

UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA



*Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione*

***Progettazione e sviluppo di un sistema operativo per
architettura Intel IA-32 in Rust***

***Design and development of an operating system for
Intel IA-32 architecture using Rust***

Relatore:
PROF. MANCINI ADRIANO

Laureando:
PALMIERI GIANMATTEO

ANNO ACCADEMICO 2022-2023

Sommario

Il presente lavoro di tesi tratta la progettazione e lo sviluppo di un sistema operativo sperimentale per architettura Intel IA-32, scritto completamente da zero nel linguaggio di programmazione Rust.

L'obbiettivo primario di questa tesi è mostrare il potenziale di Rust come linguaggio di programmazione di basso livello, e di come questo linguaggio sia particolarmente adatto allo sviluppo di sistemi critici, dove la velocità d'esecuzione e la sicurezza sono delle priorità.

Fornisce un esame completo del sistema operativo sviluppato, compresi componenti critici come scheduler della CPU, driver, gestori degli interrupt e delle chiamate di sistema.

Il lavoro discusso in questa tesi vuole essere uno stimolo per la ricerca futura nello sviluppo di sistemi operativi moderni, mostrando come nuove tecnologie come Rust possono essere la base per una nuova generazione di sistemi operativi veloci, sicuri ed affidabili.

Abstract

This thesis presents the design and development of an experimental operating system for the Intel IA-32 architecture. It was created entirely from scratch using the Rust programming language.

The primary objective of this thesis is to showcase Rust as a modern systems programming language, especially well suited for critical applications in low-level environments, where performance and safety are paramount.

This thesis offers a comprehensive analysis of the developed OS, including critical core components, such as the CPU scheduler, device drivers, interrupts and system calls handlers.

The work discussed in this thesis aims to inspire future research in the field of modern OS development, demonstrating how emerging technologies like Rust can be the foundation for a new generation of performant, secure and reliable operating systems.

Contents

1	Introduction	6
1.1	Definition and purpose of an operating system	6
1.2	Rust as systems programming language	7
1.3	Intel IA-32 architecture	8
1.4	Objectives	8
1.5	Thesis structure	9
2	Bootloader	10
2.1	Master Boot Record	10
2.2	BIOS interrupts	12
2.3	Protected mode	14
2.4	Global Descriptor Table	15
3	Kernel	18
3.1	Interrupts	18
3.1.1	Interrupt Descriptor Table	19
3.1.2	Interrupt Service Routines	20
3.1.3	CPU exceptions	21
3.1.4	Programmable Interrupt Controller	21
3.2	Drivers	23
3.2.1	Keyboard driver	23
3.2.2	Advanced Technology Attachment disk driver	24
3.3	Multitasking	26
3.3.1	Context switching	27
3.3.2	CPU scheduler	28
3.3.3	Task manager	29
3.4	System calls	31
3.5	Shell	32
4	Standard library	34

CONTENTS	5
<hr/>	
4.1 Print line macros	34
4.2 Examples	35
5 Conclusions and future development	36
Bibliography	37
Acknowledgements	37

Chapter 1

Introduction

1.1 Definition and purpose of an operating system

An *operating system* is a software that manages the computer hardware[1].

It is the core software component of many computing devices including personal computers, servers, smartphones and embedded devices. It provides several layers of abstraction over the hardware allowing users and applications to use the device through a simple interface, hiding the complexity of the hardware underneath.

Depending on the complexity, they can serve different purposes. For instance, *multitasking* operating systems, like the one discussed in this thesis, can run multiple tasks simultaneously on the same machine.

This means the OS¹ takes the responsibility of managing the available resources by lending them to the processes in an organized, optimized and secure manner. When processes are running the OS has to ensure the absence of unexpected and harmful behaviors, such as processes reading from another process's memory or seizing control of the CPU² for too much time.

For this reason the OS must be as safe as possible, and the *Rust programming language* can accomplish this requirement thanks to its compiler able to guarantee memory-safety and thread-safety at compile-time.

¹Operating System

²Central Processing Unit

1.2 Rust as systems programming language

Most operating systems are written in C^3 , because of its low-level access to hardware, allowing developers to write efficient code and providing features like direct memory manipulation. However, when programming in C, this comes with a price because developers take full responsibility over a program's memory, meaning they have to manage it manually. C requires the programmer to *allocate* and *deallocate* memory explicitly. It provides flexibility in memory usage, but if not done correctly, it can result in serious bugs/issues such as *memory leaks* and *dangling pointers*.

These types of bugs pose significant dangers since malicious actors can exploit them to execute arbitrary code or collect sensitive user data.

Higher-level programming languages address this problem by automating memory management using garbage collection techniques. They implement a *runtime*⁴ responsible for pausing the program, scanning the memory to search for unused variables and automatically deallocating them. The drawbacks are performance overheads and long pause times; for this reason those programming languages are not employed for OS development.

The Rust programming language adopts an entirely different approach to memory management. It uses the unique *ownership* feature, that can guarantee the safety of memory operations at compile time, allowing for very little overhead at runtime. At the same time, it has the performance of a low-level programming language like C and $C++$ and the memory safety of a high level programming language, taking the best from both worlds without having any noticeable drawbacks.

Actually, the concept of ownership is very straightforward - it consists of a small set of rules that governs how memory is managed:

- Each value has an owner.
- There can be only one owner at a time.
- When the owner goes out of scope, the value is automatically dropped.

These rules enable other features such as references and borrowing, allowing Rust to remain flexible as C and C++ while also ensuring memory safety.

³C is a general-purpose programming language

⁴A runtime provides an environment in which programs run

1.3 Intel IA-32 architecture

The Intel IA-32 architecture, also known as x86, is a 32-bit instruction set architecture developed by Intel.

Starting from the late 90s formed the foundation of a wide range of computing devices, most importantly personal computers and servers. It was developed with a particular focus on backward compatibility, allowing it to run a vast array of software developed over decades. For this reason it has significantly shaped the trajectory of computing becoming an industry standard.

IA-32 is a Complex Instruction Set Computer (CISC) architecture, allowing the execution of complex operations within a single instruction. It includes multiple general-purpose registers and supports various data types and addressing modes.

It also features the *protected mode*, an operating mode that introduces hardware-level security features, such as privilege levels, segmentation, virtual memory and paging.

1.4 Objectives

The main objective of this thesis is to provide a comprehensive examination of Felix.

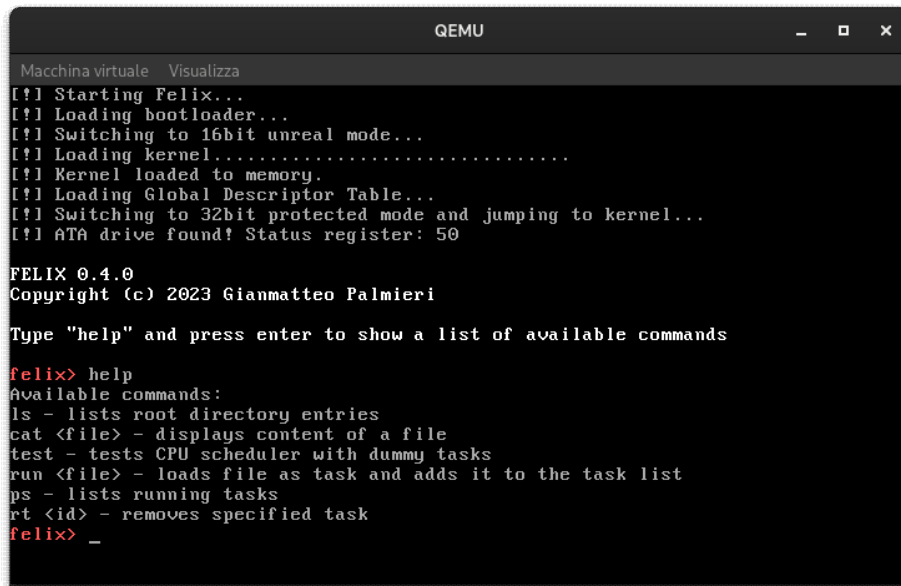
Felix⁵ is an experimental operating system written completely from scratch for the purpose of this thesis.

It is completely open source and its source code is released under the MIT license.

Its source code is freely available on this GitHub repository:

<https://github.com/mrgian/felix>

⁵This name was chosen for its assonance with Unix



```
QEMU
Macchina virtuale Visualizza
[!] Starting Felix...
[!] Loading bootloader...
[!] Switching to 16bit unreal mode...
[!] Loading kernel.....
[!] Kernel loaded to memory.
[!] Loading Global Descriptor Table...
[!] Switching to 32bit protected mode and jumping to kernel...
[!] ATA drive found! Status register: 50

FELIX 0.4.0
Copyright (c) 2023 Gianmatteo Palmieri

Type "help" and press enter to show a list of available commands

feli> help
Available commands:
ls - lists root directory entries
cat <file> - displays content of a file
test - tests CPU scheduler with dummy tasks
run <file> - loads file as task and adds it to the task list
ps - lists running tasks
rt <id> - removes specified task
feli> _
```

Figure 1.1: Felix running in QEMU

1.5 Thesis structure

This thesis is structured in the following chapters:

- Introduction
- Bootloader
- Kernel
- Standard library
- Conclusions

The main part of this thesis follows a structure similar to that of the developed operating system, with a chapter for each main component of the OS.

Chapter 2

Bootloader

The **bootloader** is a program responsible for booting the operating system. Its main task is to load the *kernel* to memory and execute it, but before that it needs to prepare the CPU by switching from *16 bit real mode* to *32 bit protected mode* and setting up memory segments.

There are many full fledged bootloaders, like GRUB¹, capable of running complex operating systems such as Linux and Windows.

However, Felix uses its own tailor made bootloader, also written in Rust. This allows to study the fundamental steps required to boot an operating system, also giving insights on other topics, such as memory segmentation and BIOS interrupts.

Felix uses a two-stage bootloader. The first stage is very memory constrained, because it has to fit in the first sector of the disk, for this reason it has the only purpose of loading the second stage, able to read from disk and load the kernel.

2.1 Master Boot Record

The **Master Boot Record** (MBR) is the very first sector of a disk, also know as *bootsector* because it contains the bootstrap program needed to boot the OS (usually the first stage of the bootloader).[2]

¹GNU GRUB, a very popular bootloader used in many Linux distributions

Offset	Size (bytes)	Description
0x000	440	MBR bootstrap code
0x1B8	4	Unique Disk ID
0x1BC	2	Reserved
0x1BE	16	Partition table entry
0x1CE	16	Partition table entry
0x1DE	16	Partition table entry
0x1EE	16	Partition table entry
0x1FE	2	0x55AA signature

When booting from a disk, BIOS² automatically loads the first sector of that disk to address 0x7c00 and then jumps to that address, actually executing the MBR³ bootstrap program.

In Felix the MBR bootstrap program is the first stage of the bootloader. This is a very constrained environment because a program with a size of only 440 bytes has to be able to load the second stage of the bootloader to memory and then execute it.

For this reason this part of the bootloader is usually written in *Assembly*⁴ code, it needs to be as optimized as possible. However the Rust compiler has the possibility to generate size optimized binaries making it suitable for this purpose. For this reason Felix's first stage bootloader is written in Rust, with the exception of a small portion of the program written in Assembly responsible for:

- Disabling hardware interrupts.
- Setting data segments to zero.
- Setting up the stack.
- Calling Rust main function.

Listing 2.1: felix/boot/src/boot.asm

```
.section .boot, "awx"
.global _start
.code16

_start:
    # disable external interrupts
    cli
```

²Basic Input Output System

³Master Boot Record

⁴Low level programming language very close to machine code

```
# set data segments to zero
xor ax, ax
mov ds, ax
mov es, ax
mov ss, ax
mov fs, ax
mov gs, ax

# setup stack
cld
mov sp, 0x7c00

# call main rust function
call main
```

2.2 BIOS interrupts

The first stage of the bootloader reads data from disk to load the second stage, however, since it operates in a very constrained memory environment, it cannot implement its own disk driver. For this reason the bootloader uses BIOS *interrupts* to access hardware for various tasks, like printing to screen or reading from disk.

BIOS interrupt calls are functions provided by the BIOS to facilitate and abstract access to the hardware. They only work in *16 bit real mode*, for this reason they are not meant to be used as hardware drivers, but only to facilitate the work of the bootloader during bootup. BIOS interrupts don't work when the CPU is in *32 bit protected mode*, so implementing custom drivers is mandatory for the kernel.

Felix uses BIOS interrupt 0x13 to read from disk; this interrupt requires to setup a *Disk Address Packet* structure that specifies how many sectors to read, from what LBA⁵ address and where to write them in memory:

Listing 2.2: felix/boot/src/disk.rs

```
#[repr(C, packed)]
struct DiskAddressPacket {
    size: u8, //size of dap
    zero: u8, //always zero
    sectors: u16, //sectors to read
    offset: u16, //target offset
    segment: u16, //target segment
```

⁵Logical Block Addressing

```
    lba: u64,      //logical block address
}
```

Before issuing the interrupt, the bootloader needs to set some CPU registers:

- **DS:SI** to the DAP's address⁶
- **AH** to 0x42
- **DL** to the drive number (0x80 for the main drive)

Then issuing `INT 0x13` will call the BIOS function that reads from `disk[3]`. If an error occurs during the read, the carry flag will be set. Felix's bootloader uses the `JC` instruction to check for this flag and notify the user.

Listing 2.3: `felix/boot/src/disk.rs`

```
unsafe {
    asm!(
        "mov {1:x}, si", //backup si
        "mov si, {0:x}", //put dap address in si
        "int 0x13",
        "jc fail",
        "mov si, {1:x}", //restore si
        in(reg) dap_address as u16,
        out(reg) _,
        in("ax") 0x4200 as u16,
        in("dx") 0x0080 as u16,
    );
}
```

Another BIOS interrupt is `INT 0x10`, used internally by the print function to print a string to screen:

Listing 2.4: `felix/boot/src/main.rs`

```
fn print(message: &str) {
    unsafe {
        asm!("mov si, {0:x}", //move given string address to si
            "2:",
            "lodsb", //load a byte (next character) from si to
                al
            "or al, al", //bitwise or on al, if al is null set
                zf to true
            "jz 1f", //if zf is true (end of string) jump to end

            "mov ah, 0x0e",
            "mov bh, 0",
```

⁶Disk Address Packet

```
        "out 0xe9, al", //e9 port hack
        "int 0x10", //tell the bios to write content of al
            to screen

        "jmp 2b", //start again
        "1:",
        in(reg) message.as_ptr());
    }
}
```

However, this interrupt is only used in the bootloader because the kernel implements a more complex `print` function that directly writes to video memory.

2.3 Protected mode

Before *protected mode* was introduced, *real mode* was the only *operating mode* on x86 CPUs. For this reason old CPUs, like the *Intel 8088*⁷, are lacking of memory protection features.

This mode allowed access to memory using 16 bit address, this means that the maximum addressable memory was 1MB. This limit was not a problem at the time since all systems had no more than 640KB of RAM.[4]

Nowadays, this is a huge constrain that led to the introduction of 32 bit CPUs and a new operating mode called *protected mode*. The *protected mode* allows accessing memory using 32 bit addresses, bringing the the limit of addressable memory to 4GB. This mode also introduced a lot of new security features that *real mode* was completely lacking, like a new type of *segmentation*, *privilege levels* and *paging*.^[5]

For compatibility reasons, all x86 CPUs begin execution in *16 bit real mode*, for this reason early stages of the bootloader use this mode. Switching to protected mode consists in three steps:

- Setup a *Global Descriptor Table* (discussed in the next paragraph).
- Set *Protection Enable* bit in CR0.
- Long jump to a protected mode code segment.

⁷A popular x86 16 bit CPU from Intel

Listing 2.5: felix/bootloader/src/main.rs

```
fn protected_mode() {
    unsafe {
        //enable protected mode in cr0 register
        asm!("mov eax, cr0", "or al, 1", "mov cr0, eax");

        //push kernel address
        asm!(
            "push {0:e}",
            in(reg) KERNEL_TARGET,
        );

        //jump to protected mode
        asm!("ljmp $0x8, $2f", "2:", options(att_syntax));

        //protected mode start
        asm!(
            ".code32",

            //setup segment registers
            "mov {0:e}, 0x10",
            "mov ds, {0:e}",
            "mov es, {0:e}",
            "mov ss, {0:e}",

            //jump to kernel
            "pop {1:e}",
            "call {1:e}",

            out(reg) _,
            in(reg) KERNEL_TARGET,
        );
    }
}
```

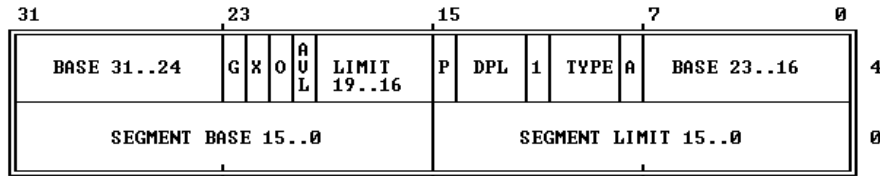
2.4 Global Descriptor Table

Switching to *Protected Mode* requires to setup a *Global Descriptor Table* beforehand. The *Global Descriptor Table* (GDT) is a data structure specific to the x86 architectures. It defines the memory segments, allowing to have hardware protection regarding how memory is accessed by the CPU.[6]

Entries in the GDT⁸ are called *Segment Descriptors* and they are 8 bytes long each. The first entry should always be null.

⁸Global Descriptor Table

The *Segment Descriptor* has a structure like this:



Felix uses a *flat memory model*, meaning its GDT contains only two entries, a *data segment* and a *code segment*. Those two segment have the base set to 0x0, the limit set to 0xffff and the granularity bit set to 1, this means they take up the entirety of the 4GB available memory.

Listing 2.6: felix/bootloader/src/gdt.rs

```
pub static GDT: GlobalDescriptorTable = {
    //segment length (0xffff means all 32bit memory)
    let limit = {
        let limit_low = 0xffff << 0;
        let limit_high = 0xf << 48;

        limit_low | limit_high
    };

    //base address
    let base = {
        let base_low = 0x0000 << 16;
        let base_high = 0x00 << 56;

        base_low | base_high
    };

    //access byte
    let access = {
        let p = 0b1 << 47; //present bit (1 for any segment)
        let dpl = 0b00 << 46; //descriptor privilege level (ring
            , 0 for highest privilege, 3 for lowest)
        let s = 0b1 << 44; //descriptor type bit
        let e = 0b0 << 43; //executable bit
        let dc = 0b0 << 42; //direction bit/conforming bit
        let rw = 0b1 << 41; //readable bit/writable bit
        let a = 0b0 << 40; //accessed bit

        p | dpl | s | e | dc | rw | a
    };

    //flags
```



```
let flags = {
  let g = 0b1 << 55; //granularity flag
  let db = 0b1 << 54; //size flag
  let l = 0b0 << 53; //long mode flag
  let r = 0b0 << 52; //reserved

  g | db | l | r
};

let executable = 0b1 << 43; //set only executable flag again
, instead of setting all values again

//first entry is always zero
//second entry is code segment (default + executable)
//third entry is data segment (default)
let zero = GdtEntry { entry: 0 };
let code = GdtEntry {
  entry: limit | base | access | flags | executable,
};
let data = GdtEntry {
  entry: limit | base | access | flags,
};

GlobalDescriptorTable {
  entries: [zero, code, data],
}
};
```

Chapter 3

Kernel

The **kernel** is the core component of an operating system. Its responsibilities include managing memory and devices, also providing an interface for software applications to use those resources.[1]

Depending on their complexity and goals, kernels can be developed following a model. There are two major models, *microkernel* and *monolithic*. The goal of a microkernel is to run most of its services in *userspace*, resulting in better security and modularity.

Felix uses a monolithic kernel. For this reason all of its services and device drivers run in a single address space running in privileged mode. This allows better efficiency and less code complexity. However, a single bug in one of its components can bring down the entire system.

3.1 Interrupts

An **interrupt** is a signal that the CPU has to respond to. When the CPU receives such signals, it stops to do whatever it was doing to handle the interrupt before returning to its previous task.[7]

There are three types of interrupts:

- **Exceptions:** generated when the CPU runs into an error, like dividing by zero or accessing an invalid memory address.
- **Hardware interrupts:** generated by hardware components when they need to be handled, like pressing a key on the keyboard or moving the mouse

- **Software interrupts:** generated by a specific CPU instruction, they are usually used to invoke system calls or to run BIOS routines

3.1.1 Interrupt Descriptor Table

The **Interrupt Descriptor Table** is a data structure specific to the x86 architecture, its purpose is to tell the CPU where the *Interrupt Service Routines* are located, so that the CPU knows where to jump its execution when an interrupt occurs.[8]

The location of the IDT¹ is stored in the IDT register of the CPU. The content of this register can be updated using the `lidt` instruction. Loading the IDT means updating the IDT register with an address pointing to the *IDT Descriptor*.

Listing 3.1: felix/kernel/src/interrupts/idt.rs

```
//load idt using lidt instruction
pub fn load(&self) {
    let descriptor = IdtDescriptor {
        size: (IDT_ENTRIES * size_of::<IdtEntry>() - 1) as u16,
        //calculate size of idt
        offset: self,
        //pointer to idt
    };

    unsafe {
        asm!("lidt [{0:e}]", in(reg) &descriptor);
    }
}
```

The **IDT Descriptor** is another data structure related to the IDT that contains data about the size and location of the IDT.

Listing 3.2: felix/kernel/src/interrupts/idt.rs

```
#[repr(C, packed)]
pub struct IdtDescriptor {
    size: u16, //idt size
    offset: *const InterruptDescriptorTable, //pointer to idt
}
```

IDT's entries are called gates, each gate corresponds to an interrupt number and has a complex structure. Other than containing the address of the interrupt handler function, gates also define their type, segment selector and flags.

¹Interrupt Descriptor Table

Listing 3.3: felix/kernel/src/interrupts/idt.rs

```
#[repr(C, packed)]
pub struct IdtEntry {
    offset_low: u16,          //lower 16 bits of handler func
        address
    segment_selector: u16, //segment selector of gdt entry
    reserved: u8,           //always zero
    flags: u8,              //entry flags
    offset_high: u16,       //higher 16 bits of handler func
        address
}
```

3.1.2 Interrupt Service Routines

An **Interrupt Service Routine** (ISR) is a routine written to handle an interrupt. When an interrupt occurs, the CPU searches for the corresponding routine in the IDT and then jumps to it.[9] Since ISRs² are called directly by the CPU, their calling protocol differs from normal C functions, for the same reason they use a particular instruction for returning, the `iret` instruction.

The most important thing to know when writing an ISR is that the CPU automatically pushes some registers to the stack before calling the ISR. Those registers are `EFLAGS`, `EIP`, and `CS`. Knowing their value when an interrupt occurs can be very useful for the handler or when debugging an exception.

Most operating systems use dedicated assembly functions as interrupt handlers. Felix however, makes use of Rust *inline assembly* to call another function from the handler and then returning.

Listing 3.4: felix/kernel/src/interrupts/exceptions.rs

```
#[naked]
pub extern "C" fn div_error() {
    unsafe {
        asm!(
            "push 0x00",
            "call exception_handler",
            "add esp, 4",
            "iretd",
            options(noreturn)
        );
    }
}
```

²Interrupt Service Routine

For example, the code above is the ISR for handling the division error exception. Notice the `naked` attribute and the `noreturn` option, those two things make sure that, when compiled, the function is only made by the instructions present in the `asm` block. Also notice the call to `exception_handler`, everything pushed to the stack before this call will be available to the function as arguments, including the registers already pushed by the CPU. This way of passing arguments is called `cdecl` convention.

Listing 3.5: `felix/kernel/src/interrupts/exceptions.rs`

```
#[no_mangle]
pub extern "C" fn exception_handler(int: u32, eip: u32, cs: u32,
    eflags: u32) {
    libfelix::println!("EIP: {:X}, CS: {:X}, EFLAGS: {:b}", eip,
        cs, eflags);
    loop {}
}
```

3.1.3 CPU exceptions

An **exception** is an interrupt triggered by the CPU itself when it runs into an error during the current instruction.^[10] On x86, there are about 20 different CPU exception types. The most important are:

- **Page fault**: occurs on illegal memory accesses.
- **Invalid opcode**: occurs when the current instruction is invalid.
- **General protection fault**: occurs on various kinds of access violations, such as trying to execute a privileged instruction in user-level code or writing reserved fields in configuration registers.
- **Double fault**: this exception is triggered if another exception occurs while calling the exception handler.
- **Triple fault**: this exception is triggered if another exception occurs while the CPU tries to call the double fault handler function. Since triple fault can't be handled, most processors react by resetting themselves and rebooting the operating system.

3.1.4 Programmable Interrupt Controller

The **Programmable Interrupt Controller** (PIC), also known as *8259 PIC*, used to be a tiny chip on the motherboard of older computers, but nowadays is integrated in the CPU die.

Since the CPU can't have an interrupt line for each hardware component, the main purpose of the PIC³ is to receive interrupt signals from the hardware and dispatch them to the CPU when it's ready.[11]

It has 28 pins, 8 of which are wired to the hardware components that can trigger an interrupt. Since 8 is a small number, modern computers started to use two PIC chips wired in cascade with a *master* and a *slave* PIC. Because of its limitations, the PIC is now deprecated in favor of the modern APIC⁴, however it's still supported in modern systems for backwards compatibility.

The most important feature of the PIC is its ability to remap interrupts, meaning it can take the interrupt line and add an offset to it before sending it to the CPU. Remapping interrupts is mandatory because hardware interrupt lines starting from zero are already occupied by CPU exceptions, for this reason hardware interrupts have to be remapped to an offset of 32 minimum. Remapping interrupts means reconfiguring the PICs by sending commands to them through two I/O ports, one command port and one data port. For the primary controller, these ports are 0x20 (command) and 0x21 (data). For the secondary controller, they are 0xa0 (command) and 0xa1 (data).

This procedure consists in initializing the PICs by sending an `init` command, setting the offset, specifying which PIC is master and which is slave and then setting the mode.

Listing 3.6: `felix/kernel/src/drivers/pic.rs`

```
pub fn init(&self) {
    //backup masks, need to restore later
    let mask1 = self.master.read_data();
    let mask2 = self.slave.read_data();

    //send init command
    self.master.send_command(COMMAND_INIT);
    wait();
    self.slave.send_command(COMMAND_INIT);
    wait();

    //set offset
    self.master.write_data(self.master.offset);
    wait();
    self.slave.write_data(self.slave.offset);
    wait();
}
```

³Programmable Interrupt Controller

⁴Advanced Programmable Interrupt Controller

```
//tell master pic that there is a connected slave pic on
    IRQ2
self.master.write_data(4);
wait();
//tell slave pic that he is slave
self.slave.write_data(2);
wait();

//set 8086 mode
self.master.write_data(MODE);
wait();
self.slave.write_data(MODE);
wait();

//restore mask
self.master.write_data(mask1);
self.slave.write_data(mask2);
}
```

3.2 Drivers

A device driver is a software component built to interface a particular hardware device with the kernel. By default an operating system doesn't know how to communicate with the hardware, for this reason it needs drivers to translate operating system's requests into commands that the hardware can understand and execute.[1]

3.2.1 Keyboard driver

The *PS/2 Keyboard* is a device that talks to a PS/2 controller using serial communication.[12]

Even though PS/2 keyboards are not used anymore, they are still supported in modern systems. New motherboards emulate modern USB keyboards as PS/2 devices to support older software.

When a key is pressed, the keyboard sends an interrupt signal telling the kernel to handle it. However the interrupt itself doesn't carry any information about the pressed key, for this reason the keyboard stores a *scancode* in its controller memory, waiting the driver to read it.

A *scancode* is a byte that can be read from the keyboard data port, it contains information about the key and if it was pressed or released.

Reading *scancodes* from the controller is not enough, the driver still has to interpret those bytes to translate them in input data before sending it to the operating system.

Listing 3.7: felix/kernel/src/drivers/keyboard.rs

```
pub extern "C" fn keyboard_handler(charset: [u8; CHAR_COUNT]) {
    //read scancode from keyboard controller
    let scancode: u8;
    unsafe {
        asm!("in al, dx", out("al") scancode, in("dx")
            KEYBOARD_CONTROLLER as u16);
    }

    //notify pics end of interrupt
    PICS.end_interrupt(KEYBOARD_INT);

    //print char
    let key = scancode_to_char(scancode, charset);

    if key != '\0' {
        unsafe {
            SHELL.add(key);
        }
    }
}
```

3.2.2 Advanced Technology Attachment disk driver

Advanced Technology Attachment (ATA), also known as IDE⁵, is a standard connection interface designed for storage devices, such as hard disk drives, floppy disk drives and optical disk drives.

ATA⁶ has been replaced by SATA⁷, that features a more efficient and faster data transfer. However, ATA drives controllers support *ATA PIO Mode*. Even though this mode uses a lot of CPU resources, it makes the implementation the driver very simple. [13] For this reason the ATA disk driver is the first storage driver implemented in Felix.

The most important function of this driver is `read`, it takes three arguments: an LBA⁸ address, the number of sectors to read and a pointer to a target where to write data in memory.

⁵Integrated Drive Electronics

⁶Advanced Technology Attachment

⁷Serial ATA

⁸Logical Block Addressing

Listing 3.8: felix/kernel/src/drivers/disk.rs

```

//read multiple sectors from lba to specified target
pub fn read<T>(&self, target: *mut T, lba: u64, sectors: u16) {
    if !self.enabled {
        libfelix::println!("[ERROR] Cannot read! Disk not
            enabled");
        return;
    }

    //wait until not busy
    while self.is_busy() {}

    unsafe {
        //disable ata interrupt
        asm!("out dx, al", in("dx") 0x3f6, in("al") 0b00000010
            as u8);

        //setup registers
        asm!("out dx, al", in("dx") SECTOR_COUNT_REGISTER, in("
            al") sectors as u8); //number of setcors to read
        asm!("out dx, al", in("dx") LBA_LOW_REGISTER, in("al")
            lba as u8); //low 8 bits of lba
        asm!("out dx, al", in("dx") LBA_MID_REGISTER, in("al") (
            lba >> 8) as u8); //next 8 bits of lba
        asm!("out dx, al", in("dx") LBA_HIGH_REGISTER, in("al")
            (lba >> 16) as u8); //next 8 bits of lba
        asm!("out dx, al", in("dx") DRIVE_REGISTER, in("al") (0
            xE0 | ((lba >> 24) & 0xF)) as u8); //0xe0 (master
            drive) ORed with highest 4 bits of lba

        //send read command to port
        asm!("out dx, al", in("dx") STATUS_COMMAND_REGISTER, in
            ("al") READ_COMMAND);
    }

    let mut sectors_left = sectors;
    let mut target_pointer = target;
    while sectors_left > 0 {
        //a sector is 512 byte, buffer size is 4 byte, so loop
        for 512/4
        for _i in 0..128 {
            //wait until not busy
            while self.is_busy() {}

            //wait until ready
            while !self.is_ready() {}

            let buffer: u32;
            unsafe {

```

```

        //read 16 bit from controller buffer
        asm!("in eax, dx", out("eax") buffer, in("dx")
            DATA_REGISTER);

        //copy buffer in memory pointed by target
        /*(target_pointer as *mut u32) = buffer;
        core::ptr::write_unaligned(target_pointer as *
            mut u32, buffer);

        target_pointer = target_pointer.byte_add(4);
    }
}
sectors_left -= 1;
}

self.reset();
}

```

Notice how `target` is a pointer to a generic data type, allowing this function to write to every data structure.

The function first checks if the drive is enabled, waits until it's not busy and disables interrupts coming from the drive. Then fills up the controller registers with the data needed for the reading, such as the LBA address and how many sectors to read. Finally it sends the read command to the controller.

At this point the drive controller starts filling its buffer with the data that's being read, so a loop copies from the buffer to the target pointer increasing it by the size of the buffer on every iteration. Since a sector is 512 bytes large and the buffer is 4 bytes large, this loop continues to run for 128 (512/4) iterations for each sector.

Reading from the buffer has the effect of erasing it and telling the controller to proceed by reading the next 4 bytes. For this reason the driver checks if the drive is ready in each iteration, this avoids reading from the buffer before it has been filled.

3.3 Multitasking

Felix is a **multitasking** operating system. When the user starts a task, it gets added to a list, then the kernel is responsible to execute that task whenever possible.

Since Felix doesn't support multi-threaded execution, everything runs on a

single core, so it's responsible for sharing available processor time between multiple tasks automatically, giving the user the illusion of the tasks being run simultaneously. This type of execution is called concurrency, and it differs from parallelism because tasks aren't really executed at the same time.

Felix is a preemptive multitasking operating system. It differs from cooperative multitasking systems because tasks don't voluntarily give up the CPU, but instead a task switch is triggered at every timer tick. Every time an interrupt signal arrives from the timer, the handler calls a scheduler function that decides which task will be executed next.[14]

3.3.1 Context switching

The most crucial part of a multitasking operating system is **context switching**. It involves storing the old state of the CPU and retrieving the new state[15]. In this way when a task is stopped and then resumed later it continues its execution like if it was never stopped.

There are two main things that defines *context*, the task stack and the CPU state. Every task in Felix has its own stack, so that it can be restored by the scheduler when it's needed.

This is the timer IRQ⁹ responsible for triggering the scheduler at every timer tick:

Listing 3.9: felix/kernel/src/interrupts/timer.rs

```
//TIMER IRQ
#[naked]
pub extern "C" fn timer() {
    unsafe {
        asm!(
            //disable interrupts
            "cli",
            //save registers
            "push ebp",
            "push edi",
            "push esi",
            "push edx",
            "push ecx",
            "push ebx",
            "push eax",
            //call c function with esp as argument
            "push esp",
            "call timer_handler",
```

⁹Interrupt Request

```

        //set esp to return value of c func
        "mov esp, eax",
        //restore registers
        "pop eax",
        "pop ebx",
        "pop ecx",
        "pop edx",
        "pop esi",
        "pop edi",
        "pop ebp",
        //re-enable interrupts
        "sti",
        //return irq
        "iretd",
        options(noreturn)
    );
}

#[no_mangle]
pub extern "C" fn timer_handler(esp: u32) -> u32 {
    //trigger scheduler and return the esp returned by scheduler
    unsafe {
        let new_esp: u32 = TASK_MANAGER.schedule(esp as *mut
            CPUState) as u32;

        PICS.end_interrupt(TIMER_INT);

        return new_esp;
    }
}

```

The most important instruction to notice in this IRQ is `mov esp, eax`, this instruction sets the stack pointer to the value that just returned from `timer_handler`. This function returns the stack pointer of the next task to be executed, but this stack contains the old CPU state in its bottom part, so popping the CPU registers has the effect to restore the CPU to the state it had before the task was stopped.

3.3.2 CPU scheduler

Felix uses a *round robin algorithm* for its **scheduler**, it gets triggered by the timer interrupt handler. At every timer tick the algorithm chooses the next task in the list and returns a pointer to its CPU state.

Listing 3.10: felix/kernel/src/multitasking/task.rs

```
pub fn schedule(&mut self, cpu_state: *mut CPUState) -> *mut
CPUState {
    //if no tasks return current state
    if self.task_count <= 0 {
        return cpu_state;
    }

    //save current state of current task
    if self.current_task >= 0 {
        self.tasks[self.current_task as usize].cpu_state_ptr =
            cpu_state as u32;
    }

    self.current_task = self.get_next_task();

    self.tasks[self.current_task as usize].cpu_state_ptr as *mut
        CPUState
}
```

Notice how the return value of this function is a pointer to the CPU state of the chosen task, so that the context switcher can set the stack pointer to this pointer and pop the registers to restore the state.

3.3.3 Task manager

The **task manager** is the component responsible for managing the insertion, removal and scheduling of tasks from its tasks list.

It consists of several functions to manage the tasks list and a data structure containing the array of tasks, the number of tasks and the ID of the current running task.

Listing 3.11: felix/kernel/src/multitasking/task.rs

```
pub struct TaskManager {
    tasks: [Task; MAX_TASKS as usize], //array of tasks
    task_count: i8,                    //how many tasks are in
        the queue
    current_task: i8,                  //current running task
}
```

A task is defined by a data structure containing an array of 4K bytes that makes the stack, a pointer to its CPU state and a value that marks if the task wants to run or not.

Listing 3.12: felix/kernel/src/multitasking/task.rs

```
pub struct Task {
```

```
pub stack: [u8; STACK_SIZE],
pub cpu_state_ptr: u32, //pub cpu_state: *mut CPUState,
pub running: bool,
}
```

Adding a new task to the queue means putting the new task in the first available slot in the task array.

Listing 3.13: felix/kernel/src/multitasking/task.rs

```
pub fn add_task(&mut self, entry_point: u32) {
    let free_slot = self.get_free_slot();
    self.tasks[free_slot as usize].init(entry_point);
    self.task_count += 1;
}
```

However, before doing that the task needs to be initialized. Initializing a task means setting up its stack by putting a blank CPU state in its bottom part. In particular this new CPU state has its EIP register set to the entry point of the task, so that the CPU jumps to that address when the task is executed.

Listing 3.14: felix/kernel/src/multitasking/task.rs

```
pub fn init(&mut self, entry_point: u32) {
    //mark task as running
    self.running = true;

    //set cpu state pointer to the bottom part of its stack
    let mut state = &self.stack as *const u8;
    unsafe {
        state = state.byte_add(STACK_SIZE);
        state = state.byte_sub(core::mem::size_of::<CPUState>())
        ;
    }

    //update cpu state pointer
    self.cpu_state_ptr = state as u32;

    let cpu_state = self.cpu_state_ptr as *mut CPUState;

    unsafe {
        //init registers
        (*cpu_state).eax = 0;
        (*cpu_state).ebx = 0;
        (*cpu_state).ecx = 0;
        (*cpu_state).edx = 0;
        (*cpu_state).esi = 0;
        (*cpu_state).edi = 0;
    }
}
```

```

    (*cpu_state).ebp = 0;

    //set instruction pointer to entry point of task
    (*cpu_state).eip = entry_point;

    //set code segment
    (*cpu_state).cs = 0x8;

    //set eflags
    (*cpu_state).eflags = 0x202;
}
}

```

3.4 System calls

A **system call** is a way in which a program in userland can request a service from the kernel[16]. This interface allows the kernel to setup a security model by putting restrictions on the type and scope of operations that can be performed by certain processes.

The most common way to implement system calls is using software interrupts. Felix uses a similar approach to Linux on x86. It consists in setting the arguments in the CPU registers and then issuing interrupt 0x80.

When issuing a system call, the *EAX* register is used to set the number of the system call, the other two registers *EBX* and *ECX* are used for storing arguments.

Felix currently only supports two systems calls:

- **syscall 0**: prints the string pointed by the address in *EBX* with the length specified in *ECX*
- **syscall 1**: removes the current running task from the task list

Listing 3.15: felix/kernel/src/multitasking/task.rs

```

pub extern "C" fn syscall_handler(ecx: u32, ebx: u32, eax: u32)
{
    unsafe {
        match eax {
            //SYSCALL 0, print string pointed by ebx with lenght
            //specified in ecx
            0 => {
                let s = {
                    let slice = slice::from_raw_parts(ebx as *
                        const u8, ecx as usize);
                }
            }
        }
    }
}

```

```
        str::from_utf8(slice)
    };

    print::PRINTER.prints(s.unwrap());
}

//SYSCALL 1, remove current active task
1 => {
    TASK_MANAGER.remove_current_task();
}

_ => {}
}

PICS.end_interrupt(SYSCALL_INT);
}
}
```

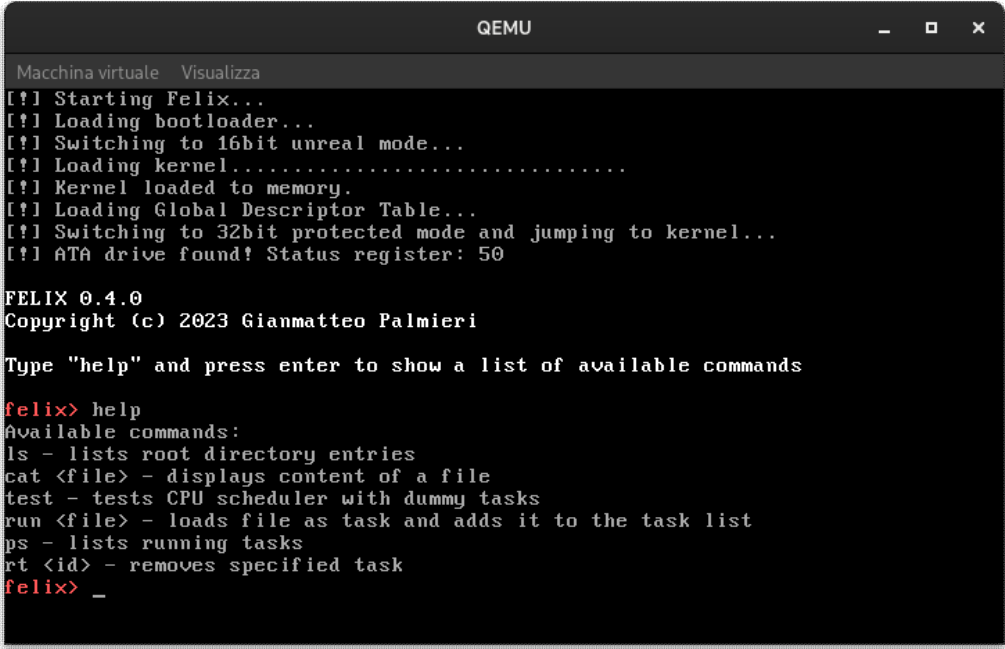
3.5 Shell

The **shell** is a simple interface for the user to interact with the operating system.

It allows the user to input commands for the operating system, those commands are interpreted by the shell so that the kernel can respond (see Figure 3.5).

Felix's shell supports various commands to interact with the OS:

- **help**: shows a list of available commands
- **ls**: lists the entries in the root directory
- **cat *filename***: displays the content of a file in ASCII representation
- **run *filename***: loads an executable file as a task and adds it to the task queue
- **ps**: shows a list of the running tasks
- **rt *id***: removes the specified task from the queue



```
QEMU
Macchina virtuale Visualizza
[!] Starting Felix...
[!] Loading bootloader...
[!] Switching to 16bit unreal mode...
[!] Loading kernel.....
[!] Kernel loaded to memory.
[!] Loading Global Descriptor Table...
[!] Switching to 32bit protected mode and jumping to kernel...
[!] ATA drive found! Status register: 50

FELIX 0.4.0
Copyright (c) 2023 Gianmatteo Palmieri

Type "help" and press enter to show a list of available commands

felix> help
Available commands:
ls - lists root directory entries
cat <file> - displays content of a file
test - tests CPU scheduler with dummy tasks
run <file> - loads file as task and adds it to the task list
ps - lists running tasks
rt <id> - removes specified task
felix> _
```

Figure 3.1: Felix running in QEMU showing its shell

Chapter 4

Standard library

The standard library provides all the necessary implementations to compile and run a program on an operating system.

It implements various useful algorithms and data structures, but most importantly it provides a wrapper around system calls, so that the programmer can interact with the OS more easily without directly calling system calls.

`libfelix` is the standard library implementation for Felix.

4.1 Print line macros

`libfelix` currently provides only one macro, used for printing to the screen.

This macro supports string formatting, allowing to print various data types in text representation to the screen.

Since the kernel only supports printing raw strings through system calls, all the formatting code has to be included in this library.

Luckily the Rust *core library* used in `libfelix` already implements the formatting logic, so the only thing `libfelix` needs to do is to tell the *core library* how to print a single unformatted string, and it does so by implementing a wrapper around `syscall 0`.

Listing 4.1: `felix/lib/src/print.rs`

```
//core lib needs to know how to print a string to implement its
print formatted func
impl fmt::Write for Printer {
    fn write_str(&mut self, s: &str) -> fmt::Result {
```

```

        self.prints(s);
        Ok(())
    }
}

impl Printer {
    pub fn prints(&self, s: &str) {
        unsafe {
            let ptr = s.as_ptr();
            let len = s.len();

            asm!("push eax", "push ebx", "push ecx", "int 0x80",
                "pop ecx", "pop ebx", "pop eax", in("eax") 0, in
                ("ebx") ptr as u32, in("ecx") len as u32);
        }
    }
}

```

4.2 Examples

`libfelix` allows programmers to write and compile programs able to run on Felix. Here is an example of a program that uses `libfelix` to print to screen:

Listing 4.2: `felix/apps/hello/src/main.rs`

```

//HELLO
//Simple program to test libfelix

#![no_std]
#![no_main]

use core::panic::PanicInfo;
use libfelix;

#[no_mangle]
#[link_section = ".start"]
pub extern "C" fn _start() {
    let a = 0xFFFF;
    libfelix::println!("Hello world! {:X}", a);

    loop {}
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}

```

Chapter 5

Conclusions and future development

This thesis has analyzed the design and development of Felix. It has accomplished the goal to demonstrate the power of Rust in low-level and critical environments.

Developing an operating system completely from scratch is a highly complex and challenging task that requires deep understanding of CPU architecture, hardware interaction, embedded software and system-level programming. This project gave me the opportunity to gain knowledge on such topics, but also the ability to leverage this knowledge to develop other projects.

During the development, Felix gained a lot of popularity from other developers, some of which contributed to this project, giving me the opportunity to get to know new talented people in the open source community.

Felix is still missing some key features to make it a truly safe and modern operating system, such as an hardened memory allocator and support for modern architectures like *ARM* and *RISC-V*.

For this reason the development of Felix doesn't stop here but it will continue, hoping to be one of the first steps into the beginning of a new generation of secure and reliable operating systems.

Bibliography

- [1] Silberschatz A., Galvin P. B., Gagne G., Operating System Concepts, Wiley Publishing, 2008.
- [2] [https://wiki.osdev.org/MBR_\(x86\)](https://wiki.osdev.org/MBR_(x86)).
- [3] [https://wiki.osdev.org/Disk_access_using_the_BIOS_\(INT_13h\)](https://wiki.osdev.org/Disk_access_using_the_BIOS_(INT_13h)).
- [4] https://wiki.osdev.org/Real_Mode.
- [5] https://wiki.osdev.org/Protected_Mode.
- [6] https://wiki.osdev.org/Global_Descriptor_Table.
- [7] <https://wiki.osdev.org/Interrupts>.
- [8] https://wiki.osdev.org/Interrupt_Descriptor_Table.
- [9] https://wiki.osdev.org/Interrupt_Service_Routines.
- [10] <https://wiki.osdev.org/Exceptions>.
- [11] https://wiki.osdev.org/8259_PIC.
- [12] https://wiki.osdev.org/PS/2_Keyboard.
- [13] https://wiki.osdev.org/ATA_PIO_Mode.
- [14] https://wiki.osdev.org/Multitasking_Systems.
- [15] https://wiki.osdev.org/Context_Switching.
- [16] https://wiki.osdev.org/System_Calls.

Acknowledgements

I would like to give my warmest thanks to my friends and family, in particular my parents, for supporting me during this whole time, even in my darkest moments. All this would not have been possible without you.

I would also like to give special thanks to everyone who took an interest in Felix and to my secret weapon for supporting and showing me the way during the development of Felix.

Finally, I would like everyone in the open source community for making the world better every day.