# A PROJECT REPORT ON

# OPERATING SYSTEM DEVELOPMENT

*Submitted in partial fulfillment of the requirements for the award of the degree of*

# Bachelor of Technology

In

# Computer Science And Engineering

*By*

**Betsy L Kalathoor**
**Jom Jose**
**Krishnaprasad K**


**FEDERAL INSTITUTE OF SCIENCE AND TECHNOLOGY (FISAT)**
**ANGAMALY-683577, ERNAKULAM (DIST)**
***Affiliated to***
**MAHATMA GANDHI UNIVERSITY**
**Kottayam-686560**

**May 2012**

# FEDERAL INSTITUTE OF SCIENCE AND TECHNOLOGY (FISAT)


## CERTIFICATE

Certified that project work entitled "**OPERATING SYSTEM DEVELOPMENT**" is a bonafide work carried out by **Betsy L Kalathoor, Jom Jose, Krishnaprasad K** in partial fulfilment for the award of Bachelor of Technology in Computer Science and Engineering from Mahatma Gandhi University, Kottayam, Kerala during the academic year 2011-2012.

**Mr. Prasad J C**
*Head of the Dept*
*Asst. Professor*
*Dept of Computer Science and Engineering*
*FISAT*

**Ms. Divya T V**
*Project Guide*
*Asst. Professor*
*Dept. of Computer Science and Engineering*
*FISAT*

## ABSTRACT

The project aims at developing an operating system using only assembly language.The OS is a 32 bit DOS like oprating system,which has a Character User Interface for user interaction.The OS is equipped with a kernel which is only capable handling basic operations.The OS is a single tasking,single user operating system.The operating system has a bootloader which loads the kernel.The Operating system contains an application for playing a piano on the keyboard by pressing keys on the keyboard.Also a screensaver application is present inorder to save the screen.

## ACKNOWLEDGMENT

If words are considered as symbols of approval and tokens of acknowledgement,then let the words play the heralding role of expressing our gratitude.First and foremost,we praise the God almighty for the grace he showered on us during our studies as well as our day to day activities

We are extremely thankful to the Chairman,**Mr.P.V.Mathew**,who provided us withall the vital facilities required for the successful completion of the project.

We also thank our Principal, **Dr. K S M Panicker**, for the amenities he provided, which helped us in the fulfilment of our project. We would like to express our sincere thanks to our beloved **Prof. Prasad J.C (H.O.D)** who always guidedus and rendered his help in all the phases of our project.

We would like to pay our gratitude to project in charge, **Mr. Mahesh C**,for his constant support and encouragement and for guiding us with patience and enthusiasm in all the stages.We are extremely thankful to **Ms. Divya T.V**, who always extended help and support for the successful completion of our project.

We would like to express our sincere gratitude to all the faculties of the Computer Science Department, FISAT. We would like to express special thanks to our Lab instructors for providing us with all the necessary Lab facilities and helping us throughout this project to make it a success.

We also sincerely thank our parents and friends for giving us moral support and encouragement in all possible ways.

Betsy L Kalathoor
Jom Jose
Krishnaprasad K

# Contents

# List of Figures

# Chapter 1

# INTRODUCTION

## 1.1 Overview

An operating system (OS) is a set of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs require an operating system to function.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and will frequently make a system call to an OS function or be interrupted by it. Operating systems can be found on almost any device that contains a computerfrom cellular phones and video game consoles to supercomputers and web servers.

At the foundation of all system software, the OS performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking, and managing files. It also may provides a character user interface for basic functioning and a graphical user interface for higher level functions.

Various services performed by operating systems are discussed below.

**Process management**
It deals with running multiple processes. Most operating system allow a process to be assigned a priority which affects its allocation of CPU time. Interactive operating systems also employ some level of feedback in which the task with which the user is working receives higher priority. In many systems there is a background process which runs when no other process is waiting for the CPU.

**Memory management**
The memory manager in an OS coordinates the memories by tracking which one is available, which is to be allocated or deallocated and how to swap between the main memory and secondary memories. The operating system tracks all memory used by each process so that when a process terminates, all memory used by that process will be available for other processes.

**Disk and file systems**
Operating systems have a variety of native file systems that controls the creation, deletion, and access of files of data and programs.

**Networking**
Most current operating systems are capable of using the TCP/IP networking protocols. This means that one system can appear on a network of the other and share resources such as files, printers, and scanners. Many operating systems also support one or more vendor-specific legacy networking protocols as well.
**Security**
Most operating systems include some level of security.
**Device drivers**
A device driver is a specific type of computer software developed to allow interaction with hardware devices. Typically this constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications. Examples of popular modern operating systems include Android, BSD, iOS, Linux, Mac OS X, Microsoft Windows, Windows Phone, and IBM z/OS. All these, except Windows and z/OS, share roots in UNIX.

## 1.2 Types of Operating Systems

### Real-time

A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main objective of real-time operating systems is their quick and predictable response to events. They have an event-driven or time-sharing design and often aspects of both. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

### Multi-user

A multi-user operating system allows multiple users to access a computer system concurrently. Time-sharing system can be classified as multi-user systems as they enable a multiple user access to a computer through the sharing of time. Single-user operating systems, as opposed to a multi-user operating system, are usable by a single user at a time. Being able to use multiple accounts on a Windows operating system does not make it a multi-user system. Rather, only the network administrator is the real user. But for a UNIX-like operating system, it is possible for two users to login at a time and this capability of the OS makes it a multi-user operating system.

### Multi-tasking vs. Single-tasking

When only a single program is allowed to run at a time, the system is grouped under a single-tasking system. However, when the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system. Multi-tasking can be of two types: pre-emptive or co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programs. Unix-like operating systems such as Solaris and Linux support pre-emptive multitasking, as does AmigaOS. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. 16-bit versions of Microsoft Windows used cooperative multi-tasking. 32-bit versions, both Windows NT and Win9x, used pre-emptive multi-tasking. Mac OS prior to OS X used to support cooperative multitasking.

### Distributed

A distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

### Embedded

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

## 1.3 BROS VERSION 1.0

BROS VERSION 1.0 was venture towards developing an operating system exploring the finer details of how a layer of software could takecare of the technical aspects of a computers operation and mediate access between physical and application layers of a system.

BROS is a Single-Tasking operating system developed from the bootloader stage to a Chatacter User Interface(CUI) level , where the user executes basic operations. It has a basic 32-bit kernal without complex input output functionalities.
Basic components of BROS are:

### Bootloader

Bootloaders are small pieces of software that play a role in getting an operating system loaded and ready for execution when a computer is turned on. The way this happens varies between different computer designs (early computers required

a person to manually set the computer up whenever it was turned on), and often there are several stages in the process of boot loading.It's crucial to understand that the term "bootloader" is simply a classification of software (and sometimes a blurry one). To the processor, a bootloader is just another piece of code that it blindly executes. There are many different kinds of boot loaders. Some are small, others are large; some follow very simple rules while others show fancy screens and give the user a selection to choose from.On IBM PC compatibles, the first program to load is the Basic Input/Output System (BIOS). The BIOS performs many tests and initialisations, and if everything is OK, the BIOS's boot loader begins. Its purpose is to load another boot loader! It selects a disk (or some other storage media) from which it loads a secondary boot loader.In some cases, this boot loader loads enough of an operating system to start running it. In other cases, it loads yet another boot loader from somewhere else. This often happens when multiple operating systems are installed on a single computer; each OS may have its own specific bootloader, with a "central" bootloader that loads one of the specific ones according to the user's selection.Most bootloaders are written exclusively in assembly language (or even machine code), because they need to be compact, they don't have access to OS routines (such as memory allocation) that other languages might require, they need to follow some unusual requirements, and they make frequent use of low-level features. However some bootloaders, particularly those that have many features and allow user input, are quite heavyweight. These are often written in a combination of assembly and C. The GRand Unified Bootloader (GRUB) is an example of such.Some boot loaders are highly OS-specific, while others are less so - certainly the BIOS boot loader is not OS-specific. The MS-DOS boot loader (which was placed on all MS-DOS formatted floppy disks) simply checks if the files IO.SYS and MSDOS.SYS exist; if they are not present it displays the error "Non-System disk or disk error" otherwise it loads and begins execution of IO.SYS.The final stage boot loader may be expected (by the OS) to prepare the computer in some way, for instance by placing the processor in protected mode and programming the interrupt controller. While it would be possible to do these things inside the OS's initialisation procedure, moving them into the bootloader can simplify the OS design. Some operating systems require their bootloader to set up a small, basic GDT (Global Descriptor Table) and enter protected mode, in order to remove the need for the OS to have any 16-bit code. However, the OS might replace this with its own sophisticated GDT soon after.
A bootloader runs under certain conditions that the programmer must appreciate in order to make a successful bootloader. The following pertains to bootloaders initiated by the PC BIOS:

- The first sector of a drive contains its boot loader.
- One sector is 512 bytes the last two bytes of which must be 0xAA55 (i.e. 0x55 followed by 0xAA), or else the BIOS will treat the drive as unbootable.
- If everything is in order, said first sector will be placed at RAM address 0000:7C00, and the BIOS's role is over as it transfers control to 0000:7C00. (I.e. it JMPs to that address) The DL register will contain the drive number

that is being booted from, useful if you want to read more data from elsewhere on the drive.

- The BIOS leaves behind a lot of code, both to handle hardware interrupts (such as a keypress) and to provide services to the bootloader and OS (such as keyboard input, disk read, and writing to the screen). You must understand the purpose of the Interrupt Vector Table (IVT), and be careful not to interfere with the parts of the BIOS that you depend on. Most operating systems replace the BIOS code with their own code, but the boot loader can't use anything but its own code and what the BIOS provides. Useful BIOS serve include int 10h (for displaying text/graphics), int 13h (disk functions) and int 16h (keyboard input).

- This means that any code or data that the boot loader needs must either be included in the first sector (be careful not to accidentally execute data) or manually loaded from another sector of the disk to somewhere in RAM. Because the OS is not running yet, most of the RAM will be unused. However, you must take care not to interfere with the RAM that is required by the BIOS interrupt handlers and services mentioned above.

- The OS code itself (or the next bootloader) will need to be loaded into RAM as well.

- The BIOS places the stack pointer 512 bytes beyond the end of the boot sector, meaning that the stack cannot exceed 512 bytes. It may be necessary to move the stack to a larger area.

- There are some conventions that need to be respected if the disk is to be readable under mainstream operating systems. For instance you may wish to include a BIOS Parameter Block on a floppy disk to render the disk readable under most PC operating systems.

**Kernel**

The kernel is the core of an operating system. It is the software responsible for running programs and providing secure access to the machine's hardware. Since there are many programs, and resources are limited, the kernel also decides when and how long a program should run. This is called scheduling. Accessing the hardware directly can be very complex, since there are many different hardware designs for the same type of component. Kernels usually implement some level of hardware abstraction (a set of instructions universal to all devices of a certain type) to hide the underlying complexity from applications and provide a clean and uniform interface. This helps application programmers to develop programs without having to know how to program for specific devices. The kernel relies upon software drivers that translate the generic command into instructions specific to that device. An operating system kernel is not strictly needed to run a computer. Programs can be directly loaded and executed on the "bare metal" machine, provided that the authors of those programs are willing to do without any hardware abstraction or operating system support. This was the normal operating method of many early computers, which were reset and reloaded between the running of different programs. Eventually, small ancillary programs such as program loaders and debuggers

were typically left in-core between runs, or loaded from read-only memory. As these were developed, they formed the basis of what became early operating system kernels. The "bare metal" approach is still used today on many video game consoles and embedded systems, but in general, newer systems use kernels and operating systems.

The basic facilities provided by a kernel include :

- Process management:The main task of a kernel is to allow the execution of applications and support them with features such as hardware abstractions. A process defines which memory portions the application can access.Kernel process management must take into account the hardware built-in equipment for memory protection.To run an application, a kernel typically sets up an address space for the application, loads the file containing the application's code into memory (perhaps via demand paging), sets up a stack for the program and branches to a given location inside the program, thus starting its execution.

- Memory management:The kernel has full access to the system's memory and must allow processes to safely access this memory as they require it. Often the first step in doing this is virtual addressing, usually achieved by paging and/or segmentation. Virtual addressing allows the kernel to make a given physical address appear to be another address, the virtual address. Virtual address spaces may be different for different processes; the memory that one process accesses at a particular (virtual) address may be different memory from what another process accesses at the same address. This allows every program to behave as if it is the only one (apart from the kernel) running and thus prevents applications from crashing each other.

- Device management:To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. A device driver is a computer program that enables the operating system to interact with a hardware device. It provides the operating system with information of how to control and communicate with a certain piece of hardware. The driver is an important and vital piece to a program application. The design goal of a driver is abstraction; the function of the driver is to translate the OS-mandated function calls (programming calls) into device-specific calls. In theory, the device should work correctly with the suitable driver. Device drivers are used for such things as video cards, sound cards, printers, scanners, modems, and LAN cards.

- System calls:A system call is a mechanism that is used by the application program to request a service from the operating system. They use a machine-code instruction that causes the processor to change mode. An example would be from supervisor mode to protected mode. This is where the operating system performs actions like accessing hardware devices or the memory management unit. Generally the operating system provides a library that sits between the operating system and normal programs. Usually it is a C library such as Glibc or Windows API. The library handles the low-level details of passing information to the kernel and switching to supervisor mode. System calls include close, open, read, wait and write.To actually perform useful work, a process

must be able to access the services provided by the kernel. This is implemented differently by each kernel, but most provide a C library or an API, which in turn invokes the related kernel functions.The method of invoking the kernel function varies from kernel to kernel. If memory isolation is in use, it is impossible for a user process to call the kernel directly, because that would be a violation of the processor's access control rules.

**Shell**
A shell is a piece of software that provides an interface for users of an operating system which provides access to the services of a kernel. The shell used here is a Command Line Shell that provides a Command Line Interface(CLI) to the user. It allows communication with the operating system via a control language letting the user control the peripherals without understanding the characteristics of the hardware used , management of the physical addresses etc.

**QEMU**
QEMU is an open source emulator for complete PC systems. In addition to emulating a processor, QEMU permits emulation of all necessary subsystems, such as networking and video hardware. It also permits emulation of advanced concepts, such as symmetric multiprocessing systems (up to 255 CPUs) and other processor architectures, such as ARM or PowerPC. QEMU supports two operating modes: user-mode emulation and system-mode emulation. User-mode emulation allows a process built for one CPU to be executed on another (performing dynamic translation of the instructions for the host CPU and converting Linux system calls appropriately). System-mode emulation allows emulation of a full system, including processor and assorted peripherals. When x86 code is being emulated on an x86 host system, near native performance can be achieved using what is called the QEMU accelerator. This permits execution of the emulated code directly on the host CPU (on Linux through a kernel module). But what makes QEMU interesting from a technical perspective is its fast and portable dynamic translator.QEMU achieves dynamic translation by first converting target instructions into micro operations. These micro operations are bits of C code that are compiled into objects. The core translator is then built. It maps the target instructions to micro operations for dynamic translation. This is not only efficient, but also portable.

# Chapter 2

# RELATED WORK

## 2.1 MenuetOS

MenuetOS is an Operating System in development for the PC written entirely in 32/64 bit assembly language. Menuet64 is released under License and Menuet32 under GPL. Menuet supports 32/64 bit x86 assembly programming for smaller, faster and less resource hungry applications. Menuet isn't based on other operating system nor has it roots within UNIX or the POSIX standards. The design goal, since the first release in year 2000, has been to remove the extra layers between different parts of an OS, which normally complicate programming and create bugs. Menuet's application structure isn't specifically reserved for asm programming since the header can be produced with practically any other language. However, the overall application programming design is intended for 32/64 bit asm programming. Menuet programming is fast and easy to learn. Menuet's responsive GUI is easy to handle with assembly language. And Menuet64 is capable of running Menuet32 applications.

- Pre-emptive multitasking with 1000hz scheduler, multithreading, multiprocessor, ring-3 protection - Responsive GUI with resolutions up to 1920x1080, 16 million colours
- Free-form, transparent and skinnable application windows, drag'n drop
- SMP multiprocessor support with currently up to 8 cpus
- IDE: Editor/Assembler for applications
- USB 2.0 Hi-speed storage, webcam, printer class and TV/Radio support
- USB 1.1 Keyboard and mouse support
- TCP/IP stack with Loopback  Ethernet drivers
- Email/ftp/http/chess clients and ftp/mp3/http servers
- Hard real-time data fetch
- Fits on a single floppy, boots also from CD and USB drives

## 2.2 MikeOS

MikeOS is an operating system for x86 PCs, written in assembly language. It is a learning tool to show how simple OSes work, with well-commented code and extensive documentation. Features:

A text-mode dialog and menu-driven interface

Boots from a floppy disk, CD-ROM or USB key

Over 60 system calls for use by third-party programs

File manager, text editor, image viewer, game...

Includes a BASIC interpreter with 42 instructions

PC speaker sound and serial terminal connection

## 2.3 BareMetal OS

BareMetal is a 64-bit OS for x86-64 based computers. The OS is written entirely in Assembly, while applications can be written in Assembly or C/C++. Development of the Operating System is guided by its 3 target segments:

- High Performance Computing - Act as the base OS for a HPC cluster node. Running advanced computation workloads is ideal for a mono-tasking Operating System.
- Embedded Applications - Provide a platform for embedded applications running on commodity x86-64 hardware.
- Education - Provide an environment for learning and experimenting with programming in x86-64 Assembly as well as Operating System fundamentals.

BareMetal boots via Pure64 and has a command line interface with the ability to load programs/data from a hard drive. Current plans for v0.6.0 call for basic TCP/IP support, a native File System, as well as general bug fixes and optimizations. The creation of BareMetal was inspired by MikeOS - A 16-bit OS written in Assembly used as a learning tool to show how simple Operating Systems work.

## 2.4 DexOS

DexOS is a x86 32-bit console like operating system, coded in 100Now when most people think of consoles they think of playing games, that is true, but these very same console can just as easy be used for web browsing, interfacing with electronics, writting supper fast apps etc. From the start, as you would expect from a console like operating system, optimizing for speed has been of paramount importance in

the overall design of dexos. To this end there's no virtual memory paging, and only a single process is allowed (though that process can spawn multiple threads). The program runs in ring0, so you have direct access to all hardware (including CPU and graphics). Memory allocation is the responsibility of the app - there's no front-end memory allocation. The entire operating system will fit into less than 64KB of memory. dexos can boot from a floppy disk, cd, usb flash drive or directly from hard drive. It has lots of built-in functions and libraries to help programmers. You can boot without touching the underlining OS. A small team of coders are working on tools to help you develop applications There's an IDE, a Fasm port, and a full TCP/IP stack implemented, plus a console like GUI.

Dexos is one of the fastest operating system available for the x86 PC! Everything in this system is focused on speed: the architecture, avoidance of virtualization ... everything. And of course, there's the choice of development language: dexos is written in 100%-pure 32-bit assembly!

Because of its clear, modular architecture, we hope it will become a great development platform. The functionality of all 'parts' of the system can be easily accessed from your applications and if you miss a part, it's very easy to add it later. That's the idea behind the architecture of dexos. Everything can be connected, and everything is replaceable. We invite everyone to take a test drive, and start using the OS as a development platform for your applications or for educational purposes. The use of the system is completely free, all we hope is that you post the programs you have created back to us, so they can be shared with the rest of the world .

## 2.5 TomOS

TomOS is a simple 16 bit operating system for x86. It is based on MikeOS version 2.0.0. It is entirely written in the assembly language. TomOS is an educational project.

- Mouse support.
- 8KB RAM available for applications.
- FAT 12 support.
- Approximately 90 system calls.

# Chapter 3

# DESIGN

## 3.1 Introduction

Design is the first step in the development phase for an engineered product or system. It may be defined as the process of applying various techniques and principles for the purpose of defining a devise, a process or a system in sufficient detail to permit its physical realization. Computer software design like engineering design approaches in other disciplines changes continually as new methods, better analysis, and broader understanding evolve.

Using one of a number of design methods the design step produces a data design,an architectural design and a procedural design. Preliminary design is concerned with the transformation requirements to data and software architectures. Detail design Focus on refinements to architectural representation for software. The data design transforms the information domain model created during analysis in to the data structures that will be required to implement the software. The architectural design defines the relationship among major structural components in to a procedural description of the software.

### 3.1.1 Elements of Design

The components of an information system described during requirement analysis are the focal point in system design. Analysis must design the following elements

Data flows

- Data stores
- Processes
- Procedures
- Controls
- Roles

Data flow analysis permits to isolate areas of interest in the organization and to study them by examining the data that enter the process and seeing how they are changes when they leave the process. In the Data Flow diagrams, the physical system is translated into a logical description that focuses on data and processes. It is advantageous to emphasis data and processes in order to focus on the actual activities that occur and the resources needed to perform them, rather than on who performs the work.

### 3.1.2 Logical Designs

In the logical design, description of the inputs, outputs, databases and procedures are given I a format that meets the requirements. Data Flow Diagrams A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system. A Data Flow Diagram can also be used for the visualization of data processing (structured design). It is common practice for a designer to draw a context-level DFD first which shows the interaction between the system and outside entities. This context-level DFD is then "exploded" to show more details of the system being modelled.

Data Flow Diagrams (DFDs) are one of the three essential perspectives of Structured System Analysis and Design Method SSADM. The sponsor of a project and the end users will need to be briefed and consulted throughout all stages of a system's evolution.With a Data Flow Diagram, users are able to visualize how the system will operate, what the system will accomplish, and how the system will be implemented. The old system's dataflow diagrams can be drawn up and compared with new systems Data Flow Diagrams to draw comparisons to implement a more efficient system. Data Flow Diagrams can be used to provide the end user with a physical idea of how data is transferred in the system.
The four key elements in a Data Flow Diagram are Processes, Data flows, Data Stores and External entities.

The process entity identifies a process taking place; it must have at least one input and output. A process with no input is known as a miracle process an output is a black hole process.
The Data Flow entity identifies the flow of data between processes, data stores and external entities. A data flow cannot connect an external entity to a data source; at least one connection must be with a process. There are also physical flows, i.e. those that use a physical medium like membership card.
The Data Store entity identifies stores of data, both manual and electronic. Electronicor digital stores are identified by the letter D, and manual filing systems by the letter M, e.g. D1 could be a SQL Server database, and M$ could be filing cabinet.
The External Entity identifies external entities which interact with the system, usually clients but can be within the same organization. Multiple existences of the same entity, e.g. the same doctor shown twice on the same diagram, can be identified by a horizontal line in the top left corner of the symbol. The various DFD levels are context level, level 0 and level 1.

The context level shows the overall context of the system and its operating environment and shows the whole systems as just one process. It does not usually show data stores, unless they are "owned" by external systems, e.g. are accessed by but not maintained by this system, however, these are often shown as external entities.

Level 0 shows all processes at the first level of numbering, data stores, external entities and the data flows between them. The purpose of this level is to show the major high level processes of the system and their interrelation. A process model will have one, and only one, level 0 diagram. A level 0 diagram must be balanced with its parent context level diagram, i.e. there must be the same external entities and the same data flows, these can be broken down to more details in the level 0, e.g. the "enquiry" data flow could be split into "enquiry" results and still be valid.

Level 1 is a decomposition of a process shown in a level 0 diagrams, as such there should be a level 1 diagram for each and every process shown in a level 0 diagram. In this examples process 1.1, 1.2, and 1.3 are all children of process 1, together they wholly and completely describe process 1, and combined must perform the full capacity of this parent process. As before, a level 1 diagram must be balanced with its parent level 0 diagrams.

Data Flow Diagram (DFD) is used to define the flow of the system and its resources such as information. Data Flow Diagrams are a way of expressing system requirements in a graphical manner. DFD represents one of the most ingenious tools used for structured analysis. A DFD is also known as a bubble chart. It has the purpose of clarifying system design.

### 3.1.3 Physical Design

**Input Design and Output Design**

Input design is the process of converting user inputs into computer-based format. The project requires a set of information from the user to prepare a report. In order to prepare a report, when-organized input data are needed.

In the system design phase, the expanded DFD identifies logical data flow, data stores and destination. Input data is collected and organized into groups of similar data. The goal behind designing input data is to make the data entry easy and make it free from logical errors.
**Objectives**

- To produce a cost-effective method of input.
- To achieve the highest possible level of accuracy.
- To increase that the input is acceptable and understandable.

Outputs are the most important and direct source of information to the user and to the management. Efficient and eligible output design should improve the systems relationship with the user and help in decision making.

### 3.1.4 Code Design

A code is an ordered collection symbols to provide unique identification of data. Codes can be used by people who do not with data processing; the following are characters of a good code generation.

**Characteristics of Good Coding**

- Uniqueness
- Meaningfulness
- Stability
- Uniform size and Format
- Simplicity
- Conciseness

## 3.2 Design of Modules

The project is divided into following modules:

### 3.2.1 Bootsector

The First and the most important part of the OS ,which helps the OS to get started is the Bootsector.A boot sector or boot block is a region of a hard disk, floppy disk, optical disc, or other data storage device that contains machine code to be loaded into random-access memory (RAM) by a computer system's built-in firmware. The purpose of a boot sector is to allow the boot process of a computer to load a program stored on the same storage device. The location and size of the boot sector is specified by the design of the computing platform.Since its the Kernel thats to be loaded next,the bootsectors purpose here is to find that second stage bootsector and load it in to the memory.

At the completion of the system's Power On Self Test (POST), INT 19 is called. Usually INT 19 tries to read a boot sector from the first floppy drive. If a boot sector is found on the floppy disk, the that boot sector is read into memory at location

0000:7C00 and INT 19 jumps to memory location 0000:7C00.Since the OS is written on the floppy disk,the boot sector is found on the floppy disk.

The Boot section module consistes of a collection of submodules which are as below:

### Loading the File Allocation Tables(FATs)

This section of the Bootsector loads the FATs.File Allocation Table (FAT) is the name of a computer file system architecture and a family of industry standard file systems utilizing it.The FAT contains mappings to the clusters.The name of the file system originates from the file system's prominent usage of an index table, the FAT, statically allocated at the time of formatting. The table contains entries for each cluster, a contiguous area of disk storage. Each entry contains either the number of the next cluster in the file, or else a marker indicating end of file, unused disk space, or special reserved areas of the disk. The root file directory of the disk contains the number of the first cluster; the operating system can then traverse the FAT table, looking up the cluster number of each successive part of the disk file as a cluster chain until the end of the file is reached.Clusters are basically a series of contiguous sectors on the disk.Since the second stage bootloader is needed to be loaded,the FATs needed to be loaded first so that the clusters containing the second stage bootloaded can be effectively found out and the program control can be transferred to the corresponding starting location of the Kernel.

### Loading the Root Directory Table (RDT)

The root directory is a table of 32 byte values that represent information reguarding file and directories.Since as a part of the design the name of the second stage bootloader is already known,the RDT can be used to find the cluster numbers of the Kernel by searching the RDT with the name of the Kernel file.Once the Kernel file is successfully searched,then the cluster number is used as a key to the FATs and doing so will yeald the clusters required.

### Jumping to the Kernel Address

On the Succcessfull loading of the sectors containing the Kernel,the progra control can be transferred to the starting address of the kernel.This address is specified while loading the sectors containing the kernel.

### Specifying the boot signature

To be a valid boot sector, the two-byte hexadecimal sequence 0x55,0xAA ,called the boot sector signature,must exist at the end of the sector - if not, either the BIOS or MBR code will report an error message and halt any OS bootstrapping process.The boot signature marks the end of the boot sector and this is the final submodule of the bootsector module.

### 3.2.2 A20 Line

The A20 Address Line is the physical representation of the 21st bit (number 20, counting from 0) of any memory access. When the IBM-AT (Intel 286) was introduced, it was able to access up to sixteen megabytes of memory (instead of the 1 MByte of the 8086). But to remain compatible with the 8086, a quirk in the 8086 architecture (memory wraparound) had to be duplicated in the AT. To achieve this, the A20 line on the address bus was disabled by default. The wraparound was caused by the fact the 8086 could only access 1 megabyte of memory, but because of the segmented memory model they could effectively address up to 1 megabyte and 64 kilobytes (minus 16 bytes). Because there are twenty address lines on the 8086 (A0 through A19), any address above the 1 megabyte mark wraps around to zero. For some reason a few short-sighted programmers decided to write programs that actually used this wraparound (rather than directly addressing the memory at its normal location at the bottom of memory). Therefore in order to support these 8086-era programs on the new processors, this wraparound had to be emulated on the IBM AT and its compatibles; this was originally achieved by way of a latch that by default set the A20 line to zero. Later the 486 added the logic into the processor and introduced the A20M pin to control it. For an operating system developer (or Bootloader developer) this means the A20 line has to be enabled so that all memory can be accessed. This started off as a simple hack but as simpler methods were added to do it, it became harder to program code that would definitely enable it and even harder to program code that would definitely disable it. The traditional method for A20 line enabling is to directly probe the keyboard controller. The reason for this is that Intel's 8042 keyboard controller had a spare pin which they decided to route the A20 line through. This seems foolish now given their unrelated nature, but at the time computers weren't quite so standardized. Keyboard controllers are usually derivatives of the 8042 chip. By programming that chip accurately, we can either enable or disable bit 20 on the address bus.

### 3.2.3 Global Descriptor Table

This modules defines the Global descriptor table(GDT).The Global Descriptor Table or GDT is a data structure used by Intel x86-family processors starting with the 80286 in order to define the characteristics of the various memory areas used during program execution, including the base address, the size and access privileges like executability and writability. These memory areas are called segments in Intel terminology.The GDT can hold things other than segment descriptors as well. Every 8-byte entry in the GDT is a descriptor, but these can be Task State Segment (or TSS) descriptors, Local Descriptor Table (LDT) descriptors, or Call Gate descriptors. The last one, Call Gates, are particularly important for transferring control between x86 privilege levels although this mechanism is not used on most modern operating systems.There is also an LDT or Local Descriptor Table. The LDT is supposed to contain memory segments which are private to a specific program, while the GDT is supposed to contain global segments. The x86 processors contain facilities for automatically switching the current LDT on specific machine events, but no fa-

cilities for automatically switching the GDT.Every memory access which a program can perform always goes through a segment. On the 386 processor and later, because of 32-bit segment offsets and limits, it is possible to make segments cover the entire addressable memory, which makes segment-relative addressing transparent to the user.In order to reference a segment, a program must use its index inside the GDT or the LDT. Such an index is called a segment selector or selector in short. The selector must generally be loaded into a segment register to be used. Apart from the machine instructions which allow one to set/get the position of the GDT (and of the Interrupt Descriptor Table) in memory, every machine instruction referencing memory has an implicit Segment Register, occasionally two. Most of the time this Segment Register can be overridden by adding a Segment Prefix before the instruction.Loading a selector into a segment register automatically reads the GDT or the LDT and stores the properties of the segment inside the processor itself. Subsequent modifications to the GDT or LDT will not be effective unless the segment register is reloaded.

### 3.2.4 Protected Mode

In this module the system jumps from the real mode to the protected mode.In protected mode operation, the x86 can address 16 Mb or 4 GB of address space. This may map directly onto the physical RAM (in which case, if there is less than 4 GB of RAM, some address space is unused), or paging may be used to arbitrarily translate between virtual addresses and physical addresses. In Protected mode, the segments in memory can be assigned protection, and attempts to violate this protection cause a "General Protection" exception.Protected mode in the 386, amongst other things, is controlled by the Control Registers, which are labelled CR0, CR2, CR3, and CR4.Protected mode in the 286 is controlled by the Machine Status Word. Until the release of the 386, protected mode did not offer a direct method to switch back into real mode once protected mode was entered. IBM created a workaround which involved resetting the CPU via the keyboard controller and saving the system registers, stack pointer and often the interrupt mask in the real-time clock chip's RAM. This allowed the BIOS to restore the CPU to a similar state and begin executing code before the reset.Later, a Triple fault was used to reset the 286 CPU, which was a lot faster and cleaner than the keyboard controller method.To enter protected mode, the Global Descriptor Table (GDT) must first be created with a minimum of three entries: a null descriptor, a code segment descriptor and data segment descriptor. The 21st address line (A20 line) also must be enabled to allow the use of all the address lines so that the CPU can access beyond 1 megabyte of memory (only the first 20 are allowed to be used after power-up to guarantee compatibility with older software). After performing those two steps, the PE bit must be set in the CR0 register and a far jump must be made to clear the prefetch input queue.

### Programmable Interrupt Controller

A programmable interrupt controller (PIC) is a device that is used to combine several sources of interrupt onto one or more CPU lines, while allowing priority levels to be assigned to its interrupt outputs. When the device has multiple interrupt outputs to assert, it will assert them in the order of their relative priority.The function of the 8259A is actually relatively simple. Each PIC has 8 input lines, called Interrupt Requests (IRQ), numbered from 0 to 7. When one of these lines goes high, the PIC alerts the CPU and sends the appropriate interrupt number. This number is calculated by adding the IRQ number (0 to 7) to an internal "vector offset" value. The CPU uses this value to execute an appropriate Interrupt Service Routine. (For furthur information, see Advanced Interrupts). Of course, it's not quite as simple as that, because each system has two PICS, a "master" and a "slave". So when the slave raises an interrupt, its actually sent to the master, which sends that to the CPU. In this way, interrupts cascade and a processor can have 16 IRQ lines. Of these 16, one is needed for the two PIC chips to interface with eachother, so the number of available IRQs is decreased to 15. While cli and sti can be used to enable and disable all hardware interrupts, it's sometimes desirable to selectively disable interrupts from certain devices. For this purpose, PICs have an internal 8-bit register called the Interrupt Mask Register (IMR). The bits in this register determine with IRQs are passed on to the CPU. If an IRQ is raised but the corresponding bit in the IMR is set, it is ignored and nothing is sent to the CPU.In this module the PIT is remapped specifically for the interrupts uses.That is, changing their internal vector offsets, thereby altering the interrupt numbers they send. The initial vector offset of PIC1 is 8, so it raises interrupt numbers 8 to 15. Unfortunately, some of the low 32 interrupts are used by the CPU for exceptions (divide-by-zero, page fault, etc.), causing a conflict between hardware and software interrupts. The usual solution to this is remapping the PIC1 to start at 32, and often the PIC2 right after it at 40. This requires a complete restart of the PICs, but is not actually too difficult, requiring just eight outs.

### Interrupt Service Routine

The x86 architecture is an interrupt driven system. External events trigger an interrupt - the normal control flow is interrupted and a Interrupt Service Routine (ISR) is called.Such events can be triggered by hardware or software.For the system to know which interrupt service routine to call when a certain interrupt occurs, offsets to the ISR's are stored in the Interrupt Descriptor Table when you're in Protected mode, or in the Interrupt Vector Table when you're in Real Mode.When an exception occurs, a program invokes an interrupt, or the hardware raises an interrupt, the processor uses one of several methods to transfer control to the ISR, whilst allowing the ISR to safely return control to whatever it interrupted after execution is complete. At minimum, FLAGS and CS:IP are saved and the ISR's CS:IP loaded; however, some mechanisms cause a full task switch to occur before the ISR begins (and another task switch when it ends).Since the interrupt vector table we use is the default table defined by the BIOS,only the IDT needs to be defined which is done in the next module.

### Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is specific to the I386 architecture. It is the Protected mode counterpart to the Real Mode Interrupt Vector Table (IVT) telling where the Interrupt Service Routines (ISR) are located. It is similar to the Global Descriptor Table in structure.Use of the IDT is triggered by three types of events: hardware interrupts, software interrupts, and processor exceptions, which together are referred to as "interrupts". In general ,the IDT consists of 256 interrupt vectorsthe first 32 (0-31 or 00-1F) of which are reserved for processor exceptions.The IDT only comes into play when the processor is in protected mode. Much like the IVT, the IDT contains a listing of pointers to the ISR routine;, however, there are now three ways to invoke ISRs:

- Task Gates: These cause a task switch, allowing the ISR to run in its own context (with its own LDT, etc.). Note that IRET may still be used to return from the ISR, since the processor sets a bit in the ISR's task segment that causes IRET to perform a task switch to return to the previous task.
- Interrupt Gates: These are similar to the original interrupt mechanism, placing EFLAGS, CS and EIP on the stack. The ISR may be located in a segment of equal or higher privilege to the currently executing segment, but not of lower privilege (higher privileges are numerically lower, with level 0 being the highest privilege).
- Trap Gates: These are identical to interrupt gates, except they do not clear the interrupt flag.

### Programmable Interval Timer

The Programmable Interval Timer (PIT) chip (also called an 8253/8254 chip) basically consists of an oscillator, a prescaler and 3 independent frequency dividers. Each frequency divider has an output, which is used to allow the timer to control external circuitry (for example, IRQ 0).The frequecy of the sound produced by the PC Speaker is determined by the PIT.Since the PIANO app used this , PIT needs to be configured often throughout the execution.

### Keyboard Setup

Basically, when a key is pressed, the keyboard controller tells a device called the Programmable Interrupt Controller, or PIC, to cause an interrupt. Because of the wiring of keyboard and PIC, IRQ 1 is the keyboard interrupt, so when a key is pressed, IRQ 1 is sent to the PIC. The role of the PIC will be to decide whether the CPU should be immediately notified of that IRQ or not and to translate the IRQ number into an interrupt vector (i.e. a number between 0 and 255) for the CPU's

table. The OS is supposed to handle the interrupt by talking to the keyboard, via in and out instructions, asking what key was pressed, doing something about it (such as displaying the key on the screen, and notifying the current application that a key has been pressed), and returning to whatever code was executing when the interrupt came in.This module sets up the keyboard layout which tells the basic layout of the keyboard and it is used by the ISR for keyboard intterupt to recognize the key which is pressed.

### 3.2.5 Shell

The Shell module displays a basic shell and is used by the user to execute commands.The shell exected the commands using calls to the kernel.The shell in this OS accepts the following commands:

- Help
- About
- Gettime
- Settime
- Piano
- Screensaver
- Restart

#### Gettime and Settime

The time module is used to display the system time and date or set the system time or the system date.The user can execute the gettime command to show the date and time .Also this module is used in conjunction with the shell module to show the system time at the fast end of the line showing the shell .

#### Piano Application

The Piano module accepts keystrokes and produces MIDI tones corresponding to the key pressed.Upto 4 octaves of the Piano are being mapped to four layers of the keyboard keys.This module uses the PIT module to set the appropriate frequencies and produces the right sound.

#### Screensaver Application

The screensaver module displays a marquee on a blank screen.This text keeps on floating untill the tilde( ) is pressed.This module saves the entire screenbuffer to the memory on the beginning of the command execution and restores it at the end of

execution.This gives the user a feel that the system resumes at the same checkpoint where is stopped for the screensaver.

## 3.3 Control Flow Diagram

**Control flow diagram(CFD)** is a diagram to describe the control flow of a business process, process or program. They are one of the classic business process modeling methodologies, along with flow charts,data flow diagrams, functional flow block diagram, Gantt charts,PERT diagrams.

A control flow diagram can consist of a subdivision to show sequential steps, with if-then-else conditions, repetition, and/or case conditions. Suitably annotated geometrical figures are used to represent operations, data, or equipment, and arrows are used to indicate the sequential flow from one to another. How any system is developed can be determined through a Data Flow Diagram, There are different notations to draw DFDs, defining different visual representations for processes, data stores, data flow,and external entities.

## 3.4 Data Flow Diagram

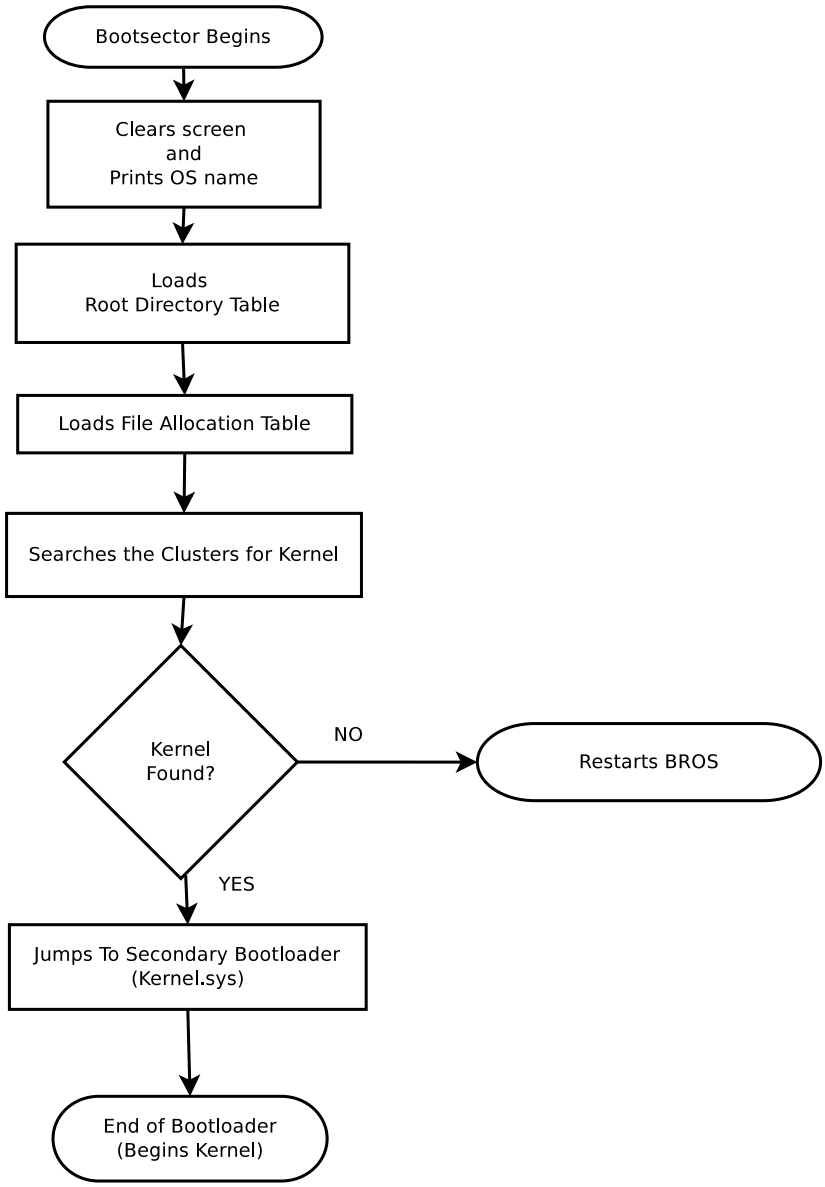


Figure 3.1: Stage 1

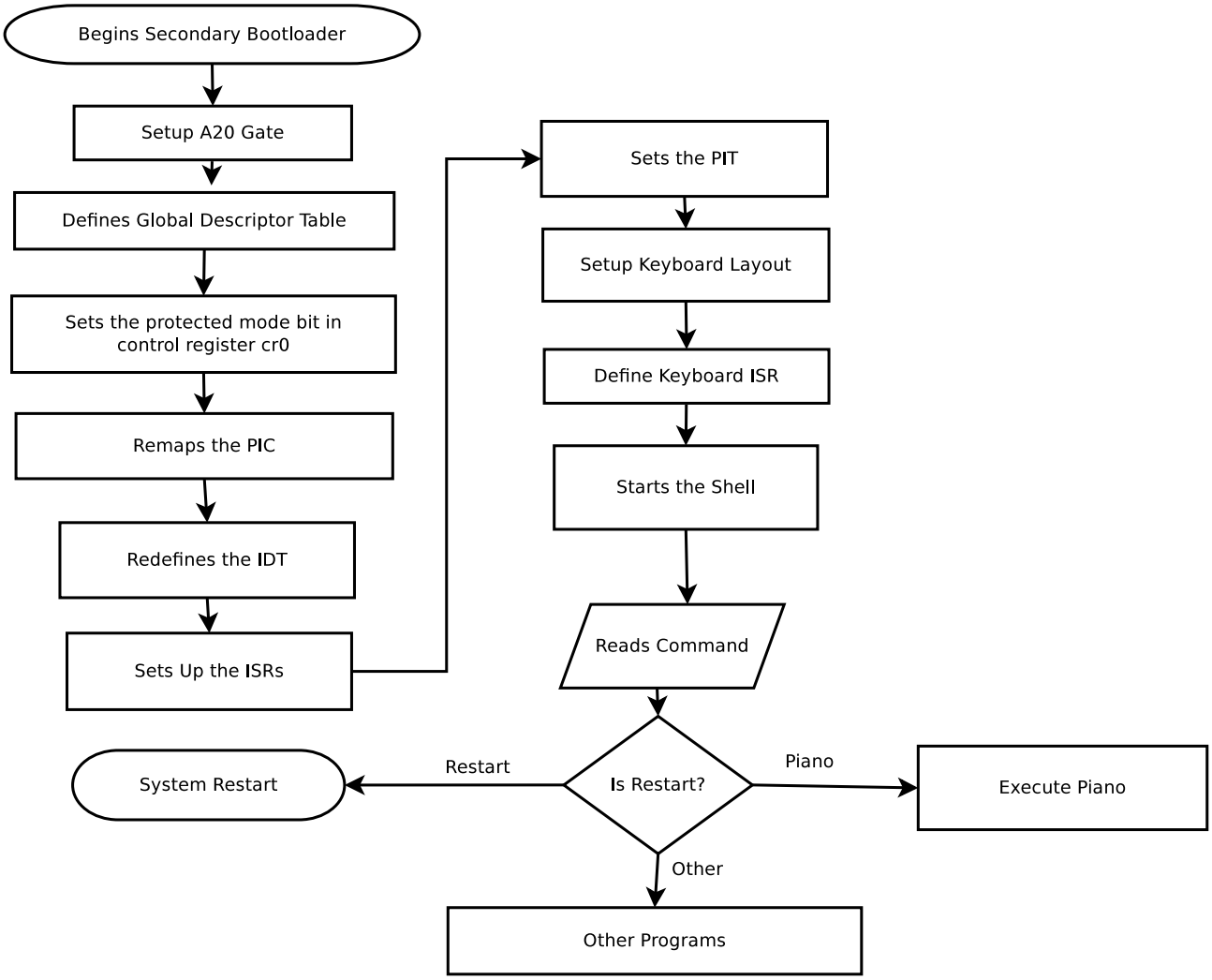


Figure 3.2: Stage 2

# Chapter 4

# IMPLEMENTATION AND TESTING

## 4.1 Implementation

Implementation is an activity that is contained throughout the development phase. It is a process of bringing a developed system into operational use and turning it over to the user. The new system and its components are to be tested in a structured and planned manner. A successful system should be delivered and users should have confidence that the system would work efficiently and effectively. The more complex the system being implemented the more involved will be the system analysis and design effort required for implementation.

There are three types of implementation that exist:

- Implementation of a computer system to replace a manual system. The problems encountered are converting files, training users, creating accurate files, and verifying printouts for integrity.
- Implementation of a new computer system to replace an existing one. This usually a difficult conversion. If not properly planned, there can be many problems. Some large computer systems have taken as long as year to convert.
- Implementation of a modified application to replace an existing one using the same computer. This type of conversion is relatively easy to handle, provided there are no major changes in the files.

### 4.1.1 Implementation Plan

Initailly the bootloader needs to be implemented.Only then the rest of the system can be coded.Since assembly language is used and there is no IDE , error debugging will be difficult and therefore we plan to code the rest slowly and steadly.After implementing the bootloader,kernel needs to be implemented and then the applications

such as piano.
Basic steps are as follows:

- Make the Bootloader
- Set up the protected mode
- Make a shell interface
- Do the Piano Application
- Test the System for errors

## 4.2 Testing

Testing is the phase of software development in which the software to be tested is executed with a set of test cases and the behavior of the system for the test cases is evaluated to determine if the system is performing as expected. Testing is a critical activity in software engineering and should be performed as systematically as possible by stating clearly what result one expects from it and how one expects to obtain that result. The success of testing in revealing errors depends on the test cases. Testing should help locate errors, not just detect their presence. Test should be organized in a way that helps isolate errors. A program testing can be used to show the presence of bugs, but never to show the absence. If the results delivered by the system are different from the expected ones even in one case, this unequivocally shows that the system is incorrect; by contrast a correct number of cases do not guarantee correctness in the general case. Thus testing should be considered only one of the means to analyze the behavior of a system and should be integrated with other verification techniques in order to enhance our confidence in system qualities as much as possible.

### 4.2.1 System testing

A test plan has the following steps:

- Prepare test plan
- Specify conditions for user acceptance testing
- Prepare test data for program testing
- Prepare test data for transaction path testing
- Plan user training
- Compile/assemble programs
- Prepare job performance aids
- Prepare operational documents

### 4.2.2 Test Cases

The different levels of testing are used in the process, each level of testing aims to test the different aspects of the system. It is important to perform testing at each level. The first level of testing is the unit testing in which the modules are tested by the individual developing it. A module is taken for integration testing only after unit testing is done. Then the system and acceptance testing are performed. These different levels of testing attempt to detect different types of faults. In case of this project, test data are stored in the existing database. Then data migration process is carried on for each of the table separately. After this, data migration operation is performed on multiple tables simultaneously. This is known as pre-migration test. Its only after that the migration plan is applied on the real database to perform data migration.

#### Unit testing

Here the different modules are tested against the specification produced during the design of the modules. Unit testing is essential for verification of the code produced during the coding phase and hence the internal logic of the modules is tested. In it the programmer of the particular module does the testing. A module is con- sidered for integration and used by the others only after it has been unit tested satisfactorily. Unit testing comprises the set of tests performed by an individual programmer prior to integration of the unit into a larger system. There are four categories of tests that can be performed on a program unit.

- Functional unit
- Performance unit
- Stress unit
- Structure unit

In unit testing, module interface is tested to ensure that the information properly into and out of the program under test. Local datastructures are examined to ensure that data stored temporarily maintains its integrity during all steps in algorithm execution. Boundary condition is tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths through the control structures are executed atleast once. Error handling paths are also tested. This test focuses verification effort on the smallest unit of software design, the module. Here the module interfaces local data structure, boundary conditions, and all independent paths and last but not the least,all error handling paths were verified by inputting false data. Tests of dataflow across each module interface of this software were done before any test was initiated.

### Integration testing

In this, many unit tested modules were compiled into subsystems which were then tested. The goal is to see if the modules can be integrated properly. Hence the emphasis is on testing interfaces between modules. This testing activity can be con- sidered as testing the design. Integration testing is a systematic technique for constructing the program struc- ture while at the same time conducting tests to uncover errors associated with interfacing.Unit testing modules were taken and a single program structure was built that has been dictated by the design.Incremental integration has been adopted here. The entire software was developed and testing in small segments, where errors were easy to locate and rectify.Program builds(groups of modules) were constructed corresponding to the successful testing of user interaction, data manipulation anal- ysis, and display processing and database management. Integration testing involves bottom-up integration, top-down integration and sandwich integration strategy. Bottom-up integration the traditional strategy used to integrate the components of software system into a functioning whole. Top-down integration starts with main routine and one or two immediate subroutine in the system structure. Sandwich integration is predominantly top-down, but bottom-up techniques are used in some modules and sub systems.

### Acceptance testing

Acceptance testing is performed with realistic data of the client to demonstrate that the software is working satisfactorily. Testing here focuses on the external behavior of the system, the internal logic of the program is not emphasized. Mostly functional testing is performed at these levels. These levels of testing are performed when a system is being built from the components that have been coded. Acceptance testing involves planning and execution of functional tests, performance tests and stress tests in order to demonstrate the implemented system satisfies its requirements. Functional tests causes involve exercising the code with nominal files for which the expected outputs are known. Thus the software system developed in the above manner is one that satisfies the user needs, confirms to its requirements and design satisfactions and exhibits an absence of errors.

### System Testing

System testing is performed to verify that all system elements have been properly integrated and perform allocated functions. Security testing was done to check the security mechanism built into the system, which will protect it from improper penetration, performance testing was done to test the runtime performance of the software. For user acceptance testing the system was given to the end user to use. The errors found are rectified.

**Validation Testing**

Validation testing is done to ensure complete assembly of the error-free software.Validation can be said to be successful only if it functions in manner. Reasonably expected by the customer under validation is alpha and beta testing. Alpha testing is where end user tests the system rather than the developer, but in a controlled environment. The software is used on a natural setting with the developer monitoring the user using the system. The developer records the errors and usage problems encountered by the user. The sales person conducts the beta testing at one or more sites. The developer is not present during these tests. Hence, beta tests can be said as the live application of the software on an environment ttohat cannot be controlled by the developer. The sales person takes down the probles encountered during beta testing and reports these to the developer at regular intervals. The developer makes suitable modifications to the software henceforth. Validation testing is done to ensure complete assembly of the error-free software. Validation can be said to be successful only if it functions in manner. Reasonably expected by the customer under validation is alpha and beta testing. Alpha testing is where end user tests the system rather than the developer, but in a controlled environment.The software is used on a natural setting with the developer monitor- ing the user using the system. The developer records the errors and usage problems encountered by the user.

**Syntax testing**

Syntax testing is used to eliminate errors in software. In the system we have input fields like text, numeric fields. User should allow only numeric fields and should avoid any alphabets. The program wont accept any alphabets in a numeric field.

System testing involves unit testing, integration testing, acceptance testing. Careful planning and scheduling are required to ensure that modules will be available for integration into the evolving software.

### 4.2.3 Testing Process

BROS is tested using QEMU.QEMU stands for Quick EMUlator and is a processor emulator that relies on dynamic binary translation to achieve a reasonable speed while being easy to port to new host CPU architectures.The testing is done as show in the steps below:

- Compiling the assembly code:Basically two assembly files should be compiled in order to get the OS working.The compilation id done using the FASM assembler.
- Creating a Image file for installation:An iso image file is used to mount the OS.This image file is used by the QEMU as a virtual disk.The compiled binary files should be written in to this ISO image file and then exectued by the QEMU.

- Mounting ISO image and copying binaries :Now the ISO image needs to be mounted on to the filesystem and then the binaries are copieed in to the mounted ISO using the 'dd' command in unix system.
- Executing OS using QEMU:The QEMU is used to run the OS from the ISO file.It uses the command '*qemu -soundhw pcspk -fda bos.img*'.

## 4.3 Maintenance

Once the software is fully developed and implemented, the company starts to use the software. The company also grows and more divisions can be attached to the company, or the database of the company can grow in size. So after sometime the software, which has been installed, needs some modification. If the software needs modification all the steps needed to develop new software has to be executed.

The need has to be studied, the design has to be made and coding has to be done. Then the new module has to be connected to the existing software modules. Once the software is working perfect also we have to do routine testing and any new bug is found out, immediately it has to be fixed. No software ever developed will be bug free forever. When a new situation arises, the software can create an error, but if its found out and repaired the software will not be causing more problems. Always maintenance has to be done on the software, for to make the software work perfectly without any errors.

# Chapter 5

# CONCLUSION AND FUTURE WORK

Using the assembly language BROS was successfully developed and capable of single tasking.The OS can be used mainly as a piano interface using which user can play MIDI music.Instead of being limited to a 16bit operating system,BROS is a 32 operating system although its full utilisation is not yet made.The unavailability of mouse support and a Graphical User Interface is not a burder because the OS is developed to be User friendly in a Character User Interface Enviornment.

## 5.1 Future Works

The BROS is a 32bit Operating System.Since the entire code is in assembly languages the BROS has its limitations.The Following are the possible improvements to be added to the BROS in the future:

- Implement Multitasking
- Implement a Login security system
- Adding support for multiboot
- Support for HardDisk and CDRom
- Upgrade to 64bit OS
- Support for USB Drives
- Network Support
- GUI and Mouse support
- Implementing process,Memory Management
- Scheduling Algorithms
- Adding More Applications

Apart from all these ,the main task it to port the OS to a C language interface for more Flexibility and ease of programming.

## Bibliography


[1] . http://wiki.osdev.org/Expanded$_{M}ain_{P}age$

[2] . http://www.brokenthorn.com/Resources/OSDevIndex.html

[3] . https://viralpatel.net/taj/tutorial/hello$_{w}orld_{b}ootloader.php$

[4] . http://en.wikibooks.org/wiki/X86$_{A}ssembly/Bootloaders$

[5] . http://www.ctyme.com/rbrown.htm

[6] . http://www.jamesmolloy.co.uk/tutorial$_{h}tml/index.html$

[7] . http://fly.cc.fer.hr/GDM/articles/sndmus/speaker1.html

[8] . http://www.osdever.net/tutorials/index

# Appendix A

# IMPLEMENTATION

## A.1 Sample Program code

### A.1.1 Bootsector

```
use16
org 0x7C00

BOOTSECTOR:     jmp     near MAIN
                nop

;------------------------------------------------;
;   Standard BIOS Parameter Block, "BPB".        ;
;------------------------------------------------;
        brOSOEM                 db  'brOS 0.1 '
        brOSSectorSize          dw  512
        brOSClusterSize         db  1
        brOSReservedSectors     dw  1
        brOSFats                db  2
        brOSRootDirectorySize   dw  224
        brOSTotalSectors        dw  2880
        brOSMedia               db  240
        brOSFatSize             dw  9
        brOSTracksPerSector     dw  18
        brOSHeads               dw  2
        brOSHiddenSectors       dd  0
        brOSBigSectors          dd  0
        brOSDriveNo             db  0
        brOSUnused              db  0
        brOSBootSignature       db  41
        brOSID                  dd  1
        brOSVolumeLabel         db  'BOOT DISK A'
        brOSFileSystem          db  'FAT12   '
```

```
;-----------------------------------------;
;    starting point of bootsector code    ;
;-----------------------------------------;
MAIN:
            cli

            xor     ax, ax
            mov     ds, ax
            mov     es, ax
            mov     ss, ax
            mov     sp, 0xFFFF

            mov     [brOSDriveNo], dl

            sti


;-----------------------------------------;
;    clear screen and print some          ;
;-----------------------------------------;
            mov     ax, 3
            int     0x10

            mov     bp, LOADINGMESG
            mov     ax, 0x1301
            mov     bx, 4
            mov     cx, 44
            mov     dx, 0x0102
            int     0x10

            mov     bl, 4
            mov     ah, 2
            mov     dx, 0x0202
            int     0x10

            mov     ah, 9
            mov     al, '_'
            mov     cx, 44
            int     0x10


;-----------------------------------;
;     load FAT and root             ;
;-----------------------------------;
            mov     di, 0x0050
```

```
        mov     ax, 19
        mov     cx, 14
        call    READSECTORS

        mov     di, 0x0210
        mov     ax, 1
        mov     cx, 9
        call    READSECTORS


;-------------------------------;
;   search for the file         ;
;-------------------------------;
        mov     dx, [brOSRootDirectorySize]
        mov     bx, 0x0500
SEARCHFILE:
        cld
        mov     si, KERNELFILENAME
        mov     cx, 11
        mov     di, bx
        repe    cmpsb
        je      LOADFILE
        add     bx, 32
        dec     dx
        jz      ERROR
        jmp     SEARCHFILE



;-------------------------------------------------------------;
;    read a number of sectors (one at a time)                 ;
;-------------------------------------------------------------;
;   in:                                                       ;
;      di = segment to save at                                ;
;      ax = sector to read                                    ;
;      cx = number of sectors                                 ;
;   out:                                                      ;
;      di = updated (added for next read)                     ;
;      ax = updated (added for next read)                     ;
;-------------------------------------------------------------;
READSECTORS:
        pusha
        mov     bl, byte [brOSTracksPerSector]
        div     bl
        mov     cl, ah
        add     cl, 1
```

```
        xor     ah, ah
        mov     bl, byte [brOSHeads]
        div     bl
        mov     ch, al
        mov     dh, ah

        mov     ax, cx
        mov     cx, 6
.TRYNEXTSECT:
        push    es
        push    cx
        mov     cx, ax
        push    cx

        xor     ax, ax
        mov     dl, [brOSDriveNo]
        push    dx
        int     0x13
        jc      .FAILED

        pop     dx
        pop     cx
        xor     bx, bx
        mov     es, di
        mov     ax, 0x0201
        int     0x13
        jnc     .SUCCESS

.FAILED:
        pop     dx
        pop     ax

        pop     cx
        pop     es
        loop    .TRYNEXTSECT
        jmp     ERROR
.SUCCESS:
        pop     cx
        pop     es
        popa
        add     di, 32
        inc     ax
        loop    READSECTORS
        ret
```

```
        ;-------------------------------------------------------------;
        ;   show a message and wait for a key before reboot           ;
        ;-------------------------------------------------------------;
        ERROR:
                mov     bp, FAILTURE
                mov     ax, 0x1301
                mov     bx, 4
                mov     cx, 11
                mov     dx, 0x0401
                int     0x10

                mov     ah, 0
                int     0x16
                int     0x19


;-----------------------------------------;
;    the file is found, load it.          ;
;-----------------------------------------;
LOADFILE:
                mov     bp, [bx+26]
                mov     di, 0x1000

        .NEXTBLOCK:
                xor     cx, cx
                mov     cl, [brOSClusterSize]
                mov     si, bp

        .NEXTCLUSTER:
                mov     ax, 3
                mul     si
                shr     ax, 1
                mov     bx, ax
                mov     ax, word [(0x2100+bx)]
                jc      .ODDCLUSTER
                and     ax, 0x0FFF
                jmp     .CLUSTERFOUND
        .ODDCLUSTER:
                shr     ax, 4

        .CLUSTERFOUND:
                inc     si
                cmp     ax, si
                jne     .FORCEREAD

                add     cl, [brOSClusterSize]
                adc     ch, 0
```

```
            jmp     .NEXTCLUSTER

        .FORCEREAD:
            xchg    bp, ax
            dec     ax
            dec     ax

            movzx   dx, byte [brOSClusterSize]
            mul     dx
            add     ax, 33
            call    READSECTORS

            cmp     bp, 0x0FF8
            jb      .NEXTBLOCK

;-----------------------------;
;   the file is loaded    ;
;-----------------------------;
ENDOFCODE:
            jmp     0x1000:0x0000


;-----------------------------------------;
;       variables & functions             ;
;-----------------------------------------;
            LOADINGMESG     db  'brOS Operating System v0.1 @ FISAT 2011-
            KERNELFILENAME  db  'KERNEL  SYS'
            FAILTURE        db  'Read error!'

;-----------------------------------------;
; set the BOOT-signature at byte 510. ;
;-----------------------------------------;
            rb BOOTSECTOR+512-2-$
            dw 0xAA55
```

### A.1.2 A20 Gate Setup

```
;-----------------------------;
;    enable a20 gate      ;
;-----------------------------;
enable_a20:
            in      al, 0x64
            test    al, 2
            jnz     enable_a20
            mov     al, 0xD1
            out     0x64, al
        .d6:
```

```
        in      al, 0x64
        and     ax, 2
        jnz     .d6
        mov     al, 0xDF
        out     0x60, al

        call    a20_test
        jz      .ok

        in      al, 0x92
        or      al, 0x02
        out     0x92, al

        call    a20_test
        jz      .ok



.ok:
        ret
```

### A.1.3 Global Descriptor Table

```
;-----------------------;
;   pointer to GDT      ;
;-----------------------;
gdtr:
    .size           dw  gdt_end - gdt - 1
    .address        dd  0x7400

;-----------------------------------------;
;   Global Descriptor Table (GDT).        ;
;-----------------------------------------;
gdt:
    .null           dw  0x0000,0x0000,0x0000,0x0000 ; null desc.
    .BOS_code:      dw  0xFFFF,0x0000,0x9A00,0x00CF ; 0x08
    .BOS_data:      dw  0xFFFF,0x0000,0x9200,0x00CF ; 0x10
    .BOS_16code:    dw  0xFFFF,0x0000,0x9A00,0x0000 ; 0x18
    .BOS_16data:    dw  0xFFFF,0x0000,0x9200,0x0000 ; 0x20

gdt_end:
```

### A.1.4 Switching to Protected Mode

```
        mov     eax, cr0
        or      al, 1
        mov     cr0, eax
```

### A.1.5 Sample ISR

```
;-------------------------------------;
;   Exception Interrupt no. 00    ;
;-------------------------------------;
isr00:
            mov     [dbg_keypress], 0
            mov     [dbg_error], 'D'
            mov     [dbg_error+1], 'i'
            mov     [dbg_error+2], 'v'
            mov     [dbg_error+3], 'i'
            mov     [dbg_error+4], 'd'
            mov     [dbg_error+5], 'e'
            mov     [dbg_error+6], ' '
            mov     [dbg_error+7], 'e'
            mov     [dbg_error+8], 'r'
            mov     [dbg_error+9], 'r'
            mov     [dbg_error+10], 'o'
            mov     [dbg_error+11], 'r'
            mov     [dbg_error+12], ' '
            mov     [dbg_error+13], ' '
            mov     [dbg_error+14], 0
            call    dump_regs
```

### A.1.6 Getting Time

```
get_time:
            push    ax

            mov     al, 0x00
            out     0x70, al
            in      al, 0x71
            mov     [stime], al

            mov     al, 0x02
            out     0x70, al
            in      al, 0x71
            mov     [mtime], al

            mov     al, 0x04
            out     0x70, al
            in      al, 0x71
            mov     [htime], al
            pop   ax
            ret
```

### A.1.7 Sample Piano Code

```
.play:
        call getc

        push bx
        mov         bl, al
        mov         bh, 0x03
        call        print_char
        pop bx


        push bx
        mov         bl, 32
        mov         bh, 0x0E
        call        print_char
        pop bx

        ;-------------------;
        ;Check 'a'          ;
        ;-------------------;
        .check_a:
                cmp al,97
                jne .check_b
                call play_a
                jmp .done
```

### A.1.8 Some Standard Modules

```
get_ascii:
            push    ecx
            mov     ah, al
            and     ax, 0xF00F
            shr     ah, 4
            or      ax, 0x3030
            xchg    ah, al
            pop     ecx
            ret


show_time_shell:
            push    eax
            push    ebx
            push    esi
            push    ecx
```

```
  call get_time

  call getcursor
  mov ax,bx
  mov cx,59
  add ax,cx

  mov [cursor_pos_time],ax

  mov al,[htime]


  mov     bh, 0x04
  cmp al,12

  jng .jump_am

.jump_pm:

  sub al,12
  mov [day_flag],byte 1
  jmp .continue


.jump_am:
  mov [day_flag],byte 0

.continue:
  call get_ascii
  push    eax
  mov     bl, al
  call    print_time_char
  pop     eax
  mov     bl, ah
  call    print_time_char

  mov     bl, ':'
  call    print_time_char



  mov al,[mtime]
  call get_ascii
  push    eax
  mov     bl, al
  call    print_time_char
  pop     eax
```

```
  mov     bl, ah
  call    print_time_char

  mov     bl, ':'
  call    print_time_char

  mov al,[stime]
  call get_ascii
  push    eax
  mov     bl, al
  call    print_time_char
  pop     eax
  mov     bl, ah
  call    print_time_char

  mov     bl, '␣'
  call    print_time_char

  cmp [day_flag],byte 0
  je .set_am

  mov     bl, 'P'
  call    print_time_char

  mov     bl, 'M'
  call    print_time_char

  jmp .end_set_am_pm


.set_am:
  mov     bl, 'A'
  call    print_time_char

  mov     bl, 'M'
  call    print_time_char

.end_set_am_pm:



  pop ecx
  pop     esi
  pop     ebx
  pop     eax
  ret
```

```
;-------------------------------------------------;
;   print char                                     ;
;             in: bl = char, bh = attrib          ;
;-------------------------------------------------;
print_time_char:
            push    eax
            push    bx

      .cont:
            push    bx
            mov     bx,[cursor_pos_time]
            movzx   eax, bx
            pop     bx

            mov     [es:(eax*2 + 0xB8000)], bx

            add     [cursor_pos_time],word 1
      .done:
            pop     bx
            pop     eax
            ret

show_date:
            push    eax
            push    ebx
            push    esi

           ; mov     esi, msg_date
           ;; mov    bl, 0x07
           ; call    print

            mov     bh,0x09
            call    get_date
            mov     al, [ydate]
            call    get_ascii
            push    eax
            mov     bl, al
            call    print_char
            pop     eax
            mov     bl, ah
            call    print_char

            mov     bl, '/'
            call    print_char

            mov     al, [mdate]
```

```
            call    get_ascii
            push    eax
            mov     bl, al
            call    print_char
            pop     eax
            mov     bl, ah
            call    print_char

            mov     bl, '/'
            call    print_char

            mov     al, [ddate]
            call    get_ascii
            push    eax
            mov     bl, al
            call    print_char
            pop     eax
            mov     bl, ah
            call    print_char

            pop     esi
            pop     ebx
            pop     eax
            ret


;-------------------------------------------------------------------------;
;   time, prints current time and goes back to the      ;
;   prompt.                                             ;
;-------------------------------------------------------------------------;
show_time:
            push    eax
            push    ebx
            push    esi

           ; mov     esi, msg_time
          ;  mov     bl, 0x07
           ; call    print
            call    get_time
            mov     al, [htime]
            call    get_ascii
            push    eax
            mov     bh, 0x09
            mov     bl, al
            call    print_char
            pop     eax
            mov     bl, ah
```

```
            call    print_char

            mov     bl, ':'
            call    print_char

            mov     al, [mtime]
            call    get_ascii
            push    eax
            mov     bl, al
            call    print_char
            pop     eax
            mov     bl, ah
            call    print_char

            mov     bl, ':'
            call    print_char

            mov     al, [stime]
            call    get_ascii
            push    eax
            mov     bl, al
            call    print_char
            pop     eax
            mov     bl, ah
            call    print_char

            pop     esi
            pop     ebx
            pop     eax
            ret
;-------------------------------------------------------------------------------
ask_time db 13,10,13,10,"Type Time in the format hh:mm:ss in 24 hour fo
hrh      db 0
hrl      db 0
mh       db 0
ml       db 0
sh       db 0
sl       db 0
tmpchar  db 0
;-------------------------------------------------------------------------------

set_system_time:

            mov     esi, ask_time
            mov     bl, 0x0E
            call    print
```

```
.start:
    call getc
    mov [tmpchar],al
    cmp al,48
    jnge .start
    cmp al,57
    jge .start

    cmp al,'2'
    jg .start
    mov [hrh],al

    push bx
    mov      bl, al
    mov      bh, 0x09
    call     print_char
    pop bx
 .start2:
    call getc
    cmp [hrh],2
    je  .not_20s
    cmp al,'4'
    jg .start2

 .not_20s:
    mov [hrl],al
    push bx
    mov      bl, al
    mov      bh, 0x09
    call     print_char
    pop bx



    push bx
    mov      bl, ':'
    mov      bh, 0x09
    call     print_char
    pop bx

    cmp [hrh],'2'
    jne .start3
    cmp [hrl],'4'
    jne .start3
    mov [hrh],'0'
    mov [hrl],'0'
```

```
.start3:
   call getc
   cmp al,'6'
   jg .start3
   jne .c2
.c2:
   mov [mh],al
   push bx
   mov      bl, al
   mov      bh, 0x09
   call     print_char
   pop bx

.start4:
   call getc
   cmp [mh],'6'
   jnge .c3
   cmp al,'0'
   je .c3
   jmp .start4
.c3:
   mov [ml],al
   push bx
   mov      bl, al
   mov      bh, 0x09
   call     print_char
   pop bx
   push bx
   mov      bl, ':'
   mov      bh, 0x09
   call     print_char
   pop bx

.start5:
   call getc
   cmp al,'6'
   jg .start5
   jne .c4
.c4:
   mov [sh],al
   push bx
   mov      bl, al
   mov      bh, 0x09
   call     print_char
   pop bx
```

```
        .start6:
           call getc
           cmp [sh],'6'
           jnge .c5
           cmp al,'0'
           je .c5
           jmp .start6
        .c5:
           mov [sl],al
           push bx
           mov      bl, al
           mov      bh, 0x09
           call     print_char
           pop bx


          ret

end_set_time:

           push bx
           mov      bl, '0'
           mov      bh, 0x09
           call     print_char
           pop bx
            push bx
           mov      bl, '0'
           mov      bh, 0x09
           call     print_char
           pop bx
            push bx
           mov      bl, ':'
           mov      bh, 0x09
           call     print_char
           pop bx
            push bx
           mov      bl, '0'
           mov      bh, 0x09
           call     print_char
           pop bx
            push bx
           mov      bl, '0'
           mov      bh, 0x09
           call     print_char
           pop bx

          ret
```

### A.1.9 Piano sound processing

```
;-------------------------------------;
;Stop the PC Speaker                  ;
;-------------------------------------;
set_pit_freq:
        push    eax
        mov     al, 0x34
        out     0x43, al
        mov     al, 0x9B
        out     0x40, al
        mov     al, 0x2E
        out     0x40, al
        pop     eax
        ret



delay2:
        mov     [timer_wait], ecx
    .loop:
        cmp     [timer_wait], 0
        jne     .loop
        ret

enable_timer_irq:
        push cx
        mov     cl, 0
        call    enable_irq
        pop cx
        ret


stop_speaker:
        push    ax
        in      al, 0x61
        and     al, 0xFC
        out     0x61, al
        pop     ax
        ret
;-------------------------------------;
;Beep the PC Speaker                  ;
;in: ax=freq                          ;
;-------------------------------------;
beep:
        push eax
        push edx
```

```
        push ecx
        push ebx

        mov ebx,eax

        mov al,10110110B
        out 0x43,al
        xor ax,ax
        mov eax,1193180
        div ebx
        mov dx,ax
        xchg dx,ax

        out 0x42,al
        xchg al,ah
        out 0x42,al

        in      al, 0x61
        or      al, 3
        out     0x61, al
        mov     ecx, 7
        call    delay2
        call    stop_speaker



        pop ebx
        pop ecx
        pop edx
        pop eax
        ret
;-------------------------------------;
; Start the PC Speaker                ;
;-------------------------------------;
start_speaker:
        push    eax
        in      al, 0x61
        or      al, 3
        out     0x61, al
        pop     eax
        ret


;-------------------------------------;
; Give a delay                        ;
;       count-> in:cx                 ;
;-------------------------------------;
```

```
delay:
    .loop:
        loop    .loop
        ret


;-----------------------;
;Play a                 ;
;-----------------------;
play_a:
        push eax
        mov eax,dword 1109
        call beep
        pop eax
        ret


;-----------------------;
;Play b                 ;
;-----------------------;
play_b:
        push eax
        mov eax,dword 1760
        call beep
        pop eax
        ret



;-----------------------;
;Play c                 ;
;-----------------------;
play_c:
        push eax
        mov eax,dword 1397
        call beep
        pop eax
        ret
```

### A.1.10 Screensaver

```
screensaver_init:
                pusha
                mov cx,4000

                push bx
                call getcursor
```

```
                mov [cursor_backup],bx
                pop bx
                xor eax,eax
                mov edi,screen_buffer
        .store:
                xor dx,dx
                mov dl,byte [es:(eax + 0xB8000)]
                mov byte [edi],dl
                inc edi
                inc eax
                dec cx
                jnz .store

                call screensaver_start
                popa
                ret

screensaver_start:
                pusha
                call screen_clear
                xor eax,eax
                xor ecx,ecx
                xor edx,edx
                mov dl,1
                mov bh,0
                mov bl,10
                call setcursorxy


        .jump_incinc:
                call screen_clear
                push bx
                mov     esi, bros_saver
                mov     bl,dl
                call    print

                inc dx
                cmp dl,15
                jne .c1
                mov dl,1
              .c1:
                pop bx
                push ecx
                mov     ecx, 3
                call    delay2
                pop ecx
                push eax
```

```
                push ebx
                mov al,[row_val]
                mov bl,[col_val]
                ;inc ax
                cmp ax,20
                jnge .c3
                ;mov ax,0


        .c3:
                inc bx
                cmp bl,[col_limit]
                jnge .c4
                mov bx,0



        .c4:
                mov [row_val],al
                mov [col_val],bl

                pop ebx
                pop eax


                mov bl,byte [row_val]
                mov bh,byte [col_val]
                call setcursorxy



                call screen_getc
                cmp al,''
                jne .c2
                call screensaver_stop
                jmp .end
        .c2:
                jmp .jump_incinc
        .end:
                popa
                ret

screensaver_stop:
                pusha
                mov cx,4000
```

```
                xor eax,eax
                mov edi,screen_buffer
        .store:
                xor dx,dx
                mov dl,byte [edi]
                mov byte [es:(eax + 0xB8000)],dl
                inc edi
                inc eax
                dec cx
                jnz .store

                mov bx,[cursor_backup]
                call setcursor
                popa
                ret

screen_clear:
        push    bx
        push    cx

        movzx   cx, [screen_rows]
    .loop:
        call    _scroll_up
        loop    .loop
        pop     cx
        pop     bx
        ret

screen_getc:
        push    esi

    .no_new:
        mov     al, [kbd_head]
        mov     ah, [kbd_tail]
        cmp     ah, 63
        jne     .check2

        cmp     al, 0
        je      .end2
        mov     [kbd_tail], 0
        jmp     .done_check
    .check2:
        mov     bl, ah
        inc     bl
        cmp     bl, al
        je      .end2
        inc     [kbd_tail]
```

```
.done_check:

        mov     esi, kbd_buffer
        movzx   eax, [kbd_tail]
        add     esi, eax
        mov     ah, byte [esi]                  ; ah = scancode
        movzx   esi, byte [esi]                 ; esi = scancode

;-----------------------;

;   some checks..     ;
;-----------------------;
        cmp     ah, 0xFA
        je      .no_new
        cmp     ah, 0xE0
        je      .no_new
        cmp     ah, 0xE1
        je      .no_new
        test    ah, 0x80
        jnz     .no_new

;---------------------------------------;
;   check for caps, shift & alt     ;
;---------------------------------------;
        test    [kbd_status], 00000100b
        jnz     .caps
        test    [kbd_flags], 00000001b
        jnz     .shift

;-----------------------;
;   normal keymap     ;
;-----------------------;
        mov     al, [esi+keymap]                ; scancode + keyn
        jmp     .end

;-----------------------;
;   capslock keymap     ;
;-----------------------;
.caps:
        test    [kbd_flags], 00000001b
        jnz     .caps_and_shift

        mov     al, [esi+keymap_caps]
        jmp     .end

;-------------------------------;
;   caps and shift keymap     ;
```

```
;-----------------------------------;
.caps_and_shift:
        mov     al, [esi+keymap_caps_shift]
        jmp     .end

;--------------------;
;   shift keymap     ;
;--------------------;
.shift:
        mov     al, [esi+keymap_shift]
        jmp     .end

;-----------------------------------;
;   set registers and exit    ;
;-----------------------------------;
.end:
        mov     bl, [kbd_status]
        mov     bh, [kbd_flags]
.end2:
        pop     esi
        ret
```

## A.2 Technology Explanation

### A.2.1 QEMU

QEMU stands for "Quick EMUlator" and is a processor emulator that relies on dynamic binary translation to achieve a reasonable speed while being easy to port to new host CPU architectures. In conjunction with CPU emulation, it also provides a set of device models, allowing it to run a variety of unmodified guest operating systems; it can thus be viewed as a hosted virtual machine monitor. It also provides an accelerated mode for supporting a mixture of binary translation (for kernel code) and native execution (for user code), in the same fashion as VMware Workstation and VirtualBox. QEMU can also be used purely for CPU emulation for user level processes, allowing applications compiled for one architecture to be run on another.

QEMU has two operating modes:

#### User mode emulation

In this mode QEMU runs single Linux or Darwin/Mac OS X programs that were compiled for a different CPU. System calls are thunked for endianness and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.

**Computer emulation**

In this mode QEMU emulates a full computer system, including peripherals. It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD ; it supports emulating several hardware platforms, including x86, x86-64, ARM, ETRAX CRIS, MIPS, MicroBlaze, PowerPC and SPARC.

The virtual machine can interface with many types of physical host hardware. Some of these are: hard disks, CD-ROM drives, network cards, audio interfaces, and USB devices. USB devices can be completely emulated (mass storage from image files, input devices), or the host's USB devices can be used (however, this requires administrator privileges and does not work with all devices).

Virtual hard disk images can be stored in a special format (qcow or qcow2) that only take up disk space that the guest OS actually uses. This way, an emulated 120 GB disk can occupy only several hundred megabytes on the host. The QCOW2 format also allows the creation of overlay images that record the difference from another base image file which is not modified. This provides the possibility for reverting the emulated disk's contents to an earlier state. For example, a base image could hold a fresh install of an operating system that is known to work, and the overlay images are used. Should the guest system become unusable (virus attack, accidental system destruction, ...), the overlay can be deleted and an earlier emulated disk image version recreated.

QEMU does not depend on the presence of graphical output methods on the host system. Instead, it can allow one to access the screen of the guest OS via an integrated VNC server. It can also use an emulated serial line, without any screen, with applicable operating systems.

QEMU does not depend on the presence of graphical output methods on the host system. Instead, it can allow one to access the screen of the guest OS via an integrated VNC server. It can also use an emulated serial line, without any screen, with applicable operating systems.

### A.2.2 x86 assembly language

x86 assembly language is a family of backward-compatible assembly languages, which provide some level of compatibility all the way back to the Intel 8008. x86 assembly languages are used to produce object code for the x86 class of processors, which includes Intel's Core series and AMD's Phenom and Phenom II series. Like all assembly languages, it uses short mnemonics to represent the fundamental operations that the CPU in a computer can perform. Compilers sometimes produce assembly code as an intermediate step when translating a high level program into machine code. Regarded as a programming language, assembly coding is machine-specific and low level. Assembly languages are more typically used for detailed and/or time critical applications such as small real-time embedded systems or operating system kernels and device drivers.

The modern x86 instruction set is a superset of 8086 instructions and a series of extensions to this instruction set that began with the Intel 8008 microprocessor.

Nearly full binary backward compatibility exists between the Intel 8086 chip through to the current generation of x86 processors, although certain exceptions do exist. In practice it is typical to use instructions which will execute on anything later than an Intel 80386 (or fully compatible clone) processor or else anything later than an Intel Pentium (or compatible clone) processor but in recent years various operating systems and application software have begun to require more modern processors or at least support for later specific extensions to the instruction set

### Segmented addressing

The x86 architecture in real and virtual 8086 mode uses a process known as segmentation to address memory, not the flat memory model used in many other environments. Segmentation involves composing a memory address from two parts, a segment and an offset; the segment points to the beginning of a 64 KB group of addresses and the offset determines how far from this beginning address the desired address is. In segmented addressing, two registers are required for a complete memory address: one to hold the segment, the other to hold the offset. In order to translate back into a flat address, the segment value is shifted four bits left then added to the offset to form the full address, which allows breaking the 64k barrier through clever choice of addresses, though it makes programming considerably more complex.

The 32-bit flat memory model of the 80386's extended protected mode may be the most important feature change for the x86 processor family until AMD released x86-64 in 2003, as it helped drive large scale adoption of Windows 3.1 (which relied on protected mode) since Windows could now run many applications at once, including DOS applications, by using virtual memory and simple multitasking.

### Execution modes

The x86 processors support five modes of operation for x86 code, Real Mode, Protected Mode, Long Mode, Virtual 86 Mode, and System Management Mode, in which some instructions are available and others are not. A 16-bit subset of instructions are available in real mode (all x86 processors), 16-bit protected mode (80286 onwards), V86 mode (80386 and later) and SMM (Some Intel i386SL, i486 and later). In 32-bit protected mode (Intel 80386 onwards), 32-bit instructions (including later extensions) are also available; in long mode (AMD Opteron onwards), 64-bit instructions, and more registers, are also available. The instruction set is similar in each mode but memory addressing and word size vary, requiring different programming strategies.

### Instruction types

In general, the features of the modern x86 instruction set are:

- A compact encoding
  - Variable length and alignment independent (encoded as little endian, as is all data in the x86 architecture)

  - Mainly one-address and two-address instructions, that is to say, the first operand is also the destination.
  - Memory operands as both source and destination are supported (frequently used to read/write stack elements addressed using small immediate offsets).
  - Both general and implicit register usage; although all seven (counting ebp) general registers in 32-bit mode, and all fifteen (counting rbp) general registers in 64-bit mode, can be freely used as accumulators or for addressing, most of them are also implicitly used by certain (more or less) special instructions; affected registers must therefore be temporarily preserved (normally stacked), if active during such instruction sequences.

- Produces conditional flags implicitly through most integer ALU instructions.
- Supports various addressing modes including immediate, offset, and scaled index but not PC-relative, except jumps (introduced as an improvement in the x86-64 architecture).
- Includes floating point to a stack of registers.
- Contains special support for atomic instructions (xchg, cmpxchg/cmpxchg8b, xadd, and integer instructions which combine with the lock prefix)
- SIMD instructions (instructions which perform parallel simultaneous single instructions on many operands encoded in adjacent cells of wider registers).

### A.2.3 Netwide Assembler

The Netwide Assembler (NASM) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. NASM is considered to be one of the most popular assemblers for Linux. NASM can output several binary formats including COFF, Portable Executable, a.out, ELF and Mach-O, though position-independent code is only supported for ELF object files. NASM also has its own binary format called RDOFF.The variety of output formats allows programs to be retargetted to virtually any x86 operating system. In addition, NASM can create flat binary files, usable in writing boot loaders, ROM images, and in various facets of OS development. NASM can run on non-x86 platforms, such as SPARC and PowerPC, though it cannot generate programs usable by those machines. NASM uses variation of Intel assembly syntax instead of ATT syntax.It also avoids features such as automatic generation of segment overrides (and the related ASSUME directive) used by MASM and compatible assemblers. NASM principally outputs object files, which are generally not executable in and of themselves. The only exception to this are flat binaries which are inherently limited in modern use. To translate the object files into executable programs, an appropriate linker must be used, such as the Visual Studio "LINK" utility for Windows or ld for UNIX-like systems.

## A.3 Screenshots

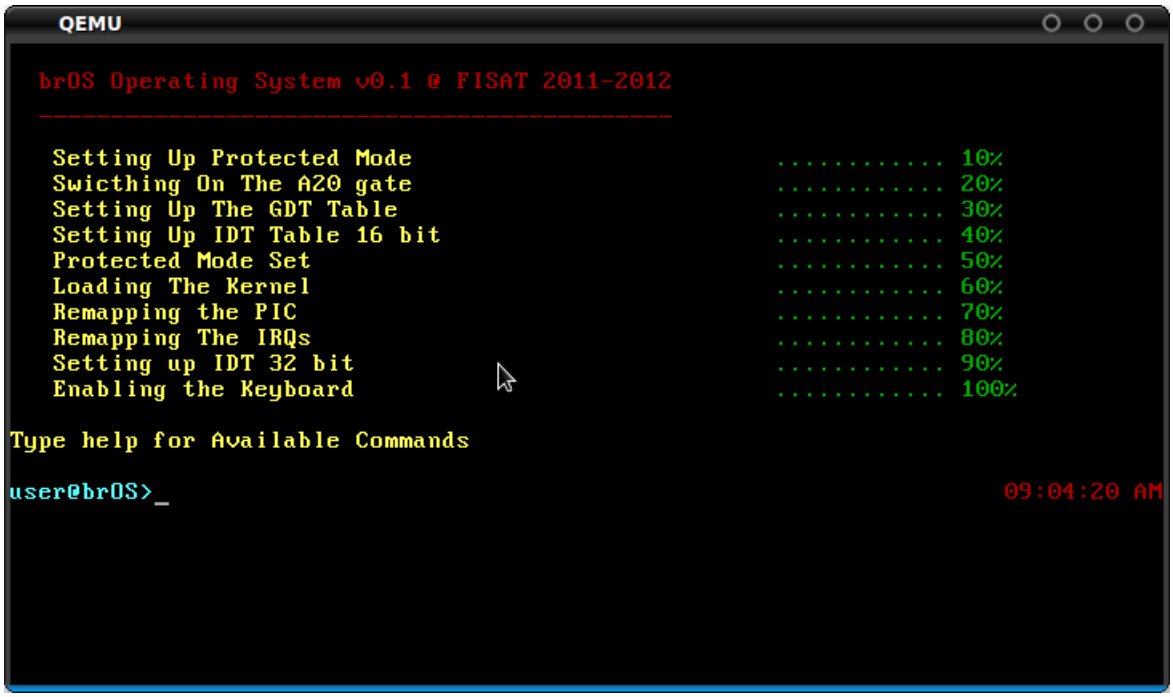


Figure A.1: Start Screen of BROS


Figure A.2: Available commands shown using help command

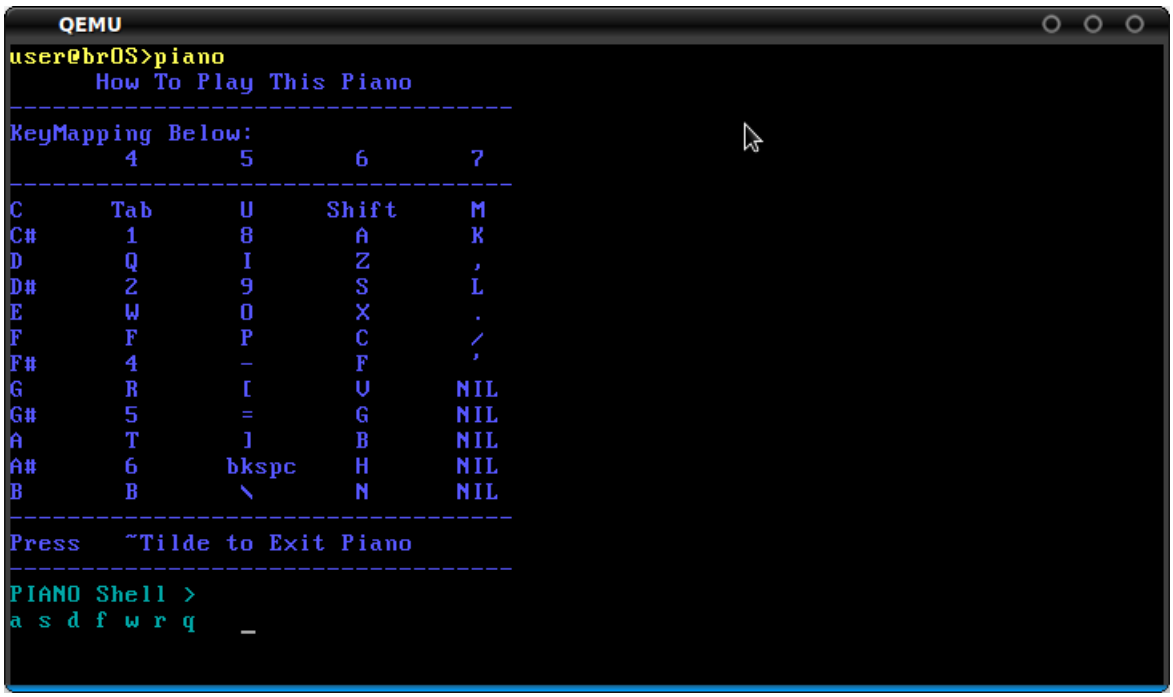


Figure A.3: Piano Interface

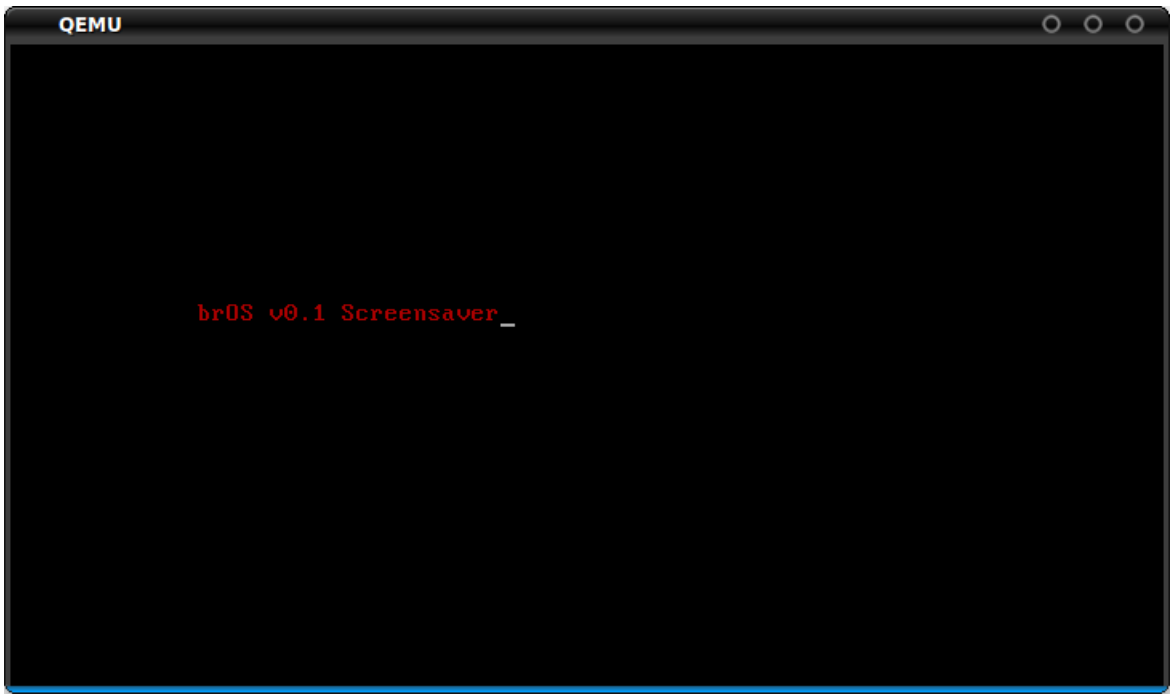


Figure A.4: Screensaver