

Indian Institute of Technology Jodhpur



॥ त्वं ज्ञानमयो विज्ञानमयोऽसि ॥

Building a x86 architecture compatible Operating system

by

Saurabh Burewar (B18CSE050)
Rituraj Kulshresth (B18CSE046)

Department of Computer Science and Engineering

*A Project Report Submitted to the
Department of Computer Science and Engineering
For the course of Operating System
CSE IIT Jodhpur*

Supervisor:

Dr. Suchetana Chakraborty
Dr. Ravi Bhandari

Department of Computer Science and Engineering
IIT Jodhpur

Abstract

In this project, we build a very basic operating system with limited functions. We first build a boot loader which is the first thing that is run when you start your system. All the code is stored at the start of the disk, known as the boot sector, so we load the disk. The operating system goes into 32-bit mode because it is necessary to access 32-bit registers, memory addressing, etc. Next we build a kernel in C, but for that we first need a cross compiler that can run on host OS but create executable code for our OS. We build a cross compiler using GCC 4.9 and then build a simple kernel in C. Now we create drivers in order to interact with the I/O devices. The first driver is video driver, to print characters on the screen and the second is keyboard driver to take input from the keyboard. We also build a memory allocation function which provides a constant bytes of memory every time it is called. Lastly, we build a basic shell with limited commands to provide an interface for the user.

Contents

Abstract	i
1 Introduction	1
1.1 Main Objective	1
1.1.1 Specific Objectives	1
1.2 Scope of the Project	1
2 Methodology	2
2.1 Introduction	2
2.2 Bootloader	2
2.2.1 Boot the disk	2
2.2.2 Segmentation	3
2.3 Moving from 16 to 32-bit mode	3
2.3.1 Disable interrupts	3
2.3.2 Load GDT descriptor	3
2.3.3 Set the CPU control register to 32-bit mode	3
2.3.4 Far jump to a different segment	4
2.3.5 Update all the segment registers and stack	4
2.4 Building a kernel	4
2.4.1 Entry routine for the kernel	4
2.4.2 Compiling C files	4
2.5 Building drivers	4
2.5.1 Video Driver	5
2.5.2 Interrupts	5
2.5.3 Keyboard Driver	6
2.5.4 Malloc	6
2.6 Building a shell	6
3 System Study and Analysis	8
3.1 System	8
3.2 Analysis	8

3.3	Path taken by the OS	8
4	Presentation of Results	10
4.1	Start of the OS	10
4.2	Shell commands	11
5	Results and Conclusion	13
5.1	Future Prospects	13

Chapter 1

Introduction

1.1 Main Objective

To build a x86 architecture operating system from scratch.

1.1.1 Specific Objectives

The specific objectives of the study were:

- Build a bootloader in machine code.
- Create a 32 bit mode for the operating system
- Build a kernel in C.
- Build drivers to interact with the I/O devices.
- Build drivers to print text on the VGA screen.
- Build a shell to interact with the user.

1.2 Scope of the Project

This project is limited to the development of a LINUX like fully functional operating system, with a file system to manage folders and files. A process management system to manage and run multiple processes at the same time along with process scheduling. Create a simple text editor. Create fully functional shell just like LINUX. A graphic user interface (GUI) and Networking via the shell.

Chapter 2

Methodology

2.1 Introduction

Building an operating system includes building of various parts of it one by one. The process involves building a bootloader moving from 16 bit to 32-bit mode, in order to use 32-bit registers and memory addressing, virtual memory, protected memory and other advantages. Finally, there is building of kernel using a higher level language, drivers to transfer data between I/O devices and a simple shell.

2.2 Bootloader

The bootloader is written in machine code and is assembled using the Netwide Assembler (NASM). It is the first program that is run on a CPU. It starts off the basic processes and allocates the interrupt descriptor tables.

2.2.1 Boot the disk

The first thing it does is boot the drive where the operating system is stored. The disk is divided in different sectors. Since this is the first thing to start when you start your system, it makes sense to keep it at the start of the disk. Therefore, the boot sector (the area where the code for booting system is kept) is normally kept at start of the disk. The first sector of the disk here is of 512 bytes which is not enough to store our data of the OS, which is why the first 31 sectors are used to store the data and the code necessary for our OS. We do not use an actual disk but some BIOS routines to do the job. The drive number, sector number, cylinder number and head number are all stored in different registers.

2.2.2 Segmentation

Segmentation means that you can specify an offset to all the data you refer to. This is done by using special registers: cs, ds, ss and es, for Code, Data, Stack and Extra. This is done to access the data stored at a particular memory location whenever necessary.

Next thing is switching from 16 to 32-bit which is done through machine code.

2.3 Moving from 16 to 32-bit mode

16 bit mode gives us the use of predefined interrupts like 0x10 to print a character and allows the user to change the value stored in the registers that store the data for the OS. This is unsafe for the OS as any one will be able to change the data on the go and harm the OS. Hence we use 32 bit mode which though takes away the interrupts but gives us many features and additional security. In order to move from 16 to 32-bit mode, we need to do the following

-

2.3.1 Disable interrupts

This is done using the clear interrupt flag (cli).

2.3.2 Load GDT descriptor

GDT is a data structure where characteristics of memory areas including base address, size and flags (permissions, writability, etc.) are stored. In 32-bit mode, segmentation works differently. Now, the offset is an index to the segment descriptor in GDT. The way we make a GDT is, define it in two segments, one for code and another for data and define the size, flags etc. We also make a descriptor because the CPU can't directly load the GDT address and requires a meta structure called "GDT descriptor".

2.3.3 Set the CPU control register to 32-bit mode

We set the the CPU control register (cr0) to 32-bit mode. We set it to "0x1" which is the 32-bit mode bit

2.3.4 Far jump to a different segment

The reason for jumping to a different segment like this is to flush the CPU pipeline.

2.3.5 Update all the segment registers and stack

Now we are in 32-bit mode but before moving forward we still need to update all the segment registers and update the stack at the top of the free space.

Now we are ready to call a function in 32-bit mode which in return will call the kernel.

2.4 Building a kernel

The kernel is written in C language. The C file contains the main kernel function. All the subsequent things to be done, including interrupt handling, keyboard inputs, etc. will be done through this main function.

2.4.1 Entry routine for the kernel

The main function is called through the assembler via an entry routine which is called from the bootsector. This entry routine is written in another .asm file. The only job of the entry routine is to call the kernel main function.

2.4.2 Compiling C files

We use the cross compiler in order to create an executable file from the C file. We talk more about the cross compiler in system and design section.

Once the kernel is running, we need drivers to get input from and show output to the user.

2.5 Building drivers

Drivers are necessary codes required to get the information to and from an I/O device.

2.5.1 Video Driver

So far our kernel is able to print text on the screen but with low level address manipulation only. We are still unable to print a text in an easy way without addressing all the memory addresses. We will use the driver for this purpose.

We use the VGA's text mode to output text in a 80x25 character area. An offset variable maintains the position of the cursor to know where a character is to be printed. Once a character is printed, the offset is changed to print the next one. We get the cursor position from the I/O port and can use it to print a character on the screen. When the offset is manipulated correctly and the print function is called with -1,-1 it prints the text in the current position and is again updated. When a '\n' is encountered we reset the offset to the next line. We have a function that handles printing characters and one that sets the column, row and offset to feed to the first function. There is another function which is used to clear the screen by printing a blank character and resetting the offset to 0.

In case of an overflow of text, the top row is erased and the subsequent rows are moved one row up, thus leaving the last row empty. This gives us a scrolling feature.

2.5.2 Interrupts

Before the keyboard driver, we need to handle interrupts. Interrupts are handled on a vector, with entries which are similar to those of the GDT. However, instead of programming the IDT in assembly, we have done it in C. We define the structure of entries in the IDT and define handlers for the structs.

Interrupt Service Routines (ISR) are run everytime there is an interrupt. We have defined 32 of them manually in a header file with a function to install ISRs and load IDT and also a handler for the ISRs.

The actual implementation of all the ISRs is done in assembler. It just saves the state, calls the handler and then restores the state.

The ISRs are invoked by interrupt requests (IRQ). So, now we need to define those. When the CPU boots, the PIC (Programmable Interrupt Controller) maps the IRQs 0-15 with the ISRs by default but that actually conflicts with the the ISRs we programmed since we did it 0-31. Now, we have to map IRQs from 32 to 47. Similar to ISR, we need to add them in the IDT and

write a handler for the IRQs. We also implement them in the assembler like before, to save/restore state and call IRQ handler. As for the IRQ handler, it sends EOIs (End of Interrupt) to the PIC and calls the appropriate handler based on the interrupt. We store all these handlers in an array and call them by index.

Now the IRQ structure is ready and we have an array of handlers but we haven't define the handlers yet. We first make a timer to compute clock frequency and send the bytes to the appropriate ports. Then, initialize the timer from the kernel.

2.5.3 Keyboard Driver

Now, we can move to keyboard. The PIC sends us the scancode for the key-down and key-up events (rather than the ASCII codes of the keys). We convert the scan codes into ASCII but we have only implemented a simple keyboard, so it might not feel the same as LINUX. We also create a callback that gets the ASCII code and initialization which configures the interrupt callbacks. The callback, once it gets the ASCII, prints the character on the screen and also appends it to a `key_buffer`. Our function can also parse backspace which removes the last character from the `key_buffer` and deletes it from the screen. When a `'\n'` is encountered we go to a function which checks the input in `key_buffer` and checks it to existing text to run some other functions.

2.5.4 Malloc

We built a Malloc function returns a pointer to a new unused memory address with an offset of 4096 bytes. This address can be used by the kernel as the user wants.

2.6 Building a shell

We already have an interface where everything that the user types is printed on the screen and it can also handle scrolling of the screen. To make the simplest shell, we just need to define some command keywords. When the user presses ENTER, the keyboard callback gets a newline character, which then calls the kernel. It then compares the input string with predefined strings and if it is the same, it runs the corresponding function.

Now, we have a simple shell with a few commands including "CLR" (clear screen), "ROC" (Rocket animation), "MEM" (Memory allocation), "END" (Halt the CPU).

Chapter 3

System Study and Analysis

3.1 System

For this project we have used QEMU-x86_64 along with a cross compiler built using GCC 4.9, Bison 3.5.1, GNU make 4.2.1, FLEX 2.6.4, libgmp3-dev, libmpc-dev, libmpfr-dev, texinfo, libisl-dev.

3.2 Analysis

The OS is run from an os-image.bin file created by concatenating using the bootsector.bin file and kernel.bin file. The bootsector.bin file is created using the various boot routines along with the bootsector.asm file. The kernel.bin file is generated by linking the kernel_entry.o file and kernel.o file. The kernel_entry.o file is generated from the kernel_entry.asm file which calls the kernel.c file which created the kernel.o file. The kernel.0 file is linked with all the C code that we have written for drivers and shell.

3.3 Path taken by the OS

The OS starts in 16 bit mode and can be seen on the starting screen of the OS. Then the disc is loaded to the memory. Then after all unnecessary functions are stopped and new necessary variables and functions are created. Then we switch to 32 bit mode which is necessary for a secure and smooth OS. Then the linked kernel_entry function is started which calls to the kernel_main function. In the kernel_main function we set the ISRs and IRQs. The IRQ initializes the keyboard and the keyboard_callback function is called. The keyboard_callback function is called which scans for a text input and when

the input character is '\n' the user_input is called which checks the current input for all the given specifics. This keyboard_callback function is called till the user enters an 'END' command.

Chapter 4

Presentation of Results

4.1 Start of the OS

```
32-bit Protected Mode.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8CB00+07ECCB00 CA00

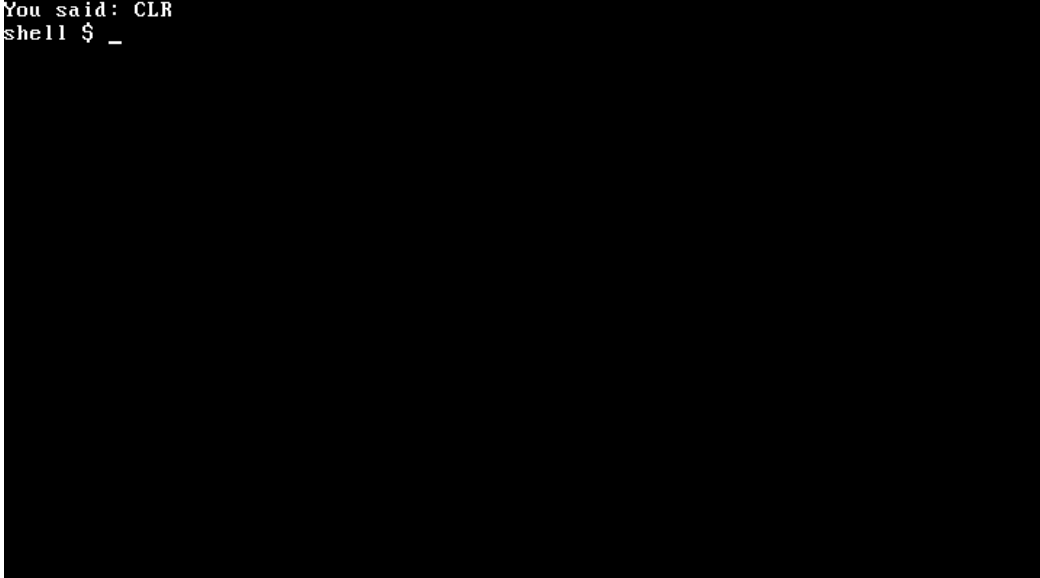
Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
16-bit Real Mode
Loading kernel into memory
Type something, it will go through the kernel
Type ROC for animation
Type CLR to clear the screen
Type END to halt the CPU
Type MEM to request a page address
Type any other text to reflect
shell $
```

Start screen of the OS

4.2 Shell commands

```
You said: CLR  
shell $ _
```



The clear screen command "CLR", clears the screen



```
  ^  
 / \  
| - |  
| R |  
| O |  
| C |  
| K |  
| E |  
| T |  
| | | | |
| | |  
|_|_|_|_|  
  _ _ _ _ _
```

The rocket command "ROC", animation of a Rocket

```
You said: CLR
shell $ MEM
Page: 0x11000, physical address: 0x11000
You said: MEM
shell $ MEM
Page: 0x12000, physical address: 0x12000
You said: MEM
shell $ END
Stopping the CPU. Bye!
-
```

The memory allocator command "MEM", gives an unused memory address of 4096 bytes

Chapter 5

Results and Conclusion

We were able to create an Operating System from scratch. We started with printing character in 16 bit mode and later went on to switch to a proper kernel written in a higher level language. Our OS currently is able to boot in 32 bit mode and recognize inputs from the user and print using the VGA memory. It also has a basic shell which is able to run the most basic commands.

5.1 Future Prospects

Future prospects for this project are:

- Create a file system for the OS
- Load and store files
- Create a directory tree
- Run multiple processes
- Memory management
- Create LINUX like commands in the custom shell
- Create a GUI and Networking with the shell

References

- Nick Blundell, Writing a Simple Operating System — from Scratch
- Nick Blundell - Youtube
- OS-Dev
- osdever.net
- OS-series by David Callanan
- OS-tutorial by Carlos Fenollosa
- OS-dev.pdf