Mentoring Operating System (MentOS)

fundamental concepts

Alessandro Danese

University of Verona
alessandro.danese@univr.it

Version 1.0.0

Table of Contents

Mentoring Operating System

MentOS

What...

MentOS (Mentoring Operating system) is an open source educational operating system. MentOS can be freely downloaded from a public github repository: *https://luigicapogrosso.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

Why...

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

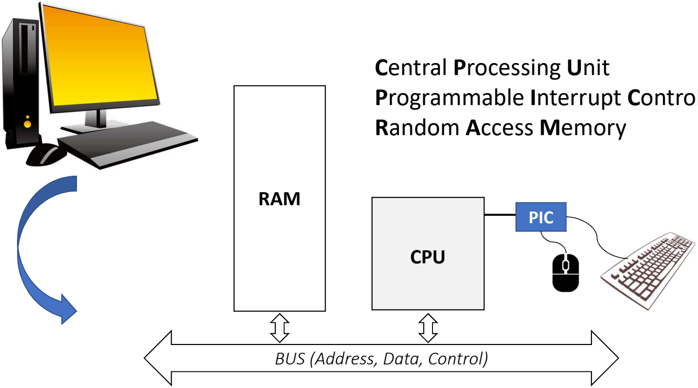
It is true, there are a lot of education operating system, BUT
how many of them follow the guideline defined by Linux?
MentOs aims to have the same Linux's data structures and algorithms. It has a well-documented source code, and you can compile it on your laptop in a few seconds!

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

Fundamental concepts

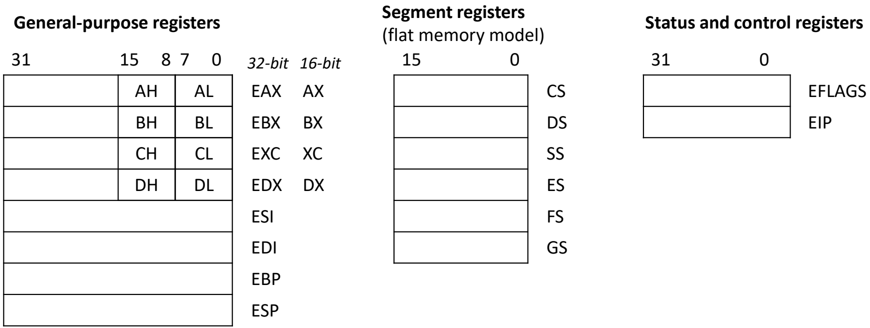
The big picture

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



CPU registers

There are three types of registers: general-purpose data registers, segment registers, and 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

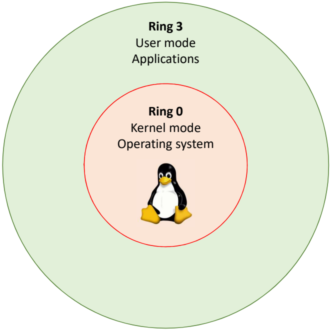
- EIP: Instruction pointer (also be called "program counter").
- EFLAGS register contains a group of status, control, system flags.

Bit	Description	Category
0	Carry flag	Status
2	Parity flag	Status
4	Adjust flag	Status
6	Zero flag	Status
7	Sign flag	Status
8	Trap flag	Control
9	Interrupt enable flag	Control
10	Direction flag	Control

Bit	Description	Category
11	Overflow flag	Status
12-13	Privilege level	System
16	Resume flag	System
17	Virtual 8086 mode	System
18	Alignment check	System
19	Virtual interrupt flag	System
20	Virtual interrupt pending	System
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.

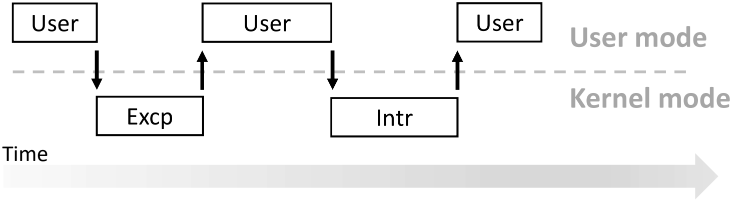
Which of the following operations can process do when the CPU is in user mode?

- open a file
- print on screen
- allocate memory

Context switch (Overview)

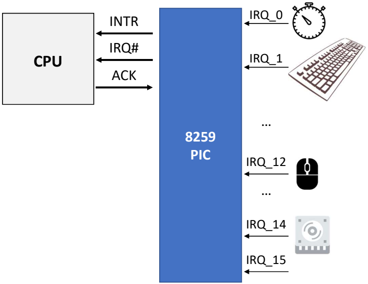
Every time CPU changes privilege level, a context switch occurs!

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?

Programmable Interrupt Controller (PIC)



16 IRQ lines, numbered from 0 (highest priority) to 15 (lowest priority)

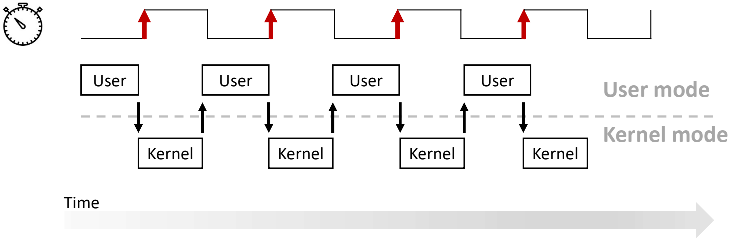
Why do we have a timer in IRQ_0?

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.

Memory organization (32-bit system)

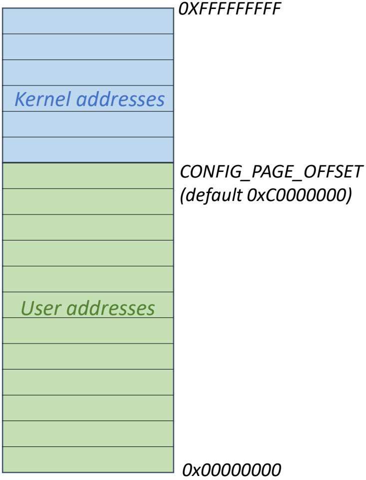


Figure: Kernel and User space.

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

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

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

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

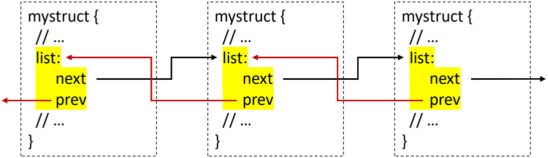
A *list_head* represent a node of a list!

Circular, doubly-linked list

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

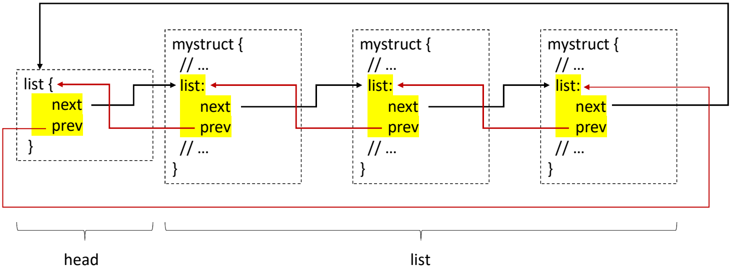
```
struct mystruct {
  //...
  struct list_head list;
  //...
};
```

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



Circular, doubly-linked list

The head of the list <u>must be</u> a standalone list_head structure.



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

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

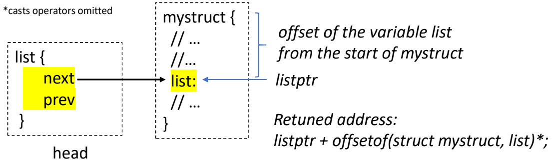
- *list_head_empty(struct list_head *head)*:
  Returns a nonzero value if the given list is empty.
- *list_head_add(struct list_head *new, struct list_head *listnode)*:
  This function adds the *new* entry immediately after the *listnode*.
- *list_head_add_tail(struct list_head *new, struct list_head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

Circular, double-linked list

Support functions... (continue)

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

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



Circular, double-linked list

Support functions... (continue)

- *list_for_each(struct list_head *ptr, struct list_head *head)*:
  Iterates over each item of a doubly-linked list. In detail:
  ptr is a free variable pointer of type struct list_head, and *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
struct list_head *ptr;
struct mystruct *entry;
// Inter over each mystruct item in list
list_for_each(ptr, &head) {
  entry = list_entry(ptr, struct mystruct, list);
  // ...
}
```