Guide to develop Swayam

January 1, 2020

By: Rahul Khandebharad

References

Documentations:

1. Writing a Simple Operating System — from Scratch

   By Nick Blundell (School of Computer Science, University of Birmingham, UK)

2. The little book about OS development

   By Erik Helin, Adam Renberg

3. Operating System Development Series

   By Mike (BrokenThron Entertainment)

4. How To Program Your Very Own Operating Systems (OS)

   By Jon Penland

YouTube Channels:

1. IKnow
2. Tinkernut
3. Write your own Operating System
4. TechMedia
5. pritam zope
6. hkdwrld

Content

History

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 faucet 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.

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.

1950's

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 (or just "DOS") was a Disk Operating System was originally announced by IBM to be released on the last day of 1964. Do 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

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 (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

The first **Windows 1.0** 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

The **Windows 2.0** 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.

1987

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 is supports **multitasking** with DOS programs.

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

Theory

Basic Concepts

- Memory Management
- Program Management
- Multitasking
- Memory Protection
- Fixed Base Address
- Multiuser
- Kernel
- File System
- Command Shell
- Graphical User Interface (GUI)
- Graphical Shell
- Linear Block Addressing (LBA)
- Bootloader

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**.

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).

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. Another new term!**
- Executing multiple programs simultaneously.

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.

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.

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.

Kernel

The Kernel is the heart of the Operating System. It provides the basic foundation, memory management, file systems, program execution, etc.

File System

In OS Development, there is no such thing as a "file". Everything could be pure binary code (from the boot sector); from the start.

A File System is simply a specification that describes information regarding files. In most cases, this refers to Clusters, Segments, segment address, root directories, etc. the OS has to find the exact starting address of the file in order to load it.

File Systems also describe file names. There are **external** and **internal** file names. For example, the FAT12 specification states a filename can only be 11 characters. No more, no less.

Command Shell

A Command Shell sits on top the Kernel as a separate program. The Command Shell provides basic input and output through the use of typing commands. The Command Shell uses the Kernel to help with this, and complete low level tasks.

Graphical User Interface (GUI)

The Graphical User Interface (GUI) simply refers to the graphical interface and interactions between the Graphical Shell and the user.

Graphical Shell

The Graphical Shell provides video routines and low level graphical abilities. It normally will be executed by the Command Shell. (As in Windows 1.0, 2.0, and 3.0). Usually this is automatic these days, however.

Preface

How to use C in kernel land

16 bit and 32 bit C

At power on, the system is also operating in 16 bit **real mode** which 32 bit compilers do not support. This is the first important thing: 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.

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.

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 or **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. One way is used in Windows. 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.

Dynamic Memory Allocation

In the application programming world, we would normally call **malloc()** and **free()** or **new** and **delete** to allocate a block of memory from the heap. This is different in the system programming world.

The important thing here is that there is no dynamic memory allocation. Dynamic memory allocation in C and C++ are **system services** and require an OS to be running.

We will need to write our own memory management services and routines in order to be able to provide a malloc() and free() or new and delete.

Until then, the only way to "allocate" a buffer is to use some unused location in the address space.

Inline Assembly

The idea behind **Inline Assembly** is to embed assembler instructions in our C/C++ code, using the asm keyword, when there's no option but to use Assembly language.

There are some things that C cannot natively do. We will need to use assembly language for system services and talking to hardware devices.

Most compilers provide a keyword that allows inline assembly.

Standard Library and the Run Time Library (RTL)

The RTL is the set of services and routines that our application program uses at run time. These, by their nature, require an OS to already be running and to support them. Because of this, we will need to develop our own RTL.

The startup RTL code is responsible for calling C++ constructors and destructors. If we want to use C++, we must develop the RTL code to support it. This uses compiler extensions.

We can use external libraries - if and only if those routines do not use system services. Anything like printf(), scanf(), memory routines, or, virtually everything but the bare minimum routines can be used. About 90% of it will be needed to be rewritten for your own OS, so it is best to write your own.

Fixing Errors

Debugging

We use the Bochs Debugger, which is a debugger that comes with the Bochs emulator. This can be used to run our OS as well as for aiding in fixing most of the more common errors that you may run into.

The only other way is to develop our own routines that will allow us to output information. At the most this might be able to tell us how far the software gets to before crashing.

Introduction

What is an Operating System?

An Operating System is not a single program, but a collection of software that work and communicate with each other. So it is called an "Operating Environment".

Because Operating Systems are a collection of software, in order to develop an Operating System, one must know computer programming.

Requirements

Knowledge of the C Programming Language

Using a high level language, such as C, can make OS development much easier. The most common languages that are used in OS development are C, C++, and Perl.

C and C++ are the most common, with C being the most used. Because the C programming language was originally *designed* for system level and embedded software development.

Because of this, we are going to be using C for most of the OS.

Knowledge of x86 Assembly Language

80x86 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.

Tools

In developing low level code, we will need specialized low level software to help us out.

NASM - The Assembler

The Netwide Assembler (NASM) can generate flat binary 16bit programs, while most other assemblers (Turbo Assembler (TASM), Microsoft's Macro Assembler (MASM)) cannot.

During the development of the OS, some programs must be pure binary executables. Because of this, NASM is a great choice to use.

Microsoft Visual C++

Because portability is a concern, most of the code for our operating system will be developed in C.

During OS Development, there are some things that we must have control over that not all compilers may support, however Depending on the design of your system, we may also need to support or change more detailed properties: Such as loading at a specific address, adding your own internal sections in your programs' binary, etc.) The basic idea is that not all compilers out there are capable of developing operating system code.

Copying the Boot Loader

The bootloader is a pure binary program that is stored in a single 512 byte sector. It is a very important program as it is impossible to create an OS without it. It is the very first program of your OS that is loaded directly by the BIOS, and executed directly by the processor.

There are a lot of ways we can do this. Here there are two ways stated:

1. PartCopy - Low Level Disk Copier

PartCopy allows the copying of sectors from one drive to another.

PartCopy stands for "Partial copy". Its function is to copy a certain number of sectors from one location to another, to and from a specific address.

2. Windows DEBUG Command

Windows provides a small command line debugger that may be used through the command line. There are quite a bit of different things that we can do with this software, but all we need it to do is copy our boot loader to the first 512 bytes on disk.

VFD - Virtual Floppy Drive

Weather you have a floppy drive or not, this program is very useful. It can simulate a real floppy drive from a stored floppy image, or even in RAM. This program creates a virtual floppy image, allows formatting, and copying files directly using Windows Explorer.

QEMU

QEMU is a hosted virtual machine monitor: it emulates the machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems.

The Build Process

There are a lot of tools listed above. To better understand how they can be useful, we should take a look at the entire build process of the OS.

- **Setting everything up**
    1. Use VFD to create and format a virtual floppy image to use.
    2. Set up Bochs Emulator to boot from the floppy image.
- **The bootloader**
    1. Assemble the bootloader with NASM to create a flat binary program.
    2. Use PartCopy or the DEBUG command to copy the bootloader to the bootsector of the virtual floppy image.
- **The Kernel (And basically all other programs)**
    1. Assembly and/or compile all sources into an object format (Such as ELF or PE) that can be loaded and executed by the boot loader.
    2. Copy kernel into floppy disk using Windows Explorer.
- **Test it!**
    1. Using Qemu , using a real floppy disk, or by using MagicISO to create a bootable CD.

Bootloaders (Part-I)

The Boot Process

Pressing the power button

When the power button is pressed, the wires connected to the button send an electronic signal to the motherboard. The motherboard simply reroutes this signal to the power supply (PSU).

This signal contains a single bit of data. If it is 0, there is, of course, no power (so the computer is off, or the motherboard is dead). If it is a 1 (meaning an active signal), it means that power is being supplied

When the PSU receives this active signal, it begins supplying power to the rest of the system. When the correct amount of power is supplied to all devices, the PSU will be able to continue supplying that power without any major problems.

The PSU then sends a signal, called the "power good" signal into the motherboard to the Basic Input Output System (BIOS).

BIOS POST

When the BIOS receive this "power good" signal, the BIOS begins initializing a process called POST (Power On Self Test). The POST then tests to insure there is good amount of power being supplied, the devices installed (such as keyboard, mouse, USB, serial ports, etc.), and insures the memory is good (By testing for memory corruption).

The POST then gives control to the BIOS. The POST loads the BIOS at the end of memory (Might be 0xFFFFF0) and puts a jump instruction at the first byte in memory.

The processors Instruction Pointer (CS:IP) is set to 0, and the processor takes control.

This mean the processor starts executing instructions at address 0x0. In this case, it is the jump instruction placed by the POST. This jump instruction jumps to 0xFFFFF0 (or wherever the BIOS was loaded), and the processor starts executing the BIOS.

The BIOS takes control...

The BIOS

The Basic Input Output System (BIOS) does several things. It creates an Interrupt Vector Table (IVT), and provides some basic interrupt services. The BIOS then does some more tests to insure there is no hardware problem. The BIOS also supplies a Setup utility.

The BIOS then needs to find an OS. Based on the boot order that you set in the BIOS Setup, the BIOS will execute Interrupt (INT) 0x19 to attempt to find a bootable device.

If no bootable device is found (INT 0x19 returns), the BIOS goes on to the next device listed in the boot order. If there is no more devices, it will print an error similar to "No Operating System found" and halt the system.

Interrupts and the Interrupt Vector Table (IVT)

An Interrupt is a subroutine that can be executed from many different programs. These interrupts are stored at address 0x0 into a table called the Interrupt Vector Table. A common interrupts, for example, is **INT 0x21** used for DOS.

Note: There is no DOS! The *Only* interrupts available are the interrupts set up by the BIOS, and no more! The use of other interrupts will cause the system to execute a nonexistent routine, causing your program to crash.

Note: If you switch processor modes, the IVT will not be available. This means absolutely *no* interrupts--neither software nor hardware, will be available, Not even the BIOS.. For a 32 bit OS, we are going to have to do this. Not yet, though.

BIOS Interrupt 0x19

INT 0x19 - SYSTEM: BOOTSTRAP LOADER

Reboots the system through a Warm Reboot without clearing memory or restoring the Interrupt Vector Table (IVT).

This interrupt is executed by the BIOS. It reads the first sector (Sector 1, Head 0, Track 0) of the first hard disk.

Sectors

A "Sector" simply represents a group of 512 bytes. So, Sector 1 represents the first 512 bytes of a disk.

Heads

A "Head" (or Face) represents the side of the disk. Head 0 is the front side, Head 1 is the back side. Most disks only have 1 side, hence only 1 head ("Head 1")

Tracks

In this picture, This disk could represent a hard disk or floppy disk. Here, we are looking at Head 1 (The front side), and the Sector represents 512 bytes. A Track is a collection of sectors.

Note: Remember that 1 sector is 512 bytes, and there are 18 sectors per track on floppy disks. This will be important when loading files.

If the disk is bootable, **then the boot sector will be loaded at 0x7C00**, and INT 0x19 will jump to it, thereby giving control to the bootloader.

Note: Remember that the bootloader is loaded at 0x7C00. This is important!

Note: On some systems, you can also execute a warm boot by putting 0x1234 at address 0x0040:0072, and jumping to 0xFFFF:0. For a cold reboot, store 0x0 instead.

Bootloader Theory

In Assembly Language, we can very easily go beyond the 512 byte mark. So, the code could look just fine, but only a **part** of it will be in memory. For example, consider this:

```
mov     ax, 4ch

inc     bx      ; 512 byte

mov     [var], bx  ; 514 byte
```

In Assembly language, execution begins from the top of the file downward. However, remember that, when loading files in memory, we are loading sectors. Each of these sectors is 512 bytes, so it will only copy 512 bytes of the file into memory.

If the above code was executed, and only the first sector was loaded in memory, it will only copy up to the 512 byte (The **inc bx** instruction). So, while the last mov instruction is still on disk, **It isn't in memory!**.

What will the processor do after **inc bx** then? It will still continue on to the 514 byte. As this was not in memory, **It will execute past the end of our file!** The end result? A crash.

However, it is possible to load the second sector (or more) at a given address and execute it. Then the rest of the file will be in memory, and everything will work just fine.

This approach will work, but it will be hard hacked. The most common approach is keeping the bootloader at 512 bytes in size, searching, loading, and executing a second stage bootloader.

Hardware Exceptions

Hardware Exceptions are just like Software Exceptions, however the **processor** will execute them rather then software.

There are times when one must stop all exceptions for happening. For example, when switching computer modes, the entire Interrupt Vector Table is not available, so **any interrupt-hardware or software, will cause your system to crash**.

CLI and STI Instructions

You can use the STI and CLI instructions to enable and disable all interrupts. Most systems do not allow these instructions for applications as it can cause big problems (Although systems can emulate them).

```
cli             ; clear interrupts

; do something...

sti             ; enable interrupts--we're in the clear!
```

Double Fault Hardware Exception

If the processor finds a problem during execution (Such as an invalid instruction, division by 0, etc.) It executes a Second Fault Exception Handler (Double Fault), which is Interrupt 0x8.

We will be looking a Double Faults later. If the processor still cannot continue after a double fault, it will execute a **Triple Fault**.

Triple Fault

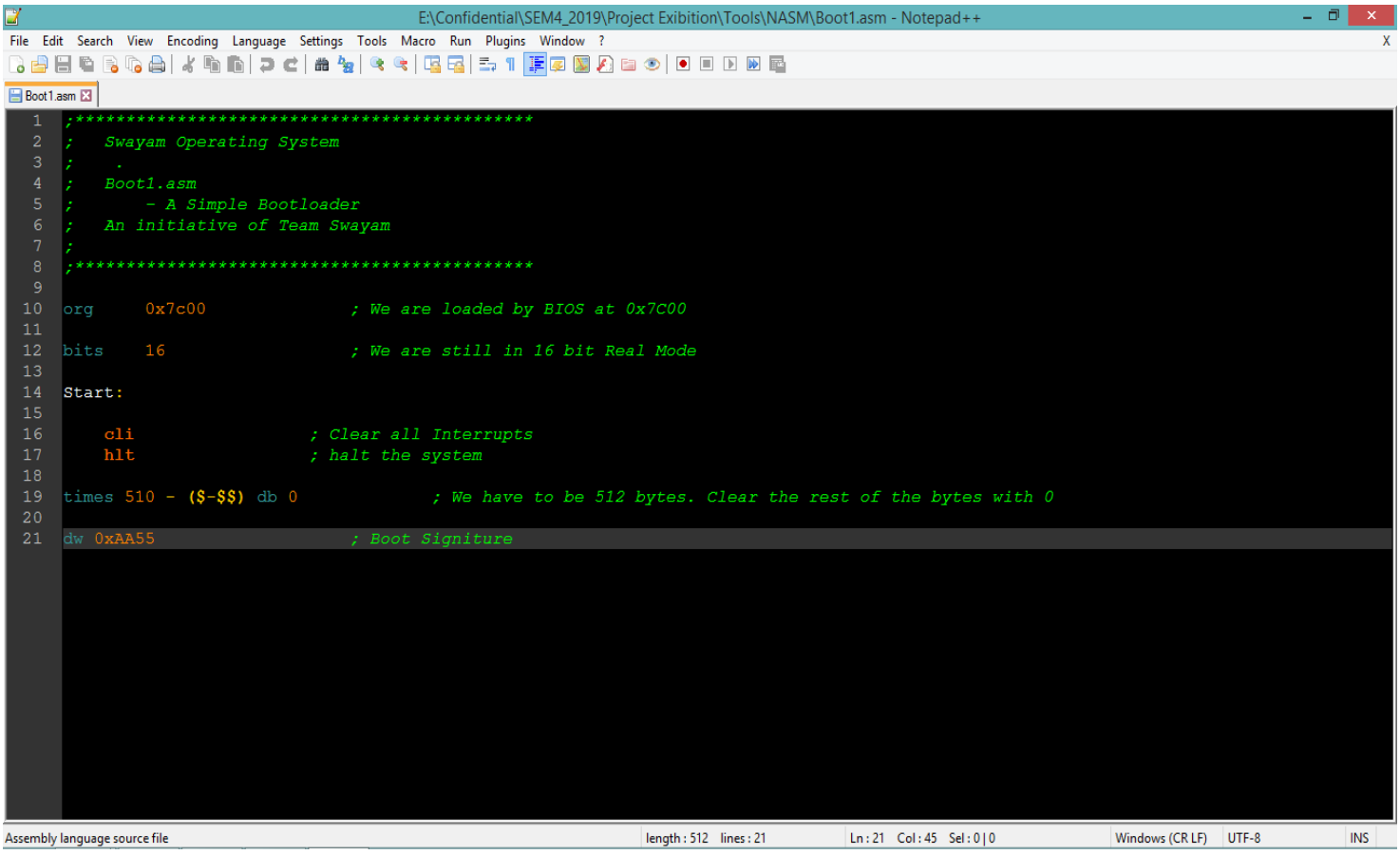
In a CPU "Triple Faults" simply means the system hard reboots.

In early stages, such as the bootloader, whenever there is a bug in your code, the system will triple fault. This indicates a problem in your code.

Developing a simple Bootloader

Open up any ordinary text editor but Notepad will be sufficient.

Here's the bootloader (Boot1.asm)...



Assembling with NASM

NASM is a command line assembler, and hence must be executed either through command line or a batch script. To assemble **Boot1.asm** do this:

```
nasm -f bin Boot1.asm -o Boot1.bin
```

The **-f** option is used to tell NASM what type of output to generate. In this case, it is a binary program.

-o option is used to give your generated file a different output name. In this case, its **Boot1.bin**

After assembling, you should have an exact 512 byte file named "Boot1.bin".

Note: For some reason, Windows Explorer limits displaying file sizes to 1 KB. Just see the properties of the file, and it should say 512 bytes.

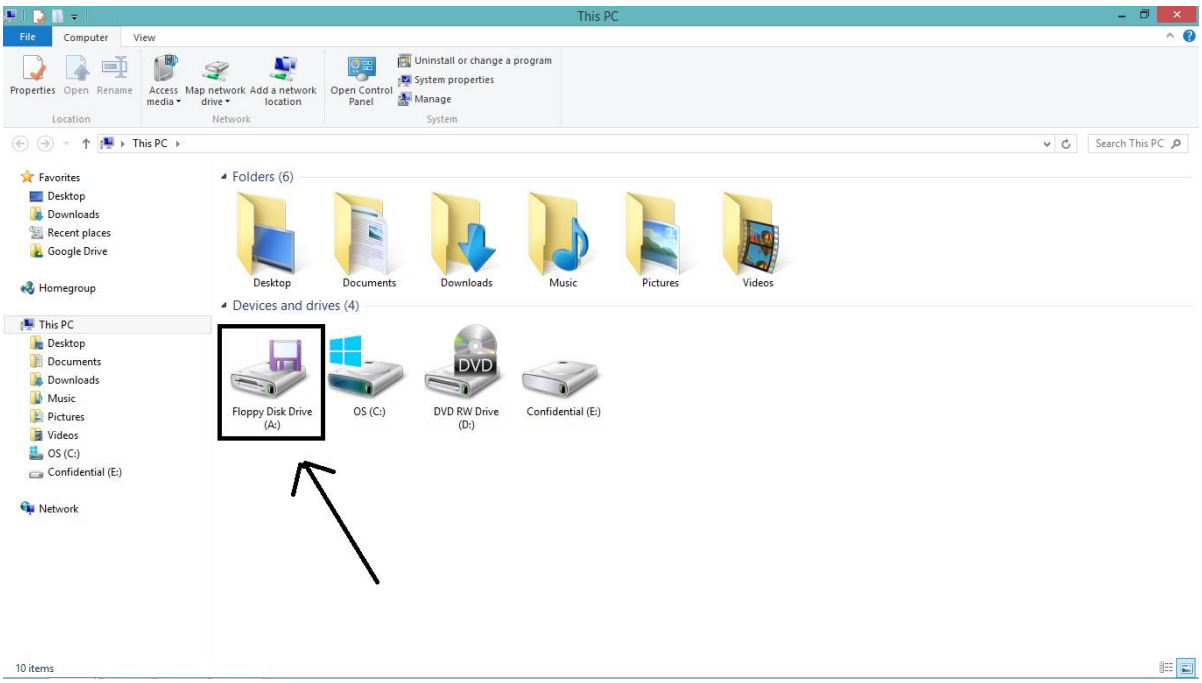
How to use VFD (Virtual Floppy Drive)

We will use VFD to create a virtual floppy image to copy our OS to. This will explain how to use it.

1. Open vfdwin.exe.
2. Under the Driver tab, click the Start button. This starts the driver.
3. Click either the Drive0 or Drive1 tab.
4. Click Open

Insure Media Type is a standard 3.5" 1.44 MB floppy, and disk type is in RAM. Also, insure Write Protect is disabled. Click "Create".

Go to My Computer and you should see a new floppy drive as shown below:



Part Copy - Copying to the Boot sector

PartCopy is a command line program. It uses the following syntax:

```
Partcopy  file first_byte  last_byte  drive
```

PartCopy can be used for more then just copying files. It can be used for copying certain bytes to and from sectors. Thinking of its format (Shown above) is a safe method.

Because you have an emulated floppy drive, you can reference the drive name by letter (Like A:).

To copy our bootloader, this will work:

```
partcopy  Boot1.bin  0 200  -f0
```

f0 represents Floppy Disk 0. You can change between f0, f1; etc. based on what drive your floppy disk is in. Boot1.bin is our file to copy. This copies from the first byte (0x0) of the file to the last byte (0x200, which is 512 decimal). Notice that partcopy only accepts hexadecimal numbers.

Warning: Remember using this program can cause permanent disk corruption if you are not careful. The above command line commands will only work for floppy disks. Do not attempt to try on hard disks!

QEMU: Emulator and Bootloader

QEMU can save and restore the state of the virtual machine with all programs running. Guest operating systems do not need patching in order to run inside QEMU.

QEMU supports the emulation of various architectures, including:

- IA-32 (x86) PCs
- x86-64 PCs etc

The virtual machine can interface with many types of physical host hardware, including the user's hard disks, floppy disk, CD-ROM drives, network cards, audio interfaces, and USB devices.

The following command will emulate the Bootloader on system:

```
qemu-system-x86_64.exe Boot1.bin
```

Locating the BIOS ROM

A lot of the lines in the configuration file are very simple. There is however one line that we need to look at here however:

```
romimage:    file=BIOS-bochs-latest, address=0xf0000
```

This line tells Qemu where to place the BIOS in its memory (Virtual RAM). Remember that BIOS sizes may differ, Also remember that the BIOS must end at the end of the first megabyte (0xFFFFF) in memory?

Because of this, you may need to change this line to reposition the Bios. This can be done by getting the size of the Bios image (It should be named **BIOS-Qemu-latest** in your Qemu directory). Get the size in bytes.

After this, simply subtract 0xFFFFF - size of qemu file (in bytes). This will be the new Bios address, so update the **address** on this line to move the Bios to its new location.

We may or may not need to do this step. If you get an error from qemu telling you that the Bios must end at 0xFFFFF, then you do need to complete this step and it should work.

We will get the following output when we load our bootloader and test it.