# Welcome!

## Ever thought of making a Kernel, for flex reasons, educational, or even commercial reasons, but don't have any clue what you're doing?

Well, today's your ***lucky day!*** This book contains everything you'll need to know to make a basic **DOS-Like Kernel**. Obviously, it won't be as good or as big as MS-DOS , but you could make it bigger if you wanted to.

## What You Will Learn:

By the end of this tutorial, you'll have a functional kernel that can interact with hardware and manage basic tasks. It's designed for beginners, so you don't need any previous experience with operating systems or low-level programming. Here's a sneak peek at what you'll cover:

- Basic Theory: Understand the fundamentals of what a kernel is and how low-level hardware interaction works.
- Basic Hello World: Create your first "hello world" program running on your kernel.
- Basic Input Handling: Learn how to receive and process user input from the keyboard.
- Command Parsing: Dive into how commands are parsed and executed by your kernel.
- Basic Errors: Handle basic error messages and outputs to improve your kernel's usability.
- System Calls: Implement basic system calls for interacting with user programs.
- Advanced Errors: Learn how to handle more complex errors with better feedback.
- Inline Assembly: Add assembly code for critical, low-level operations and optimizations.
- Type Conversion: Understand how to convert data types (like integers, strings) in a low-level language.

- Pointers: Master the concept of pointers to manage memory directly.

## Requirements:

**Basic programming knowledge**: You should have a basic understanding of programming concepts, especially in C and assembly. Though don't worry too much if you don't, I'll walk you through it anyway.

**Development environment**: You'll need a development setup that includes tools like GCC and a virtual machine like QEMU for testing.

**Patience**: Building an operating system is no easy task, but it's an incredibly rewarding process. If you run into problems, don't be discouraged—this is a learning journey!

## Why a DOS-Like Kernel?

You might wonder why we're focusing on a **DOS-like kernel**. The reason is simple: it's a great starting point! By mimicking older systems like **MS-DOS**, we can focus on essential tasks like **interacting with hardware, handling basic input/output, and writing to the screen**, without getting bogged down by more advanced, complex features.

## Who Is This For?

### This tutorial is perfect for anyone who:

- Wants to learn more about how operating systems work from the ground up.
- Has an interest in low-level programming or kernel development.
- Wants to make something cool just for the fun of it or for educational purposes.
- Has tried other kernel tutorials but struggled with complexity.

# Chapter 1 - Basic Theory

# Welcome! This chapter will teach you all about the basic theory needed for this tutorial

## Video Graphics Array

- `Video Graphics Array`, or more commonly reffered to as `VGA`, is a graphical mode used & manipulated by programmers to display things onto the screen, usually only used at a low level. Even though it's old, it's still used in things like Kernel development and bootloaders.
- It is an `array of characters`, usually 16-bit. It uses indexes (place/position in an array) to know where to place the character. In memory, a single VGA index looks something like:

`(Color (high-byte) | Character (low-byte)).`

If unfamiliar with arrays, arrays are an `organised way to store data`. Example:

```
int example[25]; // This sets an array of integers (whole numbers) with a
fixed size of 25
example[0] = 15; // Sets the first value of example as 15
```

- In the tutorial, color will follow the format 0xHH, where HH are hex values (base-16 number system), e.g. *0x1B* for cyan on dark blue, *0x0F* for white on black, *0x07* for Grey on black (common for early boot logs)

## Kernel

- If for some reason, you're diving into making a kernel without even **knowing what it is**, here's a crash course.

- **Main Jobs**
- ◦ Manipulate hardware

- Provide `system calls` (functions used by userland to tell the computer what to do)
- Execute software programs
- `Core of the OS`
- "Oversees" interactions `between software and hardware`.

Kernels are often seen as hard to develop, and if you've ever been on guide websites before, you'll see they tell you that you need ***decades of experience.*** That is partially true - If you are aiming for something to rival the NT Kernel or the Linux Kernel, yes, you should have a decent amount of experience before attempting it. But a little DOS or CP/M like kernel? A `week's experience` should be enough (it was for me, anyway).

## Assembly

- Assembly is a programming language that's goal is to be a human readable form of machine code (0, 1). It was developed in ~1940s by some of the earliest computer scientists. Though really old and verbose, it was used in projects like BIOS, 86-DOS, and much more. More info

## C

- C is an "intermediate" level programming language. It was originally constructed to re-write the Unix kernel in a higher-level language. It is a successor to B (who would have guessed), and was made at Bell Labs by Dennis Ritchie in the 1970s, and though being over FIFTY years old, we still use it today. Some examples of use are: Linux kernel, NT Kernel, C itself (Bootstrap) and more.

## Type conversion

- Usually, in C, you can quite easily convert between types, as you will see in the source code of "Hello, world!" where we convert VGAMEM to a 16 bit unsigned integer (uint16_t)

## Boot Process

- Usually, the boot process of a computer is as follows: Boot architecture (BIOS/UEFI)

--> 0x7c00 (Memory location of the bootloader) --> Kernel

- **In more detail:**

- ◦ The BIOS/UEFI reads provided disks for a bootable binary (determined by the boot signature 55 AA)
- ◦ If found, it executes the code in the binary. Usually, this binary tells the computer to load a kernel*
- ◦ The kernel then prepares hardware, loads software, Modules, and possibly a shell.

*This is because the bootloader has to fit in 510 bytes, since the boot signature takes up 2 and the BIOS can only read that much.

# Welcome! This part will cover how to write a Hello world kernel.

**This bit is actually straightforward**.

```
typedef unsigned int uint32_t; // Defining a non negative (unsigned) integer
as uint32 (32 bit)
typedef unsigned short uint16_t;
typedef unsigned char uint8_t;
typedef unsigned long size_t;

#define VGAW 80 // This defines the width of the VGA. Text mode VGA Uses
80x25
#define VGAH 25
#define VGAMEM 0xB8000 // This is where VGA begins in computer memory.
```

Before we continue, I would highly recommend sinking that in before proceeding. Do some research specifically on the var types (uintX_t) because they can be hard to grasp.

```
static uint8_t defaultVGAcol = 0x1B; // 8 bit value defining the default VGA
color (this isn't needed now, but will be)

static uint16_t* vmem = (uint16_t*)VGAMEM;
/* The line above establishes a 16 bit pointer to VGAMEM (which contains
0xB8000), but we're making VGAMEM a 16 bit value aswell */
```

So now, We have different bit types, VGA nailed down, and a default color (feel free to change!). All we need now is a way to manipulate it. First we'll define the text cursor's virtual x & y positions.

```
static size_t cursorx = 0;
static size_t cursory = 0;
```

Now, we'll make a global variable for the VGA color (useful for changing during runtime, like for errors or theme hopping)

```
static uint8_t vgcol = 0x1B;
```

The reason VGA colours are 8 bit is because VGA requires 2 values for an index (because it's an array, remember?): Color & Character. Since VGA here is 16-bit, and we need 2 value, it's only logical to split it into 2 8 bit values (color and char), or else errors will happen.

Now, we will write a function/macro to write a character to the screen at the correct index (or else everything would be put at the top left). We will need to calculate the correct index, because computers are mean :-(

```
void putchar(char c) {
    if (c == '\n') { // Checking if the char is equal to a new line
        cursorx = 0; // Resetting the cursor's x position to the far left
        if (++cursory >= VGAH) { // Checking if when we add 1 to cursor's y
position, it exceeds or is equal to VGAH
            cursory = VGAH - 1; // Stay at the bottom line to prevent
overflow
        }

        return;
    }

    size_t index = cursory * VGAW + cursorx;
    vmem[index] = (uint16_t)c | (vgcol << 8);

    if (++cursorx >= VGAW) { // Checking if cursor x exceeds VGA Width bounds
(80)
        cursorx = 0;
        if (++cursory >= VGAH) { // Checking if when we add 1 to cursor's y
position, it exceeds or is equal to VGAH
            cursory = VGAH - 1;
        }

        return;
    }

}
```

**Woah, woah!** What's all of ***THAT?!*** Good question. The bit to most likely confuse you is `calculating the index`, or in the code:

```
size_t index = cursory * VGAW + cursorx;
vmem[index] = (uint16_t)c | (vgcol << 8);
```

First off, calculating the index in VGA is done with the formula: *row * Screen Width + Column*. If we replace the variables, we get cursory * VGAW + cursorx

And the second bit is telling the computer this (I have written it in pseudocode just for you!): **The position we just calculated inside the array vmem = 16 bit version of C + the color (but push color into the high byte)**

Now you may be thinking: *"B-but I don't want to write out large strings 1 char at a time"* Then you my friend, are part of the majority. But how will we handle writing strings? Simple. **Check if the string hasn't ended, print the character currently in memory, then move on until the string no longer exists.**

```
void putstr(const char* str) {
    while (*str) {
        putchar(*str++);
    }
}
```

Well that was easy wasn't it?

Now we need a main kernel function. Something to bring all the code together. I'll call mine krnlMain. *NOTE: You will need an entry point that calls this, probably in ASM*.

```
void krnlMain() {
    putstr("Hello, world!");
    while (1) {
        // dead hang for now
    }
}
```

You may be asking: "wElL wHy Do We nEeD a WhIlE lOoP?", and because you're new, I'll let it slide. We need a while loop because if we don't have one, the kernel will exit as soon as Hello world is printed (and it's much faster than we can see it), so we won't see it!

# Basic Input Handling!

So, now that we have a way to *output*, we now need a way to *input*. Before going through this, I would recommend getting a basic grasp of ports. If you already know, that's good! You can continue.

So, you will know that the keyboard port is 0x64. We need to interact with this port to get KB input. We will need some inline asm for this. In your kernel file, add this:

```
static inline uint8_t inb(uint16_t port) { // static func containing inline
assembly that returns an 8 bit value (requires 16 bit port)
    uint8_t ret;
    __asm__ __volatile__ ("inb %1, %0" : "=a"(ret) : "Nd"(ret));
    return ret;
}
```

We will now need a keyboard mapping. Typically, it's qwerty. If you don't wanna write your own from scratch, copy this:

```
static const char asciimap[128] = {
    0,  27, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', '\b',
    '\t', 'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', '[', ']', '\n',
    0, 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ';', '\'', '`',
    0, '\\', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', ',', '.', '/', 0,
    '*', 0, ' ', 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

Unfortunately, neither of these functions do something on their own. We will need a couple functions.

First, a useful clear screen function.

```
void clrscr() {
    for (size_t y = 0; y < VGAH; y++) {
        for (size_t x = 0; x < VGAW; x++;) {
            vmem[y * VGAW + x] = (uint16_t)' ' | (vgcol << 8);
        }
    }

    cursorx = 0;
    cursory = 0;

    putstr("Welcome to my kernel!\n"); // This will print every time the
screen is cleared!
}
```

The above writes a blank (' ') for every time y is less than VGAH and x is less than VGAW. It then resets cursor position.

Now, to convert raw input from the kb port:

```
uint8_t getscan() {
    while(!(inb(0x64) & 1)) { } // While the keyboard port isn't being
actively used, poll it until it's being used
    return inb(0x60)
}
```

The above function doesn't convert raw bytes, but gets the raw bytes and returns it (useful)

```
char getch() {
    uint8_t scancd = getscan() // the return value of getscan is held in
scancd
    if (scancd < 128) {
        return asciimap[scancd];
    }

    return 0;
}
```

Now, the above function is what gets the 8 bit scancode of the key pressed from getscan, then looks for the matching value in asciimap.

But none of those are very practical. We need to write a custom scanf, so we can actually handle input. We will need a buffer to store input.

```
void readstr(char* buffer, size_t bufsize) {
    size_t pos = 0;
    while (1) {
        char c = getch(); // c is the return value of getch
        if (c) {
            if (c == '\b' && pos > 0) { // if the key is backspace:
                pos--; // take 1 away from virtual position
                cursorx = (cursorx == 0) ? VGAWID - 1 : cursorx - 1; // new
cursor x
                puts(" \b");
            }

            else if (c == '\n') {
                putchr('\n');
                buffer[pos] = '\0';
                break; // stop reading if enter is pressed!
            }

            else if (pos < bufsize - 1) {
                buffer[pos++] = c;
                putchr(c);
            }


        }
    }
}
```

The above code is simple, just long. So if you're finding it complicated, break it down logically.

Now that we have all of that, we can add it to our main kernel function.

```
void krnlMain() {
    clrscr();

    char inpbuffer[128];
    while (1) {
        putstr("Type 'help' for a list of cmds!\n");
        readstr(inpbuffer, sizeof(inpbuffer));
        puts("\n");
    }
}
```

## And boom! Now you can handle input!

# Command parsing

Welcome! In this chapter, we will cover command parsing. This is easily the most straightforward chapter. All we need to do is make a custom strcmp, strncmp (a function that compares strings until n is 0) and a command handler

First, the custom string comparisons.

```
int cmp(const char *str1, const char *str2) {
    while (*str1 && (*str1 == *str2)) {
        str1++;
        str2++;
    }

    return *(const unsigned char*)str1 - *(const unsigned char*)str2;
}

int cmpn(const char *str1, const char *str2, size_t n) {
    while (n && *str1 && (*str1 == *str2)) {
        n--;
        str1++;
        str2++;
    }

    if (n == 0) {
        return 0;
    }

    return *(const unsigned char*)str1 - *(const unsigned char*)str2;
}
```

Now, just a command handler. This part is super duper simple.

```
void reboot() {
    __asm__ __volatile__ (
        "int $0x19"
        :
        :
        : "memory"
    );
}

void cmdhandle(const char* cmd) {
    if (cmpn(cmd, "ECHO ", 5) == 0) {
        putstr(cmd + 5);
        putstr("\n");
    }

    else if (cmp(cmd, "REBOOT") == 0) {
        reboot(); // Here you will implement a reboot function using int
0x19. If you don't know how, check out the example
    }

}
```

# Hello! This covers basic error handling. It is a small part of a medium large section.

Error handling is a core part of any system. If the computer doesn't know what to do, it may start **freaking out**. As you can imagine, that is NOT a good thing. In this section, we will add error handling to the command parser.

```
void cmdhandle(const char* cmd) {
    if (cmpn(cmd, "ECHO ", 5) == 0) {
        putstr(cmd + 5);
        putstr("\n");
    }

    else if (cmp(cmd, "REBOOT") == 0) {
        reboot(); // Here you will implement a reboot function using int
0x19. If you don't know how, check out the example
    }
    /* Now to implement error handling, we need an else statement. */
    else {
        putstr("Invalid command!");
        if (cmpn(cmd, "ECHO", 2) == 0) { // If the first 2 letters of ECHO
and whatever the user typed are the same:
            putstr(" Did you mean echo?")
        }
        else if (cmpn(cmd, "REBOOT", 2) == 0) {
            putstr(" Did you mean reboot?");
        }
        putstr("\n");
    }

}
```

# Memory management

With memory management, you will need a method to allocate and free memory (that is managing memory after all).

### WARNING: This will be complex and if you aren't good in C, I would refrain from adding this.

*Also, if you are making this for a hobby, refrain from adding this unless you're planning to get it somewhere!*

If you want to see memory management and how to make it, check the github repo.