BrokenThorn Entertainment

Operating Systems Development Series

Operating Systems Development - Introduction

by Mike, 2008, Updated 2010

This series is intended to demonstrate and teach operating system development from the ground up.

What is this about?

Operating systems can be a very complex topic. Learning how operating systems work can be a great learning experience.

The purpose of this series is to teach the black art of Operating System (OS) Development, from the ground up. Whether you want to make your own OS, or simply to learn how they work, this series is for you.

What is an Operating System?

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.

If you have never programmed before

Computer programming is designing and writing software, or programs, for the computer to load and execute. However, the Operating System needs to be designed with this functionality.

An Operating System is not a single program, but a collection of software that work and communicate with each other. This is what I mean by "Operating Environment".

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

If you have never programmed before, take a look at the Requirements section below, and look no further. This section will have links to good tutorials and articles that could help you to learn computer programming with C++ and 80x86 Assembly Language.

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. Do not think these are the only languages that may be used; It is possible in other languages. I have even seen one with FreeBASIC! Getting higher level languages to work properly can also make it harder to work within the long run, however.

C and C++ are the most common, with C being the most used. C, as being a middle level language, provides high level constructs while still providing low level details that are closer to assembly language, and hence, the system. Because of this, using C is fairly easy in OS development. This is one of the primary reasons why it is the most commonly 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.

C is a complex programming language, that can take a book to cover. If you do not know C, the following may help:

- Learn C – A complete resource for a beginner
- cprogramming.com
- Thinking in C++

I personally learned from the original "The C++ Programming language", which is now obsolete, though.

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.

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.

Assembly Language is another complex language that can take a book to fill. If you do not know x86 Assembly Language, the following may help:

- Assembly Language: Step by Step
- Art of Assembly

I personally learned from Assembly Language Step by Step (Excellent beginning book) and the Art of Assembly Language. Both are very great books.

Getting ready

That is all you need to know--Everything else I'll teach along the way. Be forewarned: From here on out, I will not be explaining C or x86 Assembly Language concepts. I will still explain new instructions that you may not be familiar with, such as **lgdt**, and the use of **sti, cli, bt, cpuid** and some others, however.

Tools of the trade

In developing low level code, we will need specialized low level software to help us out. Some of these tools are not needed, however, they are highly recommended as they can significantly aid in development.

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.

You can download NASM from here.

Microsoft Visual C++ 2005 or 2008

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. For example, say good bye to all runtime compiler support (templates, exceptions) and the good old standard library! Depending on the design of your system, you 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.

I will be using Microsoft Visual C++ for developing the system. However, it is also possible to develop in other compilers such as DJGPP, GCC or even Cygwin. Cygwin is a command shell program that is designed to emulate Linux command shell. There is a GCC port for Cygwin.

You can get Visual C++ 2008 from here

You can also still get Visual C++ 2005 from here.

Support for other compilers

As previously stated, it is possible to develop an operating system using other compilers. While my primary compiler of use will be Visual C++, I will explain on how to setup the working environments so that you will be able to use your favorite compiler.

Currently, I plan on describing on setting up the environments for:

- DJGPP
- Microsoft Visual Studio 2005
- GCC

  I will also try to support the following compilers, if possible:

- Mingw
- Pelles C

If you would like to add more to this list, please contact me.

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.

We can use NASM to assemble the program, but how do we get it on a floppy disk? We cannot just copy the file. Instead, we have to overwrite the boot record that Windows places (after formatting the disk) with our bootloader. Why do we need to do this? Remember that the BIOS only looks at the bootsector when finding a bootable disk. The bootsector, and the "boot record" are both in the same sector! Hence, we have to overwrite it.

There are alot of ways we can do this. Here, I will present two. If you are unable to get one method working on your system, our readers may try the other method.

Warning: Do Not attempt to play with the following software until I explain how to use it. Using this oftware incorrectly can corrupt the data on your disk or make your PC unable to boot.

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.

You can download it from here.

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.

Go to the command prompt, and type **debug**. You will be greeted by a little prompt (-):

```
C:\Documents and Settings\Michael>debug
-
```

Here is where you enter your commands. **h** is the help command, **q** is the quit command. The **w** (write) command is the most important for us.

You can have debug load a file into memory such as, say, our boot loader:

```
C:\Documents and Settings\Michael>debug boot_loader.bin
-
```

This allows us to perform operations on it. (We can also use debugs **L (Load)** command to load the file is we wanted to). In the above example, **boot_loader.bin** will be loaded at address 0x100.

To write the file to the first sector of our disk, we need to use the **W (Write)** command which takes the following form:

```
W [address] [drive] [firstsector] [number]
```

Okay... so let's see: The file is at address 0x100. We want the floppy drive (Drive 0). The first sector is the first sector on the disk (sector 0) and the number of sectors is ehm... 1.

Putting this together, this is our command to write **boot_loader.bin** to the boot sector of a floppy:

```
C:\Documents and Settings\Michael>debug boot_loader.bin
-w 100 0 0 1
-q
```

If you would like to learn more about this command, take a look at this tutorial.

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 (Such as, your kernel perhaps?) directly using Windows Explorer.

You can download it from here.

Bochs Emulator - PC Emulator and Debugger

You pop in a floppy disk into a computer, in hopes that it works. You boot your computer and look in aw at your greatest creation! ...Until your floppy motor dies out because you forgot to send the command to the controller in your bootloader.

When working with low level code, it is possible to destroy hardware if you are not careful. Also, to test your OS, you will need to reboot your computers hundreds of times during development.

Also, what do you do if the computer just reboots? What do you do if your Kernel crashes? Because there is no debugger for your OS, it is virtually impossible to debug.

The solution? A PC Emulator. There are plenty available, two of them being VMWare and Bochs Emulator. I will be using Bochs and Microsoft Virtual PC for testing.

You can download Bochs from here.

Thats all, fokes

You do not need to know how to use the software I listed. I will explain how to use them as we start using them.

If you would like to run your system on a real computer that does not have a floppy drive, it is still possible to boot from CD even though it is a floppy image. This is done through **Floppy Emulation** that which most of BIOSs support.

Simply get a CD burning software (I personally use MagicISO) that can create a bootable ISO from a floppy image. Then, simply burn the ISO image to a CD and it should work.

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 Bochs emulator and debugger, using a real floppy disk, or by using MagicISO to create a bootable CD.

Until next time

Some of the terms and concepts listed here may be new to you. Do not worry--everything will be explained in the next few articles.

The purpose of this tutorial is to create a stepping stone for the rest of the series. It provides a basic introduction, and a listing of the tools we will be using. I will explain how to use these programs as we need to, so you do not need a tutorial on anything listed here besides what has been listed in the Requirements section.

We also have taken a look at the building process for developing an operating system. For the most part, its fairly simple, however it provides a way to see **when** the programs listed will be used.

In the next tutorial we are going to go back in time from the first Disk Operating System (DOS) and take a little tour through history. We will also look at some basic OS concepts.

We will not be using any of the tools listed above just yet, so you do not need to download them just yet.

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Chapter 0 Home Chapter 2