RedFox32
A self learning exercise in low level x86 programming

RedFox0x20

August 13, 2020

Contents

List of Figures

1 Introduction

1.1 whoami

I am a hobbyist developer with an interest in low level programming and cyber security.
You can find me here:

- Website: https://redfox32.xyz/
- Twitter: RedFox0x20
- Instagram: RedFox0x20
- GitHub: RedFox0x20
- TryHackMe: RedFox0x20
- Twitch: RedFoxCreates

1.2 Why?

The aim of this project is to develop an understanding for how software is developed for use on an x86 (and subsequently x86_64) processor. The project is purely a self learning exercise however every step into producing a functioning piece of software will be documented to allow for this write-up document to provide as much information as possible to future developers who wish to give a project like this an attempt.

1.3 How?

To produce this project a mixture of C and x86 Assembly will be used. The project code will be tested on both virtual and physical hardware using time appropriate hardware. Virtual testing is used to avoid causing damage to physical hardware during the development period. To produce the virtual hardware QEMU will be used, it is worth noting that some aspects will act differently on QEMU compared to physical hardware. Differences between virtual and physical hardware should always be thought of as it may require slight variations in our code.

Test hardware specification:

- Model: CLEVO CO 2200C
- CPU: 800MHz Intel Celeron
- MEM: 256MB RAM
- VRAM: BIOS configurable 8MB-64MB
- BIOS: 1983-1996 SystemSoft BIOS V1.0.03
- HDD: 8GB
- Ports: PS2, Printer, VGA, 2x USB 1.0, Analogue Video, Firewire, 3.5mm Audio In, 3.5mm Audio Out, RJ45 Ethernet
- External Storage: Memory card, Floppy drive, DVD-ROM CD-RW Drive

1.4 Toolchain

To develop RedFox32 I used the following software:

- git - Version control system, allows for tracking of changes to files which can be useful for data recovery and debugging of errors.
- NASM - NASM is the assembler software used to convert our assembly to machine code.
- GCC - The GNU Cross Compiler is a compiler capable of compiling a variety of languages to machine code, we will use this for compiling the C code.
- ld - The GNU Linker is used to link our compiled C binaries together to produce single files for aspects such as the kernel.
- QEMU - Virtualisation software used to test the software we develop
- Make - The make system defines a method for creating "make" jobs that only compiles our source code when it has been changed, making it quicker and easier to build and manage the compilation of software
- dd - A linux utility that allows for the copying and modification of files. dd is used to create our disk image and write our compiled software to the disk in addition to other data.

2 The Bootloader

2.1 What is a Bootloader?

The Bootloader is the first section of our code that will be run. We are limited to 512 bytes of memory in the bootloader... This may not sound like a lot, however, when we are handed control the system exists in "Real mode", a 16 bit operating mode which has access to BIOS functions, these BIOS functions are going to be the only method of communication with the system we have until we enter 32 bit protected mode where we lose access to the BIOS functions. On x86_64 systems you can move from 32 bit protected mode to 64 bit long mode, however, this will not be covered here as it is out of scope for the project in it's current state. The Bootloader sector is identified using the magic bytes 0x55 and 0xAA at position 511 and 512 respectively at the end of the sector.

2.2 What should the Bootloader do?

The bootloader should both initialise the system to our needs in addition to loading our Kernel. There are three types of Bootloader:

- Single Stage Bootloader - A Single Stage Bootloader prepares the system for the kernel then immediately afterwards jumps into the kernel environment. This is not always the best option due to the size constraints, if the bootloader exceeds 512 bytes there is a risk for features to work incorrectly in particular those that are expected to be at specific positions such as the magic bytes and FAT headers.

- Two-Stage-Bootloader - A Two-Stage-Bootloader uses the first 512 bytes to load additional code which is used to handle kernel environment preparation and jumping into the kernel. We will be using this bootloader format for ease of use and to maximise free space for additional code for error handling and identification.

- Mixed Bootloader - A mixed bootloader uses the first sector to load the second sector directly afterwards to produce a larger available space for the bootloader to use without long initial load times until it is time to load the kernel.

2.3 Testing the boot sector and how it can be used

The Boot sector is located at sector 0 of the bootable medium (In our case a floppy disc). In some versions of GCC it is possible to compile to 16 bit (Real mode) code however we are going to use assembly as the most common GCC distributions do not support 16 bit code. The Boot Loader code is loaded into 0x0000:0x7C00 of memory, we need to inform NASM of this fact to ensure that any jumps are performed correctly, this is done using the ORG identifier, additionally we need to inform nasm that this will be 16 bit code. When calling BIOS interrupt function parameters are passed using the available registers.

```
; Boot.asm
[BITS 16]
[ORG 0x7C00]
Entry:
; Let’s use a BIOS function to write to the screen
; If we’re in a colour mode then we will get green text
mov ah, 0x0E
mov al, ’H’
mov bh, 0
mov bl, 0x0A
int 0x10

mov al, ’E’
int 0x10

; The function doesn’t return any data leaving the registers
; This allows us to make repeat calls using the same data
mov al, ’L’
int 0x10
int 0x10

mov al, ’O’
int 0x10

End:    ; An end function to catch the program
hlt     ; Stop CPU execution until an interrupt occurs
jmp End ; Catch after an interrupt and halt again forever
; Fill the remaining space with zeroes until 510
times 510 - ($ - $$) db 0
db 0x55, 0xAA
```

Here we have a basic boot sector that will print the word ”Hello” to the screen using the BIOS interrupt 0x10 command 0x0E.

To compile the code we need to use NASM with the appropriate arguments/flags:

```
nasm -fbin Boot.asm -o Boot.bin
```

The ”-fbin” argument compiles the file in binary mode, this causes the file to be directly converted to machine code without any additional features such as ELF compilation which adds extra data and is unsupported in this use case.

2.4 Implementing the Boot sector (Stage 1)

So we now that we know how to use a boot sector and compile for it we should think about how we are going to use it for it's intended purpose; setting up and loading the kernel. To do this we are going to use a two stage bootloader as described previously, this will ensure to provide us a large memory space to work with in addition to providing ease of use to the code. The bootloader will not be the most memory efficient however ultimately this does not matter as the 16 bit code can no longer be executed once we enter protected mode, additionally we will be defining data regions that will be used by other code sections that may need to be accessed from protected mode.

2.4.1 Structuring our Bootloader

Our bootsector is going to have the following structure:

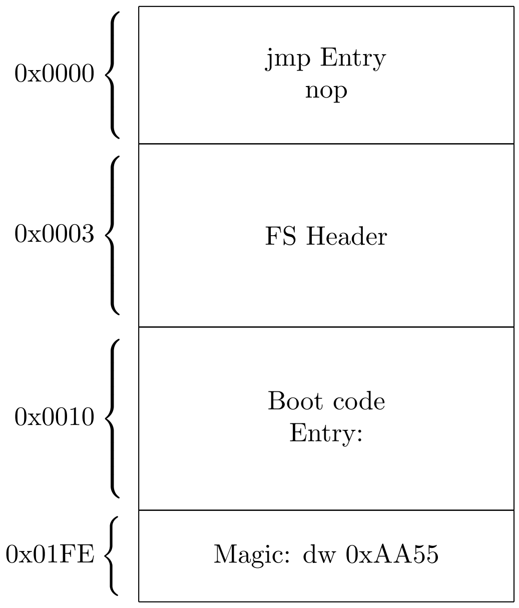

Figure 1: Bootloader structure

By using features of NASM we will be able to ensure that the bootloader does not exceed 512 bytes, additionally available space within the boot sector should not be a concern given we are creating a two stage bootloader.

2.4.2 A look at BIOS functions

BIOS functions as suggested by their name are functions provided by the BIOS. To call these functions we use the CPU Registers to provide parameters and use software interrupts to call them. BIOS functions are broken up into sections and command identifiers, I will use the code from earlier as an example:

```
mov ah, 0x0E
mov al, 'H'
mov bh, 0
mov bl, 0x0A
int 0x10
```

Here the function section is 0x10 (10h), this section is used for graphics commands, we put the function section after the "int" (interrupt) command once we have set the other registers to the correct parameters. AH is typically used to identify the exact command we are wanting to use, here AH is set to 0x0E which is the command for "Teletype output" used when writing characters to the screen. The command takes three parameters, AL is the character we want to print, BH is the page to print to and BL is used to state the colours to use. Other instructions will use the same idea for using BX, CX, DX and ES:BP for parameters, tables containing each BIOS function, their parameters and return values can be found online. (I.E. Wikipedia INT_10H).

The most common functions that will be necessary are the following, however more functions can exist and some may act differently on different BIOS's:

- INT 0x10 - Video function
- INT 0x13 - Storage device access
- INT 0x15 - Memory mapping functions
- INT 0x16 - Keyboard functions

In my case all of the instructions act as expected on both the QEMU BIOS and hardware BIOS so I will not need to make changes to my code. If you find there are issues with commands you should lookup your specific BIOS version and see what their documentation states and substitute their equivalent within your code.

2.4.3 Loading stage two into memory

The BIOS is there to be our friend, it loads data, allows some system configuration and then it loads our boot sector and jumps to it. In most(?) cases the BIOS will provide us the ID of the boot disk in the BL register, we should save this value for later use as we may need to use the register between the beginning of execution and when we want to use it. Before loading data we need to know which parts of memory are free for use as some memory may be used for hardware devices or other system data, this will be shown in more detail later however know that the addresses used here are free for use in QEMU. Here we are going to load the second stage to memory location 0x0500. To load the data we can use BIOS function AH 0x02, INT 0x13. This function has been found to work differently between QEMU and physical hardware, QEMU allows for the reading of 0xFF sectors at a time which is not possible on hardware that does not support multi-track reading. It is also important that we correctly prepare the segment registers.

```
    ; Prepare segment registers - Set to zero
    xor ax, ax
    cli
    mov ds, ax
    mov es, ax
    sti
    ; Store the boot device to a memory location for future use
    mov [BootDevice], dl
```

The load sectors command uses ES:BX as the location to write to hence we set ES to zero to ensure that we are writing to BX = 0x0500. When changing the segment registers we disable interrupts with the cli command to ensure that nothing causes errors, once we are done we can re-enable them with the sti command.

Loading the second stage, isn't as simple as just calling the read function and jumping to our location, we need to ensure that the read has actually been successful. The process is dependant on making an attempt to read the disk and then returning whether a successful read has occurred using the carry flag. If the carry flag is set we have encountered an error and need to prevent the code from potentially executing junk. An example of an error condition would be attempting to read the medium without the boot disk being inserted such as with a floppy drive, this scenario may not occur (and is very unlikely) in the boot sector however it is always something that should be thought of.

```
Reset:
    xor ax, ax
    mov dl, [BootDevice]
    int 0x13
    jc Reset
```

First we will reset the medium to reduce the risks of any errors occurring. The reset follows suit and checks for any errors, this will effectively throw the system into an infinite loop until the reset was successful. This may be a good time for feedback however I will leave that to you (I do use this for the "LoadKernel" label in the RedFox32 source.

```
Load:
    mov     ah, 0x02            ; 0x02 means to read from disk
    mov     al, 15              ; read 15 sectors from the drive
    mov     bx, 0x0500          ; Destination
    mov     ch, 0               ; sector 4 is still on track 1
    mov     cl, 4               ; sector to read from
    mov     dh, 0               ; head number
    mov     dl, [BootDevice]    ; drive 0 is floppy drive
    int     0x13                ; read the
    jc Load                     ; If Carry is set then retry
```

The code above is starting to read from the 4th sector, the function takes the read location using Cylinder-Head-Sector addressing so to correctly read the 4th sector from the start of the drive we need to use position 0-0-4. In CHS addressing the cylinders start from 1 through 18. On a 3.5" Floppy disc there are 18 sectors per track therefore the code above would not work on a BIOS that does not allow multi-track reading, we would require an algorithm to read 18-StartSector sectors per track, while correctly calculating the CHS value, this can be done using Logical Block Addressing however requires more complicated code (Ultimately this will be what I do in the RedFox32 source code) however this function will work for a virtual system in QEMU. Now that we have loaded the second stage we are able to jump to our loaded code and have it execute!

```
jmp 0x0000:0x0500
```

Here we do the same as before and provide the jump with 0x0000 before the jump location to ensure that certain registers are set correctly and have not been modified. Now that we have an idea of how to load things in Real Mode, we're nearing the point of leaving it, that was short lived, however we aren't finished, the boot sector needs another compatibility structure and the magic byte to be added.

This section may be subject to change during project development, if you can read this then the project is still in development!

2.4.4 Writing the magic bootsector bytes

We are not quite finished, last but not least we need to provide the magic bytes to the end of the boot sector, this is easily done by skipping to byte 510 and writing the magic bytes.

```
times 510 - ($ - $$) db 0
Magic: dw 0xAA55
```

Now we should have a bootable medium, if you do try to run this it will just result in undefined behaviour as we have yet to create the second stage however the boot sector is now complete for use in a virtual QEMU machine!

2.5 Stage Two

2.5.1 How are we going to use the second stage?

We are going to use the second stage for switching to Protected Mode, loading then executing Kernel code and to load device drivers. The switch to protected mode needs to be taken in a few different stages to ensure that the system will act as we expect and not doing the unexpected and executing unknown data.

The second stage we create here will do the following steps in the given order:

1. Initialize the Global Descriptor Table
2. Initialize the segment registers
3. Enter 32 bit Protected Mode
4. Enable the A20 memory line
5. Prepare a stack
6. Call into our C code

In the first stage we loaded 15 sectors into memory at 0x0500; This gives us 7.5KB of memory in the range 0x0500-0x2300. This should be plenty of space to do what we need however our code may not be as verbose as we want. It is possible to run the load command as shown previously and starting the second load on the next cylinder starting from 0x2300, this would then allow us to load an additional 18 sectors per call if we need it. The second stage is primarily dedicated to further system setup with the ultimate goal of ending in a 32 bit environment with our kernel loaded and ready to move into so let's get into it!

2.5.2 The Global Descriptor Table

The Global Descriptor Table (GDT) is used to define memory usage characteristics on x86 processors. For simplicity of this project being solely for learning purposes we will use a ”flat” style GDT which just allows free access to all 4GB of potential memory! To do this we will use the following data structures:

```
; offset 0x0
.null descriptor:
dq 0

; offset 0x8
.code: ; cs should point to this descriptor
dw 0xffff ; segment limit first 0-15 bits
dw 0 ; base first 0-15 bits
db 0 ; base 16-23 bits
db 0x9a ; access byte
db 11001111b ; high 4 bits (flags) low 4 bits (limit 4 last bits)
             ; (limit is 20 bit wide)
db 0 ; base 24-31 bits

; offset 0x10
.data: ; ds, ss, es, fs, and gs should point to this descriptor
dw 0xffff ; segment limit first 0-15 bits
dw 0 ; base first 0-15 bits
db 0 ; base 16-23 bits
db 0x92 ; access byte
db 11001111b ; high 4 bits (flags) low 4 bits (limit 4 last bits)
             ; (limit is 20 bit wide)
db 0 ; base 24-31 bits

end_of_gdt:                     ; Save the end point
dw end_of_gdt - gdt_data - 1    ; limit (Size of GDT)
dd gdt_data                     ; base of GDT
```

If you do what I did and put this at the top of your stage 2 assembly file, ensure to have a jmp so that you don’t accidentally start running your GDT as if it’s code... It won’t go down too well. Now that we have defined our GDT we need to do some additional setup before we can actually tell the system to use it.

2.5.3 Switching to 32 bit protected mode

We need to ensure that we have the machine in a known state to help reduce errors. The first thing we will do is enter a known graphics state, for this we will use 80 column, 16 colour mode using the BIOS interrupts as we can still use these with ease until we have actually made the switch to protected mode.

```
; Switch to 80 column text mode
mov ax, 0x0003
int 0x10

; Disable the text mode cursor
mov ax, 0x0100
mov cx, 0x3F00
int 0x10
```

I am ignoring to show code that outputs characters in this document as there would just be lots of repeat code. For future reference the BIOS print character command is: AH = 0x0E, AL = char, BX = 0, int 0x10.

Next we will clear the segment registers to help avoid issues when we load our GDT.

```
; Disable interrupts, this prevents errors.
cli
; Set segment registers
xor ax, ax ; We have to use a register transfer
mov ds, ax
mov cs, ax
mov es, ax
mov ax, 0x9000
mov ss, ax
mov sp, 0xFFFF ; We can get away with this apparently
```

Okay so now the magic moment, we will load the GDT. This will do pretty much nothing except potentially cause errors if it is setup wrong.

```
; Load Global Descriptor Table
lgdt [end_of_gdt]
; We can now re-enable interrupts
; -- This allows us to print again as we couldn't previously
; -- If you don't intend on calling any BIOS functions
; -- then don't bother as we need to disable interrupts again
sti
```

Okay so we are ready to enable protected mode! This is incredibly underwhelming as it's just a flag we have to set in the cr0 register... It just takes a fair amount of system initialization

```
cli
; We can actually access the 32 bit registers in real mode
; one of those strange quirks.
mov eax, cr0
```

```
or eax, 1
mov cr0, eax
; we won't re-enable interrupts as we need to make further
; -- changes once we have jumped to 32 bit
```

Congrats! We are now in 32 bit mode. Before we can do any true 32 bit magic we need to jump to some 32 bit code using a long jump using a value defined by our GDT.

```
jmp 0x08:STAGE2_32BIT
```

If everything breaks something is wrong.
Please refer to the RedFox32/BootloaderStage2/Stage2A.asm file to see how I did this, there are NASM specifics for compilation that I have neglected to mention as this document should not be used as a direct tutorial but as a guide.

2.5.4 32 bit configuration - Segment registers

Segment registers
We're getting there, this is the last of the assembly that we need to do as far as setting up the system is concerned, after this we can use C for almost everything if we want, you could continue with assembly if you wanted however using C will be useful.

Firstly we will setup our segment registers to fit the GDT for protected mode:

```
; Initialize the segment registers
mov ax, 0x10    ; Load the value 0x10 into EAX
mov ds, ax      ; DS = 0x10
mov es, ax      ; ES = 0x10
mov fs, ax      ; FS = 0x10
mov gs, ax      ; GS = 0x10
mov ss, ax      ; SS = 0x10 as per GDT

; Setup the stack (Ensure that there's no incorrect bits in high)
; The stack is set to be the memory region: 0x9000-0xFFFF
; Set of e-- versions to ensure clean high bits
mov ebp, 0x9000     ; Set the base pointer
mov esp, 0xFFFF     ; Set the top of the stack to 0xFFFF
```

It's a good thing registers are easy to work with! Ultimately the original idea of the segment registers was to use segmented memory so that each memory segment could be used for a specific task with given management requirements however, these are typically unused in favour of other memory management solutions such as paging.

2.5.5 32 bit configuration - A20 addressing line

You would expect us to already have the ability to address the full range of system memory... However surprise surprise, with x86 this is not the case! We need to enable the A20 address line so that we can access ”high memory”.