Boot Process

Introduction

There are several things that must happen when you turn on a IBM PC. The purpose of this document is to describe the PC boot process and a demonstrate the process though small code samples. It is assumed that the reader is versed in IA-32 assembly language. By the end of this document, you should be well armed to get the i386 in to full protected mode and jump to a kernel. All assembly language in this document is in Intel syntax. The tool required to compile sample code, is Netwide Assembler (NASM)

Overview

When dealing with the IBM PC the one thing that you must be aware of is that it was not designed in one sitting. Rather is was an architecture that has been added to for the last 20+ years. What this means is that there are many tricks that must be performed by you the systems designer to get the system in to the state desired. All of this was done to have backwards compatibility. The good news is once you have got the system set up, its becomes a much friendlier environment for the systems programmer. The i386 an above processors provide a very robust set of instructions that deal with protection, multi-tasking, and virtual memory.

System Startup

When you first turn on the computer the first thing that executes is the BIOS. The acronym BIOS stands for "Basic Input/Output System". This is software that is programmed in to a ROM (Read Only Memory) attached to the mother board. The BIOS provides basic input/output functions as the name implied. The BIOS is what allows a PC to load an operating system from disk. For more information on the functions that BIOS provides, please refer to Ralf Browns interrupt list. The physical memory below 1 megabyte has certain parts that map to special purposes. The following table is the layout.

Start	End	Description
0x000000	0x0003FF	Real-mode interrupt vector table
0x000400	0x0004FF	BIOS data segment
0x000500	0x09FBFF	Free conventional

		memory
0x09FC00	0x09FFFF	Extended BIOS data area (EBDA)
0x0A0000	0x0BFFFF	Video memory
0x0C0000	0x0FFFFF	Video and motherboard BIOS ROMs
0x100000	0x10FFFF	Hi Memory Area (HMA)
0x110000	0x???????	Free extended memory

Once the BIOS has finished everything that it needed to do (i.e. initialize all devices), it will attempt to load code from the bootable devices assigned in the BIOS configuration.

Floppy Disk

If the BIOS is attempting to boot from a floppy the first thing it will do is load the first sector on the disk. A sector is 512 bytes in size on a typical floppy disk. This first sector is called the "Boot Sector". This sector is loaded to the address 0000:0x7C00. At this point the BIOS gives control to the first instruction located at 0000:07C0. The layout of the boot sector is laid out in the diagram that follows.

Boot Sector

Boot Code	510 bytes
Signature	2 bytes (always 0x55AA)

Hard Disk

If the Bios is attempting to boot from a hard drive the first thing it will do is load the first sector of the disk. A sector is 512 bytes in size on a typical hard drive. This first sector is called the "Partition Sector" This sector is loaded to the address 0000:07C0. The difference between a hard disk and a floppy disk is that a hard drive contains a partition table . This table allows a hard drive to spit up in to sections. This allows multiple operating systems to co-exist, as each one can only see what is available to that partition. The layout of the partition sector is layout out in the diagram following.

Partition Sector

Boot Code	446 bytes

Partition Table	64 bytes
Signature	2 bytes (always 0x55AA)

Partition Table

Offset To Start of Sector	Size (Bytes)	Content
0x01BE	16	Partition 4
0x01CE	16	Partition 3
0x01DE	16	Partition 2
0x01EE	16	Partition 1

Partition Table Entry

Offset to Start of Sector	Size (Bytes)	Content
0x00	1	Boot Flag **(1)**
0x01	3	Start of Partition
0x04	1	System Flag **(2)**
0x05	3	End of Partition
0x08	4	Start Sector Relative to Start of Disk **(3)**
0x0C	4	Number of Sectors in Partition **(3)**

- **(1)** 0x80 = Bootable, 0x00 = Non Bootable
- **(2)** 0 = No DOS FAT, 1=DOS With 12-Bit FAT, 4=DOS With 16 Bit FAT, 5=Extended DOS Partition (DOS 3.30 ff), 6=DOS Partion Larger 32MBytes (DOS 4.00 ff), etc
- **(3)** Intel Format Low-High

The Boot loader

The code that was located in the boot sector now has control of the system. Since our destination is protected mode there are several things that need to be done. I have chosen to use the two stage boot loader approach. This is where the first loader merely loads the data files and hands control over to the second boot loader. The reason for this is that 512 (446 on a hard disk) bytes of code is not really enough to do what we need to do.

1st Stage The boot Loader

- (hard drive only, check to ensure the partition table is valid)
- Load kernel from a floppy disk with the FAT12 file system

2nd Stage The OS Loader

- Mask Interrupts
- Shutoff Non-Mask able Interrupts (NMI)
- Load/Create Global Descriptors for Kernel Code, Data, and Stack
- Turn on address line 20 (A20 gate)
- Switch Processor in to Protected mode

Stage 1 The Boot Loader defined

This is the code that is initially loaded in to memory from the BIOS. As a developer of the boot loader there is no real structure that you "must" follow. Most people find it reasonable to have the boot load the kernel and get the system running. This is a good approach when starting out. In the day of DOS, there existed the FAT (**F**ile **A**llocation **T**able) file system which has become the most widely used file system to date. The reason is that its simple to implement. Almost all modern operating systems support FAT, so its a good idea to make your boot disk fat compatible. This allows your disk to be used other then just a boot disk. It is also a good file system to get your OS up an running with. To make your disk readable by another system that supports FAT, you must include in the boot loader the BPB (Bios Parameter Block). Do not be fooled by the name of this structure, as it really has nothing to do with the BIOS. Its structure is defined below:

Description	Size (Bytes)
	3 bytes
OEM Name and Number	8 bytes
Bytes per Sector	2 bytes
Sectors per Allocation Unit (Cluster)	1 bytes
Reserved Sectors (For Boot Record)	2 bytes
Number of FATs	1 bytes

Number of Root Directory Entries	2 bytes
Number of Logical Sectors	2 bytes
Medium Descriptor Byte (**depreciated**)	1 bytes
Sectors Per FAT	2 bytes
Sectors Per Track	2 bytes
Number of Heads	2 bytes
Number of Hidden Sectors	2 bytes

To see how this would be laid out in assembly code, here is a snippet of the beginning of a boot loader.

```
;------------------------------------------------------------------------
-------
        jmp short main ; Jump over declarations and the BPB to the main
tag
        nop
;------------------------------------------------------------------------
-------
; Next, the BPB (BIOS Parameter Block) for FAT compatability
; The following figures are specific to a 1.44M 3 1/2 inch floppy disk
;------------------------------------------------------------------------
-------
OEM_Name                db 'Halos1.0'   ; 8 bytes for OEM Name and
Version
nBytesPerSec            dw 0200h        ; 512 bytes per Sector
nSecPerClust            db 01h          ; Sectors per Cluster
nSecRes                 dw 01h          ; Sectors reserved for Boot
Record
nFATs                   db 02h          ; Number of FATs
nRootEnts               dw 0E0h         ; 0E0h (224) Root Directory
Entries
nSecs                   dw 0B40h        ; Number of Logical Sectors 0B40h
= 2880
                                        ; 00h when > 65,535 sectors
mDesc                   db 0F0h         ; Medium Descriptor Byte
nSecPerFat              dw 09h          ; Sectors per FAT
nSecPerTrack            dw 012h         ; Sectors per Track
nHeads                  dw 02h          ; Number of Heads
nLogSec0
nSecHidden              dd 00h          ; Number of Hidden Sectors
```

The boot loader would then to proceed and load the loader, and possible load the kernel at this time as well. This is up to what is being attempted. In fact you might not even need another loader and you might just want to load the kernel and go. Some systems make the entry point of the kernel the actual loader, and do a lot of initial set up at that

point. If you building a 32-bit operating system I would strongly suggest using a 16 bit second stage loader, that in turn, once the environment is setup calls the kernel's main function. It is to the intention of teaching the FAT file system in this document, but it might be in the future. For now I will consider it an exercise for the reader to get acquainted with the FAT file system and how to write a FAT file loader.

Stage Two The OS Loader

The only reason to have a second stage in the boot process is to allow you the freedom of not having to squeze everything that needs to be done in the first stage. If you do not want to support the FAT file system then you might not need the second loader. The first boot loader that I wrote, fit well with in the 512 bytes, granted it was a dead disk to all systems other then a boot disk for my OS (No file system). Each item that was in the intial list of thing that need to happen on the second stage of the boot process follow:

Mask Interrupts

There are two types of interrupts on the IA-32 platform. The first set is called maskable interrupts. These are interrupts that can easily be turned off, or masked, by the processor. The instructions that handles enabling and disabling are CLI and STI, which stand for clear and set respectively. Maskable interrupts are generated by the PIC (Primary Interrupt Controller) which is usually an Intel 8259 microprocessor. These interrupts are used to single the processor of events such as the keyboard, disk I/O, and timer events. The other type of Interrupt is the non-maskable interrupt These interrupts normally only occur when a hardware failure occurs and the processor must know. There is a way to disable these, but this is done in the CMOS of the system. Code is below to disable both types

```
; ------------------------------------------------------------
;  mask(turn off) maskable interupts
; ------------------------------------------------------------
        cli                     ; we are now done with interrupts
        mov al, 11111111b        ; select to mask of all irq's
        out 0x21, al             ; write it to the PIC controller
; ------------------------------------------------------------
;  turn off non-maskable interupts
; ------------------------------------------------------------
        in al, 0x70              ; read a value
        or al, 10000000b         ; set the nmi(non maskable interupt)
disable bit
        out 0x70, al             ; write it back again
```

Load/Create Global Descriptors for Kernel Code, Data, and Stack

The global descriptor table is how the IA-32 platform handles paging, protection, and multitasking. To operate in protected mode you must have at least one GDT defined for

code and data. They can overlap. The easiest approach to this is to create the code and data segments to be 0-4gigabytes in size. This is known as the flat memory model. With the approach all addresses will be mapped to physical addresses. This is convenient because there is not paging or memory management available at the instant you go in to protected mode. To get started use the following GDT definitions. You will notice the pointer assuming the the GDT will be moved to the location of 0x800. This can be done with the code snippet following the definitions. And last but not least you must load the GDT, which I show in the 3 snippet.

```
        ;
------------------------------------------------------------------------
        ; Pointer to where the GDT will live and the limit
        ;
------------------------------------------------------------------------
GDTptr:
        dw 17FFh                ; limit, 768 slots
        dd 0x800                ; base is at 0x800 physical
        ;
------------------------------------------------------------------------
        ; Global Descriptor Table
        ;
------------------------------------------------------------------------
gdt:
        dw 0,0,0,0              ; dummy descriptor
        ; code descriptor
        dw 0xFFFF               ; segment limit bits 15-00
        dw 0x00                 ; base address bits 15-00
        db 0x00                 ; base address bits 23-16
        db 0x98                 ; Present, DPL=00,App/Sys=1
(1001=0x9),code execute only (0x08)
        db 0xCF                 ; Granularity=1,DB=1,0,0 (1100=0xc),
segment limit bits 19-16 (0xF)
        db 0x00                 ; base address bytes 31-24
        ; data descriptor
        dw 0xFFFF               ; segment limit bits 15-00
        dw 0x00                 ; base address bits 15-00
        db 0x00                 ; base address bits 23-16
        db 0x92                 ; Preset, DPL=00,App/Sys=1 (1001=0x9),
data read/write (0x02)
        db 0xCF                 ; Granularity=1,DB=1,0,0 (1100=0xc),
segment limit bits 19-16 (0xF)
        db 0x00                 ; base address bytes 31-24
        ; stack descriptor
        dw 0xFFFF               ; segment limit bits 15-00
        dw 0x00                 ; base address bits 15-00
        db 0x00                 ; base address bits 23-16
        db 0x92                 ; Preset, DPL=00,App/Sys=1 (1001=0x9),
data read/write (0x02)
        db 0xCF                 ; Granularity=1,DB=1,0,0 (1100=0xc),
segment limit bits 19-16 (0xF)
        db 0x00                 ; base address bytes 31-24


        ; ---------------------------------------------------------------
        ; relocate the global descriptor table to 0x800
```

```
        ; ------------------------------------------------------------
        mov ax,0
        mov es,ax               ;set segment for indexes
        mov di, 0x800           ;to location
        mov si,gdt              ;from location
        mov cx,32               ;32 times
        rep movsb               ;move byte

        ; ------------------------------------------------------------
        ; load the GDT
        ; ------------------------------------------------------------
        lgdt [GDTptr]           ; Load the GDT pointer!
```

Turn on address line 20 (A20 gate)

Once upon a time there were only 20 address lines on the 80x86. Since the processor had a nice feature of memory wrap around programmers did not worry about using addresses larger then what was physically available. When the x86 platform needed more address lines, IBM had the idea of just leaving the A20 line off for compatibilities. IBM found that the keyboard controller had a free port and decided that this port would be used to turn on the A20 line if it was desired. This solved the problem of backwards compatibility. To turn on the A20 line, use the code below.

```
; ------------------------------------------------------------
; Its time to turn on A20
; ------------------------------------------------------------
        xor cx,cx               ; set to 0, so loop will decrement to 255
A20step1:
        in al,64h               ; get data from keyboard controller
        test al,02h             ; test to see if the buffer is full
        loopnz A20step1         ; try 255 times
        mov al,0xd1             ; this code tells the control expect a
byte
        out 64h,al              ; send the code to the controller
        xor cx,cx               ; again reset the counter
A20step2:
        in al,64h               ; get data from keyboard controller
        test al,02h             ; test to see if the buffer is full
        loopnz A20step2         ; try 255 times
        mov al,0dfh             ; here is the code to turn on Gate 20
        out 60h,al              ; send the code to the controller
        xor cx,cx               ; again reset the counter
A20step3:
        in al,64h               ; get data from keyboard controller
        test al,02h             ; test to see if the buffer is full
        jnz A20step3            ; try 255 times
        xor cx,cx               ; just clean up cx, to 0
```

Switch to Protected mode

To switch the processor in to protected mode there is only one bit that must be set. That is bit 0 of control register 1. The GDT register must be populated with a pointer to an valid GDT at this point for this to be successful. If its not the processor will throw an

exception and since there are no exception handlers it will eventually triple fault and go in to shutdown mode.

```
; ------------------------------------------------------------
; switch to protected mode
; ------------------------------------------------------------
        mov eax,cr0             ; get the control register 0
        or al,1                 ; set PE bit
        mov cr0,eax             ; JUMP!!
; ------------------------------------------------------------------------
; the intel doc states that the pipline is not flushed when
; protected mode is enabled.  So we need to manually flush the
; pipeline
; ------------------------------------------------------------------------
        jmp $+1                 ; this clears the prefetch queue of any
        nop                     ; 16-bit instructions
```

At this point you are ready to just to the kernel. You must also ensure all segments are setup with the new descriptors rather then the 16 bit segments that you were using prior to the switch to protected mode.