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