NOTE: This book is not yet complete, incomplete sections are marked with "TODO".

1 Introduction

The purpose of this introduction is to set scene for what should hopefully be an enjoyable delve into the world of operating systems and operating system design. This book does

not represent ground breaking research by an means at the time of writing, but could be considered an insight into the history of how we got here and should start to build a basic understanding of the complexities of operating systems and operating system design.

How this book differs from others is it tries to assume as little as possible, with only requiring a simple background in programming and data structures. That said, even this is something that a smart individual may not need in order to understand this book. Another place in which this book differs from others is the speed at which we attempt to cover subjects. A competent programmer could complete this book in an extremely short amount of time, as a lot of the theory could be skipped and simply the code read to gain an understanding. Less able readers may take longer, but in general the journey taken should be as short as the individual needs in order to cover this subject.

It is recommended that you read this book from start to finish and do the practicals as and when they appear in the book, as this is how the book was designed to be used. In the section "Book Layout" you will also see more detail about how this book is formatted in order for key information to be recognised in the text. Once comfortable with the text in that section, it is then suggested that you tackle the text at your own pace.

1.1 Book Layout

This section contains information about the formatting of this book. It is highly recommended that you read this section in it's entirety before picking and choosing which parts of this book are appropriate to you in order for the other sections in this book to make sense.

1.1.1 In-line Code

In-line code can be either an explanation of code seen in a "Code Block" or complete in it's own right. It should represent something that is something a machine would understand, but also be too short that it doesn't make sense to be in it's own code block.

The formatting is as follows:

```
Code Here
```

In context, this may be used as follows:

You may want to move the value to the `AX` register.

1.1.2 Code Block

A code block should be one line or many, although if it is one line it should be of a considerable length such that it would not format well as "In-line Code". Like the in-line code, this text represents something that a machine would understand.

The formatting is as follows:

```
Code Here
More Code Here
```

In context, this may be used as follows:

```
; Make AL equal to 21
mov al, 10
add al, 10
inc al
```

1.1.3 Notes

Notes should represent something that you can choose to read or not. They usually are relatively small but can be very useful and read as advice.

The formatting is as follows:

NOTE: Text Here.

In context, this may be used as follows:

NOTE: Be sure to read the rest of this section.

1.1.4 Warnings

Warnings are something you are highly suggested to read. Some things we do may be dangerous, or simply cost you time or effort if you get them wrong. Sure, you don't have to read them - but if something goes terribly wrong the responsibility is on you. It doesn't matter how experienced or intelligent you are, mistakes are a very human aspect we all share.

The formatting is as follows:

WARNING: Text Here.

In context, this may be used as follows:

WARNING: Do not skip over these messages.

1.2 What Is An OS?

We'll first start by defining a selection of OSes exist out there, which should make the explanation easier for most people out there. The following are very popular operating systems (the list is far from complete):

- Android (Google)
- DOS (Microsoft)
- iOS (Apple)
- Linux
- Mac OS (Apple)
- Windows (Microsoft)

Hopefully at least one of those rings a bell, better still you've used one of these at some point.

The core job of an OS is to provide a level of abstraction from the hardware it's running on. This can be an extremely simple level of abstraction, meaning the underlying is still very much visible, all the way to a program having no idea what computer it is running on.

The reason for this abstraction is to allow a program to easily be used on a piece of hardware without having to worry about the hardware's specifics, allow multiple programs to easily be run on the same platform and for many OSes it will also allow for multiple programs to be run at the same time with minimal awareness of the fact (in most cases).

We will concentrate on getting simple programs to run fairly oblivious to the hardware they are running on - but more on this later.

1.3 Why Build An OS?

There are many reasons to build an operating system (OS) and those reasons will vary depending on who you are and what your goals are. Here are just the few of those:

- *General curiosity* - These computer things surround us everywhere in desktops, laptops, phones, microwaves, routers, vacuum cleaners, watches, toasters, washing machines, light bulbs - the list really does go on. These are just things you have in

your modern home, most of which will be running one of many different types of operating system.
- *Advancing knowledge* - This is a rather pure pursuit and a difficult thing to do. This book will try to guide you by using understandable English for the most part and attempting to describe concepts in ways that are understandable. For those that are already advanced, this can be a quick and satisfying journey to adding another notch to your rather formidable belt.
- *Fun project* - Here we should hopefully not only complete the task of creating a fun side project, but also motivate many future side projects! Once you have read and understood this book, without a doubt you will have more questions than what you started off with. This is of course good and is what drives Science and explorers of the unknown in general.
- *I always wanted my own [My Name] OS* - Doesn't everybody? By the end of this book you should have something to show the nerdiest of your friends, colleagues, students or family. For those who do not know, some of this can also seem like impossible magic.
- *I want to be [Bill Gates/Steve Jobs/Linus Torvalds]* - Whilst not an easy thing to do, it's also not impossible. New ideas happen all the time and they often come from surprising places, all of which share in common some dedicated and motivated individual/s that allowed these large goals to become reality.

If you don't consider yourself to be one of these categories, you are also welcome. At the end of this book will be general information, some of which will contain various contact information. Please be sure to explain what has motivated you to pick up this book and we will look to adding it to the list.

1.4 What OS Will We Build?

Please bare in mind that the following may require some technical knowledge that will be learned during the development of this system. If you see words or concepts you don't understand, we will visit them later. For those in the know, we thought it best to justify design decisions before starting so that the design decisions make sense if you choose to jump further ahead.

The Operating System we plan to build is the following, with the attached reasoning:

- *16 Bit* - The registers, memory model and boot process is much simpler for a 16 bit OS. Switching to 32 bit requires more knowledge and comes with it's own difficulties, including but not limited to the supported instruction sets for 32 bit machines not always being the same. The X86 architecture is old and gold, with it's standard well defined. It's likely that we would be able to run our code on any desktop/laptop computer built within the last 10 years with an Intel processor.

- *16kB RAM* - This means that any machine that supports the architecture would also have enough RAM to run our system, also meaning that we don't have to switch to different memory contexts which is required for addressing larger amounts of RAM.
- *Custom File System* - For a simple OS, we don't care about file sizes, when the file was created, who created it, who has permission to edit it, etc. We care about the name and where it's stored. For this reason, we can keep the file table (more on this later) small and store it completely in our small amount of RAM.
- *Command Line* - It is possible to draw graphics in 16 bit mode and quite good graphics at that - what becomes difficult is providing an interface in this mode that is fast enough to be usable. This requires more effort and defeats the object of this book.
- *Monolithic* - The kernel will only be able to run one program at a time. This is again for the ease of the reader, as debugging a program with multiple threads can become a headache for anybody. Design wise, we also don't need to then create task switching capability, memory management, shared resources, etc.

2 Build Environment

A build environment is simply a place where we will build our system (or any software). The reason for creating this environment is we will need specific tools to achieve our goals, something most operating systems are not capable of out of the box. The reason for this is that most users will not want to do anything this advanced - so consider yourself one of the privileged few who will venture under the hood of the computer!

2.1 VirtualBox

To keep things simple, here we will suggest the use of VirtualBox, which can be downloaded from https://www.virtualbox.org/wiki/Downloads. The idea is that we should be able to abstract your choice of Operating System from the work we wish to do in this book. Of course this can be done on the native operating system, but explaining how to do this three or more times for each step is somewhat tedious. For the simplicity of the book, we will be using a very small Linux operating system to do our tests. Insider that operating system, we will then run another virtual machine to minimise the risk to your real machine (as we will discuss later, there is the potential to wipe you bootloader and partition table if care is not taken).

VirtualBox has been chosen for being open source, supporting Windows, OS X, Linux and Solaris - it seems to be the best compromise. There are of course others, but when in doubt we all tend to stick with what we know. Once downloaded and installed, we can now look to download the image we will use to run the system.

2.2 Development Environment

The development tools can be found at http://coffeespace.org.uk/downloads/os-from-scratch.zip, which you should be able to easily run without modification. Please note that these tools are very simple and not hardended in any sense, so pleae only use these for the purposes shown in this book. The following will be a description of how to use these tools to succesfully build your code.

Fully extract the ZIP file for full access to the tools.

2.2.1 Image Creator

For this tool you will need the latest version of Java installed (Java 8 at the time of writing). To run, simply type `java -jar ic.jar` into your terminal (command prompt, console, etc). A list of uses should then appear. The default configuration is that of this book, which is a simple floppy drive. Further on we will discuss how to build images using this tool.

2.2.2 NASM

NASM (Netwide Assembler) is a compiler for assembly into machine level code. This can be found both in the ZIP file which has been tested for every line featured here, or at http://www.nasm.us/ which is untested. The included version is a snapshot of multiple versions from their site and is unmodified, including all instructions and liscensing.

3 Basics

Here, we run through the basics in order to make sure that we have the best chance of getting through the material. Feel free to skip through this material as you feel comfortable, for programmers, engineers and any person with a scientific background it is expected that this will be trivial. For those that are still here, we all had to start somewhere and now is as good a time as any.

3.1 Base

If you are not aware about number bases, prepare to have your mind blown. Your whole life you have been using one without any particular reason and without any decision being made. Our ability to change between these bases will be useful later on.

When you count, if you are Western you will count from zero to nine, before adding a number to the left and contuing. An example would be to have nine, add one and have 10. Zero to nine is only ten numbers, the reason for which is the number of fingers we have. For humans, it's easy to count using your fingers as tools for counting. Other places in the world use joints in the fingers, allowing them to increase the *base* of their counting system.

Okay, so why is this interesting? Well, a computer really only has one finger. On or off. True or false. One or zero. Base 2. Binary. This makes counting interesting. We have zero and add one, now we have one. We have one and... Well, we have 10.

Let us visualise this for 128. In base 10 (the one you were likely taught), we have the following:

```
100's 10's  1's
    1    2    8
```

To find out the number, we effectively do the following:

```
(100 x 1) +
( 10 x 2) +
(  1 * 8) =
128
```

Now we want to represent the base 10 number in base 2. We again visualise this:

```
128's 64's 32's 16's  8's  4's  2's  1's
    1    0    0    0    0    0    0    0
```

And this is calculated out to be:

```
(128 x 1) +
( 64 x 0) +
( 32 x 0) +
( 16 x 0) +
(  8 x 0) +
(  4 x 0) +
(  2 x 0) +
(  1 x 0) =
128
```

Hmm, this is interesting, but it takes more space? Well, computer scientists have this covered too - base 16, or hexadecimal. Base 16 is 0 to 15, which causes a problem. To

solve this, new numbers are created. We now count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

To put this into an example, we now see:

```
16's 1's
   8   0
```

And calculated out is:

```
( 16 * 8) +
(  1 * 0) =
128
```

Awesome, we saved a character when writing that number! When writing a lot of numbers, this will make some serious difference!

But... There is an issue here. Using all these different bases cause confusion in some cases. The number 10 could be the following:

```
Base  2: ( 2 * 1) + ( 1 * 0) =  2 (in normal counting)
Base 10: (10 * 1) + ( 1 * 0) = 10 (in normal counting)
Base 16: (16 * 1) + ( 1 * 0) = 16 (in normal counting)
```

There is a simple solution for this, in which we add some identification to the number in order for us to know which base we are dealing with. The following is the notation used:

```
Base  2: 10b ('b' for binary)
Base 10: 10 or 10d or 10.0 ('d' for decimal)
Base 16: 10h or 0x10 ('h' for hexidecimal)
```

Hexadecimal (base 16) is typically the one used for computers as computers often work with bytes, a format that only requires two character to represent a number from 0 to 255, or 0h to FFh.

3.2 Basic Computer Math

Now we start with some basic computer math so that we understand the computer instructions and logic to come. Very basic math is of course assumed and built upon.

- **/** or **//** = Integer Divide: An integer is a whole number, 1 is an integer but 1.5 isn't.

`5/2` in this case would be 2, remainder 1. In this case we don't care about the remainder and integers don't deal with decimal places. The answer would be 2.

- **%** = Modulo or Modulus: `5%2` in this case would be 2, remainder 1 as before, but in this case we care about the remainder, therefore the answer is 1.
- **<<** = Bitshift Left: Shifts the digits in a binary number to the left and makes the right most digit a zero. An example would be `1101b<<1`, meaning we want to shift the binary number one place to the left. The answer in this case would be 1010b.
- **>>** = Bitshift Right: Shifts the digits in a binary number to the right and makes the left most digit a zero. An example would be `1011b>>1`, meaning we want to shift the binary number one place to the right. The answer in this case would be 0101b.

If these concepts are still confusing, please take some time to reread and make sure that you are confident with these ideas. These, along with basic mathematical concepts will be very important in understanding what is happening as we build our own OS.

4 Bootloader

"What is this bootloader you speak of?" you may ask yourself. Let us break it down. It's the words "boot" and "loader" together for a reason. The "boot" process is the starting of the computer and the "loader" is the getting the more interesting parts into action. On boot (now only valid for older systems running a traditional BIOS), the computer will load a small piece of code into RAM that effectively has instructions for the rest of the system to run.

4.1 Boot Process

When a computer is booting, the first thing that will run is the BIOS software (this doesn't include computers running UEFI). This is usually responsible for loading the kernel from the storage medium and starting the process of executing from RAM. Of course for our initial simple example, we will simply just load the bootloader and prove to ourselves we have booted. Later, we will look to loading our own custom kernel.

4.2 RAM Layout

For now, we are only interested in the following:

- 0x0000:0000 to 0x0000:7C00 - Reserved for the BIOS and other low level code.
- 0x0000:7C00 - The position in RAM that the bootloader will be loaded into.

In theory we have access to 1MB (One Mega Byte) of RAM (Random Access Memory) available to us, but due to simplicity we will only have access to a much smaller amount to prevent us having to switch the RAM space we are working in. At this low level, unfortunately we don't have the luck of getting any memory management, meaning that we have to be careful about how we use memory!

4.3 Registers

- `AX` - (General Purpose) Multiply, divide, load and store
- `BX` - (General Purpose) Index register for move operations
- `CX` - (General Purpose) Counting for operations and shifts
- `DX` - (General Purpose) Port address for IN and OUT operations
- `SP` - Top of stack pointer
- `BP` - Bottom of stack pointer
- `SI` - Source index for streams
- `DI` - Destination index for streams
- `IP` - Instruction pointer
- Flags (see https://en.wikipedia.org/wiki/FLAGS_register)

More information on registers can be found https://en.wikipedia.org/wiki/X86_assembly_language.

NOTE: Some registers may be processor specific - the fact they work on your machine or in a virtual machine does not mean they will work for a wide variety of hardware. Be sure to use the most generic instructions possible to make sure you have as large support as possible.

4.4 Operations

The following operations are not the complete set of possible operations, that list is very long indeed! But here are a few instructions that are used regularly and their purpose - enough to get you setup.

- `add X, Y` - Adds X and Y and stores the result in X.

- `call X` - Pops the current position in memory onto the stack and jumps to the position X. To return from a function, see `ret`.
- `cmp X, Y` - Short for "compare", this operation compares the values X and Y, storing the result in various flags (which can be used by jumping, via `je`, `jl`, `jg`, etc).
- `dec X` - Decrements (subtract one) from X and stores the value in X.
- `div X` - **??**
- `inc X` - Increments (add one) to X and stores the value in X.
- `int X` - Calls an interrupt, X, which in turn performs an operation depending on the register values.
- `jmp X` - Jump to a position in memory denoted by X.
- `lodsb X` - Loads a byte from memory into X, the position denoted by registers.
- `mov X, Y` - This instruction is short for "move" and moves the contents of Y into X, but Y remains the same.
- `mul X` - **??**
- `pop X` - Pops a register from the stack denoted by X.
- `popa` - Pops all the general purpose registers onto the stack.
- `push X` - Pushes a register to the stack denoted by X.
- `pusha` - Pushes all the general purpose registers to the stack.
- `ret` - Stands for "return", allowing a return to the previous position of a call operation.
- `stosb X` - Stores a byte from X into memory, the position denoted by registers.
- `sub X, Y` - Subtracts Y from X and stores the result in X.

Further information about operations can be found at https://en.wikipedia.org/wiki/X86_instruction_listings#x86_integer_instructions. We will try to explain operations as we go in the text, comments of the code and operation of the code.

4.5 Simple Bootloader

Now it's time to boot the first piece of code!

First, write the following code to a file called `boot.asm`. We aren't going to concern

ourselves with exactly how the it works for this moment, we will discuss this later.

NOTE: You may be able to work out exactly how this works by looking at the above operation codes as well as the comments. Searching online may also be required.

```
; Align the code to the position in RAM we will load from
org 7C00h
boot_start:
  ; TODO: Write some code here.
boot_stop:
  ; Jump infinitely
  jmp $
; Pad out the bootloader to 510 bytes
times 0200h - 2 - ($ - $$) db 0
; Add the boot signature to the code
boot_signature dw 0AA55h
```

Now we will need access to the terminal, or this may be called "Command Prompt" or "Shell" depending on your machine. This will be different per operating system, so it's reccomended that you learn how to do this for yours. You will also need the ability to navigate the directories of the operating system via this terminal.

Next, we will have to compile this code with the following command:

```
nasm boot.asm -o boot.bin
```

This means that we are using the source file, `boot.asm`, and writing an output (`-o`) file, `boot.bin`.

NOTE: If the terminal complains about the program `nasm` not existing, try moving your code to the same place that the `nasm` binary is located.

A new file should now be located where you compiled your program. The next step will be to copy this file to the location of `ic.jar`, our image creator for this operating system. To use this program, run the following:

```
java -jar ic.jar *.bin
```

This tells the temrinal to run `java`, the file we want to run is a JAR (`-jar`), the program to be run which is `ic.jar` and what to load into the program (`*.bin`). The `*` is a

wildcard, meaning that any files we compile with the `.bin` extension on them will be included in the image generated.

Once this program has complete, we should see some output such as:

```
[ADD] `boot.bin`
[DEL] `boot.bin`
[SET] bootloader -> boot.bin
[SET] output -> os.img
[SET] mediaSize -> 1474560
[>>>] Finished generation
```

There should now be a new file called `os.img` in the directory. This is our floppy disk image! We will now need to load this image into the floppy disk section of our virtual machine. When we do so we should see that the it boots... That's it. Okay, it's not exciting - so let us create something more interesting!

NOTE: If the virtual machine complains about not having an bootable media, it means that you have not mounted the files correctly. If you are not able to easily work this out, please use your favourite search engine to achieve this task for your specific virtual machine.

4.6 Simple Printing Bootloader

Next, we look to create a bootloader that does something more useful to us. In this case, we look to display some text with a basic printing function that we write. The below code is broken up with various comments about what each part is doing in more detail.

```
; Align the code to the position in RAM we set ourselves
org 0h
```

The first notable difference, we have organised our code at offset `0x0000:0000`. This is because we orient ourselves from where this bootloader starts below.

```
; boot_start()
;
; Starts the boot process.
boot_start:
  mov ax, 08E0h
  ; Disable interrupts while changing stack
  cli
  mov ss, ax
  mov sp, 4096
  ; Restore interrupts
  sti

  ; Direction for string operations 'up' - incrementing addresses
  cld

  ; Set all segments to match where kernel is loaded
  mov ax, 07C0h
  ; Handle the segments
  mov ds, ax
  mov es, ax
  mov fs, ax
  mov gs, ax
```

Here we can see we align our position in RAM by setting the offset of `0x0000:07C0` in the register `AX` and setting the various segment registers to the content of `AX`. This means that anything that is RAM position dependant before this code will not have the correct alignment, as the segments will still be at the start of RAM instead of our ofset.

```
  ; Needed for some older BIOSes
  mov eax, 0
```

Sometimes these quirks creep in, usually due to bad design at some stage. Possibly some BIOS assumed this would be set and as a result, other bootloaders must set this value in order to boot those machines.

```
  ; Make sure we start in correct text mode
  mov ax, 03h
  int 10h
```

Before continuing, we want to make sure that we are in the right text mode to print an error or prompt if we have to. In normal operation we would start looking for a kernel -

which may or may not be on the disk. The only thing we can assume is that the bootloader has been correctly loaded, everything else is a variable we must consider.

```
; boot_main()
;
; The main code to be run in the boot process.
boot_main:
  mov si, message
  call boot_print
```

Here, we load the position in RAM of the message into the register `SI`, indicating the index of the source information. We then call our `boot_print` function to print the message.

```
; boot_stop()
;
; Stops the bootloader from running.
boot_stop:
  ; Jump infinitely
  jmp $
```

When the above call to `boot_print` finsihes, it will continue execution to this code section. `jmp` tells the processor to start executing at a position, `$` indicating the current position. This means we have an infinite loop - there is no way to escape until the computer is switched off.

```
; boot_print()
;
; Prints a string to the display.
;
; @param SI Position of the string.
boot_print:
  pusha
  mov ah, 0Eh
.repeat:
  lodsb
  int 10h
  cmp al, 0
  jne .repeat
.done:
  popa
  ret
```

`pusha` tells the processor to put all the general purpose registers onto the stack for safe keeping. `popa` in turn returns them from the stack, back into the general purpose registers. Next we see that we put `0Eh` into `AH` - this indicates the later intterupt should be to print some teletype text. This means the text will automatically wrap at the end of the display and will increment the cursor (the blinking line in the text editor so you know where the keyboard is) position.

`lodsb` loads the next byte from the position indicated by `SI` into `AL`. `SI` is incremented every time it is called. `int 10h` is a video interrupt, in this case to print our character stored in `AL`. There are various others with advantages and disadvantages.

We then compare the contents of `AL` to zero. If it's not equal to zero, we jump back to `.repeat` and continue printing. The jump is `jne`, standing for "jump if not equal". If `AL` is equal to zero, no jump occurs and the code runs through to `.done` section. At the end, we have `ret` to return to where we were called from.

NOTE: The `.` in front of the label means that the label is only in scope for the above label without a `.`. This means that the label cannot be referenced from outside it's scope.

```
message db 'Hello World!', 0
```

We store our message in bytes (`db`), with a terminator byte being `0` to indicate the end of the string.

```
; Pad out the bootloader to 510 bytes
times 0200h - 2 - ($ - $$) db 0
```

We pad the bootloader with zeros to make sure it is of the right size to be accepted.

```
; Add the boot signature to the code
boot_signature dw 0AA55h
```

The `boot_signature` is very simple way for the BIOS to know whether there is a valid bootloader to be loaded. Not all medium attached to the computer will contain runnable code - this is how the computer knows whether to execute the code or not.

There we have it, our first interesting bootloader!

4.7 Very Simple Kernel

The below code is for a very simple kernel, something that we will not go through in detail her. As you can see, it very much looks like the bootloader code and you should be able to work out it's purpose.

```
; Align the code to the position in RAM we will be loaded to
; NOTE: We leave a 512 byte gap for the file table to be loaded.
org 400h

; kernel_start()
;
; Starts the kernel process.
kernel_start:

; kernel_main()
;
; The main code to be run in the kernel process.
kernel_main:
  mov si, message
  call kernel_print

; kernel_stop()
;
; Stops the kernel from running.
kernel_stop:
  ; Jump infinitely
  jmp $

; kernel_print()
;
; Prints a string to the display.
;
; @param SI Position of the string.
kernel_print:
  pusha
  mov ah, 0Eh
.repeat:
  lodsb
  int 10h
  cmp al, 0
  jne .repeat
.done:
  popa
  ret

message db 'Hello World!', 0
```

Compile the code as `kern.asm` and include it on the command line if the image creation utility.

4.8 Booting the Kernel

Now for the exciting part, we are finally going to boot our first kernel! First, we must think about the steps required in the boot process in order to get us there. We must do the following to boot the kernel:

1. Load the file table into RAM.
2. Search file table for the kernel entry, `kern`.
3. Load the kernel sectors into RAM.
4. Begin executing the kernel.

If we fail to load the kernel, we will want to do thefollowing steps after the previous third step:

4. Display error message.
5. Jump infinitely to prevent bad code execution.

For this to work, we are going to want some basic functions that we can use over and over:

- Abiltiy to load a file from a given position on the disk.
- A very simple comparison of a string and a position in the file table.

Effectively, what we are aiming to do is build a small, read-only command line operating system so that we can load files into RAM and run them, in this case the kernel. We will also want to display an error message if we fail to boot for some reason and allow a user to type in a different kernel name from the default.

Below is original bootloader with additions that will allow us to boot our kernel:

```
; Align the code to the position in RAM we set ourselves
org 0h

; boot_start()
;
; Starts the boot process.
boot_start:
  mov ax, 08E0h
  ; Disable interrupts while changing stack
  cli
```

```
  cli
  mov ss, ax
  mov sp, 4096
  ; Restore interrupts
  sti

  ; Direction for string operations 'up' - incrementing addresses
  cld

  ; Set all segments to match where kernel is loaded
  mov ax, 07C0h
  ; Handle the segments
  mov ds, ax
  mov es, ax
  mov fs, ax
  mov gs, ax

  ; Needed for some older BIOSes
  mov eax, 0

  ; Make sure we start in correct text mode
  mov ax, 03h
  int 10h

; boot_main()
;
; The main code to be run in the boot process.
boot_main:
  mov si, msg_loading
  call boot_print

  ; Load the file table into RAM
  mov cx, 2
  mov si, position_table
  call boot_load

  ; Search the loaded file table
  mov cx, 0
  mov di, str_file
  call boot_table_search

  ; Check that we managed to find a kernel
  cmp cx, 0
  je boot_stop
```

```
  je boot_stop

  ; Load the kernel into RAM
  mov si, position_kernel
  call boot_load

  ; Start executing our kernel
  ; NOTE: We call the kernel encase the kernel does a bad return.
  call position_kernel
```

Above we have our boot logic, where we run through the steps we previosuly mentioned. A lot of code will jump to the kernel, which is tecnhically correct. The issue with this is when writing a potentially bad kernel, a bad `ret` could have unknown concequences. In our code, this simply writes an error to the display.

```
; boot_stop()
;
; Stops the bootloader from running.
boot_stop:
  ; Display an error message
  mov si, msg_error
  call boot_print
  ; Jump infinitely
  jmp $
```

Here, we are printing a message to indicate there has been an error - both for our use and the use of any potential user. A better system would print some error code for the purpose of debugging too.

We should technically call the video mode interrupt encase somebody in the higher level decided to switch to a graphics code. Technically, the processor may even be in 32 or 64 bit mode, somethign that we should be able to test for.

```
; boot_print()
;
; Prints a string to the display.
;
; @param SI Position of the string.
boot_print:
  pusha
  mov ah, 0Eh
.repeat:
  lodsb
  int 10h
```

```
  int 10h
  cmp al, 0
  jne .repeat
.done:
  popa
  ret

; boot_table_search()
;
; Search the filetable for a file and return it's position, otherwize zero.
;
; @param CX Offset to search from, starting from zero.
; @return CX Sector containing the file, otherwise zero.
boot_table_search:
  push ax
  push bx
  push dx
  mov si, position_table
.loop:
  inc cx
  mov dx, 0
.check_pos_1:
  lodsb
  mov ah, [str_file + 0]
  cmp al, ah
  jne .check_pos_2
  inc dx
.check_pos_2:
  lodsb
  mov ah, [str_file + 1]
  cmp al, ah
  jne .check_pos_3
  inc dx
.check_pos_3:
  lodsb
  mov ah, [str_file + 2]
  cmp al, ah
  jne .check_pos_4
  inc dx
.check_pos_4:
  lodsb
  mov ah, [str_file + 3]
  cmp al, ah
  jne .check_loop
```

```
  jne .check_loop
  inc dx
.check_find:
  cmp dx, 4
  je .done
.check_loop:
  cmp cx, 512 / 4
  jl .loop
.not_found:
  mov cx, 0
  jmp .done
.done:
  pop dx
  pop bx
  pop ax
  ret
```

This method is by far the largest we've added so far, but is relatively simple. It's purpose is to find the position of a file on the disk, by checking for four letter filenames stored back-to-back in the file table. There are no folders, permissions, links - anything. Just a four character filename and where it points to.

What this method does is simple, it tests each character one by one and collects the number of matched chracters in `DX`, so that we know we have a full match if we have 4 characters. This code isn't as efficient as it could be, but it is easy to understand given some time to read through it. A more complex version would use a proper implementation of a string comparison.

```
; boot_load()
;
; Load a sector from the disk.
;
; @param CX The sector to load from the disk.
; @param SI The position to load the file into.
boot_load:
  mov ax, 0201h
  jmp boot_disk_manage

; boot_write()
;
; Write a sector to the disk.
;
; @param CX The sector to write to the disk.
; @param SI The position to read the sector from.
boot_write:
  mov ax, 0201h

; boot_disk_manage()
;
; Handle the disk operation.
;
; @param AL Number of sectors.
; @param AH The disk mode to operate.
; @param CX The sector to perform the operation on.
; @param SI The position in RAM to perform the operation with.
boot_disk_manage:
  push bx
  push dx
  mov dx, 0
  mov bx, ds
  mov es, bx
  mov bx, si
  int 13h
  pop dx
  pop bx
  ret
```

We implement both read and write functions to disk as they are extremely similar - almost no cost to having both exist. The function simply correctly sets registers and uses `int 13h` to perform the disk operation. As you may imagine, the time this was written

meant that it was very much designed to be used with a floppy drive, hence is a very limited way to communicate with devices.

```
msg_loading db 'Loading...', 10, 13, 0
msg_error   db 'Error', 10, 13, 0
str_file    db 'kern', 0

; Pad out the bootloader to 510 bytes
times 0200h - 2 - ($ - $$) db 0
; Add the boot signature to the code
boot_signature dw 0AA55h

; Positions of various important sections in RAM
position_table  equ 200h
position_kernel equ 400h
```

Congratulations, we can now boot our simple kernel successfully! There are some things we should bare in mind now that we are loading files:

- Our floppy disk is not compatible with any operating system, it's very custom!
- The kernel with this system can only be 512 bytes or less in size, anything larger and it will not be loaded. This is the same limitation with all files with this read method.
- We can only load a small number of files onto the disk as the filetable is very small. A larger file table will allow for more files.

NOTE: The limitation of the kernel size can be fixed later with the addition of loop to keep searching for a file of the same name on the disk. This does make our task more complex as we must then loop to find files which may be fragmented. Another complication is with removal, addition or editing files as we can no longer easily make any assertions about the files without loading them. With this filesystem, it's costly to check the size of a file for example. Modern operating systems typically precompute these values and have them with the filename making searching much easier. In fact, almost all data that doesn't require you to load the file to disk will be stored outside of the file.

5 Kernel

TODO: Write this section.

6 Programs

TODO: Write this section.

6.1 Hello World (`test`)

TODO: Write this section.

6.2 File Viewer (`ls`)

TODO: Write this section.

6.3 Cat (`cat`)

TODO: Write this section.

7 The Future

Firstly, congratulations on getting this far. You now have a working operating system, upload it to the internet and see where it takes you! Show your friends, family, colleagues, class mates, children, cats, plants - let them all know what you have achieved.

But now you may be wondering, "What next?"... Well, the journey is not yet over and if you want to continue - all the more to you. The following is a list of ideas that could be extended on the make this system much better.

- *More Programs* - You can never have enough programs! How about a simple text editor, simple game, serial port program - the list goes on. Anything you can think of can be created, given enough time. You will start to run into some limitations though, the first being the kernel...
- *Kernel* - Extend on the kernel, multi-core support, memory management, drivers, GUI support, etc. You'll have to do some additional reading for these concepts, but it's worth it.
- *Internet* - Computers really got interesting once we connected them together and how else better to test your programming skills than to connect your newly formed

operating system to the rest of the world? Start simple with a telnet server and see how far you can go!