CS344 (OS LAB) ASSIGNMENT 1
GROUP G15

Name	Rahul Mala	Aryan Chauhan	Kotkar Anket Sanjay	Ritik Mandloi
Roll. No.	180101062	180101012	180101037	180101066

Question 1:

```c
// Simple inline assembly example
//
#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("add $1,%0":"=r"(x) : "r"(x));
    printf("Hello x = %d after increment\n", x);


    if(x == 2){
        printf("OK\n");
    }else{
        printf("ERROR\n");
    }
}
```

Question 2: Use of **si** command after starting the BIOS.

```
(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)
```

Showing 10 consecutive words and 10 consecutive instructions using **x/Nx** and **x/Ni** command after using a breakpoint at the address **0x7c00**.

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/10x
0x7c01: 0xd88ec031  0xd08ec08e  0x02a864e4  0xd1b0fa75
0x7c11: 0x64e464e6  0xfa7502a8  0x60e6dfb0  0x7816010f
0x7c21: 0xc0200f7c  0x01c88366
(gdb) x/10i 0x7c00
=> 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
(gdb)
```

Question 3: In the following images, the comparison between corresponding lines can be observed. E.g. The instruction: xorw %ax, %ax in BootASM.S correspond to instruction at 0x7c01 which can be seen in BootBlock.asm as well as in terminal as the first instruction after 0x7c00.

GDB **BootBlock.asm**

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/i
   0x7c01:  xor    %eax,%eax
(gdb) x/i
   0x7c03:  mov    %eax,%ds
(gdb) x/i
   0x7c05:  mov    %eax,%es
(gdb) x/i
   0x7c07:  mov    %eax,%ss
(gdb) x/i
   0x7c09:  in     $0x64,%al
(gdb) x/i
   0x7c0b:  test   $0x2,%al
(gdb)
```

```
00007c00 <start>:
# with %cs=0 %ip=7c00.

.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
```

GDB **Bootasm.s**

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/i
   0x7c01:  xor    %eax,%eax
(gdb) x/i
   0x7c03:  mov    %eax,%ds
(gdb) x/i
   0x7c05:  mov    %eax,%es
(gdb) x/i
   0x7c07:  mov    %eax,%ss
(gdb) x/i
   0x7c09:  in     $0x64,%al
(gdb) x/i
   0x7c0b:  test   $0x2,%al
(gdb)
```

```
.code16                     # Assemble for 16-bit mode
.globl start
start:
  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
```

Assembly language corresponding to readsect() function in bootmain.c

```
void
readsect(void *dst, uint offset)
{
    7c90:   55                      push   %ebp
    7c91:   89 e5                   mov    %esp,%ebp
    7c93:   57                      push   %edi
    7c94:   53                      push   %ebx
    7c95:   8b 5d 0c                mov    0xc(%ebp),%ebx
  // Issue command.
  waitdisk();
    7c98:   e8 e1 ff ff ff          call   7c7e <waitdisk>
}

static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
    7c9d:   b8 01 00 00 00          mov    $0x1,%eax
    7ca2:   ba f2 01 00 00          mov    $0x1f2,%edx
    7ca7:   ee                      out    %al,(%dx)
    7ca8:   ba f3 01 00 00          mov    $0x1f3,%edx
    7cad:   89 d8                   mov    %ebx,%eax
    7caf:   ee                      out    %al,(%dx)
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
    7cb0:   89 d8                   mov    %ebx,%eax
    7cb2:   c1 e8 08                shr    $0x8,%eax
    7cb5:   ba f4 01 00 00          mov    $0x1f4,%edx
    7cba:   ee                      out    %al,(%dx)
  outb(0x1F5, offset >> 16);
    7cbb:   89 d8                   mov    %ebx,%eax
    7cbd:   c1 e8 10                shr    $0x10,%eax
    7cc0:   ba f5 01 00 00          mov    $0x1f5,%edx
    7cc5:   ee                      out    %al,(%dx)
  outb(0x1F6, (offset >> 24) | 0xE0);
    7cc6:   89 d8                   mov    %ebx,%eax
    7cc8:   c1 e8 18                shr    $0x18,%eax
    7ccb:   83 c8 e0                or     $0xffffffe0,%eax
    7cce:   ba f6 01 00 00          mov    $0x1f6,%edx
    7cd3:   ee                      out    %al,(%dx)
    7cd4:   b8 20 00 00 00          mov    $0x20,%eax
    7cd9:   ba f7 01 00 00          mov    $0x1f7,%edx
    7cde:   ee                      out    %al,(%dx)
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // Read data.
  waitdisk();
    7cdf:   e8 9a ff ff ff          call   7c7e <waitdisk>
  asm volatile("cld; rep insl" :
    7ce4:   8b 7d 08                mov    0x8(%ebp),%edi
    7ce7:   b9 80 00 00 00          mov    $0x80,%ecx
    7cec:   ba f0 01 00 00          mov    $0x1f0,%edx
    7cf1:   fc                      cld
    7cf2:   f3 6d                   rep insl (%dx),%es:(%edi)
  insl(0x1F0, dst, SECTSIZE/4);
}
    7cf4:   5b                      pop    %ebx
    7cf5:   5f                      pop    %edi
    7cf6:   5d                      pop    %ebp
    7cf7:   c3                      ret
```

```
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);
}
```

```
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>
```

a) First Instruction in 32 bit protected mode

```
.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
    7c31:   66 b8 10 00             mov    $0x10,%ax
```

What exactly causes this jump:

```
  lgdt    gdtdesc
    7c1d:   0f 01 16              lgdtl  (%esi)
    7c20:   78 7c                 js     7c9e <readsect+0xe>
  movl    %cr0, %eax
    7c22:   0f 20 c0              mov    %cr0,%eax
  orl     $CR0_PE, %eax
    7c25:   66 83 c8 01           or     $0x1,%ax
  movl    %eax, %cr0
    7c29:   0f 22 c0              mov    %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
    7c2c:   ea                    .byte 0xea
    7c2d:   31 7c 08 00           xor    %edi,0x0(%eax,%ecx,1)
```

b)Last executed C code:

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

Its assembly code :

```
7d87:    ff 15 18 00 01 00          call   *0x10018
7d8d:    eb d5                      jmp    7d64 <bootmain+0x29>
```

First instruction of Kernel loaded:

```
0x10000c:    mov    %cr4,%eax   present at 0x0010000c location.
```

c)

C code that determines the number of segments to be read in order to load the whole kernel from the disk.
The boot loader reads the number of the program headers in the ELF header and loads them all.
The **elf->phnum** attribute is the location where the number of segments to be read can be found. The for loop in bootmain.c, following the below code, reads through those many segments of the disk to load the complete kernel in the memory.

```
  readseg((uchar*)elf, 4096, 0);

  // Is this an ELF executable?
  if(elf->magic != ELF_MAGIC)
    return;  // let bootasm.S handle error

  // Load each program segment (ignores ph flags).
  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
  eph = ph + elf->phnum;
```

Question 4:

Full list of the names, sizes, and link addresses of all the sections in the Kernel executable.

```
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    00000e6b  00000000  00000000  00021d08  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_loc    00005281  00000000  00000000  00022b73  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_ranges 00000700  00000000  00000000  00027df4  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .comment      00000029  00000000  00000000  000284f4  2**0
```

Full list of the names, sizes, and link addresses of all the sections in the BootLoader.

```
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    000001e0  00000000  00000000  00000b7a  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_loc    0000022a  00000000  00000000  00000d5a  2**0
                  CONTENTS, READONLY, DEBUGGING
```

Question 5:

After changing the Link address of the bootloader the first instruction that gets wrongly is shown.It is the point where real mode(16 bit) gets converted to protected mode (32 bit).

```
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 0x7D00 -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
```

(**Changes done in the makefile**)
The red line shows the link address being changed to 0x7D00.

```
WRONG ONE::
  ljmp    $(SEG_KCODE<<3), $start32
    7d2c:   ea                      .byte 0xea
    7d2d:   31 7d 08                xor    %edi,0x8(%ebp)
    ...

CORRECT:

ljmp    $(SEG_KCODE<<3), $start32
7c2c:   ea                      .byte 0xea
7c2d:   31 7c 08 00             xor    %edi,0x0(%eax,%ecx,1)
```

objdump -f kernel: Shows the info about architecture and start address of kernel.

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

Question 6:

8 words of memory at address 0x00100000 at the point the BIOS enters the boot loader and when boot loader enters the kernel

```
(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)
```

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

What is there at the second breakpoint ?

=> The second breakpoint is at the memory address where the kernel is called which in our case is 0x7d87 (**We concluded that this point is system architecture dependent as I got this address as 0x7d87 but my team mate got this point as 0x7d91**).

Why are they different ?

=> The address 0x00100000 is actually the start address of the **.text section** of the kernel executable. Hence when the kernel gets loaded into the memory the words present at that location changes. That's why the words present later are different from the ones that are present when bootloader is loaded.

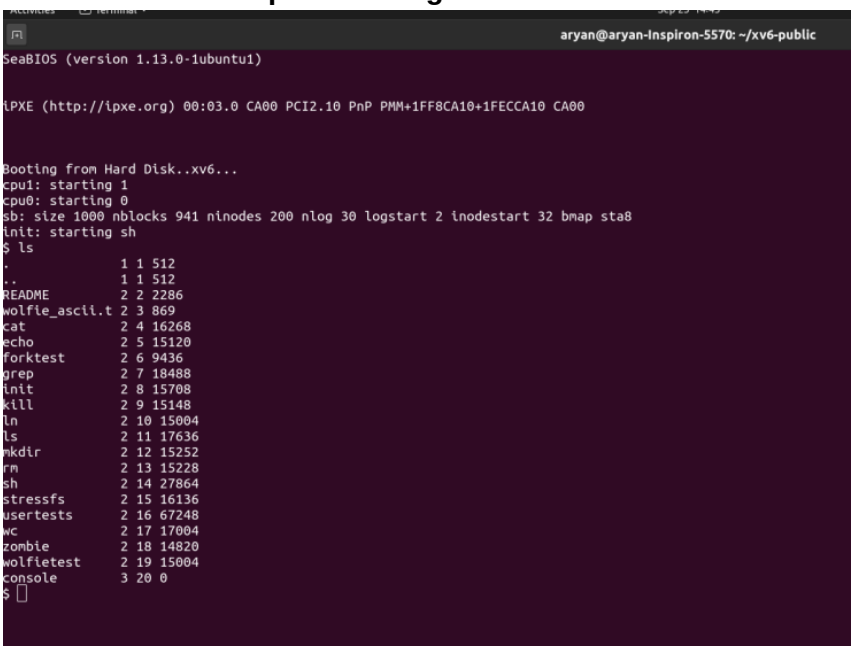
Question 7: This is the system call function we added in **sysfile.c**. The code first stores the ascii image in a string(char *) and then assigns it to the buffer provided by the user. The function returns -1 if the buffer size allocated is smaller than the size of the ascii art image. (**RELEVANT COMMENTS ARE ADDED IN THE CODE**)

```
int sys_wolfie(void)
{
  int buffer_size;
  char *buffer;
  char *asciiart = "                      .\n"
                   "                     / V\ \n"
                   "                   / `   /\n"
                   "                  <<    |\n"
                   "                  /     |\n"
                   "                /       |\n"
                   "              /         |\n"
                   "            /    \  \ /\n"
                   "           (       ) | |\n"
                   "  __|   /_  | |\n"
                   "<__\__)\__)\n";
  int sizeofart = 0;
  while (ascii_art[sizeofart] != '\0')
    sizeofart++;
  if (argint(1, &buffer_size) < 0)
    return -1;
  if (argptr(0, &buffer, buffer_size) < 0)
    return -1;
  if (sizeofart > buffer_size)
    return -1;
  for (int i = 0; i < sizeofart; i++)
  {
    buffer[i] = ascii_art[i];
  }
  return sizeofart;
}
```

Question 8:

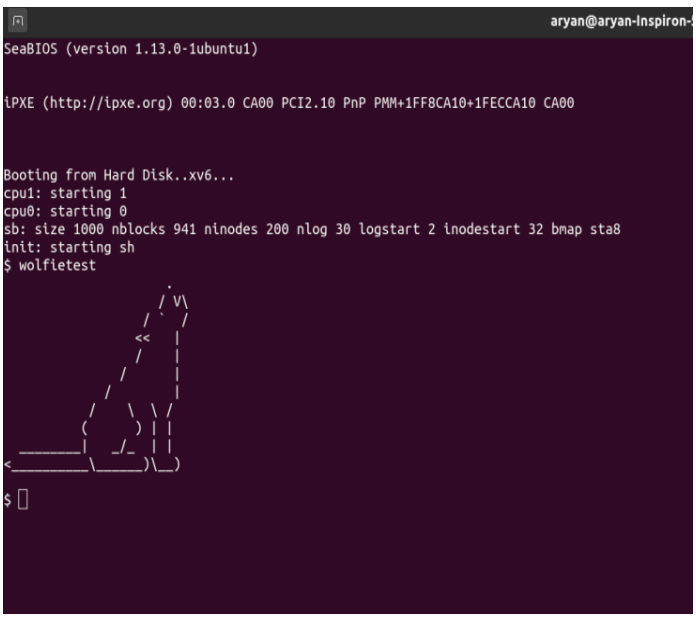
```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main()
{
    //sets buffer to the address of the newly alloted memory
    char *buffer = malloc(10000);
    //system call to wolfie
    if (wolfie(buffer,10000)!=-1)
        //prints the buffer
        printf(1,"%s\n",buffer);
    else printf(1,"Unable to print");
    exit();
}
```

Wolfietest shows up after using ls command



Working of wolfietest:

If buffer size is large enough



If Buffer size is small