CS-344 | OS LAB | ASSIGNMENT-1 | GROUP-5 |
DRISTIRON SAIKIA, KRISHNA PANDE, NIKUNJ HEDA, NIYATI CHAUDHARY |
180101022, 180101038, 180101049, 18101053

EXERCISE-1:

This is the final code of ex.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
 asm("inc %%eax;": "=a"(x):  "a"(x) );
 // the value of x by 1 using in-line assembly
 //
 printf("Hello x = %d after increment\n", x);
 if(x == 2){
 printf("OK\n");
 }
 else{
 printf("ERROR\n");
 }
}
```

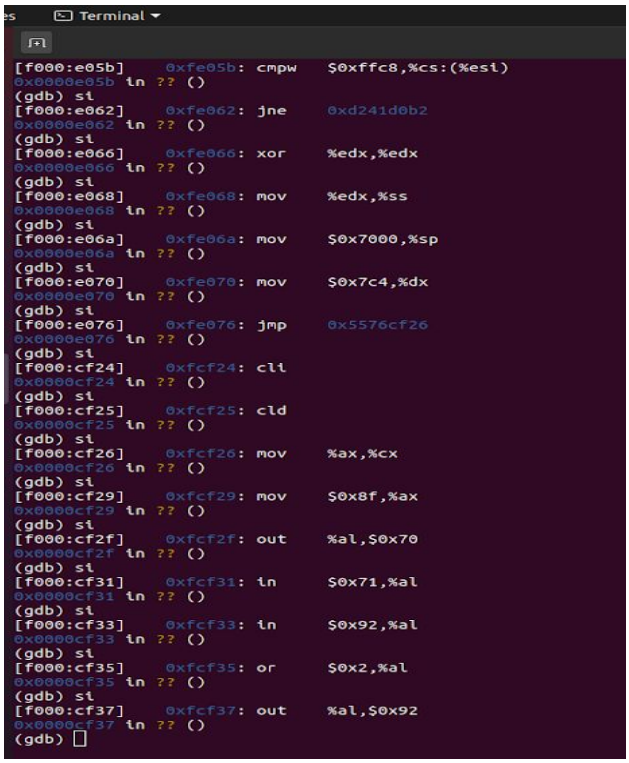
We had to write a line of asm inline code that increments the value of x by 1.

Output:-

```
PS D:\> ./a.exe
Hello x = 1
Hello x = 2 after increment
OK
```

EXERCISE-2:

Initial instructions of the BIOS:-



Explanation:-

1. The first instruction(cmpw) compares the contents of the memory address 0xffc8 with the program counter value, and sets/resets zero flag accordingly.
2. Second instruction (jne) is a conditional jump if the value of zero flag is reset.
3. The third is a xor instruction that performs xor of two 32 bit registers.
4. Fourth is a move instruction that moves the content of the stack segment register into a 32 bit register.
5. Fifth instruction moves the content of the stack pointer register into memory location 0x7000.
6. Sixth instruction moves the content of the lower half of the 32-bit register which contains the stack segment register to memory location 0x7c4.
7. Seventh instruction transfers the control to the address 0x5576cf26.
8. The next two instructions clears the two flags: interrupt and direction respectively.

EXERCISE-3:

1) This is **bootblock.asm**

```
58  # Switch from real to protected mode.  Use a bootstrap GDT that makes
59  # virtual addresses map directly to physical addresses so that the
60  # effective memory map doesn't change during the transition.
61  lgdt    gdtdesc
62    7c1d:       0f 01 16                lgdtl  (%esi)
63    7c20:       78 7c                   js     7c9e <readsect+0xe>
64  movl    %cr0, %eax
65    7c22:       0f 20 c0                mov    %cr0,%eax
66  orl     $CR0_PE, %eax
67    7c25:       66 83 c8 01             or     $0x1,%ax
68  movl    %eax, %cr0
69    7c29:       0f 22 c0                mov    %eax,%cr0
70
71 //PAGEBREAK!
72  # Complete the transition to 32-bit protected mode by using a long jmp
73  # to reload %cs and %eip.  The segment descriptors are set up with no
74  # translation, so that the mapping is still the identity mapping.
75  ljmp    $(SEG_KCODE<<3), $start32
76    7c2c:       ea                      .byte 0xea
77    7c2d:       31 7c 08 00             xor    %edi,0x0(%eax,%ecx,1)
78
79 00007c31 <start32>:
```

```
(gdb) x/6i 0x7c1d
   0x7c1d:      lgdtl  (%esi)
   0x7c20:      js     0x7c9e
   0x7c22:      mov    %cr0,%eax
   0x7c25:      or     $0x1,%ax
   0x7c29:      mov    %eax,%cr0
   0x7c2c:      ljmp   $0xb866,$0x87c31
```

- This loads the GDT: `lgdt gdtdesc`
  Where gdtdesc is a region in memory that stores the content of what the GDT should load. The Bootstrap GDT makes virtual addresses map directly to physical addresses so the effective memory map doesn't change during the transition
- Set the protected mode enable flag by some instructions.
- Then perform the jump to load CS segment register properly(0x7c2c ljmp instruction). Rest segment registers are loaded regularly.

And thus at that point(00007c31) does the processor start executing 32-bit code.

2) Last executed instruction is:

Bootmain.c:-

```
43
44  // Call the entry point from the ELF header.
45  // Does not return!
46  entry = (void(*)(void))(elf->entry);
47  entry();
48 }
```

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 }
```

```
(gdb) x/1w 0x10018
0x10018:        0x0010000c
(gdb) si
=> 0x10000c:    mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) x/1i 0x0010000c
=> 0x10000c:    mov    %cr4,%eax
```

Notice that elf has been copied starting from address 0x10000, and 0x10018 is an offset of 24 bytes from the beginning of where this struct is in memory. If we look at the definition of `struct Elf`, we see that `entry` is at offset (32+8*12+16+16+32)/8 = 24.so if we use gdb to see what's stored in there, we observe 0x001000c is the next instruction and 1st instruction of kernel.asm.

First kernel instruction is 0x10000c: mov %cr4,%eax.

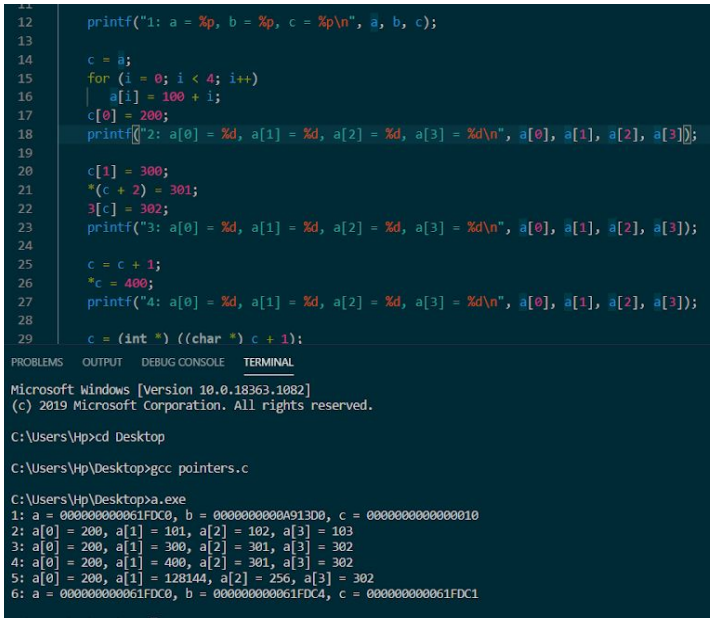
```
13 8010000c <entry>:
14
15 # Entering xv6 on boot processor, with paging off.
16 .globl entry
17 entry:
18   # Turn on page size extension for 4Mbyte pages
19   movl    %cr4, %eax
20 8010000c:       0f 20 e0                mov    %cr4,%eax
```

3) The kernel itself is an ELF file so what the bootloader does is it parses the ELF format to see how many sectors there are.
In this part of Bootmain.c :-

```
34  // Load each program segment (ignores ph flags).
35  ph = (struct proghdr*)((uchar*)elf + elf->phoff);
36  eph = ph + elf->phnum;
37  for(; ph < eph; ph++){
38    pa = (uchar*)ph->paddr;
39    readseg(pa, ph->filesz, ph->off);
40    if(ph->memsz > ph->filesz)
41      stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
42  }
```

EXERCISE-4:

The following code was run on a 64 bit system:
Executing the given C code, we get the following output



```
    printf("1: a = %p, b = %p, c = %p\n", a, b, c);

    c = a;
    for (i = 0; i < 4; i++)
      a[i] = 100 + i;
    c[0] = 200;
    printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", a[0], a[1], a[2], a[3]);

    c[1] = 300;
    *(c + 2) = 301;
    3[c] = 302;
    printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", a[0], a[1], a[2], a[3]);

    c = c + 1;
    *c = 400;
    printf("4: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n", a[0], a[1], a[2], a[3]);

    c = (int *) ((char *) c + 1);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Hp>cd Desktop

C:\Users\Hp\Desktop>gcc pointers.c

C:\Users\Hp\Desktop>a.exe
1: a = 000000000061FDC0, b = 0000000000A913D0, c = 0000000000000010
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 = 000000000061FDC0, b = 000000000061FDC4, c = 000000000061FDC1
```

`

1: a = 000000000061FDC0, b = 0000000000A913D0, c = 0000000000000010
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 = 000000000061FDC0, b = 000000000061FDC4, c = 000000000061FDC1

`

Line 1: a stores the address of the first element of an array that is stored in the stack memory. b points to a memory address in the heap memory, since b refers to dynamically allocated memory. c points to a random junk in the left over memory, since c is declared but not yet initialised.

Line 2: Pointer c was initialised to point to the address; pointer a was referring to. Array a was initialized with values (100, 101, 102,103). Value contained by the address pointed by c was changed to 200. Therefore both *c and a[0] became 200, since they both point to the same memory address.

Line 3: Since `c` and `a` are referring to the same address, therefore `**c[1]**` is the same as `**a[1]**`. Assigning `**c[1] = 300**` implies `**a[1] = 300**` and `***(c+2) => c[2] = 301**` implies `**a[2] = 301**`. As we know, `3[c]` is the same as `**c[3] => a[3]**`, therefore `a[3]` gets 302 assigned to it.

Line 4: `c = c+1` increments the c pointer to point to the next memory address = current memory address + 4 (since c is an int pointer and int data type occupies 4 bytes). It then assigns value 400 to the memory address pointed by c, which is the same as a[1]. Therefore a[1] becomes 400.

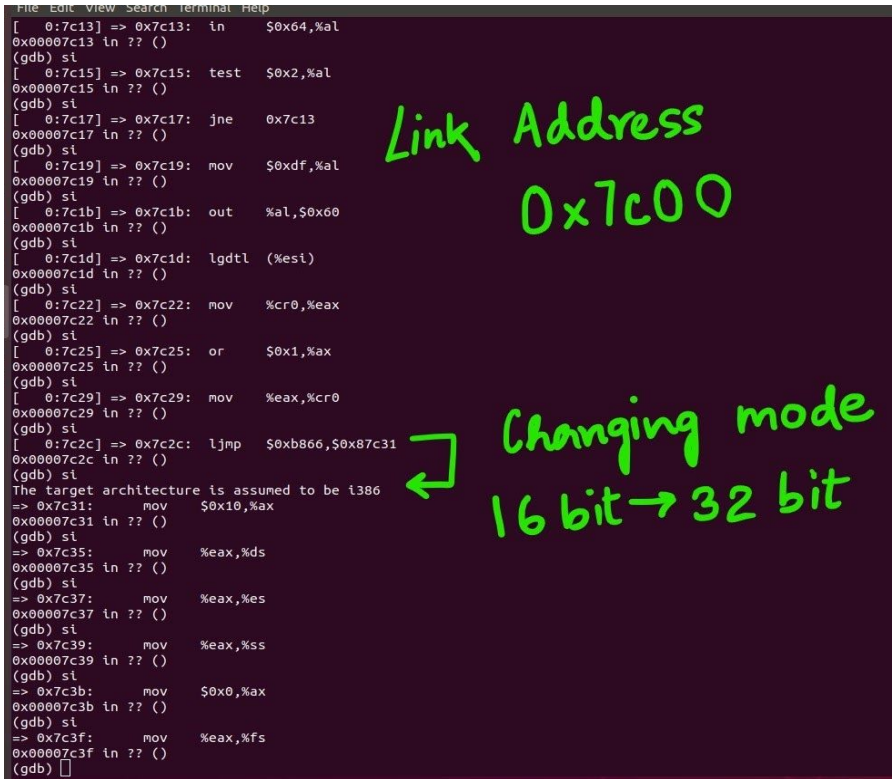
Line 5: As for now, the situation is as follows:
`c[0] = a[1] => c = 000000000061FDC4
c[1] = a[2] => c+1 = 000000000061FDC8
c[2] = a[3] => c+2 = 000000000061FDCC`
Let's take a look at this part of the code: `c = (int *) ((char *) c + 1);`
What happens here is we type c to a character pointer and increment by 1, i.e. since data type char occupies 1 byte, therefore address of c increases by 1, which is then typecasted to a int pointer, and assigned to c. Thus the new address of c is 000000000061FDC5.
Since we are using 64 bit x86-64 architecture which is little endian, so the LSB comes first.
a[1] in binary `09 10 00 00` = 400
a[2] in binary `D2 10 00 00` = 301

Now assigning `*c = 500 ` , let us see what happens:
We know 500 in hexadecimal and little endian is `4F 10 00 00`. Therefore,
a[1] becomes `09 4F 10 00` = 128144
a[2] becomes `00 10 00 00` = 256

Line 6: This code `b = (int *) a + 1;` means that pointer b is assigned to `memory address pointed by a + 4`. It is because pointer a is of int data type which occupies 4 bytes, and incrementing it takes us to the 4th byte from a. Since a is pointing to address ` 000000000061FDC0`, therefore, b points to address ` **000000000061FDC4**`. Then , `c = (int *) ((char *) a + 1);` we assign to c the immediate next byte pointed by pointer a. It works so, because we typecast a to character pointer, and since char data type has size of 1 byte, therefore incrementing it takes us to the next byte, i.e. `**c = 000000000061FDC0 + 1 => 000000000061FDC1**`.

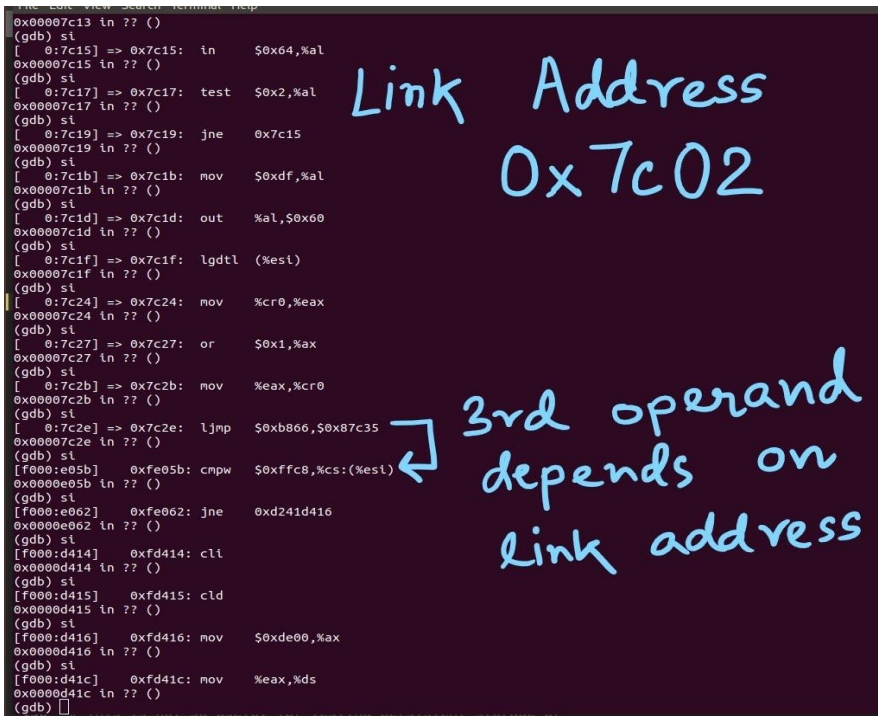
EXERCISE-5:

These are the next some instructions after **boot loader starts(0x7c00)**:-




```
File  Edit  View  Search  Terminal  Help
[   0:7c13] => 0x7c13:  in     $0x64,%al
0x00007c13 in ?? ()
(gdb) si
[   0:7c15] => 0x7c15:  test   $0x2,%al
0x00007c15 in ?? ()
(gdb) si
[   0:7c17] => 0x7c17:  jne    0x7c13
0x00007c17 in ?? ()
(gdb) si
[   0:7c19] => 0x7c19:  mov    $0xdf,%al
0x00007c19 in ?? ()
(gdb) si
[   0:7c1b] => 0x7c1b:  out    %al,$0x60
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  lgdtl  (%esi)
0x00007c1d in ?? ()
(gdb) si
[   0:7c22] => 0x7c22:  mov    %cr0,%eax
0x00007c22 in ?? ()
(gdb) si
[   0:7c25] => 0x7c25:  or     $0x1,%ax
0x00007c25 in ?? ()
(gdb) si
[   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) 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)
```

These are following instructions executed when the **link address is changed to 0x7c02:**




```
0x00007c13 in ?? ()
(gdb) si
[   0:7c15] => 0x7c15:  in     $0x64,%al
0x00007c15 in ?? ()
(gdb) si
[   0:7c17] => 0x7c17:  test   $0x2,%al
0x00007c17 in ?? ()
(gdb) si
[   0:7c19] => 0x7c19:  jne    0x7c15
0x00007c19 in ?? ()
(gdb) si
[   0:7c1b] => 0x7c1b:  mov    $0xdf,%al
0x00007c1b in ?? ()
(gdb) si
[   0:7c1d] => 0x7c1d:  out    %al,$0x60
0x00007c1d in ?? ()
(gdb) si
[   0:7c1f] => 0x7c1f:  lgdtl  (%esi)
0x00007c1f in ?? ()
(gdb) si
[   0:7c24] => 0x7c24:  mov    %cr0,%eax
0x00007c24 in ?? ()
(gdb) si
[   0:7c27] => 0x7c27:  or     $0x1,%ax
0x00007c27 in ?? ()
(gdb) si
[   0:7c2b] => 0x7c2b:  mov    %eax,%cr0
0x00007c2b in ?? ()
(gdb) si
[   0:7c2e] => 0x7c2e:  ljmp   $0xb866,$0x87c35
0x00007c2e in ?? ()
(gdb) si
[f000:e05b]    0xfe05b: cmpw   $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062]    0xfe062: jne    0xd241d416
0x0000e062 in ?? ()
(gdb) si
[f000:d414]    0xfd414: cli
0x0000d414 in ?? ()
(gdb) si
[f000:d415]    0xfd415: cld
0x0000d415 in ?? ()
(gdb) si
[f000:d416]    0xfd416: mov    $0xde00,%ax
0x0000d416 in ?? ()
(gdb) si
[f000:d41c]    0xfd41c: mov    %eax,%ds
0x0000d41c in ?? ()
(gdb)
```

We know that CS register stores the starting address of the code segment, and IP register stores the offset within the code segment.
By changing the link address we could see that the first difference occurs after the **ljmp instruction**.
We know that ljmp instruction transitions from 16 bit to 32-bit protected mode which

---

EXERCISE-6:

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

When BIOS enters the boot loader and when the boot loader enters the kernel the next 8 words are different because before there is nothing at 0x00100000 it's only after the kernel is loaded we can see further words. (we basically copy the code of the kernel into the memory location 0x00100000)

EXERCISE-7:

Studying the files **syscall.c** (the kernel side of the system call table), **user.h**(the user-level header for the system calls), and **usys.S** (the user-level system call definitions) and **sysproc.c** we understand how system calls are defined and coded in the os.
We used the same method to code the wolfie system call and added the function to implement the call **sys_wolfie()** to **sysproc.c**.
The function definition in user.h takes two parameters (the char buffer pointer and the integer size of the buffer as given in the question) which are accessed in system calls using the functions argint() for integers and argptr() for char pointers.
The first parameter to both argint() and argptr() is the zero-based index of the corresponding parameter in the list of formal parameters of the function definition.
The given wolf image in the form of ascii characters is copied into the input buffer and the size of the image is returned if input size is sufficiently large, else we return -1.

EXERCISE-8:

The user level application wolfietest.c was added to the os for the user to be able to access the system call. This c file was also added to the makefile with the other calls to compile and execute. The call is executed using "wolfietest n" , we define a buffer of this size n and invoke the system call. If the system call returns a negative integer we output an error message while if its positive the contents of the buffer are displayed on the screen.