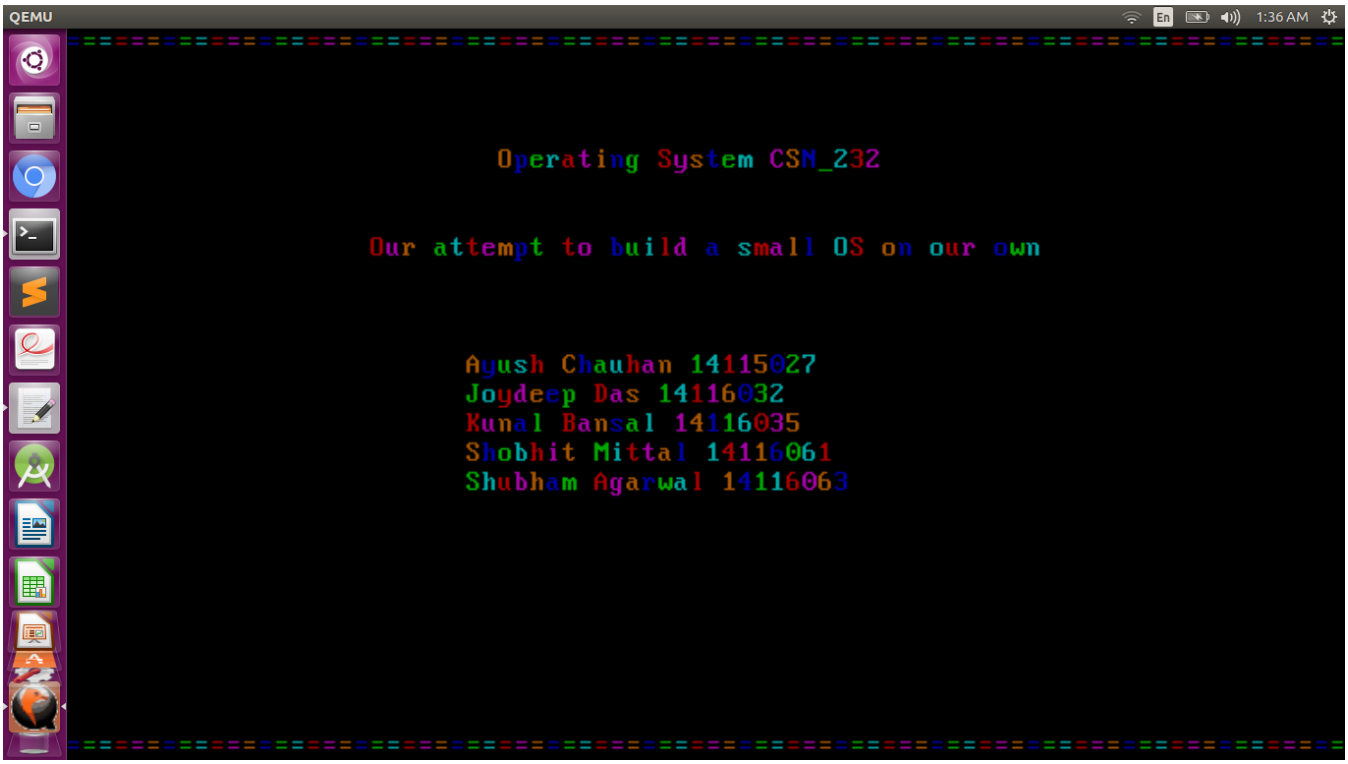
Operating Systems CSN – 232

Course Project – Building a Simple Operating System from Scratch

Report File



Submitted By:

Ayush Chauhan – 14115027

Joydeep Das – 14116032

Kunal Bansal – 14116035

Shobhit Mittal – 14116061

Shubham Agarwal - 14116063

Submitted To:

Dr. R. Balasubramanian

Course Instructor

Introduction

This project is a successful attempt to learn how one can go about building an entire Operating System from scratch. We have concentrated our work on the widely used x86 architecture CPU. The project does not aim to be extensive in terms of practical operating system functionality but instead to give us a hands-on experience of low-level programming, how operating systems are written, and the kind of problems they must solve. Along the way, we learnt quite a few things like how a computer boots, how to configure the CPU so that we can begin to use its extended functionality, how to bootstrap code written in a higher-level language, how to create some fundamental operating system services, etc. Now, we shall begin our discussion on the various stages in the process.

The Boot Process

When we reboot our computer, it must start up again, initially without any notion of an operating system. Somehow, it must load the operating system --- whatever variant that may be --- from some permanent storage device that is currently attached to the computer (e.g. a floppy disk, a hard disk, a USB dongle, etc.).

So, what we do have is the *Basic Input/Output Software (BIOS)*, a collection of software routines that are initially loaded from a chip into memory and initialised when the computer is switched on. BIOS provides auto-detection and basic control of your computer's essential devices, such as the screen, keyboard, and hard disks.

So, the easiest place for BIOS to find our OS is in the first sector of one of the disks (i.e. Cylinder 0, Head 0, Sector 0), known as the *boot sector*.

An unsophisticated means is adopted here by BIOS, whereby the last two bytes of an intended boot sector must be set to the magic number 0xaa55. So, BIOS loops through each storage device (e.g. floppy drive, hard disk, CD drive, etc.), reads the boot sector into memory, and instructs the CPU to begin executing the first boot sector it finds that ends with the magic number.
This is where we seize control of the computer.

```
e9 fd ff 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

Note that, in figure above, the three important features are:
• The initial three bytes, in hexadecimal as 0xe9, 0xfd and 0xff, are actually machine code instructions, as defined by the CPU manufacturer, to perform an endless jump.
• The last two bytes, 0x55 and 0xaa, make up the magic number, which tells BIOS that this is indeed a boot block and not just data that happens to be on a drive's boot sector.
• The file is padded with zeros ('*' indicates zeros omitted for brevity), basically to position the magic BIOS number at the end of the 512 byte disk sector.

CPU Emulation

Emulation involves a program that behaves like a specific CPU architecture, using variables to represent CPU registers and high-level control structures to simulate lower level jumps and so on, so is much slower but often better suited for development and debugging such systems. Note that, in order to do anything useful with an emulator, you need to give it some code to run in the form of a disk image file. An image file simply is the raw data (i.e. machine code and data) that would otherwise have been written to medium of a hard disk, a floppy disk, a CDROM, USB stick, etc.

The emulator used in this project is *QEmu*. It is similar to Bochs, though is much more efficient and capable also of emulating architectures other than x86. Though QEmu is less well documented than Bochs, a need for no configuration file means it is easier to get running, as follows:

```
$qemu <your-os-boot-disk-image-file>
```

The Real 16-bit processor

Boot Sector Programming

You'd have to remember, or continuously reference, which of many possible machine codes cause the CPU to do certain functions. Luckily, you are not alone, and so assemblers have been written that translate more human friendly instructions into machine code for a particular CPU.

Now, we will re-create the binary-edited boot sector using assembly language, so that we can really appreciate the value even of a very low-level language .We can assemble this into actual machine code (a sequence of bytes that our CPU can interpret as instructions) as follows:

```
$nasm boot_sect.asm -f bin -o boot_sect.bin
```

Here *boot_sect.asm* is the file into which we saved the source code in figure below and *boot_sect.bin* is the assembled machine code that we can install as a boot sector on a disk.
Note that we used the *-f bin* option to instruct *nasm* to produce raw machine code, rather than a code package that has additional meta-information for linking in other routines that we would expect to use when programming in a more typical operating system Environment.

```
;
; A simple boot sector program that loops forever.
;

loop:                   ; Define a label, "loop", that will allow
                        ; us to jump back to it, forever.

  jmp loop              ; Use a simple CPU instruction that jumps
                        ; to a new memory address to continue execution.
                        ; In our case, jump to the address of the current
                        ; instruction.

times 510-($-$$) db 0 ; When compiled, our program must fit into 512 bytes,
                        ; with the last two bytes being the magic number,
                        ; so here, tell our assembly compiler to pad out our
                        ; program with enough zero bytes (db 0) to bring us to the
                        ; 510th byte.

dw 0xaa55               ; Last two bytes (one word) form the magic number,
                        ; so BIOS knows we are a boot sector.
```

The figure above shows a simple boot sector written in assembly language. We can easily test this program by:

```
$qemu boot_sect.bin
```

Why 16-bit processing?

CPU manufacturers must go to great lengths to keep their CPUs (i.e. their specific instruction set) compatible with earlier CPUs, so that older software, and in particular older operating systems, can still run on the most modern CPUs. The solution implemented by Intel and compatible CPUs is to emulate the oldest CPU in the family: the Intel 8086, which had support for 16-bit instructions.

So, for backward compatibility, it is important that CPUs boot initially in 16-bit real mode, requiring modern operating systems explicitly to switch up into the more advanced 32-bit (or 64-bit) protected mode, but allowing older operating systems to carry on, blissfully unaware that they are running on a modern CPU.

Generally, when we say that a CPU is 16-bit, we mean that its instructions can work with a maximum of 16-bits at once, for example: a 16-bit CPU will have a particular instruction that can add two 16-bit numbers together in one CPU cycle; if it was necessary for a process to add together two 32-bit numbers, then it would take more cycles, that make use of 16-bit addition.

Writing Something on screen

Now we are going to write a seemingly simple boot sector program that prints a short message on the screen. We'd like to print a character on the screen but we do not know exactly how to communicate with the screen device, since there may be many different kinds of screen devices and they may have different interfaces. This is why we need to use BIOS, since BIOS has already done some auto detection of the hardware and, evidently by the fact that BIOS earlier printed information on the screen about self-testing and so on, so can offer us a hand.
Here we can make use of a fundamental mechanism of the computer: *interrupts.*

Interrupts

Each interrupt is represented by a unique number that is an index to the interrupt vector, a table initially set up by BIOS at the start of memory (i.e. at physical address 0x0) that contains address pointers to interrupt service routines (ISRs).

So, in a nutshell, BIOS adds some of its own ISRs to the interrupt vector that specialise in certain aspects of the computer, for example: interrupt 0x10 causes the screen-related ISR to be invoked; and interrupt 0x13, the disk-related I/O ISR.

CPU Registers

Just as we use variables in a higher level language, it is useful if we can store data temporarily during a particular routine. All x86 CPUs have four general purpose registers, ax, bx, cx, and dx, for exactly that

purpose. Also, these registers, which can each hold a word (two bytes, 16 bits) of data, can be read and written by the CPU with negligible delay as compared with accessing main memory. In assembly programs, one of the most common operations is moving (or more accurately, copying) data between these registers:

```
mov ax, 1234        ; store the decimal number 1234 in ax
mov cx, 0x234       ; store the hex number 0x234 in cx
mov dx, 't'         ; store the ASCII code for letter 't' in dx
mov bx, ax          ; copy the value of ax into bx, so now bx == 1234
```

Notice that the destination is the first and not second argument of the mov operation, but this convention varies with different assemblers.

Combining all the steps

So, recall that we'd like BIOS to print a character on the screen for us, and that we can invoke a specific BIOS routine by setting ax to some BIOS-defined value and then triggering a specific interrupt. The specific routine we want is the BIOS scrolling tele-type routine, which will print a single character on the screen and advance the cursor, ready for the next character. There is a whole list of BIOS routines published that show you which interrupt to use and how to set the registers prior to the interrupt. Here, we need interrupt 0x10 and to set ah to 0x0e (to indicate tele-type mode) and al to the ASCII code of the character we wish to print.

Defining Strings

We have to remind ourselves that our computer knows nothing about strings, and that a string is merely a sequence of data units (e.g. bytes, words, etc.) held somewhere in memory.
In the assembler we can define a string as follows:

```
my_string:
  db 'Booting OS'
```

We've actually already seen db, which translates to "declare byte(s) of data", which tells the assembler to write the subsequent bytes directly to the binary output file (i.e. do not interpret them as processor instructions).
This leads to the creation of *print_string.asm*.

Control Structures

After compilation, these high-level control structures reduce to simple jump statements. Actually, we've already seen the simplest example of loops:

```
some_label:
  jmp some_label   ; jump to address of label
```

Or alternatively, with identical effect:

```
jmp $              ; jump to address of current instruction
```

Calling Functions

The CPU keeps track of the current instruction being executed in the special register ip (instruction pointer), which, sadly, we cannot access directly. However, the CPU provides a pair of instructions, call and ret, which do exactly what we want: call behaves like jmp but additionally, before actually jumping, pushes the return address on to the stack; ret then pops the return address off the stack and jumps to it, as follows:

```
   mov al, 'H'         ; Store 'H' in al so our function will print it.
   call my_print_function
   ...
   ...

my_print_function:
   mov ah, 0x0e        ; int=10/ah=0x0e -> BIOS tele-type output
   int 0x10            ; print the character in al
   ret
```

Include Files

After slaving away even on the seemingly simplest of assembly routines, you will likely want to reuse your code in multiple programs. nasm allows you to include external files literally as follows:

```
%include "my_print_function.asm"    ; this will simply get replaced by
                                    ; the contents of the file

...
mov al, 'H'         ; Store 'H' in al so our function will print it.
call my_print_function
```

Entering 32-bit Protected Mode

Instead of continuing the further work in the 16-bit real mode, we switch over to a 32-bit protected mode in order to make fuller use of the CPU, and take advantage of developments of CPU architectures for memory protection in hardware. The different steps involved in this change are:

Instructions without BIOS

BIOS routines, which are coded to work only in 16-bit real mode, are no longer valid in 32-bit protected mode; indeed, attempting to use them would likely crash the machine. So, a 32-bit operating system must provide its own drivers for all hardware of the machine. Even to print a message on the screen, we have to do it the hard way. And to do so, we need to understand that the display device can be configured into one of several resolutions in one of two modes, *text mode* and *graphics mode*; and that

what is displayed on the screen is a visual representation of a specific range of memory. So, in order to manipulate the screen, we must manipulate the specific memory range that it is using in its current mode.

When most computers boot, they begin in a simple Video Graphics Array (VGA) colour text mode with dimensions 80x25 characters. In text mode, each character cell of the screen is represented by two bytes in memory: the first byte is the ASCII code of the character to be displayed, and the second byte encodes the characters attributes, such as the foreground and background colour and if the character should be blinking. So, to display a character on the screen, then we need to set its ASCII code and attributes at the correct memory address for the current VGA mode, which is usually at address 0xb8000. Making these modifications to the original *print_string.asm*, we create a 32-bit protected mode routine that writes directly to video memory, as in *print_string_pm.asm*.

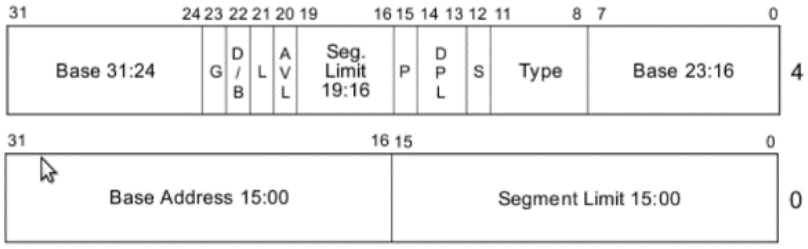
Global Descriptor Table (GDT) in Assembly

This is the most difficult part about switching the CPU from 16-bit real mode into 32-bit protected mode. We must prepare a complex data structure in memory called the global descriptor table (GDT), which defines memory segments and their protected-mode attributes. Once we have defined the GDT, we can use a special instruction to load it into the CPU, before setting a single bit in a special CPU control register to make the actual switch.

Now, to translate logical addresses (i.e. the combination of a segment register and an offset) to physical address, the segment register is used as an index to a particular segment descriptor (SD) in the GDT. A segment descriptor is an 8-byte structure that defines the following properties of a protected-mode segment:

- Base address (32 bits), which defines where the segment begins in physical memory
- Segment Limit (20 bits), which defines the size of the segment
- Various flags, which affect how the CPU interprets the segment, such as the privilege level of code that runs within it or whether it is read- or write-only.

Figure below shows the actual structure of the segment descriptor.



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

In addition to the code and data segments, the CPU requires that the first entry in the GDT purposely be an invalid null descriptor (i.e. a structure of 8 zero bytes) to catch mistakes where we forget to set a particular segment register before accessing an address. Moreover, since we are using a simple flat model, the two overlapping code and data segments are identical except for the type flags.

Also, for the simple reason that the CPU needs to know how long our GDT is, we don't actually directly give the CPU the start address of our GDT but instead give it the address of a much simpler structure called the GDT, a 6-byte structure containing:

- GDT size (16 bits)
- GDT address (32 bits)

The code file *gdt.asm* defines our GDT and the GDT descriptor. In the code, we use db, dw and dd assembly directives to fill out parts of the structure and the flags are more conveniently defined using literal binary numbers, that are suffixed with b.

Making the switchover from 16-bit mode to 32-bit mode

Once both the GDT and the GDT descriptor have been prepared within our boot sector, we are ready to instruct the CPU to switch from 16-bit real mode into 32-bit protected mode. The first thing we have to do is disable interrupts using the cli (clear interrupt) instruction, which means the CPU will simply ignore any future interrupts that may happen, unless they are later enabled. Then, to tell the CPU about the prepared GDT, we use a single instruction to do this, to which we pass the GDT descriptor:

```
lgdt [gdt_descriptor]
```

Now, to make the actual switch over, we set the first bit of a special CPU control register, cr0. However, we cannot set that bit directly on the register, so we must load it into a general purpose register, set the bit, and then store it back into cr0. Similar to the and instruction, we can use the or instruction to include certain bits into a value without disturbing any other bits.

```
mov eax, cr0    ; To make the switch to protected mode, we set
or eax, 0x1     ; the first bit of CR0, a control register
mov cr0, eax    ; Update the control register
```

After cr0 has been updated, the CPU is in 32-bit protected mode.

Next, immediately after instructing the CPU to switch mode, we issue a far jump to another segment to force the CPU to flush the pipeline (i.e. complete all of instructions currently in different stages of the pipeline). This is done to avoid the risk that the CPU may process some stages of an instruction's execution in the wrong mode. To issue a far jump, as opposed to a near (i.e. standard) jump, we additionally provide the target segment, as follows:

```
jmp <segment>:<address offset>
```

However, a near jump, such as jmp start protected mode (a label that marks the beginning of our 32-bit Code) may not be sufficient to flush the pipeline, and, besides our current code segment (i.e. cs) will not be valid in protected mode. So, we must update our cs register to the offset of the code segment descriptor of our GDT which is 0x8. This will automatically cause the CPU to update our cs register to the

target segment. Using the assembler to calculate these segment descriptor offsets and storing them as the constants CODE SEG and DATA SEG, we now arrive at our jump instruction:

```
    jmp CODE_SEG:start_protected_mode

[bits 32]

start_protected_mode:
    ...             ; By now we are assuredly in 32-bit protected mode.
    ...
```

Note that we need to use the [bits 32] directive to tell our assembler that, from that point onwards, it should encode in 32-bit mode instructions. However, we can still use 32-bit instructions in 16-bit real mode as we made use of the 32-bit register eax to set the control bit.
We combine this whole process into a re-usable routine to arrive at our code file *switch_to_pm.asm*.

Finally, we can include all of our routines into a boot sector *boot.asm* that demonstrates the switch from 16-bit real mode into 32-bit protected mode.

Writing, Building and Loading our Kernel

We have learnt a lot about how the computer really works, by communicating with it in the low-level assembly language, but we've also seen how it can be very slow to progress in such a language. Another drawback of us continuing in assembly language is that it is closely tied to the specific CPU architecture, and so it would be harder for us to port our operating system to another CPU architecture (e.g. ARM, PowerPC). So, we decided to write higher-level language compilers (e.g. FORTRAN, C, Pascal, etc.), that would transform more intuitive code into assembly language. The idea of these compilers is to map higher level constructs, such as control structures and function calls onto assembly template code.

Generating Raw Machine Code

We will produce an object file which is annotated machine code, where meta information, such as textual labels remain present. One big advantage of this format (.o) is that the code may be easily relocated into a larger binary file when linked in with routines from other routines in other libraries, since code in the object file uses relative rather than absolute internal memory references. In order to create the actual executable code, the linker links together all the routines described in the input object files into one executable binary file.

We specify an output format of binary because it would be no good trying to run machine code intermingled with meta data on CPU since we are interested in writing operating system. We have compiled the C code into a raw machine code file. The C code is decoded into 32-bit assembly instructions using disassembler. Now, we can write a simple kernel in C.

Calling Functions

The default calling convention used by any high-level language compiler is to push arguments onto the stack in reverse order, so the first argument is on the top of the stack.

Executing our Kernel Code

The involved steps in developing operating system are as follows:

- Write and compile the kernel code
- Write and assemble the boot sector code
- Create a kernel image that includes not only our boot sector but our compiled kernel code
- Load our kernel code into memory
- Switch to 32-bit protected mode
- Begin executing our kernel code

Writing our Kernel

The main function of Kernel is to let us know it has been successfully loaded and executed. We tell the linker the origin of code because that is where BIOS loads and then executes it.

Creating Boot Sector

The BIOS will load only our boot sector and not our kernel. To simplify the problem of which disk and from which sectors to load the kernel code, the boot sector and kernel of an operating system can be grafted together into a kernel image.
We can create our kernel image with the command:

```
cat boot_sect.bin kernel.bin > os-image
```

To enter the kernel correctly, a very simple assembly routine is attached to start of kernel machine code. So, we can make sure that the first instruction will eventually result in the kernel's entry function being reached.
Content of ***entry.asm*** (Kernel's entry function):

```
call main       ; invoke main() in our C kernel
jmp $           ; Hang forever when we return from the kernel
```

Build with make

We will consider 'make' for building amongst other operating systems and applications. The basic principle of make is that we specify in a configuration file (usually called MakeFile) how to convert one file into another, such that the generation of one file may be described to depend on the existence of one or more other file. An example for compiling a C file into an object file is:

```
kernel.o : kernel.c
        gcc -ffreestanding -c kernel.c -o kernel.o
```

The final code base of the project is organised into the following folders:

- Boot: anything related to booting and the boot sector (e.g. *print_string.asm, gdt.asm, switch_to_pm.asm, boot_sect.asm*, etc.)
- Kernel: the kernel's main file, *kernel.c*
- Drivers: any hardware specific driver code

Additional Work

So far, our kernel is capable of printing an 'X' in the corner of the screen, which, whilst is sufficient to let us know our kernel has been successfully loaded and executed, doesn't tell us much about what is happening on the computer. It would be much nicer if we could create an abstraction of the screen that would allow us to write print(``Hello''), Not only would this sort of abstraction make it easier to display information within other code of our kernel, but it would allow us to easily substitute one display driver for another at a later date.

Screen Driver

Characters are displayed on the screen by writing them somewhere within the display buffer at address 0xb8000. Although we could write all of this code in *kernel.c*, that contains the kernel's entry function, *main(),* it is good to organise such functionality-specific code into it's own file, which can be compiled and linked to our kernel code, ulimately with the same effect as putting it all into one file. The files in folder 'drivers' are:

- *screen.c* - New Driver Implementation File
- *screen.h* - a driver interface file

Summary and Procedure to run the OS

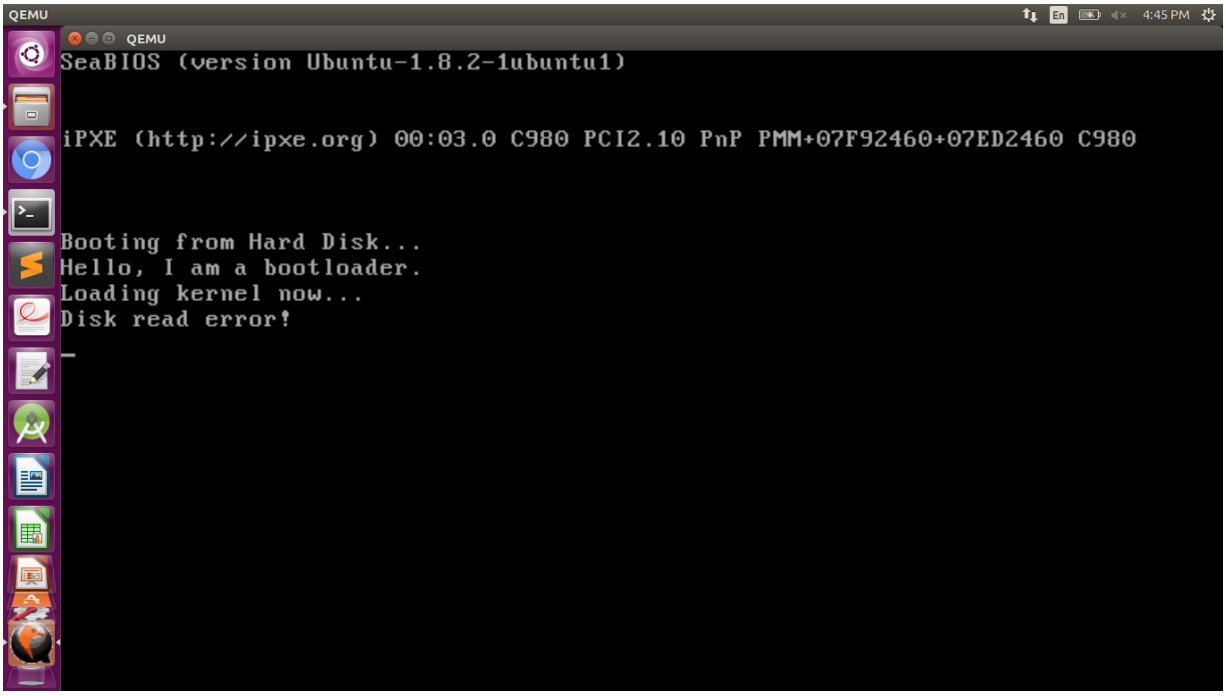
In order to run the OS, one simply needs to run the Makefile**,** present inside the Code folder, with the help of the command make in Linux Terminal since the project has been built on a Linux platform. The file contains all the necessary commands nested within one another which automate everything and load the OS.

The overall system is set in motion with the call of *os-image* which is run by the QEmu system as described earlier. The os-image file is actually a kernel image file which results from the concatenation in memory of two binary files *boot_sect.bin* and *kernel.bin* followed by conversion to the image file. The two binary files, in turn, are created as follows:

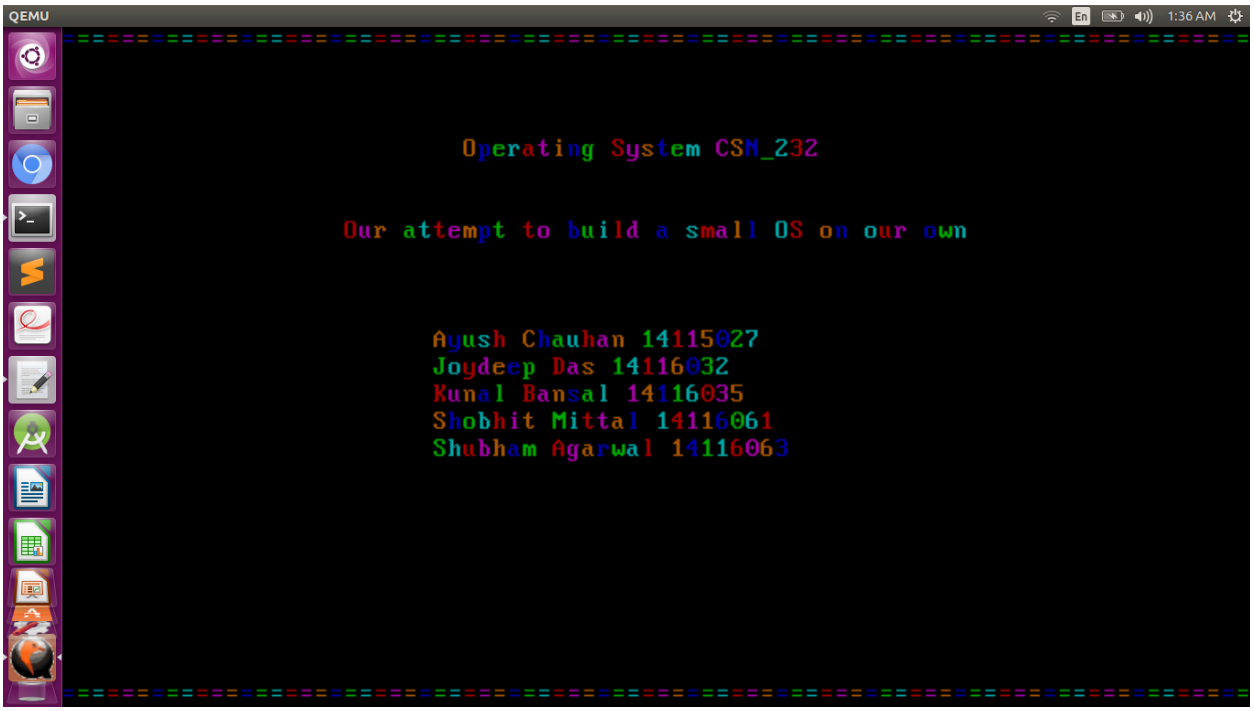
- The nasm assembler runs on the assembly language file *boot.asm* which has been discussed above and converts it the binary file *boot_sect.bin*.
- *kernel.bin* results from the conversion of the combined object file of *kernel.o* and *entry.o* object files. These two object files are obtained from high-level language file *kernel.c* and assembly language file *entry.asm*. This is done so to ensure that the main() function in *kernel.c* is the one which is called, upon the entry to the file. Also, since it is difficult to combine two binary files, the files are first converted to object files and then to binary.

Results

Initially when our kernel is successfully loaded and executed, it is capable of just printing an 'X' in the corner of the screen, which, although may be considered sufficient to let us know that it is working but doesn't appear very nice. The output as this stage looks something like this:



Hence, we went ahead to create an abstraction of the screen that allowed us to print strings on the display as shown below within the code of our kernel. It would also allow us to easily substitute one display driver for another at a later date.



Future Work

Scrolling the screen automatically when your cursor reached the bottom of the screen. It seems like a natural thing, we have complete control over the hardware, and can be implemented further. Another work could be implementation of keyboard drivers.

References

- https://www.tutorialspoint.com/operating_system/
- https://www.geeksforgeeks.org/operating-systems/
- https://samypesse.gitbooks.io/how-to-create-an-operating-system/
- Operating Systems Concepts by Silberschatz, Galvin and Gagne

Source Code

https://github.com/kunal017/Operating_Systems_CSN232