**BLOCKS OS**

**An Engineering Project in Community Service**

**Phase–II Report**

***Submitted by***

| **SERIAL NO** | **NAME** | **REGISTRATION NUMBER** |
| --- | --- | --- |
| 1 | Anjali Singh | 19BCG10003 |
| 2 | Abhishek Srivastava | 19BCE10071 |
| 3 | V Surya Kumar | 19BCE10286 |
| 4 | Pratul Maurya | 19BCY10036 |
| 5 | Pravir Kadian | 19BCE10006 |
| 6 | Saransh Pratap Singh | 19BCY10035 |
| 7 | Viplav Khubchandani | 19BAI10106 |
| 8 | C S Soujanya Mudliar | 19BCG10094 |

***in partial fulfilment of the requirements for the degree of***

***Bachelor of Engineering and Technology***

****

**VIT Bhopal University**

**Bhopal**

**Madhya Pradesh**

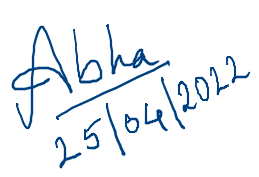
**21 April 2022**

****

**Bonafide Certificate**

Certified that this project report titled **“BLOCKS OS”** is the bonafide work of **19BCG10003 Anjali Singh, 19BCE10071 Abhishek Srivastava, 19BCE10286 V Surya Kumar, 19BCE10006 Pravir Kadian, 19BCY10036 Pratul Maurya, 19BCY10035 Saransh Pratap Singh, 19BAI10106** **Viplav Khubchandani, 19BCG10094 C.S. Soujanya Mudliar** who carried out the project work under my supervision.

This project report (Phase II) is submitted for the Project Viva-Voce examination held on 21st April 2022.



Dr. Abha Trivedi

**Supervisor**

**Comments & Signature (Reviewer 1)**

**Comments & Signature (Reviewer 2)**

|  |  |  |
| --- | --- | --- |
| **Serial Number** | **Topic** | **Page Number** |
| 1 | Introduction | 4 |
| 1.1 | Motivation | 4 |
| 1.2 | Objective | 4 |
| 2 | Existing Work | 5-6 |
| 3 | Topic of the Work | 6-12 |
| 3.1 | System Design / Architecture | 6-8 |
| 3.2 | Working Principle | 8-10 |
| 3.3 | Results and Discussion | 10-12 |
| 4 | My Contribution | 13-15 |
| 5 | Conclusion | 16 |
| 6 | References | 16 |
| 7 | Plagiarism Report | 16 |

# 

**INTRODUCTION**

An operating system is one of the major parts of an undergraduate computer science education that majorly has two main approaches to learning. The first one is theoretical concepts of working algorithms and the second one is the OS architecture and how it is built. In both cases the implementation part is something that lags, leaving students only with theoretical knowledge unknown about how in the real world these concepts are implemented using various high level and assembly level languages.

BLOCK -OS is a teaching-based open-source operating system with 32-bit system architecture. It is a simple and small os built-in C and assembly language that teaches one how to implement basic concepts of operating systems from scratch providing one with experience of in-depth understanding of concepts of operating systems.

The concepts of operating systems are implemented and divided into various modules to ease down the process of understanding and implementing more clearly. Various modules that are implemented in BLOCK OS are Boot sector and boot loader, GDT (Global Descriptor Table), IDT (Interrupt Descriptor Table), Keyboard Driver, Graphic Driver, Main memory (ATA), and audio. To run BLOCK-OS we need an x86 emulator, QEMU.

**MOTIVATION:**

The main driving force of the project was to get the experience of working on a low-level project. This was done using the knowledge of core subjects like OS, C programming, and assembly language to create a project that enhances our level of understanding and serves functionality using the same.

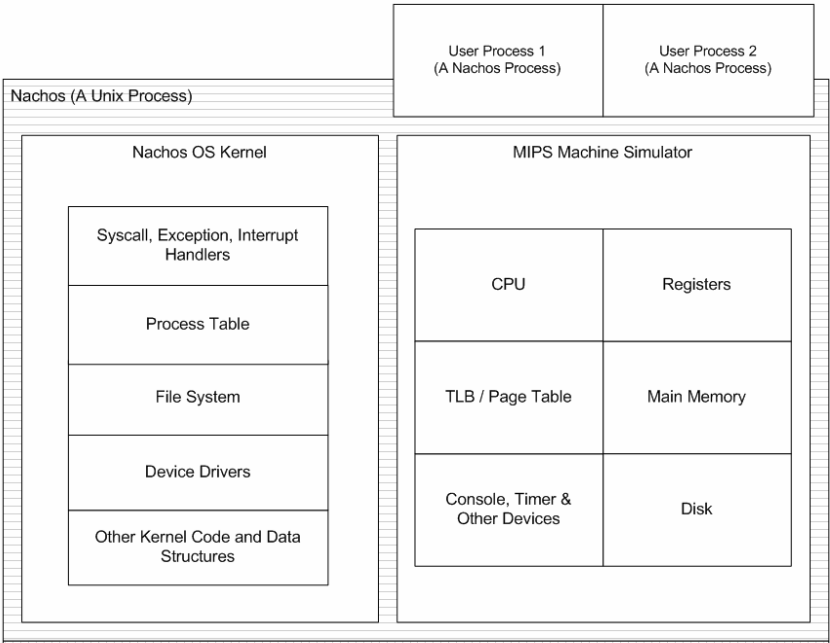
**OBJECTIVE:**

There are various factors in choosing a project whose research resources were not widely available. First off, for the group to understand the core working and structure of working of the Operating Systems with the implementation. Moreover, enhancing research skills was another aspect to build a project that would function as building blocks for the student developer community and aspiring OS Architects while learning at the same time.

**LITERATURE REVIEW:**

An operating system is a set of programs that acts as an interface between a user and computer hardware. OS is one of the base subjects for the field of computer science. To understand and know in-depth understanding of OS already various projects and small OS are created.

NachOS, one example of an operating system developed by the developer community or by developers, is used in the implementation of instructional operating systems. It operates as a standard UNIX process. Functionalities implemented in NachOS include: threads and remote procedure calls (thread management), file systems, virtual memory, multiprogramming and networking as shown below:



**NACH-OS ARCHITECTURE**

PintOS, is an operating system framework for 80x86 architecture, inspired by the NachOS educational operating system from the University of California, Berkeley, is another example of an operating system. PintOS is written in C language and at a basic level, it implements several capabilities such as kernel threads support, loading and running user programmes, and a file system. In a system simulator, PintOS may be readily executed.

PintOS and NachOS are similar in design, but PintOS has certain advantages over NachOS. First, unlike most real-world operating systems, PintOS runs on hardware that uses 80x86 microprocessors, rather than running as a process on the host operating system. Second, unlike most operating systems, PintOS is written in C++, rather than C.

Further, we have used Wiki OS Dev and a book by TINU TOM as main resources for developing this project. Wiki OS Dev contains information about the development of operating systems and serves as a community for those interested in OS development, with 693 wiki articles. However, while Wiki OS Dev might be useful to provide an overview of the functionality of each operating system, it lacks practical details.

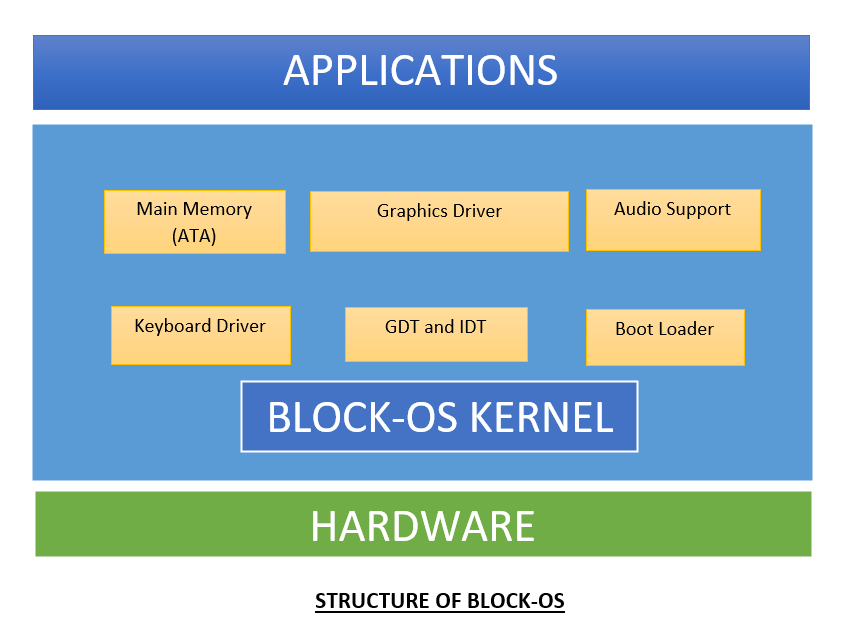
TINU TOM’s book, “Developing a Computer Operating System from Scratch”, connects the Wiki OS Dev articles and provides a more guided approach for an operating system development.

After reviewing and studying various existing operating systems, we decided to design our new one based on an existing OS with some added features and detailed documentation.

**TOPIC OF WORK:**

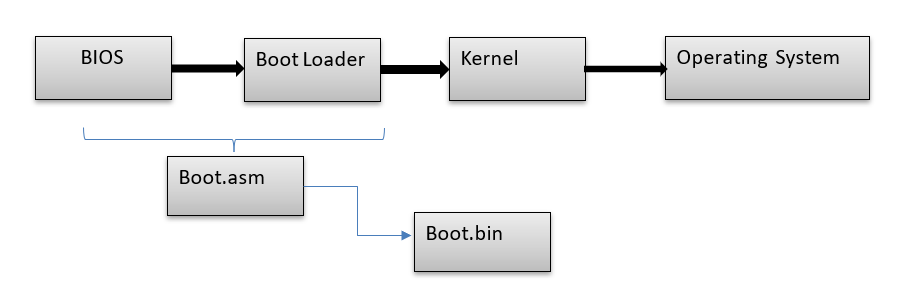
**a) System Design / Architecture**

BLOCK-OS is built on a monolithic architecture. This architecture allows our operating system kernel to access all the resources within the system as all the elements and modules are contained inside the kernel only. The overall architecture of BLOCK-OS is shown below:

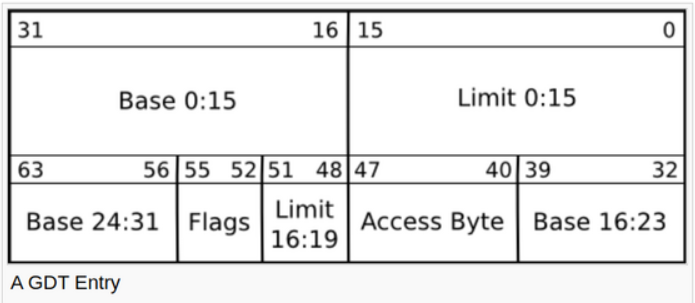
****

The monolithic structure approach allows BLOCK-OS to efficiently use resources and allow them to directly communicate with each other easily. This in turn makes our operating system fast as it allows a method to run fully in a very single address space.

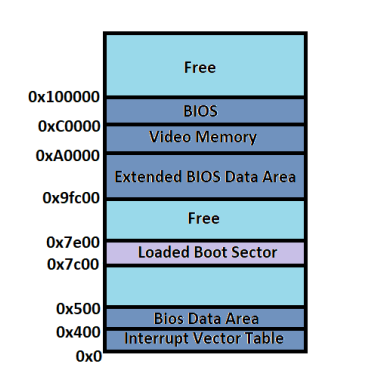
Boot sector is an important part of building an operating system. Boot sector is implemented in the boot.asm file that creates a file name boot.bin during its execution. The overall linking of BIOS and boot sector is shown below in the diagram:



We have implemented protected mode with the help of GDT (Global Descriptor Table). Each segment in Protected mode is implemented with the help of GDT using the Flat memory model. The structure of GDT entry is as shown below:

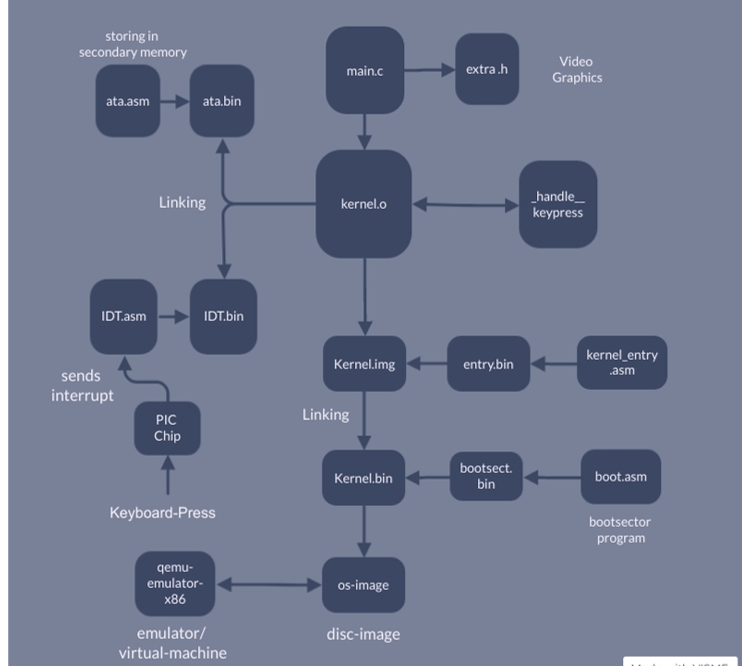


The code and other resources are present in the memory as shown in below diagram:



**b) Working Principle**

As we followed monolithic architecture each module file is connected with kernel file i.e. **kernel.o** as shown in overall directory structure below:

****

When BLOCK-OS starts, in **Kernel\_entry.asm** file it goes to command:

**jmp $**

**Boot Sector and Boot Loader:**

BIOS is the first piece of code that is loaded from ROM to main memory (RAM). After that BIOS loads the first 512 bytes from the hard disk that is the boot sector. This command initially refers to command in the **boot.asm** file where the initial magic numbers are initialised that starts the boot loader (as boot sector is initialised in boot.asm file: first 512 bytes of the sector).

**Defining GDT and Switch to Protected Mode:**

32-bit operating systems enable the user mode to have more memory protection than in 16-bit ones. If a program could access these special locations where the Kernel of the OS resides, they could take control of the CPO so our OS would not have any control.

Therefore, to switch to 32-bit i.e. protected mode we first implemented GDT(Global descriptor table) in which each element of protected mode has its defined location and properties are stored. We first implemented GDT loading and definition in the **boot.asm** file only and then switched to protected mode finally.

**Support for C:**

To add support for C, we declared a command in **Kernel.asm** file:

**[extern\_start]**

**call\_start**

When the linker encounters code (extern\_start) that references a symbol named \_start, it will attempt to search for the \_start in an external object file i.e., a C file (**main.c**) and link it in with the resulting executable.

**Graphic Driver:**

Various printf commands and colour change functions have been implemented using the graphic driver. The graphic driver functions are implemented in C as various functions in **main.c** file as explained below:

**void cls();** // to clear screen

**void setMonitorColor(char);** //to set color of monitor screen

**void printString(char\*);** //to print string the to screen

**void printChar(char);** //to print character to the screen

**void scroll();** //scrolling activity

**void printColorString(char\* , char);** //to print string of particular background and foreground color to the screen

**void printColorChar(char , char);** //to print character of particular background and foreground color to the screen

**PIC, IDT and Keyboard Driver:**

The **start()** function in the **main.c** file calls the initIDT() function, which sets up the computer's Interrupt Descriptor Table so that user input is routed to the **isr1\_Handler()** function whenever a key on the keyboard is pressed.

The PIC is used to generate interrupts. When a key in the keyboard is pressed, the chip inside the keyboard tells the PIC to generate an interrupt. The PIC will then notify the CPU about the interrupt. When the CPU receives this message and detects that a key has been pressed, it executes a set of code we pre-set in the CPU to execute when a key is pressed.

In the main.c file, we have a function named **start()**. We call the **initIDT()** function from within this function to set the Interrupt Descriptor Table (IDT) so that when a key is pressed, it calls the **isr1\_Handler()** function.

We used the **IDT\_ENTRY** structure to define an IDT entry. There are 256 possible entries, so we created 256 of them with the **IDT\_ENTRY idt[256]**.

**Hard Disk:**

We implemented a hard disk driver for ATA technology, so that hard disks and CD-ROMs could be internally connected to the motherboard and perform input/output functions.

We implemented two main commands **PUT()** and **GET()**.The PUT command copies the string of J’s to At and writes it to the hard disk. The GET command reads from the hard disk and outputs the string of J’s previously.

**Audio:**

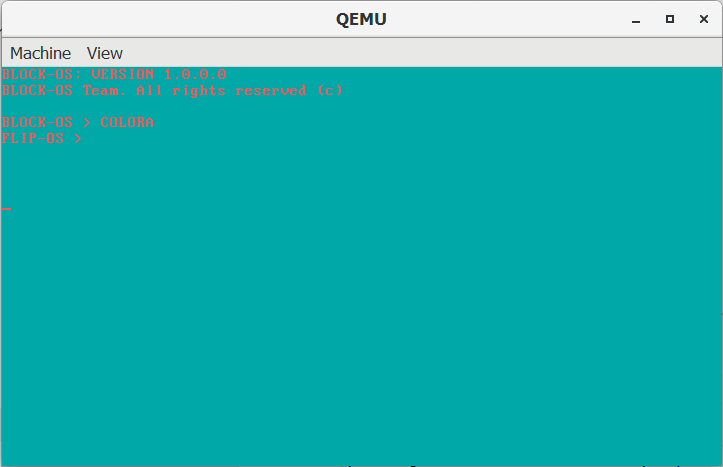
In the **extra.h** file we defined scream() function in which frequency bits are passed as an argument and executes it until stop() is executed.

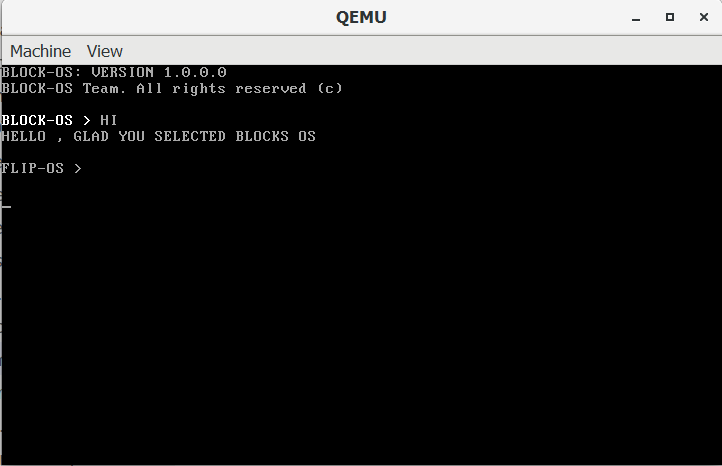
When a user enters PLAY() in the command line , the audio plays.

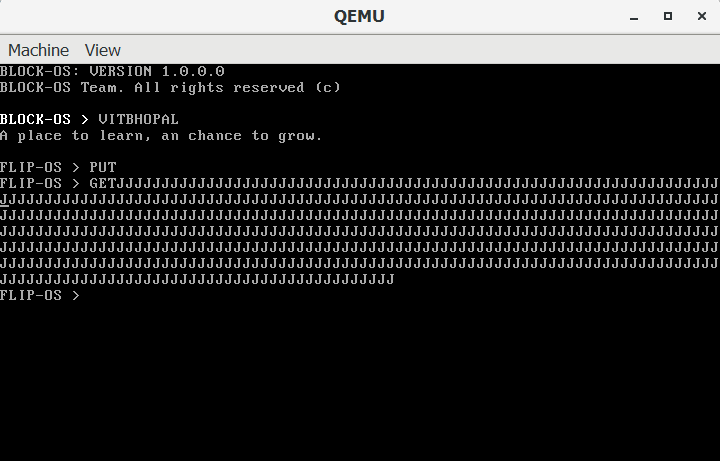
**c) Results and Discussion:**

As a result, we were able to implement following features in our BLOCK-OS that are:

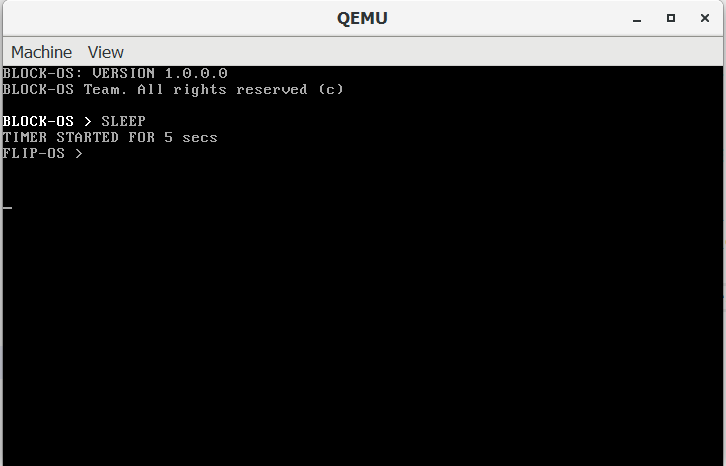
* Set up the BIOS and our kernel at initial level.
* Boot Sector and Boot Loader for booting process.
* Implementation of GDT (Global descriptor table).
* Switch to 32-bit protected mode from 16-bit mode.
* Added support for C.
* Implemented a Graphic Driver that displays graphics on screen.
* Added functions like: COLORA (for changing screen monitor color) , VITBHOPAL(to display messages on the screen) etc.







* Implemented ISR (Interrupt Descriptor Table).
* Set up a Keyboard driver using ISR and PIC.
* Hard disk writing and reading function using ATA.
* Set up audio support using the PLAY() command.
* Global constructor.
* Programmable Interval Timer.



Apart from implementing all these features as a result of processed development simultaneously we were able to create well-defined documentation of BLOCK-OS that will serve as a base for teaching faculties to assist them to provide students with in-depth knowledge along with hands-on experience on implementation of basic operating system concepts.

Also, this project is open-source. Hence, providing student developers to use this code as a boiler-plate code for initial stage understanding and further development.

**MY CONTRIBUTION:**

**(a) Boot Sector:**

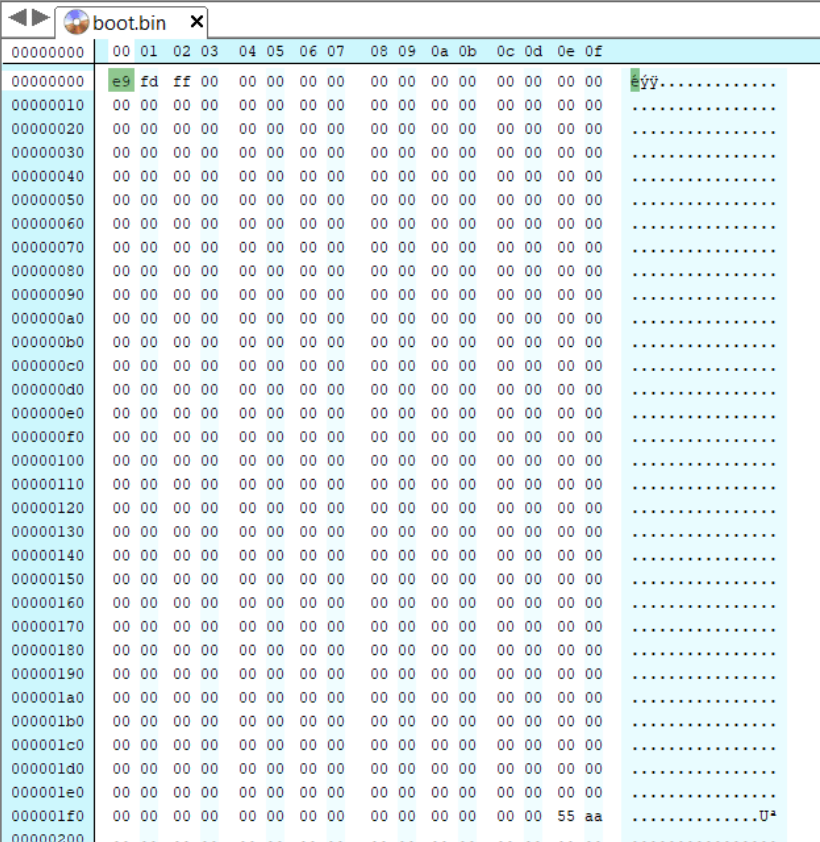
We implemented boot sector by initialising first 510 byte of hard disk as 0 and then initialising last two bytes with value aa and 55 as shown in code below:

**times 510-($-$$) db 0**

**dw 0xaa55**

These last two bytes are known as **magic numbers.** Magic numbers are first or last bits of any file that is unique to any particular file type and helps to identify a particular file without any extension also.

I used a magic number as the last two bytes so that whenever the computer starts the booting process it first checks whether the last two numbers are magic numbers or not. If it is a magic number then only it starts executing the whole code. We can see these magic numbers using hex editor as shown below:



**(b) Protected Mode and GDT implementation:**

Since we wanted to implement an operating system in 32 bits, we switched to Protected Mode i.e. 32 bit operation mode that allows us to use memory up to 4GB.

In protected mode, we also defined an area of memory that is directly accessible by user mode and that is not using GDT (Global Descriptor Table). Global descriptor table is a data structure that is mainly used in intel x86 family processors. Each segment in protected mode has a set of properties and defined location, that is defined in GDT where descriptor- lists all properties of a segment.

There are three types of memory model available to implement GDT that are: Flat memory model, paging and segmentation. We have implemented GDT using the **Flat memory model** that is a single contiguous space model.

GDT must contain two parts that are data segment and code segment. In each segment we have specific two parts that are: base→ defines location and limit→ defines size. We have implemented GDT with the following code below:

**lgdt [GDT\_DESC]** //Loads GDT  
**GDT\_BEGIN:**

**GDT\_NULL\_DESC:** //The Mandatory Null Descriptor

**dd 0x0**

**dd 0x0**

Then we have implemented data segment and code segment as shown below:

**GDT\_CODE\_SEG:**

**dw 0xffff** //Limit

**dw 0x0** //Base

**db 0x0** //Base

**db 10011010b** //Type Flags

**db 11001111b** //Other Flags

**db 0x0 ;Base**

**GDT\_DATA\_SEG:**

**dw 0xffff**  //Limit

**dw 0x0** //Base

**db 0x0** //Base

**db 10010010b** //Type Flags

**db 11001111b** //Other Flags

**db 0x0 ;Base**

Switch to protected mode:

**mov eax , cr0**

**or eax , 0x1** //xor with 1 to change last bit of cr0 as 1

**mov cr0 , eax**

**(c) Boot Loader:**

Boot loader is a part of the boot sector that loads the remaining part of the operating system. I have implemented a boot loader in c file which loads 30 sectors in main memory.

In order to execute this c file we introduced START() function in our c file and implemented a linker file that automatically calculates where the entry function is :

**START:**

**[bits 32]**

**[extern \_start]**

**call \_start**

**jmp $**

**(d) Audio:**

A speaker can have positions "in" and "out". A setting of 1 directs the speaker to move to the "out" position, and a setting of 0 directs the speaker to move to the "in" position. Moving in and out repeatedly produces audible tones if the speed of repetition is within the range that the speaker can produce and that the human ear can hear.

The PC Speaker can be connected directly to the port 0x61 of the PIC peripheral timer 2 by setting bit 0 (=1). In this mode, when the PIC peripheral timer 2 "ticks", the speaker moves to the "out" position. Likewise, when the PIC peripheral timer 2 "ticks", the speaker moves to the "in" position. By changing the frequency at which PIC peripheral timer 2 "ticks", the PC Speaker can be made to output sound of a frequency equal to that of its ticks.

To implement this we have three functions implemented in extra.h file that are:

**void scream(int fr);** // passes frequency of sound to be played

**void play(unsigned int fr );** // plays the sound

**void stop();** //stops the sound

When a user enters the PLAY() command then the audio gets played.

**Results and Discussion:**

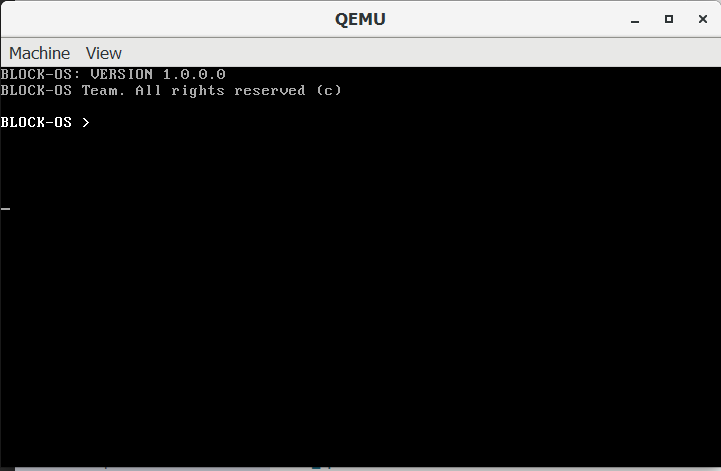
I have researched various operating system related topics like how the boot sector works, what is the magic number, GDT, etc to implement these all features in our operating system.

I have implemented a boot sector using magic number and boot loader using C that resulted in successfully loading our operating system to main memory.

I have also switched our OS from 16 bit mode to 32 bit mode i.e. protected mode. As a result, switching to protected mode allowed us to write our code in C and flexibility to implement higher concepts for the future like paging and multitasking.

Implemented GDT that allowed us to define which part of our OS is directly accessible by the user in user mode and which part is not accessible, providing better security and control to our kernel and OS.

I also implemented audio support that gets triggered when the PLAY() command is entered by the user.



**CONCLUSION:**

We have written an open-source operating system “BLOCK-OS” that is capable of performing basic operating system functionalities along with some added functions. This operating system project serves as a project that provides student developers with in-depth knowledge of operating systems that is a core subject for computer science undergraduates.

Further talking about future development various modules can be added on this depending on the feasibility of time and process of development.

**REFERENCES:**

1. <https://wiki.osdev.org/Main_Page>
2. <https://github.com/cfenollosa/os-tutorial>
3. <https://drive.google.com/file/d/1qviDBcRat4fuSdgx_IYXii2_ZFFmZK8Y/view?usp=sharing>
4. <https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html>
5. <https://cseweb.ucsd.edu/classes/sp09/cse120/projects/nachos.pdf>
6. <https://student.cs.uwaterloo.ca/~cs350/common/os_overview.html>
7. <https://littleosbook.github.io/>
8. <https://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf>

**PLAGIARISM REPORT:**

(Report is in 2 parts due to word limit 1500 in plagiarism checkers).

<https://drive.google.com/drive/folders/19OEX9tMRVaHTAR0a4Nf9ByFiGbU9XQ0V?usp=sharing>