Operating Systems

(Building an OS)

Team Members Involved

Lakshmi Priya Hariharan

Mili Upadhyaya Koirala

Experience and Computer Skills of Team Members:

Member 1: Lakshmi Priya Hariharan (Current MS Computer Science Student)

Experience

Programmer, Senior design Project, City College of New York, NY, August 2014- May 2015

Building Solar Panel under Prof. Ho and programming using Arduino for the micro-inverter.

Assistant, City College of New York, NY, NY May 2014-June 2014

Assisted Dr. Samleo.L. Joseph with his thesis on "Visual semantic parameterization - To enhance blind user perception for indoor navigation".

Student Assistant, High School of Math Science and Engineering, CCNY, NY, NY January 2012-May 2012

Assisted Professor Seth Guinals Kupperman in building the robots for the FTC robot challenge.

Computer Skills

C++, Python, Modelsim, Quartus, Multisim, Microsoft Office, Linux, Ubuntu, Data Science, HTML, Matlab, Microsoft Web Expression, Web design, File Maker Pro, Dream weaver, Visual Studio, Visio, Algorithms, Assembly Language, Emulator, Photoshop, Hadoop, R language.

Member 2: Mili Upadhyaya Koirala (Current MS Computer Science Student)

Experience

Senior Executive Officer, Prism International P Ltd, Kupondole, Lalitpur, Nepal, May 2010 – May 2011

Network Design and Implementation, Management, Project Execution.

Lecturer, Nagarjuna College of Information Technology, Pulchowk, Lalitpur, Nepal, May 2010 – Aug 2010

Undergraduate level course "Computer Organization" taught.

Senior Officer – IT, Prism International P Ltd, Kupondole, Lalitpur, Nepal, May 2008 – April 2010

Hardware debugging, Computer Networking and Troubleshooting, Client Monitoring, Wireless Network Setup, Router Configurations, Switch Configurations with VLANs, VPN Tunnel creations, VoIP Device Configurations. Also Micro Ethernet Switch Ring test for Redundancy.

Network Engineer, Prism International P Ltd, Kupondole, Lalitpur, Nepal, May 2007 – April 2008

Hardware debugging, Computer Networking and Troubleshooting, Client Monitoring, Lease Modem P2P Connections, SIP Device Configurations, ADSL Technical Support.

Projects:

Expense Claim System- a web application used by the IT firm for the Expense Management.

Technology/ Environment: J2EE with Spring Framework, MySQL with Hibernate Framework.

Academic Portal- a web application where a registered faculty can upload course materials and registered student can learn from those materials and give exams which were designed by facukty along with chat functionality.

Technology/ Environment: J2EE, MySQL, Apache Tomcat

Computer Skills:

Programming in C, C++, C#.Net, Java, Spring Framework, MySQL, Hibernate Framework, Unix, Linux, Photoshop, Visio, Electronic Workbench, MATLAB, AUTOCAD.

Lab 1 Session

Objective: Setup an environment to create a new Operating System.

Introduction: The Operating System is the name assigned to a group of computer programs, device drivers, kernel, and other things that let a user work with a computer. An operating system has various jobs. It is there to make sure that all the programs can use the CPU, system memory, displays, input devices and other hardware in a systematic way.

Initially, we create a team of two members. Focusing on the mobility and speed, we decide not to use the lab computers and used the computer with the below specification given in section 1.1.

Section 1

Requirements:

The system requires minimum of 512MB of RAM for the Virtual box installation and 1GB or more is preferred.

Section 1.1

Basic Information about the system used for the lab project:

Window 10 Pro

System:

Manufacturer: Microsoft Corporation

Processor: Intel® Core(TM) i5-6300U CPU @ 2.40GHz 2.50GHz

Installed memory: 4.00 GB

System type: 64-bit Operating System, x64-based processor

Section 2

Initially, the latest version of the VirtualBox 5.1.14 and Ubuntu 16.04.1 LTS were decided to be used for the project.

Section 2.1

Definition Oracle VirtualBox

VirtualBox is powerful Cross-platform Virtualization Software for x86-based systems.
"Cross-platform" means that it installs on Windows, Linux, Mac OS X and Solaris x86 computers. And "Virtualization Software" means that we can create and run multiple Virtual Machines, running different operating systems, on the same computer at the same time.

Section 2.2

Steps involved VirtualBox Setup

1. Download VirtualBox 5.1.14 from Oracle website. (Chose the package for Window 64-bit platform)
2. Double Click the setup file downloaded and follow the prompts to install. After the installation is complete we see the Virtual Box icon in your desktop as below:

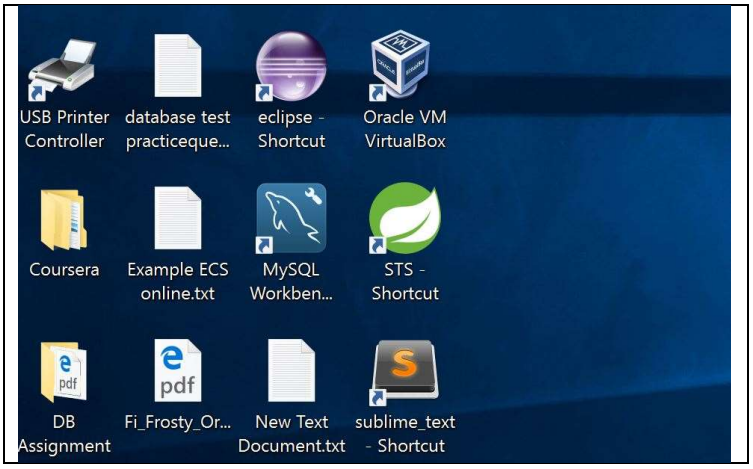
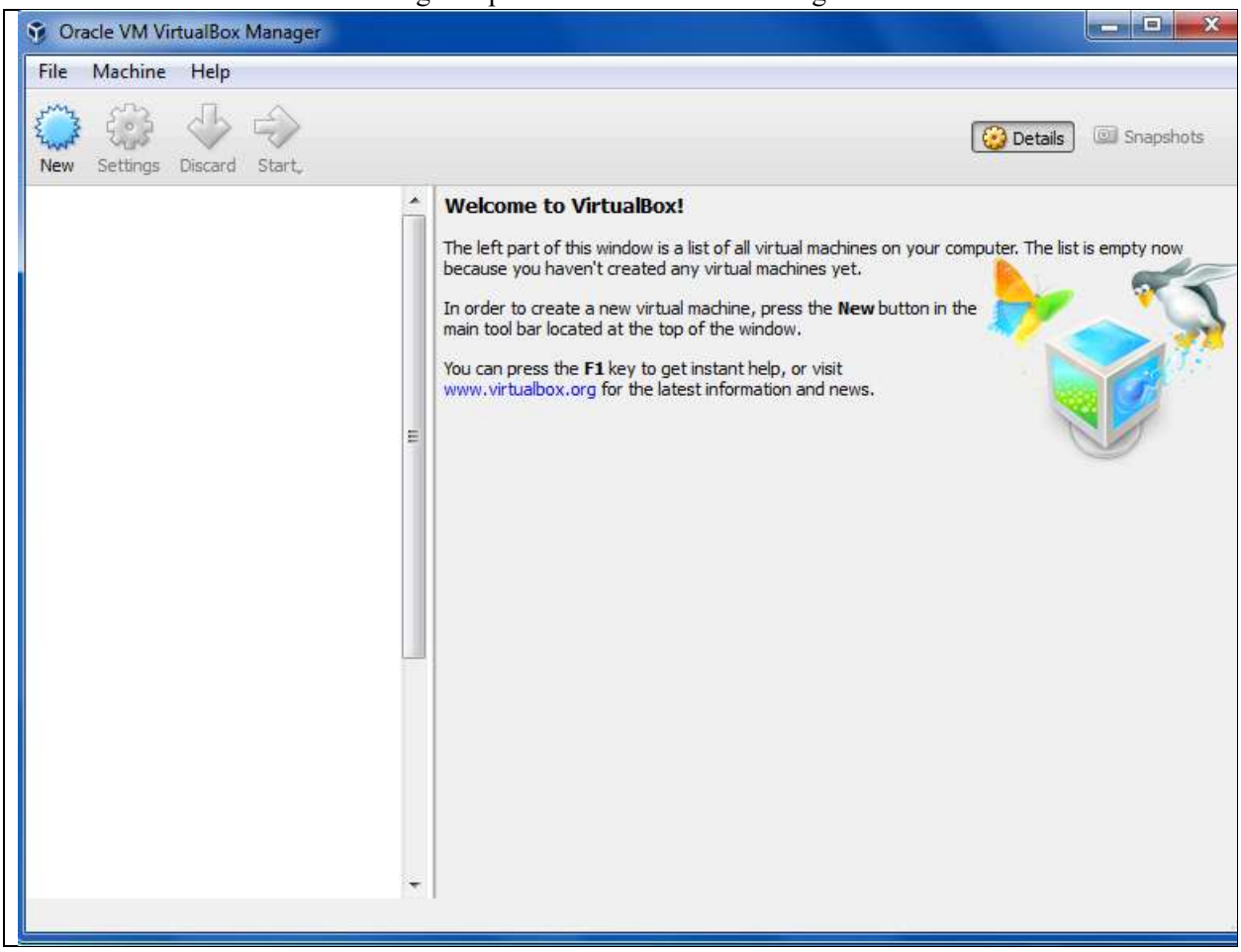


Fig: VirtualBox icon on the top right in desktop

3. Since we wish to have Ubuntu 16.04.1 LTS as Operating System in our new VirtualBox. We downloaded a .iso file named "ubuntu-16.04.1-desktop-amd64.iso"
4. Double click the desktop icon of VirtualBox to start Oracle VM VirtualBox Manager.

Fig. Snapshot Virtualbox VM Manager



5. We clicked the new tab at the top left corner for setting up the required Ubuntu setup. We already have the required Ubuntu downloaded in our machine in step 4.

Section 2.3

Definition Ubuntu

Ubuntu is a computer operating system based on the Debian Linux distribution and distributed as free and open source software, using its own desktop environment. It is named after the Southern African philosophy of ubuntu ("humanity towards others").

Recommended system requirements for Ubuntu: 2 GHz dual core processor or better

- 2 GB system memory
- 25 GB of free hard drive space
- Either a DVD drive or a USB port for the installer media
- Internet access is helpful

Section 2.4

Steps involved in Ubuntu Setup inside installed VirtualBox

1. Click new in the Oracle VM Virtual Manager and follow the instructions on the wizard. Give name as Ubuntu to OS.

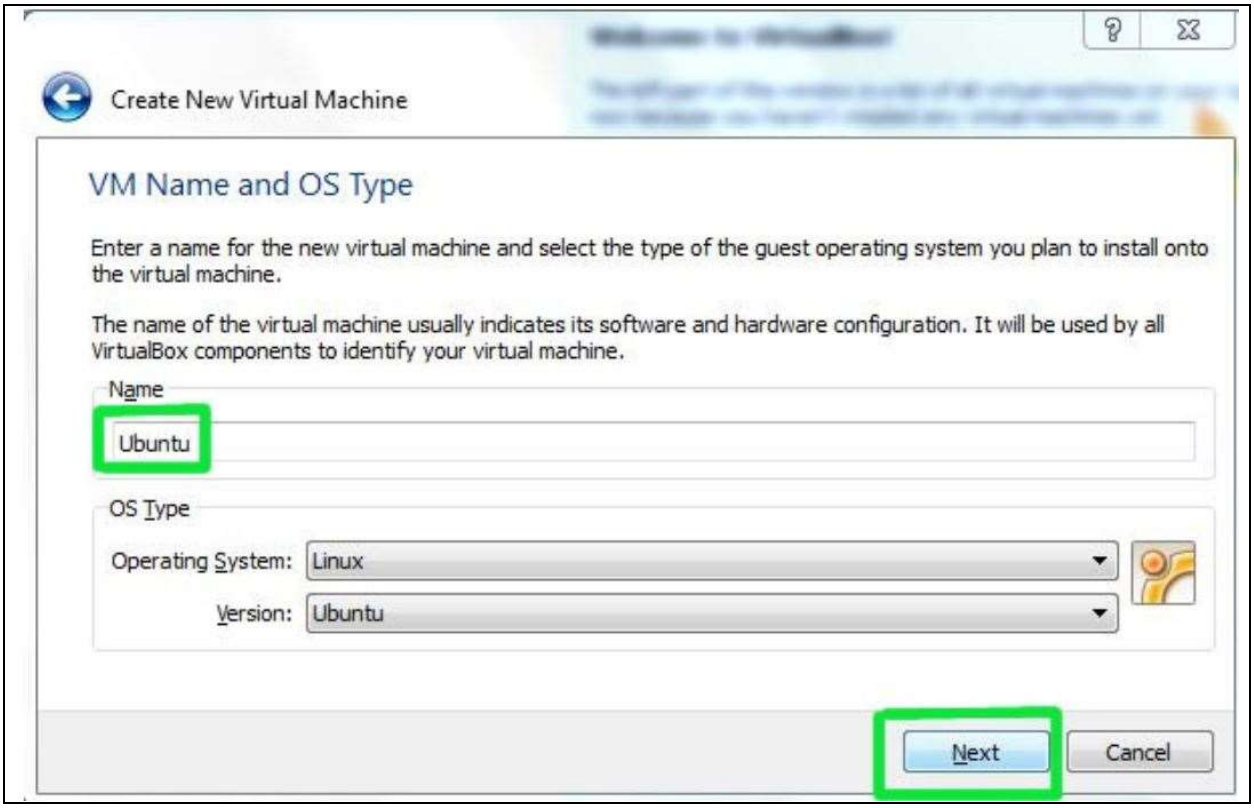


Fig. Snapshot Give name to the OS (Ref: http://www.psychocats.net/ubuntu/virtualbox)

2. Recommended memory(RAM) to allocate for Virtual machine. Since I have 4 GB RAM which is more than 1GB so I can allocate more than 512MB. I allocated 1536MB as the Base Memory of RAM.
3. Select a new hard disk and select a Fixed size in the Virtual disk storage.
4. Virtual disk file location and size. Ubuntu default installation is less than 3GB but I have allocated 10GB. And click on create and wait for virtual hard drive to be created.
5. Currently blank virtual hard drive is used to add the downloaded Ubuntu disk image (.iso) boot on our virtual machine. We click on settings and storage. Then under CD/DVD Device, next to empty, click the little folder icon.

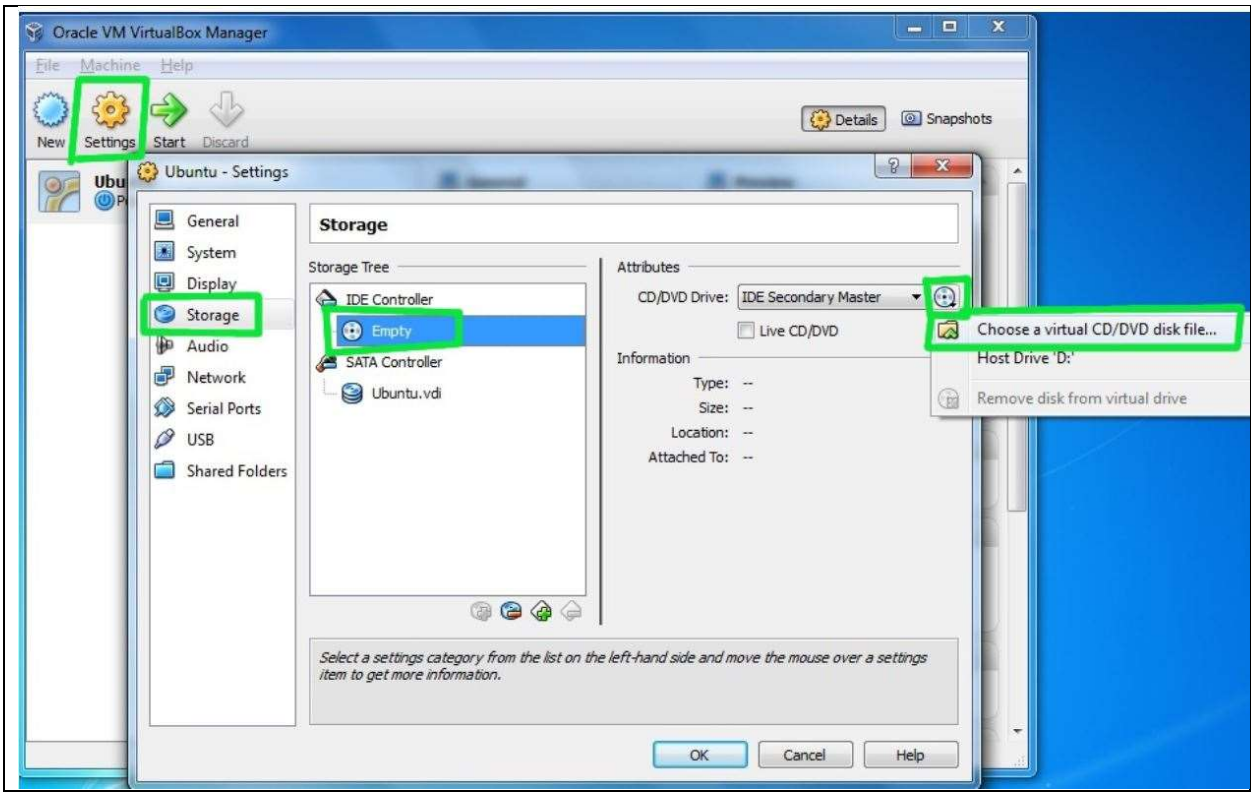


Fig. Snapshot Add the downloaded Ubuntu .iso file to boot (Ref: http://www.psychocats.net/ubuntu/virtualbox)

6) The iso image is downloaded in Section 2.2, Step 3 is loaded to the virtual CD/DVD for the Ubuntu boot and clicked OK.

7) Now Double Click on the virtual machine to start it up. Read the initial instructions and do as needed. Wait for Ubuntu to boot up. Once the boot process starts, install the Ubuntu as other OS installations.

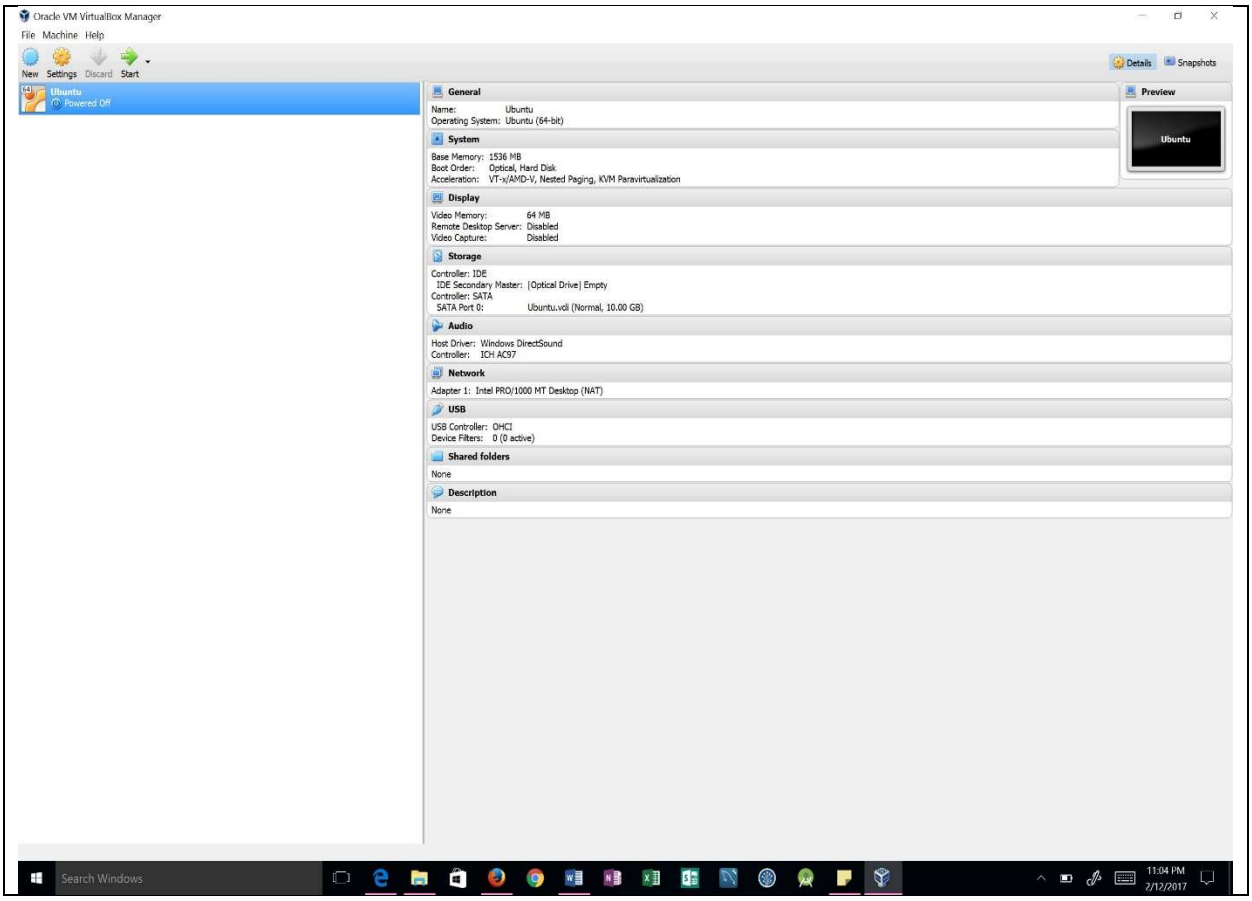


Fig. Snapshot Virtualbox startup with Ubuntu installed

8. Click on the Start to start Ubuntu installed.

9. Enter the valid password for loggin in. And the Ubuntu Desktop appeared.

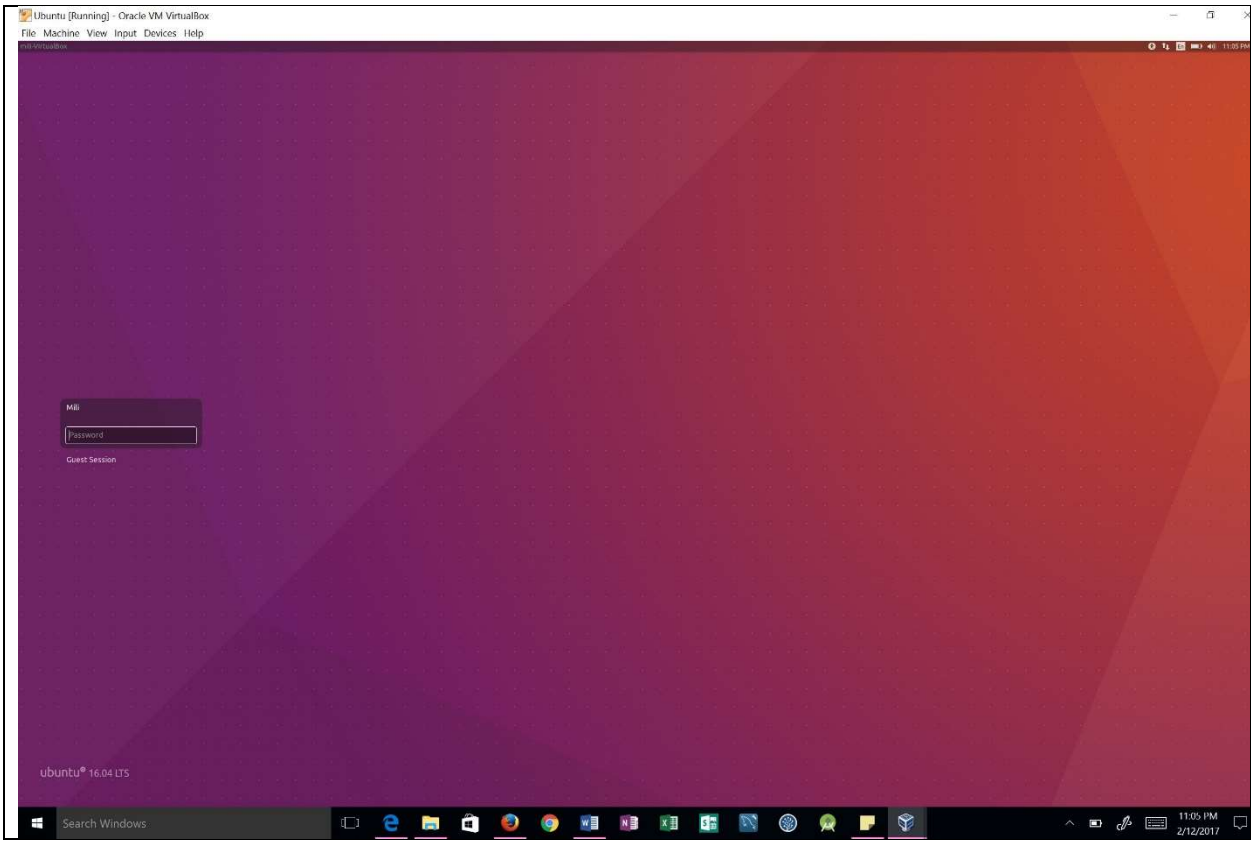


Fig. Snapshot Ubuntu startup inside Virtualbox

Section 3

We already have created the working environment for out Little OS project. Now we will be referring the Little OS Book for further developments. We begin creating a new folder named **"oslab"** in the desktop. This is where we keep all the files we create for the project.

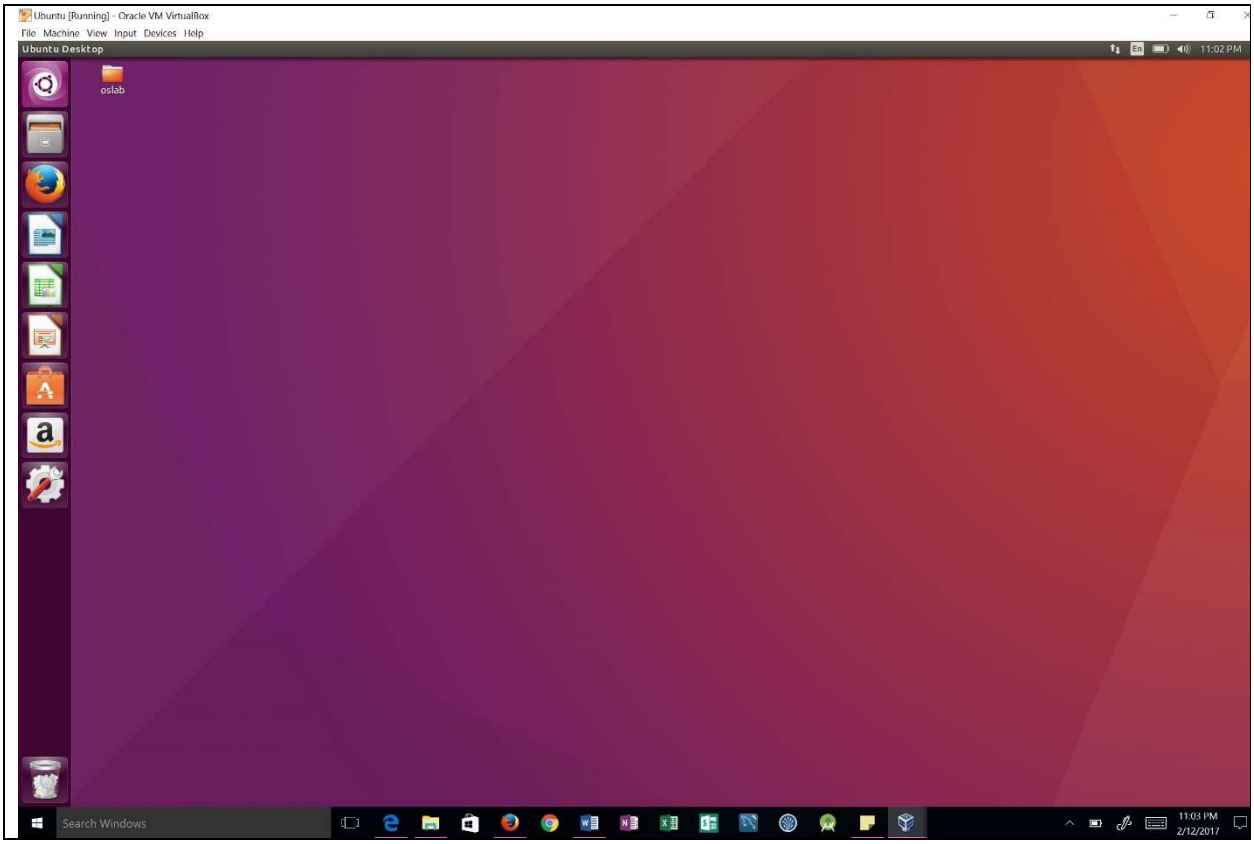


Fig. Snapshot Ubuntu after successful login, created oslab folder in the Ubuntu desktop

Section 3.1

Steps followed from the OS Book Section 1 and Section 2:

1. Open the Ubuntu terminal (Click Ubuntu icon in the upper-left and type terminal OR press Ctrl-Alt +T).
2. We will require an emulator to develop the OS and referring to the Section 2.1.1 of little OS book, we first install the emulator named Bochs using the below command.

   Bochs is a portable IA-32 and x86-64 IBM PC compatible emulator and debugger and distributed as free software under the GNU Lesser General Public License.

   Command in the terminal to install Bochs:

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

Section 3.2

Compiling the Operating System. Here we use some assembly code

1. Create a loader.s file with the following content given in the below fig. and save it to "oslab" folder previously created.

```
global loader                   ; the entry symbol for ELF

MAGIC_NUMBER equ 0x1BADB002     ; define the magic number constant
FLAGS        equ 0x0            ; multiboot flags
CHECKSUM     equ -MAGIC_NUMBER  ; calculate the checksum
                                ; (magic number + checksum + flags should equal 0)

section .text:                  ; start of the text (code) section
align 4                         ; the code must be 4 byte aligned
    dd MAGIC_NUMBER             ; write the magic number to the machine code,
    dd FLAGS                    ; the flags,
    dd CHECKSUM                 ; and the checksum

loader:                         ; the loader label (defined as entry point in linker script)
    mov eax, 0xCAFEBABE         ; place the number 0xCAFEBABE in the register eax
.loop:
    jmp .loop                   ; loop forever
```

Fig. loader.s file content

2. Create loader.s using gedit in the terminal and save it in the os lab folder.

>gedit

3. Following linux commands were helpful finding the root folder and reaching the "oslab" folder where we want to create, and update files and directories.

Find the root folder using pwd command

>pwd

/home/mili

>cd Desktop

Use ls command to list the files inside the folder

>ls

Oslab

>cd oslab

4. We used the below command to compile the loader.s file

>nasm -f elf32 loader.s

This created the loader.o object file in the oslab folder

Section 3.3

Linking the Kernel(Little Book Section 2.3.2), the code must now be linked to produce the executable file. We want the GRUB to load the kernel at the memory address larger than or equal to 0x00100000 (1MB), because addresses lower than 1MB are used by GRUB itself, BIOS and memory-mapped I/O. So, the linker script is required.

Steps required to create a linker

1. We created a file named link.ld using gedit and saved it in the "oslab" folder. And the contents of the link.ld file is in figure below.

```
ENTRY(loader)                /* the name of the entry label */

SECTIONS {
    . = 0x00100000;          /* the code should be loaded at 1 MB */

    .text ALIGN (0x1000) :   /* align at 4 KB */
    {
        *(.text)             /* all text sections from all files */
    }

    .rodata ALIGN (0x1000) : /* align at 4 KB */
    {
        *(.rodata*)          /* all read-only data sections from all files */
    }

    .data ALIGN (0x1000) :   /* align at 4 KB */
    {
        *(.data)             /* all data sections from all files */
    }

    .bss ALIGN (0x1000) :    /* align at 4 KB */
    {
        *(COMMON)            /* all COMMON sections from all files */
        *(.bss)              /* all bss sections from all files */
    }
}
```

Fig. Snapshot Contents of linker.ld

2. We linked the executable file using the following command:

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

   So, this created kernel.elf file in the oslab folder.

Section 3.4

Steps required for Obtaining GRUB (Little OS Book Section 2.3.3)

1. We downloaded the file named stage2_eltorito from the link given in the little book (http://littleosbook.github.com/files/stage2_eltorito) and saved in the same folder oslab that contains loader.s and link.ld

Section 3.5

Steps required for Building an ISO image (Little OS Book Section 2.3.4)

1. We created a Kernel ISO image using the program genisoimage. Folders named iso/boot/grub are created inside the "oslab" folder and the stage2_eltorito and kernel.elf are saved to their respective folders using the following commands

```
>mkdir -p iso/boot/grub              #create the folder structure

>cp stage2_eltorito iso/boot/grub/          #copy the bootloader

>cp kernel.elf iso/boot/             #copy the kernel
```

2. We created a file menu.lst using gedit and saved it to iso/boot/grub. Then we generated the ISO image using the following commands in the terminal.

```
>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. The ISO image os.iso now contains the kernel executable, the GRUB bootloader and the configuration file.

Section 3.6

Steps for Running Bochs (Little OS Book Section 2.3.5)

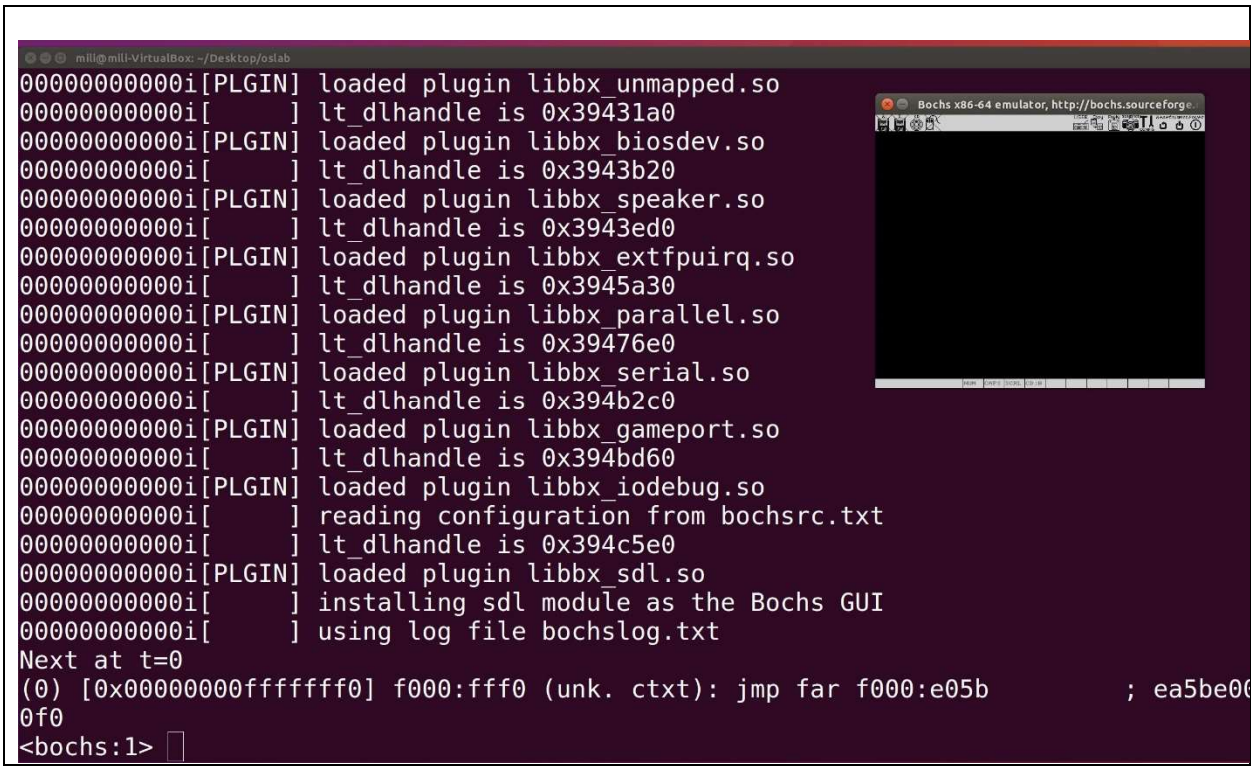
1. Bochs needs a configuration file to start. And we created a file named bochsrc.txt in the same folder "oslab" that contains "iso" folder. The contents of the bochsrc.txt is in the figure below.

```
megs:            32
display_library: sdl
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
```

Fig. Snapshot Contents of bochsrc.txt

2. Now in the command line type the command:

>bochs -f bochsrc.txt -q



We have the Bochs emulator window pop up. Now in the terminal we type c and type enter to continue to start the OS boot.

>c

3. The OS is created. The snapshot is below.

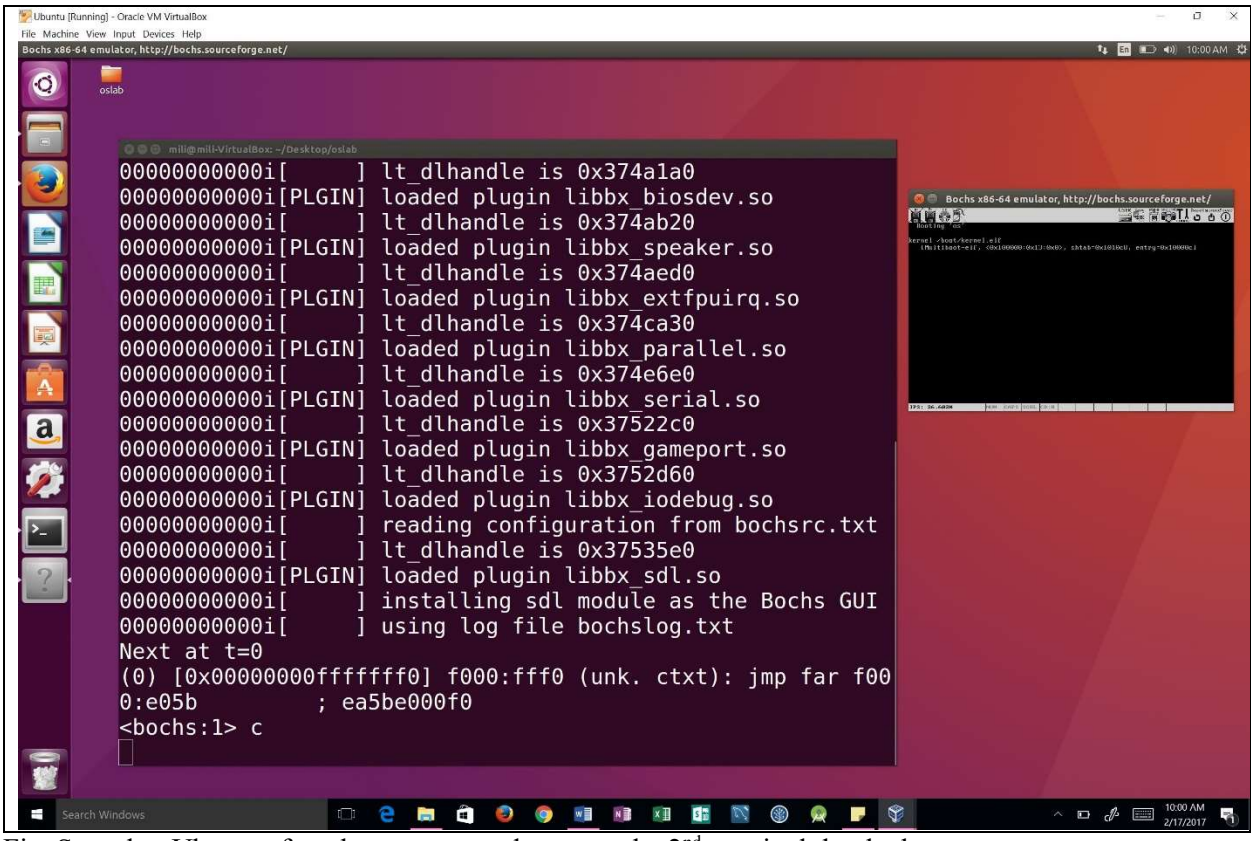


Fig. Snapshot Ubuntu after above command appears the 2nd terminal that loabs our new os.

3. To display the log produced by bochs we type this command to the terminal.

>cat bochslog.txt

4. For the OS started properly, we should see the contents of the registers of the CPU simulated by Bochs and find EAX = CAFEBABE (since our system is 64 bit). So we quit the Bochs and type the cat command of step 3 in the terminal and check for the cafebabe in the terminal.

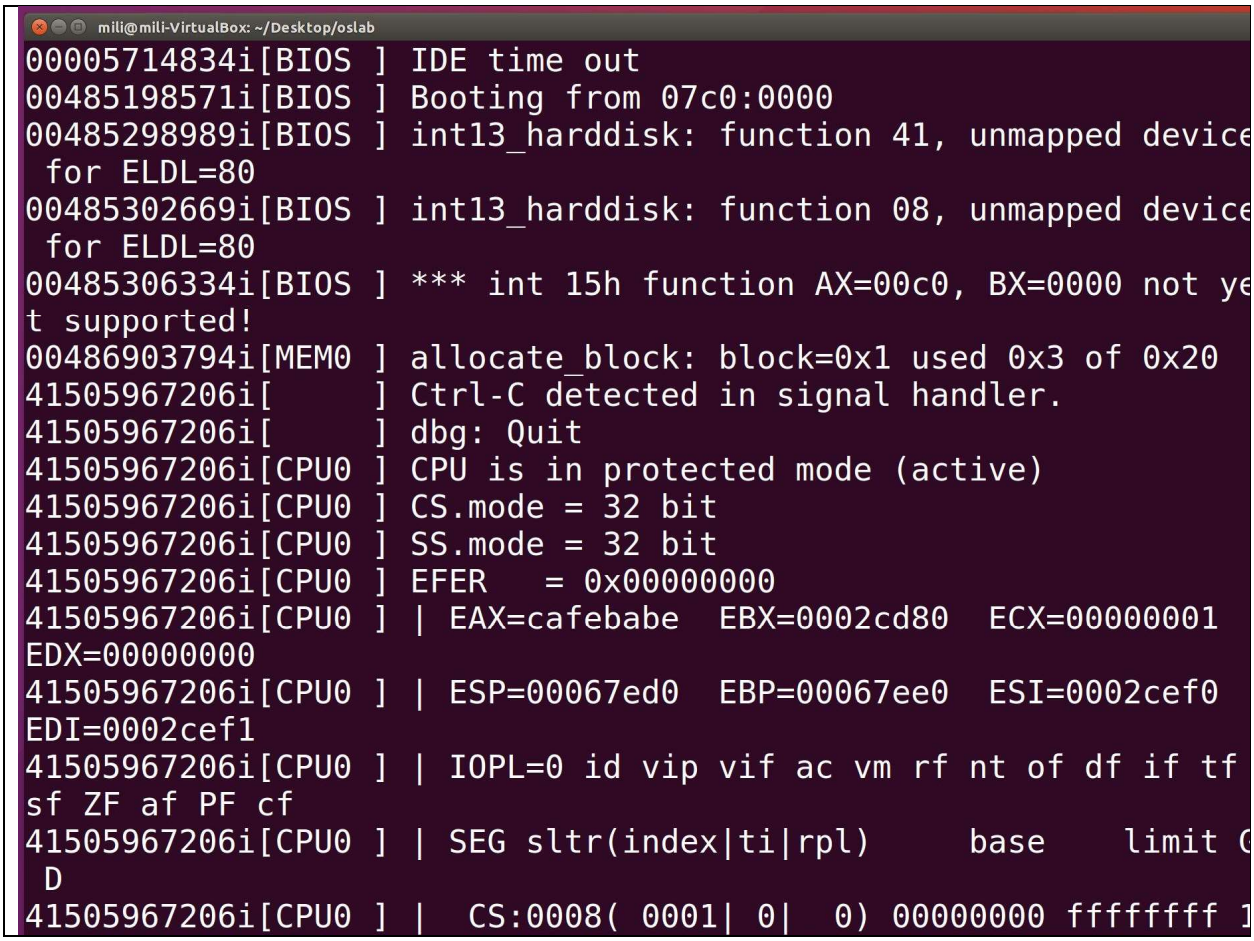


Fig. Snapshot of the bochlog.text file with the cafebabe highlighted.

5. We also found the cafebabe in our bochclog.text file. Hurray! os is running correctly.

References:

Section 2.3: Ubuntu 16.04 LTS release notes

Section 2.4 and few images: http://www.psychocats.net/ubuntu/virtualbox

Section 3.1: https://littleosbook.github.io

Lab 2 Session

In lab session 1, we were able to run the bochs emulator and made the basic setup of our new Operating System boot through.

In the lab session - 2, we will be setting up a stack for our new Operating System. We use C codes in most places and only a few assembly language code where necessary. We cover Section 3 and Section 4 in this lab. Section 4 will be completed on the Lab 3 session.

Section 3:

Setting up the stack

Open the loader.s and add up the following codes to set up the stack of 4 KiloBytes

```
KERNEL_STACK_SIZE equ 4096                  ; size of stack in bytes

section .bss
align 4                                     ; align at 4 bytes
kernel_stack:                               ; label points to beginning of memory
    resb KERNEL_STACK_SIZE                  ; reserve stack for the kernel
```

The Stack pointer is then set up by pointing esp to the end of the kernel stack memory.

```
mov esp, kernel_stack + KERNEL_STACK_SIZE   ; point esp to the start of the
                                            ; stack (end of memory area)
```

Section 3.2:

Calling C code from the assembly

We call C function from the assembly Code. We wrote the C function with three arguments in the kmain.c and we used the following code to write them.

```
/* The C function */
int sum_of_three(int arg1, int arg2, int arg3)
{
    return arg1 + arg2 + arg3;
}
```

Then we used the following assembly code to call the arguments in the kmain.c. We added and saved this code to the loader.s so that we can call the C function.

```
; The assembly code
external sum_of_three   ; the function sum_of_three is defined elsewhere

push dword 3            ; arg3
push dword 2            ; arg2
push dword 1            ; arg1
call sum_of_three       ; call the function, the result will be in eax
```

Issues:

We have to face some issues while compiling the code and we later understood from the instructor that it was because of indentation and formatting and the structure of the flow of instructions. We took help from the lab instructor in formatting the code and finally we were able to compile.

The final loader.s structured as below: We used the following order in the code, first section : .bss, second section: .text, third section: loader and fourth section: .loop

```
global loader                   ; the entry symbol for ELF

    MAGIC_NUMBER equ 0x1BADB002     ; define the magic number constant
    FLAGS        equ 0x0            ; multiboot flags
    CHECKSUM     equ -MAGIC_NUMBER  ; calculate the checksum
                                    ; (magic number + checksum + flags should equal 0)
    KERNEL_STACK_SIZE equ 4096                  ; size of stack in bytes

    section .bss
    align 4                                     ; align at 4 bytes
    kernel_stack:                               ; label points to beginning of memory
        resb KERNEL_STACK_SIZE                  ; reserve stack for the kernel



    section .text:                  ; start of the text (code) section
    align 4                         ; the code must be 4 byte aligned
        dd MAGIC_NUMBER             ; write the magic number to the machine code,
        dd FLAGS                    ; the flags,
        dd CHECKSUM                 ; and the checksum


    loader:                         ; the loader label (defined as entry point in linker script)
        mov eax, 0xCAFEBABE         ; place the number 0xCAFEBABE in the register eax


        mov esp, kernel_stack + KERNEL_STACK_SIZE   ; point esp to the start of the
                                                    ; stack (end of memory area)



    ; The assembly code
        extern sum_of_three     ; the function sum_of_three is defined elsewhere

        push dword 3            ; arg3
        push dword 2            ; arg2
        push dword 1            ; arg1
        call sum_of_three       ; call the function, the result will be in eax



    .loop:
        jmp .loop                   ; loop forever
```

After make we typed the make command at the terminal, output at the terminal is shown below:

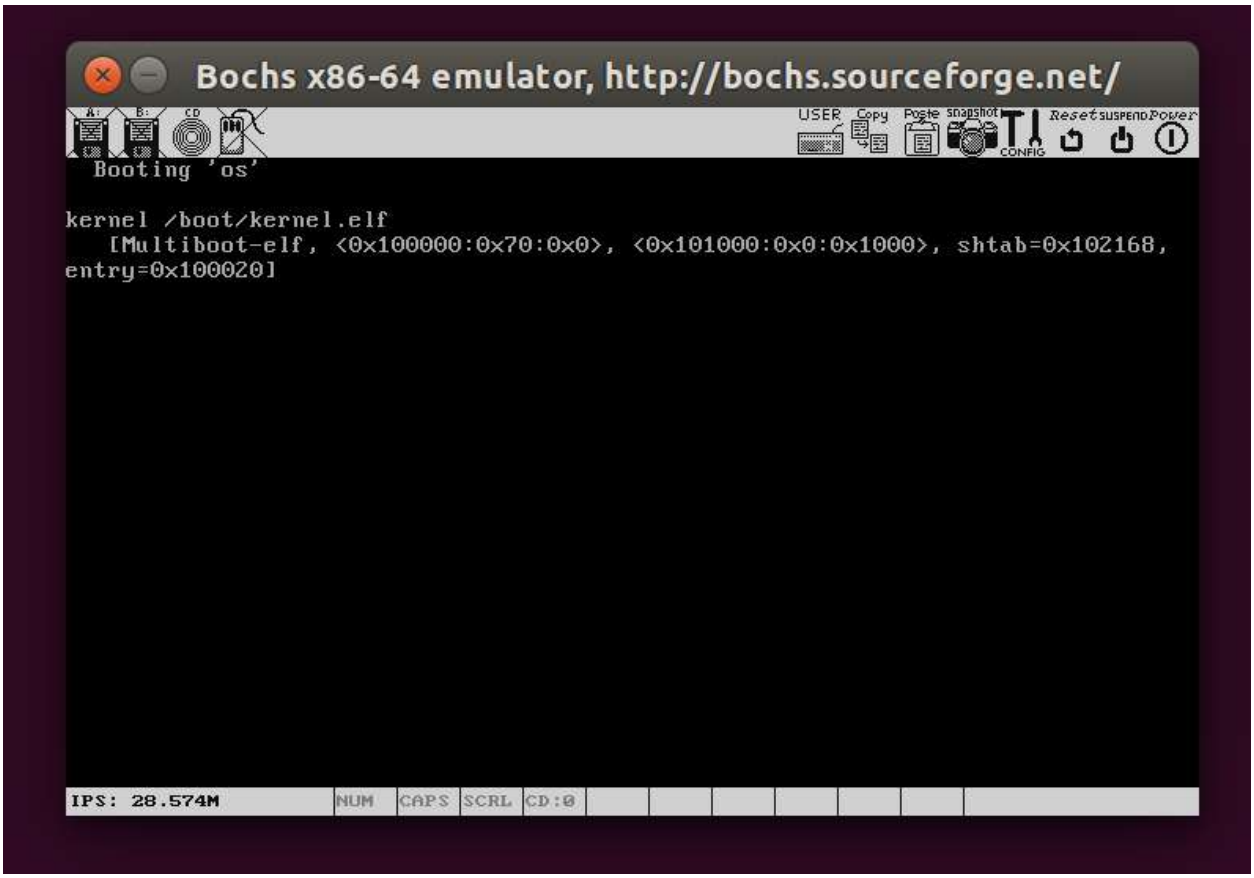
```
mili@mili-VirtualBox:~/Desktop/oslab$ make
nasm -f elf loader.s -o loader.o
gcc -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles -nodefaultli
bs -Wall -Wextra -Werror -c  kmain.c -o kmain.o
ld -T link.ld -melf_i386 loader.o kmain.o -o kernel.elf
```

Now we type the make run command at the terminal. And the result is shown below:

```
mili@mili-VirtualBox:~/Desktop/oslab$ make run
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
bochs -f bochsrc.txt -q
========================================================================
                      Bochs x86 Emulator 2.6
            Built from SVN snapshot on September 2nd, 2012
========================================================================
00000000000i[     ] LTDL_LIBRARY_PATH not set. using compile time default '/usr/lib/bochs
/plugins'
00000000000i[     ] BXSHARE not set. using compile time default '/usr/share/bochs'
00000000000i[     ] lt_dlhandle is 0x45c6560
00000000000i[PLGIN] loaded plugin libbx_unmapped.so
00000000000i[     ] lt_dlhandle is 0x45c71c0
00000000000i[PLGIN] loaded plugin libbx_biosdev.so
00000000000i[     ] lt_dlhandle is 0x45c7b40
00000000000i[PLGIN] loaded plugin libbx_speaker.so
00000000000i[     ] lt_dlhandle is 0x45c7ef0
00000000000i[PLGIN] loaded plugin libbx_extfpuirq.so
```

```
00000000000i[     ] lt_dlhandle is 0x45c9a50
00000000000i[PLGIN] loaded plugin libbx_parallel.so
00000000000i[     ] lt_dlhandle is 0x45cb700
00000000000i[PLGIN] loaded plugin libbx_serial.so
00000000000i[     ] lt_dlhandle is 0x45cf2e0
00000000000i[PLGIN] loaded plugin libbx_gameport.so
00000000000i[     ] lt_dlhandle is 0x45cfd80
00000000000i[PLGIN] loaded plugin libbx_iodebug.so
00000000000i[     ] reading configuration from bochsrc.txt
00000000000i[     ] lt_dlhandle is 0x45d0600
00000000000i[PLGIN] loaded plugin libbx_sdl.so
00000000000i[     ] installing sdl module as the Bochs GUI
00000000000i[     ] using log file bochslog.txt
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> c
```

And we type c in the Bochs terminal and press enter to continue the booting process. The booting of the OS is show in the image below:


Bochs x86-64 emulator, http://bochs.sourceforge.net/
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
suspend
Power
Booting 'os'
kernel /boot/kernel.elf
[Multiboot-elf, <0x100000:0x70:0x0>, <0x101000:0x0:0x1000>, shtab=0x102168,
entry=0x100020]
IPS: 28.574M
NUM
CAPS
SCRL
CD:0

Alternate Lab

Thread Sample Program:

Normally when a program starts up and becomes a process, it starts with a default thread. So we can say that every process has at least one thread of control. A process can create an extra thread using the following function:

#include <pthread.h>

int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void), coid *restrict arg)

1. The first argument is a pthread_t type address. The variable whose address is passed as first argument will hold the thread ID of the newly created thread.
2. The second argument may contain certain attributes which we want the new thread to contain. But in out sample program we replace it by NULL.
3. The third argument is a function pointer. Each thread starts with a function and that function address is passed here as third argument so that the kernel knows which function to start the thread from.
4. As the function(whose address is passed in the third argument above) may accept some arguments so we can pass these arguments in the form of pointer to a void type. This is because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

The following thread program were created to see the thread executions:

```c
/* Create and Terminate Pthread */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  5

 /*Define a pthread function named HelloWorldPrint*/

void *HelloWorldPrint(void *threadid)
{
        long tid;       //thread id
        tid = (long) threadid;
        printf(" #%ld! Hello World!  \n", tid);
        pthread_exit(NULL);
}

 /*Main Function */

int main(int argc, char *argv[])
{
        pthread_t threads[NUM_THREADS];
        int rc;
        long thr;

        for(thr = 0 ; thr< NUM_THREADS; thr++){

                printf("Create Thread  %ld\n", thr);
                rc = pthread_create(&threads[thr], NULL, HelloWorldPrint, (void *)thr);
                if(rc){
                        printf("Error: Return Code from pthread_create() is %d\n", rc);
                        exit(-1);
                }
        }
        /* Exit pthread*/
        pthread_exit(NULL);
}
```

We save it to a file named pthread.c and use the following command in the terminal to compile the pthread program.

>gcc -pthread pthread.c

We see the output file a.out is created after the compilation. Now, we are able to execute the code using

>./a.out

And the result is:

```
mili@mili-VirtualBox:~/Desktop/ThreadEx$ ls
pthread.c
mili@mili-VirtualBox:~/Desktop/ThreadEx$ gcc -pthread pthread.c
mili@mili-VirtualBox:~/Desktop/ThreadEx$ ls
a.out  pthread.c
mili@mili-VirtualBox:~/Desktop/ThreadEx$ ./a.out
Create Thread  0
Create Thread  1
Create Thread  2
Create Thread  3
Create Thread  4
 #4! Hello World!
 #3! Hello World!
 #2! Hello World!
 #1! Hello World!
 #0! Hello World!
mili@mili-VirtualBox:~/Desktop/ThreadEx$
```

As seen in the output, first thread is created then the second thread is created and then the remaining upto five threads i.e, threads [0 to 4] as declared by the global variable "NUM_THREADS" which is defined to be 5. One point to be noted is that the order of execution of threads is not always fixed. It depends on the OS scheduling algorithm.

During the time of execution, the program executes the most recent thread first, which is thread 4 in this program.

****

Now Back to the section 4 of the Little Books:

Section 4: Output

We are adding the following code to kmain.c in order to call assembly functions.

```
/** fb_write_cell:
 *  Writes a character with the given foreground and background to position i
 *  in the framebuffer.
 *
 *  @param i  The location in the framebuffer
 *  @param c  The character
 *  @param fg The foreground color
 *  @param bg The background color
 */
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)
}
```

We created a file io.s and we wrote the following code in that file.

```
global outb             ; make the label outb visible outside this file

; outb - send a byte to an I/O port
; stack: [esp + 8] the data byte
;        [esp + 4] the I/O port
;        [esp    ] return address
outb:
    mov al, [esp + 8]    ; move the data to be sent into the al register
    mov dx, [esp + 4]    ; move the address of the I/O port into the dx register
    out dx, al           ; send the data to the I/O port
    ret                  ; return to the calling function
```

After creating io.s we also created io.h header file with the following code

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

/** outb:
 *  Sends the given data to the given I/O port. Defined in io.s
 *
 *  @param port The I/O port to send the data to
 *  @param data The data to send to the I/O port
 */
void outb(unsigned short port, unsigned char data);

#endif /* INCLUDE_IO_H */
```

After creating io.s and io.h we made some changes to the loader.s file where we called the kmain function using the following code.

```
; The assembly code
    extern kmain    ; the function sum_of_three is defined elsewhere

    ;push dword 3           ; arg3
    ;push dword 2           ; arg2
    ;push dword 1           ; arg1
    call kmain      ; call the function, the result will be in eax
```

After changing loader.s we added a call function for write_cell in kmain.c using the following code

```
void kmain(){
fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
}
```

The final content in the kmain.c is below:

```
/*The C function*/
/*int sum_of_three(int arg1, int arg2, int arg3)
{
return arg1 + arg2 +arg3;
}

*/
#include "io.h"

#define FB_GREEN     2
#define FB_DARK_GREY 8
/* The I/O ports */
#define FB_COMMAND_PORT         0x3D4
#define FB_DATA_PORT            0x3D5
/* The I/O port commands */
#define FB_HIGH_BYTE_COMMAND    14
#define FB_LOW_BYTE_COMMAND     15

 /** fb_write_cell:
     *  Writes a character with the given foreground and background to position i
     *  in the framebuffer.
     *
     *  @param i  The location in the framebuffer
     *  @param c  The character
     *  @param fg The foreground color
     *  @param bg The background color
     */
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)
}



void kmain(){
fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
}



   | /** fb_move_cursor:
     *  Moves the cursor of the framebuffer to the given position
     *
     *  @param pos The new position of the cursor
     */
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);
}
```

The final content of the file loader.s file at this stage:

```
global loader                   ; the entry symbol for ELF

    MAGIC_NUMBER equ 0x1BADB002     ; define the magic number constant
    FLAGS        equ 0x0            ; multiboot flags
    CHECKSUM     equ -MAGIC_NUMBER  ; calculate the checksum
                                    ; (magic number + checksum + flags should equal 0)
    KERNEL_STACK_SIZE equ 4096                  ; size of stack in bytes

    section .bss
    align 4                                     ; align at 4 bytes
    kernel_stack:                               ; label points to beginning of memory
        resb KERNEL_STACK_SIZE                  ; reserve stack for the kernel



    section .text:                  ; start of the text (code) section
    align 4                         ; the code must be 4 byte aligned
        dd MAGIC_NUMBER             ; write the magic number to the machine code,
        dd FLAGS                    ; the flags,
        dd CHECKSUM                 ; and the checksum


    loader:                         ; the loader label (defined as entry point in linker script)
        mov eax, 0xCAFEBABE         ; place the number 0xCAFEBABE in the register eax


        mov esp, kernel_stack + KERNEL_STACK_SIZE   ; point esp to the start of the
                                                    ; stack (end of memory area)



    ; The assembly code
        extern kmain   ; the function sum_of_three is defined elsewhere

        ;push dword 3            ; arg3
        ;push dword 2            ; arg2
        ;push dword 1            ; arg1
        call kmain       ; call the function, the result will be in eax



    .loop:
        jmp .loop                   ; loop forever
```

The content of the file io file is below:

```
global outb     ; make the label outb visible outside this file

; outb - send a byte to an I/O port
; stack: [esp + 8] the data byte
;        [esp + 4] the I/O port
;        [esp    ] return address
outb:
        mov al, [esp + 8]       ; move the data to be sent into the al register
        mov dx, [esp + 4]       ; move the address of the I/O port into the dx register
        out dx, al              ; send the data to the I/O port
        ret                     ; return to the calling function
```

The content of the file io.h is below:

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

   /** outb:
     *  Sends the given data to the given I/O port. Defined in io.s
     *
     *  @param port The I/O port to send the data to
     *  @param data The data to send to the I/O port
     */
void outb(unsigned short port, unsigned char data);
#endif /* INCLUDE_IO_H */
```

Lab 3 and remaining Sessions

In the lab session 1, we were able to run the bochs emulator and made the basic setup of our new Operating System boot through.

In the lab session - 2, we will be setting up a stack for our new Operating System. We use C codes in most places and only a few assembly language code where necessary. We cover Section 3 and Section 4 in this lab. Section 4 will be completed on the Lab 3 session.

Continuing where we left on Section 4.2.3 from Little's Book

Section 4.2.2 (We commented the function call here beacause this was also called in the file io.s)

Add the following code in the kmain.c

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

/** outb:
 *  Sends the given data to the given I/O port. Defined in io.s
 *
 *  @param port The I/O port to send the data to
 *  @param data The data to send to the I/O port
 */
void outb(unsigned short port, unsigned char data);

#endif /* INCLUDE_IO_H */
```

Include Section 4.2.2 also in the before lab – Not Necessary since we comment this section

Section 4.2.3

The Driver

The driver should provide an interface that the rest of the codes we write can be interacting with the framebuffer. We use the following code in our write function:

```
int write(char *buf, unsigned int len);
```

This write function should automatically advance the cursor after a character has been written and scroll the screen if necessary.

Section 4.3

The Serial Port

Configuring the Serial Port

The Serial Ports is an interface communicating between the hardware. The serial port is easy to configure and is mostly used for logging utility in Bochs.

Section 4.3.1

The first data to be sent to the serial port is the configuration data. In order to set the two hardware to communicate we have to consider following things:

-The speed used for sending data (bit or baud rate)

- Error check for data (parity bits, stop bits)

- Number of bits that represent a unit of data (data bits)

Section 4.3.2

Configuring the line

Add following code to kmain.c

```
#include "io.h" /* io.h is implement in the section "Moving the cursor" */

/* The I/O ports */

/* All the I/O ports are calculated relative to the data port. This is because
 * all serial ports (COM1, COM2, COM3, COM4) have their ports in the same
 * order, but they start at different values.
 */

#define SERIAL_COM1_BASE                0x3F8      /* COM1 base port */

#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)

/* The I/O port commands */

/* SERIAL_LINE_ENABLE_DLAB:
 * Tells the serial port to expect first the highest 8 bits on the data port,
 * then the lowest 8 bits will follow
 */
#define SERIAL_LINE_ENABLE_DLAB         0x80

/** serial_configure_baud_rate:
 *  Sets the speed of the data being sent. The default speed of a serial
 *  port is 115200 bits/s. The argument is a divisor of that number, hence
 *  the resulting speed becomes (115200 / divisor) bits/s.
 *
```

```
 *  @param com      The COM port to configure
 *  @param divisor  The divisor
 */
void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
{
    outb(SERIAL_LINE_COMMAND_PORT(com),
         SERIAL_LINE_ENABLE_DLAB);
    outb(SERIAL_DATA_PORT(com),
         (divisor >> 8) & 0x00FF);
    outb(SERIAL_DATA_PORT(com),
         divisor & 0x00FF);
}
```

The way that data should be sent must be configured. This is also done via the line command port by sending a byte. Little OS book has the layout of the 8 bit look.

We use standard value 0x03[31], which means a length of 8 bits, no parity bit, one stop bit and break control disabled. This is sent to the command line port using below commands which we further add in kmain.c

```
/** serial_configure_line:
 *  Configures the line of the given serial port. The port is set to have a
 *  data length of 8 bits, no parity bits, one stop bit and break control
 *  disabled.
 *
 *  @param com  The serial port to configure
 */
void serial_configure_line(unsigned short com)
{
    /* Bit:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
     * Content: | d | b | prty  | s | dl  |
     * Value:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
     */
    outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}
```

Section 4.3.3

Configuring the Buffers

The transmission of data via the serial port is placed in buffers, if the data baud rate is more than the capacity of the wire the data will be buffered and is there is huge data transfer the buffer will be full and data will be lost. We know Buffer are FIFO queues.

We use the value 0xC7 = 11000111 that enables FIFO, clear both Rx and Tx and use 14 bytes as size of queue.

Section 4.3.4

Configuring the Modem

When configuring the serial port we want RTS and DTR to be 1, so that the data can be sent. Since we don't need to enable interrupts, as we don't do any receiving of the data. So, the configuration value 0x03 =00000011 (as RTS = 1, DTS = 1).

Section 4.3.5

Writing Data to the Serial Port

Writing data to the serial port is done using I/O port. The following code is added to the kmain.c.

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

We than just moved to see how the output looks till now and we observed as below:

Since during the compilation of io.s creates the io.o file, we have to mention the io.o file in the makefile.

Now make changes to the make file by adding io.o in the make file.

```
OBJECTS = loader.o kmain.o io.o
CC = gcc
CFLAGS = -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector \
         -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c
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
```

Rearrange the codes in the kmain.c file as below:

```
/*The C function*/
/*int sum_of_three(int arg1, int arg2, int arg3)
{
return arg1 + arg2 +arg3;
}

*/
#include "io.h"

//#ifndef INCLUDE_IO_H
//#define INCLUDE_IO_H

#define FB_GREEN     2
#define FB_DARK_GREY 8

/* Section 4.2.2 The I/O ports */
#define FB_COMMAND_PORT         0x3D4
#define FB_DATA_PORT            0x3D5

/* Section 4.2.2 The I/O port commands */
#define FB_HIGH_BYTE_COMMAND    14
#define FB_LOW_BYTE_COMMAND     15


/* Section 4.3 The I/O ports */

    /* All the I/O ports are calculated relative to the data port. This is because
     * all serial ports (COM1, COM2, COM3, COM4) have their ports in the same

     * order, but they start at different values.
     */

#define SERIAL_COM1_BASE                0x3F8      /* COM1 base port */

#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)

    /* The I/O port commands */

    /* SERIAL_LINE_ENABLE_DLAB:
     * Tells the serial port to expect first the highest 8 bits on the data port,
     * then the lowest 8 bits will follow
     */
#define SERIAL_LINE_ENABLE_DLAB         0x80


 /** fb_write_cell:
     *  Writes a character with the given foreground and background to position i
     *  in the framebuffer.
     *
     *  @param i  The location in the framebuffer
     *  @param c  The character
     *  @param fg The foreground color
```

```
     *  @param bg The background color
     */

char *fb = (char *) 0x000B8000;
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);
}

/** Section 4.2.2 outb:
  *  Sends the given data to the given I/O port. Defined in io.s
  *
  *  @param port The I/O port to send the data to
  *  @param data The data to send to the I/O port
  */
//void outb(unsigned short port, unsigned char data);

//      #endif /* INCLUDE_IO_H */

/** Section 4.2.2 fb_move_cursor:
  *  Moves the cursor of the framebuffer to the given position
  *
  *  @param pos The new position of the cursor
  */
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);
}

/** Section 4.3 serial_configure_baud_rate:
  *  Sets the speed of the data being sent. The default speed of a serial
     *  port is 115200 bits/s. The argument is a divisor of that number, hence
     *  the resulting speed becomes (115200 / divisor) bits/s.
     *
     *  @param com      The COM port to configure
     *  @param divisor  The divisor
*/
void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
{
        outb(SERIAL_LINE_COMMAND_PORT(com),
             SERIAL_LINE_ENABLE_DLAB);
        outb(SERIAL_DATA_PORT(com),
             (divisor >> 8) & 0x00FF);
        outb(SERIAL_DATA_PORT(com),
             divisor & 0x00FF);
}
```

```
/** Section 4.3 serial_configure_line:
     *  Configures the line of the given serial port. The port is set to have a
     *  data length of 8 bits, no parity bits, one stop bit and break control
     *  disabled.
     *
     *  @param com  The serial port to configure
*/
void serial_configure_line(unsigned short com)
{
/* Bit:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
 * Content: | d | b | prty  | s | dl  |
 * Value:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
 */
        outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
}
//Section 4.3
int serialwrite(char *buf, unsigned int len);

void kmain(){
        fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
}
```

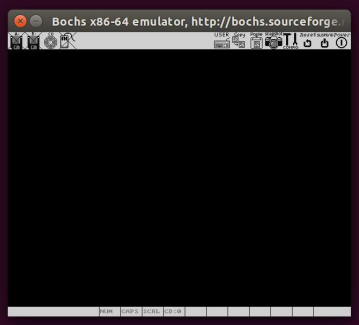
Now, use the terminal as below:

>make

```
mili@mili-VirtualBox:~/Desktop/oslab$ make
gcc -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector -nostartfiles -no
defaultlibs -Wall -Wextra -Werror -c  kmain.c -o kmain.o
ld -T link.ld -melf_i386 loader.o kmain.o io.o -o kernel.elf
mili@mili-VirtualBox:~/Desktop/oslab$
```

>make run

```
000000000000i[PLGIN] loaded plugin libbx_unmapped.so
000000000000i[     ] lt_dlhandle is 0x47211c0
000000000000i[PLGIN] loaded plugin libbx_biosdev.so
000000000000i[     ] lt_dlhandle is 0x4721b40
000000000000i[PLGIN] loaded plugin libbx_speaker.so
000000000000i[     ] lt_dlhandle is 0x4721ef0
000000000000i[PLGIN] loaded plugin libbx_extfpuirq.so
000000000000i[     ] lt_dlhandle is 0x4723a50
000000000000i[PLGIN] loaded plugin libbx_parallel.so
000000000000i[     ] lt_dlhandle is 0x4725700
000000000000i[PLGIN] loaded plugin libbx_serial.so
000000000000i[     ] lt_dlhandle is 0x47292e0
000000000000i[PLGIN] loaded plugin libbx_gameport.so
000000000000i[     ] lt_dlhandle is 0x4729d80
000000000000i[PLGIN] loaded plugin libbx_iodebug.so
000000000000i[     ] reading configuration from bochsrc.txt
000000000000i[     ] lt_dlhandle is 0x472a600
000000000000i[PLGIN] loaded plugin libbx_sdl.so
000000000000i[     ] installing sdl module as the Bochs GUI
000000000000i[     ] using log file bochslog.txt
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5b
e000f0
<bochs:1>
```



>c

```
000000000000i[     ] lt_dlhandle is 0x47211c0
000000000000i[PLGIN] loaded plugin libbx_biosdev.so
000000000000i[     ] lt_dlhandle is 0x4721b40
000000000000i[PLGIN] loaded plugin libbx_speaker.so
000000000000i[     ] lt_dlhandle is 0x4721ef0
000000000000i[PLGIN] loaded plugin libbx_extfpuirq.so
000000000000i[     ] lt_dlhandle is 0x4723a50
000000000000i[PLGIN] loaded plugin libbx_parallel
000000000000i[     ] lt_dlhandle is 0x4725700
000000000000i[PLGIN] loaded plugin libbx_serial.s
000000000000i[     ] lt_dlhandle is 0x47292e0
000000000000i[PLGIN] loaded plugin libbx_gameport
000000000000i[     ] lt_dlhandle is 0x4729d80
000000000000i[PLGIN] loaded plugin libbx_iodebug.
000000000000i[     ] reading configuration from b
000000000000i[     ] lt_dlhandle is 0x472a600
000000000000i[PLGIN] loaded plugin libbx_sdl.so
000000000000i[     ] installing sdl module as the Bochs GUI
000000000000i[     ] using log file bochslog.txt
Next at t=0
(0) [0x00000000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5b
e000f0
<bochs:1> c
```

Bochs Terminal with the green background with grey letter A seen.

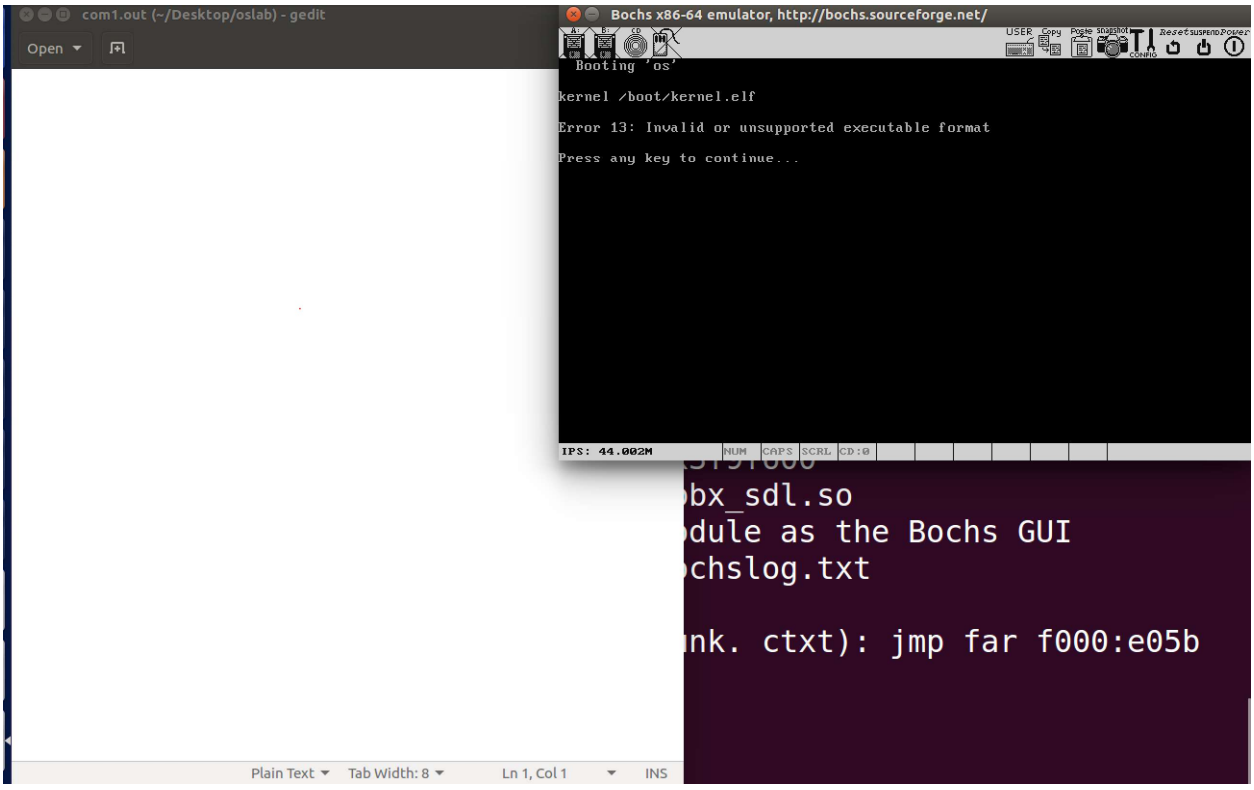
Section 4.3.5

Some changes done in the kmain.c file

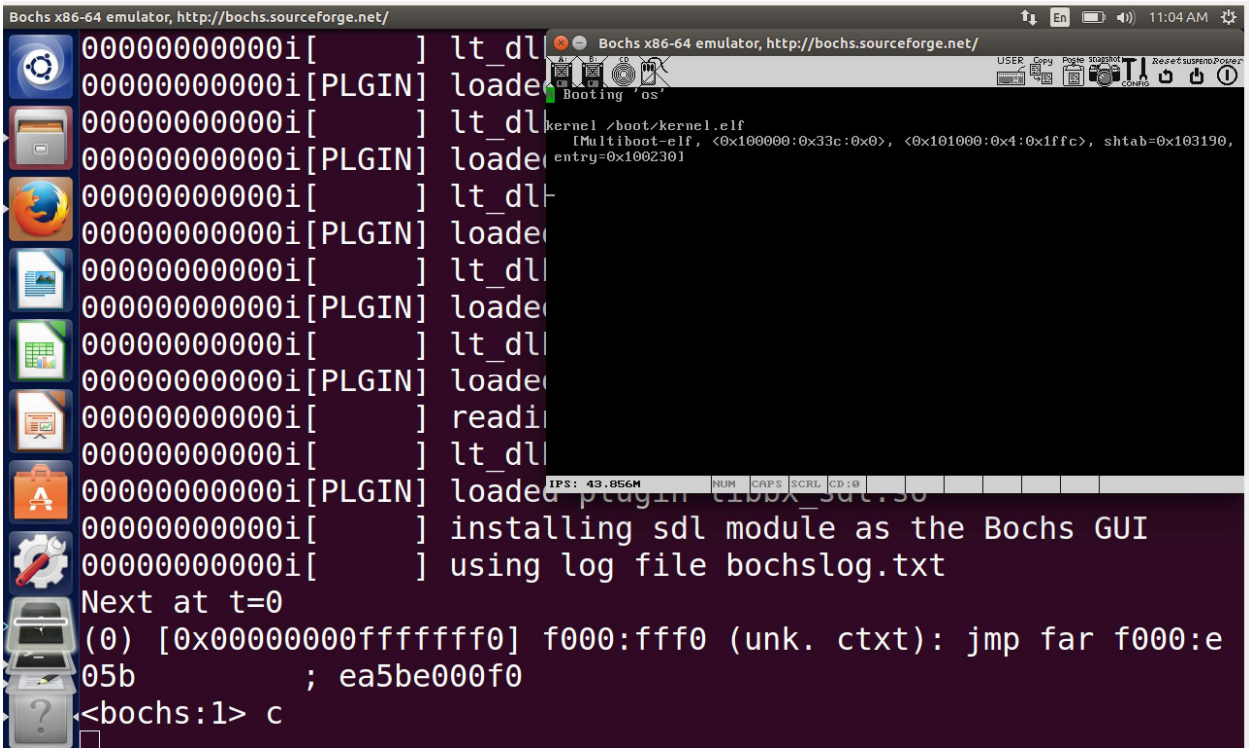
>make

>make run

After including section 4.3.5 and making many changes in kmain.c we compiled it in the terminal and we encountered some simple syntactical errors. The images are listed below:



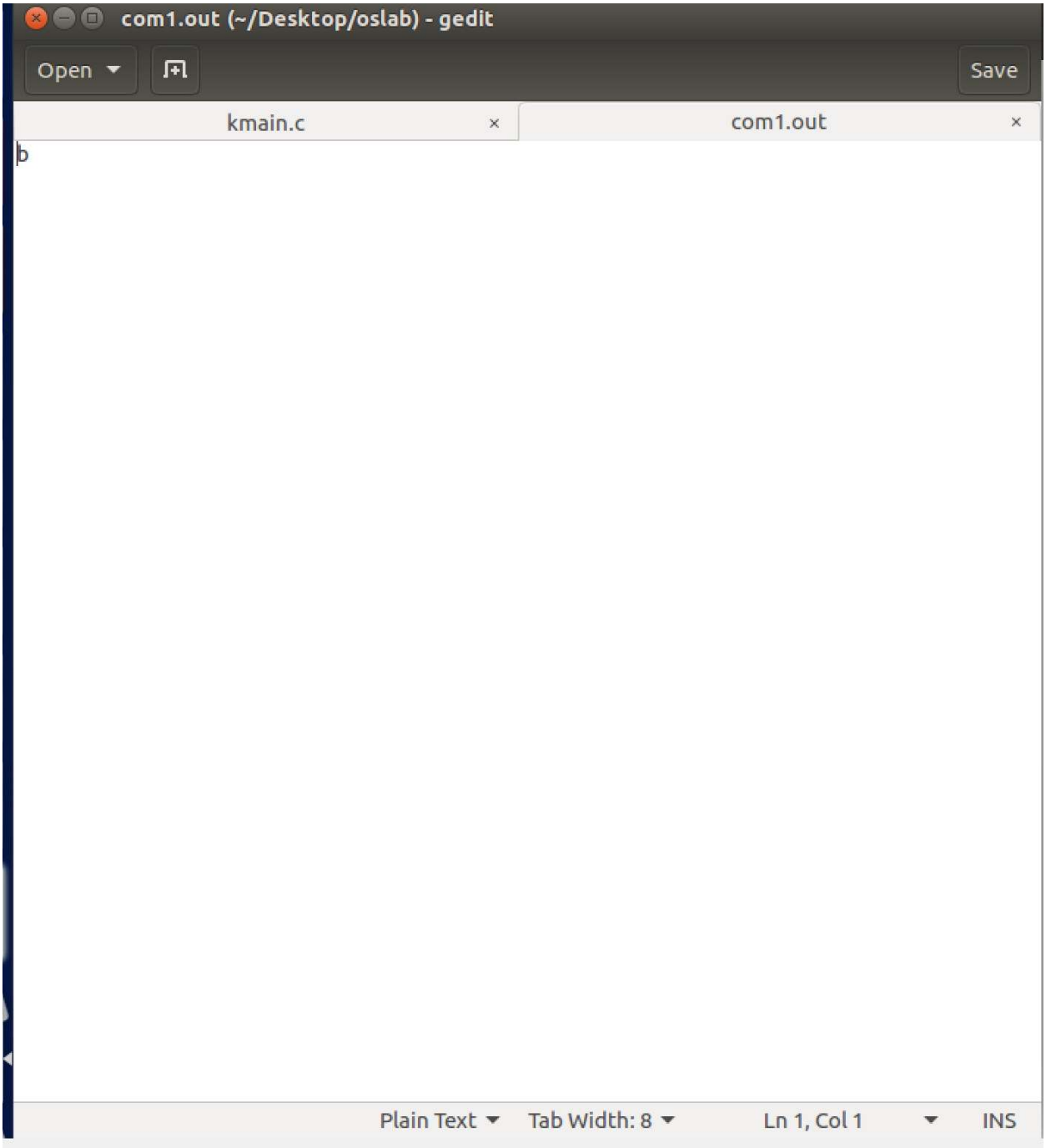
After rectifying the pointer error in the call of serial write function, we got the desired output.



The modified kmain.c figure is as follows.

```
        return i;
}

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

//Section 4.3.5 Writing to the serial port

/** serial_is_transmit_fifo_empty:
     *  Checks whether the transmit FIFO queue is empty or not for the given
COM
     *  port.
     *
     *  @param  com The COM port
     *  @return 0 if the transmit FIFO queue is not empty
     *          1 if the transmit FIFO queue is empty
*/
int serial_is_transmit_fifo_empty(unsigned int com)
{
        /* 0x20 = 0010 0000 */
return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}



void kmain(){
        fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);

        //int serial_is_transmit_fifo_empty(SERIAL_COM1_BASE);

        serial_configure_baud_rate(SERIAL_COM1_BASE, 2);
        serial_configure_line(SERIAL_COM1_BASE);
        char buf[]="b";
        serialwrite(buf, 1);
}
```

After we changed the pointer error by including char buf[]=”b” on our code we got the desired result from the com1.out. The com1.out file printed b which was our output. The image of com1.out is as follows.



Section 5: Segmentation

Segmentation in x86 means accessing the memory through segments. Segments are the portions of the address space, possibly overlapping, specified by a base address and a limit. To address a byte in the segmented memory we use a 48- bit logical address: 16 bits for the segment and 32- bits that specifies what offset within that segment we want.

Segmentation is required only when the memory size defined for the OS is more. Memory segmentation is the division of a computer's primary memory into segments or sections. We prefer not to use the segmentation here and use a flat- memory model.

Section 6: Interrupts and Inputs

An interrupt occurs when a hardware device, such as the keyboard, the serial port or the timer, signals the CPU that the state of the device has changed. The processor itself can also send interrupts due to program errors(example: when a program accesses memory it does not have access to, or when a program divides a number by zero). There are also some software interrupts, which are caused by the int assembly code instruction, and they are often used for system calls.

Section 6.1 Interrupt Handlers

Interrupts are handled via the Interrupt Descriptor Table(IDT). The IDT describes a handler for each interrupt. The interrupts are numbered (0 - 255) and the handler for interrupt i is defined at the ith position in the table. There are three different kinds of handlers for interrupts:

(1) Task Handler (Uses the functionality specific to the Intel version of x86, they won't be covered here)
(2) Interrupt Handler (This disables interrupts)
(3) Trap Handler ()

Section 6.2 Creating an Entry in the IDT

Entry in the IDT for an interrupt handler consists of 64 bits.

Name	Description
offset high	The 16 highest bits of the 32 bit address in the segment.
offset low	The 16 lowest bits of the 32 bits address in the segment.
p	If the handler is present in memory or not (1 = present, 0 = not present).
DPL	Descriptor Privilige 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.
r	Reserved.

The offset is a pointer to the code.

Section 6.3 Handling an Interrupt

When an interrupt occurs the CPU will push some information about the interrupt onto the stack, then look up the appropriate interrupt hander in the IDT and jump to it.

```
[esp + 12] eflags
[esp + 8]  cs
[esp + 4]  eip
[esp]      error code?
```

We created the interrupt handler with the assembly code which was named as interrupt_asm.s and the interrupt.c and interrupt.h files for the interrupt_handler function definition and header files used.

```
#include "interrupt.h"
#include "pic.h"


static interrupt_handler interrupt_handlers[IDT_NUM_ENTRIES];

void interrupt_handler(struct cpu_state cpu, struct stack_state stack, unsigned int interrupt){

    if (interrupt_handlers[info.idt_index] != NULL) {
        interrupt_handlers[info.idt_index](state, info, exec);
    }
    else {

        log_info("interrupt_handler",
                 "unhandled interrupt: %u, eip: %X, cs: %X, eflags: %X\n",
                 info.idt_index, exec.eip, exec.cs, exec.eflags);
        }

}
```

The header file for interrupt is interrupt.h

```
struct cpu_state {
    unsigned int eax;
    unsigned int ebx;
    unsigned int ecx;
    unsigned int esp;
} __attribute__((packed));

struct stack_state {
    unsigned int error_code;
    unsigned int eip;
    unsigned int cs;
    unsigned int eflags;
} __attribute__((packed));

void interrupt_handler(struct cpu_state cpu, struct stack_state stack, unsigned int interrupt);
```

Section 6.4 Creating a Generic Interrupt Handler

This section uses macros to push the interrupts on the stack. We created interrupt_asm.s and it is mentioned below.

```
%macro no_error_code_interrupt_handler 1
   global interrupt_handler_%1
   interrupt_handler_%1:
       push    dword 0                     ; push 0 as error code
       push    dword %1                    ; push the interrupt number
       jmp     common_interrupt_handler    ; jump to the common handler
   %endmacro

   %macro error_code_interrupt_handler 1
   global interrupt_handler_%1
   interrupt_handler_%1:
       push    dword %1                    ; push the interrupt number
       jmp     common_interrupt_handler    ; jump to the common handler
   %endmacro

   common_interrupt_handler:               ; the common parts of the generic interrupt handler
       ; save the registers
       push    eax
       push    ebx
       .
       .
       .
       push    ebp

       ; call the C function
       call    interrupt_handler

       ; restore the registers
       pop     ebp
       .
       .
       .
       pop     ebx
       pop     eax

       ; restore the esp
       add     esp, 8

       ; return to the code that got interrupted
       iret


;some parts were removed as instructed by the lab instructor as we use only one hardware interrupt
no_error_code_interrupt_handler 1       ; create handler for interrupt 1
```

Section 6.5 Loading the IDT

The IDT is loaded with the lidt assembly code instruction which takes the address of the first element in the table. We are loading it in the pic.s which is given below:

```
global  load_idt

   ; load_idt - Loads the interrupt descriptor table (IDT).
   ; stack: [esp + 4] the address of the first entry in the IDT
   ;        [esp    ] the return address
   load_idt:
       mov     eax, [esp+4]    ; load the address of the IDT into register eax
       lidt    [eax]             ; load the IDT
       ret                     ; return to the calling function
```

Section 6.6 Programmable Interrupt Controller (PIC)

The PIC makes it possible to map signals from the hardware to interrupts.

The reasons for configuring the PIC are:

- Remap the interrupts. The PIC uses interrupts 0 - 15 for hardware interrupts by default, which conflicts with the CPU interrupts. Therefore, the PIC interrupts must be remapped to another interval.
- Select which interrupts to receive. You probably don't want to receive interrupts from all devices since you don't have code that handles these interrupts anyway.
- Set up the correct mode for the PIC.

The content of the pic.c file is given below:

```
/* The PIC interrupts have been remapped */
#define PIC1_START_INTERRUPT 0x20
#define PIC2_START_INTERRUPT 0x28
#define PIC2_END_INTERRUPT   PIC2_START_INTERRUPT + 7



//******

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

#define PIC_ACK     0x20

/* Information about how to program the PIC was found at
 * http://www.acm.uiuc.edu/sigops/roll_your_own/i386/irq.html
 */

#define PIC1_PORT_A 0x20
#define PIC1_PORT_B 0x21

#define PIC2_PORT_A 0xA0
#define PIC2_PORT_B 0xA1

#define PIC1_ICW1   0x11 /* Initialize the PIC and enable ICW4 */
#define PIC2_ICW1   PIC1_ICW2

#define PIC1_ICW2   0x20 /* IRQ 0-7 will be remapped to IDT index 32 - 39 */
#define PIC2_ICW2   0x28 /* IRQ 8-15 will be remapped to IDT index 40 - 47 */

#define PIC1_ICW3   0x04 /* PIC1 is connected to PIC2 via IRQ2 */
#define PIC2_ICW3   0x02 /* PIC2 is connected to PIC1 via IRQ1 */

#define PIC1_ICW4   0x05 /* 8086/88 mode is enabled and PIC1 is master */
#define PIC2_ICW4   0x01 /* 8086/88 mode is enabled */

#define PIC_EOI     0x20
```

```
void pic_init(void)
{
    /* ICW1 */
    outb(PIC1_PORT_A, PIC1_ICW1);
    outb(PIC2_PORT_A, PIC2_ICW1);

    /* ICW2 */
    outb(PIC1_PORT_B, PIC1_ICW2);
    outb(PIC2_PORT_B, PIC2_ICW2);

    /* ICW3 */
    outb(PIC1_PORT_B, PIC1_ICW3);
    outb(PIC2_PORT_B, PIC2_ICW3);

    /* ICW4 */
    outb(PIC1_PORT_B, PIC1_ICW4);
    outb(PIC2_PORT_B, PIC2_ICW4);

    //pic_mask(0xEC, 0xFF);
}

/** pic_acknowledge:
  *  Acknowledges an interrupt from either PIC 1 or PIC 2.
  *
     *  @param num The number of the interrupt
     */
    void pic_acknowledge(unsigned int interrupt)
    {
        if (interrupt < PIC1_START_INTERRUPT || interrupt > PIC2_END_INTERRUPT) {
          return;
        }

        if (interrupt < PIC2_START_INTERRUPT) {
          outb(PIC1_PORT_A, PIC_ACK);
        } else {
          outb(PIC2_PORT_A, PIC_ACK);
        }
    }
```

```
/*void pic_mask(uint8_t mask1, uint8_t mask2)
{
    outb(PIC1_PORT_B, mask1);
    outb(PIC2_PORT_B, mask2);
}

*/
```

Section 6.7 Reading Input from the Input

The keyboard does not generate ASCII characters, it generates scan codes. since the keyboard interrupt is raised by the PIC, we should call pic_acknowledge function. The results after compiling all files together is given below:

```
mili@mili-VirtualBox:~/Desktop/oslab$ make
gcc -m32 -nostdlib -nostdinc -fno-builtin -fno-stack-protector
 -nostartfiles -nodefaultlibs -Wall -Wextra -Werror -c  interr
upt.c -o interrupt.o
interrupt.c:5:26: error: expected '=', ',', ';', 'asm' or '__a
ttribute__' before 'interrupt_handlers'
 static interrupt_handler interrupt_handlers[IDT_NUM_ENTRIES];
                          ^
interrupt.c: In function 'interrupt_handler':
interrupt.c:9:9: error: 'interrupt_handlers' undeclared (first
 use in this function)
     if (interrupt_handlers[info.idt_index] != NULL) {
         ^
interrupt.c:9:9: note: each undeclared identifier is reported
only once for each function it appears in
interrupt.c:9:28: error: 'info' undeclared (first use in this
function)
     if (interrupt_handlers[info.idt_index] != NULL) {
                            ^
```

Issues: We are getting some errors in the interrupt handler. As we are running out of time, we have decided to fix these errors in the future.

Conclusion: It was a challenging lab for Building an OS. We were to complete the halfway through the Little OS Book due to the lesser amount of time. We plan to continue with the remaining sections soon.

References:

Section 4: https://littleosbook.github.io

Section 5: https://en.wikibooks.org/wiki/Operating_System_Design/Segmentation

Section 6: http://wiki.osdev.org/Interrupts

Section 6: https://littleosbook.github.io

Section 6: https://github.com/littleosbook/aenix/tree/master/src/kernel