Operating Systems Development Series

Operating Systems Development - Preface

by Mike, 2009, Updated 2010

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

Introduction

Welcome!

This is a series of chapters, tutorials, and articles about computers and operating systems. The series focuses on a new direction in developing an operating system from scratch during this process, while describing architecture, and concepts that are in system-level programming.

The goal of this series is to provide the most comprehensive guide in operating systems and computer systems, while attempting to cover every bit of it (pun intended).

Before moving on with the series, however, I feel that we should cover our chosen languages, and what is required for our readers to know, along with an overview of important concepts in the languages, and how to work with them. We will also cover concepts that are only used in embedded platforms and system-level software.

This series uses C and x86 Assembly Language. It is very important to have a good understanding of both of these languages before moving on. This chapter includes a review of both of these languages.

If you have never programmmed before...

We would first like to welcome you to the world of programming and computer science. Computer scientists and software engineers use programming methods to build complex software systems. If you have never programmed before, you will be learning a bit of all three right from the start. This tends to make learning the first programming language difficult. However, it becomes easier to learn other programming languages over time.

It is more recommended to start with a simpler language, such as Python or Visual Basic, and build from there. Then move on to Java or C++. Java is eaiser then C++ but it shares a lot of its syntax, so if you pick up Java first, moving to C++ would be a lot easier. In addition, by learning C++ you are also learning a a large subset of C.

We recommend the following sites:

- cprogramming.com
- cplusplus.com
- codecademy.com

Finally, YouTube has a lot of really good videos and lectures on software engineering, computer science, and programming.

- Introduction to Programming
- The Great Debate: Which Programming Language Should You Learn First?

Additional links

- Beginner's Resources to Learn Programming Languages
- Title Pro's Tinkering Guide
- A History of computer programming languages
- A beginners guide to programming languages

Overview of C

It is assumed that you already know how to program in C. This is a quick overview of some of the more important parts of the language, and also how they will work for us.

How to use C in kernel land

16 bit and 32 bit C

In the beginning of programming your system you will find out that there is nothing at all to help you. 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.

In the series, 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. Its 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 both Windows and our own in-house operating system, Neptune. 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.

Doing this can be fairly complicated and tricky to implement. Because of this, we stick with just using assembly language in the series boot loader. We might cover an advanced tutorial later that can describe methods of using C later on however if the reader demand is great enough.

Calling a C kernel

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. In the series, we cover how to do this a little later. This allows us to use C for the kernel and other libraries that we build.

Pointers

Introduction

Because you are reading this, I assume that you are already good with pointers. In system software, they are used everywhere. Because of this, it is very important to master pointers.

A pointer is simply a variable that holds the address of something. To define a pointer, we use the * operator:

```
char* pointer;
```

Remember that a pointer stores an "address"? We do not set the above pointer to anything, so what "address" does it refer to? The above code is an example of a **wild pointer**. A wild pointer is a pointer that can point to anything. **Remember that C does not initialize anything for you.** Because of this, the above pointer can point to anything. Another variable, address 0, some other piece of data, your own code, *a hardware address*.

The Physical Address Space (PAS)

The Physical Address Space (PAS) defines all of the "Addresses" that you can use. These addresses can refer to anything that is inside of the PAS. This includes physical memory (RAM), hardware devices, or even nothingness. This is very different then in applications programming in a protected mode OS, like Windows, where all "addresses" are memory.

Here is an example. In applications programming, the following would cause a segmentation fault error and crash your program:

```
char* pointer = 0;
*pointer = 0;
```

This creates a pointer and points it to memory address 0, which you do not "own". Because of this, the system does not allow you to write to it.

Now, lets try that same exact code again in our future C kernel ... no crash! Instead of crashing, it overwrites the first byte of the **Interrupt Vector Table (IVT)**.

From this, we can make a few important differences:

- The system will not crash if you use null pointers
- Pointers can point to any "address" in the PAS, which may or may not be memory

If you attempt to read from a memory address that does not exist, you will get garbage (whatever was on the system data bus at that time.) An attempt to write to a memory address that does not exist does nothing. Writing to a non existent memory address and immediately reading it back may or may not give you the same result just "written"...It depends if the data "written" is still on the data bus or not.

Things get more interesting here. ROM devices are mapped into the same PAS. This means that it is possible for you to read or write certain parts of ROM devices using pointers. A good example of a ROM device is the system BIOS. Because ROM devices are read only, writing to a ROM device is the same effect as writing to a non existent memory location. You can read from ROM devices, however.

Other devices may also be mapped into the PAS. This depends on your system configuration. This means reading or writing different parts of the PAS may yeild different types of results.

As you can tell, pointers play a much bigger role in systems programming then they did in the applications programming world. It may be easier to think of pointers not as a "variable that points to a memory location" but rather a "variable that points to an address in the PAS" as it may or may not be RAM.

Dynamic Memory Allocation

In the application programming world, you 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. To allocate memory, we do this:

```
char* pointer = (char*)0x5000;
```

That is it. Cool, huh? Because we have control over everything, we can just point a pointer to some address in the PAS (would have to be RAM) and say "theres our new buffer of 1024 bytes" or something like that.

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. But, wait! Aren't we developing our own OS? That is the problem :) 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

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

Most compilers provide a keyword that allows inline assembly. For example, Microsoft Visual C++ uses _asm:

```
_asm cli ; disable interrupts
```

We can also have blocks of assembly code:

```
_asm {
        cli
        hlt
}
```

Standard Library and the Run Time Library (RTL)

You 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.

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

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

In the series, we develop both an RTL that supports C and C++ features as well as a basic standard library as needed.

Fixing Errors

Debugging

Because there is no printf() or any way to use a debugger, what are you going to do if something is not working? The series uses (and explains) how to use the Bochs Debugger, which is a debugger that comes with the Bochs emulator. This can be used to run your 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 your own routines that will allow you to output information. At the most this might be able to tell you how far the software gets to before crashing.

Until next time

That is all that there is for this chapter :) In the next chapter, we begin the adventure into the world of operating systems by looking at what they are, as well as some tools that we will be using throughout the series.

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!

Home

Chapter 1 

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

Operating Systems Development Series

Operating Systems Development - Basic Theory

by Mike, 2009

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

Introduction

Welcome to the wonderful and crazy world of operating systems!

In the previous tutorial, we have defined what an operating system is. An operating system provides the basic interface between the user and the computer system. It provides the basic look and feel for the system.

We have also taken a look at alot of tools that will help us. Assemblers, Compliers, Linkers, PartCopy, MagicISO, and Bochs.

For anyone who has no programming experence who is reading this (I know there is some) shame on you :) Please go back and reread the "Prerequisites" section from the first tutorial. Why are you still reading me? Go go go!

In this article, we are going to look at operating systems in a different way. We will first go Back in Time(tm) to look at the history of operating systems. You will find there are many similarities between these operating systems. These similarities will then be classified into the basic things operating systems have in common--and the building blocks for your own.

Blast from the Past

Most of todays operating systems are graphical. These graphical user interfaces (GUI), however, provide a large abstraction layer to what is really going on in an OS.

Alot of the concepts about operating systems date back sinse programs were on tape. Alot 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 faucet of the computer hardware. Each peice 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 feasable. 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.

What is this ultra cool interface? Why, its that sweet cuddley (sometimes nasty) thing we call an Operating System! :)

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 (Not really :)), **Operating Systems are responsible for Memory Management and Program Execution/Management**

Because this isnt the history of the world (nor computers) that I am describing, lets jump now ahead to good old DOS.

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. Do to some problems, IBM was forced to ship DOS/360 with 3 versions, each released June 1966.

The versions were:

- BOS/360 - 8KB Configuation.
- DOS/360 - 16KB Configuation with disk.
- TOS/360 - 16KB Configuation 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 Varable Transaction), it can support varies program sizes.

Now we have a few more interesting words--**Multitasking, Memory Protection,** and **Fixed Base Address**. Adding to before, we also have **Program execution** and **Memory Management**.

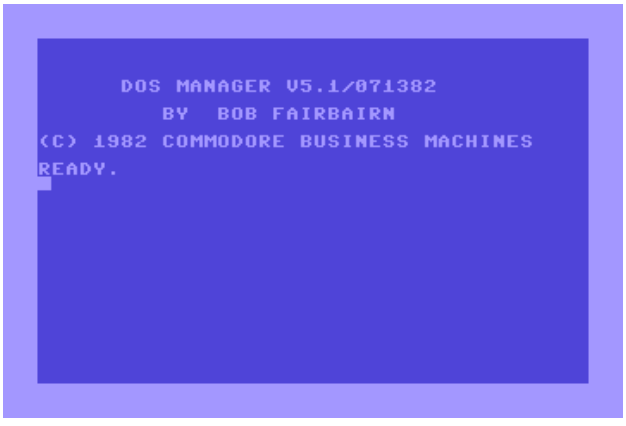
1969 - Its Unix!

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 acedimic institutions, causing it to be ported to a wider varity of machine families then any other OS.

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

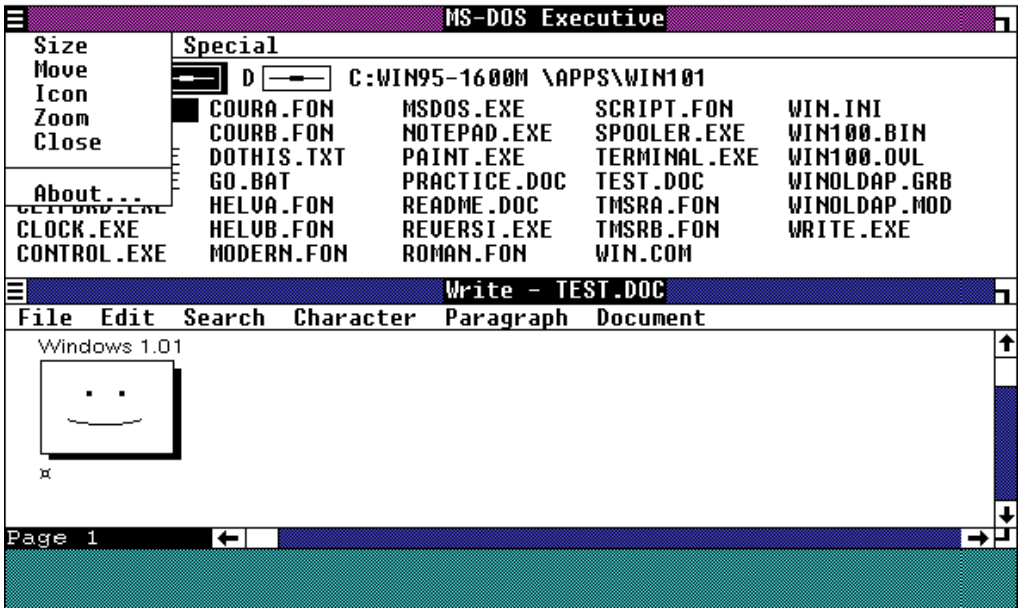
Unix includes a **Kernel**, **File System** and a **Command Shell**. There are alot of **Graphical User Interfaces (GUI)** that uses the **Command Shell** to interact with the OS, and provide a much friendler and nicer look.

1982 - Commodore DOS



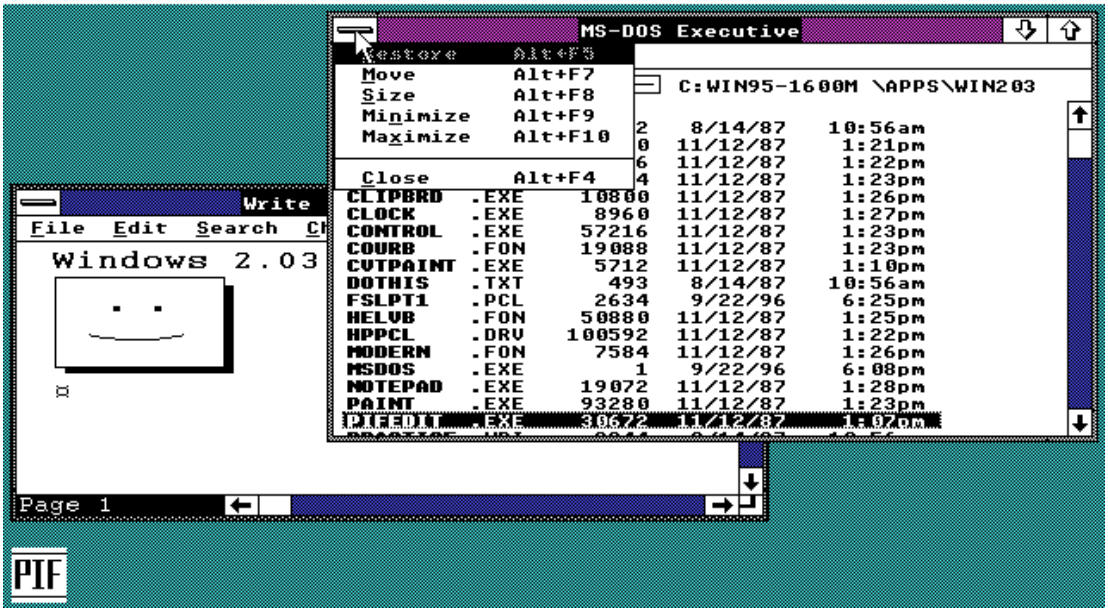
Commodore DOS (CBM DOS) was used with Commodore's 8 bit computers. Unlike the other computers before or since-which booted from disk into the systems memory at startup, CBM DOS executed internally within the drive-internal ROM chips, and was executed by an MOS 6502 CPU.

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



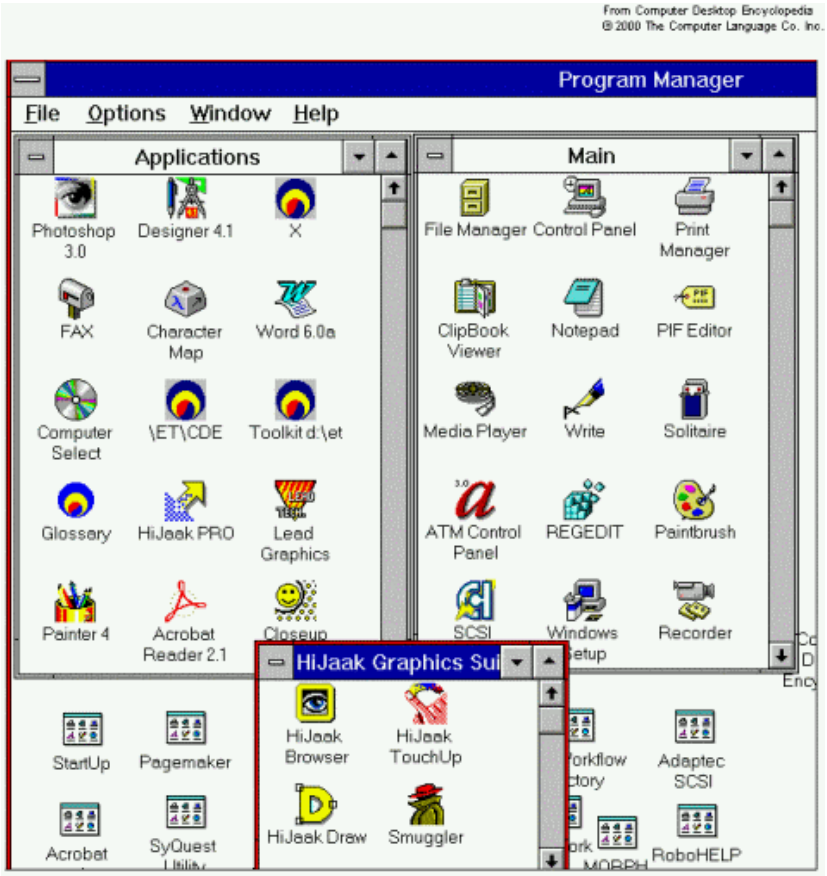
The second version of Windows was still a DOS **Graphical Shell**, but supported overlapping windows, and more colors. However, do to the limitation of DOS, it was not widley used.

Note: DOS is a 16 bit operating system. During this timeframe, DOS had to refrence 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 then 1 MB of memory. This is solved, today, by enabling the 20th address line through the Keyboard Controller. We will go over this later.

Because of this 1 MB limitation, Windows was far to 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 is supports **multitasking** with DOS programs.

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

Windows in relation to OS Developers

I have seen quite a few beginning OS developers want to develope the next Windows. While it is possible, it is extremily difficault, and is impossible with a one person team. Take a look at the above picture again. Remember that it is a **graphical shell** over a **command shell**, being executed by a **Kernel**. Also, remember that even Windows had to start here. The Command Shell was DOS. the Graphical Shell was "Windows".

Basic Concepts

Looking back though our small trip to memory lane brings some important new terms with it. So far, we only gave "operating system" a small definition. The previous section should help us in defining a better, more descriptive definition of what an operating system is.

To help define a better definition, lets put the above bolded terms into a list:

- 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 (From the previous tutorial)

Thats alot to think about, huh? And yet--The above list is techincally still an abstraction layer itself.

Lets take a look closer, shall we?

Memory Management

Memory Management refers to:

- Dynamically giving and using memory to and from programs that request it.

- Implimenting a form of **Paging**, or even **Virtual Memory**.
- Insuring the OS Kernel does not read or write to unkown or invalid memory.
- Watching and handling **Memory Fragmentation**.

Program Management

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

- Insuring the program doesnt write over another program.
- Insuring the program does not currupt 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 simulataniously.

Memory Protection

This referrs 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 wouldnt 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 curruption of data.

Kernel

The Kernel is the heart of the Operating System. It provides the basic foundation, memory management, file systems, program execution, etc. We will take a closer look at the kernel very soon, dont worry :)

File System

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

A File System is simply a specification that describes information reguarding files. In most cases, this referrs 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. Seriously. This means the filename "KRNL.sys", for example, will have the internal file name

```
"KRNL    SYS"
```

We will be using FAT12 and be discussing it in detail later.

Command Shell

A Command Shell sits ontop the Kernel as a seperate 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). Useually this is automatic these days, however.

Linear Block Addressing (LBA)

Operating Systems have control over **every single little byte in memory**. Linear Addressing refers to directly accessing linear memory. For example:

```
mov     ax, [09000h]            ; There is no such thing as Access Violations in OS Development
```

This is a good thing, but is also a very bad thing. For example:

```
                mov     bx, [07bffh]            ; or some other address less then 7c00h
                mov     cx, 10
        .loop1:
                mov     [bx], 0h                ; clear bx
                inc     bx                      ; go to next address
                loop    .loop1                  ; loop until cx=0
```

The above code seems harmless. However, if the above code was found in a bootloader, the above code will overwrite itself by 10 bytes. Ouch. The reason is that bootloaders are loaded with a Fixed address of 0x7c00:0, and the above code starts writing from 07bffh: One byte before 07c00h.

Bootloader

The bootloader. We seen this term from the previous tutorial. From the previous tutorial, we know the bootloader is loaded by the BIOS, and is the very first program to execute for your operaing system.

The bootloader is loaded by the BIOS at absolute address 0x7c00:0. After loading, CS:IP is set to your bootloader, and the bootloader takes full control.

A Floppy Sector is only 512 bytes in size. Remember that the bootloader has to fit in a single bootsector. What does this mean? The bootloader is very limited in size, and cannot exceed 512 bytes.

Most of the time, the bootloader will either just load and execute the kernel, or a **Second Stage Bootloader**.

We will take a closer look at the booting process very soon. That is when we will look at bootloaders.

Conclusion

We have taken a look into the past, and learned a few more terms to our list. After the history lesson, we took the terms and built a more broader perspective on how everything works. We even got to see some code. A small amount, though.

After all of this, one should be able to develope a more concise definition of what we are doing.

"An interactive envirement that provides an interface for the user and supplied programs, providing a stable and safe envirement, an interface layer to System services and computer hardware."

Yep. Thats my new definition of "Operating System", what is yours?

In the next tutorial, we are going to take a look on the booting process in descrete detail. Afterwords, we will take a look at building and assembling of a real bootloader.

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 1 Home Chapter 3 

Operating Systems Development Series

Operating Systems Development - Bootloaders

by Mike, 2008, 2009

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

Introduction

Welcome! This is the tutorial you have been waiting for. We will cover many topics such as:

- The booting process - How it works
- Bootloader Theory
- Developing a simple bootloader
- Assembling the bootloader with NASM
- Using the VFD (Virtual Floppy Drive) software; Creating a floppy image
- Using PartCopy; Copying our bootloader to the floppy image
- Using Bochs - Basic Setup and Use; Testing the bootloader

Ready?

The Boot Process

Pressing the power button

What actually happens when you press the power button? When this 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.

To better understand this, remember the basics of binary logic in computers. 8 "bits" simply represent 8 "wires" or "lines" where electricity can go. A 0 represents no current, while a 1 represents current within a line. This, along with Logic Gates, is the bases of Digital Logic Electronics, at which computers were built.

When the PSU recieves 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 suppling 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 recieves 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 curruption).

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.

What does 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 problems. 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 simular 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 Interupt 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 nonexistant routine, causing your program to crash.

Note: If you switch processor modes, the IVT will not be avilable. This means absolutley *no* interrupts--neither software nor hardware, will be avilable, 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

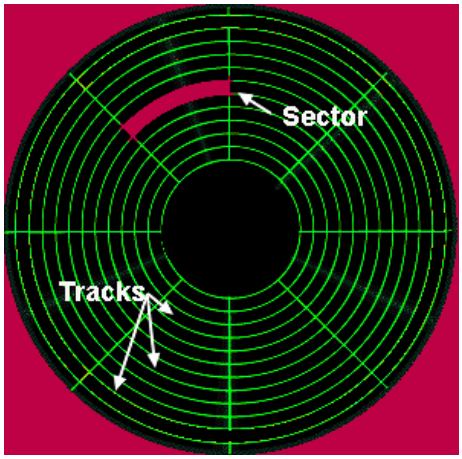
A "Sector" simply represents a goupe of 512 bytes. So, Sector 1 represents the first 512 bytes of a disk.

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

To understand tracks, we should look at a picture:



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 wil be important when loading files.

If the disk is bootable, **Then the bootsector will be loaded at 0x7C00**, and INT 0x19 will jump to it, therby 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.

Now, our 1337 bootloader is in control!

Bootloader Theory

We have talked alot about bootloaders. Lets put the important parts together, shall we?

So far, bootloaders...

- ...Are stored with the Master Boot Record (MBR).
- ...Are in the first sector of the disk.
- ...Is the size of a single sector (512) bytes.
- ...Are loaded by the BIOS INT 0x19 at address 0x7C00.

As you can imagine, we cannot do a whole lot in 512 bytes. What do we do?

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, coinsider 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 isnt in memory!**.

What will the processor do after **inc bx** then? It will stll 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 approch will work, but it will be hard hacked. The most common approch is keeping the bootloader at 512 bytes in size, searching, loading, and executing a second stage bootloader. We will look more into this later.

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 avilable, so **any interrupt-hardware or software, will cause your system to crash**. More on this later.

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

We seen this term before, havn't we? A CPU that "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

Yiippee! *drum rolls* The moment we have been waiting for! :)

Lets take another look at our list:

- Are stored with the Master Boot Record (MBR).
- Are in the first sector of the disk.
- Is the size of a single sector (512) bytes.
- Are loaded by the BIOS INT 0x19 at address 0x7C00.

Open up any ordinary text editor (I am using Visual Studio 2005), but Notepad will suffice.

Heres the bootloader (Boot1.asm)...

```
;*********************************************
;       Boot1.asm
;               - A Simple Bootloader
;
;       Operating Systems Development Tutorial
;*********************************************

org             0x7c00                          ; We are loaded by BIOS at 0x7C00

bits    16                                      ; We are still in 16 bit Real Mode

Start:

        cli                                     ; Clear all Interrupts
        hlt                                     ; halt the system

times 510 - ($-$$) db 0                         ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55                                       ; Boot Signiture
```

Some of this should not come to much of a surprise. Lets analize line by line:

```
org             0x7c00                          ; We are loaded by BIOS at 0x7C00
```

Remember: **The BIOS loads us at 0x7C00.** The above code tells NASM to insure all addresses are relitave to 0x7C00. This means, the first instruction will be at 0x7C00.

```
bits    16                                      ; We are still in 16 bit Real Mode
```

Remember tutorial two? In that tutorial, I explained how the x86 family is backward compatible with the old DOS systems. Because the old DOS systems were 16 bit, **All x86 compatible computers boot into 16 bit mode**. This means:

- We are limited to 1 MB (+64k) of memory.

We will need to switch the computer into a 32 bit mode. We will do this later.

```
times 510 - ($-$$) db 0                    ; We have to be 512 bytes. Clear the rest of the bytes with 0
```

I wish this was more documented. In NASM, the dollar operator ($) represents the address of the current line. $$ represents the address of the first instruction (Should be 0x7C00). So, **$-$$** returns the number of bytes from the current line to the start (In this case, the size of the program).

```
dw 0xAA55                                  ; Boot Signiture
```

This needs some explanation.

Remember that the BIOS INT 0x19 searches for a bootable disk. How does it know if the disk is bootable? The boot signiture. If the 511 byte is 0xAA and the 512 byte is 0x55, INT 0x19 will load and execute the bootloader.

Because the boot signiture must be the last two bytes in the bootsector, We use the **times** keyword to calculate the size different to fill in up to the 510th byte, rather then the 512th byte.

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".

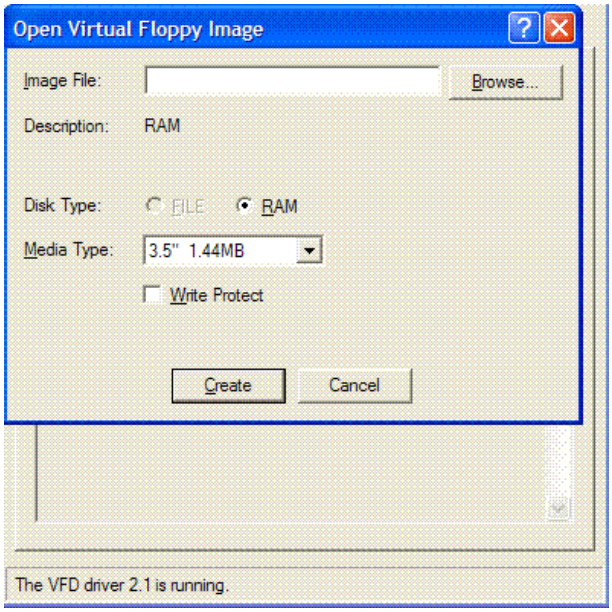
Note: For some reason, Windows Explorer limits displaying file sizes to 1 KB. Just see the properties of the file, and it should say 512 bytes.

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

You should see this:



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 (On *your* computer ;)) and you should see a new floppy drive.

To format the disk, right click the drive and go to Properties. Under the VFD Tab will be a format option.

PartCopy - Copying to the Bootsector

Great... Now that we have our boot loader ready, how do we copy it to our disk? As you probably know, Windows will not allow us to directly copy it to the first sector of a disk. Because of this, we need to use a command to do it.

In the first tutorial we have looked at one of these commands: **debug**. If you have decided to use this command, you can skip this section on **partcopy**.

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

```
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 refrence 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 parmenant disk curruption if you are not carefull. The above command line commands will only work for floppy disks. Do not attempt to try on hard disks!

Bochs: Testing the bootloader

Bochs is a 32 bit PC emulator. We are going to use Bochs for debugging and testing.

Bochs uses a configuation file that describes the hardware to emulate. For example, this is the configuation file I am using:

```
# ROM and VGA BIOS images ----------------------------------------------

romimage:    file=BIOS-bochs-latest, address=0xf0000
vgaromimage: VGABIOS-lgpl-latest

# boot from floppy using our disk image --------------------------------

floppya: 1_44=a:, status=inserted  # Boot from drive A

# logging and reporting ------------------------------------------------

log:         OSDev.log             # All errors and info logs will output to OSDev.log
error:       action=report
info:        action=report
```

The configuation file uses # for comments. It will attempt to boot from whatever floppy disk image (Like the one we created in VFD) in drive A.

The ROM BIOS and VGA BIOS images come with Bochs, so you dont need to worry about that.

Locating the BIOS ROM

Alot of the lines in the configuation 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 Bochs 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-bochs-latest** in your Bochs directory). Get the size in bytes.

After this, simply subtract 0xFFFFF - size of bochs 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.

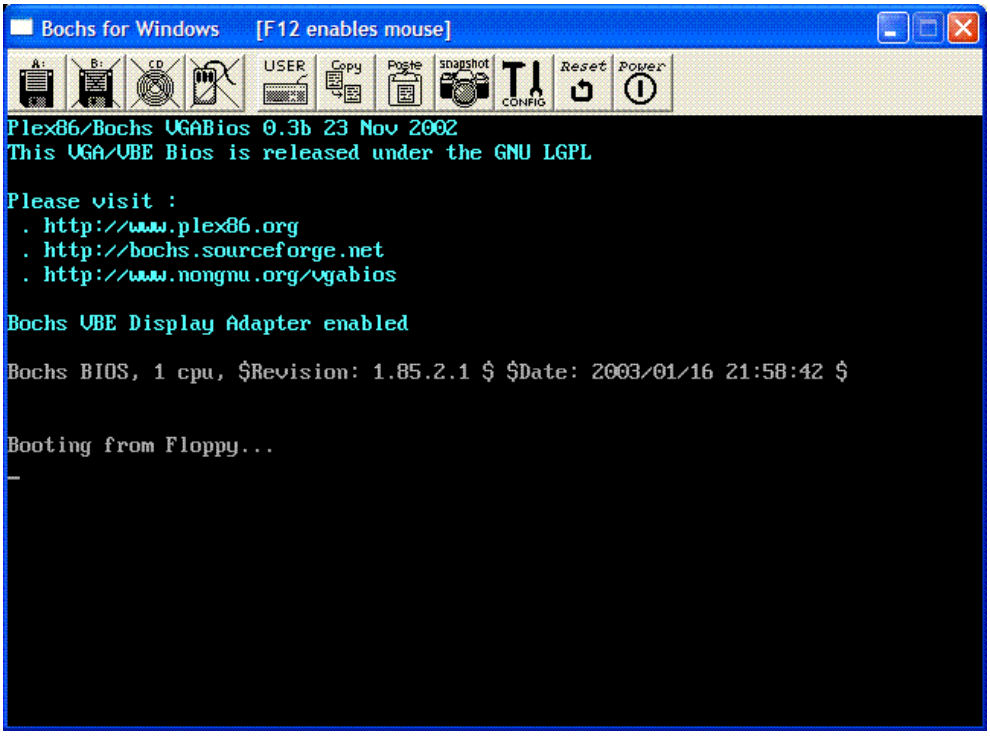
You may or may not need to do this step. If you get an error from Bochs telling you that the Bios must end at 0xFFFFF, then you do need to complete this step and it should work.

How to use Bochs

To use Bochs:

1. Execute bochs.exe
2. Select option 2 (Read options form); hit enter.
3. Type in the configuation files name (The one we created above); hit enter.
4. You will be back to the main menu. Select option 5: Begin Simulation, and hit enter.

A new window will open, and this is what you should see:



If Bochs just quits or restarts

...Then you have just experenced a Triple Fault. Go back to the code and try to find where the problem is at. If you need any help, feel free to contact me.

If the Window appears, but does nothing

Congrats! That is our **cli** and **hlt** instructions haulting the system, so we know our bootloader is being executed.

The Build Process - Abstract

Compare what we have done with the Build process we looked at in the previous tutorial. Its very simple once you get used to it.

From here on out, I wont be describing these steps of the build process in detail anymore.

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 2 Home Chapter 4

Operating Systems Development - Bootloaders 2

by Mike, 2009

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

Introduction

Welcome!

We have went over alot in the previus tutorial. We took a look at what exactally happens when you press the power button, and how the BIOS boots. We also looked at the BIOS Interrupt (INT) 0x19, which searches for a boot signiture (0xAA55), and, if found, loads and executes our bootloader at 0x7C00.

We also developed a simple bootloader, and got some experience with the entire build process.

In this tutorial, we will expand on our bootloader. We will cover:

- BIOS Paramater Block and the MBR
- Processor Modes
- Interrups - Printing text and more
- Segment:Offset Addressing Mode

Note: From here on out, our bootloader has full control of the entire system. What this means, is simply that everything relys on us writing the code. Everything now is up to us! In short: There will be alot more code comming up.

From here on out, things are going to get more complex. To insure the structure of this series stays solid, I will have a downloadable demo with each and upcomming tutorials. This will aid in understanding the concepts. Dont worry--I will still be explaining everything in detail here.

Ready?

Processor Modes

Well, well...Where have we heard this term before? Lets see... In every single tutorial!

And yet, we have not really talked much about it. Understanding the different processor modes will be very important to us. Why is this?

In the previous 2 tutorials, we talked about how and why the x86 family boots up in a 16 bit envirement. We want to develope a 32 bit operating system (OS), so we will need to switch the processor from its 16 bit mode into a 32 bit mode.

There are more then two modes. Lets go over each one, shall we?

Real Mode

As you know, the x86 processor boots into a 16 bit envirement. What is this mode? (Hint: Its not Real Mode) ... Okay, it is :)

What is so real abut real mode, anyway? Real Mode...

- Uses the native **segment:offset memory model**.
- Is limited to 1 MB (+64k) of memory.
- No **Virtual Memory** or **Memory Protection**.

Some of these are faily easy. Others require some explanation though. One thing to note is that, all of the above indirectly or directly relates to memory.

Lets take a closer look, shall we?

Segment:Offset Memory Mode - History

Lets go Back In Time(tm) again, and look at Tutorial 2. The concept of Memory and use of Operating Systems dated back since the 1950s. These computers were not personal computers, but instead large mainframe computers.

Remember that, back then, all computers had very large and bulky hardware devices. Through time (Looking back at Tutorial 2), you can see not only advances in Operating Systems, but computers as well.

As computer poplurity gained, so did its demand. When computers were 8 bit, many wanted 16. When the 16 bit era came, Microsoft was already thinking 32 bit. As soon as the 32 bit area came, 64 bit was already mainstream. Okay, the last one isnt true :) but, 128 bit is on its way.

The primary problem is the computer industry moves to fast.

When Intel was designing the 8086 Processor, the processor used 16 bit registers, and could only access up to 64 KB of memory. The problem, however, was that alot of software required more memory then this.

The 8086 was being designed the same time the 8088 was. The 8088, however, was going to be Intels "next generation" processor, except it was taking longer then expected. To challenge the other companies, Intel quicky wanted to develope and release a processor, the 8086, to hold off until the 8088 was released.

The problem here is that, software demanded more memory then 64 KB, and Intels processor, the 8086, was to challenge their competitors who is already building 16 bit processors, until the 8088 was released. Intel needed a stragety.

The designers of the 8086 proposed a solution. This solution will allow the 8086 to stay 16 bit, while being able to access up to 1 MB of memory. They aggreed, and Intel approved.

The segment:offset memory scheme was born.

To understad the segment:offset scheme, lets break it down and look at segments and offsets first.

Segments

A Segment is simply a part of a whole. In our case, A segment is a section of memory. Yep--Thats basically all it is.

Imagine dividing memory into sections. These sections represent segments.

The x86 Family of processors uses 4 primary registes that store the beginning location of a segment. Its like a **base address**--It provides the start of a segment.

Normally, a segment may be 64 KB in size, and are freely movable.

Remember that segments simply epresent a section in memory. In this case, if the segment base address is 0, then this represents the segment between byte 0 and 64 KB.

The registers are **CS, DS, ES,** and **SS**. These registers store the base address for the segments. We will take a closer look at them after we look at addressing with this mode.

Offsets

An offset is a number that is added to a base number. For example, if the base number is 3:

Offset = base number (3) + the offset number
Offset 2 is 3+2 = 5
Offset 4 is 3+4 = 7

Okay, so how does this relate to us? Well, in segment:offset addressing, we add the Base Address (Remember that a segment represents a base address) with the offset address!

Pretty simple, huh? Now, lets put it all together.

Segment:Offset Addressing

In Segment:Offset Addressing, we simply add the offset address with the segment address. However, in the previous section, I mentioned that each segment address in Real Mode is **16 bits** in size. This means we also have to multiply our segment by 16(decimal), and then we add the offset. So, heres our current formula:

```
Absolute (Exact) Memory Address = (Segment Address * 16(decimal)) + Offset
```

Thats all there is to it :)

Segment:Offset Conventions

Segment and offset addresses are useually seperated by a colen (:). They are useually of the form **Segment : Offset**. For example:

```
07C0:0000     < 07C0 is the segment, and 0 is the offset
```

We can convert the above to the absolute address 0x7C00 by using our formula:

```
                base address = base address * segment size (16) + offset
                       07C0:0000 = 07C0 * 16 (decimal) + 0
                                 = 07C00 + 0 = 0x7C00
```

Segment:Offset Problems

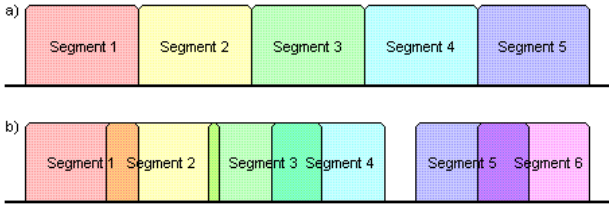
Segment:Offset is quite unique. By changing the segment and offset values, you can find different segment:offset pairs will yeld the same absolute address. Why? Because they both **refer to the same memory location!**

For example, all of the below addresses refer to our bootloader at 0x7C00:

```
0007:7B90   0008:7B80   0009:7B70   000A:7B60   000B:7B50   000C:7B40
0047:7790   0048:7780   0049:7770   004A:7760   004B:7750   004C:7740
0077:7490   0078:7480   0079:7470   007A:7460   007B:7450   007C:7440
01FF:5C10   0200:5C00   0201:5BF0   0202:5BE0   0203:5BD0   0204:5BC0
07BB:0050   07BC:0040   07BD:0030   07BE:0020   07BF:0010   07C0:0000
```

These are only a few. Technically, there is exactly **4,096** different combinations of segment:offset that can refer to the same byte in memory -- This is for each byte in memory!

What if we have two segment addresses that are **within** 64 KB? Remember that the size of a segment (and offset) are 16 bits. And the segment addresses refer only to the **base** of a segment. This is what an **Overlapped Segment** is:



Imagine layers ontop of layers that lay over other segments. This could cause problems.

This means, in Real Mode, you can access every byte in memory, over 4,000 different ways, being able to overlap segments that could potentally currupt that area of memory without you knowing. This is what is ment by Real Mode not having any **Memory Protection**.

The registers the x86 use for segment refrencing are as followes:

- CS (Code Segment) - Stores base segment address for code
- DS (Data Segment) - Stores base segment address for data
- ES (Extra Segment) - Stores base segment address for anything
- SS (Stack Segment) - Stores base segment address for the stack

Wow, Real Mode has alot of problems. What will protect little old us from it?

Protected Mode

Protected Mode (PMode) is a term you heard alot, and will here alot more. PMode allows Memory Protection through the use of a **Descriptor Tables** that describe your memory layout.

PMode is a 32 bit processor modes, so it also allows you to use 32 bit registers, and access up to **4 GB of RAM**. A huge improvment over Real Mode.

We will be using PMode. And yes, before you ask, Windows is a PMode OS. :)

PMode is a bit tricky to set up and to fully understand how it works. We will discuss more about PMode later.

Unreal Mode

It is possible to switch from processor modes whenever you want. The term "Unreal Mode" is a pun that represents Real Mode with the address space (4 GB limit) of PMode.

To enable Unreal Mode, simply switch the processor from Real Mode into PMode, and back again after loading a new **Descriptor**.

Descriptor Tables can be quite confusing. I will talk about them in detail when we talk more about Protected Mode (PMode).

Virtual 8086 Mode

Virtual 8086 Mode (v86 Mode) is a Mode that reresents Protected Mode with a 16 bit Real Mode emulated envirement.

This might seem kind of strange, dont you think? v86 can be usefull, however. **All BIOS interrupts are *only* avilable in real mode!** v86 Mode provides a way of executing BIOS interrupts from within PMode. More on this later.

Switching processor modes

We wont cover the code to switch processor modes just yet. Instead, I want to take a step back and explain some important concepts.

The only two built in actual modes are Real Mode and Potected Mode. In other words, the other modes, Unreal Mode and v86 Mode, are built from these two modes.

Remember that Unreal Mode is in Real Mode, but uses the Protected Mode (PMode) Addressing system. And, Virtual 8086 Mode is in PMode, but uses Real Mode to execute 16 bit code.

As you can see, both v86 and Unreal mode are simply based off of Real Mode and Protected Modes. Because of this, it might be hard to understand how these modes work without an understanding of PMode.

We will take a closer look at PMode, Unreal Mode, and v86 Mode soon, so dont worry :)

There is some important things to remember about PMode however:

- Absolutley no interrupts will be avilable. You will need to write them yourself. The use of any interrupt--hardware or software will cause a Triple Fault
- Once you switch into pmode, the ***slightest*** mistake will cause a Triple Fault. Be carefull.
- PMode requires the use of **Descriptor Tables**, such as the **GDT, LDT, and IDT**s.
- PMode gives you access to 4 GB of Memory, With Memory Protection
- Segment:Offset Addressing is used along with Linear Addressing
- Access and use of 32 bit registers

We will cover PMode in alot more detail later.

Expanding the bootloader

Wow, we went over alot so far, didnt we? We went over basic theory of Protected Mode, Unreal Mode, and v86 Mode. We covered Real Mode in depth though. Why? Because, remember that the computer boots in a 16 bit envirement for backward compatibility with DOS. **This 16 bit envirement is Real Mode**.

So, yes-When our bootloader is executed, we are in Real Mode. Wait! This means we can use BIOS Interrupts, right? Yep :) Ths includes VGA Video interrupts, and any other interrupt mapped directly from hardware :)

Usefull Routines and BIOS Interrupts

OEM Parameter Block

The OEM Parameter Block stores the Windows MBR and Boot Record information. Its primary purpose is to describe the filesystem on the disk. We will not describe this table until we look at filesystems. However, we can go no further without it.

This will also fix the "Not formatted" message from Windows.

For now, think of this table as a simple neccessity. I will explain it in detail later when we talk about File Systems, and loading Files off disk.

Here is the bootloader with the table:

```
;*********************************************
;       Boot1.asm
;               - A Simple Bootloader
;
;       Operating Systems Development Tutorial
;*********************************************

bits    16                                                      ; We are still in 16 bit Real Mode

org             0x7c00                                          ; We are loaded by BIOS at 0x7C00

start:          jmp loader                                      ; jump over OEM block

;*************************************************;
;       OEM Parameter block
;*************************************************;

TIMES 0Bh-$+start DB 0

bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
bpbReservedSectors:     DW 1
bpbNumberOfFATs:            DB 2
bpbRootEntries:             DW 224
bpbTotalSectors:            DW 2880
bpbMedia:                   DB 0xF0
bpbSectorsPerFAT:           DW 9
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
bpbHiddenSectors:           DD 0
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
bsUnused:                   DB 0
bsExtBootSignature:     DB 0x29
```

```
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "

;*************************************************;
;       Bootloader Entry Point
;*************************************************;

loader:

        cli                     ; Clear all Interrupts
        hlt                     ; halt the system

times 510 - ($-$$) db 0         ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55                       ; Boot Signiture
```

Printing Text - Interrupt 0x10 Function 0x0E

You an use INT 0x10 for video interrupts. Remember, however, that only basic interrupts will work.

INT 0x10 - VIDEO TELETYPE OUTPUT

AH = 0x0E
AL = Character to write
BH - Page Number (Should be 0)
BL = Foreground color (Graphics Modes Only)
For example:

```
        xor     bx, bx          ; A faster method of clearing BX to 0
        mov     ah, 0x0e
        mov     al, 'A'
        int     0x10
```

This will print the character 'A' on the screen.

Printing Strings - Interrupt 0x10 Function 0x0E

Using the same interrupt, we can easily print out a 0 terminated string:

```
msg     db      "Welcome to My Operating System!", 0

;************************************
;       Prints a string
;       DS=>SI: 0 terminated string
;************************************

Print:
                        lodsb
                        or                      al, al                          ; al=current character
                        jz                      PrintDone                       ; null terminator found
                        mov                     ah,     0eh                     ; get next character
                        int                     10h
                        jmp                     Print
PrintDone:
                        ret

;*************************************************;
;       Bootloader Entry Point
;*************************************************;

loader:

; Error Fix 1 -----------------------------------------

        xor     ax, ax          ; Setup segments to insure they are 0. Remember that
        mov     ds, ax          ; we have ORG 0x7c00. This means all addresses are based
        mov     es, ax          ; from 0x7c00:0. Because the data segments are within the same
                                ; code segment, null em.

        mov     si, msg
        call    Print

        cli                     ; Clear all Interrupts
        hlt                     ; halt the system

times 510 - ($-$$) db 0         ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55                       ; Boot Signiture
```

Getting amount of RAM

This is too easy:

INT 0x12 - BIOS GET MEMORY SIZE
Returns: AX = Kilobytes of contiguous memory starting from absolute address 0x0.

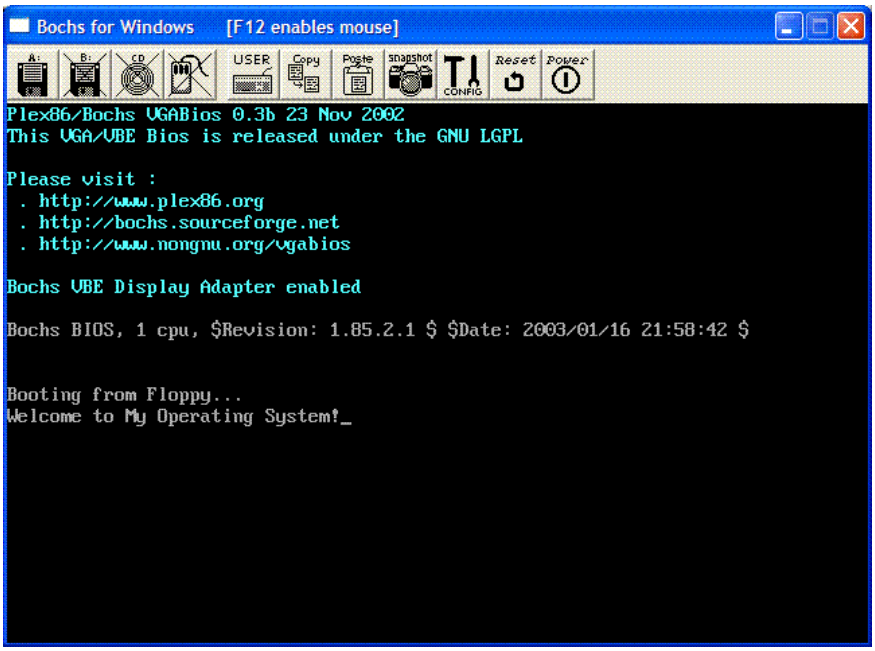
Heres an example:

```
xor     ax, ax
int     0x12
; Now AX = Amount of KB in system recorded by BIOS
```

Wow... That was hard, wasnt it? :) Actually, it can be very hard to do in Protected Mode (PMode) as you will not have any interrupts available.

Note: The amount of memory actually returned from the BIOS might not be accurate! We will look at some other methods later.

Demo




```
;*********************************************
;       Boot1.asm
;               - A Simple Bootloader
;
;       Operating Systems Development Tutorial
;*********************************************

bits    16                                              ; We are still in 16 bit Real Mode

org             0x7c00                                  ; We are loaded by BIOS at 0x7C00

start:          jmp loader                              ; jump over OEM block

;*************************************************;
;       OEM Parameter block
;*************************************************;

; Error Fix 2 - Removing the ugly TIMES directive -----------------------------------

;;      TIMES 0Bh-$+start DB 0                          ; The OEM Parameter Block is exactally 3 bytes
                                                        ; from where we are loaded at. This fills in those
                                                        ; 3 bytes, along with 8 more. Why?

bpbOEM                  db "My OS   "                   ; This member must be exactally 8 bytes. It is just
                                                        ; the name of your OS :) Everything else remains the same.

bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
bpbReservedSectors:     DW 1
bpbNumberOfFATs:            DB 2
bpbRootEntries:             DW 224
bpbTotalSectors:            DW 2880
bpbMedia:                   DB 0xF0
bpbSectorsPerFAT:           DW 9
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
bpbHiddenSectors:           DD 0
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
bsUnused:                   DB 0
bsExtBootSignature:     DB 0x29
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "

msg     db      "Welcome to My Operating System!", 0            ; the string to print

;*************************************
;       Prints a string
;       DS=>SI: 0 terminated string
;*************************************

Print:
                        lodsb                           ; load next byte from string from SI to AL
                        or                      al, al          ; Does AL=0?
                        jz                      PrintDone       ; Yep, null terminator found-bail out
                        mov                     ah,     0eh     ; Nope-Print the character
                        int                     10h
                        jmp                     Print           ; Repeat until null terminator found
PrintDone:
                        ret                                     ; we are done, so return

;*************************************************;
;       Bootloader Entry Point
;*************************************************;

loader:
```

```
        xor     ax, ax          ; Setup segments to insure they are 0. Remember that
        mov     ds, ax          ; we have ORG 0x7c00. This means all addresses are based
        mov     es, ax          ; from 0x7c00:0. Because the data segments are within the same
                                ; code segment, null em.

        mov     si, msg                                         ; our message to print
        call    Print                                           ; call our print function

        xor     ax, ax                                          ; clear ax
        int     0x12                                            ; get the amount of KB from the BIOS

        cli                                                     ; Clear all Interrupts
        hlt                                                     ; halt the system

times 510 - ($-$$) db 0                                         ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55                                                       ; Boot Signiture
```

Conclusion

Give yourself a pat on the back for making this far. :)

This tutorial was a nasty one. I had to find a very good way of explaining segment:offset addressing and the processor modes without going into to much depth. I think I did well :)

We talked about the different processor modes--Real Mode, Protected Mode, Unreal Mode, and v86. We looked at Real Mode in depth because we will be in that mode when developing the bootloader. We also went over segent:offset addressing. This might be a refresher course for some of our DOS programmers out there. We also looked at some BIOS interrupts, and ended with a complete example.

In the next tutorial, we are going to decypher that ugly OEM Parameter Block that we added. We are also going to take a look at basic File System theory, and loading sectors off disk.

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 3 Home Chapter 5

Operating Systems Development Series

Operating Systems Development - Bootloaders 3

by Mike, 2009

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

Please note: This chapter is planned to be updated soon to fix errors and provide more through information on topics.

Introduction

Welcome!

In the previous tutorial, we took a look at the different processor modes, and some easy BIOS Interrupts. We also looked at segment:offset addressing in Real Mod and explained Real Mode in depth. We also expanded our bootloader with a mysterous "OEM Parameter Block", along with the ability to print a string on screen (pu intended :))

In this tutorial, we are going to take a look at the different "Rings" that describe differences between **Application Programming** and **Systems Programming**.

We will also take a look at single stage and multi stage bootloaders, and pros and cons of each.

Lastly, we will take a look at **BIOS Interrupt 0x13**, the **OEM Parameter Block**, and **Reading, loading, and executing a program**. This program will be our **Se Stage Bootloader**. Our **Second Stage Bootloader** will then set the 32 bit envirement, and prepare to load our **C Kernel**.

Ready?

The Rings of Assembly Language

In Assembly Language, you might here the term "Ring 0 program", or "This program is running in Ring 3". Understanding the different rings (and what they are) ca usefull in OS Development.

Rings - Theory

Okay, so what is a ring? A "Ring", in Assembly Language, represents the level of protection and control the program has over the system. There are 4 rings: Ring 0 1, Ring 2, and Ring 3.

Ring 0 programs have absolute control over everything within a system, while ring 3 has less control. The smaller the ring number, the more control (and less leve protection), the software has.

A Ring is more then a concept--it is built into the processor arhcitecture.

When the computer boots up, even when your Bootloader executes, the processor is in Ring 0. Most applications, such as DOS applications, run in Ring 3. This mea Operating Systems, as running in Ring 0, have far more control over everything then normal Ring 3 applications.

Switching Rings

Because Rings are part of the processor architecture, the processor changes states whenever it needs to. It may change when...

- A directed instruction executes a program at a different ring level, such as a far jump, far call, far return, etc.
- A **trap** instruction, such as INT, **SYSCALL**, **SYSENTER**
- **Exceptions**

We will cover Exception Handling later, as well as the **SYSCALL** and **SYSENTER** instructions.

Multi Stage Bootloaders

Single Stage Bootloaders

Remember that bootloaders, and bootsectors, are only 512 bytes in size. If the bootloader, within that same 512 bytes, executed the kernel directly, it is called a **Si Stage Bootloader**.

The problem with this, however, is that of its size. There is so little room to do alot within those 512 bytes. It will be very difficault to set up, load and execute a 32 kernel within a 16 bit bootloader. This does not include error handling code. This includes code for: **GDT, IDT, A20, PMode, Loading and finding 32 bit kernel, executing kernel, and error handling.** Fitting all of this code within 512 bytes is impossible. Because of this, **Single stage bootloaders have to load and exe a 16 bit kernel.**

Because of this problem, most bootloaders are Multi Stage Loaders.

Multi Stage Bootloaders

A Multi Stage Bootloader consists of a single 512 byte bootloader (The Single Stage Loader), however it just loads and executes another loader - A Second Stage Bootloader. the Second Stage Bootloader is normally 16 bit, however it will include all of the code (listed in the previous section), and more. It will be able to load a execute a 32 bit Kernel.

The reason this works is because the only 512 byte limitation is the bootloader. As long as the bootloader loads all of the sectors for the Second Stage loader in goo manner, the Second Stage Loader has no limitation in size. This makes things much easier to set up for the Kernel.

We will be using a 2 Stage Bootloader.

Loading Sectors Off Disk

Remember that Bootloaders are limited to 512 bytes. Because of this, there is not a whole lot we can do. As stated in the previous section, we are going to be usin Stage Bootloader. This means, we will need our Bootloader to load and execute our Stage 2 program -- The Kernel Loader.

If you wanted to, The Stage 2 loader is the place to include your own "Choose your Operating System" and "Advanced Options" menu :) Come on, I know you wan :)

BIOS Interrupt (INT) 0x13 Function 0 - Reset Floppy Disk

The BIOS Interrupt 0x13 is used for disk access. You can use INT 0x13, Function 0 to reset the floppy drive. What this means is, wherever the Floppy Controller is reading from, it will immediately go to the first Sector on disk.

INT 0x13/AH=0x0 - DISK : RESET DISK SYSTEM
AH = 0x0
DL = Drive to Reset

Returns:
AH = Status Code
CF (Carry Flag) is clear if success, it is set if failure

Here is a complete example. This resets the floppy drive, and will try again if there is an error:

```
.Reset:
        mov             ah, 0                                   ; reset floppy disk function
        mov             dl, 0                                   ; drive 0 is floppy drive
        int             0x13                                    ; call BIOS
        jc              .Reset                                  ; If Carry Flag (CF) is set, there was an error. Try resetting again
```

Why is this interrupt important to us? Before reading any sectors, we have to insure we begin from sector 0. We dont know what sector the floppy controller is rea from. This is bad, as it can change from any time you reboot. Reseting the disk to sector 0 will insure you are reading the same sectors each time.

BIOS Interrupt (INT) 0x13 Function 0x02 - Reading Sectors

INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY
AH = 0x02
AL = Number of sectors to read
CH = Low eight bits of cylinder number
CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only
DH = Head Number
DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to

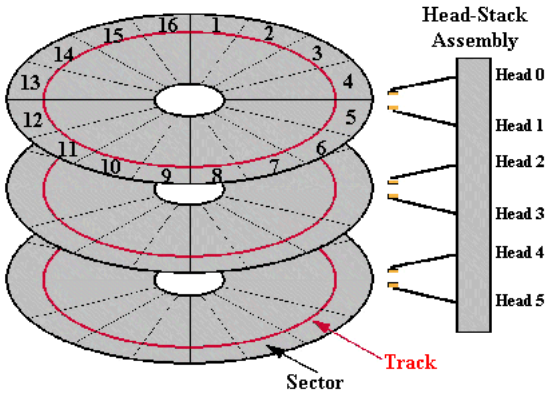
Returns:
AH = Status Code
AL = Number of sectors read
CF = set if failure, cleared is successfull

Okay, This is alot to think about. Some of this is fairly easy, others should be explained more. Lets take a look closer, shall we?

```
CH = Low eight bits of cylinder number
```

What is a **Cylinder**? A cylinder is a group of tracks (with the same radius) on the disk. To better understand this, lets look at a picture:

Drive Physical and Logical Organization



Looking at the above picture, remember:

- Each Track is useually divided into 512 byte sectors. On floppies, there are 18 sectors per track.
- A Cylinder is a group of tracks with the same radius (The Red tracks in the picture above are one cylinder)
- Floppy Disks have two heads (Displayed in the picture)
- There is 2880 Sectors total.

What does this mean for us? The Cylinder number basically represents a track number on a single disk. In the case of a floppy disk, **It represents the Track to r from**.

In summary, there are 512 bytes per sector, 18 sectors per track, and 80 tracks per side.

```
CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only
```

This is the first sector to begin reading from. Remember: **There is only 18 sectors per track. This means that this value can only be between 0 and 17. Y have to increase the current track number, and insure the sector number is correctly set to read the correct sector.**

If this value is greater then 18, The Floppy Controller will generate an exception, because the sector does not exist. Because there is no handler fo The CPU will generate a second fault exception, which will ultimately lead to a Triple Fault.

```
DH = Head Number
```

Remember that some floppys have **two heads**, or sides, to them. **Head 0** is on the front side, where sector 0 is. Because of this, **We are going to be reading fr Head 0.**

If this value is greater then 2, The Floppy Controller will generate an exception, because the head does not exist. Because there is no handler for i CPU will generate a second fault exception, which will ultimately lead to a Triple Fault.

```
DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to
```

What is a **Drive Number**? Its simply a number that, of course, represents a drive. **Drive Number 0 useually represents a floppy drive.** Drive Number 1 is use for 5-1/4" Floppy drives.

Because we are on a floppy, we want to read from the floppy disk. So, **the drive number to read from is 0.**

ES:BX stores the segment:offset base address to read the sectors into. Remember that the Base Address represents the **starting** address.

With this all in mind, lets try to read a sector.

Reading and loading a sector

To read a sector from disk, first reset the floppy drive, and just read:

```
.Reset:
        mov             ah, 0                                   ; reset floppy disk function
        mov             dl, 0                                   ; drive 0 is floppy drive
        int             0x13                                    ; call BIOS
        jc              .Reset                                  ; If Carry Flag (CF) is set, there was an error. Try resetting again

        mov             ax, 0x1000                              ; we are going to read sector to into address 0x1000:0
        mov             es, ax
        xor             bx, bx

.Read:
        mov             ah, 0x02                                ; function 2
        mov             al, 1                                   ; read 1 sector
        mov             ch, 1                                   ; we are reading the second sector past us, so its still on track 1
        mov             cl, 2                                   ; sector to read (The second sector)
        mov             dh, 0                                   ; head number
        mov             dl, 0                                   ; drive number. Remember Drive 0 is floppy drive.
        int             0x13                                    ; call BIOS - Read the sector
        jc              .Read                                   ; Error, so try again

        jmp             0x1000:0x0                              ; jump to execute the sector!
```

Note: If there is a problem reading the sectors, and you try to jump to it to execute it, The CPU will exeute whatever instructions at that address, weather or not the sector was loaded. This useually means the CPU will run into either an invalid/unkown instruction, or the end of memory, both result in a Triple Fault.

The above code only reads and executes a raw sector, which is kind of pointless to our needs. For one,**We currently have PartCopy set up to copy only 512 by** which means: Where and how are we going to create a raw sector?

Also, it is impossible for us to give this Raw Sector a "filename" because it does not exist. Its just a raw sector.

Finally, We currently have the bootloader setup for a FAT12 File System. Windows will attempt to read certain tables (**File Allocation Tables**) from Sector 2 and 3 However, with a Raw Sector, these tables are nonexistant, so Windows will take garbage values (as if it was the table). The result? When reading the floppy disk fr Windows, you will see files and directories with currupt names, and enormous sizes (Have you ever seen a 2.5 Gigabyte file on a 3.14 MB Floppy? I have :))

Of course, **We Will need to load sectors this way.** Before we do, however, we have to find the **Starting Sector, Number of sectors, base address, etc.** of a order to load it properly. This is the bases of loading files off disk.

We will look at this next.

Navigating The FAT12 FileSystem

OEM Parameter Block - Detail

In the previous artical, we dumped an ugly table in our code. What was it? Oh yeah...

```
bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
bpbReservedSectors:     DW 1
bpbNumberOfFATs:            DB 2
bpbRootEntries:             DW 224
bpbTotalSectors:            DW 2880
bpbMedia:                   DB 0xF0
bpbSectorsPerFAT:           DW 9
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
bpbHiddenSectors:           DD 0
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
bsUnused:                   DB 0
bsExtBootSignature:     DB 0x29
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "
```

Alot of this is pretty simple. Lets analyze this in some detail:

```
bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
```

bpbBytesPerSector indicates the number of bytes that represents a sector. This must be a power of 2. Normally for floppy disks, it is 512 bytes.

bpbSectorsPerCluster indicates the number of sectors per **cluster**. In our case, we want one sectorper cluster.

```
bpbReservedSectors:     DW 1
bpbNumberOfFATs:            DB 2
```

A **Reserved Sector** is the number of sectors not included in FAT12. ie, not part of the **Root Directory**. In our case, The **Bootsector**, which contains our bootload not be part of this directory. Because of this, **bpbReservedSectors** should be 1.

This also means that the reserved sectors (Our bootloader) will not contain a File Allocation Table.

bpbNumberOfFATs rpresents the number of **File Allocation Tables (FATs)** on the disk. **The FAT12 File System always has 2 FATs.**

Normally, you would need to create these FAT tables. However, Because we are using VFD, **We can have Windows/VFD to create these tables for us when it formats the disk.**

Note: These tables will also be written to by Windows/VFD when you add or delete entrys. ie, when you add a new file or directory.

```
bpbRootEntries:             DW 224
bpbTotalSectors:            DW 2880
```

Floppy Disks have a maximum of 224 directories within its **Root Directory**. Also, **Remember that there are 2,880 sectors in a floppy disk.**

```
bpbMedia:                   DB 0xF0
bpbSectorsPerFAT:           DW 9
```

The **Media Descriptor Byte** (bpbMedia) is a byte code that contains information about the disk. This byte is a Bit Pattern:

- **Bits 0: Sides/Heads =** 0 if it is single sided, 1 if its double sided
- **Bits 1: Size =** 0 if it has 9 sectors per FAT, 1 if it has 8.
- **Bits 2: Density =** 0 if it has 80 tracks, 1 if it is 40 tracks.
- **Bits 3: Type =** 0 if its a fixed disk (Such as hard drive), 1 if removable (Such as floppy drive)
- **Bits 4 to 7** are unused, and always 1.

0xF0 = 11110000 binary. This means it **is a single sided, 9 sectors per FAT, 80 tracks, and is a movable disk.** Look at **bpbSectorsPerFAT** and you will see is also 9.

```
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
```

Remember: from the previous tutorials/ **There is 18 sectors per track.** bpbHeadsPerCylinder simply represents that there are 2 heads that represents a cylinder. you dont know what a Cylinder is, please read the section "BIOS Interrupt (INT) 0x13" on Reading Sectors.)

```
bpbHiddenSectors:           DD 0
```

This represents the number of sectors from the start of the physical disk and the start of the volume.

```
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
```

Remember that the floppy drive is Drive 0?

```
bsUnused:                   DB 0
bsExtBootSignature:     DB 0x29
```

The Boot Signiture represents the type and version of this **BIOS Parameter Block** (This OEM Table) is. The values are:

- 0x28 and 0x29 indicate this is a MS/PC-DOS version 4.0 Bios Parameter Block (BPB)

We have 0x29, so this is the version we are using.

```
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "
```

The serial number is assigned by the utility that formats it. The serial number is unique to that particular floppy disk, and no two serial numbers should be identica

Microsoft, PC, and DR-DOS base the Seral number off of the current time and date like this:

```
Low 16 bits = ((seconds + month) << 8) + (hundredths + day_of_month)

High 16 bits = (hours << 8) + minutes + year
```

Because the serial number is overwritten, we could put whatever we want in it--it doesnt matter.

The Volume Lable is a string to indicate what is on the disk. Some OSs display this as its name. **Note: This string *Must* be 11 bytes. No more, and no less.**

The Filesystem string is used for the same purpose, and no more. **Note: This string *must* be 8 bytes, no more and no less.**

Demo

Wow, thats alot of stuff, huh? The following is the bootloader I developed in this tutorial, that puts everything together.

Please note: This demo will not work *as is*. It was originally intended for demonstration purposes only, and is not buildable in its current state. I plan to upd this tutorial and make the demo buildable sometime during the next revision of the series.

```
;*****************************************
;       Boot1.asm
;               - A Simple Bootloader
;
;       Operating Systems Development Tutorial
;*****************************************

bits    16                                              ; We are still in 16 bit Real Mode
```

```
org             0x7c00                                  ; We are loaded by BIOS at 0x7C00

start:          jmp loader                              ; jump over OEM block

;*************************************************;
;       OEM Parameter block / BIOS Parameter Block
;*************************************************;

TIMES 0Bh-$+start DB 0

bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
bpbReservedSectors:     DW 1
bpbNumberOfFATs:        DB 2
bpbRootEntries:         DW 224
bpbTotalSectors:        DW 2880
bpbMedia:               DB 0xF0
bpbSectorsPerFAT:       DW 9
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
bpbHiddenSectors:       DD 0
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
bsUnused:               DB 0
bsExtBootSignature:     DB 0x29
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "

;************************************
;       Prints a string
;       DS=>SI: 0 terminated string
;************************************

Print:
                        lodsb                               ; load next byte from string from SI to AL
                        or                      al, al      ; Does AL=0?
                        jz                      PrintDone   ; Yep, null terminator found-bail out
                        mov                     ah,     0eh ; Nope-Print the character
                        int                     10h
                        jmp                     Print       ; Repeat until null terminator found
PrintDone:
                        ret                                 ; we are done, so return

;*************************************************;
;       Bootloader Entry Point
;*************************************************;

loader:

.Reset:
        mov             ah, 0                               ; reset floppy disk function
        mov             dl, 0                               ; drive 0 is floppy drive
        int             0x13                                ; call BIOS
        jc              .Reset                              ; If Carry Flag (CF) is set, there was an error. Try resetting again

        mov             ax, 0x1000                          ; we are going to read sector to into address 0x1000:0
        mov             es, ax
        xor             bx, bx

        mov             ah, 0x02                            ; read floppy sector function
        mov             al, 1                               ; read 1 sector
        mov             ch, 1                               ; we are reading the second sector past us, so its still on track 1
        mov             cl, 2                               ; sector to read (The second sector)
        mov             dh, 0                               ; head number
        mov             dl, 0                               ; drive number. Remember Drive 0 is floppy drive.
        int             0x13                                ; call BIOS - Read the sector


        jmp             0x1000:0x0                          ; jump to execute the sector!


times 510 - ($-$$) db 0                                     ; We have to be 512 bytes. Clear the rest of the bytes with 0

dw 0xAA55                                                   ; Boot Signiture

; End of sector 1, beginning of sector 2 --------------------------------


org 0x1000                                                  ; This sector is loaded at 0x1000:0 by the bootsector

cli                                                         ; just halt the system
hlt
```

Conclusion

We went in alot of detail about disk reading and the BIOS Parameter Block (BPB). We even developed a simple demo that combined everything together.

We also have taken a look at the different rings in assembly language, and learned that our OS is at Ring 0, which differenates it then most other programs. This a us to use more special privedged instructions that application programs dont have.

Now, we have everything we need to find and load our second stage loader! We will learn everything about FAT12, and load the second stage in the next tutorial. I wait! :) 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 4 Home Chapter 6

Operating Systems Development Series

Operating Systems Development - Bootloaders 4

by Mike 2009

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

Introduction

Welcome! In the previous tutorial we talked about how to load and execute a sector. We also have taken a look at the Rings in assembly language, and a detailed look at the BIOS Parameter Block (BPB).

In this tutorial, we are going to use everything we learned to parse the FAT12 file system, and load our second stage loader by name,

This tutorial is going to have alot of code in it. I will do my best to explain everything in detail. Also, this tutorial will have some math in it as well.

Ready?

cli and hlt

You might be curious at why I end all of my demo programs with the instructions "cli" and "hlt". Its actualy pretty simple. If there is no way of stopping the program in some manner, the CPU will just go beyond your program and execute random instructions. This will, ultimately, end in a Triple Fault.

The reason I clear interrupts (cli) as because the interrupts will execute (hence the system is not haulted) even when I wanted to. This can cause problems. So, just having a **hlt** instruction (without **cli**) can Triple Fault the cpu.

Because of this, I always end all of my demos with **cli and hlt.**

Filesystems - Theory

Yippe! Its time to talk about filesystems :)

A **File System** is nothing more then a specification. It helps create the concept of a "file" on a disk.

A **file** is a group of data that has represents something. This data can be anything we want it to be. It all depends on how we interprate the data.

As you know, a sector is 512 bytes in size. A **file** is stored across these sectors on disk. If the file exceeds 512 bytes, we have to give it more sectors. Because not all files are evenly 512 bytes in size, we will need to fill in the rest of the bytes (That the file doesnt use). Kind of like what we did for our bootloader.

If a file spans accross several sectors, we call these sectors a **Cluster** in the FAT file systems. For example, our kernel will most likley span alot of sectors. To load our kernel, we will need to load the **cluster** (The sectors) from where it is located.

If a file spans across different sectors (Not contigous) across different clusters, it is said to be **Fragmented**. We will need to collect the different parts of the file.

There are alot of different kinds of file systems. Some are widley use (Like FAT12, FAT16, FAT32, NTFS, ext (Linux), HFS (Used in older MACs); other filesystems are only used by specific companies for in house use (Like the GFS - Google File System).

Alot of OS developers also create there on versions of the FAT file systems (or even something completely new). These are useually not as good as the most common filesystems though (Like FAT and NTFS).

Okay, so we know a little about file systems now. We are going to be using FAT12 for its simplicity. If we decide, we can always use a different one. :)

FAT12 Filesystem - Theory

FAT12 is the first FAT (File Allocation Table) Filesystem released in 1977, and used in Microsoft Disk BASIC. FAT12, as being an older filesystem generally released for floppr disks, had a number of limtations.

- FAT12 has no support for hierarchical directories. This means there is only one directory-**Thr Root Directory**.
- Cluster Addresses were only 12 bits long, which limits the maximum number of clusters to 4096
- **The Filenames are stored in the FAT as a 12 bit identifier. The Cluster Addresses represent the starting clusters of the files.**
- Because of the limited cluster size, **The maximum number of files possible is 4,077**
- The Disk Size is stored only as a 16 bit count of sctors, limiting it to 32 MiB in size
- FAT12 uses the value "0x01" to identify partitions

These are some big limitations. Why do we want FAT12 then?

FAT16 has support for directories, and over 64,000 files as it uses a 16 bit cluster (file) address, as apposed to FAT12. However, **FAT16 and FAT12 are very simular.**

To make things simple, we are going to use FAT12. We might spruce things up with FAT16 (or even use FAT32) later :) (FAT32 is quite different then FAT 12/16, so we might ust use FAT16 later.)

FAT12 FileSystem - Disk Storage

To understand more about FAT12, and how it works, it is better to look at the structure of a typical formatted disk.

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.

This is a typical formatted FAT12 disk, from the bootsector to the very last sector on disk.

Understanding this structure will be important when loading and searching for our file.

Note that there are 2 FATs on a disk. It is located *right after* the reserved sectors (or the bootloader, if there is none).

Also note: The Root Directory is right after all of the FATs. This means...

if we add up the number of sectors per FAT, and the reserved sectors, we can get the first sector to the Root Directory. By searching the Root Directory for a simple string (our filename), we can effectivly find the exact sector of the file on disk :)

Lets look closer...

Boot Sector

This section contains the BIOS Parameter Block and the bootloader. Yep--Ours. The BIOS Parameter Block contains information tat help describe our disk.

Extra Reserved Sectors

Remember the bpbReservedSectors member of our BPB? Any extra reserved sectors are stored here, right after the bootsector.

File Allocation Tables (FATs)

Remember that a cluster represents a series of contiguous sectors on disk. the size of each cluster is normally 2 KB to 32 KiB. The file peices are linked (from one cluster to another using a common data structure, such as a **Linked List**.

There are two FATs. However, one is just a copy of the first one for data recovery purposes. It useally isnt used.

The File Allocation Table (FAT) is a list of entrys that map to each of these clusters. They help identify important information to aid in storing data to these clusters.

Each entry is a 12 bit value that represents a cluster. **The FAT is a linked list-like structure with these entrys that helps identify what clusters are in use**.

To better nderstand this lets look at the possible values:

- **Value marks free cluster :** 0x00
- **Value marks Reserved cluster :** 0x01
- **This cluster is in use--the value represents next cluster :** 0x002 through 0xFEF
- **Reserved values :** 0xFF0 through 0xFF6
- **Value marks bad clusters :** 0xFF7
- **Value marks this cluster as the last in the file :** 0xFF8 through 0xFFF

A FAT is just an array of these values--thats all. When we find the starting sector form the Root Directory, we can look through the FAT to find which clusters to load. How? We simply check the value. If the value is between 0x02 and 0xfef, this value represents the next cluster to load for the file.

Lets look at this in a deeper way. A **cluster**, as you know, represents a series of sectors. We define the amount of sectors it represents from the BIOS Paramete Block:

```
bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
```

In our case, each cluster is 1 sector. When we get the first sector of Stage 2 (We get this from the root directory), we use this sector as the starting cluster number in the FAT. Once we find the starting cluster, we just refrence the FAT to determin the cluster (The FAT is just an array of 32 bit numbers. We just compare this number with the list above to determin what to do with it.)

The Root Directory Table

Now, **THIS** will be important to us :)

The root directory is a table of 32 byte values that represent information reguarding file and directories. This 32 byte value uses the format:

- **Bytes 0-7 : DOS File name (Padded with spaces)**
- **Bytes 8-10 : DOS File extension (Padded with spaces)**
- **Bytes 11 :** File attributes. This is a bit pattern:
  - **Bit 0 :** Read Only
  - **Bit 1 :** Hidden
  - **Bit 2 :** System
  - **Bit 3 :** Volume Label
  - **Bit 4 :** This is a subdirectory
  - **Bit 5 :** Archive
  - **Bit 6 :** Device (Internal use)
  - **Bit 6 :** Unused
- **Bytes 12 :** Unused
- **Bytes 13 :** Create time in ms
- **Bytes 14-15 :** Created time, using the following format:
  - **Bit 0-4 :** Seconds (0-29)
  - **Bit 5-10 :** Minutes (0-59)
  - **Bit 11-15 :** Hours (0-23)
- **Bytes 16-17 :** Created year in the following format:
  - **Bit 0-4 :** Year (0=1980; 127=2107
  - **Bit 5-8 :** Month (1=January; 12=December)
  - **Bit 9-15 :** Hours (0-23)

- **Bytes 18-19 :** Last access date (Uses same format as above)
- **Bytes 20-21 :** EA Index (Used in OS/2 and NT, dont worry about it)
- **Bytes 22-23 :** Last Modified time (See byte 14-15 for format)
- **Bytes 24-25 :** Last modified date (See bytes 16-17 for format)
- **Bytes 26-27 : First Cluster**
- **Bytes 28-32 : File Size**

I bolded the important parts--everything else is just junk Microsoft added that we can add to when we create a FAT12 driver, much later.

Wait a sec! Remember that DOS filenames are limited to 11 bytes? This is why:

- **Bytes 0-7 : DOS File name (Padded with spaces)**
- **Bytes 8-10 : DOS File extension (Padded with spaces)**

0 through 10, hmm... thats 11 bytes. Having a filename less then 11 bytes will miss up the data entry (The 32 byte entry table displayed above). This, of course, is bad :) Because of this, we have to pad the filenames with characters, and insure it is 11 bytes.

Remember in a previous tutorial I explained how their are **internal** and **external** filenames? The filename structure I explained is the internal filename. As it is limited to 11 bytes, the filename **"Stage2.sys" has to become**

```
"STAGE2  SYS" (Note the padding!)
```

Searching and reading FAT12 - Theory

Okay, after reading all of the above, you are probably tired of me saying "FAT12" :)

Anywhoo... How is hie information usefull to us?

We are going to be refrencing the BIOS Parameter Block (BPB) alot. Here is the BPB that we created from the prevuis tutorials for refrence:

```
bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
bpbReservedSectors:     DW 1
bpbNumberOfFATs:        DB 2
bpbRootEntries:         DW 224
bpbTotalSectors:        DW 2880
bpbMedia:               DB 0xF0
bpbSectorsPerFAT:       DW 9
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
bpbHiddenSectors:       DD 0
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
bsUnused:               DB 0
bsExtBootSignature:     DB 0x29
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "
```

Please see the previous tutorial for a detailed explanation of each member.

What we are trying to do is to load a second stage loader. Lets look at what we need to do in detail:

Beginning with a filename

The first thing to do is to create a good filename. Remember: **The Filenames must be exactally 11 bytes to insure we don't corrupt the root directory.**

I am using "STAGE2.SYS", for my second stage. You can look at an example of its internal filename in the above section.

Creating Stage 2

Okay, Stage2 is a seperate program then the bootloader. Our Stage2 will be very simular to a DOS COM program, sound cool?

All Stage2 does right now is print a message and halt. Everything you have already seen from the bootloader:

```
; Note: Here, we are executed like a normal
; COM program, but we are still in Ring 0.
; We will use this loader to set up 32 bit
; mode and basic exception handling

; This loaded program will be our 32 bit Kernel.

; We do not have the limitation of 512 bytes here,
; so we can add anything we want here!

org 0x0         ; offset to 0, we will set segments later

bits 16         ; we are still in real mode

; we are loaded at linear address 0x10000

jmp main        ; jump to main

;*********************************************;
;       Prints a string
```

```
;       DS=>SI: 0 terminated string
;*********************************************;

Print:
                        lodsb           ; load next byte from string from SI to AL
                        or      al, al  ; Does AL=0?
                        jz      PrintDone       ; Yep, null terminator found-bail out
                        mov     ah,     0eh     ; Nope-Print the character
                        int     10h
                        jmp     Print   ; Repeat until null terminator found
PrintDone:
                        ret             ; we are done, so return

;*********************************************;
;       Second Stage Loader Entry Point
;*********************************************;

main:
                        cli             ; clear interrupts
                        push    cs      ; Insure DS=CS
                        pop     ds

                        mov     si, Msg
                        call    Print

                        cli             ; clear interrupts to prevent triple faults
                        hlt             ; hault the system

;*********************************************;
;       Data Section
;*********************************************;

Msg     db      "Preparing to load operating system...",13,10,0
```

To assemble with NASM, just assemble it as a binary program (COM programs are binary), and copy it into the floppy disk image. For example:

```
nasm -f bin Stage2.asm -o STAGE2.SYS

copy STAGE2.SYS  A:\STAGE2.SYS
```

No PARTCOPY required :)

Step 1: Loading the Root Directory Table

Now its time to load Stage2.sys! We will be refrencing the Root directory table alot here, along with the BIOS parameter block for disk information.

Step 1: Get size of root directory

Okay, first we need to get the size of the root directory.

To get the size, just multiply the number of entrys in the root directory. Seems simple enough :)

In Windows, whenever you add a file or directory to a FAT12 formatted disk, Windows automatically adds the file information to the root directory, so we dont need to worry about it. This makes things much simpler.

Dividing the number of root entrys by bytes per sector will tell us how many sectors the root entry uses.

Here is an example:

```
        mov     ax, 0x0020        ; 32 byte directory entry
        mul     WORD [bpbRootEntries]  ; number of root entrys
        div     WORD [bpbBytesPerSector] ; get sectors used by root directory
```

Remember that the root directory table is a table of **32 byte values (entrys)** that reprsent the file information.

Yippe--Okay, we know how much sectors to load in for the root directory. Now, lets find the starting sector to load from :)

Step 2: Get start of root directory

This is another easy one. First, lets look at a FAT12 formatted disk again:

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.

Okay, note that the **Root Directory is located directly after both FATs and reserved sectors**. In other words, just add the FATs + reserved sectors, and you found the root direcory!

For example...

```
        mov     al, [bpbNumberOfFATs]  ; Get number of FATs (Useually 2)
        mul     [bpbSectorsPerFAT]  ; number of FATs * sectors per FAT; get number of sectors
        add     ax, [bpbReservedSectors] ; add reserved sectors

      ; Now, AX = starting sector of root directory
```

Pretty easy, huh? Now, we just read the sector to some location in memory:

```
        mov     bx, 0x0200  ; load root directory to 7c00:0x0200
        call    ReadSectors
```

Root Directory - Complete example

This example code is taken directly from the bootloader a the end of the tutorial. It loads the root directory:

```
LOAD_ROOT:

 ; compute size of root directory and store in "cx"

      xor     cx, cx
      xor     dx, dx
      mov     ax, 0x0020                      ; 32 byte directory entry
      mul     WORD [bpbRootEntries]           ; total size of directory
      div     WORD [bpbBytesPerSector]        ; sectors used by directory
      xchg    ax, cx

 ; compute location of root directory and store in "ax"

      mov     al, BYTE [bpbNumberOfFATs]       ; number of FATs
      mul     WORD [bpbSectorsPerFAT]          ; sectors used by FATs
      add     ax, WORD [bpbReservedSectors]    ; adjust for bootsector
      mov     WORD [datasector], ax            ; base of root directory
      add     WORD [datasector], cx

 ; read root directory into memory (7C00:0200)

      mov     bx, 0x0200                        ; copy root dir above bootcode
      call    ReadSectors
```

Step 2: Find Stage 2

Okay, now the root directory table is loaded. Looking at the above code, **we loaded it to 0x200.** Now, to find our file.

Lets look back at the 32 byte root directory table again (Section **Root Directory Table**. Remember **the first 11 bytes represent the filename**. Also remember that, **because each root directory entry is 32 bytes, Every 32 bytes will be the start of the next entry - Pointing us back to the first 11 bytes of the next entry.**

Hence, all we need to do is compare filenames, and jump to the next entry (32bytes), and test again until we reach the end of the sector. For example...

```
 ; browse root directory for binary image
      mov     cx, [bpbRootEntries]        ; the number of entrys. If we reach 0, file doesnt exist
      mov     di, 0x0200          ; Root directory was loaded here
 .LOOP:
      push    cx
      mov     cx, 11              ; eleven character name
      mov     si, ImageName       ; compare the 11 bytes with the name of our file
      push    di
 rep  cmpsb                       ; test for entry match
      pop     di
      je      LOAD_FAT            ; they match, so begin loading FAT
      pop     cx
      add     di, 32              ; they dont match, so go to next entry (32 bytes)
      loop    .LOOP
      jmp     FAILURE             ; no more entrys left, file doesnt exist :(
```

On to the next step...

Step 3: Loading FAT

Step 1: Get start cluster

Okay, so the root directory is loaded and we found the files entry. How do we get its starting cluster?

- Bytes 26-27 : First Cluster
- Bytes 28-32 : File Size

This should look familiar :) To get the starting cluster, just refrence byte 26 in the file entry:

```
mov     dx, [di + 0x001A]   ; di contains starting address of entry. Just refrence byte 26 (0x1A) of entry

; Yippe--dx now stores the starting cluster number
```

The starting cluster will be important to us when loading the file.

Step 2: Get size of FAT

Lets look at the BIOS parameter block again. More specifically...

```
bpbNumberOfFATs:        DB 2
bpbSectorsPerFAT:       DW 9
```

Okay, so how do we find out how many sectors there are in both FATs? Just multiply sectors per FAT by the number of sectors :) Seems simple, ...but...

```
        xor     ax, ax
        mov     al, [bpbNumberOfFATs]               ; number of FATs
        mul     WORD [bpbSectorsPerFAT]             ; multiply by number of sectors per FAT

      ; ax = number of sectors the FATs use!
```

No, never mind, it is simple ^^

Step 3: Load the FAT

Now that we know how many sectors to read. Just, um... read it :)

```
        mov     bx, 0x0200                          ; address to load to
        call    ReadSectors                         ; load the FAT table
```

Yey! Now with the FAT stuff out of the way (Not completely!), load in stage 2!

FAT - Complete example

Here is the complete code taken directly from the bootloader:

```
LOAD_FAT:

 ; save starting cluster of boot image

        mov     si, msgCRLF
        call    Print
        mov     dx, WORD [di + 0x001A]
        mov     WORD [cluster], dx                  ; file's first cluster

 ; compute size of FAT and store in "cx"

        xor     ax, ax
        mov     al, BYTE [bpbNumberOfFATs]              ; number of FATs
        mul     WORD [bpbSectorsPerFAT]               ; sectors used by FATs
        mov     cx, ax

 ; compute location of FAT and store in "ax"

        mov     ax, WORD [bpbReservedSectors]         ; adjust for bootsector

 ; read FAT into memory (7C00:0200)

        mov     bx, 0x0200                          ; copy FAT above bootcode
        call    ReadSectors
```

LBA and CHS

In loading the image, all we need to do is load each cluster by refrencing the FAT.

There is one little problem we havnt discussed yet though. Okay, We have a cluster number from the FAT. But, **How do we use it?**

The problem is that this cluster represents a linear address, while, in order to load sectors, we will need a segment/track/head address. (Interrupt 0x13)

There are **two** ways to access a disk. Either through **Cylinder/Head/Sector (CHS) addressing** or **Liogical Block Addressing (LBA)**.

The **LBA** represents an indexed location on disk. The first block being 0, then 1, and so on. LBA simply represents sectors are sequentially numbered with LBA 0. Cant get more basic then that.

You will need to know how to convert between LBA and CHS.

Converting CHS to LBA

The formula to convert CHS to LBA:

```
LBA     =       (cluster - 2 ) * sectors per cluster
```

That is simple enough. :) Heres an example:

```
        sub     ax, 0x0002                          ; subtract 2 from cluster number
        xor     cx, cx
        mov     cl, BYTE [bpbSectorsPerCluster]     ; get sectors per cluster
        mul     cx                                  ; multply
```

Converting LBA to CHS

This is a little bit more complex, but still is relaitevly easy:

```
     absolute sector   =       (LBA % sectors per track) + 1
     absolute head     =       (LBA / sectors per track) % number of heads
     absolute track    =        LBA / (sectors per track * number of heads)
```

Heres an example...

```
LBACHS:
         xor     dx, dx                          ; prepare dx:ax for operation
         div     WORD [bpbSectorsPerTrack]       ; divide by sectors per track
         inc     dl                              ; add 1 (obsolute sector formula)
         mov     BYTE [absoluteSector], dl

; these forumlas are very simular...

         xor     dx, dx                          ; prepare dx:ax for operation
         div     WORD [bpbHeadsPerCylinder]      ; mod by number of heads (Absolue head formula)
         mov     BYTE [absoluteHead], dl         ; everything else was already done from the first formula

         mov     BYTE [absoluteTrack], al        ; not much else to do :)
         ret
```

Not to hard, I hope :)

Load the cluster

Okay, in loading Stage 2, we first need to refrence the cluster from the FAT. Pretty simple. Then, conver the cluster number to LBA so we can read it in:

```
         mov     ax, [cluster]                   ; cluster to read
         pop     bx                              ; buffer to read into
         call    ClusterLBA                      ; convert cluster to LBA
         xor     cx, cx
         mov     cl, [bpbSectorsPerCluster]      ; sectors to read
         call    ReadSectors                     ; read in cluster
         push    bx
```

Get next cluster

This is tricky.

Okay, remember **each cluster number in the FAT entry is 12 bits**. This is a problem. **If we read in 1 byte, we are only copying a part of the cluster number!**.

Because of this, we have to read a WORD (2 byte) value.

Yet, then again, we run into a problem. Copying 2 bytes (from a 12 bit value) means that we will copy a **part of the next cluster entry**. For example, imagine this is your FAT:

```
             Note: Binary numbers seperated in bytes.
             Each 12 bit FAT cluster entry is displayed.

                                 |                           |
  01011101       0111010   01110101  00111101  0011101  0111010  0011110 0011110
  |                |             |              |           |               |
  |                |1st cluster  |              |3rd cluster-|               |
  |-0 cluster  ----|             |2nd cluster---|           |4th cluster----|
```

Notice all even clusters accopy all of the first byte, but part of the second. Also notice that all odd clusters occopy a part of their first byte, but all of the second!

Okay, so what we need to do is to read a 2byte (word) value from the FAT (This is our cluster).

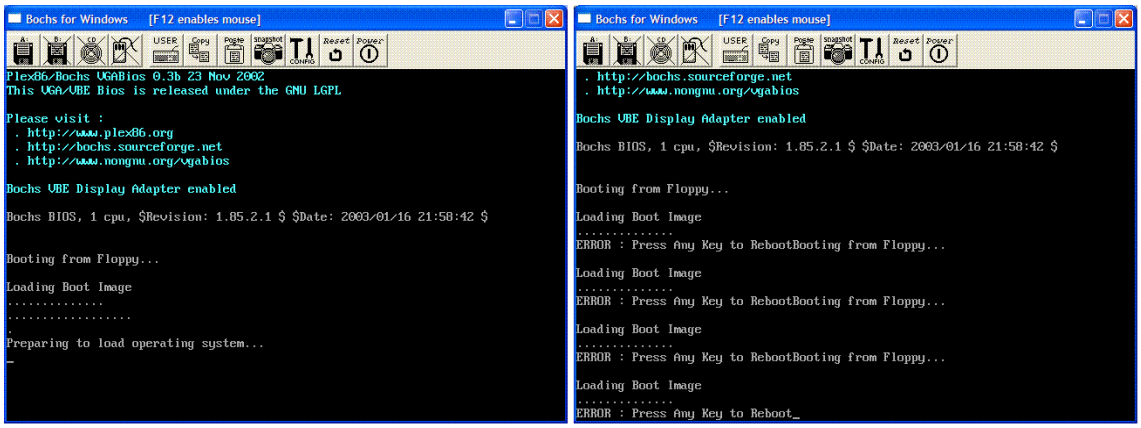
If the cluster is even, **Mask out the top 4 bits, as it belongs to the next cluster.**

If it is odd, **shift it down 4 bits (to discard the bits used by the first cluster.)** For example...

```
    ; compute next cluster

         mov     ax, WORD [cluster]  ; identify current cluster from FAT

        ; is the cluster odd or even? Just divide it by 2 and test!

         mov     cx, ax              ; copy current cluster
         mov     dx, ax              ; copy current cluster
         shr     dx, 0x0001          ; divide by two
         add     cx, dx              ; sum for (3/2)

         mov     bx, 0x0200          ; location of FAT in memory
         add     bx, cx              ; index into FAT
         mov     dx, WORD [bx]       ; read two bytes from FAT
```

```
        test    ax, 0x0001
        jnz     .ODD_CLUSTER

; Remember that each entry in the FAT is a 12 but value. If it represents
; a cluster (0x002 through 0xFEF) then we only want to get those 12 bits
; that represent the next cluster

    .EVEN_CLUSTER:

        and     dx, 0000111111111111b      ; take low twelve bits
       jmp     .DONE

    .ODD_CLUSTER:

        shr     dx, 0x0004                 ; take high twelve bits

    .DONE:

        mov     WORD [cluster], dx         ; store new cluster
        cmp     dx, 0x0FF0                 ; test for end of file
        jb      LOAD_IMAGE    ; we are not done yet--go to next cluster
```

Demo



The first shot contains the bootloader loading Stage 2 successfully. Stage 2 prints the loading operating system message.

The second shot displays an error message when it cannot find the file (within the root directory).

This demo contains most of the code in this lesson, 2 source files, 2 directories, and 2 batch programs. The first directory contains the stage 1 program -- our bootloader, the second directory contains our stage 2 program - STAGE2.SYS.

DEMO DOWNLOAD HERE

Conclusion

Wow, this tutorial was hard to write. Simply because it is hard explaining such a complex topic in every detail, while still trying to make it very easy to follow. I hope I did well :)

If you have any suggestions that could imporove this tutorial, please let me know :)

Well... I guess thats it: **Good bye bootloader!**

In the next tutorial, we will begin building on Stage 2. We will talk about A20, and look at **Protected Mode** in alot more detail...

See you there!

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 5 Home Chapter 7

Operating Systems Development - System Architecture

by Mike, 2008

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

Introduction

Welcome! :) In the previous tutorial we *finally* finished the bootloader! Yey! For now, anyway :)

We covered the FAT12 file system in detail, and looked at loading, parsing, and executing stage 2.

This tutorial will continue where the last one went off. We are going to first look at the x86 architecture in detail. This will be important to us, espically in protected mode, and understanding how protected mode works.

We are going to cover every single thing of how the computer works and operates down to the bit level. To understand how this fits in with the BIOS during bootup, you have to remember that you can "start" other processors. The BIOS does just this with the main processor, and we can do the same to support multi processor systems.

We will cover:

- The 80x86 Registers
- System Orginazation
- The System Bus
- Real Mode Memory Map
- How an instruction executes
- Software Ports

In some ways, this is like a system architecture tutorial. However, we are going to look at the architecture from an OS development point of view. Also, **We will cover every single thing within the architecture.**

Understanding the basic concepts will make understanding **Protected Mode** in alot more detail. In the next tutorial, we are going to use everything we learn here to switch into protected mode.

Lets have some fun, shall we...?

The World of Protected Mode

We all heard this term before, havnt we? Protected Mode (PMode) is an operation mode avilable from the 80286 and later processors. PMode was primarily designed to increase the stabaility of the systems.

As you know from the previous tutorials, Real Mode has some big problems. For one, we can write a byte anywhere we want. This can overwrite code or data, that may be used by software ports, the processor, or even ourself. And yet, we can do this in over 4,000 different ways--both directly and indirectly!

Real Mode has no **Memory Protection**. All data and code are dumped into a single all purpose use memory block.

In Real Mode, you are limited to 16 bit regisers. Because of this, you are limited to 1 MB of memory.

No support for hardware level **Memory Protection** or **Multitasking**.

Quite possibly the biggest problem, was that there is no such thing as "rings". All programs execute at Ring 0 level, as every program has full control over the system. This means, in a single tasking envirement, a single instruction (such as **cli/hlt**) can crash the entire OS if you are not carefull.

Alot of this should sound familiar from when we covered Real Mode in depth. Protected Mode fixes all of these problems.

Protected Mode:

- Has **Memory Protection**
- Has hardware support for **Virtual Memory** and **Task State Switching (TSS)**
- Hardware support for interruting programs and executing another
- 4 Operating Modes: **Ring 0, Ring 1, Ring 2, Ring 3**
- Access to 32 bit registers
- Access to up to 4 GB of memory

We covered the Rings of Assembly Language in a previous tutorial. Remember that **we are in Ring 0, while normal applications are in Ring 3 (Useually).** We have access to special instructions and registers that normal applications do not. In this tutorial, we are going to be using the **LGDT** instruction, along with a **far jump** using our own defined segment, and the use of the **processor control registers.** None of this is avilable in normal programs.

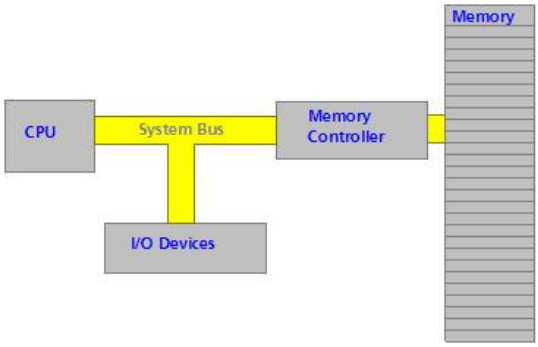
Understanding the system architecture and how the processor works will help us understand this alot better.

System Architecture

The x86 family of computers follow the *Van Neumann Architecture*. The Van Nuemann Architecture is a design specification that states a typical computer system has three main components:

- Central Processing Unit (CPU)
- Memory
- Input/Output (IO)

For example:



There are a couple of important things to note. As you know, the CPU fetches data and instructions from memory. The Memory Controller is responsible for calculating the exact RAM chip and memory cell that it resides in. Because of this, **The CPU communicates with the Memory Controller.**

Also, notice the "I/O Devices". They are connected to the system bus. **All I/O Ports are mapped to a given memory location. This allows us to use the IN and OUT instructions.**

The hardware devices can access memory through the System Bus. It also allows us to notify a device when something is happening. For example, if we write a byte to a memory location for a **hardware device controller** to read, **the processor can signal the device that there are data at that address.** It does this through the **Control Bus** part of the entire **System Bus**. This is basically how software interacts with hardware devices. We will go into much more detail later as this is the **only** way to communicate to devices in protected mode, hence this is important.

We will cover everything in detail first. Then, we will combine them and learn how they all work together, by watching an instruction get executed at the hardware level. From here, we will talk about I/O Ports, and how software interacts with hardware.

As you have experience with x86 Assembly, Some or even alot of this should be familiar. However, we are going to cover alot of things most assembly books dont cover in detail. More specifically, things specific to Ring 0 programs.

The System Bus

The **System Bus** is also known as the **Front Side Bus** that connects the CPU to the **Northbridge** on a motherboard.

The System Bus is a combination of the **Data Bus**, **Address Bus**, and **Control Bus**. **Each electronic line on this bus represents a single bit**. The voltage level used to represent a "zero" and "one" is based off **Standard Transistor-Transistor Logic (TTL) Levels.** We dont need to know this though. TTL is a part of **Digital Logic Electronics**, at which computers are built.

As you know, the System Bus is made up of 3 buses. Lets look at them in detail, shall we?

Data Bus

The data bus is the series of electronic lines that which data can be carried over. The size of the data bus is 16 lines/bits, 32 lines/bits, or 64 lines/bits. Note the direct relationship between an electronic line and a single bit.

This means, **A 32 bit processor has and uses a 32 bit data bus.** This means, it can handle a 4 byte peice of data simutaniously. Knowing this, we can watch the data sizes in our programs, and help increase speed.

How? The processor will need to pad 1,2,4,8, and 16 bit data to the size of the data bus with 0's. Larger peices of data will need to be broken (and padded) so the processor can send the bytes correctly over the data bus. **Sending a peice of data that is the size of the data bus will be faster because no extra processing is done.**

For example, lets say we have a 64 bit data type, but a 32 bit data bus. In the first **Clock Cycle**, only the first 32 bits are sent through the data bus to the Memory Controller. In the second **Clock Cycle**, the processor refrences the last 32 bits. **Note: Notice that, the larger the data type, the more clock cycles it will take!**

Generally, the terms "32 bit processor", "16 bit processor", etc. generally referrs to the size of the data bus. So, a "32 bit processor" uses a 32 bit data bus.

Address Bus

Whenever the processor or an I/O device needs to refrence memory, it places its address on the Address Bus. Okay, we all know that a **Memory Address** represents a location in memory. This is an abstraction though.

A "Memory Address" is **just a number used by the Memory Controller.** Thats it. The Memory Controller takes the number from this bus, and interprets it as a memory location. **Knowing the size of each RAM chip The Memory Controller could easily refrence the exact RAM chip and byte offset in it.** Beginning with **Memory Cell 0**, the Memory Controller intereprates this offset as the **Address that we want**.

The Address Bus is connected to the processor through the **Control Unit (CU)**, and the **I/O Controller**. The **Control Unit** is inside the processor, so we will look at that later. The **I/O Controller** controls the interface to hardware devices. We will look at that later.

Just like with the Data Bus, **Each Electronic line represents a single bit.** Because there are only two unique values in a bit, **There is exactly 2^n unique address that a CPU can access.** Therfore, **The number of bits/lines in the Address Bus represents the maximum memory the CPU can access.**

In the 8080 through 80186 processor each had **20 line/bit address busses**. The 80286 and 80386 has 24 lines/bits, and the 80386+ has 32 lines/bits.

Remember that the entire x86 family is designed to be portable with all older processors. This is why it starts in Real Mode. **Their processor architectures were limited to 1 MB because they only had access to 20 address lines -- line 0 through line 19.**

This is important to us, **Because this limtation still applies to us!** What we need to do is **Enable access through the 20th address line. This will allow our OS to access more then 4 GB of memory.** More on this later.

Control Bus

Okay, we could place data on the Data Bus, and refrence memory addresses using the Address Bus. But, how do we know what to do with this data? Are we reading it from memory? Or we writing the data?

The Control Bus is a series of lines/bits that represent what a device is trying to do. For example, the processor would set the READ bit or WRITE bit to let the Memory Controller know it wants to read or write the data in the Data Bus from the memory location stored in the Address Bus.

The Control Bus also allows the processor to signal a device. This lets a device that we need its attention. For example, perhaps we need the device to read from the memory location from the Address Bus? This will let the device know of what we need. **This is important in I/O Software ports.**

Of course, remember the system bus is not directly connected to hardware devices. Instead, it is connected to a central controller--**The I/O Controller**, which, in turn, signals the devices.

Thats all

Thats all there is to the system bus. It is the pathway for accessing and reading memory from the processor (Through its **Control Unit (CU)**) and the I/O devices (**Through the I/O Controller**) to the **Memory Controller**, which is responsible for calculating the exact RAM chip and finding the memory cell we want to access.

"Controller"...You will hear me say this term alot. I will explain why later.

Memory Controller

The Memory Controller is the primary interface between the System Bus (ak, Front Side Bus(FSB)) on the motherboard to the physical RAM chips.

We seen the term **Controller** before, havnt we? What exactally is a controller?

Controllers

A Controller provides basic hardware control functionaility. **It also provides the basic interface between hardware and software.** This is important to us. Remember that in protected mode, we will **not have *any* interrupts avilable to us**. In the bootloader, we used several interrupts to communicate with the hardware. **Using these interrupts in protected mode will cause a Triple Fault.** Yikes--so what are we to do?

We will need to communicate to the hardware directly. We do this through the controllers. (We will talk more about how controllers work later when we cover the I/O SubSystem).

Memory Controller

The Memory Controller provides a way of readng and writing memory locations through software. The Memory Controller is also responsible for the constant refreshing of the RAM chips to insure they retain the information.

Memory Controllers a **Multiplexer** and **Demultiplexer** circuits to select the exact RAM chip, and location that refrences the address in the Address Bus.

Double Data Rate (DDR) Controller

A DDR Controller is used to refresh DDR SDRAM, which uses the **System Clock** pulse to allow reading and writing memory.

Dual Channel Controller

A Dual Channel Controller are used where DRAM devices are seperated into two small busses, allowing reading and writing two memory locations at once. This helps increasing speed when accessing RAM.

Memory Controller Conclusion

The Memory Controller takes the address we put into the Address Bus. This is good and all, but how do we tell the Memory Controller to read or write memory? And where does it get its data from? When reading memory, **The processor sets the Read bit in the Control Bus**. Simularily, **The processor sets the Write bit when writing memory on the Control Bus**.

Remember that the **Control Bus** allows the processor to control how other devices use the bus.

The data the Memory Controller uses is inside the Data Bus. The Address to use is in the Address Bus.

Reading Memory

When reading memory, The Processor places the absolute address to read from on the Address Bus. The processor then sets the READ control line.

The Memory Controller now has control. The controller converts the absolute address into a physical RAM location using its **Multiplexer** circuit, and places the data into the Data Bus. It then resets the READ bit to 0, and sets the READY bit.

The processor now knows the data is now in the data bus. It copyies this data, and executes the rest of the instruction...perhaps store it in BX?

Writing Memory

The process of writing memory is simular.

First, the processor places the memory address into the Address Bus. It then places the data to write into the Data Bus. Then, it sets the WRITE bit in the Control Bus.

This lets the Memory Controller know to write the data in the Data Bus to the Absolute address in the Address Bus. When done, the Memory Controller resets the WRITE bit, and sets the READY bit on the Control Bus.

Conclusion

We do not communicate directly with the Memory Controller through software, but instead, we communicate indirectly with it. **Whenever we read or write memory, we are using the Memory Controller. This is the interface between our software and the Memory Controller / RAM Chip Hardware.**

Yippe--Lets take a look at the I/O SubSystem now, shall we? Oh wait! What about that 1337 **Multiplexer** circuit? That is a physical electronic circuit in the Memory Controller. To understand how it works, one has to know **Digital Logic Electronics**. Because this is irrilivent to us, we are not going to cover it here. If you would like to know more, Google!

I/O SubSystem

The I/O SubSystem simply represents **Port I/O**. **This is the basic system that provides the interface between software and hardware controllers.**

Lets look closer...

Ports

A **Port** simply provides an interface between two devices. There are two types of ports: **Hardware Ports** and **Software Ports**.

Hardware Ports

A Hardware Port provides the interface between two physical devices. This port is useually a connection device of sorts. This includes, but is not limited to: **Serial Ports, Parallel ports, PS/2 Ports, 1394, FireWire, USB Ports, etc.**

These ports are useually on the sides/back/or front of a typical computer system.

Okay... um, if you want to see a port, just follow any line that connects to your computer. Please, for the sake of jeeves, Dont ask me what these do--you have got to already now! Seriously!

In typical electronics, the pins in these ports carry signals that represent different things depending on the hardware device. These pins represent, just like the system bus--wait for it... Bits! Each pin represents a single bit. Yep--thats it.

Two general classifications for Hardware Ports include "Male" and "Female" ports. Male ports are connections where the pins emerge from the connector. Female ports are the opposite of this. Hardware ports are accessed through Controllers. More on this later...

Software Ports

THIS will be very important to us. This is our interface to the hardware. A **Software Port** is a number. Thats it. This number represents a hardware controller... Kind of.

You may know that several port numbers could represent the same controller. The reason? **Memory Mapped I/O**. The basic idea is that we communicate to hardware by specifying certain memory addresses. **The port number represents this address.**...Once more, kind of. The meaning of the addresses could represent a specific register in a device, or a control register.

We will look more closer later.

Memory Mapping

On the x86 Architecture, the processor uses specific memory locations to represent certain things.

For example, **The address 0xA000:0 represents the start of VRAM in the video card.** By writing bytes to this location, you effectivly change what is currently in video memory, and effectivly, what is displayed on screen.

Other memory addresses can represent something else--lets say, a register perhaps for the Floppy Drive Controller (FDC)?

Understanding what addresses are what is critical, and very important to us.

x86 Real Mode Memory Map

General x86 Real Mode Memory Map:

- **0x00000000 - 0x000003FF** - Real Mode Interrupt Vector Table
- **0x00000400 - 0x000004FF** - BIOS Data Area
- **0x00000500 - 0x00007BFF** - Unused
- **0x00007C00 - 0x00007DFF** - Our Bootloader
- **0x00007E00 - 0x0009FFFF** - Unused
- **0x000A0000 - 0x000BFFFF** - Video RAM (VRAM) Memory
- **0x000B0000 - 0x000B7777** - Monochrome Video Memory
- **0x000B8000 - 0x000BFFFF** - Color Video Memory
- **0x000C0000 - 0x000C7FFF** - Video ROM BIOS
- **0x000C8000 - 0x000EFFFF** - BIOS Shadow Area
- **0x000F0000 - 0x000FFFFF** - System BIOS

Note: It is possible to remap all of the above devices to use different regions of memory. This is what the BIOS POST does to map the devices to the table above.

Okay, this is cool and all. Because these addresses represent different things, by reading (or writing) to specific addresses, we get obtain (or change) information with ease from different parts of the computer.

For example, remember when we talked about **INT 0x19**? We refrenced that writing the value 0x1234 at 0x0040:0x0072, and jumping to 0xFFFF:0, we effectivly warm reboot the computer. (Simular to Windows ctrl+alt+del.) Remembering the conversion between seg:offset addressing mode and absolute addressing, we can convert **0x0040:0x0072** to the absolute address **0x000000472**, a byte within the BIOS data area.

Another example is text output. But writing two bytes into **0x000B8000**, we can effectivly change what is in text mode memory. Because this is constantly refreshed when displayed, it effectivly displays the character on screen. Cool?

Lets go back to port mapping, shall we? We will look back at this table alot more later.

Port Mapping - Memory Mapped I/O

A "Port Address" is a special number that each Controller listens to. **When booting, the ROM BIOS assigns different numbers to these controller devices.** It starts the primary processor, loads the BIOS program at 0xFFFF:0 (**Remember this? Compare this with the table in the previous section**).

The ROM BIOS Assigns these numbers to different controllers, so controllers have a way to identify themselves. This allows the BIOS to set up the Interrupt Vector Table, which communicates to the hardware using this special number.

The processor uses the same system bus when working with I/O Controllers. **The processor puts the special port number into the Address Bus**, as if it was reading memory. It also sets the READ or WRITE lines on the control bus as well. This is cool, but theres a problem: How does the processor differenate between writing memory and accessing a controller?

The processor sets another line on the control bus--An I/O ACCESS line. **If this line is set, The I/O Controllers from within the I/O SubSystem watches the Address Bus. If the Address Bus curresponds to a number that is assigned to the device, that device takes the value from the data bus and acts upon it.** The Memory Controller ignores any request if this line is set. So, if the port number has not been assigned, absolutley nothing happens. No controller acts on it, and the Memory Controller ignores it.

Lets take a look at these port addresses. **This is very important! This is the *only* way of communicating with hardware in protected mode!**:

Warning: This table is large!

Default x86 Port Address Assignments				
Address Range	First QWORD	Second QWORD	Third QWORD	Fourth QWORD
0x000-0x00F	DMA Controller Channels 0-3			
0x010-0x01F	System Use			
0x020-0x02F	Interrupt Controller 1	System Use		
0x030-0x03F	System Use			
0x040-0x04F	System Timers	System Use		
0x050-0x05F	System Use			
0x060-0x06F	Keyboard/PS2 Moude (Port 0x60) Speaker (0x61)	Keyboard/PS2 Mouse (0x64)	System Use	
0x070-0x07F	RTC/CMOS/NMI (0x70, 0x71)	DMA Controller Channels 0-3		
0x080-0x08F	DMA Page Register 0-2 (0x81 - 0x83)	DMA Page Register 3 (0x87)	DMA Page Register 4-6 (0x89-0x8B)	DMA Page Register 7 (0x8F)
0x090-0x09F	System Use			
0x0A0-0x0AF	Interrupt Controller 2 (0xA0-0xA1)	System Use		
0x0B0-0x0BF	System Use			
0x0C0-0x0CF	DMA Controller Channels 4-7 (0x0C0-0x0DF), bytes 1-16			
0x0D0-0x0DF	DMA Controller Channels 4-7 (0x0C0-0x0DF), bytes 16-32			
0x0E0-0x0EF	System Use			
0x0F0-0x0FF	Floating Point Unit (FPU/NPU/Mah Copprocessor)			
0x100-0x10F	System Use			
0x110-0x11F	System Use			
0x120-0x12F	System Use			
0x130-0x13F	SCSI Host Adapter (0x130-0x14F), bytes 1-16			
0x140-0x14F	SCSI Host Adapter (0x130-0x14F), bytes 17-32		SCSI Host Adapter (0x140-0x15F), bytes 1-16	
0x150-0x15F	SCSI Host Adapter (0x140-0x15F), bytes 17-32			
0x160-0x16F	System Use		Quaternary IDE Controller, master slave	
0x170-0x17F	Secondary IDE Controller, Master drive		System Use	
0x180-0x18F	System Use			
0x190-0x19F	System Use			
0x1A0-0x1AF	System Use			
0x1B0-0x1BF	System Use			
0x1C0-	System Use			

0x1CF			
0x1D0-0x1DF	System Use		
0x1E0-0x1EF	System Use	Tertiary IDE Controller, master slave	
0x1F0-0x1FF	Primary IDE Controller, master slave	System Use	
0x200-0x20F	Joystick Port	System Use	
0x210-0x21F	System Use		
0x220-0x22F	Sound Card		
	Non-NE2000 Network Card	System Use	
0x230-0x23F	SCSI Host Adapter (0x220-0x23F), bytes 17-32)		
0x240-0x24F	Sound Card		
	Non-NE2000 Network Card	System Use	
	NE2000 Network Card (0x240-0x25F) Bytes 1-16		
0x250-0x25F	NE2000 Network Card (0x240-0x25F) Bytes 17-32		
0x260-0x26F	Sound Card		
	Non-NE2000 Network Card	System Use	
	NE2000 Network Card (0x240-0x27F) Bytes 1-16		
0x270-0x27F	System Use	Plug and Play System Devices	LPT2 - Second Parallel Port
	System Use		LPT3 - Third Parallel Port (Monochrome Systems)
	NE2000 Network Card (0x260-0x27F) Bytes 17-32		
0x280-0x28F	Sound Card		
	Non NE2000 Network Card	System Use	
	NE2000 Network Card (0x280-0x29F) Bytes 1-16		
0x290-0x29F	NE2000 Network Card (0x280-0x29F) Bytes 17-32		
0x2A0-0x2AF	Non NE2000 Network Card	System Use	
	NE2000 Network Card (0x280-0x29F) Bytes 1-16		
0x2B0-0x2BF	NE2000 Network Card (0x280-0x29F) Bytes 17-32		
0x2C0-0x2CF	System Use		
0x2D0-0x2DF	System Use		
0x2E0-0x2EF	System Use	COM4 - Fourth Serial Port	
0x2F0-0x2FF	System Use	COM2 - Second Serial Port	
0x300-0x30F	Sound Card / MIDI Port	System Use	
	Non NE2000 Network Card	System Use	
	NE2000 Network Card (0x300-0x31F) Bytes 1-16		
0x310-0x31F	NE2000 Network Card (0x300-0x32F) Bytes 17-32		
0x320-0x32F	Sound Card / MIDI Port (0x330, 0x331)	System Use	
	NE2000 Network Card (0x300-0x31F) Bytes 17-32		
	SCSI Host Adapter (0x330-0x34F) Bytes 1-16		
0x330-0x33F	Sound Card / MIDI Port	System Use	
	Non NE2000 Network Card	System Use	
	NE2000 Network Card (0x300-0x31F) Bytes 1-16		
0x340-0x34F	SCSI Host Adapter (0x330-0x34F) Bytes 17-32		
	SCSI Host Adapter (0x340-0x35F) Bytes 1-16		
	Non NE2000 Network Card	System Use	
	NE2000 Network Card (0x340-0x35F) Bytes 1-16		
0x350-0x35F	SCSI Host Adapter (0x340-0x35F) Bytes 17-32		
	NE2000 Network Card (0x300-0x31F) Bytes 1-16		
0x360-0x36F	Tape Accelerator Card (0x360)	System Use	Quaternary IDE Controller (Slave Drive) (0x36E-0x36F)
	Non NE2000 Network Card	System Use	
	NE2000 Network Card (0x300-0x31F) Bytes 1-16		
0x370-	Tape Accelerator Card	Secondary IDE Controller (Slave	LPT1 - First Parallel Port (Color systems)

0x37F	(0x370)	Drive)		
	System Use		LPT2 - Second Parallel Port (Monochrome Systems)	
	NE2000 Network Card (0x360-0x37F) Bytes 1-16			
0x380-0x38F	System Use		Sound Card (FM Synthesizer)	System Use
0x390-0x39F	System Use			
0x3A0-0x3AF	System Use			
0x3B0-0x3BF	VGA/Monochrome Video			LPT1 - First Parallel Port (Monochrome Systems)
0x3C0-0x3CF	VGA/CGA Video			
0x3D0-0x3DF	VGA/CGA Video			
0x3E0-0x3EF	Tape Accelerator Card (0x370)	System Use	COM3 - Third Serial Port	
	System Use			Tertiary IDE Controller (Slave Drive) (0x3EE-0x3EF)
0x3F0-0x3FF	Floppy Disk Controller		COM1 - First Serial Port	
	Tape Accelerator Card (0x3F0)	Primary IDE Controller (Slave Drive) (0x3F6-0x3F7)	System Use	

This table is not complete, and hopefully has no errors in it. I will add to this table as time goes on, and more devices are developed.

All of these memory ranges are used by certain controllers--as the above table displays. The exact meaing of a port address depends on the controller. It could represent a control register, state registerm virtually anything. This is unfortanate.

I highley recommend to print out a copy of the above table. We will need to refrence it every time we are communicating with hardware.

I will update (at the beginning of the tutorial), if I have updated the table. That way, you can print out the table again, insuring everyone has the latest copy.

With all of this in mind, lets put it all together...

IN and OUT Instructions

The x86 processor has two instructions used for port I/O. They are **IN** and **OUT**.

These instructions tell the processor that we want to communicate to a device. This insures the processor sets the I/O DEVICE line on the control bus.

Lets have a complete example, and try to see if we can read from the keyboard controllers input buffer.

Lets see...Looking at our trusty Port table above, we can see the Keyboard Controller is in port addresses **0x60 through 0x6F**. The table displays that the first QWORD and second QWORD (Starting from port address 0x60) is for the keyboard and PS/2 Mouse. The last two QWORDS are for system use, so we will ignore it.

Okay, so our keyboard controller is mapped to ports 0x60 through, technically, port 0x68. This is cool, but what does it mean to us? This is device specific, remember?

For our keyboard, **Port 0x60 is a control register, Port 0x64 is a status register.** Remember from before--I said we would here these terms alot more, and in different contexts. **If bit 1 in the status register is set, data is inside the input buffer.** So, lessee... If we set the CONTROL register to READ, we can copy the contents of the input buffer somewhere...

```
WaitLoop:    in      al, 64h                    ; Get status register value
             and     al, 10b                    ; Test bit 1 of status register
             jz      WaitLoop                   ; If status register bit not set, no data is in buffer
             in      al, 60h                    ; Its set--Get the byte from the buffer (Port 0x60), and store it
```

This, right here, is the bases of hardware programming and device drivers.

In an **IN** instruction, the processor places the port address--like 0x64--into the Address Bus, and sets the I/O DEVICE line in the control bus, followed by the READ line. The device that has been assigned to 0x60 by the ROM BIOS-- In this case, the Status Register in the keyboard Controller, knows its a read operation because the READ line is set. So, it copies data from some location inside the keyboard registers onto the Data Bus, resets the READ and I/O DEVICE lines on the control bus, and sets the READY line. Now, the processor has the data from the Data Bus that was read.

An **OUT** instruction is simular. The processor copies the byte to be written into the Data Bus (Zero extending it to the Data Bus Width). Then, it sets the WRITE and I/O DEVICE lines on the control bus. It then copies the port address--lets say--0x60, into the Address Bus. **Because the I/O DEVICE Line is set, it is a signal that tells all controllers to watch the address bus. If the number on the address bus currosponds with there assigned number, the device acts on that data.** In our case--The Keyboard Controller. The Keyboard controller knows its a WRITE operation because the WRITE line is set in the control bus. So, it copies the value on the data bus into its Control Register--which was assigned port address 0x60. The Keyboard Controller the resets the WRITE and I/O DEVICE line, sets the READY line on the control bus, and the processor is back in control.

Port mapping and Port I/O are very important. It is our only way of cummunicating with hardware in protected mode. Remember: Interrupts are not avilable until we write them. To write them, along with any hardware routine--such as input and output, requires us to write drivers. All of this requires direct hardware access. If you dont feel confortable with this, practice a little first, and reread this section. If you have any questions, let me know.

The Processor

Special Instructions

Most 80x86 instructions can be executed by any program. However, there are some instructions that only Kernel-level software can access. Because of this, some of these instructions may not be familiar to our readers. We will require the use of most of these instructions, so understanding them is important.

Privileged Level (Ring 0) Instructions	
Instruction	Description
LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
LTR	Loads a Task Register into TR
MOV *Control Register*	Copy data and store in Control Registers
LMSW	Load a new Machine Status WORD
CLTS	Clear Task Switch Flag in Control Register CR0
MOV *Debug Register*	Copy data and store in debug registers
INVD	Invalidate Cache without writeback
INVLPG	Invalidate TLB Entry
WBINVD	Invalidate Cache with writeback
HLT	Halt Processor
RDMSR	Read Model Specific Registers (MSR)
WRMSR	Write Model Specific Registers (MSR)
RDPMC	Read Performance Monitoring Counter
RDTSC	Read time Stamp Counter

Executing any of the above instructions by any other program that does not have Kernel mode access (Ring 0) will generate a **General Protection Fault**, or a **Triple Fault**.

Do not worry if you do not understand these instructions. I will cover each of them throughout the series as we need them.

80x86 Registers

The x86 processor has alot of different **registers** for storing its current state. Most applications only have access to the **general, segment, and eflags**. Other registers are specific to Ring 0 programs, such as our Kernel.

The x86 family has the following registers: **RAX (EAX(AX/AH/AL)), RBX (EBX(BX/BH/BL)), RCX (ECX(CX/CH/CL)), RDX (EDX(DX/DH/DL)), CS,SS,ES,DS,FS,GS, RSI (ESI (SI)), RDI (EDI (DI)), RBP (EBP (BP)). RSP (ESP (SP)), RIP (EIP (IP)), RFLAGS (EFLAGS (FLAGS)), DR0, DR1, DR2, DR3, DR4, DR5, DR6, DR7, TR1, TR2, TR3, TR4, TR5, TR6, TR7, CR0, CR1, CR2, CR3, CR4, CR8, ST, mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7, xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7, GDTR, LDTR, IDTR, MSR, and TR**. All of these registers are stored in a special area of memory inside the processor called a **Register File**. Please see the **Processor Architecture** section for more information. Other registers include, but may not be in the **Resgister File**, include: **PC, IR, vector registers, and Hardware Registers.**

Alot of these registers are **only** avilable to real mode ring 0 programs. And for very good reasons, too. Most of these registers effect alot of states within the processor. Incorrectly setting them can easily triple fault the CPU. Other cases, might cause the CPU to malfunction. (Most notably, the use of TR4,TR5,TR6,TR7)

Some of the other registers are **internal to the CPU**, and **cannot** be accessed through normal means. One would need to reprogram the processor itself in order to access them. Most notably, IR, the vector registers.

We will need to know some of these special registers, so lets take a look closer, shall we?

Note: Think of the CPU as any normal device that we need to comunicate with. The concept of Control Registers (and registers themselves) will be important later on when we talk to other devices.

Also, please note that some registers are undocumented. Because of this, there may be more registers then those listed. If you know of any, please Let me know, so that I can add them. :)

General Purpose Registers

These are 32 bit registers that can be used for almost any purpose. Each of these registers have a special purpose as well, however.

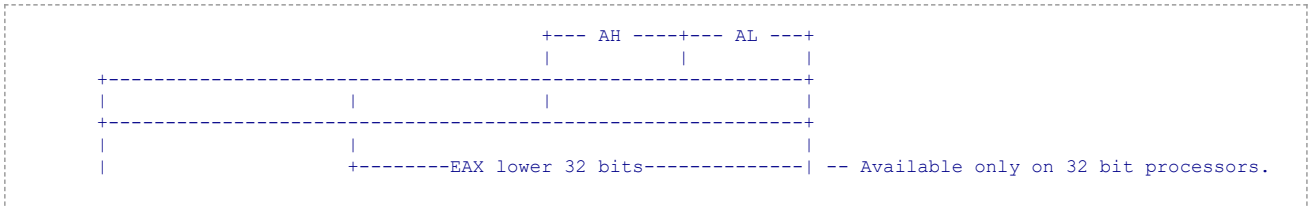
- EAX - Accumlator Register. Primary purpose: Math calculations
- EBX - Base Address Register. Primary purpose: Indirectly access memory through a base address.
- ECX - Counter Register. Primary purpose: Use in counting and looping.
- EDX - Data Register. Primary purpose: um... store data. Yep, thats about it :)

Each of these 32 bit registers has two parts. The **High order word** and **low order word**. The high order word is th upper 16 bits. **The low order word is the lower 16 bits.**

On 64 bit processors, these registers are 64 bits wide, and or named **RAX, RBX, RCX, RDX**. The lower 32 bits is the 32 bit **EAX** register.

The upper 16 bits does not have a special name associatd with them. However, **The lower 16 bits do. These names have an oppended 'H' (for higher 8 bits in low word), or an appended 'L' for lower 8 bits.**

For example, in RAX, we have:



```
        |                                                       |
        |------------------ RAX Complete 64 bits----------------| -- Available only on 64 bit processors.
```

What does this mean? **AH and AL are a part of AX, which, in turn, is a part of EAX. Thus, modifying any of these names effectivly modifies the same register - EAX.**

This, in turns, modifies RAX, on 64 bit machines.

The above is also true with BX, CX, and DX.

General purpose registers can be used within any program, from Ring 0 to Ring 4. Because they are basic assembly language, I will assume you already know how they work.

Segment Registers

The segment registers modify the current segment addresses in real mode. They are all 16 bit.

- CS - Segment address of code segment
- DS - Segment address of data segment
- ES - Segment address of extra segment
- SS - Segment address of stack segment
- FS - Far Segment address
- GS - General Purpose Register

Remember: **Real Mode uses the segment:offset memory addressing model.** The **segment address** is stored within a segment register. Another register, such as **BP, SP, or BX** can store the **offset** address.

It is useually refrenced like: **DS:SI**, where DS contains the segment address, and SI contains the offset address.

Segment registers can be used within any program, from Ring 0 to Ring 4. Because they are basic assembly language, I will assume you already know how they work.

Index Registers

The x86 uses several registers that help when access memory.

- SI - Source Index
- DI - Destination Index
- BP - Base Pointer
- SP - Stack Pointer

Each of these registers store a 16 bit base address (that may be used as an offset address as well.)

On 32 bit processors, these registers are **32 bits** and have the names **ESI, EDI, EBP, and ESP**.

On 64 bit processors, each register is 64 bits in size, and have the names **RSI, RDI, RBP, and RSP**.

The 16 bit registers are a subset of the 32 bit registers, which is a subset of the 64 bit registers; the same way with RAX.

The **Stack Pointer** is automatically incremented and decremented a certain amount of bytes whenever certain instructions are encountered. Such instructions include **push*, pop* instructions, ret/iret, call, syscall** etc.

The C Programming Language, in fact most languages, use the stack regularly. We will need to insure we set the stack up at a good address to insure C works properly. Also, remember: The stack grows *downward*!

Instruction Pointer / Program Counter

The Instruction Pointer (IP) register stores the current offset address of the currently exectuting instruction. Remember: **This is an offset address, *Not* an absolute address!**

The Instruction Pointer (IP) is sometimes also called the Program Counter (PC).

On 32 bit machines, IP is 32 bits in size and uses the name **EIP**.

On 64 bit machines, IP is 64 bits in size, and uses the name **RIP**.

Instruction Register

This is an internal processor register that cannot be accessed through normal means. It is stored within the **Control Unit (CU)** of the processor inside the **Instruction Cache**. It stores the current instruction that is being translated to **Microinstructions** for use internally by the processor. Please see **Processor Architecture** for more information.

EFlags Register

The EFLAGS register is the x86 Processor Status Register. It is used to determin the um.. current status. We have actually used this alot already so far. A simple example: **jc, jnc, jb, jnb Instruction**

Most instructions manupulate the EFLAGS register so that you can test for conditions (Like if the value was lower or higher then another).

EFLAGS is composed of the **FLAGS** register. Simularity, **RFLAGS** is composed of **EFLAGS** and **FLAGS**. ie:

```
 +---------- EFLAGS (32 Bits) -----+
 |                                 |
 |-- FLAGS (16 bits)-              |
 |                  |              |
 ===================================================================  < Register Bits
 |                                                                 |
 +------------------------ RFLAGS (64 Bits) ----------------------+
 |                                                                 |
Bit 0                                                            Bit 63
```

FLAGS Register Status Bits		
Bit Number	Abbrivation	Description
0	CF	Carry Flag - Status bit
1		Reserved
PF	CF	Parity Flag
3		Reserved
4	AF	Adjust Flag - Status bit
5		Reserved
6	ZF	Zero Flag - Status bit
7	SF	Sign Flag - Status bit
8	TF	Trap Flag (Single Step) - System Flag
9	IF	Interrupt Enabled Flag - System Flag
10	DF	Direction Flag - Control Flag
11	OF	Overflow Flag - Status bit
12-13	IOPL	I/O Priviledge Level (286+ Only) - Control Flag
14	NT	Nested Task Flag (286+ Only) - Control Flag
15		Reserved
EFLAGS Register Status Bits		
Bit Number	Abbrivation	Description
16	RF	Resume Flag (386+ Only) - Control Flag
17	VM	v8086 Mode Flag (386+ Only) - Control Flag
18	AC	Alignment Check (486SX+ Only) - Control Flag
19	VIF	Virtual Interrupt Flag (Pentium+ Only) - Control Flag
20	VIP	Virtual Interrupt Pending (Pentium+ Only) - Control Flag
21	ID	Identification (Pentium+ Only) - Control Flag
22-31		Reserved
RFLAGS Register Status Bits		
Bit Number	Abbrivation	Description
32-63		Reserved

The **IO Privilege Level (IOPL)** controls the current ring level required to use certain instructions. For example, the **CLI, STI, IN and OUT** instructions will only execute if the current Privilege Level is equal, or greater, then the IOPL. If not, a **General Protection Fault (GPF)** will be generated by the processor.

Most operating systems set the IOPF to 0 or 1. This means that only Kernel level software can use these instructions. This is a very good thing. After all, if an application issues a CLI, it can effectivly stop the Kernel from running.

For most operations, we only need to use the FLAGS register. Notice that the last 32 bits of the RFLAGS register is nill, null, non existant, there for viewing pleasure. So, um... yeah. For speed puposes, of course, but alot of bytes being wasted... ...yeah.

Because of the size of this table, I recommend printing it out for refrence later.

Test Registers

The x86 family uses some registers for testing purposes. Many of these registers are undocumented. On the x86 series, these registers are **TR4,TR5,TR6,TR7**.

TR6 is the most commonly used for command testing, and **TR7** for a test data register. One can use the **MOV** instruction to access them. **There are only avilable in Ring 0 for both pmode and real mode. Any other attempt will cause a General Protection Fault (GPF) leading to a Triple Fault.**

Debug Registers

These registers are used for program debugging. They are **DR0,DR1,DR2,DR3,DR4,DR5,DR6,DR7**. Just like the test registers, they can be accessed using the **MOV** instruction, and only in Ring 0. **Any other attempt will cause a General Protection Fault (GPF) leading to a Triple Fault.**

Beakpoint Registers

The registers **DR0, DR1, DR2, DR3** store an **Absolute Address** of a breakpoint condition. If **Paging** is enabled, the address will be converted to its Absolute address. These breakpoint conditions are further defined in DR7.

Debug Control Register

DR7 is a 32 bit register that uses a bit pattern to identify the current debugging task. Here it is:

- **Bit 0...7** - Enable the four debug registers (See below)
- **Bit 8...14** - ?
- **Bit 15...23** - When the breakpoints will trigger. Each 2 bits represents a single Debug Register. This can be one of the following:
  - **00** - Break on execution
  - **01** - Break on data write
  - **10** - Break on IO read or write. No hardware currently supports this.
  - **11** - Break on data read or write
- **Bit 24...31** - Defines how large of memory to watch. Each 2 bits represents a single Debug Register. This can be one of the following:
  - **00** - One bytes
  - **01** - Two bytes
  - **10** - Eight bytes
  - **11** - Four bytes

The Debug Registers uses two methods to enable. This is **Local** and **Global** levels. If you are using different **Tasks** (such as in **Paging**, all **Local debug changes only effect that task.** The processor automatically clears all local changes when switching between tasks. Global tasks, however, are not.

In Bits 0...7 in the above list:

- **Bit 0:** Enable local DR0 register
- **Bit 1:** Enable global DR0 register
- **Bit 2:** Enable local DR1 register
- **Bit 3:** Enable global DR1 register
- **Bit 4:** Enable local DR2 register
- **Bit 5:** Enable global DR2 register
- **Bit 6:** Enable local DR3 register
- **Bit 7:** Enable global DR3 register

Debug Status Register

This is used by debuggers to determin what happened when the error accured. **When the processor runs into an enabled exception error, it sets the low 4 bits of this register and executes the Exception Handler.**

Warning: The debug status regster, DR6, is never cleared. If you have the program continue, insure you clear this register!

Model Specific Register

This is a special control register that provides special processor specific features that may not be on others. As these are system level, Only Ring 0 programs can access this register.

Because these registers are specific to each processor, the actual register may change.

The x86 has two special instructions that are used to access this register:

- **RDMSR** - Read from MSR
- **WRMSR** - Write from MSR

The registers are very processor specific. Because of this, it is wise to use the CPUID instruction before using them.

To access a given register, one must pass the instructions an **Address** which represents the register you want access to.

Through the years, Intel has used some MSRs that are not machine specific. These MSRs are common within the x86 architecture.

Model Specific Registers (MSRs)		
Register Address	Register Name	IA-32 Processor Family
0x0	IA32_PS_MC_ADDR	Pentium Processors
0x1	IA32_PS_MC_TYPE	Pentium 4 Processors
0x6	IA32_PS_MONITOR_FILTER_SIZE	Pentium Processors
0x10	IA32_TIME_STAMP_COUNTER	Pentium Processors
0x17	IA32_PLATFORM_ID	P6 Processors
0x1B	IA32_APIC_BASE	P6 Processors
0x3A	IA32_FEATURE_CONTROL	Pentium 4 / Processor 673
0x79	IA32_BIOS_UPDT_TRIG	P6 Processors
0x8B	IA32_BIOS_SIGN_ID	P6 Processors
0x9B	IA32_SMM_MONITOR_CTL	Pentium 4 / Processor 672
0xC1	IA32_PMC0	Intel Core Duo
0xC2	IA32_PMC1	Intel Core Duo
0xE7	IA32_MPERF	Intel Core Duo
0xE8	IA32_APERF	Intel Core Duo
0xFE	IA32_MTRRCAP	P6 Processors
0x174	IA32_SYSENTER_CS	P6 Processors
0x175	IA32_SYSENTER_ESP	P6 Processors
0x176	IA32_SYSENTER_IP	P6 Processors

There are alot more MSR's then those that are listed. Please see Appendex B from the Intel Development Manuals for the complete list.

Im not sure what MSRs we will be refrencing as the Series is still in development. I will add onto this list as needed.

RDMSR Instruction

This instruction loads the MSR specified by CX into EDX:EAX.

This instruction is a **priviledged** instruction, and can only be executed at **Ring 0 (Kernel Level).** A **General Protection Fault,** or **Triple Fault** will accure if a nonprivelged application attempts to execute this instruction, or the value in CS does not represent a valid MSR Address.

This instruction does not effect any flags.

Here is an exaple of using this instruction (You will see this again later in the tutorials):

```
; This reads from the IA32_SYSENTER_CS MSR

mov     cx, 0x174                       ; Register 0x174: IA32_SYSENTER_CS
rdmsr                                   ; Read in the MSR

; Now EDX:EAX contains the lower and upper 32 bits of the 64 bit register
```

Cool, huh?

WRMSR Instruction

This instruction writes to the MSR specified by CX the 64 bit value stored in EDX:EAX.

This instruction is a **priviledged** instruction, and can only be executed at **Ring 0 (Kernel Level).** A **General Protection Fault,** or **Triple Fault** will accure if a nonprivelged application attempts to execute this instruction, or the value in CS does not represent a valid MSR Address.

This instruction does not effect any flags.

Here is an example of how it is used:

```
; This writes to the IA32_SYSENTER_CS MSR

mov     cx, 0x174                       ; Register 0x174: IA32_SYSENTER_CS
wrmsr                                   ; Write EDX:EAX into the MSR
```

Control Registers

THIS is going to be important to us.

The control registers allow us to change the behavior of the processor. They are: **CR0, CR1, CR2, CR3, CR4**.

CR0 Control Register

CR0 is the primary control register. It is 32 bits, which are defined as follows:

- **Bit 0 (PE) : Puts the system into protected mode**
- **Bit 1 (MP) : Monitor Coprocessor Flag** This controls the operation of the **WAIT** instruction.
- **Bit 2 (EM) : Emulate Flag**. When set, **coprocessor instructions will generate an exception**
- **Bit 3 (TS) : Task Switched Flag** This will be set when the processor switches to another **task**.
- **Bit 4 (ET) : ExtensionType Flag. This tells us what type of coprocesor is installed.**
  - 0 - 80287 is installed
  - 1 - 80387 is installed.
- **Bit 5 (NE):** Numeric Error
  - 0 - Enable standard error reporting
  - 1 - Enable internal x87 FPU error reporting
- **Bits 6-15 :** Unused
- **Bit 16 (WP):** Write Protect
- **Bit 17:** Unused
- **Bit 18 (AM):** Alignment Mask
  - 0 - Alignment Check Disable
  - 1 - Alignment Check Enabled (Also requires AC flag set in EFLAGS and ring 3)
- **Bits 19-28:** Unused
- **Bit 29 (NW):** Not Write-Through
- **Bit 30 (CD):** Cache Disable
- **Bit 31 (PG) : Enables Memory Paging.**
  - 0 - Disable
  - 1 - Enabled and use CR3 register

Wow... alot of new stuff, huh? Lets look at Bit 0--**Puts system in protected mode.** This means, **By setting Bit 0 in the CR0 register, we effectivly enter protected mode.**

For example:

```
            mov     ax, cr0                 ; get value in CR0
            or      ax, 1                   ; set bit 0--enter protected mode
            mov     cr0, ax                 ; Bit 0 now set, we are in 32 bit mode!
```

Wow! Its that easy!! Not quite :)

If you dump this code into our bootloader, its almost guaranteed to Triple Fault. Protected mode uses a different memory addressing system then real mode. Also, **Remember that pmode has no interrupts**. A single timer interrupt will triple fault. Also, because we use a different addressing model, **CS is invalid**. We would need to update CS to go into 32 bit code. And yet, **We didnt set privedlge levels for the memory map.**

We will go more into detail later.

CR1 Control Register

Reserved by Intel, do not use.

CR2 Control Register

Page Fault Linear Address. If a Page Fault Exception accures, CR2 contains the address that was attempted accessed

CR3 Control Register

Used when the PG bit in CR0 is set. Last 20 bits Contains the Page Directory Base Register (PDBR)

CR4 Control Register

Used in protected mode to control operations, such as v8086 mode, enabling I/O breakpoints, Page size extension and machine check exceptions.

I don't know if we will use any of these flags or not. I decided to include it here for completness sake. Don't worry to much if you don't understand what these are.

- **Bit 0 (VME) :** Enables Virtual 8086 Mode Extensions

- **Bit 1 (PVI) :** Enables Protected Mode Virtual Interrupts
- **Bit 2 (TSD) :** Time Stamp Enable
  - 0 - RDTSC instruction can be used in any privilege level
  - 1 - RDTSC instruction can only be used in ring 0
- **Bit 3 (DE) :** Enable Debugging Extensions
- **Bit 4 (PSE) :** Page Size Extension
  - 0 - Page size is 4KB
  - 1 - Page size is 4MB. With PAE, it is 2MB.
- **Bit 5 (PAE) :** Physical Address Extension
- **Bits 6 (MCE) :** Machine Check Exception
- **Bits 7 (PGE) :** Page Global Enable
- **Bits 8 (PCE) :** Performance Monitoring Counter Enable
  - 0 - RDPMC instruction can be used in any privilege level
  - 1 - RDPMC instruction can only be used in ring 0
- **Bits 9 (OSFXSR) :** OS Support for FXSAVE and FXSTOR instructions (SSE)
- **Bits 10 (OSXMMEXCPT) :** OS Support for unmasked SIMD FPU exceptions
- **Bits 11-12 :** Unused
- **Bits 13 (VMXE) :** VMX Enable

CR8 Control Register

Provides Read and Write access to the **Task Prority Register (TPR)**

PMode Segmentation Registers

The x86 family uses several registers to store the current linear address of each **Segment Descriptor**. More on this later.

These registers are:

- **GDTR -** Global Descriptor Table Register
- **IDTR -** Interrupt Descriptor Table Register
- **GDTR -** Local Descriptor Table Register
- **TR -** Task Register

We will take a look closer at these registers in the next section.

Processor Architecture

Throughout this series, you will notice alot of simularities between the processor and microcontrollers. That is, Microcontrollers have registers, and execute instructions simular to the processor. The CPU itself **is nothing more then a specialized controller chip**.

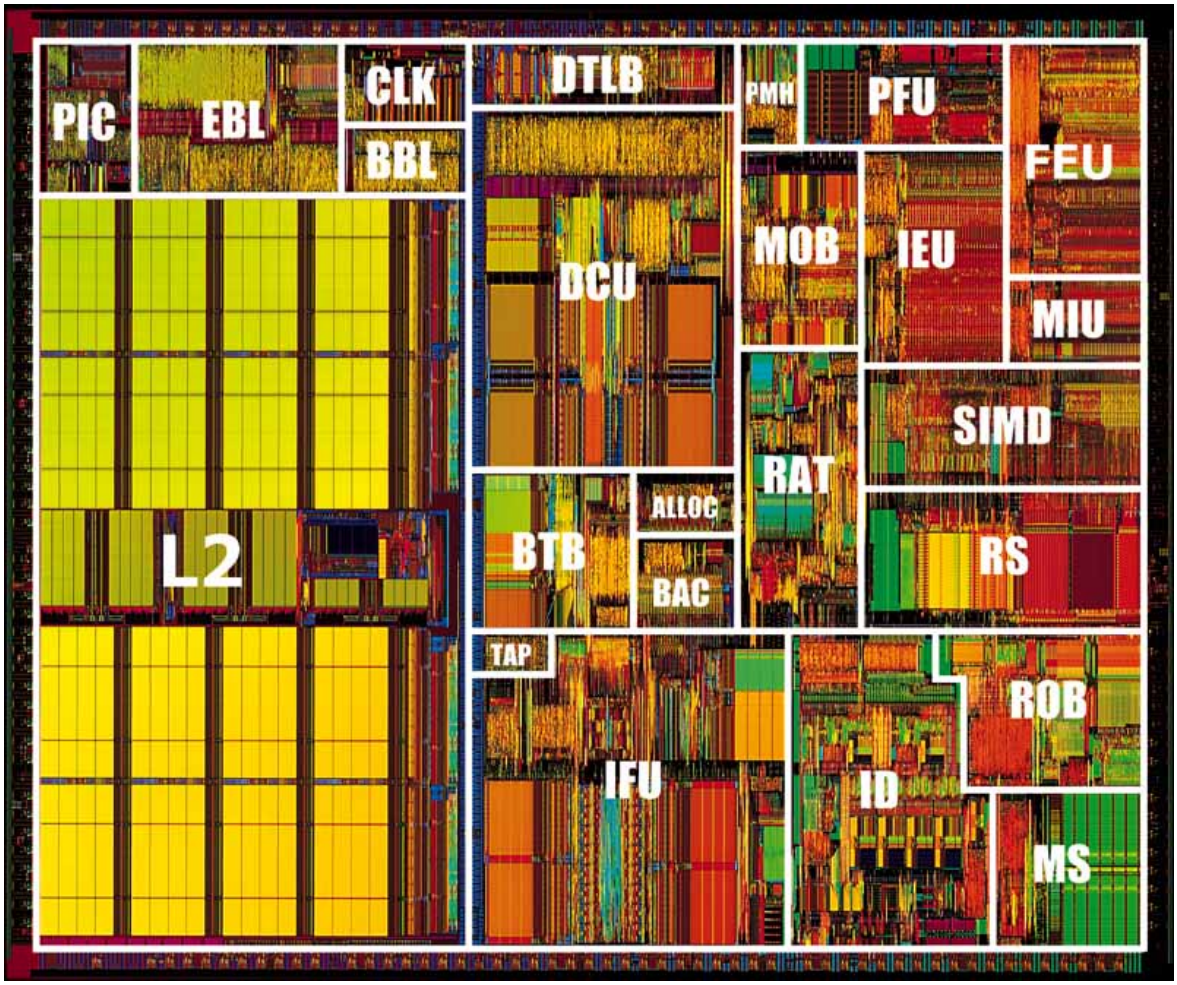
We will look at the boot process again a little later, but from a very low level perspective. This will answer alot of questions reguarding how the BIOS POST actually starts, and how it executes the POST, starts the primary processor, and load the BIOS. We have covered the **what**, but we have not covered the **how** yet.

Note: This section is fairly technical. If you do not understand everything, do not worry, as we do not need to understand everything. I am including this section for completness sake; to dive into the main component required in any computer system, and the one that is responsible for executing our code. How does it execute our given code? What is so special about machine language? All of this will be answered here.

Later on when we dive into the **Kernel** and **Device Driver** development, you will learn that understanding the basic hardware controller components themselves can not only be a great learning expierence, but sometimes a **neccessity** in understanding how to program that controller.

Breaking apart a processor

We will be looking at the Pentium III processor here for explination purposes. Lets open up and disect this processor into it's individuale components first:



Alot of things in the processor, huh? Notice how complex this is. We are not going to learn much from this picture alone, so lets look at each component.

- **L2:** Level 2 Cache
- **CLK:** Clock
- **PIC: Programmable Interrupt Controller**
- **EBL:** Front Bus Logic
- **BBL:** Back Bus Logic
- **IEU:** Integer Execution Unit
- **FEU:** Floating Point Execution Unit
- **MOB:** Memory Order Buffer
- **MIU / MMU:** Memory Interface Unit / Memory Management Unit
- **DCU:** Data Cach Unit
- **IFU:** Instruction Fetch Unit
- **ID:** Instruction Decoder
- **ROB:** Re-Order Buffer
- **MS: Microinstruction** Sequencer
- **BTB:** Branch Target Buffer
- **BAC:** Branch Allocater Buffer
- **RAT:** Register Alias Table
- **SIMD:** Packed floating point
- **DTLB:** Data TLB
- **RS:** Reservation Station
- **PMH:** Page Miss Handler
- **PFU:** Pre-Fetch Unit
- **TAP:** Test Access Port

I plan on adding to this section.

How instructions execute

Okay, **Remember that the IP Register contains the offset address of the currently exectuting instruction. CS contains the segment address.**

Okay, so what exactally happens when the processor needs to execute an instruction?

It first calculates the absolute address that it needs to read from. **Remember that in the segment:offset model, the absolute address = segment * 16 + offset**. Or, essentally, **absolute address = CS * 16 + IP**.

The processor copies this address into the Address Bus. **Remember that the address bytes is just a series of electronic lines, each representing a single bit. This bit pattern represents the binary form of the absolute address of the next instruction.**

After this, the processor enables the "Read Memory" line (By setting its bit to 1). This tells the **Memory Controller** that we need to read from memory.

The **Memory Controller** takes control. The Memory Controller copies the address from the Address Bus, anc calculates its exact location in a paricular RAM chip. The Memory Controller refrences this location, and copies it into the Data Bus. It does this because the "Read Memory" line is

set on the Control Bus.

The Memory Controller resets the Control Bus, so the processor knows its done executing. The processor takes the value(s) within the Data Bus, and uses its Digital Logic gates to "execute" it. This "value" is just a binary representation of a machine instruction, encoded as a series of electronic pulses.

Lets say, the instruction is **mov ax, 0x4c00** The value **0xB8004C** will be on the data bus for the processor. 0xB8004C is what is known as an **operation code (OPCode)**. Every instruction has an opcode associated with it. For the i86 architecture, our instruction would evaluate to the opcode 0xB8004C. We can convert this number to binary and can see the pattern as electronic lines, where a 1 means the line is high (active), and a 0 means the line is low:

```
101110000000000001001100
```

The processor follows a series of descrete instructions hard built into the digital logic circuit of the CPU. These instructions tells the processor how to encode a series of bits. All x86 processors will encode this bit pattern as our **mov ax, 0x4c00** instruction.

Do to the increasing complexity of instructions, most newer processors actually follow their own internal instruction sets. This is not new to processors--alot of microcontrollers may use multiple internal instruction sets to decrease the complexity of the electronics. Normally these are **Macrocode** and **Microcode**.

Macrocode is an abstract set of instructions the processor uses to decode an instruction into microcode. Macrocode is normally written in a special macro language developed by the electronic engineers stored on a ROM chip inside of the controller and compiled in a macro assembler. The macro assembler assembles the macrocode into an even lower level language, which is the language of the controller: Microcode.

Microcode is a very low level language developed by the electronic engineers. Microcode is used by the controller or processor to decode an instruction...such as our 0xB8004C (mov ax, 0x4c00) instruction.

Using its **Arithmitic Logic Unit (ALU)** the CPU can get a number - 0x4C00. And copy it into AX (A simple bit copy).

This example demonstrates how everything comes together. How the CPU uses the system bus, **and relys on the memory controller to decode the memory location, and follow the Control Bus**.

This is an important concept. Software Ports rely on the Memory Controller in a simular fashion.

Protected Mode - Theory

Okay...So why did we talk about architecture? The unfortanate truth is that in protected mode, there is no interrupts. So lessee... No interrupts. No System calls. No Standard Library. No nothing. Everything we do, we have to do ourselves. Because of this, there is no helping hand to guide us. One mistake can either crash your system, or even destroy your hardware if you are not carefull. Not only floppy disks, but hard disks, external (and internal) devices, etc.

Understanding the system architecture will help us understand everything alot better, to insure we dont make too many mistakes. Also, it gives us an introduction to direct hardware programming-- because this is the only thing we can do.

You might be thinking: Wait--What about the uber-1337 C Kernel you promised us!? Well...Remember that C is a low level language, in a way. Through inline assembly, we could create an interface to the hardware. And C, just like C++, only produces x86 machine instructions that could be executed directly by the processor. **Just remember, however, that there is no standard library. And, that you are programming in a very low level envirement, even if you are using a high level language.**

We will go over this when we start the Kernel.

Conclusion

I never like writing these types of tutorials. They are packed with huge amounts of information, little code, displaying concepts in descrete detail for better understanding. They are simply hard to write, you know?

I hope I explained everything well enough. We want over *alot*-Memory Mapping, Port Mapping, the x86 Port Addresses, All x86 registers, The x86 Memory Map, System Architecture, IN/OUT keywords and how they execyte, and how instructions execute-step by step. We also have taken a look at basic hardware programming--in which we will be doing alot.

In the next tutorial, we are going to make the switch--**Welcome to the 32 bit world!** We are also going to be looking at the **GDT** in detail--as we will need that to make the switch. I am also going to give warnings for common errors every step of the way. As I said before, a small itty-bitty mistake when entering protected mode **will** crash your program.

Its going to be fun... :)

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 6 Home Chapter 8 

Operating Systems Development Series

Operating Systems Development - Protected Mode

by Mike, 2008

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

Introduction

Welcome! :)

We covered alot so far throughout this series. We looked at bootloaders, system architecture, file systems, and real mode addressing in depth. This is cool--but we have yet to look at the 32 bit world. And, are we not building a 32bit OS?

In this tutorial, we are going to make the jump--**Welcome to the 32 bit world!** Granted, we are not done with the 16 bit world just yet, however it will be much easier to get entering protected mode done now.

So, lets get started then! This tutorial covers:

- Protected Mode Theory
- Protected Mode Addressing
- Entering Protected Mode
- Global Descriptor Table (GDT)

Ready?

stdio.inc

To make things more object oriented, I have moved all input/output routines into a stdio.inc file. Please, Do not associate this with the C standard stdio.lib library. They have almost nothing in common. We will start work on the standard library while working on the Kernel.

Anyways...Heres the file:

```
;*************************************************
;	stdio.inc
;		-Input/Output routines
;
;	OS Development Series
;*************************************************

%ifndef __STDIO_INC_67343546FDCC56AAB872_INCLUDED__
%define __STDIO_INC_67343546FDCC56AAB872_INCLUDED__

;*************************************************;
;	Puts16 ()
;		-Prints a null terminated string
;	DS=>SI: 0 terminated string
;*************************************************;

bits	16

Puts16:
		pusha				; save registers
.Loop1:
		lodsb				; load next byte from string from SI to AL
		or	al, al			; Does AL=0?
		jz	Puts16Done		; Yep, null terminator found-bail out
		mov	ah, 0eh			; Nope-Print the character
		int	10h			; invoke BIOS
		jmp	.Loop1			; Repeat until null terminator found
Puts16Done:
		popa				; restore registers
		ret				; we are done, so return

%endif ;__STDIO_INC_67343546FDCC56AAB872_INCLUDED__
```

For those who do not know--*.INC files are **Include** files. We will add more to this file as needed. I'm not going to explain the **puts16** function--It's the exact same routine we used in the bootloader, just with an added pusha/popa.

Welcome to Stage 2

The bootloader is small. Way to small to do anything usefully. Remember that the bootloader is limited to 512 bytes. No more, No less. Seriously--our code to load Stage 2 was almost 512 bytes already! Its simply way to small.

This is why we want the bootloader to *just* load another program. Because of the FAT12 file system, our second program can be of almost any amount of sectors. Because of this, there is no 512 byte limitation. This is great for us. This, our readers, is Stage 2.

The Stage 2 bootloader will set everything up for the kernel. This is similar to **NTLDR** (NT Loader), in Windows. In fact, I am naming the program **KRNLDR** (Kernel Loader). Stage 2 will be responsible for loading our kernel, hence KRNLDR.SYS.

KRNLDR -- Our Stage 2 bootloader, will do several things. It can:

- **Enable and go into protected mode**
- **Retrieve BIOS information**
- **Load and execute the kernel**
- Provide advance boot options (Such as Safe Mode, for example)
- Through configuration files, you can have KRNLDR boot from multiple operating system kernels
- **Enable the 20th address line for access up to 4 GB of memory**
- **Provide basic interrupt handling**

...And more. It also sets up the environment for a high level language, like C. In fact, alot of times the Stage 2 loader is a mixture of C and x86 assembly.

As you can imagine--Writing the stage 2 bootloader can be a large project itself. And yet, its nearly impossible to develop an advanced bootloader without an already working Kernel. Because of this, we are only going to worry about the important details--**bolded** above. When we get a working kernel, we may come back to the bootloader.

We are going to look at entering protected mode first. I'm sure alot of you are itching to get into the 32 bit world--I know I am!

World of Protected Mode

Yippie!! Its finally time! You have heard me say "protected mode" alot--and we have described it in some detail before. As you know, protected mode is supposed to offer memory protection. By defining how the memory is used, we can insure certain memory locations cannot be modified, or executed as code. The 80x86 processor maps the memory regions based off the **Global Descriptor Table (GDT).**. **The processor will generate a General Protection Fault (GPF) exception if you do not follow the GDT. Because we have not set up interrupt handlers, this will result in a triple fault.**

Lets take a closer look, shall we?

Descriptor Tables

A Descriptor Table defines or map something--in our case, memory, and how the memory is used. There are three types of descriptor tables: **Global Descriptor Table (GDT), Local Descriptor Table (LDT), and Interrupt Descriptor Table (IDT); each base address is stored in the GDTR, LDTR, and IDTR x86 processor registers.** Because they use special registers, they require special instructions. **Note: Some of these instructions are specific to Ring 0 kernel level programs. If a general Ring 3 program attempts to use them, a General Protection Fault (GPF) exception will occur.** In our case, because we are not handling interrupts yet, a triple fault will occur.

Global Descriptor Table

THIS will be important to us--and you will see it both in the bootloader and Kernel.

The Global Descriptor Table (GDT) defines the global memory map. It defines what memory can be executed (The **Code Descriptor**), and what area contains data (**Data Descriptor**).

Remember that a descriptor defines properties--i.e., it describes something. In the case of the GDT, it describes starting and base addresses, segment limits, and even virtual memory. This will be more clear when we see it all in action, don't worry :)

The GDT usually has three descriptors--a **Null descriptor** (Contains all zeros), a **Code Descriptor**, and a **Data Descriptor**.

Okay.....So, what is a "Descriptor"? For the GDT, a "Descriptor" is an 8 byte QWORD value that describes properties for the descriptor. They are of the format:

- **Bits 56-63:** Bits 24-32 of the base address
- **Bit 55:** Granularity
  - **0:** None
  - **1:** Limit gets multiplied by 4K
- **Bit 54:** Segment type
  - **0:** 16 bit
  - **1:** 32 bit
- **Bit 53:** Reserved-Should be zero
- **Bits 52:** Reserved for OS use
- **Bits 48-51:** Bits 16-19 of the segment limit
- **Bit 47** Segment is in memory (Used with Virtual Memory)
- **Bits 45-46:** Descriptor Privilege Level
  - **0:** (Ring 0) Highest
  - **3:** (Ring 3) Lowest
- **Bit 44:** Descriptor Bit
  - **0:** System Descriptor
  - **1:** Code or Data Descriptor
- **Bits 41-43:** Descriptor Type
  - **Bit 43:** Executable segment
    - **0:** Data Segment
    - **1:** Code Segment
  - **Bit 42:** Expansion direction (Data segments), conforming (Code Segments)
  - **Bit 41:** Readable and Writable
    - **0:** Read only (Data Segments); Execute only (Code Segments)
    - **1:** Read and write (Data Segments); Read and Execute (Code Segments)
- **Bit 40:** Access bit (Used with Virtual Memory)
- **Bits 16-39:** Bits 0-23 of the Base Address
- **Bits 0-15:** Bits 0-15 of the Segment Limit

...Pretty ugly, huh? Basically, by building up a bit pattern, the 8 byte bit pattern will describe various properties of the descriptor. Each descriptor defines properties for its memory segment.

To make things simple, lets build a table that defines a code and data descriptors with read and write permissions from the first byte to byte 0xFFFFFFFF in memory. This just means we could read or write any location in memory.

We are first going to look at a GDT:

```
  ; This is the beginning of the GDT. Because of this, its offset is 0.

; null descriptor
       dd 0                            ; null descriptor--just fill 8 bytes with zero
       dd 0

; Notice that each descriptor is exactally 8 bytes in size. THIS IS IMPORTANT.
; Because of this, the code descriptor has offset 0x8.

; code descriptor:                      ; code descriptor. Right after null descriptor
       dw 0FFFFh                       ; limit low
       dw 0                            ; base low
       db 0                            ; base middle
       db 10011010b                    ; access
       db 11001111b                    ; granularity
       db 0                            ; base high

; Because each descriptor is 8 bytes in size, the Data descritpor is at offset 0x10 from
; the beginning of the GDT, or 16 (decimal) bytes from start.

; data descriptor:                      ; data descriptor
       dw 0FFFFh                       ; limit low (Same as code)
       dw 0                            ; base low
       db 0                            ; base middle
       db 10010010b                    ; access
       db 11001111b                    ; granularity
       db 0                            ; base high
```

That's it. The infamous GDT. This GDT contains three descriptors--each 8 bytes in size. A null descriptor, code, and data descriptors. **Each bit in each descriptor corresponds directly with that represented in the above bit table (Shown above the code).**

Lets break each down into its bits to see what's going on. The null descriptor is all zeros, so we will focus on the other two.

Breaking the code selector down

Lets look at it again:

```
; code descriptor:                      ; code descriptor. Right after null descriptor
       dw 0FFFFh                       ; limit low
       dw 0                            ; base low
       db 0                            ; base middle
       db 10011010b                    ; access
       db 11001111b                    ; granularity
       db 0                            ; base high
```

Remember that, in assembly language, each declared byte, word, dword, qword, instruction, whatever is literally right after each other. In the above, 0xffff is, of course, two bytes filled with ones. We can easily break this up into its binary form because most of it is already done:

```
11111111 11111111 00000000 00000000 00000000 10011010 11001111 00000000
```

Remember (From the above bit table), that **Bits 0-15 (The first two bytes)** represents the segment limit. This just means, we cannot use an address greater then 0xffff (Which is in the first 2 bytes) within a segment. Doing so will cause a GPF.

Bits 16-39 (The next three bytes) represent Bits 0-23 of the Base Address (The starting address of the segment). In our case, its 0x0. **Because the base address is 0x0, and the limit address is 0xFFFF, the code selector can access every byte from 0x0 through 0xFFFF**. Cool?

The next byte (Byte 6) is where the interesting stuff happens. Lets break it bit by bit--literally:

```
db 10011010b                    ; access
```

- **Bit 0 (Bit 40 in GDT):** Access bit (Used with Virtual Memory). Because we don't use virtual memory (Yet, anyway), we will ignore it. Hence, it is 0
- **Bit 1 (Bit 41 in GDT):** is the readable/writable bit. Its set (for code selector), so we can read and execute data in the segment (From 0x0 through 0xFFFF) as code
- **Bit 2 (Bit 42 in GDT):** is the "expansion direction" bit. We will look more at this later. For now, ignore it.
- **Bit 3 (Bit 43 in GDT):** tells the processor this is a code or data descriptor. (It is set, so we have a code descriptor)
- **Bit 4 (Bit 44 in GDT):** Represents this as a "system" or "code/data" descriptor. This is a code selector, so the bit is set to 1.
- **Bits 5-6 (Bits 45-46 in GDT):** is the privilege level (i.e., Ring 0 or Ring 3). We are in ring 0, so both bits are 0.
- **Bit 7 (Bit 47 in GDT):** Used to indicate the segment is in memory (Used with virtual memory). Set to zero for now, since we are not using virtual memory yet

  The access byte is very important! We will need to define different descriptors in order to execute Ring 3 applications and software. We will look at this alot more closer when we start getting into the Kernel.

  Putting this together, this byte indicates: **This is a readable and writable segment, we are a code descriptor, at Ring 0.**

Lets look at the next bytes:

```
db 11001111b                    ; granularity
        db 0                    ; base high
```

Looking at the granularity byte, lets break it down. Remember to use the GDT bit table above:

- **Bit 0-3 (Bits 48-51 in GDT):** Represents bits 16-19 of the segment limit. So, lessee... 1111b is equal to 0xf. Remember that, in the first two bytes if this descriptor, we set 0xffff as the first 15 bites. Grouping the low and high bits, **it means we can access up to 0xFFFFF**. Cool? It gets better... By enabling the 20th address line, we can access **up to 4 GB of memory** using this descriptor. We will look closer at this later...
- **Bit 4 (Bit 52 in GDT):** Reserved for our OS's use--we could do whatever we want here. Its set to 0.
- **Bit 5 (Bit 53 in GDT):** Reserved for something. Future options, maybe? Who knows. Set to 0.
- **Bit 6 (Bit 54 in GDT):** is the segment type (16 or 32 bit). Lessee.... we want 32 bits, don't we? After all-we are building a 32 bit OS! So, yeah--Set to 1.
- **Bit 7 (Bit 55 in GDT):** Granularity. By setting to 1, each segment will be bounded by 4KB.

The last byte is bits 24-32 of the base (Starting) address--which, of course is 0.

That's all there is to it!

The Data Descriptor

Okay then--go back up to the GDT that we made, and compare the code and data selectors: **They are exactaly the same, except for one single bit. Bit 43. Looking back at the above, you can see why: It is set if its a code selector, not set if its a data selector.**

Conclusion

This is the most comprehensive GDT description I have ever seen (and written!) That's a good thing though, right?

Okay, Okay--I know, the GDT is ugly. Loading it for use is very easy though--so it has benefits! Actually, all you need to do is load the address of a pointer.

This GDT pointer stores the size of the GDT (**Minus one!**), and the beginning address of the GDT. For example:

```
toc:
        dw end_of_gdt - gdt_data - 1    ; limit (Size of GDT)
        dd gdt_data                     ; base of GDT
```

gdt_data is the beginning of the GDT. **end_of_gdt** is, of course, a label at the end of the GDT. Notice the size of this pointer, and note its format. **The GDT pointer must follow this format.** Not doing so will cause unpredictable results--Most likely a triple fault.

The processor uses a special register--**GDTR**, that stores the data within the base GDT pointer. To load the GDT into the GDTR register, we will need a special instruction...**LGDT** (Load GDT). It is very easy to use:

```
lgdt    [toc]           ; load GDT into GDTR
```

This is not a joke--it really is that simple. Not much times do you actually get nice breaks like this one is OS Dev. Brace it while it lasts!

Local Descriptor Table

The Local Descriptor Table (LDT) is a smaller form of the GDT defined for specialized uses. It does not define the entire memory map of the system, but instead, only up to 8,191 memory segments. We will go into this more later, as it does not have to do with protected mode. Cool?

Interrupt Descriptor Table

THIS will be important. Not yet, though. The Interrupt Descriptor Table (IDT) defines the Interrupt Vector Table (IVT). It always resides from address 0x0 to 0x3ff. The first 32 vectors are reserved for hardware exceptions generated by the processor. For example, a **General Protection Fault**, or a **Double Fault Exception**. This allows us to trap processor errors without triple faulting. More on this later, though.

The other interrupt vectors are mapped through a **Programmable Interrupt Controller** chip on the motherboard. We will need to program this chip directly while in protected mode. More on this later...

PMode Memory Addressing

Remember that **PMode** (Protected Mode) uses a different addressing scheme then real mode. Real mode uses the **Segment:Offset** memory model, However PMode uses the **Descriptor:Offset** model.

This means, in order to access memory in PMode, we have to go through the correct descriptor in the GDT. The descriptor is stored in CS. This allows us to indirectly refrence memory within the current descriptor.

For example, if we need to read from a memory location, we do not need to describe what descriptor to use; it will use the one currently in CS. So, this will work:

```
mov bx, byte [0x1000]
```

This is great, but sometimes we need to refrence a specific despcriptor. For example, Real Mode does not use a GDT, While PMode requires it. Because of this, when entering protected mode, **We need to select what descriptor to use** to continue execution in protected mode. After all, because Real Mode does not know what a GDT is, there is no guarantee that CS will contain the correct descriptor in CS, so we need to set it.

To do this, we need to set the descriptor directly:

```
jmp     0x8:Stage2
```

You will see this code again. Remember that the first number is the **descriptor** (Remember PMode uses descriptor:address memory model?)

You might be courius at where the 0x8 came from. Please look back at the above GDT. **Remember that each descriptor is 8 bytes in size**. Because our **Code descriptor** is 8 bytes from the start of the GDT, we need to offset 0x8 bytes in the GDT.

Understanding this memory model is very important in understanding how protected mode works.

Entering Protected Mode

To enter protected mode is fairly simple. At the same time, it can be a complete pain. To enter protected mode, we have to load a new GDT which describes permission levels when accessing memory. We then need to actually switch the processor into protected mode, and jump into the 32 bit world. Sounds easy, don't you think?

The problem is the details. **One little mistake can triple fault the CPU**. In other words, watch out!

Step 1: Load the Global Descriptor Table

Remember that the GDT describes how we can access memory. If we do not set a GDT, the default GDT will be used (Which is set by the BIOS--Not the ROM BIOS). As you can imagine, this is by no means standard among BIOS's. And, **if we do not watch the limitations of the GDT (ie...if we access the code selector as data), the processor will generate a General Protection Fault (GPF). Because no interrupt handler is set, the processor will also generate a second fault exception--which will lead to a triple fault.**

Anywho...Basically, all we need to do is create the table. For example:

```
; Offset 0 in GDT: Descriptor code=0

gdt_data:
        dd 0                            ; null descriptor
        dd 0

; Offset 0x8 bytes from start of GDT: Descriptor code therfore is 8

; gdt code:                             ; code descriptor
        dw 0FFFFh                       ; limit low
        dw 0                            ; base low
        db 0                            ; base middle
        db 10011010b                    ; access
        db 11001111b                    ; granularity
        db 0                            ; base high

; Offset 16 bytes (0x10) from start of GDT. Descriptor code therfore is 0x10.

; gdt data:                             ; data descriptor
        dw 0FFFFh                       ; limit low (Same as code)
        dw 0                            ; base low
        db 0                            ; base middle
        db 10010010b                    ; access
        db 11001111b                    ; granularity
        db 0                            ; base high

;...Other descriptors begin at offset 0x18. Remember that each descriptor is 8 bytes in size?
; Add other descriptors for Ring 3 applications, stack, whatever here...

end_of_gdt:
toc:
        dw end_of_gdt - gdt_data - 1    ; limit (Size of GDT)
        dd gdt_data                     ; base of GDT
```

This will do for now. Notice **toc**. This is the pointer to the table. The first word in the pointer is the size of the GDT - 1. The second dword is the actual address of the GDT. **This pointer must follow this format. Do NOT forget to subtract the 1!**

We use a special Ring 0-only instruction - **LGDT** to load the GDT (Based on this pointer), into the **GDTR** register. Its a single, simple, one line instruction:

```
cli                     ; make sure to clear interrupts first!
lgdt    [toc]           ; load GDT into GDTR
sti
```

That's it! Simple, huh? Now... Onto protected mode! Um...Oh yeah! Heres **Gdt.inc** to hide all the ugly GDT stuff:

```
;*********************************************
;       Gdt.inc
;               -GDT Routines
;
;       OS Development Series
;*********************************************

%ifndef __GDT_INC_67343546FDCC56AAB872_INCLUDED__
%define __GDT_INC_67343546FDCC56AAB872_INCLUDED__

bits    16

;*****************************************
; InstallGDT()
;       - Install our GDT
;*****************************************

InstallGDT:

        cli                             ; clear interrupts
        pusha                           ; save registers
        lgdt    [toc]                   ; load GDT into GDTR
        sti                             ; enable interrupts
        popa                            ; restore registers
        ret                             ; All done!

;*****************************************
; Global Descriptor Table (GDT)
;*****************************************

gdt_data:
        dd 0                            ; null descriptor
        dd 0

; gdt code:                             ; code descriptor
        dw 0FFFFh                       ; limit low
        dw 0                            ; base low
        db 0                            ; base middle
        db 10011010b                    ; access
        db 11001111b                    ; granularity
        db 0                            ; base high

; gdt data:                             ; data descriptor
        dw 0FFFFh                       ; limit low (Same as code)
        dw 0                            ; base low
        db 0                            ; base middle
        db 10010010b                    ; access
        db 11001111b                    ; granularity
        db 0                            ; base high

end_of_gdt:
toc:
        dw end_of_gdt - gdt_data - 1    ; limit (Size of GDT)
        dd gdt_data                     ; base of GDT


%endif ;__GDT_INC_67343546FDCC56AAB872_INCLUDED__
```

Step 2: Entering Protected Mode

Remember that bit table of the CR0 register? What was it? Oh yeah...

- **Bit 0 (PE) : Puts the system into protected mode**
- **Bit 1 (MP) : Monitor Coprocessor Flag** This controls the operation of the **WAIT** instruction.
- **Bit 2 (EM) : Emulate Flag**. When set, **coprocessor instructions will generate an exception**
- **Bit 3 (TS) : Task Switched Flag** This will be set when the processor switches to another **task**.
- **Bit 4 (ET) : ExtensionType Flag. This tells us what type of coprocessor is installed.**
  - 0 - 80287 is installed
  - 1 - 80387 is installed.
- **Bit 5 :** Unused.
- **Bit 6 (PG) : Enables Memory Paging.**

The important bit is bit 0. **By setting bit 0, the processor continues execution in a 32 bit state.** That is, **Setting bit 0 enables protected mode.**

Here is an example:

```
                mov             eax, cr0                        ; set bit 0 in CR0-go to pmode
                or              eax, 1
                mov             cr0, eax
```

That's it! If bit 0 is set, Bochs Emulator will know that you are in protected mode (PMode).

Remember: The code is still 16 bit until you specify **bits 32**. As long as you code is in 16bit, you can use segment:offset memory model.

Warning! Insure interrupts are DISABLED before going into the 32 bit code! If it is enabled, the processor will triple fault. (Remember that we cannot access the IVT from pmode?)

After entering protected mode, we run into an immediate problem. Remember that, in Real Mode, we used the **Segment:Offset** memory model? However, **Protected Mode** relies on the **Descriptor:Address** memory model.

Also, remember that Real Mode does not know what a GDT is, while in PMode, the use of it is **Required**, because of its addressing mode. Because of this, in real mode, CS still contains the last segment address used, **Not the descriptor to use**.

Remember that PMode uses CS to store the current code descriptor? So, in order to fix CS (So that it is set to our code descriptor) we need to **far jump**, using our code descriptor.

Because our code descriptor is 0x8 (8 bytes offset from start of GDT), just jump like so:

```
        jmp     08h:Stage3              ; far jump to fix CS. Remember that the code selector is 0x8!
```

Also, once in PMode, we have to reset all of the segments (As they are incorrect) to their correct descriptor numbers.

```
        mov             ax, 0x10                ; set data segments to data selector (0x10)
        mov             ds, ax
        mov             ss, ax
        mov             es, ax
```

Remember that our data descriptor was 16 (0x10) bytes from the start of the GDT?

You might be curius at why all of the refrences inside the GDT (to select the descriptor) are offsets. Offsets of what? Remember the GDT pointer that we loaded in via the **LGDT** instruction? The processor bases all offset address off of the base address that we set the GDT pointer to point to.

Heres the entire Stage 2 bootloader in its entirety:

```
bits    16

; Remember the memory map-- 0x500 through 0x7bff is unused above the BIOS data area.
; We are loaded at 0x500 (0x50:0)

org 0x500

jmp     main                            ; go to start

;*************************************************
;       Preprocessor directives
;*************************************************

%include "stdio.inc"                    ; basic i/o routines
%include "Gdt.inc"                      ; Gdt routines

;*************************************************
;       Data Section
;*************************************************

LoadingMsg db "Preparing to load operating system...", 0x0D, 0x0A, 0x00

;*************************************************
;       STAGE 2 ENTRY POINT
;
;               -Store BIOS information
;               -Load Kernel
;               -Install GDT; go into protected mode (pmode)
;               -Jump to Stage 3
;*************************************************

main:

        ;-----------------------------;
        ;   Setup segments and stack    ;
        ;-----------------------------;

        cli                             ; clear interrupts
        xor     ax, ax                  ; null segments
        mov     ds, ax
        mov     es, ax
        mov     ax, 0x9000              ; stack begins at 0x9000-0xffff
        mov     ss, ax
        mov     sp, 0xFFFF
        sti                             ; enable interrupts

        ;-----------------------------;
        ;   Print loading message       ;
        ;-----------------------------;

        mov     si, LoadingMsg
        call    Puts16

        ;-----------------------------;
```

```
        ;   Install our GDT             ;
        ;-------------------------------;

        call    InstallGDT              ; install our GDT

        ;-------------------------------;
        ;   Go into pmode               ;
        ;-------------------------------;

        cli                             ; clear interrupts
        mov     eax, cr0                ; set bit 0 in cr0--enter pmode
        or      eax, 1
        mov     cr0, eax

        jmp     08h:Stage3              ; far jump to fix CS. Remember that the code selector is 0x8!

        ; Note: Do NOT re-enable interrupts! Doing so will triple fault!
        ; We will fix this in Stage 3.

;*****************************************************
;       ENTRY POINT FOR STAGE 3
;*****************************************************

bits 32                                 ; Welcome to the 32 bit world!

Stage3:

        ;-------------------------------;
        ;   Set registers               ;
        ;-------------------------------;

        mov             ax, 0x10                ; set data segments to data selector (0x10)
        mov             ds, ax
        mov             ss, ax
        mov             es, ax
        mov             esp, 90000h             ; stack begins from 90000h

;*****************************************************
;       Stop execution
;*****************************************************

STOP:

        cli
        hlt
```

Conclusion

I'm excited, are you? We went over alot in this tutorial. We talked about the GDT, descriptor tables, and getting into protected mode.

Welcome to the 32 bit world!

This is great for us. Most compilers only generate 32 bit code, so protected mode is necessary. Now, we would be able to execute the 32 bit programs written from almost any language - C or assembly.

We are not done with the 16 bit world yet though. In the next tutorial, we are going to get BIOS information, and loading the kernel through FAT12. This also means, of course, we will create a small little stub kernel. Cool, huh?

Hope to see you there!

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 7 Home Chapter 9

Operating Systems Development Series

Operating Systems Development - Enabling A20

by Mike, 2008

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

Introduction

Welcome! :)

In the previus tutorial, we looked at how to switch the processor into a 32 bit mode. We also learned how we can access up to 4 GB of memory. This is great--but, **how**?

Also, remember that the PC boots into real mode, which has the limitation of 16 bit registers. And, hence, 16 bit segment addressing. This limits the amount of memory you can access in real mode. Because of this, we still cannot access even up to 1 GB of memory yet. Heck, we cannot even go passed the 1 MB barrier yet! What to do? We have to enable the 20th address line. This will require direct hardware programming, so we will talk about that as well.

So, This is what's on the menu:

- Direct Hardware Programming - Theory
- Direct Hardware Programming and Controllers
- Keyboard Controller Programming - Basics
- Enabling A20
- Pizza :)

Do to the use of high level languages, like C, being able to access more then 1 MB of memory can be critical. Because of this, enabling A20 (Address line 20) will be important!

Note: Remember that we cannot access over 1 MB yet! Doing so will cause a triple fault.

Also, because we are going to go over direct hardware programming, this tutorial will be a little more complicated then previous ones. Don't worry - You will get more expeirence with direct hardware programming later when we develop device drivers for the Kernel.

Ready?

Get Ready

For those who have been with me this far, I am certain you know how hard OS development is. However, we have not touched anything close to hard. All of the concepts listed here is still very basic, and yet quite advanced. However: **Things are only going to get much more harder.**

Every single controller must be programmed a special way in order to work correctly. For example, to write (or read) a hard drive, you must first determin if it is an IDE or SCSI drive. Then, you have to determin the drive number it is, and program it using either the **IDE Controller** or **SCSI Controller**, which control the IDE and SCSI connections, respecitvley. Both of these controllers are different.

To add more complexity, a "Sector" might not be 512 bytes. Hence, "Reading and writing sectors" is vauge.

Then comes memory management and fragmentation. This is where **Paging**, **Virtual Address Space**, and the **Memory Management Unit (MMU)** comes into play.

Reading and writing any drive is very different then any other drive. This is even true at the bootsector level. The typical format and file system is different between media, so code that boots from a FAT12 floppy will **Not** work to boot a CDFS Filesystem CD ROM. By Abstracting hardware specific (And low level code), we can make most of the code, however, work for these devices.

When we say "Write a file to hard drive", we normally don't want to define what a "file" is, because we shouldn't. We shouldn't have to worry about what controller to read to/from, nor the exact location on disk. **This is why abstraction is *very* important!**

Everything here is primairly for protected mode (i.e., it is 32 bit code), although it will work in real mode as well. Because of this, remember the rule of protected mode:

- **No interrupts are avilable! The use of any interrupt will cause a triple fault**

...Hence, you are **Completely** on your own.

Kernel Debugging

Debugging is an art form. It provides a way of trapping problems, and fixing errors through software before they become serious. **Kernel Debugging** relates to debugging kernel-level Ring 0 programs. This is never an easy task.

Debuggers in High Level Languages

Most debuggers in languages, like C and C++, provide a way of displaying varable and routine names, and their values, locations, etc during runtime. The problem? We don't have any symbolic names yet in any of our programs. We are still working at the **Binary** level.

What this means is that we need a debugger that could work and display memory directly. Bochs has a debugger just for us.

Bochs Debugger

Bochs comes with a debugger called **bochsdbg.exe**. When you launch it, you will be given the same startup screen from Bochs.exe. Load your configuation file, and start the emulation.

Bochs debugger and display window will now appear, and you should see the line:

```
[0x000ffff0] f000:fff0 (unk. ctxt): jmp f000:e05b         ; ea5be000f0
< bochs:1> _
```

In the second line, bochs tells you the number of commands sent to it (In this case, this is the first command, so a 1 is displayed). You can type your commans here.

The first line is the important line. It tells you the current instruction, absolute address, and seg:offset address. It also gives you the machine language **Operation Code (Opcode)** equivelant.

HELP command

The **help** command gives you a list of available commands.

BREAK command

The **b (break)** command allows you to set breakpoints at addresses in memory. For example, if we are trying to debug our OS, we need to start at the bootloader (07c00:0). However, Bochs Debugger starts where the BIOS is at. Because of this, we will need to set a breakpoint to 0x7c00:0, and continue execution until that breakpoint is reached:

```
                        // BIOS is at 0xea5be000f0
[0x000ffff0] f000:fff0 (unk. ctxt): jmp f000:e05b
< bochs:1> b 0x7c00                    // Sets the breakpoint to 0x7c00:0
< bochs:2> c                           // Continue executon
< 0> Breakpoint 1, 0x7c00 in ?? < >    // Our breakpoint is hit
Next at t=834339
                        // We are now at our bootloaders first instruction
< 0> [0x00007c00] 0000:7c00 (unk. ctxt): jmp 7cb5         ; e9b200/DIV>
```

The above tells us that our main() function in our bootloader is at 0x7cb5. This makes sinse because, remember that the OEM Parameter Block is between this jump instruction, and the start of main().

Knowing that the bootloader loads stage 2 at 0x500, lets break to it:

```
< bochs:3> b 0x500
< bochs:4> c
< 0> Breakpoint 2, 0x500 in ?? < >
Next at t=934606
<0> [0x000000500] 0050:0000 (ink. ctxt): jmp 00a0        ; e99d00
< bochs:5> _
```

Now, we are at the beginning of stage 2, and could follow the debugger with our assembly file! Cool, huh? Best of all, you can see the window dynamically update to display the output of your system.

Single Step

The **s (Single Step)** command is used to walk through one instruction at a time:

```
< bochs:6> s
Next at t=934607
<0> [0x0000005a0] 0050:00a0 (ink. ctxt): cli              ; fa
< bochs:7> s
Next at t=934608
<0> [0x0000005a1] 0050:00a1 (ink. ctxt): xor AX, AX       ; 31c0
< bochs:8> _
```

dump_cpu

This command displays the current value of all cpu registers, including RFLAGS, General Purpose, Test, Debug, Control, and Segment registers. It also includes GDTR, IDTR, LDTR, TR, and EIP.

print_stack

This displays the current values of the stack. This is critical coinsidering that we use the stack very often.

Conclusion

There are more commands then this, however these are the most usefull. Learning how to use the debugger is very important, espically in the early stages like we are in now.

Direct Hardware Programming - Theory

This is where things start getting very hard in operating system development.

"Direct hardware programming" simply referrs to communicating directly (and controlling) individual chips. As long as these chips are programmable (In some way), we can control them.

In Tutorial 7, we took a very detailed look at how the system works. We also talked about how software ports work, port mapping, the IN and OUT instructions, and I gave a huge table with common port mappings on x86 architectures.

Remember that, whenever the processor recieves an **IN** or **OUT** instruction, It enables the **I/O Access Line** on the **Control Bus**, which, of course, is part of the **System Bus**, in the **Motherboards North Bridge**. Because the system bus is connected to both the **Memory Controller** and **I/O Controller**, both controllers listen for specific addresses and activated lines in the control bus. If the **I/O Access line** is set (Electricity runs through it--which means it is active (1), The I/O controller takes the address.

The I/O Controller then gives the port address to every other device, and awaits a signal from a controller chip (meaning that it belongs to some device--so give whatever data to that device). If no controller chip responds, and the port address is set back, it is ignored.

This is how port mapping works. (Please see tutorial 7 for more detail.)

Also, remember that a single controller chip may be assigned a range of port addresses. Port addresses are assigned by the BIOS POST, even before the BIOS is loaded and executed. Why? Alot of devices need different types of information. Some ports may represent "registers", while others may be "data" or "ready" ports. It's ugly, I know. But it gets worse. On different systems, port addresses may vary widely. Because x86 architectures are backword compatible, basic devices (Such as keyboards and mice) are useually always the same address. More complex devices, however, may not be.

Direct Hardware Programming and Controllers

To better understand how everything works, lets look at controllers. After all, we will be talking to them alot--espically in protected mode.

Many PCs are based off of the early **Intel 8042 Microcontroller chip**. This controller chip is either embedded as an IC (Intergrated Circuit) chip, or directly in the motherboard. It is useually locaed in the **South Bridge**.

This microcontroller communcates through a cord connecting to your keyboard, to another microcontroller chip in your keyboard.

When you press a key on the keyboard, It presses down on a rubber dome setting beneath the key itself. On the underside of the rubber dome is a conductive contact that, when pressed down, comes in contact with two conductive contacts on the keyboard circuit. Because of this, current can flow through. Each key is connected by a pair of electrical lines. As each signal changes (Depending on weather keys are pressed), a make code is generated (From the series of lines). This make code is sent to the microcontroller chip inside of the keyboard, and sent through the cord connecting to the computer hardware port. It is sent through as a series of on and off electrical pusles. Depending on the clock cycles, each pulse can be converted to a series of bits representing a bit pattern.

We are on the motherboard. This series of bits goes through the south bridge as electrical signals, all the way to the 8042 microcontroller. This microcontroller decodes the make code into a scan code, and stores it within an internal register. That is, our buffer. The internal registers can be an EEPROM chip, or simular, so we can electrically overwrite the data whenever we want.

When booting, the BIOS POST assigns each device (Through the I/O Controller) port addresses. It does this by quering the devices. In the useual case, the BIOS POST sets this internal register at port address 0x60. This means, **Whenever we refrence port 0x60, we are requesting to read from this internal register.**

You know the rest of the story reguarding port mapping, and IN/ OUT instructions, so lets read from that register:

```
in al, 0x60             ; get byte from 8042 microcontroller input register
```

As you can probably guess, **The 8042 Microcontroller is the Keyboard Controller**. By communicating with the various of registers with the chip, we can read input from the keyboard, map scan codes, even several other things: **Like enabling A20.**

You might be wondering why you have to communicate to the keyboard controller to enable A20. We will look at this next.

Gate A20 - Theory

Finally we cover A20. I know, I know...Most of this tutorial so far covers other topics that are not directly related to A20. However, I wanted to start with the basics of direct hardware programming first before going into A20..as enabling A20 requires it, along with any microcontroller programming.

Enabling the A20 line may require programming the keyboard microcontroller. Because of this, I will cover a little bit about programming the keyboard controller but will not go into keyboard programming just yet.

A little history

When IBM designed the **IBM PC AT** machines, it used their newer **Intel 80286 microprocessor**, which was not entirely compatible with previous x86 microprocessors when in real mode. The problem? The older x86 processors **did not** have address lines A20 through A31. They did not have an address bus that size yet. **Any programs that go beyond the first 1 MB would appear to wrap around.** While it worked back then, the 80286's address space required 32 address lines. However, if all 32 lines are accessable, we get the wrapping problem again.

To fix this problem, Intel put a logic gate on the 20th address line between the processor and system bus. This logic gate got named **Gate A20**, as it can be enabled and disabled. For older programs, it an be disabled for programs that rely on the wrap wround, and enabled for newer programs.

When booting, the BIOS enables A20 when counting and testing memory, and then disables it again before giving our operating system control.

There are alot of ways to enable A20. By enabling the A20 gate, we have access to all 32 lines on the address bus, and hence, can refrence 32 bit addresses, or up to 0xFFFFFFFF - 4 GB of memory.

The Gate A20 is an electronic OR gate that was originally connected to the P21 electrical line of the 8042 microcontroller (The keyboard controller). This gate is an output line that is treated as **Bit 1** of the output port data. We can send a command to recieve this data and even modify it. By setting this bit, and writing the output line data we can have the microcontroller set the OR gate thus enabling the A20 line. We can either do this ourself directly or indirectly. We will look more in the next section.

During bootup, the BIOS enables the A20 line to test the memory. After the memory test, the BIOS disables the A20 line to retain compatability with older processors. Because of this, by default, the A20 line is disabled for our operating system.

There can be several different ways to re-enable gate A20 depending on the motherboard configuation. Because of this, I will cover several different more common methods to enable A20.

Lets look at this next. ;)

Gate A20 - Enabling

Remember that there are alot of different ways to enable A20. If you are wanting to just get your system working, all you need to do is use a method that works for you. If portability is a requirement, you may be required to use a mixture of methods.

Method 1: System Control Port A

This is a very fast, yet less portable method of enabling the A20 address line.

Sone systems, including MCA and EISA we can control A20 from the system control port I/O 0x92. The exact details and functions of port 0x92 vary greatly by manufacturer. There are several bits that are commonly used though:

- **Bit 0** - Setting to 1 causes a fast reset (Used to switch back to real mode)
- **Bit 1** - 0: disable A20; 1: enable A20
- **Bit 2** - Manufacturer defined
- **Bit 3** - power on password bytes (CMOS bytes 0x38-0x3f or 0x36-0x3f). 0: accessible, 1: inaccessible
- **Bits 4-5** - Manufacturer defined
- **Bits 6-7** - 00: HDD activity LED off; any other value is "on"

Here is an example that enables A20 using this method:

```
mov     al, 2   ; set bit 2 (enable a20)
out     0x92, al
```

Notice there is alot of other things we can do with this port:

```
mov     al, 1   ; set bit 1 (fast reset)
out     0x92, al
```

This method seems to work with Bochs as well.

Warning!

While this is one of the easier methods, I have seen this method conflict with some other hardware devices. It would normally cause the system to halt. If you want to use this method (and it works for you), I would stick with using it, but please keep this in mind.

Other Ports...

I feel that I should mention that some systems allow the use of other I/O ports to enable and disable A20.

The most common of these are I/O port 0xEE. If I/O port 0xEE ("FAST A20 GATE") is enabled on these systems, reading from this port enables A20, and writing to it disables A20. A simular effect accors for port 0xEF ("FAST CPU RESET") as well for resetting the system.

Other systems may use different ports (ie; AT&T 6300+ needs a write of 0x90 to I/O port 0x3f20 to enable A20, and a write of 0 to disable A20). There are also rumours of systems that exist that use bit 2 of I/O port 0x65 or bit 0 of I/O port 0x1f8 for enabling

and disabling A20 (0: disable, 1: enable).

As you can see, there are alot of headaches when it comes to working with A20. The only way to be sure is with your motherboard manufacturer.

Method 2: Bios

Alot of Bios's make interrupts avilable for enabling and disabling A20.

Bochs Support

It seems some versions of Bochs recognizes these methods but it may not be supported on some versions of Bochs.

INT 0x15 Function 2400 - Disable A20

This function disables the A20 gate. It is very easy to use:

```
mov ax, 0x2400
int 0x15
```

Returns:

CF = clear if success
AH = 0
CF = set on error
AH = status (01=keyboard controller is in secure mode, 0x86=function not supported)

INT 0x15 Function 2401 - Enable A20

This function enables the A20 gate.

```
mov ax, 0x2401
int 0x15
```

Returns:

CF = clear if success
AH = 0
CF = set on error
AH = status (01=keyboard controller is in secure mode, 0x86=function not supported)

INT 0x15 Function 2402 - A20 Status

This function returns the current status of the A20 gate.

```
mov ax, 0x2402
int 0x15
```

Returns:

CF = clear if success
AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)
AL = current state (00: disabled, 01: enabled)
CX = set to 0xffff is keyboard controller is no ready in 0xc000 read attempts
CF = set on error

INT 0x15 Function 2403 - Query A20 support

This function is used to query the system for A20 support.

```
mov ax, 0x2403
int 0x15
```

Returns:

CF = clear if success
AH = status (01: keyboard controller is in secure mode; 0x86: function not supported)
BX = status.

BX contains a bit pattern:

- **Bit 0** - 1 if supported on keyboard controller
- **Bit 1** - 1 if supported on bit 1 of I/O port 0x92
- **Bits 2-14** - Reserved
- **Bit 15** - 1 if additional data is available.

Method 3: Keyboard Controller

This is probably the most common method of enabling A20. Its quite easy, but requires some knowledge of programming the keyboard microcontroller. This will be the method I will be using as it seems it is also the most portable. Because this requires some knowledge of programming the keyboard microcontroller, we should look at that a little bit first.

This is also the reson why I wanted to cover hardware programming first. This will be our first glimps into direct hardware programming, and what it is all about. Don't worry, it's not too bad ;) It can get quite complex at times though :)

8043 Keyboard Controller - Port Mapping

Remember that -- in order for us to communicate with this controller, we must know what I/O ports the controller uses.

This controller has the following port mapping:

Port Mapping		
Port	Read/Write	Descripton
0x60	Read	Read Input Buffer
0x60	Write	Write Output Buffer
0x64	Read	Read Status Register
0x64	Write	Send Command to controller

We send commands to this controller by writing the command byte to I/O Port 0x64. If the command accepts a parameter, this parameter is sent to port 0x60. Likewise, any results returned by the command may be read from port 0x60.

We must note that the keyboard controller itself is quite slow. Because our code will be executing faster then the keyboard controller, we must provide a way to wait for the controller to be ready before we continue on.

This is useually done by quering for the controllers status. If this seems confusing, don't worry--everything will be clear soon enough.

8043 Keyboard Controller Status Register

Okay, how do we get the status of the controller? Looking at the table above, we can see that we must read from I/O port 0x64. The value read from this register is an 8 bit value that follows a specific format. Here it is...

- **Bit 0: Output Buffer Status**
  - 0: Output buffer empty, dont read yet
  - 1: Output buffer full, please read me :)
- **Bit 1: Input Buffer Status**
  - 0: Input buffer empty, can be written
  - 1: Input buffer full, dont write yet
- **Bit 2:** System flag
  - 0: Set after power on reset
  - 1: Set after successfull completion of the keyboard controllers self-test (Basic Assurance Test, BAT)
- **Bit 3:** Command Data
  - 0: Last write to input buffer was data (via port 0x60)
  - 1: Last write to input buffer was a command (via port 0x64)
- **Bit 4:** Keyboard Locked
  - 0: Locked
  - 1: Not locked
- **Bit 5:** Auxiliary Output buffer full
  - PS/2 Systems:
    - 0: Determins if read from port 0x60 is valid If valid, 0=Keyboard data
    - 1: Mouse data, only if you can read from port 0x60
  - AT Systems:
    - 0: OK flag
    - 1: Timeout on transmission from keyboard controller to keyboard. **This may indicate no keyboard is present.**
- **Bit 6:** Timeout
  - 0: OK flag
  - 1: Timeout
  - PS/2:
    - General Timeout
  - AT:
    - Timeout on transmission from keyboard to keyboard controller. **Possibly parity error (In which case both bits 6 and 7 are set)**
- **Bit 7:** Parity error
  - 0: OK flag, no error
  - 1: Parity error with last byte

As you can see, there is alot going on here! The important bits are bolded above--they will tell us if the controllers output or input buffers are full or not.

Here is an example. Lets say we send a command to the controller. This is placed in the controllers input buffer. So, as long as this buffer is still full, we know our command is still being performed. Here is what our code might look like:

```
wait_input:
        in      al,0x64         ; read status register
        test    al,2            ; test bit 2 (Input buffer status)
        jnz     wait_input      ; jump if its not 0 (not empty) to continue waiting
```

We will be needing to do this for both the input and output buffers.

Now that we are able to wait for the controller, we must be able to actually tell the controller what we need it to do. This is done through command bytes. Lets take a look!

8043 Keyboard Controller Command Register

Looking back at the I/O port table, we can tell that we need to write to I/O Port 0x64 to send commands to the controller.

The keyboard controller has **alot** of commands. Because this is not a tutorial on keyboard programming, I will not list them all here. However, I will list the more important ones:

Keyboard Controller Commands	
Keyboard Command	Descripton
0x20	Read Keyboard Controller Command Byte
0x60	Write Keyboard Controller Command Byte
0xAA	Self Test
0xAB	Interface Test
0xAD	Disable Keyboard
0xAE	Enable Keyboard
0xC0	Read Input Port
0xD0	Read Output Port
0xD1	Write Output Port
0xDD	Enable A20 Address Line
0xDF	Disable A20 Address Line
0xE0	Read Test Inputs
0xFE	System Reset
Mouse Command	Descripton
0xA7	Disable Mouse Port
0xA8	Enable Mouse Port
0xA9	Test Mouse Port
0xD4	Write to mouse

Again, please take note there are **alot** more commands then this. We will look at them all later, don't worry :)

Method 3.1: Enabling A20 through keyboard controller

Notice the command bytes **0xDD** and **0xDF** in the above table. This is one way to enable A20 using the keyboard controller:

```
; Method 3.1: Enables A20 through keyboard controller
mov al, 0xdd    ; command 0xdd: enable a20
out 0x64, al    ; send command to controller
```

Not all keyboard controllers support this function. If it works, I would stick with it for its simplicity ;)

Method 3.2: Enabling A20 through Output Port

Yet *another* method of enabling A20 is through the keyboard controllers output port. To do this, we need to use commands D0 and D1 to read and write the output port (Please see the **Keyboard Controller Commands** table again for refrence.)

This method is a little bit more complex then the other methods, but it is not too bad. Basically, we can disable the keyboard and read the output port from the controller. The 8042 containes thee ports: One is input, the other is output. Oh right... The third is for testing. These "ports" are just the hardware pins on the microcontroller.

To keep things simple (And because this isnt a keyboard programming tutorial), we will just look at the output port for now.

Okay... read from the output port, simply send the...erm...read output port command (0xD0) to the controller: (Please see the keyboard controller commands table for refrence)

```
; read output port into al
mov     al,0xD0
out     0x64,al
```

Now we have gotten the output port data. Great, but that isnt very useful, is it? Well, actually the output port data follows..yet again..a specific bit format.

Lets take a look...

- **Bit 0: System Reset**
  - 0: Reset computer
  - 1: Normal operation
- **Bit 1: A20**
  - 0: Disabled
  - 1: Enabled
- **Bit 2-3:** Undefined
- **Bit 4:** Input buffer full
- **Bit 5:** Output Buffer Empty
- **Bit 6:** Keyboard Clock
  - 0: High-Z
  - 1: Pull Clock Low
- **Bit 6:** Keyboard Data
  - 0: High-Z
  - 1: Pull Data Low

Most of these bits we do not want to change. Setting bit 0 to 1 resets the computer; setting bit 1 enables gate A20. You should OR this value to set the bit to insure none of the other bits get touched. After setting the bit, just write the value back (Command byte 0xD1).

The commands used to read and write the output port use the input and output buffers of the controller for its data.

This means, if we read the output port, the data read will be in the controllers input buffer register. Looking back at the I/O port table, this means to get the data we read from I/O port 0x60.

Lets look at an example. During any read or write operation, we will want to wait for the controller to be ready. **wait_input** waits for the input buffer to be empty, while **wait_output** waits for the output buffer to be empty.

```
; send read output port command
mov     al,0xD0
out     0x64,al
call    wait_output

; read input buffer and store on stack. This is the data read from the output port
in      al,0x60
push    eax
call    wait_input

; send write output port command
mov     al,0xD1
out     0x64,al
call    wait_input

; pop the output port data from stack and set bit 1 (A20) to enable
pop     eax
or      al,2            // 2 = 10 binary
out     0x60,al         // write the data to the output port. This is done through the output buffer
```

Thats all there is to it! :) This method is a bit more complex then the other methods, but it is also the most portable.

Cautions to look for

Because of it's emulation, most of these do not apply with Bochs, but to real hardware instead.

Controller executes the wrong command

If the controller executes the wrong command, it will useually do something you don't want. Like, perhaps, read data from a port instead of write data), which may currupt your data. For example, using **in al, 0x61** instead of **in al, 0x60**, which will read from a different register in the keyboard microcontroller, instead of the status register (Port 0x60).

Unkown Controller Command

Most controllers ignore commands it does not know, and just discards it (Clears it's command register, if any.)

Some controllers may malfunction, however. Please see the "Malfunction" section for more information.

Controller Malfunctions

This happens rairly, but is possible. Two notible instances are both with the Pentium processor, including the infamous FDIV and foof bugs. The FDIV bug was an internal CPU design flaw, in which the FPU inside the processor gives incorrect results.

The foof problem is more series. When the processor is given the command bytes 0xf0 0x0f 0xc7 0xc8, which is an example of an **Hault and Catch Fire (HCF)** instruction. (An **Undocumented Instruction**). Most of these instructions may lock up the processor itself, forcing the user to hard reboot. Others may cause unusual side effects from the use of these instructions.

One should coinsider these problems that may happen. It does happen, and controllers are no exception. (Remember that we send instruction bytes to individual ports? For example, Port 0x64--The Command Register in the Keyboard Controller).

Most of these malfunctions can easily be coinsidered "Design Flaws" of the device, though.

Physical Hardware damage

Although also rare, it is possible to infict hardware damage through software. An easy example is the floppy drive. You have to control the **floppy drive motor** directly through the **Floppy Drive Controller (FDC)**. Forgetting to send the command to stop the motor can cause the floppy drive to wear out and break. Be careful!

Triple Fault

The microcontroller may signal the primary processor that there is a problem via the Control Bus, in which case the processor signals an exception, which will, of course, reboot the computer.

Controller problems in Bochs

If there is a controller problem, Bochs will provoke a Triple fault, and log the information (The problem) into the log.

For example, if you try to send an unkown command (Such as 0) to the keyboard controller:

```
mov             al, 0x00                ; some random command
out             0x64, al                ; try to send command to controller
```

Bochs will provoke a triple fault, and log the information:

```
[KBD   ] unsupported io write to keyboard port 64, value = 0
```

"KBD" represents that the log was written by the keyboard controller device.

Demo

All of the A20 code is in **A20.inc**. I wrote several different routines that uses different methods of enabling A20. So if one method fails, try using another method.

Do to the increase in complexity, I have decided to have this demo downloadable. The current **Stage2.asm** has not changed that much either.

Because the demo does not display anything new, there is not a new picture to display.

Download the latest demo (*.ZIP: 8KB) Here.

Conclusion

Wow. Just, Wow. This tutorial is bigger then I originally expected.

We looked at alot of new concepts here. We also got experience with hardware programming. Remember: **This is the only way of communicating with hardware in protected mode!** Good bye interrupts. Good bye BIOS. Good bye everything--we are now completely on our own.

Right now, you can probably start appreciating Windows a little more :) After all, they all had to start at our level.

Don't worry if you do not understand everything yet--It is complicated, I know. When we get to our Kernel, we are going to have an entire tutorial dedicated to programming the keyboard microcontroller, and writing a driver for it. Cool?

The next tutorial will be much easier. We are going to put Protected Mode on hold for now, and go back to the real mode code. We will add the FAT12 loading code to load our kernel. Now that A20 is enabled, we can load it at 1MB!

Also, we will get some BIOS information, and anything else that comes to mind :) See you there.

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!

Operating Systems Development Series

Operating Systems Development - Prepare for the Kernel part 1

by Mike, 2008

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

Introduction

Welcome! :)

We have went over alot so far, havn't we? You should now be aware of the amount of complexity there is in OS development. And yet--it only gets harder.

This is our first two-part tutorial. The first part will describe all of the new code in detail. We will cover basic 32 bit graphics programming in assembly. This includes: Basic VGA Programming concepts, accessing video display, printing strings, clearing the screen, and updating the hardware cursor. There is a little math involved, but not too much :)

The demo itself is completed. It will be shown in the second part of this tutorial, along with an overview of the completed Stage 2 source code, including its new little FAT12 driver, Floppy driver. These are not "real" drivers by definition that we will add. However, they will help demenstrate the functionality of drivers, and why they are usefull. All of the code is a heavily modified version of our FAT12 loading code from the bootloader, so I will not be describing FAT12 again in detail.

With that, Part two--as being the last tutorial for Stage 2--will go over the loading and executing of a basic (pure binary) Kernel image at 1 MB.

This two part tutorial is the last tutorial for Stage 2! When we start the Kernel, we will need to cover different executable format files. We will need to insure Stage 2 executes the object files correctly. Because of this, when we start the Kernel, we will add the loader to our current Stage 2 bootloader to insure it loads our Kernel correctly. This is later, though :)

With all of that in mind, **Part 1 of this tutorial covers**:

- Basic VGA Programming Concepts
- Accessing the Display
- Printing characters
- Printing strings
- CRT Microcontroller theory and updating the hardware cursor
- Clearing the screen

This tutorial references The infamous Tutorial 7 alot. That is, the **Real Mode Addressing Map** and **Default I/O Port Addresses**. It may be helpfull to have that tutorial up when we talk about video address space and VGA port access.

Ready?

The Display

VGA - Theory

The **Video Graphics Array (VGA)** is an anolog computer display standard marketed in 1987 by IBM. It is called an "Array" because it was originally developed as a single chip, replacing dozens of logic chips in a Industry Standard Architecture (ISA) board that the **MDA**, **CGA**, and **EGA** used. Because this was all on a single ISA board, it was very easy to connect it to the motherboard.

The VGA consists of the **video buffer, video DAC, CRT Controller, Sequencer unit, Graphics Controller, and an Attribute Controller**. Please note that, we will not cover everything in detail yet until we start talking about video drivers. This is primarily to preseve space, and to make things more easier as programming the VGA can get quite complex.

Video Buffer

The Video Buffer is a segment of memory mapped as Video Memory. We can change what region of memory is mapped to video memory. **At startup, the BIOS maps it to 0xA0000.**, which means that video memory is mapped to 0xA0000. (Remember the Real Mode Address Map from Tutorial 7?) **This is important!**

Video DAC

The Video Digital to Analog Converter (DAC) contains the **color palette** that is used to convert the video data into an analog video signal that is sent to the display. This signal indicates the **red, green, and blue intensities** in analog form. We will go into more detail later, so don't worry if you do not understand this yet.

CRT Controller

This controller generates horizontal and vertical synchronization signal timings, **addressing for the video buffer, cursor and underline timings.** We will go into more detail later in this tutorial, as we need to go through the CRT Controller when updating the cursor.

Sequencer

The Sequencer generates basic memory timings for video memory and the character clock for controlling regenerative buffer fetches. It allows the system to access memory during active display intervals. Once more, we will not cover this in detail yet. We will cover everything in great detail later when looking at Video Drivers, don't worry :)

Graphics Controller

This is the interface between video memory and the attribute controller, and between video memory and the CPU. **During active display times, memory data is sent from the video buffer (Video Memory) and sent to the Attribute Controller.** In Graphics Modes, this data is converted from parallel to a serial bit plane data before being sent. In text modes, Just the parallel data is sent.

Don't worry if you do not understand these yet. I do not plan on going into much detail here. We will cover everything in detail later when we talk about developing a video driver. For now, just remember that: **The Graphics Controller refreshes the display from the parallel data from video memory.** This is automatic based on the active display times. This simply means, that **By writing to video memory (Default mapped to 0xA0000) we effectivly write to video display, depending on the current mode.** This is important when printing characters.

Remember that it is possible to change the address range used by the Graphics Cotroller. When initializing, the BIOS does just this to map video memory to 0xA0000.

Video Modes

A "Video Mode" is a specification of display. That is, it describes how **Video Memory** is refrenced, and how this data is displayed by the video adapter.

The VGA supports two types of modes: **APA Graphics**, and **Text**.

APA Graphics

All Points Addressable (APA) is a display mode, that, on a video monitor, dot matrix, or any device that consists of a pixel array, where every cell can be refrenced individually. In the case of video display, where every cell represents a "pixel", where every pixel can be manipulated directly. Because of this, almost all graphic modes use this method. **By modifying this pixel buffer, we effectivly modify individual pixels on screen.**

Pixel

A "Pixel" is the smallest unit that can be represented on a display. On a display, it represents the smallest unit of color. That is, basically, a single dot. The size of each pixel depends heavily on the current resolution and video mode.

Text Modes

A Text Mode is a display mode where the content on the screen is internally represented in terms of characters rather then pixels, as with APA.

A Video Controller implimenting text mode uses two buffers: A character map representing the pixels for each individual character to be displayed, and a buffer that represents what characters are in each cell. By changing the character map buffer, we effectivly change the characters themselves, allowing us to create a new character set. By changing the **Screen Buffer**, which represents what characters are in each cell, **we effectivly change what characters are displayed on screen.** Some text modes also allow attributes, which may provide a character color, or even blinking, underlined, inversed, brightened, etc.

MDA, CGA, EGA

Remember that VGA is based off of MDA, CGA, and EGA. VGA also supports alot of the modes these adapters do. Understanding these modes will help in better understanding VGA.

MDA - Theory

Back before I was born (Seriously :)) in 1981, IBM developed a standard video display card for the PC. They were the **Monochrome Display Adapter (MDA), and Monochrome Display and Printer Adapter (MDPA)**.

The MDA did not have any graphics mode of any kind. **It only had a single text mode, (Mode 7) which could display 80 columns by 25 lines of high resolution text characters.**

This display adapter was a common standard used in older PC's.

CGA - Theory

In 1981, IBM also developed the **Color Graphics Adapter (CGA)**, coinsidered the first color display standard for PC's.

The CGA only supported a **Color Palette** of 16 colors, because it was limited to **4 bytes per pixel.**

CGA supported two text modes and two graphics modes, including:

- 40x25 characters (16 color) text mode
- 18x25 characters (16 color) text mode
- 320x200 pixels (4 colors) graphics modes
- 640x200 pixels (Monochrome) graphics mode

It is possible to treak the display adapter in creating and discovering new, "undocumented" video modes. More on this later.

EGA - Theory

Introduced in 1984 by IBM, The **Enhanced Graphics Adapter (EGA)** produced a display of 16 colors at a resolution up to 640x350 pixels.

Remember that the VGA adapters are backward compatible, simular to the 80x86 microprocessor family. Because of this, and to insure backward compatibility, the BIOS starts up in Mode 7 (Originally from the MDA), which supports 80 columns, by 25 lines. This is important to us, because this is the mode we are in!

VGA Memory Addressing

Video memory used by the VGA Controller is mapped to the PC's memory from 0xA0000 to 0xBFFFF. **Remember the Real Mode Memory Map from Tutorial 7**!

Typically, the Video Memory is mapped as the following:

- **0xA0000 - 0xBFFFF** Video Memory used for graphics modes
  - **0xB0000 - 0xB7777** Monochrome Text mode
  - **0xB8000 - 0xBFFFF** Color text mode and CGA compatible graphics modes

Do to the different addresses used in the memory mapping, it is possible to have both ECG, CGA, and VGA display adapters installed on the same machine.

It is possible to change the memory mappings used by the video adapter cards through the CRT Microcontroller. Normally this is done through Video Drivers. More on this later, though.

One can also modify how the Video Controller uses this memory. In doing so, we can create "new", or rather, "undocumented" modes. One common mode is the infamous "Mode X".

Remember that modifying the display buffer and text buffers effectivly change what is displayed on screen? This is do to the video controller refreshing the display based on the current refresh rate. The Video Controller sends commands to the CRT Controller inside the Monitor through the VGA Port. This generates a **Vertical and Horizontal Retrace** of the CRT to refresh the monitors' display. And, because the text and display adapter is mapped to the above PC memory addresses:

Writing to this region of memory changes what is displayed on screen

For an example, remember that we are in Mode 7? Mode 7 is a color text mode, hence uses memory that begins at 0xB8000. Because this is the text buffer used by the Video Controller to determin what to display, **Writing to 0xB8000 effectivley displays text on screen.**

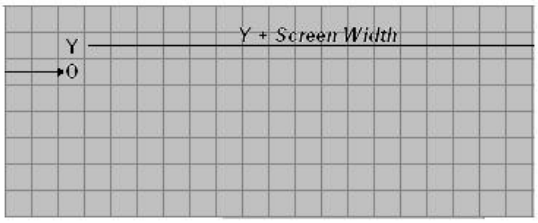
```
%define VIDMEM  0xB8000         ; video memory

mov     edi, VIDMEM             ; get pointer to video memory
mov     [edi], 'A'              ; print character 'A'
mov     [edi+1], 0x7            ; character attribute
```

The above will display the character "A", in white, black background (The attribute), in the top left corner of the display. Too cool for school :)

Printing characters

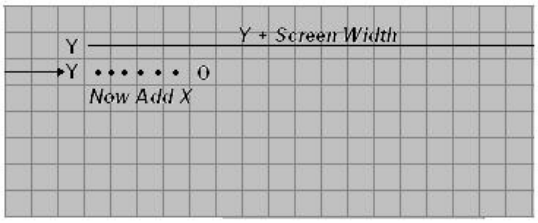
Okay, so how do we print a character at any x/y location on screen?

A special property about memory is how it is linear. If we reach the end of a line being displayed, the next byte is on the line right below it. Because of linear addressing, we have to be able to convert an x/y location to a linear address to render it to screen. And, a special forumula to do that is: **x + y * screen width**.



Notice by multiplying screen width by a value of Y, we effectivly move down one row on the screen within video memory.

Because of this, multiplying it by a value of Y means that, by incrementing the value of Y, we effectivly go down Y rows.



Now, we add the value of "X" to this location, giving us a forumla: ***Location = x + (y * screen width)***

Remember that memory is linear!

Here is an example. Lets say, we want to print a character, 'A', into location x/y (5,5). Knowing that video memory begins at 0xb8000, and is linear, we can use the formula to convert this x/y location to an absolute address:

```
address = x + (y * screen width)
address = 5 + (5 * 80)
address = 5 + 400
address = 405

This means, from the start of video memory, location 5,5 is 405 bytes away.
So, add this to the base address of video memory:

0xB8000 + 405 = 0xB8195
```

So... by writing character 'A' to 0xB8195, we effectivly write to x/y location (5,5). Cool, huh?

Knowing this, lets first provide a way to store the current location at where we are on screen. This is so that we can act like the BIOS, so that the rest of the program does not need to:

```
_CurX db 0                                  ; current x/y location
_CurY db 0

%define         VIDMEM  0xB8000             ; video memory
%define         COLS    80                  ; width and height of screen
%define         LINES   25
%define         CHAR_ATTRIB 14              ; character attribute (White text on black background)
```

Remember that we are in Mode 7. This mode has 80 colums of characters per row, and 25 lines. And, of course, video memory begins at 0xB8000. But wait! What is the character attribute?

Text Mode 7 actually uses **two** bytes per character, not one. **Remember this!** The first byte represents the actual character, and the second byte is a ...wait for it... attribute byte! Because of this, when writing a character to screen in Mode 7, you will need to write **two** bytes, not one.

The attribute byte provides a way of supplying color, as well as certain attributes, such as blinking. The values can be...

- **0** - Black
- **1** - Blue
- **2** - Green
- **3** - Cyan
- **4** - Red
- **5** - Magenta
- **6** - Brown

- **7** - Light Gray
- **8** - Dark Gray
- **9** - Light Blue
- **10** - Light Green
- **11** - Light Cyan
- **12** - Light Red
- **13** - Light Magenta
- **14** - Light Brown
- **15** - White

The attribute byte is a byte that defines certain attributes, and defining both foreground and background colors. The byte follows the format:

- **Bits 0 - 2:** Foreground color
  - Bit 0: Red
  - Bit 1: Green
  - Bit 2: Blue
- **Bit 3:** Foreground Intensity
- **Bits 4 - 6:** Background color
  - Bit 4: Red
  - Bit 5: Green
  - Bit 6: Blue
- **Bit 7:** Blinking or background intensity

Okay, now that we have everything set up, lets print a character!

Setting up

Printing characters is a little complex because we have to track where we are, both in current x/y location and when writing to video memory. We also need to track certain characters, such as the newline character, and to watch for the end of line. And yet, we still need to update the hardware cursor to this position as well.

Putch32 is the pmode routine that will display a character in stage 2. Don't worry, we will rewrite these routines for the Kernel using C. By showing how it's done in assembly, we can compare assembly language relationships with C. More on this later.

Anyways, heres the startup code:

```
bits 32

%define         VIDMEM  0xB8000                 ; video memory
%define         COLS    80                      ; width and height of screen
%define         LINES   25
%define         CHAR_ATTRIB 14                  ; character attribute (White text on black background)

_CurX db 0                                      ; current x/y location
_CurY db 0

;***********************************************;
;       Putch32 ()
;               - Prints a character to screen
;       BL => Character to print
;***********************************************;

Putch32:

        pusha                           ; save registers
        mov     edi, VIDMEM             ; get pointer to video memory
```

Okay, we have some basic definitions. _CurX and _CurY will contain the current x/y location to write the character to. By incrementing _CurX, we effectivly go to the next character in the line. Also note that EDI contains the base address of video memory. Now, by writing to video memory [EDI], we an display characters on screen do to the current video memory map.

Before displaying characters, we have to find out where to display it. To do this, just write it to the current x/y location (_CurX and _CurY). This is not quite simple though.

As you remember, video memory is linear, so we have to convert the x/y location into linear memory. Remember our formula **x + y * screen width**. This can be easily computed. However, **remember that every character is two bytes in size. Remember that _CurX, _CurY, COLS, LINES, are based off characters, not bytes.** i.e., COLS=80 characters. Because there are two bytes per character, we have to compare with 80*2. Simple, huh?

This makes things a little more complex, but not that hard:

```
        ;--------------------------------;
        ;   Get current position         ;
        ;--------------------------------;

        xor     eax, eax                ; clear eax

                ;--------------------------------
                ; Remember: currentPos = x + y * COLS! x and y are in _CurX and _CurY.
                ; Because there are two bytes per character, COLS=number of characters in a line.
                ; We have to multiply this by 2 to get number of bytes per line. This is the screen width,
                ; so multiply screen with * _CurY to get current line
                ;--------------------------------

                mov     ecx, COLS*2             ; Mode 7 has 2 bytes per char, so its COLS*2 bytes per line
                mov     al, byte [_CurY]        ; get y pos
                mul     ecx                     ; multiply y*COLS
                push    eax                     ; save eax--the multiplication
```

This is the first part of the formula: **y * screen width (in bytes)**, or _CurY * (COLS*bytes per character). We store it on the stack so that we could finish the formula.

```
                ;--------------------------------
                ; Now y * screen width is in eax. Now, just add _CurX. But, again remember that _CurX is relative
                ; to the current character count, not byte count. Because there are two bytes per character, we
                ; have to multiply _CurX by 2 first, then add it to our screen width * y.
                ;--------------------------------

                mov     al, byte [_CurX]        ; multiply _CurX by 2 because it is 2 bytes per char
                mov     cl, 2
                mul     cl
                pop     ecx                     ; pop y*COLS result
                add     eax, ecx
```

Okay then! Notice that we multiply _CurX by 2 to get the current byte location. Then, we pop the result of y * COLS and add it to the x position--completing our x+y*COLS formula.

Yey! Okay, now EAX contains the offset byte to print our character to, so lets add it to EDI--which holds the base address of video memory:

```
                ;--------------------------------
                ; Now eax contains the offset address to draw the character at, so just add it to the base address
                ; of video memory (Stored in edi)
                ;--------------------------------

                xor     ecx, ecx
                add     edi, eax                ; add it to the base address
```

Okay, now EDI contains the exact byte to write to. BL contains the character to write. If the character is a newline character, we will want to move to the next row. Else, just print the character:

```
        ;--------------------------------;
        ;   Watch for new line           ;
        ;--------------------------------;

        cmp     bl, 0x0A                ; is it a newline character?
        je      .Row                    ; yep--go to next row

        ;--------------------------------;
        ;   Print a character            ;
        ;--------------------------------;

        mov     dl, bl                  ; Get character
        mov     dh, CHAR_ATTRIB         ; the character attribute
        mov     word [edi], dx          ; write to video display

        ;--------------------------------;
        ;   Update next position         ;
        ;--------------------------------;

        inc     byte [_CurX]            ; go to next character
        cmp     [_CurX], COLS           ; are we at the end of the line?
        je      .Row                    ; yep-go to next row
        jmp     .done                   ; nope, bail out
```

Okay then! Pretty easy, huh? Oh right..to go to the next row is easy:

```
        ;--------------------------------;
        ;   Go to next row               ;
        ;--------------------------------;

.Row:
        mov     byte [_CurX], 0         ; go back to col 0
        inc     byte [_CurY]            ; go to next row

        ;--------------------------------;
        ;   Restore registers & return   ;
        ;--------------------------------;

.done:
        popa                            ; restore registers and return
        ret
```

Working with strings

Okay, so we can print a character. Yippe. I am very excited to see a single character. Yeah, I don't think so :)

To print actual information, we will need a way to print full strings. Because we already have a routine that tracks current position (and updates it), and prints the characters, all we need to do to print a string is a simple loop.

```
Puts32:

        ;--------------------------------;
        ;   Store registers              ;
        ;--------------------------------;
```

```
        pusha                           ; save registers
        push    ebx                     ; copy the string address
        pop     edi
```

Okay, Heres our Puts32() function. It takes one parameter: EBX, which contains the address of a null terminated string to print. Because out Putch32() function requires that BL store the character to print, we need to save a copy of EBX, so we do it here.

Now, we loop:

```
.loop:

        ;--------------------------------;
        ;   Get character                ;
        ;--------------------------------;

        mov     bl, byte [edi]          ; get next character
        cmp     bl, 0                   ; is it 0 (Null terminator)?
        je      .done                   ; yep-bail out
```

We use EDI to derefrence the string to get the current character to display. Note the test for the null terminator. If found, we bail out. Now, to display the character... The most complex code you will ever see:

```
        ;--------------------------------;
        ;   Print the character          ;
        ;--------------------------------;

        call    Putch32                 ; Nope-print it out
```

...Or not :)

All we need to do now is to go to the next character, and loop:

```
        ;--------------------------------;
        ;   Go to next character         ;
        ;--------------------------------;

.Next:
        inc     edi                     ; go to next character
        jmp     .loop

.done:
        ;--------------------------------;
        ;   Update hardware cursor       ;
        ;--------------------------------;

        ; Its more efficiant to update the cursor after displaying
        ; the complete string because direct VGA is slow

        mov     bh, byte [_CurY]        ; get current position
        mov     bl, byte [_CurX]
        call    MovCur                  ; update cursor

        popa                            ; restore registers, and return
        ret
```

Voila! We got ourselves a way to print strings in 32 bit protected mode. Not to hard, is it? Oh wait.. What is MovCur for? We will look at that next.

Updating the hardware cursor

Okay, so we can print characters and strings out now. You might notice something though: the cursor does not move! Because of this, it just stays no matter what we do. This cursor is a simple underline that the BIOS uses to indicate the current position when printing text.

This cursor is handled by the hardware. The **CRT Microcontroller**, in fact. So, we have to know some basic vga programming in order to move this cursor.

CRT Microcontroller

Warning for CRT users

While I incourage practicing and trying new things, please remember that, in an OS envirement, you are working directly with the hardware, and have direct control over everything.

CRT Monitor failures are violent in nature, and can explode and produce sharp glass fragments to fly at high speeds. It is possible to change frequency settings greater then the devices can handle. **This may increase the chanches of a device or microchip to malfunction, producing unpredictable or desasterous results.**

Because of this, if you, the reader, like experementing with the code, I recommend testing all experemental code in an emulator to its fullest first, before attempting real hardware.

I will not explain everything reguarding video programming yet until we talk about Video Drivers. We will look at everything in detail then, cool?

Anywhoo...On to the CRT Controller!

Port Mapping

The CRT Controller uses a single **Data Register** which is mapped to **port 0x3D5**. Remember the Port table from Tutorial 7? The CRT Controller uses a special register - an **Index Register**, to determin the type of data in the Data Register is.

So, in order to give data to the CRT Controller, **we have too write two values. One to the Index Register (Containing the type of data we are writing), and one to the Data Register (Containing the data).** Not too hard :)

The Index Register is mapped to ports 0x3D5 or 0x3B5.
The Data Register is mapped to ports 0x3D4 or 0x3B4.

There are more registers then these two (Such as the Misc. Output Register), but we will focus on these two for now.

Index Register Mapping

By default, the indices for the Index Register are mapped to the following:

CRT Microcontroller - Index Register	
Index Offset	CRT Controller Register
0x0	Horizontal Total
0x1	Horizontal Display Enable End
0x2	Start Horizontal Blanking
0x3	End Horizontal Blanking
0x4	Start Horizontal Retrace Pulse
0x5	End Horizontal Retrace
0x6	Vertical Total
0x7	Overflow
0x8	Preset Row Scan
0x9	Maximum Scan Line
0xA	Cursor Start
0xB	Cursor End
0xC	Start Address High
0xD	Start Address Low
0xE	Cursor Location High
0xF	Cursor Location Low
0x10	Vertical Retrace Start
0x11	Vertical Retrace End
0x12	Vertical Display Enable End
0x13	Offset
0x14	Underline Location
0x15	Start Vertical Blanking
0x16	End Vertical Blanking
0x17	CRT Mode Control
0x18	Line Compare

By writing an index offset value into the index Register, it indicates what register the Data Register points to (That is, what it refrences.)

Most of what is in the above table we don't need to worry about right now. However, look at indices 0xE and 0xF for a moment:

- **0x0E:** Cursor Location High Byte
- **0x0F:** Cursor Location Low Byte

Yippe! These indices refer to the current offset location of the hardware cursor. This offset is just an x/y location (as a linear location - remember the formula **x + y * screen width!**), split into its high and low bytes.

Moving the hardware cursor

Okay, first remember that the indices for the cursor are 0x0E and 0x0F, which we have to first put into the Index Register at port 0x3D4:

```
        mov     al, 0x0f
        mov     dx, 0x03D4
        out     dx, al
```

This puts index 0x0F (the cursor low byte address) into the index register. Now, this means the value put into the Data Register (Port 0x3d5) indicates the low byte of the cursor location:

```
        mov     al, bl                  ; al contains the low byte address
        mov     dx, 0x03D5
        out     dx, al                  ; low byte
```

This sets the new low byte location for the cursor! Cool, huh? Setting the high byte is exactally the same, except we have to set the index to 0x0E, which is, again, the high byte index.

Here is the complete routine:

```
;*********************************************;
;       MoveCur ()
;               - Update hardware cursor
;       parm/ bh = Y pos
;       parm/ bl = x pos
;*********************************************;
```

```
bits 32

MovCur:

	pusha				; save registers (aren't you getting tired of this comment?)

	;-------------------------------;
	;   Get current position        ;
	;-------------------------------;

	; Here, _CurX and _CurY are relitave to the current position on screen, not in memory.
	; That is, we don't need to worry about the byte alignment we do when displaying characters,
	; so just follow the forumla: location = _CurX + _CurY * COLS

	xor     eax, eax
	mov     ecx, COLS
	mov     al, bh                  ; get y pos
	mul     ecx                     ; multiply y*COLS
	add     al, bl                  ; Now add x
	mov     ebx, eax

	;--------------------------------------;
	;   Set low byte index to VGA register ;
	;--------------------------------------;

	mov     al, 0x0f                ; Cursor location low byte index
	mov     dx, 0x03D4              ; Write it to the CRT index register
	out     dx, al

	mov     al, bl                  ; The current location is in EBX. BL contains the low byte, BH high byte
	mov     dx, 0x03D5              ; Write it to the data register
	out     dx, al                  ; low byte

	;---------------------------------------;
	;   Set high byte index to VGA register ;
	;---------------------------------------;

	xor     eax, eax

	mov     al, 0x0e                ; Cursor location high byte index
	mov     dx, 0x03D4              ; Write to the CRT index register
	out     dx, al

	mov     al, bh                  ; the current location is in EBX. BL contains low byte, BH high byte
	mov     dx, 0x03D5              ; Write it to the data register
	out     dx, al                  ; high byte

	popa
	ret
```

That was easy, huh?

Next up: Clearing the screen!

Clearing the screen

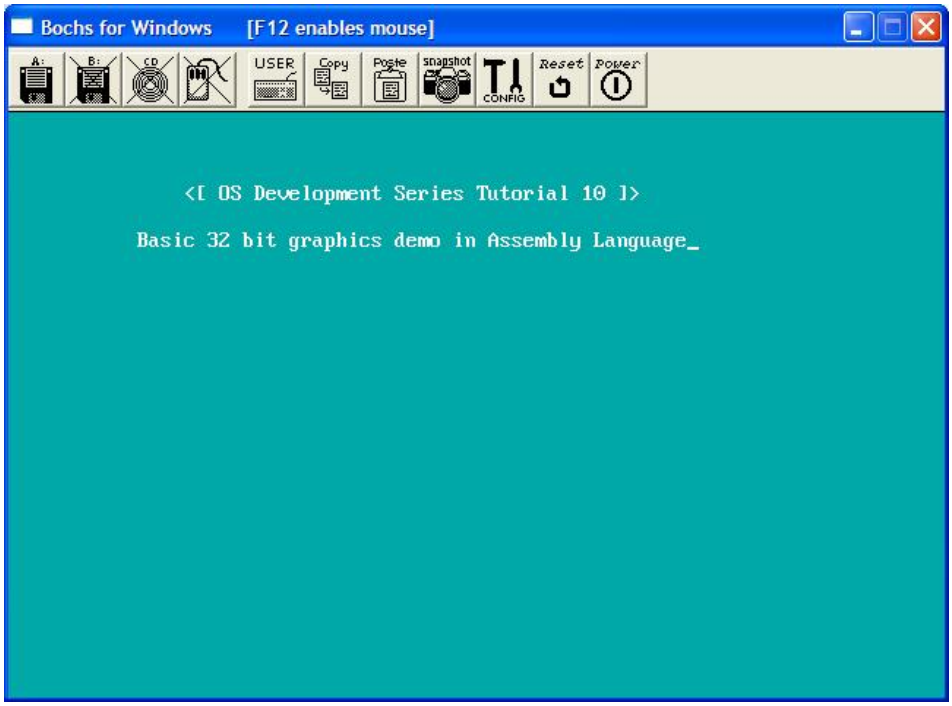
Because we already have a way to display text, just loop, and reset the current position to 0! This is surprisingly simple...

```
;************************************************;
;	ClrScr32 ()
;		- Clears screen
;************************************************;

bits 32

ClrScr32:

	pusha
	cld
	mov     edi, VIDMEM
	mov     cx, 2000
	mov     ah, CHAR_ATTRIB
	mov     al, ' '
	rep     stosw

	mov     byte [_CurX], 0
	mov     byte [_CurY], 0
	popa
	ret
```

Easy, huh?

Okay, so we have a way to print text, which also updates the hardware cursor, and clear the screen. If we wanted to, we can expand this stage 2 loader to include a small menu and advanced options when giving control to the Kernel. More on this later...

Demo



I decided to create a little demo to demenstrate everything in this tutorial. The next tutorial builds directly on this code.

This tutorial uses everything we talked about in this tutorial. It sets the foreground and background colors based on the character attribute byte. And, because of our ClrScr32() routine, effectivly clears the screen to that background color. Cool, huh?

You can download the demo Here.

Conclusion

I was pretty stumped on how to go about these next tutorials. I believe (Hope!) splitting it in two parts was a good solution.

We have went over alot of stuff here, more specifically graphics concepts. We talked about basic VGA concepts, printing characters, strings, clearing the screen, and updating the hardware cursor. By changing the **attribute byte** of the text we print out, we could easily print characters out in all shorts of colors! You can even get a new background by changing the color in the attribute byte, and calling our **ClrScr32 ()** function! Cool, dont you think? It certanly beets the boring black and white... :)

The next tutorial finishes Stage 2, and loads and executes a basic pure binary 32 bit Kernel image at 1 MB. Don't worry--When we get into the Kernel section of this series, we will change the way the Kernel is built, and modify how it is loaded. This will allow us to load the Kernel as an object format--allowing it to import or export symbols, and mix it in with C. I cannot wait!

The next tutorial is not a tutorial in a sense of learning new things. Instead, it covers all of the code that has already been explained. This code, however, is modified for better code layout, and provide the interface (and seperation) between a basic FileSystem (FAT12) Driver and a Floppy Driver. Nontheless, it is the closing tutorial for Stage 2.

We will go back to Stage 2 a bit later, as Stage 2 can be modified to provide more options, or even to support **Multibooting**, and **Boot Options**. We shall see... ;)

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 9 Home Chapter 11

Operating Systems Development Series

Operating Systems Development - Prepare for the Kernel part 2

by Mike, 2008

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

Introduction

Welcome! :)

In the previous tutorial, we talked about basic VGA programming in protected mode, and even built a 1337 demo too!

This is the tutorial you have been waiting for. It builds directly on the all of the previus code, and loads our Kernel at te 1 MB mark, and executes our Kernel.

The Kernel is the most important part of our OS. The Kernel...We have talked a little about this mysterous foe before, havn't we? We will talk about the Kernel alot more in the next few tutorials, including design, structure, and development.

Right now, we already have everything set up... It's time to load the Kernel and say good bye to Stage 2!

Note: This tutorial requires a basic understanding of the Bootloaders 3 and 4 tutorials. We cover everything in detail here, but all of the concepts are explained in depth in the Bootloaders 3 and 4 Tutorials. If you have not read those tutorials, Please look at those tutorials first.

OS Development Series Tutorial 5: Bootloaders 3

OS Development Series Tutorial 6: Bootloaders 4

If you have read them, this tutorial should not be that hard.

Ready?

A Basic Kernel Stub

This is the Kernel we will load:

```
; We are still pure binary. We will fix this in the next few tutorials :)

org     0x10000                 ; Kernel starts at 1 MB
bits    32                      ; 32 bit code

jmp     Stage3                  ; jump to stage 3

%include "stdio.inc"            ; Our stdio.inc file we developed from the previous tutorial

msg db  0x0A, 0x0A, "Welcome to Kernel Land!!", 0x0A, 0

Stage3:

        ;------------------------------;
        ;   Set registers              ;
        ;------------------------------;

        mov             ax, 0x10                ; set data segments to data selector (0x10)
        mov             ds, ax
        mov             ss, ax
        mov             es, ax
        mov             esp, 90000h             ; stack begins from 90000h

        ;--------------------------------------;
        ;   Clear screen and print success     ;
        ;--------------------------------------;

        call            ClrScr32
        mov             ebx, msg
        call            Puts32

        ;--------------------------------------;
        ;   Stop execution                     ;
        ;--------------------------------------;

        cli
        hlt
```

Okay, there is nothing much here. We will build on this program heavily in the next section.

Notice that it is all 32 bit. Sweet, huh? We are going to be out of the 16 bit world completely here.

For now, we just hault the system when we get to the Kernel.

Please note that we will not be using this file probably at all in the rest of the series. Rather, we will be using a 32 bit C++ compilier. After we load the kernel image in memory, we can parse the file in memory for the kernel entry routine and call the C main() routine directly from our 2nd stage boot loader. Cool, huh? In other words, we will go from our 2nd stage boot loader directly into the C++ world without any stub file or program. However, we need a starting point. Because of this, we will use a basic stub file in this tutorial to help test and demenstrate it working.

In the next few tutorials we will be getting our compiliers up and working and use that instead. But now we are getting ahead of ourselves here ;)

The floppy interface

Yey! Its time to finish off stage 2! In oder to load the Kernel we need to traverse FAT12 again. But before that, we have to get sectors off disk.

This code is EXACTALLY the same from our bootloader, and uses the BIOS INT 0x13 to load sectors off disk.

Because this tutorial is also a complete review, lets break each routine into sections and describe exactally what is going on.

Reading a sector - BIOS INT 0x13

We talked about everything requarding loading sectors in our **Bootloaders 3**. Looking back at the tutorial, remember that we can use the **BIOS Interrupt 0x13 function 2** to read a sector. Okay, then. The problem here is that **We have to load sectors before going into protected mode.** If we attempt to call a BIOS interrupt from protected mode, the processor will triple fault, remember?

Anyways, what was the interrupt? Right....

INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY
AH = 0x02
AL = Number of sectors to read
CH = Low eight bits of cylinder number
CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only
DH = Head Number
DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to

Returns:
AH = Status Code
AL = Number of sectors read
CF = set if failure, cleared is successfull

This is not THAT hard. Remember from the Bootloaders tutorial though. That is, we need to keep track of the sector, track, and head number, and insure we don't load attempt to load a sector beyond the track. That is, **Remember that there are 18 sectors per track? Setting the sector number greater then 18 will cause the controller to fail, and processor to triple fault.**

Okay...18 sectors per track. Remember that each sector if 512 bytes. Also, remember that there are 80 tracks per side.

Okay then! All of this information... Sectors per track, the number of tracks, number of heads, the size of a sector, completely depend on the disk itself. Remember that a sector does not NEED to be 512 bytes?

We describe everything in the OEM Parameter Block:

```
bpbOEM                  db "My OS   "
bpbBytesPerSector:      DW 512
bpbSectorsPerCluster:   DB 1
bpbReservedSectors:     DW 1
bpbNumberOfFATs:        DB 2
bpbRootEntries:         DW 224
bpbTotalSectors:        DW 2880
bpbMedia:               DB 0xf0  ;; 0xF1
bpbSectorsPerFAT:       DW 9
bpbSectorsPerTrack:     DW 18
bpbHeadsPerCylinder:    DW 2
bpbHiddenSectors:       DD 0
bpbTotalSectorsBig:     DD 0
bsDriveNumber:          DB 0
bsUnused:               DB 0
bsExtBootSignature:     DB 0x29
bsSerialNumber:         DD 0xa0a1a2a3
bsVolumeLabel:          DB "MOS FLOPPY "
bsFileSystem:           DB "FAT12   "
```

This should look familiar! Each member has been described in Tutorial 5--Please see that tutorial for a full detailed explination of everything here.

Now, all we need to have is a method so that we can load any number of sectors from disk to some location in memory. We immediately run into a problem though. Okay--**We know what sector we want to load**. However, **BIOS INT 0x13 does not**

work with sectors. Okay, it does--but it also works with cylinders (Remember that a cylinder is just a head?) and tracks.

So what does this have to do with anything? Imagine if we want to load sector 20. We cannot directly use this number, because **there are only 18 sectors per track**. Attempting to read from the 20th sector on the current track will cause the floppy controller to fail, and processor to triple fault, as that sector does not exist. **In order to read the 20'th sector, we have to read Track 2 Sector 2, Head 0** We will verify this later.

What this means as that, if we want to specify a sector to load, we need to convert our linear sector number into the exact cylinder, track, and sector location on disk.

Wait for it...Aha! Remember our **CHS to LBA** conversition routines?

Converting LBA to CHS

This should sound familiar, doesn't it? **Linear Block Addressing (LBA)** simply represents an indexed location on disk. The first block being 0, the second block being 1. In other words, LBA simply represents the sector number, beginning with 0, where each "block" is a single "sector".

Anywhoo...We have to find a way to convert this sector number (LBA) to the exact cylinder/head/secor location on disk. **Remember this from Bootloaders 4 tutorial?**

Some of our readers exclaimed this code was fairly tricky--and I am to admit it is. So, I am going to excplain it in detail here.

First, lets look at the forumlas again:

```
absolute sector         =       (logical sector / sectors per track) + 1
      absolute head     =       (logical sector / sectors per track) MOD number of heads
      absolute track    =        logical sector / (sectors per track * number of heads)
```

Okay! This is pretty easy, huh? The "logical sector" is the actual sector number we want. Note that the **logical sector / sectors per track** is inside of all of the above equations.

Because this division is inside of all of these equations, we can store it's result and use it for the other two expressions.

Lets put this into an example. We already said the 20th sector should be Track 2, Sector 2, remember? Lets try to put this formula to the test then:

```
absolute sector         =       (logical sector / sectors per track) + 1

       2.11111111111111111111111111111111       =       20 / 18 (sectors per track) + 1
```

We only keep the absolute number (2)--Aha! Sector 2! Note that we need to add 1 here because LBA addressing begins from 0. Remember that the basic formula "logical sector / sectors per track" is in ALL of these formulas. It is simply 1.11111111111111111111111111111111 in this example (Note in the above formula, we added 1 more). Because we are working with whole numbers, this is simply 1.

```
absolute head           =       (logical sector / sectors per track) MOD number of heads

                                (1) MOD Number of heads (2)
                                = Head 1
```

Remember from the OEM Block that we specified 2 heads per cylinder. So far, this indicates sector 2 on Head 1. Great--but what track are we on?

```
absolute track  =        logical sector / (sectors per track * number of heads)

                                (1) * Number of heads (2)
                                = Track 2
```

Notice that this is the exact same formula as above. The ONLY difference is that simple operation.

Anywhoo... following the formula we have: **Logical Sector 20 is on Sector 2 Track 2 Head 0**. Compare this with what we originally said in the previous section, and notice how this forumla works ;)

Okay, so now lets try to apply these formulas in the code:

LBACHS Explanation: Detail

Okay, this routine takes one parameter: AX, which contains the logical sector to convert into CHS. Note the formula **(logical sector / sectors per track)** is part of all three formulas. Rather then recalculating this over and over, it is more efficiant to just calculate it **once**, and use that result in all of the other calculations... This is how this routine works.

```
LBACHS:
          xor     dx, dx                          ; prepare dx:ax for operation
          div     WORD [bpbSectorsPerTrack]       ; calculate
```

Now AX contains the logical sector / sectors per track operation.

Begin with sector 1 (Remember the + 1 in logical sector / sectors per track ?)

```
        inc     dl                                  ; adjust for sector 0
        mov     BYTE [absoluteSector], dl
```

Clear DX. AX still contains the result of logical sector / sectors per track

```
        xor     dx, dx                              ; prepare dx:ax for operation
```

Now for the formulas...

absolute head = (logical sector / sectors per track) MOD number of heads

absolute track = logical sector / (sectors per track * number of heads)

The multiplication results into a **division** by the number of heads. So the only difference between these two is the operation--one is division, and one is the remainder of that division (The Modulus).

Okay, lessee...What instruction can we use that could return both the remainder (MOD) and division result? DIV!

Remember that (logical sector / sectors per track) is still in AX, so all we need to do is divide by number of heads per cylinder...

```
        div     WORD [bpbHeadsPerCylinder]          ; calculate
```

The equtions for absolute head and absolute track are very simular. The only actual difference is the operation. **This simple DIV instruction sets both DX and AX. AX Now stores the DIVISION of HeadsPerCylinder; DX now contains the REMAINDER (Modolus) of the same operation)**

```
        mov     BYTE [absoluteHead], dl
        mov     BYTE [absoluteTrack], al
        ret
```

I hope this clears things up a bit. If not, please let me know ;)

Converting CHS to LBA

This is alot more simpler:

```
ClusterLBA:
        ; LBA   =       (cluster - 2 ) * sectors per cluster

          sub     ax, 0x0002                        ; subtract 2 from cluster number
          xor     cx, cx
          mov     cl, BYTE [bpbSectorsPerCluster]   ; get sectors per cluster
          mul     cx                                ; multply
```

Reading in sectors

Okay, so now we have everything to read in sectors. This code is also exactally the same from the bootloader.

```
;*********************************************;
; Reads a series of sectors
; CX=>Number of sectors to read
; AX=>Starting sector
; ES:BX=>Buffer to read to
;*********************************************;

ReadSectors:
     .MAIN
          mov     di, 0x0005                        ; five retries for error
```

Okay, here we attempt to read the sectors 5 times.

```
     .SECTORLOOP
          push    ax
          push    bx
          push    cx
          call    LBACHS                            ; convert starting sector to CHS
```

We store the registers on the stack. The starting sector is a linear sector number (Stored in AX). Because we are using BIOS INT 0x13, We need to convert this to CHS before reading from the disk. So, we use our LBA to CHS coversition routine. Now, **absoluteTrack** contains the track number, **absoluteSector** contains the sector within the track, and **absoluteHead** contains the head number. All of this was set by our LBA to CHA conversition routine, remember?

```
        mov     ah, 0x02                        ; BIOS read sector
        mov     al, 0x01                        ; read one sector
        mov     ch, BYTE [absoluteTrack]        ; track
        mov     cl, BYTE [absoluteSector]       ; sector
        mov     dh, BYTE [absoluteHead]         ; head
        mov     dl, BYTE [bsDriveNumber]        ; drive
        int     0x13                            ; invoke BIOS
```

Now we set up to read a sector, and envoke the BIOS to read it. For simplicity, lets take another look at the BIOS INT 0x13 routine that we are executing:

INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY
AH = 0x02
AL = Number of sectors to read
CH = Low eight bits of cylinder number
CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only
DH = Head Number
DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to

Compare this with how we execute the code above--fairly simple, huh?

Remember that the buffer to write to is in ES:BX, which INT 0x13 refrences as the buffer. We passed ES:BX into this routine, so that is the location to load the sectors to.

```
        jnc     .SUCCESS                        ; test for read error
        xor     ax, ax                          ; BIOS reset disk
        int     0x13                            ; invoke BIOS
        dec     di                              ; decrement error counter
        pop     cx
        pop     bx
        pop     ax
        jnz     .SECTORLOOP                     ; attempt to read again
```

The BIOS INT 0x13 function 2 sets the Carry Flag (CF) is there is an error. If there is an error, decrement the counter (Remember we set up the loop to try 5 times?), and then try again!

If all 5 attemps failed (CX=0, Zero flag set), then we fall down to the INT 0x18 instruction:

```
        int     0x18
```

...Which reboots the computer.

If the Carry Flag was NOT set (CF=0), then the **jnz** instruction jumps here, as it indicates that there was no error. The sector was read successfully.

```
    .SUCCESS
        pop     cx
        pop     bx
        pop     ax
        add     bx, WORD [bpbBytesPerSector]    ; queue next buffer
        inc     ax                              ; queue next sector
        loop    .MAIN                           ; read next sector
        ret
```

Now, just restore the registers, and go to the next sector. Not to hard :) Note that, because ES:BX contains the address to load the sectors to, we need to increment BX by the bytes per sector to go to the next sector.

AX contained the **starting sector** to read from, so we need to increment that too.

I guess thats all for now. Please refrence **Bootloaders 4** for a full explaination of this routine.

Floppy16.inc

In the example demo, all of the floppy access routines are inside of **Floppy16.inc**.

FAT12 Interface

Yey--We can load sectors. Woohoo... :(As you know, we cannot really do much with that. What we need to do next is create a basic definition of a "file" and what a "file" is. We do this by means of a **Filesystem**.

Filesystems can get quite complex. Please refrence **Bootloaders 4** while I explain this code to fully understand how this code works.

Constants

During parsing Fat12, we will be needing a location to load the root directory table and the FAT table. To make things somewhat easier, lets hide these locations behind constants:

```
%define ROOT_OFFSET 0x2e00
%define FAT_SEG 0x2c0
%define ROOT_SEG 0x2e0
```

We will be loading our root directory table to 0x2e00 and our FAT to 0x2c00. FAT_SEG and ROOT_SEG are used for loading into segment registers.

Traversing FAT12

As you know, some OS code can simply get ugly. Filesystem code, in my opinion, is one of them. This is one of the reasons why I decided to go over this code in this review-like tutorial. The FAT12 code is basically the same as the bootloaders, but I decided to modify it to decrease dependencies with the main program. Because of this, I decided to describe it in detail here.

Please note, I will not be going over FAT12 in detail here. Please see the **Bootloaders 4** tutorial for complete details.

Anywhoo, as you know, in order to traverse FAT12 the first thing we need to load is the **Root Directory Table**, so lets look at that first.

Loading the Root Directory Table

Disk structure:

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.

Remember that the Root Directory Table is located right after the FAT's and Reserved sectors?

In loading the root directory table, we need to find a location in memory that we do not currently need and copy it there. For now, I chose 0x7E00 (Real mode: 0x7E0:0). This is right above our bootloader, which is **still in memory** because we have never overwritten it.

There is an important concept here. Notice that we have to load everything at absolute memory locations. This is very bad, as we have to physically keep track of where things are located. This is where a **Low level memory manager** comes into play. More later...

```
;*****************************************
; LoadRoot ()
;       - Load Root Directory Table
;*****************************************

LoadRoot:

       pusha                                    ; store registers
       push    es
```

We first store the current state of the registers. Not doing so will effect the rest of the program that uses it, which is very bad.

Now we get the size of the root directory table, so that we know the number of sectors to load.

Remember from **Bootloaders 4**: Each entry is 32 bytes in size. When we add a new file in a FAT12 formatted disk, Windows automatically appends to the root directory for us, and adds to the **bpbRootEntries** byte offset varable of the **OEM Parameter Block**

See...Windows is nice :)

So...lessee, knowing each entry is 32 bytes in size, **multiplying 32 bytes by the number of root directories will tell us how many bytes there are in the Root Directory Table**. Simple enough, but we need the number of **sectors**--so we need to divide this result by the number of sectors:

```
    ; compute size of root directory and store in "cx"

       xor     cx, cx                           ; clear registers
       xor     dx, dx
       mov     ax, 32                           ; 32 byte directory entry
       mul     WORD [bpbRootEntries]            ; total size of directory
       div     WORD [bpbBytesPerSector]         ; sectors used by directory
       xchg    ax, cx                           ; move into AX
```

OKAY, so now AX=number of sectors the root directory takes. Now, we have to find the starting location.

Remember from **Bootloaders 4: The Root Directory table is Right after both FAT's and reserved sectors on the disk.** Please look at the above disk structure table to see where the root directory table is located.

So...All we need to do is get the amount of sectors for the FAT's, and add that to the reserved sectors to get the exact location on disk:

```
    ; compute location of root directory and store in "ax"

       mov     al, byte [bpbNumberOfFATs]       ; number of FATs
       mul     word [bpbSectorsPerFAT]          ; sectors used by FATs
```

```
        add     ax, word [bpbReservedSectors]           ; adjust for bootsector
        mov     word [datasector], ax                   ; base of root directory
        add     word [datasector], cx
```

Now that we have the number of sectors to read in, and the exact starting sector, lets read it in!

```
    ; read root directory

        push    word ROOT_SEG
        pop     es
        mov     bx, 0x0                                 ; copy root dir
        call    ReadSectors                             ; read in directory table
        pop     es
        popa                                            ; restore registers and return
        ret
```

Notice that we set the seg:offset location to read into ROOT_SEG:0.

Next up, loading the FAT!

Loading the FAT

Okay...Remember from **Bootloaders 4**, we talked about the disk structure of a FAT12 formatted disk. Going Back in Time(tm), lets take another look:

Disk structure:

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.

Remember that there are either one or two FATs? Also notice that they are **right after** the reserved sectors on disk. **This should look familar!**

```
;*****************************************
; LoadFAT ()
;       - Loads FAT table
;
;       Parm/ ES:DI => Root Directory Table
;*****************************************

LoadFAT:

        pusha                           ; store registers
        push    es
```

First we need to know how many sectors to load. Look back at the disk structure again. We store the number of FATs (and the sectors per FAT) in the OEM Parameter Block. So to get the total sectors, just multiply them:

```
    ; compute size of FAT and store in "cx"

        xor     ax, ax
        mov     al, BYTE [bpbNumberOfFATs]              ; number of FATs
        mul     word [bpbSectorsPerFAT]                 ; sectors used by FATs
        mov     cx, ax
```

Now, we need to take the reserved sectors into coinsideration, as they are before the FAT...

```
    ; compute location of FAT and store in "ax"

        mov     ax, word [bpbReservedSectors]
```

Yippe! Now, CX contains the number of sectors to load, so call our routine to load the sectors!

```
    ; read FAT into memory (Overwrite our bootloader at 0x7c00)

        push    word FAT_SEG
        pop     es
        xor     bx, bx
        call  ReadSectors
        pop     es
        popa            ; restore registers and return
        ret
```

Thats all there is to it ;)

Searching for a file

In searching for a file, we need the filename to search with. Remember that DOS uses 11 byte file names followng the common 8.3 naming convention (8 byte file name, 3 character extension.) Because of the way the enteries in the Root directory is structured, **This MUST be 11 bytes--no exceptions**.

Remember the format of the Root Directory Table: The filename is stored within the **first** 11 bytes of an entry. Lets take another look at the format of each directory entry:

- **Bytes 0-7 : DOS File name (Padded with spaces)**
- **Bytes 8-10 : DOS File extension (Padded with spaces)**
- **Bytes 11 :** File attributes. This is a bit pattern:
  - **Bit 0 :** Read Only
  - **Bit 1 :** Hidden
  - **Bit 2 :** System
  - **Bit 3 :** Volume Label
  - **Bit 4 :** This is a subdirectory
  - **Bit 5 :** Archive
  - **Bit 6 :** Device (Internal use)
  - **Bit 6 :** Unused
- **Bytes 12 :** Unused
- **Bytes 13 :** Create time in ms
- **Bytes 14-15 :** Created time, using the following format:
  - **Bit 0-4 :** Seconds (0-29)
  - **Bit 5-10 :** Minutes (0-59)
  - **Bit 11-15 :** Hours (0-23)
- **Bytes 16-17 :** Created year in the following format:
  - **Bit 0-4 :** Year (0=1980; 127=2107
  - **Bit 5-8 :** Month (1=January; 12=December)
  - **Bit 9-15 :** Hours (0-23)
- **Bytes 18-19 :** Last access date (Uses same format as above)
- **Bytes 20-21 :** EA Index (Used in OS/2 and NT, dont worry about it)
- **Bytes 22-23 :** Last Modified time (See byte 14-15 for format)
- **Bytes 24-25 :** Last modified date (See bytes 16-17 for format)
- **Bytes 26-27 : First Cluster**
- **Bytes 28-32 : File Size**

All **Bolded** entrys are the important ones. We must compare the **first 11 bytes** of each entry, as they contain the filename.

Once we find a match, **We need to refrence byte 26 of the entry to get it's current cluster**. All of this should sound familiar.

Now...On to the code!

```
;*****************************************
; FindFile ()
;       - Search for filename in root table
;
; parm/ DS:SI => File name
; ret/ AX => File index number in directory table. -1 if error
;*****************************************

FindFile:

        push    cx                              ; store registers
        push    dx
        push    bx
        mov     bx, si                          ; copy filename for later
```

We first store the current register states. We need to use SI, so we need to save the current filename somewhere...BX, perhaps?

Remember that we need to parse the Root Directory table to find the image name. To do this, we need to check the first 11 bytes of each entry in the directory table to see if we found a match. Sounds simple, huh?

To do this, we need to know how many entries there are...

```
    ; browse root directory for binary image

        mov     cx, word [bpbRootEntries]       ; load loop counter
        mov     di, ROOT_OFFSET                 ; locate first root entry
        cld                                     ; clear direction flag
```

Okay, so CX now contains the number of entries to look in. All we need to do now is loop and compare the 11 byte character filename. Because we are using string instructions, we want to first insure the direction flag is cleared, which is what **cld** does.

DI is set to the current offset into the directory table. This is the location of the table. i.e., ES:DI points to the starting location of the table, so lets parse it!

```
.LOOP:
        push    cx
        mov     cx, 11                          ; eleven character name. Image name is in SI
        mov     si, bx                          ; image name is in BX
```

```
        push    di
        rep  cmpsb                              ; test for entry match
```

If the 11 bytes match, the file was found. Because DI contains the location of the entry within the table, we immediately jump to .Found.

If it does not match, we need to try the next entry in the table. We add **32 bytes** onto DI. (**Remember that each entry is 32 bytes?**)

```
        pop     di
        je      .Found
        pop     cx
        add     di, 32                          ; queue next directory entry
        loop    .LOOP
```

If the file was not found, restore only the registers that are still on the stack, and return -1 (error)

```
.NotFound:
        pop     bx                              ; restore registers and return
        pop     dx
        pop     cx
        mov     ax, -1                          ; set error code
        ret
```

If the file was found, restore all of the registers. AX contains the entry location within the Root Directory Table so that it can be loaded.

```
.Found:
        pop     ax                              ; return value into AX contains entry of file
        pop     bx                              ; restore registers and return
        pop     dx
        pop     cx
        ret
```

Yey! Now that we can find the file (and get it's location within the Root Directory Table), lets load it!

Loading a file

Now that everything is finally set up, it is finally time to load the file!

Most of this is pretty easy, as it calls our other routines. It is here that we loop, and insure that all of the file's clusters are loaded into memory.

```
;*****************************************
; LoadFile ()
;       - Load file
; parm/ ES:SI => File to load
; parm/ BX:BP => Buffer to load file to
; ret/ AX => -1 on error, 0 on success
; ret/ CX => Number of sectors loaded
;*****************************************

LoadFile:

        xor             ecx, ecx
        push            ecx
```

Here we just save the registers. We need to keep a copy of the buffer to write to somewhere, so we keep that on the stack as well. CX is used to keep track of how many sectors we have loaded. We store this on the stack for later.

In loading the file, we will need to first find it (Kind of obvious, don't you think? ^^) We can easily use our FindFile routine here. FindFile sets AX to -1 on error, **or the starting entry location within the Root Directory Table upon success**. We can use this index to get anything we ever wanted to know about the file.

```
.FIND_FILE:

        push    bx      ; BX=>BP points to buffer to write to; store it for later
        push    bp

        call    FindFile                        ; find our file. ES:SI contains our filename

        cmp     ax, -1                          ; check for error
        jne     .LOAD_IMAGE_PRE                 ; No error :) Load the FAT
        pop     bp                              ; Nope :( Restore registers, set error code and return
        pop     bx
        pop     ecx
```

```
        mov     ax, -1
        ret
```

Okay, so if we get here, the file was found. ES:DI contains the location of the first root entry, which was set by FindFile(), so by refrencing ES:DI we effectivly get the file's entry.

Look back at the entry description table above in the previous section. Notice that we can offset 0x1A bytes to get to byte 26 (The starting cluster number), so store it...

```
.LOAD_IMAGE_PRE:

        sub     edi, ROOT_OFFSET
        sub     eax, ROOT_OFFSET

        ; get starting cluster

        push    word ROOT_SEG
        pop     es
        mov     dx, word [es:di + 0x001A]; ES:DI points to file entry in root directory table.
        mov     word [cluster], dx      ; Refrence the table for file's first cluster
        pop     bx                      ; get location to write to so we dont screw up the stack
        pop     es
        push    bx                      ; store location for later again
        push    es
```

The above is messy, I know. Remember that AX was set to the entry number by the call to FindFile? We need to store that here, but need to keep the buffer to write to on the **top** of the stack still. This is why I played with the stack a little here :)

Anyways, next we load the FAT. This is incredably easy...

```
        call    LoadFAT                 ; Load the FAT to 0x7c00
```

OKAY then! Now that the FAT is loaded, and that we have the starting file cluster, it is time to actually read in the file's sectors.

```
.LOAD_IMAGE:

        mov     ax, WORD [cluster]              ; cluster to read
        pop     es
        pop     bx
        call    ClusterLBA                      ; convert cluster to LBA
        xor     cx, cx
        mov     cl, BYTE [bpbSectorsPerCluster] ; sectors to read
        call    ReadSectors                     ; Read in cluster

        pop     ecx                             ; increment sector count
        inc     ecx
        push    ecx

        push    bx                              ; save registers for next iteration
        push    es

        mov     ax, FAT_SEG
        mov     es, ax
        xor     bx, bx
```

This code is not that bad. Remember that, for FAT12, **each cluster is just 512 bytes**? i.e., each cluster simply represents a "sector". We first get the starting cluster/sector number. We cannot do much with just a cluster number though, as it is a **linear** number. That is, it is the sector number in **CHS** Not **LBA** format--It assumes we have the track and head information. Because our ReadSectors() requires an LBA linear sector number, **We convert this CHS to an LBA address**. Then, get the sectors per cluster, and read it in!

Note that we pop ES and BX--They were pushed on the stack from the beginning. **ES:BX points to the ES:BP buffer that was passed to this routine--It contains the buffer to load the sectors into.**

OKAY, so now that a cluster was loaded, we have to check with the FAT to determin if the end of file is reached. However, **Remember that each FAT entry is 12 bytes?** We found out from **Bootloaders 4** that there is a **pattern** when reading the FAT:

For every even cluster, take the low twelve bits; for every high cluster take the heigh twelve bits

Please see **Bootloaders 4** to see this in detail.

To determin if it is even or odd, just divide by 2:

```
    ; compute next cluster

        mov     ax, WORD [cluster]      ; identify current cluster
        mov     cx, ax                  ; copy current cluster
        mov     dx, ax                  ; copy current cluster
        shr     dx, 0x0001              ; divide by two
```

```
        add     cx, dx                  ; sum for (3/2)

        mov     bx, 0                   ; location of FAT in memory
        add     bx, cx                  ; index into FAT
        mov     dx, WORD [es:bx]        ; read two bytes from FAT
        test    ax, 0x0001
        jnz     .ODD_CLUSTER

    .EVEN_CLUSTER:
        and     dx, 0000111111111111b   ; take low twelve bits
        jmp     .DONE

    .ODD_CLUSTER:
        shr     dx, 0x0004              ; take high twelve bits

    .DONE:
        mov     WORD [cluster], dx      ; store new cluster
        cmp     dx, 0x0FF0              ; test for end of file marker (0xFF)
        jb      LOAD_IMAGE              ; No? Go on to next cluster then

    DONE:
        pop     es                      ; restore all registers
        pop     bx
        pop     ecx
        xor     ax, ax                  ; return success code
        ret
```

Thats all there is too it! Granted a little complex, but not to hard, I hope ;)

Fat12.inc

Great! All of the FAT12 code is in **Fat12.inc**.

Finishing Stage 2

Back to Stage 2 - Loading and Executing the Kernel

Now that the messy code is over, all we need to do is load our Kernel image into memory from Stage 2, and execute our kernel. The problem is: Where?

While we do want to load it to 1MB, we cannot do this directly yet. The reason is that we are still in real mode. Because of this, we will first need to load the image to a lower address first. After we switch into protected mode, we can copy our kernel to a new location. This can be 1MB, or even 3GB if paging is enabled.

```
        call    LoadRoot                ; Load root directory table

        mov     ebx, 0                  ; BX:BP points to buffer to load to
        mov     ebp, IMAGE_RMODE_BASE
        mov     Esi, ImageName          ; our file to load
        call    LoadFile                ; load our file
        MOV     dword [ImageSize], ecx  ; size of kernel
        cmp     ax, 0                   ; Test for success
        je      EnterStage3             ; yep--onto Stage 3!
        mov     si, msgFailure          ; Nope--print error
        call    Puts16
        mov     ah, 0
        int     0x16                    ; await keypress
        int     0x19                    ; warm boot computer
        cli                             ; If we get here, something really went wong
        hlt
```

Now our kernel is loaded to IMAGE_RMODE_BASE:0. ImageSize containes the number of sectors loaded (The size of the kernel).

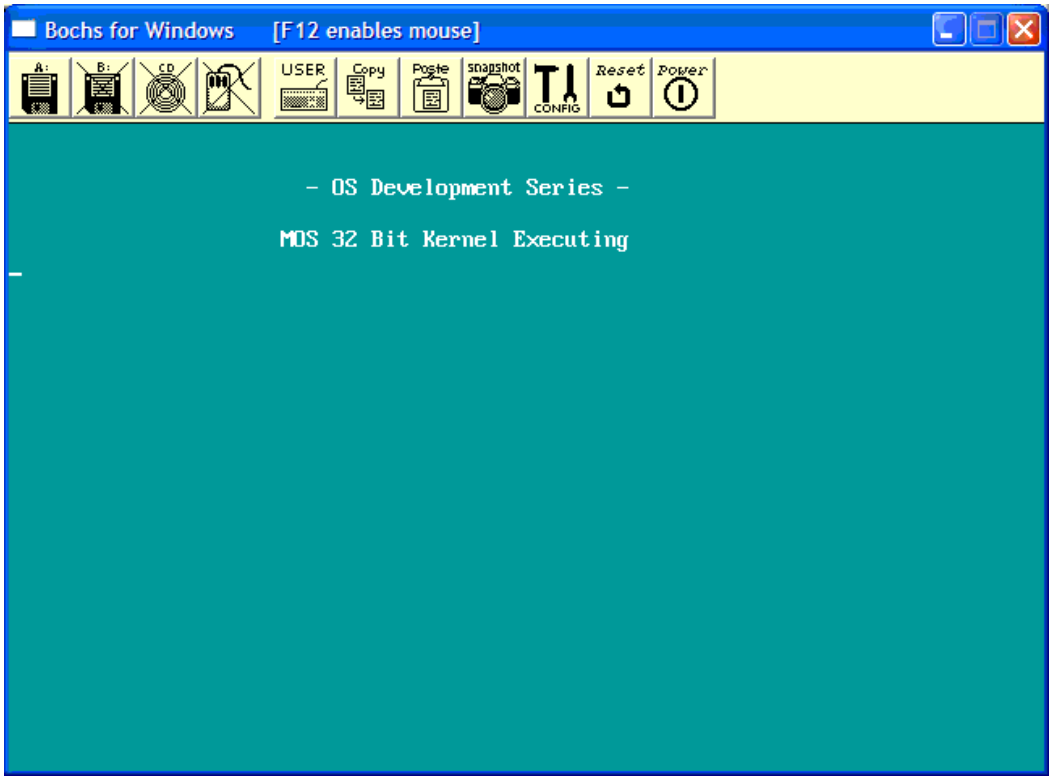
To execute inside of protected mode, all we need to do is jump or call it. Because we want our kernel at 1MB, we first need to copy it before we execute it:

```
bits 32

Stage3:

        mov     ax, DATA_DESC           ; set data segments to data selector (0x10)
        mov     ds, ax
        mov     ss, ax
        mov     es, ax
        mov     esp, 90000h             ; stack begins from 90000h

; Copy kernel to 1MB (0x10000)

CopyImage:
         mov    eax, dword [ImageSize]
         movzx  ebx, word [bpbBytesPerSector]
```

```
        mul     ebx
        mov     ebx, 4
        div     ebx
        cld
        mov     esi, IMAGE_RMODE_BASE
        mov     edi, IMAGE_PMODE_BASE
        mov     ecx, eax
        rep     movsd                   ; copy image to its protected mode address

         call   CODE_DESC:IMAGE_PMODE_BASE; execute our kernel!
```

There is a little problem here, though. This assumes the Kernel is a **pure binary file**. We cannot have this, because C does not support this. We need the Kernel to be a binary format that C supports, and we will need to parse it in order to load the Kernel using C. For now, we will keep this pure binary, but will fix this within the next few tutorials. Sound cool?

Demo



Our pure uber-1337 32 bit Kernel executing.

DOWNLOAD DEMO HERE

Conclusion

W00t! It's Kernel time!! :)

This tutorial covered alot of new code. Most of the concepts in tis tutorial we have went over before, so I hope this tutorial was not that hard ;)

In this tutorial, we covered these concepts in a new perspective, however. This may help understanding these topics a little bit more, and to see them being implimented into seperate routines.

We have developed code to load sectors off disk, and parse FAT12 to load our Kernel at whatever location we want. Cool, huh? In this Series, we are loading the Kernel at 1 MB.

With a basic full 32 bit Kernel finally loaded and executing, we can finally start focusing our attention to the most important part of any operating system -- The Kernel.

In the next few tutorials, we will cover Kernel Theory, Revolutions, and Designs. We will then start covering **Low Level C Programming**, and Low level programming with high level language concepts and theory.

There are **Alot** of freedom when programming C at Kernel Level, that most other programming fields do not allow. For example, There still is no such thing as an "Access Violation", so you still have direct control over every byte in memory. The bad news: There is also no such thing as a "standard library" either. To add more bad news, you still have to remember that you are programming a **low level envirement**, just with another abstraction layer that is C.

We will cover everything within the next few tutorials, and setting up C to work with our Kernel. I cannot wait!

See you there.... ;)

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 10 Home Chapter 12 

Operating Systems Development - Kernel: Basic Concepts Part 1

by Mike, 2009

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

Introduction

Welcome! :)

Well...We have finally made it to the most important part of any operating system: The **Kernel**.

We have heard this term alot so far throughout the series, already. The reason is because of how important it really is.

The Kernel is the Core of all operating systems. Understanding what it is and how it effects the operating system is important.

In this tutorial, We will look at what goes behind Kernels, what they are, and what they are responsible for. Understading these concepts is essental in coming up with a good design.

Ready?

Kernel: Basic Definition

In order to understand what an OS **Kernel** is, we need to first understand what a **"kernel"** is at it's basic definitions. Dictionaries define "kernel" as "core", "essental part", or even "The body of something". When applying this definition to an Operating System envirement, we can easily state that:

The Kernel is the core component of an operating system.

Okay, but what does this mean for us? What exactally is an OS Kernel, and why should we care for it?

There is no rule that states a kernel is mandatory. We can easily just load and execute programs at specific addresses without any "kernel". In fact, all of the early computer systems started this way. Some modern systems also use this. A notable example of this are the early counsole video game systems, which required **rebooting** the system in order to execute **one** of the games designed for that console.

So, what is the point of a Kernel? In a computing envirement, it is impractical to restart every time to execute a program. This will means that each program itself would need its own bootloaders and direct hardware controlling. After all, if the programs need to be executed at bootup, there would be no such thing as an operating system.

What we need is an abstraction layer to provide the capability of executing multiple programs, and manage their memory allocations. It also can provide an abstraction to the hardware, which will not be possible if each program had to start on bootup without an OS. After all, the software will be running on raw hardware.

The keyword here is **Abstraction**. Lets look closer...

The need for Kernels

The Kernel provides the primary abstraction layer to the hardware itself. The Kernel is useually at Ring 0 because of this very reason: **It has direct control over every little thing**. Because we are still at Ring 0, we already experenced this.

This is good--but what about other software? Remember that we are deveoping an **operating envirement**? Our primary goal is providing a safe, and effective envirement for applications and other software to execute. If we let all software to run at Ring 0, alongside the Kernel, there would be no need for a kernel, would there be? If there was, **The ring 0 software may conflict with the ring 0 Kernel**, causing unpredictable results. After all, they all have complete control over every byte in the system. Any software can overwrite the kernel, or any other software without any problems. Ouch.

Yet, that is only the beginning of the problems. It is impossible to have multitasking, or multiprocessing as there is no common ground to switch between programs and processes. Only one program can execute at a time.

The basic idea is that a Kernel is a neccessity. Not only do we want to **prevent** other software direct control over everything, but we want to create an **abstraction layer** for it.

Understanding where and how the Kernel fits in with the rest of the system is very important.

Abstraction Layers of Software

Software has alot of abstractions. All of these abstractions is ment to provide a core and basic interfaces to not only hide implimentation detail, but to **shield** you from it. Having direct control over everything might seem cool--but imagine how much problems would be caused by doing this.

You might be curious as of what problems I am referring to. Remember that, it its core, electronics does only what we tell it. We can control the software down to the **hardware** level, and in some cases, **electronics** level. Making a mistake at these levels can physically cause damage to those devices.

Lets take a look at each abstraction layer to understand what I mean, and to see where our Kernel fits in.

Relationship with PMode Protection Ring Levels

In **Bootloaders 3 Tutorial**, we have took a detailed look at the Rings of Assembly Language. We also looked at how this related to **protected mode**.

Remember that **Ring 0 software has the lowest protection level.** This means that we have direct control over everything, and are **expected** to never crash. If **any** Ring 0 program was to crash, it will take the system down with it (Triple Fault).

Because of this, not only do we want to **shield** everything else from direct control, but we want to only give software the protection level needed to run it. Because of this, normally:

- Kernels work in Ring 0 ("Supervisor Mode")
- Device Drivers work in Rings 1 and 2, as they require direct access to hardware devices
- Normal application software work in Ring 3 ("User Mode")

Okay... **how** does this all fit together? Lets take a closer look...

Level 1: Hardware Level

This is the actual physical component. The actual microcontroller chips on the motherboard. They send low level commands to other microcontrollers on other devices that pysically control this device. How? We will look at that in Level 2.

Examples of hardware are the microcontroller chipset (The "Motherboard Chipset'), disk drives, SATA, IDE, hard drives, memory, the processor (Which is also a controller--Please see Level 2 more more information).

This is the lowest level, and the most detailed as it is pure electronics.

Level 2: Firmware Level

The Firmware sets ontop of the electronics level. It contains the software needed by each hardware device and microcontroller. One example of firmware is the BIOS POST.

Remember the processor itself is nothing more then a controller--and just like other controllers, rely on its firmware. The **Instruction Decoder** within the processor dissects a single machine instruction into either **Macrocode**, or directly to **Microcode**.

Please see the **Tutorial 7: System Architecture Tutorial** for more information.

Microcode

Firmware is useually developed using microcode, and either assembled (With a microassembler) and uploaded into a storage area (Such as the BIOS POST), or hardwired into the logic circuits of the device through various of means.

Microcode is useually stored within a ROM chip, such as EEPROM.

Microcode is very hardware specific. Whenever there is a new change or revision, a new Microcode instruction set and Microassembler needs to be developed. On some systems, Microcode has been used to control individual electronic gates and switches within the circuit. Yes, It is that low level.

Macrocode

Microcode is **very** low level, and can be **very** hard to develop with, espically in complex systems, such as a microprocessor or CPU. It also must be reimplimented whenever a change happens--Not only the code, but the Microprograms as well.

Because of this, some systems have implimented a more higher level language called **Macrocode** ontop of Microcode. Because of this abstraction layer, Macrocode changes less frequently then that of Microcode, and is more portable. Also, do to its abstraction layer, is more easier to work with.

It is still, however, very low level. It is used as the internal logic instruction set to convert higher level machine language into Microcode--which is translated by the Instruction Decoder.

Level 3: Ring 0 - Kernel Level

This is where we are at. The Stage 2 Bootloaders only focus was to set everything up so that our Kernel has an envirement to run in.

Our Kernel provides the abstraction between Device Drivers and Applications software, and the firmware that the hardware uses.

Level 4: Rings 1 and 2 - Device Drivers

Device Drivers go through the Kernel to access the hardware. Device Drivers need alot of freedom and control because they require direct control over specific microcontrollers. Having **to much** control, however, can crash the system. For example, what would happen if a driver modified the GDT, or set up its own? Doing so will immediately crash the kernel. Because of this, we will want to insure these drivers cannot use **LGDT** to load its own GDT. **This is why we want these drivers to operate at either Ring 1 or Ring 2--Not ring 0**.

For an example, a **Keyboard Device Driver** will need to provide the interface between **Applications software** and the **Keyboard Microcontroller**. The driver may be loaded by the Kernel as a library providing the routines to indirectly access the controller.

As long as there is a standard interface used, we can provide a very portable Kernel as long as we hide all hardware dependencies.

Level 5: Ring 3 - Applications Level

This is where the software are at. They use the interfaces provided by the System API and Device Driver interfaces. Normally they do not access the Kernel directly.

Conclusion

This Series will be developing the drivers during the development of the Kernel. This will allow us to keep things object orianted, and provide abstraction layer for the Kernel.

With that in mind, notice where we are at--**Level 0**. All other programs rely on the Kernel. Why? Lets look at the Kernel...

The Kernel

Because the Kernel is the Core component, it needs to provide the management for everything that relys on it. **The primary purpose of the Kernel is to manage system resources, and provide an interface so other programs can access these resources.** In alot of cases, the Kernel itself is unable to use the interface it provides to other resources. **It has been stated that the Kernel is the most complex and difficault tasks in programming.**

This implies that designing and implimenting a good Kernel is very difficault.

In **Tutorial 2** we took a brief look at different past operating systems. We have bolded alot of new terms inside that tutorial--and have compilied a list of those terms at the end of the tutorial. This is where that list starts getting implimented.

Lets first look at that list again, and look at how it related to the Kernel. Everything **Bolded** is handled by the Kernel:

- **Memory Management**
- **Program Management**
- **Multitasking**
- **Memory Protection**
- Fixed Base Address - This was covered in Tutorial 2
- Multiuser - This is useually implimented by a shell

- **Kernel** - Of course
- **File System**
- Command Shell
- Graphical User Interface (GUI)
- Graphical Shell
- Linear Block Addressing (LBA) - This was covered in Tutorial 2
- Bootloader -Completed

Some of the above can be implimented as seperate drivers, used by the Kernel. For example, Windows uses **ntfs.sys** as an NTFS Filesystem Driver.

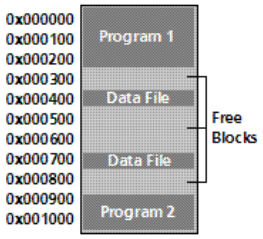
This list should look familiar from **Tutorial 2**. We have also covered some of these terms. Lets look at the **bolded** terms, and see how they relate to the Kernel. We will also look at some new concepts.

Memory Management

This is quite possibly the most important part of any Kernel. And rightfully so--all programs and data require it. As you know, in the Kernel, because we are still in **Supervisor Mode** (Ring 0), **We have direct access to every byte in memory**. This is very powerful, but also produces problems, espically in a multitasking evirement, where multiple programs and data require memory.

One of the primary problems we have to solve is: What do we do when we run out of memory?

Another problem is **fragmentation**. **It is not always possible to load a file or program into a sequencal area of memory**. For an example, lets say we have 2 programs loaded. One at 0x0, the other at 0x900. Both of these programs requested to load files, so we load the data files:



Notice what is happening here. There is alot of unused memory between all of these programs and files. Okay...What happens if we add a bigger file that is unable to fit in the above? This is when big problems arise with the current scheme. We cannot directly manipulate memory in any specific way, as it will currupt the currently executing programs and loaded files.

Then there is the problems of where each program is loaded at. Each program will be required to be **Position Indipendent** or provide **relocation Tables**. Without this, we will not know what base address the program is supposed to be loaded at.

Lets look at these deeper. Remember the **ORG** directive? This directive sets the location where your program is expected to load from. By loading the program at a different location, the program will refrence incorrect addresses, and will crash. We can easily test this theory. Right now, Stage2 expects to be loaded at 0x500. However, if we load it at 0x400 within Stage1 (While keeping the **ORG 0x500** within Stage2), a triple fault will accure.

This adds on two new problems. How do we know where to load a program at? Coinsidering all we have is a binary image, we **cannot** know. However, if we make it standard that all programs begin at the same address--lets say, 0x0, then we can know. This would work--but is impossible to impliment if we plan to support multitasking. **However, if we give each program there own memory space, that virtually begins at 0x0, this will work.** After all, from each programs' perspective, they are all loaded at the same base address--even if they are different in the real (physical) memory.

What we need is some way to abstract the physical memory. Lets look closer.

Virtual Address Space (VAS)

A **Virtual Address Space** is a **Program's Address Space**. One needs to take note that this does **not** have to do with **System Memory**. The idea is **so that each program has their own independent address space. This insures one program cannot access another program, because they are using a different address space.**

Because **VAS** is **Virtual** and not directly used with the physical memory, it allows the use of other sources, such as disk drives, as if it was memory. That is, **It allows us to use more memory then what is physically installed in the system.**

This fixes the "Not enough memory" problem.

Also, as each program uses its own **VAS**, we can have each program always begin at base 0x0000:0000. This solves the relocation problems discussed ealier, as well as memory fragmentation--as we no longer need to worry about allocating continous physical blocks of memory for each program.

Virtual Addresses are mapped by the Kernel trough the MMU. More on this a little later.

Virtual Memory: Abstract

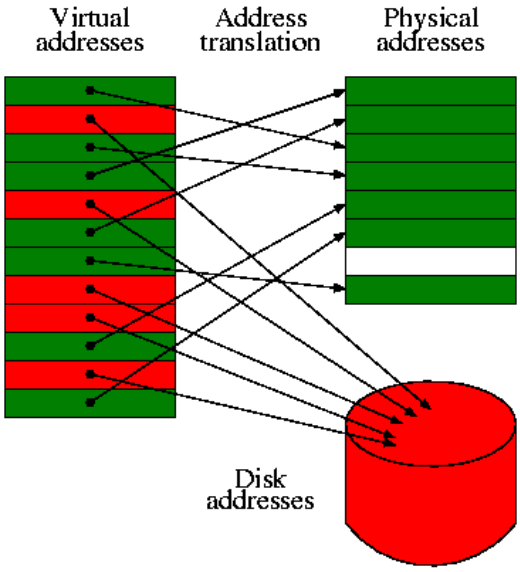
Virtual Memory is a special **Memory Addressing Scheme** implimented by both the hardware and software. It allows non contigous memory to act as if it was contigius memory.

Virtual Memory is based off the **Virtual Address Space** concepts. It provides every program its own Virtual Address Space, allowing memory protection, and decreasing memory fragmentation.

Virtual Memory also provides a way to indirectly use more memory then we actually have within the system. One common way of approching this is by using **Page files**, stored on a **hard drive**.

Virtual Memory needs to be mapped through a hardware device controller in order to work, as it is handled at the hardware level. This is normally done through the **MMU**, which we will look at later.

For an example of seeing virtual memory in use, lets look at it in action:



Notice what is going on here. Each memory block within the **Virtual Addresses** are linear. Each Memory Block is **mapped** to either it's location within the real physical RAM, or another device, such as a hard disk. The blocks are swapped between these devices as an as needed bases. This might seem slow, but it is very fast thanks to the MMU.

Remember: Each program will have its own Virtual Address Space--shown above. Because each address space is linear, and begins from 0x0000:00000, this immiedately fixes alot of the problems relating to memory fragmentation and program relocation issues.

Also, because **Virtual Memory** uses different devices in using memory blocks, it can easily manage more then the amount of memory within the system. i.e., If there is no more system memory, we can allocate blocks on the hard drive instead. If we run out of memory, we can either increase this page file on an as needed bases, or display a warning/error message,

Each memory "Block" is known as a **Page**, which is useually **4096 bytes** in size.

Once again, we will cover everything in much detail later.

Memory Management Unit (MMU): Abstract

My, oh my, where have we heard this term before? o.0 :)

The MMU, Also known as **Paged Memory Management Unit (PMMU)** is a component inside the microprocessor responsible for the management of the memory requested by the CPU. It has a number of responsibilities, including **Translating Virtual Addresses to Physical Addresses, Memory Protection, Cache Control, and more.**

Segmentation: Abstract

Segmentation is a method of **Memory Protection**. In Segmentation, we only allocate a certain address space from the currently running program. This is done through the **hardware registers**.

Segmentation is one of the most widly used memory protection scheme. On the x86, it is useually handled by the **segment registers**: CS, SS, DS, and ES.

We have seen the use of this through Real Mode.

Paging: Abstract

THIS will be important to us. Paging is the process of managing program access to the virtual memory pages that are not in RAM. We will cover this alot more later.

Program Management

THIS is where the ring levels start getting important.

As you know, Our Kernel is at Ring 0, while the applications are at Ring 3. This is good, as it prevents the applications direct access to certain system resources. This is also bad, as alot of these resources are needed by the applications.

You might be curious on how the processor knows what ring level it is in, and how we can switch ring levels. The processor simply uses an internal flag to store the current ring level. Okay, but how does the processor know what ring to execute the code in?

This is where the **GDT and LDT** become important.

As you know, in Real Mode, there is no protection levels. Because of this, everything is "Ring 0". Remember that **we have to set up a GDT prior to going into protected mode?** Also, remember that we needed to execute a **far jump** to enter the 32 bit mode. Lets go over this in more detail here, as they will play very important roles here.

Supervisor Mode

Ring 0 is known as **supervisor mode**. It has access to every instruction, register, table, and other, more privedged resources that no other applications with higher ring levels can access.

Ring 0 is also known as **Kernel level**, and is **expected** never to fail. If a ring 0 program crashes, it will take the system down with it. Remember that: ***"With great power comes great responsibility"***. This is the primary reason for protected mode. ;)

Supervisor Mode utilizes a hardware flag that can be changed by system level software. System level sftware (Ring 0) will have this flag set, while application level software (Ring 3) will not.

There are alot of things that only Ring 0 code can do, that Ring 3 code cannot. Remember the flags register from **Tutorial 7**? The **IOPL Flag** of the RFLAGS register determins what level is required to execute certain instructions, such as **IN and OUT** instructions. Because the IOPL is useually 0, this means that **Only Ring 0 programs have direct access to hardware via software ports.** Because of this, we will need to switch back to Ring 0 often.

Kernel Space

Kernel Space referrs to a special region of memory that is reserved for the Kernel, and Rng 0 device drivers. In most cases, **Kernel Space should never be swapped out to disk, like virtual memory**.

If an operating software runs in **User Space**, it is often known as **"Userland"**.

User Space

This is normally the **Ring 3 application programs**. Each application useually executes in its own **Virtual Address Space (VAS)** and can be swapped from different disk devices. **Because each application is within their own virtual memory, they are unable to access another programs memory directly.** Because of this, they will be required to go through a Ring 0 program to do this. This is neccessary for **Debuggers**.

Applications are normally the least priveldged. Because of this, they useually need to request support from a ring 0 Kernel level software to access system resources.

Switching Protection Levels

What we need is a way so that these applications can query the system for these resources. However, to do this, we need to be in Ring 0, Not Ring 3. Because of this, we need a way to switch the processor state from Ring 3 to Ring 0, and allow applications to quey our system.

Remember back in **Tutorial 5** we covered the rings of assembly language. Remember that the processor will change the current ring level under these conditions:

- A directed instruction, such as a **far jump, far call, fat ret** etc.
- A **trap** instructon, such as **INT, SYSCALL, SYSEXIT, SYSENTER, SYSRETURN** etc.
- **Exceptions**

So...In order for an application to execute a system routine (while switching to Ring 0), the application must either **far jump**, execute an **Interrupt**, or use a special instruction, such as **SYSENTER**.

This is great--but how does the processor know what ring level to switch into? This is where the GDT comes into play.

Remember that, in each descriptor of the GDT, we had to set up the **Ring Level** for each descriptor? In our current GDT, We have 2 descriptors: Each for Kernel Mode Ring 0. **This is our Kernel Space**.

All we need to do is to add 2 mode descriptors to our current GDT, **but set for Ring 3 access**. **This is our User Space**.

Lets take a closer look.

Remember from tutorial 8 that the important byte here is the access byte!. Because of this, here is the byte pattern again:

- Bit 0 (Bit 40 in GDT): Access bit (Used with Virtual Memory). Because we don't use virtual memory (Yet, anyway), we will ignore it. Hence, it is 0
- Bit 1 (Bit 41 in GDT): is the readable/writable bit. Its set (for code selector), so we can read and execute data in the segment (From 0x0 through 0xFFFF) as code
- Bit 2 (Bit 42 in GDT): is the "expansion direction" bit. We will look more at this later. For now, ignore it.
- Bit 3 (Bit 43 in GDT): tells the processor this is a code or data descriptor. (It is set, so we have a code descriptor)
- Bit 4 (Bit 44 in GDT): Represents this as a "system" or "code/data" descriptor. This is a code selector, so the bit is set to 1.
- Bits 5-6 (Bits 45-46 in GDT): is the privilege level (i.e., Ring 0 or Ring 3). We are in ring 0, so both bits are 0.
- Bit 7 (Bit 47 in GDT): Used to indicate the segment is in memory (Used with virtual memory). Set to zero for now, since we are not using virtual memory yet

```
;*****************************************
; Global Descriptor Table (GDT)
;*****************************************

gdt_data:

; Null descriptor (Offset: 0x0)--Remember each descriptor is 8 bytes!
        dd 0                            ; null descriptor
        dd 0

; Kernel Space code (Offset: 0x8 bytes)
        dw 0FFFFh                       ; limit low
        dw 0                            ; base low
        db 0                            ; base middle
        db 10011010b                    ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0
        db 11001111b                    ; granularity
        db 0                            ; base high

; Kernel Space data (Offset: 16 (0x10) bytes
        dw 0FFFFh                       ; limit low (Same as code)10:56 AM 7/8/2007
        dw 0                            ; base low
        db 0                            ; base middle
        db 10010010b                    ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0
        db 11001111b                    ; granularity
        db 0                            ; base high

; User Space code (Offset: 24 (0x18) bytes)
        dw 0FFFFh                       ; limit low
        dw 0                            ; base low
        db 0                            ; base middle
        db 11111010b                    ; access - Notice that bits 5 and 6 (privilege level) are 11b for Ring 3
        db 11001111b                    ; granularity
        db 0                            ; base high

; User Space data (Offset: 32 (0x20) bytes
        dw 0FFFFh                       ; limit low (Same as code)10:56 AM 7/8/2007
        dw 0                            ; base low
        db 0                            ; base middle
        db 11110010b                    ; access - Notice that bits 5 and 6 (privilege level) are 11b for Ring 3
```

```
        db 11001111b                    ; granularity
        db 0                            ; base high
```

Notice what is happening here. All code and data have the same range values--the only difference is that of the **Ring** levels.

As you know, **protected mode** uses CS to store the **Current Privilege Level (CPL)**. When entering protected mode for the first time, **We needed to switch to Ring 0**. Because the value of CS was invalid (From real mode), we need to choose the correct descriptor from the GDT into CS. **Please see Tutorial 8 for more information**.

This required a far jump, as we needed to upload a new value into CS. By **far jumping** to a Ring 3 descriptor, we can effectivly enter a Ring 3 state.

As, as you know, we can use a **INT, SYSCALL/SYSEXIT/SYSENTER/SYSRET, far call, or an exception** to have the processor switch back to Ring 0.

Lets take a look closer at these methods...

System API: Abstract

The program relies on the System API to access system resources. Most applications refrence the System API directly, or through their language API--Such as the **C runtime library**.

The System API provides the **Interface** between applications and system resources through **System Calls**.

Interrupts

A **Software Interrupt** is a special type of interrupt implimented in software. Interrupts are used quite often, and rely on the use of a special table--the **Interrupt Descriptor Table (IDT)**. We will look at Interrupts alot more closer later, as it is the first thing we will impliment in our Kernel.

Linux uses INT 0x80 for all system calls.

Interrupts are the most portable way to impliment system calls. Because of this, we will be using interrupts as the first way of invoking a system routine.

Call Gates

Call Gates provide a way for Ring 3 applications to execute more prividged (Ring 0,1,2) code. The **Call gate** interfaces between the Ring 0 routines and the Ring 3 applications, and is normally set up by the Kernel.

Call Gates provide a single gate (Entry point) to FAR CALL. This entry point is defined within the GDT or LDT.

It is much easier to understand a call gate with an example.

```
;*****************************************
; Global Descriptor Table (GDT)
;*****************************************

gdt_data:

; Null descriptor (Offset: 0x0)--Remember each descriptor is 8 bytes!
        dd 0                            ; null descriptor
        dd 0

; Kernel Space code (Offset: 0x8 bytes)
        dw 0FFFFh                       ; limit low
        dw 0                            ; base low
        db 0                            ; base middle
        db 10011010b                    ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0
        db 11001111b                    ; granularity
        db 0                            ; base high

; Kernel Space data (Offset: 16 (0x10) bytes
        dw 0FFFFh                       ; limit low (Same as code)10:56 AM 7/8/2007
        dw 0                            ; base low
        db 0                            ; base middle
        db 10010010b                    ; access - Notice that bits 5 and 6 (privilege level) are 0 for Ring 0
        db 11001111b                    ; granularity
        db 0                            ; base high

; Call gate (Offset: 24 (0x18) bytes

 CallGate1:
        dw (Gate1 & 0xFFFF)             ; limit low address of gate routine
        dw 0x8                          ; code segment selector
        db 0                            ; base middle
        db 11101100b                    ; access - Notice that bits 5 and 6 (privilege level) are 11 for Ring 3
        db 0                            ; granularity
        db (Gate1 >> 16)                ; base high of gate routine

; End of the GDT. Define the routine wherever

; The call gate routine

Gate1:
        ; do something special here at Ring 3

        retf                    ; far return back to calling routine
```

The above is an example of a call gate.

To execute the call gate, we offset from the **descriptor code** within the GDT. Notice how simular this is from our **jmp 0x8:Stage2** instruction:

```
; execute the call gate
        call far        0x18:0                  ; far call--calls our Gate1 routine
```

Call Gates are not used too often in modern operating systems. One of the reasons is that most architectures do not support Call Gates. They are also quite slow as they require a **FAR CALL** and **FAR RET** instructions.

On systems where the GDT is not in protected memory, it is also possible for other programs to create their own Call Gates to raise its protection level (and get Ring 0 access.) They have also been known to have security issues. One notable worm, for example, is **Gurong**, which installs its own call gate in the Windows Operating System.

SYSENTER / SYSEXIT Instructions

These instructions were introduced from the Pentium II and later CPUs. Some recent AMD processors also support these instructions.

SYSENTER can be executed by any application. **SYSRET** can only be executed by Ring 0 programs.

These instructions are used as a fast way to transfer control from a User Mode (Ring 3) to a Privildge Mode (Ring 0), and back quickly. This allows a fast and safe way to execute system routines from user mode.

These instructions directly rely on the Model Specific Registers (MSR's). Please see Tutorial 7 for an explination of MSRs, and the RDMSR and WRMSR instructions.

SYSENTER

The **SYSENTER** instruction automatically sets the following registers to their locations defined within the MSR:

- CS = IA32_SYSENTER_CS MSR + the value 8
- ESP = IA32_SYSENTER_ESP MSR
- EIP = IA32_SYSENTER_IP MSR
- SS = IA32_SYSENTER_SS MSR

This instruction is only used to transfer control from a Ring 3 code to Ring 0. At startup, we will need to set these MSR's to point to a **Starting location** which will be our **Syscall Entry Point** for all system calls.

Lets take a look at SYSEXIT.

SYSEXIT

The **SYSEXIT** instruction automatically sets the following registers to their locations defined within the MSR:

- CS = IA32_SYSENTER_CS MSR + the value 16
- ESP = ECX Register
- EIP = EDX Register
- SS = IA32_SYSENTER_CS MSR MSR + 24

Using SYSENTER/SYSEXIT

Okay, using these instructions might seem complicated, but they are not too hard ;)

Because SYSENTER and SYSEXIT require that the MSR's are set up **prior** to calling them, we first need to initialize those MSRs.

Remember that IA32_SYSENTER_CS is index 0x174, IA32_SYSENTER_ESP is 0x175, and IA32_SYSENTER_IP is 0x176 within the MSR. Remember from tutorial 7?

Knowing this, lets set them up for SYSENTER:

```
        %define IA32_SYSENTER_CS 0x174
        %define IA32_SYSENTER_ESP 0x175
        %define IA32_SYSENTER_EIP 0x176

        mov     eax, 0x8                                ; kernel code descriptor
        mov     edx, 0
        mov     ecx, IA32_SYSENTER_CS
        wrmsr

        mov     eax, esp
        mov     edx, 0
        mov     ecx, IA32_SYSENTER_ESP
        wrmsr

        mov     eax, Sysenter_Entry
        mov     edx, 0
        mov     ecx, IA32_SYSENTER_EIP
        wrmsr

        ; Now, we can use sysenter to execute Sysenter_Entry at ring 0 from either a Ring 0 program or Ring 3:
        sysenter

Sysenter_Entry:

        ; sysenter jumps here, is is executing this code at prividege level 0. Simular to Call Gates, normally we will
        ; provide a single entry point for all system calls.
```

If the code that executes **sysenter** is at Ring 3, and **Sysenter_Entry** is at protection level 0, the processor will switch modes within the **SYSENTER** instruction.

in the above code, both are at Protection Level 0, so the processor will just call the routine without changing modes.

As you can see, there is a bit of work that must be done prior to calling SYSENTER.\ and SYSEXIT.

SYSENTER and SYSEEXIT are not portable. Because of this, it is wise to impliment another, more portable, method alongside SYSENTER/SYSEXIT.

SYSCALL / SYSRET Instructions

[I plan on adding a section for SYSCALL and SYSRET here soon]

Error Handling

What do we do if a program causes a problem? How will we know what that problem is and how to handle it?

Normally this is done by means of **Exception Handling**. Whenever the procesor enters an invalid state caused by an invalid instruction, divide by 0, etc; the processor triggers an **Interrupt Service Routine (ISR)**. If you have mapped our own ISR's, it will call our routines.

The **ISR** called depends on what the problem was. This is great, as we know what the problem is, and can try finding the program that originally caused the problem.

One way of doing this is simply getting the last program that you have given processor time to. That is guaranteed to be the one that has generated the ISR.

Once you have the programs information, then one can either output an error or attempt to shutdown the program.

IRQs are mapped by the internal Programmable Interrupt Controller (PIC) inside the processor. They are mapped to interrupt entrys within the Interrupt Descriptor Table (IDT). This is the first thing we will work on inside the Kernel, so we will cover everything later.

Conclusion

We looked at alot of different concepts in this tutorial, ranging from Kernel theory, memory management concepts, Virtual Memory Addressing (VMA), and program management, including seperating Ring 0 from Ring 3, and providing the interface between applications and system software. Whew! Thats alot, don't you think?

Alot of the concepts in this tutorial may be new to you--don't worry. This is more of a "Get your feet wet" tutorial, where we cover all of the basic concepts related to Kernels.

This tutorial has bairly scratched the surface of what a Kernel must do. That is a start, though. ;)

In the next tutorial, we are going to look at Kernels from another perspective. We will cover some new concepts yet again, and talk about Kernel designs and implimentations. Afterwords, we will start building our compiliers and toolchains to work with C and C++. Sound fun?

I am currently using MSVC++ 2005 for my Kernel.

We will also finish off other concepts that we have not looked at here, including **multitasking, TSS, Filesystems**, and more. *Its going to be fun ;)*

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 11 Home Chapter 13

Operating Systems Development Series

Operating Systems Development - Kernel: Basic Concepts Part 2

by Mike, 2007

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

Introduction

Welcome! :)

We have covered alot of concepts in the previous tutorial. In this tutorial, we will continue looking at these concepts. We will be looking at alot of very important concepts in this tutorial. Heres whats on the menu for today:

- Hardware Abstraction
- Kernel: A new perspective
- Kernel designs: Abstract: Primary Design Models
- Kernel designs: Abstract: Secondary Design Models

This tutorial will cover the glue between the bootloader and beginning kernel design. This will be the glue that brings everything together for us to see what kernels really are, and to understand what we need to do.

All the concepts listed here will be very important in the next few tutorials, as we will start designing and developing our Hardware Abstraction Layer (HAL) and our uber-1337 kernel. er... wait, thats actually the next tutorial!!

So, lets both lay back in our confortable seats and take a look at another fun happy tutorial! Oh right, and this tutorial is not that big either, which is nice :)

Note: This tutorial recommends that our readers have read through Tutorial 2 before proceeding.

I plan on adding a section describing exokernels. They are a realitvely new kernel design concept. I do not plan on implimenting this design within this series, but feel our readers might be interested in the kernel design. Perhaps use the design in their own OS?

Ready?

Hardware Abstraction

Hardware Abstraction is very important. By now, you may know how complex hardware programming can be, and how very hardware dependent it is. This is where a **Hardware Abstraction Layer (HAL)** comes in.

A HAL is a software abstraction layer used to provide an interface to the physical hardware. It is an abstraction layer. These abstractions provide a way to interact with devices, while not needing to know the details of a particular device or controller.

Normally in modern OSs, the HAL is a basic **Motherboard chipset driver**. It provides a basic interface between the kernel and the hardware of the machine, including the processor. This is great, as the Kernel can interact with the HAL whenever it needs access to the hardware. This also means that the kernel can be completely hardware independent.

This also allows us to think in terms of the device itself, rather then specific controllers or mappings. This helps make the kernel itself cleaner as well.

Another great benefit comes from abstraction itself. If we decide to port our OS to a system with different hardware, all we need to do is develop a new HAL for it. This assumes that the HAL is designed very well to allow this.

Most modern operating systems use a HAL in some way. We will also be developing a HAL to act as a motherboard chipset driver between the chipset hardware and the kernel. We will start developing on our HAL within the next tutorial, when we abstract the processor itself behind the HAL.

Kernel: A new perspective

So... What exactally is Kernel? Kernel is a Scheme like programming language developed by John N. Shutt (Serously ;))

Anywhoo, lets look at another definition. A "Kernel" is the central component of a system. This system can be anything. The Kernel is the core of the system; it provides the very basic facilities for the management of effeciant execution of the system.

In an operating system, this all-so-powefull Kernel provides the most basic interface to the system hardware and resources. It also provides the most basic management facilities, such as processor management, I/O management, memory management, and process management. The Kernel can contain more, depending on the complexity of the system being developed.

Okay...The previous list might sound familiar...hm..Where have we seen that before? We actually looked at each inside of Tutorial 2.

Lets look at this closer for better understanding.

Kernel: Putting everything together

Memory Management

Okay, then! Remember again from Tutorial 2. We have created a basic list of items reguarding memory management and protection. Lets take another look at that again:

Memory Management refers to:

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

Memory Protection 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.

Right about now, you should have a better understanding of everything within Tutorial 2.

Remember that the Kernel, as it is running in ring 0, has direct control over every single byte in memory...Even if there is none. Also, remember that this is all running directly on physical memory. What happens if we run out of memory? What would happen if we write to nonexistant memory? What about memory locations used by hardware devices? This also does not touch the "gaps" that may be found throughout memory as well.

Warning: Writing random locations in physical memory may (Depending on where you write to) can cause malfunctions (Depending if a hardware device uses that area of memory), or completely make the system unbootable, and completely useless. (Depending if you write over the BIOS data area.) Never directly probe memory!

With all of this in mind, you should begin to see how important it is to properly manage physical memory.

A Kernel that properly manages the physical memory can create a virtual interface between applications and memory. This can be done through seperating **User Space** and **Kernel Space** code and data, and through **Virtual Addressing.**

Because of the many problems with directly running in physical memory, we can emulate a better method of memory. Thie memory emulation (Virtual Memory) can emulate a system with alot more memory then physical RAM, where each application uses its own virtual memory address space.

Processor Management

This is a new one. As you know, the BIOS ROM initializes and starts up the primary processor. It only starts a single core. If you are running your OS on a system with a muticore processor, or a system with multiple processors, you will need to start up the other processors and cores manually.

Letting applications play with the different processors at any time can cause fatal system problems. Because of this, we should never allow applications the ability to do this.

I/O Device Management

Simular to physical memory, allowing applications direct access to controller ports and registers can cause the controller to malfunction, or system to crash. With this, depending on the complixity of the device, some devices can get surprisingly complex to program, and uses several different controllers. Because of this, providing a more abstract interface to manage the device is important. This interface is normally done by a **Device Driver** or **Hardware Abstraction Layer**. This allows us to think in terms of the device, rather then its details.

Frequently, applications will require access to these devices. The Kernel must maintain the list of these devices by querying the system for them in some way. This can be done through the BIOS, or through one of the various system buses (Such as PCI/PCIE, or USB.) When an application requests an operation on a device (Such as, displaying a character), the kernel needs to send this request to the current active video driver. The video driver, in turn, needs to carry out this request. This is an example of **Inter Process Communication (IPC).**

Process Management

This is the most important task of the Kernel, and any computer for that matter. The Kernel needs a way of allocating execution time, and executing and management of different applications and processes.

This is where Program Management, and Multitasking comes in. These terms should sound familiar from Tutorial 2. Lets take another look at that from tutorial 2:

Program Management is responsible for:

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

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 simulataniously.

To execute an application, the Kernel must set up the applications own **Virtual Address Space (VAS)**, and load the file into the VAS. the Kernel then sets up the applications stack, and jumps to it to begin execution.

Through Virtual Addressing, we can insure the application does not run into system memory problems.

In a multitasking system, the Task Manager will allocate a certain amount of time to each process, and only execute them within that time frame. It will then switch between running applications. Because the time allocated is small, the Task Manager can switch between running process quickly, giving the illusion of multiplt processes running simulataniously.

This can either be done through hardware or software. The processor supports hardware task switching through the use of its **Task State Segment (TSS)** register.

The System API

By now, you should start being able to understand how everything fits together, and where alot of the concepts from Tutorial 2 starts to come in. Yet, there is one little detail that we have not covered yet.

How does the application ask the Kernel for request to a device or system resource? We have seen methods on how the OS manages and control the application, but how does the application control the system?

This is where the system **Application Programming Interface (API)** comes in. The System API is an API that applications may use to interact with the Kernel and other system software.

There are alot of methods for creating the System API. Most systems support System API routines through interrupts. For an example, The Linux Kernel System API primarily uses interrupt number 0x80 for system routines.

Conclusion

Wow, that is alot of stuff, huh? Dont worry if you dont understand this yet. Everything will be clear soon enough :)

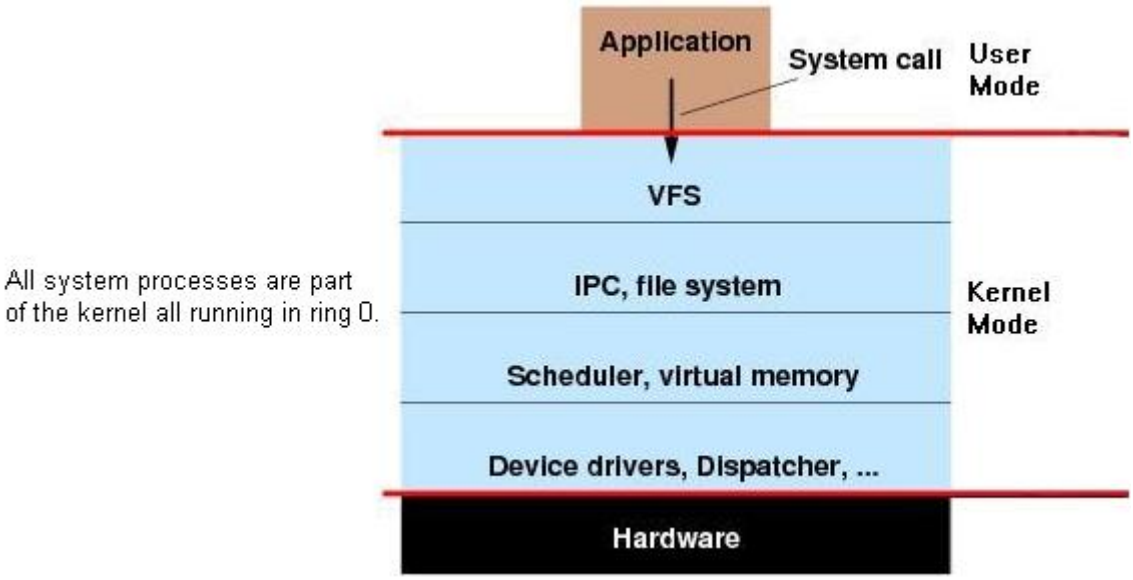
Kernel Designs - Abstract: Primary Design Models

By now, you may start realizing how important Kernels are, and where they fit in.

There has been alot of operating systems that have been developed using alot of different designs and setups. Alot of these designs have some simular basic concepts.

There are alot of different ways to construct kernels. We will look some of the more used designs here.

Monolithic kernel Design



In a Monolithic Kernel, all system process run as part of the Kernel at Ring 0

Lets first look at the term "Monolithic". The first part--"Mono" means "one". The second part--"lithic" means "it is of or like stone".

In a Monolithic kernel, the entire Kernel executes in Kernel space at Ring 0. It provides a higher level interface over computer resources and hardware. It also provides a basic set of system calls via the System API.

In a monolithic kernel, most (if not all) Kernel services are part of the kernel, itself. This does not mean that the services cannot be independent of each other. However, the software is very tightly integrated to the rest of the kernel. This makes monolithic kernels very fast and effeciant, compared to other designs.

Because all OS services run in kernel space as part of the kernel (or as an extension to the kernel), if there is a problem with a device driver or a system service program, it can cause the entire system to crash.

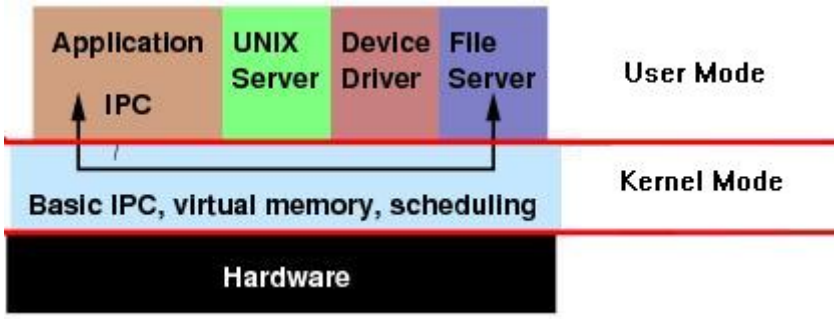
When an application requests a system service, it executes a system call through the System API.

Examples

Several large scale operating systems use a hybrid kernel, including but not limited to:

- Unix-like kernels
  - Linux
  - Syllable
  - Unix kernels
    - BSD
      - FreeBSD
      - NetBSD
  - Solaris
  - AI
- DOS
  - DR-DOS
  - MS-DOS
    - Microsoft Windows 9x series (95, 98, Windows 98SE, Me)
- Mac OS kernel, up to Mac OS 8.6
- OpenVMS
- XTS-400

Microkernel Design



In a Microkernel design, the Kernel only provides basic functionaility needed for usermode system services.

A Microkernel is a kernel design that provides no OS services at all, only the mechanisms needed to impliment those services. Because of this, the Kernel itself is useually quite small compared to Monolithic kernels. For example, a microkernel may impliment low level memory management and thread management, and Inter Process Communication (IPC).

The above image displays a microkernel. Notice the kernel only impliments the very basics of basics. In this case, it impliments basic process management and scheduling, Inter Process Communication (IPC), and basic virtual memory management.

The Kernel would use external usermode services, such as Device Drivers and Filesystems, rather then everything implimented as part of the kernel (As with Monolithic kernels.) Because of this, if an external service crashes, the system may still be functional, and the system will not crash.

Inter Process Communication (IPC) and understanding Servers and Device Drivers is important in understanding how Microkernels work.

Microkernel Servers

Microkernel "Servers" are external programs that is granted special privileges by the kernel that normal programs do not have. These "privileges" may be direct access to hardware, or even physical memory. This allows server programs to enteract directly with the hardware devices they are controlling. Wait...It sounds like a device driver, doesnt it? Yep :) Thats basically what they are.

Remember that microkernels are very minimal. They rely on external programs - servers - to help out.

Servers needed by the kernel itself is normally loaded into memory before the kernel is executed. An example that will be needed is a file system server, that will contain the code for parsing the filesystem.

Because the Kernel has no filesystem code, it has no way of loading the filesystem server! Because of this, it needs to be loaded **before** the kernel is executed.

How can we do this? There are several ways. One method is loading a complete RAM image containing both the kernel and supported severs in it. Another method is simply loading the necessary servers at startup within the bootloader, and in someway giving the server information to the kernel upon executing. In both cases, the bootloader can determin what filesystem loading code to use, however the code can interact with the filesystem server without ever needing to load it in the first place! Cool, huh?

Note: A "server" may also be called a "daemon".

Inter Process Communication (IPC)

IPC is very important in microkernels. It allows seperate processes to communicate with each other, useually by sending messages, but it can also be envoked by using shared memory.

There are alot of ways a process can "signal" another process. With reguards to microkernel severs, the most commonly used is also one of the easiest to understand - message passing.

IPC allows the servers and kernel to interact with each other.

Synchronous IPC

In **Synchronous IPC**, The process sending the message is suspended until the other process responds. If the other process is busy, the message is stored in a queue for that process to act upon when ready.

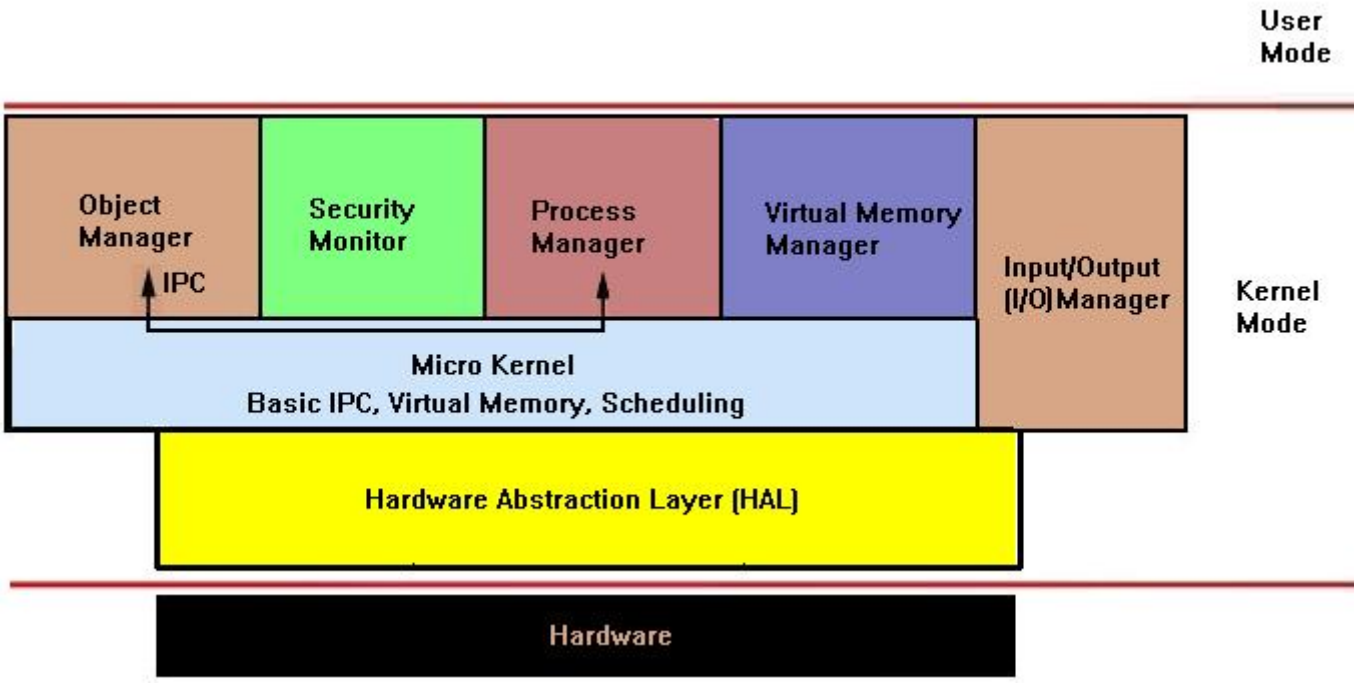
Asynchronous IPC

Simular to **Synchronous IPC**, however both processes continue executing. That is, the process is not suspended.

Kernel Designs - Abstract: Secondary Design Models

Remember that there are countless of ways that kernels may be designed. The following are common design models that are based off of the primary design models (Monolithic and Microkernels).

Hybrid kernels



Hybrid kernels are Microkernels with aspects from Monolithic kernels

A Hybrid kernel is a kernel combining aspects from both Monolithic and Microkernel designs.

Hybrid kernels useually has a structure simular to microkernels, but implimented as a monolithic kernel. Lets look at this another way for better understanding.

Hybrid Kernels, simular to microkernels, use seperate sever programs for filesystems, device drivers, etc. However, like Monolithic Kernels, these severs execute as part of the Kernel, instead of user space.

There are some controversy on what the term "Hybrid Kernel" applies to. It may be called a "Microkernel", "Macrokernel", or "modified microkernel or modified macrokernel". Hrbrid kernels are not in their own design; they are just modified Microkernels with some aspects from monolithic kernels. Do to this, there are some controversy on what to coinsider hybrid kernels.

Microsoft's NT Architecture uses a hybrid approch to their kernel design model. Microsoft describes their kernel as a "Modified microkernel".

Examples

Several large scale operating systems use a hybrid kernel, including but not limited to:

- BeOS Kernel
  - Haiku Kernel
- BSD
  - DragonFly BSD Kernel
  - XNU Kernel
- NetWare Kernel
- Plan 9 Kernel
  - Inferno Kernel
- Windows (NT,2000,2003,XP,Vista) NT Kernel
  - ReactOS Kernel

Nanokernel

Nanokernels, also known as **Picokernels**, are a very small kernel. Normally, this would be a minimal microkernel structure. As the kernel itself is very small, it must rely on other software and drivers for the basic resources within the system.

Conclusion

Okay, okay...I have to admit this tutorial is not that complex. It covers alot of very important topics that we needed to cover, however. Hopefully, this will help our readers gain a better understanding of kernels, and what they are responsible for. After all, this is what we will be building in the upcoming chapter and tutorials. Not just a kernel, but working on a basic hardware abstraction layer (HAL) for it as well.

We will be developing a modified microkernel in this series. This will allow our readers to gain some experience and understanding in both monolithic and microkernel designs, as well as mixing the approches into a Hybrid Microkernel. In fact, our kernel will look simular to that is displayed in this tutorial. We will touch upon the full design of our kernel within the next tutorial, along with developing the basic building blocks for our HAL to abstract processor dependencies using C++.

I will build several versions of the demos from now on to support multiple compiliers and platforms. As well as supporting both as C++ and as the C language. Cool?

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!

Operating Systems Development Series

Operating Systems Development - MSVC++ 2005, 2008, 2010

by Mike, 2008, Updated 2010

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

Introduction

Welcome! :)

This tutorial covers setting up Microsoft Visual C++ 2005 to work in Kernel Land. This will allow us to use, and completely work in, the nice MSVC++ 2005 IDE.

We will be using the Express Edition. If you have the professional edition, some of the screen shots will be different. Do not worry as most of the options are still the same.

This tutorial will also work fine if you have Visual C++ 2008 or Visual C++ 2010 (with some slight modifications in what options to set).

Getting a high level language to work can be tricky to work and set up with. Getting the build envirement to work itself adds on even more complexity.

The first problem, for example, is that MSVC++ only outputs Win32-compilent PE executables and DLL's. And, at our current stage, we only have a flat, pure binary program loaded at absolute address 1 MB. How do we execute a full blown C++ program, from our flat pure-assembler program?

And yet, that is the beginning of the problems. C++ itself relies on the C++ runtime library. So? **The C++ runtime library relies on the operating system.** Because we are developing the operating system, **C++ has no standard library to work with within our operating system**. This means, that we have to work without any standard runtime.

But wait! Remember that, in application software, the **runtime** needs to initialize everything for our C++ envirement (Such as executing global constructors, providing basic C++ operation support, and executing our main())? Because we have no runtime, we have to do everything ourselves. This produces an interesting chicken and egg scenario: How can we develop a runtime envirement without a runtime for it to work in?

As you can probably see, it can be quite tricky to get C++ to even **work** properly.

In this tutorial, we are going to set up MSVC++ 2005 for kernel development, and set up the language to work for us. We will also be watching for runtime integrations (i.e, watching where basic C++ relies on the runtime, so we can build our own).

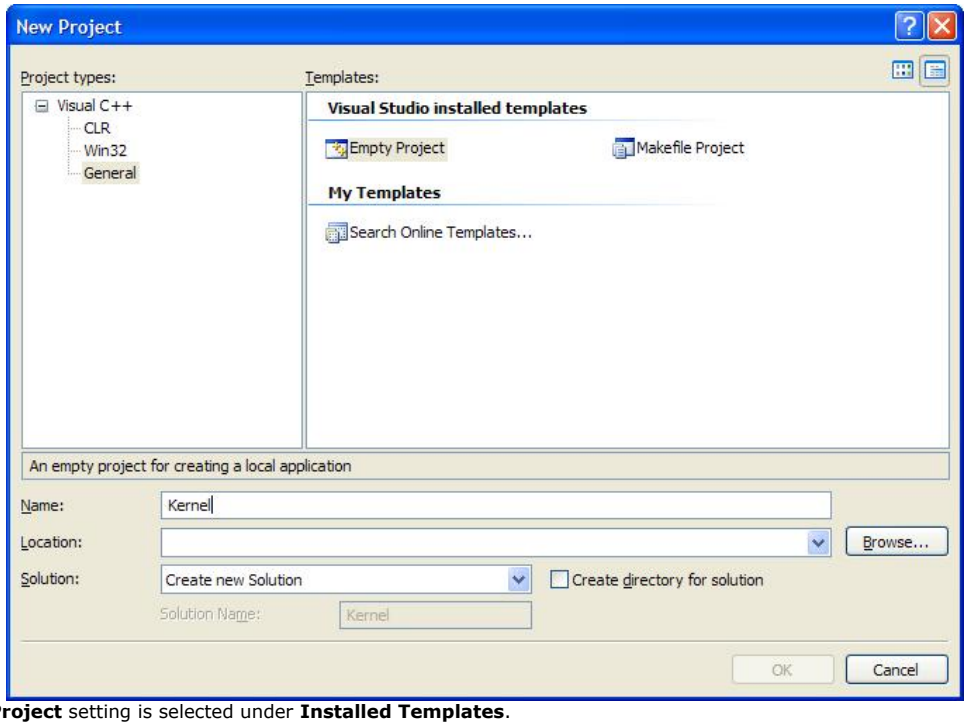
Please note, however, that some C++ features rely on details that we have yet to impliment. These details are fairly advanced. For example, the C++ **new** and **delete** operaters require that we have a working **memory manager** already. As such, this section cannot cover everything just yet. Nontheless, we will set as much things up as possible here.

Ready?

Setting up a new Project

Just like almost any other project, creating a project is fairly straightfoward.

Within the IDE, Select **File->New->Project**. You should see a nice dialog:



Notice that the **Empty Project** setting is selected under **Installed Templates**.

Now, type in the name of the project, and choose the project location. When you are done, select the "OK" button, which should be highlighted.

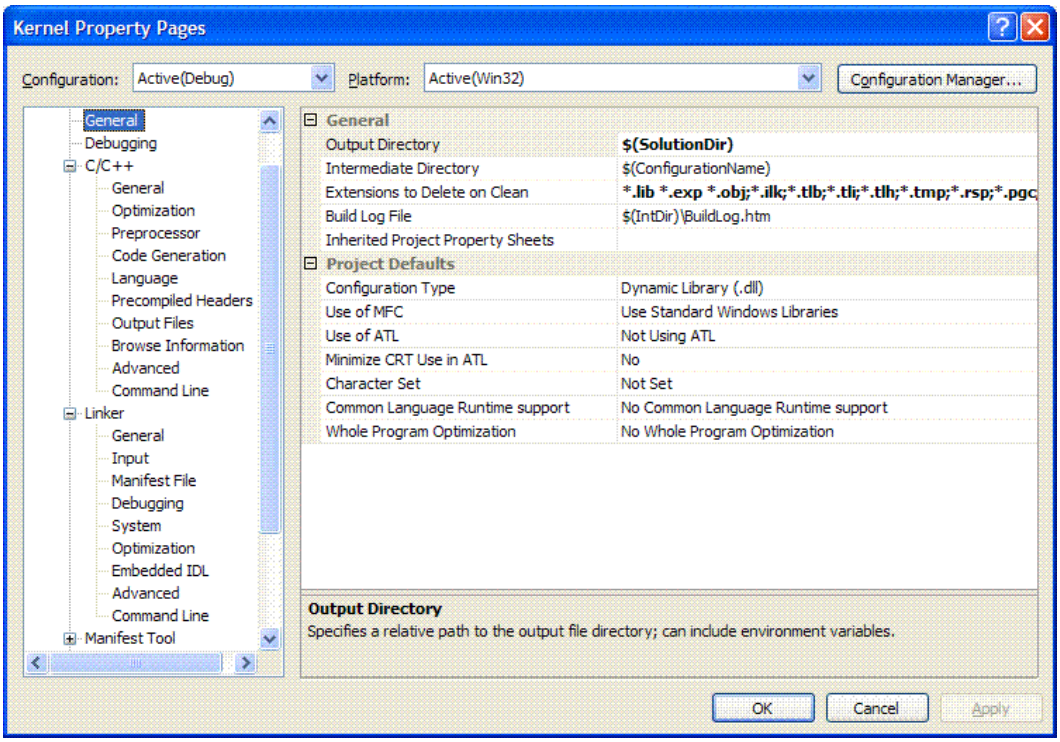
Your new project should now be created.

The Build Envirement - Project Properties

This is great! The problem, however, is that the compilier assumes we are building a basic Win32 application. As such, it sets all of the project properties to it's default configuation, and links the **Standard C++ Runtime** libraries--which will not work for us.

Because of this, we need to change the configuation settings.

To access **Project Properties**, **Right Click** the project name, and select **Properties**. You should recieve a dialog, simular to the following:



All of the configuation settings are discussed in the same format listed in the above picture (Please look at the left plane).

Project Wide Settings

Most configuation settings depend entirely on the envirement that is being built on. **Remember that we are still in a very low level envirement.** A **Single** wrong instruction produced by the compilier can make our code triple fault.

Also, remember that **We are still at Ring 0**. As such, we still have control over every little thing. This is important, because if the code the compilier produces does not work with our current operating system setup, the code will triple fault.

This is a problem. **Alot of the configuation settings effect the output code.** Weather these changes cause a triple fault, or work completely depends on the layout of your operating system, and its configuations. Alot of these changes do not need to be changed. However, because certain configuations may cause a triple fault, this indicates to be extra careful when trying to produce "The fastest and greatest code" from your C++ compilier.

There are, however, some options that **REQUIRE** changing. These are the options that I will look at here. I will also cover some other options that one may find helpful. I will explain these specific options in depth.

At the end of this section, I will post my current configuation, cool? This way, you can compare your configuations, and learn about setting up MSVC++ 2005.

Configuation Type

We are going to be building the kernel as an executable, so keep this as **Application (.exe)**. This setting is located under **General** (Please see the image shown in the above section to locate **Configuation Type**).

C++ Configuation Settings

Looking at the above image again, you will see the **C/C++** and **Linker** properties expanded. the C/C++ properties include General, Optomization, Preprocessor, Code Generation, etc. We will cover these properties here.

C/C++ > General

None of these options we are not required to change. There are a couple of options I want to look at here, however.

Additional Include Directories

This option allows us to provide our own path for INCLUDE files. This will allow us to use the format when including our own files:

```
#include <myheader.h>
```

This can be usefull when seperating the inlcusion of files within the same directory (**#include "myheader.h"**) and including files within a standard Kernel include directory (**#include <myheader.h>**)

Debug Information Format

Although we are using MSVC++, we **Cannot** use it's debugging features. Debuggers require a runtime envirement to run in (and to hook) with an application. Because we do not have a runtime envirement, you should **Disable** this.

Normally, Adding debuggng information will not cause any problems. However, because we are unable to use the debugger right now, there is no reason to generate debug info.

Warning Level

Operating System code can get very complex. It is important that we can track even the slightest of potential problems. Because of this, I recommend to **Set this to the highest level**.

C/C++ > Optomization

Optimization

This can be set to any option. If a specific setting causes the code to crash, disassemble the code and try to find out why. All source code in this series have been tested with all optomization levels to work.

Omit Frame Pointers

This is not required; but setting this option frees up ESP so we may use it.

C/C++ > Preprocessor

Preprocessor Definitions

Throughout the source code, I will be hiding all x86 architecture dependencies behind a special preprocessor constant. This makes it easier to port non-portable code to other architectures. This constant is **ARCH_X86**. This can be #define'd, but putting ARCH_X86 here is easier ;)

Ignore Standard Include Path

Remember we do not have a standard library anymore? :)

C/C++ > Code Generation

There are a few options that I want to go over.

Enable C++ Exceptions

This requires runtime support, which we do not have. I plan on implimenting exception handling soon, though. Until then, this should be set to **No**.

Struct Member Alignment

While writing the Kernel, we will be using alot of classes and structures. Most of these must be byte aligned. Most compiliers add extra padding (for speed) to these structures, which will throw off the alignment that we need. Because of this, set this to **1 Byte (/Zp1)**

Buffer Security Check

When enabled, MSVC++ adds extra code to test for possible buffer under and overruns. This relies on the MSVC++ runtime, which we cannot use. Because of this, we cannot use this. **Set this to No (/GS-)**

C/C++ > Language

Enable Run-Time Type Info

Run Time Type Info (RTTI) requires runtime support. Because we are disabling the runtime, we cannot use this. Set to **No (/GR-)**

C/C++ > Advanced

No changes needed here. I do personally recommend using **__cdecl** over **__stdcall**, as __cdecl seems to have a cleaner symbol names. It does not really matter though.

C/C++ > Command Line

Here is the command line that I am using. If you are having difficulty, feel free to use this as a refrence. You can also see all of these options set in the demo at the end of this chapter.

```
/O2 /Oy /I "..\Include\\" /D "ARCH_X86" /X /FD /MT /Zp1 /GS- /GR- /FAs /Fa"Debug\\"
/Fo"Debug\\" /Fd"Debug\vc80.pdb" /W4 /nologo /c /Gd /TP /errorReport:prompt
```

Compare this command line with the one you currently have. You may have additional options, depending on weather you decided to add more options.

Linker Configuation Settings

The linker is very important to us for a number of reasons. It is responsible for creating the final symbolic names the compilier produces. These symbolic names represent numerical addresses for varables, routines, and constants. For example, the routine "main()" might be compilied into the symbolic name "_main". **In assembly language, we refrence varables and routines by their symbolic names.** Because of this, to call a C++ main() routine, we will normally do this:

```
call _main   ; call C++ main() routine
```

The linker produces a linker map with all of these symbolic names. This will be very important with debugging and testing. With this, there are some linker settings that are required, while others are optional. The optional settings may or may not work depending on your envirement settings and configuations.

Linker > General

There are no options that require changing here. If you are using a real (or virtual) floppy drive, I personally recommend setting the **Output File** to point to the floppy drive. This way, the final binary will be placed into the floppy disk, allowing us to immediately test the Kernel in the emulator.

Linker > Input

Additional Dependencies

By default, MSVC++ automatically links in a number of its libraries, including kernel32.lib, user32.lib, gdi32.lib, winspool.lib and more. Because we never use them, they will not cause any problems. They will add extra uneeded bloat to your kernel, though. Setting this to **$(NOINHERIT)** will fix this so they will not be linked.

Ignore Default Libraries

There is no standard library, so set this to **Yes (/NODEFAULTLIB)**

Linker > Debugging

Remember that the linker can generate a mapfile? This allows us to see the relitive address locations of all global symbolic names. This will be very important to us, coinsidering we are still at the binary image level. To do this:

- Set **Generate Map File** to **Yes (/MAP)**
- Set **Map File Name** to the name of the mapfile to generate.
- Hit **Apply**

Linker > System

SubSystem

This value is stored within the program file. It tells the operating system how to run the application. Because this is a driver application, set this to **Native (/SUBSYSTEM:NATIVE)**.

Driver

This option insures to build this program as a kernel-level driver. This automatically envokes the **/FIXED:NO** option (instead of the standard **/FIXED** option), which generates a relocation section, instead of a fixed base address. Because we are developing a driver application, set this option to **Driver (/DRIVER)**).

Linker > Optimization

Setting **Refrences** to **Eleminate unrefrenced data** removes all unrefrenced symbols (such as varables and functions that are never used.) **I recommend setting this option to reduce the number of symbols from the linker map, and to reduce kernel size.**

Setting **Enable COMDAT folding** to **Remove redundent COMDATS** will also reduce the size of the Kernel, and the number of redundent COMDATS.

Linker > Advanced

Entry Point

This should be set to the entry point of your kernel. In our system, this will be **kernel_entry**.

Base Address

This is the base address that the image will be loaded to. Remember that the kernel is loaded to 1MB? Because of this, this should be set to 0x100000.

Fixed Base Address

This will be automatically envoked by the linker. Set this to **Generate a relocation section (/FIXED:NO)**.

Linker > Command Line

Additional options

Add **/ALIGN:512** to the **Additional options** text box. This is required to insure proper section alignment. **Not doing so will cause problems executing the kernel or triple faults**.

The command line

Thats all! Compare your command line with the following. You may have additional options depending on your envirement setup.

```
/OUT:"A:\KRNL32.EXE" /INCREMENTAL:NO /NOLOGO /LIBPATH:"..\Lib\\" /MANIFEST:NO
/NODEFAULTLIB /MAP:"Kernel.map" /SUBSYSTEM:NATIVE /DRIVER /OPT:REF /OPT:ICF /ENTRY:"kernel_entry"
/BASE:"0x100000" /FIXED:No /ERRORREPORT:PROMPT
```

Executing the PE Kernel

I do not plan on going over the entire format of executable files here. Not until we cover program and task managers, anyways.

The problem, however, is that MSVC++ can only output COFF and PE file formats. Because of this, we have to find a way of parsing it from within our Stage 2 bootloader.

Because I do not plan on describing the PE format in detail yet, I will first describe its basic format, and how the code works. Lets take a look!

File Format

Once we have loaded a file image into memory, it is simply a direct copy of the image file on disk. Because of this, in order to parse the file, all we need to do is read it from where we loaded it from memory.

Understanding how to parse file formats are very important. Remember that the FIRST byte within the FIRST structure actually represents the FIRST byte from where it is loaded at in memory.

_IMAGE_DOS_HEADER

This structure is the very first structure within the PE file.

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    USHORT e_magic;         // Magic number (Should be MZ)
    USHORT e_cblp;          // Bytes on last page of file
    USHORT e_cp;            // Pages in file
    USHORT e_crlc;          // Relocations
    USHORT e_cparhdr;       // Size of header in paragraphs
```

```
    USHORT e_minalloc;      // Minimum extra paragraphs needed
    USHORT e_maxalloc;      // Maximum extra paragraphs needed
    USHORT e_ss;            // Initial (relative) SS value
    USHORT e_sp;            // Initial SP value
    USHORT e_csum;          // Checksum
    USHORT e_ip;            // Initial IP value
    USHORT e_cs;            // Initial (relative) CS value
    USHORT e_lfarlc;        // File address of relocation table
    USHORT e_ovno;          // Overlay number
    USHORT e_res[4];        // Reserved words
    USHORT e_oemid;         // OEM identifier (for e_oeminfo)
    USHORT e_oeminfo;       // OEM information; e_oemid specific
    USHORT e_res2[10];      // Reserved words
    LONG   e_lfanew;        // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

We do not need to fully understand this yet, until we create a full PE loader. For now, because we are only looking for the entry routine address, we need to find the entry routine address from the _IMAGE_FILE_HEADER structure, which contains the start of the PE header.

The address of the _IMAGE_FILE_HEADER structure is inside of the **e_lfanew** member of the _IMAGE_DOS_HEADER. So, in order to access this member, we refrence the byte offset from where it is loaded in memory:

```
mov    ebx, [IMAGE_PMODE_BASE+60]       ; e_lfanew is a 4 byte offset address of the PE header; it is 60th byte. Get it
add    ebx, IMAGE_PMODE_BASE            ; add base
```

Yey! Noe EBX contains the starting address of the _IMAGE_FILE_HEADER structure. This assumes our PE kernel image was loaded at **IMAGE_PMODE_BASE**.

Real Mode DOS Stub Program

Okay then! Lets look back up at the PE file image structure again (The above picture.) Notice how a DOS stub program is right after the _IMAGE_DOS_HEADER. This is a useful program, actually. This is the program that displays "This program cannot be run in DOS Mode", if you try to execute a Windows program from within DOS.

It is possible to change the program that is executed. This allows us to embed our own program to execute, instead of the dull default one. We do this is MSVC++ using the **STUB** command line option. For example:

```
/STUB=myprog.exe
```

As long as **myprog.exe** is a 32 bit application, MSVC++ will embed that program as the DOS stub program, insead of the dull default one. Cool? This can be usefull for a verity of reasons. Who knows--Perhaps provide a specialized DOS version of your program?

Because our kernel is an EXE file, it is possible for users to double-click and attempt to run it from Windows. Instead, this DOS stub program will run instead. Cool, huh?

Anywhoo...Because the size of this program is not constant, we need to jump over it to the next section--the _IMAGE_FILE_HEADER. This is why we needed to get the location of _IMAGE_FILE_HEADER from the _IMAGE_DOS_HEADER struct.

_IMAGE_FILE_HEADER

```
typedef struct _IMAGE_FILE_HEADER {
    USHORT  Machine;
    USHORT  NumberOfSections;
    ULONG   TimeDateStamp;
    ULONG   PointerToSymbolTable;
    ULONG   NumberOfSymbols;
    USHORT  SizeOfOptionalHeader;
    USHORT  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Okay...Remember that EBX now contains the starting address of this structure. This structure is useful, but not for what we need. We need a way of executing the entry point routine, remember? Knowing that this struct is 24 bytes in size, and the _IMAGE_OPTIONAL_HEADER structure is right after it, we can just skip this structure for now:

```
mov    ebx, [IMAGE_PMODE_BASE+60]       ; e_lfanew is a 4 byte offset address of the PE header; it is 60th byte. Get it
add    ebx, IMAGE_PMODE_BASE            ; add base

; EBX now points to beginning of _IMAGE_FILE_HEADER. Jump over it to the next section (_IMAGE_OPTIONAL_HEADER)
add     ebx, 24
```

_IMAGE_OPTIONAL_HEADER

```
struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    USHORT  Magic;
    UCHAR   MajorLinkerVersion;
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;
    ULONG   SizeOfInitializedData;
    ULONG   SizeOfUninitializedData;
    ULONG   AddressOfEntryPoint;                    << IMPORTANT!
    ULONG   BaseOfCode;
    ULONG   BaseOfData;
    //
    // NT additional fields.
    //
    ULONG   ImageBase;
```

```
    ULONG   SectionAlignment;
    ULONG   FileAlignment;
    USHORT  MajorOperatingSystemVersion;
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;
    USHORT  MinorSubsystemVersion;
    ULONG   Reserved1;
    ULONG   SizeOfImage;
    ULONG   SizeOfHeaders;
    ULONG   CheckSum;
    USHORT  Subsystem;
    USHORT  DllCharacteristics;
    ULONG   SizeOfStackReserve;
    ULONG   SizeOfStackCommit;
    ULONG   SizeOfHeapReserve;
    ULONG   SizeOfHeapCommit;
    ULONG   LoaderFlags;
    ULONG   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

THIS is an important structure. While its name states that it is "optional"--It is not, so do not worry about that. It is a required structure for all PE programs.

The important member here is **AddressOfEntryPoint** which contains...erm... The address of the entry point routine. For example...main(), mainCRTStartup(), whatever suits your needs.

Knowing that EBX points to the beginning of this structure, all we now need to do is refrence EBX+AddressOfEntryPoint. Read from that location, and we have the beginning address of the starting routine to call. After we get this address, all we need to do is far jump to that location, and we effectivly call our C++ entry point!

Putting it together

Now that everything is set up, lets try to put this all together.

Remember that the image is loaded at IMAGE_PMODE_BASE:

```
        mov     ebx, [IMAGE_PMODE_BASE+60]      ; e_lfanew is 60th byte. Get it
        add     ebx, IMAGE_PMODE_BASE           ; Add base address. EBX now points to file sig (PE00)
```

Because the image is loaded at PMODE_IMAGE_BASE, that is where the first byte of the first structure--_IMAGE_DOS_HEADER, is located. Remember that the e_lfanew member of the _DOS_IMAGE_FILE structure contains the address of the _IMAGE_FILE_HEADER. **Because this is an offset address (Assuming base 0), we have to add the base address to where we loaded it in memory.**

```
        ; jump over to optional header (Although it isnt optional o.0 )

        add     ebx, 24
        mov     eax, [ebx]              ; _IMAGE_FILE_HEADER is 20 bytes + size of sig (4 bytes)
        add     ebx, 16                 ; address of entry point is now in ebx
```

Now EBX points to the beginning of _IMAGE_FILE_HEADER. The first lines jumps over this section (As we do not need it now). So, **after the first instruction here, EBX now points to the beginning of the _IMAGE_OPTONAL_HEADER structure**, where we can begin looking for the **AddressOfEntryPoint** member. This member is 16 bytes from the start, so we add 16 to EBX.

Now, EBX contains the address of the entry point routine. Before calling it, however, we need to add the image base address to the entry point address. That is, the entry point address is just an offset.

Looking back at the _IMAGE_OPTIONAL_HEADER structure, we can see the **ImageBase** member. This is 12 bytes (A ULONG is 4 bytes) from **AddressOfEntryPoint**. Knowing EBX already points to **AddressOfEntryPoint**, this is very easy:

```
        mov     ebp, dword [ebx]        ; store entry point address
        add     ebx, 12                 ; ImageBase member is 12 bytes from AddressOfEntryPoint member
        mov     eax, dword [ebx]        ; gets image base
        add     ebp, eax                ; add image base to entry point address
```

Now that ebp contains the entry point address, call it:

```
        call    ebp     ; Execute Kernel
```

Not to hard, huh? Notice that we do not need to specify the code selector (0x8) here. The reason is that CS already contains 0x8.

Developing a C++ Runtime Envirement for MSVC++

As you know, we cannot use the runtime that was provided with Windows. The reason is fairly simple. The C++ Windows runtime relies heavily on an existing Windows Operating system. Because we are developing a new operating system, this runtime is nonexistant.

Because of this, We have to create our own C++ runtime code. This can get tricky. Alot of C++ features requires the use of a runtime. However, because we have disabled the runtime, the compilier will generate interesting errors when using these features. Other times are simply unpredictable, and may cause a triple fault.

Lets look at this, for a moment. In applications, what calls main()? The runtime library. What calls and initialzes all global objects? The runtime library. What provides certain keyword supports that which tie with the system (such as new and delete)? The runtime library. What sets up the initil stack information? Again: **The runtime library.**

Not defining a runtime library can cause unpredictable results. For one, global and static objects will never be initialized. Another problem is that the use of certain keywords are unpredictable. Global and static objects will never be deallocated. Also, the compilier relies on certain routines--useually defined

by the standard runtime. Defining and calling virtual functions may become unpredictable. The calling of pure virtual routines will immediately crash. And, say goodby to **new, delete, typeid, and exceptions**.

To make a story short: Creating a small C++ runtime is essental to get C++, the language itself, to even work properly for us.

Global Operators

You will need to define the global new and delete operators. The problem, however, is that we have no memory manager to work around with. Because of this, for now, don't do anything:

```
void* __cdecl ::operator new (unsigned int size) { return 0; }
void* __cdecl operator new[] (unsigned int size) { return 0; }
void __cdecl ::operator delete (void * p) {}
void __cdecl operator delete[] (void * p) { }
```

Now we can use the **new** and **delete** keywords without error--although they do absolutly nothing...yet, anyways.

Pure virtual function call handler

Pure virtual functions are functions declared in the class, but contain no definition. Their primary purpose is to **force** derived classes to overload that function.

It is not possible to call a pure virtual function directly through normal means. Calling a pure virtual function will result into undefined behavior, because that function does not actually exist--It was never defined.

If a pure virtual function has some how been called, the compilier attempts to use **_purecall()** as the call handler. If this does not exist, the result is unpredictable--That is, a triple fault.

Because of this, our C++ runtime will need to define it:

```
int __cdecl ::_purecall() { for (;;); return 0; }
```

Normally you would want to assert () when this happens as it should never happen.

Floating Point Support

Everything is working great with our new MSVC++ kernel. That is, until we try to compile **float i=2/2;** and BAM! We get hit by errors. More specifically, unresolved external errors...Our favoriate ;)

There is nothing wrong with this...Just like with using the new and delete operators without them being defined. Simulariy, MSVC++ needs some routines defined for working with floating point math.

_fltused

This is used by MSVC++ to determin if floating point is currently in use. This should be set to 1 and must be given C linkage if building for C++:

```
extern "C" int _fltused = 1;
```

_ftol2_sse()

Depending on your optomization level, MSVC++ may embed calls to _ftol2_sse() to convert a float to a long. I wont be using SSE here, so I will just write my implimentation using the FPU:

```
extern "C" long __declspec (naked) _ftol2_sse() {

        int a;
#ifdef ARCH_X86
        _asm    fistp   [a]
        _asm    mov     ebx, a
        _asm    ret
#endif
}
```

Other routines

I have defined other routines, which are _CIcos(), _CIsqrt(), and _CIsin(). Until we can verify these routines are needed, I will keep them inside of our runtime library.

Initializing globals and static data

Everything is good so far, except what about globals? Remember that the runtime is responsible for executing all global executed routines and initializing all global and static objects? Because we have disabled the runtime, we have to do it.

To do this, we have to watch exactally how MSVC++ handles the constructors (ctors).

MSVC++ uses a special section (Simular to .data, .bss, .text, etc.) within the final binary image for ctors. When the MSVC++ compilier finds an object that must be executed by the startup code, it places a **Dynamic initializer** inside of this section. This means that -- for every dynamic initializer that needs to be executed at startup, they can all be found by looking inside of this special section.

This section is the **.CRT** section. **These dynamic initializers are a an array of 4 byte function pointers, which are stored within .CRT** So, if we can find a way of parsing this section, we can call each function pointer that MSVC++ set for us to call each routine that needs to be called at startup.

We cannot do this with C++ alone, however, as these section names are very MSVC++ specific. Also, as we are building without any runtime, the .CRT section is currently nonexistant. We have to add this section ourselves. The **only** way of doing this is using the preprocessor.

Naming Conventions

Alright...This section can be a little confusing. The section names used in MSVC++ are very strange. Serously--*.CRT$XCU*? What were they thinking!?

Actually, these section names do have a purpose. the section names are composed of two parts, seperated by the dollar sign "$". **The first part is used as the base section name**. The second part indicates where it is located at in the final image.

That is, we can think of a section name having this format:

```
.section_name$location_name
```

The **section name** can be .code, .data, .bss, .CRT, or any other section name. The **location_name** is an alpabitic name that represents where we are in the section. For an example, in **.CRT$XCA**, .CRT is the section name, XCA is its location in that section. It does not matter what this location name is; the important thing to remember is that it is in **alphabitical order**.

Here is an example:

```
        .CRT$XCA
        .CRT$XCU
        .CRT$XCZ
```

Notice the sections are arranged in the letters of the alphabit. The arrangement of the second part determins their location, at which they are stored within the final image. In the above, **.CRT$XCA will be first, .CRT$XCU is second, .CRT$XCZ is the last one.**

Lets look at this anoher way, by mixing these up...

```
        .CRT$XCZ
        .CRT$XCA
        .CRT$XCU
```

The same thing applies here. .CRT$XCA is the first section, again. This example illustrates that: **The order of these sections depends on their alphavalue--a comes before z, so .CRT$XCA comes before .CRT$XCZ.** Notice the last character. Not too hard, I hope :)

We will be able to see these sections inside of the linker map, when we set them up.

Creating new segment names

In order to create a new section, we need to use the **#pragma data_seg()** directive. This directive insures that all data allocated after it is placed within this new section.

This directive takes the form:

```
        #pragma data_seg( ["section-name"[, "section-class"] ] )
```

"section_class" is retained for compatibility purposes only, and is now ignored by MSVC++.

The important part, thus, is the first parameter--"section-name", which gives the new section name a...er...name:

```
#pragma data_seg(".CRT$XCA")

// All varables allocated here are now placed within the .CRT$XCA section, rather then .data section
```

To go back to the default (.data) section, use this pragma without parameters:

```
//! Select the default data segment again (.data) for the rest of the unit
#pragma data_seg()
```

Merging Sections

By default, we are unable to read or write to the .CRT section. However, we can read and wite to the .data section just fine. What we want is the same **permissions** for **both** section names.

We can fix this by combining the two sections together, which will insure both sections have read and write abilities:

```
//! Now, move the CRT data into .data section so we can read/write to it
#pragma comment(linker, "/merge:.CRT=.data")
```

Okay... So lets see... All global initializer routines are stored as function pointers within the .CRT$XCU section of the binary image. We can declare a section right before and after this section, and insure they are right after each other, thanks to the naming convention, and the linker. Because they are right next to each other, we can effectivly declare a varable to point to these sections--effectivly pointing to the first and last function pointer in the initializer array. Lets look at this next...

Initializing globals - Setup

Lets look at actual code, and break it down:

```
//! Function pointer typedef for less typing
typedef void (__cdecl *_PVFV)(void);
```

We first typedef a function pointer to improve readability. This function pointer is used to point to each global initializer.

```
/**
*       MSVC++ creates dynamic initializers and deallocaters, which help us in calling the routines.
*       The compilier and linker bind all dynamic initializers into a function pointer table inside a
*       section called .CRT$XCU.
*/

// Standard C++ Runtime (STD CRT) __xc_a points to beginning of initializer table
```

```
#pragma data_seg(".CRT$XCA")
_PVFV __xc_a[] = { NULL };
```

The above code creates the .CRT$XCA section. By declaring this with an "A", we insure this will be right before the next .CRT section defined, so it is guaranteed to be right before our .CRT$XCU section.

__xc_a is a standard MSVC++ CRT name used as a pointer to the beginning of the initializer list, stored in .CRT$XCU

```
//! .CRT$XCU is located here.
```

Our .CRT$XCU is giaranteed to be located before .CRT$XCZ and after .CRT$XCA because of the naming convention used.

```
//! Standard C++ Runtime (STD CRT) __xc_z points to end of initializer table
#pragma data_seg(".CRT$XCZ")
_PVFV __xc_z[] = { NULL };
```

This is the .CRT$XCZ section. Again, because of naming conventions, it is guaranteed to be right after the initialzer list within .CRT$XCU. By defining a function pointer here, **It is guaranteed to point to the last initializer routine - 1, within the initializer array within .CRT$XCU.** __xc_z is the standard name used by the MSVC++ CRT.

```
//! Select the default data segment again (.data) for the rest of the unit
#pragma data_seg()
```

For all other data, we want to use .data section, so switch back to that section...

```
//! Now, move the CRT data into .data section so we can read/write to it
#pragma comment(linker, "/merge:.CRT=.data")
```

...And merge the .CRT section with our .data section. This insures we can access the .CRT section from the .data section. To initialize each routine, just loop through each function pointer and call it. **Warning: Beware of null function pointers. Calling a null function pointer will result into an invalid jump to some random location in memory, which will result in triple fault.**

```
void __cdecl _initterm ( _PVFV * pfbegin, _PVFV * pfend )
{
    // Go through each initializer
    while ( pfbegin < pfend )
    {
      // Execute the global initializer
      if ( *pfbegin != NULL )
            (**pfbegin) ();

        // Go to next initializer inside the initializer table
        ++pfbegin;
    }
}

// This initializes all global initializer routines:
_initterm(__xc_a, __xc_z);
```

Cleaning up the envirement

Yippee! We now have all global initializer routines being executed. Whats next? Cleaning up after ourselves, of course.

The good news is that this is much easier to work with then the initializer routines. All we need to define a location to store an array of global deinitializer function pointers at some location in memory:

```
//! function pointer table to global deinitializer table
static _PVFV * pf_atexitlist = 0;

//! Maximum entries allowed in table
static unsigned max_atexitlist_entries = 32;

//! Current amount of entries in table
static unsigned cur_atexitlist_entries = 0;
```

These are our function pointers that we use to keep track of where we are at in the global deinitializer array.

We define where these arrays is located. MSVC++ adds deintializer code for each global object, that adds a function pointer to the global deinitializer array. It does this by calling a specially defined routine, **atexit ()**.

Note: MSVC++ requires this routine. Not defining this routine will result in errors when defining a dtor of any kind.

The actual routine is simple. Remember that, for each global object, MSVC++ embeds code that will call this routine. The dtor as an object is passed into this routine as a paramater. Because of this, all we need to do is to add it to the end of our dtor array:

```
//! For every global object created, MSVC++ calls this routine with a function ptr to each dtor
int __cdecl atexit(_PVFV fn)
{
        //! Insure we have enough free space
        if (cur_atexitlist_entries>=max_atexitlist_entries)
                return 1;
        else {

                // Add the exit routine
                *(pf_atexitlist++) = fn;
                cur_atexitlist_entries++;
```

```
        }

        return 0;
}
```

So...Now that we have a way of adding dtors to the list (Remember that MSVC++ automaticlly does this through our function), All we need to do is initialize the original function pointer array:

```
void __cdecl _atexit_init(void)
{
    max_atexitlist_entries = 32;

        // Warning: Normally, the STDC will dynamically allocate this. Because we have no memory manager, just choose
        // a base address that you will never use for now
    pf_atexitlist = (_PVFV *)0x500000;
}
```

Not too hard, I hope :)

There are alot of possibly new concepts for our readers, however, so all of this may be better in an example demo.

The Entry Point

Okay, so lets see...We have covered getting the entry address from the PE image. The entry point routine is immediately executed by the 2nd stage loader. We have set up our entry point to be **kernel_entry** so lets define it:

```
void _cdecl kernel_entry () {
```

We need to insure the registers and stack are setup before any code is executed. This is very important to insure we refrence the correct descriptors in the bootloaders' GDT. We also need to setup the stack, as C++ uses the stack reguarly.

```
#ifdef ARCH_X86
        _asm {
                cli                                     // clear interrupts--Do not enable them yet
                mov ax, 10h                             // offset 0x10 in gdt for data selector, remember?
                mov ds, ax
                mov es, ax
                mov fs, ax
                mov gs, ax
                mov ss, ax                              // Set up base stack
                mov esp, 0x90000
```

Next, we store the current stack frame pointer. This will insure that any routines we call has a place to return to.

```
                mov ebp, esp
                push ebp
        }
#endif
```

Now, we call our main() routine!

After calling main(), just halt the system, to insure we don't return (As there is nowhere to return to.)

```
        //! Execute global constructors
        InitializeConstructors();

        //! Call kernel entry point
        main ();

        //! Cleanup all dynamic dtors
        Exit ();

#ifdef ARCH_X86
        _asm cli
#endif
        for (;;);
}
```

Thats all thats needed! As long as the entry poin is set to **kernel_entry**, This routine will be placed at the starting base address--Which should be set to 1 MB

Demo

Demo Download (MSVC++)

This demo loads and executes a 32 bit Kernel written in MSVC++ 2005. It also includes all of the source code in this tutorial, as well.

Conclusion

Yey! Alot of concepts in this tutorial is farily simple, isn't it? We covered setting up MSVC++ 2005 so that we can use the compilier for use in operating system Kernels. We also looked at creating a basic C++ runtime envirement, ctor and dtor calling, virtual function handling, and global operators.

In the next few tutorials, we are looking at creating envirements for different compiliers. This tutorial has covered setting up MSVC++ 2005.

This tutorial was hard to write--and I am yet to finish it. There are simply so many options that MSVC++ has, and describing these options in detail will take a long time. I wanted to find a way of combining context, not just a "do this do that" option setting list. I am still deciding on a format style for that. I hope I did Okay :)

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!

Operating Systems Development Series

Operating Systems Development - Basic CRT and Code Design

by Mike, 2008

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

Introduction

Woohoo!! Its finally time to start developing our kernel and hardware abstraction layer (HAL).

In the previous tutorial, we have taken a look at putting basic kernel concepts together, and looking at the different basic kernel design layouts. We have also decided that we will be developing a hybrid kernel design for our operating system, as it uses some concepts derived from Microkernels and Monolithic kernel designs. This will allow us to look at some concepts from both worlds.

In this tutorial, we will start building our Kernel program, and start developing our Hardware Abstraction Layer library. At the moment, we have our system set up so tht we can develop the kernel and Hardware Abstraction Layer within a higher level programming language - C or C++, depending on the compilier being used.

To keep compatability with C compiliers, we will be using C instead of C++. However, I might be developing C++ versions of the source as I personally prefer C++ over C :)

So, heres what is on the list for this week:

- Promoting Good Coding Practices
- Code Design and Layout
- Abstracting data types and basic declarations
- CRT: _null.h
- CRT: size_t.h
- CRT: ctype.h and cctype
- CRT: va_list.h and stdarg.h/csdtarg
- Demo: Writing Debug Printf (Will be uploaded soon)

...Thats it! This tutorial only covers the basic setup of the HAL and Kernel.

Lets start!

Before we Begin...

This is our first step away from the bootloader world. Within our bootloader, we did not need to worry much about portability nor system dependency. After all, the bootloader - by its very nature - is very system dependent.

However, now we have made the jump from the bootloader to the Kernel, being developed in C or C++. This is also the beginning of our own runtime library, and Hardware Abstraction Layer (HAL)--Alot of things going on, huh?

However, it does not get easier. Operating Systems can get very large in size. Because we do not know how large this system will be, we need to stress good coding practices from the start. Many development projects fail. It is not because it is to complex, however. Any project can be made with less complexity if designed right. This is what I want to look at next...

Pandora's Boxes

The truth is, simply put, code is evil. Code can get very disorderly and ugly. It is this that adds on more complexity do to the chaotic and recursive nature of code and design. Dont get me wrong, we will still need to rewrite alot of code. The reason for this is because there is no right design. This is what makes code chaotic: After the initil write and rewrites the code itself can be very malformed. This has the tendensy to stop an entire project, specifically large projects, as the rest of the system needs to rely on the chaotic nature of this ugly written, and poorly designed code. Its like a plauge...Where it starts in one area of the system and spreads to the rest of the software.

How do we stop this from happening?

"People say Pandora's Box was evil, but they're wrong. The stuff inside it was evil. The box ain't nothin' but a box." - Anonymous

As long as the code is contained within a nice little box, it does not matter how disordley or ugly the code gets on the inside. As long as this code is tucked away within a nice little box where no one can see it. On the inside of the box can contain deamons, creatures, and other things -- we dont care as that cannot get out of the box. After all, all we see is the box. We dont need to care how it works, it simply works.

This, my friends, is the bases of isolation and containement. That is, ***Encapsulation***, and the bases for nearly all of software engineering.

We first write what the inside of the box does. After this, when this module is completed, we close the box and connect it to the rest of the system. But, you *never, ever* open the box after its been closed. Doing so lets out all of the evils within the box, as it

breaks encapsulation. Once you open a box, **it will infect as many peices of code as it can though compile, linker, or runtime errors, and leave the whole project into a big mess.**

A solid, well designed system will treat all of its components as isloated ("encapsulated") boxes connected together and nested within other larger boxes.

Encapsulation is a very important concept in software engineering. Even if you are not an Object Orianted Programmer, the concept of encapsulation is still there.

Interface and Implimentation

Using the Pondora's Box anology again, we can say that the "interface" is the box, and the "implimentation" is what is within the box. The interface ("public") part of the box is the connection from that box to the outside world. It is what connects our box to other boxes within this subsystem. The interface itself containes all of the function prototypes, structures, classes, and other definitions that the box exposes to the outside world so the outside world can use and interact with the box. This is the **Interface**. All of the evil code that dwells within this box that define the module, all of its functions, class routines, etc. is the modules **implimentation**.

It is important to construct each box ("component") with an interface that is simple and to the point. It should also be clear at what each componenet does. In C, the global namespace can get very cluttered with tons of routines. Because of this, it is important to name these routines and interfaces to help clearly identify them. You will also need to insure that the implimentation detailes of the box (The "private" part) are kept as private members. Putting any part of this in the interface is bad, as it can open up the box (Which is bad.)

In C, we can insure routines stay as part of the implimentation by using the **static** keyword. Interfaces can be made by using the **extern** keyword. Within C++, It is encouraged to use classes, with the **private, public,** and **protected** keywords.

Get Ready

We will be using the above concepts with developing our system to promote good programming practices with large scale software.

Because portability between compiliers is a concern, we will be developing the system using the C programming language. Please keep in mind, however, that you may use C++ if you prefer.

Our primary focus is that of expandability and portability. Because of this, we will be hiding all hardware dependent implimentations behind its own little box - The **Hardware Abstraction Layer (HAL).** Because the C++ startup runtime code it compilier dependent, we will put that in its own little box - The **CRT (C++ Runtime) Library**. All of this will be completely independent of the rest of the system.

Remember: The key is isolation. It does not matter how they are isolated, so long as the interfaces are clean and nice. The more isolation you have, the better, and remember to never open a box once it has been closed.

With all of this in mind, lets take the first step into our system...

Code Layout and Design

This tutorial containes our most complex demo so far. Because of this, I would like our readers to open up the demo source, and follow along with the tutorial for better understanding of everything.

Code Design

It is very important to understand why we have chosen this structure for this series. The primary reason is that of **encapsulation**, where each directory containes a separate **library module**. That is, **Each of these modules is a pondora's box**. It is **extremily** important to keep these modules as separate as possible in order to maintain code stability, structure, and portability. In order to do this, I have decided to treat each module as independent library modules.

```
Our two stage bootloader (We have already constructed this from our previous tutorials)
=======================================

SysBoot\

      Stage1\                          - Stage1 bootstrap loader
      Stage2\                          - Stage2 KRNLDR bootloder

Our System Core
=======================================

SysCore\

      Debug\                           - Pre-Release complete builds
      Release\                         - Release builds
      Include\                         - Standard Library Include directory

      Lib\                             - Standard Library Runtime. Outputs Crtlib.lib or Crtlib.dll.
      Hal\                             - Hardware Abstraction Layer. Outputs Hal.lib or Hal.dll.
      Kernel\                          - Kernel Program. Outputs Krnl32.lib or KRNL32.EXE
```

The only thing that does not need to be built as a library module are the files within the Include/ Directory. As they are only header files, they should never have the need to contain implimentations. Because of this, there is no box to open.

As with applications, I have decided to make the C++ runtime code the first code to be executed. In other words, the bootloader does NOT execute the kernel. Instead, it executes the runtime code (CRTLIB), which sets up the envirement for the kernel, and then executes the kernel.

_null.h

Yey!! Its time to start getting down to the nitty griddy of the tutorial!

About C++ includes...

If you are using C++, you might be interested about the library header files. That is, in C++, the appended *.h is dropped, and a **c** is prepended to all C headers. So, instead of **#include <stdlib.h>**, C++ uses **#include <cstdlib>** We would like to encourage creating an interface compatable with both languages. However, you might be wondering how do we do that?

Its very simple, actually. In all compiliers standard include/ directory, you will see different variants of the same file. i.e., you will see **stdlib.h** and **cstdlib**. **cstdlib** is simply a header file that #includes **stdlib.h** and no more. We will be doing the same with our library.

This will allow the developers using C to use **stdlib.h**, while our C++ developers can still use **cstdlib**. This way we can both encourage good habits.

Back on topic

The first abstraction I would like to look at is NULL. There really is not that much to say here. However, there is one small detail: The way NULL is defined depends on whether you are using C or C++.

Within standard C, NULL is defined as (void*)0. Within C++, it is simply 0. We can determin this by using the fairly standard **__cplusplus** predefined constant:

```
// Undefines NULL
#ifdef NULL
#  undef NULL
#endif

#ifdef __cplusplus
extern "C"
{
#endif
/* standard NULL declaration */
#define NULL    0
#ifdef __cplusplus
}
#else
/* standard NULL declaration */
#define NULL    (void*)0
#endif
```

There is more in this header do to the template, but this is the important part. Everything else is quite easy.

size_t.h

About Data Hiding...

Remember the Pandora's Box theory. The data types within a box are on **implimentation detail**. Some data types are okay, however some or better kept within the implimentation. **size_t** is one of them. By keeping the implimentation details, we can modify anything we like about the data type, without effecting anything that uses that type, so long as we remain backward compatable.

Back on topic

There isnt much to say about this one...

```
#ifdef __cplusplus
extern "C"
{
#endif

/* standard size_t type */
typedef unsigned size_t;

#ifdef __cplusplus
}
#endif
```

Data Type Hiding - stdint.h and cstdint

Within the previous section, we were encouraging the importance of data hiding within an interface, However, we did not stress the importance of it with relation to portability.

Each data type has a specified size to them. However, the size of each data type completely depends on the compilier and system this is being built for. Because of this, it is important to hide the data types behind a standard interface, specifically because we are working in an envirement where Size Does Matter(tm).

stdint.h

This is a fairly big file at about 150 lines. None of it is very hard, however. It defines different integral data types that are guaranteed to be a certain size.

Lets look at the fundemetal types, as we will be using them throughout the system:

```
typedef signed char          int8_t;
typedef unsigned char        uint8_t;
typedef short                int16_t;
typedef unsigned short       uint16_t;
typedef int                  int32_t;
typedef unsigned             uint32_t;
typedef long long            int64_t;
typedef unsigned long long   uint64_t;
```

When compiling for a 32bit system, the above data types are guaranteed to be the same. That is, **uint8_t** is guaranteed to be 8 bits. **uint16_t** is guaranteed to be the size of a WORD (2 bytes), and so on. The size of the data type is encoded in its name, so we will always know its size.

There is alot more code in this file, but most of it is fairly easy.

The file **cstdint** simply #includes stdint.h. This allows us to include these declarations in two ways:

```
#include <stdint.h> // C
#include <cstdint>  // C++ only
```

Please see **About C++ includes...** section for more information of why we have done this.

ctype.h and cctype

ctype.h is a set of macros that help determin what type of character in a string is. It does this by following the different properties of the standard **ASCII Character Set**. You can get it from asciitable.com

This header file includes several macros and constants:

```
extern char _ctype[];

#define CT_UP   0x01    /* upper case */
#define CT_LOW  0x02    /* lower case */
#define CT_DIG  0x04    /* digit */
#define CT_CTL  0x08    /* control */
#define CT_PUN  0x10    /* punctuation */
#define CT_WHT  0x20    /* white space (space/cr/lf/tab) */
#define CT_HEX  0x40    /* hex digit */
#define CT_SP   0x80    /* hard space (0x20) */

#define isalnum(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_UP | CT_LOW | CT_DIG))
#define isalpha(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_UP | CT_LOW))
#define iscntrl(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_CTL))
#define isdigit(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_DIG))
#define isgraph(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_PUN | CT_UP | CT_LOW | CT_DIG))
#define islower(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_LOW))
#define isprint(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_PUN | CT_UP | CT_LOW | CT_DIG | CT_SP))
#define ispunct(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_PUN))
#define isspace(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_WHT))
#define isupper(c)      ((_ctype + 1)[(unsigned)(c)] & (CT_UP))
#define isxdigit(c)     ((_ctype + 1)[(unsigned)(c)] & (CT_DIG | CT_HEX))
#define isascii(c)      ((unsigned)(c) <= 0x7F)
#define toascii(c)      ((unsigned)(c) & 0x7F)
#define tolower(c)      (isupper(c) ? c + 'a' - 'A' : c)
#define toupper(c)      (islower(c) ? c + 'A' - 'a' : c)
```

Pretty simple stuff so far. The above macros may be used to determin and modify individual characters.

For C++, There is also **cctype** that may be used instead of **ctype.h**.

va_list.h and stdarg

These are standard headers containing macros for accessing unnamed parameters whithen a variable argument list.

va_list.h

va_list.h abstracts the data type used for varable leangth parameter lists.

```
/* va list parameter list */
typedef unsigned char *va_list;
```

stdarg.h and cstdarg

This is our final basic library include file that we will look at. It defines some nice macros that we may use for C and C++ varable length parameter lists.

These macros are fairly tricky, so lets look at them one at a time.

VA_SIZE

```
/* width of stack == width of int */
#define STACKITEM       int

/* round up width of objects pushed on stack. The expression before the
& ensures that we get 0 for objects of size 0. */
#define VA_SIZE(TYPE)                                           \
        ((sizeof(TYPE) + sizeof(STACKITEM) - 1) \
                & ~(sizeof(STACKITEM) - 1))
```

This is a little tricky. VA_SIZE returns the size of the parameters pushed on the stack. Remember that C and C++ uses the stack to pass parameters to routines. On 32bit machines, each stack item is normally 32 bits.

va_start

```
/* &(LASTARG) points to the LEFTMOST argument of the function call
(before the ...) */
#define va_start(AP, LASTARG)    \
        (AP=((va_list)&(LASTARG) + VA_SIZE(LASTARG)))
```

The standard va_start macro takes two parameters. AP is a pointer to the parameter list (of type va_list), and LASTARG, which is the last parameter in the parameter list (The parameter right before the ...).

All this routine does is get the address of the last parameter, and adds the size of the parameter size to that address. If the stack size is 32, then all it does it add 32 to the last parameters address on the stack, which is where the first parameter in the parameter list is at.

va_end

```
/* nothing for va_end */
#define va_end(AP)
```

Thre isnt much to do here.

va_arg

```
#define va_arg(AP, TYPE)         \
        (AP += VA_SIZE(TYPE), *((TYPE *)(AP - VA_SIZE(TYPE))))
```

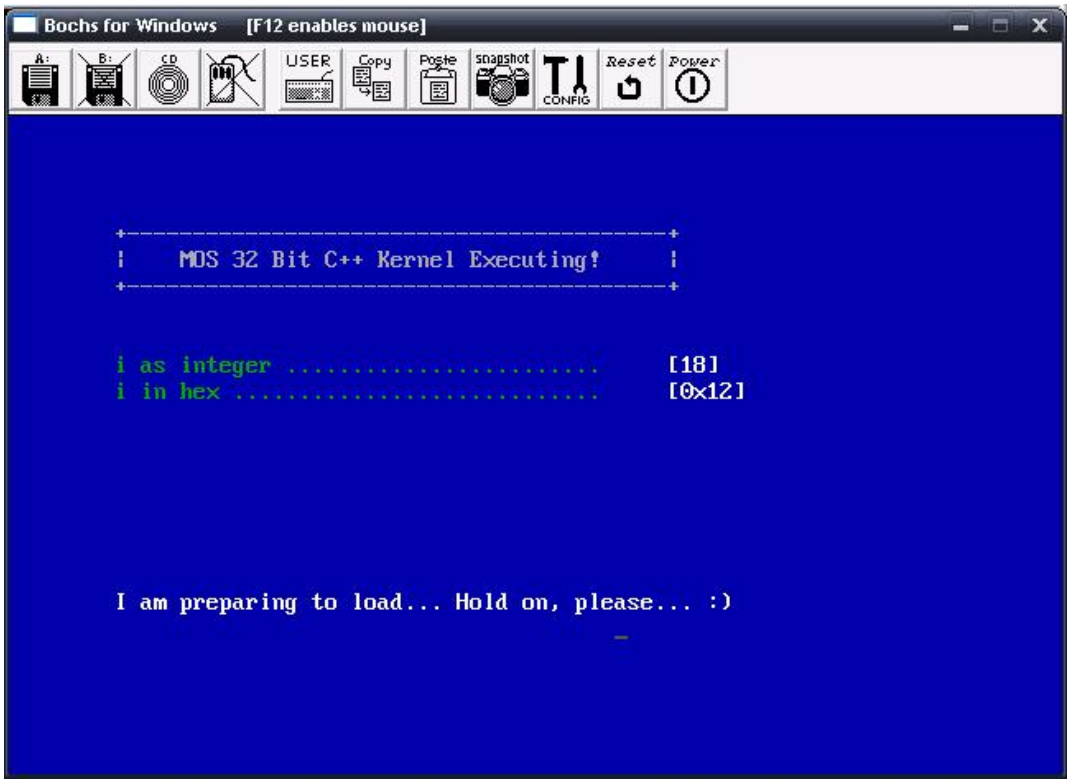
This is a little tricky. va_arg() returns the next parameter in the parameter list. AP should contain the pointer to the parameter list that we are working with. TYPE containes the data type (int, char, etc.)

All we need to do is add the number of bytes of the data type (TYPE) to the varable parameter list pointer (AP). This insures the varable parameter list pointer now points to the **next** parameter in the list.

After this, we derefrence that data that we have just passed (by incrementing the pointers location) and return that data.

Demo

Jeeze, there is alot of stuff in this tutorial. The demo is even more funner, as it developes its own printf() routine that we can use for debugging and displaying text.



This demo is fairly complex. I wanted to provide some basic C++ library routines, as well as a way to provide displaying text for debugging purposes. With this, all of the project files include the libraries for the Hardware Abstraction Layer (HAL), Kernel, and C++ Library code. In other words...It looks more complex then it actually is :)

Demo Download (MSVC++)

Conclusion

Wow--Alot of things in this tutorial! We covered a bit of stuff - some may be new concepts to some of our readers.

This tutorial I personally did not want to write. I wanted to find a nice and good way of covering some basic ground, theory, and design concepts before diving into the code. We have also looked at a few basic standard library headers as well, and looked at the basic structure of our system.

Now that the basic neccessities are taken care of, in the next tutorial we will start building the actual Kernel and Hardware Abstraction Layer (HAL). We will cover error and exception handling theory and concepts, interrupt handling, the Interrupt Descriptor Table (IDT), and how to trap processor exceptions so it will no longer triple fault. We can also build our own super 1337 BSoD too ;)

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 13 Home Chapter 15

Operating Systems Development Series

Operating Systems Development - Errors, Exceptions, Interruptions

by Mike, 2008

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

Please note: This tutorial covers software interrupt handling, not hardware interrupt handling. If you are looking for hardware interrupts, please see our 8259A PIC tutorial. The software side of handling hardware interrupts is discussed here.

Introduction

Welcome back! :)

In the last tutorial we have covered the ground basics and design of our system. Yes, pretty basic and easy so far, right?

You might be wondering when we will start getting into the system-level code again. Well.... *ahem* ...welcome back :)

This tutorial will cover a very important concept: **Error Handling**. Error Handling involves alot more then simply handling problems, but **catching** them as well. This is where **Exception Handling** comes in. Because **Exception Handling** requires **interrupts**, we will also cover **interrupt handling** as well.

Interrupts are architecture dependent. Because of this, we will be developing an interface for managing interrupts through our uber 1337, yet very empty (at the moment) **Hardware Abstraction Layer**, and interfacing with our Kernel to install our own **Trap Gates** which will be used to catch processor exception errors, and allow us to prevent triple faults now, and forever, while remaining completely hardware independent.

Sound fun? So, heres whats up:

- Error Handling
- Exception Handling
- IRs, IRQs, ISRs
- Gates: Traps, Interrupts, Tasks
- IDTs and IVTs
- IDTR processor register
- LIDT and SIDT instructions
- FLIHs and SLIHs
- How interrupts work, stack, error codes
- Developing a kernel panic error screen. ie, BSoD

...Alot of stuff going on here, so lets get started, shall we?

Errors, Errors, Errors

Okay, lets face it: No one is perfect. With computers, this is even truer. As we are working in the wonderful world of kernel land, things are even worse as a simple error can cause unpredictable software to hardware problems.

I am sure a lot of our readers have already experienced this through Triple Faults. In applications programming, we are not working directly with the hardware. Because of this, there are less problems that can result to errors. In kernel land, things are a bit different. Triple faults are caused do to errors with our instructions or data. If there is a problem that the processor cannot resolve, it reboots the system before it gets worse.

Triple faults and no error handling is very bad in OS development, as the problems can get much worse?rom data corruption to hardware failures, to even completely destroying the system. Knowing the importance of error handling is critical in resolving these, and insuring that our system stays stable to its end release.

Exception Handling

?xception Handling comes in two flavors: A programming language construct (For example, standard C++? try/catch/throw keywords. Some compilers also include additional keywords like _except; or mechanisms like SEH, or VEH). The other flavor is the one we are interested in: Hardware mechanisms that are designed to change (?nterrupt? the current flow of execution. The condition that changes this flow of execution is called an exception. Exceptions should only be used to signal error (exceptional) conditions, and not for conditionals that are used for normal operation.

When an exception occurs, the flow of execution changes as a subroutine (the ?xception handler? is executed. This allows the subroutine to handle the error condition in some way. Normally, the current state will be saved before the handler is called. This will allow the handler to continue execution later, if possible.

Remember that exceptions are designed from the hardware. i.e., They are hardware mechanisms. This is similar to hardware interrupts, and the bases of interrupt handling, as they are related.

Because of this, in order to understand exception handling from the hardware, we need to look at interrupts. Lets look at that next.

Interrupt Handling

Interrupts

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.

Not to hard. 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.

Other interrupts may be used to provide a way to service software as routines. These interrupts can be called by any software from within the system. This is used alot for System API's, which provide a way for ring 3 applications to execute ring 0 level routines.

Interrupts provide alot of use, especially as a way of receiving information from hardware that may change its state at asynchronous times.

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 alsu 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.

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 to hard, huh? This special routine determins 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.

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.

Interrupt Service Routines (ISRs)

Interrupt Service Routines (ISRs) is an Interrupt Handler. These are important to understand, so lets 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 then 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.

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.

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.

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.

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**, simular 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 then 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.

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 determins size
    - **01110** - 32 bit descriptor
    - **00110** - 16 bit descriptor
  - **Task Gate:** Must be 00101
  - **Trap Gate:** Of the format 0D111, where D determins size
    - **01111** - 32 bit descriptor
    - **00111** - 16 bit descriptor
- Bits 42...44: Descriptor Privedlge 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

Thats it!? Yep--Thats all there is to it ;)

All we need to do is fill in our IDT, and install it, just like what we done with the GDT. The IDT is alot more simpler then the IDT, so its even easier :) The above list is the complete descriptor format. We only need to worry about developing an interrupt gate for now, so we will only focus on that.

Interrupt Descriptor: Example

Just like with the GDT, we will create an example at the bit level to help describe exactally how everything works.

First, lets look at an example interrupt descriptor. This is going to be in shown in assembly so we can get a better view of everything.

```
idt_descriptor:
    .m_baseLow          dw    0
    .m_selector         dw    0x8
    .m_reserved         db    0
    .m_flags            db    010001110b
    .m_baseHi           dw    0
```

Yep--Thats all there is to a descriptor. Thats not that hard, is it?

Lets see how this relates to our table above by breaking it down and seeing each bit:

```
00000000 00000000 00000000 00001000 00000000 10001110 00000000 00000000
```

This is our descriptor, but in binary form. For the most part this is easy as most of it is all 0's.

The first two bytes is our **m_baseLow** member shown in the above code. Looking at the table above, we can see that this is the first 16 bits of the descriptor. **Because this is an interrupt gate, this represents bits 0-15 of the base address of the IR**. This means, if this was our field, our IR would be located at address 0. (This normally would NOT be the case, as the location of the IR varies. This works for this example, though.)

The next 2 bytes is our **m_selector** field. This is bytes 16-31 of the descriptor. Looking at our table, we can see that this represents our segment selector. Our interrupt handler containes code, so it should be using one of our code selectors. This is defined at offset 0x8 within the GDT, so that is our segment selector.

The next few bits are not used. We can see that bits 31-35 are not used, while bits 36-38 must be 0 for interrupt gates. Because of this, we can saftley say bits 31-38 are 0. This is the size of a byte, which is our **m_reserved** member.

The next byte is where the interesting stuff happens. Lets break it down, bit by bit--literally:

```
10001110
```

Okay...Right now we are at bit 39. Looking at our table above, we can see **bits 39-41 must be 0D110. If the D bit is set, this is a 32bit descriptor.** It is equal to 01110, so it is indeed a 32bit descriptor.

The next two bits (00 above) are bytes 42-45 of the descriptor, which represents the privledge level (DPL). It is 00, so the DPL is to execute at ring 0.

The final two bytes within our example are the last two bytes within the above table. This is the high 16 bits of the IR base address (Which, in our case, is 0.) This is the **m_baseHi** member displayed above.

As you can see, there really is not that much going on here. The selector is always going to be that of the code selector within the GDT (0x8 for our needs); then all we need to do is set the flag bits and the IR base address within **m_baseLow** and **m_baseHi**. We will see a complete example a little later which will help in understanding everything.

IDTR Processor Register

The IDTR register is the processor register that stores the base address of the IDT.

The IDTR register has the following format:

IDTR Register	
Bits 16...46 (IDT Base Address)	Bits 0...15 (IDT Limit)

Simple enough, huh? Notice that the **base address** of our created IDT is stored in this register. The processor uses this register to determin where our IDT is located at.

Knowing this format is very important, as it containes **both** the limit and base address. Because of this, simply giving it the base address of our idt will NOT work. This is useually resolved by creating a new structure in the format shown above like this:

```
idt_ptr:
        .limit  dw idt_end - idt_start  ; bits 0...15 is size of idt
        .base   dd idt_start            ; base of idt

; load register with idt_ptr
```

Oh, wait...How do we even access this register? Oh right...

LIDT Instruction - Loading our IDT

This instruction is used to store a new address of an IDT into the IDTR register. This instruction can only be used if the Current Protection Level (CPL) is 0 (Ring 0). It is very easy to use:

```
lidt [idt_ptr]
```

Thats all there is to it. As long as idt_base is the base address of the IDT, this will copy the address into IDTR.

SIDT Instruction - Storing our IDT

This instruction is used to store the value in IDTR into a 6 byte memory location. This instruction may be used in both ring 0 and ring 3 applications.

```
sidt [idt_ptr]
```

How Interrupts Work: Detail

Finding the interrupt procedure to call

When an interrupt or exception is fired, the processor uses the exception or interrupt number as an index into the IDT. As you know, our IDT is nothing more then an array of 256 descriptors of the format shown above. The processor performs the calculation **IDTR.baseAddress + index * 8**, where 8 is the size of a descriptor (Remember that descriptors are 8 bytes in size?), and index is the interrupt number. IDTR.baseAddress is the base address of the IDT stored within the upper bits of IDTR. This allows the processor to retrieve the base address of the descriptor index for the interrupt handler. **If the value of the calculation is greater then the IDT limit size (stored in IDTR.limit), the processor will execute a General Protection Fault (GPF)** as this will result into a call beyond the size of the IDT.

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 simular 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 simular 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 abtained 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. We will look at this next.

Inside of our interrupt handler

Because the location of our interrupt handler is stored within the descriptor, the processor is now able to execute our handler.

As you know, when the processor executes our handler, it pushes some extra information on the stack. If our handler is running in the same ring level as ours (As it will be), then we must remember that **the processor will push EFLAGS, CS, EIP** and an **Error code** on **our current** stack. This allows us to continue execution if we are able to.

Putting all of this together, when our handler is called, our stack will be set up like this:

```
+---------------+ -- Bottom of stack
|   EFLAGS      |
+---------------+
|   Return CS   |
+---------------+
|  Return EIP   |
+---------------+
|   Error Code  |
+---------------+ -- ESP points here when handler is executed.
                    If there is no error code, ESP points to return EIP
```

We use this information to return back from our handler, and to determin what caused the exception (If there is an error code.)

Inside of our interrupt handler: Error code format

If an error code is pushed on the stack when our handler is called, we can use its information to help in determing the error.

It has the following format:

- Bit 0: External event
    - 0: Internal or software event triggered the error.
    - 1: External or hardware event triggered the error.
- Bit 1: Description location
    - 0: Index portion of error code refers to descriptor in GDT or current LDT.
    - 1: Index portion of error code refers to gate descriptor in IDT.
- Bit 2: GDT/LDT. Only use if the descriptor location is 0.
    - 0: This indicates the index portion of the error code refers to a descriptor in the current GDT.
    - 1: This indicates the index portion of the error code refers to a segment or gate descriptor in the LDT.
- Bits 3-15: Segment selector index. This is an index into the IDT, GDT, or current LDT to the segment or gate selector bring refrenced by the error code.
- Bits 16-31: Reserved

Error codes are not pushed on the stack for exceptions that are generated externally (via the INTR,LINT0,LINT1 pins), or INT n instruction.

The error code format is different for page fault exception errors. We will look at that in the next section.

Returning from a handler

All handlers must use either **IRET** or **IRETD** instructions to return. IRET is simular to RET except that it restores the saved EFLAGS (that was pushed on the stack when the handler was getting executed), and the IOPL field in EFLAGS is only set to 0 if the current protection level (CPL) is 0. The IF flag is also changed only if the CPL is less then or equal to the IOPL.

If a stack switch occured when executing the handler, IRET switches back to the interrupted procedures stack as well.

x86 Exceptions

Exceptions: Listing

All of the exceptions are defined as the first few interrupts within the IVT or IDT. Here is the complete list of generated exceptions from the x86 class of processors.

- **Fault** - the return address (Return CS:EIP that was pushed on stack when handler was called. See **Inside of our interrupt handler** for more information.) points to the instruction that caused the exception. The exception handler may fix the problem and then restart the program, making it look like nothing has happened.
- **Trap** - the return address points to the instruction after the one that has just completed.
- **Abort** - the return address is not always reliably supplied. A program which causes an abort is never meant to be continued.

x86 Processor Exceptions			
Interrupt Number	Class	Description	Error Code
0	Fault	Divide by 0	None
1	Trap or Fault	Single step (Debugger)	None. Can be retrived from debug registers
2	Unclassed	Non Maskable Interrupt (NMI) Pin	Not applicable
3	Trap	Breakpoint (Debugger)	None
4	Trap	Overflow	None
5	Fault	Bounds check	None
6	Fault	Unvalid OPCode	None
7	Fault	Device not available	None
8	Abort	Double Fault	Always 0
9	Abort (Reserved, do not use)	Coprocessor Segment Overrun	None
10	Fault	Invalid Task State Segment (TSS)	See error code below
11	Fault	Segment Not Present	See error code below
12	Fault	Stack Fault Exception	See error code below
13	Fault	General Protection Fault (GPF)	See error code below
14	Fault	Page Fault	See error code below
15	-	Unassigned	-
16	Fault	x87 FPU Error	None. x87 FPU provides own error information
17	Fault	Alignment Check (486+ Only)	Always 0
18	Abort	Machine Check (Pentium/586+ Only)	None. Error information abtained from MSRs
19	Fault	SIMD FPU Exception	None
20-31	-	Reserved	-
32-255	-	Avilable for software use	Not applicable

IRQ 0 and the System Timer

As you know, if we enter protected mode all interrupts must be disabled. If we have not done this, our system will triple fault immediately on the next clock tick. Why is this?

The **System Timer**, useually a form of the **8253 Programmable Interval Timer (PIT)** uses **IRQ 0** to let us know when a clock tick happens. This device is configured this way by the system BIOS.

But WAIT! *looks at above table*, Isn't that the **Divide by 0** error? ***Bingo.***

Because the tables are now invalid because we switched to protected mode, Who knows where this will lead us. Because of this, an immediate triple fault on the next system tick, and the reason we must disable interrupts before switching.

We should also note that the **8253 Programmable Interval Timer (PIT)** is a **hardware device**. Notice how, using the table above, it will fire an exception (IRQ 0)? How will we know its an actual error, or just a simple tick?

Lets take a look closer...

Remapping the 8259A Programmable Interrupt Controller (PIC)

The 8259A PIC is a standard controller used to control hardware interrupts. Hardware microcontrollers signal the PIC on their respective IR line that connects to the PIC. This allows the PIC to "know" a hardware device needs attention, and to signal the processor to fire an interrupt to handle the devices request.

In our above example, the 8253 PIT was signalling the 8259A PIC to handle a system tick in this manner, which caused IRQ 0 (Remember that the 8253 PIT uses IRQ 0) to fire--which caused a triple fault as it was also a) a divide by 0 exception, and b) invalid code as we have not written it yet.

To resolve this problem, we will need to reprogram the 8259A PIC Microcontroller to remap the hardware devices to use different IRQs.

Please keep in mind that we *can* still use software interrupts if the IF is 0 (interrupts disabled), as IF only applies to hardware interrupts. However, if we want to re-enable hardware interrupts, we must reprogram the PIC.

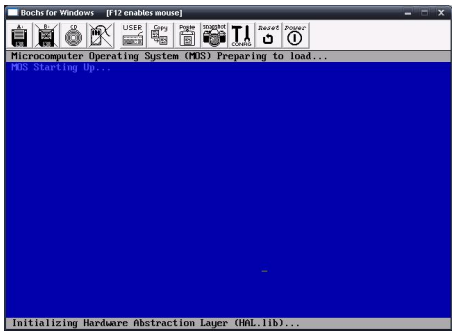
The 8259A PIC is a fairly complex microcontroller to program. Luckly, most of its modes do not apply to us.

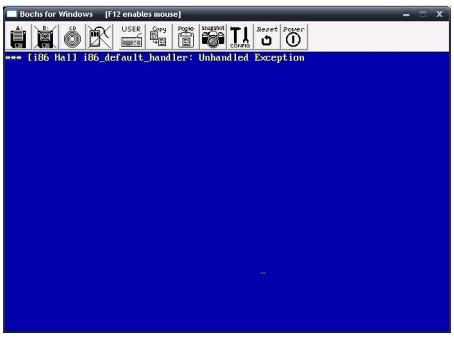
The demo at the end reprograms the PIC and re-enable interrupts. In order to completely get the most out of this tutorial, it is recommended to read our 8259A Programmable Interrupt Controller tutorial.

Demo

SKIP TO DEMO CONCLUSION

Note: These images have been scaled down to better fit the screen.





The first screenshot displays the kernel initializing the HAL. The second screenshot displays what happens when an interrupt is fired. Notice how our default handler catches the interrupt.

This demo is fairly complex, as we have covered a lot of material in this tutorial.

This demo installs a new Global Descriptor Table (GDT) for the kernel to use and an Interrupt Descriptor Table (IDT). It also creates a nice interface that we can use to handle software interrupts. Note that we do NOT cover hardware interrupts yet. In the next tutorial, we will be adding both **8259A PIC** and **8253 PIT** microcontroller interfaces to the HAL. This will allow us to catch hardware interrupts, enable hardware interrupts, and provide ourself with a system timer. It will be fun :)

Lets cover the demo a little more so that we can see everything working.

Hardware Abstraction

This demo includes alot of extra files that we have not seen until now. Because of this, it is kind of like a code dump, which is what I want to avoid here. Alot of it is very simple and things that we have looked at and even implimented in our bootloader. Some of it (Inside **idt.h** and **idt.cpp** may be new to you, and covers what we have learned here: The **Interrupt Descriptor Table (IDT)**.

This is also the beginning of our **Hardware Abstraction Layer (HAL)**!

As you know, I have been stressing hardware abstraction, and the importance of it, sense this series begun. You will see why soon as we continue to build on the Hal. You might even see the pluses of keeping the Hal completely independent of the kernel here!

Without further ado, lets look at the beginnings of the primary interface for the HAL.

Hal - include/hal.h - Platform independent interface for the HAL

This is the interface between the HAL and the kernel. It is part of the standard include directory, and is completely separate from its implimentation. All routines are declared **extern** as the header file is meant to be used by any implimentation that defines the routines inside of it. The implimentations are architecture specific, but the interface is in no way coupled to any specific implimentation, making it completely hardware independent.

While the implimentations themselves are architecture specific, we can simply build the implimentations for different architectures. Because each implimentation uses this common interface, and we can support dynamic loading (like hal.dll), we can either a)link what static hal implimentation to use when building for different architectures, or b)Build the different Hals independently, and choose which HAL to use at startup. Because they all use the same interface (Hal.h), we dont need any changes in the kernel to use different implimentations (and hence different hardware setups.) Cool?

There are currently only two functions in it. We will add more when we need to:

```
//! Initialize and shutdown hal
extern int Hal_Initialize ();
extern int Hal_Shutdown ();
```

I will most likley change the prototypes of these routines to allow startup and shutdown paramaters. In any case, these are very generic routines that is meant to provide a way to setup and shutdown the hardware, if needed, for the implimentation.

There are a couple of very simple layers of software within the hal for the gdt, idt, and cpu, and hal.cpp. Because all they do is initialize the layer below it (Hal.cpp calls the cpu initialize routine, which calls the gdt and idt initialize methods), I am not going to post it here as it may add more complexity in this tutorial then is required.

Instead, lets focus on the bulk of the hal: The gdt setup code, The idt setup code (This containes the bulk of what we looked at in this tutorial), and the kenrel's main() routine. Cool?

We will not cover the GDT in detail. Plaese see Tutorial 8 for a complete description of it.

Hal - hal/gdt.h - global descriptor table

Descriptor Tables ...again!

Yes, the GDT has come back to haunt you!!! ...yes, YOU!!

Anywhoo... the GDT is quite a complex structure, huh? As you know, a GDT is an array of descriptors. What was the format of a GDT descriptor again? Right, okay then...

- **Bits 56-63:** Bits 24-32 of the base address
- **Bit 55:** Granularity
  - **0:** None
  - **1:** Limit gets multiplied by 4K
- **Bit 54:** Segment type
  - **0:** 16 bit
  - **1:** 32 bit
- **Bit 53:** Reserved-Should be zero
- **Bits 52:** Reserved for OS use
- **Bits 48-51:** Bits 16-19 of the segment limit
- **Bit 47** Segment is in memory (Used with Virtual Memory)
- **Bits 45-46:** Descriptor Privilege Level
  - **0:** (Ring 0) Highest
  - **3:** (Ring 3) Lowest
- **Bit 44:** Descriptor Bit
  - **0:** System Descriptor
  - **1:** Code or Data Descriptor
- **Bits 41-43:** Descriptor Type
  - **Bit 43:** Executable segment
    - **0:** Data Segment
    - **1:** Code Segment
  - **Bit 42:** Expansion direction (Data segments), conforming (Code Segments)
  - **Bit 41:** Readable and Writable
    - **0:** Read only (Data Segments); Execute only (Code Segments)
    - **1:** Read and write (Data Segments); Read and Execute (Code Segments)
- **Bit 40:** Access bit (Used with Virtual Memory)
- **Bits 16-39:** Bits 0-23 of the Base Address
- **Bits 0-15:** Bits 0-15 of the Segment Limit

runs off screeming in horror

Okay, okay I'll stop now :) But serously, Intel could have made the structure more nicer, don't you think? :)

Building the C structure

We can hide this structure behind a nice C style structure using C's built in types. Knowing that the first 15 bits is the segment limit (Size of an uint16_t), **thats data member one**. The next 16 bits is bits 0-23 of the base address, and that can be either expressed as 1 uint16_t or 2 uint8_t's. **Thats data member two and/or three**. The next 16 bits (bits 41-56 of GDT) is 16 bits. This is the bulk of the ugly structure that containes flag values, and can be represented, of course, using either 2 uint8_t's or 1 uint16_t. **Thats the next data member.** The last byte is our base address. **Thats the last data member!**

Looking at the above, that ugly structure can be represented in 4 to 5 nice members within a structure. Here is our structure. Try to compare this structure with the above description and table to see where everything fits in. Also, remember that this structure is packed to 1 byte, so it is guaranteed to be 64 bits in size.

```
#ifdef _MSC_VER
#pragma pack (push, 1)
#endif

//! gdt descriptor. A gdt descriptor defines the properties of a specific
//! memory block and permissions.

struct gdt_descriptor {

        //! bits 0-15 of segment limit
        uint16_t                limit;

        //! bits 0-23 of base address
        uint16_t                baseLo;
        uint8_t                 baseMid;

        //! descriptor bit flags. Set using bit masks above
        uint16_t                flags;

        //! bits 24-32 of base address
        uint8_t                 baseHi;
};

#ifdef _MSC_VER
#pragma pack (pop, 1)
#endif
```

Easy enough! There are alot of bit flags that can be set to help build the flags bytes within the structure. Please see the header file to see them all, and notice how they work. Basically, we would bitwise OR the bit flags that we want to set. You will see us do this in the next section.

gdtr abstraction

Remember from tutorial 8, we have covered protected mode, the gdt, and gdtr? gdtr Is the processors internal register used to point to the GDT to be used. It is a 48 bit pointer that must follow the following format:

- **Bits 0-15:** size of entire gdt
- **Bits 16-48:** base address of gdt

Okay...This one is an easy one to convert to a C struct. Notice how it follows the above format:

```
#ifdef _MSC_VER
#pragma pack (push, 1)
#endif

//! processor gdtr register points to base of gdt. This helps
//! us set up the pointer
struct gdtr {

        //! size of gdt
        uint16_t                m_limit;

        //! base address of gdt
        uint32_t                m_base;
};

#ifdef _MSC_VER
#pragma pack (pop, 1)
#endif

// Global Descriptor Table (GDT)
static struct gdt_descriptor                    _gdt [MAX_DESCRIPTORS];

//! gdtr data
static struct gdtr                              _gdtr;
```

Here you can also see our new GDT and _gdtr, which will be used for refrence when setting up the processors GDTR register.

gdt_install(): Installs a gdt into gdtr

This routine is a very simple one. All it does is use the lgdt instruction to load GDTR with our gdtr pointer. We do not need to do any far jumps here, though, as CS should never change.

```
//! installs gdtr
static void gdt_install () {
#ifdef _MSC_VER
        _asm lgdt [_gdtr]
#endif
}
```

gdt_set_descriptor(): Sets up a new descriptor in the gdt

This routine is used to install a new descriptor in the GDT. For the most part, it is not too hard; the ugly code is when we get to setting up the flags.

```
//! Setup a descriptor in the Global Descriptor Table
void gdt_set_descriptor(uint32_t i, uint64_t base, uint64_t limit, uint8_t access, uint8_t grand)
{
        if (i > MAX_DESCRIPTORS)
                return;

        //! null out the descriptor
        memset ((void*)&_gdt[i], 0, sizeof (gdt_descriptor));

        //! set limit and base addresses
        _gdt[i].baseLo  = base & 0xffff;
        _gdt[i].baseMid = (base >> 16) & 0xff;
        _gdt[i].baseHi  = (base >> 24) & 0xff;
        _gdt[i].limit   = limit & 0xffff;

        //! set flags and grandularity bytes
        _gdt[i].flags = access;
        _gdt[i].grand = (limit >> 16) & 0x0f;
        _gdt[i].grand |= grand & 0xf0;

}
```

i86_gdt_initialize() - initializes the gdt

This brings everything together. All it does is set up our GDTR structure, installs some default descriptors into our GDT, and finally installs the GDT. To make things easier, this GDT is the same one we have used for our bootloader. The base address is 0, the limit (Maximum addressable address) is 4GB (0xffffffff). All of the flags are defined in **gdt.h**. They are defined to increase readability and to get rid of ugly magic numbers. It should be much easier to see what the descriptors are for with the flags!

```
//! initialize gdt
int i86_gdt_initialize () {

        //! set up gdtr
        _gdtr.m_limit = (sizeof (struct gdt_descriptor) * MAX_DESCRIPTORS)-1;
        _gdtr.m_base = (uint32_t)&_gdt[0];

        //! set null descriptor
        gdt_set_descriptor(0, 0, 0, 0, 0);

        //! set default code descriptor
        gdt_set_descriptor (1,0,0xffffffff,
                I86_GDT_DESC_READWRITE|I86_GDT_DESC_EXEC_CODE|I86_GDT_DESC_CODEDATA|I86_GDT_DESC_MEMORY,
                I86_GDT_GRAND_4K | I86_GDT_GRAND_32BIT | I86_GDT_GRAND_LIMITHI_MASK);

        //! set default data descriptor
        gdt_set_descriptor (2,0,0xffffffff,
                I86_GDT_DESC_READWRITE|I86_GDT_DESC_CODEDATA|I86_GDT_DESC_MEMORY,
                I86_GDT_GRAND_4K | I86_GDT_GRAND_32BIT | I86_GDT_GRAND_LIMITHI_MASK);

        //! install gdtr
        gdt_install ();

        return 0;
}
```

Hal: Interrupt Descriptor Table

THIS is where the fun stuff is at! The IDT interface is within the idt.h and idt.cpp source files.

hal.h - idt_descriptor

This is the structure for an interrupt descriptor. Compare this format with the descriptor format we looked at in this tutorial, and you should notice that they follow exactally the same format:

```
#ifdef _MSC_VER
#pragma pack (push, 1)
#endif

//! interrupt descriptor
struct idt_descriptor {

        //! bits 0-16 of interrupt routine (ir) address
        uint16_t                baseLo;

        //! code selector in gdt
        uint16_t                sel;

        //! reserved, shold be 0
        uint8_t                 reserved;

        //! bit flags. Set with flags above
        uint8_t                 flags;

        //! bits 16-32 of ir address
        uint16_t                baseHi;
};

#ifdef _MSC_VER
#pragma pack (pop, 1)
#endif
```

Lets look at what each member represents, and where at within the interrupt descriptor:

- **baseLo** - This is the first 16 bits of the base address of the interrupt routine (IR)
  - This is bits 0-15 within the overall interrupt descriptor. Compare this with the table listed in **Interrupt Descriptor: Structure**
- **sel** - Segment Selector
  - This is bits 16-31 within the overall interrupt descriptor.
- **reserved** - er... very usefull information here ;)
  - This is bits 31-38 within the overall interrupt descriptor.
- **flags** - Where the fun stuff is at!
  - Interrupt Descriptor Bits 39-41. This is where the bit flags are at
  - Interrupt Descriptor Bits 42-45. This is the Descriptor Priveldge Level (DPL)
- **baseHi** - Bits 16-31 of base address of the IR
  - This is bits 46-64 within the overall interrupt descriptor.

Simple enough! Notice how this struct directly matches how the interrupt descriptor is laid out. So, now that we have the description for an interrupt descriptor, lets take a look at installing a IDT!

idt.cpp - idtr

Simular to how we set up the gdtr structure, we also have one for idtr. Notice how this structure followes the exact structure for the idtr register.

```
#ifdef _MSC_VER
#pragma pack (push, 1)
#endif

//! describes the structure for the processors idtr register
struct idtr {

        //! size of the interrupt descriptor table (idt)
        uint16_t                limit;

        //! base address of idt
        uint32_t                base;
};

#ifdef _MSC_VER
#pragma pack (pop, 1)
#endif


//! interrupt descriptor table
static struct idt_descriptor    _idt [I86_MAX_INTERRUPTS];

//! idtr structure used to help define the cpu's idtr register
static struct idtr                              _idtr;
```

Okay... Remember that the IDT is nothing more then an array of interrupt descriptors? With this, _idtr is here for refrence; it stores the current information in the processors IDTR register for our use. Basically, all we need to do from here is to set up the IDT, and _idtr; then install the IDT! Not to hard :)

idt_install() - installs a new IDT

This is used to install the IDT into IDTR, no more, and no less. It is a helper method used to abstract the inline assembly language (Which is compiler dependent) behind a common interface to help with portability between compilers.

```
//! installs idtr into processors idtr register
static void idt_install () {
#ifdef _MSC_VER
        _asm lidt [_idtr]
#endif}
```

i86_default_handler() - default interrupt handler

Our IDT interface will provide a way to install our own interrupt handling routines directly into the IDT, which is as cool as it can get! Because there are 256 interrupts, there are 256 interrupt handlers. Odds are, we will not be using every one of them early on. So, what happens if an interrupt is generated that the kernel does not yet handle?

That is what this is for! This is a basic unhandled exception handler that our IDT interface will install (You will see this later on.) All it does is, if being built for debug mode, prints out an error. It then halts the system.

```
//! default handler to catch unhandled system interrupts.
void i86_default_handler () {

#ifdef _DEBUG
        DebugClrScr (0x18);
        DebugGotoXY (0,0);
        DebugSetColor (0x1e);
        DebugPrintf ("*** [i86 Hal] i86_default_handler: Unhandled Exception");
#endif

        for(;;);
}
```

Returning from an interrupt...

C and C++ automatically pops the values off the stack and issues a RET instruction when returning from an IR. This is bad! Because of this, we need to issue our own way of returning through an IRET instruction.

geninterrupt() - generrate interrupt call

This is a little tricky. This is another helper method provided to abstract the inline assembly language behind a common interface for better portability for more compiliers. However, it also hides the challenge of generating an abritary interrupt call.

The problem is that of the OPCode for an interrupt (INT instruction) only has one format: 0xCDimm, where imm is an intermediate value. Because of this, we cannot use any registers nor memory locations in the INT instruction; as there is no OPCode form that accepts that (Invalid instruction.) So, how do we fix this? There are alot of different ways, of course. I opted to use a fast and small solution: self modifying code.

Basically, all we need to do is modify the second byte of the INT OPCode. Knowing it is always two bytes (First being 0xCD, second is the interrupt number to call) it is quite an easy solution:

```
//! generate interrupt call
void geninterrupt (int n) {
#ifdef _MSC_VER
        _asm {
                mov al, byte ptr [n]
                mov byte ptr [genint+1], al
                jmp genint
        genint:
                int 0   // above code modifies the 0 to int number to generate
        }
#endif
}
```

i86_install_ir () - installs interrupt handler into IDT

This is a little tricky, but not too hard. Remember that the **baseLo** and **baseHi** members of our structure contain the high and low bits of our **Interrupt Routine (IR)**? So, all we need to do is get the address of the IR function, and store its high and low bits. This is done here by means of a **function pointer**.

We pass in a function pointer as a parameter. This routine gets the address of the function the pointer points to, and masks out the low and high bits, storing it into the structure at **_idt [i]**, where i is the descriptor offset (the Interrupt number) in the IDT.

```
//! installs a new interrupt handler
int i86_install_ir (uint32_t i, uint16_t flags, uint16_t sel, I86_IRQ_HANDLER irq) {

        if (i>I86_MAX_INTERRUPTS)
                return 0;

        if (!irq)
                return 0;

        //! get base address of interrupt handler
        uint64_t                uiBase = (uint64_t)&(*irq);

        //! store base address into idt
        _idt[i].baseLo          =       uiBase & 0xffff;
        _idt[i].baseHi          =       (uiBase >> 16) & 0xffff;
        _idt[i].reserved        =       0;
        _idt[i].flags           =       flags;
        _idt[i].sel             =       sel;

        return  0;
}
```

There is some bueaty in this. Remember that, when an interrupt is generated, the processor pushes some information on the stack for us? This information will now be in the paramaters list when our routine is called! Cool, huh? This also means, however, that we need to be careful as only some interrupts push error codes, others do not.

i86_idt_initialize () - Initialize IDT Interface

Now, lets bring everything together. The following code sets up IDTR, sets our default interrupt handler to catch all interrupts (This is so we only need to define the needed interrupts in the Kernel); and finally installs the IDT using the above methods.

The bit flags used for setting up the IDT are defined in **idt.h** and are provided to make the code more readable and easier to modify.

```
//! initialize idt
int i86_idt_initialize (uint16_t codeSel) {

        //! set up idtr for processor
        _idtr.limit = sizeof (struct idt_descriptor) * I86_MAX_INTERRUPTS -1;
        _idtr.base      = (uint32_t)&_idt[0];

        //! null out the idt
        memset ((void*)&_idt[0], 0, sizeof (idt_descriptor) * I86_MAX_INTERRUPTS-1);

        //! register default handlers
        for (int i=0; i<I86_MAX_INTERRUPTS; i++)
                i86_install_ir (i, I86_IDT_DESC_PRESENT | I86_IDT_DESC_BIT32,
                        codeSel, (I86_IRQ_HANDLER)i86_default_handler);

        //! install our idt
        idt_install ();
```

```
        return 0;
}
```

Demo Conclusion

This demo is a bit complex, I am to admit. At least we got the ugly neccessities out of the way! You will see that, if you issue any INT instruction, the default handler will be called. If you install your own interrupt handlers, try to experiement with them - both the ones with the error codes, and the ones without them. You will see the interrupts being fired. Anytime you call geninterrupt() or an INT instruction, you will see that the correct interrupt handler (Or, if the interrupt handler was not defined, the default handler) is executed.

To keep this tutorial from getting much more complex, I decided to NOT handle hardware interrupts yet. We will cover this in the next tutorial, as well as developing the code for the **8253 Programmable Interval Timer (PIT)** for use as the Kernels System Timer as well as for the **8259A Programmable Interrupt Controller**, which is needed for hardware interrupts.

Study the demo well and how everything works. Modify a few things; try to register your own interrupt handlers using **i86_install_ir()** and generrating the interrupts. To do this, all we need to do is:

```
//! our uber 1337 interrupt handler. handles int 5 request
void int_handler_5 () {

        _asm add esp, 12
        _asm pushad

        // do whatever...

        _asm popad
        _asm iretd
}

//! registers our interrupt handler
i86_install_ir (5, I86_IDT_DESC_PRESENT | I86_IDT_DESC_BIT32,
        0x8, (I86_IRQ_HANDLER)int_handler_5);

//! generates int 5 instruction. You can also use inline assembly, of course
geninterrupt (5);
```

I decided to leave out the paramater lists for the interrupt handlers as the format may change. So, in order to access paramaters, we would need to access it through ESP. I might decide to give it paramaters later on to make things easier, though.

Demo Download Here (MSVC++)

Conclusion

A lot of fun stuff this time, huh? We covered a lot of ground in this tutorial. We have looked at a lot of important topics, covered exception and interrupt handling, and have re-enabled interrupts in our system. This might even be the last time we ever see a triple fault. Woohoo!

I am to admit this tutorial is a bit complex. *Isn't OS programming fun?* ^_^

In the next tutorial, we will be starting to develop our kernel even more. We will be handling timing through the **8254 Programmable Interval Timer (PIT) microcontroller**, which will be covered simular to the 8259A PIC tutorial. Afterwords, we plan on moving onto more memory management and process management. ...We might even develop a basic debugging text based console to spice things up a bit ;)

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!

Operating Systems Development - PIC, PIT, and exceptions

by Mike, 2008

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

Please note: This tutorial covers hardware interrupt handling, not software interrupt handling. If you are looking for software interrupts, please see Tutorial 15. This tutorial requires knowledge of software interrupt handling.

Introduction

Welcome to...what? Tutorial 16 already?

In the last tutorial, we have dived deep into the world of interrupt handling. We have covered, and even implimented, interfaces for the GDT and IDT inside of our **hardware abstraction layer**. We have covered almost everything we needed for software interrupt handling to work. ...But WAIT! What about hardware interrupts? o_0

Because alot of critical system devices use interrupts, it is necessary for us to be able to handle and catch interrupts triggered by hardware devices. The good news? This is already done for us! By what? The **8259 Programmable Interrupt Controller (PIC)**. We will look closer in the next section.

Even if we get hardware interrupts working by itself, we will still be running into problems do to the system timer. As long as the system timer does not use a valid interrupt handler that we have set up for it, it will triple fault a few milliseconds after you enable hardware interrupts. After all, it will call an invalid interrupt handler, remember? Thus, we will also fix this small problem by reprogramming the system timer, otherwise known as the **Programmable Interval Timer (PIT)**.

Alot of stuff in this tutorial. We will look at:

- Hardware Interrupts
- Interrupt Chaining
- Hal: Programmable Interrupt Controller
- Hal: Programmable Interval Timer
- Hardware Abstraction
- Interrupts Implimentation and Design for our HAL

Please note that this does not cover interrupt handling. Please see Tutorial 15 for interrupt handlng.

With all of that in mind, lets take a look...

Hardware Interrupts

There are two types of interrupts, those generated by software (Useually by an instruction, such as INT, INT 3, BOUND, INTO), and an interrupt generated by hardware.

Hardware interrupts are very important for PC's. It allows other hardware devices to signal the CPU that something is about to happen. For example, a keystroke on the keyboard, or a single clock tick on the internal timer, for example.

We will need to map what Interrupt Request (IRQ) to generate when these interrupts happen. This way, we have a way to track these hardware changes.

Lets take a look at these hardware interrupts.

x86 Hardware Interrupts		
8259A Input pin	Interrupt Number	Description
IRQ0	0x08	Timer
IRQ1	0x09	Keyboard
IRQ2	0x0A	Cascade for 8259A Slave controller
IRQ3	0x0B	Serial port 2
IRQ4	0x0C	Serial port 1
IRQ5	0x0D	AT systems: Parallel Port 2. PS/2 systems: reserved
IRQ6	0x0E	Diskette drive
IRQ7	0x0F	Parallel Port 1
IRQ8/IRQ0	0x70	CMOS Real time clock
IRQ9/IRQ1	0x71	CGA vertical retrace
IRQ10/IRQ2	0x72	Reserved
IRQ11/IRQ3	0x73	Reserved
IRQ12/IRQ4	0x74	AT systems: reserved. PS/2: auxiliary device
IRQ13/IRQ5	0x75	FPU
IRQ14/IRQ6	0x76	Hard disk controller
IRQ15/IRQ7	0x77	Reserved

You do not need to worry to much about each device just yet. The 8259A Pins are described in detail in the 8259 PIC tutorial. The Interrupt Numbers listed in this table are the default DOS **interrupt requests (IRQ)** to execute when these events trigger.

In most cases, we will need to recreate a new interrupt table. As such, most operating systems need to remap the interrupts the PIC's use to insure they call the proper IRQ within their IVT. This is done for us by the BIOS for the real mode IVT. We will cover how to do this later in this tutorial as well.

Wait...What is this PIC thing? All of these hardware devices that can signal hardware devices are connected indirectly to the **8259A Programmable Interrupt Controller (PIC)**. This is a special, and very important microcontroller that is used to signal the microprocessor when it needs to fire a hardware interrupt.

We will be programming this microcontroller a little later in this tutorial. Because this microcontroller is fairly complex, we have dedicated another tutorial for it. Please read it here.

Interrupt Chaining

We will be able to install our own interrupt handlers within the **Interrupt Descriptor Table (IDT)** very easily. We create interrupt handlers to handle not only software interrupts, but interrupts triggered by hardware devices. **Remember: The hardware devices signal the Programmable Interrupt Controller to signal the processor to request a hardware interrupt to be triggered.** The PIC lets the processor know what **Interrupt Request (IRQ)** to call within out **Interrupt Descriptor Table (IDT)**.

But wait... How does the PIC know what IRQs to call within our IDT? *We tell it*.

This is why we must reprogram the PIC in order to let it know what interrupts to use.

Okay, lets say we now have interrupt handlers to handle software and hardware interrupts. What now? How does this work from our perspecitive? Sure, we can easily install handlers for different devices, but what if multiple devices require the same interrupt? What about multiple functions for a software interrupt? This is where **Interrupt Chaining** comes in.

Interrupt chaning is a technique used to restore and call all of the interrupt handlers that share that same interrupt number. This is done by saving the previus **Interrupt Routine (IR)** in a function pointer. Then, installing the new handler, and calling the previus interrupt handler whenever the new IR is called.

Here is an example:

```
void deviceInitialize () {

        //store previus interrupt handler
        prevhandler = getvect (0);

        //install new interrupt handler
        setvect (0, handler);
}

void deviceShutdown () {

        //install previus interrupt handler
        setvect (0, prevhandler);
}

void handler () {

        // do stuff...

        // call previus interrupt handler
        (*prevhandler) ();
}
```

As you can see, interrupt chaining is rather easy. Notice how the previus interrupt handlers will always be called whenever this interrupt fires? **setvect()** installs a new interrupt vector. **getvect()** returns an interrupt vector. These interrupt vectors can be stored in either the **Interrupt Vector Table (IVT)** or **Interrupt Descriptor Table (IDT)**. Wait, what? Thats right--ours :)

Get Ready - Implimenting Interrupt Handling

We have covered alot of ground with interrupts and interrupt handling. Text alone can only go so far. We have even looked a little bit about how hardware interrupt handling works, but not enough to get very far.

We cannot impliment hardware interrupt handling until we learn how to program the **Programmable Interrupt Controller**. Also, we cannot enable hardware interrupts until we fix the timing problem (Remember that the **Programmable Interval Timer** is still connected to IRQ8 thanks to the BIOS? That means, as soon as we re-enable hadware interrupts, the next timer tick will result in a double fault.) Because of this, we also have to learn how to reprogram the **Programmable Interval Timer**.

This, dear readers, is where things get complex. **Welcome back to the world of hardware programming** :)

There is good news, however... None of these microcontrollers are very complex. However, to keep the main series from getting too complex, I have decided to write **2 tutorials dedicated to these microcontrollers.** This is required reading for understanding of the demo and code that lies ahead.

Because of this, I recommend for all of our readers to read the following tutorials before continuing:

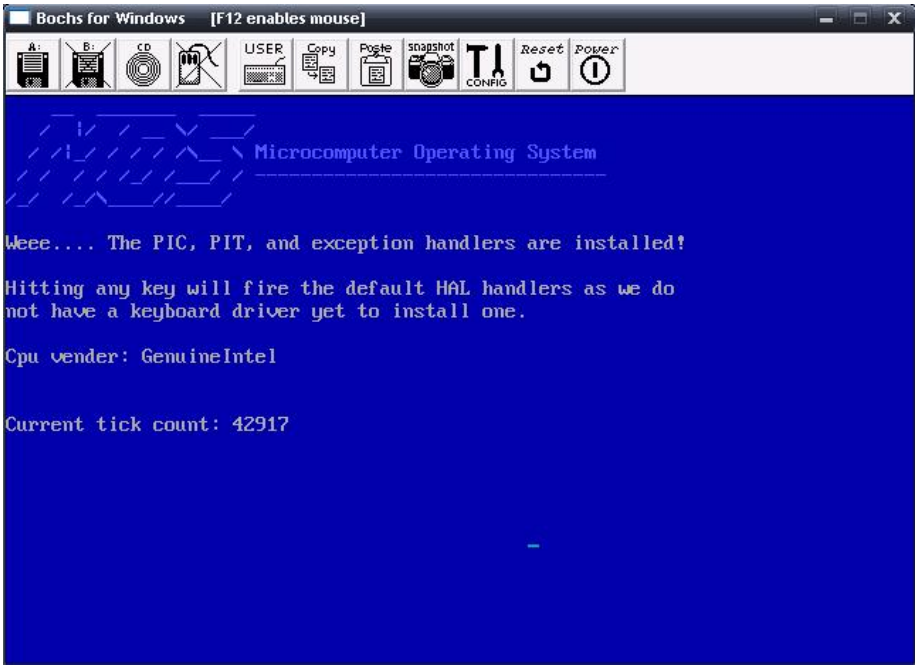
- 8259 Programmable Interrupt Controller
- 8253 Programmable Interval Timer

Do not worry if you do not understand everything in these tutorials yet. You will see us implimenting everything within the above tutorials in the sections to come. :) I will also still be describing everything here, so I will not lead you in the dark.

It may also be helpful to use the above tutorials as a refrence, as we will be refrencing the above tutorials alot throughout the upcoming sections.

So..? What are you waiting for? Jump into those tutorials! And come back here when you are done. Don't worry, I am just text... I will still be here when you get back.. ...unless I am deleted of course :)

Demo



The new demo in action

Demo Download (MSVC++)

I am going to admit, this demo is a little complex. It is not too hard, though, as most of the code should look familiar.

This demo uses our **hardware abstraction layer (HAL)** to install its own exception handlers. This allows us to catch processor exception errors (Like divide by 0.) It also initializes the HAL, which in turns initializes processor tables, the PIC, and the PIT. This allows us to enable hardware exceptions (Finally!) and start a timer. **The demo displays the current tick count (Updated by the PIT's interrupt handler) on screen.** Any other interrupt generated will be handled by the IDT's default interrupt handler. We have set this up in our IDT tutorial remember?

Because the keyboard generates IRQ 9, and we have not defined an interrupt for it yet, anytime a key is pressed the default handler for the IDT will execute.

Okay... I have tried to make this demo look a little cooler then the other ones. At the same time, I did not want to add even more complexity to it. Because we have looked at creating interfaces for the PIC and PIT microcontrollers, lets look a little bit at the demo code itself, shall we..?

Hardware Abstraction

The first interface that we will look at is the one provided by the hardware abstraction layer. This can be seen by looking at **include/hal.h** and **hal/hal.cpp**. I will not be describing the routines in depth as most of it is very simple and simply use the other interfaces (GDT, IDT, CPU, PIC, PIT, etc) that we have developed (And are about to develop). Instead, I want to look at the interface itself. This will be the interface used by the kernel and device drivers, so why not?

The new hal.h

This is where we start seeing how useful hardware abstraction can be. I wanted to provide a "DOS"-like interface that is just as easy to use as programming 16bit DOS is. In doing so, I came up with a very easy list of routines that can be used for alot of different purposes. Looking at these routines, you will see there is absolutley no refrence to the hardware devices or tables that are used by them. This is what hardware abstraction is all about. It does not abstract the architecture; but rather the hardware that it uses.

Alot of the code that we use later use the routines within the HAL to perform its task. Because of this, I wanted you to take a look at the hardware abstraction layer now, and the routines it provides.

```
extern  int                     _cdecl  hal_initialize ();
extern  int                     _cdecl  hal_shutdown ();
extern  void                    _cdecl  enable ();
extern  void                    _cdecl  disable ();
extern  void                    _cdecl  geninterrupt (int n);
extern  unsigned char           _cdecl  inportb (unsigned short id);
extern  void                    _cdecl  outportb (unsigned short id, unsigned char value);
extern  void                    _cdecl  setvect (int intno, void (_cdecl far &vect) ( ) );
extern  void (_cdecl    far *   _cdecl  getvect (int intno)) ( );
extern  bool                    _cdecl  interruptmask (uint8_t intno, bool enable);
extern  inline void             _cdecl  interruptdone (unsigned int intno);
extern  void                    _cdecl  sound (unsigned frequency);
extern  const char*             _cdecl  get_cpu_vender ();
extern  int                     _cdecl  get_tick_count ();
```

If you have ever programmed 16bit DOS, you should feel at home right about now! :)

Programmable Interrupt Controller

8259: Microcontroller

The 8259 Microcontroller familiy is a set of **Programmable Interrupt Controller (PIC) Integrated Circuits (ICs)**. Hardware controllers are indirectly connected to the PIC when a hardware interrupt is requested. Because of this, in order to handle hardware interrupts, we must have an understanding of how to program this microcontroller.

I will still go over everything here, however the 8259 is a complex microcontroller. Because of this, we have dedicated a full tutorial to cover just this controller. Because of this, **in order to get the most out of this section, Please see (and refrence) the following tutorial to learn about the**

PIC:

8259A Programmable Interrupt Controller Tutorial

Please note: **We will not cover everything about the PIC nor hardware interrupt handling here. Please see the above tutorial for this.**

8259: Abstract

The **Programmable Interrupt Controller (PIC)** is a microcontroller used to provide the connection between devices and the processor through **interrupt lines**. This allows devices to signal the processor whenever it requires attention from the system software or executive. This is the **Interrupt Request (IRQ)**.

The PIC controls all of the hardware interrupt requests. It allows us to recieve signals from different hardware devices whenever they require attention. When a device, such as the **Floppy Disk Controller (FDC)** requires attention, it tells the PIC to fire the IRQ it is assigned to. From here, the PIC will signal the processor, and give the interrupt number to call. The processor then offsets into the IDT, and executes the interrupt handler at ring 0. We define all of the interrupt handlers, so we now take control.

The best thing about this is that it is all **automatic** thanks to the PIC. Whenever a device signals the PIC, our interrupt handler will be executed automatically. The processor also performs a **task switch** to ring 0, so we will always end up in kernal land to handle the request. Cool, huh?

The PIC itself is a complex microcontroller. I will try to cover everything in detail here, but please keep in mind that--in order to get the most out of this tutorial, we encourage our readers to read the above PIC tutorial.

With all of that in mind--lets dive into the interface. All of this code can be found in the demo at the end of this tutorial.

Operation Commands

An **Operation Command** is a special command that is composed of a bit pattern. This bit pattern must be set up to describe the command for a microcontroller. There are basically two types of operation commands: **Initialization Command Words (ICWs)** and **Operation Command Words (OCWs)**.

ICWs are operation commands that must only be used during the initialization of the device. OCWs are used to control the device after the device has been initialized.

pic.h: Interface

This file provides the overall minidriver interface for the rest of the system. This is the interface to controlling and managing the PIC. I define a "minidriver" as a driver embedded in a peice of software, and not as stand alone software.

pic.h: Device Connections

In the PIC tutorial, we have looked at hardware interrupts in alot of depth. We have looked at how hardware devices signal the PIC whenever it requires attention of the system software or executive. For this to work, each device is indirectly connected to an **Interrupt Request (IR)** line on the PIC. This line not only represents the **Interrupt Request (IRQ)** the device uses but also its pririty level (The lower the IRQ number, the higher pririty.)

To help when working with individual devices and their IRQs, we will want to abstract the IRQ they use. This helps increase portability but also readability as they are behind nice constants. Remember: Magic numbers are bad!

```
//! The following devices use PIC 1 to generate interrupts
#define         I86_PIC_IRQ_TIMER               0
#define         I86_PIC_IRQ_KEYBOARD            1
#define         I86_PIC_IRQ_SERIAL2             3
#define         I86_PIC_IRQ_SERIAL1             4
#define         I86_PIC_IRQ_PARALLEL2           5
#define         I86_PIC_IRQ_DISKETTE            6
#define         I86_PIC_IRQ_PARALLEL1           7

//! The following devices use PIC 2 to generate interrupts
#define         I86_PIC_IRQ_CMOSTIMER           0
#define         I86_PIC_IRQ_CGARETRACE          1
#define         I86_PIC_IRQ_AUXILIARY           4
#define         I86_PIC_IRQ_FPU                 5
#define         I86_PIC_IRQ_HDC                 6
```

The above constants list all of the devices (along with their IRQ line/number) that they use. **There are only 8 IR lines per PIC**, hence only 8 possible IRQs per PIC. **Remember that PIC's can be cascaded with secondary PICs** (Up to 8 PICs can be cascaded with each other.) Typical x86 architectures only have 2--One primary and one secondary.

The two most important devices for us right now are the timer (I86_PIC_IRQ_TIMER) and keyboard (I86_PIC_IRQ_KEYBOARD). We will be using I86_PIC_IRQ_TIMER in this tutorial, so you will see how everything works together, cool?

pic: 8259 Commands

Setting up the PIC is farily complex. It is done through a series of **Command Words**, which are a bit pattern that containes various of states used for initialization and operation. This might seem a little complex, but it is not to hard. We will first look at the **Operation Command Word (OCW)** that are used to control the PIC. We will look at the initialization commands a little later.

pic: Operation Command Word 1

This represents the value in the **Interrupt Mask Register (IMR)**. It does not have a special format, so it is handled directly in the implimentation file to enable and disable hardware interrupts. It is a single byte in size. We enable and disable ("mask and unmask") an interrupt request line by setting the correct bit. Remember that there are only 8 IRQ's per PIC? So, bit 0 in the IMR is IRQ 0, bit 1 is IRQ 1, bit 2 is IRQ 2, and so on.

We will take a look at the **Interrupt Mask register** a little later on, cool?

pic: Operation Command Word 2

This is the primary control word used to control the PIC. Lets take a look...

Operation Command Word (OCW) 2		
Bit Number	Value	Description
0-2	L0/L1/L2	Interrupt level upon which the controller must react

3-4	0	Reserved, must be 0
5	EOI	End of Interrupt (EOI) request
6	SL	Selection
7	R	Rotation option

Okay then!

The format of OCW 2 is very easy. The first three bits are the current interrupt level. Bits 3-4 are reserved (Must be 0). Bit 5 represents **End of Interrupt (EOI)**. Bit 6 is the **Selection** bit. Bit 7 provides a **Rotation** command.

Because each command is selected in individual bits, we can **bitwise OR** these commands together to produce OCW 2.

```
//! Command Word 2 bit masks. Use when sending commands
#define         I86_PIC_OCW2_MASK_L1            1               //00000001      //Level 1 interrupt level
#define         I86_PIC_OCW2_MASK_L2            2               //00000010      //Level 2 interrupt level
#define         I86_PIC_OCW2_MASK_L3            4               //00000100      //Level 3 interrupt level
#define         I86_PIC_OCW2_MASK_EOI           0x20            //00100000      //End of Interrupt command
#define         I86_PIC_OCW2_MASK_SL            0x40            //01000000      //Select command
#define         I86_PIC_OCW2_MASK_ROTATE        0x80            //10000000      //Rotation command
```

There you have it! This is an important command word for us. We will be required to send this command word from all interrupt handlers.

Remember that the PIC masks off the interrupt when it gets executed? This means that no more interrupt requests on that IR line can execute until the processor acknowledges the PIC. This is done by sending an **End of Interrupt** command word to the correct PIC. We can do this by **masking off** the EOI bit in the command word. This is what **I86_PIC_OCW2_MASK_EOI** is used for.

A little later on, you will see that the interface has a **i86_pic_send_command** routine that is used to...erm...send commands to the PIC. Lets look at an example of sending the EOI command using this routine so that you can see how it works:

```
i86_pic_send_command (I86_PIC_OCW2_MASK_EOI, picNumber);
```

The above code will send an EOI command to the pic in **picNumber**, cool?

I suppose thats it for OCW 2. On to the next one!

pic: Operation Command Word 3

I plan on adding to this section.

```
//! Command Word 3 bit masks. Use when sending commands
#define         I86_PIC_OCW3_MASK_RIS           1               //00000001
#define         I86_PIC_OCW3_MASK_RIR           2               //00000010
#define         I86_PIC_OCW3_MASK_MODE          4               //00000100
#define         I86_PIC_OCW3_MASK_SMM           0x20            //00100000
#define         I86_PIC_OCW3_MASK_ESMM          0x40            //01000000
#define         I86_PIC_OCW3_MASK_D7            0x80            //10000000
```

pic.cpp: Implimentation

Okay...Everything was easy so far, right? You are probably asking "Where is the challenge!??" Well, okay then.

pic.cpp provides the implimentation for our PIC interface. First thing we must look at are the registers.

pic.cpp: Register constants

This is where we define the constants to abstract the port locations for the PICs. Notice that I have defined constants for all register names, even though they share the same port address. The reason is for completness: Even though they share the same port location, they still are different registers.

```
//! PIC 1 register port addresses
#define I86_PIC1_REG_COMMAND                    0x20                    // command register
#define I86_PIC1_REG_STATUS                     0x20                    // status register
#define I86_PIC1_REG_DATA                       0x21                    // data register
#define I86_PIC1_REG_IMR                        0x21                    // interrupt mask register (imr)

//! PIC 2 register port addresses
#define I86_PIC2_REG_COMMAND                    0xA0                    // ^ see above register names
#define I86_PIC2_REG_STATUS                     0xA0
#define I86_PIC2_REG_DATA                       0xA1
#define I86_PIC2_REG_IMR                        0xA1
```

Not to hard. We send commands to the **command register**, and read data from the **data register**. If we are writing from a data register, we are accessing the **Interrupt Mask Register (IMR)** which can be used to manually mask off or unmask interrupt requests. This is how we enable or disable interrupt requests.

The register we are accessing depends on wether it is a write or read operation. If we **write** to port 0x20, we are accessing the command register. If we are **reading** from it, we are accessing the status register.

Lastly, because this is an implimentation detail, it is part of the implimentation (pic.cpp), not interface.

Lets take a look at the constants used during initialization next.

pic.cpp: Initialization Control Word 1

This is the primary control word used when initializing the PICs. This is a 7 bit value that must be put in the primary PIC command register. This is the format:

Initialization Control Word (ICW) 1		
Bit Number	Value	Description
0	IC4	If set(1), the PIC expects to recieve IC4 during initialization.
1	SNGL	If set(1), only one PIC in system. If cleared, PIC is cascaded with slave PICs, and ICW3 must be sent to controller.
2	ADI	If set (1), CALL address interval is 4, else 8. This is useually ignored by x86, and is default to 0
3	LTIM	If set (1), Operate in Level Triggered Mode. If Not set (0), Operate in Edge Triggered Mode
4	1	Initialization bit. Set 1 if PIC is to be initialized
5	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0
6	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0
7	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0

As you can see, there is alot going on here. We have seen some of these before. This is not as hard as it seems, as most of these bits are not used on the x86 platform.

There are two types of constants for each command word. The first type are **bit masks** that are used to mask off the bits that the data represents. The second type of constants used are **command control bits** which are used in conjunction with the masks to set them to their correct values.

Lets look closer. Here are the ICW 1 bit masks. Notice how they follow the format shown in the above table. We do not define anything for the last three bits as they are always zero for x86 architectures.

```
//! Initialization Control Word 1 bit masks
#define I86_PIC_ICW1_MASK_IC4                   0x1     //00000001      // Expect ICW 4 bit
#define I86_PIC_ICW1_MASK_SNGL                  0x2     //00000010      // Single or Cascaded
#define I86_PIC_ICW1_MASK_ADI                   0x4     //00000100      // Call Address Interval
#define I86_PIC_ICW1_MASK_LTIM                  0x8     //00001000      // Operation Mode
#define I86_PIC_ICW1_MASK_INIT                  0x10    //00010000      // Initialization Command
```

Okay...We can easily use the above bit masks to set the bits in the ICW 1. But, how do we know what they mean? That is, when we mask off the bits that we are wanting to set, how do we know what the value we are setting them to mean? This is where **command control bits** come in.

They contain the constant values that may be used to set the above masked off bits to. This helps increase readability and extendability alot.

Here are the command control bits for ICW 1. Lets take a look...

```
#define I86_PIC_ICW1_IC4_EXPECT                 1       //1             //Use when setting I86_PIC_ICW1_MASK_IC4
#define I86_PIC_ICW1_IC4_NO                     0       //0
#define I86_PIC_ICW1_SNGL_YES                   2       //10            //Use when setting I86_PIC_ICW1_MASK_SNGL
#define I86_PIC_ICW1_SNGL_NO                    0       //00
#define I86_PIC_ICW1_ADI_CALLINTERVAL4          4       //100           //Use when setting I86_PIC_ICW1_MASK_ADI
#define I86_PIC_ICW1_ADI_CALLINTERVAL8          0       //000
#define I86_PIC_ICW1_LTIM_LEVELTRIGGERED        8       //1000          //Use when setting I86_PIC_ICW1_MASK_LTIM
#define I86_PIC_ICW1_LTIM_EDGETRIGGERED         0       //0000
#define I86_PIC_ICW1_INIT_YES                   0x10    //10000         //Use when setting I86_PIC_ICW1_MASK_INIT
#define I86_PIC_ICW1_INIT_NO                    0       //00000
```

Not to hard. The naming convention used allows us to easily know what to use, and where. For example, **I86_PIC_ICW1_SNGL_YES** is used with **I86_PIC_ICW1_MASK_SNGL**, **I86_PIC_ICW1_LTIM_EDGETRIGGERED** is used with **I86_PIC_ICW1_MASK_LTIM**.

Here is an example of how they work together. When we are initializing the PIC, we will need to enable initialization, and to send ICW 4. To do this, we simply set up ICW 1 like this:

```
        uint8_t icw=0;
        icw = (icw & ~I86_PIC_ICW1_MASK_INIT) | I86_PIC_ICW1_INIT_YES;
        icw = (icw & ~I86_PIC_ICW1_MASK_IC4) | I86_PIC_ICW1_IC4_EXPECT;
```

Thats it!? Yep. Notice how everything works and fits together. This is used throughout the implimentations to set specific bits (or a series of bits) to known values. The best thing here as that--just by looking at the above code--you know what it is doing. (Begin initialization, and to expect ICW 4). Pretty cool, huh? We will be using this method throughout this series when needed when setting and masking off bits.

Initialization Control Word 2

This control word is used to map the base address of the IVT of which the PIC are to use.

Initialization Control Word (ICW) 2		
Bit Number	Value	Description
0-2	A8/A9/A10	Address bits A8-A10 for IVT when in MCS-80/85 mode.
3-7	A11(T3)/A12(T4)/A13(T5)/A14(T6)/A15(T7)	Address bits A11-A15 for IVT when in MCS-80/85 mode. **In 80x86 mode, specifies the interrupt vector address.** May be set to 0 in x86 mode.

During initialization, we need to send ICW 2 to the PICs to tell them where the base address of the IRQ's to use. If an ICW1 was sent to the PICs (With the initialization bit set), you must send ICW2 next. **Not doing so can result in undefined results.** Most likley the incorrect interrupt handler will be executed.

Because this command does not have a complex format, it is handled directly inside of **pic.cpp** and does not have any constants.

Initialization Control Word 3

This command word is used to let the PIC controllers know how they are cascaded. To cascade multiple PICs, we must connect one of the PIC's IR lines to each other. We use this command word to let them know what line it is.

Initialization Control Word (ICW) 3		
Bit Number	Value	Description
0-7	S0-S7	Specifies what Interrupt Request (IRQ) is connected to slave PIC

Because this command does not have a complex format, it is handled directly inside of **pic.cpp** and does not have any constants.

Initialization Control Word 4

Yey! This is the final initialization control word. This controls how everything is to operate.

Initialization Control Word (ICW) 4		
Bit Number	Value	Description
0	uPM	If set (1), it is in 80x86 mode. Cleared if MCS-80/86 mode
1	AEOI	If set, on the last interrupt acknowledge pulse, controller automatically performs End of Interrupt (EOI) operation
2	M/S	Only use if BUF is set. If set (1), selects buffer master. Cleared if buffer slave.
3	BUF	If set, controller operates in buffered mode
4	SFNM	Special Fully Nested Mode. Used in systems with a large amount of cascaded controllers.
5-7	0	Reserved, must be 0

This is a pretty complex command word, but not to bad. Lets take a look at our defined bit masks. Notice how they follow the format shown above.

```
//! Initialization Control Word 4 bit masks
#define I86_PIC_ICW4_MASK_UPM           0x1     //00000001      // Mode
#define I86_PIC_ICW4_MASK_AEOI          0x2     //00000010      // Automatic EOI
#define I86_PIC_ICW4_MASK_MS            0x4     //00000100      // Selects buffer type
#define I86_PIC_ICW4_MASK_BUF           0x8     //00001000      // Buffered mode
#define I86_PIC_ICW4_MASK_SFNM          0x10    //00010000      // Special fully-nested mode
```

Simular to **ICW 1**, we have a set of control bits that are used in conjunction with the bit masks to set properties. Here they are...

```
#define I86_PIC_ICW4_UPM_86MODE         1       //1             //Use when setting I86_PIC_ICW4_MASK_UPM
#define I86_PIC_ICW4_UPM_MCSMODE        0       //0
#define I86_PIC_ICW4_AEOI_AUTOEOI       2       //10            //Use when setting I86_PIC_ICW4_MASK_AEOI
#define I86_PIC_ICW4_AEOI_NOAUTOEOI     0       //00
#define I86_PIC_ICW4_MS_BUFFERMASTER    4       //100           //Use when setting I86_PIC_ICW4_MASK_MS
#define I86_PIC_ICW4_MS_BUFFERSLAVE     0       //000
#define I86_PIC_ICW4_BUF_MODEYES        8       //1000          //Use when setting I86_PIC_ICW4_MASK_BUF
#define I86_PIC_ICW4_BUF_MODENO         0       //0000
#define I86_PIC_ICW4_SFNM_NESTEDMODE    0x10    //10000         //Use when setting I86_PIC_ICW4_MASK_SFNM
#define I86_PIC_ICW4_SFNM_NOTNESTED     0       //00000
```

This is simple snough, huh? ^_^ We can use the above control bits in conjunction with the bit masks to build up the control word. The naming convention used allows us to easily identify what bit masks they are used with.

I suppose thats it for the constants used in the implimentation. Lets get on with the functions...

i86_pic_send_command (): Sends a command to a PIC

This routine sends a command byte to the PIC's command register. **picNum** is a zero-based index representing the PIC we are accessing. On x86, this should either be a 0 or 1. Notice that we test what PIC we are working with in order to get the correct command register.

While this is part of the interface, it should not be used that much outside of the interface. It provides a method so we can manually send and control the PICs, if needed. **This will be required by the interrupt handlers to send the EOI command.**

```
inline void i86_pic_send_command (uint8_t cmd, uint8_t picNum) {

        if (picNum > 1)
                return;

        uint8_t reg = (picNum==1) ? I86_PIC2_REG_COMMAND : I86_PIC1_REG_COMMAND;
        outportb (reg, cmd);
}
```

i86_pic_send_data () and i86_pic_read_data (): Send and return a data byte to or from a PIC

These routine are very simular to the above routine, however it writes or reads to the PIC's data register depending on the PIC in **picNum**. Notice how both of these routines are **inline**. Because these routines are small, we want to take out the function call.

```
inline void i86_pic_send_data (uint8_t data, uint8_t picNum) {

        if (picNum > 1)
                return;

        uint8_t reg = (picNum==1) ? I86_PIC2_REG_DATA : I86_PIC1_REG_DATA;
        outportb (reg, data);
}

inline uint8_t i86_pic_read_data (uint8_t picNum) {

        if (picNum > 1)
                return 0;

        uint8_t reg = (picNum==1) ? I86_PIC2_REG_DATA : I86_PIC1_REG_DATA;
        return inportb (reg);
}
```

i86_pic_initialize (): Initializes the PICs

This is the final routine for the PIC interface. This initializes both PICs for operation using all of the routines above, and our constants defined for the initialization control words.

This routine is not too complex. Or, rather, not as complex as it looks ;) All it does is send the initialization command to the PIC. It does this by setting the **I86_PIC_ICW1_INIT_YES** bit in the command word. We also set the **I86_PIC_ICW1_IC4_EXPECT** bit. This insures that the controller expects us to send ICW 4. Notice how the constants help improve readability?

The ICW is stored in..well... **icw**. We send the command to both PICs using our **i86_pic_send_command()** routine.

After ICW 1 is sent, we begin initialization by sending ICW 2. Remember that ICW 2 containes the base interrupt numbers? This is passed into the **base0** and **base1** parameters.

Afterwords, we simply send ICW 3. Remember that ICW 3 provides the connection between the master and secondary PIC controllers.

Lastly is ICW 4. We set up x86 mode by setting the **I86_PIC_ICW4_UPM_86MODE** bit. Compare this routine with the example found in the PIC tutorial and be amazed...very amazed on their simularities!

```
//! Initialize pic
void i86_pic_initialize (uint8_t base0, uint8_t base1) {

        uint8_t         icw     = 0;

        //! Begin initialization of PIC

        icw = (icw & ~I86_PIC_ICW1_MASK_INIT) | I86_PIC_ICW1_INIT_YES;
        icw = (icw & ~I86_PIC_ICW1_MASK_IC4) | I86_PIC_ICW1_IC4_EXPECT;

        i86_pic_send_command (icw, 0);
        i86_pic_send_command (icw, 1);

        //! Send initialization control word 2. This is the base addresses of the irq's

        i86_pic_send_data (base0, 0);
        i86_pic_send_data (base1, 1);

        //! Send initialization control word 3. This is the connection between master and slave.
        //! ICW3 for master PIC is the IR that connects to secondary pic in binary format
        //! ICW3 for secondary PIC is the IR that connects to master pic in decimal format

        i86_pic_send_data (0x04, 0);
        i86_pic_send_data (0x02, 1);

        //! Send Initialization control word 4. Enables i86 mode

        icw = (icw & ~I86_PIC_ICW4_MASK_UPM) | I86_PIC_ICW4_UPM_86MODE;

        i86_pic_send_data (icw, 0);
        i86_pic_send_data (icw, 1);
}
```

Whew, I guess thats all of the big stuff for the PIC. All that is left is reprogramming the PIT. Don't worry--its not as complex as the PIC is. Lets take a look...

Programmable Interval Timer

Okay... So the PIC is ready to go, so we can now enable hardware interrupts, right? Yep--Kind of. While everything is okay so far, we still do not have an interrupt handler installed for the PIT yet. So, what will happen on the next timer tick? ...I think you know where I am getting at :)

A **Programmable Interval Timer (PIT)** is a counter which triggers an interrupts when they reach their programmed count. The 8253 and 8254 microcontrollers are PITs avialble for the i86 architectures used as timer for i86-compatable systems.

On x86 architectures, **The PIT acts as the system timer**, and is **connected to the PIC's IR0 line.** This allows the PIT to fire **IRQ 0** each timer tick. Because of this, we will need to reprogram this microcontroller before we can use it.

The PIT is a complex microcontroller to program. Because of this, we have created a separate tutorial for it. While I will still try to cover everything in detail, **I will not cover everything about the PIT here.**

Please see (and refrence) the following tutorial to learn about the PIT:

8253 Programmable Interval Timer Tutorial

pit.h: Interface

The good thing about the PIT is that it is not that complex to program. It does not contain that much commands, and yet does not need that much commands. It is a small, but powerful chip used for hardware timing and requests.

Operation Command Word

The PIT only containes one **Operation Command Word (OCW)** which is used to initialize a counter. It sets up the counters counting mode, operation mode, and allows us to set up an initil count value.

The command word is a little complex. Here is the complete command word:

- **Bit 0: (BCP)** Binary Counter
  - **0:** Binary
  - **1:** Binary Coded Decimal (BCD)
- **Bit 1-3: (M0, M1, M2)** Operating Mode. See above sections for a description of each.
  - **000:** Mode 0: Interrupt or Terminal Count
  - **001:** Mode 1: Programmable one-shot
  - **010:** Mode 2: Rate Generator
  - **011:** Mode 3: Square Wave Generator
  - **100:** Mode 4: Software Triggered Strobe
  - **101:** Mode 5: Hardware Triggered Strobe
  - **110:** Undefined; Don't use

    - **111:** Undefined; Don't use
- **Bits 4-5: (RL0, RL1)** Read/Load Mode. We are going to read or send data to a counter register
    - **00:** Counter value is latched into an internal control register at the time of the I/O write operation.
    - **01:** Read or Load Least Significant Byte (LSB) only
    - **10:** Read or Load Most Significant Byte (MSB) only
    - **11:** Read or Load LSB first then MSB
- **Bits 6-7: (SC0-SC1)** Select Counter. See above sections for a description of each.
    - **00:** Counter 0
    - **01:** Counter 1
    - **10:** Counter 2
    - **11:** Illegal value

Simular to the PIC's interface, we set up several bit masks that are used to describe the format of the command. Here it is...

```
#define         I86_PIT_OCW_MASK_BINCOUNT                       1       //00000001
#define         I86_PIT_OCW_MASK_MODE                           0xE     //00001110
#define         I86_PIT_OCW_MASK_RL                             0x30    //00110000
#define         I86_PIT_OCW_MASK_COUNTER                        0xC0    //11000000
```

Okay...While this is smaller then the ICWs and OCWs we set up in the PIC, this is actually more complex. The commands used in the PIC are simple in that they are 1 bit in size. The commands used in this operation command word are not.

This is where **Command Control Bits** shine. These help define the different settings and bit combinations for the different bit masks above. Here they are.

```
#define         I86_PIT_OCW_BINCOUNT_BINARY     0       //0             //! Use when setting I86_PIT_OCW_MASK_BINCOUNT
#define         I86_PIT_OCW_BINCOUNT_BCD        1       //1
#define         I86_PIT_OCW_MODE_TERMINALCOUNT  0       //0000          //! Use when setting I86_PIT_OCW_MASK_MODE
#define         I86_PIT_OCW_MODE_ONESHOT        0x2     //0010
#define         I86_PIT_OCW_MODE_RATEGEN        0x4     //0100
#define         I86_PIT_OCW_MODE_SQUAREWAVEGEN  0x6     //0110
#define         I86_PIT_OCW_MODE_SOFTWARETRIG   0x8     //1000
#define         I86_PIT_OCW_MODE_HARDWARETRIG   0xA     //1010
#define         I86_PIT_OCW_RL_LATCH            0       //000000        //! Use when setting I86_PIT_OCW_MASK_RL
#define         I86_PIT_OCW_RL_LSBONLY          0x10    //010000
#define         I86_PIT_OCW_RL_MSBONLY          0x20    //100000
#define         I86_PIT_OCW_RL_DATA             0x30    //110000
#define         I86_PIT_OCW_COUNTER_0           0       //00000000      //! Use when setting I86_PIT_OCW_MASK_COUNTER
#define         I86_PIT_OCW_COUNTER_1           0x40    //01000000
#define         I86_PIT_OCW_COUNTER_2           0x80    //10000000
```

Lets look at an example. Lets say we want to initialize counter 0 as a square wave generator in binary count mode. This is how we can do it:

```
        uint8_t ocw=0;
        ocw = (ocw & ~I86_PIT_OCW_MASK_MODE) | I86_PIT_OCW_MODE_SQUAREWAVEGEN;
        ocw = (ocw & ~I86_PIT_OCW_MASK_BINCOUNT) | I86_PIT_OCW_BINCOUNT_BINARY;
        ocw = (ocw & ~I86_PIT_OCW_MASK_COUNTER) | I86_PIT_OCW_COUNTER_0;
```

I think I am making this too easy, what do you think? :p This is all you need to do, and **ocw** will contain the operation command word that can be sent to the PIC. Notice how using these constants help both improve readabilty, but also to decrease the possibility for errors.

I guess thats all there is to pit.h. Lets dive into pit.cpp next, shall we? Wee...!!

pit.cpp: Implimentation

This containes the bulk of the PIT minidriver. It containes the implimentations of each routine used by both the interface and implimentation.

pit.cpp: Registers

This is where we define the constants to abstract the port locations for the PIT.

```
#define         I86_PIT_REG_COUNTER0            0x40
#define         I86_PIT_REG_COUNTER1            0x41
#define         I86_PIT_REG_COUNTER2            0x42
#define         I86_PIT_REG_COMMAND             0x43

//! Global Tick count
uint32_t                _pit_ticks=0;
```

Not to bad. **I86_PIT_REG_COUNTER0, I86_PIT_REG_COUNTER1,** and **I86_PIT_REG_COUNTER2** are the data registers for each counter. Remember that the PIT has three internal counters? **I86_PIT_REG_COMMAND** is our command register. We will need to write commands to the command register to control and operate the PIT.

Also, notice **_pit_ticks**. This is a very special and important global.

Remember that the PIT counter 0 connects to the IR0 line on the PIC? This means, when counter 0 fires, it will generate **Interrupt Request (IRQ) 0**. We will need to create and install an interrupt handler to handle this request.

All the interrupt handler needs to do is update the **Global Tick Count** for the system. That is what **_pit_ticks** is for.

i86_pit_irq(): PIT Counter 0 Interrupt Handler

This is the interrupt handler that handles the IRQ 0 request. Whenever Counter 0 fires, it will call this interrupt handler.

All it does is increment the global tick count whenever it fires. Note the general format for an interrupt handler.

intstart() is a macro used to disable hardware interrupts and save the stack frame so that we can return to the task without missing up its stack. **intret()** is a macro that disables hardware interrupts, restors the stack frame and returns from the handler using the **IRETD** instruction. The purpose of this is simply so that we can protect the current stack from being changed, and return back to the task with its stack intact. These macros are defined in **asm/system.h** so they can be used by the kernel and device drivers interrupt handlers.

interrupt is a special constant that is only used on certain compiliers. For MSVC++, it is defined as **__declspec (naked)**. This is so we don't need to worry about the compiliers added code. Some compiliers support this keyword directly (Most notably 16 bit compiliers). Others (Like MSVC++) do not, so we must define it.

interruptdone() is a special routine defined in the **Hardware Abstraction Layer**. It is resposible for sending the **End of Interrupt** commands to the PIC.

This is the generic format that all of our interrupt handlers will use.

```
void interrupt _cdecl i86_pit_irq () {

	//! macro to hide interrupt start code
	intstart ();

	//! increment tick count
	_pit_ticks++;

	//! tell hal we are done
	interruptdone(0);

	//! macro used with intstart to return from interrupt handler
	intret ();
}
```

i86_pit_send_command (): Send Command to PIT

This is a very important routine that allows us to send command to the PIT. This hides the command port we are sending it to, which is nice if we need to change the port name. The command is in the form of an **Operation Command Word (OCW)**.

```
//! send command to pic
void i86_pit_send_command (uint8_t cmd) {

	outportb (I86_PIT_REG_COMMAND, cmd);
}
```

For an example, we can build up an OCW using our bit masks and command control bits above. Then, we can use **i86_pit_send_command()** to send the OCW to the PIT.

i86_pit_send_data() and i86_pit_read_data(): Sends and reads data from counter

These routines help abstract the port name used when reading or writing to a counter. These are used to set and get the current count value. All they do is test the counter passed in **counter** to insure we get the correct port. Then, its just a simple read or write operation through that port.

```
//! send data to a counter
void i86_pit_send_data (uint16_t data, uint8_t counter) {

	uint8_t port= (counter==I86_PIT_OCW_COUNTER_0) ? I86_PIT_REG_COUNTER0 :
		((counter==I86_PIT_OCW_COUNTER_1) ? I86_PIT_REG_COUNTER1 : I86_PIT_REG_COUNTER2);

	outportb (port, data);
}

//! read data from counter
uint8_t i86_pit_read_data (uint16_t counter) {

	uint8_t port= (counter==I86_PIT_OCW_COUNTER_0) ? I86_PIT_REG_COUNTER0 :
		((counter==I86_PIT_OCW_COUNTER_1) ? I86_PIT_REG_COUNTER1 : I86_PIT_REG_COUNTER2);

	return inportb (port);
}
```

i86_pit_initialize (): Initialize the PIT

Okay, lets talk about initializing the PIT. Yeah! Well... er.. There really is not much to talk about, as it really does not require initialization. What we will need to do, however, is provide a way to install our interrupt handler. **irq** is the interrupt number to use and **irCodeSeg** is the **code seletor** offset into the **Global Descriptor Table (GDT)**.

We use our **i86_install_ir()** routine to install our interrupt handler (**i86_pit_irq**) into the **Interrupt Descriptor Table**. From here on out, IRQ 0 is mapped to our interrupt handler at **irq**. **irq** should be the same base IRQ number that the primary PIC was mapped to use to insure it is mapped to IRQ 0.

```
//! initialize minidriver
void i86_pit_initialize (uint8_t irq, uint8_t irCodeSeg) {

	//! Install our interrupt handler
	i86_install_ir (irq, I86_IDT_DESC_PRESENT | I86_IDT_DESC_BIT32,
		irCodeSeg, i86_pit_irq);
}
```

i86_pit_start_counter (): Starts an internal counter

This is the final routine in the PIT interface. This starts up a counter. We pass the counter into **counter** that we want to start (Such as **I86_PIT_REG_COUNTER0**). **mode** containes the operation mode that we want the counter to use (Such as **I86_PIT_OCW_MODE_SQUAREWAVEGEN**). **freq** containes the frequency rate that we want the counter to operate at.

This routine builds up the **operational command word** based on the paramaters passed into the routine.

```
void i86_pit_start_counter (uint32_t freq, uint8_t counter, uint8_t mode) {
```

```
        if (freq==0)
                return;

        uint16_t divisor = 1193180 / freq;

        //! send operational command
        uint8_t ocw=0;
        ocw = (ocw & ~I86_PIT_OCW_MASK_MODE) | mode;
        ocw = (ocw & ~I86_PIT_OCW_MASK_RL) | I86_PIT_OCW_RL_DATA;
        ocw = (ocw & ~I86_PIT_OCW_MASK_COUNTER) | counter;
        i86_pit_send_command (ocw);

        //! set frequency rate
        i86_pit_send_data (divisor & 0xff, 0);
        i86_pit_send_data ((divisor >> 8) & 0xff, 0);

        //! reset tick count
        _pit_ticks=0;
}
```

Conclusion

From here on out, all of the basics are completed. We have covered alot in this series, from processor modes and architecture, to processor tables, interrupts, interrupt management, and more. This is the beginning of the kernel, and where the kernel builds off from.

In this tutorial, we have added support for the PIC, PIT, exceptions, and hardware interrupt management. This is a important steps, as alot of important devices use hardware interrupts. Also, this gives us a chanch to re-enable hardware interrupts (Remember that we needed to disable hardware interrupts before the switch to protected mode?)

In the next tutorial, we will go back to the kernel itself. Its time to talk about one of the most fundemental aspects of any computer system: **Paging** and **Low Level Memory Management**. This will also be the foundation of our own **System API**.

I'll see you there... ;)

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 15 Home Chapter 17 

Operating Systems Development Series

Operating Systems Development - Physical Memory

by Mike, 2008

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

Introduction

Welcome!

In this tutorial we will be looking at managing one of our most important resources within a computer system: Physical Memory. We will be looking at how to get memory information, some more BIOS interrupts, memory manager concepts, and the design and implimentation for our complete physical memory manager.

This is one of those things that no one likes to do--but in the end it makes things much easier to work with. With that in mind, lets take a look at the list for this tutorial:

- Physical Memory
- Translation Lookaside Buffer (TLB)
- Memory Management Unit (MMU)
- Memory Managers
- Abtaining memory information
- Passing information from bootloader to kernel
- Designing and developing a physical memory manager

Alirighty then! Notice that I am not covering paging or virtual memory here. Instead, I want to keep the concepts of physical memory management and virtual memory management completely separate. The reasons for this is simplicity; we can focus on one without the other. Don't worry--we will cover paging and virtual memory in the next tutorial when we cover the development of a virtual memory manager.

Memory: A deeper look

Rather then jumping directly into memory management, I want to take a different approch here. That is, How can we even understand what memory management is about without even understanding what memory *itself* is? That is, we should know what is it that we are trying to manage, right?

Because of this, we will first look at what physical memory is first. You know... Those little RAM chips inside of your computer? :)

Here we go...!

Physical Memory

Physical Memory: Abstract

Physical Memory is an abstract block of memory stored within the computers **Random Access Memory (RAM)**. The way physical memory is "stored" within the RAM depends on the type of RAM the system uses. For example, **Dynamic Random Access Memory (DRAM)** stores each bit of data in its own **capacitor** that must be refreshed periodically. A **capacitor** is an electronic device that stores a current for a limited time. This allows it to either store a current (a binary 1), or no current (a binary 0). This is how DRAM chips store individual bits of data in a computer.

Most of the time, the memory types (RAM, SRAM, DRAM, etc.) require a specific type of **Memory Controller** to interface with the processor and **System Bus**.

The Memory Controller provides a way of reading and writing memory locations through software. The Memory Controller is also responsible for the constant refreshing of the RAM chips to insure they retain their information.

Memory Controllers contain Multiplexer and Demultiplexer circuits to select the exact RAM chip, and location that refrences the address in the **Address Bus**. This allows the **processor** to refrence a specific memory location by sending the memory address through the address bus.

...This is where the software come in, as they tell the processor what memory address to read :)

The Memory Controller selects the location within the RAM chip in a sequence manner. This means, if we access a physical memory location greater then the total amount of memory in the system, nothing will happen. That is, you can write a value to that memory location and read it back--you will get whatever left over data on the data bus.

It is possible for **memory holes** to appear in the **Physical Address Space**. This can happen if, for example, a RAM chip is in slots 1 and 3, with no RAM chip in slot 2. This means that there is an area of memory between the last byte stored in the RAM at slot 1 and the first byte-1 in slot 3 that does not exist. Reading or writing to these locations have almost the same effect when reading or writing beyond memory. If this nonexistant memory location has been remapped by the memory controller, you may be reading or writing to a different part of memory. If the memory has not been remapped (Which most of memory is not), **Reading or writing to a memory location that does not exist does nothing at all.** That is, writing to the non existant memory location will not write anything anywhere, reading from a non existant memory location reads whatever garbage was left over on the data bus. Knowing that writing a value to a non existant location and reading it back will NOT yeld the same value, methods have come up of manually parsing memory via pointers to determin what areas of memory are good or not. However, doing this can be dangerous as we will look at later.

Well, Thats all for what physical memory really is. Knowing how memory stores each bit you can probably start seeing where the bytes, words, dword, qwords, tbytes, etc.. start to come in. The most important of these are the **byte**, as that is the smallest data that the processor can access. But how does the processor know where a byte is located in memory? This is where the **Physical Address Space** comes in. Lets take a look :)

Physical Address Space (PAS)

This is the address space used by the processor (and translated by the memory controller) to refer to an 8 bit peice of data (ie, a byte) stored in physical memory (RAM). A **Memory Address** is just a number selected by the **Memory Controller** for a byte of data. For example, memory

address 0 can refer to the first 8 bits of physical memory, memory address 1 can refer to the next 8 bits, and so on. The **Physical Address Space** is the array of these **memory addresses** and the actual memory that the memory addresses refer to.

The **physical address space** is accessed by the **processor** through the systems **address bus** (Remember this from Chapter 7?)

Okay, so lessee... the processor now can use an address to refer to a byte of memory. It useually starts from address 0, and increments for each byte in memory. Thats as simple as it can get! But, it still doesnt describe how the software can access memory. Sure, the *processor itself* now has a way of refrencing memory, but the *software does not*. The processor, depending on its needs, need to provide specific ways for software to provide software a way to refrence memory. Wait, what? Thats right... different ways of addressing and accessing memory...

Addressing Modes

The **Addressing Mode** is an abstraction made by the processor to manage how the software accesses the **Physical Address Space**. This useually allows the software to set up the processors' registers so the processor knows how to refrence memory. We have already seen two: **segment:offset memory addressing** and **descriptor:offset memory addressing**.

This is the interface given by the processor for the software to allow a way to access memory.

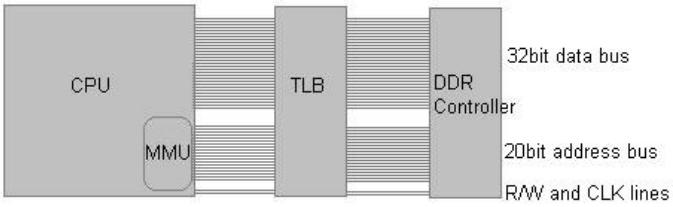
We have covered the segment:offset addressing mode in Chapter 4 and the descriptor:offset memory addressing mode in Chapter 8.

How Memory Works: Detail

Okay, now lets look at memory in a new way. We have already covered alot of details on what memory is, address space and addressing modes. Now, lets put everything together, shall we?

Alot of the information in this section is not needed, but I decided to include it for completness sake. Don't worry to much if you do not understand everything here.

In Chapter 7, we have looked at a basic overview of a computer system and system architecture. We have talked about how the processor's **system bus** connects to the **memory controller** which is used to provide the system a way to control physical RAM. Kind of like this:



Thats it! This is how the physical RAM connects and communicates with the rest of the system. In the above image, the **DDR Controller** is the memory controller. The **Translation Lookaside Buffer (TLB)** sits between the memory controller and processor. With this, the **system bus** connects all three of them through its **address bus**, **data bus** and **control bus**. The only 2 lines from the control bus that are important to us right now are the **RW line** and **CLK line.**

The TLB is only used when paging is enabled. Because of this, we will look at it a little more later.

Okay, so what actually happens when we write data to a physical memory location? During a **write operation**, the processor sets the RW pin to **high** (a logical 1). This tells the devices it connects to that a write operation is to take place. The processor resets the **IO Control line** to **low** (a logical 0). This insures the IO SubSystem ignores the command (Meaning its not an IN/OUT port instruction), but rather for the memory controller. The processor then copies the address to write to onto the address bus and the data to write onto the data bus. Because these lines are connected indirectly to the memory controller, the memory controller is able to see it is a write operation. So, all the memory controller needs to do is translate the memory address on the address bus using its demultiplexer circuit to find the RAM chip to use, and linear offset byte into the RAM chips memory space. The memory controller then copies the data from the data bus to this location, and refreshes the memory state on the next clock signal.

During a **read operation** it follows almost the same process as a write operation. Except, the **RW line** is set **low** to indicate a read operation. Also, the memory controller, after translating the memory address into an offset into the RAM chip, copies the data stored in that location and places it on the data bus for the processor. The memory controller then refreshes the memory state on the next clock signal.

The **CLK** signal is used to synchronize the exchange of addresses and data values through the reads and writes. Communication with the memory chip is started when the CLK line is a logical 1 (set high). During this period that the CLK line is held high the address is placed on the address lines and the R/W line is either taken high for a write or low for a read.

During execution the processor will constantly be flipping the clock line high and low in order to perform reads and writes with the memory controller.

While paging is disabled, the TLB itself does nothing at all. Notice that the TLB is not used at all when reading or writing memory.

Physical Memory Managers

As you may know, managing memory is critically important. All of our data and code share the same physical address space. If we attempt to load and work with more data or programs, we will need to some how find a way of managing memory to allow this.

At this stage, our kernel has full control of all of the hardware and memory in your computer. This is great but bad at the same time. We have no way of knowing what areas of memory are currently in use nor what areas are free. Because of this, there is no real way of working with memory without the possibility of problems: program corruption, data corruption, no way of knowing how memory is mapped, triple faults or other exception errors, etc... The results may be unpredictable.

Because of this, effectivly managing physical memory is very important. Lets look closer!

Detecting Memory

Abstract

The first thing we need to do is abtain the amount of RAM inside of the computer system. There are alot of different ways to do this. Some methods may work on some systems while others may not.

Getting the amount of memory can be very system dependent. More specifically, motherboard chipset dependent. During initialization, the BIOS abtains memory information from the memory controller and configures the chipset to work with the detected memory. Because of this, **Operating systems need to go through the system BIOS to abtain this information.** But Wait! I thought we were not able to use the Bios in protected mode!? Thats right...we can't. Instead we must get this information some other way... The bootloader, perhaps?

I should point out that there is indeed other methods that we can do to abtain the amount of memory within the system. For example, the CMOS, PnP, and SMBios. However the only way to guarantee that you get the correct amount is from the device that configuares it: The BIOS.

The last thing to note here is that all PCs are required to have a memory region just below the 4 GB mark for use by additional possible devices (Memory mapped hardware or the BIOS ROM). We will look at how to get around this a little later, don't worry :)

"Low Memory", otherwise known as **Conventional Memory** is memory below the 1MB mark. Memory above 1MB is known as **Extended Memory**.

With all of that in mind, lets take a look at some nice Bios interrupts to help us out...

Bios: Getting Memory Size

All of the following routines can be found in **memory.inc** in the demo at the end of this tutorial inside of the 2nd stage boot loader.

BIOS INT 0x12 - Get Memory Size (Conventional Memory)

Return

CF = Clear if successful
AX = Number of KB conventional memory
AH = status if error (0x80: Invalid command; 0x86) Unsupported function

This is quite possible the easiest method. This interrupt returns the value found in the BIOS data area (The word at physical address 0x413). Because it returns a WORD sized value, it is limited to 0xFFFF (65535 decimal). That is, it will only detect memory below 64KB. Because of this, on systems with more then 64KB of memory, it will not return the correct size. Thus, I will not be using this method.

While this method may not return the complete memory size, it is quite possibly the only one guranteed to work on almost all, if not all, PCs.

```
BiosGetMemorySize:
        int     0x12
        jc      .error
        test    ax, ax          ; if size=0
        je      .error
        cmp     ah, 0x86        ;unsupported function
        je      .error
        cmp     ah, 0x80        ;invalid command
        je      .error
        ret
.error:
        mov     ax, -1
        ret
```

BIOS INT 0x15 Function 0x88 - Get Extended Memory Size

Return

CF = Clear if successful
AX = Number of contiguous KB starting at 1MB physical address
AH = status if error (0x80: Invalid command; 0x86) Unsupported function

This interrupt returns the amount of KB extended memory in AX. Because it uses 16 bit registers, it is limited to returning 64MB, or 0xFFFFF (65535). On some versions of Windows, this function may return 15MB instead.

```
BiosGetExtendedMemorySize:
        mov     ax, 0x88
        int     0x15
        jc      .error
        test    ax, ax          ; if size=0
        je      .error
        cmp     ah, 0x86        ;unsupported function
        je      .error
        cmp     ah, 0x80        ;invalid command
        je      .error
        ret
.error:
        mov     ax, -1
        ret
```

BIOS INT 0x15 Function 0xE881 - Get Memory Size For > 64 MB Configuations (32 Bit)

Return

CF = Clear if successful
EAX = Extended memory between 1MB and 16MB in KB
EBX = Extended memory above 16MB, in 64KB blocks
ECX = Configured memory 1MB to 16MB in KB
EDX = Configured memory above 16MB in 64JB blocks

This interrupt is exactally like INT 0x15 Function 0xE801 except it uses the extended registers (EAX/EBX/ECX/EDX).

BIOS INT 0x15 Function 0xE801 - Get Memory Size For > 64 MB Configuations

Return

CF = Clear if successful
EAX = Extended memory between 1MB and 16MB in KB
EBX = Extended memory above 16MB, in 64KB blocks
ECX = Configured memory 1MB to 16MB in KB
EDX = Configured memory above 16MB in 64JB blocks

This is the method that I tend to use. This interrupt is used by both Windows NT and Linux during boot up to detect memory size if INT 0x15 function 0xe820 is not supported (Get System Memory Map). We will look at that later. This method has been around since around 1994 so older systems may not support this method.

The "Extended Memory" and "Configured Memory" values are almost always the same. Some BIOSs may store the results in either EAX and EBX or ECX and EDX. In other words, some BIOS may use EAX and EBX but leave ECX and EDX alone. Other BIOSs may do the exact opposite. Yey for standards! :) Uh, okay...Sorry ;)

The typical way of using this method is to null out all of the general-purpose registers first before calling the BIOS. This way, after the BIOS call, we can test if a register is null or not so we know what pair of registers to use: EAX/EBX or ECX/EDX:

```
;-------------------------------------------
;       Get memory size for >64M configuations
;       ret\ ax=KB between 1MB and 16MB
;       ret\ bx=number of 64K blocks above 16MB
;       ret\ bx=0 and ax= -1 on error
;-------------------------------------------

BiosGetMemorySize64MB:
        push    ecx
        push    edx
        xor     ecx, ecx                ;clear all registers. This is needed for testing later
        xor     edx, edx
        mov     ax, 0xe801
        int     0x15
        jc      .error
        cmp     ah, 0x86                ;unsupported function
        je      .error
        cmp     ah, 0x80                ;invalid command
        je      .error
        jcxz    .use_ax                 ;bios may have stored it in ax,bx or cx,dx. test if cx is 0
        mov     ax, cx                  ;its not, so it should contain mem size; store it
        mov     bx, dx

.use_ax:
        pop     edx                     ;mem size is in ax and bx already, return it
        pop     ecx
        ret

.error:
        mov     ax, -1
        mov     bx, 0
        pop     edx
        pop     ecx
        ret
```

Notice what this routine returns. In order for us to get the amount of KB in the system, we need to do some math. EBX contains the number of 64KB blocks of memory. If we multiply this by 64, we effectivly convert the value in EBX into the amount of KB above 16MB. Afterwords, simply add this number to the number returned in EAX to get the amount of KB above 1MB. Knowing there is 1024 KB in a one megabyte, add 1024 to this number and we now have the total amount of KB in the system!

Manually Probing Memory

Manually probing memory means to manually detect memory from a pointer by directly accessing memory. While it is possible that doing this may detect all of memory, it is also the most dangerous. Remember that there may be devices that we may or may not know of that uses regions of memory for different things. There may also be memory mapped devices, the ROM BIOS, and other devices that use memory. We also have not taken into account memory holes within the physical address space.

To directly probe memory comes from the fact that reading or writing to nonexistant memory does nothing. That is, if you write to a physical memory address that does not exist, you will not get an error. However, if you attempt to read from that same location again, the value that you get back may be completely random garbage--whatever was left on the data bus.

Thus to probe memory, just go into a loop for every 1k (or so) of memory. Use a pointer to read and write to a memory location. Continue incrementing the pointer (Hence reading from another location in memory) until the value read from the pointer containes an invalid value.

I might create a little demo code for this method, but we would probably never use it do to its many problems. I decided to include this method as it is a method to detect memory, although it is the most unsafe method to use and can cause unpredictable results. Use at your own risk!

Getting the Memory Map

Yippie! Now we have gotten the amount of memory in the system. But Wait! Not all of this memory is available to us, remember?

This is where a **Memory Map** comes in. A memory map defines what regions of memory are used for what. Using this, we can also abtain what regions that are safe for us to use.

BIOS INT 0x15 Function 0xE820 - Get Memory Map

Input

EAX = 0x0000E820
EBX = continuation value or 0 to start at beginning of map
ECX = size of buffer for result (Must be >= 20 bytes)

EDX = 0x534D4150h ('SMAP')
ES:DI = Buffer for result

Return

CF = clear if successful
EAX = 0x534D4150h ('SMAP')
EBX = offset of next entry to copy from or 0 if done
ECX = actual length returned in bytes
ES:DI = buffer filled
If error, AH containes error code

Address Range Descriptor

The buffer used by this interrupt as an array of descriptors that follow the following format:

```
struc   MemoryMapEntry
        .baseAddress    resq    1       ; base address of address range
        .length         resq    1       ; length of address range in bytes
        .type           resd    1       ; type of address range
        .acpi_null      resd    1       ; reserved
endstruc
```

Types of Address Ranges

The types of address ranges defined for this function is shown below:

- 1: Available Memory
- 2: Reserved, do not use. (e.g. system ROM, memory-mapped device)
- 3: ACPI Reclaim Memory (usable by OS after reading ACPI tables)
- 4: ACPI NVS Memory (OS is required to save this memory between NVS sessions)
- All other values should be treated as undefined.

Getting the Memory Map

This interrupt might seem a little complex, but its not to bad.

First take a look at the **inputs** that this interrupt requires. We put the function number (0xe820), of course, in AX. However some BIOSs **require** that the upper half of EAX is zero. Because of this, you should us EAX here instead of AX.

Also notice that EDX must contain the value of 'SMAP'. This is another **requirement**. Some BIOSs may trash this register after calling the interrupt.

Okay... When we execute this interrupt, the BIOS will return a single entry of the memory map (This entry has a format. Please see the **Address Range Descriptor** above). If, after calling the interrupt, EBX is NOT zero, then there are more entries in the memory map. We will need to loop for each entry in the map. If the entry length is 0, then skip the entry as there is nothing there and go to the next entry in the list until we reach the end.

This routine uses the MemoryMapEntry structure we have defined above to get information from the entries that we abtain from the bios.

```
;-----------------------------------------
;       Get memory map from bios
;       /in es:di->destination buffer for entries
;       /ret bp=entry count
;-----------------------------------------

BiosGetMemoryMap:
        pushad
        xor     ebx, ebx
        xor     bp, bp                  ; number of entries stored here
        mov     edx, 'PAMS'             ; 'SMAP'
        mov     eax, 0xe820
        mov     ecx, 24                 ; memory map entry struct is 24 bytes
        int     0x15                    ; get first entry
        jc      .error
        cmp     eax, 'PAMS'             ; bios returns SMAP in eax
        jne     .error
        test    ebx, ebx                ; if ebx=0 then list is one entry long; bail out
        je      .error
        jmp     .start
.next_entry:
        mov     edx, 'PAMS'             ; some bios's trash this register
        mov     ecx, 24                 ; entry is 24 bytes
        mov     eax, 0xe820
        int     0x15                    ; get next entry
.start:
        jcxz    .skip_entry             ; if actual returned bytes is 0, skip entry
.notext:
        mov     ecx, [es:di + MemoryMapEntry.length]    ; get length (low dword)
        test    ecx, ecx                ; if length is 0 skip it
        jne     short .good_entry
        mov     ecx, [es:di + MemoryMapEntry.length + 4]; get length (upper dword)
        jecxz   .skip_entry             ; if length is 0 skip it
.good_entry:
        inc     bp                      ; increment entry count
        add     di, 24                  ; point di to next entry in buffer
.skip_entry:
        cmp     ebx, 0                  ; if ebx return is 0, list is done
        jne     .next_entry             ; get next entry
        jmp     .done
.error:
```

```
        stc
.done:
        popad
        ret
```

Multiboot Specification

I do not plan on covering the multiboot specification too soon. Mabey in the future, but not now. However, we need a way to pass the information that we abtained from the BIOS inside of our bootloader to our Kernel. We can do this any way that we want. Because the multiboot specification defines a standard boot time information structure, and I dont know if we will fully support the multiboot standard, I decided why not use the same structure?

Also, if you decide to use another boot loader (Such as GRUB), we can have it boot our kernel as well.

Anyways, the entire specification itself is rather large, so covering it in a tutorial about memory management is not a good idea ;) Thus, I will cover just enough so we can use it to pass the information that we need, sound cool?

Abstract

The Multiboot specification is a list of standards used to describe standards for boot loaders for loading and executing operating system kernels. This specification makes it easier to boot multiple operating systems because the specification describes a standard state that the machine must be in before the operating system takes control. **This also includes how and what information is passed from the bootloader to the kernel.**

I will not be covering the complete multiboot specification right now. However, we will be looking at what the machine state must be in when the kernel is executed. We will also be looking a little bit at the **Multiboot information** structure which containes the information passed from the boot loader to the kernel. We will also be using this structure to pass our bootloader memory information using this structure as well.

Machine State

The Multiboot specification states that, when we invoke a 32 bit operating system (That is, execute our kernel), the machine registers must be set to a specific state. More specifically: **When we execute our kernel, set up the registers to the following values:**

- EAX - Magic Number. Must be **0x2BADB002**. This will indicate to the kernel that our boot loader is multiboot standard
- EBX - Containes the physical address of the **Multiboot information structure**
- CS - Must be a 32-bit read/execute code segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact value is undefined.
- DS,ES,FS,GS,SS - Must be a 32-bit read/write data segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact values are all undefined.
- A20 gate must be enabled
- CR0 - Bit 31 (PG) bit must be cleared (paging disabled) and Bit 0 (PE) bit must be set (Protected Mode enabled). Other bits undefined

All other registers are undefined. Most of this is already done in our existing boot loader. The only additional two things we must add are for the EAX register and EBX.

The most important one for us is stored in EBX. This will contain the physical address of the multiboot information structure. *Lets take a look!*

Multiboot Information Structure

This is possibly one of the most important structures contained in the multiboot specification. The information in this structure is passed to the kernel from the EBX register, **This allows a standard way for the boot loader to pass information to the kernel**.

This is a fairly big structure but isnt to bad. Not all of these members are required. The specification states that the operating system must use the flags member to determin what members in the structure exist and what do not.

```
struc multiboot_info
        .flags          resd    1       ; required
        .memoryLo       resd    1       ; memory size. Present if flags[0] is set
        .memoryHi       resd    1
        .bootDevice     resd    1       ; boot device. Present if flags[1] is set
        .cmdLine        resd    1       ; kernel command line. Present if flags[2] is set
        .mods_count     resd    1       ; number of modules loaded along with kernel. present if flags[3] is set
        .mods_addr      resd    1
        .syms0          resd    1       ; symbol table info. present if flags[4] or flags[5] is set
        .syms1          resd    1
        .syms2          resd    1
        .mmap_length    resd    1       ; memory map. Present if flags[6] is set
        .mmap_addr      resd    1
        .drives_length  resd    1       ; phys address of first drive structure. present if flags[7] is set
        .drives_addr    resd    1
        .config_table   resd    1       ; ROM configuation table. present if flags[8] is set
        .bootloader_name resd   1       ; Bootloader name. present if flags[9] is set
        .apm_table      resd    1       ; advanced power management (apm) table. present if flags[10] is set
        .vbe_control_info resd  1       ; video bios extension (vbe). present if flags[11] is set
        .vbe_mode_info  resd    1
        .vbe_mode       resw    1
        .vbe_interface_seg resw 1
        .vbe_interface_off resw 1
        .vbe_interface_len resw 1
endstruc
```

Alot of information in this structure! Most of this does not apply to use, so we will only focus on a few. **memLo** and **memHi** containes the amount of memory we detected from the BIOS. **mmap_length** and **mmap_addr** will point to the memory map that we abtained from the BIOS.

Thats it! Now we have a nice way of passing our memory information (And mabey more ;)) to the kernel:

```
        mov     eax, 0x2BADB002         ; multiboot specs say eax should be this
        mov     ebx, 0
        mov     edx, [ImageSize]

        push    dword boot_info
        call    ebp                     ; Execute Kernel
        add     esp, 4

        cli
        hlt
```

...And inside of our kernel:

```
//! kernel entry point is called by boot loader
void __cdecl  kernel_entry (multiboot_info* bootinfo) {

        //*snip*
}
```

The kernels multiboot_info structure is the same one shown above but in C. Thanks to this setup, all the kernel needs to do is access the memory information (And any information passed from the bootloader) through **bootinfo**. Cool, huh?

Now that we got the memory information from the Bios and passed it to the kernel, the kernel can use it for its physical memory manager. Thats right...Its finally time to develop a physical memory manager!

Physical Memory Management

We have covered alot already, don't you think? We have looked at how to abtain memory information from the BIOS and using the multiboot information structure to pass this information to the kernel. This allows the kernel to be able to retrieve this memory information anytime it wants. Yet, we have yet to cover the most important topic: **Managing** this memory. Lets look closer...

Memory Management: Abstract

Okay, so we know we need a way to manage memory. To do this, we--of course--need a way of keeping track of how memory is being used. Doing this for every byte in memory is simply not possible: How can we store information about every byte in memory without running out of memory? Because of this, we need to come up with another way.

What we need to insure is that, whatever data structure we use to manage the rest of memory, is smaller then the total size of memory. For example, we can use an array of bytes, for example. Every byte can store information about a larger block of memory. This is the only way we can insure we do not run out of memory.

The size of a "block" of memory must be a feasable and efficiant size. Using this method, we can split up the physical address space into "block" sized chunks. When we allocate memory, we do not allocate bytes. Rather, we allocate blocks of memory. **This is what a physical memory manager does**.

The purpose of a physical memory manager is to split up the computers physical address space into block-sized chunks of memory and provide a method to allocate and release them.

On the x86 architecture, when paging is enabled, each page represents a 4KB block of memory. Because of this, to keep things simple, each memory block for our physical memory manager will also be 4KB in size.

Setting up

Allright...We know that a physical memory manager is important. We also know that the physical memory manager needs to split up the physical address space and keep track of what memory blocks are being used or available. But wait! We immediately run into a problem: The kernel needs an area of memory for managing memory. How can we allocate an area of memory before we can allocate memory?

We cant. Because of this, the only method that we have is using a pointer to a location in memory. Think of this location as simply some more reserved memory, simular to the BIOS, Bios Data Area (BDA) and the kernel itself. We want to stick this somewhere in reserved memory, so how about at the end of the kernel? Afterwords, we can mark this region (along with the kernel itself) as reserved within the data structure to insure nothing touches it.

Great! Now that we have a pointer to some location in memory, we can store the information needed to keep track of each block in memory. But...*how?* That is, all we have is a pointer. The data this pointer points to must be in some useable structure so that we can use the area of memory effectivly. How can we create a structure to manage all of physical memory?

There are two common solutions to this: Stack or a bit map.

Stack based allocation

Bit map based allocation

This is the simplest to impliment. All our physical memory manager needs to know is if a block of memory is allocated or not. If it is allocated, we can use a binary bit 1. If it is not, a binary bit 0. In other words, for every block in memory, we use a single bit to represent if it has been allocated or not. This is the method that we will be using. However, the physical memory manager is designed in a way to allow other methods (Like the stack-based approch) if we decide too switch. ;)

The bit map approach is very efficiant in size. Because each bit represents a block of memory, a single 32 bits using this bit map approach represents 32 blocks. Because 32 bits is 4 bytes, this means we can watch--out of 32 blocks of memory--using only 4 bytes of memory.

This approach is a bit slower though as it requires searching the bit map for a free block (The first bit that is 0) every time we want to allocate a block of memory.

Developing a Physical Memory Manager (PMM)

In the upcoming demo code, the entire physical memory manager can be found in **mmngr_phys.h** and **mmngr_phys.cpp**. It may also help to study the updated 2nd stage boot loader to see how the memory information is passed from the boot loader to the kernel, and how the kernel initializes the PMM.

Globals and Constants

I never like "magic numbers" as you may have noticed ;) This is why I tend to hide all of these numbers behind more readable constants.

```
//! 8 blocks per byte
#define PMMNGR_BLOCKS_PER_BYTE 8

//! block size (4k)
#define PMMNGR_BLOCK_SIZE       4096

//! block alignment
#define PMMNGR_BLOCK_ALIGN      PMMNGR_BLOCK_SIZE
```

These are just to help with the readability of the code. The PMM creates an abstraction known as a **Memory Block**. A Memory Block is 4096 bytes in size (4K). This is important as it is also the size of a page when we enable paging.

There are also several globals defined for keeping track of everything.

```
//! size of physical memory
static  uint32_t        _mmngr_memory_size=0;

//! number of blocks currently in use
static  uint32_t        _mmngr_used_blocks=0;

//! maximum number of available memory blocks
static  uint32_t        _mmngr_max_blocks=0;

//! memory map bit array. Each bit represents a memory block
static  uint32_t*       _mmngr_memory_map= 0;
```

The most important of these is **_mmngr_memory_map**. This is a pointer to the bit map structure that we use to keep track of all of physical memory. **_mmngr_max_blocks** containes the amount of memory blocks available. This is the size of physical memory (Retrieved from the BIOS from the boot loader) divide by PMMNGR_BLOCK_SIZE. This essentally divides the physical address space into memory blocks (Remember this from before?) **_mmngr_used_blocks** containes the amount of blocks currently in use, **_mmngr_memory_size** is for refrence only--it containes the amount of physical memory in KB.

Memory Bit Map

Okay then! **_mmngr_memory_map** is a pointer to an uint32_t...right? Well, of course... sort of. Rather, we should think of it as "a pointer to a series of bits" as that is the way we treat it. Each bit is a 0 if that block has not been allocated (Useable) or a 1 if it is reserved (In use). The number of bits in this array is **_mmngr_max_blocks**. In other words, each bit represents a single memory block, which in turn, is 4KB of physical memory.

Knowing this, all we need to do with the bit map is be able to set a bit, unset a bit, and test if a bit is set. Lets take a look...

mmap_set () - Sets a bit in the bit map

What we want to do is provide a way so that we can think of the memory map as an array of bits rather then an array of ints. This is not to hard:

```
inline void mmap_set (int bit) {

  _mmngr_memory_map[bit / 32] |= (1 << (bit % 32));
}
```

The bit is a value from 0...x, where x is the bit that we want to set in the memory map. We divide the bit by 32 to get the integer index in _mmngr_memory_map that the bit is in.

To use this routine, simply call it passing in the bit that you want to set. You are not limited to 32 bits: mmp_set(62) sets the 62nd bit in the memory map bit array.

mmap_unset () - Unsets a bit in the bit map

This is very simular to the above routine but clears the bit instead:

```
inline void mmap_unset (int bit) {

  _mmngr_memory_map[bit / 32] &= ~ (1 << (bit % 32));
}
```

mmap_test () - Test if a bit is set

This routine simply returns true if a bit is 1 or false if it is 0. It is very simular to the above routine, but instead of setting the bit we use it as a mask and return its value:

```
inline bool mmap_test (int bit) {

 return _mmngr_memory_map[bit / 32] &  (1 << (bit % 32));
}
```

Thats all too it! Now that we have a way to set, unset, and test bits inside of the bit map, we need a way of searching through the bit map for free bits. These will be used so we can find free memory blocks that we can use.

mmap_first_free () - Returns index of first free bit in bit map

This routine is a little complex. We have a way to set, clear, and test a bit in the memory bit map. Lets say that we want to allocate a block of memory. How do we find a free block of memory? Thanks to our bit map, all we need to do is traverse the bit map for a bit that is not set. This isnt to complex:

```
int mmap_first_free () {

	//! find the first free bit
	for (uint32_t i=0; i< pmmngr_get_block_count() / 32; i++)
		if (_mmngr_memory_map[i] != 0xffffffff)
			for (int j=0; j<32; j++) {		//! test each bit in the dword

				int bit = 1 << j;
				if (! (_mmngr_memory_map[i] & bit) )
					return i*4*8+j;
			}

	return -1;
}
```

pmmngr_get_block_count() returns the maximum number of memory blocks in this system (Remember this is also the number of bits in the bit array?) We divide this by 32 (32 bits per dword) to get the amount of integers in ths bit map. In other words: The outmost loop simply loops through each integer in the array.

We then test to see if the dword is all set. We loop this in dwords rather then bits because it is much more efficiant and faster. We test it by insuring its not 0xffffffff. If it is, go on to the next dword. If it is not, then a bit must be clear. Afterwords we simply go through each bit in that dword to find the free bit and returns its physical frame address.

The physical memory manager includes another version of this routine-- **mmap_first_free_s()** that returns the index of the first free series of frames of a specific size. This allows us to insure a certain region of memory blocks are free rather then a single block. This routine is a little tricky I admit; if any of our readers do not understand the code I would be glad to describe it in more detail in this tutorial.

Physical Memory Allocater

We now have a way of managing memory. Wait, what? Thats right. The way this works is to remember that each bit in the bit map represents 4KB of physical memory. If we want to allocate the first memory block (The first 4k) just set bit 0. If you want to allocate the second 4k, just set bit 1. This continues to the end of memory. This provides a way for us to not only work in 4k blocks of memory, but also to know what memory is currently in use or reserved (The bit is 1) or free for use (Bit 0). All of this provided by the three simple routines above and our bit map array. Cool, huh?

Now we just need the actual allocation and deallocation routines. Before that, however, we need to initialize the bit map regions to that of the BIOS memory map. And even before THAT, we need to provide a way so the kernel can provide some information for our physical memory manager to use. Lets take a look...

pmmngr_init () - Initialize the physical memory manager

This routine is called by the kernel to initialize the physical memory manager (PMM). **memSize** is the maximum amount of memory the PMM is allowed to access. This should be the size of RAM in KB passed to the kernel from the bootloader. **bitmap** is the location that the PMM will use for its memory bit map structure. Another important thing to note is how we set all the bits in the memory bit map using a memset() call. There is a reason for this which will be looked at very soon.

```
void	pmmngr_init (size_t memSize, physical_addr bitmap) {

	_mmngr_memory_size	=	memSize;
	_mmngr_memory_map	=	(uint32_t*) bitmap;
	_mmngr_max_blocks	=	(pmmngr_get_memory_size()*1024) / PMMNGR_BLOCK_SIZE;
	_mmngr_used_blocks	=	pmmngr_get_block_count();

	//! By default, all of memory is in use
	memset (_mmngr_memory_map, 0xf, pmmngr_get_block_count() / PMMNGR_BLOCKS_PER_BYTE );
}
```

pmmngr_init_region () - Initializes a region of memory for use

Remember the memory map? We do not know what areas of memory are safe to work with, only the kernel does. Because of this, by default all of memory is in use. The kernel abtains the memory map from the kernel and uses this routine to initialize available regions of memory that we can use.

The routine is very simple. It just finds out how much memory blocks to set and loops--clearing the appropriate bits in the memory bit map. This allows the allocation routines to use these now free areas of memory again:

```
void	pmmngr_init_region (physical_addr base, size_t size) {

	int align = base / PMMNGR_BLOCK_SIZE;
	int blocks = size / PMMNGR_BLOCK_SIZE;

	for (; blocks>0; blocks--) {
		mmap_unset (align++);
		_mmngr_used_blocks--;
	}

	mmap_set (0);	//first block is always set. This insures allocs cant be 0
}
```

Notice the call to mmap_set() at the end. In our PMM, the first memory block block will always be set. This insures us that the PMM can return null pointers for allocation errors. This also insures that any data structures defined within the first 64 KB of memory are not overwritten or touched, including the Interrupt Vector Table (IVT) and Bios Data Area (BDA).

pmmngr_deinit_region () - Deinitializes a region of memory for use

This routine is simular to the above routine, but sets the bits instead of clearing them. Because the bits become a binary 1, that 4KB block of memory the bit represents is effectivly set to reserved so that area of memory will never be touched when this routine is called.

```
void    pmmngr_deinit_region (physical_addr base, size_t size) {

        int align = base / PMMNGR_BLOCK_SIZE;
        int blocks = size / PMMNGR_BLOCK_SIZE;

        for (; blocks>0; blocks--) {
                mmap_set (align++);
                _mmngr_used_blocks++;
        }
}
```

Woohoo! Now that we have a way to initialize and deinitialize regions of memory for use, and initialize the PMM we can work on allocating and deallocating blocks next!

pmmngr_alloc_block () and pmmngr_alloc_blocks () - Allocates a single block of physical memory

To allocate a block of memory is very simple. All of physical memory is already there, all we need to do is return a pointer to a free block of memory. We can find a free block of memory by looking through the bit map using out **mmap_first_free()** routine. Also notice that we call **mmap_set** to set the same frame returned from **mmap_first_frame ()**. This is what marks that block of memory just allocated is now "in use". This routine returns a void* to the 4KB of physical memory just allocated.

```
void*   pmmngr_alloc_block () {

        if (pmmngr_get_free_block_count() <= 0)
                return 0;       //out of memory

        int frame = mmap_first_free ();

        if (frame == -1)
                return 0;       //out of memory

        mmap_set (frame);

        physical_addr addr = frame * PMMNGR_BLOCK_SIZE;
        _mmngr_used_blocks++;

        return (void*)addr;
}
```

The PMM also containes another allocation routine: **pmmngr_alloc_blocks()**. This routine is almost exactly like the above so I decided to leave it out of this tutorial for space purposes. It provides a way to allocate a sequencial amount of blocks rather then a single block.

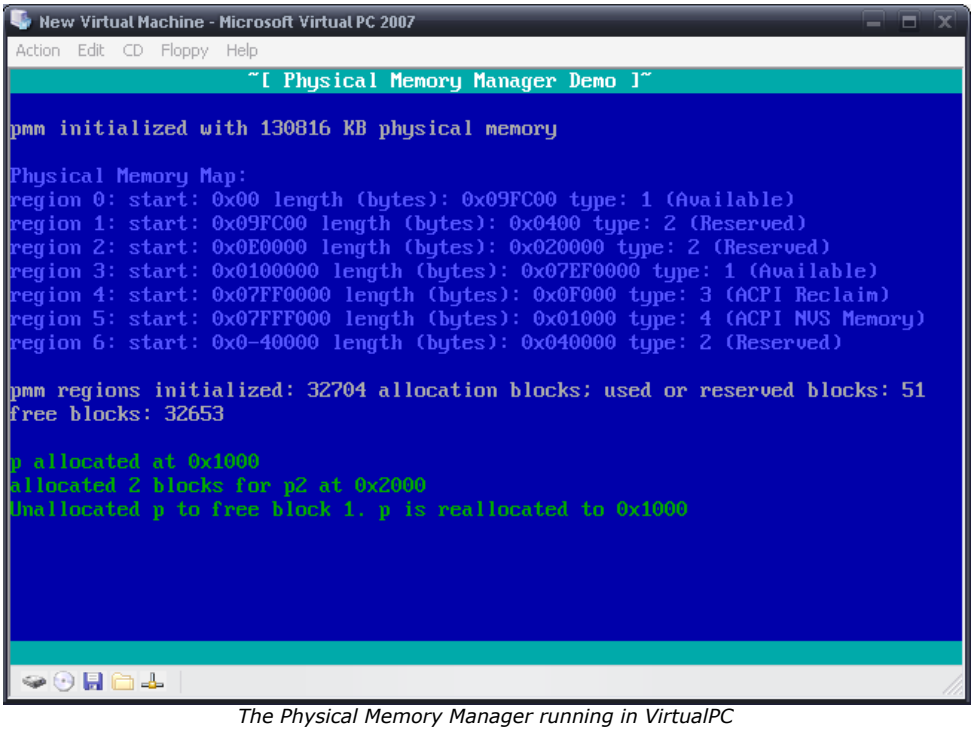
pmmngr_free_block () and pmmngr_free_blocks () - Releases a block of physical memory

Alright...We now have a way to allocate blocks of physical memory. Now we need a way to free these blocks of memory so we do not run out of memory. This is too easy:

```
void    pmmngr_free_block (void* p) {

        physical_addr addr = (physical_addr)p;
        int frame = addr / PMMNGR_BLOCK_SIZE;

        mmap_unset (frame);

        _mmngr_used_blocks--;
}
```

pmmngr_free_blocks() works almost the same way but is used in conjunction with **pmmngr_alloc_blocks()** used to free a sequencial amount of blocks rather then a single block.

Demo



The Physical Memory Manager running in VirtualPC

Demo Download (MSVC++)

I decided to spice things up a little. The memory map from Bochs is boring ;) This demo is quite nice in a way that you can run it on alot of different machines and see how different computer systems map the regions of physical memory. This demo should work with all optomization levels as well.

This demo uses the passed multiboot information structure from the bootloader to get the size of physical memory. My VirtualPC is set to use 130 MB of memory so I would say it detected it pretty well :)

There is alot going on in this demo. Some updates in the second stage boot loader is used to detect memory using the methods we looked at above, and passes this information to the kernel using the multiboot information structure in which we looked at above, remeber?

Play around with the allocations and deallocations and study the way the code works. If any of the allocation routines return null (0) then it indicates that there is no more free blocks left. If you are out of memory, try to free some memory or allocate the objects on the stack or globally instead.

There is an important thing to note here: Notice how all allocations are aligned on 4k boundaries? This is a very important characteristic when we start getting into pages and virtual memory in the next tutorial.

Conclusion

This tutorial was not to bad, huh?

We first looked at physical memory itself; understanding what it is and how it works, the physical address space and addressing modes. We have also looked at how to abtain memory information from the BIOS and give it to the kernel as well as the development of a physical memory manager.

Now we have a way of allocating and releasing physical memory blocks. This is great, but it still has some problems. That is, if we load a file or program we can simply use our physical memory manager to allocate an area of memory large enough for the file or program. But...What if there is no area big enough for it? This also means any programs loaded must be linked to a specific address that it is loaded at by the kernel.

This is where virtual memory and paging comes in. In the next tutorial, we will be looking at paging and virtual memory. We will learn how we can map and control the full 4GB address space. We will look at what virtual addressing is all about, and how we can use it to fix all of the above problems and more. I'll see you there.

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 16 Home Chapter 18

Operating Systems Development - Virtual Memory

by Mike, 2008

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

Introduction

Welcome back! Jeeze, I cant believe we are already going on tutorial eighteen. See? OS development isn't too bad ;)

In the last tutorial we have looked at physical memory management and even developed a full working physical memory manager. In this tutorial, we will take it to a new level by introducing paging and virtual memory. We will learn how we can mimic a full virtual address space for our programs and learn how we can manage virtual memory.

Heres the list for this chapter:

- Virtual Memory
- Memory Management Unit (MMU)
- Translation Lookaside Buffer (TLB)
- PAE and PSE
- Paging Methods
- Pages and Page Faults
- The Page Table
- The Page Directory Table
- Implimenting Paging

...And a whole lot more!

This tutorial will build off of the physical memory manager we developed in the last chapter. This may also be the last chapter on memory management!

With that in mind, lets get started!

Virtual Memory Concepts

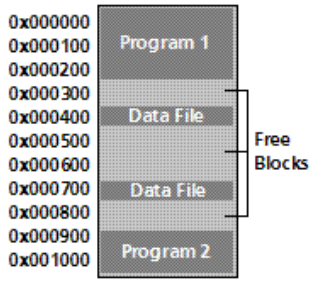
The need for Virtualization

You might be curious as to why we should worry about this "virtual memory" thing. After all, we already have a nice and effective way of managing memory, right? Well, sort of. While it manages blocks of memory well, thats all our physical memory manager does. This alone is pretty useless, don't you think?

There are alot of very important concepts that we should look at to better understand virtual memory and the need for it.

Right now all we have is a way to directly and indirectly work with physical memory. There are alot of big problems with this that you may already know (or even have experience with yourself ;)) One that we have just seen was when we would access to a block of memory that does not exist. Knowing that both programs and data are in memory, it is also possible for programs to access each others memory spaces, or even corrupt and overwrite themselves or other programs without knowing it. After all, there is no memory protection.

Also, it is not always possible to load a file or program into a sequencial area of memory. This is when fragmentation happens. For an example, lets say we have 2 programs loaded. One at 0x0, the other at 0x900. Both of these programs requested to load files, so we load the data files:



Notice what is happening here. There is alot of unused memory between all of these programs and files. Okay...What happens if we add a bigger file that is unable to fit in the above? This is when big problems arise with the current scheme. We cannot directly manipulate memory in any specific way, as it will currupt the currently executing programs and loaded files.

As you can see, there are alot of problems that will arise when working with physical memory. If your operating system is single-tasking (Where only one ring 0 program runs at a time), then this might be fine. For anything more complex, we will be needing more control over how memory works within the system. What we need is a way to abstract physical memory in such

a way that we do not need to worry about these details anymore. I think you know where I am getting at here -- this is where virtualization comes in. Lets take a look!

Virtual Memory

Concepts

Understanding what virtual memory is can be a little tricky. Virtual Memory is a special Memory Addressing Scheme implimented by both the hardware and software. It allows non contigous physical memory to act as if it was contigius memory.

Notice that I said *"Memory Addressing Scheme"*. What this means is that virtual memory allows us to control what a **Memory Address** refers to.

Virtual Address Space (VAS)

A Virtual Address Space is a Program's Address Space. One needs to take note that this does not have to do with **Physical Memory**. The idea is **so that each program has their own independent address space. This insures one program cannot access another program, because they are using a different address space.**

Because **VAS** is **Virtual** and not directly used with the physical memory, it allows the use of other sources, such as disk drives, as if it was memory. That is, **It allows us to use more "memory" then what is physically installed in the system.**

This fixes the "Not enough memory" problem.

Also, as each program uses its own VAS, we can have each program always begin at base 0x0000:0000. This solves the relocation problems discussed ealier, as well as memory fragmentation--as we no longer need to worry about allocating continous physical blocks of memory for each program.

Virtual Addresses are mapped by the Kernel trough the MMU. More on this a little later.

Memory Management Unit (MMU)

The **Memory Management Unit (MMU)** (Also known as **Paged Memory Management Unit (PMMU)**) sets between (Or as part of) the **microprocessor** and the **memory controller**. While the **memory controller's** primary function is the translation of memory addresses into a physical memory location, the **MMU**'s purpose is the translation of virtual memory addresses into a memory address for use by the **memory controller**.

This means--**when paging is enabled, all of our memory refrences go through the MMU first!**

Translation Lookaside Buffer (TLB)

This is a cache stored within the processor used to improve the speed of virtual address translation. It is useually a type of **Content-addressable memory (CAM)** where the search key is the virtual address to translate, and the result is the physical frame address. If the address is not in the TLB (A **TLB miss**), the MMU searches through the page table to find it. If it is found in the TLB, it is a **TLB Hit**. If the page is not found or invalid inside of the page table during a TLB miss, the processor will raise a **Page Fault** exception for us.

Think of a TLB as a table of pages stored in a cache instead of in RAM--as that is basically what it is.

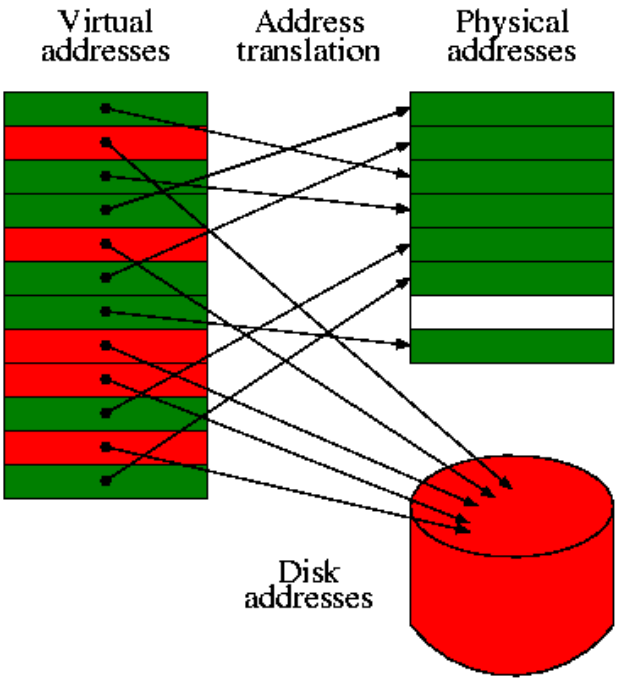
This is important! The pages are stored in **page tables**. We set up these page tables to describe how physical addresses translate to virtual addresses. In other words: **The TLB translates virtual addresses into physical addresses using the page tables *we* set up for it to use!** Yes, thats right--we set up what virtual addresses map to what. We will look at how to do this a little later, cool? Dont worry--its not that bad ;)

Paged Virtual Memory

Virtual Memory also provides a way to indirectly use more memory then we actually have within the system. One common way of approching this is by using **Page files**, stored on a hard drive or a **swap partition**.

Virtual Memory needs to be mapped through a hardware device controller in order to work, as it is handled at the hardware level. This is normally done through the **MMU**, which we will look at later.

For an example of seeing virtual memory in use, lets look at it in action:



Notice what is going on here. Each memory block within the Virtual Addresses are linear. Each Memory Block is mapped to either it's location within the real physical RAM, or another device, such as a hard disk. The blocks are swapped between these devices as an as needed bases. This might seem slow, but it is very fast thanks to the MMU.

Remember: Each program will have its own Virtual Address Space--shown above. Because each address space is linear, and begins from 0x0000:00000, this immiedately fixes alot of the problems relating to memory fragmentation and program relocation issues.

Also, because Virtual Memory uses different devices in using memory blocks, it can easily manage more then the amount of memory within the system. i.e., If there is no more system memory, we can allocate blocks on the hard drive instead. If we run out of memory, we can either increase this page file on an as needed bases, or display a warning/error message,

Each memory "Block" is known as a **Page**, which is useually 4096 bytes in size. We will cover **Pages** a little later.

Okay, so a **Page** is a memory block. **This memory block can either be mapped to a location in memory, or to another device location, such as a hard disk.** This is an **unmapped** page. If software accessed an unmapped page (The page is not currently in memory), it needs to be loaded somehow. This is done by our **Page fault handler**.

We will cover everything later, so do not worry if this sounds hard :)

Because we are talking about paging in general, I think now would be a good idea to look at some extensions that may be used with paging. Lets have a look!

PAE and PSE

Physical Address Extension (PAE)

PAE is a feature in x86 microprocessors that allows 32 bit systems to access up to 64 GB of physical memory. PAE supported motherboards use a 36 line address bus to achieve this. Paging support with PAE enabled (Bit 5 in the cr4 register) is a little different then what we looked at so far. I might decide to cover this a little later, however to keep this tutorial from getting even more complex, we will not look at it now. However, I do encourage readers to look into it if you are interested. ;)

Page Size Extension (PSE)

PSE is a feature in x86 microprocessors that allows pages more then 4KB in size. This allows the x86 architecture to support 4MB page sizes (Also called "huge pages" or "large pages") along side 4KB pages.

The World of Paging

Let the madness begin :)

Introduction

Woo-hoo! Welcome to the wonderful and twisted-minded world of paging! With all of the fundemental concepts that we have went over already, you should have a nice and good grasp at what paging and virtual memory is all about. This is a great start, don't you think?

Okay, cool...but, how do we actually impliment it? How does paging work on the x86 architecture? Lets take a look!

Pages

A **Page** (Also known as a **memory page** or **virtual page**) is a fixed-length block of memory. This block of memory can reside in physical memory. Think of it like this: A page describes a memory block, and where it is located at. This allows us to "map" or "find" the location of where that memory block is at. We will look at mapping pages and how to impliment paging a little later :)

The i86 architecture uses a specific format for just this. It allows us to keep track of a single page, and where it is currently located at. Lets take a look..

Page Table Entries (PTE)

A page table entry is what represents a page. We will not cover the page table until a little later so dont worry too much about it. However we will need to look at what an entry in the table looks like now. The x86 architecture defines a specific bit format for working with pages, so lets take a look at it.

- **Bit 0 (P):** Present flag
  - 0: Page is not in memory
  - 1: Page is present (in memory)
- **Bit 1 (R/W):** Read/Write flag
  - 0: Page is read only
  - 1: Page is writable
- **Bit 2 (U/S):**User mode/Supervisor mode flag
  - 0: Page is kernel (supervisor) mode
  - 1: Page is user mode. Cannot read or write supervisor pages
- **Bits 3-4 (RSVD):** Reserved by Intel
- **Bit 5 (A):** Access flag. Set by processor
  - 0: Page has not been accessed
  - 1: Page has been accessed
- **Bit 6 (D):** Dirty flag. Set by processor
  - 0: Page has not been written to
  - 1: Page has been written to
- **Bits 7-8 (RSVD):** Reserved
- **Bits 9-11 (AVAIL):** Available for use
- **Bits 12-31 (FRAME):** Frame address

Cooldos! Thats all? Well.. I never said it was hard ;)

Quite possibly the most important thing here is the **frame address**. **The frame address represents the 4KB physical memory location that the page manages.** This is *vital* to know when understanding paging, however it is hard to describe why it is so right now. For now, just remember that **each and every page manages a block of memory. If the page is present, it manages a 4KB physical address space in physical memory.**

The **Dirty Flag and Access Flag are set by the processor, not software.** You might wonder on how the processor knows what bits to set; ie, where they are located in memory. We will look at that a little later. Just rememeber that, this will allow the software or executive to test if a page has been accessed or not.

The **present flag** is an important one. This one single bit is used to determin if a page is currently in physical memory or not. If it is currently in physical memory, the frame address is the 32 bit linear address for where it is located at. If it is not in physical memory, the page must reside on another location--such as a hard disk.

If the present flag is not set, the processor will ignore the rest of the bits in the structure. This allows us to use the rest of the bits for whatever purpose...perhaps where the page is located at on disk? This will allow--when our page fault handler gets called--for us to locate the page on disk and swap the page into memory when needed.

Lets give out a simple example. Lets say that we want this page to manage the 4KB address space beginning at physical location 1MB (0x100000). What this means--to put in other words--**is that this page is *"mapped"* to address 1MB**.

To create this page, simply set 0x100000 in bits 12-31 (the frame address) of the page, and set the present bit. Voila--the page is mapped to 1MB. :) For example:

```
%define         PRIV            3

mov             ebx, 0x100000 | PRIV    ; this page is mapped to 1MB
```

Notice that 0x100000 is 4KB aligned? It ORs it with 3 (11 binary which sets the first two bits. Looking at the above table, we can see that it sets the present and read/write flags, making this page present (Meaning its in physical memory. This is true as it is mapped from physical address 0x100000), and is writable.

Thats it! You will see this example expand further in the next few sections so that you can start seeing how everything fits in, so don't worry to much if you still do not understand.

Also notice that there is nothing special about PTEs--they are simply 32 bit data. What is special about them is how they are *used*. We will look at that a little later...

pte.h and pte.cpp - Abstracting page table entries and pages

The demo hides all of the code to set and get the individual properties of the page table entries inside of these two files. All these do is set and get the bits and frame address from the 32 bit pattern that we have looked at in the list above. This interface does have a little overhead but greatly improves readability and makes it easier to work with them.

The first thing we do is to abstract the bit pattern used by page table entries. This is too easy:

```
enum PAGE_PTE_FLAGS {

        I86_PTE_PRESENT                 =       1,              //00000000000000000000000000000001
        I86_PTE_WRITABLE                =       2,              //00000000000000000000000000000010
        I86_PTE_USER                    =       4,              //00000000000000000000000000000100
        I86_PTE_WRITETHOUGH             =       8,              //00000000000000000000000000001000
        I86_PTE_NOT_CACHEABLE           =       0x10,           //00000000000000000000000000010000
        I86_PTE_ACCESSED                =       0x20,           //00000000000000000000000000100000
        I86_PTE_DIRTY                   =       0x40,           //00000000000000000000000001000000
        I86_PTE_PAT                     =       0x80,           //00000000000000000000000010000000
        I86_PTE_CPU_GLOBAL              =       0x100,          //00000000000000000000000100000000
        I86_PTE_LV4_GLOBAL              =       0x200,          //00000000000000000000001000000000
        I86_PTE_FRAME                   =       0x7FFFF000      //11111111111111111111000000000000
};
```

Notice how this matches up with the bit format that we looked at in the above list. What we want is a way to abstract the setting and getting of these properties (ie, bits) behind the interface.

To do this, we first abstract the data type used to store a page table entry. In our case its a simple uint32_t:

```
//! page table entry
typedef uint32_t pt_entry;
```

Simple enough. Next up is the interface routines that are used to set and get these bits. I dont want to look at the implimentation of it as all it does is (litterally) set or get individual bits within a pt_entry. So instead I want to focus on the interface:

```
extern void             pt_entry_add_attrib (pt_entry* e, uint32_t attrib);
extern void             pt_entry_del_attrib (pt_entry* e, uint32_t attrib);
extern void             pt_entry_set_frame (pt_entry*, physical_addr);
extern bool             pt_entry_is_present (pt_entry e);
extern bool             pt_entry_is_writable (pt_entry e);
extern physical_addr    pt_entry_pfn (pt_entry e);
```

pt_entry_add_attrib() sets a single bit within the pt_entry. We pass it a mask (like our I86_PTE_PRESENT bit mask) to set it. **pt_entry_del_attrib()** does the same but clears the bit.

pt_entry_set_frame() masks out the frame address (I86_PTE_FRAME mask) to set our frame address to it. **pt_entry_pfn()** returns this address.

There is nothing special about these routines--we can easily set and get these attributes manually if we wanted to via bit masks or (if you wanted) bit fields. I personally feel this setup makes it much easier to work with though ;)

Okay, this is great as this setup allows us to keep track of a single page. However, it is useless by itself as a typical system will need to have alot of pages. This is where a page table comes in.

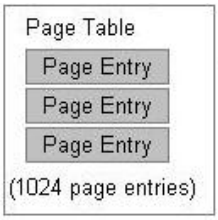
Page Tables

The page table...hm...where oh where did we hear that term before? *looks one line up*. Oh, right ;)

A **Page Table** is..well..a table of pages. (Surprised?) A page table allows us to keep track of how the pages are mapped between physical and virtual addresses. **Each page entry in this table follows the format shown in the previous section.** In other words, **a page table is an array of page table entries (PTEs)**.

While it is a very simple structure, it has a very important purpose. The page table containes a list of all the pages it containes, and how they are mapped. By "mapping", We refer to how the virtual address "maps" to the physical frame address. The page table also manages the pages, weather they are present, how they are stored, or even what process they belong to (This can be set by using the AVAIL bits of a page. This may not be needed, it depends on the implimentation of the system.)

Lets stop for a moment. **Remember that a page manages 4KB of physical address space?** By itself, a page is nothing more then a 32 bit data structure that describes the properties of a specific 4KB region of physical memory (Remember this from before?) Because each page "manages" 4KB of physical memory, putting 1024 pages together we have 1024*4KB=4MB of managed virtual memory. Lets take a look at how its set up:




```
struct page_table {

   page_entry m_entries[1024];
};
```

Thats an example of a page table. Notice how it is nothing more then an array 1024 page entries. Knowing that each page manages 4KB of physical memory, we can actually turn this little table into its own **virtual address space**. How can we do this? Simple: By deciding the format of a **virtual address**.

Heres an example: Lets say we have designed a new virtual address format like this:

```
AAAAAAAAAA        BBBBBBBBBBBB
page table index  offset into page
```

This is our format for a virtual address. So, when paging is enabled, all memory addresses will now follow the above format. For example, lets say we have the following instruction:

```
mov     ecx, [0xc0000]
```

Here, **0xc0000** will be treated like a **virtual address**. Lets break it apart:

```
 11000000        000000000000  ; 0xc0000 in binary form
AAAAAAAAAA        BBBBBBBBBBBB
page table index  offset into page
```

What we are now doing is an example of **address translating**. We are actually translating this virtual address to see what physical location it refers to. The page table index, 11000000b = 192. This is the page entry inside of our page table. We can now get the base physical address of the 4KB that this page manages. If this page is present (Pages **present** flag is set), all we need to do is access the pages **frame address** to access the memory. If this page is NOT present, then generate a page fault--The page data might be somewhere on disk. The page fault handler will allow us to copy the 4KB data for the page into memory somewhere and set the page to **present** and update its **frame address** to point to this new 4KB block of physical memory.

Okay okay, I know. This little example of creating a fake "virtual address" might seem silly, but guess what? ***This is how its actually done!*** The actual format of a virtual address is a *little* bit more complex in that there are *three* sections instead of 2. However, if we omit the first section of the real virtual address format then it would be *exactally* the same as our above example.

I hope by now you are starting to see how everything fits together, and the importance of page tables.

Page Size

A system with smaller page sizes will require more pages then a system with larger page sizes. Because the table keeps track of all pages, a system with smaller page sizes will also require a larger page table because there are more pages to keep track of. Simple enough, huh?

The i86 architecture supports 4MB (2MB pages if using **Page Address Extension (PAE)**) and 4KB sized pages.

The important things to note are: Notice how page size may effect the size of page tables.

The Page Directory Table (PDT)

Okay... We are almost done! A page table is a very powerful structure as you have seen. Remember our previous virtual address example? I gave an example of a virtual addressing system where each virtual address was composed of two parts: A page table entry and a offset into that page.

On the x86 architecture, the virtual address format actually uses three sections instead of two: The entry number in a **page directory table**, the page table index, and the offset into that page.

A **Page Directory Table** is nothing more then an array of **Page Directory Entries**. I know I know... How useless and non-informative was that last sentence? ;)

So, anyways, lets first look at a page directory entry. Then we will start looking at the directory table, and where it all fits in...

Page Directory Entries (PDEs)

Page directory entries help provide a way to manage a single page table. Not only do they contain the address of a page table, but they provide properties that we can use to manage them. You will see how all of this fits in within the next section, so dont worry if you dont understand it yet.

Page directory tables are very simularly structured in the way page tables are structured. They are an array of 1024 entries, where the entries follow a specific bit format. The nice thing about the format of page directory entries (PDEs) is that they follow almost the exact same format that page table entries (PTEs) do (in fact they can be interchangeable). There is only a few little bit of details (pun intended ;)).

Here is the format of a page directory entry:

- **Bit 0 (P):** Present flag
  - 0: Page is not in memory
  - 1: Page is present (in memory)
- **Bit 1 (R/W):** Read/Write flag
  - 0: Page is read only
  - 1: Page is writable
- **Bit 2 (U/S):**User mode/Supervisor mode flag
  - 0: Page is kernel (supervisor) mode
  - 1: Page is user mode. Cannot read or write supervisor pages
- **Bit 3 (PWT):**Write-through flag
  - 0: Write back caching is enabled
  - 1: Write through caching is enabled
- **Bit 4 (PCD):**Cache disabled
  - 0: Page table will not be cached
  - 1: Page table will be cached
- **Bit 5 (A):** Access flag. Set by processor
  - 0: Page has not been accessed
  - 1: Page has been accessed
- **Bit 6 (D):** Reserved by Intel
- **Bit 7 (PS):** Page Size
  - 0: 4 KB pages
  - 1: 4 MB pages
- **Bit 8 (G):** Global Page (Ignored)
- **Bits 9-11 (AVAIL):** Available for use
- **Bits 12-31 (FRAME):** Page Table Base address

Alot of the members here should look familiar from the page table entry (PTE) list that we looked at ealier.

The **Present**, **Read/Write**, and **access** flags are the same as it was with PTEs, however they apply to a **page table** rather then a **page**.

page size determins if the pages inside of the page table are **4KB** or **4MB**.

Page Table Base address bits contain the 4K aligned address of a **page table**.

pde.h and pde.cpp - Abstracting Page Directory Entries

Simular to what we did with PTEs, we have created an interface to abstract PDEs in the same manner.

```
enum PAGE_PDE_FLAGS {

	I86_PDE_PRESENT			=	1,		//00000000000000000000000000000001
	I86_PDE_WRITABLE		=	2,		//00000000000000000000000000000010
	I86_PDE_USER			=	4,		//00000000000000000000000000000100
	I86_PDE_PWT			=	8,		//00000000000000000000000000001000
	I86_PDE_PCD			=	0x10,		//00000000000000000000000000010000
	I86_PDE_ACCESSED		=	0x20,		//00000000000000000000000000100000
	I86_PDE_DIRTY			=	0x40,		//00000000000000000000000001000000
	I86_PDE_4MB			=	0x80,		//00000000000000000000000010000000
	I86_PDE_CPU_GLOBAL		=	0x100,		//00000000000000000000000100000000
	I86_PDE_LV4_GLOBAL		=	0x200,		//00000000000000000000001000000000
	I86_PDE_FRAME			=	0x7FFFF000	//11111111111111111110000000000000
};

//! a page directery entry
typedef uint32_t pd_entry;
```

Not to hard. We use the new type pd_entry to represent a page directory entry. Also, with the PTE interface, we provide a small set of routines used to provide a nice way of setting and getting the bits within the page directory entry:

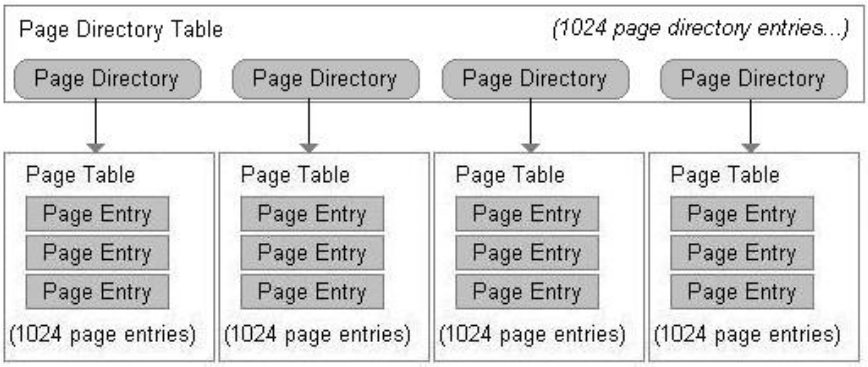
```
extern void		pd_entry_add_attrib (pd_entry* e, uint32_t attrib);
extern void		pd_entry_del_attrib (pd_entry* e, uint32_t attrib);
extern void		pd_entry_set_frame (pd_entry*, physical_addr);
extern bool		pd_entry_is_present (pd_entry e);
extern bool		pd_entry_is_user (pd_entry);
extern bool		pd_entry_is_4mb (pd_entry);
extern bool		pd_entry_is_writable (pd_entry e);
extern physical_addr	pd_entry_pfn (pd_entry e);
extern void		pd_entry_enable_global (pd_entry e);
```

Understanding the Page Directory Table

The Page Directory Table is sort of like an array of 1024 page tables. Remember that each page table manages 4MB of a virtual address space? Well... Putting 1024 page tables together we can manage a full 4GB of virtual addresses. Sweet, huh?

Okay, its a little more complex then that, but not that much. The **Page Directory Table** is actually an array of **1024 page directory entries** that follow the format above. Look back at the format of an entry and notice the **Page Table Base address** bits. This is the address of the page table this directory entry manages.

It may be easier to see it visually, so here you go:



Notice what is happening here. Each page directory entry points to a page table. Remember that each page manages 4KB of physical (and hence virtual) memory? Also, remember that a page table is nothing more then an array of 1024 pages? 1024*4KB = 4MB. This means that each page table manages its own 4MB of address space.

Each page directory entry provides us a way to manage each page table much easier. Because the complete page directory table is an array of 1024 directory entries, and that each entry manages its own table, we effectivly have 1024 page tables. From our previous calculation we know each page table manages 4MB of address space. So 1024 page tables*4MB size= 4GB of virtual address space.

I guess thats it for ... believe it or not... everything. See, its not that hard, is it? In the next section, we will be revisiting the *real* format of an x86 virtual address, and you will get to see how everything works together!

Use in Multitasking

We run into a small problem here. Remember that a page directory table represents a 4GB address space? How can we allow multiple programs a 4GB address space if we can only have one page directory at a time?

We cant. Not nativly, anyways. Alot of mutitasking operating systems map the high 2 GB address space for its own use as "kernel space" and the low 2 GB as "user space". The user space cannot touch kernel space. With the kernel address space being mapped to every processes 4GB virtual address space, we can simply switch the current page directory without error using the kernel no matter what process is currently running. This is possible do to the kernel always being located at the same place in the processes address space. This also makes scheduling possible. More on that later though...

Virtual Memory Management

We have covered everything we need to develop a good virtual memory manager. A virtual memory manager must provide methods to allocate and manage pages, page tables, and page directory tables. We have looked at each of these in separate, but have not looked at how they work together.

Higher Half Kernels

Abstract

A **Higher Half Kernel** is a kernel that has a virtual base address of 2GB or above. A lot of operating systems have a higher half kernel. Some examples include the Windows and Linux Kernels. The Windows Kernel gets mapped to either 2GB or 3GB virtual address (depending on if /3gb kernel switch is used), the Linux Kernel gets mapped to 3GB virtual address. The series uses a higher half kernel mapped to 3GB. Higher half kernels must be mapped properly into the virtual address space. There are several methods to achieve this, some of which is listed here.

You might be interested on why we would want a higher half kernel. We can very well run our kernel at some lower virtual address. One reason has to do with v86 tasks. If you want to support v86 tasks, v86 tasks can only run in user mode and within the real mode address limits (0xffff:0xffff), or about 1MB+64k linear address. It is also typical to run user mode programs in the first 2GB (or 3GB on some OSs) as software typically never has a need to access high memory locations.

Method 1

The first design is that we can have the boot loader set up a temporary page directory. With this, the base address of the kernel can be 3GB. The boot loader maps a physical address (typically 1MB) to this base address and calls the kernel's entry point.

This method works, but creates a problem of how the kernel is going to work with managing virtual memory. The kernel can either try to work with the page directory and tables set up by the boot loader, or create a new page directory to manage. If

we create a new page directory, the kernel will need to remap itself (1MB physical to the base virtual address of the kernel) or cloning the existing temporary page directory to the new page directory.

At this time, this is the method the series uses. The series boot loader will set up a temporary page directory and maps the kernel to 3GB virtual. The kernel then creates a new page directory during VMM initialization and remaps itself. The kernel must remain position-independent during this set up phase. This is the method we use in our in-house OS.

Method 2

Another possible design is that the boot loader loads the kernel into a physical memory location and keeps paging disabled. The kernel virtual base address would be the virtual address it is supposed to execute at. For example, the boot loader can load and execute the kernel at 1MB physical, although the kernels base address is 3GB.

This method is a little tricky. There has to be a way for the boot loader to know what physical address to load and execute the kernel at, and the kernel has to map itself to its real base virtual address. This is usually done during kernel startup in position-independent code. This can be used in position-dependent code, but the kernel must be able to fix the addresses when accessing data or calling functions. This is the method used in our in-house OS.

Method 3

This method uses Tim Robinson's GDT trick. This can be found in his documentation located here (*.pdf) This allows your kernel to run at a higher address (its base address) even though it is not loaded there. This trick works do to address wrap around. For example, lets say our kernel is loaded at 1MB physical address, but we want it to appear to be running at 3GB Virtual. The base that we want is X + 3GB = 1MB in this case. Lets look closer.

Remember that the GDT descriptor base address is a DWORD. If the value becomes greater then 0xffffffff, it will wrap around back to 0. 3GB = 0xC0000000. 0xffffffff - 0xc0000000 = 0x3FFFFFFF bytes left until it wraps. We need to add an address that will make this address to point to our physical location (1MB). Knowing we have 0x3FFFFFFF bytes left until our DWORD wraps back to 0, we can add 0x100000 (1MB) + 0x3FFFFFFF = 0x400FFFFF + 1 = 0x40100000.

So, by using the above example, if our kernel is loaded at 1MB physical address but has a real base address of 3GB virtual, we can create a temporary GDT with a base code and data selector of 0x40100000. The processor automatically adds the base selector addresses to the addresses it is accessing. After using LGDT to install this new GDT. After this we are now running at 3GB. This works because the processor will add the cs and ds selector base (40100000) to whatever address that is being referenced. For example, 3GB would be translated by the processor to 1MB in our example as 3GB+base selector ((40100000) = 1MB physical.

This trick is fairly easy to impliment and works well but wont work for 64 bit (Long Mode). After the kernel performs this trick it can set up its page directory and map itself with ease after which can enable paging.

Virtual Addressing and Mapping Addresses

When we enable paging, **all memory refrences** will be treated as a **virtual address**. This is very important to know. This means we must set up the structures properly first before enabling paging. If we do not, we can run into an immiedate triple fault--with or without valid exception handlers.

Remember the format of a virtual address? **This is the format of a x86 virtual address**:

```
AAAAAAAAAA          BBBBBBBBBB          CCCCCCCCCCCC
directory index     page table index    offset into page
```

This is very important! This tells the processor (And *us*) alot of information.

The **directory index** portion tells us what index into the current **page directory** to look in. Look back up to the Directory Entry Structure format in the previous section. **Notice that each directory table entry containes a pointer to a page table**. You can also see this within the image again in that section.

Because each index within the directory table points to a page table, this tells us what page table we are accessing.

The **page table index** portion tells us what **page entry** within this page table we are accessing.

...And remember that each page entry manages a full 4KB of physical address space? The **offset into page** portion tells us what byte within this pages physical address space we are refrencing.

Notice what happened here. We have just translated a virtual address into a physical address using our page tables. Yes, its that easy. No trickery involved.

Lets look at another example. Lets assumed that virtual address 0xC0000000 was mapped to physical address 0x100000. How do we do this? We need to find the page in our structures that 0xC0000000 refer to -- just like we did above. In this case 0xC0000000 is the virtual address, so lets look at its format:

```
1100000000          0000000000          000000000000                ; 0xC0000000 in binary form

AAAAAAAAAA          BBBBBBBBBB          CCCCCCCCCCCC
directory index     page table index    offset into page
```

Remember that the directory index tells us what page table we are accessing within the page directory table? So... 1100000000b (The directory index) = 768th page table.

Remember that the page table index is the page we are accessing within this page table? That is 0, so its the first page. Also note the offset byte in this page is 0.

Now, all we need to do is set the **frame address** of the first page in the 768th page table to 0x100000 and voila! You have just mapped 3GB virtual address to 1MB physical! Knowing that each page is 4KB aligned, we can keep doing this in increments of 4KB physical addresses.

Identity Mapping

Identity Mapping is nothing more then mapping a virtual address to the same physical address. For example, virtual address 0x100000 is mapped to physical address 0x100000. Yep--Thats all there is to it. The only real time this is required is when first setting up paging. It helps insure the memory addresses of your current running code of where they are at stays the same when paging is enabled. Not doing this will result in immediate triple fault. You will see an example of this in our Virtual Memory Manager initialization routine.

Memory Managment: Implimentation

Implimentation

I suppose that is everything. What we will look at next is the virtual memory manager (VMM) itself that has been developed for this tutorial. This will bring everything that we have looked at together so that you can see how everything works.

I have tried to make the routines small so that we can focus on one topic at a time as there is a couple of new things that we still need to look at.

Alrighty...First lets take a look at the page table and directory table themselves:

```
//! virtual address
typedef uint32_t virtual_addr;

//! i86 architecture defines 1024 entries per table--do not change
#define PAGES_PER_TABLE 1024
#define PAGES_PER_DIR   1024

#define PAGE_DIRECTORY_INDEX(x) (((x) >> 22) & 0x3ff)
#define PAGE_TABLE_INDEX(x) (((x) >> 12) & 0x3ff)
#define PAGE_GET_PHYSICAL_ADDRESS(x) (*x & ~0xfff)

//! page table represents 4mb address space
#define PTABLE_ADDR_SPACE_SIZE 0x400000

//! directory table represents 4gb address space
#define DTABLE_ADDR_SPACE_SIZE 0x100000000

//! page sizes are 4k
#define PAGE_SIZE 4096

//! page table
struct ptable {

        pt_entry m_entries[PAGES_PER_TABLE];
};

//! page directory
struct pdirectory {

        pd_entry m_entries[PAGES_PER_DIR];
};
```

Simular to our **physical_addr** type, I created a new address type for virtual memory--**virtual_addr**. Notice that a page table is nothing more then an array of 1024 page table entries? Same thing with the page directory table, but its an array of page directory entries instead. Nothing special yet ;)

PAGE_DIRECTORY_INDEX, PAGE_TABLE_INDEX, PAGE_GET_PHYSICAL_ADDRESS are macros that just returns the respective partion of a virtual address. Remember that a virtual address has a specific format, these macros allow us to obtain the information from the virtual address.

PTABLE_ADDR_SPACE_SIZE represents the size (in bytes) that a page table represents. A page table is 1024 pages, where a page is 4K in size, so it is 1024 * 4k = 4MB. **DTABLE_ADDR_SPACE_SIZE** represents the number of bytes a page directory manages, which is the size of the virtual address space. Knowing a page table represents 4MB of the address space, and that a page directory contains 1024 page tables, 4MB * 1024 = 4GB.

The virtual memory manager presented here does not handle **large pages**. Instead, it only manages 4K pages.

The Virtual Memory Manager (VMM) we use relies on these structures heavily. Lets take a look at some of the routines in the VMM to learn how they work.

vmmngr_alloc_page () - allocates a page in physical memory

To allocate a page, all we need to do is allocate a 4K block of physical memory for the page to refer to, then simply create a page table entry from it:

```
bool vmmngr_alloc_page (pt_entry* e) {

	//! allocate a free physical frame
	void* p = pmmngr_alloc_block ();
	if (!p)
		return false;

	//! map it to the page
	pt_entry_set_frame (e, (physical_addr)p);
	pt_entry_add_attrib (e, I86_PTE_PRESENT);

	return true;
}
```

Notice how our PTE routines make this much easier to do? The above sets the PRESENT bit in the page table entry and sets its FRAME address to point to our allocated block of memory. Thus the page is present and points to a valid block of physical memory and is ready for use. Cool, huh?

Also, notice how we "map" the physical address to the page. All this means is that we set the page to point to a physical address. Thus the page is "mapped" to that address.

vmmngr_free_page () - frees a page in physical memory

To free a page is even easier. Simply free the block of memory using our physical memory manager, and clear the page table entries PRESENT bit (marking it NOT PRESENT) :

```
void vmmngr_free_page (pt_entry* e) {

	void* p = (void*)pt_entry_pfn (*e);
	if (p)
		pmmngr_free_block (p);

	pt_entry_del_attrib (e, I86_PTE_PRESENT);
}
```

Thats it! Now that we have a way to allocate and free a single page, lets see if we can put them together in full page tables...

vmmngr_ptable_lookup_entry () - get page table entry from page table by address

Now that we have a way of abtaining the page table entry number from a virtual address, we need a way to get it from the page table. This routine does just that! It uses the above function to convert the virtual address into an index into the page table array, and returns the page table entry from it.

```
inline pt_entry* vmmngr_ptable_lookup_entry (ptable* p,virtual_addr addr) {

	if (p)
		return &p->m_entries[ PAGE_TABLE_INDEX (addr) ];
	return 0;
}
```

Because this routine returns a pointer, we can modify the entry as much as we need to as well. Cool?

Thats it for the page table routines. See how easy paging is? ;)

Next up...The page directory routines!

vmmngr_pdirectory_lookup_entry () - get directory entry from directory table by address

Now that we have a way to covert a virtual address into a page directory table index, we need to provide a way to get the page directory entry from it. This is exactally the same with the page table routine counterpart:

```
inline pd_entry* vmmngr_pdirectory_lookup_entry (pdirectory* p, virtual_addr addr) {

	if (p)
		return &p->m_entries[ PAGE_TABLE_INDEX (addr) ];
	return 0;
}
```

vmmngr_switch_pdirectory () - switch to a new page directory

Notice how small all of these routines are. They provide a minimal but very effective interface for easily working with page tables and directories. When we set up a page directory, we need to provide a way to install it for our use.

In the previous tutorial, we added two routines: **pmmngr_load_PDBR()** and **pmmngr_get_PDBR()** to set and get the **Page Directory Base Register (PDBR)**. This is the register that stores the current page directory table. On the x86 architecture, the PDBR is the **cr3** processor register. Thus, these routines simply set and gets the cr3 register.

vmmngr_switch_pdirectory () uses these routines to load the PDBR and set the current directory:

```
//! current directory table (global)
pdirectory*             _cur_directory=0;

inline bool vmmngr_switch_pdirectory (pdirectory* dir) {

        if (!dir)
                return false;

        _cur_directory = dir;
        pmmngr_load_PDBR (_cur_pdbr);
        return true;
}

pdirectory* vmmngr_get_directory () {

        return _cur_directory;
}
```

vmmngr_flush_tlb_entry () - flushes a TLB entry

Remember how the TLB caches the current page table? Sometimes it may be necessary to flush (invalidate) the TLB or individual entries so that it can get updated to the current value. This may be done automatically by the processor (Like during a mov instruction involving a control register).

The processor provides a method for us to manually flush individual TLB entries ourself. This is done using the **INVLPG** instruction.

We simply pass it the virtual address and the resulting page entry will be invalidated:

```
void vmmngr_flush_tlb_entry (virtual_addr addr) {

#ifdef _MSC_VER
        _asm {
                cli
                invlpg  addr
                sti
        }
#endif
}
```

Keep in mind that **INVLPG** is a **privlidged instruction**. Thus you must be running in **supervisor mode** to use it.

vmmngr_map_page () - maps pages

This is one of the most important routines. This routine allows us to map any physical address to a virtual address. Its a little complicated so lets break it down:

```
void vmmngr_map_page (void* phys, void* virt) {

   //! get page directory
   pdirectory* pageDirectory = vmmngr_get_directory ();

   //! get page table
   pd_entry* e = &pageDirectory->m_entries [PAGE_DIRECTORY_INDEX ((uint32_t) virt) ];
   if ( (*e & I86_PTE_PRESENT) != I86_PTE_PRESENT) {
```

We are given a physical and virtual address as paramaters. The first thing that must be done is to verify that the page directory entry that this virtual address is located in is valid (That is, has been allocated before and its PRESENT bit is set.)

The page directory index is part of the virtual address itself, so we use PAGE_DIRECTORY_INDEX() to obtain the page directory index. Then we just index into the page directory array to obtain a pointer to the page directory entry. Then the test to see if I86_PTE_PRESENT bit is set or not. If it is not set, then the page directory entry does not exist so we must create it...

```
      //! page table not present, allocate it
      ptable* table = (ptable*) pmmngr_alloc_block ();
      if (!table)
         return;

      //! clear page table
      memset (table, 0, sizeof(ptable));

      //! create a new entry
      pd_entry* entry =
         &pageDirectory->m_entries [PAGE_DIRECTORY_INDEX ( (uint32_t) virt) ];

      //! map in the table (Can also just do *entry |= 3) to enable these bits
      pd_entry_add_attrib (entry, I86_PDE_PRESENT);
      pd_entry_add_attrib (entry, I86_PDE_WRITABLE);
      pd_entry_set_frame (entry, (physical_addr)table);
   }
```

The first thing the above does is to allocate a new page for the new page table and clears it. After words, it uses PAGE_DIRECTORY_INDEX() again to get the directory index from the virtual address, and indexes into the page directory to get a pointer to the page table entry. Then it sets the page table entry to point to our new allocate page table, and sets its PRESENT and WRITABLE bits so that it can be used.

At this point, the page table is guaranteed to be valid at that virtual address. So the routine now just needs to map the address...

```
   //! get table
   ptable* table = (ptable*) PAGE_GET_PHYSICAL_ADDRESS ( e );

   //! get page
   pt_entry* page = &table->m_entries [ PAGE_TABLE_INDEX ( (uint32_t) virt) ];

   //! map it in (Can also do (*page |= 3 to enable..)
   pt_entry_set_frame ( page, (physical_addr) phys);
   pt_entry_add_attrib ( page, I86_PTE_PRESENT);
}
```

The above calls PAGE_GET_PHYSICAL_ADDRESS() to get the physical frame that the page directory entry points to in order to get the page table entry. Then, using **PAGE_TABLE_INDEX** to get the page table index from the virtual address, indexing into the page table it obtains the page table entry. Then it sets the page to point to the physical address and sets the pages PRESENT bit.

vmmngr_initialize () - initialize the VMM

This is an important routine. This uses all of the above routines (Well, most of them ;)) to set up the default page directory, install it, and enable paging. We can also use this an example of how everything works and fits together. Because this routine creates a new page directory, we also need to map 1MB physical to 3GB virtual in order for the kernel.

This is a fairly big routine so lets break it down and see whats going on:

```
void vmmngr_initialize () {

	//! allocate default page table
	ptable* table = (ptable*) pmmngr_alloc_block ();
	if (!table)
		return;

	//! allocates 3gb page table
	ptable* table2 = (ptable*) pmmngr_alloc_block ();
	if (!table2)
		return;

	//! clear page table
	vmmngr_ptable_clear (table);
```

Remember how page tables must be located at 4K aligned addresses? Thanks to out physical memory manager (PMM), our **pmmngr_alloc_block()** already does just this so we do not need to worry about it. Because a single block allocated is already 4K in size, the page table has enough storage space for its entries as well (1024 page table entries * 4 bytes per entry (size of page table entry) = 4K) so all we need is a single block.

Afterwords we clear out the page table to clean it up for our use.

```
	//! 1st 4mb are idenitity mapped
	for (int i=0, frame=0x0, virt=0x00000000; i<1024; i++, frame+=4096, virt+=4096) {

		//! create a new page
```

```
		pt_entry page=0;
		pt_entry_add_attrib (&page, I86_PTE_PRESENT);
		pt_entry_set_frame (&page, frame);

		//! ...and add it to the page table
		table2->m_entries [PAGE_TABLE_INDEX (virt) ] = page;
	}
```

This parts a little tricky. Remember that **as soon as paging is enabled, all address become virtual?** This poses a problem. To fix this, we must map the virtual addresses to the same physical addresses so they refer to the same thing. **This is idenitity mapping**.

The above code idenitity maps the page table to the first 4MB of physical memory (the entire page table). It creates a new page and sets its PRESENT bit followed by the frame address we want the page to refer to. Afterwords it converts the current virtual address we are mapping (stored in "frame") to a page table index to set that page table entry.

We increment "frame" for each page in the page table (stored in "i") by 4K (4096) as that is the block of memory each page refrences. (Remember page table index 0 references address 0 - 4093, index 1 refrences address 4096--etc..?)

Here we run into a problem. Because the boot loader maps and loads the kernel directly to 3gb virtual, we also need to remap the area where the kernel is at:

```
	//! map 1mb to 3gb (where we are at)
	for (int i=0, frame=0x100000, virt=0xc0000000; i<1024; i++, frame+=4096, virt+=4096) {

		//! create a new page
		pt_entry page=0;
		pt_entry_add_attrib (&page, I86_PTE_PRESENT);
		pt_entry_set_frame (&page, frame);

		//! ...and add it to the page table
		table->m_entries [PAGE_TABLE_INDEX (virt) ] = page;
	}
```

This code is pretty much the same as the above loop and maps 1MB physical to 3GB virtual. This is what maps the kernel into the address space and allows the kernel to continue running at 3GB virtual address.

```
	//! create default directory table
	pdirectory*	dir = (pdirectory*) pmmngr_alloc_blocks (3);
	if (!dir)
		return;

	//! clear directory table and set it as current
	memset (dir, 0, sizeof (pdirectory));
```

The above creates a new page directory and clears it for our use.

```
	pd_entry* entry = &dir->m_entries [PAGE_DIRECTORY_INDEX (0xc0000000) ];
	pd_entry_add_attrib (entry, I86_PDE_PRESENT);
	pd_entry_add_attrib (entry, I86_PDE_WRITABLE);
	pd_entry_set_frame (entry, (physical_addr)table);

	pd_entry* entry2 = &dir->m_entries [PAGE_DIRECTORY_INDEX (0x00000000) ];
	pd_entry_add_attrib (entry2, I86_PDE_PRESENT);
	pd_entry_add_attrib (entry2, I86_PDE_WRITABLE);
	pd_entry_set_frame (entry2, (physical_addr)table2);
```

Remember that each page table represents a full 4MB virtual address space? Knowing that each page directory entry points to a page table, we can saftley say that each page directory entry represents the same 4MB address space inside of the 4GB virtual address space of the entire directory table. The first entry in the page directory is for the first 4MB, the second is for the next 4MB and so on. Because we are only mapping the first 4MB right now, all we need to do is set the first entry to point to our page table.

In a simular way, we set up a page directory entry for 3GB. This is needed so we can map the kernel in.

Notice that we also set the page directory entries PAGE and PRESENT bit as well. This will tell the processor that the page table is present and writable.

```
	//! store current PDBR
	_cur_pdbr = (physical_addr) &dir->m_entries;

	//! switch to our page directory
	vmmngr_switch_pdirectory (dir);

	//! enable paging
```

```
        pmmngr_paging_enable (true);
}
```

Now that the page directory is set up, we install the page directory and enable paging. If everything worked as expected, your program should not crash. If it does not work, it will probably triple fault.

Page Faults

As you know, as soon as we enable paging all addresses become virtual. All of these virtual addresses rely heavily on the page tables and page directory data structures. This is fine, but there will be alot of times when a virtual address requires the cpu to access a page that is not yet valid. This is when a **page fault exception (#PF)** is raised by the processor. A will only occur when a page is marked **not present**. A **General Protecton Fault (#GPF)** will occur if the page is not properly mapped but marked present and accessable. A **#GPF** will also occur if the page is not accessable.

A page fault is cpu interrupt 14 which also pushes an error code so that we can abtain information. The error code pushed by the processor has the following format:

- **Bit 0:**
  - 0: #PF occured because page was present
  - 1: #PF occured NOT because the page was present
- **Bit 1:**
  - 0: Operation that caused the #PF was a read
  - 1: Operation that caused the #PF was a write
- **Bit 2:**
  - 0: Processor was running in ring 0 (kernel mode)
  - 1: Processor was running in ring 3 (user mode)
- **Bit 3:**
  - 0: #PF did not occure because reserved bits were written over
  - 1: #PF occured becaused reserved bits were written over
- **Bit 4:**
  - 0: #PF did not occure during an instruction fetch
  - 1: #PF occured during an instruction fetch

All other bits are 0.

When a #PF occures, the processor also stores the address that caused the fault in the **CR2** register.

Normally when a #PF occurs, an operating system will need to fetch the page from the faulting address of the currently running program from disk. This requires several different components of an OS (disk driver, file system driver, volume/mount points management) that we do not yet have. Because of this, we will return back to page fault handling a little later when we have a more evolved OS.

Demo

This demo includes all of the source code in this tutorial, and more. This demo includes paging code inside of the bootloader and kernel to include the complete virtual memory manager (VMM) and to map the kernel to the 3GB mark within its own virtual address space.

There is nothing new visually with this demo. Because of this, there is no new pics. However it does demenstrate the concepts described in this chapter in both assembly language source (The bootloaders Paging.asm file) and C source (The VMM that we have developed in this chapter.)

DEMO DOWNLOAD

Conclusion

I am very glad to get this one done! We have covered alot of information and ground in this tutorial: Virtual Memory, Virtual addressing and translation, paging, methods, and more. With this tutorial, we are not out of the paging word yet! However, we can all saftely go to bed tonight knowing that we have a better understanding of it, how it works, and hot to work with it. See? Its not so bad :)

Inside of the next tutorial I am thinking about going back to the fun stuff with developing a keyboard driver. Because we already have a form of output, and we will be able to retrieve input, we may even make a simple command line as well ;)

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!

Operating Systems Development Series

Operating Systems Development - Keyboard

by Mike, 2009

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

Introduction

Welcome!

In this chapter we will cover something a bit more complex: The keyboard. We will learn about the keyboard itself, a little history lesson, keyboard internals, the 8042 and 8048 microcontrollers, and developing a keyboard driver.

This will also be the first device that we will program within this series. Excited? We have already learned how hardware programming works and have experience in it; now it time to put it to the test. *Ready?* This is also the first device that we will be programming that not only requires us to work with one controller but *two*. These controllers communicate with each other and our system. To make things more complex, both controllers have their own set of commands and way to work with them. Because of this, this chapter is fairly detailed in a couple of places.

This chapter also includes the first interactive demo: A basic command line parser. *Excited?*

This is also the first chapter that we look at device drivers in more depth: The importance of hardware abstraction and device drivers.

Heres the list:

- Keyboard - Back in Time and Keyboard Layout
- Inside the keyboard
- Keyboard Protocols
- Keyboard Encoder
- Keyboard Controller
- Scan Code Sets
- Keyboard IRQ

Lets get going!

Keyboard - History

Back in time

A keyboard is an input device that we use as a form of input to a computer. They were originally modeled off of a typical typewriter when they were first introduced. However the creation of the keyboard was not directly modeled from it.

When the typewriter was patented in by Christopher Latham Sholes in 1877 several manufacturers and people further developed the original design. What evolved through a series of inventions was the Telegraph. Around this same time, inside of the 1930s, IBM was using Keypunches (punch card machines combined with typewriters) in their adding machines. Early computer keyboards adapted from both the keypunch and telegraph designs.

The ENIAC (Electronic Numerical Integrator And Computer) was the first general purpose computer. The ENIAC used a punchcard reader as both an input and output device. in 1946.

In 1948, the BINAC (BINary Automatic Computer) used an electromechanically controlled typewriter as both an input and output device.

When does the keyboard evolve from these inventions? The computer keyboard that we all know does not evolve into what it is today until 1964 when MIT (with Fernando Corbató), Bell Laboratories and General Electric joined together to create the Multics (Multiplexed Information and Computing Service) machine. With the Multics, a new interface was at hand: They combined the technology of the cathode ray tube (CRT) used in televisions and electric typewriters to create a Video Display Terminal (VDT). The VDTs allowed a way for the users to be able to