

**REPUBLIQUE DU CAMEROUN**

**Paix- Travail-Patrie**

**MINISTRE DE**

**L'ENSEIGNEMENT SUPERIEUR**

**FACULTY DE L'INGENIEURIE**

**ET TECHNOLOGIE**



**REPUBLIC OF CAMEROON**

**Peace- Work-Fatherland**

**MINISTER OF HIGHER**

**EDUCATION**

**FACULTY OF ENGINEERING**

**AND TECHNOLOGY**

## **UNIVERSITY OF BUEA**

**DEPARTMENT OF COMPUTER ENGINEERING**

**COURSE: OPERATING SYSTEM**

**COURSE CODE: CEF347**

### **REQUIREMENT, ANALYSIS, DESIGN AND IMPLEMENTATION OF AN OPERATING SYSTEM**

**Report presented by members of group 18**

**Supervised by: Dr Nkemeni Valery**

**Presented by members of group 18:**

<b>1. POKAM NGOUFFO TANEKOU</b>	<b>FE21A299</b>
<b>2. DJEUTIO QUOIMON ANDERSON ROY</b>	<b>FE21A169</b>
<b>3. NOUPOUWO DONGMO STEPHANE MERCI</b>	<b>FE21A283</b>
<b>4. TIANI PEKINS EBIKA</b>	<b>FE21A325</b>
<b>5. AKENGNI KEANLI EMMANUEL</b>	<b>FE21A132</b>
<b>6. JERRY AYUKNKEM EBAI</b>	<b>FE21A370</b>
<b>7. FOLAMO BENOIT</b>	<b>FE21A188</b>
<b>8. DEGAULLE NJUKANG NKENGAFAC</b>	<b>FE21A353</b>
<b>9. AGBOR LOVETH</b>	<b>FE20A003</b>
<b>10. DJIGHO FOSSO JUNIOR</b>	<b>FE19A139</b>

**On Sunday 08-January-2023**

## **AIM: To Design an operating System**

# **Introduction:**

An Operating System is a system software used to link the application software to the hardware. An operating system is made up of the following components:

### **1. The Shell**

- ❖ The Shell is a collection of Commands.
- ❖ The Shell acts as an interface between the user and the kernel.
- ❖ The Shell is a Command Line Interpreter-Translate the commands provided by the user and converts it into a language that is understood by the Kernel.
- ❖ Only one Kernel running on the system, but several shells in action-one for each user who is logged in.
- ❖ E.g.: C Shell, Bourne Shell, Korn Shell etc.

### **2. The Kernel**

- ❖ The Kernel is the heart of the Operating System
- ❖ It interfaces between Shell and Hardware.
- ❖ It performs Low Level Task.
- ❖ E.g. Device Management, Memory Management etc.

### **3. Files and Processes**

- ❖ A File is an array of bytes and can contain anything.
- ❖ All data is organized into files.
- ❖ A process is a program file under execution.
- ❖ Files and Processes belongs to a separate hierarchical structure.

### **4. System Calls**

- ❖ Thousands of commands in the system uses a handful of functions called System Calls to communicate with the kernel.

We are presenting our operating system in this work as shown below;

# **Requirement:**

To build this operating system, we used different soft wares ,with each performing their various tasks .A list of these software are listed below:

1. QEMU
2. HEX EDITOR
3. NOTEPAD++
4. NASM
5. SASM
6. CODE::BLOCKS

## **Requirement Analysis :**

### **1. QEMU:**

To run the operating system we developed, we used an Emulator named QEMU

### **2. HEX EDITOR:**

Used to see the hexadecimal output in this system.

### **3. NOTEPAD++:**

Used to write assembly language codes which are later stored as SASM

### **4. NASM:**

NASM is a widely used assembler. NASM was used as our assembler.

### **5. SASM:**

SASM is an ide which helps build assembly programs easily. SASM was used to write all our assembly codes

### **6. CODE::BLOCKS:**

An ide used for compiling C programs. We used CODE::BLOCKS to write and compile our C programs.

After downloading the above applications, environment variables were been put in place.

# Design:

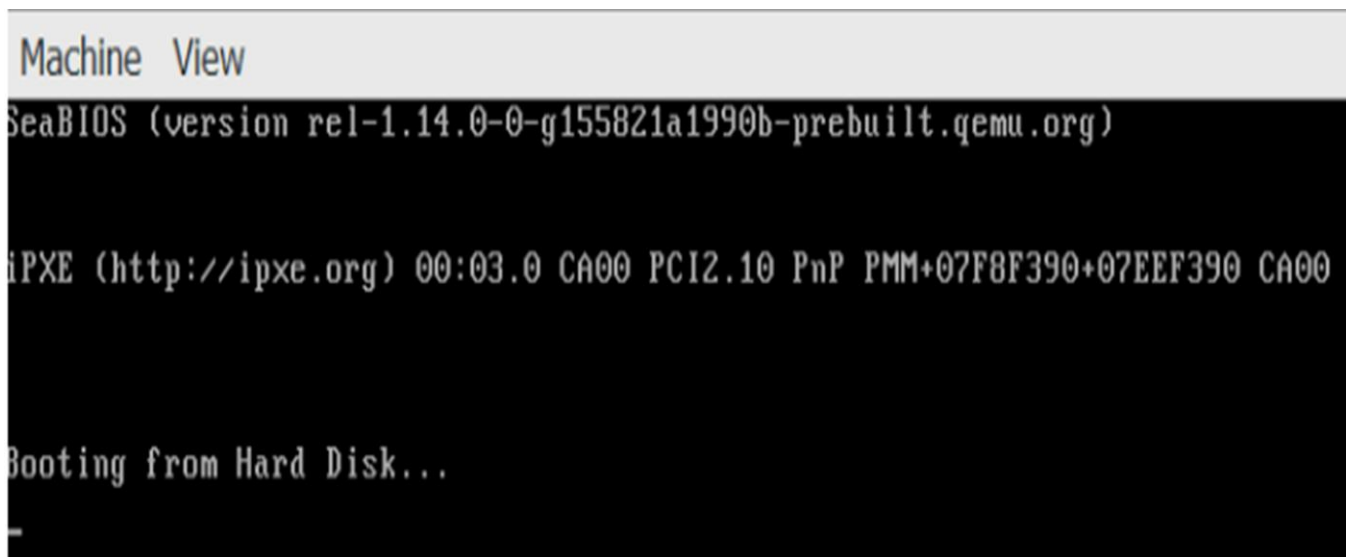
The design of our operating system was done by coding the following commands chronologically

## ❖ **Boot Sector:**

Boot Sector is a 512 bytes location in the Hard Disk where we placed the codes that we ran. To be able to boot a file we typed the following commands in the command prompt:

1. To tell the computer it is an executable file, we typed the following code in sasm an assembled it:  
loop:  
    jmp loop  
    times 510-(\$-\$\$) db 0  
    dw 0xaa55
2. To create a bootable file, we typed: `nasm boot-sect0.asm -f bin -o filename.`
3. To both the file, we typed: `qemu-system-i386 -drive format=raw ,file=filename.`

After the following operations our output was as follows:



```
Machine View
SeaBIOS (version rel-1.14.0-0-g155821a1990b-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8F390+07EEF390 CA00

Booting from Hard Disk...
```

### ❖ Printing to the screen:

Here we printed ***hello world*** to the screen by writing and assembly code to move data into some 16 bit registers, with our computer initially in 16 bit mode after boot-up. Our output was as follows:

```
Machine View
SeaBIOS (version rel-1.14.0-0-g155821a1990b-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8F390+07EEF390 CA00

Booting from Hard Disk...
HELLO WORLD_
```

### ❖ Filling the screen with colors:

To our hello world assembly code, we added the following piece of code to print a green color to the screen:

```
mov ah , 0x0b ; by changing the value
mov bh , 0x0  ; assigned to b1
mov bl , 0xff  ; (0xff) we could change
int 0x10      ; the screen colors
```

We obtained the output below:

```
Machine View
SeaBIOS (version rel-1.14.0-0-g155821a1990b-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8F390+07EEF390 CA00

Booting from Hard Disk...
HELLO WORLD
```

## ❖ Switching to 32bits protected mode:

We switched to 32 bits mode to allow us to protect some memory locations (where the kernel of our OS is found) from user mode programs cause these programs could take control of CPU and our OS would have no control anymore. To perform this switch we used a ready-made Global Descriptor Table(GDT) which is a special data structure which the processor directly validates. After defining and loading the GDT we could now switch to 32 bit mode. We had no special output(the same output as that of the boot sector above) since we only switched to 32 bit mode. The assembly code for our GDT and switch is given below:

```
15 ;Switch To Protected Mode
16 cli ; Turns Interrupts off
17 lgdt [GDT_DESC] ; Loads Our GDT
18
19 mov eax , cr0
20 or  eax , 0x1
21 mov cr0 , eax ; Switch To Protected Mode
22
23 jmp  CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
24 ;Forces the cpu to flush out contents in cache memory
```

```
47 GDT_BEGIN:
48
49 GDT_NULL_DESC: ;The Mandatory Null Descriptor
50     dd 0x0
51     dd 0x0
52
53 GDT_CODE_SEG:
54     dw 0xffff ;Limit
55     dw 0x0    ;Base
56     db 0x0    ;Base
57     db 10011010b ;Flags
58     db 11001111b ;Flags
59     db 0x0    ;Base
60
61 GDT_DATA_SEG:
62     dw 0xffff ;Limit
63     dw 0x0    ;Base
64     db 0x0    ;Base
65     db 10010010b ;Flags
66     db 11001111b ;Flags
67     db 0x0    ;Base
68
69 GDT_END:
70
71 GDT_DESC:
72     dw GDT_END - GDT_BEGIN - 1
73     dd GDT_BEGIN
74
75 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
76 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
77
78
79 times 510-($-$$) db 0
80 dw 0xaa55
```

### ❖ Making a boot loader:

A boot loader is a piece of code in Boot Sector which loads the remaining part of the OS to memory that is the C program will always be greater than 512 bytes. This piece of code we implemented as shown below:

```
4 ;Boot Loader
5 mov bx , 0x1000 ; Memory offset to which kernel will be loaded
6 mov ah , 0x02 ; Bios Read Sector Function
7 mov al , 30 ; No. of sectors to read(If your kernel won't fit into 30 sectors
8 mov ch , 0x00 ; Select Cylinder 0 from harddisk
9 mov dh , 0x00 ; Select head 0 from hard disk
10 mov cl , 0x02 ; Start Reading from Second sector(Sector just after boot sector)
11
12 int 0x13 ; Bios Interrupt Relating to Disk functions
```

### ❖ Calling our C Kernel:

In order to execute the C program we made, we used a linker to automatically calculate where our entry C function will be positioned. Since we found it difficult to know where our entry C function is positioned in memory .Some changes were made as we modified our C program .The linker used is the kernel entry.asm where the code used by this named linker are shown

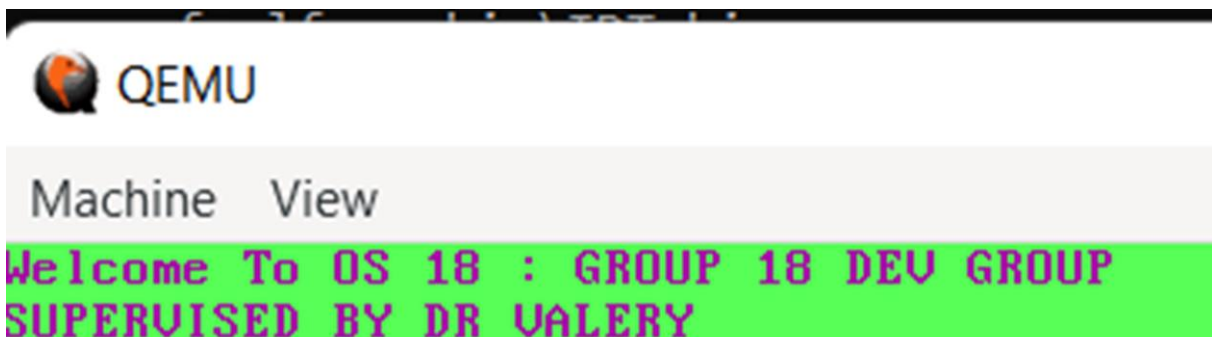
```
1 START:
2 [bits 32]
3 [extern _start]
4 call _start
5 jmp $
```

```
int start(){
    char* video_memory = (char*) 0xb8000;
    *video_memory = 'K';
}
```

Assembly program

C program

From here we call our start function in our assembly program and obtained the below output:





we changed the K assigned to the video memory pointer to the text displayed on the output.

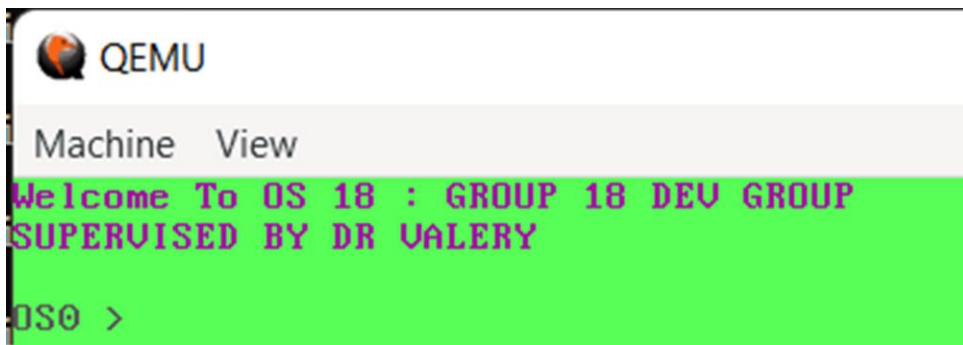
### ❖ Video Graphics:

Displaying units and factors were the most crucial part when developing our OS as we needed to print out units to the screen, thus debugging any arising issues. Normally after the boot up the computer video card will be in text mode, we could print something (characters and colors) to the screen the screen by poking (writing) memory location starting from 0xb800.

Looking at the text mode video graphics it was hard for us to print strings to the screen, so we implemented our own 'print' (print function) that we later called to do the printing operation.

**THEORY:** Here we poked the video memory, as changing the data contained in the video memory continuously gives us a video feel since human eyes see about 60 frames per second.

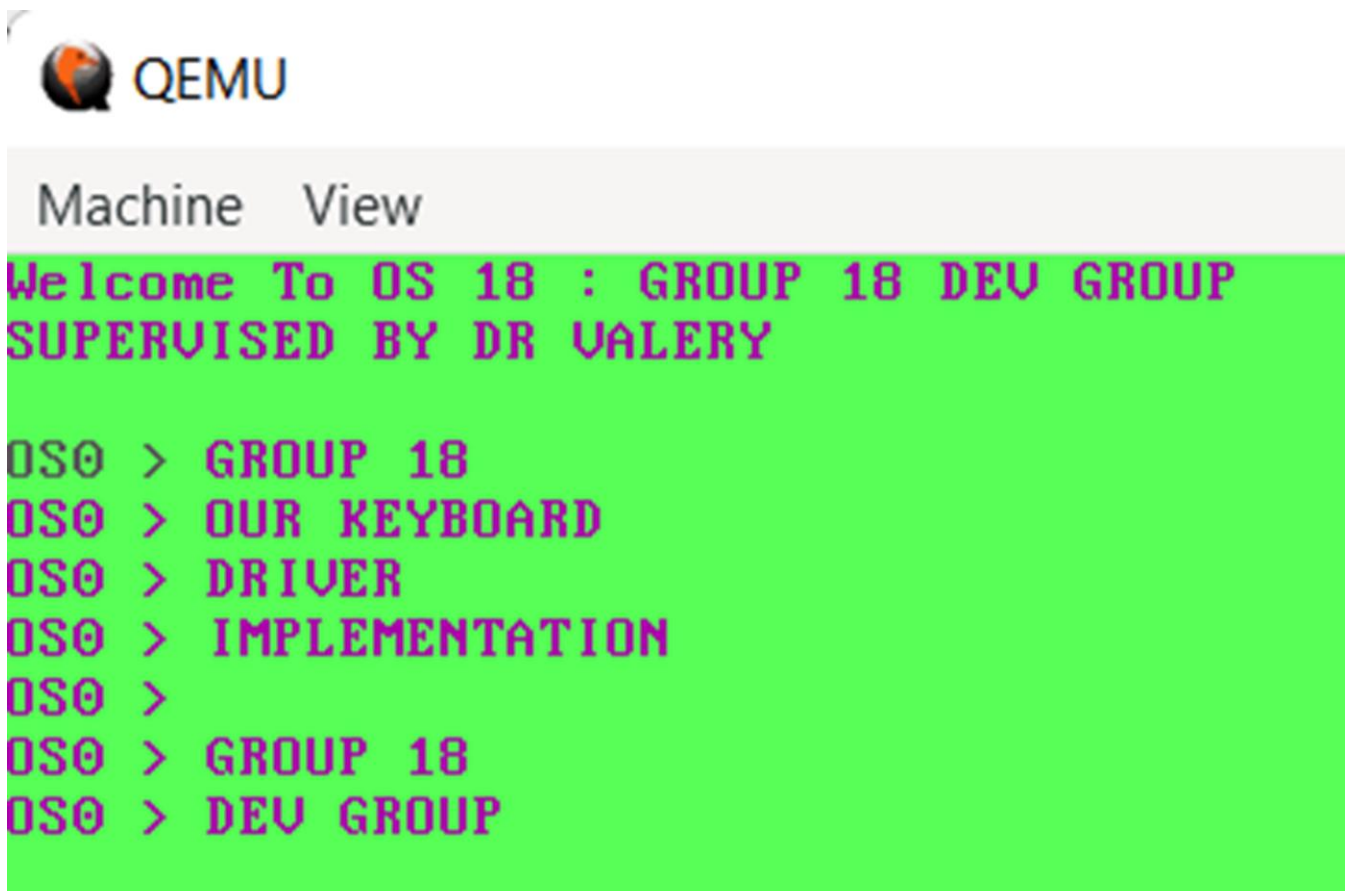
our output for the video implementation is shown below:



### ❖ Implementing keyboard driver:

Keyboard is the primary input device for the computer so working with the keyboard was an inevitable factor in the OS development. Our keyboard is slower than our processor hence it affects our processor speed. Whenever a key is being pressed, the CPU will call a function that is pre-defined to it. The processor won't do the keyboard logic, but it will let us execute some code whenever a key is being pressed. We need to make some code to handle the keyboard input, and pass its address to the interrupt descriptor table and say to the processor to load it. After loading all the parameters and the location of our keyboard handling code, it will call that code whenever a key is being pressed. When we get this interrupt, we could try reading from the keyboard which gives us the key being inputted/pressed.

What keyboard gives as the value for pressed key is not ascii. It is named as scan codes. PIC or Programmable Interrupt Controller is a chip in the computer whose main job is to generate interrupts. When a key in keyboard is pressed, The Chip inside keyboard tells to the pic chip inside our computer to generate a #1 Interrupt. The pic chip will then decide the time to notify the CPU about the interrupt. When the CPU gets the message which says a key is being pressed, It executes a set of code which we told earlier to the CPU to execute when a key is pressed. After our implementation we entered some instructions and obtained the following output:



After this successful implementation we were able to input characters using the keyboard

#### ❖ **First prototype OS 18:**

Here we implemented our first prototype **OS 18** which was a combination of what we have done so far. By entering a series of commands one could clear the screen, change the screen color, play videos and “print a string” to the screen. when a videos is played it can only be stopped by closing the Qemu software. The different commands use to perform the above listed

Operations are given below:

**CLS**: this command clears the entire screen content

**COLORA**: changes screen color to

**COLORB**: changes screen color to

**COLORC**: changes screen color to

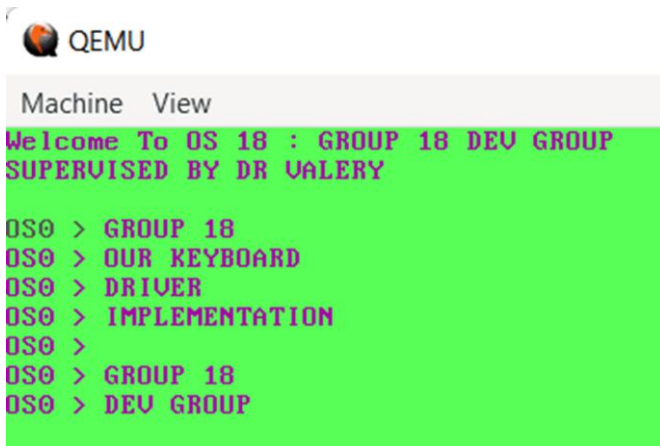
**COLORDEF**: changes screen color to definition color (green)

**VID**: plays video

**HI**: displays a string

The output of the following commands is shown below:

COLORDEF

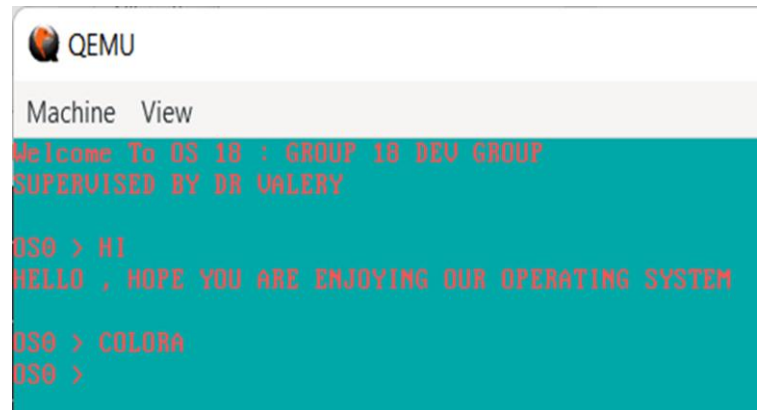


A QEMU terminal window with a light gray title bar containing the QEMU logo and the text 'QEMU'. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal background is bright green. The text is in a purple/magenta monospace font. It displays a welcome message, followed by a series of commands and their outputs.

```
Machine View
Welcome To OS 18 : GROUP 18 DEV GROUP
SUPERVISED BY DR VALERY

OS0 > GROUP 18
OS0 > OUR KEYBOARD
OS0 > DRIVER
OS0 > IMPLEMENTATION
OS0 >
OS0 > GROUP 18
OS0 > DEV GROUP
```

COLORA



A QEMU terminal window with a light gray title bar containing the QEMU logo and the text 'QEMU'. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal background is teal. The text is in a red/orange monospace font. It displays a welcome message, followed by a series of commands and their outputs.

```
Machine View
Welcome To OS 18 : GROUP 18 DEV GROUP
SUPERVISED BY DR VALERY

OS0 > HI
HELLO , HOPE YOU ARE ENJOYING OUR OPERATING SYSTEM

OS0 > COLORA
OS0 >
```

COLORB



A QEMU terminal window with a light gray title bar containing the QEMU logo and the text 'QEMU'. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal background is purple. The text is in a green monospace font. It displays a welcome message, followed by a series of commands and their outputs.

```
Machine View
Welcome To OS 18 : GROUP 18 DEV GROUP
SUPERVISED BY DR VALERY

OS0 > HI
HELLO , HOPE YOU ARE ENJOYING OUR OPERATING SYSTEM

OS0 > COLORB
OS0 >
```

COLORC

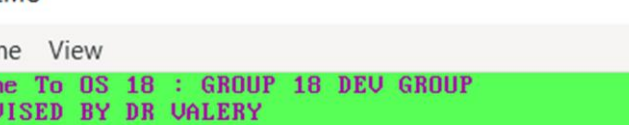


A QEMU terminal window with a light gray title bar containing the QEMU logo and the text 'QEMU'. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal background is bright green. The text is in a green monospace font. It displays a welcome message, followed by a series of commands and their outputs.

```
Machine View
Welcome To OS 18 : GROUP 18 DEV GROUP
SUPERVISED BY DR VALERY

OS0 > HI
HELLO , HOPE YOU ARE ENJOYING OUR OPERATING SYSTEM

OS0 > COLORC
OS0 >
```



QEMU

Machine View

Welcome To OS 18 : GROUP 18 DEV GROUP  
SUPERVISED BY DR VALERY

OS0 > HI  
HELLO , HOPE YOU ARE ENJOYING OUR OPERATING SYSTEM

OS0 > COLORA  
OS0 > COLORDEF  
OS0 >

QEMU

Machine View

DS0 >

The screenshot shows a QEMU window titled "Machine View". The main area displays a corrupted screen with a repeating pattern of garbled text and symbols, including "Bπeroδ n±j±~J", "EGIKMU", and "YI". The text is rendered in various colors (red, green, blue, yellow) against a black background, indicating a severe display or memory corruption issue within the virtual machine.

## ❖ Accessing hard disk

We implemented a Hard drive for the ATA technology. We Read/Wrote from the hard disk using LBA mode since we needed to pass the Block address of sector. Passing 0 will give us access to the first sector (Boot sector).

We added two commands to the **strEval** function which GET and PUT commands. When the PUT command is typed on the QEMU, it first copies the number 0 to **blockAddr** and then proceeds to initialize every cell in at character array with 'J' and finally adds a null character. We set J as default character to check our implementation

If we try running the GET command first, we'll get some random character as output but if the PUT command is passed first then later on the GET command then we'll get a string of **J's**

[illegible]



## ❖ Creating a simple file

We'll build a file system which allows us to give names to a file and store up to 512-byte data in a file. Before creating a file, the user must format the hard disk using the FORMAT command. The formatting operation first stores four random bytes in the beginning of sector 0.

We should note that the sector 0 is always used to store the boot loader. Writing other data to sector 0 will make the system un-bootable. So we'll need to choose different Sectors.

**FAT, NTFS, EXT**, etc. are examples of file systems

### **a. TO CREATE A FILE**

We check the first four bytes stored in sector 0 and if it is same as what we stored in the formatting section, we do the next operation. Our main aim for this check is to see if we have previously formatted the disk. If the check fails, say to the user to enter the command to format it. Now we store the name of file in sector 0 just after the name of last file we created. Name of the very first file can be stored just after the first four bytes.

### **b. TO SAVE A FILE**

To save a file, we need to have created the file before. We shall use the SAVE command to do the save operation. After the user enters the save command, we obtain the index of the file specified. We will get 1 as index if the file specified is the first file defined at sector 0. Like that, we will get 2 as index if the file specified is the second file defined at sector 0. We then take this index as the sector count where the string should be saved to.

## ❖ GUI creation

Since we're to switch from text mode to graphics mode, we have different video modes to choose from. These different modes have their abilities with their own supported resolution and colors. Graphics mode mostly deals with text mode graphics. Switching to graphics mode is done using BIOS but since we're in 32 bit protected mode we won't use BIOS on this mode. We'll switch down to 16bit mode.

**mov ah, 0x00;** This option in BIOS is to switch mode

**mov al, 0x13;** This option switches to video mode, VGA320\*200 256

**int 0x10;** With this function, we call the BIOS

We shall create a folder for this since our OS is in 32bit protected mode. The folder we'll create is just a crash file for the GUI. We've a whole lot of codes that we'll specify later on, for now we'll show our **main.c** file and the output generated when we include it in our **compile.bat** file.

```
int start () {  
  
    char* vbuff = (char*) 0xA0000;  
    char colr = 0x00;  
  
    int i = 0;  
  
    while (i < 320 * 200){  
  
        *(vbuff + i) = colr;  
  
        colr++;  
  
        i++;  
  
    } }  

```



## ❖ Implementing mouse driver

We are to generate a PS2 mouse. A PS2 mouse generates IRQ12. Once we initialize a mouse, it sends ; 3 or 4 byte packets to communicate mouse movement,( mouse button press/release events). The PS2 mouse won't give us any special data to represent double clicks. The best way to check if a double click occur is by looking at the time difference between first and second click. If the time difference is less, then we threat is a double click.

We didn't have specific output for the MOUSE driver but we understood the coding during its implementation.

## ❖ Audio

Generating Sound was complicated which wasn't resolved since the **qemu-system-i386 -soundhw pcspk -drive format=raw, file=bin\os-image** kept on displaying errors about the sound pack.

```
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>nasm boot.asm -f bin -o bin\bootsect.bin
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>nasm IDT.asm -f elf -o bin\IDT.bin
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>nasm ata.asm -f elf -o bin\ata.bin
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o bin\IDT.
bin bin\ata.bin
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>objcopy -O binary -j .text bin\kernel.img bin\kernel.bin
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
bin\bootsect.bin
bin\kernel.bin
1 file(s) copied.
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>qemu-system-i386 -soundhw pcspk -drive format=raw,file=bin\os-image
qemu-system-i386: -soundhw: invalid option
C:\Users\djeut\OneDrive\Desktop\OS PROJECT>pause
Press any key to continue . . .
```

Here we've an overview of the problem faced when running the sound pack. To get back to our normal command line we just need to remove the **-soundhw pcspk** which is normally there to get a sound.



## **CONCLUSION**

Though the numerous difficulties encountered like the AUDIO and FILE SYSTEM, we succeeded in implementing the basic parts of our operating system.