RV Educational Institutions ®
RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE, New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

REPORT

Submitted by

SHREEJAY PANDEY	**1RV22CS186**
SHREYANSH SING	**1RV22CS190**
UJJWAL S G	**1RV22CS418**
VED M REVANKAR	**1RV22CS419**

Computer Science and Engineering
2023-2024

Table of Contents:

1. **Introduction**
2. **System Architecture**
3. **Methodology**
4. **Systems calls used**
5. **Output/results**
6. **Conclusion**

INTRODUCTION

Kenel using x86 architecture. At its core lies a series of intricate processes that power the bootstrapping of an operating system kernel. This report delves into the fundamental steps involved in constructing a kernel tailored to the x86 architecture, offering insights into each crucial stage of the process.

The journey begins with the pivotal act of booting the kernel from disk. This initial step sets the stage for the entire system to come to life. Through a series of intricate operations, the bootloader, residing in the Master Boot Record (MBR), takes charge, orchestrating the loading of the kernel from disk into memory. This seamless transition from disk to memory lays the groundwork for the subsequent stages of kernel initialization.

Central to the process is the utilization of the FAT16 file system, a widely adopted format renowned for its simplicity and compatibility. This file system structure facilitates the storage and retrieval of crucial kernel components, ensuring efficient access during the bootstrapping process. By adhering to the FAT16 standard, developers harness a reliable mechanism for organizing and managing essential system files, paving the way for a smooth boot sequence.

A pivotal player in the bootstrapping saga, the bootloader serves as the bridge between the hardware and the kernel. Nestled within the confines of the MBR, the bootloader executes a delicate dance, initializing hardware components, configuring memory, and ultimately transferring control to the loaded kernel. This intermediary stage is essential for establishing the groundwork necessary for the kernel's operation, ensuring seamless communication between the hardware and software layers.

As the kernel takes its place in memory, a critical transformation occurs—the transition from real mode to protected mode. In this metamorphosis, the processor shifts gears, unlocking a realm of enhanced capabilities and security features. By transitioning into protected mode, the kernel gains access to vital resources, enabling it to execute complex operations and interact with hardware peripherals with heightened efficiency and control.

SYSTEM ARCHITECTURE

The x86 architecture, also referred to as IA-32 (Intel Architecture 32-bit), stands as a cornerstone in computing, boasting a rich lineage of instruction set architectures (ISAs) primarily spearheaded by Intel and later embraced by AMD and other prominent CPU manufacturers.

Instruction Set: At the heart of x86 processors lies an extensive repertoire of instructions encapsulated within the x86 instruction set. These instructions encompass a broad spectrum of functionalities, ranging from fundamental arithmetic and logical operations to intricate system-level tasks vital for comprehensive computing tasks.

Registers: Integral to the processing power of x86 CPUs are their ensemble of general-purpose registers, including EAX, EBX, ECX, and EDX. These registers serve as dynamic reservoirs for data manipulation, memory addressing, and control flow, enabling efficient execution of diverse computational tasks.

Memory Addressing: The x86 architecture boasts versatile memory addressing capabilities, accommodating various addressing modes to cater to the diverse needs of software developers. These modes encompass direct addressing, indirect addressing, and base/index addressing, providing flexibility in accessing and manipulating memory resources.

Operating Modes: x86 processors exhibit a dynamic operational landscape, with distinct operating modes tailored to different computing environments. Among these modes, real mode, protected mode, and long mode (the latter prevalent in modern x86-64 processors) stand out, each offering unique features and capabilities to cater to specific computational requirements.

From its inception, the x86 architecture has evolved into a stalwart pillar of computing, underpinning a myriad of software applications, operating systems, and technological advancements. Its enduring legacy continues to shape the digital landscape, driving innovation and powering the next generation of computing experiences.

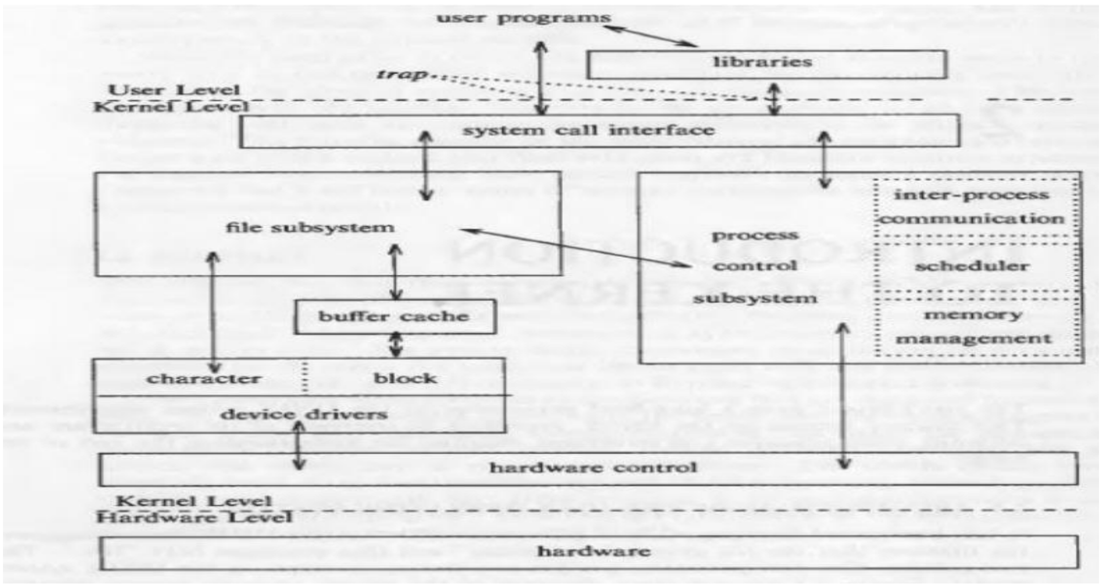
METHODOLOGY

Research Gathering: Initial research was conducted to understand the x86 architecture comprehensively. This involved studying materials provided by Intel, AMD, and other reputable sources to grasp the fundamental concepts and features.

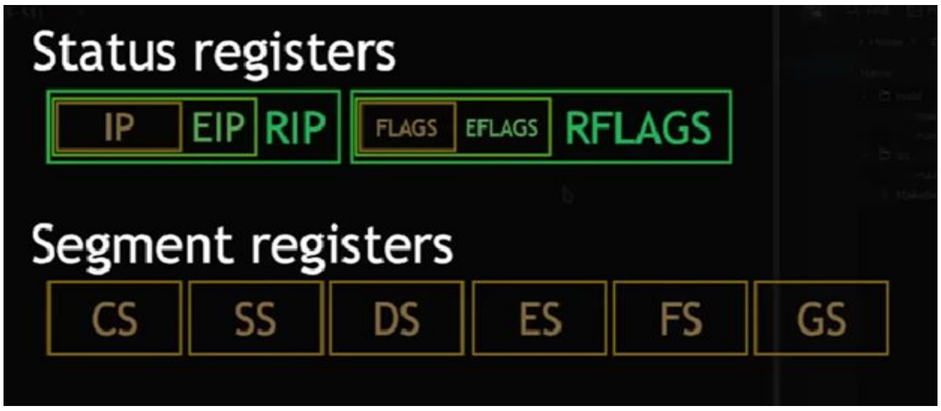
Simplification: Complex technical jargon and concepts were simplified to make the content easily understandable for a wide audience. This involved breaking down intricate details into digestible pieces without sacrificing accuracy.

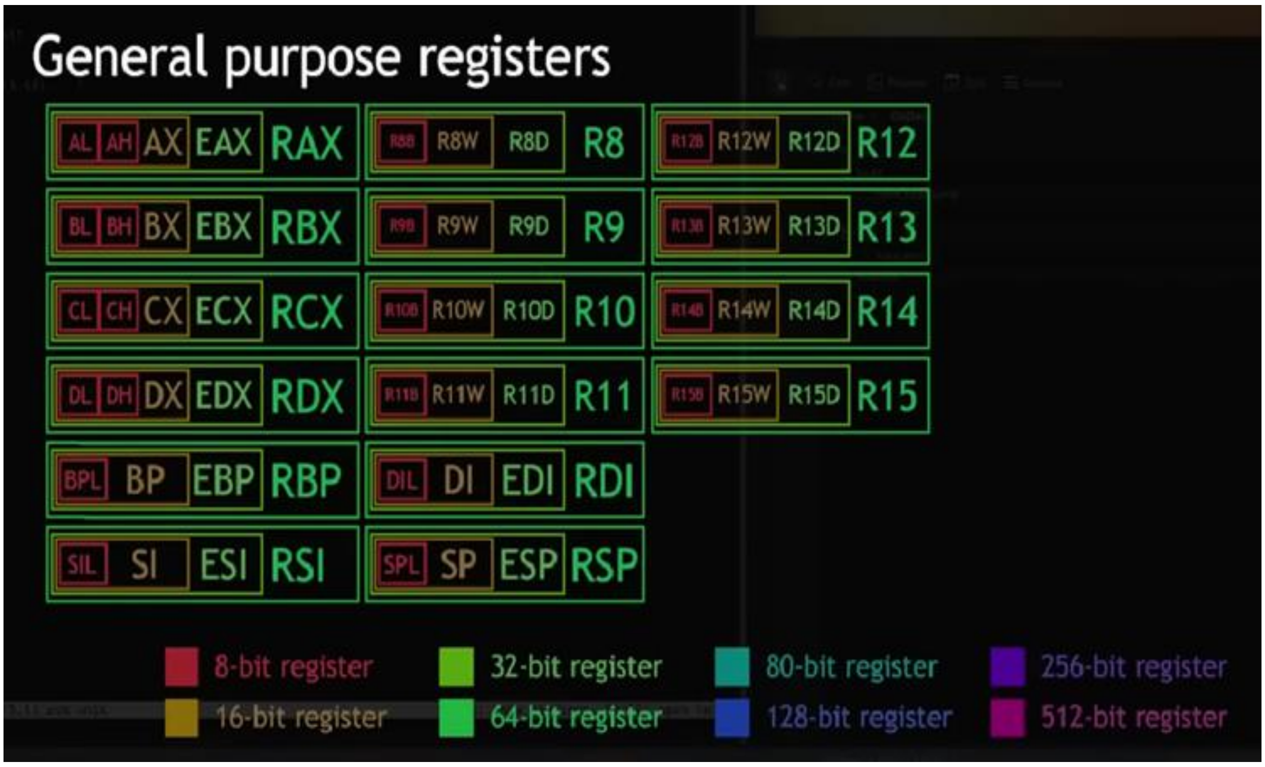
Structured Organization: The content was organized into distinct sections covering essential aspects of the x86 architecture, including its instruction set, registers, memory addressing, and operating modes. This structured approach helped in presenting the information in a logical sequence.

Block diagram

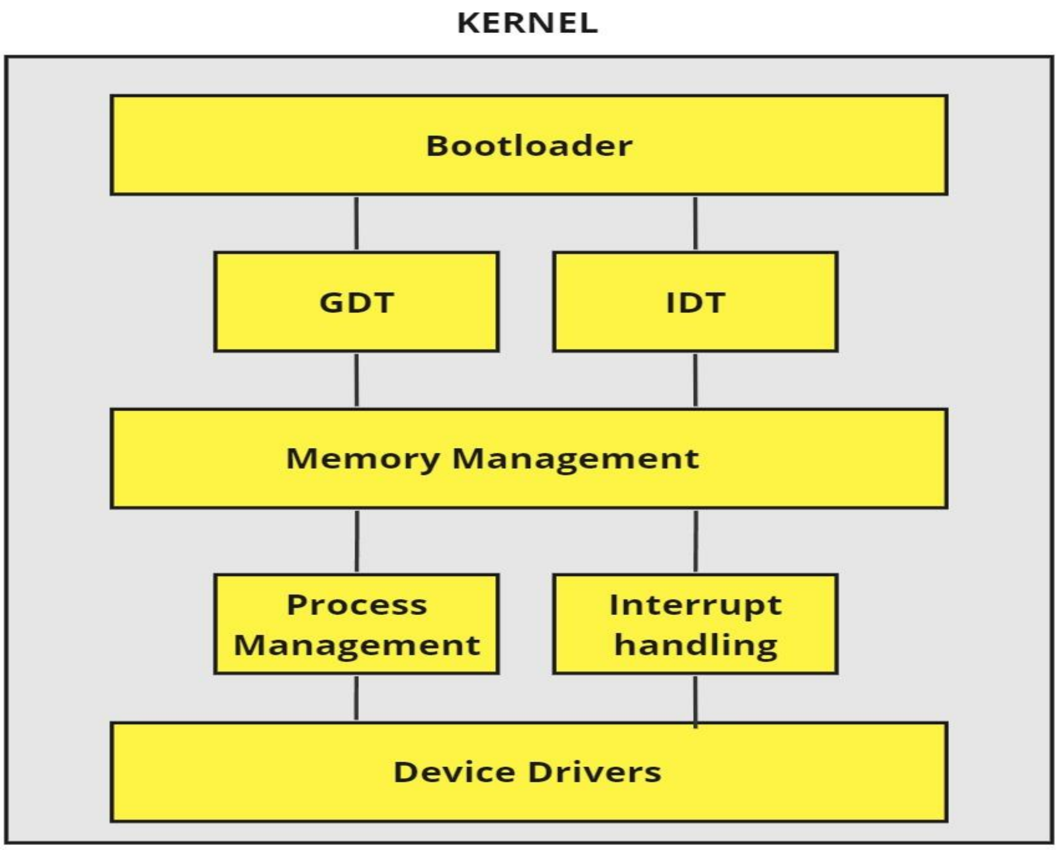


X86 REGISTERS





Software Design



SYSTEM CALL

1.File System Related System Calls:

Open File (sys_open):

```
int sys_open(const char *filename, int flags, mode_t mode);
```

Assembly Syntax:

```
mov eax, 5          ; System call number for sys_open
mov ebx, filename   ; Pointer to filename
mov ecx, flags      ; File access mode
mov edx, mode       ; File permission mode
int 0x80            ; Interrupt to invoke the system call
```

Read File (sys_read):

```
ssize_t sys_read(int fd, void *buf, size_t count);
```

Assembly Syntax:

```
mov eax, 3        ; System call number for sys_read
mov ebx, fd       ; File descriptor
mov ecx, buf      ; Buffer to read into
mov edx, count    ; Number of bytes to read
int 0x80          ; Interrupt to invoke the system call
```

Write File (sys_write):

```
ssize_t sys_write(int fd, const void *buf, size_t count);
```

Assembly Syntax:

```
mov eax, 4        ; System call number for sys_write
mov ebx, fd       ; File descriptor
mov ecx, buf      ; Buffer containing data to write
mov edx, count    ; Number of bytes to write
int 0x80          ; Interrupt to invoke the system call
```

2.Memory Management System Calls:

Allocate Memory (sys_brk):

```
void *sys_brk(void *addr);
```

Assembly Syntax:

```
mov eax, 45       ; System call number for sys_brk
mov ebx, addr     ; New end of data segment
int 0x80          ; Interrupt to invoke the system call
```

3.Process Management System Calls:

```
int sys_execve(const char *filename, char *const argv[], char *const envp[]);
```

Execute Program (sys_execve):

```
mov eax, 11           ; System call number for sys_execve
mov ebx, filename     ; Pointer to the filename
mov ecx, argv         ; Pointer to the arguments array
mov edx, envp         ; Pointer to the environment variables array
int 0x80              ; Interrupt to invoke the system call
```

4.Miscellaneous System Calls

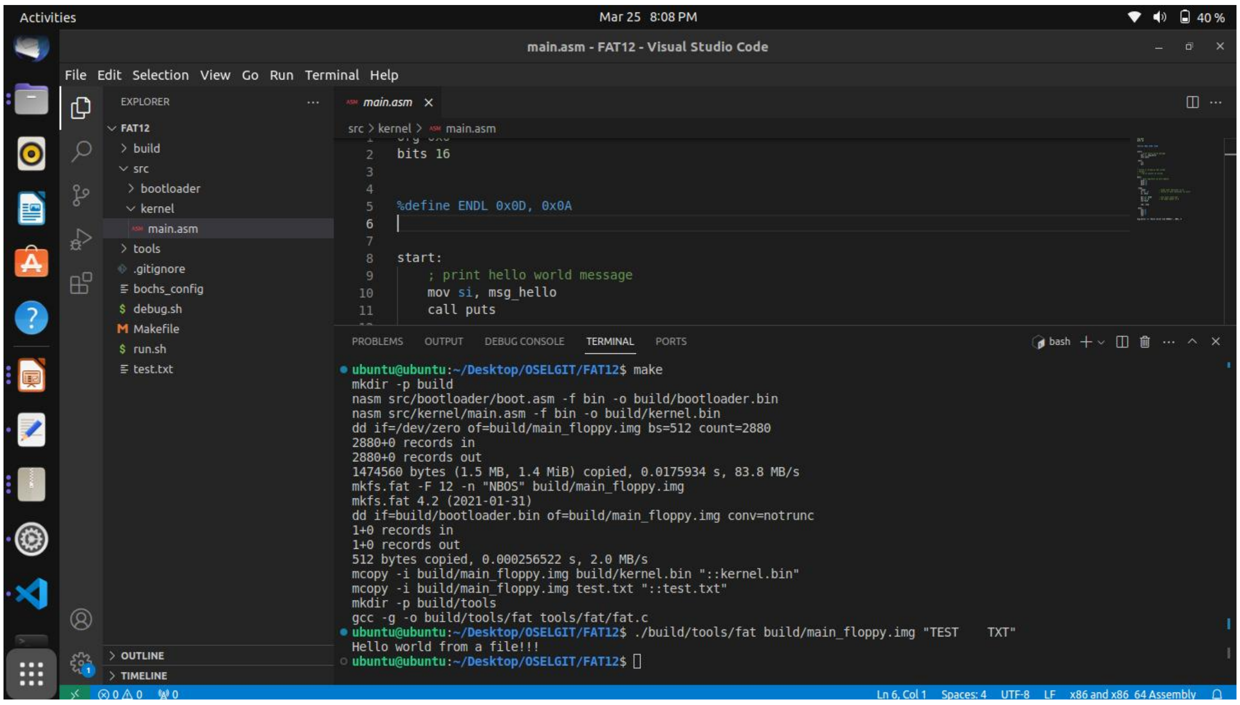
Exit Program (sys_exit):

```
void sys_exit(int status);
```

Assembly Syntax:

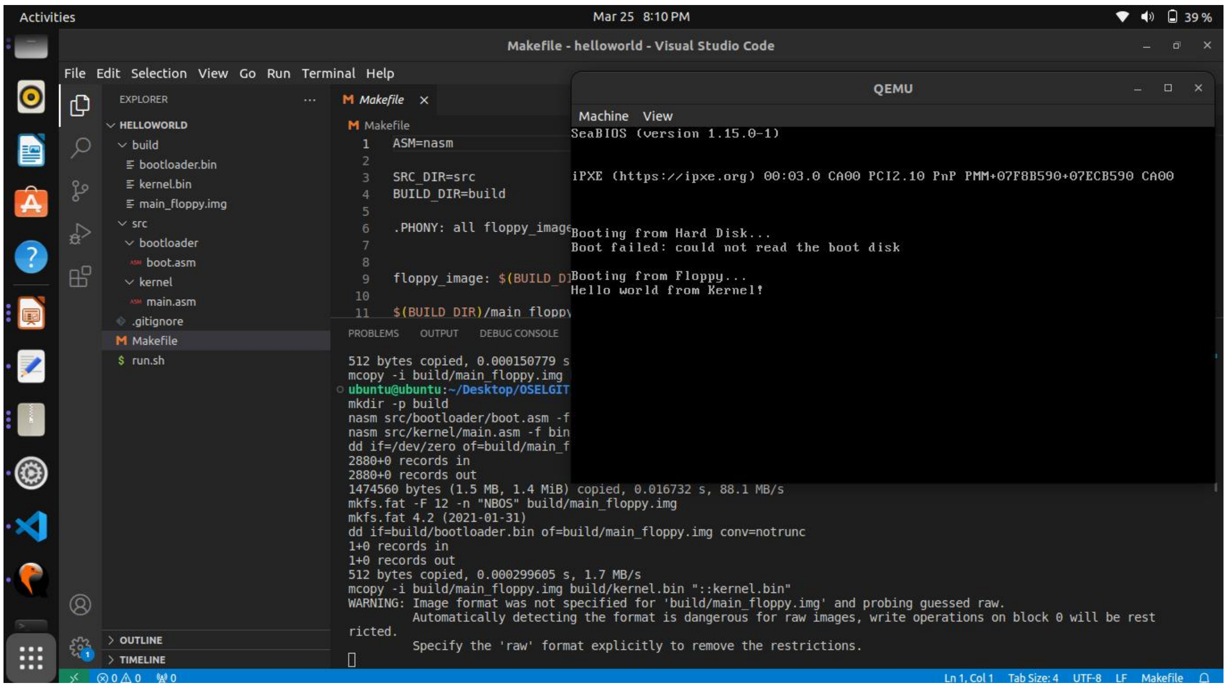
```
mov eax, 1          ; System call number for sys_exit
mov ebx, status     ; Exit status
int 0x80            ; Interrupt to invoke the system call
```

OUTPUT/RESULT



```
bits 16


%define ENDL 0x0D, 0x0A


start:
    ; print hello world message
    mov si, msg_hello
    call puts
```

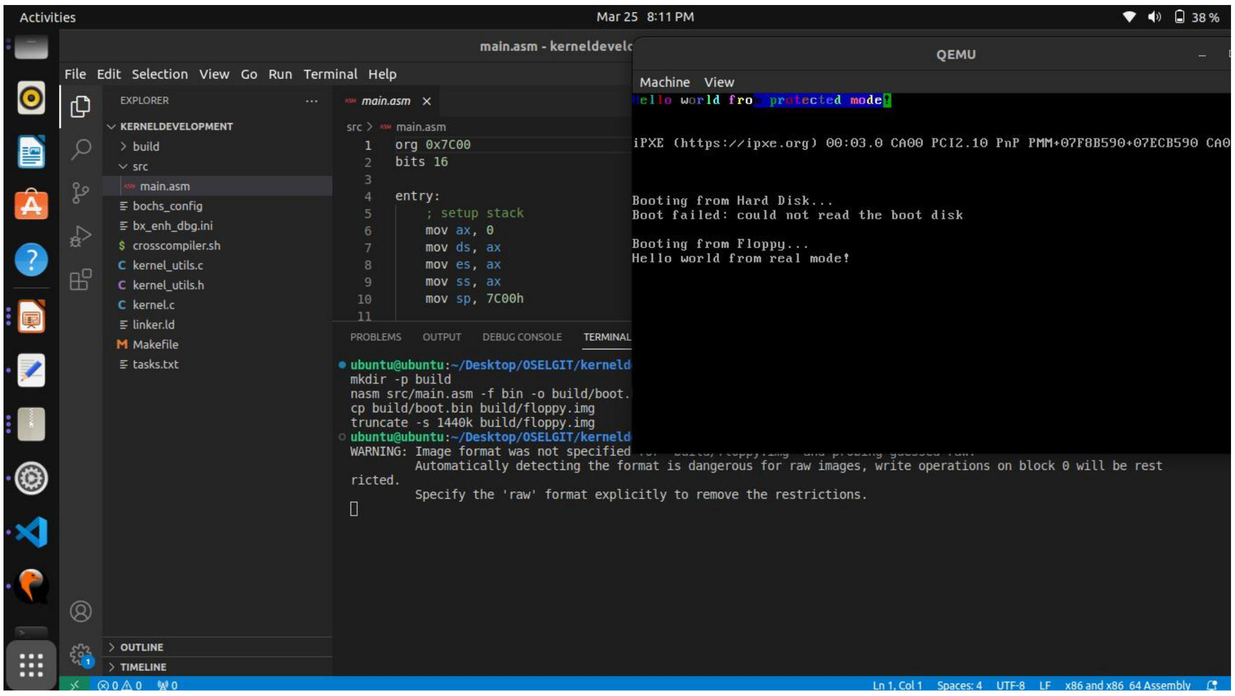
```
ubuntu@ubuntu:~/Desktop/OSELGIT/FAT12$ make
mkdir -p build
nasm src/bootloader/boot.asm -f bin -o build/bootloader.bin
nasm src/kernel/main.asm -f bin -o build/kernel.bin
dd if=/dev/zero of=build/main_floppy.img bs=512 count=2880
2880+0 records in
2880+0 records out
1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.0175934 s, 83.8 MB/s
mkfs.fat -F 12 -n "NBOS" build/main_floppy.img
mkfs.fat 4.2 (2021-01-31)
dd if=build/bootloader.bin of=build/main_floppy.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000256522 s, 2.0 MB/s
mcopy -i build/main_floppy.img build/kernel.bin "::kernel.bin"
mcopy -i build/main_floppy.img test.txt "::test.txt"
mkdir -p build/tools
gcc -g -o build/tools/fat tools/fat/fat.c
ubuntu@ubuntu:~/Desktop/OSELGIT/FAT12$ ./build/tools/fat build/main_floppy.img "TEST    TXT"
Hello world from a file!!!
ubuntu@ubuntu:~/Desktop/OSELGIT/FAT12$ 
```



```
ASM=nasm

SRC_DIR=src
BUILD_DIR=build

.PHONY: all floppy_image


floppy_image: $(BUILD_DI

$(BUILD_DIR)/main_floppy
```

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA00



Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Hello world from Kernel!
```

```
512 bytes copied, 0.000150779 s
mcopy -i build/main_floppy.img
ubuntu@ubuntu:~/Desktop/OSELGIT
mkdir -p build
nasm src/bootloader/boot.asm -f
nasm src/kernel/main.asm -f bin
dd if=/dev/zero of=build/main_f
2880+0 records in
2880+0 records out
1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.016732 s, 88.1 MB/s
mkfs.fat -F 12 -n "NBOS" build/main_floppy.img
mkfs.fat 4.2 (2021-01-31)
dd if=build/bootloader.bin of=build/main_floppy.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000299605 s, 1.7 MB/s
mcopy -i build/main_floppy.img build/kernel.bin "::kernel.bin"
WARNING: Image format was not specified for 'build/main_floppy.img' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be rest
ricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```



```
org 0x7C00
bits 16

entry:
    ; setup stack
    mov ax, 0
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, 7C00h
```

```
ubuntu@ubuntu:~/Desktop/OSELGIT/kerneld
mkdir -p build
nasm src/main.asm -f bin -o build/boot.
cp build/boot.bin build/floppy.img
truncate -s 1440k build/floppy.img
ubuntu@ubuntu:~/Desktop/OSELGIT/kerneld
WARNING: Image format was not specified
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be rest
ricted.
         Specify the 'raw' format explicitly to remove the restrictions.
```

```
ello world fro  protected mode!

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8B590+07ECB590 CA0


Booting from Hard Disk...
Boot failed: could not read the boot disk

Booting from Floppy...
Hello world from real mode!
```

CONCLUSION

In conclusion, the journey of building a kernel using the x86 architecture unveils a fascinating blend of technical intricacies and human ingenuity. Through the exploration of x86's versatile instruction set, dynamic registers, and robust memory addressing capabilities, we delve into the essence of computing itself.

Beyond the realm of hardware, the kernel emerges as the heart and soul of an operating system, orchestrating the harmonious interaction between software and hardware components. As we navigate through the complexities of bootstrapping, memory management, and system call interfaces, we are reminded of the collaborative effort required to bring a kernel to life.

Moreover, the x86 architecture serves as a testament to the evolution of computing, embodying decades of innovation and progress. Its enduring legacy continues to shape the digital landscape, empowering developers to push the boundaries of what is possible.

Ultimately, in our quest to harness the power of x86, we not only unravel the mysteries of hardware and software integration but also embark on a journey of discovery, creativity, and endless possibilities.