,%ds                 # -> Data Segment
  movw    %ax,%es                 # -> Extra Segment
  movw    %ax,%ss                 # -> Stack Segment

  # Physical address line A20 is tied to zero so that the first PCs
  # with 2 MB would run software that assumed 1 MB.  Undo that.
seta20.1:
  inb     $0x64,%al                   # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.1

  movb    $0xd1,%al                   # 0xd1 -> port 0x64
  outb    %al,$0x64

seta20.2:
  inb     $0x64,%al                   # Wait for not busy
  testb   $0x2,%al
  jnz     seta20.2

  movb    $0xdf,%al                   # 0xdf -> port 0x60
  outb    %al,$0x60
```

Image 3.3 : GDB

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/15i $eip
=> 0x7c00:      cli
   0x7c01:      xor    %eax,%eax
   0x7c03:      mov    %eax,%ds
   0x7c05:      mov    %eax,%es
   0x7c07:      mov    %eax,%ss
   0x7c09:      in     $0x64,%al
   0x7c0b:      test   $0x2,%al
   0x7c0d:      jne    0x7c09
   0x7c0f:      mov    $0xd1,%al
   0x7c11:      out    %al,$0x64
   0x7c13:      in     $0x64,%al
   0x7c15:      test   $0x2,%al
   0x7c17:      jne    0x7c13
   0x7c19:      mov    $0xdf,%al
   0x7c1b:      out    %al,$0x60
```

Image 3.3 shows how the breakpoint was set at the address *0x7c00* (where the boot sector was loaded). Using the *x/I* GDB command, we disassembled the next 15 instructions. Images 3.2 and 3.1 show the source code in **bootasm.s** and the disassembled instructions in **bootblock.asm**, respectively. All 3 images show the first 15 instructions starting at *0x7c00*. Comparing the 3 images, we see that the first 15 instructions are identical in all of them, except a few differences in how some keywords are written.

```
// 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);
```

Image 3.4 : readsect( ) in **bootmain.c**

```
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>
}
```

Image 3.5 : readsect( ) in **bootblock.asm**

Image 3.6 : Reading remaining sectors of the kernel, **bootblock.asm**

```
315   for(; ph < eph; ph++){
316     7d8d:       39 f3                   cmp    %esi,%ebx
317     7d8f:       72 15                   jb     7da6 <bootmain+0x5d>
318   entry();
319     7d91:       ff 15 18 00 01 00       call   *0x10018
320 }
321     7d97:       8d 65 f4                lea    -0xc(%ebp),%esp
322     7d9a:       5b                      pop    %ebx
323     7d9b:       5e                      pop    %esi
324     7d9c:       5f                      pop    %edi
325     7d9d:       5d                      pop    %ebp
326     7d9e:       c3                      ret
327   for(; ph < eph; ph++){
328     7d9f:       83 c3 20                add    $0x20,%ebx
329     7da2:       39 de                   cmp    %ebx,%esi
330     7da4:       76 eb                   jbe    7d91 <bootmain+0x48>
331     pa = (uchar*)ph->paddr;
332     7da6:       8b 7b 0c                mov    0xc(%ebx),%edi
333     readseg(pa, ph->filesz, ph->off);
334     7da9:       83 ec 04                sub    $0x4,%esp
335     7dac:       ff 73 04                pushl  0x4(%ebx)
336     7daf:       ff 73 10                pushl  0x10(%ebx)
337     7db2:       57                      push   %edi
338     7db3:       e8 44 ff ff ff          call   7cfc <readseg>
339     if(ph->memsz > ph->filesz)
340     7db8:       8b 4b 14                mov    0x14(%ebx),%ecx
341     7dbb:       8b 43 10                mov    0x10(%ebx),%eax
342     7dbe:       83 c4 10                add    $0x10,%esp
343     7dc1:       39 c1                   cmp    %eax,%ecx
344     7dc3:       76 da                   jbe    7d9f <bootmain+0x56>
345       stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
346     7dc5:       01 c7                   add    %eax,%edi
347     7dc7:       29 c1                   sub    %eax,%ecx
348 }
```

The instructions from line 327 to line 348 read the remaining sectors of the kernel from the disk. When the loop is finished, the instruction in line 319 *call *0x10018* is executed. We set a breakpoint at the address *0x7d91* using GDB, and continue until we reach that breakpoint.

```
39   # Switch from real to protected mode.  Use a bootstrap GDT that makes
40   # virtual addresses map directly to physical addresses so that the
41   # effective memory map doesn't change during the transition.
42   lgdt    gdtdesc
43   movl    %cr0, %eax
44   orl     $CR0_PE, %eax
45   movl    %eax, %cr0
46
47 //PAGEBREAK!
48   # Complete the transition to 32-bit protected mode by using a long jmp
49   # to reload %cs and %eip.  The segment descriptors are set up with no
50   # translation, so that the mapping is still the identity mapping.
51   ljmp    $(SEG_KCODE<<3), $start32
```

Image 3.7 : Transition from 16-bit mode to 32-bit mode, **bootasm.s**

This section in **bootasm.s** switches the processor from 16-bit mode to 32-bit mode. The instruction in line 51 causes this switch. All the instructions unto this point were executed in 16-bit mode, and hereafter all instructions will be executed in 32-bit mode.

```
(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) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
```

Image 3.8 : Final boot loader instruction and first kernel instruction

*0x7d91: call *0x10018* —— last boot loader instruction executed
0x10000c: mov %cr4, %eax —— first kernel instruction loaded

```
  // 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);
  }
```

Image 3.9 : Reading remaining sectors of the kernel, **bootblock.asm**

As shown in image 3.9, the boot loader runs a loop starting from *ph* to *eph* to load the kernel. Both these values are obtained from the ELF header. *elf->phnum* determines the size of the loop.

Exercise 4.

Image 4.1 : *objdump -h* ***bootmain.o***

```
parthbakare@parthbakare-VirtualBox:~/xv6-public$ objdump -h bootmain.o

bootmain.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000155  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000189  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000189  2**0
                  ALLOC
  3 .debug_info   000005ac  00000000  00000000  00000189  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  4 .debug_abbrev 00000218  00000000  00000000  00000735  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  5 .debug_loc    000002bb  00000000  00000000  0000094d  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  6 .debug_aranges 00000020  00000000  00000000  00000c08  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  7 .debug_ranges 00000078  00000000  00000000  00000c28  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  8 .debug_line   0000023f  00000000  00000000  00000ca0  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  9 .debug_str    00000221  00000000  00000000  00000edf  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .comment      00000025  00000000  00000000  00001100  2**0
                  CONTENTS, READONLY
 11 .note.GNU-stack 00000000  00000000  00000000  00001125  2**0
                  CONTENTS, READONLY
 12 .note.gnu.property 0000001c  00000000  00000000  00001128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 13 .eh_frame     000000b0  00000000  00000000  00001144  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

Image 4.2 : *objdump -h kernel*

```
parthbakare@parthbakare-VirtualBox:~/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    00000eb0  00000000  00000000  00028718  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  9 .debug_loc    0000681e  00000000  00000000  000295c8  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 10 .debug_ranges 00000d08  00000000  00000000  0002fde6  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
 11 .comment      00000024  00000000  00000000  00030aee  2**0
                  CONTENTS, READONLY
```

Size : Size of the section
VMA : Link address of the section
LMA : Load address of the section

Link address : Memory address from where the section begins to execute
Load address : Memory address where the section should be loaded

Exercise 5.

```
39   # Switch from real to protected mode.  Use a bootstrap GDT that makes
40   # virtual addresses map directly to physical addresses so that the
41   # effective memory map doesn't change during the transition.
42   lgdt    gdtdesc
43   movl    %cr0, %eax
44   orl     $CR0_PE, %eax
45   movl    %eax, %cr0
46
47 //PAGEBREAK!
48   # Complete the transition to 32-bit protected mode by using a long jmp
49   # to reload %cs and %eip.  The segment descriptors are set up with no
50   # translation, so that the mapping is still the identity mapping.
51   ljmp    $(SEG_KCODE<<3), $start32
```

Image 5.1 : Transition from 16-bit mode to 32-bit mode, **bootasm.s**

The instruction in line 51 will be the first to break if the provided link address is wrong.

Image 5.2 : Correct link address

```
[   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) 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 ?? ()
(gdb) si
=> 0x7c41:      mov    %eax,%gs
0x00007c41 in ?? ()
(gdb) si
=> 0x7c43:      mov    $0x7c00,%esp
0x00007c43 in ?? ()
(gdb) si
=> 0x7c48:      call   0x7d49
0x00007c48 in ?? ()
(gdb)
```

Image 5.3 : Wrong link address

```
[   0:7c2c] => 0x7c2c:  ljmp   $0xb866,$0x87d31
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 ?? ()
(gdb) si
[f000:d0b8]    0xfd0b8: mov    %eax,%ds
0x0000d0b8 in ?? ()
(gdb) si
[f000:d0ba]    0xfd0ba: mov    %eax,%ss
0x0000d0ba in ?? ()
(gdb) si
[f000:d0bc]    0xfd0bc: mov    $0xf898,%sp
0x0000d0bc in ?? ()
(gdb) si
[f000:d0c2]    0xfd0c2: jmp    0x5476ca07
0x0000d0c2 in ?? ()
(gdb)
```

The correct link address is *0x7c00*. We had changed it to *0x7d00*, and then used *make qemu* (after *make clean*) to reload the boot loader. Then we used GDB to compare the instructions executed in both the correct and wrong versions. Before the instruction *ljmp $0xb866, $0x87c31* (line 51 in **bootasm.s**), the same instructions were executed in both the versions. This instruction was executed incorrectly in the wrong version, and thereafter, all instructions in the wrong version were different from those in the correct version.

Image 5.4 : *objdump -f kernel*   entry point address : *0x0010000c*

```
parthbakare@parthbakare-VirtualBox:~/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.

Image 6.1 : *x/8x 0x00100000*, at *0x7c00* and at *0x7d91*

```
(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 *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
```

The boot loader loads the kernel into the main memory starting from the address 0x0010000. Before the boot loader starts running, there is no useful data at this location and hence when we inspect the given address at the first breakpoint (beginning of the boot loader), all the words are 0s.

However, when we look into this address at the second breakpoint, we see that some useful data has been written into these locations. This is because the second breakpoint is just at the end of the boot loader, and the kernel has been fully loaded into the main memory starting from the given address.

PART III

Exercise 7.

The following files were edited :

1) **sysproc.c** : Implemented a function *int sys_wolfie(void)*. If the buffer is too small, it returns a negative value i.e -1. If the call succeeds, it returns the number of bytes copied to the buffer.
2) **syscall.h** : Defines the position of the system call vector that connected to our implementation.
3) **syscall.c** : *extern int sys_wolfie(void);*. This external function is visible to the whole program. It connects the shell and the kernel, and the system call function was added to the system call vector at the position defined in **syscall.h**.
4) **usys.S** : Creates a user level system call definition for the system call *sys_wolfie*. Used this to connect the user's call to system call function.
5) **user.h** : Includes the system call which copies the ASCII image of a wolf picture i.e. *int wolfie(void *buf, uint size);* in the user header file.

The following file was created :

1) **wolfietest.c** : A C code which includes *user, types* and *stat* header files and prints the image of the wolfie on the console if the size of buffer is greater than size of the image, or else prints a error message.

Exercise 8.

1) **Makefile** was edited : **Makefile** needs to be edited before our program **wolfietest.c** is available for xv6 source code for compilation. We made only one change in **Makefile** i.e included *_wolfietest* in the *USER PROGRAMS (UPROGS)* section in the **Makefile**.

2) We executed the following commands in the terminal :

```
tejas@tejas-XPS-13-9380:~$ cd xv6-public
tejas@tejas-XPS-13-9380:~/xv6-public$ make clean
tejas@tejas-XPS-13-9380:~/xv6-public$ make
tejas@tejas-XPS-13-9380:~/xv6-public$ make qemu
```

Image 8.1 : Reloading the QEMU terminal

3) After entering the xv6 shell's command prompt, we checked the contents of **fs.img** by using *ls* and then executed *wolfietest* to get the wolfie image from the kernel to print it in the console.

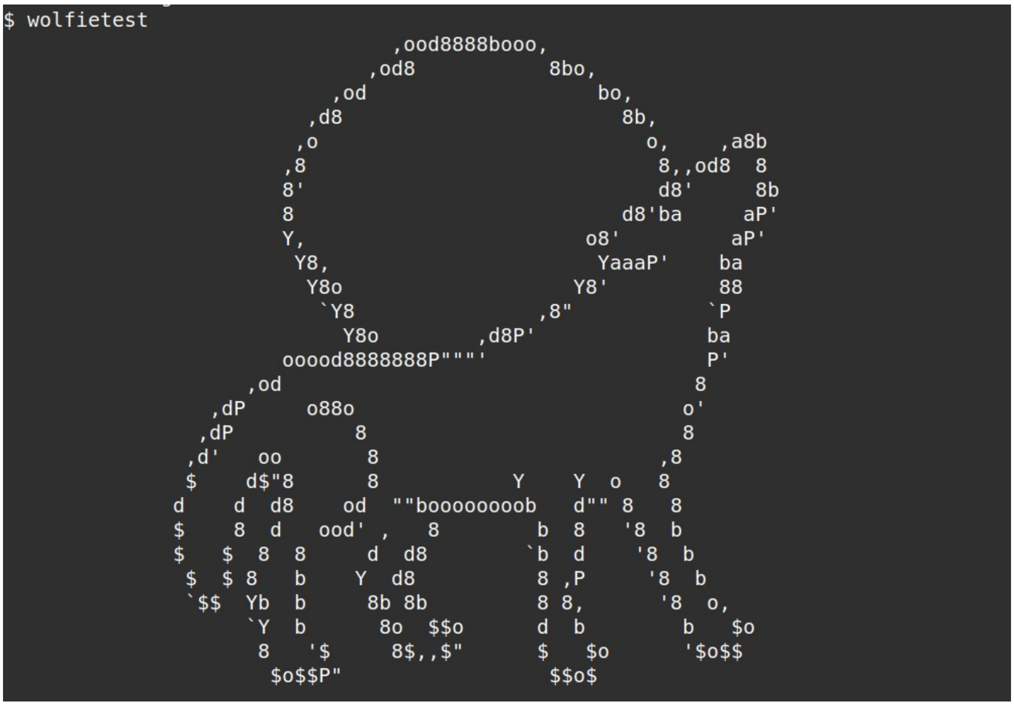


Image 8.2 : Printing wolfie image to the console