DEVELOPING AN x64 OPERATING SYSTEM

A REPORT

submitted by

Lakshmi K G (19BCE1114)

Nihal Mubeen (19BCE1213)

in partial fulfilment for the award

of

B. Tech. Computer Science and Engineering

School of Computer Science and Engineering

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

June 2021

School of Computer Science and Engineering

DECLARATION

We hereby declare that the project entitled **"Developing an x64 Operating System"** submitted by us to the School of Computer Science and Engineering, Vellore Institute of Technology, Chennai Campus, Chennai 600127 in partial fulfilment of the requirements of **Embedded Project for the Course CSE 2005 – Operating Systems** is a record of bona fide work carried out by us**.** We further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, in any other institute or university.

Signature

Lakshmi K G, 19BCE1114

Nihal Mubeen, 19BCE1213

ACKNOWLEDGEMENT

We would like to express our special thanks to ***Dr. Jagadeesh Kannan R***, Dean of the School of Computer Science & Engineering, VIT Chennai; ***Dr. Geetha S***, Associate Dean of the School of Computer Science & Engineering, VIT Chennai and our faculty ***Dr. Menaka Pushpa A*** , Assistant Professor(Sr), VIT Chennai for providing us with the facilities and other help required to complete this project.

TABLE OF CONTENTS

ABSTRACT **6**
INTRODUCTION **7**
1.1 PURPOSE OF THE PROJECT 7
PROJECT ANALYSIS **8**
2.1 INTRODUCTION 8
2.2 HARDWARE AND SOFTWARE REQUIREMENTS 8
2.2.1 Hardware Requirements: 8
2.2.2 Software Requirements: 8
2.3 LIMITATIONS 9
2.4 RELATED WORK 9
2.5 PROPOSED WORK 9
SETUP AND CREATING OUR FIRST OS **10**
3.1 PROJECT SETUP 10
3.2 CREATNG OUR FIRST OS 10
3.2.1 Loader.s 10
3.2.2 Linker Script 11
3.2.3 Creating the ISO Image for the Operating System 11
3.3 THE OUTPUT 12
SETTING UP C AND BUILD SYSTEM **13**
4.1 SETTING UP C 13
4.1.1 Creating the stack 13
4.1.2 Calling C code from assembly 13
4.2 SETTING UP THE BUILD SYSTEM 14
4.3 OUTPUT 15
MAKING THE OS DISPLAY OUTPUT **16**
5.1 THE FRAMEBUFFER 16
5.1.1 Writing Text 16
5.1.2 Moving the Cursor 16
5.1.3 The code 17
5.2 THE SERIAL PORTS 18

5.2.1 Configuring the line for sending data....19
5.2.2 Configuring the Buffers....19
5.2.3 Configuring the Modem....19
5.2.4 Writing data to Serial Port....20
5.2.5 Codes....20
5.3 KMAIN.C....21
5.4 OUTPUT....21
MEMORY SEGMENTATION....23
6.1 INTRODUCTION....23
6.2 GLOBAL DESCRIPTOR TABLE....24
6.3 THE CODE....24
6.4 OUTPUT....26
INTERRUPTS & INPUT FROM THE KEYBOARD....27
7.1 INTERRUPTS....27
7.2 INTERRUPT DESCRIPTOR TABLES....27
7.2.1 Creating the Interrupt Handlers....28
7.2.2 Loading the IDT....29
7.3 PROGRAMMABLE INTERRUPT CONTROLLER....29
7.4 READING INPUT FROM KEYBOARD....30
7.4.1 Take Input from scan code and convert to ASCII....30
7.4.2 Interrupt Handling for Keyboard....31
7.5 OUTPUTS:....33
CONCLUSION AND FUTURE ENHANCEMENT....34
REFERENCES....35
APPENDIX....35

ABSTRACT

In this project, we have created a standalone bare-bones x64 operating system. We have created our OS completely from scratch. The project was made on an Ubuntu 20.04 Virtual Machine running on a Windows 10 OS in Oracle Virtual Box Virtual Machine. Our OS was run on bochs. This document outlines the step-by-step procedure involved in making our Operating System and can be followed by anyone interested in making an OS from scratch.

CHAPTER 1

INTRODUCTION

With this project, we intend to write our own x64 operating system. We have coded the OS completely from scratch using assembly programming as well as C code. For creating this OS, we have ***set up our development environment, boot kernel in virtual machine, set up the OS to execute C code, code the OS to display text on the screen (framebuffer), code OS to send data to serial ports (serial buffer), set up memory segmentation for the OS and implement interrupts to take in input from the keyboard and print to screen.*** The procedures we followed for the above tasks have been detailed in this report.

1.1 PURPOSE OF THE PROJECT

- ✓ To develop a basic x64 operating system
- ✓ To lean the steps involved in coding an OS kernel
- ✓ To get a deeper understanding of and gain more insight into the concepts learnt by us in our Operating System Theory classes.
- ✓ Produce a step-by-step tutorial and documentation on how to create an OS from scratch
- ✓ This OS can be developed upon to make more complex operating systems.

CHAPTER 2

PROJECT ANALYSIS

2.1 INTRODUCTION

This section of the report will describe the hardware and software requirements of this project. It will also explain how we setup the development environment required for the creation of this project.

2.2 HARDWARE AND SOFTWARE REQUIREMENTS

This project was executed on an Ubuntu Virtual Machine running on a Windows 10 Operating Systems. Detailed specifications are given below.

2.2.1 Hardware Requirements:

Host Computer Specifications

- Processor : Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
- RAM : 8.00 GB (7.79 GB usable)
- System type : 64-bit operating system, x64-based processor

Virtual Machine Specifications:

- Base Memory : 2048 MB
- Video Memory : 16 GB
- Processors allotted : 1

2.2.2 Software Requirements:

Host Computer Software Requirements :

- Host OS : Windows 10 Home Single Language
- OS build 19042.985
- Virtual Machine Manager : VirtualBox 6.1

Virtual Machine Software Requirements :

- Operating System : Ubuntu 20.04
- Build System : Make
- Packages Required :
    - Build-essential
    - Nasm
    - Genisoimage
    - Bochs, Bochs-sdl, Bochs-x
- Programming Language : Assembly Language, C (complied with gcc)
- Virtual Machine Manager : bochs
- Bootloader – GRUB Legend

2.3 LIMITATIONS

This is a very bare-bones Operating System with a lot of room for improvement. It does not provide the OS development industry with any new solutions to existing problems. Rather, it only provides the authors (and anyone else who implements this project on their own) with an application of OS development principles.

2.4 RELATED WORK

This is a very basic x64 Operating Systems and can be developed into more complex OS. As a result, it is closely related to various other x64 bit Operating Systems in the market.

2.5 PROPOSED WORK

Rather than developing on existing kernels, we have, in this project, developed an entire operating system from scratch.

Advantage of Proposed Project:

- ✓ Easy to understand
- ✓ Can be used to learn about OS development
- ✓ Implements important OS Development principles
- ✓ This kernel can be used to develop more complex operating systems.

CHAPTER 3

SETUP AND CREATING OUR FIRST OS

3.1 PROJECT SETUP

Before we began coding the Operating System, we needed to set up our developing environment and install all the dependencies required for executing our project. We began by installing the dependencies using the ubuntu terminal with the following command:

sudo apt-get install build-essential nasm genisoimage bochs bochs-sdl

3.2 CREATNG OUR FIRST OS

For our first OS, we implemented an operating system that has one function and one function only – to write "cafebabe" to the EAX register. "0xCAFEBABE" is the Java class file's magic number the first four bytes of every Java class file is specified to be this magic number. This was the simplest possible OS that we could create. We chose this word since it is highly unlikely for this value to be present in the register without us putting it there.

3.2.1 Loader.s

For our first OS, we wrote the OS in assembly code since coding with C requires setting up a stack which we have not implemented yet. The code for loader.s is given below:

```
global loader

MAGIC_NUMBER equ 0x1BADB002
FLAGS        equ 0x0
CHECKSUM     equ -MAGIC_NUMBER

section .text              ; start of the text (code) section
align 4                    ; the code must be 4 bytes aligned
    dd MAGIC_NUMBER
    dd FLAGS               ; the flags,
    dd CHECKSUM            ; and the checksum

loader:
    mov eax, 0xCAFEBABE
.loop:
    jmp .loop              ; loop forever
```

3.2.2 Linker Script

We then had o to write a linker code so as to convert our object files to one executable program. This was saved as **“link.ld”**

```
ENTRY(loader)

SECTIONS {
  . = 0x00100000;

  .text ALIGN (0x1000) :
  {
    *(.text)
  }
  .rodata ALIGN (0x1000) :
  {
    *(.rodata*)
  }
  .data ALIGN (0x1000) :
  {
    *(.data)
  }
  .bss ALIGN (0x1000) :
  {
    *(COMMON)
    *(.bss)
  }
}
```

We used link.ld to link the executable with the following command :

ld -T link.ld -melf_i386 loader.o -o kernel.elf

3.2.3 Creating the ISO Image for the Operating System

In order to create the ISO image for the operating system, we have used the genisoimage command as follows:

```
genisoimage -R                    \
-b boot/grub/stage2_eltorito      \
-no-emul-boot                     \
-boot-load-size 4                 \
-A os                             \
-input-charset utf8               \
-quiet                            \
-boot-info-table                  \
-o os.iso                         \
iso
```

3.3 THE OUTPUT

The OS was run in the bochs emulator using the command

bochs -f bochsrc.txt -q[1]

Output Screenshots:

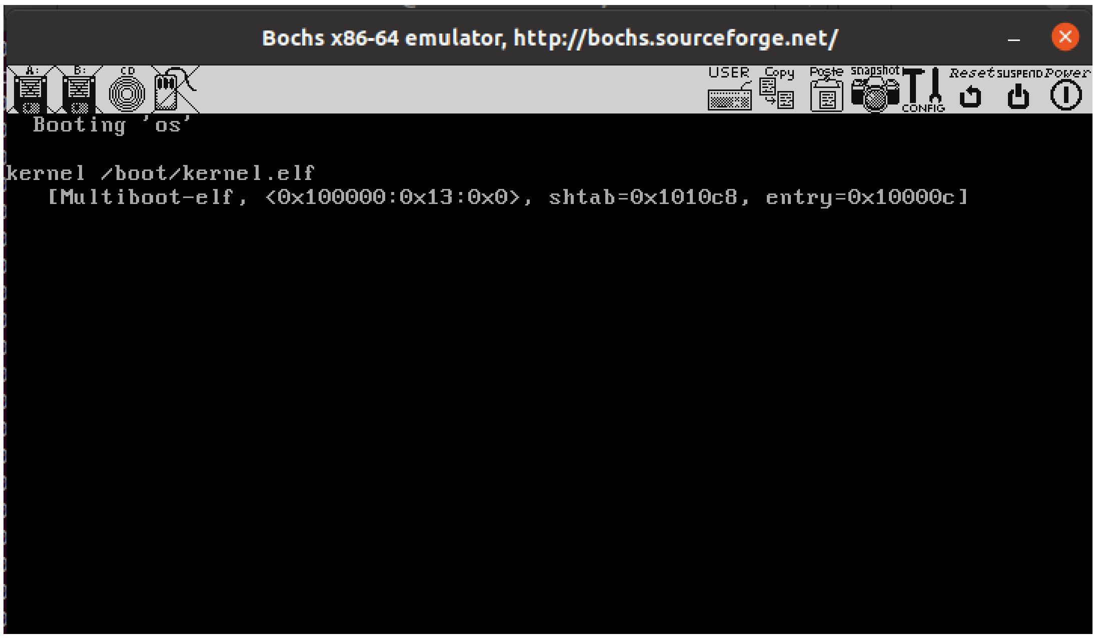


Fig1 : The OS after running in bochs emulator

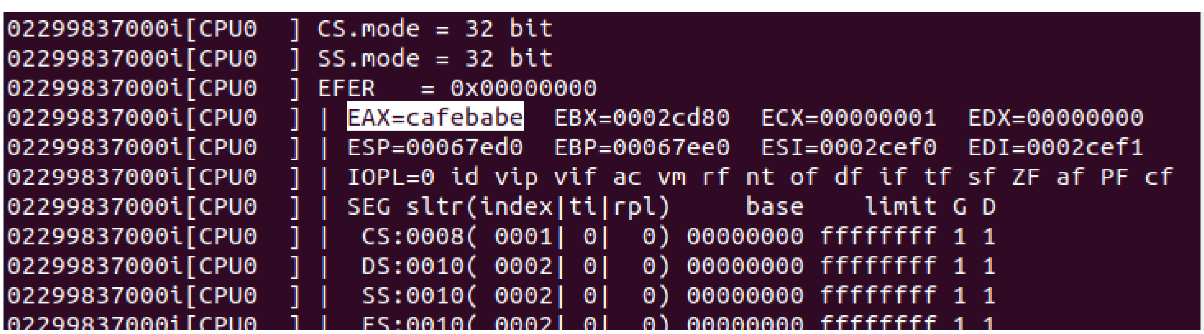


Fig 2: The bochs log which is stored in bochslog.txt. The log clearly shows that the value "cafebabe" has been inserted into the EAX register

[1] (bochsrc.txt is given in Appendix)

CHAPTER 4

SETTING UP C AND BUILD SYSTEM

4.1 SETTING UP C

After successfully running our first OS, we then proceeded to set up our OS such that we can code in C instead of assembly langauge. For this, we first created a stack.

4.1.1 Creating the stack

To create a stack, we used a .bss section (block starting symbol) of unintialized memory (4096 bytes). This was done by adding the following section in loader.s

```
section .bss:
align 4

kernel  stack:
        resb KERNEL_STACK_SIZE
```

We also set up the stack pointer by pointing esp to the end of the kernel_stack with the following line of code in the loader section:

```
mov esp, kernel_stack + KERNEL_STACK_SIZE
```

4.1.2 Calling C code from assembly

To call the C code from assembly, we first created a simple c code in a file kmain.c with the function kmain() as :

```
int kmain(int arg1, int arg2, int arg3)
   {
      return arg1 + arg2 + arg3;
   }
```

Further, we added the following code to loader.s so as to call the c code. Now the returned value from kmain will be stored in eax register

```
external kmain
push dword 3
push dword 2
push dword 1
call kmain
```

4.2 SETTING UP THE BUILD SYSTEM

We have also set up some build tools to make it easier to compile the C code and run the emulator. We have used the make build system in our project. The Makefile for our OS is:

```
OBJECTS = loader.o kmain.o
CC = gcc
CFLAGS = -m32 -nostdlib -fno-builtin -fno-stack-protector \
        -Wno-unused -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c -masm=intel
LDFLAGS = -T link.ld -melf_i386
AS = nasm
ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
	ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
	cp kernel.elf iso/boot/kernel.elf
	genisoimage -R                          \
	-b boot/grub/stage2_eltorito            \
	-no-emul-boot                           \
	-boot-load-size 4                       \
	-A os                                   \
	-input-charset utf8                     \
	-quiet                                  \
	-boot-info-table                        \
	-o os.iso                               \
	iso


run: os.iso
	bochs -f bochsrc.txt -q
%.o: %.c
	$(CC) $(CFLAGS) $< -o $@

%.o: %.s
	$(AS) $(ASFLAGS) $< -o $@

clean:
	rm -rf *.o kernel.elf os.iso
```

As we add more object files to the OS, the object files will be added to the “OBJECTS”.

4.3 OUTPUT

Upon setting up the makefile, we could run our OS easily with the command

make run

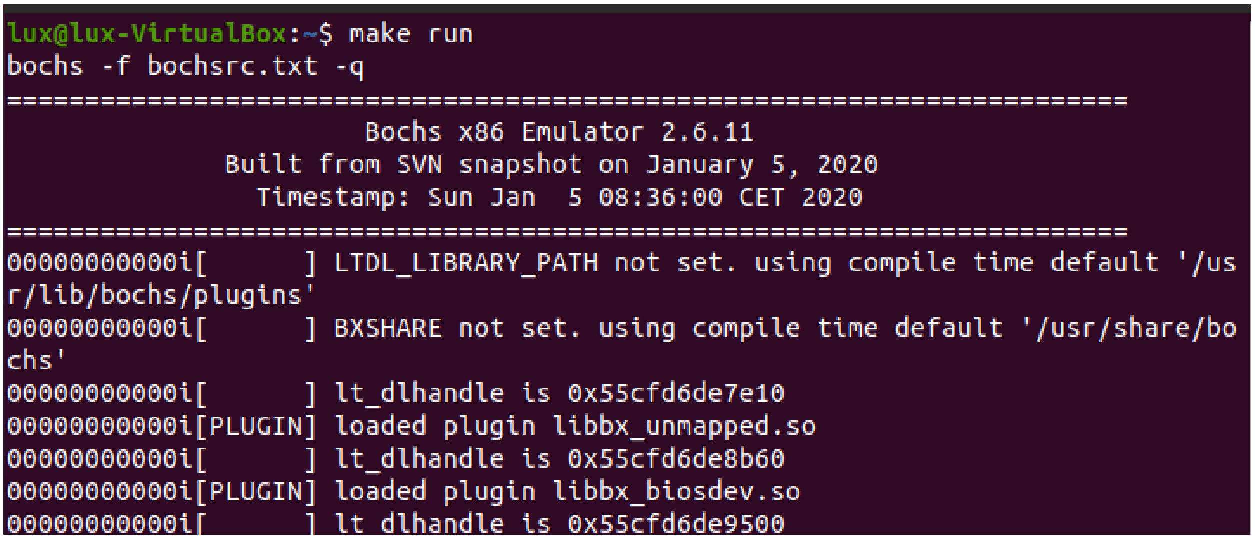


Fig 3: The Bochs emulator is started with the command "make run"

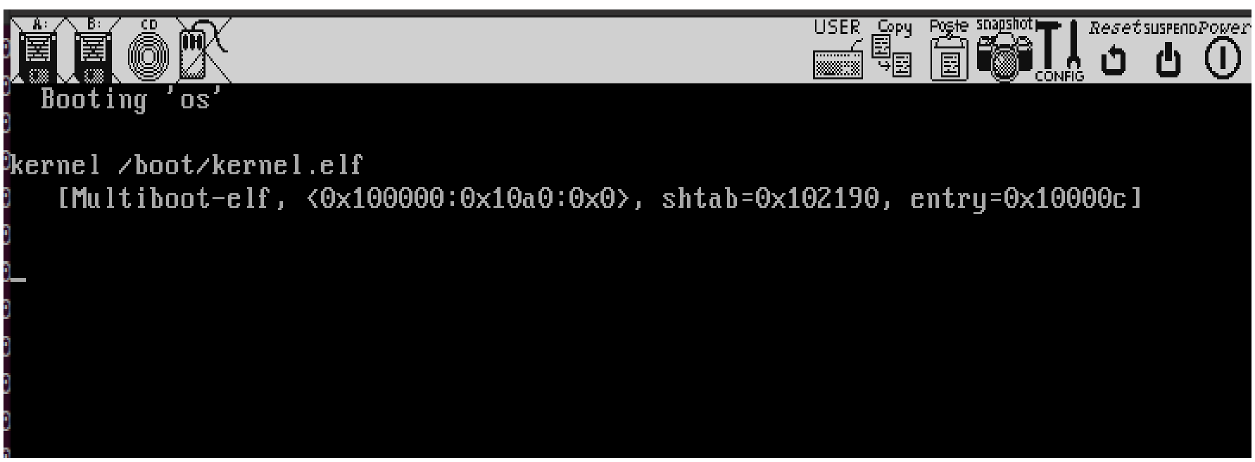


Fig 4 : The OS after running the C code.

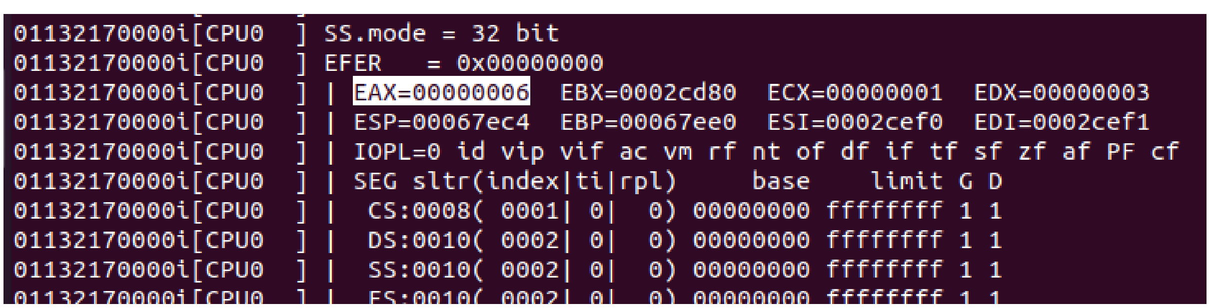


Fig 5: Bochs log shows that the value returned by kmain.c (3+2+1 =6) from the c code is stored in EAX register.

CHAPTER 5

MAKING THE OS DISPLAY OUTPUT

Our OS can display text on the console as well as write data to the serial port. For this, we have created drivers – which is a code that acts as a layer between the kernel and the hardware thereby creating a higher abstraction.

5.1 THE FRAMEBUFFER

Framebuffer is a portion of RAM containing a bitmap that drives video display. It has 80 columns and 25 rows and the indices for both start at 0.

5.1.1 Writing Text

Writing text is doing using memory mapped I/O. The memory is divided into 16-bit cells of which the first 8 bits is the ASCII value of the character, the next 4 bits describes the background and last 4 the foreground. The starting address of the framebuffer is 0x000B8000.

e.g. : mov [0x000B8000], 0x4128 //instruction to print A with a green foreground and dark grey background

5.1.2 Moving the Cursor

The position of the cursor is described using a 16-bit integer as follows :

0 = row 0, column 1 ; 1 = row 0, column 1 ; 80 = row 1, column 0 etc. To move cursor to (0,1) the instructions are as follows:

```
out 0x3D4, 14
out 0x3D5, 0x00
out 0x3D4, 15
out 0x3D5, 0x50
```

5.1.3 The code

Framebuffer.h

```
#ifndef INCLUDE_FRAMEBUFFER_H
#define INCLUDE_FRAMEBUFFER_H

#include "io.h"

#define FB_COMMAND_PORT         0x3D4
#define FB_DATA_PORT            0x3D5

#define FB_HIGH_BYTE_COMMAND    14
#define FB_LOW_BYTE_COMMAND     15

#define FRAMEBUFFER_WIDTH     80
#define FRAMEBUFFER_HEIGHT    25

extern char *__fb;

extern unsigned short __fb_present_pos;

void fb_move_cursor(unsigned short pos);

void fb_write_cell(unsigned int i, char c, unsigned char fg, unsigned char bg);

int fb_write(char *buf, unsigned int len);

#endif
```

Framebuffer.c

```
#include "framebuffer.h"
#inlude “io.h”

char *    __fb = (char *) 0x000B8000;
unsigned short __fb_present_pos = 0x00000000;

void fb_move_cursor(unsigned short pos) {
        outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
        outb(FB_DATA_PORT,       ((pos >> 8) & 0x00FF));
        outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
        outb(FB_DATA_PORT,       pos & 0x00FF);
        __fb_present_pos = pos;
}

void fb_write_cell(unsigned int i, char c, unsigned char fg, unsigned char bg) {
        __fb[i] = c;
        __fb[i + 1] = ((fg & 0x0F) << 4) | (bg & 0x0F);
}

int fb_write(char *buf, unsigned int len) {

        unsigned int i;
        for(i=0; i<len; i++) {

                fb_write_cell(2* __fb_present_pos, buf[i], (unsigned char)0, (unsigned char)15);
                fb_move_cursor(__fb_present_pos + 1);
```

```
            if(   fb  present  pos == FRAMEBUFFER  WIDTH * FRAMEBUFFER  HEIGHT) {
                fb_move_cursor((FRAMEBUFFER_HEIGHT-1)*FRAMEBUFFER_WIDTH);

                int j;
                for(j=0; j<2*FRAMEBUFFER  WIDTH*(FRAMEBUFFER_HEIGHT-1);  j++) {
                    __fb[j] = __fb[j + FRAMEBUFFER_WIDTH*2];
                }
                for(j=j; j<2*FRAMEBUFFER_WIDTH*FRAMEBUFFER_HEIGHT;  j++) {
                    __fb[j] = 0;
                }
            }
        }
        return len;
}
```

io.h

```
#ifndef INCLUDE  IO  H
#define INCLUDE_IO_H

void outb(unsigned short port, unsigned char data);

unsigned char inb(unsigned short port);                //defined in next section – for serial buffer

#endif
```

io.s

```
global outb
global inb                                              //for serial ports – next section

outb:
   mov al, [esp + 8]
   mov dx, [esp + 4]
   out dx, al
   ret

inb:
mov dx, [esp + 4]
in  al, dx
ret
```

5.2 THE SERIAL PORTS

The serial port is an interface for communicate between hardware devices. For two hardware devices to communicate with each other, they must have a common:

- The speed for sending data (baud rate)
- Error checking (parity bits, stop bits)
- The number of bits that represent a unit of data (data bits)

5.2.1 Configuring the line for sending data

The serial port has an I/O port, the line command port, that is used for configuration. To configure the line, we have set the speed for sending data. The default speed of a serial port is 115200 bits/s. In order to send 2 results, the speed is set to 115200/2 = 57600 Hz. The way that the data is to be sent is configured using a byte of the following layout.

```
void serial  configure(unsigned short com, unsigned short divisor) {
        outb(SERIAL  DATA  PORT(com) + 1, 0x00);              // Disable all interrupts
        outb(SERIAL  LINE  COMMAND  PORT(com), 0x80);   // Enable DLAB (set baud rate divisor)
        outb(SERIAL  DATA  PORT(com), divisor & 0x00FF);   // Set divisor to (lo byte)
        outb(SERIAL  DATA  PORT(com) + 1, (divisor >> 8) & 0x00FF);   //(hi byte)
        outb(SERIAL  LINE  COMMAND  PORT(com), 0x03);   // 8 bits, no parity, one stop bit
        outb(SERIAL  FIFO  COMMAND  PORT(com), 0xC7);   // Enable FIFO, clear them, with 14-byte threshold
        outb(SERIAL_MODEM_COMMAND_PORT(com), 0x0B);// IRQs enabled, RTS/DSR set
}
```

```
Bit:          | 7 | 6 | 5 4 3 | 2 | 1 0 |
Content:      | d | b | prty  | s | dl  |
```

Name	Description
d	Enables (d = 1) or disables (d = 0) DLAB
b	If break control is enabled (b = 1) or disabled (b = 0)
prty	The number of parity bits to use
s	The number of stop bits to use (s = 0 equals 1, s = 1 equals 1.5 or 2)
dl	Describes the length of the data

We have used the standard value 0x03 [31], meaning a length of 8 bits, no parity bit, one stop bit and break control disabled.

```
int serial  is  transmit  fifo  empty(unsigned short com) {
        /* 0x20 = 0010 0000 */
        return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x30;
}
```

5.2.2 Configuring the Buffers

We have set the FIFO queue configuration byte as 0xC7 that:

- Enables FIFO
- Clears both receiver and transmission FIFO queues
- Use 14 bytes as size of queue

(Refer 5.2.1 for code)

5.2.3 Configuring the Modem

We have set the modem configuration byte as 0x003 that:

- Sets RTS = 1 (Ready to Transmit)
- DTR = 1 (Data Terminal Ready)

(Refer 5.2.1 for code)

5.2.4 Writing data to Serial Port

Writing data to serial port is done via the data I/O port. The “in” assembly command cannot be called directly from c and so, it is wrapped into function inb as shown in io.h and io.c codes in section 5.1.4. We also check if the transmit FIFO queue is empty before writing.

5.2.5 Codes

Serial.h

```
#ifndef _SERIAL_H_
#define _SERIAL_H_

#define SERIAL_COM1_BASE              0x3F8
#define DEBUG_SERIAL SERIAL_COM1_BASE

#define SERIAL_DATA_PORT(base)         (base)
#define SERIAL_FIFO_COMMAND_PORT(base)  (base + 2)
#define SERIAL_LINE_COMMAND_PORT(base)  (base + 3)
#define SERIAL_MODEM_COMMAND_PORT(base) (base + 4)
#define SERIAL_LINE_STATUS_PORT(base)   (base + 5)

#define SERIAL_LINE_ENABLE_DLAB        0x80

void serial_configure_baud_rate(unsigned short com, unsigned short divisor);
void serial_configure_line(unsigned short com);

int serial_is_transmit_fifo_empty(unsigned int com);

void serial_write(char *buf, unsigned int len);

#endif
```

Serial.c

```
#include "io.h"
#include "serial.h"

void serial_configure_baud_rate(unsigned short com, unsigned short divisor) {
      outb(SERIAL_DATA_PORT(com) + 1,0x00);
      outb(SERIAL_LINE_COMMAND_PORT(com), 0x80);
      outb(SERIAL_DATA_PORT(com), (divisor >> 8) & 0x00ff);
      outb(SERIAL_LINE_COMMAND_PORT(com), 0x30);
      outb(SERIAL_LINE_COMMAND_PORT(com), 0xC7);
      outb(SERIAL_LINE_COMMAND_PORT(com), 0x0B);
}

void serial_configure_line(unsigned short com)
{
  outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}

int serial_is_transmit_fifo_empty(unsigned int com) {
  return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}
```

```
void serial_write(char *buf, unsigned int len) {
  serial_configure_baud_rate(SERIAL_COM1_BASE, 3);
  serial_configure_line(SERIAL_COM1_BASE);

  unsigned int i;
  for (i=0; i<len; i++) {
    while (!serial_is_transmit_fifo_empty(SERIAL_COM1_BASE));

    outb(SERIAL_DATA_PORT(SERIAL_COM1_BASE), buf[i]);
  }
}
```

5.3 KMAIN.C

As discussed in chapter 4, kmain will be called from the loader.s file. This file is responsible for calling all other scripts in this project

```
#include "framebuffer.h"
#include "serial.h"

char myname[] = "First text in OS by Lakshmi KG and Nihal Mubeen";
char name2[] = "Hello World";

void kmain() {
        serial_configure(DEBUG_SERIAL, 2);

        fb_write(myname, sizeof(myname));
        serial_write(name2, sizeof(name2));
        fb_move_cursor(6*80);
}
```

5.4 OUTPUT

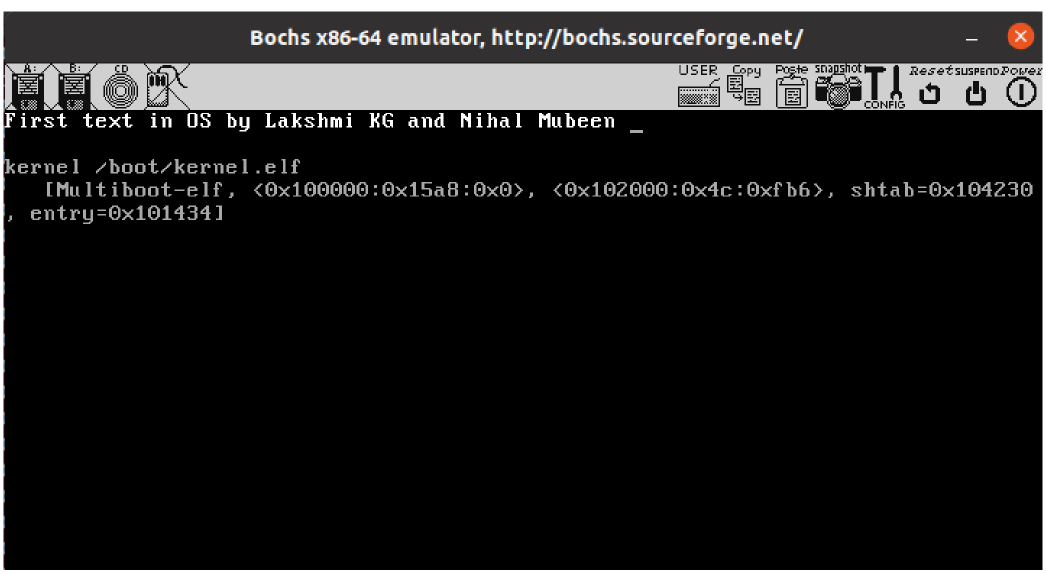


Fig 6 : Displaying the text using framebuffer. Note the position of the cursor.

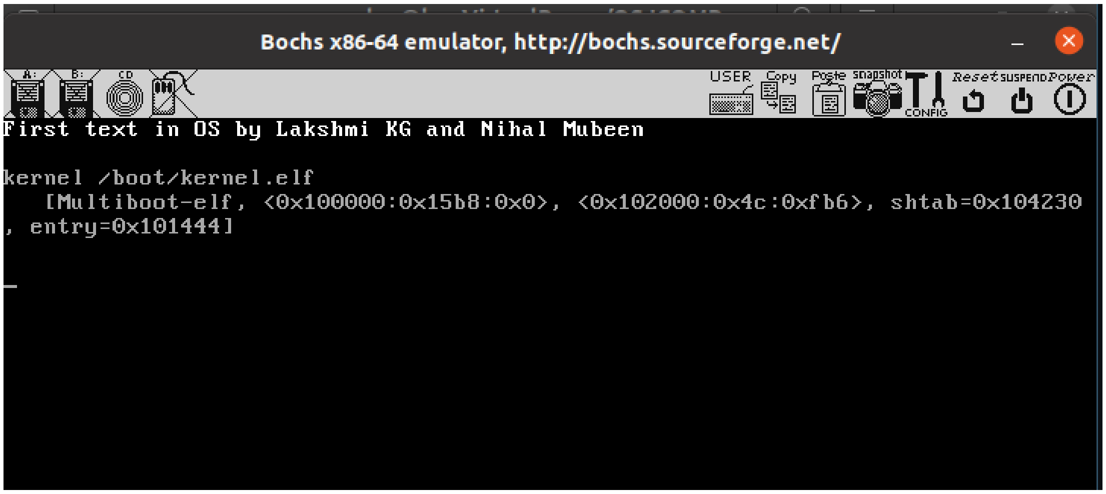


Fig 7 : Moving the cursor (from position in fig 6) using framebuffer

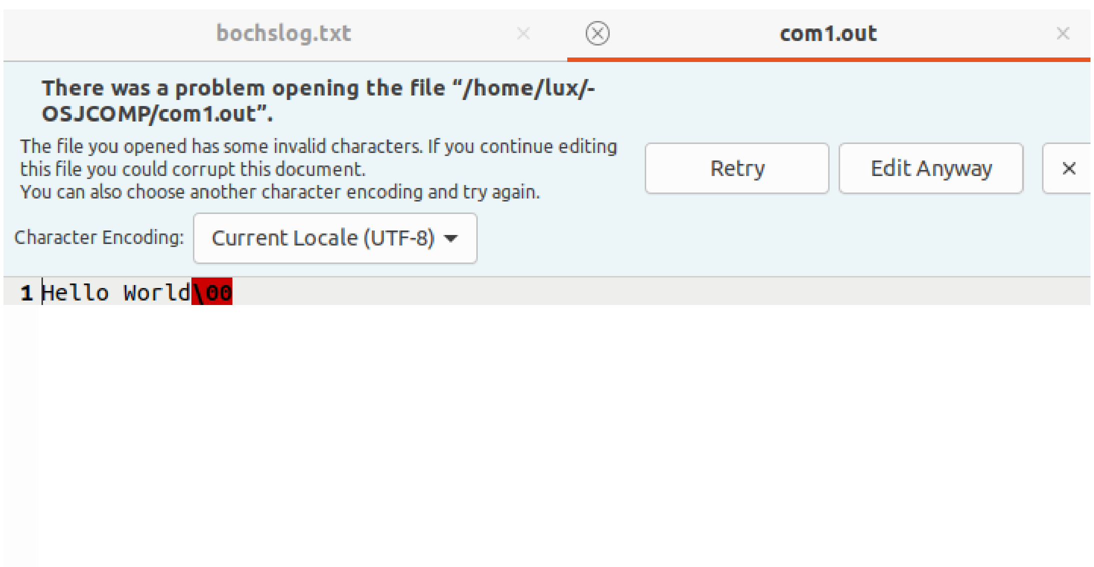


Fig 8: The output sent to serial port is saved as com1.out file.

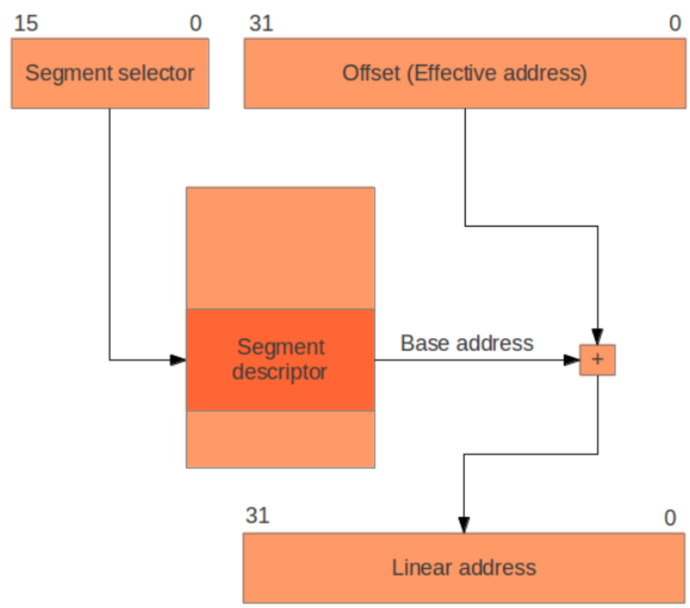
CHAPTER 6

MEMORY SEGMENTATION

6.1 INTRODUCTION

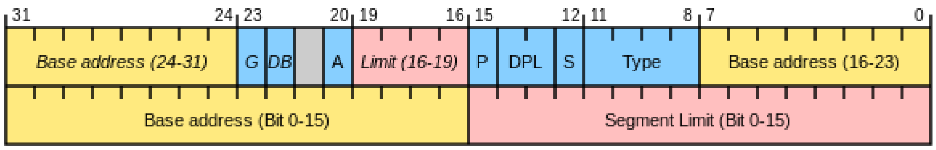
Segments are portions of the address space specifies by a base address and a limit. To address a byte within a segmented memory, we must use a 48-bit logical address constructed as follows :

- 16 bits that specify the segment
- 32 bits that specify the offset within the segment we want



To enable memory segmentation, we have created a segment descriptor table. Since we are building a comparatively simple OS, we have only built a Global Descriptor Table (GDT) and we did not make an LDT.

6.2 GLOBAL DESCRIPTOR TABLE



In this format for segmenr descriptor, the fields most important to this project are Type Field and Decriptor Priviledge Level (DPL). The Type field indicates the segment or gate type and specifies the kinds of access that can be made to the segment and the direction of growth. DPL specifies the privilege level of the segment. The privilege level can range from 0 to 3, with 0 being the most privileged level. The DPL is used to control access to the segment.

Table 3-1. Code- and Data-Segment Types

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

6.3 THE CODE

<u>Memory_Segments.h</u>

```
#ifndef INCLUDE_MEMORY_SEGMENTS
#define INCLUDE_MEMORY_SEGMENTS

struct GDT
{
        unsigned short size;
        unsigned int address;
} __attribute__((packed));

struct GDTDescriptor
{
        unsigned short limit_low;
        unsigned short base_low;
        unsigned char base_middle;
        unsigned char access_byte;
        unsigned char limit_and_flags;
```

```
	unsigned char base_high;
} __attribute__((packed));

void segments_init_descriptor(int index, unsigned int base_address, unsigned int limit, unsigned char access_byte, unsigned char flags);
void segments_install_gdt();

void segments_load_gdt(struct GDT gdt);
void segments_load_registers();

#endif
```

Memory Segments.c

```
#include "memory_segments.h"

#define SEGMENT_DESCRIPTOR_COUNT 3

#define SEGMENT_BASE 0
#define SEGMENT_LIMIT 0xFFFFF

#define SEGMENT_CODE_TYPE 0x9A
#define SEGMENT_DATA_TYPE 0x92

#define SEGMENT_FLAGS_PART 0x0C

static struct GDTDescriptor gdt_descriptors[SEGMENT_DESCRIPTOR_COUNT];

void segments_init_descriptor(int index, unsigned int base_address, unsigned int limit, unsigned char access_byte, unsigned char flags)
{
	gdt_descriptors[index].base_low = base_address & 0xFFFF;
	gdt_descriptors[index].base_middle = (base_address >> 16) & 0xFF;
	gdt_descriptors[index].base_high = (base_address >> 24) & 0xFF;

	gdt_descriptors[index].limit_low = limit & 0xFFFF;
	gdt_descriptors[index].limit_and_flags = (limit >> 16) & 0xF;
	gdt_descriptors[index].limit_and_flags |= (flags << 4) & 0xF0;

	gdt_descriptors[index].access_byte = access_byte;
}

void segments_install_gdt()
{
	gdt_descriptors[0].base_low = 0;
	gdt_descriptors[0].base_middle = 0;
	gdt_descriptors[0].base_high = 0;
	gdt_descriptors[0].limit_low = 0;
	gdt_descriptors[0].access_byte = 0;
	gdt_descriptors[0].limit_and_flags = 0;

	segments_init_descriptor(1, SEGMENT_BASE, SEGMENT_LIMIT, SEGMENT_CODE_TYPE, SEGMENT_FLAGS_PART);
	segments_init_descriptor(2, SEGMENT_BASE, SEGMENT_LIMIT, SEGMENT_DATA_TYPE, SEGMENT_FLAGS_PART);

	segments_load_gdt(*gdt_ptr);
	segments_load_registers();
}
```

Gdt.s

```
global segments_load_gdt
global segments_load_registers

segments_load_gdt:
        lgdt [esp + 4]
        ret

segments_load_registers:
        mov ax, 0x10
        mov ds, ax
        mov ss, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        jmp 0x08:flush_cs

flush_cs:
        ret
```

Kmain.c

```
#include "framebuffer.h"
#include "serial.h"
#include "memory_segments.h"

char myname[] = "First text in OS by Lakshmi KG and Nihal Mubeen";
char name2[] = "Hello World";

void kmain() {
        serial_configure(DEBUG_SERIAL, 2);

        fb_write(myname, sizeof(myname));
        serial_write(DEBUG_SERIAL, name2, sizeof(name2));
        fb_move_cursor(6*80);
        segments_install_gdt();
 }
```

6.4 OUTPUT

```
01226251000i[CPU0  ] | SEG sltr(index|ti|rpl)     base    limit G D
01226251000i[CPU0  ] |  CS:0008( 0001| 0|  0) 00000000 ffffffff 1 1
01226251000i[CPU0  ] |  DS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
01226251000i[CPU0  ] |  SS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
01226251000i[CPU0  ] |  ES:0010( 0002| 0|  0) 00000000 ffffffff 1 1
01226251000i[CPU0  ] |  FS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
01226251000i[CPU0  ] |  GS:0010( 0002| 0|  0) 00000000 ffffffff 1 1
01226251000i[CPU0  ] | EIP=001015df (001015df)
01226251000i[CPU0  ] | CR0=0x60000011 CR2=0x00000000
```

Fig 9 : The bochs log after creating the gdt and implementing memory segmentation

CHAPTER 7

INTERRUPTS & INPUT FROM THE KEYBOARD

7.1 INTERRUPTS

An interrupt occours when a hardware device connected to the computer signals to the CPU that there is a change in the state of the device. Interrupts can also be sent by the CPU in cases such as program errors. Software interrupts are called using the int assembly instruction and they are used for system calls.

7.2 INTERRUPT DESCRIPTOR TABLES

Interrupts are handled using the IDT. The interrupts are numbered (0-255) and a handler for interrupt i is defined at the i^{th} position in the table. An entry in the IDT consists of 64 bits in the following layout:

Highest 32 bits:

Bit:	31 16	15	14 13	12	11	10 9 8	7 6 5	4 3 2 1 0
Content:	offset high	P	DPL	0	D	1 1 0	0 0 0	reserved

Lowest 32 bits:

Bit:	31 16	15 0
Content:	segment selector	offset low

P – If the handler is present in memory or not (1 = present, 0 = not present)
DPL – Descriptor Privilege Level, the privilege level the handler can be called from (0, 1, 2, 3).
D – Size of gate, (1 = 32 bits, 0 = 16 bits).
segment selector – The offset in the GDT.

7.2.1 Creating the Interrupt Handlers

Interrupt_handlers.s

```
extern interrupt_handler

%macro no_error_code_interrupt_handler 1
global interrupt_handler_%1
interrupt_handler_%1:
   push      dword 0
   push      dword %1
   jmp       common_interrupt_handler
%endmacro

%macro error_code_interrupt_handler 1
global interrupt_handler_%1
interrupt_handler_%1:
   push   dword %1
   jmp    common_interrupt_handler
%endmacro

common_interrupt_handler:
   push   eax
   push   ebx
   push      ecx
   push      edx
   push      ebp
   push      esi
   push      edi

      call   interrupt_handler

   pop       edi
   pop       esi
   pop       ebp
   pop       edx
   pop       ecx
   pop       ebx
      pop    eax

   add    esp, 8

   iret

no_error_code_interrupt_handler       33                        ; interrupt 1
no_error_code_interrupt_handler       14                        ; interrupt 2
```

7.2.2 Loading the IDT

idt.s

```
global  load_idt
load_idt:
       mov eax, [esp + 4]
       lidt [eax]
       ret
```

7.3 PROGRAMMABLE INTERRUPT CONTROLLER

The PIC is an IC that helps microprocessors (or CPU) handle Interrupt Requests (IRQ) coming from multiple sources which may occour simultaneously. The PIC is responsible for:

- Remapping interrupts
- Select which interrupt you want to receive
- Set up the correct mode for the PIC

Pic.c

```
#include "io.h"
#include "pic.h"

void pic_acknowledge(unsigned int interrupt)
{
        if (interrupt < PIC_1_OFFSET || interrupt > PIC_2_END) {
                return;
        }

        if (interrupt < PIC_2_OFFSET) {
                outb(PIC_1_COMMAND_PORT, PIC_ACKNOWLEDGE);
        } else {
                outb(PIC_2_COMMAND_PORT, PIC_ACKNOWLEDGE);
        }
}

void pic_remap(int offset1, int offset2)
{
        outb(PIC_1_COMMAND, PIC_ICW1_INIT + PIC_ICW1_ICW4);
        outb(PIC_2_COMMAND, PIC_ICW1_INIT + PIC_ICW1_ICW4);
        outb(PIC_1_DATA, offset1);
        outb(PIC_2_DATA, offset2);
        outb(PIC_1_DATA, 4);
        outb(PIC_2_DATA, 2);

        outb(PIC_1_DATA, PIC_ICW4_8086);
        outb(PIC_2_DATA, PIC_ICW4_8086);

        outb(PIC_1_DATA, 0xFD);
        outb(PIC_2_DATA, 0xFF);

        asm("sti");
}
```

Pic.c

```
#ifndef INCLUDE_PIC_H
#define INCLUDE_PIC_H

#define PIC_1                  0x20
#define PIC_2                  0xA0
#define PIC_1_COMMAND          PIC_1
#define PIC_1_DATA             (PIC_1+1)
#define PIC_2_COMMAND          PIC_2
#define PIC_2_DATA             (PIC_2+1)

#define PIC_1_OFFSET 0x20
#define PIC_2_OFFSET 0x28
#define PIC_2_END PIC_2_OFFSET + 7

#define PIC_1_COMMAND_PORT 0x20
#define PIC_2_COMMAND_PORT 0xA0
#define PIC_ACKNOWLEDGE 0x20

#define PIC_ICW1_ICW4          0x01
#define PIC_ICW1_SINGLE        0x02
#define PIC_ICW1_INTERVAL4     0x04
#define PIC_ICW1_LEVEL         0x08
#define PIC_ICW1_INIT          0x10

#define PIC_ICW4_8086                  0x01
#define PIC_ICW4_AUTO                  0x02
#define PIC_ICW4_BUF_SLAVE             0x08
#define PIC_ICW4_BUF_MASTER            0x0C
#define PIC_ICW4_SFNM                  0x10

void pic_remap(int offset1, int offset2);
void pic_acknowledge(unsigned int interrupt);

#endif
```

7.4 READING INPUT FROM KEYBOARD

7.4.1 Take Input from scan code and convert to ASCII

Keyboard.h

```
#ifndef INCLUDE_KEYBOARD_H
#define INCLUDE_KEYBOARD_H

#define KEYBOARD_MAX_ASCII 83

unsigned char keyboard_read_scan_code(void);

unsigned char keyboard_scan_code_to_ascii(unsigned char);

#endif
```

Keyboard.c

```
#include "io.h"

#define KEYBOARD_DATA_PORT  0x60

unsigned char keyboard_read_scan_code(void)
{
        return inb(KEYBOARD_DATA_PORT);
}

unsigned char keyboard_scan_code_to_ascii(unsigned char scan_code)
{
        unsigned char ascii[256] =
        {
                0x0, 0x0, '1', '2', '3', '4', '5', '6',          // 0 - 7
                '7', '8', '9', '0', '-', '=', 0x0, 0x0,          // 8 - 15
                'q', 'w', 'e', 'r', 't', 'y', 'u', 'i',          // 16 - 23
                'o', 'p', '[', ']', '\n', 0x0, 'a', 's',         // 24 - 31
                'd', 'f', 'g', 'h', 'j', 'k', 'l', ';',          // 32 - 39
                '\'', '`', 0x0, '\\', 'z', 'x', 'c', 'v',        // 40 - 47
                'b', 'n', 'm', ',', '.', '/', 0x0, '*',          // 48 - 55
                0x0, ' ', 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,          // 56 - 63
                0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, '7',          // 64 - 71
                '8', '9', '-', '4', '5', '6', '+', '1',          // 72 - 79
                '2', '3', '0', '.'                               // 80 - 83
        };

        return ascii[scan_code];
}
```

7.4.2 Interrupt Handling for Keyboard

Interrupts.c

```
#include "interrupts.h"
#include "pic.h"
#include "io.h"

#include "serial.h"
#include "keyboard.h"
#include "framebuffer.h"

#define INTERRUPTS  DESCRIPTOR  COUNT  256
#define INTERRUPTS_KEYBOARD  33

struct IDTDescriptor idt_descriptors[INTERRUPTS_DESCRIPTOR_COUNT];
struct IDT idt;

void interrupts_init_descriptor(int index, unsigned int address)
{
idt  descriptors[index].offset  high = (address >> 16) & 0xFFFF;
idt_descriptors[index].offset_low = (address & 0xFFFF);

idt_descriptors[index].segment_selector = 0x08;
idt_descriptors[index].reserved = 0x00;
```

```
idt_descriptors[index].type_and_attr  = (0x01 << 7)  |
                                        (0x00 << 6) |
                                        (0x00 << 5) |
                                        0xe;
}

void interrupts_install_idt()
{
interrupts_init_descriptor(INTERRUPTS_KEYBOARD, (unsigned int) interrupt_handler_33);


idt.address = (int) &idt_descriptors;
idt.size = sizeof(struct IDTDescriptor) * INTERRUPTS_DESCRIPTOR_COUNT;
load_idt((int) &idt);

pic_remap(PIC_1_OFFSET, PIC_2_OFFSET);
}

void interrupt_handler(__attribute__((unused)) struct cpu_state cpu, unsigned int interrupt,
 __attribute__((unused)) struct stack_state stack)
{
unsigned char scan_code;
unsigned char ascii;

switch (interrupt){
        case INTERRUPTS_KEYBOARD:

                scan_code = keyboard_read_scan_code();

                if (scan_code <= KEYBOARD_MAX_ASCII) {
                ascii = keyboard_scan_code_to_ascii(scan_code);
                char str[1];
                str[0] = ascii;
                fb_write(str, 1);
                }

                pic_acknowledge(interrupt);

                break;

        default:
                break;
  }
}
```

7.5 OUTPUTS:

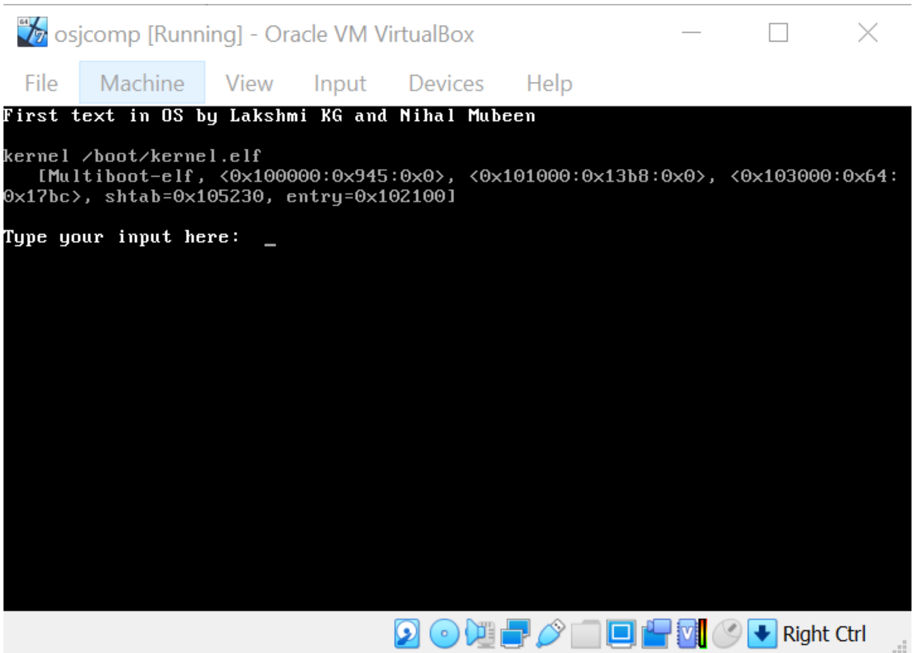


Fig 9: Before inputting from keyboard

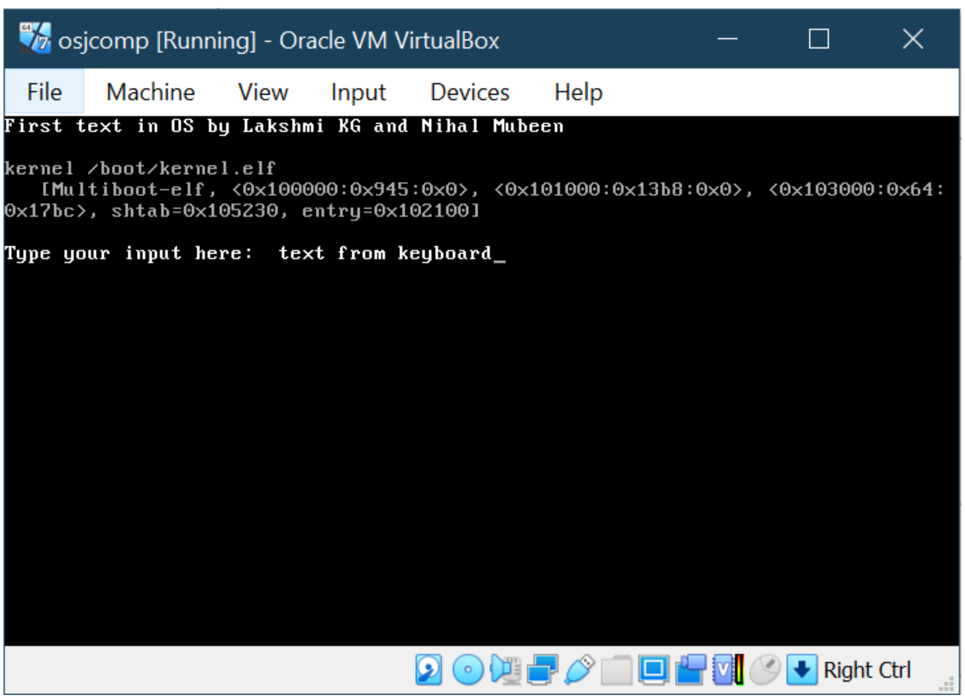


Fig 10: After inputting from keyboard

Link for video demonstration at :
https://drive.google.com/file/d/1lhRBkK9ARcCa0wvwq3MPBhlyglUvd8O1/view?usp=sharing

CHAPTER 8

CONCLUSION AND FUTURE ENHANCEMENT

We have completed our project **“Developing an x64 Operating System**”. Our current operating system can – run C code instead of assembly code, display output to the console, write to the serial outputs and implement memory segmentation. This is a very basic operating system with much room for improvement. The next step in the development of this OS would be improving on the keyboard interrupt to deal with new lines and backspace better. We will also be looking into implementing paging on this OS.

REFERENCES

Intel, *Intel Architecture Software Developer's Manual, vol. 3: System Programming*. 1999. https://web.archive.org/web/20050505161222/http://download.intel.com/design/PentiumII/manuals/24319202.pdf

OSDev Wiki. http://wiki.osdev.org/

Blundell, Nick. "Writing a Simple Operating System—from Scratch." (2009).

Helin, Erik, and Adam Renberg. "The little book about OS development."

APPENDIX

Bochsrc.txt

```
megs:            32
display_library: sdl2
romimage:        file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage:     file=/usr/share/bochs/VGABIOS-lgpl-latest
ata0-master:     type=cdrom, path=os.iso, status=inserted
boot:            cdrom
log:             bochslog.txt
clock:           sync=realtime, time0=local
cpu:             count=1, ips=1000000
com1:            enabled=1, mode=file, dev=com1.out
```

Menu.lst

```
default = 0
timeout = 0

title os
kernel /boot/kernel.elf
```

Makefile

```
OBJECTS = loader.o kmain.o framebuffer.o io.o serial.o memory_segments.o gdt.o
CC = gcc
CFLAGS = -m32 -nostdlib -fno-builtin -fno-stack-protector \
        -Wno-unused -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c -masm=intel
LDFLAGS = -T link.ld -melf_i386
```

```
AS = nasm
ASFLAGS = -f elf

all: kernel.elf

kernel.elf: $(OBJECTS)
	ld $(LDFLAGS) $(OBJECTS) -o kernel.elf

os.iso: kernel.elf
	cp kernel.elf iso/boot/kernel.elf
	genisoimage -R                              \
	-b boot/grub/stage2_eltorito    \
	-no-emul-boot                   \
	-boot-load-size 4               \
	-A os                           \
	-input-charset utf8             \
	-quiet                          \
	-boot-info-table                \
	-o os.iso                       \
	iso


run: os.iso
	bochs -f bochsrc.txt -q
%.o: %.c
	$(CC) $(CFLAGS)  $< -o $@

%.o: %.s
	$(AS) $(ASFLAGS) $< -o $@

clean:
	rm -rf *.o kernel.elf os.iso
```

Final kmain.c

```
#include "framebuffer.h"
#include "serial.h"
#include "memory  segments.h"
#include "interrupts.h"
#include "io.h"
#include "keyboard.h"

char myname[] = "First text in OS by Lakshmi KG and Nihal Mubeen";
char name2[] = "Hello World";
char sent[] = "Type your input here: ";

void kmain() {

	fb_write(myname, sizeof(myname));
	serial  write(name2, sizeof(name2));
	serial  write(sent, sizeof(sent));
	fb  move  cursor(6*80);
	fb_write(sent,sizeof(sent));
```

```
        segments_install_gdt();
        interrupts_install_idt();
}
```