

# **GROUP 9 MNC**

- **TEAM MEMBERS :**

- Chandrashekhar Dharmarajan(220123076)
- Srinjoy Som(220123074)
- Dhruva Nilesh Kuthari(220123015)
- Shubham Kumar Jha(220123081)

(\*\*Link to all relevant files are attached at the end of the report)

## OS Lab Assignment 1 PART A

### Question 1:

```
srinjoy2004@srinjoy2004-IdeaPad-Gaming-3-15IMH05:~/Downloads$ strings main.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__("addl %%ebx, %%eax;"
        : "=a"(x)
        : "a"(x), "b"(1));
printf("Hello x = %d after increment\n",
x); if(x == 2){
printf("OK\n");
}else{
printf("ERROR\n");
}
srinjoy2004@srinjoy2004-IdeaPad-Gaming-3-15IMH05:~/Downloads$ gcc main.c
srinjoy2004@srinjoy2004-IdeaPad-Gaming-3-15IMH05:~/Downloads$ ./a.out
Hello x = 1
Hello x = 2 after increment
OK
srinjoy2004@srinjoy2004-IdeaPad-Gaming-3-15IMH05:~/Downloads$
```

### Explanation:

1. `addl %%ebx, %%eax;`: This instruction adds the value in the `ebx` register to the value in the `eax` register and stores the result back in the `eax` register.
2. `addl`: Stands for "add long," indicating the operation is on 32-bit integers.
3. `%%ebx, %%eax`: These are placeholders for the `ebx` and `eax` registers in assembly code.
4. `: "=a"(x)`: This indicates that the output operand will be stored in the `eax` register (denoted by `a`). The `=` sign specifies that `x` is an output operand. After the assembly code executes, the value of the `eax` register will be copied to `x`.
5. `"a"(x), "b"(1)`: This specifies the input operands:
  - `"a"(x)`: The value of `x` is initially placed in the `eax` register (denoted by `a`).

- `"b" (1)`: The constant value `1` is placed in the `ebx` register (denoted by `b`).

## Question 2 :

### 2.1 BIOS

- **GDB, the GNU Debugger**, is a powerful tool for examining the execution of low-level software such as the BIOS, which is the first code that runs when a computer starts. The BIOS initialises the hardware and sets up the system to load the operating system. In this analysis, GDB is employed to trace the execution of the BIOS starting from the reset vector located at the address `0xFFF0`. This reset vector is the initial instruction pointer address where the CPU begins executing after a system reset.
- **Using the `si` command** : By setting breakpoints and using the `si` (Step Instruction) command in GDB, we can step through each instruction in the BIOS code, allowing us to observe and understand the exact operations performed during system initialization .

### Explaining the first 5 instructions :

#### 1. Instruction at `0xFFF0`: `ljmp`

`$0x3630, $0xf000e05b` : The current instruction lies at an address which is way up in the bios segment in the physical memory so it need to make a far jump transfers control to the main BIOS code from the beginning of the BIOS section of physical memory , starting the initialization process.

#### 2. Instruction at `0xFE05B`: `cmpw`

`$0xffc8, %cs:(%esi)` : Compares a

value in memory to 0xFFC8, setting flags for the next conditional instruction.

3. **Instruction at 0xFE062: jne**

**0xd241d0b0** : Jumps to a different address if the previous comparison was not equal, controlling program flow.

4. **Instruction at 0xFE066: xor %edx,%edx**

: Clears the EDX register, initialising it to 0

5. **Instruction at 0xFE068: mov %edx,%ss**

: Sets the stack segment register SS to 0, preparing the stack for further use.

```
The target architecture is set to "i8086".
[f000:fff0] 0xfffff0: jmp $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 0xd241d0b0
0x0000e062 in ?? ()
(gdb) si
[f000:e066] 0xfe066: xor %edx,%edx
0x0000e066 in ?? ()
(gdb) si
[f000:e068] 0xfe068: mov %edx,%ss
0x0000e068 in ?? ()
```

First 5 instructions along with addresses .

- **Transition from BIOS to Bootloader**

**Execution** : The BIOS code runs sequentially, initializing hardware and preparing the system for booting. This process continues until a breakpoint is set at the memory address **0x7C00**, where the bootloader is loaded. The BIOS searches for a bootable device and, upon finding one, loads the first 512 bytes of the bootloader into memory at this address. Once the BIOS transfers control to the bootloader by jumping to **0x7C00**, the bootloader begins executing its own code.

This code is responsible for further system initialization, such as switching the CPU from real mode to protected mode and loading the operating system kernel from the disk into memory.

## Command for Setting Breakpoint at 0x7c00

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

Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[ 0:7c01] => 0x7c01: xor    %eax,%eax
0x00007c01 in ?? ()
```

- **Explaining the x/Ni ADDR command :**

The **x/Ni addr** command in GDB disassembles and displays **N** instructions starting from the specified address **addr**.

- **x**: Examine memory.
- **N**: Number of instructions.
- **i**: Instruction format.
- **addr**: Starting memory address.

```
(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
```

## 2.2 Boot Loader

- **Introduction :** A boot loader is a small program that initialises the computer's hardware and loads

the operating system into memory after the BIOS has completed its checks. It resides in the boot sector of a disk and is essential for starting the operating system and preparing the system for operation.

- **Steps after BIOS completion :** Disks in PCs are divided into 512-byte sectors, with the first sector known as the boot sector containing the boot loader code. The BIOS loads this sector into memory at addresses `0x7C00` to `0x7DFF` and transfers control to the boot loader. This process focuses on loading only the first 512 bytes. The boot loader, written in assembly and C, initialises the system and loads the operating system kernel, making it essential for the bootstrapping process.
- **Summary of Bootloader functions :**
  - Switches the processor from real mode to 32 bit protected mode
  - Reads the kernel from the hard disk
  - Transfer control to kernel
- **Explaining the 32 bit protected mode:**
  - **Real Mode :** In real mode, the memory is constrained to 1MB, with addressable locations from `0x00000` to `0xFFFFF`, requiring a 20-bit address that doesn't fit into the 8086's 16-bit registers. Intel addressed this by introducing the segment : pair, which allows access to the entire 1MB range. However, this method has limitations. Each segment can only access 64K of memory, so programs larger than this need to be divided into segments, requiring frequent changes to the CS or DS registers. Additionally, the same physical memory address can be represented by

different segment combinations, making address comparisons more complex.

- **Introduction of 16-bit protected :** With the 80286, Intel introduced 16-bit protected mode, which still used the segment model but renamed segments as selectors. In real mode, segments are fixed in physical memory, while in protected mode, they can be moved around, allowing for virtual memory. This means that segments can be swapped between memory and disk as needed, all managed by the operating system. In protected mode, each segment is linked to an entry in a descriptor table, which contains information such as its memory location, permissions, and status. However, with offsets still limited to 16 bits, segment sizes remained capped at 64K, making it challenging to handle large data structures.
- **32 bit proctored introduction :** The 80386 introduced 32-bit protected mode, which expanded offsets to 32 bits, allowing segments to be up to 4GB in size. This mode also brought in paging, where segments are divided into smaller 4K pages, enhancing memory management by allowing only parts of a segment to be in memory at any given time. In essence, Intel's evolution from the 8086 to the 80386 introduced protected mode, which provided better memory protection and virtual memory capabilities, culminating in 32-bit protected mode with expanded memory addressing and paging, greatly improving system efficiency and memory management.

- **Note :** The explanation of bootmain.c code has been provided in the subsequent questions .
- **Conclusion :** So in this question we examined how the gdb debugger runs the BIOS code instruction by instruction , which in turn loads the bootloader code and gives control to it . Then we saw the function of the boot loader along with the recent 32 - bit protected architecture that it uses for instructions .



### Question 3:

The readsect() function of bootmain.c is used to read from disk and send the data to a specific memory address

```
// 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);
}
```

```
void
waitdisk(void)
{
    // Wait for disk ready.
    while((inb(0x1F7) & 0xC0) != 0x40)
        ;
    7c84:  83 e0 c0      and    $0xffffffffc0,%eax
    7c87:  3c 40         cmp    $0x40,%al
    7c89:  75 f8         jne    7c83 <waitdisk+0x
    ;
}
```

```

outb(0x1F4, offset >> 8);
7cac: 89 d8      mov    %ebx,%eax
7cae: c1 e8 08    shr    $0x8,%eax
7cb1: ba f4 01 00 00  mov    $0x1f4,%edx
7cb6: ee         out     %al,(%dx)
outb(0x1F5, offset >> 16);
7cb7: 89 d8      mov    %ebx,%eax
7cb9: c1 e8 10    shr    $0x10,%eax
7cbc: ba f5 01 00 00  mov    $0x1f5,%edx
7cc1: ee         out     %al,(%dx)
outb(0x1F6, (offset >> 24) | 0xE0);
7cc2: 89 d8      mov    %ebx,%eax
7cc4: c1 e8 18    shr    $0x18,%eax
7cc7: 83 c8 e0    or     $0xffffffe0,%eax
7cca: ba f6 01 00 00  mov    $0x1f6,%edx
7ccf: ee         out     %al,(%dx)
7cd0: b8 20 00 00 00  mov    $0x20,%eax
7cd5: ba f7 01 00 00  mov    $0x1f7,%edx
7cda: ee         out     %al,(%dx)
outb(0x1F7, 0x20); // cmd 0x20 - read sectors

// Read data.
waitdisk();
7cdb: e8 9e ff ff ff  call   7c7e <waitdisk>
asm volatile("cld; rep insl" :
7ce0: 8b 7d 08      mov    0x8(%ebp),%edi
7ce3: b9 80 00 00 00  mov    $0x80,%ecx
7ce8: ba f0 01 00 00  mov    $0x1f0,%edx
7ced: fc           cld
7cee: f3 6d        rep insl (%dx),%es:(%edi)
insl(0x1F0, dst, SECTSIZE/4);

7cf0: 5b           pop    %ebx
7cf1: 5f           pop    %edi
7cf2: 5d           pop    %ebp
7cf3: c3          ret

```

At first a waitdisk() function is called which is basically responsible for ensuring that the IDE (Integrated Drive Electronics ) disk is ready before any read or write operations are performed. This function waits in a loop, checking the status of the disk until it becomes ready, which is crucial for the bootloader process. The first

outb(0x1F2, 1);

Writes 1 to I/O port '0x1F2' setting count to 1 which signifies 1 sector will be read from the disk. Next few outb functions are

used for bookkeeping purpose or interacting with the primary IDE hard disk controller for storing sector information. Then again it waits for disk to be ready for data transfer and then

```
insl(0x1F0, dst, SECTSIZE/4);
```

Which reads 128 32-bit words from memory (512 byte of data ) from the disk to main memory.

```
// 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);
}
```

This is the code which loads sectors of the kernel from disk to memory ,

The corresponding assembly code is here

```
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
7d66: a1 1c 00 01 00      mov     0x1001c,%eax
7d6b: 8d 98 00 00 01 00    lea     0x10000(%eax),%ebx
eph = ph + elf->phnum;
7d71: 0f b7 35 2c 00 01 00 movzwl  0x1002c,%esi
7d78: c1 e6 05             shl     $0x5,%esi
7d7b: 01 de               add     %ebx,%esi
for(; ph < eph; ph++){
7d7d: 39 f3               cmp     %esi,%ebx
7d7f: 72 15               jb      7d96 <bootmain+0x59>
entry();
7d81: ff 15 18 00 01 00    call    *0x10018

7d87: 8d 65 f4             lea     -0xc(%ebp),%esp
7d8a: 5b                   pop     %ebx
7d8b: 5e                   pop     %esi
7d8c: 5f                   pop     %edi
7d8d: 5d                   pop     %ebp
7d8e: c3                   ret

for(; ph < eph; ph++){
7d8f: 83 c3 20             add     $0x20,%ebx
7d92: 39 de               cmp     %ebx,%esi
7d94: 76 eb               jbe     7d81 <bootmain+0x44>
pa = (uchar*)ph->paddr;
7d96: 8b 7b 0c             mov     0xc(%ebx),%edi
readseg(pa, ph->filesz, ph->off);
7d99: 83 ec 04             sub     $0x4,%esp
7d9c: ff 73 04             push    0x4(%ebx)
7d9f: ff 73 10             push    0x10(%ebx)
7da2: 57                   push    %edi
7da3: e8 4c ff ff ff       call    7cf4 <readseg>
if(ph->memsz > ph->filesz)
7da8: 8b 4b 14             mov     0x14(%ebx),%ecx
7dab: 8b 43 10             mov     0x10(%ebx),%eax
7dae: 83 c4 10             add     $0x10,%esp
7db1: 39 c1               cmp     %eax,%ecx
7db3: 76 da               jbe     7d8f <bootmain+0x52>
    stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
7db5: 01 c7               add     %eax,%edi
7db7: 29 c1               sub     %eax,%ecx
```

First instruction in the loop is

```
7d7d:39 f3 cmp %esi,%ebx
```

Which basically compares ph and eph to decide whether to get in the loop or break out of it .

Last instruction is to

```
7d81: ff 15 18 00 01 00 call *0x10018
```

Which jumps to code at the mentioned address now we will set a break point there and see what happens.

```
Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[ 0:7c01] => 0x7c01: xor    %eax,%eax
0x00007c01 in ?? ()
(gdb) si
[ 0:7c03] => 0x7c03: mov    %eax,%ds
0x00007c03 in ?? ()
(gdb) si
[ 0:7c05] => 0x7c05: mov    %eax,%es
0x00007c05 in ?? ()
(gdb) si
[ 0:7c07] => 0x7c07: mov    %eax,%ss
0x00007c07 in ?? ()
(gdb) si
[ 0:7c09] => 0x7c09: in     $0x64,%al
0x00007c09 in ?? ()
(gdb) si
[ 0:7c0b] => 0x7c0b: test   $0x2,%al
0x00007c0b in ?? ()
(gdb) si
[ 0:7c0d] => 0x7c0d: jne    0x7c09
0x00007c0d in ?? ()
(gdb) si
[ 0:7c0f] => 0x7c0f: mov    $0xd1,%al
0x00007c0f in ?? ()
(gdb) b *0x7d81
Breakpoint 2 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call   *0x10018

Thread 1 hit Breakpoint 2, 0x00007d81 in ?? ()
(gdb) si
=> 0x10000c:      mov    %cr4,%eax
0x0010000c in ?? ()
(gdb) si
=> 0x10000f:      or     $0x10,%eax
0x0010000f in ?? ()
(gdb) si
=> 0x100012:      mov    %eax,%cr4
0x00100012 in ?? ()
(gdb) si
=> 0x100015:      mov    $0x109000,%eax
0x00100015 in ?? ()
(gdb) si
=> 0x10001a:      mov    %eax,%cr3
0x0010001a in ?? ()
(gdb) si
=> 0x10001d:      mov    %cr0,%eax
0x0010001d in ?? ()
(gdb) si
=> 0x100020:      or     $0x80010000,%eax
0x00100020 in ?? ()
(gdb)
```

Break point is reached and it calls 0x10018 the subsequent steps are executed.

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

```
[ 0:7c05] => 0x7c05: mov %eax,%es
(gdb) si
[ 0:7c07] => 0x7c07: mov %eax,%ss
(gdb) si
[ 0:7c09] => 0x7c09: in $0x64,%al
(gdb) si
[ 0:7c0b] => 0x7c0b: test $0x2,%al
(gdb) si
[ 0:7c0d] => 0x7c0d: jne 0x7c09
(gdb) si
[ 0:7c0f] => 0x7c0f: mov $0xd1,%al
(gdb) si
[ 0:7c11] => 0x7c11: out %al,$0x64
(gdb) si
[ 0:7c13] => 0x7c13: in $0x64,%al
(gdb) si
[ 0:7c15] => 0x7c15: test $0x2,%al
(gdb) si
[ 0:7c17] => 0x7c17: jne 0x7c13
(gdb) si
[ 0:7c19] => 0x7c19: mov $0xdf,%al
(gdb) si
[ 0:7c1b] => 0x7c1b: out %al,$0x60
(gdb) si
[ 0:7c1d] => 0x7c1d: lgdtl (%esi)
(gdb) si
[ 0:7c22] => 0x7c22: mov %cr0,%eax
(gdb) si
[ 0:7c25] => 0x7c25: or $0x1,%ax
(gdb) si
[ 0:7c29] => 0x7c29: mov %eax,%cr0
(gdb) si
[ 0:7c2c] => 0x7c2c: jmp $0xb866,$0x87c31
(gdb) si
The target architecture is set to "i386".
=> 0x7c31: mov $0x10,%ax
[ 0:7c31] => 0x7c31: in ?? ()
(gdb) si
```

Ans- After running few si operation we come across a command saying

```
mov %cr0,%eax
```

Which moves the content of cr0 register into eax register.

In next two instruction

```
or $0x1,%ax
```

```
mov %eax,%cr0
```

The LSB of cr0 register is set to 1 which basically indicates shift from real mode to protected mode(LSB of cr0 is a flag for that) and then shifts to a far address with a different code segment with a long jump instruction.

```
ljmp $0xb866,$0x87c31
```

After that the following message is printed on the terminal

```
The target architecture is set to "i386".
```

```

38
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    gdt_desc
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
52
53 .code32 # Tell assembler to generate 32-bit code now.
54 start32:
55 # Set up the protected-mode data segment registers
56 movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
57 movw    %ax, %ds                # -> DS: Data Segment
58 movw    %ax, %es                # -> ES: Extra Segment
59 movw    %ax, %ss                # -> SS: Stack Segment
60 movw    $0, %ax                 # Zero segments not ready for use
61 movw    %ax, %fs                # -> FS
62 movw    %ax, %gs                # -> GS
63

```

(b) Question : What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

After exiting from the loop the booting is complete and it enters kernel mode the first instruction to be executed is

```
mov %cr4, %eax
```

```

(gdb) b *0x7d81
Breakpoint 2 at 0x7d81
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d81:      call    *0x10018

Thread 1 hit Breakpoint 2, 0x00007d81 in ?? ()
(gdb) si
=> 0x10000c:    mov     %cr4,%eax
0x0010000c in ?? ()

```

at address 0x10000c . If we see bootmain.c we will see after the loop it calls entry() function which is basically the call instruction.

```

// 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;
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);
}

// Call the entry point from the ELF header.
// Does not return!
entry = (void(*)(void))(elf->entry);
entry();
}

```

(c) Question: 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?

Ans: This info is already present in the elf file in bootmain.c we can see that `ph` is initialised by adding `elf->phoff` (offset of a `struct elfhdr` (executable and linkable format header structure) object named `elf` which signifies starting of program header table). How many sectors it has to copy is known to computer by the `phnum` member of `elf` (The number of entries of header table). Adding this integer with `ph`, we can get upto which sector disk has to be copied to get the kernel. In the loop `ph` is incremented by 1 (this code actually increases the value of the pointer by `sizeof(proghdr)` as `ph` is a pointer to `struct proghdr` (program header) ) until it becomes equal to `eph`.



#### Question 4.

```
#include <stdio.h>
#include <stdlib.h>

void f(void)
{
    int a[4];
    int *b = malloc(16);
    int *c;
    int i;

    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);
    *c = 500;
    printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3]
= %d\n",
        a[0], a[1], a[2], a[3]);

    b = (int *) a + 1;
    c = (int *) ((char *) a + 1);
    printf("6: a = %p, b = %p, c = %p\n", a, b, c);
}

int
main(int ac, char **av)
{
    f();
    return 0;
}

```

// Description of f function

```
int a[4];
```

statically allocate an array of int type of size 4

```
int *b = malloc(16);
```

dynamically allocate a memory of 16 bytes to an `int*`, i.e. a pointer to an integer, so 4 integers

```
int *c;
```

pointer to an integer

```
int i;
```

an integer, uninitialised, used as a loop variable later on

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

prints the addresses of a, b, and c. c may contain a garbage value because it is uninitialised.

```
1: a = 00000053621ff700, b = 000002206ce11430, c =  
0000000000000010
```

Output specific to my pc

```
c = a;
```

now c points to the start of array a

```
for (i = 0; i < 4; i++)  
    a[i] = 100 + i;
```

initialises a with `a[0] = 100`, `a[1] = 101`, `a[2] = 102` and `a[3] = 103`

```
c[0] = 200;
```

makes `a[0] = 200` as c points to the starting of a and `c[0] = *(c+0)`

```
printf("2: a[0] = %d, a[1] = %d, a[2] = %d, a[3] =  
%d\n",  
      a[0], a[1], a[2], a[3]);
```

prints values of `a[0]`, `a[1]`, `a[2]`, `a[3]`

```
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
```

```
c[1] = 300;
```

makes `a[1] = 300` as `c[1] = *(c+1)`, i.e. points to 1 location (here 4 bytes, i.e. 1 int) ahead of `c`

```
*(c + 2) = 301;
```

Makes `a[2] = 301` as `*(c+2)`, points to 2 locations (here 2 sets of 4 bytes, i.e. 2 int) ahead of `c`

```
3[c] = 302;
```

makes `a[3] = 302` as `3[c] = *(c+3)`, points 3 locations (here 3 int) ahead of `c`

```
printf("3: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",  
      a[0], a[1], a[2], a[3]);
```

Prints value of `a[0]`, `a[1]`, `a[2]` and `a[3]`

**3: `a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302`**

```
c = c + 1;
```

Moves `c` pointer to 1 location ahead (i.e. 4 bytes or 1 int), so now it points to `a[1]`

```
*c = 400;
```

Makes `a[1] = 400` as `*c = *(c+0)`, and `c` points at `a[1]`

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

Prints value of `a[0]`, `a[1]`, `a[2]` and `a[3]`

**4: `a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302`**

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

Casts `c` to `char*` type and then move it by 1 location(here **1 byte**, or **1 char**), then casts it back to `int*`

`c` before this line of code:



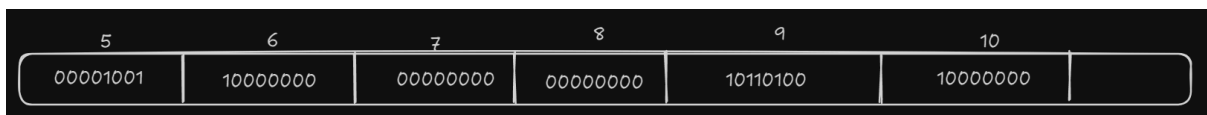
`c` after this line of code



```
*c = 500;
```

Note: `int` is stored from LSB to MSB fashion(also known as Little-Endian).

`a` before this line of code:



`a` after this line of code



```
printf("5: a[0] = %d, a[1] = %d, a[2] = %d, a[3] = %d\n",  
      a[0], a[1], a[2], a[3]);
```

Prints value of a[0], a[1], a[2] and a[3]

5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302

Calculation for a[1] and a[2]:

$$a[1] = 2^{16} + 2^{15} + 2^{14} + 2^{13} + 2^{12} + 2^{10} + 2^7 + 2^4 = 128144$$

$$a[2] = 2^8 = 256$$

```
b = (int *) a + 1;
```

Makes b point to 1 location(here 4 bytes,i.e. 1 int) ahead from base address of a, i.e. at the address of a[1]

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

casts a to char\* type and then move it by 1 location(here **1 byte**, or **1 char**), then casts it back to int\*

Makes c to point to 1 byte ahead of base address of array a

```
printf("6: a = %p, b = %p, c = %p\n", a, b, c);
```

prints the addresses of a, b, and c.

6: a = 00000053621ff700, b = 00000053621ff704, c = 00000053621ff701

a is equal to its base address, b is equal to 1 location(4 bytes,i.e. 1 int) ahead of a,

and c is equal to 1 byte ahead of a

## Question 5 :

**Objective :** We want to understand the concept of link and load addresses .So traditionally the link address for the start point of the bootloader in the makefile is 0x7c00 , and the load address is the same as well , due to which the execution happens correctly.

So let's change the link address in the makefile and examine the changes in the bootmain.asm file . We will try to identify what changes will happen in the execution and at which instruction the execution will break or things will start going wrong !! .

```
bootblock.o:      file format elf32-i386

Disassembly of section .text:

00007f00 <start>:
# with %cs=0 %ip=7c00.

.code16                      # Assemble for 16-bit mode
.globl start
start:
  cli                        # BIOS enabled interrupts; disable
    7f00:      fa            cli

# Zero data segment registers DS, ES, and SS.
xorw  %ax,%ax               # Set %ax to zero
    7f01:      31 c0        xor    %eax,%eax
movw  %ax,%ds               # -> Data Segment
    7f03:      8e d8        mov    %eax,%ds
movw  %ax,%es               # -> Extra Segment
    7f05:      8e c0        mov    %eax,%es
movw  %ax,%ss               # -> Stack Segment
    7f07:      8e d0        mov    %eax,%ss

00007f09 <seta20.1>:

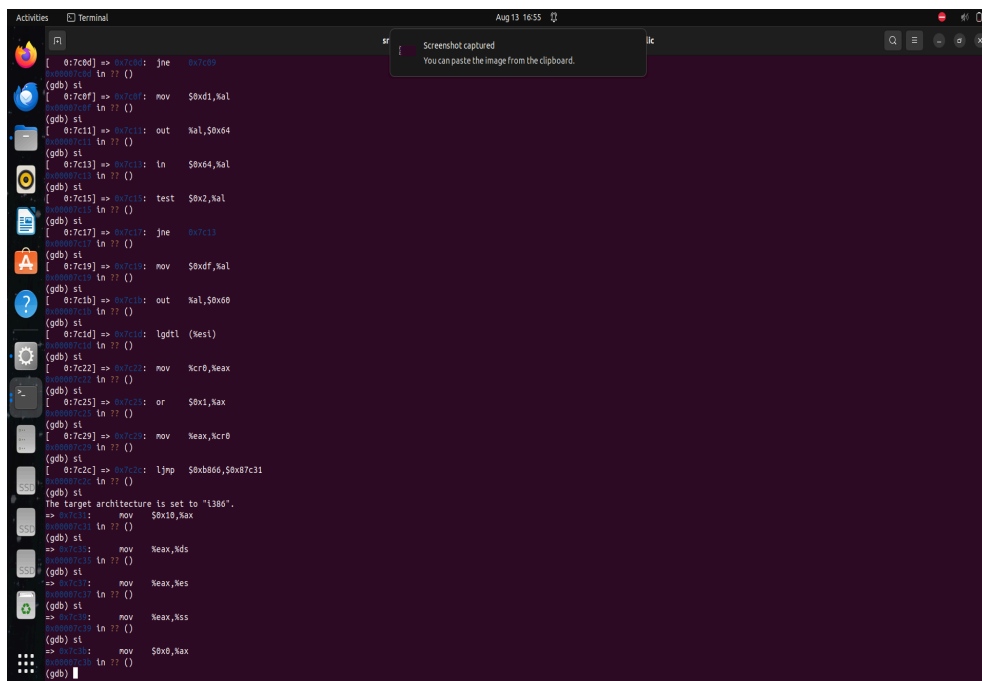
# Physical address line A20 is tied to zero so that the first PCs
# with 2 MB would run software that assumed 1 MB. Undo that.
seta20.1:
  inb  $0x64,%al            # Wait for not busy
    7f09:      e4 64        in     $0x64,%al
  testb $0x2,%al
    7f0b:      a8 02        test   $0x2,%al
  jnz  seta20.1
    7f0d:      75 fa        jne    7f09 <seta20.1>

  movb  $0xd1,%al           # 0xd1 -> port 0x64
    7f0f:      b0 d1        mov     $0xd1,%al
  outb  %al,$0x64
    7f11:      e6 64        out     %al,$0x64

00007f13 <seta20.2>:

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

In the file named makefile we changed the link address to be 0x7f00 then we run make clean and then we recompile with make .Above we can see that in bootblock.asm the starting address is changed from 7c00 to 7f00.Still after changing the bios instructions are unchanged because changing linking address should not affect that. After this we set a breakpoint at 0x7c00 and then run c and after reaching breakpoint we run a few si instructions. For the first few instructions it is unchanged but soon it starts to behave unexpectedly . The problem occurs after the long jump which is supposed to set the instruction architecture to i386.After linking address is changed this message does not pop on the screen as well as all following instructions are different from that when run with link and load address to be the same.



```

(gdb) si
[ 0:7c0d] => 0:7c0d: jne 0x7c09
(gdb) si
[ 0:7c0f] => 0:7c0f: mov $0xd1,%al
(gdb) si
[ 0:7c11] => 0:7c11: out %al,$0x04
(gdb) si
[ 0:7c13] => 0:7c13: in $0x04,%al
(gdb) si
[ 0:7c15] => 0:7c15: test $0x2,%al
(gdb) si
[ 0:7c17] => 0:7c17: jne 0x7c13
(gdb) si
[ 0:7c19] => 0:7c19: mov $0xdf,%al
(gdb) si
[ 0:7c1b] => 0:7c1b: out %al,$0x08
(gdb) si
[ 0:7c1d] => 0:7c1d: lgdtl (%esi)
(gdb) si
[ 0:7c22] => 0:7c22: mov %cr0,%eax
(gdb) si
[ 0:7c25] => 0:7c25: or $0x1,%ax
(gdb) si
[ 0:7c29] => 0:7c29: mov %eax,%cr0
(gdb) si
[ 0:7c2c] => 0:7c2c: ljmp $0xb06,$0x7c31
(gdb) si
The target architecture is set to "i386".
=> 0:7c31: mov $0x10,%ax
(gdb) si
=> 0:7c33: mov %eax,%ds
(gdb) si
=> 0:7c35: mov %eax,%es
(gdb) si
=> 0:7c37: mov %eax,%ss
(gdb) si
=> 0:7c39: mov $0x0,%ax
(gdb)

```

Instructions when link and load address is same



```

(gdb) si
[ 0:7c0d] => 0x7c0d: jne 0x7c09
0x00007c0d in ?? ()
(gdb) si
[ 0:7c0f] => 0x7c0f: mov $0xd1,%al
0x00007c0f in ?? ()
(gdb) si
[ 0:7c11] => 0x7c11: out %al,$0x64
0x00007c11 in ?? ()
(gdb) si
[ 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,$0x87f31
0x00007c2c in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpw $0xffc8,%cs:(%esi)
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xd241d0b0
0x0000e062 in ?? ()
(gdb) si
[f000:d0ae] 0xfd0ae: cli
0x0000d0ae in ?? ()
(gdb) si
[f000:d0af] 0xfd0af: cld
0x0000d0af in ?? ()
(gdb) si
[f000:d0b0] 0xfd0b0: mov $0xd980,%ax

```

Instructions when link and load address is different

Also we can see when we run c it again comes back to 0x7c00 and breaks as there was a breakpoint here which is very unexpected.

## Question 6 :

**Objective :** For this experiment, we have to examine the 8 words of memory at 0x00100000 at two different instances of time, the first when the BIOS enters the boot loader and the second when the boot loader enters the kernel. For this, we will use the command “x/8x 0x00100000” but before that we will have to set our breakpoints. The first breakpoint will be at 0x7c00 because this is the point where the BIOS hands control over to the boot loader. The second breakpoint will be at 0x0010000c because this is the point when the kernel is passed control by the boot loader.

**Showing the Process :** setting breakpoint at 0x7c00 to transfer control to bootloader from BIOS .

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

**Checking the first 8 words** at 0x00100000 : we can see that all words are initialised to 0

```
Thread 1 hit Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000: 0x00000000 0x00000000 0x00000000 0x00000000
0x100010: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) |
```

**Setting a breakpoint at 0x0010000c** to transfer control to the kernel from the bootloader .

```
(gdb) b* 0x00010000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x10000c: mov %cr4,%eax

Thread 1 hit Breakpoint 2, 0x0010000c in ?? ()
(gdb) |
```

**Checking the first 8 words** at 0x00100000 now

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

**Explanation :** At the first breakpoint, the address 0x00100000 (which is 1MB) contains only zeroes since the kernel hasn't yet been loaded into memory. In xv6, uninitialized memory is set to 0 by default, so when we initially read the 8 words at this address, we see all zeroes. By the time we reach the second breakpoint, the kernel has been loaded into this

address, filling it with meaningful data instead of zeroes. This explains the different values observed at the two breakpoints.

# GROUP 9 MNC

- **TEAM MEMBERS :**

- Chandrashekhar Dharmarajan(220123076)
- Srinjoy Som(220123074)
- Dhruva Nilesh Kuthari(220123015)
- Shubham Kumar Jha(220123081)

## OS Lab Assignment 1 Part B

### Question 1 :

**Objective :** We need to add a system call to the xv6 library called sys\_draw which takes in an ASCII generated image and copies it to a buffer and returns the size of the image if it's valid (i.e not too small or big !!!) .

Steps :

In syscall.h we make a 22nd entry of system call named draw .The hash defining is done so that we can get the function pointer from the corresponding index in the array of function pointers in syscall.c.

```
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
#define SYS_draw    22
```

Next in the syscall.c file we need to add a function pointer to this system call. We also add the following line in this file.

```
[SYS_close]    sys_close,
[SYS_draw]     sys_draw ,
};
```

To run the function we need to declare the function but we don't need to define it here .

```
extern int sys_uptime(void);
extern int sys_draw(void) ;
```

In sys\_proc.c we add another function with the already existing ones named draw().

```
int
sys_draw(void)
{
    void* buf;
    uint size;

    argptr(0, (void*)&buf, sizeof(buf));
    argptr(1, (void*)&size, sizeof(size));

    char text[] = "Simpson says Hi : \n\
| | | | | | | \n\
|           | \n\
|           | \n\
| (o)(o)   \n\
c         _)\n\
| ,_|      \n\
|   |      \n\
|   \\\n\
/       \\\n\
\n";

    if(sizeof(text)>size)
        return -1;

    strncpy((char *)buf, text, size);
    return sizeof(text);
}
```

user.S contains the code that initialises the execution environment for user programs so in usys.S we add

```
SYSCALL(uptime)
SYSCALL(draw)
```

In header we have to declare the functions so in user.h

```
char* sbrk(int);
int sleep(int);
int uptime(void);
int draw(void *, uint);
```

### Question 2 :

To show that the added system call is working we make a c file named draw\_test.c which utilises the draw system call in the xv6 folder .

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

int
main(void)
{
    static char buf[4000];
    printf(1, "Testing the draw system call %d\n", draw((void*) buf, 4000));

    printf(1, "%s", buf);
    exit();
}
```

So that qemu knows this file we add it in the file named 'makefile' then we run make clean and then recompile and run this file and you can see mr simpson saying hello to you.

### Testing Time :

```
SeaBIOS (version 1.15.0-1)

ipXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00


Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ draw_test
Testing the draw system call    402
Simpson says Hi :

      |||||
      |   |
      |   |
      | (o)(o)
      | C   _
      | , - |
      |     \
      /       \

$ |
```

## Assignment1 part 1:

[https://drive.google.com/drive/folders/1-am-bzeA4k6x\\_sC1e3wWnK\\_PC3AKOrCA?usp=sharing](https://drive.google.com/drive/folders/1-am-bzeA4k6x_sC1e3wWnK_PC3AKOrCA?usp=sharing)

## Assignment1 part2:

[https://drive.google.com/drive/folders/1dkl85PbktRWd6Btv4fwZO\\_cwLN\\_PojLDx?usp=sharing](https://drive.google.com/drive/folders/1dkl85PbktRWd6Btv4fwZO_cwLN_PojLDx?usp=sharing)