-----======Bootloader======-----

- - - - - - - - - - - -

[Planning/Setting goals]

- - - - - - - - - - - -

What is a bootloader? A bootloader is basically bootable code that is loaded by the system at startup. Usually the bootloader is used as a base for doing other operations in the PC. How is the bootloader loaded? By way of the very first sector (512bytes) of a bootable device. For example, sector 0 of the floppy disk drive would be where you would put the bootloader's code. This "bootsector" is taken by the system and loaded into memory at 0x0:0x7C0 after the POST.

After the bootsector has been loaded into the system memory it is started. One of the first things done is the actual loading of the kernel data. Then there are things that are often needed to run the kernel data like:

```
 *enabling the A20 line
     (this allows access to 'full' memory space)
 *entering Protected mode
     (this allows 32bit addressing for x86 compatible systems)
 *setting up basic memory protection
     (called the GDT which holds address space privileges)
 *setting up an extremely simple interrupt table
     (IDT for kernel-based interrupts)
 *initialising a 32bit stack
     (Pmode requires a 32bit addressable stack for function parameters)
 *jumping to the address that contains the loaded data
     (this runs the kernel's data)
```

Now that we know the basics of what the bootloader does, let's setup a simple list of goals. First thing we do after the bootloader is loaded is setup the A20 gate. After this, we do what is needed to enter protected mode. Then we need to setup our GDT, IDT, and Pmode stack(32bit stack). Note that we may also need to move certain data to a different memory areas other than it's current position (ie-GDT).

Goals:
1-enable A20
2-load kernel
3-enter Pmode
4-setup temp GDT
5-setup temp IDT(optional)
6-setup 32bit stack
7-refresh registers
8-jump to kernel

Optional:
 -move GDT & IDT to specified memory area

- - - - - - - - - - - - - - - - - -

[First goal: Running the Bootsector]

- - - - - - - - - - - - - - - - - -

There are two things required to get a basic bootsector running: keep the bootsector at exactly 512bytes in size and the last two bytes of the code must be 0x55AA.

In Asm:

```
TIMES 510-($-$$) DB 0
SIGNATURE DW 0xAA55
```

This will make sure that everything below 510 bytes that isn't filled with code is padded with 0 and that the last two bytes are 0x55AA. That little piece of code should be the last in your bootloader. What you do between the beginning of the bootloader and the end of it is totally up to you. You could use BIOS to display a message that says, 'Loaded' for example, just to make sure it works.

- - - - - - - - - - -
[Loading the kernel]
- - - - - - - - - - -

This step is by far the easiest of them all. All you do is set the proper values and call int 0x13. All assignment values are shown here. Here is a quick set of instructions to follow for you to make your own loader code.

```
set following registers to appropriate values:
 ah=BIOS function(2)
 al=number of sectors to read into memory
 es:bx=segment:offset of memory location
 ch=track number
 cl=starting sector
 dh=head number
 dl=drive number
  *ah=(on error)sectors that were read in*

call int 13h
```

[ADD INT 13H BIOS INT INFO HERE(WRITE,READ)]

When you read in the kernel, you have to read at least the whole file size of the kernel.bin. Example, my kernel is, oh say, 1KB. That means I have to read in 2 sectors because 1 sector is 512 btyes and 2 * 512 bytes = 1024 bytes or 1KB.

Remember that different storage devices have different dimensions, so be careful to pay close attention to what you use to load the kernel. A weird thing that happens sometimes when you read in more than 18 sectors at a time and read again is that BIOS trashes some registers. So don't forget to reset ALL registers for each read.

- - - - - - - -
[A20 enabler]
- - - - - - - -

This is often a big step to understand and code, unless you steal someone else's code. The A20 gate is a bit in the keyboard's controller that enables or disables a mode called "wrap-around". The A20 gate simply allows the cpu to manage memory through a 20-bit bus instead thus attaining access to over the 1MB mark. If the A20 gate isn't enabled, it would wrap back around to the beginning of memory.

To change the A20 gate, you access hardware port 0x64 and 0x60 (kbd).
Here is a quick list of the kbd ports.

-{8042 kbd controller ports}-

PORT	ACTION	PURPOSE
0x60	READ	Output register for getting data from the keyboard
0x60	WRITE	Data register for sending kbd controller commands
0x64	READ	Status register that can be read at anytime
0x64	WRITE	Commmand register used to set options(like the A20 gate)

[ADD KBD BIT INFO TABLE HERE]

Steps to take:
1-disable interrupts
2-wait for the kbd controller to clear
  (if bit 1[00000001] is set that means input port isn't open)
3-write to kbd controller to set A20 gate
4-tell kbd conroller you want to write to output port
5-wait agin for kbd to clear
6-get status value and OR it by 2(bit 2[00000010] is A20 gate bit)
7-write new data to data port
8-write 'nop' to kbd
9-wait for kbd to clear
10-enable interrupts

There ya go, 10 easy steps :) If you need any help on this, just check out the provided example code. Moving on.

- - - - - - - - - - - - - - -
[Setup basic Stack/temp GDT]
- - - - - - - - - - - - - - -

Two things that are a must to jump in to your 32-bit kernel are a piece O' pie to setup. For the stack, all you do is setup its memory position, like so:

```
 ss   |  sp
-------------
0x0100:0x0200 = 0x1200 linear
```

In Asm:
```
mov ax,0x100
mov ss,ax
mov sp,0x200
```

You may need to know that the stack counts down, toward 0, so sp = 0x200 means that the stack is 512 bytes long. SS = memory segment of stack and SP = pointer/offset/size of stack.

Next we have the GDT(Note:you may have heard of some people loading the IDT in the bootsector. We will not do this because there is no visible advantage to doing it now). The GDT(Global Descriptor Table) is a table with structures for memory setup. Each GDT entry contains the following:

-{GDT Entry Table}-

Low Byte	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	High Byte

Limit	Limit	Base	Base	Base	Type	Flags &	Base
[0-7]	[8-15]	[0-7]	[8-15]	[16-23]		Limit	[24-31]

Limit: This is the limit address of the Pmode Segment (ie-0xFFFFF)
Base: This is the base address of the Pmode Segment (ie-0x0)
Type: This conatins the type of Segment it is (ie-code/data/writable/readable/stack/ring3..0)
Flags: Another set of options like 32bit or 16bit

-{Type Byte Table}-

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
P	DPL		S	C/D -	E -	W -	A
	2 bits			The Type nibble			

(P)resent:	Tells whther Segment is present or not
DPL:	Descriptor Privilege Levevl (ring3..0)
(S)egment Type:	Tells what kind of segment it is (ie-System=1)
C/D:	Tells whether segment is code or data segment
(E)xpand-Down:	Tells whether segment expands up or down in memory
(W)rite:	Is the segment writable or read-only?
(A)ccessed:	Every time the segment is accessed this bit is set

Once you have at least a code and data segment, alls you need to do is load the data with a pointer. The GDTR(GDT register) holds this data. You can load the data to it with a special command, but first, let me show you exactly what the data looks like.

In Asm:
```
GDTR:
GDTsize DW 0x10   ; limit
GDTbase DD 0x50   ; base address
```

This means that our GDT will begin at 0x500 and span 0x10 bytes. After that, you load it with this command:

In Asm:
```
lgdt[GDTR]
```

Then you're all ready to enter Pmode and jump into your kernel code. Remember that this is only the temperary GDT. One thing, the GDT is special in that it requires a null entry before all others. This 'null' entry is simple to create:

In Asm:
```
NULL_SEL
  DD 0
  DD 0
```

Ok, one MORE thing, another name for a GDT entry is a selector. This is because it selects memory, in plainest terms :)

- - - - - - - - - - - - - -
[Entering protected Mode]
- - - - - - - - - - - - - -

Another very easy, short, and simple step in the bootloader. All you need to do is change a bit in cr3(control register 3). Take a look at the steps.

Enabling Pmode:
1-get cr0's current value
2-OR it by 1(bit 1[00000001] is the Pmode bit)
3-save new data to cr3

Hey, hey! "It's Easy as A B C,1 2 3". This will allow protection levels for your OS. Ring 0 is the highest level and ring 3 is the lowest. Think of it this way, the lower the ring, the less bull-crap the cpu gives you about hardware access. The most significant effect of Pmode is that it allows you to access up to a whopping 4GB of memory. That's a whole 4,294,967,296 bytes!!! I know I won't have that much RAM till one of two things occur:

For me to have >=4GB RAM:
1-1GB chips are up for wholesale
2-Bill Gates adopts me

[Next Goal:Kernel entry]

Ahh, the most exciting moment in OSdev, seeing your hard work workin'
for you. In this section I will only briefly discuss how to jump into the
kernel. It is up to you to study the provided code and to understand it (it
is well commented, I coded it myself).

After enabling pmode and loading the GDTR, jump into the kernel at the address where you loaded it at. For example, if I loaded it to 0x5000
linear I would use my Code Selector from the GDT to jump to it.

In Asm:
```
jmp CODESEL:0x5000
```

In Asm(usually CODESEL=0x08):
```
jmp 0x08:0x5000
```

That doesn't mean segment 0x08, because in Pmode it means the offset
to the selector in the GDT. This will not be enough to load a kernel. Read on to the next chapter where it is explained more.There is still a linker script that has to be used to load all code correctly. Then we need code that calls the kernel function:

In Asm:
```
[bits 32]        ; 32bit code here
[global start]   ; start is a global function
[extern _k_main] ; this is the kernel function
```

```
start:
call _k_main     ; jump to k_main() in kernel.c
hlt              ; halt the cpu
```

It is probably a better idea to read the next chapter and see how it
is done there. Comment out the asm jmp command for now. That way it only loads "nothing" and doesn't try to run it.

And there ya go. If all goes as planned you will have a very simple 32bit Pmode kernel loaded up(well, at least a simple 32bit bootloader). :)

- - - - - - - - - - - - -
[Compiling the Bootloader]
- - - - - - - - - - - - -

What fun would it be to have a bootloader with just code? Let's run this bad boy and see how 'bad' you did :) First things first: get the bootloader assembly file and save it as, oh say, boot.asm. Now run this command in windows.

```
In Win:
nasm -f bin boot.asm
```

There really is no difference between linux and windows, just get nasm for linux (or whatever other supported platform) and compile boot.asm with the same parameters. This command will output a binary file call 'boot' usually without a 'bin' extension. If you want a 'bin' extension just do this:

```
In Win:
nasm -f bin boot.asm -o boot.bin
```

By the way, '-f' = format and '-o' = output. Ok, now take that file and use something like rawrite or similar and write it to sector 1 of the floppy. Stick that bad boy into a real PC and hope it boots right :) It would be a good idea to have some kind of output to make sure it works
like the source example has.

If you want to use bochs to run the binary file, then look at the example source code. It is easier to explain by example than for me to try to explain it to you. Also, read the bochs' readme file. That always helps :)