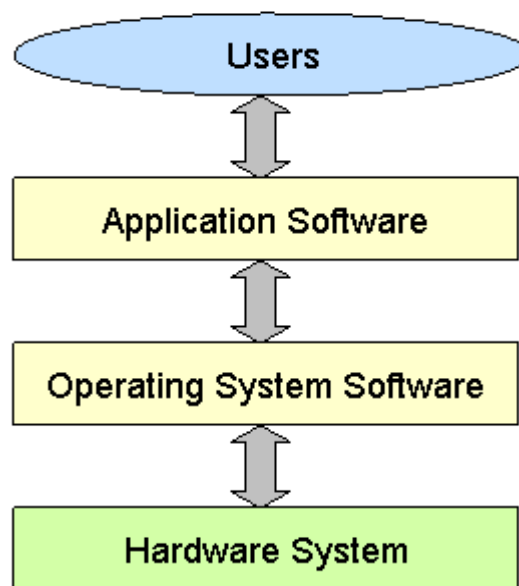


CHAPTER 1

1.1 INTRODUCTION

This project is intended to demonstrate and teach operating system development from the ground up. Operating systems can be a very complex topic. Learning how operating systems work can be a great learning experience. An Operating System provides the basic functionality, look, and feel, for a computer. The primary purpose is to create a workable Operating Environment for the user. An example of an Operating System is Windows, Linux, and Macintosh.



An operating system is system software that manages computer hardware and software resources and provides common services for computer programs. The operating system is a component of the system software in a computer system. Application programs usually require an operating system to function. For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers.

1.2 BLAST FROM THE PAST

Most of today's operating systems are graphical. These graphical user interfaces (GUI), however, provide a large abstraction layer to what is really going on in an OS. A lot of the concepts about operating systems date back since programs were on tape. A lot of these concepts still remain active today.

Prehistory - The Need for Operating Systems

Prior to the 1950s, all programs were on punch cards. These punch cards represented a form of instructions, which would control every facet of the computer hardware. Each piece of software would have full control of the system. Most of the time, the software would be completely different with each other. Even the versions of a program.

The problem was that each program was completely different. They had to be simply because they had to be always rewritten from scratch. There was no common support for the software, so the software had to communicate directly with the hardware. This also made portability and compatibility impossible.

During the realm of Mainframe computers, creating code libraries became more feasible. While it did fix some problems, such as two versions of software being completely different, each software still had full control of hardware.

If new hardware came out, the software will not work. If the software crashed, it would need to be debugged using light switches from a control panel.

The idea of an interface between hardware and programs came during the Mainframe era. By having an abstraction layer to the hardware, programs will no longer need to have full control, but instead they all would use a single common interface to the hardware

1950s - Yes there were OSs then

The first real operating system recorded, according to Wikipedia, is the GM-NAA I/O. The SHARE Operating System was a successor of the GM-NAA I/O. SHARE provided sharing programs, managed buffers, and was the first OS to allow the execution of programs written in Assembly Language. SHARE became the standard OS for IBM computers in the late 1950s.



The SHARE Operating System (SOS) was the first OS to manage buffers, provide program sharing, and allow execution of assembly language programs.

"Managing Buffers" relate to a form of "Managing Memory". "Program Sharing" relates to using libraries from different programs.

The two important things to note here are that, since the beginning of time, Operating Systems are responsible for Memory Management and Program Execution/Management

1964 - DOS/360 and OS/360

DOS/360 (or just "DOS") was a Disk Operating System was originally announced by IBM to be released on the last day of 1964. Due to some problems, IBM was forced to ship DOS/360 with 3 versions, each released June 1966.

The versions were:

- BOS/360 - 8KB Configuration.
- DOS/360 - 16KB Configuration with disk.
- TOS/360 - 16KB Configuration with tape.

A couple of important things to note is that DOS/360 offered no **Multitasking**, and no **Memory Protection**. The OS/360 was being developed about the same time by IBM. The OS/360 used "OS/MFT" (Multiple Fixed Transactions) to support multiple programs, with a **Fixed Base Address**. With OS/MVT (Multiple Variable Transaction), it can support varies program sizes.

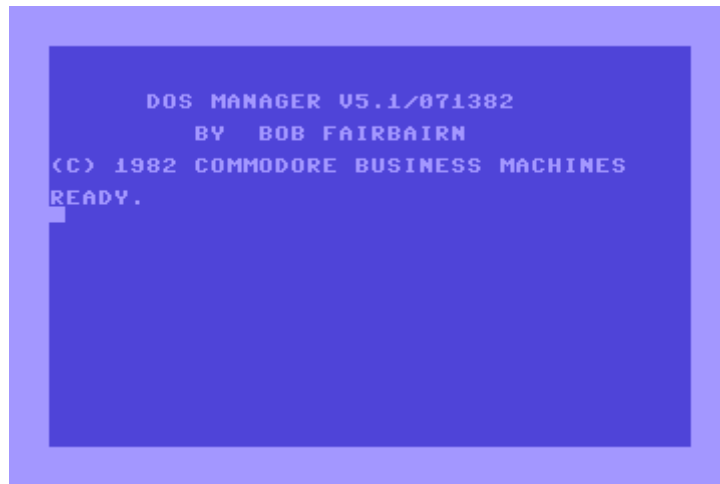
1969 - Its Unix!

The Unix Operating System was originally written in C. Both C and Unix were originally created by AT&T. Unix and C were freely distributed to government and academic institutions, causing it to be ported to a wider variety of machine families than any other OS.

Unix is a **multiuser, Multitasking** Operating System.

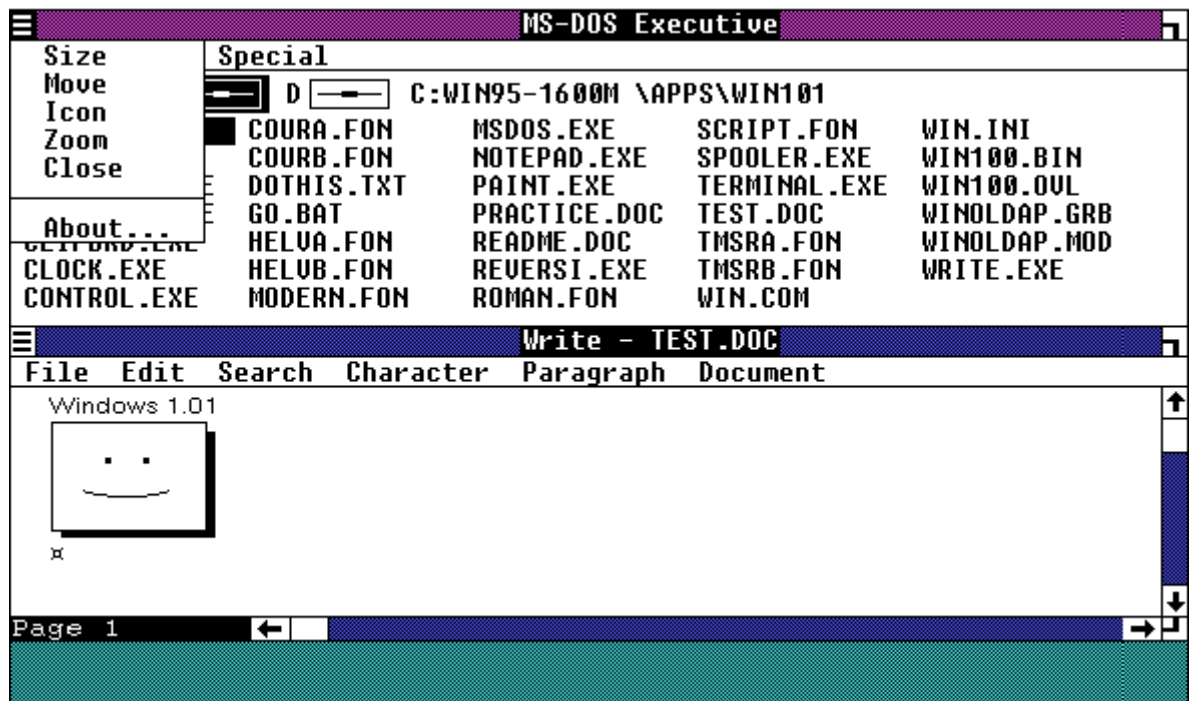
Unix includes a **Kernel, File System** and a **Command Shell**. There are a lot of **Graphical User Interfaces (GUI)** that uses the **Command Shell** to interact with the OS, and provide a much friendlier and nicer look.

1982 - Commodore DOS



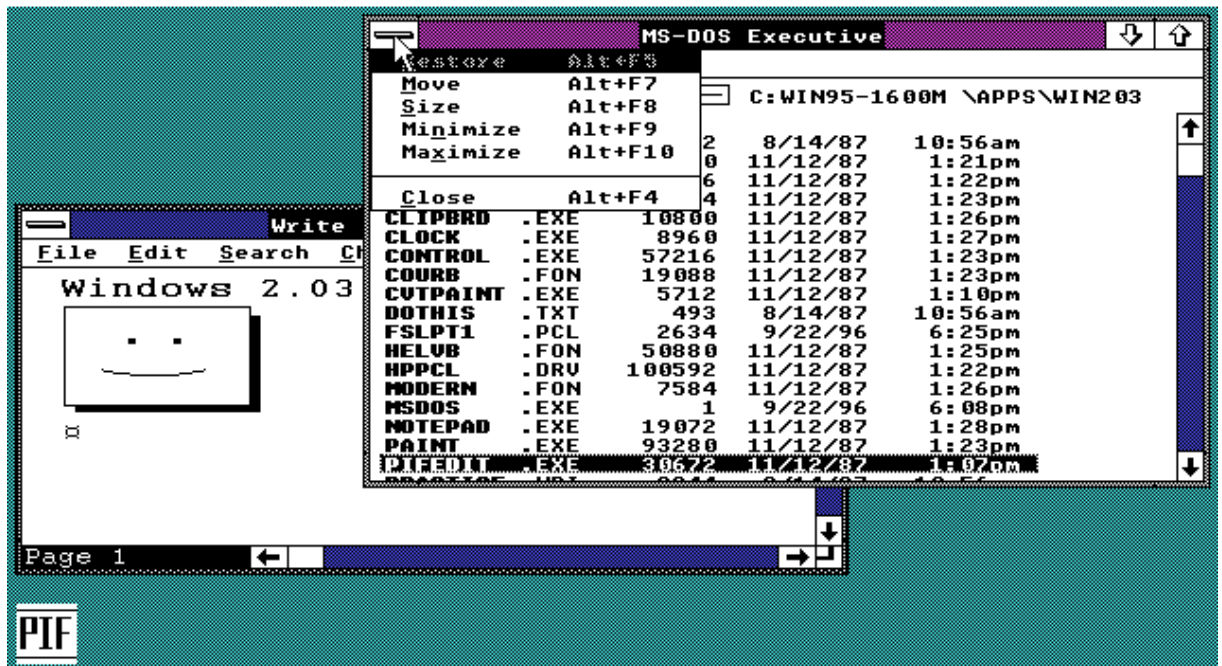
Commodore DOS (CBM DOS) was used with Commodore's 8 bit computers. Unlike the other computers before or since-which booted from disk into the systems memory at startup, CBM DOS executed internally within the drive-internal ROM chips, and was executed by an MOS 6502 CPU.

1985 - Microsoft Windows 1.0



The first Windows was a DOS application. Its "MSDOS Executive" program allows the running of a program. None of the "Windows" could overlap, however, so each "window" was displayed side to side. It was not very popular.

1987 - Microsoft Windows 2.0



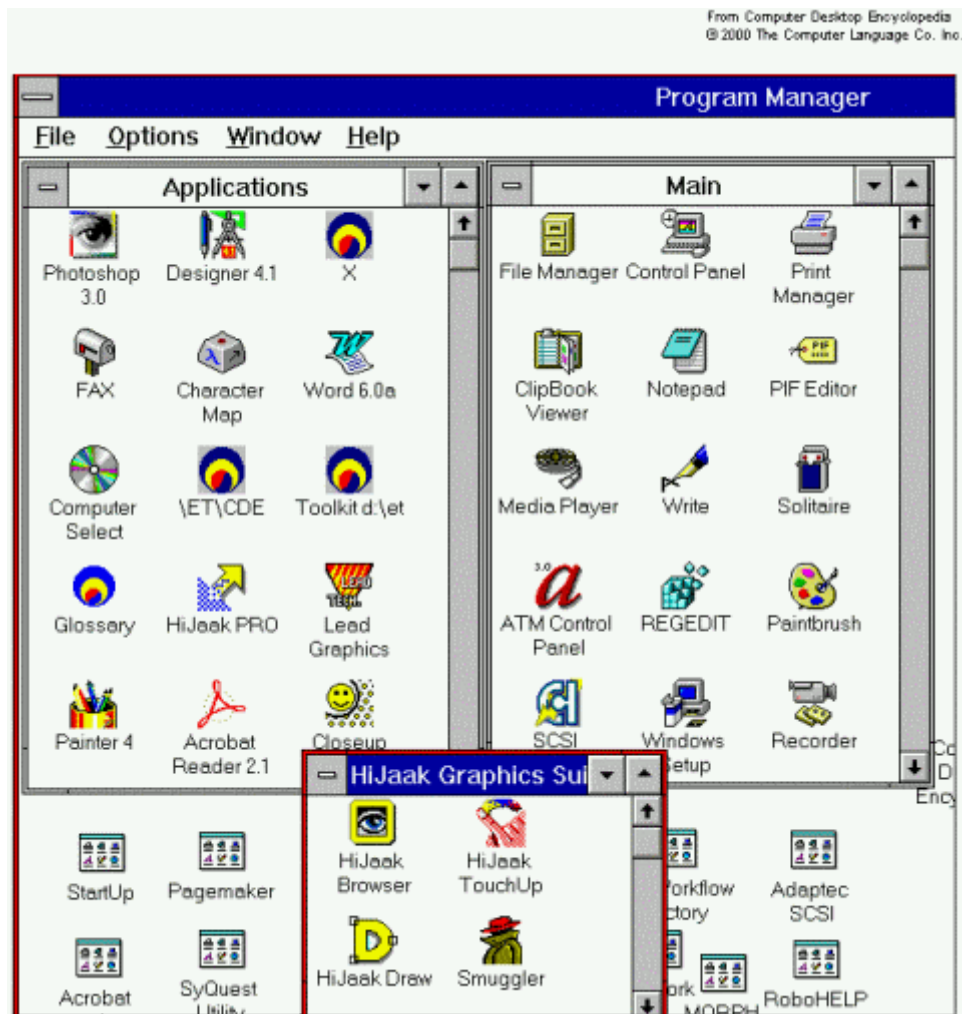
The second version of Windows was still a DOS **Graphical Shell**, but supported overlapping windows, and more colors. However, do to the limitation of DOS, it was not widely used.

Note: DOS is a 16 bit operating system. During this timeframe, DOS had to reference memory through Linear Addressing, and disks through LBA (Linear Block Addressing). Because the x86 platform is backward compatible, When the PC boots it is in 16 bit mode (Real Mode), and still has LBA. More on this later.

Do to 16 bit mode limitations, DOS could not access more than 1 MB of memory. This is solved, today, by enabling the 20th address line through the Keyboard Controller.

Because of this 1 MB limitation, Windows was far too slow, which was one primary reason of it being unpopular.

1987 - Microsoft Windows 3.0



Windows 2.0 was completely redesigned. Windows 3.0 was still a DOS Graphical Shell, however it included A "DOS Extender" to allow access to 16 MB of memory, over the 1 MB limit of DOS. It supports **multitasking** with DOS programs.

This is the Windows that made Microsoft big. It supports resizable windows, and movable windows.

1.3 FUNCTIONS OF OPERATING SYSTEM

1.3.1 Memory Management

Memory Management refers to:

- Dynamically giving and using memory to and from programs that request it.
- Implementing a form of **Paging**, or even **Virtual Memory**.
- Insuring the OS Kernel does not read or write to unknown or invalid memory.
- Watching and handling **Memory Fragmentation**.

1.3.2 Program Management

This relates closely with Memory Management. Program Management is responsible for:

- Insuring the program doesn't write over another program.
- Insuring the program does not corrupt system data.
- Handle requests from the program to complete a task (such as allocate or deallocate memory).

1.3.3 Multitasking

Multitasking refers to:

- Switching and giving multiple programs a certain timeframe to execute.
- Providing a **Task Manager** to allow switching (Such as Windows Task Manager).
- **TSS (Task State Segment) switching.**
- Executing multiple programs simultaneously.

1.3.4 Memory Protection

This refers to:

- Accessing an invalid descriptor in protected mode (Or an invalid segment address)
- Overwriting the program itself.
- Overwriting a part or parts of another file in memory.

1.3.5 Fixed Base Address

- A "Base Address" is the location where a program is loaded in memory. In normal applications programming, you wouldn't normally need this. In OS Development, however, you do.
- A "Fixed" Base Address simply means that the program will always have the same base address each time it is loaded in memory. Two example programs are the BIOS and the Bootloader.

1.3.6 Multiuser

This refers to:

- Login and Security Protection.
- Ability of multiple users to work on the computer.
- Switching between users without loss or corruption of data.

CHAPTER 2 Basic Requirements

2.1 KNOWLEDGE OF C LANGUAGE

How to use C in kernel land

16 bit and 32 bit C

If you want to create a 16 bit real mode OS, you **must** use a 16 bit C compiler. If, however, you decide that you would like to create a 32 bit OS, you **must** use a 32 bit C compiler. 16 bit C code is not compatible with 32 bit C code.

In this project, we will be creating a 32 bit operating system. Because of this, we will be using a 32 bit C compiler.

C and executable formats

A problem with C is that it does not support the ability to output **flat binary programs**. A flat binary program can basically be defined as a program where the **entry point routine** (such as `main()`) is always at the first byte of the program file. Wait, what? Why would we want this?

This goes back to the good old days of DOS COM programming. DOS COM programs were flat binary - they had no well-defined entry point nor **symbolic names** at all. To execute the program, all that needed to be done was to "jump" to the first byte of the program. Flat binary programs have no special internal format, so there was no standard. It's just a bunch of 1's and 0's. When the PC is powered on, the system BIOS ROM takes control. When it is ready to start an OS, it has no idea how to do it. Because of this, it runs another program - the **Boot Loader** to load an OS. The BIOS does not at all know what internal format this program file is or what it does. Because of this, it treats the Boot Loader as a **flat binary program**. It loads whatever is on the **Boot Sector** of the **Boot Disk** and "jumps" to the first byte of that program file.

Because of this, the first part of the boot loader, also called the **Boot Code** or **Stage 1** cannot be in C. This is because all C compilers output a program file that has a special internal format - they can be library files, object files, or executable files. There is only one language that natively supports this - assembly language.

How to use C in a boot loader

While it is true that the first part of the boot loader must be in assembly language, it is possible to use C in a boot loader. There are different ways of doing this. . We combine an assembly stub program and the C program in a single file. The assembly stub program sets up the system and calls our C program. Because both of these programs are combined into a single file, Stage 1 only needs to load a single file - which in turn loads both our stub program and C program.

This is one method - there are others. Most real boot loaders use C, including GRUB, Neptunes boot loader, Microsoft's NTLDR and Boot Manager. Because we are using 32 bit C, there are also ways that will allow us to mix 16 bit code with our 32 bit C code.

Here we have used a combination of both ‘C’ and ‘assembly language’.

When the boot loader is ready, it loads and executes our C kernel by calling its entry point routine. Because the C program follows a specific internal format, the boot loader must know how to parse the file and locate the entry point routine to call it. This allows us to use C for the kernel and other libraries that we build.

2.2 KNOWLEDGE OF ASSEMBLY LANGUAGE

Assembly Language is a low level programming language. Assembly Language provides a direct one to one relation with the processor machine instructions, which make assembly language suitable for hardware programming.

Assembly Language, as being low level, tend to be more complex and harder to develop in, then high level languages like C. Because of this, and to aid in simplicity, we are only going to use assembly language when required, and no more.

The assembly programming language is developed by using mnemonics. The microcontroller or microprocessor can understand only the binary language like 0's or 1's therefore the assembler convert the assembly language to binary language and store it the memory to perform the tasks. Before writing the program the embedded designers must have sufficient knowledge on particular hardware of the controller or processor, so first we required to know hardware of 8086 processor.

8086 Processor Architecture

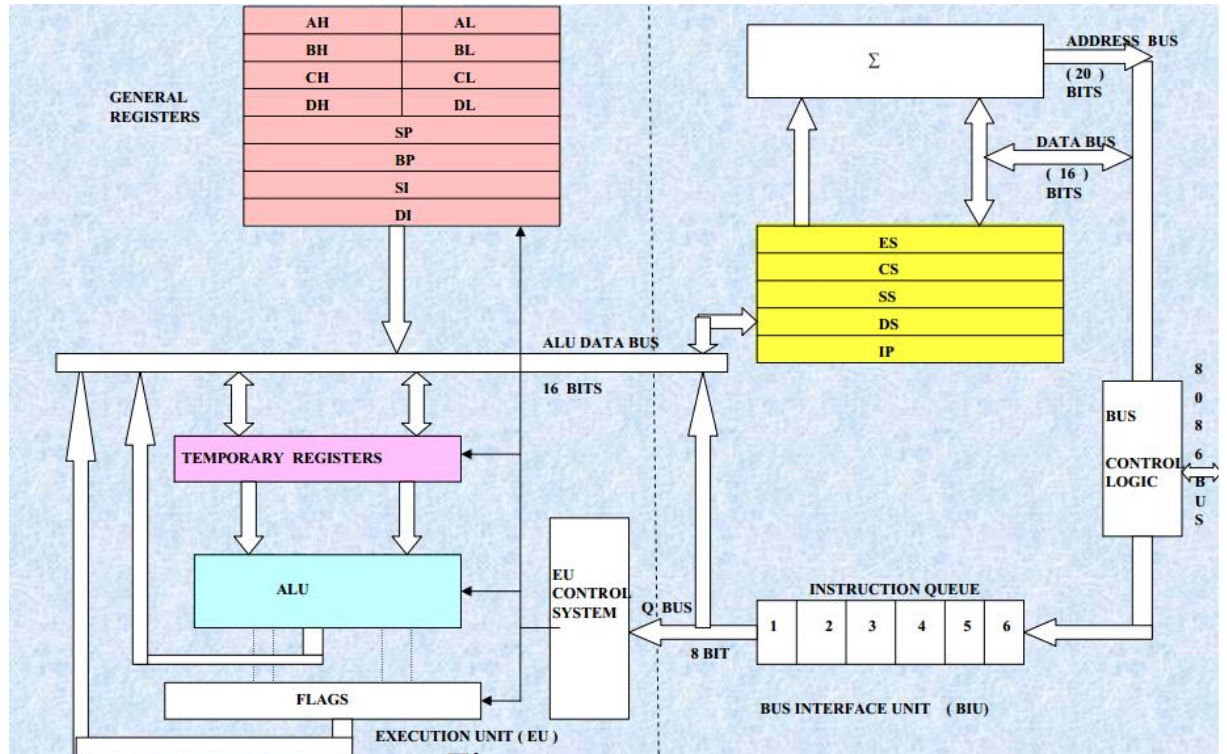
The assembly level programming 8086 is based on the memory registers. A Register is the main part of the microprocessors and controllers which are located in the memory that provides a faster way of collecting and storing the data. If we want to manipulate data to a processor or controller by performing multiplication, addition, etc., we cannot do that directly in the memory where need registers to process and to store the data. The 8086 microprocessor contains various kinds of registers that can be classified according to their instructions such as;

General purpose registers: The 8086 CPU has consisted 8-general purpose registers and each register has its own name as shown in the figure such as AX, BX, CX, DX, SI, DI, BP, SP . These all are 16-bit registers where four registers are divided into two parts such as AX, BX, CX, and DX which is mainly used to keep the numbers.

Special purpose registers: The 8086 CPU has consisted 2- special function registers such as IP and flag registers. The IP register point to the current executing instruction and always works to gather with the CS segment register. The main function of flag registers is to

modify the CPU operations after mechanical functions are completed and we cannot access directly

Segment registers: The 8086 CPU has consisted 4- segment registers such as CS, DS, ES, SS which is mainly used for possible to store any data in the segment registers and we can access a block of memory using segment registers.



Simple Assembly Language Programs 8086

The assembly language programming 8086 has some rules such as -

- The assembly level programming 8086 code must be written in upper case letters
- The labels must be followed by a colon, for example: label:
- All labels and symbols must begin with a letter
- All comments are typed in lower case
- The last line of the program must be ended with the END directive



Op-code: A single instruction is called as an op-code that can be executed by the CPU. Here the 'MOV' instruction is called as an op-code.

Operands: A single piece data are called operands that can be operated by the op-code. Example, subtraction operation is performed by the operands that are subtracted by the operand.

Syntax: SUB b, c

Write a Program For Reading and Displaying a Character

```
MOV ah, 1h      //keyboard input subprogram
INT 21h         //read character into al
MOV dl, al      //copy character to dl
MOV ah, 2h      //character output subprogram
INT 21h         // display character in dl
```

2.3 UBUNTU

Ubuntu is a Debian-based Linux operating system and distribution for personal computers, smartphones and network servers. It uses Unity as its default user interface. It is based on free software and named after the Southern African philosophy of *ubuntu* (literally, "human-ness"), which often is translated as "humanity towards others" or "the belief in a universal bond of sharing that connects all humanity".

2.4 VMWARE

In computing, a **virtual machine (VM)** is an emulation of a particular computer system. Virtual machines operate based on the computer architecture and functions of a real or hypothetical computer, and their implementations may involve specialized hardware, software, or a combination of both.

There are different kinds of virtual machines, each with different functions. System virtual machines (also known as full virtualization VMs) provide a complete substitute for the

targeted real machine and a level of functionality required for the execution of a complete operating system. A hypervisor uses native execution to share and manage hardware, allowing multiple different environments, isolated from each other, to be executed on the same physical machine. Modern hypervisors use hardware-assisted virtualization, which provides efficient and full virtualization by using virtualization-specific hardware capabilities, primarily from the host CPUs. Process virtual machines are designed to execute a single computer program by providing an abstracted and platform-independent program execution environment.

Presently, VMware runs on Windows, Linux, Macintosh, and Solaris hosts and supports a large number of guest operating systems including but not limited to Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7, Windows 8, Windows 10), DOS/Windows 3.x, Linux (2.4, 2.6, 3.x and 4.x), Solaris and OpenSolaris, OS/2, and OpenBSD

2.5 NASM

The **Netwide Assembler (NASM)** is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. NASM is considered to be one of the most popular assemblers for Linux.

NASM was originally written by Simon Tatham with assistance from Julian Hall. As of 2016, it is maintained by a small team led by H. Peter Anvin. It is open-source software released under the terms of a simplified (2-clause) BSD license.

NASM can output several binary formats including COFF, Portable Executable, Executable and Linkable Format (ELF), Mach-O and binary file (.bin, binary disk image, used to compile operating systems), though position-independent code is only supported for ELF object files. NASM also has its own binary format called RDOFF.

The variety of output formats allows retargeting programs to virtually any x86 operating system (OS). Also, NASM can create flat binary files, usable to write boot loaders, read-only memory (ROM) images, and in various facets of OS development. NASM can run on non-x86 platforms, such as PowerPC and SPARC, though it cannot generate programs usable by those machines.

NASM uses a variant of Intel assembly syntax instead of AT&T syntax. It also avoids features such as automatic generation of segment overrides (and the related ASSUME directive) used by MASM and compatible assemblers.

NASM principally outputs object files, which are generally not executable by themselves. The only exception to this are flat binaries (e.g., .COM) which are inherently limited in modern use. To translate the object files into executable programs, an appropriate linker must be used, such as the Visual Studio "LINK" utility for Windows or ld for Unix-like systems.

2.6 GCC COMPILER

GCC, the GNU Compiler Collection

The **GNU Compiler Collection (GCC)** is a compiler system produced by the GNU Project supporting various programming languages. GCC is a key component of the GNU toolchain. The Free Software Foundation (FSF) distributes GCC under the GNU General Public License (GNU GPL). GCC has played an important role in the growth of free software, as both a tool and an example.

Originally named the **GNU C Compiler**, when it only handled the C programming language, GCC 1.0 was released in 1987. It was extended to compile C++ in December of that year. Front ends were later developed for Objective-C, Objective-C++, Fortran, Java, Ada, and Go among others.

GCC has been ported to a wide variety of processor architectures, and is widely deployed as a tool in the development of both free and proprietary software. GCC is also available for most embedded platforms, including Symbian (called *gcce*), AMCC, and Freescale Power Architecture-based chips. The compiler can target a wide variety of platforms, including video game consoles such as the PlayStation 2 and Dreamcast.

As well as being the official compiler of the GNU operating system, GCC has been adopted as the standard compiler by many other modern Unix-like computer operating systems, including Linux and the BSD family, although FreeBSD and OS X have moved to the LLVM system. Versions are also available for Microsoft Windows and other operating systems; GCC can compile code for Android and iOS.

Some features of GCC include:

- **Link-time optimization** optimizes across object file boundaries to directly improve the linked binary. Link-time optimization relies on an intermediate file containing the serialization of some *Gimple* representation included in the object file. The file is generated alongside the object file during source compilation. Each source compilation generates a separate object file and link-time helper file. When the object files are linked, the compiler is executed again and uses the helper files to optimize code across the separately compiled object files.
- **Plugins** can extend the GCC compiler directly. Plugins allow a stock compiler to be tailored to specific needs by external code loaded as plugins. For example, plugins can add, replace, or even remove middle-end passes operating on *Gimple* representations. Several GCC plugins have already been published, notably the GCC Python Plugin, which links against libpython, and allows one to invoke arbitrary Python scripts from inside the compiler. The aim is to allow GCC plugins to be written in Python. The MELT plugin provides a high-level Lisp-like language to extend GCC.
- The current stable version of GCC is 6.1, which was released on April 27, 2016.
- As of version 4.8, GCC is implemented in C++.
- GCC 4.6 supports many new Objective-C features, such as declared and synthesized properties, dot syntax, fast enumeration, optional protocol methods, method/protocol/class attributes, class extensions and a new GNU Objective-C runtime API. It also supports the Go programming language and includes the

`libquadmath` library, which provides quadruple-precision mathematical functions on targets supporting the `__float128` datatype. The library is used to provide the `REAL(16)` type in GNU Fortran on such targets.

- GCC uses many standard tools in its build, including Perl, Flex, Bison, and other common tools. In addition it currently requires three additional libraries to be present in order to build: GMP, MPC, and MPFR.

2.7 TERMINAL

The terminal is an interface in which you can type and execute text based commands.

Why use it:

It can be much faster to complete some tasks using a Terminal than with graphical applications and menus. Another benefit is allowing access too many more commands and scripts.

A common terminal task of installing an application can be achieved within a single command, compared to navigating through the Software Centre or Synaptic Manager.

For example the following would install updates to the presently working system

```
sudo apt-get install update
```

A similar notation is:

```
# apt-get update
```

This means that the command should be run as root, that is, using `sudo`:

```
$ sudo apt-get update
```

Note that the `#` character is also used for *comments*.

```
# this command will give your username
whoami
# the next command will show the contents of the current directory
ls
```

How do I open a terminal:

- Open the **Dash** (Super Key) or **Applications** and type `terminal`
- Use the keyboard shortcut by pressing `Ctrl+Alt+T`.
- For older or Ubuntu versions:

Applications → Accessories → Terminal

Alternative names for the terminal:

- Console
- Shell
- Command Line
- Command Prompt

2.8 BOOTLOADER (GRUB)

GNU GRUB (short for **GNU GRand Unified Bootloader**) is a boot loader package from the GNU Project. GRUB is the reference implementation of the Free Software Foundation's Multiboot Specification, which provides a user the choice to boot one of multiple operating systems installed on a computer or select a specific kernel configuration available on a particular operating system's partitions

GRUB is a boot loader designed to boot a wide range of operating systems from a wide range of filesystems. **GRUB** is becoming popular due to the increasing number of possible root filesystems that can **Linux** can reside upon..

It is predominantly used for Unix-like systems. The GNU operating system uses GNU GRUB as its boot loader, as do most Linux distributions. The Solaris operating system has used GRUB as its boot loader on x86 systems, starting with the Solaris 10 1/06 release.

How does GRUB work?

When a computer boots, the BIOS transfers control to the first boot device, which can be a hard disk, a floppy disk, a CD-ROM, or any other BIOS-recognized device. We'll concentrate on hard disks, for the sake of simplicity.

The first sector on a hard is called the Master Boot Record (MBR). This sector is only 512 bytes long and contains a small piece of code (446 bytes) called the primary boot loader and the partition table (64 bytes) describing the primary and extended partitions.

By default, MBR code looks for the partition marked as active and once such a partition is found, it loads its boot sector into memory and passes control to it.

GRUB replaces the default MBR with its own code.

Furthermore, GRUB works in stages.

Stage 1 is located in the MBR and mainly points to Stage 2, since the MBR is too small to contain all of the needed data.

Stage 2 points to its configuration file, which contains all of the complex user interface and options we are normally familiar with when talking about GRUB. Stage 2 can be located anywhere on the disk. If Stage 2 cannot find its configuration table, GRUB will cease the boot sequence and present the user with a command line for manual configuration.

Stage 1.5 also exists and might be used if the boot information is small enough to fit in the area immediately after MBR.

The Stage architecture allows GRUB to be large (~20-30K) and therefore fairly complex and highly configurable, compared to most bootloaders, which are sparse and simple to fit within the Everything is a file

To be able to successfully master the secrets of GRUB, you must understand one of the basic foundations of *NIX-based operating systems. Everything is a file. Even hard disks and partitions are treated as files. There is no magic. If you remember this, you will find the supposedly perilous task of tampering with partitions no different than playing with files using a file explorer (or the command line). Now that we have established this, we can move on to the more technical parts of grubbing.

- Limitations of the Partition Table. *GRUB provides a true command-based, pre-OS environment on x86 machines.* This feature affords the user maximum flexibility in loading operating systems with specified options or gathering information about the system. For years, many non-x86 architectures have employed pre-OS environments that allow system booting from a command line.
- *GRUB supports Logical Block Addressing (LBA) mode.* LBA places the addressing conversion used to find files in the hard drive's firmware, and is used on many IDE and all SCSI hard devices. Before LBA, boot loaders could encounter the 1024-cylinder BIOS limitation, where the BIOS could not find a file after the 1024 cylinder head of the disk. LBA support allows GRUB to boot operating systems from partitions beyond the 1024-cylinder limit, so long as the system BIOS supports LBA mode. Most modern BIOS revisions support LBA mode.
- *GRUB can read ext2 partitions.* This functionality allows GRUB to access its configuration file, `/boot/grub/grub.conf`, every time the system boots, eliminating the need for the user to write a new version of the first stage boot loader to the MBR when configuration changes are made. The only time a user needs to reinstall GRUB on the MBR is if the physical location of the `/boot/` partition is moved on the disk.

2.9 QEMU

QEMU (short for **Quick Emulator**) is a free and open-source hosted hypervisor that performs hardware virtualization (not to be confused with hardware-assisted virtualization).

QEMU is a hosted virtual machine monitor: It emulates CPUs through dynamic binary translation and provides a set of device models, enabling it to run a variety of unmodified guest operating systems. It also can be used together with KVM in order to run virtual machines at near-native speed (requiring hardware virtualization extensions on x86 machines). QEMU can also be used purely for CPU emulation for user-level processes, allowing applications compiled for one architecture to be run on another.

QEMU is a generic and open source machine emulator and virtualizer.

When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.

When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, and S390 guests

Operation

QEMU has two operating modes:

- **Full system emulation.** In this mode, QEMU emulates a full system (for example a PC), including a processor and various peripherals. It can be used to launch different Operating Systems without rebooting the PC or to debug system code.
- **User mode emulation (Linux host only).** In this mode, QEMU can launch Linux processes compiled for one CPU on another CPU. For example, it can be used to launch Wine or to ease cross-compilation and cross-debugging.

CHAPTER 3: PROJECT IMPLEMENTATION

For implementation:-

3.1 Download the VMware virtual machine and install Ubuntu in it, in a customized way using the following steps :-



New Virtual Machine Wizard

Choose the Virtual Machine Hardware Compatibility

Which hardware features are needed for this virtual machine?

Virtual machine hardware compatibility

Hardware compatibility: Workstation 9.0

Compatible with: ☒ ESX Server

Compatible products:

Fusion 5.0
Workstation 9.0

Limitations:

64 GB memory limit
8 processor limit
10 network adapter limit
2 TB disk size limit

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Guest Operating System Installation

A virtual machine is like a physical computer; it needs an operating system. How will you install the guest operating system?

Install from:

☐ Installer disc:

DVD RW Drive (D:)

☒ Installer disc image file (iso):

C:\Users\Prabhash\Desktop\ubuntu-14.04.1-desktop-

Browse...

Ubuntu 64-bit 14.04.1 detected.

This operating system will use Easy Install. [\(What's this?\)](#)

☐ I will install the operating system later.

The virtual machine will be created with a blank hard disk.

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Easy Install Information
This is used to install Ubuntu 64-bit.

Personalize Linux

Full name:

User name:

Password:

Confirm:

Help < Back Next > Cancel

New Virtual Machine Wizard

Easy Install Information
This is used to install Ubuntu 64-bit.

Personalize Linux

Full name:

User name:

Password:

Confirm:

Help < Back Next > Cancel

New Virtual Machine Wizard

Name the Virtual Machine

What name would you like to use for this virtual machine?

Virtual machine name:

Location:

The default location can be changed at Edit > Preferences.

< Back

Next >

Cancel

New Virtual Machine Wizard

Processor Configuration

Specify the number of processors for this virtual machine.

Processors

Number of processors:

Number of cores per processor:

Total processor cores:

1

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Memory for the Virtual Machine

How much memory would you like to use for this virtual machine?

Specify the amount of memory allocated to this virtual machine. The memory size must be a multiple of 4 MB.

64 GB

32 GB

16 GB

8 GB

4 GB

2 GB

1 GB

512 MB

256 MB

128 MB

64 MB

32 MB

16 MB

8 MB

4 MB

Memory for this virtual machine:

1024

MB

Maximum recommended memory:

2900 MB

Recommended memory:

1024 MB

Guest OS recommended minimum:

512 MB

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Network Type

What type of network do you want to add?

Network connection

Use bridged networking

Give the guest operating system direct access to an external Ethernet network. The guest must have its own IP address on the external network.

Use network address translation (NAT)

Give the guest operating system access to the host computer's dial-up or external Ethernet network connection using the host's IP address.

Use host-only networking

Connect the guest operating system to a private virtual network on the host computer.

Do not use a network connection

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Select a Disk

Which disk do you want to use?

Disk

☒ Create a new virtual disk
A virtual disk is composed of one or more files on the host file system, which will appear as a single hard disk to the guest operating system. Virtual disks can easily be copied or moved on the same host or between hosts.

☐ Use an existing virtual disk
Choose this option to reuse a previously configured disk.

☐ Use a physical disk (for advanced users)
Choose this option to give the virtual machine direct access to a local hard disk.

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Select a Disk Type

What kind of disk do you want to create?

Virtual disk type

☐ IDE

☒ SCSI (Recommended)

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Specify Disk Capacity

How large do you want this disk to be?

Maximum disk size (GB):

20.0

Recommended size for Ubuntu 64-bit: 20 GB

☐ Allocate all disk space now.
Allocating the full capacity can enhance performance but requires all of the physical disk space to be available right now. If you do not allocate all the space now, the virtual disk starts small and grows as you add data to it.

☐ Store virtual disk as a single file
☒ Split virtual disk into multiple files
Splitting the disk makes it easier to move the virtual machine to another computer but may reduce performance with very large disks.

Help

< Back

Next >

Cancel

New Virtual Machine Wizard

Specify Disk File

Where would you like to store the disk file?

Disk File

One disk file will be created for each 2 GB of virtual disk capacity. File names for each file beyond the first will be automatically generated using the file name provided here as a basis.

Ubuntu 64-bit (2).vmdk

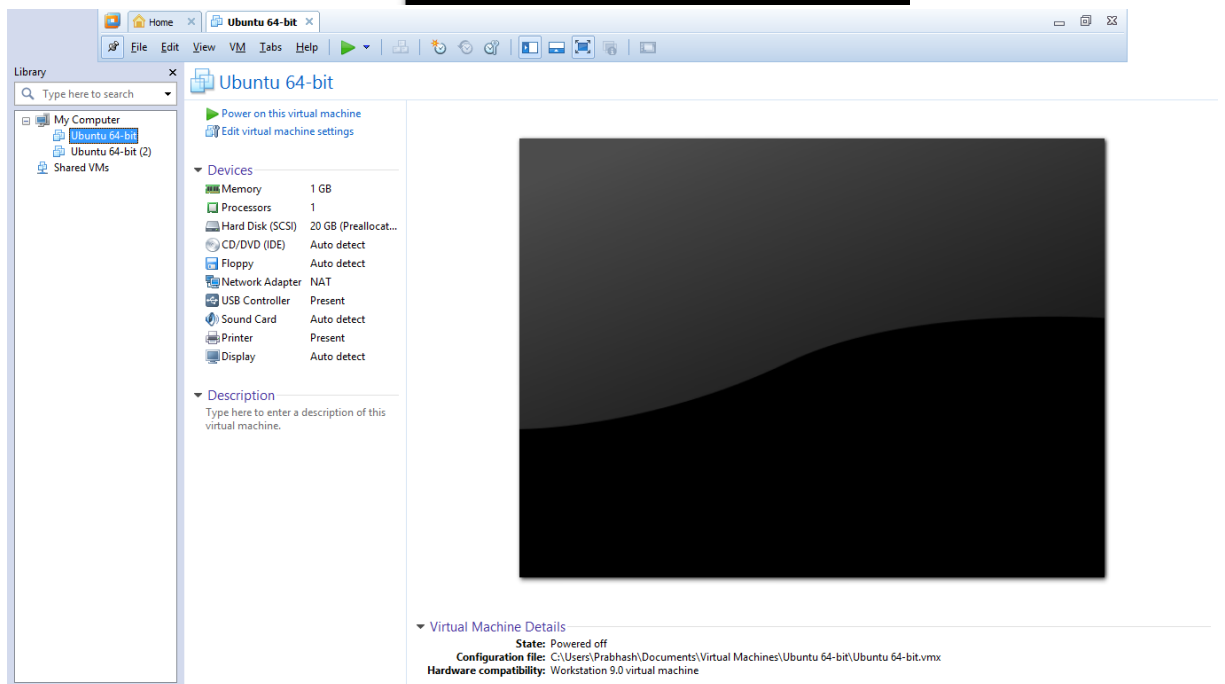
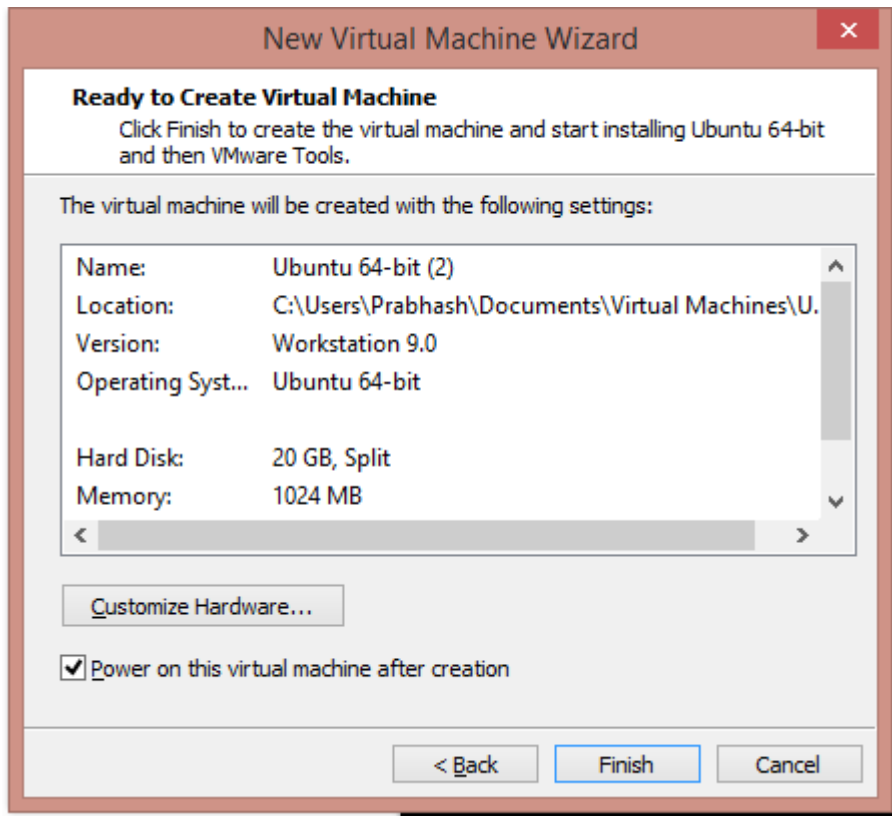
Browse...

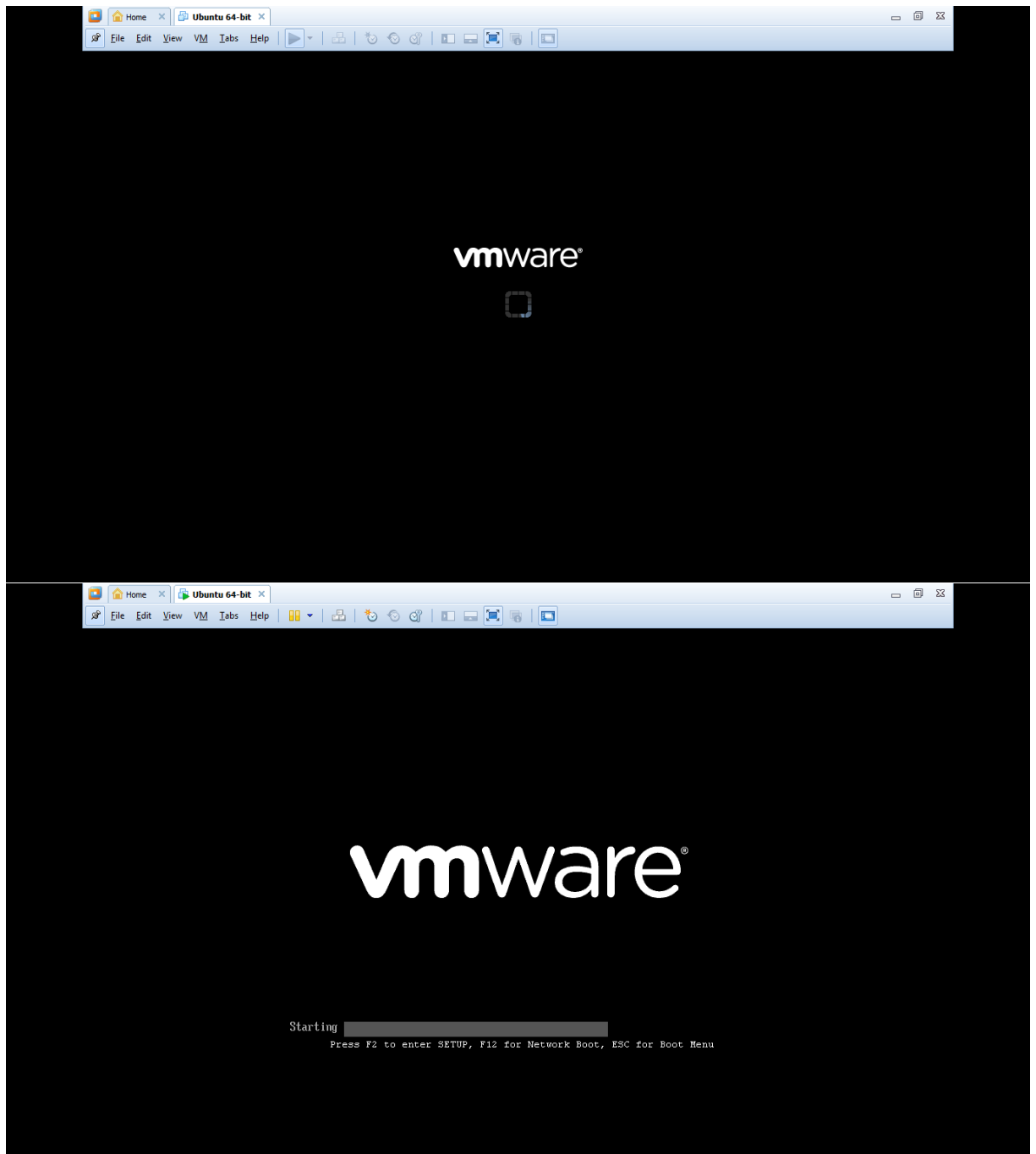
Help

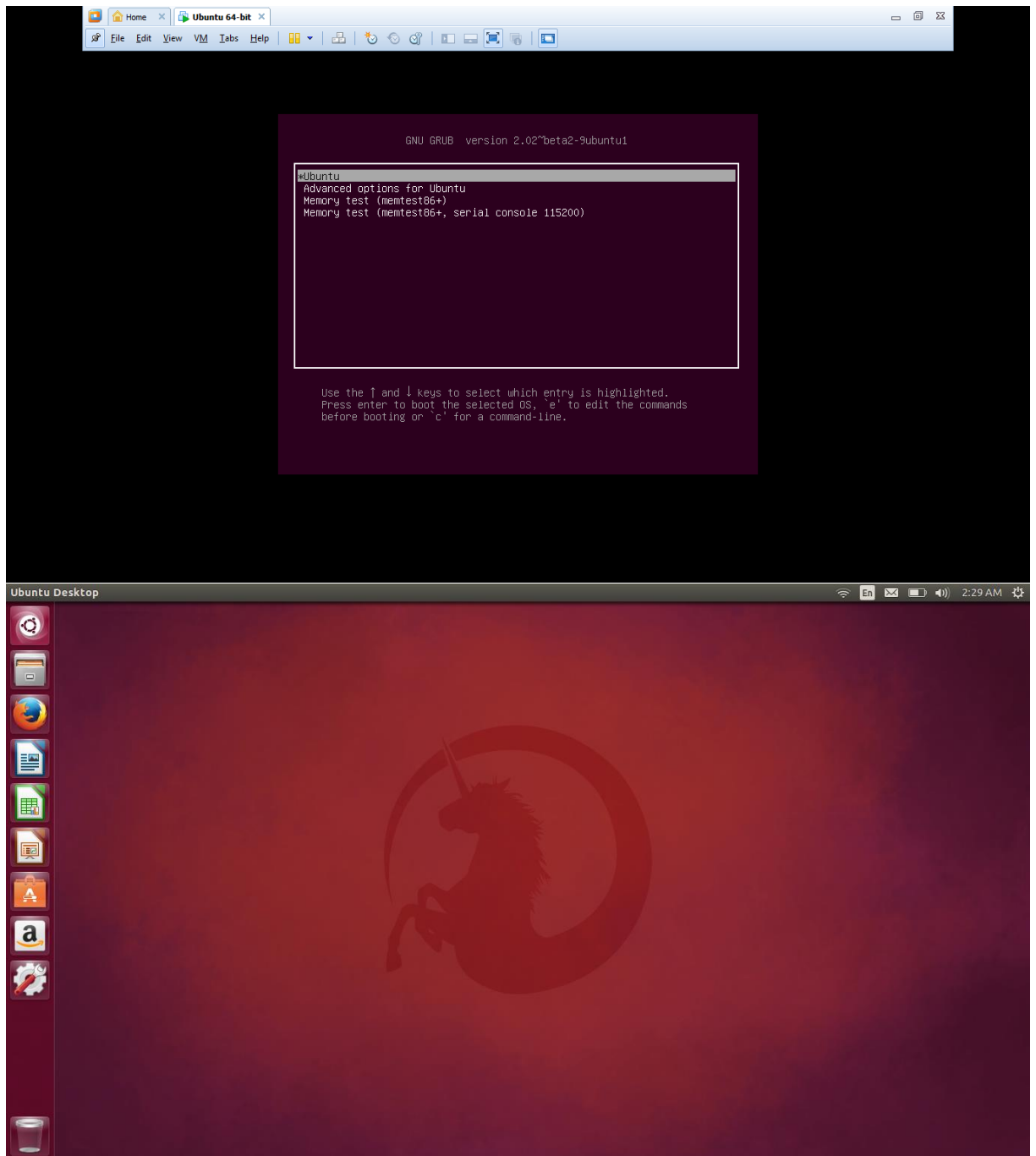
< Back

Next >

Cancel

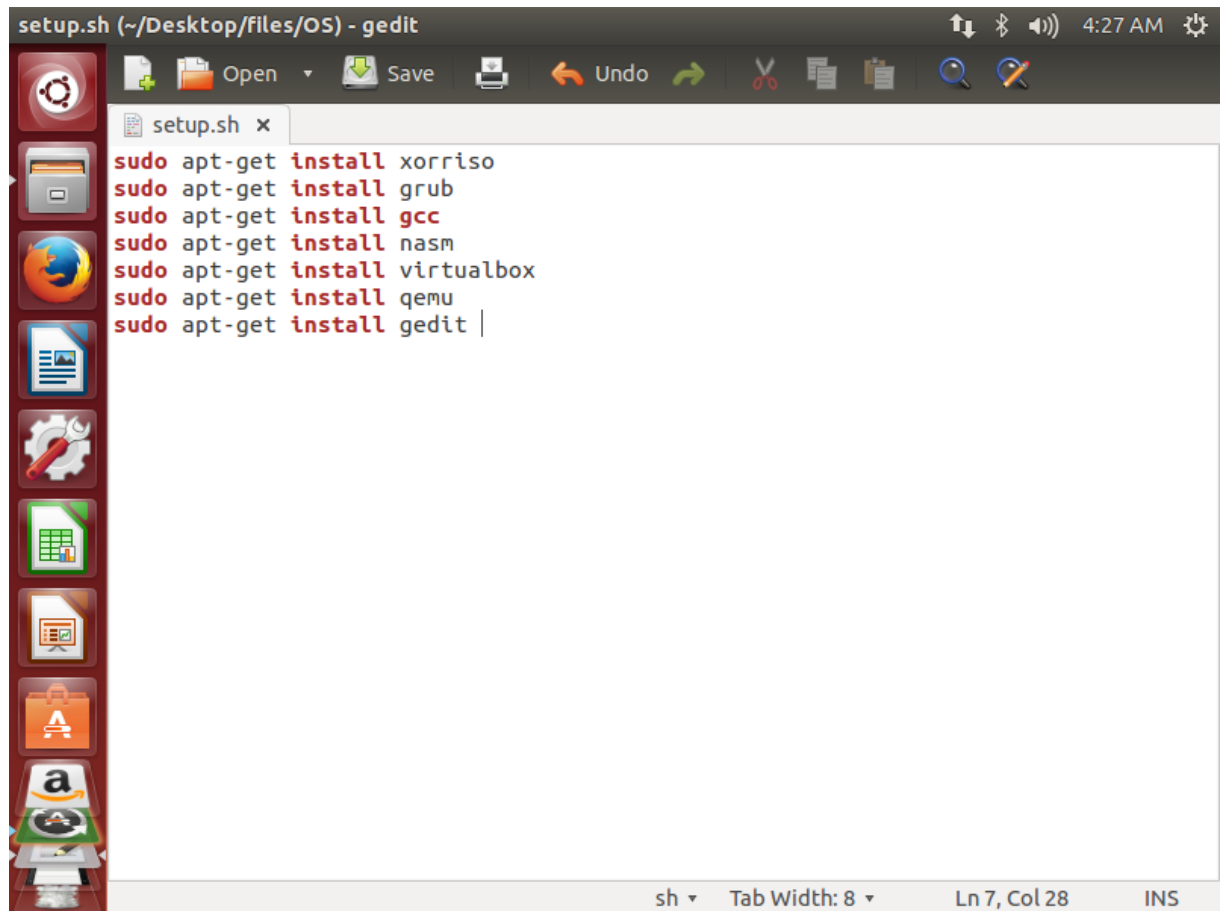






Now we are ready with Ubuntu as our working operating system and we need various setup's to develop our host operating system.
Snapshots below shows the setup.sh file consisting of various commands to setup the required environment.

3.2 Installing required packages



The screenshot shows a terminal window titled 'setup.sh (~/Desktop/files/OS) - gedit'. The window contains a list of commands to install various packages. The commands are as follows:

```
sudo apt-get install xorriso
sudo apt-get install grub
sudo apt-get install gcc
sudo apt-get install nasm
sudo apt-get install virtualbox
sudo apt-get install qemu
sudo apt-get install gedit |
```

The terminal window has a menu bar with 'Open', 'Save', 'Undo', and other standard editing functions. The status bar at the bottom indicates 'sh', 'Tab Width: 8', 'Ln 7, Col 28', and 'INS'.

To install packages or programs in your system, you first need to know the package name. If you don't know the package name is recommended you use a GUI (like Ubuntu Software Center or packagekit), search and install your program using it. If you know the package name then you only have to:

```
sudo apt-get install package_name
```

Replace `package_name` your package name. You can install several packages at once, just write them with space between them.

xorriso is a program which copies file objects from POSIX compliant filesystems into Rock Ridge enhanced ISO 9660 filesystems and allows session-wise manipulation of such filesystems. It can load the management information of existing ISO images and it writes the session results to optical media or to filesystem objects. Vice versa **xorriso** is able to copy file objects out of ISO 9660 filesystems.

A special property of **xorriso** is that it needs neither an external ISO 9660 formatter program nor an external burn program for CD, DVD or BD but rather incorporates the libraries of libburnia-project.org.

Rock Ridge is the name of a set of additional information which enhance an ISO 9660 filesystem so that it can represent a POSIX compliant filesystem with ownership, access permissions, symbolic links, and other attributes. This is what **xorriso** uses for a decent representation of the disk files within the ISO image. **xorriso** produces Rock Ridge information by default. It is strongly discouraged to disable this feature.

xorriso is not named "porriso" because POSIX only guarantees 14 characters of filename length. It is the X/Open System Interface standard XSI which demands a file name length of up to 255 characters and paths of up to 1024 characters. Rock Ridge fulfills this demand.

Operates on an existing ISO image or creates a new one.

Copies files from disk file system into the ISO image.

Copies files from ISO image to disk file system (see osirrox).

Renames or deletes file objects in the ISO image.

Changes file properties in the ISO image.

Updates ISO subtrees incrementally to match given disk subtrees.

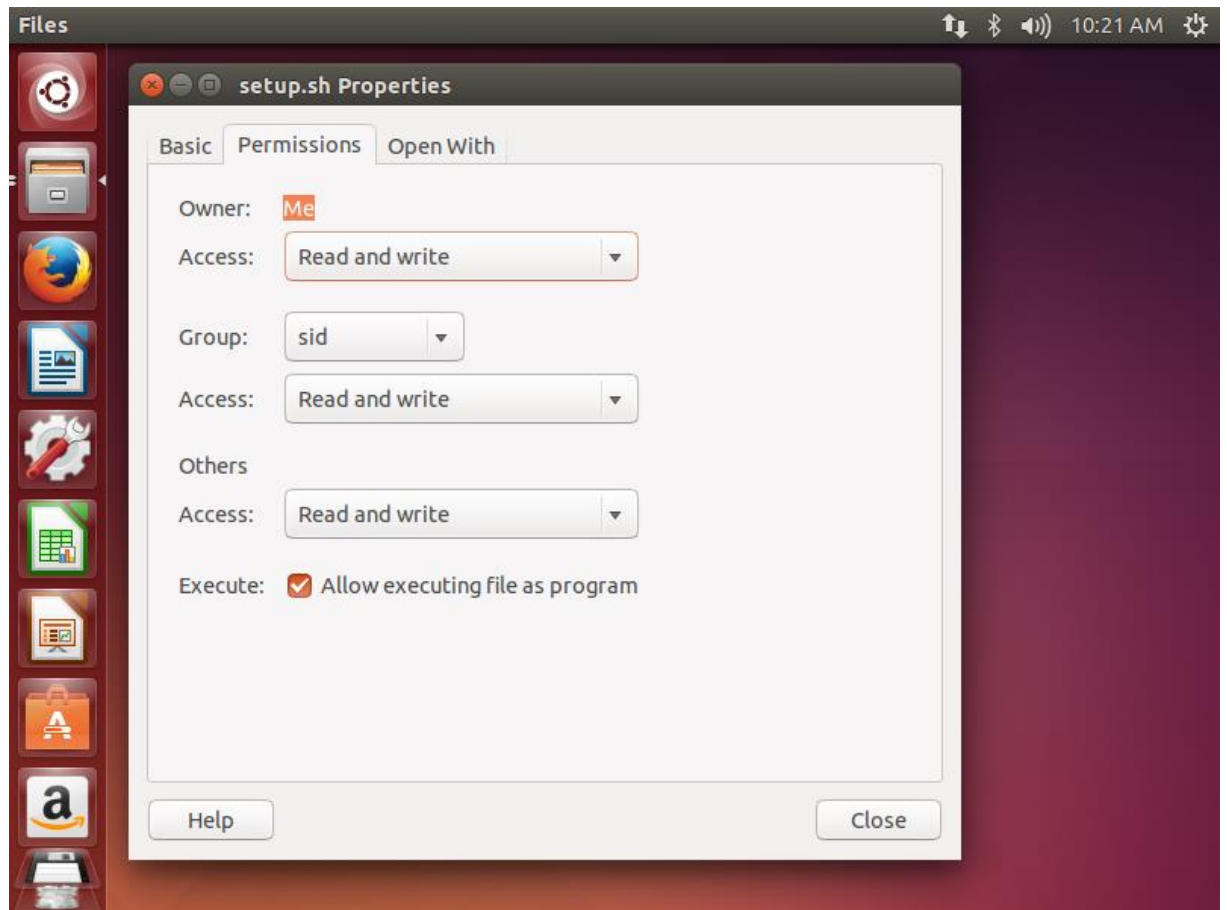
gedit is the default text editor of the GNOME desktop environment and part of the GNOME Core Applications. Designed as a general-purpose text editor, gedit emphasizes simplicity and ease of use, with a clean and simple GUI, according to the philosophy of the GNOME project. It includes tools for editing source code and structured text such as markup languages

It is free and open-source software subject to the requirements of the GNU General Public License version 2 or later.

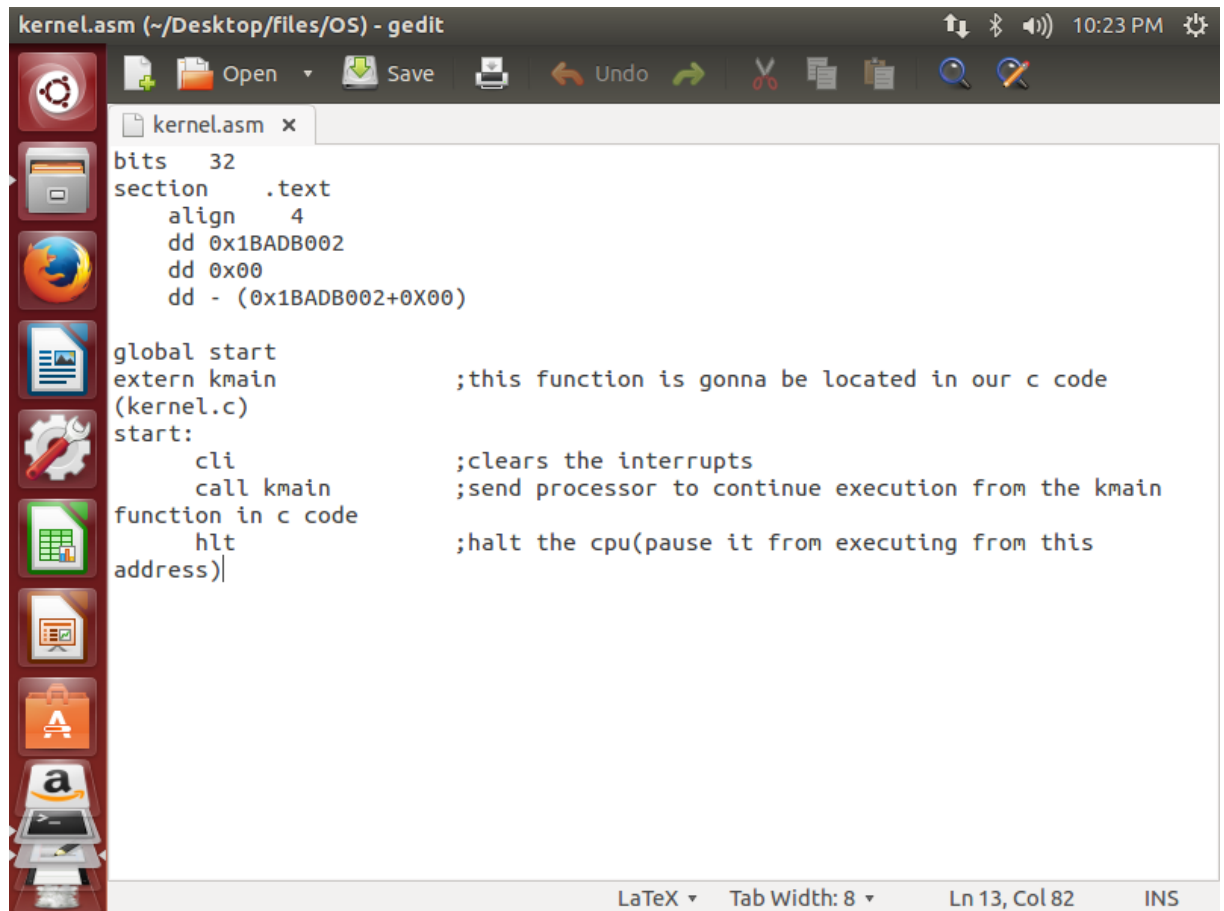
gedit is also available for Mac OS X and Microsoft Windows.

To allow executing file run as a program in linux

Change the permission for owner, group and others. Also allow the “executing file run as a program” option.



3.3 Kernel.asm:- Assembly language programming for kernel



```
kernel.asm (~/Desktop/files/OS) - gedit
bits 32
section .text
    align 4
    dd 0x1BADB002
    dd 0x00
    dd - (0x1BADB002+0x00)

global start
extern kmain
start:
    cli
    call kmain
    hlt
function in c code
address)|
;this function is gonna be located in our c code
;clears the interrupts
;send processor to continue execution from the kmain
;halt the cpu(pause it from executing from this
```

```
bits 32
section .text
;multiboot spec
align 4
dd 0x1BADB002 ;magic
dd 0x00 ;flags
dd - (0x1BADB002 + 0x00) ;checksum. m+f+c should be zero
```

The first instruction `bits 32` is not an x86 assembly instruction. It's a directive to the NASM assembler that specifies it should generate code to run on a processor operating in 32 bit mode. It is not mandatorily required in our example, however is included here as it's good practice to be explicit.

The second line begins the text section. This is where we put all our code.

`global` is another NASM directive to set symbols from source code as global. By doing so, the linker knows where the symbol `start` is; which happens to be our entry point.

`kmain` is our function that will be defined in our `kernel.c` file. `extern` declares that the function is declared elsewhere.

Then, we have the `start` function, which calls the `kmain` function and halts the CPU using the `hlt` instruction. Interrupts can awake the CPU from an `hlt` instruction. So we disable interrupts beforehand using `cli` instruction. `cli` is short for clear-interrupts.

We should ideally set aside some memory for the stack and point the stack pointer (`esp`) to it. However, it seems like GRUB does this for us and the stack pointer is already set at this point. But, just to be sure, we will allocate some space in the BSS section and point the stack pointer to the beginning of the allocated memory. We use the `resb` instruction which reserves memory given in bytes. After it, a label is left which will point to the edge of the reserved piece of memory. Just before the `kmain` is called, the stack pointer (`esp`) is made to point to this space using the `mov` instruction.

`ALIGN X`

The `ALIGN` directive is accompanied by a number (X).
This number (X) must be a power of 2. That is 2, 4, 8, 16, and so on...

The directive allows you to enforce alignment of the instruction or data immediately after the directive, on a memory address that is a multiple of the value X.

The extra space, between the previous instruction/data and the one after the `ALIGN` directive, is padded with NULL instructions (or equivalent, such as `MOV EAX,EAX`) in the case of code segments, and NULLs in the case of data segments.

The number X, cannot not be greater than the default alignment of the segment in which the `ALIGN` directive is referenced. It must be less or equal to the default alignment of the segment. More on this to follow...

A. If you use the `.386` processor directive, and you haven't explicitly declared the default alignment value for a segment, the default segment alignment is of `DWORD` (4 bytes) size. Yeah, in this case, $X = 4$. You can then use the following values with the `ALIGN` directive: ($X=2$, $X=4$). Remember, X must be less or equal than the segment alignment.

B. If you use the `.486` processor directive and above, and you haven't explicitly declared the default alignment value for a segment, the default segment alignment is of `PARAGRAPH` (16 bytes) size. In this case, $X = 16$. You can then use the following values with the `ALIGN` directive: ($X=2$, $X=4$, $X=8$, $X=16$).

Here are the aliases for segment alignment values...

Align Type	Starting Address
<code>BYTE</code>	Next available byte address.
<code>WORD</code>	Next available word address (2 bytes per word).
<code>DWORD</code>	Next available double word address (4 bytes per double word).
<code>PARA</code>	Next available paragraph address (16 bytes per paragraph).
<code>PAGE</code>	Next available page address (256 bytes per page).

The processor supports the following data sizes –

- Word: a 2-byte data item
- Doubleword: a 4-byte (32 bit) data item
- Quadword: an 8-byte (64 bit) data item
- Paragraph: a 16-byte (128 bit) area
- Kilobyte: 1024 bytes
- Megabyte: 1,048,576 bytes

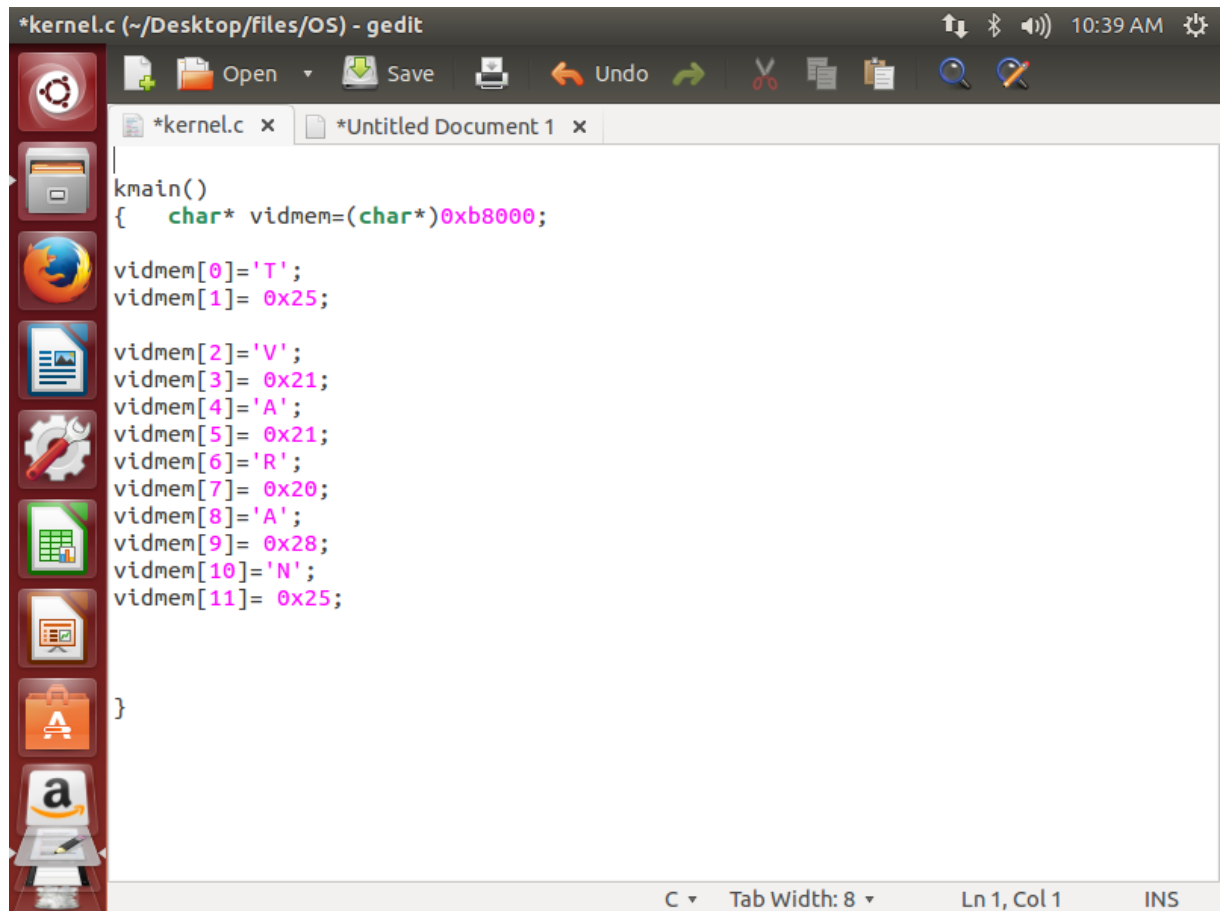
`dd - (0x1BADB002 + 0x00)` is actually saying add the two values `0x1BADB002` and `0x00` and then negate the result (similar to multiplying by -1) and yes `0x1BADB002` and `0x00 = 0x1BADB002`. Usually what you do is add the flags value to `0x1BADB002`. In this case flags is `0x00`. Depending on the functionality your bootloader needs it isn't `0x00`. I think they have done it for clarity. Although this isn't the preferred way.

The code appears to be 32-bit which is correct since a multiboot compliant bootloader will put the machine into protected mode set up with a default GDT, enabling the A20 gate and then jumping into the specified starting point of the ELF object. After the BIOS loads a boot sector it is in 16-bit real mode (or unreal mode), but a multiboot loader puts the machine in 32-bit protected mode before calling your kernel.

Multiboot compliant ELF objects like the one you are creating here have some rules to follow. A Multiboot image must contain a specific header that among other things declares the capabilities of the multiboot code, uses a magic value `0x1BADB002` so a multiboot loader can confirm what it is loading is multiboot compliant. There is a requirement that the header information fall within the first 8k of the executable and that it is aligned to a 4 byte boundary. Failure to use `align 4` before the header will likely cause the multiboot loader to see your executable as non-multiboot compliant

3.4 KERNEL.C :- Programming for kernel of operating system

The snapshot below consists of programming of kernel for booting and printing a string with different colour of all its characters.



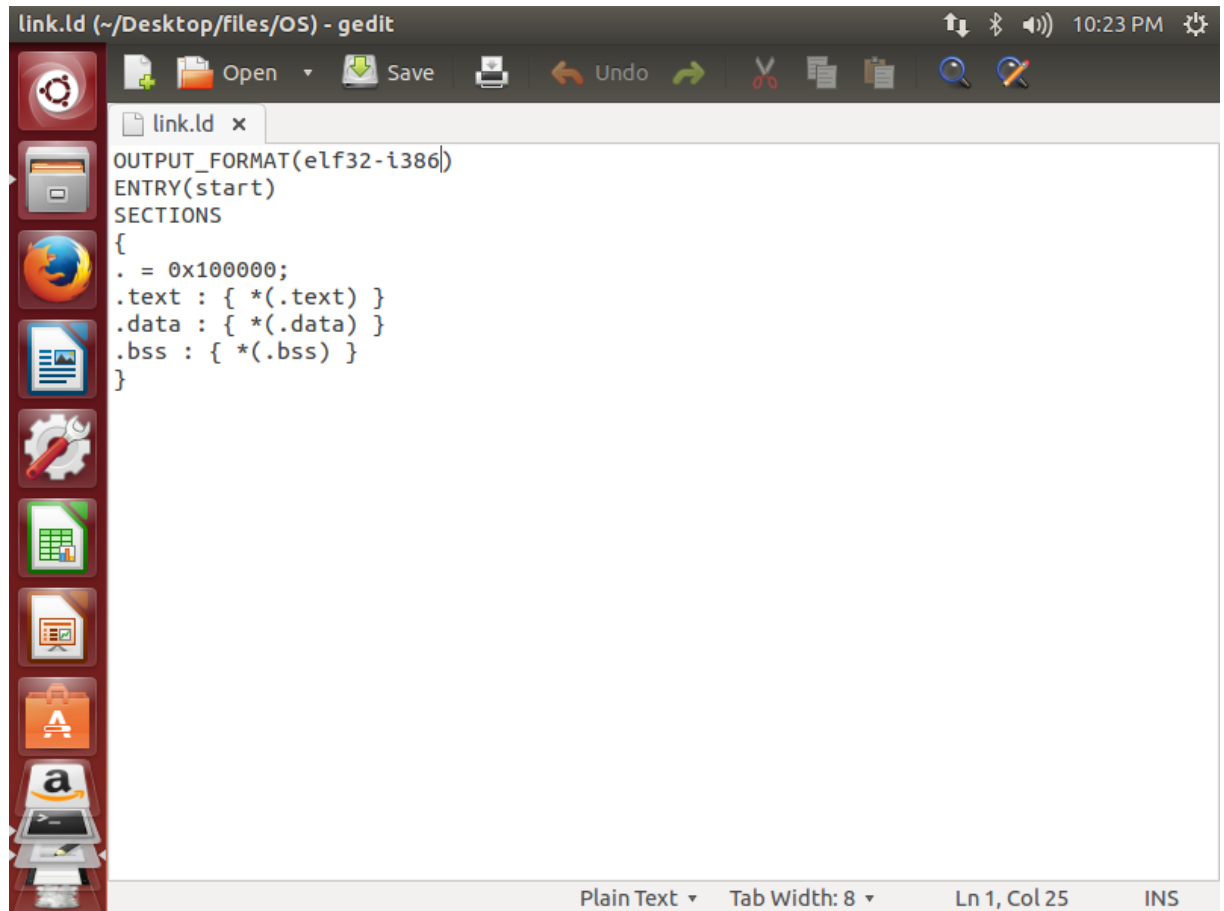
```

*kernel.c (~/Desktop/files/OS) - gedit
|
| kmain()
| {  char* vidmem=(char*)0xb8000;
|
| vidmem[0]='T';
| vidmem[1]= 0x25;
|
| vidmem[2]='V';
| vidmem[3]= 0x21;
| vidmem[4]='A';
| vidmem[5]= 0x21;
| vidmem[6]='R';
| vidmem[7]= 0x20;
| vidmem[8]='A';
| vidmem[9]= 0x28;
| vidmem[10]='N';
| vidmem[11]= 0x25;
|
| }
  
```

Vidmem (video memory) is a character pointer variable which is used to print character with the option of defining its font color as well as background color.

If vidmem[i] defines the character to be printed then vidmem[i+1] defines its font colour.

3.5 LINK.LD -- it is used for linking and loading operations between different object files.



An assembly program can be divided into three sections –

- The **data** section,
- The **bss** section, and
- The **text** section.

The *data* Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.

The syntax for declaring data section is –

```
section.data
```

The *bss* Section

The **bss** section is used for declaring variables. The syntax for declaring bss section is –

```
section.bss
```

The *text* section

The **text** section is used for keeping the actual code. This section must begin with the declaration **global _start**, which tells the kernel where the program execution begins.

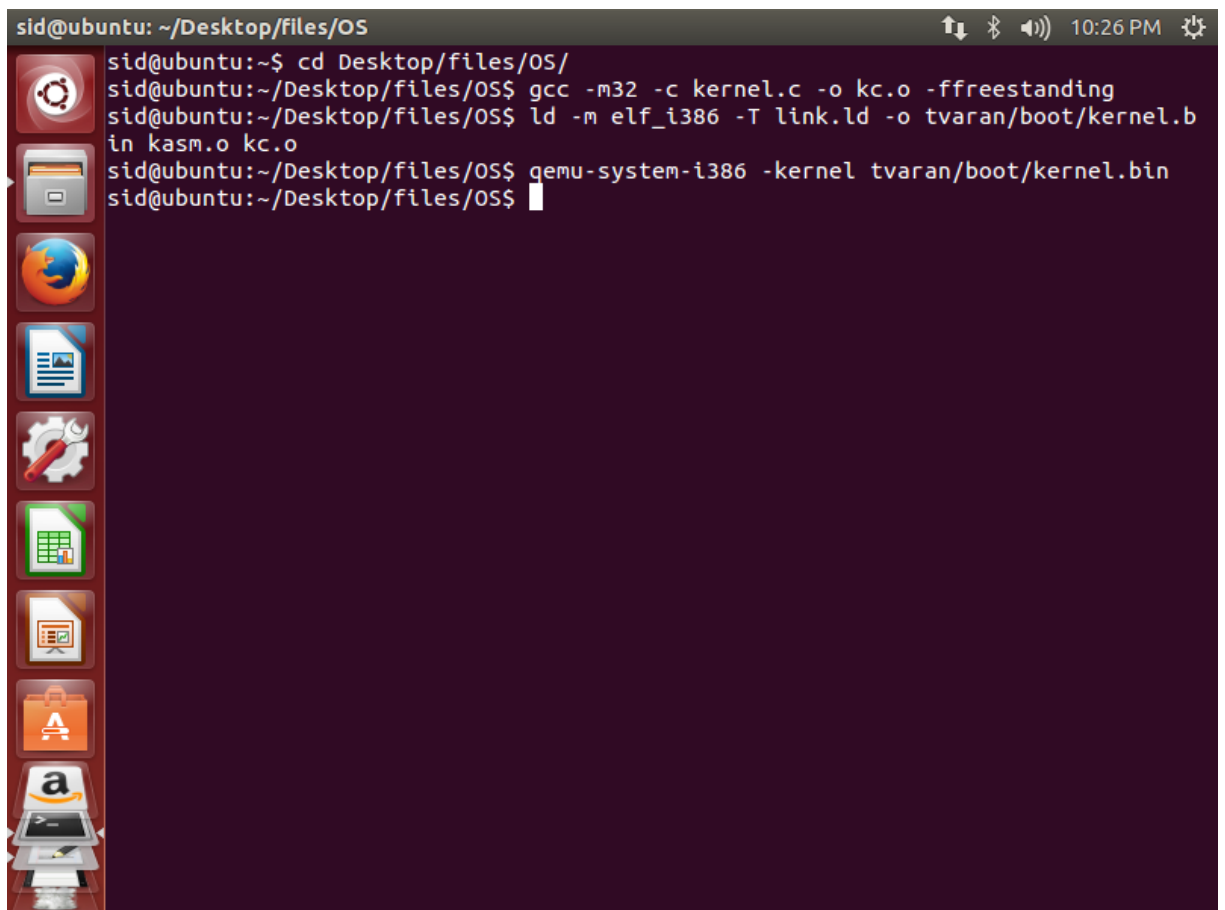
The syntax for declaring text section is –

```
section.text
global _start
_start:
```

Comments

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like –

```
; This program displays a message on screen
```



The screenshot shows a terminal window titled 'sid@ubuntu: ~/Desktop/files/OS'. The user has navigated to the directory ~/Desktop/files/OS. The terminal shows the following commands and output:

```
sid@ubuntu:~$ cd Desktop/files/OS/
sid@ubuntu:~/Desktop/files/OS$ gcc -m32 -c kernel.c -o kc.o -ffreestanding
sid@ubuntu:~/Desktop/files/OS$ ld -m elf_i386 -T link.ld -o tvaran/boot/kernel.b in kasm.o kc.o
sid@ubuntu:~/Desktop/files/OS$ qemu-system-i386 -kernel tvaran/boot/kernel.bin
sid@ubuntu:~/Desktop/files/OS$
```

The terminal window has a sidebar with various application icons including a file manager, web browser, and office applications. The top status bar shows the time as 10:26 PM.

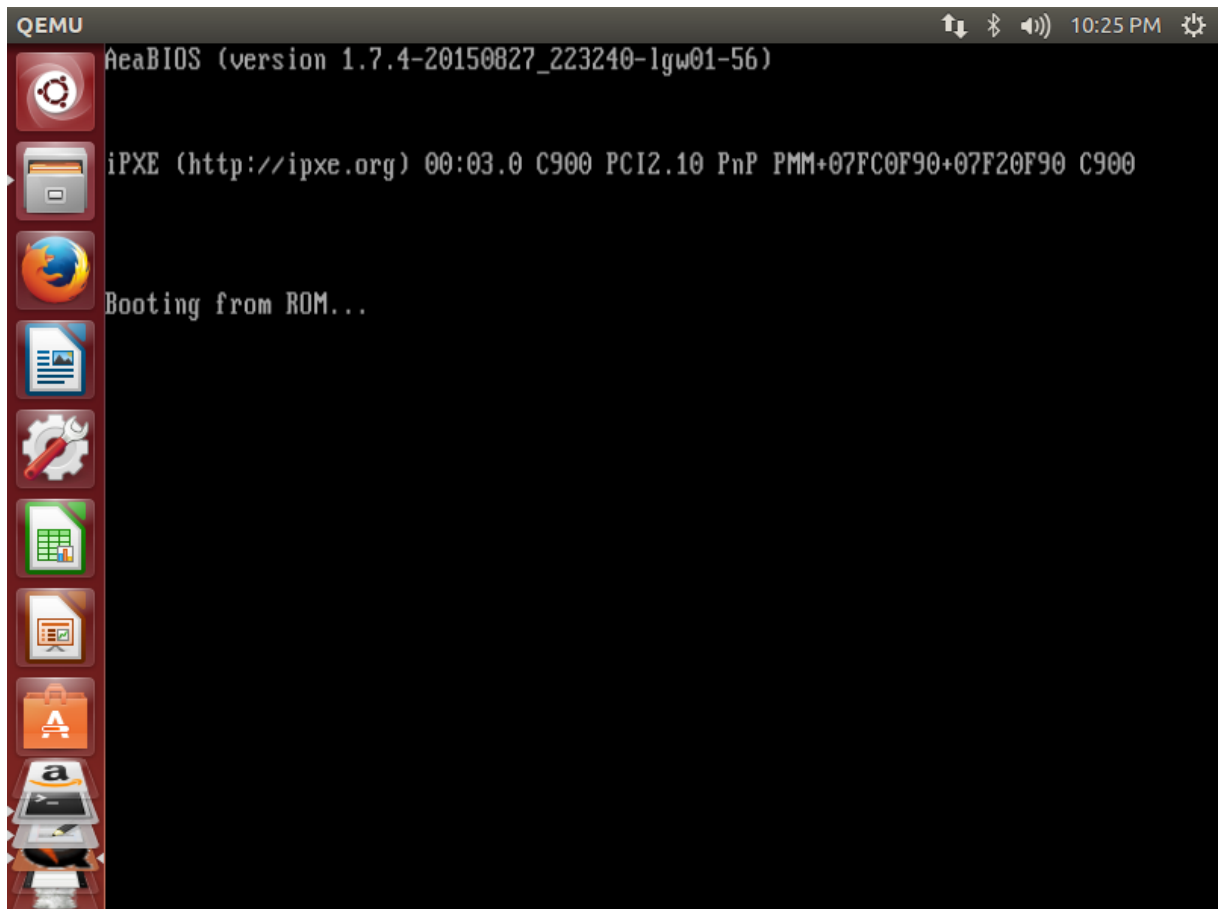
Command 1 –it is used to reach to the destination folder where the system and other target files are present.

Command 2-it is used for compiling and creating object code

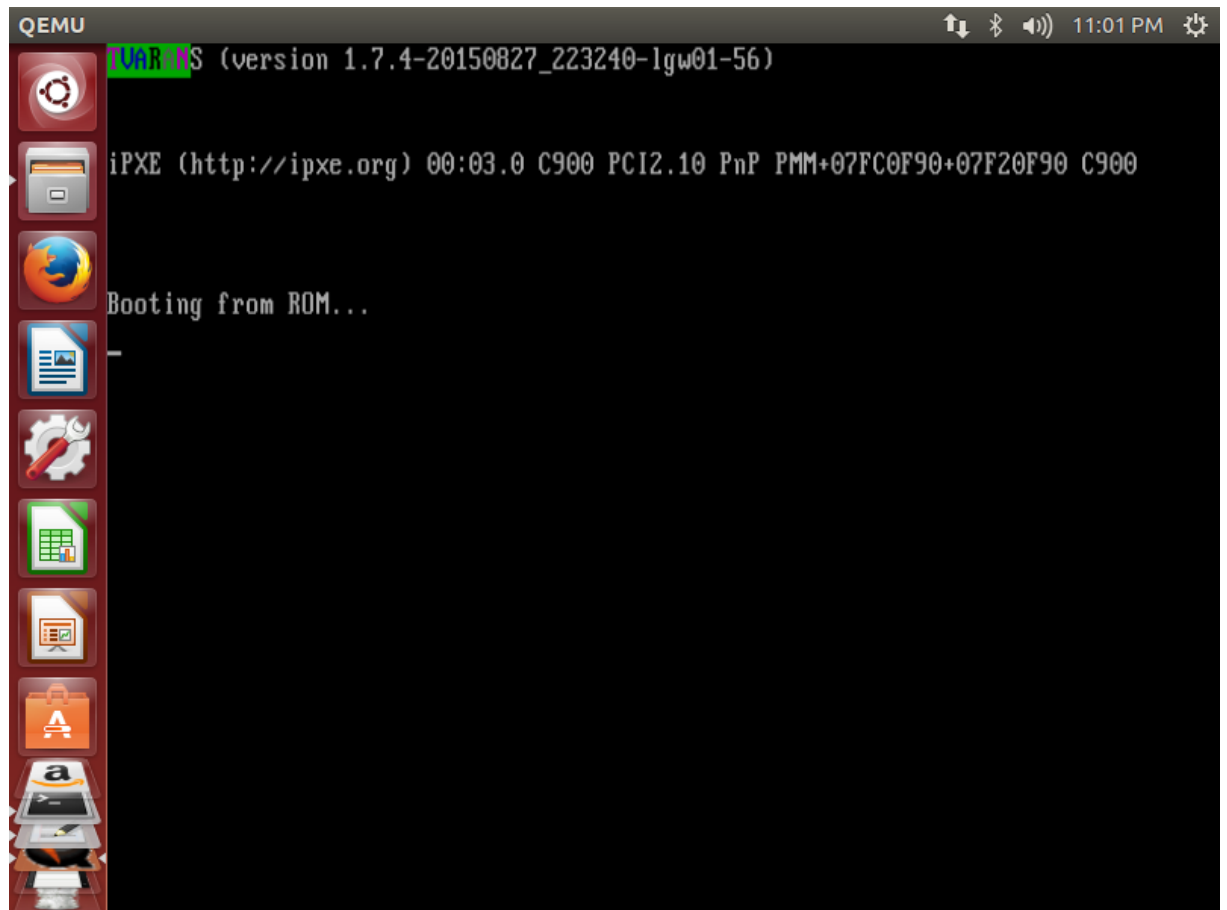
Command 3-it performs function of linking two or more object files from a specified folder and targeting the resultant file to the desired folder.

Command 4-‘qemu’ our output monitor displays the result of file created in previous step.

Snapshot below shows the process of booting in output window QEMU



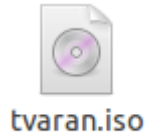
Snapshot below shows booting and printing of a colored string with each character being assigned a different color using vidmem.



3.6 Modified Build.sh

In the snapshot below-

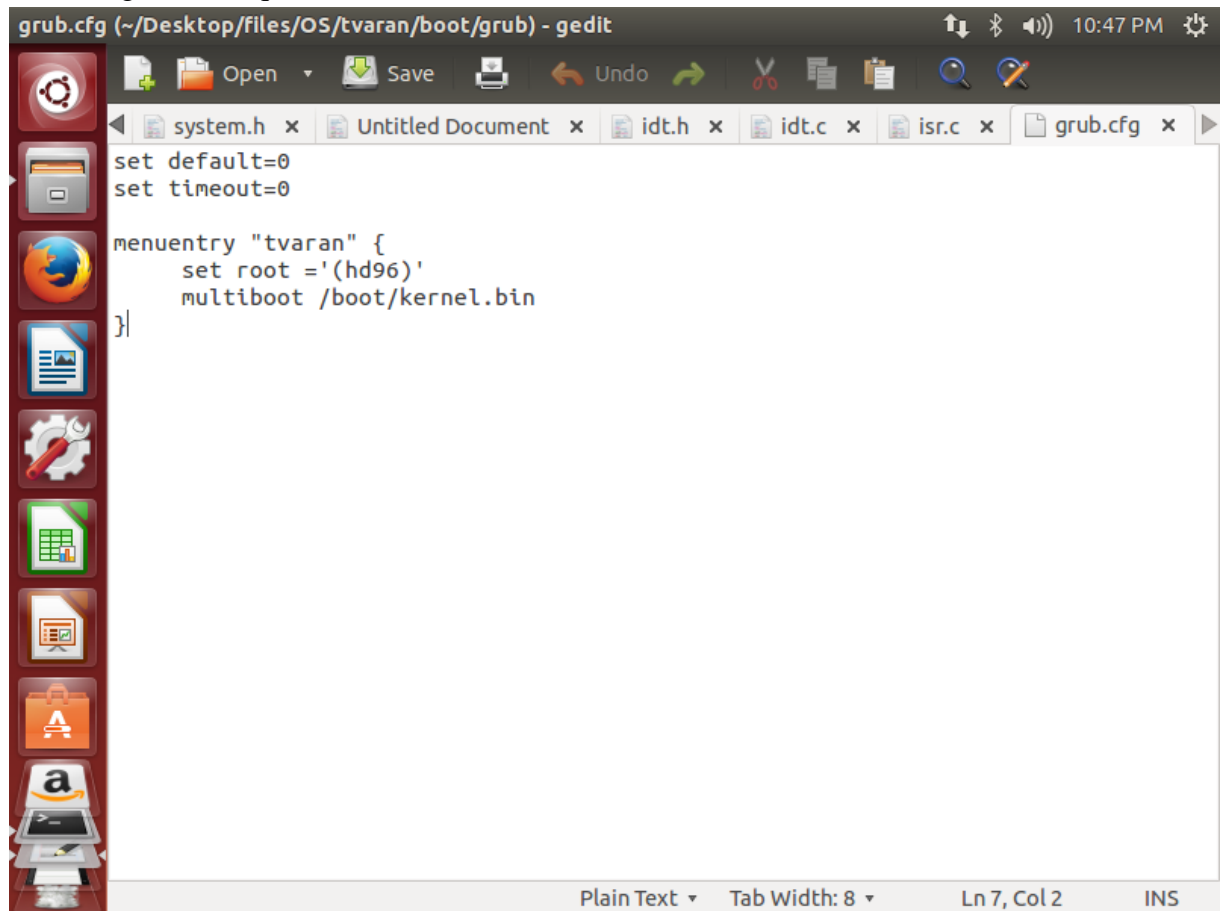
1. NASM is used for assembling and creating various object files.
2. GCC compiler is used for creating creation of object files.
3. “grub-mk rescue -o tvaran.iso tvaran/” this command is used to create iso file of name tvaran.



```
*build.sh (~/Desktop/files/OS) - gedit
nasm -f elf32 kernel.asm -o kasm.o
gcc -m32 -c kernel.c -o kc.o -ffreestanding
gcc -m32 -c include/types.h -o obj/type.o -ffreestanding
gcc -m32 -c include/system.h -o obj/system.o -ffreestanding
gcc -m32 -c include/isr.c -o obj/isr.o -ffreestanding
gcc -m32 -c include/idt.c -o obj/idt.o -ffreestanding
gcc -m32 -c include/util.c -o obj/util.o -ffreestanding
gcc -m32 -c include/string.h -o obj/string.o -ffreestanding
gcc -m32 -c include/screen.h -o obj/screen.o -ffreestanding
gcc -m32 -c include/kb.h -o obj/kb.o -ffreestanding
ld -m elf_i386 -T link.ld -o tvaran/boot/kernel.bin kasm.o kc.o
qemu-system-i386 -kernel kernel.bin
grub-mk rescue -o tvaran.iso tvaran/
```

3.7 GRUB.cfg-

Snapshot below is used to change the default factory settings and to customize our OS according to our requirements.



```
grub.cfg (~/Desktop/files/OS/tvaran/boot/grub) - gedit
set default=0
set timeout=0

menuentry "tvaran" {
    set root ='(hd96)'
    multiboot /boot/kernel.bin
}
```

In computing, **configuration files**, or **config files** configure the parameters and initial settings for some computer programs. They are used for user applications, server processes and operating system settings. The files are often written with ASCII encoding (rarely UTF-8) and line-oriented, with lines terminated by a newline or carriage return/line feed pair (CR LF), depending on the operating system. In operating systems that categorize files by extensions, software-dependent extensions .cnf, .conf, and .cfg are often used.

Some applications provide tools to create, modify, and verify the syntax of their configuration files; these sometimes have graphical interfaces. For other programs, system administrators may be expected to create and modify files by hand using a text editor. For server processes and operating-system settings, there is often no standard tool, but operating systems may provide their own graphical interfaces such as YaST or debconf.

CFG is a configuration file format used for storing settings. CFG files are created by many programs to store information and settings that differ from the factory defaults. CFG files usually appear as text documents, and can be opened by word processors though it is not recommended.

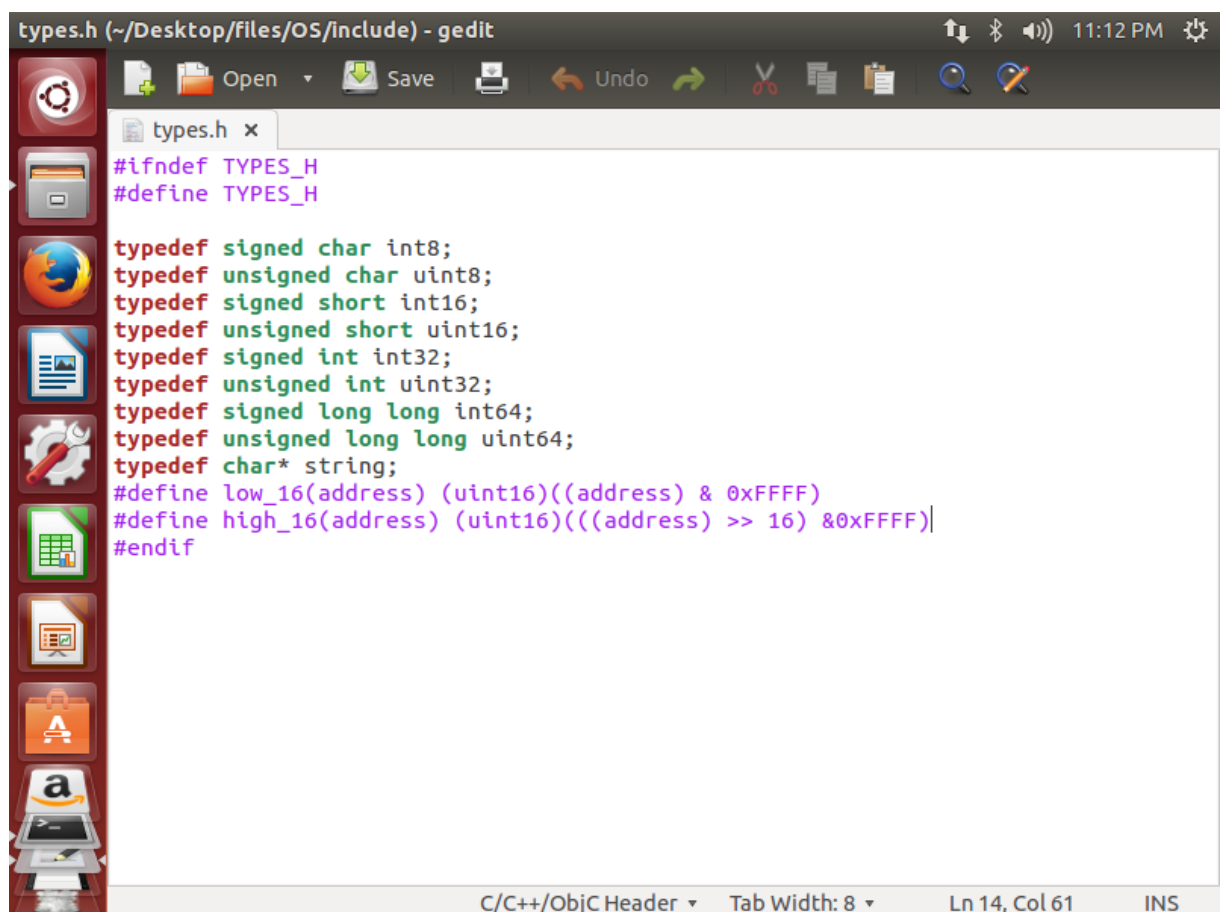
The editing or deletion of a .CFG file outside of the program it was created by will affect the settings in the program and how it runs. If one wishes to set a program back to the factory settings, however, moving or deleting the .CFG files will do so.

First of all, GRUB requires that the device name be enclosed with ‘(’ and ‘)’. The ‘fd’ part means that it is a floppy disk. The number ‘0’ is the drive number, which is counted from *zero*. This expression means that GRUB will use the whole floppy disk.

```
(hd0,msdos2)
```

Here, ‘hd’ means it is a hard disk drive. The first integer ‘0’ indicates the drive number, that is, the first hard disk, the string ‘msdos’ indicates the partition scheme, while the second integer, ‘2’, indicates the partition number (or the PC slice number in the BSD terminology). The partition numbers are counted from *one*, not from zero (as was the case in previous versions of GRUB). This expression means the second partition of the first hard disk drive. In this case, GRUB uses one partition of the disk, instead of the whole disk.

Timeout is used to set the timing for start of booting process. Here is it set to zero thus booting starts without any time delay.



```
types.h (~/Desktop/files/OS/include) - gedit
types.h x
#ifndef TYPES_H
#define TYPES_H

typedef signed char int8;
typedef unsigned char uint8;
typedef signed short int16;
typedef unsigned short uint16;
typedef signed int int32;
typedef unsigned int uint32;
typedef signed long long int64;
typedef unsigned long long uint64;
typedef char* string;
#define low_16(address) (uint16)((address) & 0xFFFF)
#define high_16(address) (uint16)(((address) >> 16) & 0xFFFF)
#endif

C/C++/ObjC Header Tab Width: 8 Ln 14, Col 61 INS
```

Those included macros are called Include guards.

```
#ifndef <token>
/* code */
#else
/* code to include if the token is defined */
#endif
```

`#ifndef` checks whether the given token has been `#defined` earlier in the file or in an included file; if not, it includes the code between it and the closing `#else` or, if no `#else` is present, `#endif` statement. `#ifndef` is often used to make header files idempotent by defining a token once the file has been included and checking that the token was not set at the top of that file.

```
#ifndef _INCL_GUARD
#define _INCL_GUARD
#endif
```

Typedef is used to rename the data types.

For example, signed char is renamed as `int8` using typedef. Similarly other data types are renamed. Its renamed according to size a data type has- char is of size 8bits and integer is of size 16bit.

`#define low_16(address) (uint16)((address) & 0xFFFF)` - it sets the lower 16bits for the given address.

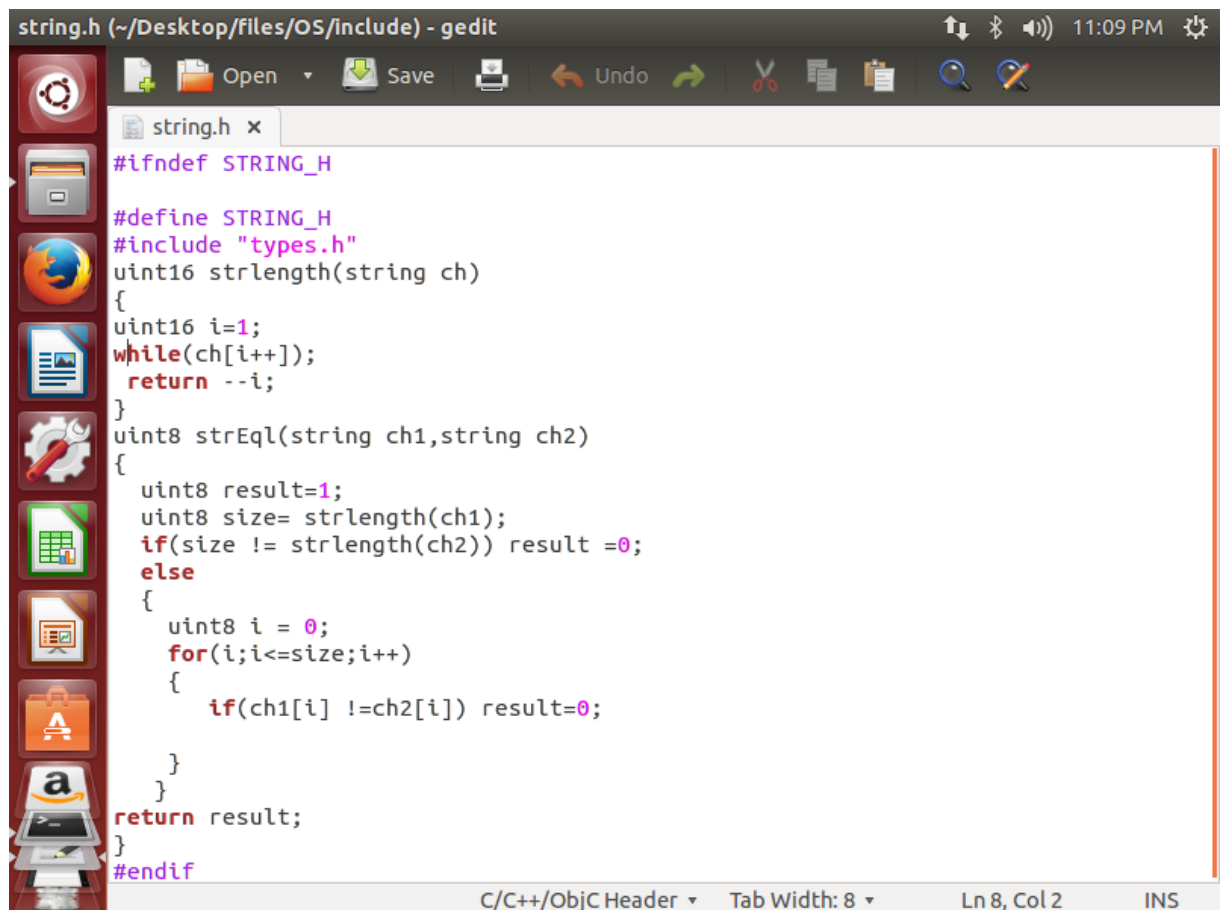
`#define high_16(address) (uint16)(((address) >> 16) & 0xFFFF)` - it sets the upper 16bits of for the given address.

3.8 Building Header files

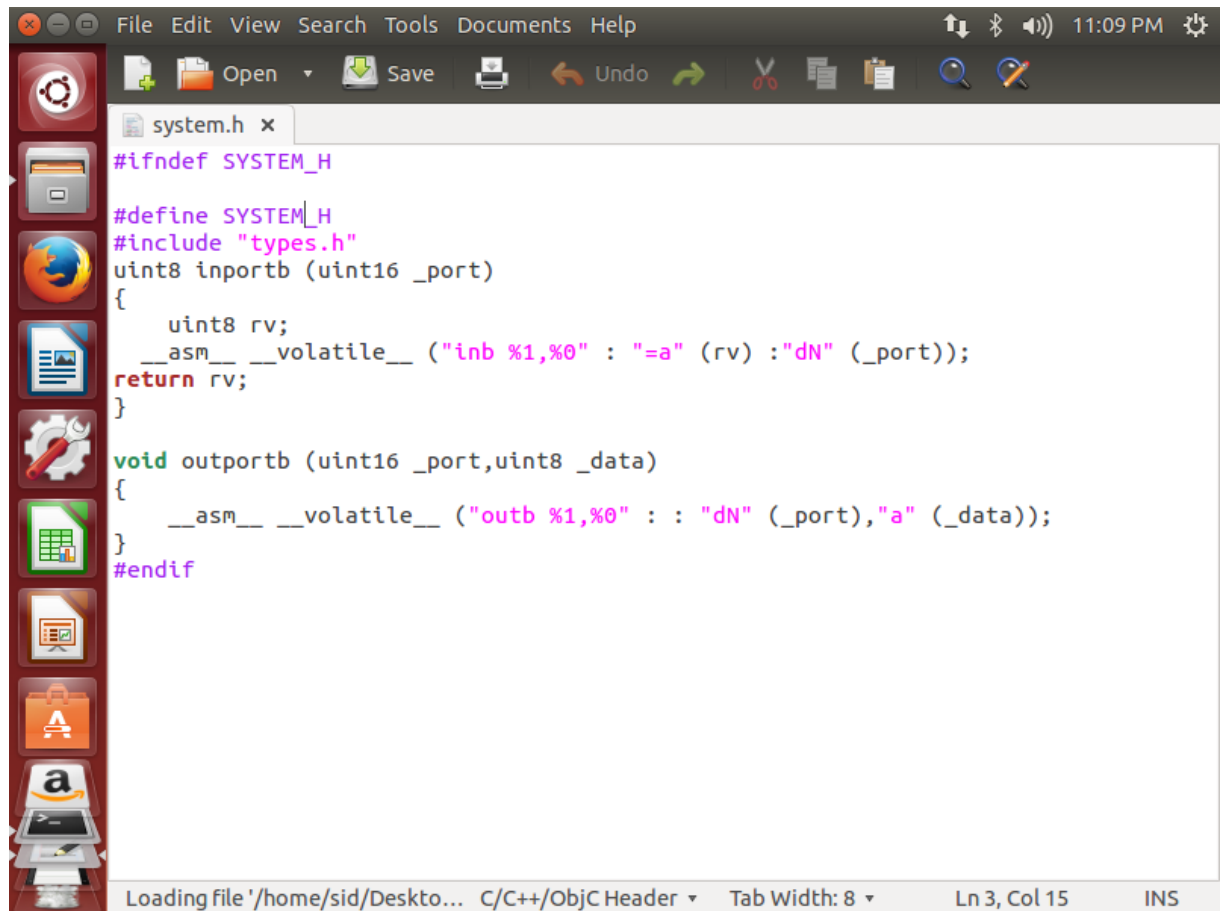
STRING.H- this consists of functions which works over string.

For instance- `strlen()`-this function calculates the length of string.

`strcmp()`-this function checks whether two given strings are equal or not.



```
string.h (~/Desktop/files/OS/include) - gedit
string.h x
#ifndef STRING_H
#define STRING_H
#include "types.h"
uint16 strlen(string ch)
{
    uint16 i=1;
    while(ch[i++]);
    return --i;
}
uint8 strcmp(string ch1,string ch2)
{
    uint8 result=1;
    uint8 size= strlen(ch1);
    if(size != strlen(ch2)) result =0;
    else
    {
        uint8 i = 0;
        for(i;i<=size;i++)
        {
            if(ch1[i] !=ch2[i]) result=0;
        }
    }
    return result;
}
#endif
C/C++/ObjC Header ▾ Tab Width: 8 ▾ Ln 8, Col 2 INS
```



```

system.h x
#ifndef SYSTEM_H
#define SYSTEM_H
#include "types.h"
uint8 inportb (uint16 _port)
{
    uint8 rv;
    __asm__ __volatile__ ("inb %1,%0" : "=a" (rv) : "dN" (_port));
    return rv;
}

void outportb (uint16 _port,uint8 _data)
{
    __asm__ __volatile__ ("outb %1,%0" : : "dN" (_port),"a" (_data));
}
#endif
    
```

Loading file '/home/sid/Desktop... C/C++/ObjC Header ▾ Tab Width: 8 ▾ Ln 3, Col 15 INS

This is the syntax for calling `asm()` in your C/C++ code:

```

asm ( assembler template
    : output operands                (optional)
    : input operands                 (optional)
    : clobbered registers list       (optional)
    );
    
```

`__asm__` is a gcc extension of permitting assembly language statements to be entered nested within your C code - used here for its property of being able to specify side effects that prevent the compiler from performing certain types of optimizations (which in this case might end up generating incorrect code).

`__volatile__` is required to ensure that the **asm** statement itself is not reordered with any other volatile accesses any (a guarantee in the C language).

`memory` is an instruction to GCC that (sort of) says that the inline asm sequence has side effects on global memory, and hence not just effects on local variables need to be taken into account. If our assembly statement must execute where we put it, (i.e. must not be moved out of a loop as an optimization), put the keyword `volatile` after `asm` and before the `()`'s. So to keep it from moving, deleting and all, we declare it as

```
asm volatile ( ... : ... : ... : ... );
```

Use `__volatile__` when we have to be very much careful.

If our assembly is just for doing some calculations and doesn't have any side effects, it's better not to use the keyword `volatile`.

SCREEN.H- it consists of functions that modifies the output and cursor positions.

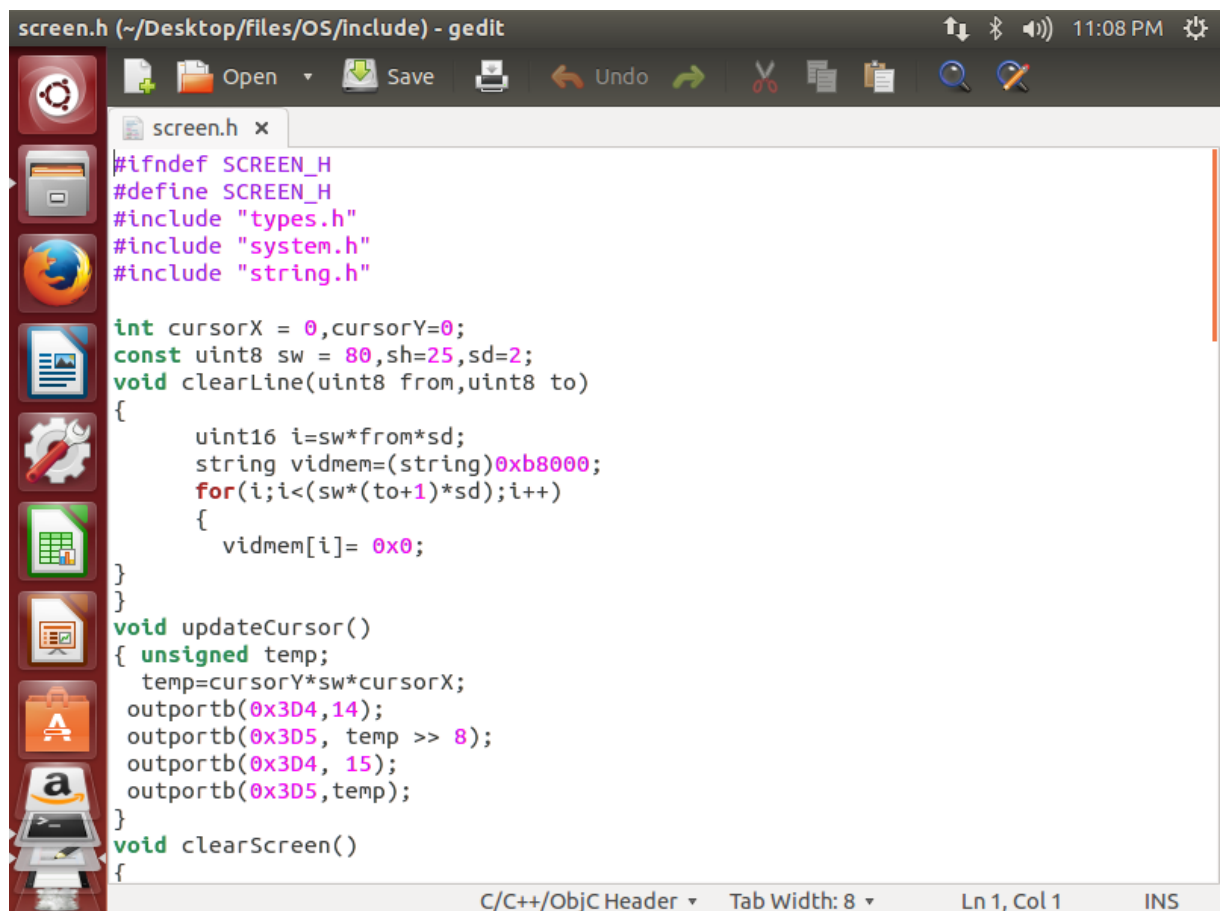
clearLine()-this function is used to remove some part of output.it takes two arguments 'from' and 'to' in order to decide from which point to point clearLine function has to act.

updateCursor()-it updates the position of cursor according to requirement.

clearScreen()-it is used to clear the screen.it just clears one row at a time. clearLine() removes pixel by pixel however clearScreen() clears one row at a time.

scrollUp()-this function just shift the contents by some row according to requirement.

Print()- this function is used to display/print character on the output screen.



```
screen.h (~/Desktop/Files/OS/include) - gedit
#ifndef SCREEN_H
#define SCREEN_H
#include "types.h"
#include "system.h"
#include "string.h"

int cursorX = 0, cursorY = 0;
const uint8 sw = 80, sh = 25, sd = 2;
void clearLine(uint8 from, uint8 to)
{
    uint16 i = sw * from * sd;
    string vidmem = (string) 0xb8000;
    for (i; i < (sw * (to + 1) * sd); i++)
    {
        vidmem[i] = 0x0;
    }
}

void updateCursor()
{
    unsigned temp;
    temp = cursorY * sw * cursorX;
    outportb(0x3D4, 14);
    outportb(0x3D5, temp >> 8);
    outportb(0x3D4, 15);
    outportb(0x3D5, temp);
}

void clearScreen()
{
}
```

Full code of screen.h

```
#ifndef SCREEN_H
#define SCREEN_H
#include "types.h"
#include "system.h"
#include "string.h"
```

```

int cursorX = 0,cursorY=0;
const uint8 sw = 80,sh=25,sd=2;
void clearLine(uint8 from,uint8 to)
{
    uint16 i=sw*from*sd;
    string vidmem=(string)0xb8000;
    for(i; i<(sw*(to+1)*sd);i++)
    {
        vidmem[i]= 0x33;
    }
}
void updateCursor()
{ unsigned temp;
  temp=cursorY*sw*cursorX;
  outportb(0x3D4,14);
  outportb(0x3D5, temp >> 8);
  outportb(0x3D4, 15);
  outportb(0x3D5,temp);
}
void clearScreen()
{
    clearLine(0,sh-1);
    cursorX=0;
    cursorY=0;
    updateCursor();
}
void scrollUp(uint8 lineNumber)
{ string vidmem= (string)0xb8000;
  uint16 i=0;
  for(i; i<sw*(sh-1)*2; i++)
  {
      vidmem[i]= vidmem[i+sw*2*lineNumber];
  }
  clearLine(sh-1-lineNumber,sh-1);
  if((cursorY - lineNumber)<0)
  { cursorY=0;
    cursorX=0;
  }
  else
  {
      cursorY-=lineNumber;
  }
  updateCursor();
}

```

```

void newLineCheck()
{
if(cursorY>=sh-1)
{
scrollUp(1);
}
}
void printch(char c)
{
string vidmem=(string) 0xb8000;
switch(c)
{
case (0x08):
if(cursorX>0)
{ cursorX--;
vidmem[(cursorY*sw+cursorX)*sd]=0x00;
}
break;
case (0x09):
cursorX=(cursorX+8) & ~(8 - 1);
break;
case ('\r'):
cursorX=0;
break;
case ('\n'):
cursorX=0;
cursorY++;
break;
default:
vidmem [((cursorY * sw +cursorX))*sd]=c;
vidmem [((cursorY * sw +cursorX))*sd+1]=0x3F;
cursorX++;
break;
}
if(cursorX >= sw)
{
cursorX = 0;
cursorY++;
}
newLineCheck();
updateCursor();
}
void print (string ch)
{

```

```
uint16 i=0;
uint8 length = strlen(ch)-1;
for(i;i<length;i++)
{
    printch(ch[i]) ;
}
}
#endif
```

Assuming that you are in protected mode and not using the BIOS to write text to screen, you will have write directly to "video" memory.

This is quite easy. The text screen video memory for colour monitors resides at 0xB8000, and for monochrome monitors it is at address 0xB0000 (see Detecting Colour and Monochrome Monitors for more information).

```
void move_csr(void)
{
    unsigned temp;

    /* The equation for finding the index in a linear
    * chunk of memory can be represented by:
    * Index = [(y * width) + x] */
    temp = csr_y * 80 + csr_x;

    /* This sends a command to indices 14 and 15 in the
    * CRT Control Register of the VGA controller. These
    * are the high and low bytes of the index that show
    * where the hardware cursor is to be 'blinking'. To
    * learn more, you should look up some VGA specific
    * programming documents. A great start to graphics:
    * http://www.brackeen.com/home/vga */
    outportb(0x3D4, 14);
    outportb(0x3D5, temp >> 8);
    outportb(0x3D4, 15);
    outportb(0x3D5, temp);
}
```

the base port (here assumed to be 0x3D4) should be read from the BIOS data area

```
unsigned char cursor_xpos = 0, cursor_ypos = 0;
// Global variables specifying cursor position
```

```
void textmode_updatecursor(void)
```

```
{
    static unsigned char xpos_store = 0;
    static unsigned char ypos_store = 0;
    // static variables so we can tell if the cursor has moved, no point updating a cursor
    which doesn't need to change

    if((xpos_store != cursor_xpos) || (ypos_store != cursor_ypos)) { // if cursor has moved
        unsigned int temp = (cursor_ypos * 80) + cursor_xpos;
        outportb(0x3D4, 14); // specify which register we want to write to (index 14)
        outportb(0x3D5, temp >> 8); // write the high order bits to index 14
        outportb(0x3D4, 15); // specify which register we want to write to (index 15)
        outportb(0x3D5, temp); // write the low order bits to index 15
        ypos_store = cursor_ypos;
        xpos_store = cursor_xpos; // save updated cursor position
    }
}

#define COLOR_BLACK 0x00
#define COLOR_BLUE 0x01
#define COLOR_GREEN 0x02
#define COLOR_CYAN 0x03
#define COLOR_RED 0x04
#define COLOR_MAGENTA 0x05
#define COLOR_BROWN 0x06
#define COLOR_LTGRAY 0x07
#define COLOR_DKGRAY 0x08
#define COLOR_LTBLUE 0x09
#define COLOR_LTGREEN 0x0A
#define COLOR_LTCYAN 0x0B
#define COLOR_LTRED 0x0C
#define COLOR_LTMAGENTA 0x0D
#define COLOR_LTBROWN 0x0E
#define COLOR_WHITE 0x0F
```

The *attribute* byte carries the *foreground color* in its lowest 4 bits and the *background color* in its highest 3 bits. The interpretation of bit #7 depends on how you (or the BIOS) configured the hardware (see VGA Resources for additional info).

For instance, using 0x00 as attribute byte means black-on-black (you'll see nothing). 0x07 is light grey-on-black (DOS default), 0x1F is white-on-blue (Win9x's blue-screen-of-death), 0x2a is for green-monochrome nostalgic.

For color video cards, you have 16kb of text video memory to use. Since 80x25 mode does not use all 16kb (80 x 25 x 2, 4000 bytes per screen), you have 8 display pages to use.

When you print to any other page than 0, it will *not* appear on screen until that page is *enabled* or *copied* into the page 0 memory space

KB.H-this file consists of codes in order to accept input by the user from keyboard.

Switch statement is used in order to having cases for all the buttons available on the keyboard, and according to cases that is the button tapped output is displayed by screen.h.

kb.h (~/.Desktop/files/OS/include) - gedit

```

#ifndef KB_H
#define KB_H
#include "screen.h"
#include "system.h"
#include "types.h"

string readStr()
{
    char buff;
    string buffstr;
    uint8 i = 0;
    uint8 reading = 1;
    while(reading)
    {
        if(inportb(0x64) & 0x1)
        {
            switch(inportb(0x60))
            {
                case 1:
                    printf((char)27); // Escape button
                    buffstr[i] = (char)27;
                    i++;
                    break;
                case 2:
                    printf('1');
                    buffstr[i] = '1';
                    i++;
            }
        }
    }
}

```

C/C++/ObjC Header ▾ Tab Width: 8 ▾ Ln 1, Col 1 INS

Complete code for keyboard KB.H –

```
#ifndef KB_H
#define KB_H
#include "screen.h"
#include "system.h"
#include "types.h"
```

```
string readStr()
{
    char buff;
    string buffstr;
    uint8 i = 0;
    uint8 reading = 1;
    while(reading)
    {
        if(inportb(0x64) & 0x1)
        {
            switch(inportb(0x60))
            {
```

```

case 1:
    printch((char)27);          // Escape button
    buffstr[i] = (char)27;
    i++;
    break;
case 2:
    printch('1');
    buffstr[i] ='1';
    i++;
    break;
case 3:
    printch('2');
    buffstr[i] ='2';
    i++;
    break;
case 4:
    printch('3');
    buffstr[i] ='3';
    i++;
    break;
case 5:
    printch('4');
    buffstr[i] ='4';
    i++;
    break;
case 6:
    printch('5');
    buffstr[i] ='5';
    i++;
    break;
case 7:
    printch('6');
    buffstr[i] ='6';
    i++;
    break;
case 8:
    printch('7');
    buffstr[i] ='7';
    i++;
    break;
case 9:
    printch('8');
    buffstr[i] ='8';
    i++;

```

```

        break;
    case 10:
        printch('9');
        buffstr[i] ='9';
        i++;
        break;
    case 11:
        printch('0');
        buffstr[i] ='0';
        i++;
        break;
    case 12:
        printch('-');
        buffstr[i] ='-';
        i++;
        break;
    case 13:
        printch('=');
        buffstr[i] ='=';
        i++;
        break;
    case 14:
        printch('\b');
        i--;
        buffstr[i] =0;
        break;
    case 15:
        printch('\t');    //Tab button
        buffstr[i] ='\t';
        i++;
        break;
    case 16:
        printch('q');
        buffstr[i] ='q';
        i++;
        break;
    case 17:
        printch('w');
        buffstr[i] ='w';
        i++;
        break;
    case 18:
        printch('e');
        buffstr[i] ='e';

```

```

        i++;
        break;
case 19:
    printch('r');
    buffstr[i] ='r';
    i++;
    break;
case 20:
    printch('t');
    buffstr[i] ='t';
    i++;
    break;
case 21:
    printch('y');
    buffstr[i] ='y';
    i++;
    break;
case 22:
    printch('u');
    buffstr[i] ='u';
    i++;
    break;
case 23:
    printch('i');
    buffstr[i] ='i';
    i++;
    break;
case 24:
    printch('o');
    buffstr[i] ='o';
    i++;
    break;
case 25:
    printch('p');
    buffstr[i] ='p';
    i++;
    break;
case 26:
    printch('{');
    buffstr[i] ='}';
    i++;
    break;
case 27:
    printch('}');

```

```

        buffstr[i] ='}';
        i++;
        break;
case 28:
    printch('\n');
    buffstr[i] ='\n';
    i++;
    reading = 0;
    break;
case 29:
    printch('q');      Left Control
    buffstr[i] ='q';
    i++;
    break; */
case 30:
    printch('a');
    buffstr[i] ='a';
    i++;
    break;
case 31:
    printch('s');
    buffstr[i] ='s';
    i++;
    break;
case 32:
    printch('d');
    buffstr[i] ='d';
    i++;
    break;
case 33:
    printch('f');
    buffstr[i] ='f';
    i++;
    break;
case 34:
    printch('g');
    buffstr[i] ='g';
    i++;
    break;
case 35:
    printch('h');
    buffstr[i] ='h';
    i++;
    break;

```

```

case 36:
    printch('j');
    buffstr[i] ='j';
    i++;
    break;
case 37:
    printch('k');
    buffstr[i] ='k';
    i++;
    break;
case 38:
    printch('l');
    buffstr[i] ='l';
    i++;
    break;
case 39:
    printch(';');
    buffstr[i] ='(';
    i++;
    break;
case 40:
    printch((char)44);           //Single quote {'}
    buffstr[i] = (char)44;
    i++;
    break;
case 41:
    printch((char)44);           //Back Tick {'}
    buffstr[i] = (char)44;
    i++;
    break;
case 42:                         //Left shift
    printch('q');
    buffstr[i] ='q';
    i++;
    break;
case 43:                         // (< for somekeyboards)
    printch((char)92);
    buffstr[i] ='q';
    i++;
    break;
case 44:
    printch('z');
    buffstr[i] ='z';
    i++;

```

```

        break;
case 45:
    printch('x');
    buffstr[i] ='x';
    i++;
    break;
case 46:
    printch('c');
    buffstr[i] ='c';
    i++;
    break;
case 47:
    printch('v');
    buffstr[i] ='v';
    i++;
    break;
case 48:
    printch('b');
    buffstr[i] ='b';
    i++;
    break;
case 49:
    printch('n');
    buffstr[i] ='n';
    i++;
    break;
case 50:
    printch('m');
    buffstr[i] ='m';
    i++;
    break;
case 51:
    printch(',');
    buffstr[i] =',';
    i++;
    break;
case 52:
    printch('.');
    buffstr[i] ='.';
    i++;
    break;
case 53:
    printch('/');
    buffstr[i] ='/';

```

```

        i++;
        break;
    case 54:
        printch('.');
        buffstr[i] = '.';
        i++;
        break;
    case 55:
        printch('/');
        buffstr[i] = '/';
        i++;
        break;
    case 56:
        printch(' ');           // Right shift
        buffstr[i] = ' ';
        i++;
        break;
    case 57:
        printch(' ');
        buffstr[i] = ' ';
        i++;
        break;
    }
}
buffstr[i] = 0;
return buffstr;
}
#endif

```

3.9 Modified KERNEL.C

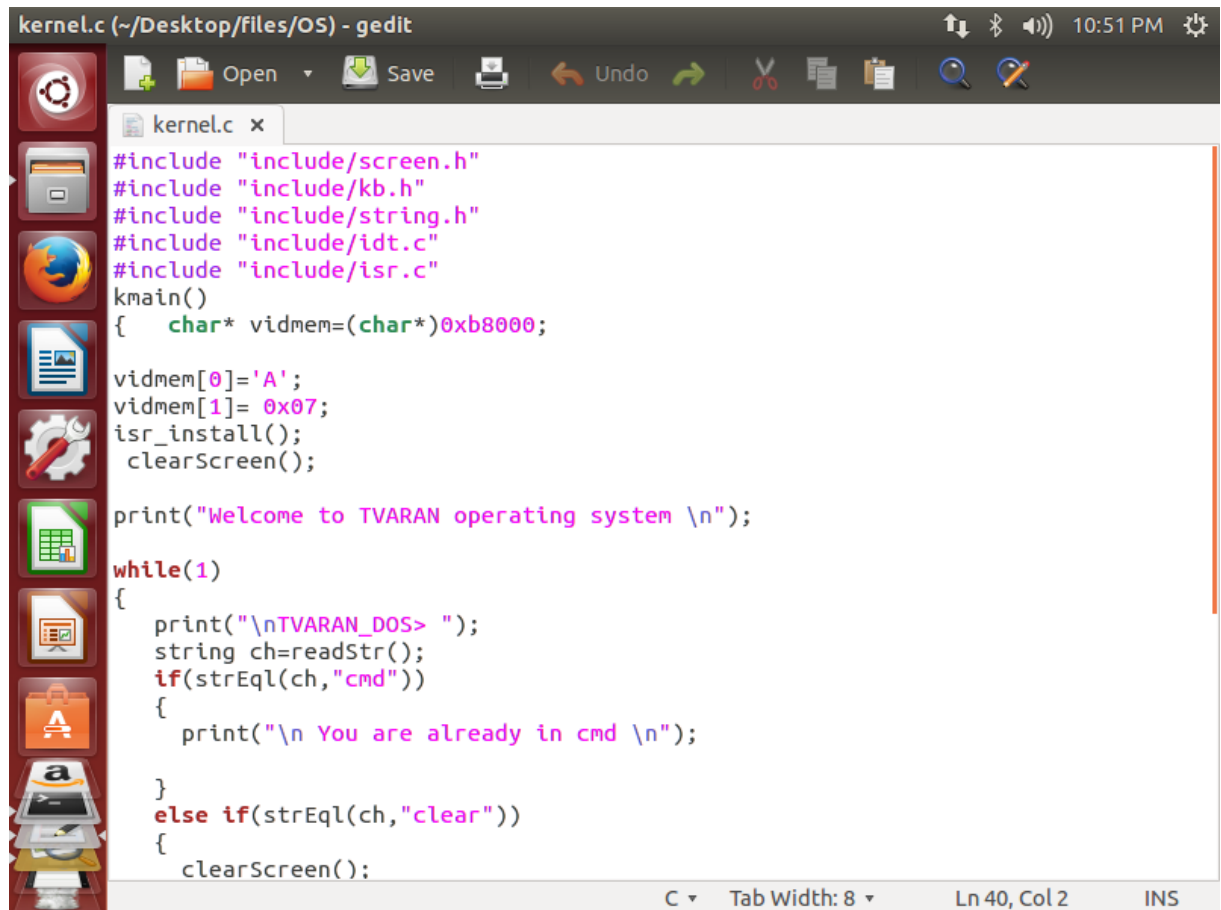
It consists of various functions to add commands to the operating system.

Here, the character entered by user through keyboard is matched with available strings and if it matches with any of the predefined commands then the respective output belonging to output set by for that command will be displayed.

For example, CLRSCR- If user enters clrscr, kernel.c will match it with all available commands and it will find a match. then body of that command will be executed which consists of clearScreen(), thus output screen will be cleared.

OSINFO- as soon as it will get matched with osinfo by strcmp function, information regarding os present in print statement will be displayed.

Default – if the entered string doesn't match with any of the available strings, BAD COMMAND will be displayed as output.



```
kernel.c (~/Desktop/files/OS) - gedit
#include "include/screen.h"
#include "include/kb.h"
#include "include/string.h"
#include "include/idt.c"
#include "include/isr.c"
kmain()
{
    char* vidmem=(char*)0xb8000;

    vidmem[0]='A';
    vidmem[1]= 0x07;
    isr_install();
    clearScreen();

    print("Welcome to TVARAN operating system \n");

    while(1)
    {
        print("\nTVARAN_DOS> ");
        string ch=readStr();
        if(strEql(ch,"cmd"))
        {
            print("\n You are already in cmd \n");
        }
        else if(strEql(ch,"clear"))
        {
            clearScreen();
        }
    }
}
```

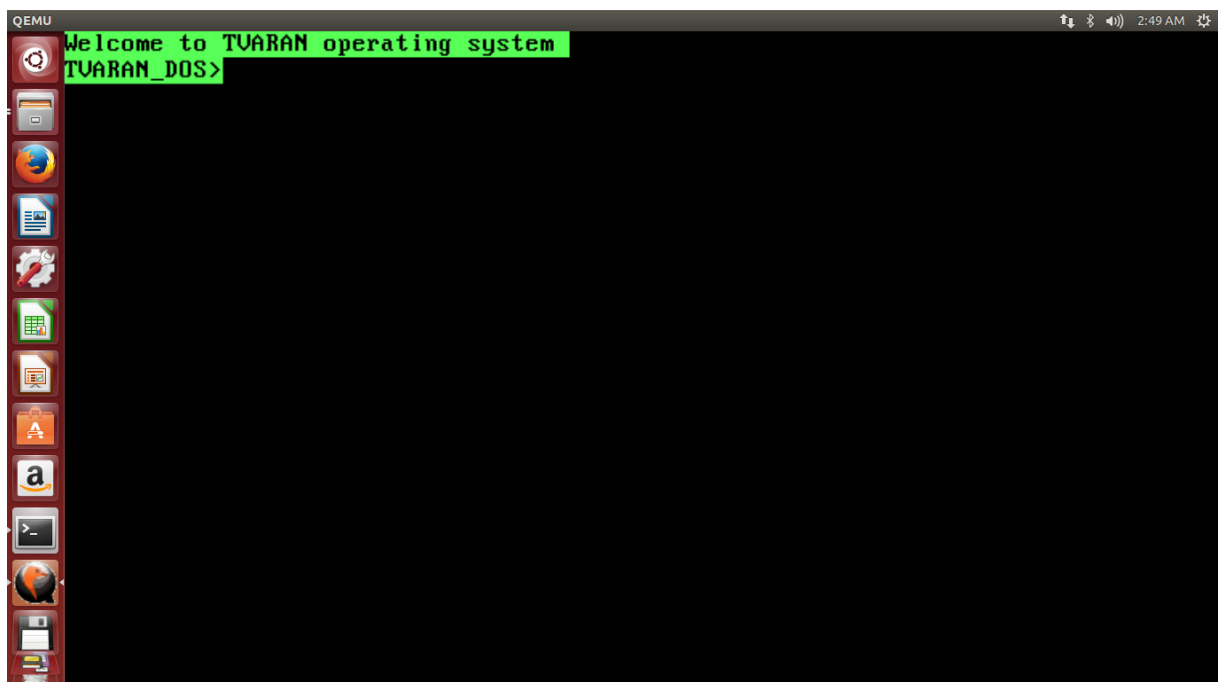
Full code of modified kernel.c file.

```
#include "include/screen.h"
#include "include/kb.h"
#include "include/string.h"
#include "include/idt.c"
#include "include/isr.c"
kmain()
{
    char* vidmem=(char*)0xb8000;
    vidmem[0]='T';
    vidmem[1]= 0x25;
    vidmem[2]='V';
    vidmem[3]= 0x21;
    vidmem[4]='A';
    vidmem[5]= 0x21;
    vidmem[6]='R';
    vidmem[7]= 0x20;
    vidmem[8]='A';
    vidmem[9]= 0x28;
    vidmem[10]='N';
    vidmem[11]= 0x25;
```

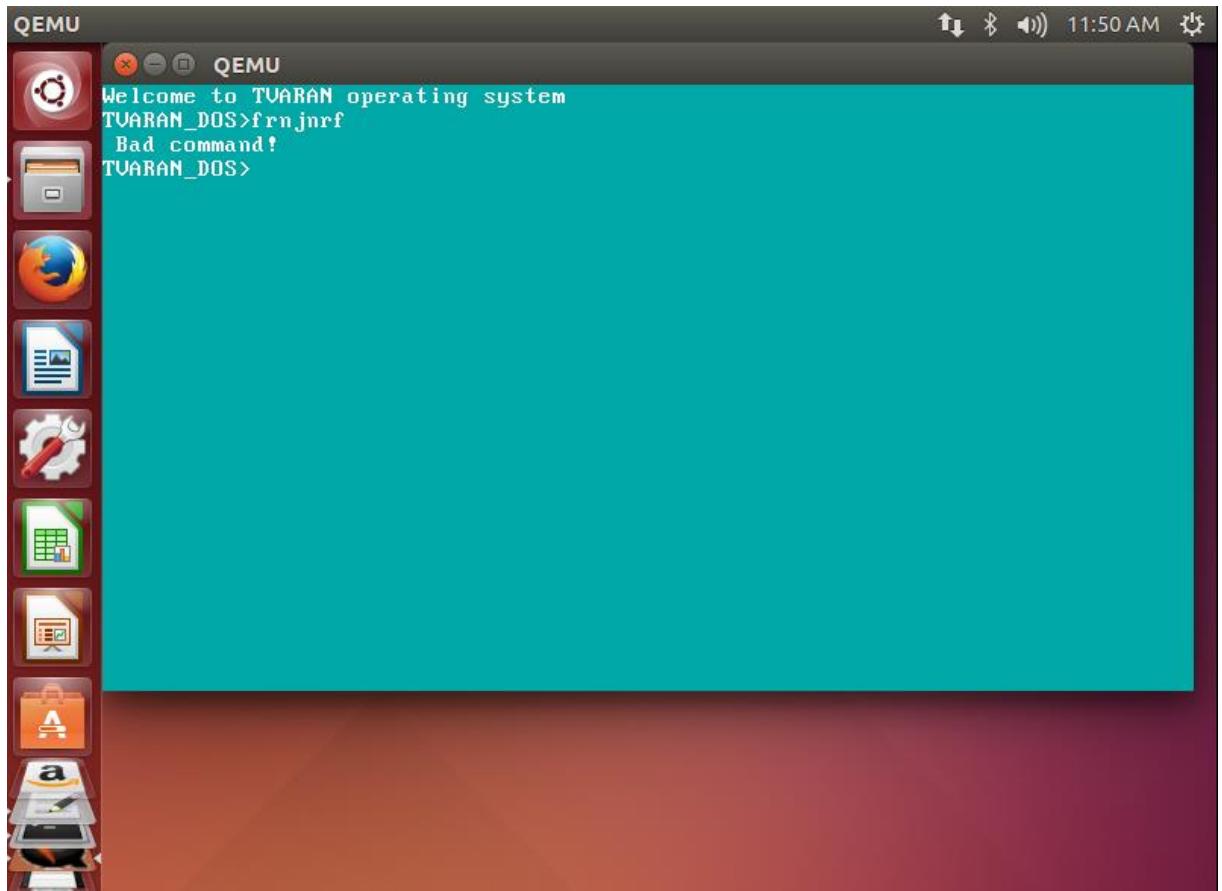
```

isr_install();
clearScreen();
print("Welcome to TVARAN operating system \n");
while(1)
{
    print("\nTVARAN_DOS> ");
    string ch=readStr();
    if(strEql(ch,"cmd"))
    {
        print("\n You are already in cmd \n");
    }
    else if(strEql(ch,"clrscr"))
    {
        clearScreen();
    }
    else if(strEql(ch,"osinfo"))
    {
        print("\n OS_name-tvaran,Version 1.7.4- 20150827_223240-lgw01-56");
    }
    else
    {
        print("\n Bad command!\n");
    }
    print("\n");
}
}

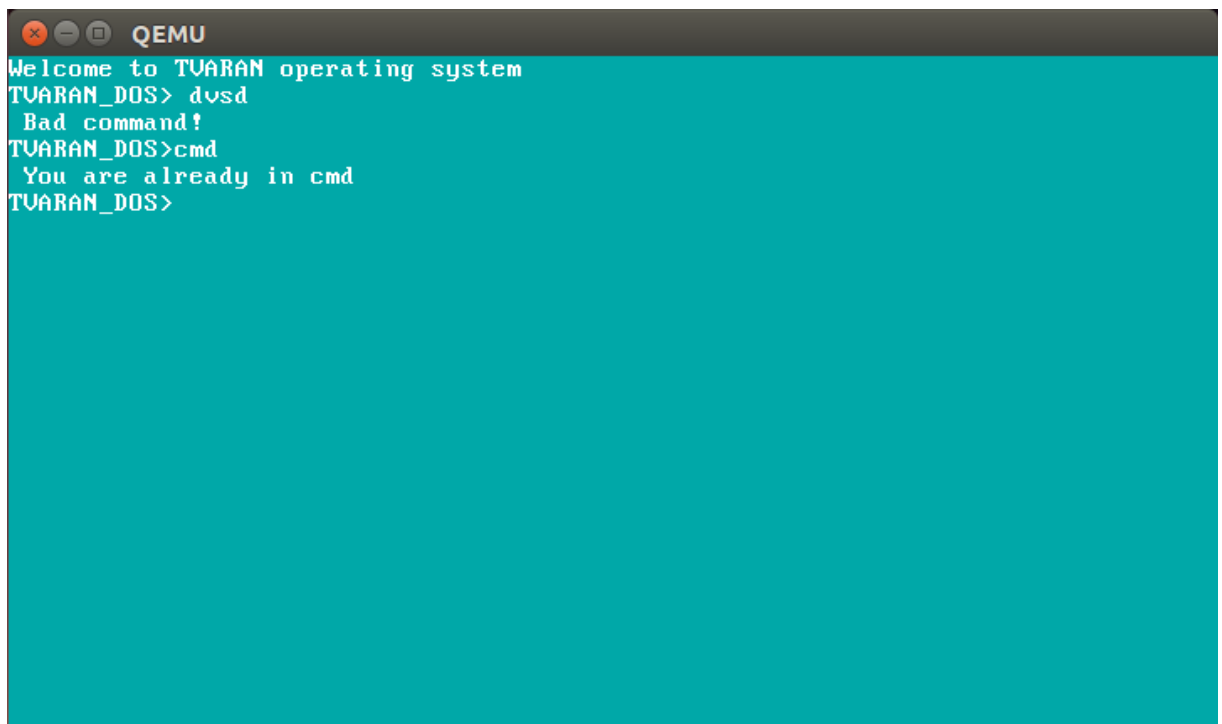
```



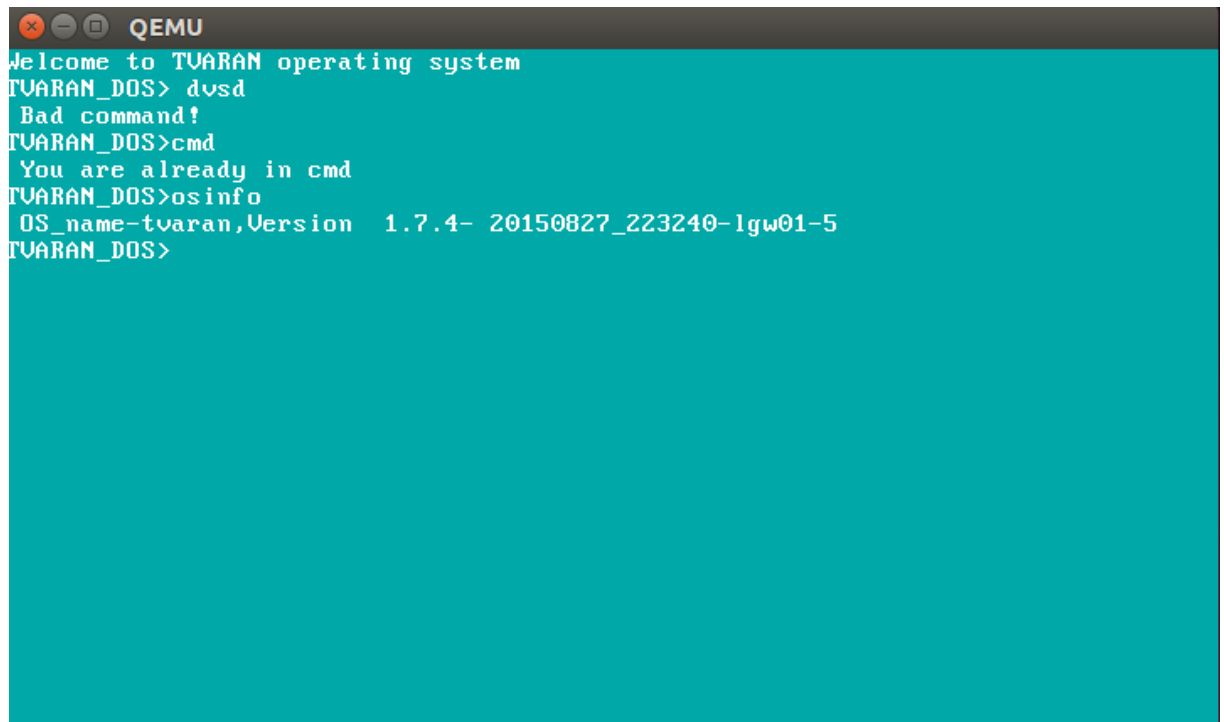
Snapshot shows default case in which entered string does not matches with any of the available commands.



Snapshot below shows the use of CMD command with its output.



Snapshot below shows the use of OSINFO command.

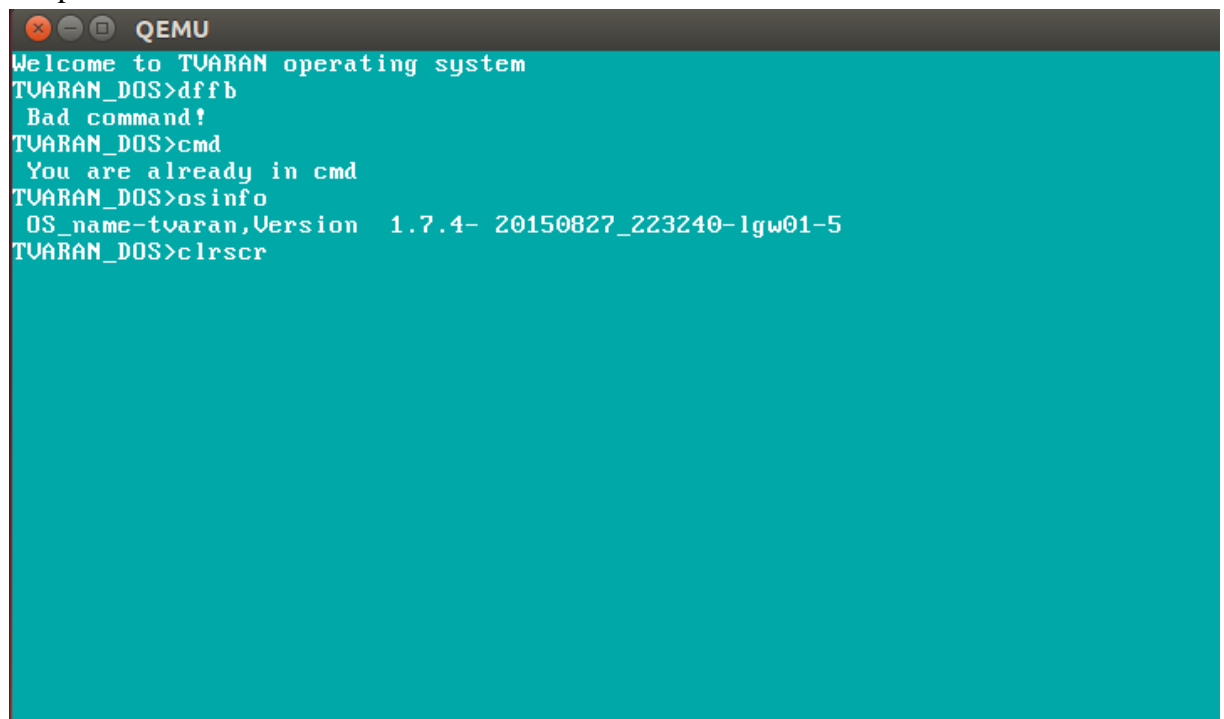


```

QEMU
Welcome to TVARAN operating system
TVARAN_DOS> d\sd
Bad command!
TVARAN_DOS>cmd
You are already in cmd
TVARAN_DOS>osinfo
OS_name-tvaran,Version 1.7.4- 20150827_223240-1gw01-5
TVARAN_DOS>

```

Snapshot below shows the use of CLRSCR command which clears the screen.

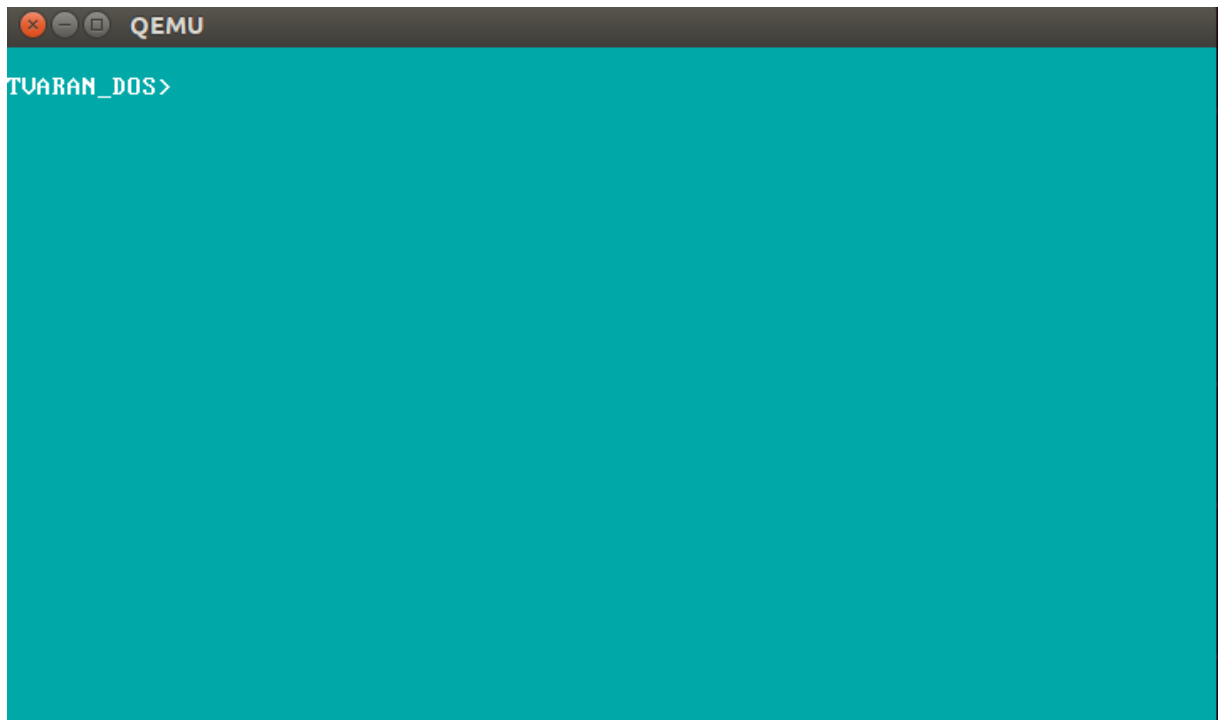


```

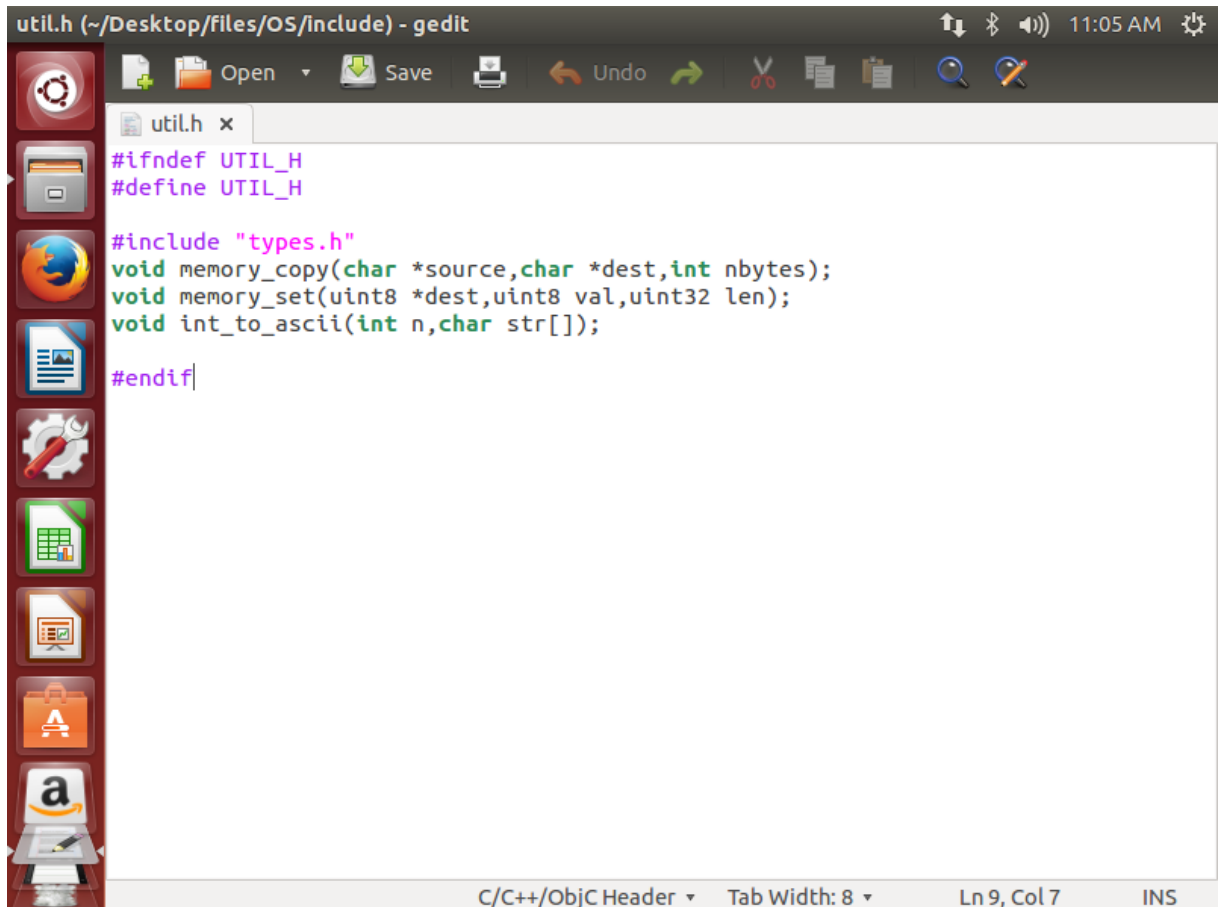
QEMU
Welcome to TVARAN operating system
TVARAN_DOS>dffb
Bad command!
TVARAN_DOS>cmd
You are already in cmd
TVARAN_DOS>osinfo
OS_name-tvaran,Version 1.7.4- 20150827_223240-1gw01-5
TVARAN_DOS>clrscr

```

Snapshot below shows resultant output screen after the use of CLRSCR command.



UTIL.H- *Utility* file is user defined file designed to help analyze, configure, optimize or maintain a computer.it consists of headers of all such functions to perform optimization such as function for copying from one part of memory to another setting the memory and conversion of string to ASCII.

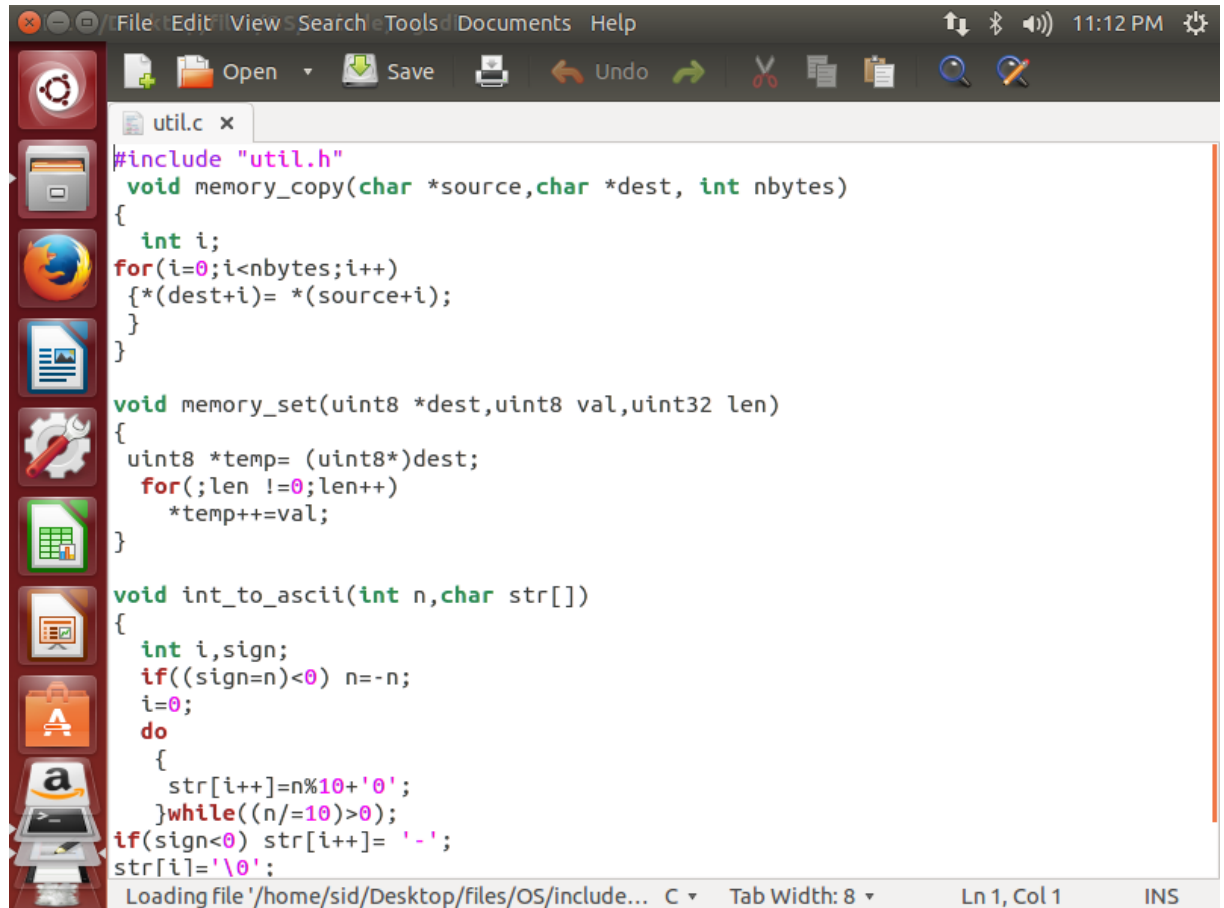


UTIL.C –it consists of definition of various functions defined in UTILITY.H .

Memory_copy() – this function takes three arguments source, destination address and no. of bytes to be transferred.it transfers the mentioned amount of bytes from source address to destination address.

Memory_set() – this function also takes three arguments destination, value and its length.it prints the value entered by us to a particular location mentioned by us in destination and upto the mentioned length only.

Int_to_ASCII() – this function converts integer into its corresponding ASCII value.



```

#include "util.h"
void memory_copy(char *source,char *dest, int nbytes)
{
    int i;
    for(i=0;i<nbytes;i++)
    {*(dest+i)= *(source+i);
    }
}

void memory_set(uint8 *dest,uint8 val,uint32 len)
{
    uint8 *temp= (uint8*)dest;
    for(;len !=0;len++)
        *temp++=val;
}

void int_to_ascii(int n,char str[])
{
    int i,sign;
    if((sign=n)<0) n=-n;
    i=0;
    do
    {
        str[i++]=n%10+'0';
    }while((n/=10)>0);
    if(sign<0) str[i++]='-';
    str[i]='\0';
}
    
```

Full code for UTILITY.C file

```

#include "util.h"

void memory_copy(char *source,char *dest, int nbytes)
{
    int i;
    for(i=0;i<nbytes;i++)
    {*(dest+i)= *(source+i);
    }
}

void memory_set(uint8 *dest,uint8 val,uint32 len)
{
    
```

```

uint8 *temp= (uint8*)dest;
for(;len !=0;len++)
    *temp++=val;
}

void int_to_ascii(int n,char str[])
{
    int i,sign;
    if((sign=n)<0) n=-n;
    i=0;
    do
    {
        str[i++]=n%10+'0';
    }while((n/=10)>0);
    if(sign<0) str[i++] = '-';
    str[i]='\0';
}

```

CHAPTER 4: INTERRUPT HANDLING

An Interrupt is an external asynchronous signal requiring a need for attention by software or hardware. It allows a way of interrupting the current task so that we can execute something more important. Interrupts provide a way to help trap problems, such as divide by zeros. If the processor finds a problem with the currently executing code, it provides the processor alternative code to execute to fix that problem

4.1 Interrupt Types

There are two types of interrupts: Hardware Interrupts and Software Interrupts. In the 8259A PIC tutorial, we have covered hardware interrupts. This tutorial focuses on software interrupts.

Hardware Interrupts

A hardware interrupt is an interrupt triggered by a hardware device. Normally, these are hardware devices that require attention. The hardware Interrupt handler will be required to service this hardware request.

This tutorial does not cover hardware interrupt handling, as that is hardware specific. For the x86 architecture, hardware interrupts are handled by programming the 8259A Programmable Interrupt Controller (PIC). Please see our 8259A PIC tutorial for more information on hardware interrupt handling.

Spurious Interrupt

This is a hardware interrupt generated by electrical interference in the interrupt line, or faulty hardware. We do NOT want this!

Software Interrupts

This is where the fun stuff is at!

Software Interrupts are interrupts implemented and triggered in software. Normally, the processor's instruction set will provide an instruction to service software interrupts. For the x86 architectures, these are normally INT imm, and INT 3. It also uses IRET and IRETD instructions.

INT imm and INT 3 instructions are used to generate an interrupt, while the IRET class of instructions are used to return from Interrupt Routines (IRs).

For example, here we generate an interrupt through a software instruction:

```
int      3                ; generates software interrupt 3
```

These instructions may be used to generate software interrupts and execute Interrupt Routines (IR)'s through software.

As you know, software interrupts were available in real mode. However, as soon as we made the jump to protected mode, the Interrupt Vector Table (IVT) became invalid. Because of this, we cannot use interrupts. Instead, we have to make our own.

We will cover software interrupt handling in this tutorial.

4.2 Interrupt Routines (IRs)

An Interrupt Routine (IR) is a special function used to handle an Interrupt Request (IRQ).

When the processor executes an interrupt instruction, such as INT, it executes the Interrupt Routine (IR) at that location within the Interrupt Vector Table (IVT).

This means, it simply executes a routine that we define. Not too hard, huh? This special routine determines the Interrupt Function to execute normally based off of the value in the AX register. This allows us to define multiple functions in an interrupt call. Such as, the DOS INT 21h function 0x4c00.

Remember: Executing an interrupt simply executes an interrupt routine that you created. For example, the instruction INT 2 will execute the IR at index 2 in the IVT. Cool?

IRs are commonly also referred to as **Interrupt Requests (IRQs)**. However, the naming convention of IRs are still used within the ISA bus, so understanding both names is important.

4.3 Interrupt Requests (IRQs)

An Interrupt Request (IRQ) refers to the act of interrupting an event by signaling the system either through the Control Bus IR line or through one of the 8259A Programmable Interrupt Controller (PIC) IR lines.

For systems with a single 8259 PIC, there are 8 IRQ lines, labeled IR0 IR7. For systems with 2 8259 PICs, there are 16 possible IRQ? labeled IR0 IR15. On the system ISA bus, these lines are labeled as IRQ0 IRQ15.

Newer intel based systems integrate an Advanced Programmable Interrupt Controller (APIC) device that allows 255 IRQs per controller.

For more information about IRQs, please see either the 8259A PIC tutorial or the APIC tutorial.

What this means is that the 8259A PIC can signal the processor to generate a software interrupt call through a hardware device by activating the processors IR line, and the processor to execute the correct interrupt handler. **This allows us to handle hardware device requests through software.** Please see the 8259A PIC tutorial for more information on this...It is very important to understand this.

4.4 Interrupt Service Routines (ISRs)

Interrupt Service Routines (ISRs) is an Interrupt Handler. These are important to understand, so let's look closer.

Interrupt Handlers

An interrupt handler is an IR for handling interrupts and IRQs. In other words, they are callback methods that we define for handling both hardware and software interrupts.

There are two types of ISRs: **FLIH**, and **SLIH**.

First Level Interrupt Handler (FLIH)

A FLIH is considered to be part of the lower half of a device driver or kernel. These interrupt handlers are platform specific, and usually service hardware requests, executing similar to Interrupt Routines (IRs) and Interrupt Requests (IRQs). They have short execution time. Their primary duty is to service the interrupt, or to record platform specific information which is only available at the time of the interrupt (As it is running in a lower level.) It may also schedule or execute a SLIH, if needed.

Second Level Interrupt Handler (SLIH)

These interrupt handlers are longer lived than FLIHs. In this way, it is similar to a task or process. SLIHs are normally executed and managed by a kernel program, or by FLIHs.

Nested Interrupt Handlers

When an interrupt handler is executed and the Interrupt Flag (IF) is set, interrupts can still be executed during the current interrupt. This is known as a nested interrupt.

4.5 Interrupts in Real Mode

Interrupts in Real Mode are handled through the Interrupt Vector Table (IVT). The Interrupt Vector Table (IVT) is a list of Interrupt Vectors. There are 256 Interrupts in the IVT.

4.6 IVT Map

The IVT is located in the first 1024 bytes of physical memory, from addresses 0x0 through 0x3FF. Each entry inside of the IVT is 4 bytes, in the following format:

- Byte 0: Offset Low Address of the Interrupt Routine (IR)
- Byte 1: Offset High Address of the IR
- Byte 2: Segment Low Address of the IR
- Byte 3: Segment High Address of the IR

Notice that each entry in the IVT simply contains the address of the IR to call. This allows us to create a simple function anywhere in memory (Our IR). As long as the IVT contains the addresses of our functions, everything will work fine.

Okay, Lets take a look at the IVT. The first few interrupts are reserved, and stay the same.

x86 Interrupt Vector Table (IVT)		
Base Address	Interrupt Number	Description
0x000	0	Divide by 0
0x004	1	Single step (Debugger)
0x008	2	Non Maskable Interrupt (NMI) Pin
0x00C	3	Breakpoint (Debugger)
0x010	4	Overflow
0x014	5	Bounds check
0x018	6	Undefined Operation Code (OPCode) instruction
0x01C	7	No coprocessor
0x020	8	Double Fault
0x024	9	Coprocessor Segment Overrun
0x028	10	Invalid Task State Segment (TSS)
0x02C	11	Segment Not Present
0x030	12	Stack Segment Overrun
0x034	13	General Protection Fault (GPF)
0x038	14	Page Fault
0x03C	15	Unassigned
0x040	16	Coprocessor error
0x044	17	Alignment Check (486+ Only)
0x048	18	Machine Check (Pentium/586+ Only)
0x05C	19-31	Reserved exceptions
0x068 - 0x3FF	32-255	Interrupts free for software use

Not to hard. Each of these interrupts are located at a base address within the IVT.

4.7 Interrupts in Protected Mode

As we are developing a protected mode operating system. This will be important to us. As you know, we cannot access the IVT in protected mode do to a lot of reasons. Because of this, we cannot access or use any more interrupts. So, instead, we need to create our own.

...And it all starts with the Interrupt Descriptor Table.

4.8 Interrupt Descriptor Table (IDT)

The Interrupt Descriptor Table (IDT) is a special table used by the processor for the management of IRs. Its use depends on the mode of the processor. The IDT itself is an array of 256 **descriptors**, similar to the LDT and GDT.

Real Mode

In Real Mode, The IDT is also known as the IVT. Please see the description of the IVT in the above sections for more information.

Protected Mode

The way the IDT works in protected mode is very different than that of Real Mode (This is one of the many reasons why we cannot use the IVT in protected mode.) The IVT is still used, however.

The IDT is an array of 256 8 byte descriptors stored consecutively in memory and indexed by an interrupt vector within the IVT. We will take a look at these descriptors, descriptor types, and the details of the IDT next.

4.9 Interrupt Descriptor: Structure

A descriptor for an IDT takes the following formats. Some of the format changes depending on what type of descriptor this is.

- Bits 0...15:
 - **Interrupt / Trap Gate:** Offset address Bits 0-15 of IR
 - **Task Gate:**

Not used.
- Bits 16...31:
 - **Interrupt / Trap Gate:** Segment Selector (Useually 0x10)
 - **Task Gate:** TSS Selector
- Bits 31...35: Not used
- Bits 36...38:
 - **Interrupt / Trap Gate:** Reserved. Must be 0.
 - **Task Gate:** Not used.
- Bits 39...41:
 - **Interrupt Gate:** Of the format 0D110, where D determines size
 - **01110** - 32 bit descriptor
 - **00110** - 16 bit descriptor
 - **Task Gate:** Must be 00101
 - **Trap Gate:** Of the format 0D111, where D determines size

- **01111** - 32 bit descriptor
 - **00111** - 16 bit descriptor
- Bits 42...44: Descriptor Privilege Level (DPL)
 - **00**: Ring 0
 - **01**: Ring 1
 - **10**: Ring 2
 - **11**: Ring 3
- Bit 45: Segment is present (1: Present, 0:Not present)
- Bits 46...62:
 - **Interrupt / Trap Gate**: Bits 16...31 of IR address
 - **Task Gate**: Not used

4.10 How Interrupts Work: Detail

Finding the interrupt procedure to call

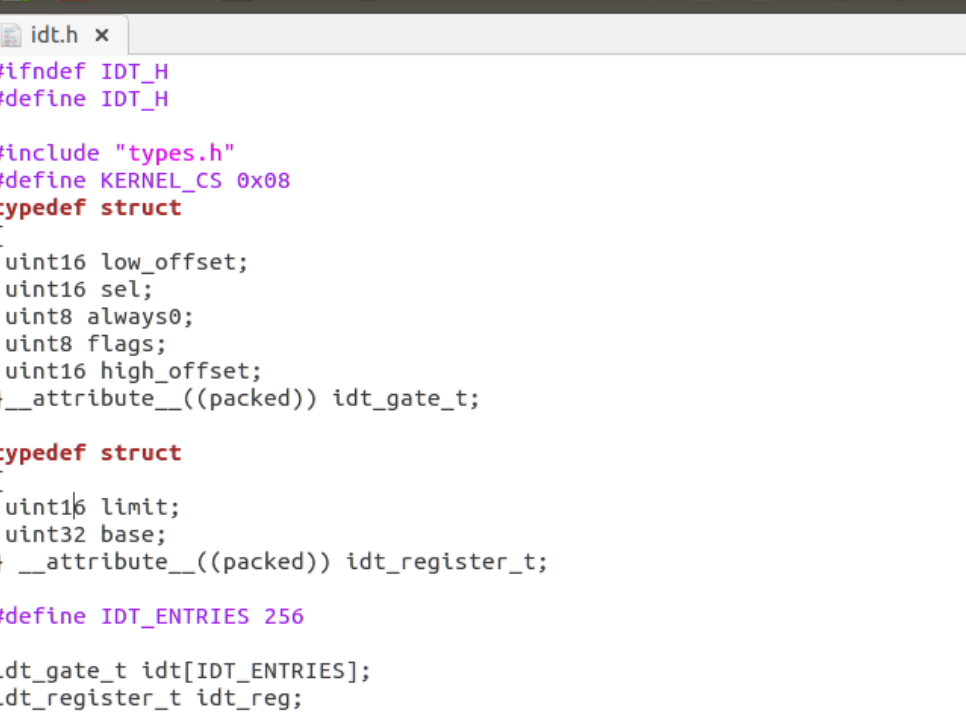
Remember that the descriptor is either an interrupt, trap, or task gate. If the index points to an interrupt or trap gate, the processor calls the exception or interrupt handler. This is done similar to CALLing a call gate. If the index points to a task gate, the processor executes a task switch to the exception or interrupt handler task similar to a CALL to a task gate.

The information and addresses for the handler are stored within this descriptor. When the processor performs the switch:

Executing the handler

- If the handler is going to be executed at a lower privilege level (bits 42-45 of descriptor), a stack switch occurs.
 1. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. The processor pushes the stack segment selector and stack pointer of the interrupt handler on this new stack.
 2. The processor saves the current state of EFLAGS, CS, and EIP on the new stack
 3. If an exception causes an **error code** to be saved, **the error code is pushed on the new stack after EIP**
- If the handler is going to be executed at the same privilege level (current privilege level (cpl) is the same as (bits 42-45 of descriptor))
 1. The processor saves the current state of EFLAGS, CS, EIP on the **current stack**.
 2. If an exception causes an error code to be saved, **the error code is pushed on the current stack after EIP**

It is very important to know how the stack is pushed when our interrupt handler is called, and what exceptions also push error codes.



idt.h (~/.Desktop/files/OS/include) - gedit

```
#ifndef IDT_H
#define IDT_H

#include "types.h"
#define KERNEL_CS 0x08
typedef struct
{
    uint16 low_offset;
    uint16 sel;
    uint8 always0;
    uint8 flags;
    uint16 high_offset;
} __attribute__((packed)) idt_gate_t;

typedef struct
{
    uint16 limit;
    uint32 base;
} __attribute__((packed)) idt_register_t;

#define IDT_ENTRIES 256

idt_gate_t idt[IDT_ENTRIES];
idt_register_t idt_reg;

void set_idt_gate(int n, uint32 handler);
void set_idt();
```

C/C++/ObjC Header ▾ Tab Width: 8 ▾ Ln 17, Col 7 INS

Full code of IDT.H with explanation

```
#ifndef IDT_H
```

```
#define IDT_H
```

```
#include "types.h"
```

```
/* Segment selectors */
```

```
#define KERNEL_CS 0x08
```

```
/* How every interrupt gate (handler) is defined */
```

```
typedef struct {
```

```
u16 low_offset; /* Lower 16 bits of handler function address */
```

```
u16 sel; /* Kernel segment selector */
```

```
u8 always0;
```

```
/* First byte
```

* Bit 7: "Interrupt is present"

* Bits 6-5: Privilege level of caller (0=kernel..3=user)

* Bit 4: Set to 0 for interrupt gates

* Bits 3-0: bits 1110 = decimal 14 = "32 bit interrupt gate" */

u8 flags;

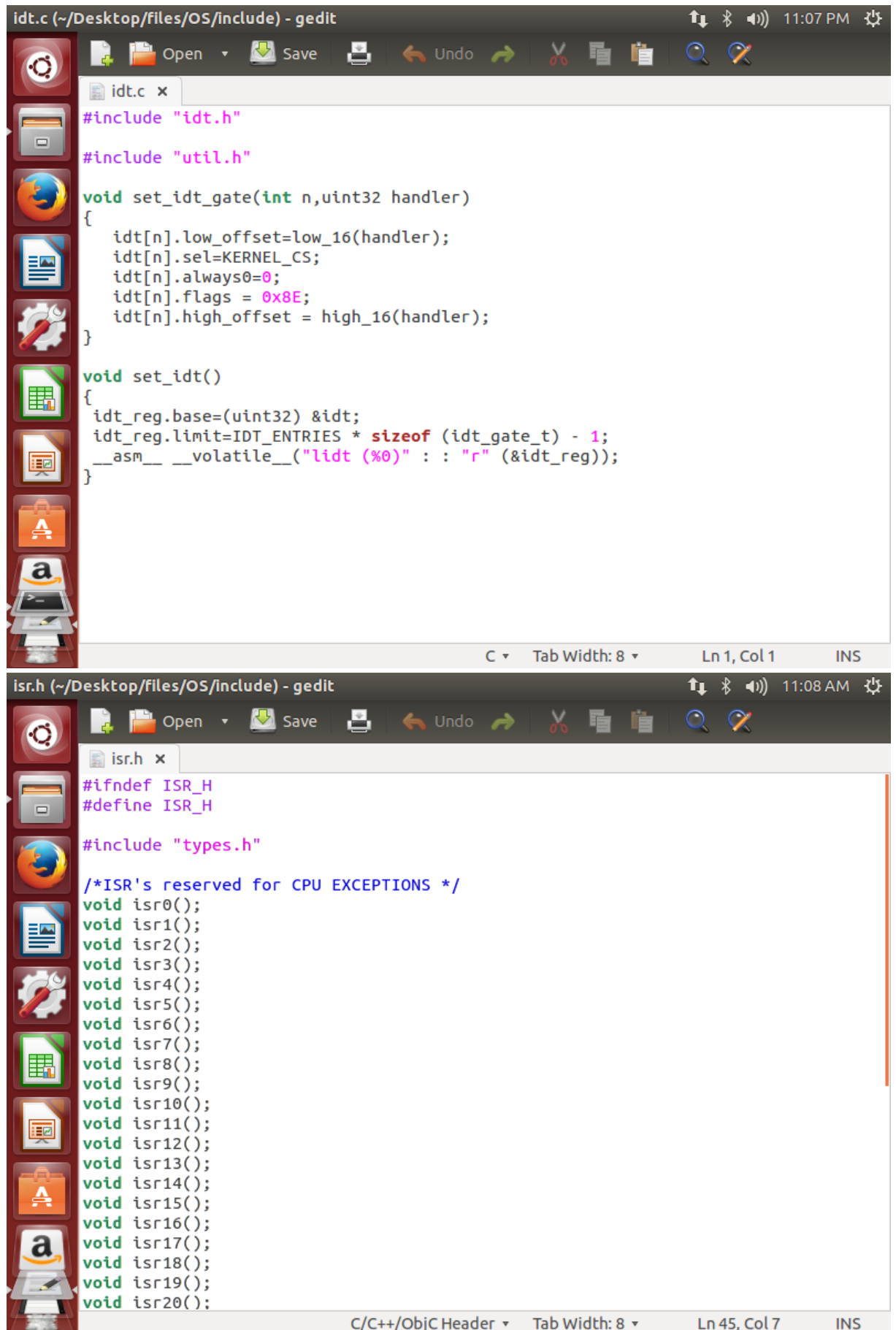
```
    u16 high_offset; /* Higher 16 bits of handler function address */
} __attribute__((packed)) idt_gate_t ;
```

```
/* A pointer to the array of interrupt handlers.
 * Assembly instruction 'lidt' will read it */
```

```
typedef struct {
    u16 limit;
    u32 base;
} __attribute__((packed)) idt_register_t;
```

```
#define IDT_ENTRIES 256
idt_gate_t idt[IDT_ENTRIES];
idt_register_t idt_reg;
/* Functions implemented in idt.c */
void set_idt_gate(int n, u32 handler);
void set_idt();
#endif
```

Snapshot of IDT.C consists of `set_idt_gate()` and `set_idt()` function in order to calculate the address or type of interrupt that has occurred. Flags help to detect whether interrupt has occurred or not for interrupt to occur flags must be set.



idt.c (~/Desktop/files/OS/include) - gedit

```
#include "idt.h"
#include "util.h"

void set_idt_gate(int n, uint32 handler)
{
    idt[n].low_offset = low_16(handler);
    idt[n].sel = KERNEL_CS;
    idt[n].always0 = 0;
    idt[n].flags = 0x8E;
    idt[n].high_offset = high_16(handler);
}

void set_idt()
{
    idt_reg.base = (uint32) &idt;
    idt_reg.limit = IDT_ENTRIES * sizeof(idt_gate_t) - 1;
    __asm__ __volatile__("lidt (%0)" : : "r" (&idt_reg));
}
```

isr.h (~/Desktop/files/OS/include) - gedit

```
#ifndef ISR_H
#define ISR_H

#include "types.h"

/*ISR's reserved for CPU EXCEPTIONS */
void isr0();
void isr1();
void isr2();
void isr3();
void isr4();
void isr5();
void isr6();
void isr7();
void isr8();
void isr9();
void isr10();
void isr11();
void isr12();
void isr13();
void isr14();
void isr15();
void isr16();
void isr17();
void isr18();
void isr19();
void isr20();
```

Full code of ISR.H with explanation


```

#ifndef ISR_H
#define ISR_H
#include "types.h"
/* ISRs reserved for CPU exceptions */
extern void isr0();
extern void isr1();
extern void isr2();
extern void isr3();
extern void isr4();
extern void isr5();
extern void isr6();
extern void isr7();
extern void isr8();
extern void isr9();
extern void isr10();
extern void isr11();
extern void isr12();
extern void isr13();
extern void isr14();
extern void isr15();
extern void isr16();
extern void isr17();
extern void isr18();
extern void isr19();
extern void isr20();
extern void isr21();
extern void isr22();
extern void isr23();
extern void isr24();
extern void isr25();
extern void isr26();
extern void isr27();
extern void isr28();
extern void isr29();
extern void isr30();
extern void isr31();

/* Struct which aggregates many registers */
typedef struct {
    u32 ds;                /* Data segment selector */
    u32 edi, esi, ebp, esp, ebx, edx, ecx, eax;    /* Pushed by pusha. */

```

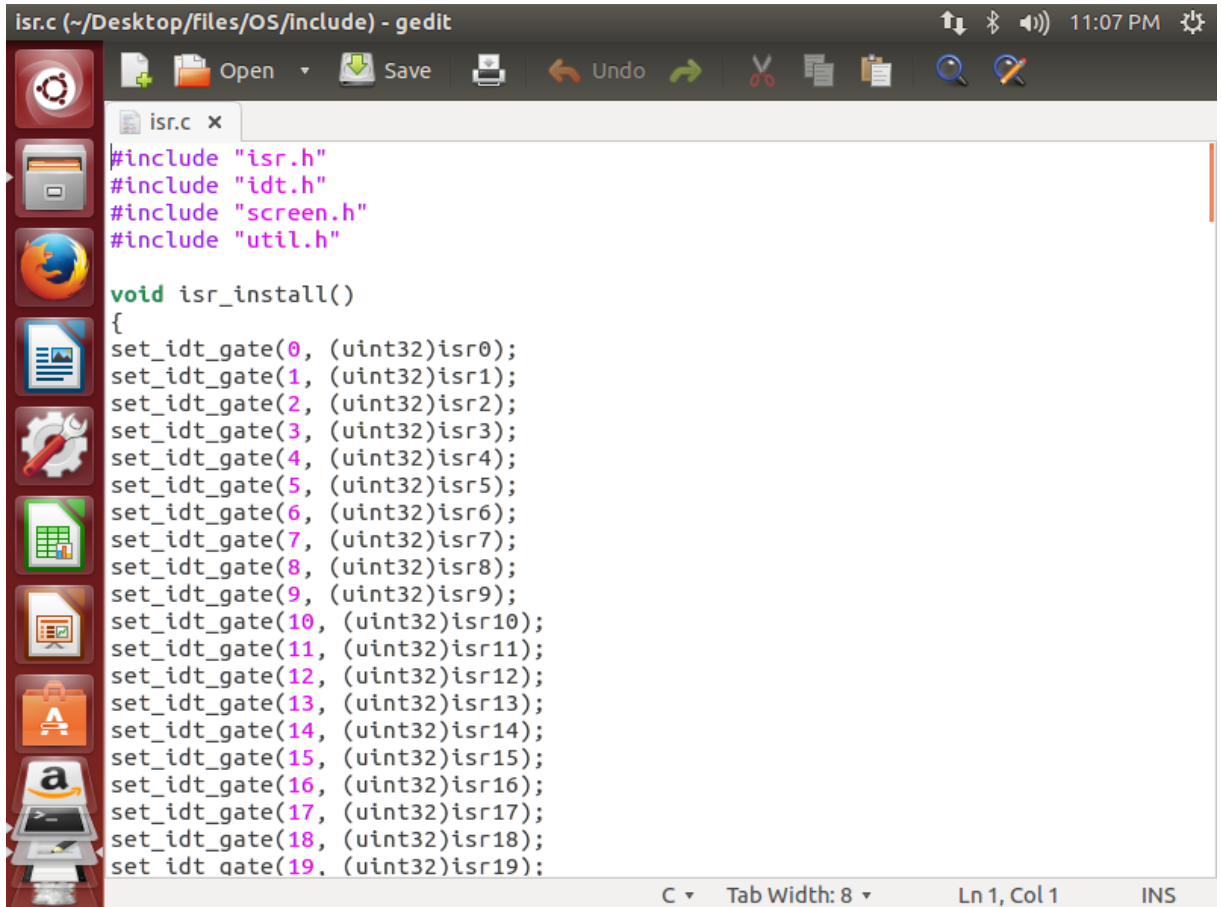
```

    u32 int_no, err_code;          /* Interrupt number and error code (if applicable) */
    u32 eip, cs, eflags, useresp, ss; /* Pushed by the processor automatically */
} registers_t;

void isr_install();
void isr_handler(registers_t r);

#endif

```



```

isr.c (~/Desktop/Files/OS/Include) - gedit
#include "isr.h"
#include "idt.h"
#include "screen.h"
#include "util.h"

void isr_install()
{
    set_idt_gate(0, (uint32)isr0);
    set_idt_gate(1, (uint32)isr1);
    set_idt_gate(2, (uint32)isr2);
    set_idt_gate(3, (uint32)isr3);
    set_idt_gate(4, (uint32)isr4);
    set_idt_gate(5, (uint32)isr5);
    set_idt_gate(6, (uint32)isr6);
    set_idt_gate(7, (uint32)isr7);
    set_idt_gate(8, (uint32)isr8);
    set_idt_gate(9, (uint32)isr9);
    set_idt_gate(10, (uint32)isr10);
    set_idt_gate(11, (uint32)isr11);
    set_idt_gate(12, (uint32)isr12);
    set_idt_gate(13, (uint32)isr13);
    set_idt_gate(14, (uint32)isr14);
    set_idt_gate(15, (uint32)isr15);
    set_idt_gate(16, (uint32)isr16);
    set_idt_gate(17, (uint32)isr17);
    set_idt_gate(18, (uint32)isr18);
    set_idt_gate(19, (uint32)isr19);
}

```

Full code of ISR.C

```

#include "isr.h"
#include "idt.h"
#include "../drivers/screen.h"
#include "../kernel/util.h"
/* Can't do this with a loop because we need the address
 * of the function names */
void isr_install() {
    set_idt_gate(0, (u32)isr0);
    set_idt_gate(1, (u32)isr1);
    set_idt_gate(2, (u32)isr2);
    set_idt_gate(3, (u32)isr3);
}

```

```

set_idt_gate(4, (u32)isr4);
set_idt_gate(5, (u32)isr5);
set_idt_gate(6, (u32)isr6);
set_idt_gate(7, (u32)isr7);
set_idt_gate(8, (u32)isr8);
set_idt_gate(9, (u32)isr9);
set_idt_gate(10, (u32)isr10);
set_idt_gate(11, (u32)isr11);
set_idt_gate(12, (u32)isr12);
set_idt_gate(13, (u32)isr13);
set_idt_gate(14, (u32)isr14);
set_idt_gate(15, (u32)isr15);
set_idt_gate(16, (u32)isr16);
set_idt_gate(17, (u32)isr17);
set_idt_gate(18, (u32)isr18);
set_idt_gate(19, (u32)isr19);
set_idt_gate(20, (u32)isr20);
set_idt_gate(21, (u32)isr21);
set_idt_gate(22, (u32)isr22);
set_idt_gate(23, (u32)isr23);
set_idt_gate(24, (u32)isr24);
set_idt_gate(25, (u32)isr25);
set_idt_gate(26, (u32)isr26);
set_idt_gate(27, (u32)isr27);
set_idt_gate(28, (u32)isr28);
set_idt_gate(29, (u32)isr29);
set_idt_gate(30, (u32)isr30);
set_idt_gate(31, (u32)isr31);
set_idt(); // Load with ASM
}

```

```

/* To print the message which defines every exception */

```

```

char *exception_messages[] = {
    "Division By Zero",
    "Debug",
    "Non Maskable Interrupt",
    "Breakpoint",
    "Into Detected Overflow",
    "Out of Bounds",
    "Invalid Opcode",
    "No Coprocessor",

```

```

        "Double Fault",
        "Coprocesor Segment Overrun",
        "Bad TSS",
        "Segment Not Present",
        "Stack Fault",
        "General Protection Fault",
        "Page Fault",
        "Unknown Interrupt",
        "Coprocesor Fault",
        "Alignment Check",
        "Machine Check",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved",
        "Reserved"
    };

```

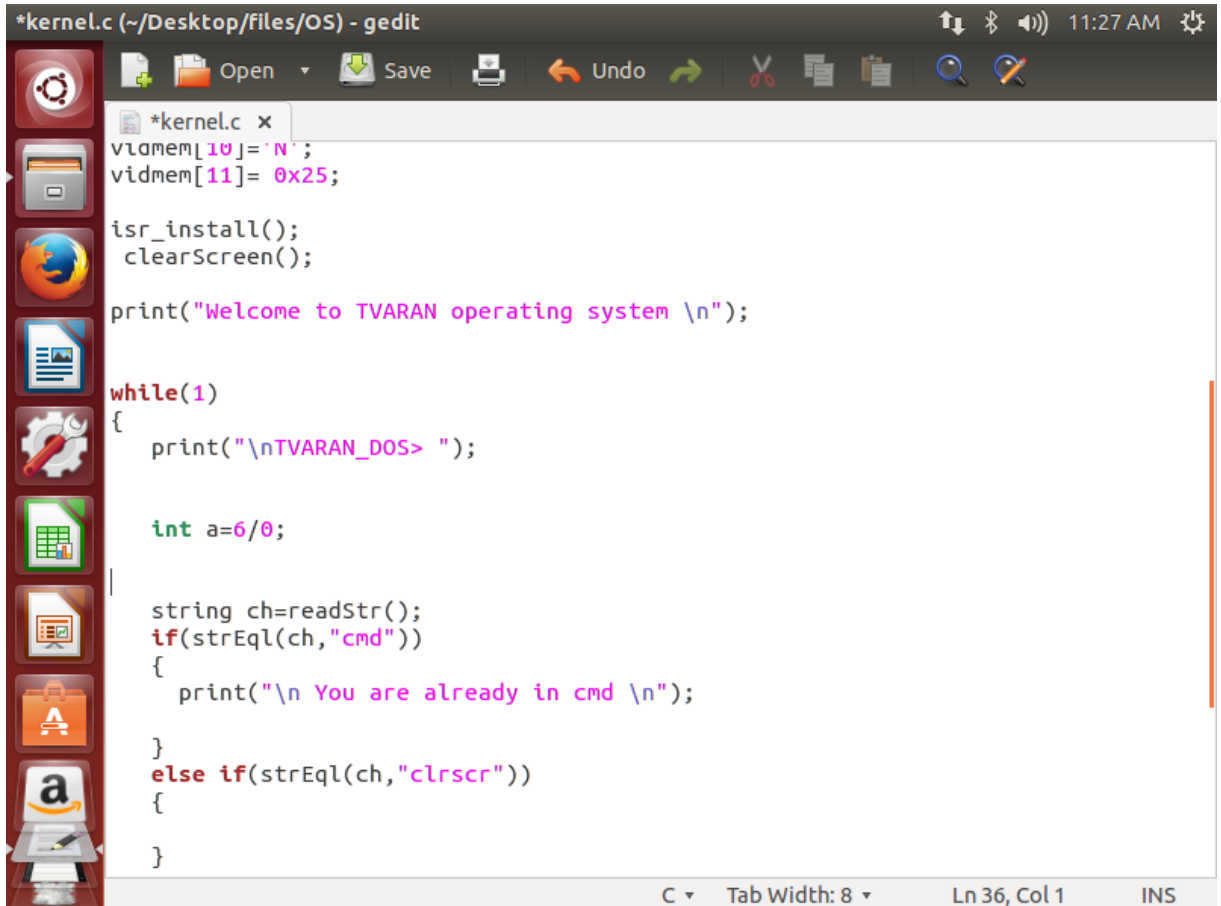
```

void isr_handler(registers_t r) {
    kprint("received interrupt: ");
    char s[3];
    int_to_ascii(r.int_no, s);
    kprint(s);
    kprint("\n");
    kprint(exception_messages[r.int_no]);
    kprint("\n");
}

```

KERNEL.C with an interrupt of division by zero
 In the main function int a is defined as
 Int a = 6/0;

Which leads to an undefined result. thus an interrupt occurs with message division by zero.



```
*kernel.c (~/Desktop/files/OS) - gedit
vrammem[10]='N';
vidmem[11]= 0x25;

isr_install();
clearScreen();

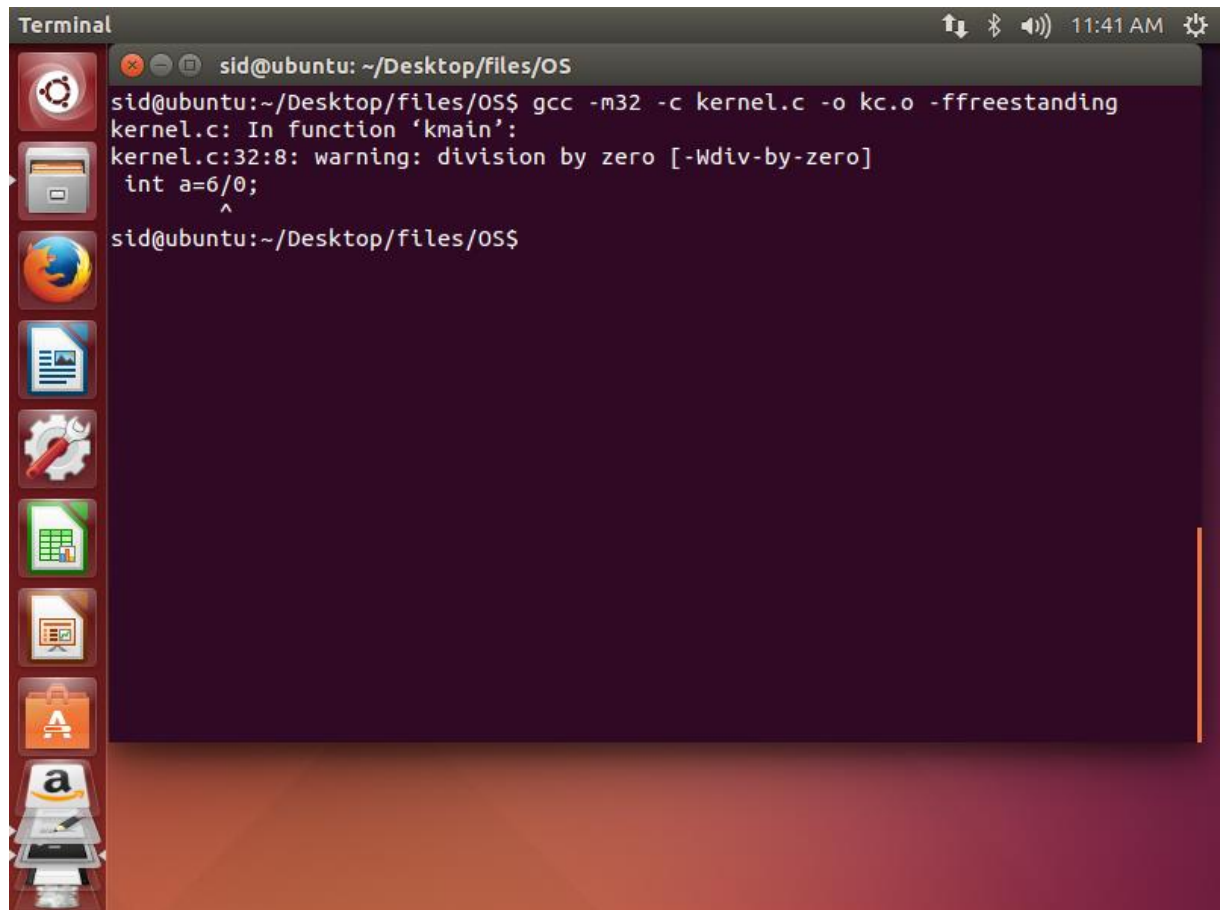
print("Welcome to TVARAN operating system \n");

while(1)
{
    print("\nTVARAN_DOS> ");

    int a=6/0;

    string ch=readStr();
    if(strEql(ch,"cmd"))
    {
        print("\n You are already in cmd \n");
    }
    else if(strEql(ch,"clrscr"))
    {
    }
}
```

Although the code is correct but due to presence of undefined value division by zero, terminal gives a warning of division by zero during compilation.



The image shows a terminal window titled "Terminal" with a dark background. The prompt is "sid@ubuntu: ~/Desktop/files/OS". The user has entered the command "gcc -m32 -c kernel.c -o kc.o -ffreestanding". The output shows the compilation of "kernel.c" in function "kmain". A warning is displayed: "kernel.c:32:8: warning: division by zero [-Wdiv-by-zero]" followed by the code snippet "int a=6/0;" with a caret pointing to the denominator. The prompt returns to "sid@ubuntu:~/Desktop/files/OS\$". The terminal window is part of a desktop environment with a sidebar containing icons for applications like Firefox, LibreOffice, and Amazon.

```
Terminal
sid@ubuntu: ~/Desktop/files/OS
sid@ubuntu:~/Desktop/files/OS$ gcc -m32 -c kernel.c -o kc.o -ffreestanding
kernel.c: In function 'kmain':
kernel.c:32:8: warning: division by zero [-Wdiv-by-zero]
  int a=6/0;
          ^
sid@ubuntu:~/Desktop/files/OS$
```

When the KERNEL.C with `int a = 6/0` is compiled, terminal shows a warning of division by zero. In spite of the warning, if the function is loaded, our output screen QEMU will display the following.

