Mentoring Operating System (MentOS)

Fundamental concepts

Enrico Fraccaroli

enrico.fraccaroli@univr.it

University of Verona
Department of Computer Science

2020/2021

Table of Contents

Mentoring **O**perating **S**ystem (**MentOS**)

MentOS

What...

MentOS (**Ment**oring **O**perating **S**ystem) is an open source educational operating system. MentOS can be freely downloaded from a public github repository: github.io/MentOS/

Goal...

The goal of MentOS is to provide a project environment that is realistic enough to show how a real Operating System work, yet simple enough that students can understand and modify it in significant ways.

MentOS
Who?


Enrico Fraccaroli	**Alessandro Danese**

2019

Mirco De Marchi **Luigi Capogrosso**

Bachelor in Comp. Science

2020

Andrea Cracco

Bachelor in Bioinformatics

Linda Sacchetto **Marco Berti**

Master in Embedded Systems

2021

Daniele Nicoletti **Filippo Ziche**

Bachelor in Comp. Science

MentOS

Why...

There are so many operating systems, why did we write MentOS?

It is true, there are a lot of education operating system, BUT how many of them follow the guideline defined by Linux?

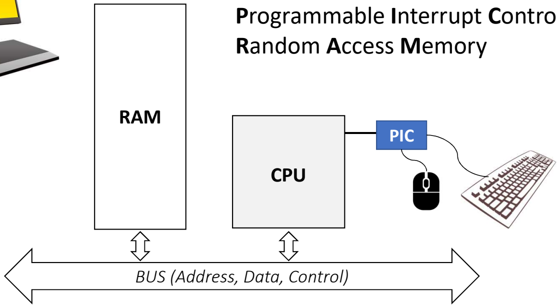
MentOS aims to have the same Linux's data structures and algorithms. It has a well-documented source code, and you can compile it on your laptop in a few seconds!

If you are a beginner in Operating-System developing, perhaps MentOS is the right operating system to start with.

Fundamental concepts

The big picture

Central **P**rocessing **U**nit
Programmable **I**nterrupt **C**ontroller
Random **A**ccess **M**emory



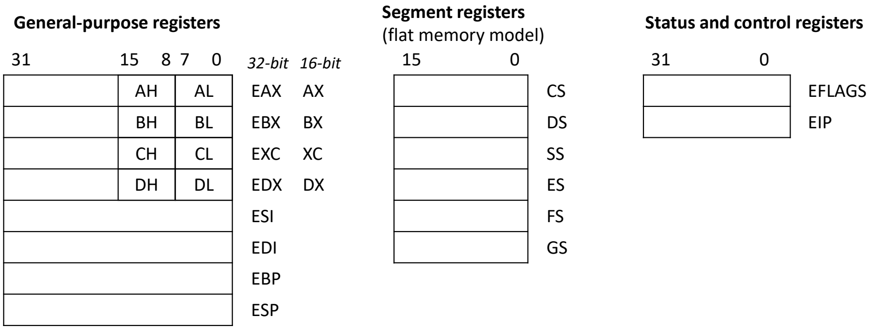
Fundamental concepts

Central Processing Unit (CPU)

CPU registers

There are **three** types of registers:

- general-purpose data registers;
- segment registers;
- status control registers.



General-purpose registers

The **eight** 32-bit **general-purpose** registers are used to hold operands for logical and arithmetic operations, operands for address calculations and memory pointers. The following shows what they are used for:

- `EAX`: Accumulator for operands and results data;
- `EBX`: Pointer to data in the DS segment;
- `ECX`: Counter for loop operations;
- `EDX`: I/O pointer;
- `ESI`: Pointer to data in the segment pointed to by the DS register;
- `EDI`: Pointer to data in the segment pointed to by the ES register;
- `EBP`: Pointer to data on the stack (in the SS segment);
- `ESP`: Stack pointer (in the SS segment).

Status and control registers

The **two** 32-bit **status control** registers are used for:

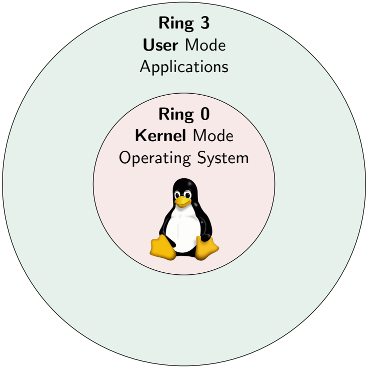
- `EIP`: Instruction pointer (also known as "program counter");
- `EFLAGS`: Mantain group of status, control, system flags.

Table with some of the flags:

Bit	Description	Category	Bit	Description	Category
0	Carry flag	Status	11	Overflow flag	Status
2	Parity flag	Status	12-13	Privilege level	System
4	Adjust flag	Status	16	Resume flag	System
6	Zero flag	Status	17	Virtual 8086 mode	System
7	Sign flag	Status	18	Alignment check	System
8	Trap flag	Control	19	Virtual interrupt flag	System
9	Interrupt enable flag	Control	20	Virtual interrupt pending	System
10	Direction flag	Control	21	Able to use CPUID instruction	System

Not listed bit are reserved. What is the privilege level of a CPU?

Privilege levels



Most modern x86 kernels use only two privilege levels, 0 and 3.

There are **four** privilege levels, numbered 0 (**most** privileged) to 3 (**least** privileged).

At any given time, an x86 CPU is running in a specific privilege level, which determines what code can and cannot execute.

Which of the following operations can process do when the CPU is in **user mode**?

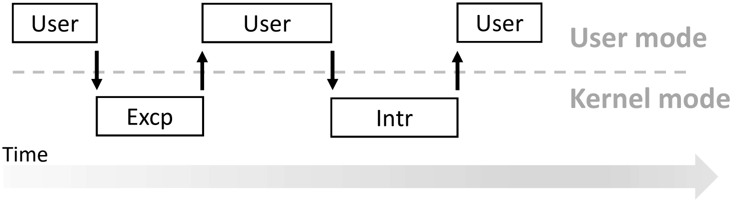
1. open a file;
2. print on screen;
3. allocate memory.

Context switch (Overview)

Every time CPU changes privilege level, a **context switch** occurs!

Example of events making CPU change execution mode:

A mouse click, type of a character on the keyboard, a system call...

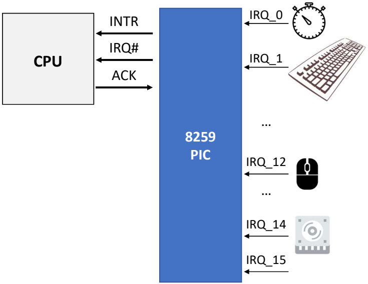


How many times does the CPU change execution mode when a user presses a key of the keyboard?

Fundamental concepts

Programmable Interrupt Controller (PIC)

Programmable Interrupt Controller (PIC)



16 IRQ lines, numbered

- from 0 (highest priority)
- to 15 (lowest priority)

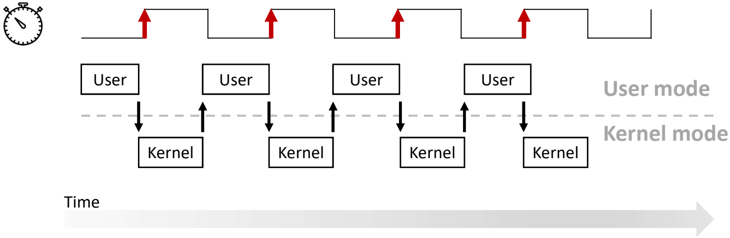
Why do we have a timer in `IRQ_0`?

A programmable interrupt controller is a components combining several interrupt requests onto one or more CPU lines.
Example of interrupt request:

- a key on the keyboard is pressed
- PIC rises INTR line and presents IRQ_1 to CPU
- CPU jumps into Kernel mode to handle the interrupt request
- CPU reads from the keyboard the key pressed
- CPU sends back ACK to notify that IRQ_1 was handled
- CPU jumps back to User mode

IRQ_0, Timer!

The timer is a hardware component aside the CPU. At a fixed frequency, the timer rises a signal connected to the IRQ_0 of PIC.



Linux fixes the timer frequency to 100 Hz. The CPU runs a user process for maximum 10 milliseconds, afterwards Kernel has back the control of CPU.

Fundamental concepts

Memory organization

Memory organization (32-bit system)

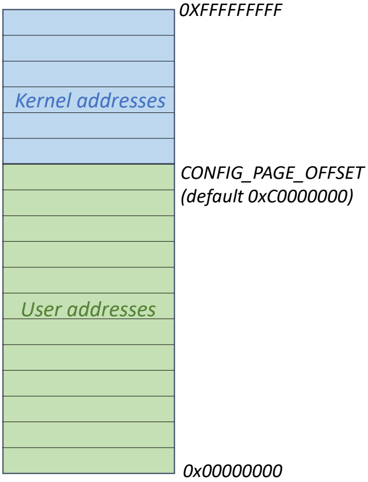


Figure: Kernel and User space.

The Kernel applies Virtual Memory to maps virtual addresses to physical addresses.

RAM is virtually split in Kernel space (1GB) and User space (3GB).

CPU in Ring 0 has visibility of the whole RAM.

CPU in Ring 3 has visibility of User space only.

Kernel doubly-linked list

Circular, doubly-linked list (1/7)

Introduction

Operating system kernels, like many other programs, often need to maintain lists of data structures. To reduce the amount of duplicated code, the kernel developers have created a **standard implementation** of circular, doubly-linked lists.

Pros:

- Safer/quicker than own ad-hoc implementation.
- Comes with several ready functions.

Cons:

- Pointer manipulation can be tricky.

Circular, doubly-linked list (2/7)

Definition

To use the list mechanism kernel developers defined the `list_head` data structure as follow:

```
typedef struct list_head {
    struct list_head *next, *prev;
} list_head_t;
```

A `list_head` represent a node of a list!

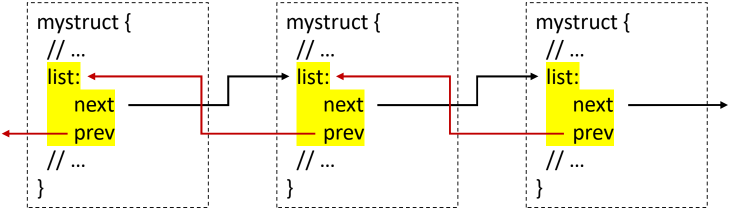
Circular, doubly-linked list (3/7)

Usage

To use the Linux list facility, we need only embed a `list_head` inside the structures that make up the list.

```
struct mystruct {
    //...
    list_head_t list;
    //...
};
```

The instances of mystruct can now be linked to create a doubly-linked list!

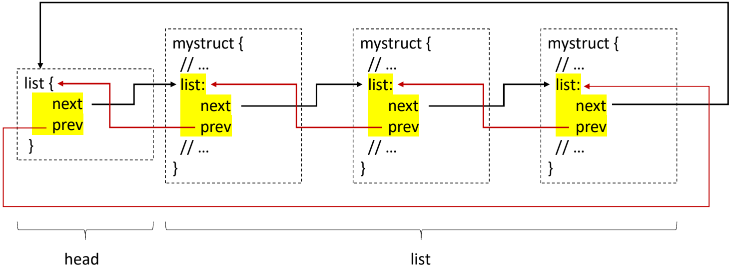


Circular, doubly-linked list (4/7)

Mechanism in Detail

The head of the list **must be** a standalone `list_head_t` structure.

```
list_head_t list;
struct mystruct {
    //...
    list_head_t list;
    //...
};
```



The head is always present in a circular, doubly-linked list!
If a list is empty, then only its head exists!

Circular, double-linked list (5/7)

Support functions (1/3)

Support functions to use with a circular, doubly-linked list.

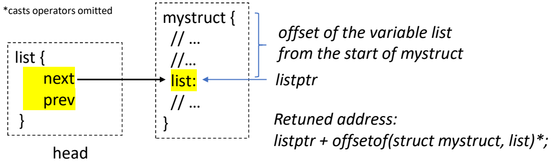
- `list_head_empty(list_head_t *head)`:
  Returns a nonzero value if the given list is empty.
- `list_head_add(list_head_t *new, list_head_t *listnode)`:
  This function adds the `new` entry immediately after the `listnode`.
- `list_head_add_tail(list_head_t *new, list_head_t *listnode)`:
  This function adds the `new` entry immediately before the `listnode`.
- `list_head_del(list_head_t *entry)`:
  The given entry is removed from the list.

Circular, double-linked list (6/7)

Support functions (2/3)

- `list_entry(list_head_t *ptr, type_of_struct, field_name)`:
  Returns the struct embedding a list_head. In detail:
  - `ptr` is a pointer to a `list_head_t`;
  - `type_of_struct` is the type name of the struct embedding a `list_head_t`;
  - `field_name` is the name of the pointed `list_head_t` within the struct.

```
// Example showing how to get the first mystruct from a list
list_head_t *listptr = head.next;
struct mystruct *item = list_entry(listptr, struct mystruct, list);
```



Circular, double-linked list (7/7)

Support functions (3/3)

- `list_for_each(list_head_t *ptr, list_head_t *head)`:

  Iterates over each item of a doubly-linked list. In detail:

  - `ptr` is a free variable pointer of type `list_head_t`;
  - `head` is a pointer to a doubly-linked list's head node.

  Starting from the first list's item, at each call `ptr` is filled with the address of the next item in the list until its head is reached.

```c
list_head_t *ptr;
struct mystruct *entry;
// Inter over each mystruct item in list
list_for_each(ptr, &head) {
    entry = list_entry(ptr, struct mystruct, list);
    // ...
}
```