theos

me.

oday

Contents

Description

To build the kernel, we've made a `Makefile` for the project. Use it with cross-compiling tools `i686-gcc` or others.

```
FILES = ./build/kernel.asm.o ./build/kernel.o ./build/idt/idt.asm.o ./build/idt/idt.o ./build/memory/memory.o \\
        ./build/io/io.asm.o ./build/memory/heap/heap.o ./build/memory/heap/kheap.o
INCLUDES= -I./src
FLAGS= -g -ffreestanding -falign-jumps -falign-functions -falign-labels -falign-loops -fstrength-reduce -fomit-frame-pointer -finline-functions -Wno-unused-function -fno-builtin -Werror -Wno-unused-label

all: ./bin/kernel.bin ./bin/boot.bin
        rm -rf ./bin/os.bin
        dd if=./bin/boot.bin >> ./bin/os.bin
        dd if=./bin/kernel.bin >> ./bin/os.bin
        dd if=/dev/zero bs=512 count=100 >> ./bin/os.bin

./bin/kernel.bin: $(FILES)
        i686-elf-ld -g -relocatable $(FILES) -o build/kernelfull.o
        i686-elf-gcc $(FLAGS) -T ./src/linker.ld -o ./bin/kernel.bin -ffreestanding -O0 -nostdlib ./build/kernelfull.o

./bin/boot.bin: ./src/boot/boot.asm
        nasm -f bin ./src/boot/boot.asm -o ./bin/boot.bin

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

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

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

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

./build/memory/heap/kheap.o: ./src/memory/heap/kheap.c
	i686-elf-gcc $(INCLUDES) -I./src/memory/heap/ $(FLAGS) -std=gnu99 -c ./src/memory/heap/kheap.c -o ./build/memory/heap/kheap.o
clean:
	rm -rf ./bin/boot.bin
	rm -rf ./bin/os.bin
	rm -rf ./bin/kernel.bin
	rm -rf $(FILES)
	rm -rf ./build/kernelfull.o
```

Building

In the main directory `theos/` run the build script. You have to change the environment variables to match the location of your cross compiller toolchain.

```
cd theos
chmod +x build.sh
./build.sh
```

Output files

In the `theos/build` directory, you will find all the object files for the compilled kernel. The one we mostly care about is the `kernelfull.o`, which is the entire kernel linked into a single file. Use it for loading all the symbols into GDB or your preferred debugger. ## Running To run the kernel, you can do one of the following: ### GDB Script You can run the kernel using the provided GDB script. The script assumes you have QEMU installed. If you don't, install it and add it to your $PATH.

```
gdb -x gdb_script
```

Using QEMU

You can also run the kernel by executing it directly with QEMU.

```
qemu-system-i386 -hda ./os/bin
```

Description

This page details some information on the current state of the OS bootloader. # Related code The related code can be found on the following directories: - src/boot - src/boot.asm # Architecture The bootloader it's not smart. At all. All it knows is that the following sector in the disk contains the kernel. Nevertheless, it has some important aspects to keep in mind. ## GDT Code Segments We define two important segments in the beginning of the `boot.asm` file: `CODE_SEG` and `DATA_SEG`. Both do important stuff later on when the GDT is loaded and the kernel is running.

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

Boot Parameter Block

At the beginning of the `boot.asm` file, we can find the Boot Parameter Block. Although it is a dummy BPB, it is a BPB nonetheless.

```
_start:
	jmp short start
	nop
times 33 db 0
```

Global Descriptor Table

We need to create a global descriptor table descriptor and send it to the CPU at boot time. This is done through the `gdt_*` labels found in `boot.asm`:

```
; 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
```

And in the label `load32`, there's the `lgdt` call.

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

Jump to 32 bit Protected Mode

Jumping to the 32 bit protected mode is done right after loading our GDT in the `.laod_protected` label.

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

And the first bits of code executed are the `load32` label.

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

Which loads our kernel via the ATA LBA driver and then jumps into it. ## LBA Read Driver. Before jumping into our kernel, we need a driver to read from the disk and actually load it into an accesible place in memory. This is done through the ATA LBA driver contained within our bootloader. The code is quite extensive and it's documented in the [[LBA ATA Related notes]] page.

```
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

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

        mov eax, ebx ; restoring the lba backup
        mov dx, 0x1F3
        out dx, al
        ; finished sending more bits of the LBA

        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

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

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

        ; read all sectors into memory
.next_sector:
        push ecx

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

Nevertheless, the `shr` instructions are calculating the LBA address (`LBA address = (ecx >> 24) + (ecx >> 16) + (ecx >> 8)`) and then reading after the `0x1F7` command is send to the I/O port. After that, it loops using the `.try_again` and `.next_sector` labels. ## Kernel execution. After we load the kernel into memory using the LBA driver, we can execute it using the `jmp` instruction. This is done in the `laod32` label.

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

Description

This page details some information on the current state of kernel and it's doings. # Related code The related code can be found on the following directories: - src/kernel.asm - src/kernel.c - src/kernel.h - src/memory/heap/kheap.c - src/memory/heap/kheap.h # Architecture The kernel is composed by several parts of code. Most important ones are `kernel.asm` and `kernel.c`. ## `kernel.asm` The `kernel.asm` binary gives execution to the C code in `kernel.c`. That's its purpose. It's initially loaded at `0x100000` in physical memory and then execution goes from there. ## Section setup The `kernel.asm` file also sets up the main section registers (`ds, ss, fs, es, gs, ax`).

```
[BITS 32]
global _start
extern kernel_main
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
```

A20 line

The `kernel.asm` file also enables de A20 line. This si to be able to access the entire memory address space.

```
        ; enabling the A20 line.
        in al, 0x92
        or al, 2
        out 0x92, al
        ; end of enabling the A20 line.
```

PIC Master Remaping

The `kernel.asm` also remaps the master PIC. Doing this allows us to add our own interrupts later on.

```
        ; remap the master PIC
        mov al, 00010001b ; init mode
        out 0x20, al
        mov al, 0x20 ; int 0x20 is where master ISR should start
        out 0x21, al

        mov al, 00000000b
        out 0x21, al
        ; end of master PIC remap
```

kernel.asm -> kernel_main

Finally, the `kernel.asm` calls `kernel_main`, which is in our `kernel.c` file. ## `kernel_main` The `kernel_main` is the main execution thread of the kernel. For now, it doesn't do much. ## `terminal_initialize` This function initializes (clears up) the screen after the kernel is loaded. ## `print(str)*` The `print` function allows the caller to print a string to the console. This is done via VGA memory manipulation. \This is a wrapper function.* ## `enable_interrupts` This function is from the [[3. Interrupts]] section and it enables interrupts via the Assembly `sti` instruction. ## `kheap_init` This functions is from the [[5. Memory]] section and it initializes the main heap of the kernel. ## `idt_init` This function is from the [[3. Interrupts]] section and it initializes the [[Interrupt Descriptor Table Related Notes|Interrupt Descriptor Table]]. ## `kernel.h` The header file contains some prototypes, mainly the `print` and `kernel_main` functions. It also contains the `VGA_HEIGHT` and `VGA_WIDTH` constants.

```
#define VGA_WIDTH 80
#define VGA_HEIGHT 20

void kernel_main();
void print(const char* str);
```

- `VGA_WIDTH`
  - Description:
    * This the width of the VGA buffer, also known as `terminal_col`.
- `VGA_HEIGHT`
  - Description:
    * This is the height of the VGA buffer, also known as `terminal_row`. # Description This page details the current state of interrupts in our kernel. # Related code The related code can be found on the following directories:
- src/idt
- src/idt/idt.asm
- src/idt/idt.c
- src/idt/idt.h # Architecture The IDT section is missing a lot of stuff. For now, it can only handle `int 21h` or a single keyboard press. Although the master PIC does receive constant acknoledgement. ## `idt_load` The `idt.asm` file contains the label `idt_load` which, as the name implies, loads the IDT using the `lidt` instruction. It also sets up its own stack and the pops it.

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

This is done using an argument passed onto it by some code in the `idt.c` file. ## `enable_interrupts` and `disable_interrupts` The `idt.asm` also creates some interfaces which can be used globally to enable and disable interrupts using `sti` and `cli` respectively.

```
enable_interrupts:
        sti
        ret

disable_interrupts:
        cli
        ret
```

int21h

The `int21h` label in `idt.asm` handles the interrupt 21h, which is related to keyboard presses. It does nothing more than that, though, and since we don't have a proper keyboard driver, it only registers a single keypress.

```
int21h:
        cli
```

```
        pushad
        call int21h_handler
        popad
        sti
        iret
```

no_interrupt

The no_interrupt label exists to be able to handle occasions where the system or the user don't send any interrupts at all. In real systems, this wouldn't happen as much, as operations are being done constantly. But in our case, we need to handle them as soon as possible. And we do.

```
no_interrupt:
        cli
        pushad
        call no_interrupt_handler
        popad
        sti
        iret
```

idt.h

Before going into `idt.c`, I need to explain the structures that are being used *in* the `idt.c` file. Without them, we won't understand the code. ### `idt_desc` First we have the `idt_desc` structure, which is our proper [[Interrupt Descriptor Table Related Notes|Interrupt Descriptor Table]].

\Many things are not well described here. It's not the purpose of this documentation to explain what each thing is and what it does. Check the related page for more info.*

```
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));
```

- `offset_1`
  - Type:
    * uint16_t
  - Description:
    * This variable is the initial section of our offset, which is divided in two parts.
- `selector`
  - Type:
    * uint16_t
  - Description:
    * This is the `selector` section, in which we set the `CODE_SEG` selector from our GDT.
- `zero`
  - Type:
    * uint8_t
  - Description:
    * Unused. Must be zero.
- `type_attr`
  - Type:
    * uint8_t
  - Description:
    * Here go the interrupt descriptor attributes and type.
- `offset_2`
  - Type:
    * uint16_t
  - Description: And finally we last bits of our offset.

idtr_desc

The IDT also needs the IDT Register, which is built in this struct.

```
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));
```

- `limit`
  - Type
    - uint16_t
  - Description:
    - This is the size of the IDT minus one.
- `base;`
  - Type:
    - uint32_t
  - Description:
    - Base address of the GDT. ## `idt.c` Now we can jump into the C code. ### `idt_descriptors[PEACHOS_TOTAL_INTERRUPTS]` This is the basis of our IDT. Our system will have 512 interrupts (as per `PEACHOS_TOTAL_INTERRUPTS` in [[6. Configuration|config.h]]). ### `idtr_descriptor` This is the IDT register. ### `int21h_handler` This is a simple function that prints out `kb pressed!` when the keyboard is pressed and then acknowledges the PIC by using the `outb` instruction implemented and documented in the [[4. IO Operations|I/O section]]. ### `no_interrupts` This function continually acknoledges the PIC. ### `idt_zero` This function just prints `divide by zero error` as per Intel's documentation and reserved interrupts. More on [[Interrupt Descriptor Table Related Notes]]. ### `idt_set` The `idt_set` function set up the interrupt descriptor table.

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

  For more information about this values, check the section on the IDT in the [[3. Protected mode development#11. Implementing the IDT in our code.|implementing IDT]] section. ### `idt_init` This function is called by the `kernel.c` [[2. Kernel#`idt_init`|routine]]. It's used to setup everything related to the IDT and its interrupts.

  ```
  void idt_init()
  {
    memset(idt_descriptors, 0, sizeof(idt_descriptors));
    idtr_descriptor.limit = sizeof(idt_descriptors) - 1;
    idtr_descriptor.base = (uint32_t) idt_descriptors;
    for (int i = 0; i < PEACHOS_TOTAL_INTERRUPTS; i++)
    {
  	idt_set(i, no_interrupt);
    }
    idt_set(0, idt_zero);
    idt_set(0x21, int21h);

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

\`memset` is part of the [[5. Memory|memory]] section of the documentation. *### `idt_load` This is where we make use of the Assembly label that we saw [[#`idt_load`|`idt.asm` section]] . Here we pass the IDT descriptor for it to be loaded by `lidt`. # Description This page details some information on the current state of I/O operations. What we have right now is not much. But hey, we're learning :) # Related code The related code can be found on the following directories: - src/io/io.asm - src/io/io.h # Architecture Most of the code we have as of right now is written in Assembly; this is because we cannot access I/O ports* directly* using Assembly in C. So, we create the labels in Assembly and then make those labels global for our C code to use. ## `outb` and `outw` `outb` and `outw` are wrapper labels for our C code. They send data to the given I/O port. `outb` uses bytes and `outw` uses words.

```
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
```

`insb` and `insw`

`insb` and `insw` are, again, wrapper labels for our C code. They receive data from the given I/O port and sent it back using the return (`eax`) register. As explained before, `insb` takes bytes and `insw` takes words.

```
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
```

Description

This page details the state of our memory management systems. As of right now, we only have a basic heap implementation, `malloc` and `free`. # Related code The related code can be found on the following directories: - src/memory/ - src/memory/memory.c - src/memory/memory.h - src/memory/heap/ - src/memory/heap/heap.c - src/memory/heap/heap.h - src/memory/heap/kheap.c - src/memory/heap/kheap.h # Architecture As of right now, the memory part of our code only has the heap, `malloc` and `free`. There are a lot of helper functions in `heap.c` that I might not document. ## `memset(void* ptr, int c, size_t size)` We have a basic implementation of `memset` in the code. It works as expected.

```
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;
}
```

`heap.h`

Before jumping into the `heap.c` code, we need to read and understand the basic structures and constants that our heap uses. ### Constants We have four basic constants in our header.

```
#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
```

- HEAP_BLOCK_TABLE_ENTRY_TAKEN
  - Value:
    - `0x01`
  - Description:
    - This value is used to mark an entry as taken or check if the heap entry is taken.
- HEAP_BLOCK_TABLE_ENTRY_FREE
  - Value:
    - `0x00`
  - Description:
    - This value is used to free an entry or to check if the heap entry is free.
- HEAP_BLOCK_HAS_NEXT
  - Value:

    * `0b10000000`
  - Description:
    * This value is used to check if a heap entry has a next entry or to mark it as a multiblock allocation.
- HEAP_BLOCK_IS_FIRST
  - Value:
    * `0b01000000`
  - Description:
    * This value is used to check if we're starting an allocation or to start an allocation. ### Type definitions We have some basic types for our table entries.

```
typedef unsigned char HEAP_BLOCK_TABLE_ENTRY;
```

- `HEAP_BLOCK_TABLE_ENTRY`
  - Size:
    * 1 byte.
  - Description:
    * Each bit in this byte has an assigned attribute. See [[#Constants]] for the attributes or [[Heap and memory alloc related notes|the heap notes]] for more information. ### `heap_table` The `heap_table` structure is the basis for our heap entry table. It serves as a map of memory entries.

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

- `entries`
  - Type:
    * `HEAP_BLOCK_TABLE_ENTRY*`
  - Description:
    * This item in our structure will contain all the entries that we'll use in `kheap.c`.
- `total`
  - Type:
    * `size_t`
  - Description:
    * This item contains the total size of our heap. ### `heap` This is the data pool of our heap implementation.

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

- `table`
  - Type:
    * `struct heap_table*`
  - Description:
    * This item contains a pointer to the heap table.
- `saddr`
  - Type:
    * `void*`
  - Description:
    * This item contains the initial address of the heap, which is `0x01000000` (check [[6. Configuration|config.h]] for more) ## `heap.c` `heap.c` is really meaty, so we'll go over only the functions that we actually care about. A more detailed documentation can be found over in [[4. The heap and memory allocation|the heap section.]] ## `heap_create(struct heap*, void* ptr, void* end, struct heap_table* table)` The `heap_create` function is supposed to be called only by kernelspace. It's used to create the initial heap of the kernel. "' 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; }

`heap_malloc(struct heap* heap, size_t size)`
The `heap_malloc` function is supposed to be called only by kernelspace. It's used to allocate stuff.

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); }

`heap_free(struct heap* heap, void* ptr)`
The `heap_free` function is used to free allocated memory for other processes. It's only supposed to be called by kernelspace.

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

**Note: these two functions use a lot of subfunctions to work. All of them are documented over at [[4. The heap and memory allocation|the heap page]] in the docs.*
`kheap.c`
We've defined some functions and wrappers in the `kheap.c` file.
`kheap_init`
We make it our responsibility to initialize the heap. This function does exactly that.

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");
    }
```

}

`kmalloc(size_t size)`
We create a wrapper function over the functions defined in the `heap.c` file. This works as a `malloc` for our kernel.

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

`kfree(void* ptr)`
We create a wrapper function over the functions defined in the `heap.c` file. This works as a `free` for our kernel.

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

Description
This page details the configuration aspects of the kernel.
Related code
The related code can be found on the following directories:
- src/config.h
Configurations
We can define several constants and things in this file. Some things are better left untouched, like the selectors.

// GDT code segment #define KERNEL_CODE_SELECTOR 0x08

// GDT data segment #define KERNEL_DATA_SELECTOR 0x10

// OS total amount of interrupts #define PEACHOS_TOTAL_INTERRUPTS 512

// 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

```
- `KERNEL_CODE_SELECTOR`
    - Description:
        - This is the memory address of the `CODE_SEG` GDT section.
- `KERNEL_DATA_SELECTOR `
    - Description:
        - This is the memory address of the `DATA_SEG` GDT section.
- `PEACHOS_TOTAL_INTERRUPTS`
    - Description:
        - This constant defines the total amount of interrupts to be initialized in the kernel.
- `PEACHOS_HEAP_SIZE_BYTES`
    - Description:
        - This constant defines the heap size in bytes. The operation `PEACHOS_HEAP_SIZE_BYTES % PEACHOS_HEAP_BLOCK_SIZE` must return 0, otherwise the initialization of the heap will fail with  `-EINVARG`
- `PEACHOS_HEAP_BLOCK_SIZE`
```

    - Description:
        - This constant defines the byte size for each block. The operation `PEACHOS_HEAP_SIZE_BYTES % PEACHOS_HEAP_BLOCK_SIZE` must return 0, otherwise the initialization of the heap will fail with `-EI
- `PEACHOS_HEAP_ADDRESS `
    - Description:
        - This constant defines the initial starting address of our heap.
- `PEACHOS_HEAP_TABLE_ADDRESS`
    - Description:
        - This constant defines the location in memory where the heap table will be located.

Description
This page details some information on the current state of disk operations. This isn't a filesystem specific page, but rather an agnostic page on drive reading, writting and general access.
Related code
The related code can be found on the following directories:
- src/disk
- src/disk/disk.c
- src/disk/disk.h

Architecture
We've implemented a small driver that will allow us to read `n` sectors using the LBA ATA ports.
`int disk_read_sector(int lba, int total, void* buffer)`
This code will take an `lba` start sector, a `total` amount of sectors to be read starting from `lba` and a `buffer[SIZE]` in which the read sectors will be read to. This is done using the previously impl

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;
```

}

`void disk_search_and_init()`
This function will initialize the `disk` struct and assign some types. It's in early alpha, since it doesn't really search for anything right now.

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_get(int index)`
This function will get you the disk from an index. Again, it in early alpha, since right now all it does is return the already existing `disk` structure as a pointer.

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

`int disk_read_block(struct disk* idisk, unsigned int lba, int total, void* buf)`
This function will be the main way in which a programmer will read sectors from the disk. It requires a `disk` structure which is obtained by using `disk-get`, an `lba` starting sector, the `total` sector

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); } “‘