theos

me.

oday

Contents

ndisasm !

Important pieces of information:

0x7c00: Where the bootloader is placed by the BIOS. 0x55AA: Boot signature (at the 511th and 512th byte) the BIOS looks for when trying to boot up the bootloader.

What is memory?

- Piece of hardware that allow computers to store information.
- RAM (Random Access Memory) is the main memory, where programs read and write information.
- ROM (Read-only Memory) is a form of memory that can only be read from.

RAM.

- Temporary.
- Writeable.
- RAM is not persistent.

ROM

- Permanent.
- Can't be written to normally.
- Persistent.

![[Pasted image 20250429224017.png]] BIOS Chip (ROM).

Boot process.

- The BIOS is executed directly from ROM.
- The BIOS loads the bootloader into 0x7c00.
- The bootloader loads the kernel.

Bootloader.

- Small program responsible for loading the kernel of an operating system.
- Generally small.

When booting...

- The CPU executes instructions directly from the BIOS's ROM.
- The BIOS generally loads itself into RAM then continues execution from RAM.
- The BIOS will initialize essential hardware.
- The BIOS looks for a bootloader to boot by searching all storage mediums for the boot signature 0x55AA at the 511th and 512th byte.
  - If the sector is found, the sector will be loaded into 0x7c00 and execute from there.
  - A sector is just a block of storage, in hard drives them being 512 bytes.
- When the bootloader is loaded, the BIOS will do an absolute jump to the address 0x7c00 and start the operating system boot process via the bootloader.
- If the BIOS can't find a bootable sector, it can't do anything more.

The BIOS is almost a kernel by itself.

- The BIOS contains routines to asssit our bootloader in booting our kernel.
- The BIOS is 16 bit code which means only 16 bit code can execute it properly.
- BIOS routines are generic and standard (more later).

Setup.

- Install NASM (nasm).
- Install QEMU (qemu-system-x86) # 1. Writting a Hello World bootloader. ## The code

```
ORG 0x7c00 ;
BITS 16

start:
    mov ah, 0eh
    mov al, 'A'
    mov bx, 0
    int 0x10

    jmp $

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

ORG 0x7c00

- This line instructs the assembler to load the binary into address 0x7c00 ## BITS 16
- This instructs the assembler to only assembly instructions into 16 bit assembly code. ## start
- This is a label to represent where we'll write our code. ## mov ah, 0eh
- 0eh or teletype output is a way to write characters in the screen. ## mov al, 'a'
- The register `al` is a register that will be written on screen. ## mov bx, 0
- This sets the background color to black. ## int 0x10
- Executes the 0eh call (teletype output). ## times 510-(\$ -
) db 0
- This tells the assembler to fill space with 510 bytes of data.
  - times: the assembly directive that tells the assembler to repeat an instruction a given amount of times.
  - 510: is the amount of bytes that we want to fill,
  - $: is the current address of the code,
  - \$$: represents the starting address of the current section or segment in the code.
  - ($ -
    ): calculates the offset from the start of the section to the current address. This gives the number of bytes that have been used so far.
  - 510-($ -
    ): calculates the difference between 510 and the number of bytes used so far. This gives the number of bytes that still need to be filled to reach a total of 510 bytes.
  - `db 0`: This is the instruction that is being repeated. `db` stands for "define byte", and it tells the assembler to reserve space for a byte and initialize it with the given value. In this case, the value is `0`, so the assembler will fill the specified number of bytes with zeros. ## dw 0xAA55
- This puts the byte 0x55AA in the last two bytes. It's reversed in the assembly, as we're working with Little Endian architecture, where the most significant bits are flipped when working with words. ## Compilation

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

Why BIN?

- The BIOS has no concept of libraries, ELFs, or EXEs. It needs to be raw binary. It's a file without headers. Just raw code.

New code.

```
ORG 0x7c00
BITS 16

start:
    mov si, message
    call print
    jmp $

print:
    mov bx, 0

.loop
    lodsb
    cmp al, 0
    je .done
    call print_char
    jmp .loop

.done:
    ret

print_char:
    mov ah, 0eh
    int 0x10
    ret

message: db 'hola cybeer y resaca :)', 0

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

mov si, message

- Move the memory address of message into the `si` register. ### si IS NOT stack index.
- By the way: the `si` register IS NOT the stack index. It's the source index, used to store the offset address of a source opearnd in string manipulation operations (what we're doing with `mov ah, 0eh`.) ## call print
- This will call the print function. ## print:
- Just another label. In this case, we're defining a subroutine. ## mov bx, 0
- As before, we're setting the background color. ## .loop:
- This is a subsubroutine only accesible to the `print` subroutine. In the code, it's used to create a loop to print the characters. ## lodsb
- The `lodsb` directive loads the value of the `si` register into the `al` register and increment it. That's why we're not moving data into `al`, because `si` points to `message`.
- For example:
  - If `si` points to the memory section where `message` ("Hello world") is stored, the `lodsb` directive will put the **H** character in the `al` register and increment it, meaning that after the initial `lodsb` instruction, `si` will now point to **e** instead of **H**, and so on and so forth.
  - Remember: the `si` register DOES NOT contain the ENTIRE message. It stores the MEMORY VALUE of where message is at, beginning at the first character of the message itself. I will repeat: **IT DOES NOT CONTAIN THE ENTIRE MESSAGE!**. ## cmp al, 0
- This instruction compares the value of `al` with 0. This is because or message is terminated with a zero (or null byte). Depending on whether the comparison is true or false, it will… ## je .done
- …jump to the .done subsubroutine. It's like an `if`, although in this case it's more like a while, since:

```
while si != "0";
do
    print_char($si)
    si=$si+1
done
```

- But the logic is the same :) ## call print_char
- If `cmp al, 0` is false, the subrountine `print_char` will be called. ## jmp .loop
- After calling `print_char`, the program will go to the `.loop` subsubroutine, rerunning the `print` subroutine. Like a while loop :) ## .done
- Again, just a subsubroutine. It will be accessed only when `si` is equal to `0`. ## ret
- Returns to the calling function.

2. Understanding Real Mode.

- We have 1 megabyte of RAM accessible.
  - Memory is accessible through the use of segments.
  - It does not matter if you have more than 1 megabyte of RAM. The realmode compatibility layer acts as an older 8086 CPU.
- Based on the original 8086 design.
  - All the code MUST be 16 bits
- It has no security at all.
  - No memory security.
  - No hardware security.
  - Simple user programs can destroy our operating system with no way for us to stop them.
- We only have 16 bits accessible at all time.
  - We can access 8 and 16 bit registers.
  - It can be overwritten by a per-instruction basis, but thats a no go.
  - We can only request memory address offsets of up to 65535 for a given segment.
  - We can use only 16 bit numbers. # 3. Segmentation Memory Model # 3. IMPORTANT FORMULA: ss * 16 + sp
- Memory is accessed by a segment and an offset.
- Programs can be oaded in different areas of memory but run without problems.

- Multiple segments are available through the use of segment registers. ## 8086 Segment Registers
- `cs` - Code segment.
- `ss` - Stack segment.
- `ds` - Data segment.
- `es` - Extra segment. ## Calculating absolute offset.
- Take the segment register and multiply it by 16 and add the offset.
  - The offset is the absolute position in RAM that a particular byte is add.
    * Code Segment = 0x7C0
    * Assembly `org` is zero.
    * Our first instruction is at origin ero so our offset is zero.
      · (0x7C0 * 16) = 0x7C00
      · 0x7C00 + 0 = 0x7C00
        · Where 0 is the offset (origin). ### Offset examples for absolute address 0x7CFF
- Segment 0 offset 0x7CFF
- Segment 0x7C0 offset 0xFF
- Segment 0x7CF offset 0x0F
  - Calculation would be:
    * 0x7CF * 16 = 0x7CF0
    * 0x7CF0 + 0x0F = 0x7CFF ## Different instructions use different segment registers.
- `lodsb` uses the `ds:si` register combination. ### Code example:

```
ORG 0

mov ax, 0x7c0
mov ds, ax
mov si, 0x1F
lodsb
```

And… 0x7C0 * 16 = 0x7C00 0x7C00 + 0x1F = 0x7C1F ![[Pasted image 20250430121649.png]]

Program can be loaded into different areas of memory and run without problems.

- Imagine we have two programs, both with origin at 0.
- Program 1 uses segment 0x7C0 for all its segment registers.
- Program 1 is loaded at address 0x7C00.
- Program 2 uses segment 0x7D0.
- Program 2 is loaded at address 0x7D00.
- We swap segment registers when switching to the other process.
- We restore all the registers of the process we are switching to.
- It resumes as if we never switched processes at all. ## Multiple segments are available through the use of segment registers.
- `mov byte al, [es:32]`
  - Extra segment, the absolute offset in RAM that it'll get one byte of data will be the extra segment * 16 + 32d.
- `mov byte al, [ds:826]`
  - Data segment, the absolute offset in RAM that it'll get one byte of data will be the extra segment * 16 + 826d.
- `mov byte al, [ss:231]`
  - Stack segment, the absolute offset in RAM that it'll get one byte of data will be the extra segment * 16 + 231d. ## The stack segment.
- `ss` (stack segment) = 0x00.
- `sp` (stack pointer) = 0x7C00.
- `push 0xFFFF`: It decrements the stack pointer by 2.
  - That's because 0xFFFF is a two byte hex value. Meaning that we're storing a word, that why we're decrementing it by 2.
  - 2 bytes = word.
- Meaning that now, our `sp` turned from 0x7C00 into: `sp` = 0x7BFE.
  - because 0x7C00 - 2 = 0x7BFE.

- The combination of the stack segment and the stack pointer (0x7C00), when we push it becomes 0x7BFE.
  - That's because, again, a word is two bytes. We're doing:
    * 0x7C00 - 2 = 0x7BFE + 1 = 0x7BFF.
    * So our word is stored at memory address 0x7BFE and 0x7BFF.
      · 0x7BFE contains our low byte (0xFF),
      · 0x7BFF contains our high byte (0xFF). ### AI Help (by Llama) Prompt: I'm studying this topic further, now getting into the segmentation memory model. I have a question regarding the `sp` and `ss` registers. See, I have the following statements:
- `ss` (stack segment) = 0x00.
- `sp` (stack pointer) = 0x7C00.
- `push 0xFFFF`: It decrements the stack pointer by 2.
- Meaning that now, our `sp`: `sp` = 0x7BFE.
- The combination of the stack segment and the stack pointer (0x7C00), when we push it becomes 0x7BFE.
- We then set the bytes 0x7BFE and 0x7BFF to 0xFFFF, because we push our word into it.

Why is this? Where does 0x7BFF comes from? I understand that 0x7C00 - 2d = 0x7BFE, but I don't understand where that 0x7BFF is coming from! Please help!

Answer:

Let's break down what's happening when you push a word onto the stack:

1. **Stack Segment (`ss`)**: This register stores the segment address of the stack. In your case, `ss = 0x0000`.
2. **Stack Pointer (`sp`)**: This register stores the offset within the stack segment where the next item will be pushed or popped. Initially, `sp = 0x7C00`.

When you push a word (2 bytes) onto the stack, here's what happens:

- The stack pointer is decremented by 2 (because you're pushing a word, which is 2 bytes long). This means `sp` becomes `0x7C00 - 2 = 0x7BFE`.
- The word being pushed is stored at the new stack pointer location. Since `sp` is now `0x7BFE`, the word will be stored at `0x7BFE` and `0x7BFF`.

The reason you're storing the word at both `0x7BFE` and `0x7BFF` is because of how memory addressing works in the 8086 architecture:

- The stack segment (`ss`) is used as the base address for the stack. In your case, `ss = 0x0000`.
- The stack pointer (`sp`) is used as an offset within the stack segment. When you access memory using `ss` and `sp`, the address is calculated as `ss * 16 + sp`.
- Since `ss = 0x0000`, the base address of the stack is `0x0000 * 16 = 0x0000`.
- When `sp = 0x7BFE`, the address of the top of the stack is `0x0000 * 16 + 0x7BFE = 0x7BFE`.
- Since you're storing a word (2 bytes) at this location, the second byte of the word will be stored at the next address, which is `0x7BFF`.

So, to summarize:

- `0x7BFE` comes from decrementing the stack pointer (`sp`) by 2.
- `0x7BFF` comes from the fact that you're storing a word (2 bytes) at the location pointed to by `sp`, and the second byte of the word needs to be stored at the next address.

The calculation `ss * 16 + sp` is used to convert the segment and offset into a linear address. In your case, `ss = 0x0000` and `sp = 0x7BFE`, so the linear address is `0x0000 * 16 + 0x7BFE = 0x7BFE`. When you store a word at this location, the second byte will be stored at `0x7BFE + 1 = 0x7BFF`.

3. Improving our bootloader.

```
; modified origin
ORG 0
BITS 16

; new code
jmp 0x7c0:start
; end new code before start: label.
start:
    ; new code
    cli ; clean interrupts.
    mov ax, 0x7c0
    mov ds, ax
    mov es, ax
    mov ax, 0x00
    mov ss, ax
    mov sp, 0x7c00

    ;; for example purposes, not in the final code
    mov si, 20
    lodsb
    ; what lodsb is doing is:
    ; DS:SI, meaning that:
    ; ds * 16 + si, or:
    ; 0x7C0 * 16 = 0x7C00
    ; 0x7C00 + 20 = 0x7C14
    ; we do this because we do not know what the ds or si registers contain before we execute our program.
    ; we need to initialize them, as we do with mov ds, ax and mov es, ax.
    ; if we dont do this, we're just hoping that the BIOS sets the registers as we need them, but if
    ; we do it manually, we're taking control of things and making it correct every single time.
    ;; end of example code
    sti ; enable interrupts.
    ; end of new code
    mov si, message
    call print
    jmp $

print:
    mov bx, 0
.loop:
    lodsb
    cmp al, 0
    je .done
    call print_char
    jmp .loop
.done:
    ret

print_char:
    mov ah, 0eh
    int 0x10
    ret

message: db 'Hola amigos.', 0

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

jmp 0x7c0:start

- This instruction will ensure that the start: label will be at 0x7C0. ## cli ; clean and disable interrupts.
- This instruction clear all the interrupts. This will guarantee that our program will do only what our program is trying to do, and not execute any other interrupts set before by the BIOS.
- This will also disable interrupts. ## mov ax, 0x7c0
- Now we're setting this value to later on assign it to the data segment and the extra segment registers. This is how the CPU works; we cannot assign 0x7C0 directly to the `ds` and `es` registers. ## mox ds, ax
- Now we're setting the data segment to `ax`, or 0x7C0.
- This is part of the stack initialization. ## mov es, ax
- Now we're setting the extra segment to `ax`, or 0x7C0.
- This is part of the stack initialization. ## mov ax, 0x00
- Now we set `ax` to 0x00, because… ## mov ss, ax
- …we'll set the stack segment to 0x00. If we remember correctly, when setting up the stack, `ss * 16 + sp` is the absolute memory segment. ## mov sp, 0x7c00
- Now we assign 0x7C00 to the stack pointer, making our origin, effectively the 0x7C00 address, which is what we need. ## sti ; enable interrupts.
- Now we enable interrupts, because we will need them :) ## Compilation

```
# as always...
nasm -f bin boot.asm -o boot.bin
```

AI Help.

Prompt:

```
My issue now is with the initialization itself.
- Why do we have to set the `es` register?
- Is `si` like the stack index on modern CPU architecture?
- What does the `es` register do?
- What would happen if instead of `sp` pointing to 0x7C00, it would be the `ss` register pointing to 0x7C00 instead?

Thanks!
```

Answer:

```
Let's address each of your questions:

### 1. Why set the `es` register?

In this code, the `es` register is set to `0x7c0`, which is the same value as the `ds` register. This is because the `es` register is used as the destination segment for string operations, such as `lodsb`

In this specific code, the `es` register is not actually being used, since we're not performing any string operations that involve the `es` register. However, it's a good practice to initialize the `es` re

### 2. Is `si` like the stack index on modern CPU architecture?

No, `si` is not like the stack index on modern CPU architecture. In x86 assembly language, `si` is the Source Index register, which is used as an index register for string operations.

In the given code, `si` is used to point to the `message` string, and the `lodsb` instruction loads a byte from the memory location pointed to by `ds:si` into the `al` register.

In modern CPU architectures, the stack index is typically implemented using a stack pointer register (such as `rsp` in x86-64) and a base pointer register (such as `rbp` in x86-64). The `si` register in x8

### 3. What does the `es` register do?

The `es` register is the Extra Segment register, which is used as the destination segment for string operations, such as `lodsb` and `stosb`. It's also used as the segment register for some instructions, s

In general, the `es` register is used to specify the destination segment for data transfer operations, while the `ds` register is used to specify the source segment.

### 4. What if `ss` pointed to 0x7C00 instead of `sp`?

If `ss` pointed to 0x7C00 instead of `sp`, it would mean that the stack segment would start at address 0x7C00. This would be incorrect, because the stack segment should start at a lower address, and the st

In the given code, `ss` is set to 0x00, which means the stack segment starts at address 0x0000. The stack pointer (`sp`) is then set to 0x7C00, which means the top of the stack is at address 0x7C00.

If `ss` pointed to 0x7C00, it would cause the stack to grow downwards from address 0x7C00, which would overlap with the code segment and potentially cause stack corruption or other issues.

To illustrate the difference:

- `ss = 0x00`, `sp = 0x7C00`: Stack segment starts at 0x0000, stack pointer points to 0x7C00 (top of stack)
- `ss = 0x7C0`, `sp = 0x00`: Stack segment starts at 0x7C00, stack pointer points to 0x0000 (bottom of stack)

The first scenario is correct, while the second scenario would cause issues with stack management.
```

4. Preparing out bootloader for real hardware.

BIOS Parameter Block (more info)

Some BIOS expect the BIOS Parameter Block. If we don't create it, the BIOS might overwrite the values of our program. #### When booting from a USB, we're doing USB emulation, meaning that we're treating our USB as a hard drive, allowing us to talk to it.

Offset (decimal)	Offset (hex)	Size (in bytes)	Meaning
0	0x00	3	The first three bytes EB 3C 90 disassemble to JMP SHORT 3C NOP. (The 3C value may be different.) The reason for this is to jump over the disk format information (the BPB and EBPB). Since the first sector of the disk is loaded into ram at location 0x0000:0x7c00 and executed, without this jump, the processor would attempt to execute data that isn't code. Even for non-bootable volumes, code matching this pattern (or using the E9 jump opcode) is required to be present by both Windows and OS X. To fulfil this requirement, an infinite loop can be placed here with the bytes EB FE 90.

Offset (decimal)	Offset (hex)	Size (in bytes)	Meaning
3	0x03	8	OEM identifier. The first 8 Bytes (3 - 10) is the version of DOS being used. The next eight Bytes 29 3A 63 7E 2D 49 48 and 43 read out the name of the version. The official FAT Specification from Microsoft says that this field is really meaningless and is ignored by MS FAT Drivers, however it does recommend the value “MSWIN4.1” as some 3rd party drivers supposedly check it and expect it to have that value. Older versions of dos also report MSDOS5.1, linux-formatted floppy will likely to carry “mkdosfs” here, and FreeDOS formatted disks have been observed to have “FRDOS5.1” here. If the string is less than 8 bytes, it is padded with spaces.
11	0x0B	2	The number of Bytes per sector (remember, all numbers are in the little-endian format).
13	0x0D	1	Number of sectors per cluster.
14	0x0E	2	Number of reserved sectors. The boot record sectors are included in this value.
16	0x10	1	Number of File Allocation Tables (FAT’s) on the storage media. Often this value is 2.
17	0x11	2	Number of root directory entries (must be set so that the root directory occupies entire sectors).

Offset (decimal)	Offset (hex)	Size (in bytes)	Meaning
19	0x13	2	The total sectors in the logical volume. If this value is 0, it means there are more than 65535 sectors in the volume, and the actual count is stored in the Large Sector Count entry at 0x20.
21	0x15	1	This Byte indicates the media descriptor type.
22	0x16	2	Number of sectors per FAT. FAT12/FAT16 only.
24	0x18	2	Number of sectors per track.
26	0x1A	2	Number of heads or sides on the storage media.
28	0x1C	4	Number of hidden sectors. (i.e. the LBA of the beginning of the partition.)
32	0x20	4	Large sector count. This field is set if there are more than 65535 sectors in the volume, resulting in a value which does not fit in the *Number of Sectors* entry at 0x13.

For us to create our own fake BPB, we'll need to fill at MOST 33 bytes of data without the first three ones, because those are reserved for `EB 3C 90` or `JMP SHORT 3c NOP`. `3c` is arbitrary and can be our own code, in our case it will be the `start` label.

The parameter block in 8086 Assembly.

```
_start:
    ; when we said that JMP SHORT 3c NOP, that 3c is arbitrary, this is it.
    jmp short start
    nop

times 33 db 0
; the actual BPB.

start:
    jmp 0x7c0:step2

step2:
    ; old code...
```

5. Writting our bootloader into a USB.

```
sudo dd if=./boot.bin of=/dev/sdb # your USB goes in of.
```

6. The Interrupt Vector Table

- Interrupts are similar to subroutines, but we don't need to know the memory address to invoke them.

- Interrupts are called via interrupt numbers rather than memory addresses.
- Interrupts can be setup by the programmer. We could point `0x32` to a piece of our code and if someone else calls `0x32`, they will invoke our code. ## What happens when we invoke an interrupt?
- The processor is interrupted.
- The old state is saved in the stack.
- The interrupt is executed. ## What is the interrupt vector table?
- Table describing all 256 interrupt handlers.
- Interrupts contain 4 bytes (offset:segment).
- Interrupts are in numerical order in the table.

Interrupts, being 4 bytes each, are in the following order: - Interrupt 0: 0x00 - Interrupt 1: 0x04 - Interrupt 2: 0x08 - Interrupt 3: 0x0C …and so on and so forth.

For example, take this IVT:

Offset	Segment	Offset	Segment	Offset	Segment	Offset	Segment
0x00 2 bytes	0x7C0 2 bytes	0x8D00 2 bytes	0x00 2 bytes	0x00 2 bytes	0x8D0 2 bytes	0x7C00 2 bytes	0x587 2 bytes

Each interrupt is four bytes, and the formula to get the physical address (segment * 16 + offset) still applies here.

The processor can also call for interrupts. For example, the Intel processor will call interrupt 0 whenever we divide anything by zero, because the interrupt zero is the interrupt for handling division by zero.

7. Implementing our own interrupts in realmode.

```
handle_zero:
    ; literally just print A when division by zero occurs.
    mov ah, 0eh
    mov al, 'A'
    mov bx, 0x00
    int 0x10
    iret

step2:
    ; old code...
    ; interrupt code now!
    mov word[ss:0x00], handle_zero
    mov word[ss:0x02], 0x7c0 ; its not 0x04 because 0x7c0 is two bytes, finishing at 0x03.
    ; this will put our handle_zero subroutine at interrupt 0.
    mov word[ss:0x04], handle_two  ; now we do use 0x04.
    mov word[ss:0x06], 0x7c0 ; and 0x7c0 finishes at 0x07.
    ; the same, but with interrupt 2.
    ; if you're thinking
```

iret

- When writting interrupts, we don't use the `ret` instruction. We use `iret`, aka ***interrupt return***. ## mov word[ss:0x00], handle_zero
- We're moving our subroutine into 0x00, or the interrupt zero, filling up 0x00 and 0x01. ## mov word[ss:0x02], 0x7c0
- Now we're moving 0x7c0 to 0x02, filling up 0x02 and 0x03, finishing our four byte interrupt.

Exceptions (read more here)

- When dividing by zero, interrupt 0 is called by an exception. So interrupt 0 is an exception.

8. Disk access and how it works.

Files do not actually exist.

- Filesystems are kernel implemented. They are not the responsibility of the disk to handle.

- Implementing a filesystem requires the kernel programmer to create a filesystem driver for the target filesystem. # Data is typically written in sectors.
- Each sector is typically 512 bytes per sector in modern disks.
- CDs, for example, blocksize might be bigger.
- Reading the sector of a disk will return 512 bytes of data for the chosen sector. ## CHS (Cylinder Head Sector)
- Secotors are read and written by specifying a “head”, “track” and “sector”.
- This is an old fashioned and more complex way of reading from a disk drive. ## LBA (Logical Block Address)
- This is the modern way of reading frmo a hard disk, rather than specify “head”, “track” and “sector”, we just specify a number that starts from zero.
- LBA allows us to read from the disk as if we are reading blocks from a very large file.
- LBA 0 and LBA 1 are the first and second sectors of a disk, and together they’re 1024 bytes. ## Calculating LBA
- If we want to read the byte at position 58376 on the disk, how do we do it?
- LBA: `58376 / 512 = 114.015625`
  - Now we have the sector, but we need to also calculate the offset, as we don’t know if it will divide correctly.
  - We load 512 bytes into memory, now we need to calculate the offset.
- LBA Offset: `58,376 % 512 = 8`
- Let’s confirm we’re right.
- `114 * 512 = 58,368`
- `58,368 + 8 = 58,376` ## BIOS Disk routines.
- In 16 bit real mode, the BIOS provides interrupt `13h` for disk related operations.
- In 32 bit protected mode, we have to write our own disk drive, which is a bit more complicated. # 9. Reading from the hard disk.
- So now we’ll read from the hard disk. Not an actual hard disk, but a sector appended to our boot.bin image with dd. Nevertheless, the operation in a real drive is the same. ## The code.

```
; new code. this goes inside the step2 label.
step2:
    ...
        ; new code
        mov ah, 2 ; read sector command
        mov al, 1 ; one sector to be read
        mov ch, 0 ; cylinder low eight bits
        mov cl, 2 ; sector number
        mov dh, 0 ; head number
        mov bx, buffer
        ; we do not need to set dl / drive number, as this is done
        ; automatically by the bios.
        int 0x13
        jc error

        mov si, buffer
        call print
        jmp $

; new!
error:
        mov si, error_message
        call print
        jmp $

print:
        mov bx, 0
.loop:
        lodsb
        cmp al, 0
        je .done
        call print_char
        jmp .loop
.done:
        ret

print_char:
        mov ah, 0eh
        int 0x10
        ret

; new!
error_message: db 'Failed to load sector.', 0

; old code. nothing new.
times 510-($-$$) db 0
dw 0xAA55

; new!
buffer:
```

mov ah, 2 ; read sector command

- We move 2 to `ah`, 2 being is the read sector command. for more info, read this! ## mov al, 1
- We move 1 to `al`, beacuse we'll be reading one sector (as per de documentation linked above). ## mov ch, 0
- This is the low eight bits of the cylinder number. We set it to zero. ## mov cl, 2
- The sector to be read, from 1 to 63. In our case, since the sector 1 is the bootloader itself, we'll read sector 2. ## mov dh, 0
- This is the head number. ## (implicit) mov dl, DRIVENUMBER
- Although we don't set this, the BIOS does. It's for the drive number. ## mov bx, buffer
- We move our buffer (outside the first sector) into `bx`, where the sector 2 will be loaded. ## int 0x13
- We run the load sector in RAM interrupt. ## jc error
- jc (jump carry) is a conditional jump where it will jump to `error` if the `cf` (carry flag) is set. As per the documentation, the `cf` register will be set if any errors are found during the load sector to RAM interrupt. ## mov si, buffer
- We load the buffer into the `si` register. ## error: mov si, error_message
- We load the `error_message` constant into the `si` register to print it out. ## buffer:
- This is a label created outside of the bootloader 512 bytes sector. It's used as a space we can use to print out the message found in the second sector. ![[Pasted image 20250501004456.png]]

1. What is protected mode?

- Processor state in x86 architectures.
- Can provide memory and hardware protection.
- It has different memory schemes.
- It has 4GB of addressable memory. ## Protected mode security rings
- Protected mode allows us to protect memory from being accessed.
- It also prevents user programs to talk directly with the hardware. ![[Pasted image 20250501004752.png]] ## Different memory schemes.
- Selectors (CS, DS, ES, SS), etc.
- Paging (Remapping Memory Addresses).

Selector memory scheme

- Our segmentation registers become selector registers.
- Selectors point to data structures that describe memory ranges and the permissions (ring level) required to access a given range. ![[Pasted image 20250501005054.png]]

Paging Memory Scheme

- Memory is virtual and what you address can point to somehwere entirely different in memory.
- Memory protection is easier to control.
- Paging is the most popular choice for emmory schemes with kernels/operating systems. ![[Pasted image 20250501005143.png]] ### 4GB of addressable memory.
- We gain access to 32 bit instructions and can easily work with 32-bit registers.
- We can address up to 4GB ofg memory at any time and we are no longer limited to the 1MB of memory provided by real mode. # 2. Switching to protected mode. This is how we access protected mode:

```
cli            ; disable interrupts
lgdt [gdtr]    ; load GDT register with start address of Global Descriptor Table
mov eax, cr0
or al, 1       ; set PE (Protection Enable) bit in CR0 (Control Register 0)
mov cr0, eax

; Perform far jump to selector 08h (offset into GDT, pointing at a 32bit PM code segment descriptor)
; to load CS with proper PM32 descriptor)
jmp 08h:PModeMain

PModeMain:
; load DS, ES, FS, GS, SS, ESP
; at this point, we're on protected mode :)
```

But we need to create a GDT or Global Descriptor Table. > Since the following code is quite large, I'll set it up in pieces.

```
ORG 0x7c0
```

- Now we'll begin from 0x7c0 directly again.

```
CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

- These we'll learn what they are in a few moments.

```
start:
    jmp 0:step2
```

- Now instead of setting 0 to 0x7c0, we'll set it to 0, since we're already at origin 0x7c0.

```
.load_protected:
        cli
        lgdt [gdt_descriptor]
        mov eax, cr0
        or eax, 0x1
        mov cr0, eax
        jmp CODE_SEG:load32
```

- The `cli` instruction clears and disables interrupts.
- `mov eax, cr0` moves the value of `eax` into the `cr0` register.
- `or eax, 0x1` we set `eax` to 0x1 to set the PE (Protection Enable) bit in `cr0` (Control Register 0)
- `mov cr0, eax` now we set the PE bit.
- `jmp CODE_SEG:load32` we'll see what this does in a bit. More info on the following GDT code: https://wiki.osdev.org/GDT_Tutorial#Flat_/_Long_Mode_Setup

```
; GDT!
gdt_start:
gdt_null:
        dd 0x0
        dd 0x0

; offset 0x8
gdt_code:               ; cs should point to this.
        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 bit flags and low 4 bit flags.
        db 0            ; base 24-31 bits.

; offset 0x10
gdt_data:               ; linked to DS, SS, ES, FS, GS
        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 bit flags and low 4 bit flags.
        db 0            ; base 24-31 bits.
gdt_end:

gdt_descriptor:
        dw gdt_end - gdt_start-1
        dd gdt_start
```

- `gdt_start:` just setting up the initial label for our GDT.
- `gdt_null:` this label points to null values, which are needed by the CPU to mark the end of the GDT.
  - `dd 0x0`: we're setting up null values.
  - `dd 0x0`: again, we're setting up null values.
- `gdt_code:` we begin the GDT code. The `cs` register should point to this. It will be used to store executable code.
  - `dw 0xffff`: This is the segment limit. It is set to 0xFFFF because the limit is 64KB. Although standard, this is an arbitrary limit.
  - `dw 0`: This is the base first 0-15 bits. For more info: https://wiki.osdev.org/Global_Descriptor_Table#Segment_D
  - `db 0`: This is the base 16-23 bits. For more info: https://wiki.osdev.org/Global_Descriptor_Table#Segment_Desc
  - `db 0x9a`: This is the access byte. It specifies the access rights and properties of the segment.
    - Types available in 32-bit protected mode:
      - **0x1:** 16-bit TSS (Available)
      - **0x2:** LDT
      - **0x3:** 16-bit TSS (Busy)
      - **0x9:** 32-bit TSS (Available)
      - **0xB:** 32-bit TSS (Busy)
    - Types available in Long Mode:

        · **0x2:** LDT
        · **0x9:** 64-bit TSS (Available)
        · **0xB:** 64-bit TSS (Busy)
  - `db 0x11001111b`: This is the flags byte. Read this if you forget about it.
  - `db 0`: This is the base 24-31 bits.
- `gdt_descriptor`: This is the GDT descriptor label.
  - `dw gdt_end - gdt_start-1`: This is the size of the GDT in bytes minus 1, as per CPU requirements.
    * `dd gdt_start`: This is the address of the GDT in memory.

```
[BITS 32]
load32:
  mov ax, DATA_SEG
  mov ds, ax
  mov es, ax
  mov fs, ax
  mov gs, ax
  mov ss, ax
  mov ebp, 0x00200000
  mov esp, ebp
  jmp $
```

- `[BITS 32]`: this sets the assembler to use 32 bit instructions and registers.
- `load32:` This is a label that will be called when we load into protected mode.
  - `mov ax, DATA_SEG`: We move `DATA_SEG` into `ax`. Remember that `DATA_SEG` we saw before? `DATA_SEG equ gdt_data - gdt_start-1`? This is why it's here. It makes the `ds` register point to offset `0x10`.
  - `mov ds, ax`: Now we set all our segments to the `0x10` offset or `DATA_SEG` of our GDT.
  - `mov es, ax`: Same here.
  - `mov fs, ax`: Same here.
  - `mov gs, ax`: Same here.
  - `mov ss, ax`: Same here.
  - `mov ebp, 0x00200000`: Here we're using the 32 bit `ebp` register which is the base pointer. We're setting it to `0x00200000`, which is a 32 bit number.
  - `mov esp, ebp`: Now we're setting the stack pointer to point to the base pointer, initializing our memory.
  - `jmp $`: We do nothing else. Pretty fucking complex right? So fun!!!

![[Sin título 4.jpg]] # 3. Setting up the A20 Line. - Setting up the A20 line is necessary, as it is "**the physical representation of the 21st bit (number 20, counting from 0) of any memory access**". Without it, we won't be able to access to the 21st bit of any memory address.

```
in al, 0x92
or al, 2
out 0x92, al
```

- in al, 0x92: This instruction reads the current state of the A20 line from the SCPA port and stores it into the `al` register.
  - `in`: This reads data from the I/O port.
  - `al`: This is the destination register that will read the data from the I/O port.
  - `0x92`: is the address of the I/O port that controls the A20 line. This port is also known as the System Control Port A or SCPA.
- or al, 2: The `or` operation sets the A20 line enable bit (second bit) in the `al` register to `1`, thus enabling the A20 line. The other bits in the `al` register remain untouched.
  - `or`: Simple bitwise OR operation.
  - `al`: Register that holds the value of the current status of the A20 line.
  - `2`: Decimal value for 00000010, meaning that it has the second bit set, that bit being the enable A20 line.
- out 0x92, al: This lines writes the changes done to the status of the SCPA port, enabling the A20 line.
  - `out`: It's an input instruction that writes data to an I/O port.
  - `0x92`: It's the destination address, meaning that we'll write data to the SCPA port.
  - `al`: It's the OR'd register that contains the previous status of the SCPA port, but now with the 2nd bit set to 1, enabling the A20 line. # 4. Setting up a cross-compiler. Just do it you pussy. Read this..

5. Loading our 32 bit kernel into memory and working with debugging symbols.

This shit got very complex very quick. Let's see it step by step. ## 5.1. Creating an initial kernel. ### 5.1.1. Makefile To create our initial kernel, we'll take the `laod32` code and put it into a different assembly file called `kernel.asm`. Now, we need to setup a real Makefile for it to work and for it to have debugging symbols, so it's easier for us to debug it in GDB.

```
FILES = ./build/kernel.asm.o

all: ./bin/kernel.bin ./bin/boot.bin
    # remove the older os.bin file if it exists.
        rm -rf ./bin/os.bin
        # write the boot.bin into os.bin in pure raw binary.
        dd if=./bin/boot.bin >> ./bin/os.bin
        # write the kernel.bin into os.bin in raw binary, at sector 2.
        dd if=./bin/kernel.bin >> ./bin/os.bin
        # fill up the remaining space. count could be just one, but we
    # use 100 for testing the LBA driver.
        dd if=/dev/zero bs=512 count=100 >> ./bin/os.bin

./bin/kernel.bin: $(FILES)
    # first we link our kernel...
        i686-elf-ld -g -relocatable $(FILES) -o build/kernelfull.o
        # now we build with our linker.ld script and the kernel object we generated before.
        # the -ffreestanding, -O0 and -nostdlib are required.
        i686-elf-gcc -T ./src/linker.ld -o ./bin/kernel.bin -ffreestanding -O0 -nostdlib ./build/kernel

./bin/boot.bin: ./src/boot/boot.asm
    # just building the bootloader as we've done before...
        nasm -f bin ./src/boot/boot.asm -o ./bin/boot.bin

./build/kernel.asm.o: ./src/kernel.asm
    # now we're building the kernel in ELF format.
        nasm -f elf -g ./src/kernel.asm -o ./build/kernel.asm.o

clean:
        rm -rf ./bin/boot.bin
        rm -rf ./bin/os.bin
        rm -rf ./build/kernel.asm.o
```

5.1.2. linker.ld

We need to create a linker file to link out bootloader with our kernel and the kernel with the kernel object:

```
# we define the entrypoint for our program. in our case, it'll be the _start symbol (see kernel.asm)
ENTRY(_start)
# now we tell the linker how to output our file, i.e., a binary format with no headers at all.
OUTPUT_FORMAT(binary)
# we start defining the sections of our program.
SECTIONS
{
    # we tell the linker to load our file at RAM address 1M or 0x100000.
        . = 1M;
        # this section defines a .text section, which contains the code of our program.
        .text :
        {
                *(.text)
        }
        # this section defines .rodata, which contains read only data.
        .rodata :
        {
                *(.rotdata)
        }
        # this section defines .data, which contains initialized data, like global variables.
```

```
	.data :
	{
		*(.data)
	}
	# this section defines .bss, which contains non initialized data, like non initialized
	# global variables.
	.bss :
	{
		*(COMMON)
		*(.bss)
	}
}
```

5.1.3. kernel.asm

The kernel.asm file is the same `load32` routine we had previously. But in this case, we renamed it to __start.

```
[BITS 32]
global _start

CODE_SEG equ 0x08
DATA_SEG equ 0x10

_start:
	; beginning of register setup
	mov ax, DATA_SEG
	mov ds, ax
	mov es, ax
	mov fs, ax
	mov gs, ax
	mov ss, ax
	mov ebp, 0x00200000
	mov esp, ebp
	; end of register setup
	; enabling the A20 line.
	in al, 0x92
	or al, 2
	out 0x92, al
	; end of enabling the A20 line.
	jmp $
```

- `[BITS 32]`: same as `[BITS 16]`, but now we're on 32 bit protected mode.
- `global _start`: we make the `_start` symbol global, so our compiler and linker can see it. Now onto the complex part of the chapter… ### 5.1.4. The LBA driver. The LBA driver is the part of the bootloader that allows us to read from the disk and put stuff into memory. It's an essential part in executing our kernel code. #### 5.1.4.1. .load_protected We still need the `load_protect` function, as we'll be working in protected mode rather than in realmode.

```
; we still have the load32 function in our bootloader.
.load_protected:
	cli
	lgdt [gdt_descriptor]
	mov eax, cr0
	or eax, 0x1
	mov cr0, eax
	jmp CODE_SEG:load32
```

5.1.4.2. New `load32`

```
[BITS 32]
load32:
	mov eax, 1
	mov ecx, 100
	mov edi, 0x0100000
	call ata_lba_read
	jmp CODE_SEG:0x0100000
```

Now, instead of `jmp $` to just be able to sit in 32 bit protected mode, we need to write a driver that allows us to read sectors and write them into memory. Get ready.

There are also a few changes done. For example, now we aren't initializing the data segments of the RAM, or enabling the A20 line. This is because now that's something the kernel will do.

- `mov eax, 1`: In here we're setting up the amount of sectors to be read during the LBA read routine. If we were to read from 0, we'd be reading the bootloader.
- `mov ecx, 100`: This is the amount of sectors to be read.
- `mov edi, 0x0100000`: This is the memory address where our code will jump to later on, which will be the kernel.
- `call ata_lba_read`: This calls the `ata_lba_read` routine.
- `jmp CODE_SEG:0x0100000`: This will execute our kernel.

5.1.4.3. ata_lba_read This is the driver. We'll be looking at it in chunks; we won't need it in kernel space, but I'd like to understand what's going on here.

To actually understand what its being done, we need to know how to calculate the LBA address. Check this formula:

Info on ATA and other I/O ports here.

- `0x1F6`: This is the port address for the ATA drive's control register. Specifically, it's used to select the drive and set the LBA mode.
- `0x1F2`: This is the port address for the ATA drive's sector count register. It's used to specify the number of sectors to read or write.
- `0x1F3`: This is the port address for the ATA drive's LBA low register. It's used to send the lower 8 bits of the LBA address to the drive.
- `0x1F4`: This is the port address for the ATA drive's LBA mid register. It's used to send the middle 8 bits of the LBA address to the drive.
- `0x1F5`: This is the port address for the ATA drive's LBA high register. It's used to send the upper 8 bits of the LBA address to the drive.
- `0x1F7`: This is the port address for the ATA drive's command register. It's used to send commands to the drive, such as "read sector" or "write sector".
- `0x1F0`: This is the port address for the ATA drive's data register. It's used to read or write data from the drive.

  ```
  LBA address = (ecx >> 24) + (ecx >> 16) + (ecx >> 8)
  ```

In the assembly, `ecx` is right shifted three times and sent to the I/O bus

```
ata_lba_read:
        mov ebx, eax ; backup the lba
        ; send the highest 8 bits to the hard disk controller
        shr eax, 24
        or eax, 0xE0 ; selects the master drive
        mov dx, 0x1F6
        out dx, al
        ; finish sending the highest 8 bits of the lba
```

- `ata_lba_read:`: define the `ata_lba_read` routine label.
- `mov ebx, eax`: we backup the value of the LBA into the `ebx` register.
- `shr eax, 24`: following the LBA address formula, we're right shifting `eax`. Although the formula states `ecx`, it doesn't matter as long as we're working with LBA values.
  - By right shifting eax (1) by 24, we're effectively doing `2^24`, which is equivalent to extracting the highest 8 bits of a 32 bit value.
  - Meaning that now, we have the highest 8 bits in our `eax` register.
- `or eax, 0xE0`: selects the master drive by ORing `eax`. if we wanted to select the slave disk, we'd use `0xF0`.
  - This value needs to be ORd and sent to the primary bus.
- `mov dx, 0x1F6`: This will set the `dx` register to `0x1F6`, which will allow us to talk with the `0x1F6` ATA port.
- `out dx, al`: Send the data.

Let's continue with the following section.

```
    ; send the total sectors to read
        mov eax, ecx
        mov dx, 0x1F2
        out dx, al
        ; done sending the sectors to read
```

- `mov eax, ecx`: We set up the total amount of sectors to read, so later we'll send this data to the ATA interface.
- `mov dx, 0x1F2`: Now we set `dx` to `0x1F2` (LBA Sector Count)
- `out dx, al`: This sends the first 8 bits of the `eax` register to the `0x1F2` ATA I/O port. See above for more info.

  ```
  mov eax, ebx ; restoring the lba backup
  mov dx, 0x1F3
  out dx, al
  ```

```
        ; finished sending more bits of the LBA
```

- `mov eax, ebx`: we restore the LBA backup.
- `mov dx, 0x1F3`: we set the I/O interface to `0x1F3` (LBA Low Register)
- `out dx, al`: we send the lower 8 bits of the EAX register to the I/O interface.

```
        mov dx, 0x1F4
        mov eax, ebx ; restoring again the LBA backup
        shr eax, 8
        out dx, al
        ; finished sending even more bits of the LBA
```

- `mov dx, 0x1F4`: we set the I/O interface to `0x1F4` (LBA Mid Register)
- `mov eax, ebx`: we restore the LBA backup yet again
- `shr eax, 8`: we right-shift the value of `eax` by 8 (as per LBA Address formula above)
- `out dx, al`: we send the data to the `0x1F4` I/O interface.

```
        ; send uppter 16 bits
        mov dx, 0x1F5
        mov eax, ebx ; restoring the LBA backup
        shr eax, 16
        out dx, al
```

- `mov dx, 0x1F5`: we set the I/O interface to `0x1F5` (LBA High Register).
- `mov eax, ebx`: we restore the LBA backup.
- `shr eax, 16`: we right-shift the value of `eax` by 16 (as per LBA Address formula above)
- `out dx, al`: we send the data to the `0x1F5` I/O interface.

```
        ; finished sending upper 16 bits of the LBA
        mov dx, 0x1F7
        mov al, 0x20
        out dx, al
```

- `mov dx, 0x1F7`: finally, we set the I/O interface to `0x1F7` (LBA Command Register).
- `mov al, 0x20`: we set the LBA action to read.
- `out dx, al`: we finally send the last bits of data to the `0x1F7` interface.

```
.next_sector:
        push ecx
```

- `.next_sector:`: we define a label that will allow us to decrement the sector value to be read.
- `push ecx`: we push the value of `ecx` (100, the amount of sectors to be read) into the stack.

```
; checking if we need to read
.try_again:
        mov dx, 0x1F7
        in al, dx
        test al, 8
        jz .try_again
```

- `.try_again:`: this label will help us to check if we have to read from disk or not, as sometimes the disk may have a bit of a delay.
- `mov dx, 0x1F7`: again as before, we set the I/O interface to `0x1F7` (LBA Command Register).
- `in al, dx`: we set the action register to read.
- `test al, 8`: we check if the number `8` is in the `al` register. if so, continue; if not…
- `jz .try_again`: try again!

```
; we need to read 256 words at a time
        mov ecx, 256
        mov dx, 0x1F0
        rep insw
        pop ecx
        loop .next_sector
        ; end of reading sectors into memory
        ret
```

- `mov ecx, 256`: we set the amount of words to be read (translate to 512 bytes).
- `mov dx, 0x1F0`: we now set the interface to be read into the `0x1F0` (LBA Data Register)
- `rep insw`: …
  - `rep`: repeats an instruction X amount of times defined in the `e[cx]` register.
  - `insw`: inputs word from I/O port specified in the `dx` register into memory pointer by the `e[di]` register.
    - in our case, `edi` is `0x0100000`. check [[#5.1.4.2. New `load32`]] for a refresher.
  - for more info, read the [[General Assembly Notes]] page.
- `pop ecx`: we now `pop` the value of `ecx` out of the stack.

- `loop .next_sector`: the `loop` instruction compares the value of `e[cx]` and checks if it's zero. If not, it decrements `ecx` by 1 and loops again.
- `ret`: when `ecx` equals zero, the function will return to `load32`.

6. Potential alignment issues.

When linking, our C code might be misaligned. For example: - In a section (.text) we might have Assembly code. But, if we have C code, there's nothing that could assure us that the C code will be interpreted as code, and not as data or rodata. - This might be an issue later on. So to fix it, we'll modify our linker script:

```
ENTRY(_start)
OUTPUT_FORMAT(binary)
SECTIONS
{
	. = 1M;
	.text : ALIGN(4096)
	{
		*(.text)
	}

	.rodata : ALIGN(4096)
	{
		*(.rotdata)
	}

	.data : ALIGN(4096)
	{
		*(.data)
	}

	.bss : ALIGN(4096)
	{
		*(COMMON)
		*(.bss)
	}
	.asm : ALIGN(4096)
	{
		*(.asm)
	}
}
```

We've added the `ALIGN(4096)` sentence after the declaration of the sections and created the new `.asm` section, where we'll put our Assembly code in the future, but NOT the kernel! The kernel will still go in the `.text` section, as it'll be the first thing to be loaded, so we need to execute it as quickly as possible.

7. C Code in Protected Mode.

First of all, we need to make a few changes in our Makefile.

```
FILES = ./build/kernel.asm.o ./build/kernel.o # added kernel.o
INCLUDES= -I./src # added this include line
FLAGS= -g -ffreestanding -falign-jumps -falign-functions -falign-labels -falign-loops -fstrength-reduce -fomit-frame-pointer -finline-functions -Wno-unused-functions -fno-builtin -Werror -Wno-unused-label
# added aaaaaaaall of this GCC flags.
[SNIP]
./bin/kernel.bin: $(FILES)
    ...
	i686-elf-gcc $(FLAGS) -T ./src/linker.ld -o ./bin/kernel.bin -ffreestanding -O0 -nostdlib ./build/kernelfull.o
    # we added the FLAGS to the linker script.
[SNIP]
./build/kernel.o: ./src/kernel.c
	i686-elf-gcc $(INCLUDES) $(FLAGS) -std=gnu99 -c ./src/kernel.c -o ./build/kernel.o
	# we created this gcc command to build the kernel.c file.
```

7.1. GCC Flags

7.1.1.1. Debugging and optimization flags

- `-g`: Generates debugging information, such as symbol tables and line numbers, that can be used by debuggers like GDB.
- `-O0`: Disables all optimizations, which can make the code easier to debug but may result in slower performance. #### 7.1.1.2. Alignment flags
- `-falign-jumps`: Aligns jump targets to a power-of-2 boundary, which can improve performance on some architectures.
- `-falign-functions`: Aligns function entries to a power-of-2 boundary, which can improve performance on some architectures.
- `-falign-labels`: Aligns labels to a power-of-2 boundary, which can improve performance on some architectures.

- `-falign-loops`: Aligns loops to a power-of-2 boundary, which can improve performance on some architectures. #### 7.1.1.3. Code generation flags
- `-ffreestanding`: Tells GCC to generate code that doesn't rely on the standard library or startup code, which is useful for embedded systems or operating system development.
- `-fstrength-reduce`: Enables strength reduction, which is an optimization that replaces expensive operations with cheaper ones.
- `-fomit-frame-pointer`: Omits the frame pointer, which can reduce code size but may make debugging more difficult.
- `-finline-functions`: Inlines functions, which can improve performance by reducing function call overhead. #### 7.1.1.4. Warning and error flags
- `-Wno-unused-functions`: Disables warnings about unused functions.
- `-Werror`: Treats all warnings as errors, which can help catch potential issues early.
- `-Wno-unused-label`: Disables warnings about unused labels.
- `-Wno-cpp`: Disables warnings about C++-specific issues, which is useful when compiling C code.
- `-Wno-unused-parameter`: Disables warnings about unused function parameters.
- `-Wall`: Enables all warnings, which can help catch potential issues early. #### 7.1.1.5. Library and startup flags
- `-nostdlib`: Disables the standard library, which is useful for embedded systems or operating system development.
- `-nostartfiles`: Disables the startup files, which are used to initialize the program.
- `-nodefaultlibs`: Disables the default libraries, which are used to provide common functions like `printf`. #### 7.1.1.6. Include flag `-Iinc`: Adds the `inc` directory to the include path, which allows GCC to find header files in that directory.

7.2. Kernel.c and Kernel.h files

We created two files that will be our kernel C code and the header file. They look like this:

```
/* kernel.c file */
#include "kernel.h"

void kernel_mail()
{

}
```

And the header file

```
/* kernel.h file */
#ifndef KERNEL_H
#define KERNEL_H
#endif

void kernel_main();
```

7.3. Modifications to `kernel.asm`

We modified a couple things in our `kernel.asm` file:

```
# we added this extern directive to call our C code from thd
# assembly code
extern kernel_main


_start:
    [snip]
    # we've also added this call kernel_main, to actually call the C code.
    call kernel_main
```

8. Text mode explained

- Text mode allows us to write ASCII to video memory.
- It supports 16 unique colors.
- There's no need to set individual screen pixels for printing characters. ## 8.1. Text mode allows us to write ASCII to video memory.
- We write ASCII characters into memory starting at address `0xB8000` for color displays.
- Or for monochrome displays at `0xB0000`

- Each ASCII character written to memory has its pixel equivalent outputted to the monitor. ## 8.2. No need to set individual screen pixels.
- In text mode, we can directly output the ASCII value to the screen instead of writting pixel by pixel. ## 8.3. Each character takes up two bytes.
- Byte 0: ASCII character; ‘A’, for example,
- Byte 1: Color code; ‘0x00’ for black, for example. ### 8.3.1. We want to set row 0 column 0 to black `a`.

```
# row 0 column 0 to black A
0xb8000 = 'A'
0xb8001 = '0x00'
# row 0 column 1 to black B
0xb8002 = 'B'
0xb8003 = '0x00'
```

9. Hello World, again!

Now that we know the VGA video memory address, we can start writting stuff to the terminal.

9.1. Simple video_mem write.

```
void kernel_mail()
{
    char* video_mem = (char*)(0xB8000);
    video_mem[0] = 'A';
    video_mem[1] = 1;
}
```

- `char* video_mem = (char*)(0xB8000);`: we create a pointer to the `0xB8000` memory address.
- `video_mem[0] = 'A';` we write A to the first byte and…
- `video_mem[1] = 1;` one to the second byte, as we want the color to be black. For demonstrative purposes, we’ll use values ‘Z’ and 15, meaning a letter Z in white color. ![[Pasted image 20250506124938.png]]

Let’s continue. Now, instead of writting byte by byte, we’ll simply write hexadecimal values.

9.2. Writting bytes to video_mem

```
#include<stdint.h>

void kernel_main()
{
    uint16_t video_mem = (uint16_t)(0xB8000);
    video_mem[0] = 0x5AF;
}
```

- `uint16_t video_mem = (uint16_t)(0xB8000);`: using a `uint16_t` type instead of a `char` will allow us to write two bytes directly into memory instead of just one.
- `video_mem[0] = 0x5AF;`: here we assign the values in hex, similar to what we did before.

But! What the hell is that?! ![[Pasted image 20250506125226.png]] While our code is correct, our bytes are not! We’re working within little endian architectures, meaning that the least significant bytes go first. So it should be `0xF5A`! ## 9.2.1. Fixing endiannes!

```
#include<stdint.h>
// this include will be put at the kernel.h header file.

void kernel_main()
{
    uint16_t video_mem = (uint16_t)(0xB8000);
    video_mem[0] = 0xF5A;
}
```

Here we just reverse the endianness of the `0xF5A` bytes.

![[Pasted image 20250506125604.png]] Now it works! But writting to the VGA memory using hexadecimal characters is kind of a pain in the ass. So, we’ll write a routine that will help us do that for us. ## 9.3. Making our program write bytes for us!

```
void terminal_make_char(char c, char color)
{
    return (color << 8) | c;
}
void kernel_main()
{
```

```
    uint16_t* video_mem = (uint16_t*)(0xB8000);
    video_mem[0] = terminal_print_char('Z', 15);
}
```

- `return (color << 8) | c;`: this simple formula will align the characters and bytes for us. For a more detailed explaination, consult [[Text mode notes]].

![[Pasted image 20250506125923.png]] Yipe!!! Now we're playing! But it looks quite ugly with all the BIOS information and stuff. We'll write a terminal initialization function that will clear the screen for us. ## 9.4. Clearing the screen.

```
#define VGA_WIDTH 80
#define VGA_HEIGHT 20
// both of this define values will be put in the kernel.h file.

uint16_t* video_mem = 0;
void terminal_print_char(char c, char color)
{
    return (color << 8) | c
}

void terminal_initialization() {
    video_mem = (uint16_t)(0xB8000);
    for (y=0; y < VGA_HEIGHT; y++)
    {
        for (x=0; x < VGA_WIDTH; x++)
        {
            video_mem[(y*VGA_WIDTH) + x] = terminal_print_char(' ', 0);
        }
    }
}
void kernel_main()
{
    terminal_initialization();
}
```

- `video_mem = (uint16_t)(0xB8000);`: as before, we assign `video_mem` as a `uint16_t` to be able to access the two bytes we need.
- `for (y=0; y < VGA_HEIGHT; y++)`: we start to go along the 80 columns of our VGA screen.
- `for (x=0; x < VGA_WIDTH; x++)`: and now we begin to go along the 20 rows of our VGA screen.
- `video_mem[(y*VGA_WIDTH) + x] = terminal_print_char(' ', 0);`: this formula helps us calculate the exact index in a one-dimensional matrix. for a more detailed explaination, consult [[Text mode notes]].

![[Pasted image 20250506130649.png]] Yay! The screen has been cleared! :) The expression ((y+VGA_WIDTH)+x) is explained in [[Text mode notes]]. ## 9.5. Creating a `print` function. Now that we have all of this functions created, we can create a new one, a `print` function. But we'll have to write a bit more code before actually having a `print` function.

9.5.1. `strlen` implementation.

We'll need to implement a simple `strlen` function. For that, we can do the following:

```
#include<stddef.h>

size_t strlen(const char* str)
{
    size_t len = 0;
    while(str[len])
    {
        len++;
    }
    return len;
}
```

- `size_t len = 0;`: we initially assign `len` to 0.
- `while(str[len])`: this will iterate over every char in our `str`. and if `str` is empty, it will return `len = 0`, which would be correct.
- `len++;`: if `str[len]` isn't zero, increment `len`.
- `return len;` after finishing reading the `str`, we return `len`. ### 9.5.2. `terminal_putchar` Now, we want to create a function that will directly put the bytes of data that we want into memory. This way, we won't have to refer directly to `video_mem`.

```
void terminal_putchar(int x, int y, char c, char color)
{
    video_mem[(y * VGA_WIDTH) + x] = terminal_make_char(c, color);
}
```

- `video_mem[(y * VGA_WIDTH) + x] = terminal_make_char(c, color);`: this is just a combination of what we've seen before. ### 9.5.3. `terminal_writechar` This function will call `terminal_putchar` and put the bytes into the correct VGA memory address in terms of columns and rows, as to not overlap characters.

```
int terminal_row = 0;
int terminal_col = 0;

void terminal_writechar(char c, char color)
{
    terminal_putchar(terminal_col, terminal_row, c, color);
    terminal_col += 1;
    if (terminal_col >= VGA_WIDTH)
    {
        terminal_row += 1;
        terminal_col = 0;
    }
}
```

- `terminal_putchar(terminal_col, terminal_row, c, color);`: now we call the previously defined function to add a character as `0,0`.
- `terminal_col += 1;`: after writting our character, we increment the `terminal_col`, as to not overlap our characters.
- `if (terminal_col >= VGA_WIDTH)`: now we check if we have to introduce a newline. if `terminal_col` is equal or higher than 80, we…
- `terminal_row += 1;`: go to a new row, and…
- `terminal_col = 0;`: reset the columns.

Now, whenever we want to print a character to the screen, we'll call `terminal_writechar`. But we need to keep abstracting it a bit more… ### 9.5.4. `print` Now we'll write the print function! There are two implementations: the one I made, and the one the instructor made. I'll show both.

```
/* i like this implementation because i did it :) */
void terminal_writestring(const char* str, char color)
{
    size_t pos = 0;
    while(str[pos])
    {
        terminal_writechar((char) (str[pos]), color);
        pos++;
    }
}
```

- `size_t pos = 0;`: we create `pos`, which will we used to iterate over our string.
- `while(str[pos])`: here we iterate over `str[pos]`, and whenever `str` becomes \x00 (null terminator), it will let go.
- `terminal_writechar((char) (c[pos]), color);`: we cast and print our `str` as a single byte.
- `pos++;`: finally we increment `pos`.

```
/* this is the instructor's implementation. i think it's a bit cleaner. */
void print(const char* str)
{
    size_t len = strlen(str);
    for (int i = 0; i < len; i++)
    {
        terminal_writechar(str[i], 15);
    }
}
```

- `size_t len = strlen(str);`: we assign `len` to the return value of `strlen(str)`.
- `for (int i = 0; i < len; i++)`: now we iterate over `len`.
- `terminal_writechar(str[i], 15);`: we write our char.

Now our print function is done. But… what about newlines??? ![[Pasted image 20250507104600.png]] Aw fuck! We need to implement newlines. ### 9.5.5. Implementing newlines. This is a very easy task. We just need to put an `if` statement at out `terminal_writechar` function to check if `c` is `\\n` and what to do then.

```
void terminal_writechar(char c, char color)
{
        if (c == '\\n')
        {
                terminal_row += 1;
                terminal_col = 0;
                return;
        }
        terminal_putchar(terminal_col, terminal_row,  c, color);
        terminal_col +=1;
        if (terminal_col >= VGA_WIDTH)
        {
                terminal_col = 0;
```

```
        terminal_row += 1;
    }
}
```

- `if (c == '\\n')`: we check if `c` is a newline character (\n). if so…
- `terminal_row += 1;`: we increment our row number and…
- `terminal_col = 0;`: reset our columns!
- `return;`: after that, we return to the calling function, as passing to `terminal_putchar` would put a new newline. Lets check it out! ![[Pasted image 20250507104948.png]] *Note: I have a small issue. There's a new character at the beginning of `hello test!`, a blank space. I don't yet know why that is happening.*

10. Interrupt Descriptor Table Explained

In protected mode, the interrupt descriptor table is the equivalent to the interrupt vector table.

- It describes how interrupts are called in protected mode.
  - Similarly to the interrupt vector table, the interrupt descriptor table describes how interrupts are setup in the CPU so that if someone causes an int 5 it will invoke the code for interrupt 5 as decsribed by the interrupt descriptor table.
- It can be mapped anwyhere in memory.
- It's different from the Interrupt Vector Table.

Name	Bit	Known As	Description
Offset	46-63	Offset 16-31	The higher part of the offset to execute.
P	47	Present	This should be set to zero for unused interrupts.
DPL	45-46	Descriptor Privilege Level	The ring level the processor requires to call this interrupt.
S	44	Storage Segment	Should be set to zero for trap gates.
Type	40-43	Gate Type	THe type of gate this interrupt is treated as.
0	32-39	Unused 0-7	Unused bits in this structure.
Selector	16-31	Select 0-15	The selector this interrupt is bounded to, i.e. the kernel code selector.
Offset	0-15	Offset 0-15	The lower part of the offset to execute.

10.1 Example implementation

```
struct idt_desc
{
    uint16_t offset_1; // offset bits 0..15
    uint16_t selector; // a code segment selector in GDT or LDT
    uint8_t zero;      // unused, set to 0
    uint8_t type_attr; // type and attributes
    uint16_t offset_2; // offset bits 16..31

} __attribute__((packed)); // we use this so the compiler doesn't rearrange our bits.
```

10.2 Gate (interrupt) types

Each entry on the Interrupt Descriptor Table is called a gate. There are several types of gates:

Name	Value	Description
80386 32 bit Task Gate	`0x05/0b0101`	Task gates reference TSS descriptors and can assist in multitasking when exceptions occur.

Name	Value	Description
80386 16-bit Interrupt Gate	`0x06/0b0110`	Interrupt gates are to be used for interrupts that we want to invoke ourselves in our code.
80386 16-bit Trap Gate	`0x07/0b0111`	Trap gates are like interrupt gates however they are used for exceptions. They also disable interrupts on entry and re-enable them on an "iret" instruction
80386 32-bit Interrupt Gate	`0x0E/0b1110`	Interrupts gates are to be used for interrupts that we want to invoke ourselves in our code.
80386 32-bit Trap Gate	`0x0F/0b1111`	Trap gates are like interrupt gates however they are used for exceptions. They also disable interrupts on entry and re-enable them on an "iret" instruction.

10.3 Interrupt Descriptor Array.

Interrupt descriptors are stored in an array with index 0 defining interrupt zero `int 0`. Index 1 defines interrupt 1, `int 1` and so on and so forth. ![[Pasted image 20250510221232.png]]

10.4 The IDTR: Interrupt Descriptor Table Register

The IDTR is a structure that points to the Interrupt Descriptor Table. It's very similar to the Global Descriptor Table's `gdt_descriptor` label that we created some time ago.

Name	Bit	Description
Limit	0-15	The length of the interrupt descriptor table minus one.
Base	16-47	The address of the interrupt descriptor table.

10.5 IDTR Example implementation

```
struct idt_desc
{
    uint16_t limit;    // our 16 bit base.
    uint32_t base;     // our 32 bit address for the interrupt descriptor table.
} __attribute__((packed)); // we use this so the compiler doesn't rearrange our bits.
```

10.6 Loading the IDT

```
idt_load:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    lidt [ebx]
    pop ebp
    ret
```

x86 INT instruction

11. Implementing the IDT in our code.

- explain memset.c and memset.h
- explain idt.c, idt.h and idt.asm

11.1 memory.c and memory.h

Before actually implementing our IDT, we need a way to zero out the descriptor.

```
// memory.h
#ifndef MEMORY_H
#define MEMORY_H
#include<stddef.h>

void memset(void* ptr, int c, size_t size);

#endif
```

- header guard.
  - `#ifndef MEMORY_H`: the preprocessor checks if the `MEMORY_H` constant is defined. if it is, it skips the header file. but if not…
  - `#define MEMORY_H`: we define `MEMORY_H` and read the rest of the header file.
  - header guard: header guard is what is done in this code by using the `ifndef` and `endif` macros. it makes sure that the header file is included just one time in our code; if, for example, `memory.h` was used in other place apart from `memory.c` and there was no header guard, the preprocessor would load the header the amount of times it is used in the code. the header guard prevents that and only allows it to be loaded once.
- `#include<stddef.h>`: standard def, for `size_t` and other types used in `memory.c`.
- `void memset(void* ptr, int c, size_t size);`: we define the prototype for our `memset` function.
- `#endif`: we end our header file.

```
// memory.c
#include "memory.h"

void* memset(void* ptr, int c, size_t size)
{
        char* c_ptr = (char*) ptr;
        for (int i = 0; i < size; i++)
        {
                c_ptr[i] = (char) c;
        }
        return ptr;
}
```

- `#include "memory.h"`: we include our header file.
- `void* memset(void* ptr, int c, size_t size)`: we start to work on the `memset` function that will take a void pointer to the descriptor table, what will be put there in integer form and the size of the descriptor.
- `char* c_ptr = (char*) ptr;`: we create a pointer to the given pointer to be able to write to it. this is because we received a void pointer, and to be able to write to such a pointer, we need to define its type. in this case, we're casting it into our `char* c_ptr` pointer.
- `for (int i = 0; i < size; i++)`: we initialize a `for` loop that will set the memory to whatever we passed as `c`.
- `c_ptr[i] = (char) c;`: we cast the `c` argument and put it in the `i` position in memory of `c_ptr`, which points to `ptr`.
- `return ptr;`: at the end, we return the `ptr` passed and set to the given `c`.

11.2 config.h

Before explaining our IDT related code, we need to talk about the newly created `config.h` file.

```
// config.h
#ifndef CONFIG_H
#define CONFIG_H

#define KERNEL_CODE_SELECTOR 0x08
#define KERNEL_DATA_SELECTOR 0x10

#define PEACHEOS_TOTAL_INTERRUPTS 512

#endif
```

- `#ifndef CONFIG_H`: header guard, standard stuff.
- `#define CONFIG_H`: header guard, standard stuff.
- `#define KERNEL_CODE_SELECTOR 0x08`: here we set `KERNEL_CODE_SELECTOR` to `0x08`, which is our `CODE_SEG` in the GDT.
- `#define KERNEL_DATA_SELECTOR 0x10`: here we set `KERNEL_DATA_SELECTOR` to `0x10`, which is our `DATA_SEG` in the GDT.
- `#define PEACHEOS_TOTAL_INTERRUPTS 512`: here we set the amount of interrupts that we'll have.
- `#endif`: we end our header file.

11.3 idt.c and idt.h

Now, we'll write the functions that create the prototype structures for our IDT and the C code that actually uses them.

```c
// idt.h
#ifndef IDT_H
#define IDT_H

#include <stdint.h>
struct idt_desc
{
        uint16_t offset_1; // offset bits 0 - 15
        uint16_t selector; // selector in our GDT
        uint8_t zero;      // does nothing; bits are reserved.
        uint8_t type_attr; // descriptor type and attributes.
        uint16_t offset_2; // offset bits 16-31

} __attribute__((packed));

struct idtr_desc
{
        uint16_t limit;    // size of the descriptor table - 1
        uint32_t base;     // base address of the start of the interrupt table.
} __attribute__((packed));

#endif
```

- `#ifndef IDT_H`: we define our header guard
- `#define IDT_H`: we set define the `IDT_H` constant, so the header isn't loaded two times.
- `#include <stdint.h>`: we include `stdint` for our types.
- `struct idt_desc`: we create our Interrupt Descriptor Table.
- `uint16_t offset_1; // offset bits 0 - 15`:
- `uint16_t selector; // selector in our GDT`
- `uint8_t zero;      // does nothing; bits are reserved.`
- `uint8_t type_attr; // descriptor type and attributes.`
- `uint16_t offset_2; // offset bits 16-31`
- `struct idtr_desc`: now we create the Interrupt Descriptor Table Register.
- `uint16_t limit;    // size of the descriptor table - 1`:
- `uint32_t base;     // base address of the start of the interrupt table.`:
- `#endif`: end the header file.

```c
// idt.c
#include "idt.h"
#include "config.h"
#include "memory.h"
#include "kernel.h"

extern void idt_load(struct idtr_desc* ptr);

struct idt_desc idt_descriptors[PEACHOS_TOTAL_INTERRUPTS];
struct idtr_desc idtr_descriptor;

void idt_zero()
{
        print("divide by zero error\\n");
}

void idt_set(int interrupt_no, void* addr)
{
        struct idt_desc* desc = &idt_descriptors[interrupt_no];
        desc->offset_1 = (uint32_t) address & 0x0000ffff;
        desc->selector = KERNEL_CODE_SELECTOR;
        desc->zero = 0x00;
        desc->type_attr = 0xEE;
        desc->offset_2 = (uint32_t) addr >> 16;
}

void idt_init()
{
        memset(idt_descriptors, 0, sizeof(idt_descriptors));
        idtr_descriptor.limit = sizeof(idt_descriptors) - 1;
        idtr_descriptor.base = idt_descriptors;
        idt_set(0, idt_zero);

        // load interrupt descriptor table
        idt_load(idtr_descriptor);
}
```

- `#include "idt.h"`: we include our header file.
- `#include "config.h"`: we include our config file.
- `#include "memory.h"`: we include our `memory.h` header file.
- `extern void idt_load(struct idtr_desc* ptr);`: now we add the externally created `idt_load` function, more on that later.
- `struct idt_desc idt_descriptors[PEACHOS_TOTAL_INTERRUPTS];`: now we create `idt_descriptors` of `idt_desc` type with [PEACHOS_TOTAL_INTERRUPTS] amount of interrupts. this is our IDT.
- `struct idtr_desc idtr_descriptor;`: now we create `idtr_descriptor` of type `idtr_desc`, which will function as our IDTR.
- `void idt_zero()`: here we define the code that will be our interrupt 0.

- `print("divide by zero error\\n");`: since Intel uses `0` for the divide by zero trap, we set it here.
- `void idt_set(int interrupt_no, void* address)`: this function will help us define our interrupts.
- `struct idt_desc* desc = &idt_descriptors[interrupt_no];`: we create a pointer structure to the memory address of `idt_descriptors` at the `interrupt_no` place.
- `desc->offset_1 = (uint32_t) address & 0x0000ffff;`: here we set our `offset_1` to the lower port of `addr`.
- `desc->selector = KERNEL_CODE_SELECTOR;`: here we set out selector, which is `0x08`.
- `desc->zero = 0x00;`: here we set the bits that need to be zero to zero.
- `desc->type_attr = 0xEE;`: here we define our attributes. in this case, `0xEE` is `11101110`. read [[Interrupt Descriptor Table Related Notes]] for more info on this.
- `desc->offset_2 = (uint32_t) address >> 16;`: here we set the offset to the higher part of the of `addr`.
- `void idt_init()`: here we start our `idt_init` function.
- `memset(idt_descriptors, 0, sizeof(idt_descriptors));`: here we zero out the IDT. even though it should be zero, we need to make sure that it is, indeed, zero.
- `idtr_descriptor.limit = sizeof(idt_descriptors) - 1;`: here we set the IDTR limit minus one. more on [[Interrupt Descriptor Table Related Notes]]
- `idtr_descriptor.base = (uint32_t) idt_descriptors;`: and here we set its base which is casted to `uint32_t`.
- `idt_set(0, idt_zero);`: now we setup our divide by zero interrupt.
- `idt_load(&idtr_descriptor)`: and here we call our `idt_load` function, which we'll see in a couple seconds. ## 11.4. idt.asm Now, we'll create an Assembly routine that loads the IDT into the CPU. We cannot do this in C, so we **need** to make it in Assembly.

```
; idt.asm
section .asm

global idt_load
idt_load:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    lidt [ebx]
    pop ebp
    ret
```

- `section .asm`: we need to add this Assembly to our previously defined `.asm` section. see the linker.ld file.
- `global idt_load`: here we make our `idt_load` function global. this will allow our C code to call it.
- `idt_load:`: here we start our `idt_load` function.
- `push ebp`: standard stuff: we push the base pointer to the stack to have it at hand.
- `mov ebp, esp`: here we move the base pointer to the stack pointer.
- `mov ebx, [ebp+8]`: here magic happens. `[ebp+8]` is a given argument. this means that `ebx` will hold our `idtr_descriptor` structure defined in our C code.
- `lidt [ebx]`: here we call `lidt` and load the Interrupt Descriptor Table.
- `pop ebp`: we pop `ebp`.
- `ret`: and we return.

now our IDT is loaded :) although I had some doubts on why we had to call `ebp+8`, so I asked Ollama:

```
In x86 assembly language, when you use the `push ebp` and `mov ebp, esp` instructions to set up a stack frame, the `ebp` register (also known as the base pointer) points to the base of the current stack f

When you call a function in C, the following things happen:

1. The caller pushes the return address onto the stack.
2. The caller pushes any arguments onto the stack, in reverse order (i.e., the last argument is pushed first).
3. The caller calls the function using the `call` instruction, which pushes the current instruction pointer onto the stack.
4. The callee (the function being called) sets up its stack frame by pushing `ebp` onto the stack and moving `esp` into `ebp`.

So, when you look at the stack frame, it looks like this:

- `ebp` (base pointer) points to the base of the stack frame
- `ebp + 0` points to the saved `ebp` value (i.e., the `ebp` value of the caller)
- `ebp + 4` points to the return address
- `ebp + 8` points to the first argument passed to the function

In your case, the `idt_load` function takes a single argument, which is the address of the IDT. This argument is passed on the stack, and its address is `ebp + 8`.

The reason you see `[ebp+8]` in the assembly code is that the compiler is using the `ebp` register as a base pointer to access the argument that was passed on the stack. By adding 8 to `ebp`, you get the

Here's a breakdown of what's on the stack:

- `ebp` (base pointer): points to the base of the stack frame
```

```
- `ebp + 0`: saved `ebp` value (i.e., the `ebp` value of the caller)
- `ebp + 4`: return address
- `ebp + 8`: first argument (address of the IDT)

So, when you do `mov ebx, [ebp+8]`, you're loading the address of the IDT into the `ebx` register, which is then used to load the IDT using the `lidt` instruction.
```

11.5 Makefile changes.

before compiling, we need to make some changes to our Makefile:

```
# Makefile
FILES = ... ./build/idt/idt.asm.o ./build/idt/idt.o ./build/memory/memory.o

./build/idt/idt.asm.o: ./src/idt/idt.asm
	nasm -f elf -g ./src/idt/idt.asm -o ./build/idt/idt.asm.o

./build/idt/idt.o: ./src/idt/idt.c
	i686-elf-gcc $(INCLUDES) -I./src/idt $(FLAGS) -std=gnu99 -c ./src/idt/idt.c -o ./build/idt/idt.o

./build/memory/memory.o: ./src/memory/memory.c
	i686-elf-gcc $(INCLUDES) -I./src/memory $(FLAGS) -std=gnu99 -c ./src/memory/memory.c -o ./build/memory/memory.o
```

And now we compile and run!

![[Pasted image 20250511022620.png]]

Interrupt 0!

12. Implementing `in` and `out` instructions.

So, we know that the x86 architecture has this two instructions `in` and `out` for reading and outputting data to IO ports. But we'll want to use them in our C code, so we'll implement them.

12.1 io.h and io.asm

```
// io.h
#ifndef IO_H
#define IO_H

unsigned char insb(unsigned char port);
unsigned char insw(unsigned short port);

void outb(unsigned char port, unsigned char val);
void outw(unsigned short port, unsigned short val);

#endif
```

- `#ifndef IO_H`: standard header guard stuff.
- `#define IO_H`: standard header guard stuff.
- `unsigned char insb(unsigned char port);`: here we define the prototype for our C `insb` call, which will get us 1 byte from `port`.
- `unsigned short insw(unsigned short port);`: here we define the prototype for our C `insw` call, which will get us 2 byte from `port`.
- `void outb(unsigned char port, unsigned char val);`: here we define the prototype for our C `outb` call, which will send 2 bytes to `port`.
- `void outw(unsigned short port, unsigned short val);`: here we define the prototype for our C `outw` call, which will send 2 bytes to `port`.
- `#endif`: end of our header guard.

```
; io.asm
section .asm

global insb
global insw
global outb
global outw

insb:
	push ebp
	mov ebp, esp
	xor eax, eax ; xor eax, since we'll use it to return the io byte
	mov edx, [ebp+8]
	in al, dx
	pop ebp
	ret

insw:
	push ebp
	mov ebp, esp
	xor eax, eax
	mov edx, [ebp+8]
```

```
        in al, dx
        pop ebp
        ret

outb:
        push ebp
        mov ebp, esp
        mov eax, [ebp+12]
        mov edx, [ebp+8]
        out dx, al
        pop ebp
        ret

outw:
        push ebp
        mov ebp, esp
        mov eax, [ebp+12]
        mov edx, [ebp+8]
        out dx, al
        pop ebp
        ret
```

I will not explain the four function, just one for each.

- `insb:`: we define our input from port function.
- `push ebp`: stack setup.
- `mov ebp, esp`: stack setup.
- `xor eax, eax ; xor eax, since we'll use it to return the io byte`: here we zero out the `eax` register, which is used to return the value of the port.
- `mov edx, [ebp+8]`: here we move the `port` argument into the `edx` register. the lower 8 bits will have our argument, in this case `port`.
- `in al, dx`: here we make the `in` call, and receive the input from the port in the `al` register, or the lower eight bits of the `eax` register.
- `pop ebp`: here we pop the base pointer.
- `ret`: now we return to the calling function.
- `outb:`: here we define our output to port function.
- `push ebp`: stack setup.
- `mov ebp, esp`: stack setup.
- `mov eax, [ebp+12]`: here we move the value to be written `val` into the `eax` register.
- `mov edx, [ebp+8]`: and here we take the first argument `port` and set it to the `edx` register.
- `out dx, al`: here we make the call to `out`, and send the byte to the port.
- `pop ebp`: popping our base pointer.
- `ret`: we return to the calling function.

12.2 Makefile changes

We had to modify our Makefile once again.

```
FILES = ... ./build/io/io.asm.o
./build/io/io.asm.o: ./src/io/io.asm
        nasm -f elf -g ./src/io/io.asm -o ./build/io/io.asm.o
```

13. Programmable Interrupt Controller

- It allows hardware to interrupt the processor state, such as the keyboard, hard disk, mouse and more.
- It's programmable (duh).
- It requires interrupt acknowledgment.

13.1 IRQs

- IRQs are mapped to a starting interrupt, for example 0x20.
- IRQ 0 would then be interrupt 0x20.
- IRQ 1 would then be interrupt 0x21.
- IRQ 2 would then be interrupt 0x22. By default, some of the IRQs are mapped to interrupts 8-15. This is a problem as these interrupts are reserved in protected mode for exceptions, so we are required to remap the PIC. ## 13.2 Master vs Slave
- The system has two IRQs: one for master ports and one for slave ports.
- The master handles IRQ 0-7.
- The slave handles IRQ 8-15. ## 13.3 Ports
- `0x20` and `0x21` = Master IRQs.
- `0xA0` and `0xA1` = Slave IRQs.

13.4 Remapping the Master PIC

```
; we havent set this code up, we're understanding the PIC, not implementing it!
setup_pic:
    ; init some flags in the PICs
    mov al, 00010001b ; b4=1: init; b3=0: Edge; b1=0: Cascade; b0=1: Need 4th init setp
    out 0x20, al ; tell master

    mov al, 0x20 ; master IRQ should be on INT 0x20 (Just after Intel exceptions)
    mov 0x21, al

    mov al, 00000001b ; b4 = 0: FNM; b3-2=00: Master/Slave set by hardware; b1=0: Not AEOI; b0=1: x86 mode.
    out 0x20, al
```

13.5 Interrupt Acknowledgement

We must acknowledge the IRQ; otherwise, it will not interrupt us again.

```
// we havent set this code up, we're understanding the PIC, not implementing it!
// we acknoledge the PIC by sending it an EOI (End of Interrupt) command.
outb(PIC1, PIC_EOI);
// this table might be handy later:
// ISR Definitions
#define PIC1 0x20           // IO base address for master PIC.
#define PIC2 0xA0           // IO base address for slave PIC.
#define PIC1_COMMAND PIC1
#define PIC1_DATA (PIC1+1)
#define PIC2_DATA PIC2
#define PIC2_DATA (PIC2+1)
#define PIC_EOI 0x20        // end of interrupt command.
```

14. Programmable Interrupt Controller Implementation

14.1 Remapping the PIC in `kernel.asm`

```
...
    mov al, 00010001b
    out 0x20, al
    mov al, 0x20
    out 0x21, al
    mov al, 00000000b
    out 0x21, al
...
```

- `mov al, 00010001b`: we set the bits for the PIC initialization mode.
- `out 0x20, al`: here we tell the master to set itself to init mode.
- `mov al, 0x20`: here we pass 0x20 to the `al` register.
- `out 0x21, al`: here we tell the master.
- `mov al, 00000000b`: here we enable 32 bit protected mode in the PIC.
- `out 0x21, al`: and we tell the master.
- explain changes to idt.c # 1. What is the heap?
- The heap is a giant memory regino that cfan be shared in a controlled manner.

- You can ask the heap for emmory and tell the heap when you're done with that memory.
- Heap implementations are essentially system memory managers.

1.0 Memory in the C Programming Language

- In C, we can point to any address in the RAM memory, regardless if we can access it or not.
- As of right now, if we were to use this C code, it would crash.

```
int main (int argc, char** argv)
{
    char* ptr = (char*)(0x100000);
    ptr[0] = 'A';
}
```

- But in our kernel, it would not, as there are no restrictions set in place yet. ## 1.1. Malloc in C
- Malloc returns a memory address that we can write to (it becomes ours).
- Ensures that any other time our program calls "malloc" it doesn't return a memory address that is unavailable.
- This code would run.

```
int main (int argc, char** argv)
{
    char* ptr = (char*) malloc(50);
    ptr[0] = 'A';
    char* ptr2 = (char*) malloc(50);
    ptr2[0] = 'A';
}
```

1.2. Free in C

- Similar to `malloc`, `free` allows us to free the memory in the heap that we previously asked for.
- After being freed, the system marks the memory as available to other processes.
- Next time `malloc` is called, we can safely end up with a previous address that was used.
- This code would run.

```
int main (int argc, char** argv)
{
    char* ptr = (char*) malloc(50);
    free(ptr);
}
```

1.3. Limits in a 32-bit protected mode kernel.

- While in protected mode we have certain restrictions, as the processor is in a 32-bit state.
- As we run ni 32 bit mode, we have access only to 32 bit memory addresses, allowing us to addres to a maximum of 4.29G of 4294967296 bytes of RAM regardless of how much system RAM is installed. # 1.4 Memory of an uninitialized system
- Video memory takes up portions of the RAM.
- Hardware memory takes up portions of the RAM.
- Unused parts of RAM are available to use.
- An array of uninitialized memory is available to us from address "`0x01000000`" which can be a lot or too little, depending on the installed memory.

```
Address `0xC0000000` is reserved. This means that the memory array we have at address `0x01000000` can g
```

1.5 Then, the heap is...

- Pointed to an address unused by hardware that is also big enough for us to use.
- THe heap data size can be defined for example, to 100MB of heap memory.
- So long as we have 100MB of memory available, our heap will work fine.
- We'll need a heap implementation to make our haep work properly.
- The heap will be responsible for storing information in our kernel.
- The heap implementation will be responsible for managing this giant chunk of memory that we call the heap.

1.6 Simplest possible heap implementation

- Start with a start address and call it a “current address” and point it to somewhere free i.e., `0x01000000`.
- Any call to malloc gets the current address, stores it in a temporary variable called `tmp`.
- Now the current address is incremented by the size provided to `malloc`.
- Temporary variable called `tmp` that contains the allocated address is returned.
- `current_address` now contains the next address for `malloc` to return when `malloc` is called again.
- **Benefits**:
  - Easy to implement.
- **Cons**:
  - Memory can never be released, which may eventually lead to the system being unusable and requiring a reset.
- This is its implementation:

```
void* current-address = (void*)(0x01000000);
void* malloc(int size)
{
    void* tmp = current_address;
    current_address += size;
    return tmp;
}

void* free(void* ptr)
{
    // we cannot free the memory due to its design.
}
```

1.7 Our heap implementation

- Will consist of a giant table which describes a giant piece of free memory in the system. This atble will describe which memory is taken, which memory is free and so on. **We will call this the “entry table”**.
- Will have another pointer to a giant piece of free memory, this will be the actual heap data itself that users of `malloc` can use. **We will call this the “data pool”**. If our heap can allocate 100 MB of RAM, then the heap data pool will be of 100 MB in size.
- Our heap implementation will be block based, each address returned from `malloc` will be aligned to 4096 and will at least be 4096 in size.
- If you requested to have `50` bytes of memory, 4096 bytes of memory will be returned to you.

1.8 The entry table

- Its composed of an array of 1 byte values that represent an entry in our heap data pool.
- Array size is calculated by taking the heap data pool size and dividing it by our block size of 4096 bytes. We are left with the total number fo entries we need in our array.
- For example:
  - We want a 100MB heap then the math would be:
    - 100MB / 4096 = 25600 bytes in our entry table.
  - If our heap data pool is at address `0x01000000` then entry zero in our table will represent address `0x01000000`.
  - Entry one will represent address `0x01001000`.
  - Entry two will represent address `0x01002000`.
  - `0x1000`: 4096 bytes. ## 1.9 The entry structure Upper 4 bits are flags. Lower 4 bits represent the entry type.

HAS_N	IS_FIRST	0	0	ET_3	ET_2	ET_1	ET_0

- `HAS_N`: Set if the entry to the right of us is part of our allocation.
  - If we allocate 2 blocks of memory, than entry 0 would have the `HAS_N` bit set, and block 1 would not.

- `IS_FIRST`: Set if this is the first entry of our allocation.
- 00: Unused.
- **Each entry byte describes 4096 bytes of data in the heap data pool**.

1.10 Entry types

- `HEAP_BLOCK_ENTRY_TAKEN`: The entry is taken and the address cannot be used.
- `HEAP_BLOCK_ENTRY_FREE`: The entry is free and may be used.

1.11 Data pool

- It's simply a raw flat array of thousands of millions of bytes that our heap implementation can give to people who need memory.

1.12 Malloc example

- First we assume our heap data pool to point to address `0x01000000` .
- We assume our heap is 100MB in size.
- We assume we have 25600 entries in our entry table that describe our 100MB of data in the data pool.
  - 100MB / 4096 = 25600. ### 1.12.1 Memory allocation process
- Take the size from `malloc` and calcyulate how many blocks we need to allocate for this size. If the user asks for "5000" bytes we will need to allocate 8192 bytes because our implementation works on 4096 byte blocks. 8192 is two blocks.
- Check the entry table for the first entry we can find that has a type of `HEAP_BLOCK_TABLE_ENTRY_FREE`, meaning that the 4096 block that this entry represents is free for use.
- Since we require two blocks, we also need to ensure the next entry is also free for use, otherwise we will need to discard the first block we found and look further in our table until we find at least two free blocks that are next to each other.
- Once we have two blocks, we mark those blocks as taken `HEAP_BLOCK_ENTRY_TAKEN`.
- We now reutnr the absolute address that the starting block represents. Calculation:
  - (heap_data_pool_start_address + (block_number * block_size)) ### 1.12.2 Finding the total blocks
- Block size: 4096.
- Get the size priovided to `malloc`. For example, value "5000" we then align it to "4096" and we get the value 8192.

```
if (( 5000 % 4096 ) == 0)
{
    return 5000;
}
uint32_t new_val = 5000 - (5000 % 4096);
new_val += 4096;

return new_val;
```

- Now we divide 8192 by our block size of "4096" which gives us `8192 / 4096 = 2`. **Two blocks to allocate**.

1.12.3 Finding the two free blocks in the table.

0xC1	0x81	0x81	0x01	0xC1	0x01	0x41	0x41	0x41	0x41
0x41	0x41	0xC1	0x81	0x81	0x81	0x81	0x81	0x81	0x81
0x81	0x01	0x00	0x00	0x41	0xC1	0x01	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

- `0xC1 | 1100 0001`: Block taken, first block, has more blocks for this allocation.
- `0x41 | 0100 0001`: Block taken, first block, no more blocks for this allocation.

- `0x81 | 1000 0001`: Block taken and we have more blocks for this allocation, we are not the first block for this allocation.
- `0x01 | 0000 0001`: Block taken, we are not the first block for this allocation, no more flobkc for this allocation.
- `0x00 | 0000 0000`: Block free or (lower 4 bits = 0) block free.

We can see in the third row and forth and fifth column that we have two unallocated blocks that we can use. After allocating them, they'll become `0xC1` and `0x01`.

1.12.4 Calculating the absolute address for the programmer to use.

The first index that we can use in our heap entry table is the 22 index of our array. The mathematical formula to get the offset is:

`(array_iudex * BLOCK_SIZE)`. In this case:

`(22 * 4096) = 90122` decimal. This is our offset. If our heap data pool starts at address `0x01000000`, we add the offset. `initial_address + offset = absolute_address`. So: `0x01000000 + 90122d` = `0x101600A` 90122 = 0x1600A

So the absolute address that `malloc` has to return is `0x101600A` or 16,867,338 in decimal. Now the programmer can safely write his 5000 bytes to the address `0x101600A`.

```
Technically, the programmer will be able to write up to 8192 bytes before overflowing to other people's
```

1.13. Free example.

- Calculate the block number based on the address provided for us to free.
- Go through the entry table starting at the block number we have calcualted, set each entry to `0x00` until we reach the last block of the allocation.
- **We know how many blocks we need to free because the current blockw e are freeing wil now have the `HAS_N` bit set in the entry byte**.

1.14. Advantages of our implementation.

- Fast allocation of memory blocks.
- Fast to free blocks of memory.
- Can be written in under 200 lines of code (easy to implement).

1.15. Disadvantages of our implementation.

- We allocate in memory blocks, meaning misaligned sizes requested from our heap will result in wasted lost bytes.
- Memory fragmentation is possible.

2. Implementing our Heap

this chapter was really long and intense. lots of code was written, so documenting it all will take me a few days. nevertheless, a proper heap, `kmalloc` and `kfree` was implemented!

2.1. changes to config.h

we had to add some values to the `config.h` file. these are mostly related to heap size, RAM block size and the heap table address.

```
// 100MB heap size, 1024*1024*100
#define PEACHOS_HEAP_SIZE_BYTES         104857600
// block size
#define PEACHOS_HEAP_BLOCK_SIZE         4096
// heap starting memory address
```

```
#define PEACHOS_HEAP_ADDRESS          0x01000000
// table address
#define PEACHOS_HEAP_TABLE_ADDRESS    0x00007E00
```

- `#define PEACHOS_HEAP_SIZE_BYTES          104857600`: this is the total heap size in bytes. this was calculated using the formula (1024*1024*\MEGS).
- `#define PEACHOS_HEAP_BLOCK_SIZE          4096`: here we define an arbitrary block size. it can be anything, but in our case, we're using 4 kilobyte blocks.
- `#define PEACHOS_HEAP_ADDRESS          0x01000000`: this address is where the heap begins. check [[Heap and memory alloc related notes]] for more info.
- `#define PEACHOS_HEAP_TABLE_ADDRESS          0x00007E00`: this address is where the heap table resides. check [[Heap and memory alloc related notes]] for more info. ## 2.2. status.h we created a `status.h` file that contains several status codes for exiting and error reporting. I won't explain the header guard, as we've explained it several times before.

```
#ifndef STATUS_H
#define STATUS_H

#define PEACHOS_ALLOK 0
#define EIO 1
#define EINVARG 2
#define ENOMEM 3

#endif
```

- `#define PEACHOS_ALLOK 0`: this is the correct exit value. if return is 0, all is good.
- `#define EIO 1`: this is an error related to input/output issues.
- `#define EINVARG 2`: this is an error related to invalid arguments, like invalid alignment bytes.
- `#define ENOMEM 3`: this is an error returned when during heap allocation, no usable memory is found.

2.3. explain heap.c, heap.h

this is the meat of the code. here's where our heap implementation resides, and it is quite extensive. I will map this later on, and when the map its done, i'll put it here; for now, let's see the code piece by piece.

![[Heap structure of execution.canvas|Heap structure of execution]]

2.3.1 heap.h

```
#ifndef HEAP_H
#define HEAP_H
#include "config.h"
#include <stdint.h>
#include <stddef.h>

#define HEAP_BLOCK_TABLE_ENTRY_TAKEN    0x01
#define HEAP_BLOCK_TABLE_ENTRY_FREE     0x00

#define HEAP_BLOCK_HAS_NEXT             0b10000000
#define HEAP_BLOCK_IS_FIRST             0b01000000

typedef unsigned char HEAP_BLOCK_TABLE_ENTRY;

struct heap_table
{
        HEAP_BLOCK_TABLE_ENTRY* entries;
        size_t total;
};

struct heap
{
        struct heap_table* table;
        // start address of the heap data pool
        void* saddr;
};

int heap_create(struct heap* heap, void* ptr, void* end, struct heap_table* table);
void* heap_malloc(struct heap* heap, size_t size);
void heap_free(struct heap* heap, void* ptr);

#endif
```

- `#define HEAP_BLOCK_TABLE_ENTRY_TAKEN          0x01`: this it the taken flag (0001) for heap entries that are taken.
- `#define HEAP_BLOCK_TABLE_ENTRY_FREE          0x00`: this is the free flag (0000) for heap entries that are free to use.
- `#define HEAP_BLOCK_HAS_NEXT          0b10000000`: this is the memory flag for heap entries that have one or more blocks in use after the one that's being accessed.

- `#define HEAP_BLOCK_IS_FIRST           0b01000000`: this is the memory flag for heap entries that are the first in their memory allocation.
- `typedef unsigned char HEAP_BLOCK_TABLE_ENTRY;`: here we define a type of `HEAP_BLOCK_TABLE_ENTRY`, which we'll use to define the heap memory entry allocation table. it's and 8 bit value.
- `struct heap_table`: here we define our `heap_table`, which we'll use to describe the heap in use and free.
- `HEAP_BLOCK_TABLE_ENTRY* entries;`: here the entries are defined.
- `size_t total;`: and here is the size of each entry.
- `struct heap`: here we define our `heap`, literal heap.
- `struct heap_table* table;`: here, when we create a `heap`, we point to the `heap_table` structure as a descriptor of the heap itself.
- `void* saddr;`: here we give the `heap` its starting address. this value will be incremented and decremented depending on heap usage.
- `int heap_create(struct heap* heap, void* ptr, void* end, struct heap_table* table);`: this interface is exposed to allow the kernel the ability to initialize the heap and be able to expose other endpoints, such as `kmalloc` and `kfree` in the future.
- `void* heap_malloc(struct heap* heap, size_t size);`: this function is used to allocate memory, we'll see it later on.
- `void heap_free(struct heap* heap, void* ptr);`: and this function is used to free those memory blocks.

2.3.2. heap.c

I will not explain the `heap.c` file from top to bottom, but rather by order of execution. First, let's see how the heap is created. the map I made (and that is shown above) will be very helpful in our process to understand how the heap is created and how it works.

2.3.2.1. Heap creation. the heap creation process goes from: - heap_create, - heap_validate_alignment, - heap_validate_table. ##### 2.3.2.1.1 `heap_create` This is the main function that will allow us to create the heap. it calls two validation functions: `heap_validate_alignment` and `heap_validate_table`. this will help making sure that the parameters passed by the kernel are correct and work within our implementation.

```
int heap_create(struct heap* heap, void* ptr, void* end, struct heap_table* table)
{
        int res = 0;
        if ( !heap_validate_alignment(ptr) || !heap_validate_alignment(end) )
        {
                res = -EINVARG;
                goto out;
        }
        memset(heap, 0, sizeof(struct heap));
        heap->saddr = ptr;
        heap->table = table;

        res = heap_validate_table(ptr, end, table);
        if (res < 0)
        {
                goto out;
        }

        size_t table_size = sizeof(HEAP_BLOCK_TABLE_ENTRY) * table->total;
        memset(table->entries, HEAP_BLOCK_TABLE_ENTRY_FREE, table_size);

out:
        return res;
}
```

- `int heap_create(struct heap* heap, void* ptr, void* end, struct heap_table* table)`: first we define `heap_create`. this function will be mainly called by the kernel at init time and no one else.
- `int res = 0;`: here we create `res` with `0` as the initialized variable.
- `if ( !heap_validate_alignment(ptr) || !heap_validate_alignment(end) )`: we need to check if the values for the beginning and end pointers are valid and see if the operation `ptr % 4096 == 0`. check [[#2.3.2.1.2 `heap_validate_alignment`]]
- `res = -EINVARG;`: if not, we know that we some of the parameters were wrong.
- `goto out;`: here wo go to the `out` label and return 0.

- `memset(heap, 0, sizeof(struct heap));`: here we zero out our heap structure, in case some other memory was there before we claimed it.
- `heap->saddr = ptr;`: here we set the starting address of the heap. in our case, it's `0x01000000`. (check `config.h`)
- `heap->table = table;`: here we pass the heap entry table passed by the kernel to the `heap` structure.
- `res = heap_validate_table(ptr, end, table);`: here we validate whether the table is correct or not. check [[#2.3.2.1.3 `heap_validate_table`]]
- `if (res < 0)`: if `heap_validate_table` is less than zero, then we know the validation failed.
- `goto out;`: goto out and return less than zero.
- `size_t table_size = sizeof(HEAP_BLOCK_TABLE_ENTRY) * table->total;`: here we get the table size, as to be able to `memset` the entries to zero, meaning that they're all unused (`b00000000`)
- `memset(table->entries, HEAP_BLOCK_TABLE_ENTRY_FREE, table_size);`: here we do the memset.
- `out:`: `out` label.
- `return res;`: here we just return either: a. `-EINVARG` or b. the validated table.
- `} ##### 2.3.2.1.2`heap_validate_alignment` this function helps us verify the alignment of the beginning and ending memory sections of our heap.

```
static bool heap_validate_alignment(void* ptr)
{
        return ((unsigned int) ptr % PEACHOS_HEAP_BLOCK_SIZE) == 0;
}
```

- `static bool heap_validate_alignment(void* ptr)`: we define our function that will take a void pointer; in our case, this takes the sections of memory `ptr` and `end` to be validated.
- `return ((unsigned int) ptr % PEACHOS_HEAP_BLOCK_SIZE) == 0;`: we return 0 only if the operation `ptr % PEACHOS_HEAP_BLOCK_SIZE` (4096) is zero. this is done so we can have a non floating amount of memory blocks to be used, because if `ptr % PEACHOS_HEAP_BLOCK_SIZE` wasn't 0, then memory would be left out and unused. ##### 2.3.2.1.3 `heap_validate_table` this functions helps us validate the table formation and whether its a valid heap table or not.

```
static int heap_validate_table(void* ptr, void* end, struct heap_table* table)
{
        int res = 0;

        size_t table_size = (size_t)(end - ptr);
        size_t total_blocks = table_size / PEACHOS_HEAP_BLOCK_SIZE;
        if (table->total != total_blocks)
        {
                res = -EINVARG;
                goto out;
        }

out:
        return res;
}
```

- `static int heap_validate_table(void* ptr, void* end, struct heap_table* table)`: we define the function, it will take the starting address, end address and the `heap_table`.
- `int res = 0;`: as before, we initialize `res` to zero.
- `size_t table_size = (size_t)(end - ptr);`: this line calculates the difference between the end and beginning memory addresses and gives us a table size. we need to cast it to `size_t`.
- `size_t total_blocks = table_size / PEACHOS_HEAP_BLOCK_SIZE;`: here we divide the table size by the block size. for more on the calculations, check [[Heap and memory alloc related notes]]
- `if (table->total != total_blocks)`: here we check if the total amount of blocks isn't the total amount calculated within this function. if so, the programmer did something wrong.
- `res = -EINVARG;`: we set it to minus `EINVARG` and return it.
- `goto out;`: return `-EINVARG`.
- `out:`: `out` label.
- `return res;`: or return zero if the function ended correctly. #### 2.3.2.2. Heap memory allocation process. Now that we've created out heap, we need to understand how our memory is being allocated. Let's check it out. ##### 2.3.2.2.1 `heap_malloc` this function will be wrapped by the `kmalloc` call. it's used to allocate memory in the heap.

```
void* heap_malloc(struct heap* heap, size_t size)
{
        size_t aligned_size = heap_align_value_to_upper(size);
        uint32_t total_blocks = aligned_size / PEACHOS_HEAP_BLOCK_SIZE;
        return heap_malloc_blocks(heap, total_blocks);
}
```

- `void* heap_malloc(struct heap* heap, size_t size)`: here we define the function.
- `size_t aligned_size = heap_align_value_to_upper(size);`: here we need to align the size given by using the modulus operation.
- `uint32_t total_blocks = aligned_size / PEACHOS_HEAP_BLOCK_SIZE;`: this will give us the total amount of blocks that are needed to be allocated.
- `return heap_malloc_blocks(heap, total_blocks);`: and this function starts doing the actual allocation. ##### 2.3.2.2.2 `heap_align_value_to_upper` this function serves the purpose of aligning the `malloc` sizes to their correspondant block size. for example, if the programmer passes `50`, this function will return `4096`.

```
static uint32_t heap_align_value_to_upper(uint32_t val)
{
        if ((val % PEACHOS_HEAP_BLOCK_SIZE) == 0)
        {
                return val;
        }
        val = (val - ( val % PEACHOS_HEAP_BLOCK_SIZE));
        val += PEACHOS_HEAP_BLOCK_SIZE;
        return val;

}
```

- `static uint32_t heap_align_value_to_upper(uint32_t val)`: definition of the function.
- `if ((val % PEACHOS_HEAP_BLOCK_SIZE) == 0)`: here we calculate the rest of the division between `val` and `PEACHOS_HEAP_BLOCK_SIZE`.
- `return val;`: if zero, (meaning a multiple of 4096), we return and don't make any changes to `val`.
- `val = (val - ( val % PEACHOS_HEAP_BLOCK_SIZE));`: otherwise, we take `val` and substract the rest of `val % PEACHOS_HEAP_BLOCK_SIZE`. check [[Heap and memory alloc related notes]] for more info.
- `val += PEACHOS_HEAP_BLOCK_SIZE;`: here we now have to add the `PEACHOS_HEAP_BLOCK_SIZE` for it to be aligned correctly.
- `return val;`: and return the value. ##### 2.3.2.2.3 `heap_malloc_blocks` this function is called by [[#2.3.2.2.1 `heap_malloc`]]. let's see what it does.

```
void* heap_malloc_blocks(struct heap* heap, uint32_t total_blocks)
{
        void* address = 0;
        int start_block = heap_get_start_block(heap, total_blocks);
        if (start_block < 0)
        {
                goto out;
        }

        address = heap_block_to_address(heap, start_block);

        // mark blocks as taken
        heap_mark_blocks_taken(heap, start_block, total_blocks);

out:
        return address;
}
```

- `void* heap_malloc_blocks(struct heap* heap, uint32_t total_blocks)`: here we define the function.
- `void* address = 0;`: now we set the address to zero.
- `int start_block = heap_get_start_block(heap, total_blocks);`: now we get the starting block number on our `heap` array. check [[#2.3.2.2.4 `heap_get_start_block`]] for more.
- `if (start_block < 0)`: if the `heap_get_start_block` returns a value minor to zero, then we have an error (potentially an ENOMEM).
- `goto out;`: return address 0.
- `address = heap_block_to_address(heap, start_block);`: here we calculate the block memory address.
- `heap_mark_blocks_taken(heap, start_block, total_blocks);`: now we the the in use bits (00000001).
- `out:`: out label.
- `return address;`: returns the address obtained by either `ENOMEM` or `heap_block_to_address`. this will be the address returned in `kmalloc`. ##### 2.3.2.2.4 `heap_get_start_block` before actually marking out blocks taken, we need to know which blocks we need to mark as taken. let's see how.

```
int heap_get_start_block(struct heap* heap, uint32_t total_blocks)
{
        struct heap_table* table = heap->table;
```

```
    int bc = 0;
    int bs = -1;
    for (size_t i = 0; i < table->total; i++)
    {
        if (heap_get_entry_type(table->entries[i]) != HEAP_BLOCK_TABLE_ENTRY_FREE)
        {
            bc = 0;
            bs = -1;
            continue;
        };
        // if this is first block
        if (bs == -1)
        {
            bs = i;
        };
        bc++;
        if (bc == total_blocks)
        {
            break;
        };
    }
    if (bs == -1)
    {
        return -ENOMEM;
    };

    return bs;
}
```

- `int heap_get_start_block(struct heap* heap, uint32_t total_blocks)`: function definition.
- `struct heap_table* table = heap->table;`: here we take the heap table to be checked.
- `int bc = 0;`: we initialize the block count.
- `int bs = -1;`: and the start block.
- `for (size_t i = 0; i < table->total; i++)`: we need to check the entirety of the heap table to check for free blocks. for that, we access `table->total` in this for loop.
- `if (heap_get_entry_type(table->entries[i]) != HEAP_BLOCK_TABLE_ENTRY_FREE)`: if the current entry doesn't have the free bits (00000001), we…
- `bc = 0;`: restart the counter and
- `bs = -1;`: restart the block size.
- `continue;`: and continue.
- `if (bs == -1)`: but if the block doesn't have the taken bits, we…
- `bs = i;`: the `bs` to the current index in the entry table.
- `bc++;`: we increment block count after finding a free block.
- `if (bc == total_blocks)`: and if the block count is equal to the total_blocks needed, we…
- `break;`: break!
- `if (bs == -1)`: but if not, and if `bs` is -1, then we…
- `return -ENOMEM;`: probably ran out of memory.
- `return bs;`: return the start block. ##### 2.3.2.2.5 `heap_get_entry_type` before calculating the block memory address, let's talk about the `heap_get_entry_type` function.

```
static int heap_get_entry_type(HEAP_BLOCK_TABLE_ENTRY entry)
{
    return entry & 0x0f;
}
```

- `static int heap_get_entry_type(HEAP_BLOCK_TABLE_ENTRY entry)`: function definition.
- `return entry & 0x0f;`: here we get the entry type using bitwise operations. check [[Heap and memory alloc related notes]] for more info. ##### 2.3.2.2.6 `heap_block_to_address` after getting the start block, we need to calculate its address in memory. let's see how.

```
void* heap_block_to_address(struct heap* heap, int block)
{
    return heap->saddr + (block * PEACHOS_HEAP_BLOCK_SIZE);
}
```

- `void* heap_block_to_address(struct heap* heap, int block)`: function definition.
- `return heap->saddr + (block * PEACHOS_HEAP_BLOCK_SIZE);`: here we calculate the starting memory address of the given block of memory. check [[Heap and memory alloc related notes]] for more.

2.3.2.2.7 `heap_mark_blocks_taken` with the block memory address and the aligned block size, we can start to mark the blocks as taken.

```
void heap_mark_blocks_taken(struct heap* heap, int start_block, int total_blocks)
{
    int end_block = (start_block + total_blocks) - 1;
```

```
	HEAP_BLOCK_TABLE_ENTRY entry = HEAP_BLOCK_TABLE_ENTRY_TAKEN | HEAP_BLOCK_IS_FIRST;
	if (total_blocks > 1)
	{
		entry |= HEAP_BLOCK_HAS_NEXT;
	}

	for (int i = start_block; i <= end_block; i++)
	{
		heap->table->entries[i] = entry;
		entry = HEAP_BLOCK_TABLE_ENTRY_TAKEN;
		if (i != end_block - 1)
		{
			entry |= HEAP_BLOCK_HAS_NEXT;
		}
	}
}
```

- `void heap_mark_blocks_taken(struct heap* heap, int start_block, int total_blocks)`: function definition.
- `int end_block = (start_block + total_blocks) - 1;`: here we set the ending block.
- `HEAP_BLOCK_TABLE_ENTRY entry = HEAP_BLOCK_TABLE_ENTRY_TAKEN | HEAP_BLOCK_IS_FIRST;`: now we create a taken and is first entry. by `OR`ing the two values.
- `if (total_blocks > 1)`: we check the total block variable to validate whether we need a `HAS_N` bit or not.
- `entry |= HEAP_BLOCK_HAS_NEXT;`: set the `HAS_N` bit if we have more than one block.
- `for (int i = start_block; i <= end_block; i++)`: here we'll iterate over all the blocks.
- `heap->table->entries[i] = entry;`: here we set the first entry in the heap table to `IS_FIRST` and `HAS_N`.
- `entry = HEAP_BLOCK_TABLE_ENTRY_TAKEN;`: now we change the value to only taken.
- `if (i != end_block - 1)`: and if the block we're currently in isn't the last block…
- `entry |= HEAP_BLOCK_HAS_NEXT;`: we set the `HAS_N` bit.

And done. Although it technically isn't a long process, it sure feels like it is. Nevertheless, this implementation is quite simple. #### 2.3.2.3. Heap memory freeing process. We've created our heap and created a way to allocate memory into it. Now we need to create a way to free that memory. ##### 2.3.2.3.1 `heap_free` This is the initial `free` call that will be abstracted later by `kfree`.

```
void heap_free(struct heap* heap, void* ptr)
{
	heap_mark_blocks_free(heap, heap_address_to_block(heap, ptr));
}
```

- `void heap_free(struct heap* heap, void* ptr)`: function definition.
- `heap_mark_blocks_free(heap, heap_address_to_block(heap, ptr));`: here we take the `heap` and `ptr` and call the `heap_mark_blocks_free` function.

2.3.2.3.2. `heap_address_to_block`

This function is used to get a block index from a memory address.

```
int heap_address_to_block(struct heap* heap, void* addr)
{

	return ((int)(addr - heap->saddr)) / PEACHOS_HEAP_BLOCK_SIZE;
}
```

- `int heap_address_to_block(struct heap* heap, void* addr)`: function definition.
- `return ((int)(addr - heap->saddr)) / PEACHOS_HEAP_BLOCK_SIZE;`: here we get the difference between the `addr` and the initial `saddr` and divide it by the BLOCK_SIZE. check [[Heap and memory alloc related notes]] for more info. ##### 2.3.2.3.3 `heap_mark_blocks_free` this is the function that does the actual job :)

```
void heap_mark_blocks_free(struct heap* heap, int starting_block)
{
	struct heap_table* table = heap->table;
	for (int i = starting_block; i < (int)table->total; i++)
	{
		HEAP_BLOCK_TABLE_ENTRY entry = table->entries[i];
		table->entries[i] = HEAP_BLOCK_TABLE_ENTRY_FREE;
		if (!(entry & HEAP_BLOCK_HAS_NEXT))
		{
			break;
		}
	}
}
```

- `void heap_mark_blocks_free(struct heap* heap, int starting_block)`: function definition.
- `struct heap_table* table = heap->table;`: we access the table by creating another table.

- `for (int i = starting_block; i < (int)table->total; i++)`: here we start iterating over all the blocks.
- `HEAP_BLOCK_TABLE_ENTRY entry = table->entries[i];`: we get the entry into a maleable variable.
- `table->entries[i] = HEAP_BLOCK_TABLE_ENTRY_FREE;`: here we set the entry to `FREE` (b00000000)
- `if (!(entry & HEAP_BLOCK_HAS_NEXT))`: now we check if the current block has the `HAS_N` bit. if it doesn't…
- `break;`: we break the loop and all our blocks have been freed.

2.4. kheap.c, kheap.h

Now we'll see how the kernel abstracts these functions into stuff that can actually be used, like `kfree` and `kmalloc`.

2.4.1. `kheap.c`

```
#include "kheap.h"
#include "heap.h"
#include "config.h"
#include "kernel.h"

struct heap kernel_heap;
struct heap_table kernel_heap_table;

void kheap_init()
{
	int total_table_entries = PEACHOS_HEAP_SIZE_BYTES / PEACHOS_HEAP_BLOCK_SIZE;
	kernel_heap_table.entries = (HEAP_BLOCK_TABLE_ENTRY*)(PEACHOS_HEAP_TABLE_ADDRESS);
	kernel_heap_table.total = total_table_entries;

	void* end = (void*)(PEACHOS_HEAP_ADDRESS + PEACHOS_HEAP_SIZE_BYTES);
	int res = heap_create(&kernel_heap, (void*)(PEACHOS_HEAP_ADDRESS), end, &kernel_heap_table);
	if (res < 0)
	{
		print("failed to create heap\\n");
	}
}

void* kmalloc(size_t size)
{
	return heap_malloc(&kernel_heap, size);
}

void kfree(void* ptr)
{
	heap_free(&kernel_heap, ptr);
}
```

- `struct heap kernel_heap;`: here the kernel creates it's heap or data pool.
- `struct heap_table kernel_heap_table;`: and here we create the heap table.
- `void kheap_init()`: we define the `kheap_init` that we'll call in our `kernel.c` file later on.
- `int total_table_entries = PEACHOS_HEAP_SIZE_BYTES / PEACHOS_HEAP_BLOCK_SIZE;`: here we set the total amount of entries by dividing the `SIZE_BYTES` of the heap and the `BLOCK_SIZE` of each block.
- `kernel_heap_table.entries = (HEAP_BLOCK_TABLE_ENTRY*)(PEACHOS_HEAP_TABLE_ADDRESS);`: here we cast our constant `0x00007E00` to `HEAP_BLOCK_TABLE_ENTRY*`.
- `kernel_heap_table.total = total_table_entries;`: here we set the total amount of blocks in our heap.
- `void* end = (void*)(PEACHOS_HEAP_ADDRESS + PEACHOS_HEAP_SIZE_BYTES);`: here we obtain the ending address of our heap.
- `int res = heap_create(&kernel_heap, (void*)(PEACHOS_HEAP_ADDRESS), end, &kernel_heap_table);`: here we finally create our heap by calling [[#2.3.2.1.1 `heap_create`]].
- `if (res < 0)`: we check the response, and if less than zero (could be `EINVARG` or `ENONEM`)…
- `print("failed to create heap\\n");`: we fail.
- `void* kmalloc(size_t size)`: now we define the abstraction for the [[#2.3.2.2.1 `heap_malloc`]] function.
- `return heap_malloc(&kernel_heap, size);`: here we do the call to `heap_malloc` and return the pointer to the address of memory reserved to the `heap` that we asked.
- `void kfree(void* ptr)`: same as above, but now we're calling [[#2.3.2.3.1 `heap_free`]].
- `heap_free(&kernel_heap, ptr);`: and here we call `heap_free` and clear the bits off of the entry table for them to be used at a later time. ### 2.4.2. `kheap.h`

```
#ifndef KHEAP_H
#define KHEAP_H
#include <stdint.h>
#include <stddef.h>

void kheap_init();
void* kmalloc(size_t size);
void kfree(void* ptr);

#endif
```

- `void kheap_init();`: prototype for `kheap_init` that will be called by `kernel.c`.
- `void* kmalloc(size_t size);`: prototype for `kmalloc` that will be called by `kernel.c`.
- `void kfree(void* ptr);`: prototype for `kfree` that will be called by `kernel.c`.

And with all the pieces combined, we can see our heap in action!

![[Pasted image 20250512005418.png]] but what about `kfree`? ![[Sin título 5.jpg]] It's working! *What I failed to mention was that I had to debug the heap implementation because I misread a `1` for an `i` and a `*` for a `+`.*

2. Understanding the concept of Paging

- It allows us to remap memory addresses to point to other memory addresses.
- Can be used to provide the illusion we have the maximum amount of RAM installed.
- Can be used to hide memory from other processes.

2.1. Remapping memory

- Paging allows us to remap one memory address to another, so `0x100000` could point to `0x200000`.
- Paging works ni 4096 byte block sizes by default. The blocks are called pages.
- When paging is enabled, the `MMU` (Memory Management Unit) will look at your allocated page tables to resolve virtual addresses into physical addresses.
- Paging allows us to pretend memory exists when it does not.

2.2. Virtual vs Physical addresses.

- Virtual addresses are addresses that are not pointing to the address in memory that their value says they are. Virtual address `0x100000` might point to physical address `0x200000` as an example.
- Physical addresses are absolute addresses in memory whose value points to the same address in memory. For eaxmple, if phyisical address `0x100000` points to `0x100000`, then this is a physical address (duh).
- Essentially, virtual and physical addresses are just terms we use to explain how a piece of memory is being accessed. ### 2.2.1. Paging illustration ![[Pasted image 20250514220523.png]] ## 2.3. Structure of Paging
- 1024 page directories that point to 1024 page tables.
- 1024 page table entries per page table.
- Each page table entry covers 4096 bytes of memory.
- Each "4096" byte block of memory is called a page.
- 1024 * 1024 * 4096 = 4.294.967.296 Bytes / 4GB of addressable memory. ## 2.4. Page directory and page entry structure
- Holds a pointer to a page table.
- Holds attributes. Check [[Paging Notes]] for the attribute table.

2.5. First Page Table Visualized

![[Pasted image 20250514221855.png]] ### 2.5.1. Second Page Table Visualized ![[Pasted image 20250514222058.png]]

2.6. Page Fault Exceptions

- The CPU will call the page fault interrupt `0x14` when there was a problem with paging.

  - The exception is invoked:
    * If you access a apge in memory that does nmot have its `P` (Present) bit set.
    * If you access a page that is for supervisor but you aren't a supervisor.
    * If you write to a page that is read-only and you are not supervisor. ## 2.7. Hiding memory from processes.
- If we give each process its own page directory table, then we can map the memory for the process however we want it to be. We can make it so the process can only see itself.
- Hiding memory can be achieved by switching the page directories when moving between processes.
- All processes can access the same virtual memory addresses but they will point to different physical addresses. ## 2.8 Illusion of more memory.
- We can pretend we have the maximum amount of memory even if we do not.
- This is achieved by creating page tables that are not present. Once a process accesses this non-present address, a page fault will occur. We can then load the page back into memory and the process had no idea of this happening.
- 100MB of system memory can act as if it has access to the full 4GB on a 32 bit architecture.
  - This would work, but accessing non-present pages would cause a page fault.

2.9. Benefits to paging.

- Processes can access the same virtual memory, but they won't overwrite each other.
- Security is an added benefit, as we can map out physical memory that we don't want processes to see.
- It can be used to prevent overwritting sensitive sections of memory.
- And many more. ## 2.10. Enabling paging Paging can be enabled using a few lines of Assembly code. It'd look like this:

```
[BITS 32]

section .asm

global paging_load_directory
global enable_paging

paging_load_directory:
    push ebp
    mov esp, ebp
    mov eax, [ebp+8]
    mov cr3, eax
    pop ebp
    ret

enable_paging:
    pusb ebp
    mov esp, ebp
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax
    pop ebp
    ret
```

This code will be explained later on when we implement paging within the kernel.

3. Implementing Paging

Paging wasn't as hard as the heap to enable. Yes, enable. We haven't yet implemented proper paging with virtual memory. But we will! For now, let's see what we did to enable it.

kheap.h, kheap.c

kheap.c

First we had to implement `kzalloc`. A function that allocates memory with `kmalloc` and then zeroes it out with `memset`.

```
void* kzalloc(size_t size)
{
        void* ptr = kmalloc(size);
        if(!ptr)
        {
                return 0;
        }
        memset(ptr, 0x00, size);
        return ptr;
```

```
}
```

- `void* kzalloc(size_t size)`: function definition. nothing fancy. we just take a size_t `size`.
- `void* ptr = kmalloc(size);`: here we create our section on the heap with `size` size.
- `if(!ptr)`: if something went wrong while getting some memory (EINVARG or ENOMEM)…
- `return 0;`: we return 0.
- `memset(ptr, 0x00, size);`: here we `memset` the entire region of memory to zero.
- `return ptr;`: and return that pointer to the memory address that we just zeroed out.

And in the `kheap.h` we just added the prototype of our function for it to be called by other files.

`kheap.h`

```
void* kzalloc(size_t size);
```

It's practically the same thing as `kmalloc`. ## paging.asm, paging.h and paging.c. Here's where the magic of enabling paging happens. ### `paging.asm` In this file, we write the routines needed to enable paging. There's two, `enable_paging` and `paging_load_directory`.

```
[BITS 32]

section .asm
global paging_load_directory
global enable_paging

paging_load_directory:
        push ebp
        mov ebp, esp
        mov eax, [ebp+8]
        mov cr3, eax
        pop ebp
        ret

enable_paging:
        push ebp
        mov ebp, esp
        mov eax, cr0
        or eax, 0x80000000
        mov cr0, eax
        pop ebp
        ret
```

- `paging_load_directory:`: label setup.
- `push ebp`: stack setup, nothing new.
- `mov ebp, esp`: stack setup, nothing new.
- `mov eax, [ebp+8]`: here we take the argument passed onto us by the C code. we'll see more on that later.
- `mov cr3, eax`: here we move the `eax` register (containing the argument) and the move it into the `cr3` register. we do this because we cannot change `cr3` directly.
- `pop ebp`: here we delete our stack.
- `ret`: and we return!
- `enable_paging:`: label setup.
- `push ebp`: stack setup, nothing new.
- `mov ebp, esp`: stack setup, nothing new.
- `mov eax, cr0`: here we move the value of `cr0` into `eax`.
- `or eax, 0x80000000`: here we set the 31st bit to 1. read the page on Assembly for info on HOW and WHY.
- `mov cr0, eax`: here we move `eax` into the `cr0` register.
- `pop ebp`: here we delete our stack.
- `ret`: and we return! ### 3.2.2. `paging.h` Before getting our hands dirty with some C code, we need to check the constants and structures created within our header file.

```
#define PAGING_CACHE_DISABLED   0b00010000
#define PAGING_WRITE_THROUGH    0b00001000
#define PAGING_ACCESS_FROM_ALL  0b00000100
#define PAGING_IS_WRITEABLE     0b00000010
#define PAGING_IS_PRESENT       0b00000001

#define PAGING_TOTAL_ENTRIES_PER_TABLE 1024
#define PAGING_PAGE_SIZE 4096

struct paging_4gb_chunk
{
        uint32_t* directory_entry;
};

uint32_t* paging_4gb_chunk_get_directory(struct paging_4gb_chunk* chunk);
struct paging_4gb_chunk* paging_new_4gb(uint8_t flags);
void paging_switch(uint32_t* directory);
void enable_paging();
```

- `#define PAGING_CACHE_DISABLED   0b00010000`: bitmask. it represents the PCD 'cache disable' bit.
- `#define PAGING_WRITE_THROUGH    0b00001000`: bitmask. it represents the PWT 'write through' bit.
- `#define PAGING_ACCESS_FROM_ALL  0b00000100`: bitmask. it represents the U/S 'user/supervisor' bit.
- `#define PAGING_IS_WRITEABLE     0b00000010`: bitmask. it represents the R/W 'read/write' bit.
- `#define PAGING_IS_PRESENT       0b00000001`: bitmask. it represents the P 'present' bit.
- `#define PAGING_TOTAL_ENTRIES_PER_TABLE 1024`: we define 1024 entries per page table.
- `#define PAGING_PAGE_SIZE 4096`: and each page table contains 4096 pages.
- `struct paging_4gb_chunk`: this is our page table.
- `uint32_t* directory_entry;`: and this is our directory entries or page directories.
- `uint32_t* paging_4gb_chunk_get_directory(struct paging_4gb_chunk* chunk);`: this is a helper function that will allow us to access page directories.
- `struct paging_4gb_chunk* paging_new_4gb(uint8_t flags);`: this function will allow us to create a page and its directories.
- `void paging_switch(uint32_t* directory);`: and this function linearly assigns a physical address to a page address or virtual address.
- `void enable_paging();`: and here we call our Assembly `enable_paging` label.

All the bitmasks are attributes that are assigned to the page directory. These are well documented in the paging notes and in the OsDev wiki, so I won't go through them here. ### 3.2.3. `paging.c` Now let's get our hands dirty. It's not that much code in comparison to the heap implementation, but it gets a bit confusing. Nevertheless, we'll go over it step by step. #### 3.2.3.1. `struct paging_4gb_chunk paging_new_4gb(uint8_t flags)`

```
void paging_load_directory(uint32_t* directory);
static uint32_t* current_directory = 0;

struct paging_4gb_chunk* paging_new_4gb(uint8_t flags)
{
        uint32_t* directory = kzalloc(sizeof(uint32_t) * PAGING_TOTAL_ENTRIES_PER_TABLE);
        int offset = 0;
        for (int i = 0; i < PAGING_TOTAL_ENTRIES_PER_TABLE; i++)
        {
                uint32_t* entry = kzalloc(sizeof(uint32_t) * PAGING_TOTAL_ENTRIES_PER_TABLE);
                for (int b = 0; b < PAGING_TOTAL_ENTRIES_PER_TABLE; b++)
                {
                        entry[b] = (offset + (b * PAGING_PAGE_SIZE)) | flags;
                }
                offset += (PAGING_TOTAL_ENTRIES_PER_TABLE * PAGING_PAGE_SIZE);
                directory[i] = (uint32_t) entry | flags | PAGING_IS_WRITEABLE;
        }
        struct paging_4gb_chunk* chunk_4gb = kzalloc(sizeof(struct paging_4gb_chunk));
        chunk_4gb -> directory_entry = directory;

        return chunk_4gb;
}
```

- `void paging_load_directory(uint32_t* directory);`: Here we make an explicit definition of the `paging_load_directory` label that we created in the `paging.asm` file. Remember, this function loads the paging directory into the `cr3` register.
- `static uint32_t* current_directory = 0;`: Here we create a global static variable because we want to be able to access this variable outside of the scope of the function that we'll see in just a bit. This is because we'll use it later on too.
- `struct paging_4gb_chunk* paging_new_4gb(uint8_t flags)`: simple function definition. it'll take a single byte which will represent the flags of the page directory. those bitmasks we saw before? yeah,

we'll use them in this function.

- `uint32_t* directory = kzalloc(sizeof(uint32_t) * PAGING_TOTAL_ENTRIES_PER_TABLE);`: Here we multiply `uint32_t` (at least 32 bits) with `PAGING_TOTAL_ENTRIES_PER_TABLE`. the result will represent the size of our `directory` allocation using the function we just created `kzalloc`, which will fill a buffer of size `(sizeof(uint32_t) \\* PAGING_TOTAL_ENTRIES_PER_TABLE)`and then zero it out.*
  - we zero it out because if any memory pointer returned by the `kzalloc` function has any bits set, it could mess up our page directory.
  - we allocate `(sizeof(uint32_t) * PAGING_TOTAL_ENTRIES_PER_TABLE)` because the result will be 4096 or `PAGING_PAGE_SIZE`.
- `int offset = 0;`: here we initialize an offset. this variable is used to keep track of the memory region that is being mapped.
- `for (int i = 0; i < PAGING_TOTAL_ENTRIES_PER_TABLE; i++)`: here we start a for loop that will populate our `directory` with it's attributes and flags.
- `uint32_t* entry = kzalloc(sizeof(uint32_t) * PAGING_TOTAL_ENTRIES_PER_TABLE);`: we'll use entry for the page entries.
- `for (int b = 0; b < PAGING_TOTAL_ENTRIES_PER_TABLE; b++)`: here we start another for loop to assign the flags to the `entry` variable for our page directory.
- `entry[b] = (offset + (b * PAGING_PAGE_SIZE)) | flags;`: now we assign the attributes at the given offset. Ait might seem confusing, but remember: each directory has entries. each entry points to a page table, which is also an array of 1024 entries. if this explaination in not enough, check the misc notes on Paging.
- `offset += (PAGING_TOTAL_ENTRIES_PER_TABLE * PAGING_PAGE_SIZE);`: Here we increment the `offset` to start populating the next four megabytes of memory.
- `directory[i] = (uint32_t) entry | flags | PAGING_IS_WRITEABLE;`: And here we set the Page Directory attributes using `OR` bitwise operations.
- `struct paging_4gb_chunk* chunk_4gb = kzalloc(sizeof(struct paging_4gb_chunk));`: Here we finally initialize the chunk of memory that we'll return and initialize it to the size of `paging_4gb_chunk` which is, indeed, four gigabytes.
- `chunk_4gb -> directory_entry = directory;`: And here we assign the PD to the newly created `chunk_4gb` chunk.
- `return chunk_4gb;`: And we return it. \ I have no idea why the instructor didn't use the PAGING_PAGE_SIZE instead. The result is the same. I'll be in touch with him and check why he chose that way of calculating the kzalloc size.* This is the most complex function of the `paging.c` section. Read it several times until you understand it. Use the Paging notes on the misc section.

3.2.3.2. `void paging_switch(uint32_t* directory)` This function will be used to load the Page Directory to the CPU register `cr3` and set the `current_directory` variable that we assigned before to the current Page Directory created with `paging_new_4gb_chunk`.

```
void paging_switch(uint32_t* directory)
{
        paging_load_directory(directory);
        current_directory = directory;
}
```

- `void paging_switch(uint32_t* directory)`: function definition. we'll expect the 4 gigabyte chunk of memory.
- `paging_load_directory(directory);`: here we load the chunk into the CPU `cr3` register.
- `current_directory = directory;`: and here we set the global variable `current_directory` to the generated chunk by `paging_new_4gb_chunk`. #### 3.2.3.3. `uint32_t* paging_4gb_chunk_get_directory(struct paging_4gb_chunk* chunk)` This function will be used in the future. It's a way to access the Page Directory directly. We do NOT want to make this accessible outside of the `paging.c` scope, as it could allow a malicious person to access the real physical memory.

```
uint32_t* paging_4gb_chunk_get_directory(struct paging_4gb_chunk* chunk)
{
        return chunk->directory_entry;
}
```

- `uint32_t* paging_4gb_chunk_get_directory(struct paging_4gb_chunk* chunk)`: here we receive the chunk of memory generated previously.
- `return chunk->directory_entry;`: and here we return the `directory_entry`. And that's it for the paging implementation. We have to make some changes to `kernel.c` and the `Makefile`, but that's just calling other functions. Let's see it.

3.2.4. `kernel.c`

In `kernel.c`, we make the calls to some functions that we've implemented.

```
static struct paging_4gb_chunk* kernel_chunk = 0;
void kernel_main()
    ...
        // setting up the pages
        kernel_chunk = paging_new_4gb(PAGING_IS_WRITEABLE | PAGING_IS_PRESENT | PAGING_ACCESS_FROM_ALL);
        // switching pages
        paging_switch(paging_4gb_chunk_get_directory(kernel_chunk));
        // enabling paging
        enable_paging();
```

- `static struct paging_4gb_chunk* kernel_chunk = 0;`: here we make our page directory and page table a global variable so it can be accessed by other functions.
    - `kernel_chunk = paging_new_4gb(PAGING_IS_WRITEABLE | PAGING_IS_PRESENT | PAGING_ACCESS_FROM_ALL)` here we generate the page directory and page table with the writeable, present and U/S flags.
    - `paging_switch(paging_4gb_chunk_get_directory(kernel_chunk));`: here we tell the CPU to load the `kernel_chunk` chunk into the `cr3` register.
    - `enable_paging();`: and here we enable paging. it's **important** to FIRST load the PD and PT and ***then*** enable paging. if we don't, the system ***WILL PANIC!***. And that's all for the `kernel.c` file changes.

3.2.5. `Makefile`

`Makefile` changes are pretty basic. We're just telling it to compile and link the `paging.o` and `paging.asm.o` objects to the kernel.

```
FILES = ... ./build/memory/paging/paging.o \\
    ./build/memory/paging/paging.asm.o

./build/memory/paging/paging.o: ./src/memory/paging/paging.c
        i686-elf-gcc $(INCLUDES) -I./src/memory/paging/ $(FLAGS) -std=gnu99 -c ./src/memory/paging/paging.c -o ./build/memory/paging/paging.o

./build/memory/paging/paging.asm.o: ./src/memory/paging/paging.asm
        nasm -f elf -g ./src/memory/paging/paging.asm -o ./build/memory/paging/paging.asm.o
```

We add the files that we expect to be there to `make all` and then just compile them! It's the same process we've done tons of times before.

Hey, it works!

Figure 1: working implementation of paging, memcpy

This is an image of the functional paging initialization, kzalloc and an implementation of memcpy (outside of the scope of the chapter; will document!). ain't that nice!

4. Modifying the Page Table.

Now we'll see how we can modify the page table from the page directory to be able to remap virtual addresses to physical addresses. ## 4.1. `paging.c` we've worked in this file before, so we'll jump right into the changes and new functions we've made. ### 4.1.1. `bool paging_is_aligned(void* addr)` This function will help us validate whether the `addr` passed on to us is aligned (check Paging notes for what *aligned* means) and return a boolean value.

```
bool paging_is_aligned(void* addr)
{
        return ((uint32_t)addr % PAGING_PAGE_SIZE) == 0;
}
```

- `bool paging_is_aligned(void* addr)`: function definition. we'll take a pointer to an addr.

- `return ((uint32_t)addr % PAGING_PAGE_SIZE) == 0;`: and here we use the modulus operand to check if the page is aligned or not. ### 4.1.2. `int paging_get_indexes(void* virtual_address, uint32_t* directory_index_out, uint32_t* table_index_out)` this function will help us get the actual indexes from a virtual address and put the values in the pointers passed onto the function.

```
int paging_get_indexes(void* virtual_address, uint32_t* directory_index_out, uint32_t* table_index_out)
{
        int res = 0;
        if (!paging_is_aligned(virtual_address))
        {
                res = -EINVARG;
                goto out;
        }
        *directory_index_out = ((uint32_t)virtual_address / (PAGING_TOTAL_ENTRIES_PER_TABLE * PAGING_PAGE_SIZE));
        *table_index_out = ((uint32_t) virtual_address % (PAGING_TOTAL_ENTRIES_PER_TABLE * PAGING_PAGE_SIZE) / PAGING_PAGE_SIZE);

out:
        return res;
}
```

- `int paging_get_indexes(void* virtual_address, uint32_t* directory_index_out, uint32_t* table_index_out)`: here we take the virtual address from where we'll calculate both `directory` and `table` indexes from.
- `int res = 0;`: here we just initialize a return value. we've seen this before :)
- `if (!paging_is_aligned(virtual_address))`: if the virtual address passed onto the function isn't aligned…
- `res = -EINVARG;`: we return `-EINVARG` and…
- `goto out;`: go to the `out` label.
- `*directory_index_out = ((uint32_t)virtual_address / (PAGING_TOTAL_ENTRIES_PER_TABLE * PAGING_PAGE_SIZE));`: here we calcuate the directory index using this formula. it's further explained on the misc section about paging.
- `*table_index_out = ((uint32_t) virtual_address % (PAGING_TOTAL_ENTRIES_PER_TABLE * PAGING_PAGE_SIZE) / PAGING_PAGE_SIZE);`: here we calcuate the table index using this formula. it's further explained on the misc section about paging.
- `out:`: just a label.
- `return res;`: and we return. ### 4.1.3. `int paging_set(uint32_t* directory, void* virt, uint32_t phys_addr)` this function will be the one that we'll use to actually point the table index's virtual address to a physical address of our choosing.

```
int paging_set(uint32_t* directory, void* virt, uint32_t phys_addr)
{
        if(!paging_is_aligned(virt))
        {
                return -EINVARG;
        }
        uint32_t directory_index = 0;
        uint32_t table_index = 0;

        int res = paging_get_indexes(virt, &directory_index, &table_index);
        if (res < 0)
        {
                return res;
        }
        uint32_t entry = directory[directory_index];
        uint32_t* table = (uint32_t*)(entry & 0xfffff000); // take out the 20 bit address
        table[table_index] = phys_addr;


        return res;
}
```

- `int paging_set(uint32_t* directory, void* virt, uint32_t phys_addr)`: function definition. we'll take a page directory, the virtual address and the physical address.
- `if(!paging_is_aligned(virt))`: here we check if the virtual address passed onto us is aligned. if not…
- `return -EINVARG;`: we return `-EINVARG`.
- `uint32_t directory_index = 0;`: here we initialize a variable which the `paging_get_indexes` will populate further on.
- `uint32_t table_index = 0;`: here we initialize a variable which the `paging_get_indexes` will populate further on.
- `int res = paging_get_indexes(virt, &directory_index, &table_index);`: here's where the previous two vars are populated with `directory` and `table` indexes.
- `if (res < 0)`: here we check if the `paging_get_indexes` failed or not.

- `return res;`: if it failed, then we fail too.
- `uint32_t entry = directory[directory_index];`: now we create a variabe that points to the given directory entry, which points to the page table and the attributes of the page table in a 32 bit value.
- `uint32_t* table = (uint32_t*)(entry & 0xfffff000);`: in this variable we'll access the first 20 bits of the entry, which is the pointer to the page table.
- `table[table_index] = phys_addr;`: and here, having the table and its index, we can asign the physical address directly.
- `return res;`: and done! we return `res`, which is the exit status of `paging_get_indexes`. and done! we now have a function that will swap around physical and virtual memory at our liking. ## 4.2. `paging.h` in here we just expose the new functions for everyone to see :)

```
...
int paging_set(uint32_t* directory, void* virt, uint32_t phys_addr);
int paging_get_indexes(void* virtual_address, uint32_t* directory_index_out, uint32_t* table_index_out);
bool paging_is_aligned(void* addr);
```

4.3. `kernel.c`

no real changes were made to `kernel.c`, but we did add some code to exemplify the usage of the new functions.

```
...
        char* ptr = kzalloc(4096);
        paging_set(paging_4gb_chunk_get_directory(kernel_chunk), (void*)0x1000, (uint32_t)ptr | PAGING_ACCESS_FROM_ALL | PAGING_IS_PRESENT | PAGING_IS_WRITEABLE);

        enable_paging();

        char* ptr2 = (char*) 0x1000;
        ptr2[0] = 'A';
        ptr2[1] = 'B';
        print(ptr2);
        print(ptr);
...
```

-
```
    `char* ptr = kzalloc(4096);`: here we assign a pointer to a memory address returned by `kzalloc` ***before*** enabling paging.
```

- `paging_set(paging_4gb_chunk_get_directory(kernel_chunk), (void*)0x1000, (uint32_t)ptr | PAGING_ACCESS_FROM_ALL | PAGING_IS_PRESENT | PAGING_IS_WRITEABLE);`: here we make the swap between the `ptr` address `0x1000` and assign the page the attributes with `|` (`OR`) operations.

- `enable_paging();`: now we enable paging.

- `char* ptr2 = (char*) 0x1000;`: here we create a pointer to the virtual address `0x1000`, which, if paging is enabled, will now also point to `ptr`.

- `ptr2[0] = 'A';`: we assign `A` to `ptr2` for example purposes.

- `ptr2[1] = 'B';`: we assign `B` to `ptr2` for example purposes.

- `print(ptr2);`: we print `ptr2`; we can do this because `ptr` called `kzalloc`, so the string is null terminated

- `print(ptr);`: and we print `ptr`; same as above.

This image will ilustrate pointer `kzalloc` allocation with and without paging enabled. As the picture shows, when we don't have paging enabled, `ptr2` is the only one being displayed, and in the right, we can see that both `ptr` and `ptr2` display `AB`. this is thanks to the magic of paging :)

1. Preparing to read from the hard hisk. ## 1.1. PCI IDE Controller - IDE refers to the electrical specification of cables which connect ATA drives to a device. - IDE allows up to four disks to be connected. - There are four types of disks: - ATA (Serial): known as SATA; used by modern hard drives. - ATA (Parallel): known as PATA; used by hard drives. - ATAPI (Serial): Used by modern optical drives. - ATAPI (Parallel): Commonly used by optical drives. - We don't care if the drive is serial or parallel. ## 1.2. Possible IDE Drive Types - Primary Master Drive. - Primary Slave Drive. - Secondary Master Drive. - Secondary Slave Drive.

We'll be using the I/O operations we've implemented in the past to be able to read the disk sectors.

2. Reading from the disk with the ATA Controller.

Let's see the implementation of the LBA driver we made in Assembly in our `boot.asm` file but in C. ## 2.1. `disk.c` ### 2.1.1. `int disk_read_sector(int lba, int total, void* buffer)`

```
int disk_read_sector(int lba, int total, void* buffer)
{
	outb(0x1F6, (lba >> 24) | 0xE0);
	outb(0x1F2, total);
	outb(0x1F3, (unsigned char)(lba & 0xff));
	outb(0x1F4, (unsigned char) lba >> 8);
	outb(0x1F4, (unsigned char) lba >> 16);
	outb(0x1F7, 0x20);

	unsigned short* ptr = (unsigned short*) buffer;

	for (int b = 0; b < total; b++)
	{
		char c = insb(0x1F7);
		while(!(c & 0x08))
		{
			c = insb(0x1F7);
		}
		// copy from hdd to memory
		for (int i = 0; i < 256; i++)
		{
			*ptr = insw(0x1F0);
			ptr++;
		}
	}
	return 0;
}
```

- `int disk_read_sector(int lba, int total, void* buffer)`: function definition. this might look very familiar. check out the `load32` and `ata_lba_read` labels in the `boot.asm` file. here, we're taking an `lba` (disk sector) initial sector (from), a `total` of sectors to read (up to) and a `buffer` in which we'll store the bytes read.
- `outb(0x1F6, (lba >> 24) | 0xE0);`: here we right shuffle `lba` by 24 bits and `OR` it with `0xE0` and sending this data to the `0x1F6` I/O port.
  - if you look at `boot.asm`, the lines 73 to 76 might be familiar. this is because we're doing exactly that but in C.
- `outb(0x1F2, total);`: here we send the total amount of sectors to read into the `0x1F2` I/O port.
- `outb(0x1F3, (unsigned char)(lba & 0xff));`: in here we're sending the lower 8 bits to the `0x1F3` port. check the LBA ATA misc notes for info on this bitwise operation.
- `outb(0x1F4, (unsigned char) lba >> 8);`: Here we send the `lba` right shifted by 8 to the `0x1F4` port.
- `outb(0x1F4, (unsigned char) lba >> 16);`: And here we send the `lba` right shifted by 16 to the `0x1F4` port.
- `outb(0x1F7, 0x20);`: And now we set the command port `0x1F7` to `0x20` or the read sector command.
- `unsigned short* ptr = (unsigned short*) buffer;`: here we make a pointer to the buffer passed onto us by the caller. we need to cast it to be able to assign stuff to it.
- `for (int b = 0; b < total; b++)`: here we start a loop. we'll read `total` amount of bytes into the `ptr` buffer pointer.
- `char c = insb(0x1F7);`: here we read from the `0x1F7` port. we're expecting for…
- `while(!(c & 0x08))`: bit 0x08. this bit is sent to us by the LBA drive and tells us that it's ready to send the data.
- `c = insb(0x1F7);`: here we continue to read and check if the `0x08` bit is set.
- `for (int i = 0; i < 256; i++)`: if the `0x08` bit is set we go into the reading loop.
- `*ptr = insw(0x1F0);`: here we read two bytes into the `*ptr`.
- `ptr++;`: and we increment the pointer until `i` is higher than 256.
- `return 0;`: and we return!

If you didn't understand the code, I wholeheartedly recommed you reading on the `boot.asm` code again on the 3-PROTECTEDMODE page. It'll help you understand this code better.

2.2 `disk.h`

For now, we just publish the function in the header file.

```
#ifndef DISK_H
#define DISK_H
```

```
int disk_read_sector(int lba, int total, void* buffer);

#endif
```

3. Implementing a disk driver.

Now, what we have already made will help us abstract more things. For example, we can give the programmer an interface to select a drive and read an amount of bytes from it. For now, we'll only have drive zero, but we'll fix that when the time is right. ## 3.1. `disk.h` We've made some changes to the `disk.h` file. Mainly, we've created some constants and a struct to help us with our drives.

```
typedef unsigned int PEACHOS_DISK_TYPE;

// real physical disk
#define PEACHOS_DISK_TYPE_REAL 0

struct disk
{
	PEACHOS_DISK_TYPE type;
	int sector_size;
};

int disk_read_block(struct disk* idisk, unsigned int lba, int total, void* buf);
struct disk* disk_get(int index);
void disk_search_and_init();
```

- `typedef unsigned int PEACHOS_DISK_TYPE;`: here we create a type definition of `unsigned int` to a DISK_TYPE. it could be a drive, a partition, or something else.
- `#define PEACHOS_DISK_TYPE_REAL 0`: here we define one of our disk types.
- `struct disk`: here we'll create a disk structure that will help us create information and attributes to our disks. for now, it'll be quite simple.
- `PEACHOS_DISK_TYPE type;`: including its type and…
- `int sector_size;`: it's sector size.
- `int disk_read_block(struct disk* idisk, unsigned int lba, int total, void* buf);`: function defintion. we'll see this later!
- `struct disk* disk_get(int index);`: function defintion. we'll see this later!
- `void disk_search_and_init();`: function defintion. we'll see this later! ## 3.2. `disk.c` We've written three functions that will help us abstract the code and make our code more flexible. Let's go over them one by one. ### 3.2.1. `void disk_search_and_init()`

```
struct disk disk;
void disk_search_and_init()
{
	memset(&disk, 0, sizeof(disk));
	disk.type = PEACHOS_DISK_TYPE_REAL;
	disk.sector_size = PEACHOS_SECTOR_SIZE;
}
```

- `struct disk disk;`: we create a structure in which we'll save our disk data.
- `void disk_search_and_init()`: this function will initialize our disks and disk structures.
- `memset(&disk, 0, sizeof(disk));`: here we zero out the structure.
- `disk.type = PEACHOS_DISK_TYPE_REAL;`: here we assign a disk type of `PECHOS_DISK_TYPE_REAL`.
- `disk.sector_size = PEACHOS_SECTOR_SIZE;`: and here we assign the sector size. we haven't seen this constant definition, but know that it is 512. Pretty simple code! Let's continue. ### 3.2.2. `struct disk* disk_get(int index)`

```
struct disk* disk_get(int index)
{
	if(index != 0)
	{
		return 0;
	}
	return &disk;
}
```

- `struct disk* disk_get(int index)`: here we take a disk index. we're only working with disk ID 0, so it'll just return the disk for now.
- `if(index != 0)`: here we check if `index` is not zero.
- `return 0;`: if it's not zero, return 0 as any non zero index is invalid.
- `return &disk;`: and return a pointer to the disk. ### 3.2.3. `int disk_read_block(struct disk* idisk, unsigned int lba, int total, void* buf)`

```
int disk_read_block(struct disk* idisk, unsigned int lba, int total, void* buf)
{
	if(idisk != &disk)
	{
		return -EIO;
	}
	return disk_read_sector(lba, total, buf);
}
```

- `int disk_read_block(struct disk* idisk, unsigned int lba, int total, void* buf)`: this function will be the new way in which programmers will access the disks. it'll take a disk, an `lba` starting block, a `total` amount of sectors to read and a `buf` in which we'll store the data read.
- `if(idisk != &disk)`: here we check if the `idisk` is not the same as the disk we already have. this is for testing purposes and will be changed in the future.
- `return -EIO;`: if not &disk, then return `-EIO`
- `return disk_read_sector(lba, total, buf);`: and return the read sectors using the already available `disk_read_sector`. As you can see, we'll now use the `disk_read_block` function directly instead of calling the `disk_read_sector`. ## 3.3. `config.h` We've also defined some new constants in our `config.h` file.

```
#define PEACHOS_SECTOR_SIZE            512
```

It's just the sector size. That's it for now! # 4. What's a filesystem? - A filesystem is a structure that describes information stored in a disk. - Disks do not have the concept of a file. They are not aware of them. - But the operating system does know about files and uses the filesystem to read and write to or from files. # 4.1. Disks and what we already know. - Disks are gigantic arrays of data that are split into several sectors. - Each sector in a disk is given an LBA (Logical Block Address) number. - But "files" don't exist in the disk in the sense that we know them. They exist as pure data. ## 4.2. Filesystem Structure - A filesystem contains raw data for files. - It contains the filesystem structure header which can explani things such as how many files are no the disk, where the root directory is in relation to its sectors and more. - They way files are laid out on disk is different depending on the filesystem at hand. For example: a "file" will not be the same or have the same structure in FAT32 versus EXT4. They might contain different metadata, different structures and so on. - Without filesystems, we'd we working with sector numbers and we'd have to write down where our files are in each sector by hand. It'd be very painful, and that's why we do filesystems. - Operating systems must have a way to read and understand these filesystems. That's why when you format a USB drive to EXT4 format, many Windows versions will say that the disk is "corrupt" (even though it isn't), because it has no idea what an EXT4 filesystem is and thus can't read from it or understand what's going on in it. ## 4.3. FAT16 (File Allocation Table); 16 bits. - The first sector in this filesystem format is the boot sector on a disk. Fields also exist in this first sector that describe the filesystem, such as how many reserved sectors follow this sector. - Then follows the reserved sectors (if any) amd these sectors are typically ignored by the filesystem. There's a field in the bootsector that tells the operating system to ignore them. This isn't automatic and we must make our OS ignore them manually. - Then we have our first file allocation table. This table contains the values that represents which clusters on the disk are taken and which ones are free for use. A cluster is just a certain number of sectors joined together to represent a cluster. - There might be a second file allocation table, although this one it's optional and depends on whether the FAT6 header in the boot sector is set or not. - Then would come the root directory. This directory explains what files/directories are in the root directory of the filesystem. Each entry has a relative name that represents the file or directory name, attributes such as read or write permissions, address of the first cluster representing the data on the disk and more. - And then we have the data region. Our data is here! :) # 5. Creating a path parser. A path parser function will divide something like `/mnt/file.txt` into `drive, path, file`. Although it sounds simple (and it is), at first it looks and feels like very complex operations. With time, we'll realize that indeed, these are very simple and logical operations. Before getting into the parser itself, we'll move around some functions and create other ones that'll help us write the parser. Let's do it! ## 5.1. `string.h` We've created some new code to be able to validate some stuff, like if a given `char` is a number, convert an ASCII number into an number of `int` type (because 1 is not the same as '1') and a couple more. I've also moved the `memcpy` function we saw before in here.

```
#ifndef STRING_H
#define STRING_H
#include <stddef.h>
#include <stdbool.h>

int strlen(const char* ptr);
```

```
void* memcpy(void* src, void *dst, size_t size);
bool isdigit(char c);
int tonumericdigit(char c);
int strnlen(const char* ptr, int max);

#endif
```

- `int strlen(const char* ptr);`: although we've already made a `strlen` function, we'll move it in here, as to keep everything well organized.
- `void* memcpy(void* src, void *dst, size_t size);`: and this `memcpy` function previously was in the `memory` section, but i've moved it here, since it makes more sense.
- `bool isdigit(char c);`: this function will help us check is a `char` is a digit or not.
- `int tonumericdigit(char c);`: this function will help us convert an ASCII digit into an actual number.
- `int strnlen(const char* ptr, int max);`: this function will do a comparison between the lenght of `ptr` and `max`, and it will return different values if `ptr` is higher or lower than `max`. ## 5.2. `string.c` Now, let's go over the actual code. ### 5.2.1. `int strlen(const char* ptr);`

```
int strlen(const char* ptr)
{
	int len = 0;
	while(*ptr != 0)
	{
		len++;
		ptr += 1;
	}
	return len;
}
```

- `int strlen(const char* ptr)`: function definition. we'll take a `const char* ptr`.
- `int len = 0;`: here we declare an initial `len` variable which we'll represent the lenght of the `ptr` passed on to us.
- `while(*ptr != 0)`: here we check if `*ptr` is NULL terminated or the end of the pointer.
- `len++;`: if not, we increment the lenght.
- `ptr += 1;`: and we also increment the pointer position.
- `return len;`: and we return the lenght. ### 5.2.2. `void* memcpy(void* src, void *dst, size_t size);` We won't go over the `memcpy` code, as it has not changed. I just moved it here. ### 5.2.3. `bool isdigit(char c);`

```
bool isdigit(char c)
{
	return c >= 48 && c <= 57;
}
```

- `bool isdigit(char c)`: here we make a function that will take a `char c`. we will check its ASCII value and determine if it is between 48 and 57, which are integer numbers (check the ASCII table).
- `return c >= 48 && c <= 57;`: here we do a conditional return and check if `c` is higher or equal to 48 (which is a numeric `0`) and if it is lower or equal to 57 (which is numeric `9`). if true, we return true, if not, we return false (1=true, 0=false). ### 5.2.4. `int tonumericdigit(char c);`

```
int tonumericdigit(char c)
{
	return c - 48;
}
```

- `int tonumericdigit(char c)`: function definition. we'll take a `char c` that is supposed to be an ASCII digit.
- `return c - 48;`: here we make a conversion from ASCII digits to decimal digits. take ASCII value 50, which is `2`. 50-48 would equal to 2, which is, well, integer 2. in the ASCII table, `2` is `start of text`, but that doesn't matter to us, because we want the raw decimal value. ### 5.2.5. `int strnlen(const char* ptr, int max);`

```
int strnlen(const char* ptr, int max)
{
	int len = 0;
	for(int i = 0; i < max; i++)
	{
		if(ptr[i] == 0)
		{
			break;
		}
	}
	return len;
}
```

- `int strnlen(const char* ptr, int max)`: as described before, we'll take a `ptr` and a `max` value and we'll use is to count and read a `max` amount of characters from `ptr` and return the `len`.
- `int i = 0;`: here we initialize `i` as integer to `0`.
- `for(i = 0; i < max; i++)`: here we start to go over the `ptr` items with `i`.
- `if(ptr[i] == 0)`: we check if `ptr[i]` is `0` (NULL terminator), and if so…
- `break;`: break. if not, keep counting and increment `i`.
- `return len;`: and return `len`: This is all very simple. Because this, although it will help us make the path parser, isn't the path parser itself :) Let's go over the changes made to `memory.c` before getting our hands ***really*** dirty. ## 5.3. `memory.h` We've added a prototype for `memcmp` and deleted some stuff that should've never been there, like `kzmalloc` and `memcpy`.

```
...
int memcmp(void* s1, void* s2, int count);
...
```

We'll go over `memcmp` right now. ## 5.4. `memory.c` ### 5.4.1. `int memcmp(void* s1, void* s2, int count)`

```
int memcmp(void* s1, void* s2, int count)
{
	char* c1 = s1;
	char* c2 = s2;

	while(count-- > 0)
	{
		if (*c1++ != *c2++)
		{
			return c1[-1] < c2[-1] ? -1 : 1;
		}
	}
	return 0;
}
```

- `int memcmp(void* s1, void* s2, int count)`: function definition. we'll take two pointers and a `count` of bytes to check.
- `char* c1 = s1;`: here we make a pointer to the provided pointer.
- `char* c2 = s2;`: here we make a pointer to the provided pointer.
- `while(count-- > 0)`: here we make a while `count` minus one is higher than zero. `count` will be decremented every time the loop begins again.
- `if (*c1++ != *c2++)`: here we check if `c1++` and `c2++` are not equal. this increments `*c1` and `*c2` every time the `while` loop runs.
- `return c1[-1] < c2[-1] ? -1 : 1;`: and here we make some tertiary operations. if the value of `c1[-1]` is lower than `c2[-1]`, then we return -1, but we return 1 if it's the other way around.
- `return 0;`: and we return 0 if the comparison is OK! For a while I wasn't able to understand that what that tertiary operation was doing. But after a couple minutes of reading, I understand. ## 5.5. `pparser.h` Now we're getting into the interesting stuff. Before reading this section, I recommend reading about linked lists. It's a fundamental data structure and you will need to understand how they work before getting into the section. Nevertheless, you can also read the path parsing notes in the misc section. Let's continue. As always, we won't go over the header guard.

```
struct path_root
{
	int drive_no;
	struct path_part* first;
};

struct path_part
{
	const char* part;
	struct path_part* next;
};

void pathparser_free(struct path_root* root);
struct path_root* pathparser_parse(const char* path, const char* current_directory_path);
```

- `struct path_root`: we create a `path_root`. this will be the root of our filesystem.
- `int drive_no;`: the `root` also contains data on the drive number.
- `struct path_part* first;`: and then it links to a `path_part*` struture `first`, which would be files and directories right after `/`. in UNIX, it could be `bin, home, usr`, etc.
- `struct path_part`: here we define the structure for everything else that is not the root.
- `const char* part;`: here we establish the identifier for that part of the path. it could be, for example, `bin`.

- `struct path_part* next;`: and here we establish a link to the `next` item within that directory, which is the same type as the one we're defining right now.
- `void pathparser_free(struct path_root* root);`: here we set a `pathparser_free` function, but we'll see it later.
- `struct path_root* pathparser_parse(const char* path, const char* current_directory_path);`: same here. we'll see it later on. ## 5.6. `pparser.c` Now we're getting into the MEAT and POTATOES! Let's go!

5.6.1. `static int pathparser_get_drive_by_path(const char** path)`

this function will get the drive number by using the `path` passed onto it.

```
static int pathparser_get_drive_by_path(const char** path)
{
	if(!pathparser_path_valid_format(*path))
	{
		return -EBADPATH;
	}
	int drive_no = tonumericdigit(*path[0]);

	// add 3 bytes to skip drive number
	*path += 3;
	return drive_no;
}
```

- `static int pathparser_get_drive_by_path(const char** path)`: here we'll take a pointer to a `char` pointer, or a pointer to a string. the `static` means that the function won't be changing any values passed onto it.
- `if(!pathparser_path_valid_format(*path))`: here we check if the passed value is a valid format. we'll check this function next.
- `return -EBADPATH;`: if not, return `-EBADPATH` or -4. we'll see this at the end.
- `int drive_no = tonumericdigit(*path[0]);`: here we convert the first element of the char array into a numeric digit from an ASCII digit into a decimal digit.
- `*path += 3;`: here we increment the pointer passed by the user by 3.
- `return drive_no;`: and we return the `drive_no`. The `path` in this case is something like `0:/bin/bash`. When we do `*path += 3`, we are effectively moving the pointer past the `0:/` section of the path. This isn't changing the pointer, it's just incrementing it, as the data contained has not been changed. ### 5.6.2. `static int pathparser_path_valid_format(const char* filename)` This function will validate the path format.

```
static int pathparser_path_valid_format(const char* filename)
{
	int len = strnlen(filename, PEACHOS_MAX_PATH);
	return (len >= 3 && isdigit(filename[0]) && memcmp((void*)&filename[1], ":/", 2) == 0);
}
```

- `static int pathparser_path_valid_format(const char* filename)`: here we take a `*filename`, effectively a path. we won't be changing the values passed onto us.
- `int len = strnlen(filename, PEACHOS_MAX_PATH);`: here we first check if the `len` of `filename` is higher than the maximum path size, which is 108. we'll see this change at the end.
- `return (len >= 3 && isdigit(filename[0]) && memcmp((void*)&filename[1], ":/", 2) == 0);`: here we make a wacky conditional return. we first check if `len` is higher or equal to 3, we then check if the first character is a digit and then we make a `memcmp` between the second and third item in the `filename` array with `:/`. if all of these conditions are met, the path is valid and we return zero. ### 5.6.3. `static struct path_root* path_parser_create_root(int drive_number)` This function will setup the root path structure.

```
static struct path_root* path_parser_create_root(int drive_number)
{
	struct path_root* path_r = kzalloc(sizeof(struct path_root));
	path_r->drive_no = drive_number;
	path_r->first = 0;
	return path_r;
}
```

- `static struct path_root* path_parser_create_root(int drive_number)`: here we take the drive number as an argument.
- `struct path_root* path_r = kzalloc(sizeof(struct path_root));`: now we create a `path_root` structure and we kzalloc

- `path_r->drive_no = drive_number;`: here we access the member `drive_no` of the pointer to the pointer that points to the `path_root` structure and set the `drive_no` item to `drive_number`.
- `path_r->first = 0;`: here we do the same as above. `0` in this case represents the first section of our path, which is `0`, like in `0:/bin/bash`.
- `return path_r;`: and here we return the pointer to the pointer to the struct. ### 5.6.4. `struct path_part* pathparser_parse_path_part(struct path_part* last_part, const char** path)` This function will parse the path part from the `last_part` using the `path`.

```
struct path_part* pathparser_parse_path_part(struct path_part* last_part, const char** path)
{
        const char* path_part_str = pathparser_get_path_part(path);
        if (!path_part_str)
        {
                return 0;
        }

        struct path_part* part = kzalloc(sizeof(struct path_part));
        part->part = path_part_str;
        part->next = 0x00;

        if (last_part)
        {
                last_part->next = part;
        }

        return part;
}
```

- `struct path_part* pathparser_parse_path_part(struct path_part* last_part, const char** path)`: function definition. we'll use this function to parse the path parts that are passed to us by `last_part` and the pointer to the char pointer `path` itself.
- `const char* path_part_str = pathparser_get_path_part(path);`: here we get the path by using the `pathparse_get_path_part`, which we'll see next.
- `if (!path_part_str)`: here we check if the `path_part_str` isn't zero. if it is…
- `return 0;`: we return zero.
- `struct path_part* part = kzalloc(sizeof(struct path_part));`: here we allocate memory for our structure with the size of the structure itself.
- `part->part = path_part_str;`: here we access the `part->part` member of the structure and assign it the `path_part_str` path.
- `part->next = 0x00;`: here we assign `0x00` as a default value; this is because we don't know if there is a next part.
- `if (last_part)`: but if there is a next part…
- `last_part->next = part;`: we assign it to our structure.
- `return part;`: and we return `part`. Kinda complex, I know. Read it all the times you need to understand it. Or mail me! ### 5.6.5. `static const char* pathparser_get_path_part(const char** path)` This function will get the path from the path part.

```
static const char* pathparser_get_path_part(const char** path)
{
        char* result_path_part = kzalloc(PEACHOS_MAX_PATH);
        int i = 0;
        while(**path != '/' && **path != 0x00)
        {
                result_path_part[i] = **path;
                *path += 1;
                i++;
        }
        if (**path == '/')
        {
                // skip / to avoid problems
                *path += 1;
        }

        if (i == 0)
        {
                kfree(result_path_part);
                result_path_part = 0;
        }
        return result_path_part;
}
```

- `static const char* pathparser_get_path_part(const char** path)`: here we take a pointer to a pointer to a char `path`.
- `char* result_path_part = kzalloc(PEACHOS_MAX_PATH);`: here we allocate `result_path_part` with size 108.
- `int i = 0;`: here we initialize a counting variable.
- `while(**path != '/' && **path != 0x00)`: here we check if the element in `path` isn't `/` and if the

element isn't `0x00`.

- `result_path_part[i] = **path;`: while in the loop, we assign `**path` (the element of the pointer to the pointer) to `result_path_part[i]`.
- `*path += 1;`: now we increment the pointer by one.
- `i++;`: and we also increment `i` by one, to populate the enxt item in the `result_path_part`, if there is an item, that is.
- `if (**path == '/')`: after finishing the loop, we check if the element in the `**path` is a `/`. if so…
- `*path += 1;`: we increment if by one, moving past the `/`
- `if (i == 0)`: we also check if `i` is zero. if it is, it means that we got the root or that we were already at the end of the path.
- `kfree(result_path_part);`: and then we free the memory in the `result_path_part` used by the structure.
- `result_path_part = 0;`: and we set the `result:_path_part` to zero.
- `return result_path_part;`: and we return! It's a little bit complex, all of this. I understand if you might feel a bit surpassed by all of this, but don't feel discouraged if you don't understand the code. Go over it, read line by line, use the debugger to go over the code execution, or hit me with an email. I'll find the time to help out :) ### 5.6.6. `struct path_root* pathparser_parse(const char* path, const char* current_directory_path)` This will be the function that the programmer will see. It's a wrapper that uses all our previous functions. It's the last complex function, just a bit more to be done!

```
struct path_root* pathparser_parse(const char* path, const char* current_directory_path)
{
	int res = 0;
	const char* tmp_path = path;
	struct path_root* path_root = 0;
	if (strlen(path) > PEACHOS_MAX_PATH)
	{
		goto out;
	}

	res = pathparser_get_drive_by_path(&tmp_path);
	if (res < 0)
	{
		goto out;
	}

	path_root = path_parser_create_root(res);
	if(!path_root)
	{
		goto out;
	}
	struct path_part* first_part = pathparser_parse_path_part(NULL, &tmp_path);
	if (!first_part)
	{
		goto out;
	}

	path_root->first = first_part;

	struct path_part* part = pathparser_parse_path_part(first_part, &tmp_path);

	while(part)
	{
		part = pathparser_parse_path_part(part, &tmp_path);
	}

out:
	return path_root;
}
```

- `struct path_root* pathparser_parse(const char* path, const char* current_directory_path)`: function definition. we'll use this function to parse the path passed on to us by the user, and in the future we'll take into consideration the `current_directory_path`, but not right now.
- `int res = 0;`: here we set `res` for use later on.
- `const char* tmp_path = path;`: and here we create a pointer to the pointer `path` as to not modify the original pointer.
- `struct path_root* path_root = 0;`: here we initialize the root_path.
- `if (strlen(path) > PEACHOS_MAX_PATH)`: here we check if the size of the `char* path` passed onto us fits in the 108 byte limit we impossed. if `path` is higher than 108…
- `goto out;`: we go to out and return 0.
- `res = pathparser_get_drive_by_path(&tmp_path);`: here we get the drive number. it will also set `res` to the `drive_no`, which we'll use later.
- `if (res < 0)`: here we check if `res` is lower than zero. if it is…

- goto out;: we go to the out label and return 0.
- path_root = path_parser_create_root(res);: here is where we modfiy the actual path_root structure, passing in our drive_no. this will return path_r, which is a path_root* structure with root values set; that is, drive_no is zero and first is 0.
- if(!path_root): here we check if path_root isn't set, and if it isn't…
- goto out;: we go to the out label and return 0.
- struct path_part* first_part = pathparser_parse_path_part(NULL, &tmp_path);: here we initialize the first_part part of our path. we pass NULL because there is no previous part of the path yet. we also pass the &tmb_path (value of tmp_path) to the function. this function will return a path_part structure with populated values of the first item in the path and 0x00 as the next value, since there is no next part of the path yet.
- if (!first_part): here we check if first_part isn't set. if it isn't…
- goto out;: we go to the out label and return 0.
- path_root->first = first_part;: now we can link our path_root with its first part, which is the first_part, a path_part structure that contains it's own value and the next one.
- struct path_part* part = pathparser_parse_path_part(first_part, &tmp_path);: now we'll initialize the next value. it'll we stored in the part pointer.
- while(part): and if part is set…
- part = pathparser_parse_path_part(part, &tmp_path);: we can recursively assign each next part of our linked list.
- out:: out label.
- return path_root;: and finally we return the linked list. the path_root contains all the items from part and whatever they might link to. That was the hardest part, the combination of everything we've written so far. Congratulations if you understood everything! It took me a couple hours to fully get, but I did it! ### 5.6.7. void pathparser_free(struct path_root* root) And this function will free the allocated spaces of heap that we used during the root creation and path parsing.

```
void pathparser_free(struct path_root* root)
{
        struct path_part* part = root->first;
        while(part)
        {
                struct path_part* next_part = part->next;
                kfree((void*)part->part);
                kfree(part);
                part = next_part;
        }
        kfree(root);
}
```

- void pathparser_free(struct path_root* root): function definition. we'll just take the path_root object, since it already contains all the values we need from the linked list after initializing it.
- struct path_part* part = root->first;: here we access the path_part via our path_root root object.
- while(part): and now we get into a loop that will run until part isn't set.
- struct path_part* next_part = part->next;: here we **must** assign the next_part before freeing the memory.
- kfree((void*)part->part);: now we free the part section of our path part
- kfree(part);: and now we can free the structure in its entirety.
- part = next_part;: and now we reassign the part pointer to the next_part pointer that we created previously.
- kfree(root);: after freeing every other member of the linked list, we can free the path_root root pointer. And done! The pathparser_free is way simpler than everything else we've seen so far, but it's an important function to make. Now, we'll continue to see the other minor changes we made to kernel.h and status.h. ## 5.7. kernel.h We've just added some constants. Nothing complex.

```
...
#define PEACHOS_MAX_PATH 108
...
```

- #define PEACHOS_MAX_PATH 108: This constant will determine the maximum lenght of a part of a path. ## 5.8 status.h We also added a new status to the file, which is just a constant.

```
...
#define EBADPATH 4
...
```

- `#define EBADPATH 4`: This status code will be used for bad paths, as we saw below.

5.9. `kernel.c`

This is the least important part. I'll show you how we can use the parser, just for demonstration purposes.

```
    struct path_root* root_path = pathparser_parse("0:/usr/bin/bash", NULL);

    if(root_path)
    {
        print("[+] pathparse yay!");
    }
```

- `struct path_root* root_path = pathparser_parse("0:/usr/bin/bash", NULL);`: here we use the wrapper, passing a path (whatever that might be) and `NULL`, since we don't a current working directory.
- `if(root_path)`: unnecesary, but it's fun to get print messages when stuff works.
- `print("[+] pathparse yay!");`: and print some message.

If we run the kernel, we won't see much happening apart from the `print` message. To check the results, we need to `print root_path` in GDB.

6. Creating a disk stream.

We will implement a way in which we can stop reading from the disk in sectors and start reading from it in bytes. ## 6.1. `streamer.h` We've created a header file for our streamer. We'll just define some structures and prototypes.

```
#ifndef DISKSTREAMER_H
#define DISKSTREAMER_H
#include "disk.h"

struct disk_stream
{
        int pos;
        struct disk* disk;
};

struct disk_stream* diskstreamer_new(int disk_id);
int diskstreamer_seek(struct disk_stream* stream, int pos);
int diskstreamer_read(struct disk_stream* stream, void* out, int total);
void diskstreamer_close(struct disk_stream* stream);


#endif
```

- `struct disk_stream`: `disk_stream` structure definition.
- `int pos;`: the `disk_stream` will have a `pos` value that will be used to know where the streamer is currently located.
- `struct disk* disk;`: and it also has a `disk disk` structure, which will point to a given disk.
- `struct disk_stream* diskstreamer_new(int disk_id);`: prototype. we'll see it later.
- `int diskstreamer_seek(struct disk_stream* stream, int pos);`: prototype. we'll see it later.
- `int diskstreamer_read(struct disk_stream* stream, void* out, int total);`: prototype. we'll see it later.
- `void diskstreamer_close(struct disk_stream* stream);`: prototype. we'll see it later. Now, let's go over the code. ## 6.2. `streamer.c`

6.2.1. `struct disk_stream* diskstreamer_new(int disk_id);`

We will use this function to create a streamer.

```
struct disk_stream* diskstreamer_new(int disk_id)
{
        struct disk* disk = disk_get(disk_id);
        if(!disk)
        {
                return 0;
        }

        struct disk_stream* streamer = kzalloc(sizeof(struct disk_stream));
        streamer->pos = 0;
```

```
        streamer->disk = disk;

        return streamer;
}
```

- `struct disk_stream* diskstreamer_new(int disk_id)`: function definition. we'll receive a disk number (usually 0) as to point the new `disk_stream` to that disk number.
- `struct disk* disk = disk_get(disk_id);`: here we get the disk. now it only returns `0` to non `0` values. if zero, it returns a disk object defined in the `disk.c` file.
- `if(!disk)`: here we check if we got the correct values from `disk_get` call.
- `return 0;`: if we didn't, return 0.
- `struct disk_stream* streamer = kzalloc(sizeof(struct disk_stream));`: here we allocate enough heap space for our `streamer` structure.
- `streamer->pos = 0;`: here we set the initial position of the streamer.
- `streamer->disk = disk;`: and here we set the disk to the disk passed onto us.
- `return streamer;`: and we return the streamer object. ### 6.2.2. `int diskstreamer_seek(struct disk_stream* stream, int pos);` We will use this function to change the `pos` value of the streamer. We'll see the usage of `pos` in the next function.

```
int diskstreamer_seek(struct disk_stream* stream, int pos)
{
        stream->pos = pos;
        return 0;
}
```

- `int diskstreamer_seek(struct disk_stream* stream, int pos)`: function definition. it takes a (previously created) streamer and it's new `pos` value.
- `stream->pos = pos;`: accessing the `pos` member and setting it to `pos` passed by the user.
- `return 0;`: and return. ### 6.2.3. `int diskstreamer_read(struct disk_stream* stream, void* out, int total);` This function will be used to read the data stream into the `out` pointer. We're supposed to pass it a `disk_stream`, the `out` pointer and the `total` amount of bytes. Here, we'll use the `stream->pos` value to read from `pos`.

```
int diskstreamer_read(struct disk_stream* stream, void* out, int total)
{
        int sector = stream->pos / PEACHOS_SECTOR_SIZE;
        int offset = stream->pos % PEACHOS_SECTOR_SIZE;
        char buf[PEACHOS_SECTOR_SIZE];

        int res = disk_read_block(stream->disk, sector, 1, buf);
        if(res < 0)
        {
                goto out;
        }

        int total_to_read = total > PEACHOS_SECTOR_SIZE ? PEACHOS_SECTOR_SIZE : total;
        for(int i = 0; i < total_to_read; i++)
        {
                *(char*)out++ = buf[offset+i];
        }
        stream->pos += total_to_read;
        if(total > PEACHOS_SECTOR_SIZE)
        {
                res = diskstreamer_read(stream, out, total-PEACHOS_SECTOR_SIZE);
        }

out:
        return res;

}
```

- `int diskstreamer_read(struct disk_stream* stream, void* out, int total)`: function definition we'll take a streamer object, a pointer and a `total` amount of bytes to read into the `out` pointer.
- `int sector = stream->pos / PEACHOS_SECTOR_SIZE;`: here we get the sector of the disk. if the value is, for example, `30`, then `30 / PEACHOS_SECTOR_SIZE` returns 0; if 513, then it's `513 / PEACHOS_SECTOR_SIZE` or 1.
- `int offset = stream->pos % PEACHOS_SECTOR_SIZE;`: now we get the offset of the bytes in relation to the sector. this will be used as our starting byte to read from. if `pos` is 30, then `30 % PEACHOS_SECTOR_SIZE` equals 30. and if `pos` 513, then `1`.
  - in the second case it returns 1 because it's indeed the first byte of sector 1 (or second sector, counting from zero).
- `char buf[PEACHOS_SECTOR_SIZE];`: here we assign a buffer of `PEACHOS_SECTOR_SIZE` bytes to read into. we don't care if we're reading less than that, we need a buffer big enough as to not cause overflows.
- `int res = disk_read_block(stream->disk, sector, 1, buf);`: here we use the `disk_read_block`

that we previously made. this will only read the `sector` calculated previously and store the data into `buf`.

- `if(res < 0)`: here we check if the read was successful.
- `goto out;`: if not, return the error.
- `int total_to_read = total > PEACHOS_SECTOR_SIZE ? PEACHOS_SECTOR_SIZE : total;`: this is a ternary assignation we first check if the `total` bytes to read is higher than the `PEACHOS_SECTOR_SIZE` and if so, assign the `PEACHOS_SECTOR_SIZE` to `total_to_read`. if lower, assign the total value.
  - this is a recursive function. if we were to read 514 bytes, the function itself would run within itself. we'll see this in a bit.
- `for(int i = 0; i < total_to_read; i++)`: `for` loop to read the bytes.
- `*(char*)out++ = buf[offset+i];`: tricky pointer magic. here, we cast the `out` pointer into a `char` pointer and then dereferencing the pointer. after that, we assign the `buf[offset+i]` to the current pointer position and finally incrementing it. we assign `buf[offset+i]` because `offset+i` it's the byte that was asked from the function.
  - `out++` and `++out` are different. `out++` increments after assignment and `++out` increments prior to assignment.
- `stream->pos += total_to_read;`: here we increment the `pos` by `total_to_read` bytes. this is because we've already read those bytes.
- `if(total > PEACHOS_SECTOR_SIZE)`: here we check again if `total` is higher than `PEACHOS_SECTOR_SIZE`. if so…
- `res = diskstreamer_read(stream, out, total-PEACHOS_SECTOR_SIZE);`: we call ourselves, but `total` is now subtracted `PEACHOS_TOTAL_SECTORS`. this changes the `total` variable for the next pass of the function, and will do until it becomes 0. this new call also modifies the `out` pointer directly (as we previously did), so there's no need to return `out`.
- `out:`: `out` label.
- `return res;`: here we return `res`. That was a little bit of pointer magic. But we did it. ### 6.2.4. `void diskstreamer_close(struct disk_stream* stream);` And when we're done using the stream, we can call this function to free the memory that it used.

```
void diskstreamer_close(struct disk_stream* stream)
{
        kfree(stream);
}
```

- `void diskstreamer_close(struct disk_stream* stream)`: function definition. it takes a previously created `streamer` object.
- `kfree(stream);`: here we free the `stream` pointer. # 7. File Allocation Table (FAT)
- It's a filesystem developed by Microsoft.
- It consists of a series of clusters of data and a table that determines the state of the clusters.
- The boot sector contains information about the filesystem. ## 7.1. FAT16 Filesystem
- This filesystem uses clusters to represent data, directories and files.
- Each and every cluster uses a fixed amount of sectors which is specified in the boot sector.
- Every file in tha FAT16 filesystem needs to use at least one cluster for its data. This means that a lot of storage is wasted for small files.
- FAT16 can't store files larger than 2 gigabytes without large file support. With large file support, it can hold files up to 4 gigabytes.
- It's very easy to implement. ## 7.2. FAT16 Disk Layout

Name	Size
Boot sector	512 bytes.
Reserved sectors.	fat_header.reserved_sectors * 512*
FAT 1	fat_header.sectors_per_fat * 512
FAT 2 (Optional)	fat_header.sectors_per_fat * 512

Name	Size
Root directory.	fat_header.root_dir_entries * sizeof(struct fat_directory_item) (**rounded if needed**)
Data clusters.	...rest of disk.

7.3. FAT16 Boot Sector

The FAT16 boot sector is basically the Boot Parameter Block. Check the BIOS BPB entry on the misc notes. ## 7.4. FAT16 File Allocation: Clusters in disk. - In the FAT table, we'll find that each cluster is represented as an entry of two bytes. Each two byte represent a cluster in the data region that is taken, free or in use. - Clusters can be chained together. An example would be a file larger than a cluster. It'd use one or more clusters, obviously. - When files use more than one cluster, the first cluster entry will too point to the next cluster in use and so on. The final cluster will contain the value `0xFFFF`, which means the end of the cluster chain. - The size of a cluster is determined by the values in the boot sector Example cluster table:

Entry zero	Entry one	Entry two	Entry three
0x0003	0xFFFF	0xFFFF	0xFFFF
Entry four	Entry five	Entry six	Entry seven
0x0000	0x0000	0x0000	0x0000

We have a cluster chain at entry zero. We know this because the first cluster entry has decimal 3. So, counting from zero, we can see that the final cluster would be located at entry three. *Note: Cluster zero isn't used most of the time. FAT usually starts from cluster three and so forth. ## 7.5. How are clusters accessed? If we wanted to, for example, access the third cluster in the FAT table, we'd take the

```
cluster index * size of cluster
```

If each cluster is 512 bytes in size and we wanted the data from the third cluster, we'd do: `3 * 512` and we'd have to access bytes 1536. ## 7.6. FAT16 Root Directory - It's the top directory, like `C:\\` or `/`. - Directories contain directory entries of a fixed size. ## 7.7. FAT16 Directory Entry A typical directory entry would look like this:

```
struct fat_directory_item
{
    uint8_t filename[8];
    uint8_t ext[3].
    uint8_t attribute;
    uint8_t reserved;
    uint8_t creation_time_tenths_of_a_sec;
    uint16_t creation_time;
    uint16_t creation_date;
    uint16_t last_access;
    uint16_t high_16_bits_first_cluster;
    uint16_t last_mod_time;
    uint16_t last_mod_date;
    uint16_t low_16_bits_first_cluster;
    uint32_t filesize;
} __attribute__((packed));
```

It's a lot of stuff, but everything is quite simple. FYI: the `attributes` item is used to declare the type of file (file or directory), read access, write access and so on. ## 7.8 Iterating through directories - The boot sector contains the maximum number of root directory entries. We should not exceed this value when iterating through the root directory. - We know that when we finish iterating through the root directory or a subdirectory, because the first byte of our filename will be equal to zero. ## 7.9. Directory entry attribute flags

Flag	Description
`0x01`	Read only.
`0x02`	File hidden.
`0x04`	System file. Do not move clusters.

Flag	Description
`0x08`	Volume label.
`0x10`	This is a directory.
`0x20`	Archived.
`0x40`	Device.
`0x80`	Reserved.

7.10. Filenames and extensions.

- The filename is 8 bytes wide and unused bytes are padded with spaces (`0x20`).
- The extension is 3 bytes wide and unused bytes are padded with spaces (`0x20`). ## 7.11. Clusters
- Each cluster represents a fixed amount of sectors in the disk, linearly to each other.
- The amount of sectors that represents a cluster is stored in the boot sector.
- The data clusters section in the filesystem contains all the clusters that make up the subdirectories and file data of files throughout the FAT filesystem. ## 7.12. Tips
- Always use `__attribute__((packed))` when working with structures that are to be stored or read from the disk. The C compiler might try to optimize the structure and change it. We **DO NOT** want this when working with raw data to or from and the disk.
- Learn to use GDB! # 8. Starting our FAT Filesystem. To begin creating our FAT filesystem, we first need to modify the bootloader (`boot.asm`). ## 8.1. `boot.asm` The added code is just added variables in Assembly. ~~Not actually variables, but you understand~~ We did make an important change, which is deleting the `times db 33` we had before. We don't need that anymore, because we'll implement a proper BPB; which is also our FAT table.

```
ORG 0x7c00
BITS 16

CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start

jmp short start
nop

; FAT16 Header
OEMIdentifier           db 'PEACHOS '
BytesPerSector          dw 0x200
SectorsPerCluster       db 0x80
ReservedSectors         dw 200
FATCopies               db 0x02
RootDirEntries          dw 0x40
NumSectors              dw 0x00
MediaType               db 0xF8
SectorsPerFAT           dw 0x100
SectorsPerTrack         dw 0x20
NumberOfHeads           dw 0x40
HiddenSectors           dd 0x00
SectorsBig              dd 0x773544

; Entended BPB
DriveNumber             db 0x80
WinNTBit                db 0x00
Signature               db 0x29
VolumeID                dd 0xD105
VolumeIDString          db 'PEACHOS BOO'
SystemIDString          db 'FAT16   '
```

- `jmp short start`: start of the BPB. we immediately make a `jump short` into the `start` label. the BPB isn't meant to be executed.
- `nop`: as per BPB, we require a `nop` instruction here.
- `OEMIdentifier           db 'PEACHOS '`: this is the OEM Identifier. we can put whatever we want here. there are some old DOS drivers that only work with some OEM ids, but those are far gone. this is an 8 byte value padded with spaces. that is, if we use less that 8 bytes, we must add the spaces.
- `BytesPerSector          dw 0x200`: these is the amount of bytes per *logical* (not physical) sectors. I emphasize *logical* because this parameter doesn't change the physical sectors byte amount, only the logical. that is, what the OS sees. `0x200` is 512 bytes.
- `SectorsPerCluster       db 0x80`: amount of sectors per cluster. we use one of the allowed values `0x80` or 128 sectors per cluster.
- `ReservedSectors         dw 200`: these are reserved sectors, i.e., sectors to be ignored by the kernel. keep in mind that these sectors must be manually ignored.

- `FATCopies` `db 0x02`: number of FAT table copies.
- `RootDirEntries` `dw 0x40`: number of maximum root directory entries in the FAT12 or FAT16 filesystem. this value must be divisible by 16. in our case, we use `0x40` or 64, which is indeed divisible by 16.
- `NumSectors` `dw 0x00`: logical sectors. we won't use this for now.
- `MediaType` `db 0xF8`: this is the media descriptor. `0xF8` describes a partition.
- `SectorsPerFAT` `dw 0x100`: these are the amount of sectors per FAT. we'll use `0x100` or 256.
- `SectorsPerTrack` `dw 0x20`: these are the sectors per track.
- `NumberOfHeads` `dw 0x40`: and these are the number of heads. it's related to CHS geometry, as the entry above.
- `HiddenSectors` `dd 0x00`: these are the hidden sectors. literally just that.
- `SectorsBig` `dd 0x773544`: since we aren't using the `NumSectors`, we must set `SectorsBig`, which is the total amount of sectors in the disk.
- `; Entended BPB`: separator. from now on, we'll see the EBPB or Extended BIOS Parameter Block.
- `DriveNumber` `db 0x80`: this is the drive number. `0x80` means "first physical disk".
- `WinNTBit` `db 0x00`: this byte is resereved.
- `Signature` `db 0x29`: this is the signature. it must be `0x29`.
- `VolumeID` `dd 0xD105`: this is the serial number of the drive.
- `VolumeIDString` `db 'PEACHOS BOO'`: this is the name or label of the drive. it must be 11 bytes, and if less than 11 bytes are used, you need to pad the rest with spaces.
- `SystemIDString` `db 'FAT16   '`: this is the file system type. this is an 8 byte value and it must be padded with spaces.

And that's that! # 8.2. `Makefile` We also modified the `Makefile` a little bit.

```
all: ./bin/kernel.bin ./bin/boot.bin
        ...
        dd if=/dev/zero bs=1048576 count=16 >> ./bin/os.bin
```

- `dd if=/dev/zero bs=1048576 count=16 >> ./bin/os.bin`: we changed the block size to 1048576 (1 megabyte) and the count to 16. # 9. Understanding VFS or Virtual File System.
- The VFS layer allows a kernel to support an infinite amount of filesystems.
- It allows us to abstract complicated low level functions with higher, simpler interfaces.
- It also allows the kernel to load and unload filesystem functionality at will.
- The VFS layer is supposed to be used by all other filesystems. ## 9.1. What happens when we insert a disk?
- The kernel checks for its filesystems and then asks the drive if it has a filesystem it can handle. This process is called resolving the filesystem.
- If the kernel has functionality for that filesystem, it will then load that functionality and binds it to itself.
- Userspace programs use syscalls to the kernel to interact with the drives. There is no direct interaction between the user and the hardware. Take the following example:

```
-> Userspace executes fopen("0:/test.txt", "r")
-> The kernel then parses the call
-> Path parser returns path root.
-> disk_read then talks to the drive
-> the drive calls the `fopen` FAT32 function and returns a file descriptor.
```

10. Implementing the VFS Core Functionality.

10.1. `config.h`

We defined some magic numbers for convenience an readability.

```
// max fs
#define PEACHOS_MAX_FILESYSTEMS        12

// max open files
#define PEACHOS_MAX_FILEDESCRIPTORS    512
```

- `#define PEACHOS_MAX_FILESYSTEMS 12`: we've added a max filesystem number. it's completely arbitrary and can be anything we want.
- `#define PEACHOS_MAX_FILEDESCRIPTORS 512`: and a maximum amount of open files in our system. as before, this is arbitrary and can be whatever we want. ## 10.2. `file.h` sometext

```
#ifndef FILE_H
#define FILE_H
#include "pparser.h"

typedef unsigned int FILE_SEEK_MODE;
enum
{
	SEEK_SET,
	SEEK_CUR,
	SEEK_END,
};

typedef unsigned int  FILE_MODE;
enum
{
	FILE_MODE_READ,
	FILE_MODE_WRITE,
	FILE_MODE_APPEND,
	FILE_MODE_INVALID
};

struct disk;
typedef void*(*FS_OPEN_FUNCTION)(struct disk* disk, struct path_part* path, FILE_MODE mode);
typedef int (*FS_RESOLVE_FUNCTION)(struct disk* disk);

struct filesystem
{
	// fs should return 0 from resolve if the disk is using its fs.
	FS_RESOLVE_FUNCTION resolve;
	FS_OPEN_FUNCTION fopen;
	char name[20];
};

struct file_descriptor
{
	int index;
	struct filesystem* filesystem;
	// private data for internal fd
	void* private;
	// disk the fd is used on
	struct disk* disk;
};

void fs_init();
int fopen(const char* filename, const char* mode);
void fs_insert_filesystem(struct filesystem* filesystem);
struct filesystem* fs_resolve(struct disk* disk);

#endif
```

- `#include "pparser.h"`: we include `pparser.h` to use it later.
- `typedef unsigned int FILE_SEEK_MODE;`: here we define a type `FILE_SEEK_MODE`. it will represent the start of the file, the current relative position or the end of the file.
- `enum`: and with the `enum` keyword, we define the values for `FILE_SEEK_MODE`.
- `SEEK_SET,`: for the start of the file,
- `SEEK_CUR,`: for the current relative position and
- `SEEK_END,`: for the end of the file.
- `typedef unsigned int  FILE_MODE;`: now we define the different file modes in which a file descriptor can be opened with.
- `enum`: again, we define the values for `FILE_MODE`.
- `FILE_MODE_READ,`: for read mode,
- `FILE_MODE_WRITE,`: for write mode,
- `FILE_MODE_APPEND,`: for append (add characters after EOF) and
- `FILE_MODE_INVALID`: for any invalid mode.
- `struct disk;`: here we define a `disk` struct which is globally accessible.
- `typedef void*(*FS_OPEN_FUNCTION)(struct disk* disk, struct path_part* path, FILE_MODE mode);`: this is a function pointer that takes a `disk`, a `path_part` pointer to a file and a `file_mode`.
  - a function pointer is used to point to a function that will be defined by the programmer later on. check the Function Pointers misc page for more.
- `typedef int (*FS_RESOLVE_FUNCTION)(struct disk* disk);`: another function pointer that will take a `disk` function. this resolve function is used to resolve the filesystem implementation as per the filesystem check.
- `struct filesystem`: here we define a `filesystem` struct, which will be used by all the fs drivers that will use this VFS interface to interact with files.

- `FS_RESOLVE_FUNCTION resolve;`: the struct must contain a function to resolve the filesystem and be able (or not) to recognize the filesystem as theirs.
- `FS_OPEN_FUNCTION fopen;`: it'll also have a `fopen` function defined by the driver itself.
- `char name[20];`: and a name of the filesystem, like FAT or EXT.
- `struct file_descriptor`: now we define the `file_descriptor` struct. all the files in our OS will fall under this struct. each filesystem driver will use this struct to return the data read from the filesystem.
- `int index;`: here we have an index, or a position in the disk in which this file is in.
- `struct filesystem* filesystem;`: the `filesystem` the file resides on,
- `void* private;`: a pointer to private data, if any.
- `struct disk* disk;`: and a `disk` struct in which the file is in.
- `void fs_init();`: function prototype. we'll see this later.
- `int fopen(const char* filename, const char* mode);`: function prototype. we'll see this later.
- `void fs_insert_filesystem(struct filesystem* filesystem);`: function prototype. we'll see this later.
- `struct filesystem* fs_resolve(struct disk* disk);`: function prototype. we'll see this later. This is the basis of our VFS layer. As mentioned before, it'll we used by all other filesystems to interface with the OS. ## 10.3. `file.c` Here's the base implementation. There's still a lot to be done, but we'll get there when we get there.

```
#include "file.h"
#include "status.h"
#include "kernel.h"
#include "memory/memory.h"
#include "memory/heap/kheap.h"
#include "config.h"

struct filesystem* filesystems[PEACHOS_MAX_FILESYSTEMS];
struct file_descriptor* file_descriptors[PEACHOS_MAX_FILEDESCRIPTORS];

static struct filesystem** fs_get_free_filesystem()
{
        int i = 0;
        for (i = 0; i < PEACHOS_MAX_FILESYSTEMS; i++)
        {
                return &filesystems[i];
        }

        return 0;
}

void fs_insert_filesystem(struct filesystem *filesystem)
{
        struct filesystem** fs;
        fs = fs_get_free_filesystem();
        if(!fs)
        {
                print("no fs!");
                while(1);
        }
        *fs = filesystem;
}

static void fs_static_load()
{
        //fs_insert_filesystem(fat16_init());
}

void fs_load()
{
        memset(filesystems, 0, sizeof(filesystems));
        fs_static_load();
}

void fs_init()
{
        memset(file_descriptors, 0, sizeof(file_descriptors));
        fs_load();
}

static int file_new_descriptor(struct file_descriptor** desc_out)
{
        int res = -ENOMEM;
        for (int i = 0; i < PEACHOS_MAX_FILEDESCRIPTORS; i++)
        {
                if (file_descriptors[i] == 0)
                {
                        struct file_descriptor* desc = kzalloc(sizeof(struct file_descriptor));
                        desc->index = i+1;
                        file_descriptors[i] = desc;
                        *desc_out = desc;
                        res = 0;
                        break;
                }
        }
        return res;
}

static struct file_descriptor* file_get_descriptor(int fd)
{
        if(fd <= 0 || fd >= PEACHOS_MAX_FILEDESCRIPTORS)
```

```
	{
		return 0;
	}
	// descriptors start at 1
	int index = fd - 1;
	return file_descriptors[index];
}

struct filesystem* fs_resolve(struct disk* disk)
{
	struct filesystem* fs = 0;
	for (int i = 0; i < PEACHOS_MAX_FILESYSTEMS; i++)
	{
		if (filesystems[i] != 0 && filesystems[i]->resolve(disk) == 0)
		{
			fs = filesystems[i];
			break;
		}
	}

	return fs;
}

int fopen(const char* filename, const char* mode)
{
	return -EIO;
}
```

We will the read the code as it would execute in memory. It's a bit much, but not complex. First, we'll go over the initialization code. Then, we'll go over the others. ### 10.3.0. Initial definitions. Before jumping into each function, we need to see the initial structs that are defined by the file.

```
struct filesystem* filesystems[PEACHOS_MAX_FILESYSTEMS];
struct file_descriptor* file_descriptors[PEACHOS_MAX_FILEDESCRIPTORS];
```

- `struct filesystem* filesystems[PEACHOS_MAX_FILESYSTEMS];`: here we define the general `filesystems` structure. it's configured to be of `PEACHOS_MAX_FILESYSTEMS` or 12.
- `struct file_descriptor* file_descriptors[PEACHOS_MAX_FILEDESCRIPTORS];`: and the general `file_descriptors` structure, which is `PEACHOS_MAX_FILEDESCRIPTORS` or 512. ### 10.3.1. void fs_init()

```
void fs_init()
{
	memset(file_descriptors, 0, sizeof(file_descriptors));
	fs_load();
}
```

- `void fs_init()`: `fs_init` function. it'll setup the filesystems as it executes; initially, it cleans up `file_descriptors`.
- `memset(file_descriptors, 0, sizeof(file_descriptors));`: here we `memset` the initial `file_descriptors` to zero.
- `fs_load();`: and we call `fs_load`. ### 10.3.2. void fs_load()

```
void fs_load()
{
	memset(filesystems, 0, sizeof(filesystems));
	fs_static_load();
}
```

- `void fs_load()`: `fs_load` function. here we `memset` the `filesystems` structure and continue executing.
- `memset(filesystems, 0, sizeof(filesystems));`: `memset` call to setup `filesystems` to 0.
- `fs_static_load();`: and we now call `fs_static_load`. ### 10.3.3. static void fs_static_load()

```
static void fs_static_load()
{
	//fs_insert_filesystem(fat16_init());
}
```

- `static void fs_static_load()`: function definition. for now, it doesn't do much.
- `//fs_insert_filesystem(fat16_init());`: we call `fs_insert_filesystem` with `fat16_init` function, which would return a `filesystem` structure. the function hasn't been implemented yet, to its commented out for now. ### 10.3.4. void fs_insert_filesystem(struct filesystem *filesystem)

```
void fs_insert_filesystem(struct filesystem *filesystem)
{
	struct filesystem** fs;
	fs = fs_get_free_filesystem();
	if(!fs)
	{
		print("no fs!");
		while(1);
	}
	*fs = filesystem;
}
```

- `void fs_insert_filesystem(struct filesystem *filesystem)`: function definition. it'll take a `filesystem` structure and then insert them into the `filesystems` structure.
- `struct filesystem** fs;`: here we define a pointer to a pointer to a `filesystem` structure.
- `fs = fs_get_free_filesystem();`: here we get the value address returned by the `fs_get_free_filesystem`. we'll see it later.
- `if(!fs)`: here we check if the `fs` pointer is null. if so…
- `print("no fs!");`: we panic!
- `while(1);`: but we don't have a `panic` function yet. so the system just enters an unescapable `while` loop.
- `*fs = filesystem;`: if not panicking, we dereference the `*fs` pointers (which in turn accesses the `filesystems` structure defined at the beginning of the file) and sets its value to `filesystem`. ### 10.3.5. static struct filesystem** fs_get_free_filesystem()

```
static struct filesystem** fs_get_free_filesystem()
{
	int i = 0;
	for (i = 0; i < PEACHOS_MAX_FILESYSTEMS; i++)
	{
		if(filesystems[i] == 0)
		{
			return &filesystems[i];
		}
	}
	return 0;
}
```

- `static struct filesystem** fs_get_free_filesystem()`: this function will return the memory address value of the first NULL pointer.
- `int i = 0;`: initial counter.
- `for (i = 0; i < PEACHOS_MAX_FILESYSTEMS; i++)`: `for` loop that will run 12 times.
- `if(filesystems[i] == 0)`: here we check if the `filesystems[i]` is NULL. if so…
- `return &filesystems[i];`: we access the memory value of the `filesystems[i]` and return it to the caller, which is the function we saw before.
- `return 0;`: and if anything fails, we return 0. And that's it! Pointers get a little bit tricky, but you'll get the hang of it. Now, we need to read the helper functions. ### 10.3.6. `static int file_new_descriptor(struct file_descriptor** desc_out)`

```
static int file_new_descriptor(struct file_descriptor** desc_out)
{
	int res = -ENOMEM;
	for (int i = 0; i < PEACHOS_MAX_FILEDESCRIPTORS; i++)
	{
		if (file_descriptors[i] == 0)
		{
			struct file_descriptor* desc = kzalloc(sizeof(struct file_descriptor));
			desc->index = i+1;
			file_descriptors[i] = desc;
			*desc_out = desc;
			res = 0;
			break;
		}
	}
	return res;
}
```

- `static int file_new_descriptor(struct file_descriptor** desc_out)`: this function will be used to create file descriptors.
- `int res = -ENOMEM;`: here we create the initial `res` value.
- `for (int i = 0; i < PEACHOS_MAX_FILEDESCRIPTORS; i++)`: here we start enumerating all the file descriptors.
- `if (file_descriptors[i] == 0)`: here we check if `file_descriptors[i]` is NULL or unset.
- `struct file_descriptor* desc = kzalloc(sizeof(struct file_descriptor));`: here we create the initial structure that will be assigned to the file descriptors, allocate memory and zero it out.
- `desc->index = i+1;`: here we increase the `index` value, which would become 1.
- `file_descriptors[i] = desc;`: and we assign the value of `desc` to the found NULL descriptor.
- `*desc_out = desc;`: and we modify the descriptor passed to us to point to the newly created file descriptor.
- `res = 0;`: and we set `res` to 0.
- `break;`: and we break the loop.
- `return res;`: and return `res`. ### 10.3.7. `static struct file_descriptor* file_get_descriptor(int`

`fd)`

```
static struct file_descriptor* file_get_descriptor(int fd)
{
	if(fd <= 0 || fd >= PEACHOS_MAX_FILEDESCRIPTORS)
	{
		return 0;
	}
	// descriptors start at 1, but arrays values start at 0.
	int index = fd - 1;
	return file_descriptors[index];
}
```

- `static struct file_descriptor* file_get_descriptor(int fd)`: this function will the return a file descriptor.
- `if(fd <= 0 || fd >= PEACHOS_MAX_FILEDESCRIPTORS)`: here we check if the `fd` value passed to us is zero or is a value higher than the allowed file descriptors.
- `return 0;`: and we return zero if any of the conditions are met.
- `int index = fd - 1;`: here we create an index value to access the `fd`. if `fd` is one, then we need to access the previous value (`0`) to access the `fd` 1.
- `return file_descriptors[index];`: and we return the file descriptor. ### 10.3.8. `struct filesystem* fs_resolve(struct disk* disk)`

```
struct filesystem* fs_resolve(struct disk* disk)
{
	struct filesystem* fs = 0;
	for (int i = 0; i < PEACHOS_MAX_FILESYSTEMS; i++)
	{
		if (filesystems[i] != 0 && filesystems[i]->resolve(disk) == 0)
		{
			fs = filesystems[i];
			break;
		}
	}

	return fs;
}
```

- `struct filesystem* fs_resolve(struct disk* disk)`: this function will use the `resolve` function pointer to get the fs type.
- `struct filesystem* fs = 0;`: here we make a `fs` pointer to `filesystem` structure to zero (null).
- `for (int i = 0; i < PEACHOS_MAX_FILESYSTEMS; i++)`: we start a loop that will run at most 12 times.
- `if (filesystems[i] != 0 && filesystems[i]->resolve(disk) == 0)`: here we make two checks: first we check if the `filesystems[i]` is NULL and if the execution of the `resolve` function pointer is zero. if so…
- `fs = filesystems[i];`: we set `fs` to `filesystems[i]`.
- `break;`: and break.
- `return fs;`: and we return the `fs` to the caller. ### 10.3.9. `int fopen(const char* filename, const char* mode)`

```
int fopen(const char* filename, const char* mode)
{
	return -EIO;
}
```

- `int fopen(const char* filename, const char* mode)`: we create the skeleton for our `fopen` function.
- `return -EIO;`: we haven't implemented it yet, so we return `-EIO`. ## 10.4. `disk.h` sometext

```
...
struct disk
{
	PEACHOS_DISK_TYPE type;
	int sector_size;
	struct filesystem* filesystem;
};
...
```

- `struct disk`:
- …
- `struct filesystem* filesystem;`: we added the `filessytem` structure to our `disk` structure, as a way to identify the disk. ## 10.5. `disk.c` sometext

```
void disk_search_and_init()
{
	memset(&disk, 0, sizeof(disk));
	disk.type = PEACHOS_DISK_TYPE_REAL;
	disk.sector_size = PEACHOS_SECTOR_SIZE;
```

```
        disk.filesystem = fs_resolve(&disk);
}
```

- `void disk_search_and_init()`:
- `disk.filesystem = fs_resolve(&disk);`: here we make a call to the function `fs_resolve`, which in turn will call the `filesystem->FS_RESOLVE_FUNCTION` function pointer defined in the `file.h` header file ## 10.6. `Makefile` sometext

```
FILES = ... ./build/fs/file.o

all: ./bin/kernel.bin ./bin/boot.bin
    ...
    sudo mount -t vfat ./bin/os.bin ./bin/mountpoint
        echo "hello world!" | sudo tee ./bin/mountpoint/helloworld.txt
        sudo umount ./bin/mountpoint

./build/fs/file.o: ./src/fs/file.c
        i686-elf-gcc $(INCLUDES) -I./src/file/ $(FLAGS) -std=gnu99 -c ./src/fs/file.c -o ./build/fs/file.o
```

- `sudo mount -t vfat ./bin/os.bin ./bin/mountpoint`: here, we mount the `os.bin` file into the `mountpoint`. we can do this because Linux recognizes the `os.bin` as a valid FAT16 partition.
- `echo "hello world!" | sudo tee ./bin/mountpoint/helloworld.txt`: here we write something to the disk. it doesn't matter what it is, we just want to have placeholder data for us to read later on.
- `sudo umount ./bin/mountpoint`: and here we unmount the disk. This is definitely not needed, but it'll help us to later on check if our implementations are working or not. # 11. Implementing FAT16 core functions. Well… we begin implementing our FAT16 driver. We have a lot of changes to cover. So, let's begin. ## 11.1. `string.c` First we need a `strcpy` function. It's quite easy to make. Let's see it.

```
char* strcpy(char* dest, const char* src)
{
        char *res = dest;
        while(*src != 0)
        {
                *dest = *src;
                src += 1;
                dest += 1;
        }
        *dest = 0x00;
        return res;
}
```

- `char* strcpy(char* dest, const char* src)`: function definition. we'll take a `dest` pointer and some text.
- `char *res = dest;`: here we make a pointer to the `dest` pointer.
- `while(*src != 0)`: `while` loop until `*src` is 0.
- `*dest = *src;`: we point the `dest` value to the `src` value.
- `src += 1;`: and we increment the `src` pointer.
- `dest += 1;`: same with `dest`.
- `*dest = 0x00;`: after finishing the loop, we must add a null byte. we do this because the `while` loop stops right at the NULL byte.
- `return res;`: and we return the `res` pointer. we return `res` instead of `dest` because `res` points to `dest`. ## 11.2. `string.h`

```
...
char* strcpy(char* dest, const char* src);
...
```

- `char* strcpy(char* dest, const char* src);`: this is just a prototype of the function we saw before. ## 11.3. `file.c`

```
...
static void fs_static_load()
{
        fs_insert_filesystem(fat16_init());
}
...
```

- `fs_insert_filesystem(fat16_init());`: here we just uncomment this line. ## 11.4. `fat16.h` We begin!

```
#ifndef FAT16_H
#define FAT16_H
#include "fs/file.h"

struct filesystem* fat16_init();

#endif
```

- `struct filesystem* fat16_init();`: function prototype for the `fat16_init` function. ## 11.5. `fat16.c` This is the meat. Here's where, in the future, things might get tricky. For now, we can breath with ease.

```
#include "fat16.h"
#include "status.h"
#include "fs/file.h"
#include "disk/disk.h"
#include "string/string.h"

void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode);
int fat16_resolve(struct disk* disk);

struct filesystem fat16_fs =
{
        .resolve = fat16_resolve,
        .open = fat16_fopen
};

struct filesystem* fat16_init()
{
        strcpy(fat16_fs.name, "FAT16");
        return &fat16_fs;
}

int fat16_resolve(struct disk* disk)
{
        return 0;
}

void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode)
{
        return 0;
}
```

- `void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode);`: we make a prototype of `fat16_fopen` here because if we didnt, the next lines won't compile
- `int fat16_resolve(struct disk* disk);`: same here.
- `struct filesystem fat16_fs =`: and this is the line that *would* cause the issue. nevertheless, it doesn't. we start to create our `fat16_fs` structure.
- `.resolve = fat16_resolve,`: here we pointer the `resolve` function pointer to `fat16_resolve`.
- `.open = fat16_fopen`: and here we point `open` to the `fat16_open` functino.
- `struct filesystem* fat16_init()`: function definition.
- `strcpy(fat16_fs.name, "FAT16");`: here we `strcpy` "FAT16" to the `name` in the struct.
- `return &fat16_fs;`: and here we return our filesystem to the caller.
- `int fat16_resolve(struct disk* disk)`: function definition. we'll take a `disk` struct.
- `return 0;`: we return zero for now. this will make the `file.c` `fs_resolve` function assign our FAT16 filesystem to the disk `0`. in the future, we would add a way to check if the disk is, indeed, fat16. baby steps!
- `void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode)`: function definition. we'd take a `disk`, a `path` and a file mode.
- `return 0;`: we return zero for now. we'll implement this later. ## 11.6. `kernel.c`

```
void kernel_main()
{
    ....
        // fs init
        fs_init();
        // disk init
        disk_search_and_init();
        ...
}
```

- `fs_init();`: here we just call the `fs_init` function.
- `disk_search_and_init();`: this function MUST come after!! if we run this before initializing the filesystem structures, the disk won't have an assigned filesystem and the system will not work properly. ## 11.7. `Makefile`

```
FILES = ... ./build/fs/fat/fat16.o

./build/fs/fat/fat16.o: ./src/fs/fat/fat16.c
        i686-elf-gcc $(INCLUDES) -I./src/fs/ -I./src/fs/fat/ $(FLAGS) -std=gnu99 -c ./src/fs/fat/fat16.c -o ./build/fs/fat/fat16.o
```

Although we have done this before, here we add a new include. Well, not new, but we include the `src/fs` folder instead of just `src/fs/fat`.

And that's it for now! # 12. Implementing FAT16 Structures ### 12.1.1. Constants

```
#define PEACHOS_FAT16_SIGNATURE 0x29
#define PEACHOS_FAT16_FAT_ENTRY_SIZE 0x02
#define PEACHOS_FAT16_BAD_SECTOR 0xFF7
```

```
#define PEACHOS_FAT16_UNUSED 0x00

typedef unsigned int FAT_ITEM_TIPE;
#define FAT_ITEM_TYPE_DIRECTORY 0
#define FAT_ITEM_TYPE_FILE 1

// fat dir entry attribute bitmasks
#define FAT_FILE_READ_ONLY 0x01
#define FAT_FILE_HIDDEN 0x02
#define FAT_FILE_SYSTEM 0x04
#define FAT_FILE_VOLUME_LABEL 0x08
#define FAT_FILE_SUBDIRECTORY 0x10
#define FAT_FILE_ARCHIVED 0x20
#define FAT_FILE_DEVICE 0x40
#define FAT_FILE_RESERVED 0x80
```

- `#define PEACHOS_FAT16_SIGNATURE 0x29`: this is the FAT signature of our filesystem.
- `#define PEACHOS_FAT16_FAT_ENTRY_SIZE 0x02`: this is the size of each FAT entry in bytes.
- `#define PEACHOS_FAT16_BAD_SECTOR 0xFF7`: this is a cluster value that represents a bad sector.
- `#define PEACHOS_FAT16_UNUSED 0x00`: this is a cluster value that represents an unused sector.
- `typedef unsigned int FAT_ITEM_TIPE;`: here we define an item type. we'll use it later on to define if an item is a directory or an item.
- `#define FAT_ITEM_TYPE_DIRECTORY 0`: we'll use this to define a directory.
- `#define FAT_ITEM_TYPE_FILE 1`: and we'll use this to define a file.
- `#define FAT_FILE_READ_ONLY 0x01`: this attribute represents a read only file.
- `#define FAT_FILE_HIDDEN 0x02`: this attribute represents a hidden file.
- `#define FAT_FILE_SYSTEM 0x04`: this attribute represents a system file and it must not be moved or deleted.
- `#define FAT_FILE_VOLUME_LABEL 0x08`: this attribute represents a volume label.
- `#define FAT_FILE_SUBDIRECTORY 0x10`: this attribute represents a subdirectory.
- `#define FAT_FILE_ARCHIVED 0x20`: this attribute represents a backed up (archived) file.
- `#define FAT_FILE_DEVICE 0x40`: this attribute represents a device file and it must not be changed.
- `#define FAT_FILE_RESERVED 0x80`: this attribute represents the reserved clusters/sectors in disk. ### 12.1.2. `struct fat_header_extended`

```
struct fat_header_extended
{
	uint8_t drive_number;
	uint8_t win_nt_bit;
	uint8_t signature;
	uint32_t volume_id;
	uint8_t volume_id_string[11];
	uint8_t systemd_id_string[8];
} __attribute__((packed));
```

- `struct fat_header_extended`: this structure is a C implementation of the extended BPB. check `boot.asm`.
- `uint8_t drive_number;`: this is the drive number.
- `uint8_t win_nt_bit;`: this is the WIN NT bit.
- `uint8_t signature;`: this is the signature of the FS.
- `uint32_t volume_id;`: this is the volume id.
- `uint8_t volume_id_string[11];`: this is the volume ID string.
- `uint8_t systemd_id_string[8];`: and this is the system ID, `FAT16`.
- `} __attribute__((packed));`: we pack this structure because it'll be used in the disk. ### 12.1.3. `struct fat_header`

```
struct fat_header
{
	uint8_t short_jmp_ins[3];
	uint8_t oem_identifier[8];
	uint16_t bytes_per_sector;
	uint8_t sectors_per_cluster;
	uint16_t reserved_sectors;
	uint8_t fat_copies;
	uint16_t root_dir_entries;
	uint16_t number_of_sectors;
	uint8_t media_type;
	uint16_t sectors_per_fat;
	uint16_t sectors_per_track;
	uint16_t number_of_heads;
	uint32_t hidden_sectors;
	uint32_t sectors_big;
} __attribute__((packed));
```

- `struct fat_header`: as before, this is a C implementation of the BPB. check `boot.asm`.
- `uint8_t short_jmp_ins[3];`: this is the `jump short main` instruction.

- `uint8_t oem_identifier[8];`: this is the OEMIdentifier.
- `uint16_t bytes_per_sector;`: this are the bytes per logical sector.
- `uint8_t sectors_per_cluster;`: this is the sectors per cluster (unused).
- `uint16_t reserved_sectors;`: this is the amount of reserved sectors we have.
- `uint8_t fat_copies;`: this is the amount of FAT copies that exist in the FAT table.
- `uint16_t root_dir_entries;`: this is the amount of root directory entries.
- `uint16_t number_of_sectors;`: this is number of logical sectors we have.
- `uint8_t media_type;`: this is the media type.
- `uint16_t sectors_per_fat;`: this is the number of sectors per FAT table.
- `uint16_t sectors_per_track;`: this is the number of sectors per track.
- `uint16_t number_of_heads;`: this is the number of drive heads.
- `uint32_t hidden_sectors;`: this is the amount of hidden or reserved sectors.
- `uint32_t sectors_big;`: this is the amount of sectors we actually have.
- `} __attribute__((packed));`: and we pack it up. ### 12.1.4. `struct fat_h`

```
struct fat_h
{
        struct fat_header primary_header;
        union fat_h_e
        {
                struct fat_header_extended extended_header;
        } shared;
};
```

- `struct fat_h`: this is the internal representation that we'll use of the FAT headers.
- `struct fat_header primary_header;`: this is the BPB.
- `union fat_h_e`: here we make a union. a union is a special data type that will allow us to store different data types in the same memory location. in practice, it's similar to a structure. this union will only be used if an EBPB is found.
- `struct fat_header_extended extended_header;`: here we set the extended BPB as a member of this union.
- `} shared;`: and `shared` is the name of the union. it's the member name that we'll use to access it. ### 12.1.5. `struct fat_directory_item`

```
struct fat_directory_item
{
        uint8_t filename[8];
        uint8_t ext[3];
        uint8_t attribute;
        uint8_t reserved;
        uint8_t creation_time_tenths_of_a_sec;
        uint16_t creation_time;
        uint16_t creation_date;
        uint16_t last_access;
        uint16_t high_16_bits_first_cluster;
        uint16_t last_mod_time;
        uint16_t last_mod_date;
        uint16_t low_16_bits_first_cluster;
        uint32_t filesize;
} __attribute__((packed));
```

- `struct fat_directory_item`: this structure will represent each entry in the FAT table, be it a directory or a file.
- `uint8_t filename[8];`: filename.
- `uint8_t ext[3];`: extension of the file.
- `uint8_t attribute;`: attributes (look above)
- `uint8_t reserved;`: this is a reserved bit, unused.
- `uint8_t creation_time_tenths_of_a_sec;`: this is the creation time in miliseconds.
- `uint16_t creation_time;`: this is the time itself.
- `uint16_t creation_date;`: this is the creation date.
- `uint16_t last_access;`: this is the last access date.
- `uint16_t high_16_bits_first_cluster;`: this are the high 16 bits of the cluster. if it's a directory, it'll represent the next entry in the directory structure. if it's a file, it'll have cluster data, position, etc.
- `uint16_t last_mod_time;`: this is the last modification time.
- `uint16_t last_mod_date;`: this is the last modification date.
- `uint16_t low_16_bits_first_cluster;`: this are the lower 16 bits.
- `uint32_t filesize;`: and this is the filesize.

- `} __attribute__((packed));`: we pack it up! ### 12.1.6. `struct fat_directory`

```
struct fat_directory
{
	struct fat_directory_item* item;
	int total;
	int sector_pos;
	int ending_sector_pos;
};
```

- `struct fat_directory`: this is a directory entry.
- `struct fat_directory_item* item;`: here we'll point to the items in the directory. they might be a file or directories.
- `int total;`: this is the total number of items in the directory.
- `int sector_pos;`: this is the sector position, or where in the disk the directory is at.
- `int ending_sector_pos;`: and the last sector (if more than one). ### 12.1.7. `struct fat_item`

```
struct fat_item
{
	union
	{
		struct fat_directory_item* item;
		struct fat_directory* directory;
	};
	FAT_ITEM_TIPE type;
};
```

- `struct fat_item`: this is the file entry.
- `union`: we create a union.
- `struct fat_directory_item* item;`: here, we might point to a file or…
- `struct fat_directory* directory;`: a directory.
- `FAT_ITEM_TIPE type;`: and the type, be it file or folder. ### 12.1.8. `struct fat_item_descriptor`

```
struct fat_item_descriptor
{
	struct fat_item* item;
	uint32_t pos;
};
```

- `struct fat_item_descriptor`: this is the file descriptor of the item.
- `struct fat_item* item;`: here we point to the item.
- `uint32_t pos;`: and here the position in which it exists. ### 12.1.9. `struct fat_private`

```
struct fat_private
{
	struct fat_h header;
	struct fat_directory root_directory;
	// used to stream data clusters
	struct disk_stream* cluster_read_stream;
	// used to stream the FAT table
	struct disk_stream* fat_read_stream;
	struct disk_stream* directory_stream;
};
```

- `struct fat_private`: here we define the private data of the FAT filesystem. this data is not supposed to be displayed.
- `struct fat_h header;`: here we set the FAT header.
- `struct fat_directory root_directory;`: here the root directory.
- `struct disk_stream* cluster_read_stream;`: we create a filestream to read clusters.
- `struct disk_stream* fat_read_stream;`: we create a filestream to read the FAT table.
- `struct disk_stream* directory_stream;`: we create a filestream to read the directory streams. For now, it isn't THAT complex. It looks like a lot of code, but it is what is it. FAT16 is one of the simplest filesystems there is, so prepare yourself :)

13. Implementing the FAT16 resolver function

This chapter is one of the most complex so far. From here, we'll put to the test everything we've made so far. Our path parser, our I/O functions, our boot FAT sector, our previously made FAT structures, disk streamers and more. As a matter of fact, a couple of thing I made previously were wrong. I mistyped some bytes and did a couple fuck ups. Those we'll fix here, too. But we'll do those at the very end. First, let's go over the FAT resolver implementation. ### 13.1.1. `disk.h` Here we did a minor change to the `disk` structure. We've added a disk `id`.

```
struct disk
{
        PEACHOS_DISK_TYPE type;
        int sector_size;
        struct filesystem* filesystem;
        int id;
        void* fs_private;
};
```

- `int id`: this `id` will hold the disk ID associated with the disk. ### 13.1.2. `disk.c` Since we've modified the disk structure, we also need to reflect the changes in the disk initialization routine.

```
void disk_search_and_init()
{
        memset(&disk, 0, sizeof(disk));
        disk.type = PEACHOS_DISK_TYPE_REAL;
        disk.sector_size = PEACHOS_SECTOR_SIZE;
        disk.filesystem = fs_resolve(&disk);
        disk.id = 0;
}
```

- `disk.id = 0;`: here we'll set 0 for now. Remember that as of right now, we don't have a way to detect disks and other stuff. Just disk 0. ## fat16.c Here's where we get into the resolver function. We'll go over it step by step, from order of execution in the main `fat16_resolve` function. ### 13.2.1. `static void fat16_init_private(struct disk* disk, struct fat_private* private)` Similar to all our other `init` functions, this function will initialize the private structures of our FAT filesystems. This data contains the FAT header, FAT extended header, data streams and other stuff. We initialize the streams, as of right now. We'll get the sector information later on.

```
static void fat16_init_private(struct disk* disk, struct fat_private* private)
{
        memset(private, 0, sizeof(struct fat_private));
        private->cluster_read_stream = diskstreamer_new(disk->id);
        private->fat_read_stream = diskstreamer_new(disk->id);
        private->directory_stream = diskstreamer_new(disk->id);
}
```

- `static void fat16_init_private(struct disk* disk, struct fat_private* private)`: function definition. we'll take a disk and a previously defined `fat_private` structure.
- `memset(private, 0, sizeof(struct fat_private));`: here we memset the entire data region to zero.
- `private->cluster_read_stream = diskstreamer_new(disk->id);`: here we initialize the cluster stream.
- `private->fat_read_stream = diskstreamer_new(disk->id);`: here we initialize the FAT reading stream.
- `private->directory_stream = diskstreamer_new(disk->id);`: here we initialize the directory stream. ### 13.2.2. `int fat16_get_root_directory(struct disk* disk, struct fat_private* fat_private, struct fat_directory* directory)` Now here we're starting to get serious. The `fat16_get_root_directory` function will do several calls to get the root directory and other fun stuff.

```
int fat16_get_root_directory(struct disk* disk, struct fat_private* private, struct fat_directory* directory)
{
        int res = 0;
        struct fat_header* primary_header = &private->header.primary_header;
        int root_dir_sector_pos = (primary_header->fat_copies * primary_header->sectors_per_fat) + primary_header->reserved_sectors;
        int root_directory_entries = private->header.primary_header.root_dir_entries;
        int root_directory_size = (root_directory_entries * sizeof(struct fat_directory_item));
        int total_sectors = root_directory_size / disk->sector_size;
        if(root_directory_size % disk->sector_size)
        {
                total_sectors += 1;
        }
        int total_items = fat16_get_total_items_per_directory(disk, root_dir_sector_pos);

        struct fat_directory_item* dir = kzalloc(sizeof(root_directory_size));
        if(!dir)
        {
                res = -ENOMEM;
                goto out;
        }
        struct disk_stream* stream = private->directory_stream;
        if(diskstreamer_seek(stream, fat16_sector_to_absolute(disk, root_dir_sector_pos)) != PEACHOS_ALLOK)
        {
                res = -EIO;
                goto out;
        }
        if(diskstreamer_read(stream, dir, root_directory_size) != PEACHOS_ALLOK)
        {
                res = -EIO;
                goto out;
        }

        directory->item = dir;
        directory->total = total_items;
```

```
	directory->sector_pos = root_dir_sector_pos;
	directory->ending_sector_pos = root_dir_sector_pos + (root_directory_size / disk->sector_size);

out:
	return res;
}
```

- `int fat16_get_root_directory(struct disk* disk, struct fat_private* private, struct fat_directory* directory)`: function definition. we'll take a disk, a fat_private and a directory structure. the directory structure is inside the `fat_private` struct.
- `int res = 0;`: here we set the initial return value.
- `struct fat_header* primary_header = &private->header.primary_header;`: here we create a pointer to the `primary_header` (FAT header). it'll be easier to work with it this way.
- `int root_dir_sector_pos = (primary_header->fat_copies * primary_header->sectors_per_fat) + primary_header->reserved_sectors;`: here we calculate the root directory by multiplying the amount of FAT table copies with the sectors per FAT value (256 or `0x100`). this is then added to the amount of `reserved_sectors`. this formula will give us the exact place in which the `root_directory` should be positioned.
- `int root_directory_entries = private->header.primary_header.root_dir_entries;`: here we get the amount of root directory entries from the primary header.
- `int root_directory_size = (root_directory_entries * sizeof(struct fat_directory_item));`: here we get the total directory size by multiplying the `root_directory_entries` with the size of the `fat_directory_item` struct.
- `int total_sectors = root_directory_size / disk->sector_size;`: here we get the total amount of sectors used by the `root_directory_size` (bytes value) and dividing it by the logical `sector_size` defined in our `disk`.
- `if(root_directory_size % disk->sector_size)`: here we check if the `root_directory_size` needs alignment. if so…
- `total_sectors += 1;`: we add `1` to the total sector count.
- `int total_items = fat16_get_total_items_per_directory(disk, root_dir_sector_pos);`: here we call the `fat16_get_total_items` function. we'll see that function right after this one. in short: it returns the amount of *stuff* in the root directory. be it files or directories.
- `struct fat_directory_item* dir = kzalloc(sizeof(root_directory_size));`: here we create and allocate zero'd out memory for the `root_directory_size` size.
- `if(!dir)`: here we check if the allocation failed. if so…
- `res = -ENOMEM;`: return `-ENOMEM`.
- `goto out;`: and go to the `out` label.
- `struct disk_stream* stream = private->directory_stream;`: here we access the previously initialized `directory_stream`
- `if(diskstreamer_seek(stream, fat16_sector_to_absolute(disk, root_dir_sector_pos)) != PEACHOS_ALLOK)`: here we `seek` to the `fat16_sector_to_absolute` byte, which we'll see later on. for now, just keep in mind that we're multiplying the sector value contained in `root_dir_sector_pos` to the byte value usable by the `diskstreamer_read`. if this call fails…
- `res = -EIO;`: we return `-EIO`
- `goto out;`: and we go to the `out` label.
- `if(diskstreamer_read(stream, dir, root_directory_size) != PEACHOS_ALLOK)`: here we read the bytes set in the `diskstreamer_seek` function, which size is `root_directory_size` into the `dir` buffer. if this call fails…
- `res = -EIO;`: we return `-EIO`
- `goto out;`: and we go to the `out` label.
- `directory->item = dir;`: here we set the `directory` item to `dir`.
- `directory->total = total_items;`: here we set the `total` to `total_items`.
- `directory->sector_pos = root_dir_sector_pos;`: here we set the `sector_pos` to `root_dir_sector_pos`.
- `directory->ending_sector_pos = root_dir_sector_pos + (root_directory_size / disk->sector_size);`: and the `ending_sector_pos` is a calculation in which we sum the ending sector position with the `root_directory_size` divided by the sector size in our `disk`.
- `out:`: `out` label.

- `return res;`: and here we return res, which, if everything went well, will be zero. ### 13.2.3. `int fat16_get_total_items_for_directory(struct disk* disk, uint32_t directory_start_sector)` Here we'll see how we can calculate the amount of *stuff* held in a directory.

```
int fat16_get_total_items_per_directory(struct disk* disk, uint32_t directory_start_sector)
{
	struct fat_directory_item item;
	struct fat_directory_item empty_item;
	memset(&empty_item, 0, sizeof(empty_item));

	struct fat_private* private = disk->fs_private;

	int res = 0;
	int i = 0;
	int directory_start_pos = directory_start_sector * disk->sector_size;
	struct disk_stream* stream = private->directory_stream;
	if(diskstreamer_seek(stream, directory_start_pos) != PEACHOS_ALLOK)
	{
		res = -EIO;
		goto out;
	}

	while(1)
	{
		if(diskstreamer_read(stream, &item, sizeof(item)) != PEACHOS_ALLOK)
		{
			res = -EIO;
			goto out;
		}
		if (item.filename[0] == 0x00)
		{
			break;
		}
		if (item.filename[0] == 0xE5)
		{
			continue;
		}
		i++;
	}
	res = i;

out:
	return res;
}
```

- `int fat16_get_total_items_per_directory(struct disk* disk, uint32_t directory_start_sector)`: here we'll get the `disk` and the starting sector from which we'll get the directory total.
- `struct fat_directory_item item;`: here we define a `fat_directory_item`.
- `struct fat_directory_item empty_item;`: here we define another one, but it won't be used.
- `memset(&empty_item, 0, sizeof(empty_item));`: here we memset the empty directory to zero.
- `struct fat_private* private = disk->fs_private;`: here we access the private data contained by our filesystem.
- `int res = 0;`: we set the `res` return value.
- `int i = 0;`: and a counter, which we'll use to count how many *things* are in a directory.
- `int directory_start_pos = directory_start_sector * disk->sector_size;`: here we get the initial position in bytes by multiplying the `directory_start_sector` with the disk's logical `sector_size`.
- `struct disk_stream* stream = private->directory_stream;`: here we access the previously initialized `directory_stream stream`.
- `if(diskstreamer_seek(stream, directory_start_pos) != PEACHOS_ALLOK)`: here we `seek` to the `directory_start_pos`. if something goes wrong…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to `out`.
- `while(1)`: now, we'll need an infinite loop to read over the items we have in the directories.
- `if(diskstreamer_read(stream, &item, sizeof(item)) != PEACHOS_ALLOK)`: here we read from the previous `seek` byte into `&item` with a size of `item`. `&item`, after each iteration, will grow in size. if the read function goes wrong…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to `out`.
- `if (item.filename[0] == 0x00)`: here we check if the `filename` starts with null bytes. if so, we have read over everything.
- `break;`: and we break the loop.
- `if (item.filename[0] == 0xE5)`: if `0xE5`, we've found something.
- `continue;`: we continue.
- `i++;`: and we add an item to the `i` counter.

- `res = i;`: here we set `res` to `i`
- `out:`: `out` label.
- `return res;`: and we return `res`. ### 13.2.4. `int fat16_sector_to_absolute(struct disk* disk, int sector)` This function is quite simple, as previously stated.

```
int fat16_sector_to_absolute(struct disk* disk, int sector)
{
	return sector * disk->sector_size;
}
```

- `int fat16_sector_to_absolute(struct disk* disk, int sector)`: function definition. we'll take a disk and a sector.
- `return sector * disk->sector_size;`: we return the sector multiplied by the `sector_size`. ### 13.2.5. `int fat16_resolve(struct disk* disk)` Here's where everything before comes together. Not really. At least not for now.

```
int fat16_resolve(struct disk* disk)
{
	int res = 0;
	struct fat_private* fat_private = kzalloc(sizeof(struct fat_private));
	fat16_init_private(disk, fat_private);

	disk->fs_private = fat_private;
	disk->filesystem = &fat16_fs;

	struct disk_stream* stream = diskstreamer_new(disk->id);
	if(!stream)
	{
		res = -ENOMEM;
		goto out;
	}
	if(diskstreamer_read(stream, &fat_private->header, sizeof(fat_private->header)) != PEACHOS_ALLOK)
	{
		res = -EIO;
		goto out;
	}

	if (fat_private->header.shared.extended_header.signature != 0x29)
	{
		res = -EFSNOTUS;
		goto out;
	}

	if (fat16_get_root_directory(disk, fat_private, &fat_private->root_directory) != PEACHOS_ALLOK)
	{
		res = -EIO;
		goto out;
	}

out:
	if(stream)
	{
		diskstreamer_close(stream);
	}

	if(res < 0)
	{
		kfree(fat_private);
		disk->fs_private = 0;
	}

	return res;
}
```

- `int fat16_resolve(struct disk* disk)`: function definition. we'll just take a disk.
- `int res = 0;`: we set the initial return value.
- `struct fat_private* fat_private = kzalloc(sizeof(struct fat_private));`: here we allocate some memory on the heap for the `fat_private` data structure.
- `fat16_init_private(disk, fat_private);`: here we initialize the `fat_private`, as seen previously.
- `disk->fs_private = fat_private;`: here we set the `disk` private data to the `fat_private`.
- `disk->filesystem = &fat16_fs;`: and here we assign the filesystem to us, for now.
- `struct disk_stream* stream = diskstreamer_new(disk->id);`: here we create a new stream.
- `if(!stream)`: if the stream wasn't created successfully…
- `res = -ENOMEM;`: we return `-ENOMEM`.
- `goto out;`: and go to `out`.
- `if(diskstreamer_read(stream, &fat_private->header, sizeof(fat_private->header)) != PEACHOS_ALLOK)`: by not using `seek`, we're reading at byte 0 into the `&fat_private->header` with a size of `fat_private->header`. if something happened…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to `out`.
- `if (fat_private->header.shared.extended_header.signature != 0x29)`: to check that the

filesystem is indeed us, we'll read the `fat_private->header.shared.extended_header.signature` value, which should be `0x29`. if not…
- `res = -EFSNOTUS;`: we return `-EFSNOTUS`.
- `goto out;`: and go to `out`.
- `if (fat16_get_root_directory(disk, fat_private, &fat_private->root_directory) != PEACHOS_ALLOK)`: and here we get the root directory. and set it to `&fat_private->root_directory`. if something goes wrong…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to `out`.
- `out:`: `out` label.
- `if(stream)`: here we check if `stream` is opened. if so…
- `diskstreamer_close(stream);`: we close it.
- `if(res < 0)`: and here we check fi `res` is less than zero. if so, something went wrong, and we should…
- `kfree(fat_private);`: free the memory used by `fat_private`
- `disk->fs_private = 0;`: and reset the `fs_private` data.
- `return res;`: and we return `res`. which, if zero, is OK; if not, NOT OK. And that's it! We have working resolver function. Not really. If you were following step by step, you'll have issues. So, let's fix them. ## 13.3. Fixing the fuckups. ### 13.3.1. `io.asm` So when making the `io.asm` labels, I mistyped some registers.

```
insw:
    ...
-       in al, dx
+       in ax, dx
    ...
```

Yes. We were reading byte (singular) instead of bytes (plural) in an `insw`, where we should be reading TWO bytes. I debugged this for like 4 hours. ### 13.3.1. `boot.asm` Here I also mistyped some bytes.

```
-SectorsBig             dd 0x773544
+SectorsBig             dd 0x773594
```

Yes. Instead of a `9`, I wrote a `4`. As before, it took me HOURS to find this. But hey, you live you learn. ### 13.3.1. `disk.c` And finally…

```
-       outb(0x1F4, (unsigned char) lba >> 8);
-       outb(0x1F4, (unsigned char) lba >> 16);
+       outb(0x1F4, (unsigned char)(lba >> 8));
+       outb(0x1F5, (unsigned char)(lba >> 16));
```

The major bug was only using the `0x1F4` interface to send the LBA values. I NEEDED to use the `0x1F5` too, alongside `0x1F3`. Nevertheless, it's working now.

14. Implementing the VFS `fopen`.

Now, we need to implement the basis of our `fopen` function at the VFS layer. This doesn't mean that we have a working FAT read function, no. We're making the structure for our virtual filesystem layer to be able to handle the calls to FAT16 and any other filesystem that might use it. ## 14.1. `string.c` Before doing the `fopen` basic structure, we need a couple tools for that. ### 14.1.1. `int strncmp(const char* str1, const char* str2, int n)` We'll implement a simple `string compare` function.

```
int strncmp(const char* str1, const char* str2, int n)
{
        unsigned char u1, u2;
        while(n-- > 0)
        {
                u1 = (unsigned char)*str1++;
                u2 = (unsigned char)*str2++;
                if (u1 != u2)
                {
                        return u1 - u2;
                }
                if (u1 == '\\0')
                {
                        return 0;
                }
        }
        return 0;
}
```

- `int strncmp(const char* str1, const char* str2, int n)`: here we'll get two strings to compare and an `n` number of bytes to compare.

- `unsigned char u1, u2;`: here we'll use two unsigned chars to work with the `str{1,2}` as pointers.
- `while(n-- > 0)`: here `while` loop until `n--` (reading all but one) value isn't zero.
- `u1 = (unsigned char)*str1++;`: we set `u1` to the first character of `*str1` to it and we increment `str1`.
- `u2 = (unsigned char)*str2++;`: we set `u2` to the first character of `*str2` to it and we increment `str2`.
- `if (u1 != u2)`: if `u1` and `u2` aren't equal…
- `return u1 - u2;`: we return `u1 - u2`, which isn't zero.
- `if (u1 == '\\0')`: we check if we arrived at a NULL terminator. if so…
- `return 0;`: we return zero.
- `return 0;`: and we return zero here in case the `while` loop finishes before the NULL terminator. ### 14.1.2. `int strnlen_terminator(const char* str, int max, char terminator)` This function will be used to check the lenght of strings with a `max` amount of characters to read and checking for a `terminator` and breaking at it.

```
int strnlen_terminator(const char* str, int max, char terminator)
{
        int i = 0;
        for(i = 0; i < max++; i++)
        {
                if (str[i] == '\\0' || str[i] == terminator)
                {
                        break;
                }
        }
        return i;
}
```

- `int strnlen_terminator(const char* str, int max, char terminator)`: function definition. we'll get a string, a `max` bytes to read and a `terminator` value.
- `int i = 0;`: here we set the initial counter to zero.
- `for(i = 0; i < max; i++)`: here we'll run a `for` loop until `max`.
- `if (str[i] == '\\0' || str[i] == terminator)`: we check if if value at `str[i]` is a NULL terminator or the `terminator` value. if so…
- `break;`: we break.
- `return i;`: and here we return the number of characters read until the `terminator` or the NULL byte. ### 14.1.3. `char tolower(char s1)` Since the cringe FAT filesystem does not distinguish between upper and lower case, we need a way to make everything lowercase.

```
char tolower(char s1)
{
        if(s1 >= 65 && s1 <= 90)
        {
                s1 += 32;
        }
        return s1;
}
```

- `char tolower(char s1)`: function definition. we'll take a single char.
- `if(s1 >= 65 && s1 <= 90)`: here we check if the `char` is between `65` and `90`. check the ASCII table to see why :) if the char is between these values…
- `s1 += 32;`: we increment it by 32, effectively making it lowercase.
- `return s1;`: and we return `s1`. ### 14.1.4. `int istrncmp(const char* s1, const char* s2, int n)` We'll also need an case insensitive `strncmp`.

```
int istrncmp(const char* s1, const char* s2, int n)
{
        unsigned char u1, u2;
        while(n-- > 0)
        {
                u1 = (unsigned char)*s1++;
                u2 = (unsigned char)*s2++;
                if(u1 != u2 && tolower(u1) != tolower(u2))
                {
                        return u1 - u2;
                }
                if (u1 == '\\0')
                {
                        return 0;
                }
        }
        return 0;
}
```

- `int istrncmp(const char* s1, const char* s2, int n)`: as before, we'll take two strings and an `n` amount of bytes to compare.

- `unsigned char u1, u2;`: we make two unsigned chars to use `s1` and `s2` as pointers.
- `while(n-- > 0)`: `while` loop that continually decrements `n` as it goes until it's lower than zero.
- `u1 = (unsigned char)*s1++;`: here we set `u1` to the first character of the `s1` casted pointer and we increment it.
- `u2 = (unsigned char)*s2++;`: here we set `u2` to the first character of the `s2` casted pointer and we increment it.
- `if(u1 != u2 && tolower(u1) != tolower(u2))`: we check if `u1` and `u2` are not the same and if converting them to lowercase is also not the same. fi so…
- `return u1 - u2;`: we return `u1` minus `u2`.
- `if (u1 == '\\0')`: and if we hit a null terminator…
- `return 0;`: we return zero.
- `return 0;`: and we also return zero outside, in case we don't hit a null terminator. ## 14.2. `string.h` Here we'll just put our prototypes.

```
int strncmp(const char* str1, const char* str2, int n);
int strnlen_terminator(const char* str, int max, char terminator);
int istrncmp(const char* s1, const char* s2, int n);
char tolower(char s1);
```

14.3. `file.c`

Before actually getting into the `fopen` function, we need to check something else hehe. ### 14.3.1. `FILE_MODE file_get_mode_by_string(const char* str)` We'll make a function that converts our `mode_string` (more on that later) into an anctual `FILE_MODE` type.

```
FILE_MODE file_get_mode_by_string(const char* str)
{
	FILE_MODE mode = FILE_MODE_INVALID;
	if(strncmp(str, "r", 1) == 0)
	{
		mode = FILE_MODE_READ;
	}
	if(strncmp(str, "w", 1) == 0)
	{
		mode = FILE_MODE_WRITE;
	}
	if(strncmp(str, "a", 1) == 0)
	{
		mode = FILE_MODE_APPEND;
	}

	return mode;

}
```

- `FILE_MODE file_get_mode_by_string(const char* str)`: here we'll receive the `mode_string`.
- `FILE_MODE mode = FILE_MODE_INVALID;`: we'll set the initial return value to `FILE_MODE_INVALID`.
- `if(strncmp(str, "r", 1) == 0)`: here we check if the initial byte of the `mode_string` is `r`. if so…
- `mode = FILE_MODE_READ;`: we set `FILE_MODE_READ.`
- `if(strncmp(str, "w", 1) == 0)`: here we check if the initial byte of the `mode_string` is `w`. if so…
- `mode = FILE_MODE_WRITE;`: we set `FILE_MODE_WRITE.`
- `if(strncmp(str, "a", 1) == 0)`: here we check if the initial byte of the `mode_string` is `a`. if so…
- `mode = FILE_MODE_APPEND;`: we set `FILE_MODE_APPEND.`
- `return mode;`: and we return the mode. ### 14.3.2. `int fopen(const char* filename, const char* mode_string)` And finally! The `fopen` basis! Let's see it!

```
int fopen(const char* filename, const char* mode_string)
{
	int res = 0;
	struct path_root* root_path = pathparser_parse(filename, NULL);
	if(!root_path)
	{
		res = -EINVARG;
		goto out;
	}
	if(!root_path->first)
	// if 0:/ and not 0:/file.txt...
	{
		res = -EINVARG;
		goto out;
	}

	struct disk* disk = disk_get(root_path->drive_no);
	// if 1:/...
	if(!disk)
	{
		res = -EIO;
		goto out;
	}
	if(!disk->filesystem)
```

```
	{
		res = -EIO;
		goto out;
	}

	FILE_MODE mode = file_get_mode_by_string(mode_string);
	if(mode == FILE_MODE_INVALID)
	{
		res = -EINVARG;
		goto out;
	}
	void* descriptor_private_data = disk->filesystem->open(disk, root_path->first, mode);
	if(ISERR(descriptor_private_data))
	{
		res = ERROR_I(descriptor_private_data);
		goto out;
	}

	struct file_descriptor* desc = 0;
	res = file_new_descriptor(&desc);
	if(res < 0)
	{
		goto out;
	}
	desc->filesystem = disk->filesystem;
	desc->private = disk->fs_private;
	desc->disk = disk;
	res = desc->index;

out:
	// fopen shouldnt return negative values.
	if(res < 0)
	{
		res = 0;
	}
	return res;
}
```

- `int fopen(const char* filename, const char* mode_string)`: we'll take a filename (path) and a mode.
- `int res = 0;`: here we set the initial return value.
- `struct path_root* root_path = pathparser_parse(filename, NULL);`: here we'll make a `path_root` using the `pathparser_parse` using the provided `filename` and without a current working directory (NULL).
- `if(!root_path)`: if something went wrong with the parsing…
  - something that could've went wrong is that the user passed "0:/" and no file.
- `res = -EINVARG;`: we return `-EINVARG`.
- `goto out;`: and we go to `out`.
- `if(!root_path->first)`: here we check if the `root_path` has a next element, that is, that we're not pointing at the root directory itself. if so…
- `res = -EINVARG;`: we return `-EINVARG`.
- `goto out;`: and we go to `out`.
- `struct disk* disk = disk_get(root_path->drive_no);`: here we get the disk structure found in the `root_path`.
- `if(!disk)`: we check if the disk not zero. if so…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to `out`.
- `if(!disk->filesystem)`: here we check if the `disk` structure has a `filesystem` associated with it. if it doesn't…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to `out`.
- `FILE_MODE mode = file_get_mode_by_string(mode_string);`: here we get the file open mode with the previously described function.
- `if(mode == FILE_MODE_INVALID)`: here we check if the `mode` is invalid. if so…
- `res = -EINVARG;`: we return `-EINVARG`.
- `goto out;`: and we go to `out`.
- `void* descriptor_private_data = disk->filesystem->open(disk, root_path->first, mode);`: here we initialize a void pointer that will hold the value of the `open` function pointer.
- `if(ISERR(descriptor_private_data))`: here we check `ISERR`. we'll see what this is later on. know for now that it's a macro that checks if the value is less than zero. if it is…
- `res = ERROR_I(descriptor_private_data);`: we res `res` to `ERROR_I`, which is just a macro that casts the error value to an integer.

- `goto out;`: and we go to `out`.
- `struct file_descriptor* desc = 0;`: here we create a new file descriptor.
- `res = file_new_descriptor(&desc);`: here we initialize the file descriptor structures.
- `if(res < 0)`: here we check if the `res` value is less than zero. if so…
- `goto out;`: we go to `out`.
- `desc->filesystem = disk->filesystem;`: here we set the new file descriptor's filesystem to the one we have in the `disk` structure.
- `desc->private = disk->fs_private;`: here we set the new file descriptor's private data to the one we have in the `disk` structure.
- `desc->disk = disk;`: and we set the file descriptor's `disk` to the `disk` strucutre.
- `res = desc->index;`: and we set `res` to the newly created file descriptor index.
- `out:`: `out` label.
- `if(res < 0)`: we check if `res` is lower than zero. if so, no file descriptor was created and…
- `res = 0;`: we set it to zero.
- `return res;`: and we return. In this last two chapters, we've really made use of everything we've made thus far. Path parsers, disk streamers, disk initailizations, heap implementation, and more. Awesome, right? ## 14.4.1. `kernel.h` We've defined some new macros in the `kernel.h` header.

```
#define ERROR(valud) (void*)(value)
#define ERROR_I(value) (int)(value)
#define ISERR(value) ((int)value < 0)
```

- `#define ERROR(valud) (void*)(value)`: here we take a value and we cast it to a void pointer.
- `#define ERROR_I(value) (int)(value)`: here we take a value and we cast it to an integer.
- `#define ISERR(value) ((int)value < 0)`: and here we take a value and return true or false depending on if `value` is less than zero or not. # 15. Implementing the FAT16 fopen function. ### 15.1.-1. typo in `kernel.h` (valud) to (value) When we defined the error macros, I made a typo. It's a quick fix, though.

```
-#define ERROR(valud) (void*)(value)
+#define ERROR(value) (void*)(value)
```

15.1.0. `fat_item_descriptor` to `fat_file_descriptor`

And we need to do a little name change to the `fat_item_descriptor` structure and call it `fat_file_descriptor`. Simple as that! ### 15.1.1. `struct fat_item* fat16_get_directory_entry(struct disk* disk, struct path_part* path)` Well gentlemen, this is the moment. Prepare yourselves. Cuz shit is gonna get whacky.

```
struct fat_item* fat16_get_directory_entry(struct disk* disk, struct path_part* path)
{
        struct fat_private* fat_private = disk->fs_private;
        struct fat_item* current_item = 0;
        struct fat_item* root_item
                = fat16_find_item_in_directory(disk, &fat_private->root_directory, path->part);

        if(!root_item)
        {
                goto out;
        }

        struct path_part* next_part = path->next;
        current_item = root_item;
        while(next_part != 0)
        {
                if(current_item->type != FAT_ITEM_TYPE_DIRECTORY)
                {
                        current_item = 0;
                        break;
                }
                struct fat_item* tmp_item = fat16_find_item_in_directory(disk, current_item->directory, next_part->part);
                fat16_fat_item_free(current_item);
                current_item = tmp_item;
                next_part = next_part->next;
        }
out:
        return current_item;

}
```

- `struct fat_item* fat16_get_directory_entry(struct disk* disk, struct path_part* path)`: function definition. we'll take a disk and a path_path structs. this function will get us the directory entry from the FAT table.

- `struct fat_private* fat_private = disk->fs_private;`: here we make a struct pointer to the private data.
- `struct fat_item* current_item = 0;`: we create an empty `fat_item*` struct.
- `struct fat_item* root_item = fat16_find_item_in_directory(disk, &fat_private->root_directory, path->part);`: now we get the first (root) item in the directory. we'll see this function right after this one.
- `if(!root_item)`: here we check if the previous call was successful. if not…
- `goto out;`: go out out!
- `struct path_part* next_part = path->next;`: now we initialize a struct that points to the `next` entry in the `path` argument.
- `current_item = root_item;`: we set the `current_item` to the `root_item`.
- `while(next_part != 0)`: we'll run a `while` loop until we have no next entries.
- `if(current_item->type != FAT_ITEM_TYPE_DIRECTORY)`: we check if the current item is not a directory. if so, we don't need to keep iterating, since we've found the last item already.
- `current_item = 0;`: we set the `current_item` to zero.
- `break;`: and we break out of the loop.
- `struct fat_item* tmp_item = fat16_find_item_in_directory(disk, current_item->directory, next_part->part);`: if we got a directory, we define a `tmp_item` `fat_item` structure that calls `fat16_find_item_in_directory`, passing the `next_part->part` and the `current_item->directory`. we'll see this function later on.
- `fat16_fat_item_free(current_item);`: when we called `fat16_find_item_in_directory`, memory was allocated to `root_item`. we free it here. remember, `current_item` holds the values held by `root_item`.
- `current_item = tmp_item;`: now we set the current item to `tmp_item`.
- `next_part = next_part->next;`: we set the `next_part` to it's `next` member, if any.
- `out:`: `out` label.
- `return current_item;`: and we finally return the `current_item`. ### 15.1.2. `struct fat_item* fat16_find_item_in_directory(struct disk* disk, struct fat_directory* directory, const char* name)` This function is used to get an item in a given directory.

```
struct fat_item* fat16_find_item_in_directory(struct disk* disk, struct fat_directory* directory, const char* name)
{
	struct fat_item* f_item = 0;
	char tmp_filename[PEACHOS_MAX_PATH];
	for(int i = 0; i < directory->total; i++)
	{
		fat16_get_full_relative_filename(&directory->item[i], tmp_filename, sizeof(tmp_filename));
		if(istrncmp(tmp_filename, name, sizeof(tmp_filename)) == 0)
		{
			f_item = fat16_new_fat_item_for_directory_item(disk, &directory->item[i]);
		}
	}
	return f_item;
}
```

- `struct fat_item* fat16_find_item_in_directory(struct disk* disk, struct fat_directory* directory, const char* name)`: function definition. we'll be called with a disk, fat_directory structures and the name of the file.
- `struct fat_item* f_item = 0;`: here we initialize `f_item`.
- `char tmp_filename[PEACHOS_MAX_PATH];`: and here we make a temporary array of size `PEACHOS_MAX_PATH` (108).
- `for(int i = 0; i < directory->total; i++)`: we'll search over the entire directory for a file.
- `fat16_get_full_relative_filename(&directory->item[i], tmp_filename, sizeof(tmp_filename));`: here we get the file's full relative name. we'll see this function in a bit.
- `if(istrncmp(tmp_filename, name, sizeof(tmp_filename)) == 0)`: we check if `istrncmp` return zero when comparing `tmp_filename` and `name`.
- `f_item = fat16_new_fat_item_for_directory_item(disk, &directory->item[i]);`: if it's a match, we'll get a new FAT item for the directory item for the given item.
- `return f_item;`: and we return `f_item`. ### 15.1.3. `void fat16_get_full_relative_filename(struct fat_directory_item* item, char* out, int max_len)` With this function, we'll get the full relative filename of a given `fat_directory_item` and pass it to `out`.

```
void fat16_get_full_relative_filename(struct fat_directory_item* item, char* out, int max_len)
```

```
{
        memset(out, 0x00, max_len);
        char* out_tmp = out;
        fat16_to_proper_string(&out_tmp, (const char*) item->filename);
        if(item->ext[0] != 0x00 && item->ext[0] != 0x20)
        {
                *out_tmp++ = '.';
                fat16_to_proper_string(&out_tmp, (const char*) item->ext);
        }

}
```

- `void fat16_get_full_relative_filename(struct fat_directory_item* item, char* out, int max_len)`: function definition. we'll take a `fat_directory_item*`, a char out pointer and a max length.
- `memset(out, 0x00, max_len);`: first we fill the pointer with zeroes until `max_len`.
- `char* out_tmp = out;`: we create a temporary pointer to out.
- `fat16_to_proper_string(&out_tmp, (const char*) item->filename);`: here we'll load the proper string from the FAT table into the `out_tmp` pointer. we'll see this function more in depth in a bit.
- `if(item->ext[0] != 0x00 && item->ext[0] != 0x20)`: we check if the extension isn't null and that the extension isn't a space character either.
- `*out_tmp++ = '.';`: we add a `.` (dot) to the name.
- `fat16_to_proper_string(&out_tmp, (const char*) item->ext);`: and now we add the extension. ### 15.1.4. `void fat16_to_proper_string(char** out, const char* in)` As mentioned previously, this function is used to get the proper string name of a file or directory from the FAT table.

```
void fat16_to_proper_string(char** out, const char* in)
{
        while(*in != 0x00 && *in != 0x20)
        {
                **out = *in;
                *out += 1;
                in += 1;
        }
        if(*in == 0x20)
        {
                **out = 0x00;
        }

}
```

- `void fat16_to_proper_string(char** out, const char* in)`: function definition. we'll a pointer to a C style string pointer and an `in`, which is a C style string.
- `while(*in != 0x00 && *in != 0x20)`: here we check that we aren't at a space character or at a NULL terminator.
- `**out = *in;`: we now set the value from `in` to the `out` pointer.
- `*out += 1;`: now we increment the pointer by one.
- `in += 1;`: and we also increment `in` by one.
- `if(*in == 0x20)`: we check if the `in` pointer has a space. if so…
- `**out = 0x00;`: terminate the filename and return to the caller. ### 15.1.5. `struct fat_item* fat16_new_fat_item_for_directory_item(struct disk* disk, struct fat_directory_item* item)` This function creates a FAT item structure out of the FAT directory items found.

```
struct fat_item* fat16_new_fat_item_for_directory_item(struct disk* disk, struct fat_directory_item* item)
{
        struct fat_item* f_item = kzalloc(sizeof(struct fat_item));
        if(!f_item)
        {
                return 0;
        }

        if(item->attribute & FAT_FILE_SUBDIRECTORY)
        {
                f_item->directory = fat16_load_fat_directory(disk, item);
                f_item->type = FAT_ITEM_TYPE_DIRECTORY;
        }
        f_item->type = FAT_ITEM_TYPE_FILE;
        f_item->item = fat16_clone_directory_item(item, sizeof(struct fat_directory_item));
        return f_item;

}
```

- `struct fat_item* fat16_new_fat_item_for_directory_item(struct disk* disk, struct fat_directory_item* item)`: function definition. it takes a disk and a fat_directory_item structures.
- `struct fat_item* f_item = kzalloc(sizeof(struct fat_item));`: first we allocate memory for

the fat_item that we'll pass return.

- if(!f_item): here we check if the f_item was allocated or not. if not…
- return 0;: we return zero.
- if(item->attribute & FAT_FILE_SUBDIRECTORY): now we check if the item passed onto us has the subdirectory attributes. we use an AND operation because it'll return true if the attribute member has the bitmask FAT_FILE_SUBDIRECTORY. and if so…
- f_item->directory = fat16_load_fat_directory(disk, item);: we load a FAT directory. we'll see this function right after this one.
- f_item->type = FAT_ITEM_TYPE_DIRECTORY;: and we set the type of the item to a directory.
- f_item->type = FAT_ITEM_TYPE_FILE;: if the condition wasn't met, we then set the item type to a file.
- f_item->item = fat16_clone_directory_item(item, sizeof(struct fat_directory_item));: and we clone the item. we do this because we don't know if we have memory allocated for it on our heap. we'll see this function after fat16_load_fat_directory.
- return f_item;: and we return the item. ### 15.1.6. struct fat_directory* fat16_load_fat_directory(struct disk* disk, struct fat_directory_item* item) As said before, this function loads the directory from the FAT into the struct.

```
struct fat_directory* fat16_load_fat_directory(struct disk* disk, struct fat_directory_item* item)
{
        int res = 0;
        struct fat_directory* directory = 0;
        struct fat_private* private = disk->fs_private;
        if (!(item->attribute & FAT_FILE_SUBDIRECTORY))
        {
                res = -EINVARG;
                goto out;
        }

        directory = kzalloc(sizeof(struct fat_directory));
        if (!directory)
        {
                res = -ENOMEM;
                goto out;
        }

        int cluster = fat16_get_first_cluster(item);
        int cluster_sector = fat16_cluster_to_sector(private, cluster);
        int total_items = fat16_get_total_items_per_directory(disk, cluster_sector);
        directory->total = total_items;
        int directory_size = directory->total * sizeof(struct fat_directory_item);
        directory->item = kzalloc(directory_size);
        if(!directory->item)
        {
                res = -ENOMEM;
                goto out;
        }
        res = fat16_read_internal(disk, cluster, 0x00, directory_size, directory->item);
        if (res != PEACHOS_ALLOK)
        {
                goto out;
        }


out:
        if (res != PEACHOS_ALLOK)
        {
                fat16_free_directory(directory);
        }
        return directory;
}
```

- struct fat_directory* fat16_load_fat_directory(struct disk* disk, struct fat_directory_item* item): function definition. we'll take a disk and a fat_directory_item.
- int res = 0;: here we set an initial return value.
- struct fat_directory* directory = 0;: we initialize a fat_directory.
- struct fat_private* private = disk->fs_private;: and we access the fs_private.
- if (!(item->attribute & FAT_FILE_SUBDIRECTORY)): here we check if the item doesn't have the subdirectory attribute. if not…
- res = -EINVARG;: we set the return value to -EINVARG.
- goto out;: and we goto out.
- directory = kzalloc(sizeof(struct fat_directory));: now we zero allocate memory for the structure.
- if (!directory): now we check for allocation issues. if something happened…
- res = -ENOMEM;: we throw an -ENOMEM.
- goto out;: and return.

- `int cluster = fat16_get_first_cluster(item);`: now we get the first cluster of the item. we'll see this function after `fat16_clone_directory_item`.
- `int cluster_sector = fat16_cluster_to_sector(private, cluster);`: and now we get the sector value of the cluster.
- `int total_items = fat16_get_total_items_per_directory(disk, cluster_sector);`: and we get the total amount of items in the directory using the sector value of the cluster.
- `directory->total = total_items;`: and we set the total items of the directory.
- `int directory_size = directory->total * sizeof(struct fat_directory_item);`: now we get the total size of the directory. we multiply the `total` value with the size of `fat_directory_item`.
- `directory->item = kzalloc(directory_size);`: now we allocate memory for the items.
- `if(!directory->item)`: now we check for allocation issues. if something happened…
- `res = -ENOMEM;`: we throw an `-ENOMEM`.
- `goto out;`: and return.
- `res = fat16_read_internal(disk, cluster, 0x00, directory_size, directory->item);`: and we go to the `fat16_read_internal`. this is one of the main and most important function of the `fat16_fopen` code. we'll see it at the end.
- `if (res != PEACHOS_ALLOK)`: we check if `res` isn't ALLOK. if not…
- `goto out;`: we go to out.
- `out:`: `out` label.
- `if (res != PEACHOS_ALLOK)`: we check `res` again. if not OK…
- `fat16_free_directory(directory);`: we free the directory. we'll see this at the end, too.
- `return directory;`: and we return the directory. ### 15.1.7. `struct fat_directory_item* fat16_clone_directory_item(struct fat_directory_item* item, int size)` This function clones the `fat_directory_item` structure passed to it and returns a new, although equal (of *n* bytes), structure to the caller.

```
struct fat_directory_item* fat16_clone_directory_item(struct fat_directory_item* item, int size)
{
        struct fat_directory_item* item_copy = 0;
        if (size < sizeof(struct fat_directory_item))
        {
                return 0;
        }
        item_copy = kzalloc(size);
        if (!item_copy)
        {
                return 0;
        }
        memcpy(item_copy, item, size);

        return item_copy;

}
```

- `struct fat_directory_item* fat16_clone_directory_item(struct fat_directory_item* item, int size)`: function definition.
- `struct fat_directory_item* item_copy = 0;`: here we create the item copy, which is the value we'll return.
- `if (size < sizeof(struct fat_directory_item))`: we check if the size is less than the `fat_directory_item` struct. if it was less than the FAT directory item struct, then we couldn't clone it, since it wouldn't fit.
- `return 0;`: and we return 0.
- `item_copy = kzalloc(size);`: now we allocate the memory in the heap and zero it out.
- `if (!item_copy)`: we check for memory allocation issues. if any…
- `return 0;`: we return 0.
- `memcpy(item_copy, item, size);`: now we copy the `item` into the `item_copy` structure.
- `return item_copy;`: and we return a new, memory allocated and equal `fat_directory_item` to the caller.

15.1.8. `static uint32_t fat16_get_first_cluster(struct fat_directory_item* item)`

As mentioned previously and as stated in the function name, this function looks up the cluster of the `fat_directory_item` structure passed to it.

```
static uint32_t fat16_get_first_cluster(struct fat_directory_item* item)
```

```
{
        return (item->high_16_bits_first_cluster) | item->low_16_bits_first_cluster;
}
```

- `static uint32_t fat16_get_first_cluster(struct fat_directory_item* item)`: function definition.
- `return (item->high_16_bits_first_cluster) | item->low_16_bits_first_cluster;`: we take both high and low bits of the first cluster and join them with an `OR` operation.
  - this is wrong. we'll we see why this is wrong later on. try to think about why it's wrong for now :) (ps: it's related to bit alignment) ### 15.1.9. `static int fat16_cluster_to_sector(struct fat_private* private, int cluster)` This function will take the cluster byte address and convert it to a sector value. The struct access gets kinda crazy, but when you get the hang of it, it ain't that hard.

```
static int fat16_cluster_to_sector(struct fat_private* private, int cluster)
{
    return private->root_directory.ending_sector_pos + ((cluster - 2) * private->header.primary_header.sectors_per_cluster);
}
```

- `static int fat16_cluster_to_sector(struct fat_private* private, int cluster)`: function definition. we'll take the `cluster` from the previous function, and the `fat_private`, which can be passed onto us by the `disk->fs_private` member.
- `return private->root_directory.ending_sector_pos + ((cluster - 2) * private->header.primary_header.` and we return the `ending_sector_pos` plus the `cluster - 2` multiplied by the `sectors_per_clusters`.
  - it isn't really specified the reason as to why we do `cluster - 2` in the official spec. nevertheless, i think this is related to the first and second `sectors` in the disk. which, remember, are the boot sector and the FAT tables. ### 15.1.10. `static int fat16_read_internal(struct disk* disk, int starting_cluster, int offset, int total, void* out)` This function is THE BEGINNING OF THE END… not really. But it's going to get more complex as we continue. Let's go!

```
static int fat16_read_internal(struct disk* disk, int starting_cluster, int offset, int total, void* out)
{
    struct fat_private* private = disk->fs_private;
    struct disk_stream* stream = private->cluster_read_stream;
    return fat16_read_internal_from_stream(disk, stream, starting_cluster, offset, total, out);
}
```

- `static int fat16_read_internal(struct disk* disk, int starting_cluster, int offset, int total, void* out)`: function definition. this is one of the main functions, since here's where we'll do the actual reading.
- `struct fat_private* private = disk->fs_private;`: we access the `fs_private`.
- `struct disk_stream* stream = private->cluster_read_stream;`: and the `cluster_read_stream`.
- `return fat16_read_internal_from_stream(disk, stream, starting_cluster, offset, total, out);`: and we call the actual read, `fat16_read_internal_from_stream`. ### 15.1.11. `static int fat16_read_internal_from_stream(struct disk* disk, struct disk_stream* stream, int cluster, int offset, int total, void* out)` Let's continue.

```
static int fat16_read_internal_from_stream(struct disk* disk, struct disk_stream* stream, int cluster, int offset, int total, void* out)
{
        int res = 0;
        struct fat_private* private = disk->fs_private;
        int size_of_cluster_bytes = private->header.primary_header.sectors_per_cluster * disk->sector_size;
        int cluster_to_use = fat16_get_cluster_for_offset(disk, cluster, offset);
        if (cluster_to_use < 0)
        {
                res = cluster_to_use;
                goto out;
        }

        int offset_from_cluster = offset % size_of_cluster_bytes;
        int starting_sector = fat16_cluster_to_sector(private, cluster_to_use);
        int starting_pos = (starting_sector * disk->sector_size) * offset_from_cluster;
        int total_to_read = total > size_of_cluster_bytes ? size_of_cluster_bytes : total;

        res = diskstreamer_seek(stream, starting_pos);
        if(res != PEACHOS_ALLOK)
        {
                goto out;
        }
        res = diskstreamer_read(stream, out, total_to_read);
        if (res != PEACHOS_ALLOK)
        {
                goto out;
        }
        total -= total_to_read;
        if (total > 0)
        {
                res = fat16_read_internal_from_stream(disk, stream, cluster, offset+total_to_read, total, out + total_to_read);
        }
out:
        return res;
```

```
}
```

- `static int fat16_read_internal_from_stream(struct disk* disk, struct disk_stream* stream, int cluster, int offset, int total, void* out)`: this function is quite similar to the one we implemented in the `diskstreamer_read` code. you'll see why in just a second.
- `int res = 0;`: initial return value.
- `struct fat_private* private = disk->fs_private;`: accessing the `fs_private`.
- `int size_of_cluster_bytes = private->header.primary_header.sectors_per_cluster * disk->sector_size;`: now we get the total amount of bytes contained in a single cluster.
- `int cluster_to_use = fat16_get_cluster_for_offset(disk, cluster, offset);`: we calculate the cluster offset. we'll see this function right after this one.
- `if (cluster_to_use < 0)`: we check if it's less than zero. if it is…
- `res = cluster_to_use;`: we set `res` to the `cluster_to_use` value.
- `goto out;`: and we go to out.
- `int offset_from_cluster = offset % size_of_cluster_bytes;`: now we calculate the offset for the seek function. it's as simple as using the mod operand with both `offset` (passed in the function args) and the `size_of_cluster_bytes`.
- `int starting_sector = fat16_cluster_to_sector(private, cluster_to_use);`: here we calculate the starting sector. we use the cluster to sector function for this.
- `int starting_pos = (starting_sector * disk->sector_size) * offset_from_cluster;`: now we get the starting position. we do this by multiplying the sector size by the starting sector, and then multiplying it by the offset.
- `int total_to_read = total > size_of_cluster_bytes ? size_of_cluster_bytes : total;`: ternary operation. if the total is higher than the cluster byte size, we'll use the cluster byte size. if not, we'll use the total.
  - we do this because we can read only one cluster at a time.
- `res = diskstreamer_seek(stream, starting_pos);`: now we seek to the starting position.
- `if(res != PEACHOS_ALLOK)`: we check the status of the seek. if something went wrong..
- `goto out;`: we go to out.
- `res = diskstreamer_read(stream, out, total_to_read);`: now we read.
- `if (res != PEACHOS_ALLOK)`: if something went wrong while reading…
- `goto out;`: we go to out.
- `total -= total_to_read;`: now we substract the `total_to_read` value from `total`.
- `if (total > 0)`: if total is HIGHER than zero…
- `res = fat16_read_internal_from_stream(disk, stream, cluster, offset+total_to_read, total, out + total_to_read);`: we recurse and run again! check `diskstreamer_read`, it's kinda similar :)
- `out:`: `out` label.
- `return res;`: and we return! ### 15.1.12. `static int fat16_get_cluster_for_offset(struct disk* disk, int starting_cluster, int offset)` Hey! Congratulations if you've got this far. We just have two more hard function to go! Keep going! This function will help us get the cluster for offset.

```
static int fat16_get_cluster_for_offset(struct disk* disk, int starting_cluster, int offset)
{
        int res = 0;
        struct fat_private* private = disk->fs_private;
        int size_of_cluster_bytes = private->header.primary_header.sectors_per_cluster * disk->sector_size;
        int cluster_to_use = starting_cluster;
        int clusters_ahead = offset / size_of_cluster_bytes;
        for (int i = 0; i < clusters_ahead; i++)
        {
                int entry = fat16_get_fat_entry(disk, cluster_to_use);
                if (entry == 0xFF8 || entry == 0xFFF)
                {
                        // last entry in file
                        res = -EIO;
                        goto out;
                }
                // is sector bad?
                if (entry == PEACHOS_FAT16_BAD_SECTOR)
                {
                        res = -EIO;
                        goto out;
                }
                if (entry == 0xFF0 || entry == 0xFF6)
                {
```

```
                res = -EIO;
                goto out;
            }
            if (entry == 0x00)
            {
                res = -EIO;
                goto out;
            }

            cluster_to_use = entry;
        }
        res = cluster_to_use;
out:
        return res;
}
```

- `static int fat16_get_cluster_for_offset(struct disk* disk, int starting_cluster, int offset)`: function definition.
- `int res = 0;`: return value initialization.
- `struct fat_private* private = disk->fs_private;`: `fat_private` access.
- `int size_of_cluster_bytes = private->header.primary_header.sectors_per_cluster * disk->sector_size;`: first we calculate the size of the clusters in bytes.
- `int cluster_to_use = starting_cluster;`: now we create a variable that holds the value of `starting_cluster`.
- `int clusters_ahead = offset / size_of_cluster_bytes;`: and we calculate the clusters ahead by dividing the offset passed onto us and the size of clusters in bytes.
- `for (int i = 0; i < clusters_ahead; i++)`: we'll run until `i` is less than `clusters_ahead`.
- `int entry = fat16_get_fat_entry(disk, cluster_to_use);`: here we get a FAT entry. this function will be explained below this one.
- `if (entry == 0xFF8 || entry == 0xFFF)`: we check if `entry` has `0xFF8` or `0xFFF`. meaning that we're checking that we aren't accessing a reserved cluster nor a end of file cluster. if we are…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to out.
- `if (entry == PEACHOS_FAT16_BAD_SECTOR)`: now we check if we're accessing a bad sector.
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to out.
- `if (entry == 0xFF0 || entry == 0xFF6)`: and now we check if we're accessing a reserved sector. (0xFF0-6, 0xFF8-0xFFE all indicate reserved sectors, but the latter might also indicate end of file). if we are accessing a reserved sector…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to out.
- `if (entry == 0x00)`: if we're accessing a free sector…
- `res = -EIO;`: we return `-EIO`.
- `goto out;`: and we go to out.
- `cluster_to_use = entry;`: and after all that, we get the cluster to use. and repeat :)
- `res = cluster_to_use;`: we set `res` to the cluster to use.
- `out:`: `out` label.
- `return res;`: and we return `res`. ### 15.1.13. `static int fat16_get_fat_entry(struct disk* disk, int cluster)` One more function to go. This one is used to get a FAT entry.

```
static int fat16_get_fat_entry(struct disk* disk, int cluster)
{
        int res = -1;

        struct fat_private* private = disk->fs_private;
        struct disk_stream* stream = private->fat_read_stream;
        if(!stream)
        {
                goto out;
        }
        uint32_t fat_table_position = fat16_get_first_fat_sector(private) * disk->sector_size;

        res = diskstreamer_seek(stream, fat_table_position * (cluster * PEACHOS_FAT16_FAT_ENTRY_SIZE));
        if(res < 0)
        {
                goto out;
        }
        uint16_t result = 0;
        res = diskstreamer_read(stream, &result, sizeof(result));
        if (res < 0)
        {
                goto out;
        }
```

```
        res = result;
out:
        return res;
}
```

- `static int fat16_get_fat_entry(struct disk* disk, int cluster)`: function definition.
- `int res = -1;`: initial return value.
- `struct fat_private* private = disk->fs_private;`: accesing the `fs_private`.
- `struct disk_stream* stream = private->fat_read_stream;`: and accessing the `fat_read_stream` streamer.
- `if(!stream)`: we check if the streamer was actually created beforehand. if not…
- `goto out;`: we go to out.
- `uint32_t fat_table_position = fat16_get_first_fat_sector(private) * disk->sector_size;`: we use the get first FAT sector to, well, get the first usable FAT sector and we multiply it by the sector size, as to obtain the sector in bytes instead of a sector number.
- `res = diskstreamer_seek(stream, fat_table_position * (cluster * PEACHOS_FAT16_FAT_ENTRY_SIZE));`: here we seek up to the `fat_table_position` multiplied by the cluster size, which is multiplied by the FAT entry size.
- `if(res < 0)`: if something went wrong…
- `goto out;`: goto out.
- `uint16_t result = 0;`: we set an initial result., in which we'll read the FAT entry to.
- `res = diskstreamer_read(stream, &result, sizeof(result));`: here we read the FAT entry into the `result` variable.
- `if (res < 0)`: if something went wrong…
- `goto out;`: goto out.
- `res = result;`: and we set `res` to `result`.
- `out:`: `out` label.
- `return res;`: and return `res`! ### 15.1.14. `static uint32_t fat16_get_first_fat_sector(struct fat_private* private)` All this function does is return the number of reserved sectors. This is useful, as the number will also indicate the first usable sector.

```
static uint32_t fat16_get_first_fat_sector(struct fat_private* private)
{
        return private->header.primary_header.reserved_sectors;
}
```

- `static uint32_t fat16_get_first_fat_sector(struct fat_private* private)`: function definition.
- `return private->header.primary_header.reserved_sectors;`: we return te `reserved_sectors` value. ### 15.1.15. `void fat16_free_directory(struct fat_directory* directory)` This function frees the memory allocated by the other functions. It frees directories and the items contained by the directories. This isn't deleting stuff from the disk, but rather deallocating the structures from the heap.

```
void fat16_free_directory(struct fat_directory* directory)
{
        if(!directory)
        {
                return;
        }
        if(directory->item)
        {
                kfree(directory->item);
        }
        kfree(directory);
}
```

- `void fat16_free_directory(struct fat_directory* directory)`: function definition.
- `if(!directory)`: if we didn't get an initialized directory…
- `return;`: we return nothing.
- `if(directory->item)`: if the directory has items…
- `kfree(directory->item);`: we free them!
- `kfree(directory);`: and we free the directory. ### 15.1.16. `void fat16_fat_item_free(struct fat_item* item)` Same here, but with items instead of directories.

```
void fat16_fat_item_free(struct fat_item* item)
{
        if(item->type == FAT_ITEM_TYPE_DIRECTORY)
```

```
	{
		fat16_free_directory(item->directory);
	}
	else if (item->type == FAT_ITEM_TYPE_FILE)
	{
		kfree(item->item);
	}
	kfree(item);
}
```

- `void fat16_fat_item_free(struct fat_item* item)`: function definition.
- `if(item->type == FAT_ITEM_TYPE_DIRECTORY)`: we check if the item type is of a directory. if it is…
- `fat16_free_directory(item->directory);`: we call the free directory function.
- `else if (item->type == FAT_ITEM_TYPE_FILE)`: if it is a file…
- `kfree(item->item);`: we free it!
- `kfree(item);`: and we free the initial structure. ### 15.1.17. `void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode)` Some modifications were made to the main `fopen` call. This is the last function of the day, so congratulations! You've made it. I'm quite proud of you.

```
void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode)
{
	if( mode != FILE_MODE_READ)
	{
		return ERROR(-ERDONLY);
	}
	struct fat_file_descriptor* descriptor = 0;
	descriptor = kzalloc(sizeof(struct fat_file_descriptor));
	if(!descriptor)
	{
		return ERROR(-ENOMEM);
	}

	descriptor->item = fat16_get_directory_entry(disk, path);
	if(!descriptor->item)
	{
		return ERROR(-EIO);
	}

	descriptor->pos = 0;
	return descriptor;
}
```

- `void* fat16_fopen(struct disk* disk, struct path_part* path, FILE_MODE mode)`: function definition.
- `if( mode != FILE_MODE_READ)`: we check if we're being asked to read. if not (since we haven't made any write operations yet)…
- `return ERROR(-ERDONLY);`: we return an error.
- `struct fat_file_descriptor* descriptor = 0;`: now we initialize a file descriptor.
- `descriptor = kzalloc(sizeof(struct fat_file_descriptor));`: and we zero allocate it.
- `if(!descriptor)`: we check if we were correctly allocated. if not…
- `return ERROR(-ENOMEM);`: we return an error.
- `descriptor->item = fat16_get_directory_entry(disk, path);`: now we start the execution.
- `if(!descriptor->item)`: we check if the item was returned. if not…
- `return ERROR(-EIO);`: there was a problem, so we return an error.
- `descriptor->pos = 0;`: we set the descriptor position to zero.
- `return descriptor;`: and we return it. And that's it! Although this is just the beginning of the FAT filesystem, we've made a lot of work together. Now, as a simple demonstration, we'll try our `fopen`.

1. Implementing the VFS Fread Function So… now we'll start writting the code to actually read the contents of a file. ## 1.1. `file.h` We've made some changes to the filesystem structure and added a function pointer.

```
#include <stdint.h>
typedef int (*FS_READ_FUNCTION)(struct disk* disk, void* private, uint32_t size, uint32_t nmemb, char* out);

struct filesystem
{
	// fs should return 0 from resolve if the disk is using its fs.
	FS_RESOLVE_FUNCTION resolve;
	FS_OPEN_FUNCTION open;
	FS_READ_FUNCTION read;
	char name[20];
};
```

- `#include <stdint.h>`: we've included this header file.
- `typedef int (*FS_READ_FUNCTION)(struct disk* disk, void* private, uint32_t size,`

uint32_t nmemb, char* out);: this is a function pointer declaration. we've seen this before, when creating the VFS structures. we're making a function type that can be any function that it is assigned to.

- FS_READ_FUNCTION read;: and we've made the function pointer part of the filesystem structure. ## 1.2. file.c Now here's the meat. ### 1.2.1. int fread(void* ptr, uint32_t size, uint32_t nmemb, int fd)

```
int fread(void* ptr, uint32_t size, uint32_t nmemb, int fd)
{
        int res = 0;
        if (size == 0 || nmemb == 0 ||fd < 1)
        {
                res = -EINVARG;
                goto out;
        }

        struct file_descriptor* descriptor = file_get_descriptor(fd);
        if(!descriptor)
        {
                res = -EINVARG;
                goto out;
        }

        res = descriptor->filesystem->read(descriptor->disk, descriptor->private, size, nmemb, (char*) ptr);
out:
        return res;
}
```

- `int fread(void* ptr, uint32_t size, uint32_t nmemb, int fd)`: function definition. we'll take a pointer, the file size, the nmemb, and the file descriptor or node index.
- `int res = 0;`: initial return value.
- `if (size == 0 || nmemb == 0 ||fd < 1)`: here we check for failure conditions. if any of these are met…
- `res = -EINVARG;`: we return invalid argument.
- `goto out;`: and go to out.
- `struct file_descriptor* descriptor = file_get_descriptor(fd);`: here we go looking for the file descriptor via the node index. we've defined this function before, so we won't be looking at it again.
- `if(!descriptor)`: we check if the descriptor wasn't created successfully. if not…
- `res = -EINVARG;`: we return invalid argument.
- `goto out;`: and go to out.
- `res = descriptor->filesystem->read(descriptor->disk, descriptor->private, size, nmemb, (char*) ptr);`: and now we call the filesystem's implementation of the `fread` funtion. we'll implement this later on.
- `out:`: `out` label.
- `return res;`: and we return the values.

2. Implementing the FAT16 fread function.

Now we'll implement the filesystem fread function. Let's go!

Note: *I was stuck on this for at least a month. I found several issues within the code, all of which I'll detail here and fix it for you guys. There were also some minor issues, such as typos that were inconsequential, as they were mistyped in all instances. Nevertheless, I fixed them too.*

2.1. fat16.c

First, we need to add the prototype of the function at the very top of our file.

```
int fat16_resolve(struct disk* disk);
void* fat16_open(struct disk* disk, struct path_part* path, FILE_MODE mode);
#### new code:
int fat16_read(struct disk* disk, void* descriptor, uint32_t size, uint32_t nmemb, char* out_ptr);
```

- `int fat16_read(struct disk* disk, void* descriptor, uint32_t size, uint32_t nmemb, char* out_ptr);`: we'll take a disk, a file descriptor, a size, an nmemb and the out pointer.

We too need to add the `fat16_read` and `fat16_fopen` to the filesystem struct. This will allow us to call `fat16_read` and `fat16_fopen` from the structure.

```
struct filesystem fat16_fs =
{
    .resolve = fat16_resolve,
    .open = fat16_fopen,
    .read = fat16_read
};
```

- `.open = fat16_open,`: this is a function pointer to the `fat16_fopen` function.
- `.read = fat16_read`: and this is another function pointer, but to the `fat16_read` function.

Now we implement `fat16_fread`.

2.1.1. `int fat16_read(struct disk* disk, void* descriptor, uint32_t size, uint32_t nmemb, char* out_ptr)`

The code is the following:

```
int fat16_read(struct disk* disk, void* descriptor, uint32_t size, uint32_t nmemb, char* out_ptr)
{
    int res = 0;
    struct fat_file_descriptor* fat_desc = descriptor;
    struct fat_directory_item* item = fat_desc->item->item;
    int offset = fat_desc->pos;
    for (uint32_t i = 0; i < nmemb; i++)
    {
        res = fat16_read_internal(disk, fat16_get_first_cluster(item), offset, size, out_ptr);
        if (ISERR(res))
        {
            goto out;
        }

        out_ptr += size;
        offset += size;
    }

    res = nmemb;
out:
    return res;
}
```

- `int fat16_read(struct disk* disk, void* descriptor, uint32_t size, uint32_t nmemb, char* out_ptr)`: function declaration.
- `int res = 0;`: we set the initial return value to zero.
- `struct fat_file_descriptor* fat_desc = descriptor;`: here we setup the FAT file descriptor by the passed descriptor. we can't directly use the `descriptor` descriptor, because it's a void pointer to a descriptor. assigning it a struct type `fat_file_descriptor*`, we'll be able to work with it.
- `struct fat_directory_item* item = fat_desc->item->item;`: and now we access the item held by the descriptor, which, if our disk was resolved correctly, should be a `fat_directory_item*` struct.
- `int offset = fat_desc->pos;`: now we set the offset to the `pos`.
- `for (uint32_t i = 0; i < nmemb; i++)`: and we start iterating through the `nmemb` (number of members) to read.
- `res = fat16_read_internal(disk, fat16_get_first_cluster(item), offset, size, out_ptr);`: and here the disk is read.
- `if (ISERR(res))`: if we have any errors in `res`…
- `goto out;`: we go to out and return the error.
- `out_ptr += size;`: now we set the `out_ptr` pointer to the size of the object..
- `offset += size;`: and we increment the `offset` by the size.
- `res = nmemb;`: and we set `res` to `nmemb`, which should be a number higher than 0.
- `out:`: out label.
- `return res;`: and we return `res`.

Now, I previously mentioned that I had made a few errors in the code. We'll look at them now. ## 2.2. Fixing bugs! ### 2.2.1. `fat16.c` #### Bug #1: Mistakenly allocating sizeof instead of actual root directory size in `fat16_get_root_directory`

```
-   struct fat_directory_item* dir = kzalloc(sizeof(root_directory_size));
+   struct fat_directory_item* dir = kzalloc(root_directory_size);
```

This one is simple. I wwas allocating the size of `root_directory_size` (or the size of an `int`) instead of the actual `root_directory_size`. #### Bug #2: Multiplication instead of adding reads sector 0 instead of intended sector in `fat16_read_internal_from_stream`

```
-   int starting_pos = (starting_sector * disk->sector_size) * offset_from_cluster;
+   int starting_pos = (starting_sector * disk->sector_size) + offset_from_cluster;
```

THIS bug held me over from developing the OS FOR WEEKS. Since I'm not extremely familiar with the FAT specs, it took way longer to debug than it should have. The logic explaination of this bug is as follow: - When getting the start position of the item to read, `offset_from_cluster` will most possibly be zero when working with small files (less than a cluster in size). This means that, when calculating the `starting_pos`, `(starting_sector * disk->sector_size) * offset_from_cluster` was always returning 0, and this meant that I was reading from the sector 0 (the boot sector) instead of the file. Not fun. #### Typo #1: `FAT_ITEM_TIPE` instead of `FAT_ITEM_TYPE` in typedef.

```
-   typedef unsigned int FAT_ITEM_TIPE;
+   typedef unsigned int FAT_ITEM_TYPE;
```

2.2.2. `string.c`

Bug #4: Operands mixed up in `memcpy` This bug is pretty simple.

```
-   void* memcpy(void* src, void *dst, size_t size)
+   void* memcpy(void* dest, void* src, int len)
```

I just messed up the order of operands in the function. :) ### 2.2.3. `file.c` #### Bug #4: Reading from disk private data instead of descriptor private data in `fopen`.

```
-   desc->private = disk->fs_private;
+   desc->private = descriptor_private_data;
```

This is another stupid bug I had.

After fixing everything, we're finally able to actually *READ* from the disk. ![[Pasted image 20250628234032.png]] # 3. Implementing the VFS `seek` function.

3.1. `file.c`

3.1.1. `fseek`

This is the body of the VFS `fseek` function.

```
int fseek(int fd, int offset, FILE_SEEK_MODE whence)
{
        int res = 0;
        struct file_descriptor* desc = file_get_descriptor(fd);
        if (!desc)
        {
                res = -EIO;
                goto out;
        }

        res = desc->filesystem->seek(desc->private, offset, whence);

out:
        return res;
}
```

- `int fseek(int fd, int offset, FILE_SEEK_MODE whence)`: function definition. we'll take a file descriptor, an offset and a SEEK mode.
- `int res = 0;`: here we set the initial return value.
- `struct file_descriptor* desc = file_get_descriptor(fd);`: here we get the file descriptor from the `int fd` passed onto us.
- `if (!desc)`: here we check if the descriptor is valid or not. if not…
- `res = -EIO;`: we return -EIO.
- `goto out;`: and go to out.
- `res = desc->filesystem->seek(desc->private, offset, whence);`: now we call the filesystem `fseek` implementation.
- `out:`: `out` label.
- `return res;`: and we return `res`. ## 3.2. `file.h` ### FS Fseek Function We need to add the function pointer. This way, the filesystems will be able to define it later on.

```
typedef int (*FS_SEEK_FUNCTION)(void* private, uint32_t offset, FILE_SEEK_MODE  seek_mode);
```

- `typedef int (*FS_SEEK_FUNCTION)(void* private, uint32_t offset, FILE_SEEK_MODE seek_mode);`: function pointer. same as above. ### Fseek prototype And finally we add the function prototype.

```
int fseek(int fd, int offset, FILE_SEEK_MODE whence);
```

Filesystem `fseek` definition.

```
struct filesystem
{
	// fs should return 0 from resolve if the disk is using its fs.
	FS_RESOLVE_FUNCTION resolve;
	FS_OPEN_FUNCTION open;
	FS_READ_FUNCTION read;
	FS_SEEK_FUNCTION seek;
	char name[20];
};
```

And we finally add the function pointer to the filesystem structure. # 4. Implementing `fstat` VFS function. ## 4.1. `file.h` First, we need to add the definitions for our `fstat` VFS function. ### 4.1.1 `FILE_STAT_FLAGS`

```
enum
{
	FILE_STAT_READ_ONLY = 0b00000001
};

typedef unsigned int FILE_STAT_FLAGS;
```

- `enum`:
- `FILE_STAT_READ_ONLY = 0b00000001`:
- `typedef unsigned int FILE_STAT_FLAGS;`: ### 4.1.2. `struct file_stat`

```
struct file_stat
{
	FILE_STAT_FLAGS flags;
	uint32_t filesize;
};
```

- `struct file_stat`: `file_stat` struct.
- `FILE_STAT_FLAGS flags;`: these are the flags. we only have read only for now.
- `uint32_t filesize;`: and the file size. ### 4.1.3. Function pointer.

```
typedef int (*FS_STAT_FUNCTION)(struct disk* disk, void* private, struct file_stat* stat);
```

4.1.4. Function prototype.

```
int fstat(int fd, struct file_stat* stat);
```

4.2. `file.c`

Now we'll define the actual function within our VFS. ### 4.2.1. `fstat`

```
int fstat(int fd, struct file_stat* stat)
{
	int res = 0;
	struct file_descriptor* desc = file_get_descriptor(fd);
	if (!desc)
	{
		res = -EIO;
		goto out;
	}
	res = desc->filesystem->stat(desc->disk, desc->private, stat);

out:
	return res;
}
```

- `int fstat(int fd, struct file_stat* stat)`: function definition. we'll take a file descriptor and a `file_stat` structure.
- `int res = 0;`: initial return value.
- `struct file_descriptor* desc = file_get_descriptor(fd);`: we get the file descriptor.
- `if (!desc)`: we check if it's a valid fd. it not…
- `res = -EIO;`: we set `-EIO`.
- `goto out;`: and we return.
- `res = desc->filesystem->stat(desc->disk, desc->private, stat);`: if the fd is valid, we call the filesystem implementation of `stat`.
- `out:`: out label.
- `return res;`: and we return.

Now, we'll go over the FAT16 implementation. # 5. FAT16 Fstat implementation. ## 5.1. `fat16.c` ### 5.1.1. Function prototype.

```
int fat16_fstat(struct disk* disk, void* private, struct file_stat* stat);
```

5.1.2. Function pointer in filesystem structure.

```
struct filesystem fat16_fs =
{
        .resolve = fat16_resolve,
        .open = fat16_fopen,
        .read = fat16_read,
        .seek = fat16_seek,
        .stat = fat16_fstat
};
```

5.1.3. `fat16_fstat`

```
int fat16_fstat(struct disk* disk, void* private, struct file_stat* stat)
{
        int res = 0;
        struct fat_file_descriptor* desc = (struct fat_file_descriptor*) private;
        struct fat_item* desc_item = desc->item;
        if (desc_item->type != FAT_ITEM_TYPE_FILE)
        {
                res = -EINVARG;
                goto out;
        }
        struct fat_directory_item* ritem = desc_item->item;
        stat->filesize = ritem->filesize;
        stat->flags = 0x00;

        if(ritem->attribute & FAT_FILE_READ_ONLY)
        {
                stat->flags |= FILE_STAT_READ_ONLY;
        }

out:
        return res;
}
```

- `int fat16_fstat(struct disk* disk, void* private, struct file_stat* stat)`: function definition. we'll take a disk, a file descriptor's `private` data and a `file_stat` structure.
- `int res = 0;`: initial return value.
- `struct fat_file_descriptor* desc = (struct fat_file_descriptor*) private;`: here we access the descriptor's private data.
- `struct fat_item* desc_item = desc->item;`: here we access the previous descriptor's item `fat_item` structure.
- `if (desc_item->type != FAT_ITEM_TYPE_FILE)`: now we check if the descriptor's type is of a file. if not…
- `res = -EINVARG;`: we return invalid argument.
- `goto out;`: and go to out.
- `struct fat_directory_item* ritem = desc_item->item;`: now we access the item itself using a `fat_directory_item*` structure.
- `stat->filesize = ritem->filesize;`: and we set the filesize using the previously created `ritem`.
- `stat->flags = 0x00;`: now we initialize the flags of the `stat`.
- `if(ritem->attribute & FAT_FILE_READ_ONLY)`: here we check if the attributes match. if so…
- `stat->flags |= FILE_STAT_READ_ONLY;`: we add the `FILE_STAT_READ_ONLY` bitmask to the `stat->flags` member of the struct.
- `out:`: out label
- `return res;`: and we return res. Simple. We're almost done with our VFS/FAT16 implementation. # 6. Implementing the VFS fclose function.

6.1. `file.h`

6.1.1. Function pointer

```
typedef int (*FS_CLOSE_FUNCTION)(void* private);
```

6.1.2. Filesystem structure update

```
struct filesystem
{
    ...
        FS_CLOSE_FUNCTION close;
        ...
};
```

6.1.3. Function prototype

```
int fclose(int fd);
```

6.2. `file.c`

6.2.1. `int fclose(int fd)`

```
int fclose(int fd)
{
        int res = 0;
        struct file_descriptor* descriptor = file_get_descriptor(fd);
        if(!descriptor)
        {
                res = -EIO;
                goto out;
        }
        res = descriptor->filesystem->close(descriptor->private);
        if(res == PEACHOS_ALLOK)
        {
                file_free_descriptor(descriptor);
        }

out:
        return res;
}
```

- `int fclose(int fd)`: function definition. we'll just take a file descriptor.
- `int res = 0;`: we set the initial return value.
- `struct file_descriptor* descriptor = file_get_descriptor(fd);`: here we get the file descriptor.
- `if(!descriptor)`: we check if it's a valid descriptor. if not…
- `res = -EIO;`: we return `-EIO`
- `goto out;`: and go to out.
- `res = descriptor->filesystem->close(descriptor->private);`: now we call the filesystem implementation of `close`.
- `if(res == PEACHOS_ALLOK)`: we check if `res` returned a positive value. if so…
- `file_free_descriptor(descriptor);`: we now free the descriptor from the VFS.
- `out:`: out label
- `return res;`: and we return. ### 6.2.2 `static void file_free_descriptor(struct file_descriptor* desc)` When freeing, we cannot just rely on the filesystem to free the descriptor, because the filesystem only frees the private data, but not the descriptor itself. We have to do that.

```
static void file_free_descriptor(struct file_descriptor* desc)
{
        file_descriptors[desc->index-1] = 0x00;
        kfree(desc);
}
```

- `static void file_free_descriptor(struct file_descriptor* desc)`: function definition. we'll take a `file_descriptor*` structure.
- `file_descriptors[desc->index-1] = 0x00;`: here we set the `file_descriptors` array at index `desc->index-1` to `0x00`, effectively freeing it.
- `kfree(desc);`: and we now free the memory associated with the descriptor itself.

We're almost done! Now we only have to implement the FAT16 fclose call and we'll we ready to go. # 7. Implementing the FAT16 `fclose` function. ## 7.1. `fat16.c` ### 7.1.1. Function prototype.

```
int fat16_close(void* private);
```

7.1.2. Modifying our filesystem.

```
struct filesystem fat16_fs =
{
    ...
    .close = fat16_close
    ...
};
```

7.1.3. `static void fat16_free_file_descriptor(struct fat_file_descriptor* desc)`

```
static void fat16_free_file_descriptor(struct fat_file_descriptor* desc)
{
        fat16_fat_item_free(desc->item);
        kfree(desc);
}
```

This function is quite simple. We leverage the previously created `fat_item_free` function to free the memory used by the item itself. After that, we just free the descriptor by calling `kfree`. ### 7.1.4. `int`

fat16_close(void* private)

```
int fat16_close(void* private)
{
	fat16_free_file_descriptor((struct fat_file_descriptor*) private);
	return 0;
}
```

This function too is simmple. We just call our previously created `fat16_free_file_descriptor`.

And our filesystem is ready. Not production ready, of course. We can't even write files. But it'll work for now.