OPERATING SYSTEMS
PROJECT REPORT
CSE-2005

Submitted in complete fulfilment for the award of the degree of B.Tech in Computer Science
under the guidance of

Prof. Vijayarajan V

BY,

NITISH GUPTA- 19BCI0189
SANSKAR TIWARI- 19BCE0955
ARIHANT DUGAR- 19BCE0250
ARADHYA BAGRODIA- 19BCE0982

BOOT LOADER

ACKNOWLEDGEMENT

We are thankful to our faculty VIJAYARAJAN V for giving us this opportunity to work on this project “BOOT LOADER”. He has guided us throughout the project and we are highly obliged to receive his precious suggestions and advice.

We are also grateful to his continuous mentorship that helped us tackle the difficulties and obstacles faced during the making of this project which eventually led to the successful completion of the project.

INDEX

ABSTRACT

A boot loader primarily manages and executes the boot sequence of a computer system. A boot loader program is typically started after the computer or the BIOS have finished performing the initial power and hardware device checks and tests. It fetches the OS kernel from the hard disk or any specified boot device within the boot sequence, into the main memory. A boot loader is associated with only a single operating system. An operating system can also have multiple boot loader programs classified as primary and secondary boot loaders, where a secondary boot loader might be larger and more capable than the primary boot loader.

KEYWORDS

1)BIOS (BASIC INPUT OUTPUT SYSTEM)
2)BOOT MAIN
3)BOOT LOADER
4)ASSEMBLY BOOT LOADER
5)EMBEDDED SYSTEM
6)OPERATING SYSTEM
7)KERNEL.

PROBLEM STATEMENT:

The objective behind selecting boot loader was the prime importance of boot loader to load an operating system . It's main role is to set up a minimal environment in which the OS can run, and run the operating system's startup procedure.

If we are able to modify the boot loader it will reduce the boot time of the computer and load the operating system faster .

Implementing a simple boot-loader and **modifying** it so as to enhance the boot time and thus improves the working efficiency of computer . It has multi-step sequence involving numerous sub-steps.
It can be used as a base for innovative boot loader in which the boot time is less.

THE BOOT LOADER SPECIFICATION

The Boot Loader Specification defines a scheme how different operating systems can cooperatively manage a boot loader configuration directory, that accepts drop-in files for boot menu items that are defined in a format that is shared between various boot loader implementations, operating systems, and userspace programs. The target audience for this specification is:

- Boot loader developers, to write a boot loader that directly reads its configuration at runtime from these drop-in snippets
- Distribution and Core OS developers, in order to create these snippets at OS/kernel package installation time
- UI developers, for implementing a user interface that discovers the available boot options
- OS Installer developers, for setting up the initial drop-in directory

WHY IS THERE A NEED FOR THIS SPECIFICATION?

Of course, without this specification things already work mostly fine. But here's why we think this specification is needed:

- To make the boot more robust, as no explicit rewriting of configuration files is required any more
- To improve dual-boot scenarios. Currently, multiple Linux installations tend to fight over which boot loader becomes the primary one in possession of the MBR, and only that one installation can then update the boot loader configuration of it freely. Other Linux installs have to be manually configured to never touch the MBR and instead install a chain-loaded boot loader in their own partition headers. In this new scheme as all installations share a loader directory no manual configuration has to take place, and all participants implicitly cooperate due to removal of name collisions and can install/remove their own boot menu entries at free will, without interfering with the entries of other installed operating systems.
- Drop-in directories are otherwise now pretty ubiquitous on Linux as an easy way to extend configuration without having to edit, regenerate or manipulate configuration files. For the sake of uniformity we should do the same for extending the boot menu.

- User space code can sanely parse boot loader configuration which is essential with modern BIOSes which do not necessarily initialize USB keyboards anymore during boot, which makes boot menus hard to reach for the user. If user space code can parse the boot loader configuration, too, this allows for UIs that can select a boot menu item to boot into, before rebooting the machine, thus not requiring interactivity during early boot.
- To unify and thus simplify configuration of the various boot loaders around, which makes configuration of the boot loading process easier for users, administrators and developers alike
- For boot loaders with configuration scripts such as grub2, adopting this spec allows for mostly static scripts that are generated only once at first installation, but then do not need to be updated anymore as that is done via drop-in files exclusively.

TECHNICAL DETAILS

We define two directories:

- $BOOT/loader/ is the directory containing all files defined by this specification
- $BOOT/loader/entries/ is the directory containing the drop-in snippets. This directory contains one .conf file for each boot menu item.

These directories are defined below the placeholder file system $BOOT. This placeholder file system shall be determined during installation time, and an fstab entry for it shall be created mounting it to /boot. The installer program should pick $BOOT according to the following rules:

- If the OS is installed on a disk with MBR disk label, and a partition with the MBR type id of 0xEA already exists it should be used as $BOOT.

- Otherwise, if the OS is installed on a disk with MBR disk label, a new partition with MBR type id of 0xEA shall be created, of a suitable size (let's say 500MB), and it should be used as $BOOT.
- If the OS is installed on a disk with GPT disk label, and a partition with the GPT type GUID of bc13c2ff-59e6-4262-a352-b275fd6f7172 already exists, it should be used as $BOOT.
- Otherwise, if the OS is installed on a disk with GPT disk label, and an ESP partition (i.e. with the GPT type UID of c12a7328-f81f-11d2-ba4b-00a0c93ec93b) already exists and is large enough (let's say 250MB) and otherwise qualifies, it should be used as $BOOT.
- Otherwise, if the OS is installed on a disk with GPT disk label, and if the ESP partition already exists but is too small, a new suitably sized (let's say 500MB) partition with GPT type GUID of bc13c2ff-59e6-4262-a352-b275fd6f7172 shall be created and it should be used as $BOOT.
- Otherwise, if the OS is installed on a disk with GPT disk label, and no ESP partition exists yet, a new suitably sized (let's say 500MB) ESP should be created and should be used as $BOOT.

LITERATURE SURVEY

In this modern arena everything relies on Computers. Users expect speed and efficiency among computers. This both features depend upon the type of machine and the type of operating system installed in that machine. The system type is based on the cost and the standards of Manufacturers who build the machine. But the performance of Operating system mostly depends upon the features provided by the loaders, as they are considered as the init of the operating systems. Every loader has its own specifications. Even such loaders possess their own abilities they have to rely BIOS as their root because is capable of loading every loaders all of operating systems. Thus their features are not a part when BIOS takes such a role. No loaders are BIOS independent.

EVOLUTION OF BOOT LOADER TECHNIQUES

The boot techniques originated with a design of a bootstrap algorithm for MIMO systems. **M.S. Ahmed and N. Sait** in September 1989, proposed this bootstrap algorithm for the self tuning control of MIMO Systems. They split the non-linear problem into 2 linear ones and the state estimation is carried out by assuming that the system parameters are known. This algorithm is used for online identification and control . This great success of boot design motivated the researchers to go on developments with the boot design structure.

Due to these developments**, Stephen A. Rogers and Leonard E. Schulwit** in the same year developed a bootstrap for a Distributed System (MULTIBUS II system) as a set of connected hosts. In this technique bootstrap parameters are located at a number of hosts and load the needed parameters when needed with a request or a call [13]. Better flexibility and a good start was achieved with the help of these boot loaders. This attracted the researchers on electrical field to implement the boot concept in the field of Electronics.

As a result in **1992 S. Xiao, R.Y.V.Chik and C.A.T. Salama** introduced selfbooting technique to increase the output resistance of MESFET .This was a considerable achievement in the field of electronics. The boot sectors that are applied in the Distributed Systems are vulnerable to viruses called Boot sector viruses, which makes the Distributed systems in a boot failure. So that entire systems get collapsed.

By taking this under consideration **Gerald J. Tesauro, Jeffrey O. Kephart, and Gregory B.Sorkin**, IBM T.J. Watson Re-search Center used a Neural Networks scheme on boot sectors to prevent the attack of boot straps from boot-sector viruses .Not only in the world of computers but in the field of embedded systems, boot loaders made a remarkable part.

Tony Benavides, Justin Treon, Jared Hulbert, and Willie Chang in 2007 implemented a Hybrid-Execute-In-Place Architecture that reduced the Boot Time to some extent . Such a design provided more efficiency in boot loaders. The arise of VLSI technology made the development of embedded loaders into a single chip.

In 2010 **Jie Zhang and Fengjing Shao** implemented a boot loader module based on the new high-security operating system with internal networking structure netOS-I. This module is capable of initializing the hardware environment of new computer architecture sCPU-dBUS, establishing the memory mapping, setting up the environment for netOS-I, and then loading and starting up netOS-I . A modified form of operating system is developed using a portable loader in 2010.

Kuan-Jen Lin and Chin-Yi Wang in 2012 extended TPM BIOS interrupt calls to support RAS encryption and decryption and accessing the NVRAM. They enhanced the security of the encrypted BIOS password using TPM RSA engine instead of conventional encryption and added techniques for storing the encrypted password to TPM NVRAM so that an attacker cannot clear it by removing the battery. They used TPM SHA-1 engine to verify data integrity of the full MBR and determine if the booting continues .

PROPOSED SYSTEM DESIGN

PROPOSED SYSTEM ARCHITECTURE

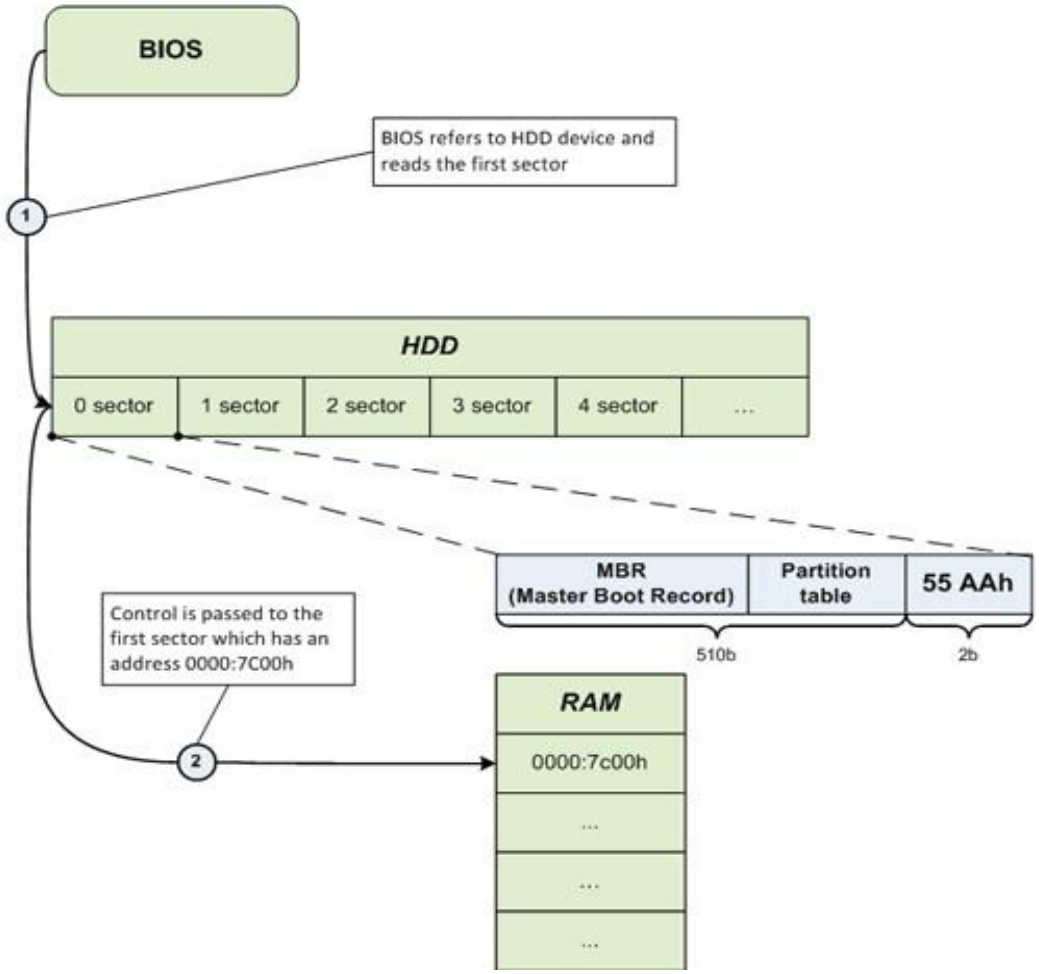
During the initial stages of computer operation, BIOS takes control over the machine hardware via the functions called interruptions.“mixed code” technique allows mixing low-level language commands and high-level constructions, which simplifies our task.

Java and C# will not be fit to the task, unfortunately, as they produce intermediate code after compilation. And in addition, a special virtual machine is used to perform conversions from intermediate code into language understood by a processor. So the code execution becomes possible only after a conversion, which doesn’t allow to take advantage of the “mixed code” technique.

To use the advantages of the “mixed code” technique, we require no less than two compilers. The first compiler is the core compiler that will be used for Assembler and C or C++. The second compiler is a linker: its task is to join the *.obj files to create a single executable file.

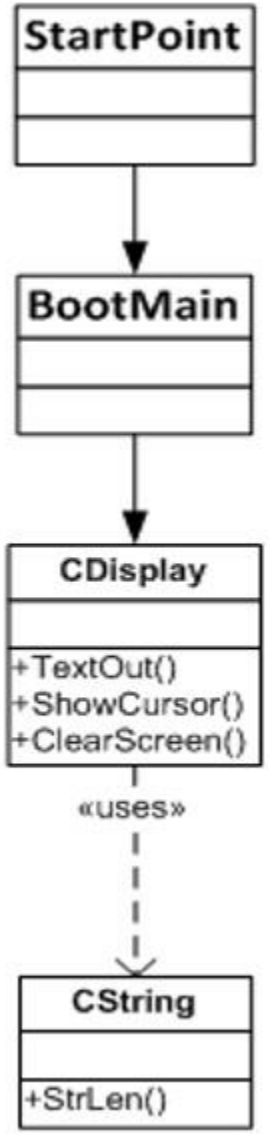
A processor functions in the 16-bit real mode, which has certain limitations, and in the 32-bit safe mode with the full available functionality. On the start, that is when the system is powered on, a processor operates in a real mode, that is why building a program and creating an executable file requires a compiler and a linker for Assembler that work in the 16-bit mode. A C or C++ compiler is required only to create *.obj files in the real mode.

The following diagram illustrates the interaction of system components during this process:



1. BIOS reads the first sector of the hard disk drive.
2. BIOS passes the control to Master Boot Record located at the address 0000:7c00, which triggers the OS booting process.

Code structure is simple and it has the following functions:



The first element is StartPoint. It is written in a low-level language. As high-level languages lack the required instructions, this element is created using Assembler only. Its task is to instruct the compiler to use specific memory model and list the address at which the loading to RAM must be performed after data from a disk was read. In addition, it fixes processor registers. After its role has been fulfilled, it eventually passes the control over to BootMain, an element written in high-level language.

BootMain takes control right after StartPoint. It is an entity similar to main, which is a primary function in which all program operations take place.And finally, the CDisplay and CString come in. They fulfill the role of the final actors displaying the message.
For bootloader development task we are using Microsoft Visual Studio.

MODULE DESCRIPTION (ALGORITHM / PSEUDOCODE)

1. BIOS interruptions and screen cleaners

Before displaying any messages, first of all, the screen must be cleared. BIOS has special interruptions for this task.

The structure of an interruption is:

```
int [number_of_interrupt];
mov al, 02h ; here we set the 80x25 graphical mode (text)
mov ah, 00h ; this is the code of the function that allows us to change the video mode
int 10h   ; here we call the interruption
```

2. Mixed code technique

Assembler instructions written in high-level code are called asm insertions, it is denoted by the introductory word _asm followed by a block of Assembler instructions enclosed in braces.

we combine the C++ code with the Assembler code that clears the screen to illustrate this technique.

```
void ClearScreen()
{
 __asm
{
 mov al, 02h ; here we set the 80x25 graphical mode (text)
mov ah, 00h ; this is the code of the function that allows us to change the video mode
int 10h    ; here we call the interruption
}
}
```

3. CString implementation

The CString class works with strings. The value of the string it contains will be used by the CDisplay class. There is the Strlen() method, which gets a pointer to a string as its parameter, counts the number of characters the obtained string contains, and returns the resulting number.

```
// CString.h
#ifndef__CSTRING__
#define__CSTRING__
#include "Types.h"
class CString
{
public:
    static byte Strlen(
        const char far* inStrSource
        );
};
#endif //__CSTRING__
```

```
// CString.cpp
#include "CString.h"
byte CString::Strlen(
        const char far* inStrSource
        )
{
        byte lenghtOfString = 0;

        while(*inStrSource++ != '\0')
        {
            ++lenghtOfString;
        }
        return lenghtOfString;
}
```

4. CDisplay implementation

As its name states, this class is developed to interact with the screen. It consists of the following methods:

- **The ShowCursor () method:** This method controls the cursor manifestation on the display. It has two values: show (enables the cursor manifestation) and hide (disables the cursor manifestation).
- **The TextOut () method:** This method simply produces the text output, i.e. displays a string on the screen.
- **The ClearScreen () method:** This method clears the screen by the means of changing the video mode.

```
// CDisplay.h
#ifndef__CDISPLAY__
#define__CDISPLAY__
//
// colors for TextOut func
//
#define BLACK         0x0
#define BLUE    0x1
#define GREEN         0x2
#define CYAN    0x3
#define RED           0x4
#define MAGENTA       0x5
#define BROWN     0x6
#define GREY          0x7
#define DARK_GREY    0x8
#define LIGHT_BLUE    0x9
#define LIGHT_GREEN       0xA
```

```
#define LIGHT_CYAN 0xB
#define LIGHT_RED          0xC
#define LIGHT_MAGENTA 0xD
#define LIGHT_BROWN      0xE
#define WHITE      0xF
#include "Types.h"
#include "CString.h"
class CDisplay
{
public:
    static void ClearScreen();
    static void TextOut(
        const char far* inStrSource,
```

```
byte        inX = 0,
        byte        inY = 0,
        byte        inBackgroundColor   = BLACK,
        byte        inTextColor         = WHITE,
        bool        inUpdateCursor      = false
        );
    static void ShowCursor(
        bool inMode
        );
};
#endif //__CDISPLAY__
// CDisplay.cpp
#include "CDisplay.h"
```

```
void CDisplay::TextOut(
        const char far* inStrSource,
        byte        inX,
        byte        inY,
        byte        inBackgroundColor,
        byte        inTextColor,
        bool        inUpdateCursor
        )
{
    byte textAttribute = ((inTextColor) |
(inBackgroundColor << 4));
    byte lengthOfString = CString::Strlen(inStrSource);
    __asm
    {
        push  bp
        mov     al, inUpdateCursor
        xor bh, bh
```

```
        xor   cx, cx
        mov cl, lengthOfString
        mov dh, inY
        mov dl, inX
        mov         es, word ptr[inStrSource + 2]
        mov         bp, word ptr[inStrSource]
        mov ah,  13h
        int    10h
        pop bp
    }
}
```

```
void
CDisplay::ClearScreen()
{
    __asm
    {
        mov     al, 02h
        mov     ah, 00h
        int  10h
    }
}
void CDisplay::ShowCursor(
        bool inMode
        )
{
    byte flag = inMode ? 0 : 0x32;
    __asm
    {
        mov     ch, flag
        mov     cl, 0Ah
        mov     ah, 01h
        int    10h
    }
}
```

5. Types.h implementation

The Types.h header file is a definition container for data types and macros.

```
// Types.h
#ifndef__TYPES__
#define__TYPES__
typedef unsigned char   byte;
typedef unsigned short  word;
typedef unsigned long   dword;
typedef char            bool;
#define true            0x1
#define false           0x0
#endif //__TYPES__
```

6. BootMain.cpp implementation

The BootMain() function serves as a starting point of the program and is its main function. This is where main operations take place.

```
// BootMain.cpp
#include "CDisplay.h"
#define HELLO_STR           "\"Hello, world...\", from low-
level..."
extern "C" void BootMain()
{
    CDisplay::ClearScreen();
    CDisplay::ShowCursor(false);
    CDisplay::TextOut(
        HELLO_STR,
        0,
        0,
        BLACK,
        WHITE,
        false
        );
    return;
}
```

7. StartPoint.asm implementation

```
;_____________________________________________
.286                ; CPU type
;-------------------------------------------------------
.model TINY                 ; memory of model
;--------------------- EXTERNS ---------------------------
extrn       _BootMain:near      ; prototype of C func
;_____________________________________________
;-------------------------------------------------------
.code
org         07c00h    ; for BootSector
main:
      jmp short start       ; go to main
      nop

;---------------------- CODE SEGMENT ----------------------
```

```
start:
      cli
      mov ax,cs             ; Setup segment registers
      mov ds,ax             ; Make DS correct
      mov es,ax             ; Make ES correct
      mov ss,ax             ; Make SS correct
      mov bp,7c00h
      mov sp,7c00h          ; Setup a stack
      sti
                        ; start the program
      call          _BootMain
      ret

      END main              ; End of program
```

8. Assembling

after the development of boot loader code is done, it is time to convert it to a file, which will be able to work on a 16-bit OS – this is a *.com file. Any compiler for Assembler or C/C++ can be started from the command line. After that we pass the required parameters to compilers; as a result, we receive object files. Then a linker comes in. We use it to merge the object files we received into a single executable file. The resulting file is a *.com executable file.

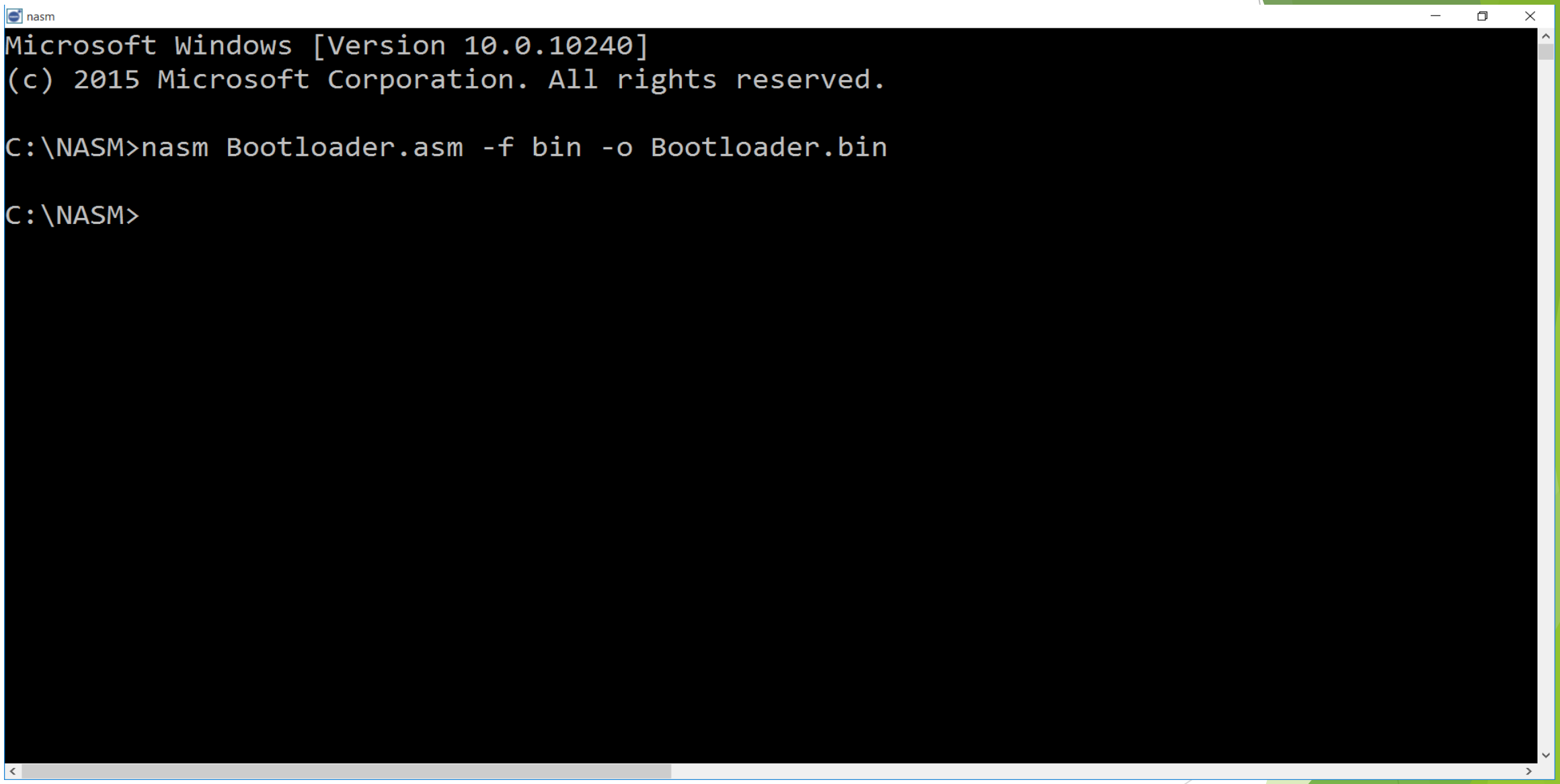
The compilers and the linker must be placed to the folder where the project is saved. In this folder, we need to place a *.bat file.

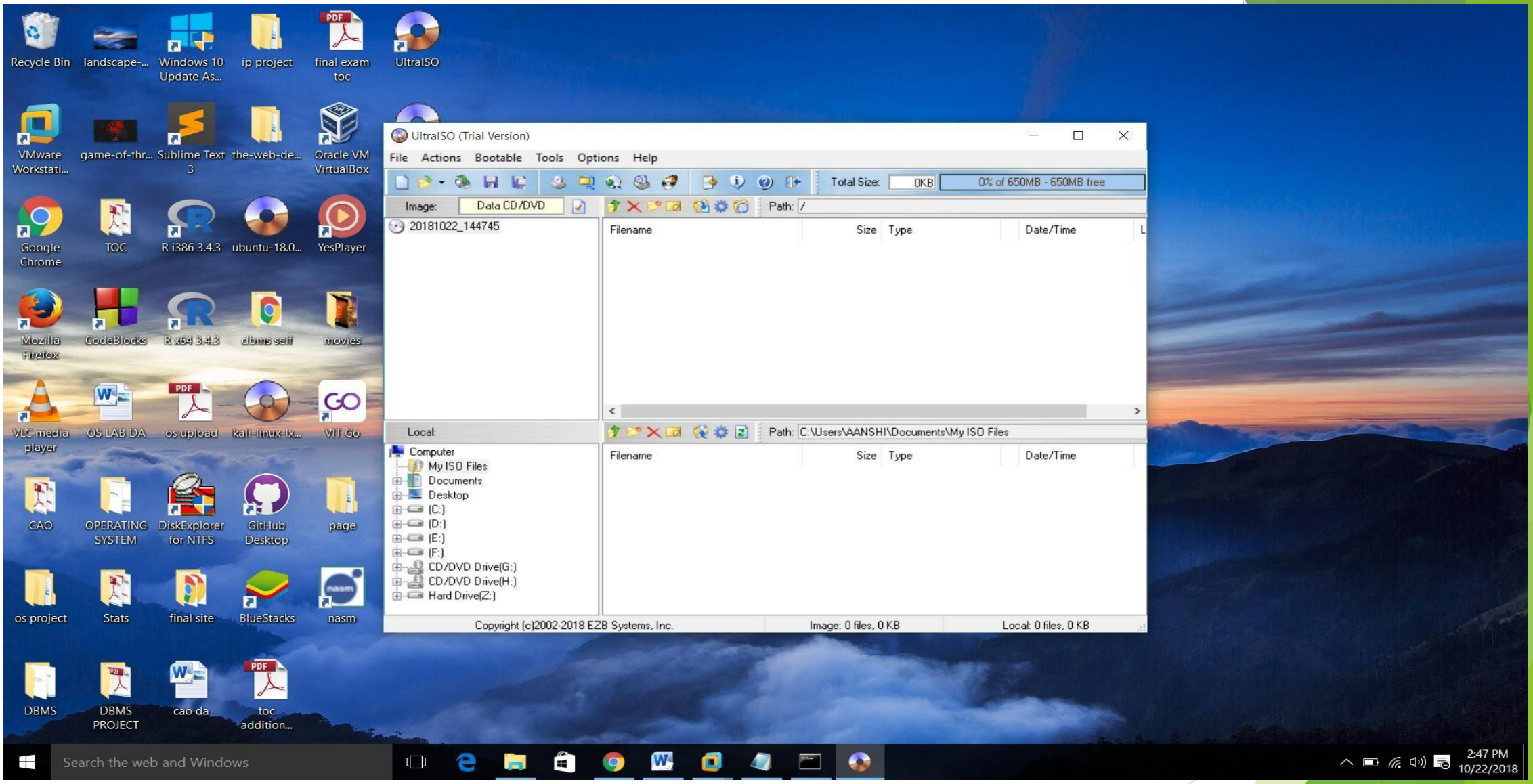
RESULTS

Screenshots of the implementation of bootloader

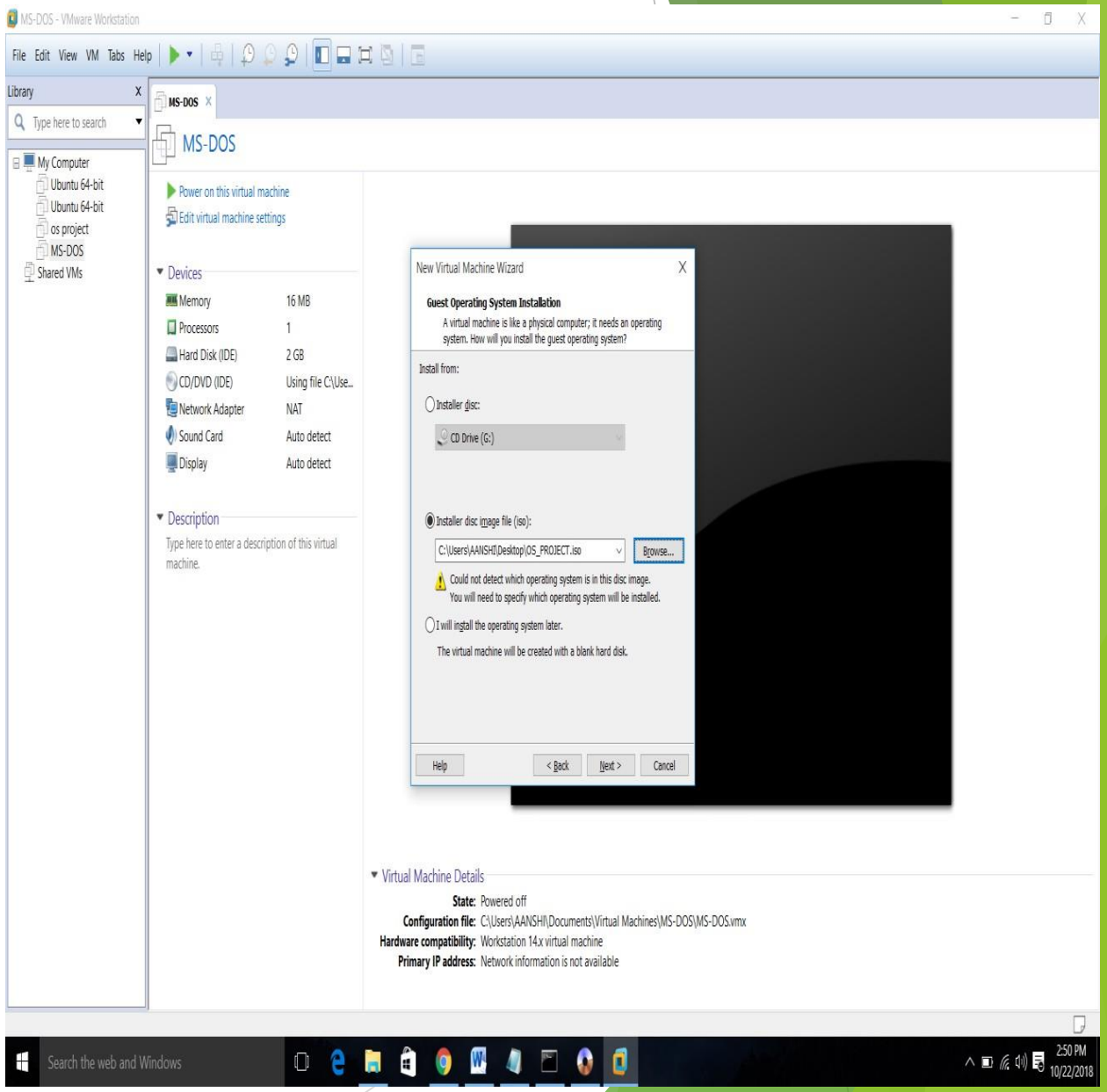
File Edit Format View Help

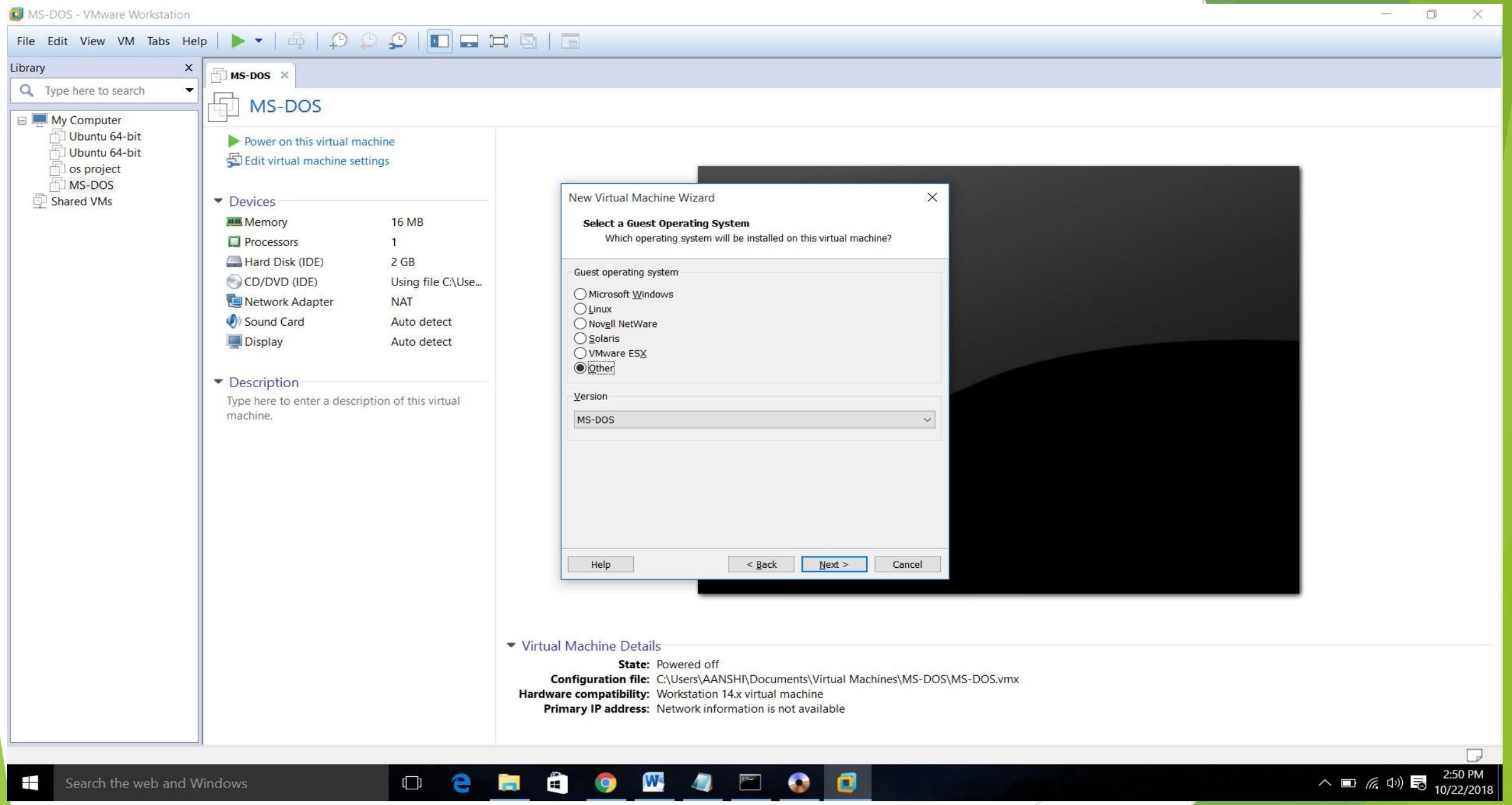
```
        BITS 16

start:
        mov ax, 07C0h           ; Set up 4K stack space after this bootloader
        add ax, 288             ; (4096 + 512) / 16 bytes per paragraph
        mov ss, ax
        mov sp, 4096

        mov ax, 07C0h           ; Set data segment to where we're loaded
        mov ds, ax


        mov si, text_string     ; Put string position into SI
        call print_string       ; Call our string-printing routine

        jmp $                   ; Jump here - infinite loop!



print_string:                   ; Routine: output string in SI to screen
        mov ah, 0Eh             ; int 10h 'print char' function

.repeat:
        lodsb                   ; Get character from string
        cmp al, 0
        je .done                ; If char is zero, end of string
        int 10h                 ; Otherwise, print it
        jmp .repeat

.done:
        ret


        times 510-($-$$) db 0   ; Pad remainder of boot sector with 0s
        dw 0xAA55               ; The standard PC boot signature
```

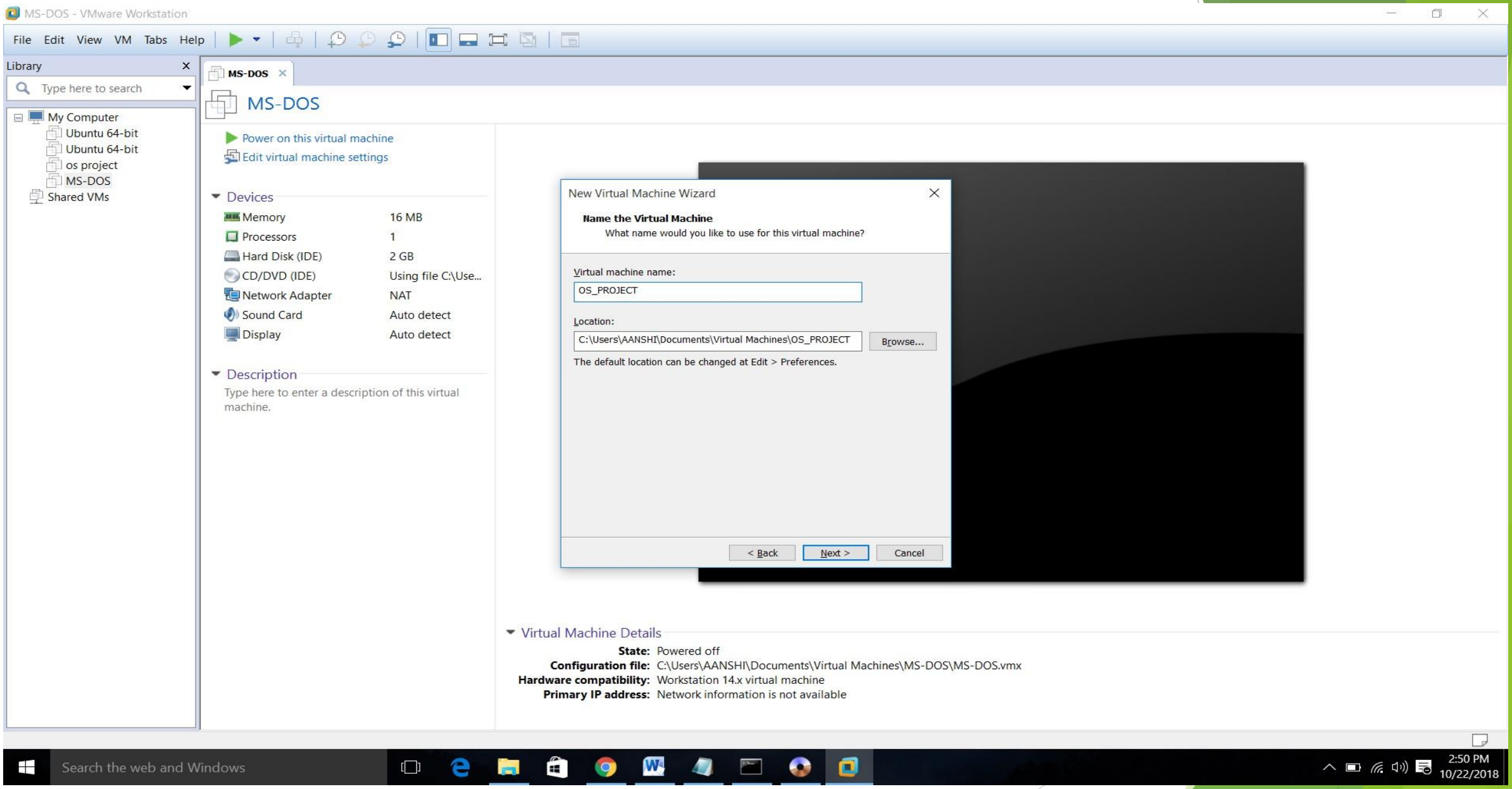

nasm
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.
C:\NASM>nasm Bootloader.asm -f bin -o Bootloader.bin
C:\NASM>

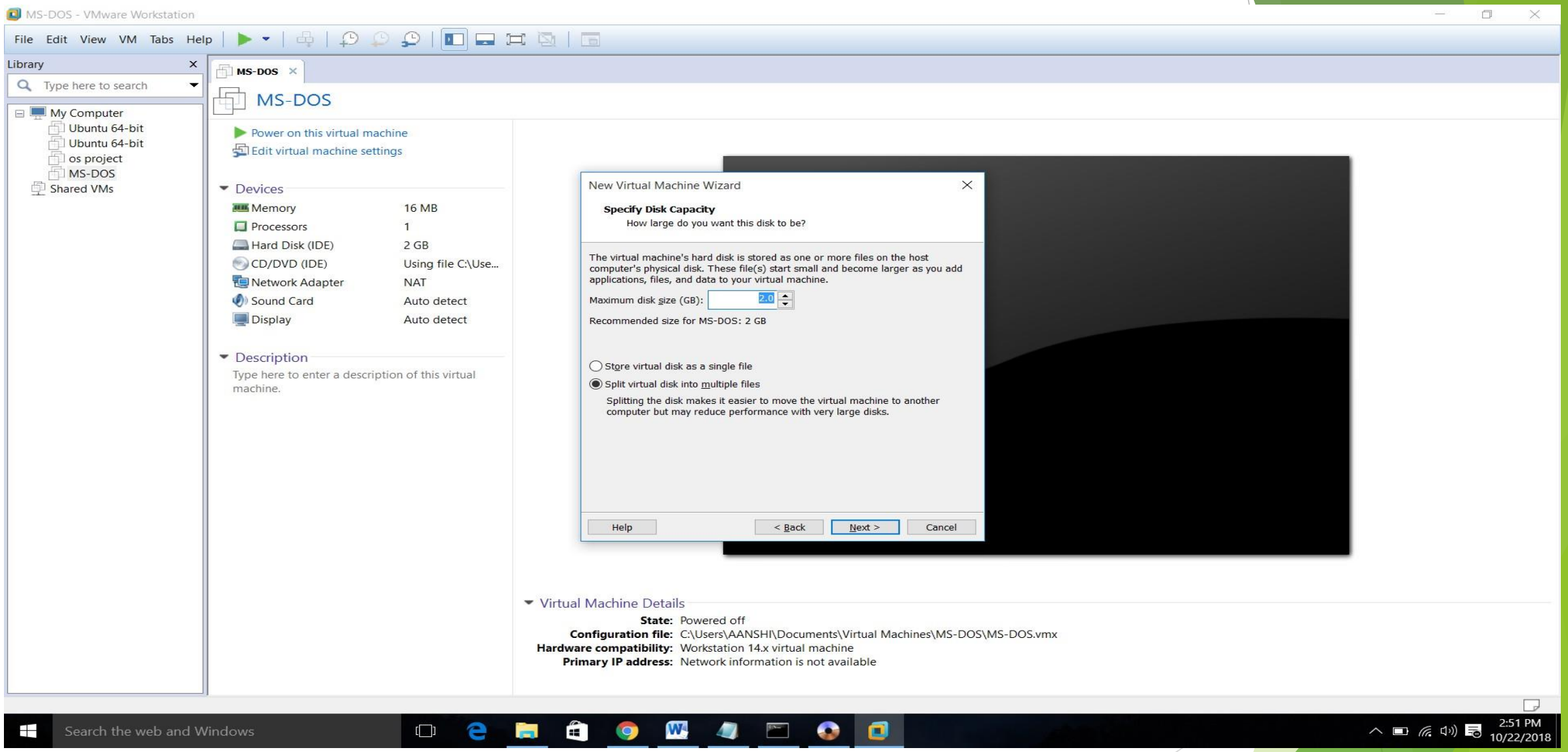

Recycle Bin
landscape-...
Windows 10 Update As...
ip project
final exam toc
UltraISO
VMware Workstati...
game-of-thr...
Sublime Text 3
the-web-de...
Oracle VM VirtualBox
Google Chrome
TOC
R i386 3.4.3
ubuntu-18.0...
YesPlayer
Mozilla Firefox
CodeBlocks
R x64 3.4.3
dbms self
movies
VLC media player
OS LAB DA
os upload
kali-linux-lx...
VIT Go
CAO
OPERATING SYSTEM
DiskExplorer for NTFS
GitHub Desktop
page
os project
Stats
final site
BlueStacks
nasm
DBMS
DBMS PROJECT
cao da
toc addition...
UltraISO (Trial Version)
File Actions Bootable Tools Options Help
Total Size: 0KB
0% of 650MB - 650MB free
Image: Data CD/DVD
Path: /
20181022_144745
Filename Size Type Date/Time
Local:
Path: C:\Users\AANSHI\Documents\My ISO Files
Computer
My ISO Files
Documents
Desktop
(C:)
(D:)
(E:)
(F:)
CD/DVD Drive(G:)
CD/DVD Drive(H:)
Hard Drive(Z:)
Copyright (c)2002-2018 EZB Systems, Inc.
Image: 0 files, 0 KB
Local: 0 files, 0 KB
Search the web and Windows
2:47 PM
10/22/2018

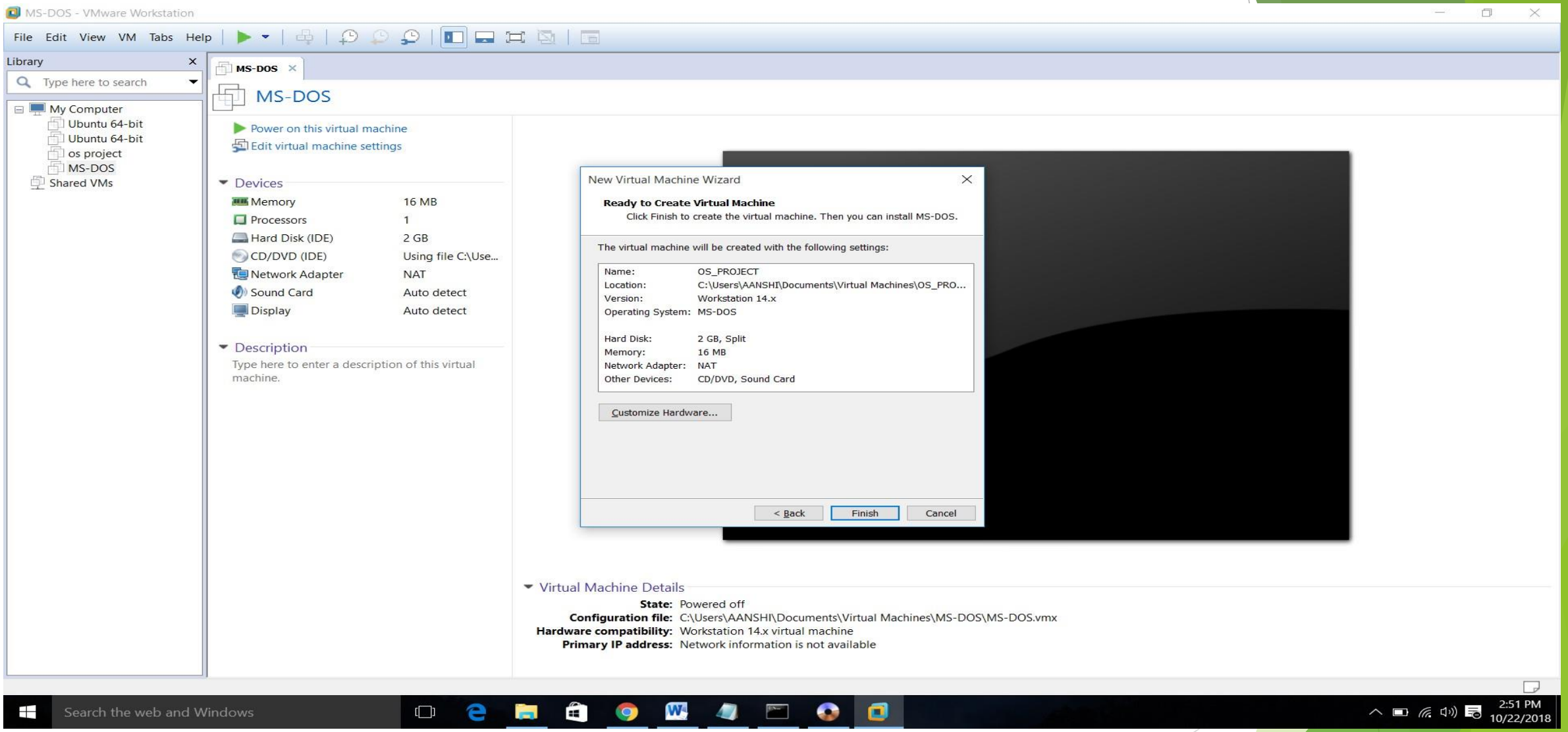
TESTING BOOTLOADER ON THE VIRTUAL MACHINE




MS-DOS - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to search
My Computer
Ubuntu 64-bit
Ubuntu 64-bit
os project
MS-DOS
Shared VMs
MS-DOS
MS-DOS
Power on this virtual machine
Edit virtual machine settings
Devices
Memory 16 MB
Processors 1
Hard Disk (IDE) 2 GB
CD/DVD (IDE) Using file C:\Use...
Network Adapter NAT
Sound Card Auto detect
Display Auto detect
Description
Type here to enter a description of this virtual machine.
New Virtual Machine Wizard
Select a Guest Operating System
Which operating system will be installed on this virtual machine?
Guest operating system
Microsoft Windows
Linux
Novell NetWare
Solaris
VMware ESX
Other
Version
MS-DOS
Help
< Back
Next >
Cancel
Virtual Machine Details
State: Powered off
Configuration file: C:\Users\AANSHI\Documents\Virtual Machines\MS-DOS\MS-DOS.vmx
Hardware compatibility: Workstation 14.x virtual machine
Primary IP address: Network information is not available
Search the web and Windows
2:50 PM
10/22/2018


MS-DOS - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to search
My Computer
Ubuntu 64-bit
Ubuntu 64-bit
os project
MS-DOS
Shared VMs
MS-DOS
MS-DOS
Power on this virtual machine
Edit virtual machine settings
Devices
Memory 16 MB
Processors 1
Hard Disk (IDE) 2 GB
CD/DVD (IDE) Using file C:\Use...
Network Adapter NAT
Sound Card Auto detect
Display Auto detect
Description
Type here to enter a description of this virtual machine.
New Virtual Machine Wizard
Name the Virtual Machine
What name would you like to use for this virtual machine?
Virtual machine name:
OS_PROJECT
Location:
C:\Users\AANSHI\Documents\Virtual Machines\OS_PROJECT
Browse...
The default location can be changed at Edit > Preferences.
< Back
Next >
Cancel
Virtual Machine Details
State: Powered off
Configuration file: C:\Users\AANSHI\Documents\Virtual Machines\MS-DOS\MS-DOS.vmx
Hardware compatibility: Workstation 14.x virtual machine
Primary IP address: Network information is not available
Search the web and Windows
2:50 PM
10/22/2018


MS-DOS - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to search
My Computer
Ubuntu 64-bit
Ubuntu 64-bit
os project
MS-DOS
Shared VMs
MS-DOS
MS-DOS
Power on this virtual machine
Edit virtual machine settings
Devices
Memory 16 MB
Processors 1
Hard Disk (IDE) 2 GB
CD/DVD (IDE) Using file C:\Use...
Network Adapter NAT
Sound Card Auto detect
Display Auto detect
Description
Type here to enter a description of this virtual machine.
New Virtual Machine Wizard
Specify Disk Capacity
How large do you want this disk to be?
The virtual machine's hard disk is stored as one or more files on the host computer's physical disk. These file(s) start small and become larger as you add applications, files, and data to your virtual machine.
Maximum disk size (GB): 2.0
Recommended size for MS-DOS: 2 GB
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
Virtual Machine Details
State: Powered off
Configuration file: C:\Users\AANSHI\Documents\Virtual Machines\MS-DOS\MS-DOS.vmx
Hardware compatibility: Workstation 14.x virtual machine
Primary IP address: Network information is not available
Search the web and Windows
2:51 PM
10/22/2018


MS-DOS - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to search
My Computer
Ubuntu 64-bit
Ubuntu 64-bit
os project
MS-DOS
Shared VMs
MS-DOS
MS-DOS
Power on this virtual machine
Edit virtual machine settings
Devices
Memory 16 MB
Processors 1
Hard Disk (IDE) 2 GB
CD/DVD (IDE) Using file C:\Use...
Network Adapter NAT
Sound Card Auto detect
Display Auto detect
Description
Type here to enter a description of this virtual machine.
New Virtual Machine Wizard
Ready to Create Virtual Machine
Click Finish to create the virtual machine. Then you can install MS-DOS.
The virtual machine will be created with the following settings:
Name: OS_PROJECT
Location: C:\Users\AANSHI\Documents\Virtual Machines\OS_PRO...
Version: Workstation 14.x
Operating System: MS-DOS
Hard Disk: 2 GB, Split
Memory: 16 MB
Network Adapter: NAT
Other Devices: CD/DVD, Sound Card
Customize Hardware...
< Back
Finish
Cancel
Virtual Machine Details
State: Powered off
Configuration file: C:\Users\AANSHI\Documents\Virtual Machines\MS-DOS\MS-DOS.vmx
Hardware compatibility: Workstation 14.x virtual machine
Primary IP address: Network information is not available
Search the web and Windows
2:51 PM
10/22/2018

CONCLUSION

Thus it can be concluded that the boot loader can affect the efficiency of the device. Since it's the first thing that is executed as soon as the system is turned on. So having a efficient and a time saving boot loader is a must.

- An efficient boot loader has a lesser boot time
- Lesser boot time will also ensure higher utilisation of CPU
- An efficient boot loader also sees to the fact that the memory is well manged and appropriately utilised

REFERENCES

- wikipedia.org
- ieeexplore.org
- researchgate.net

THANK YOU