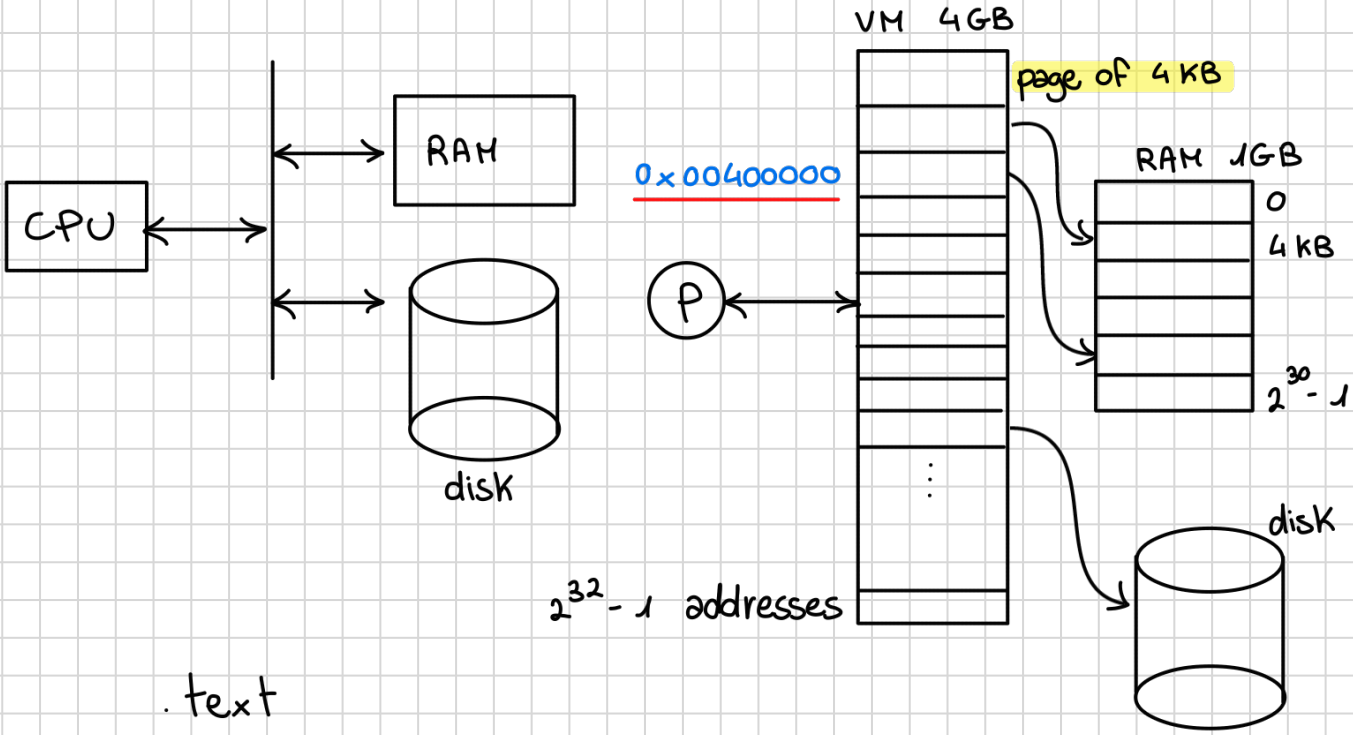
The virtual memory was introduced because in the '60s the RAM memory had a capacity of a few KB and the computer scientists were struggling to create and run large programs.

So basically the problem was: what if my program is larger than a few KB?

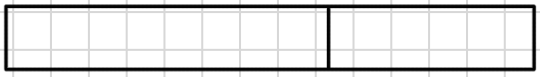
This problem was solved by introducing the virtual memory, which basically made the program believe that the RAM is large, but in fact that wasn't true.



.text

0x00400000 li t0, 0

↳ this is a virtual address that should be translated into a physical one



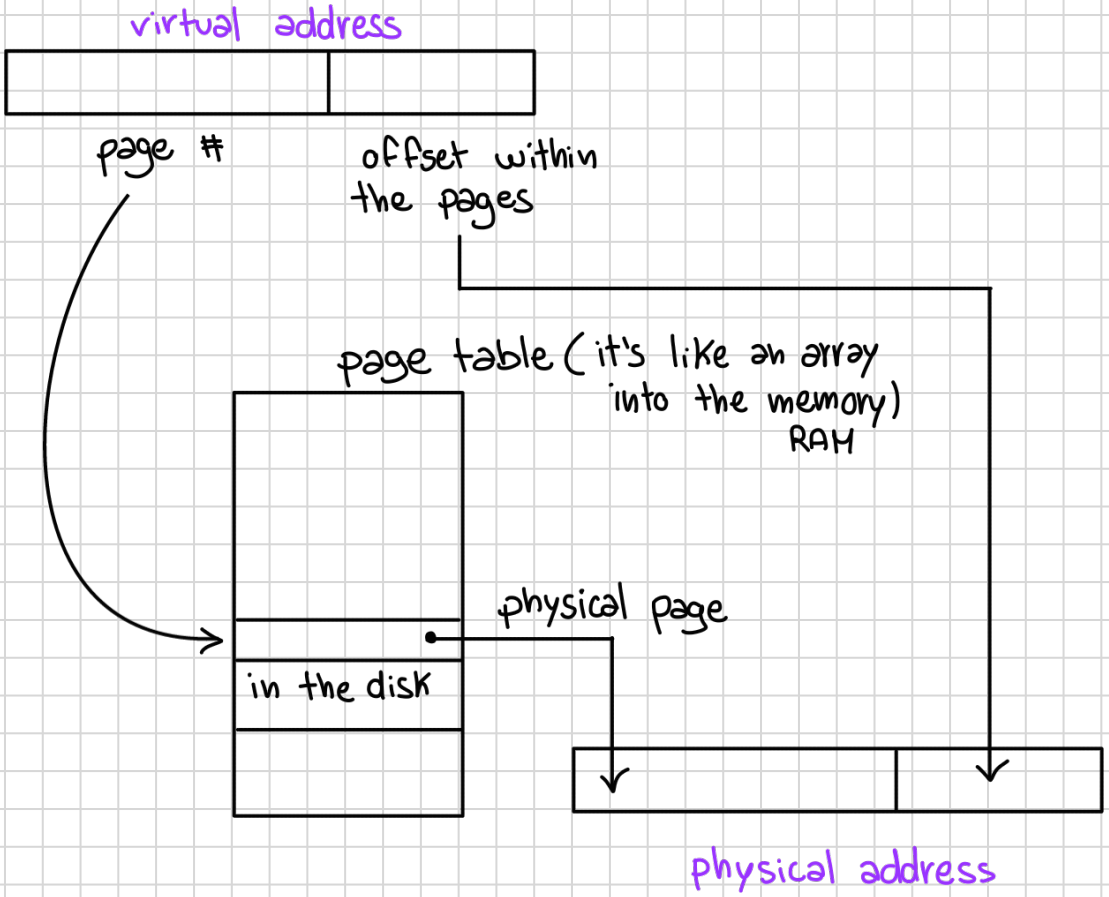
page # (20 bits) | offset within the pages (12 bits)

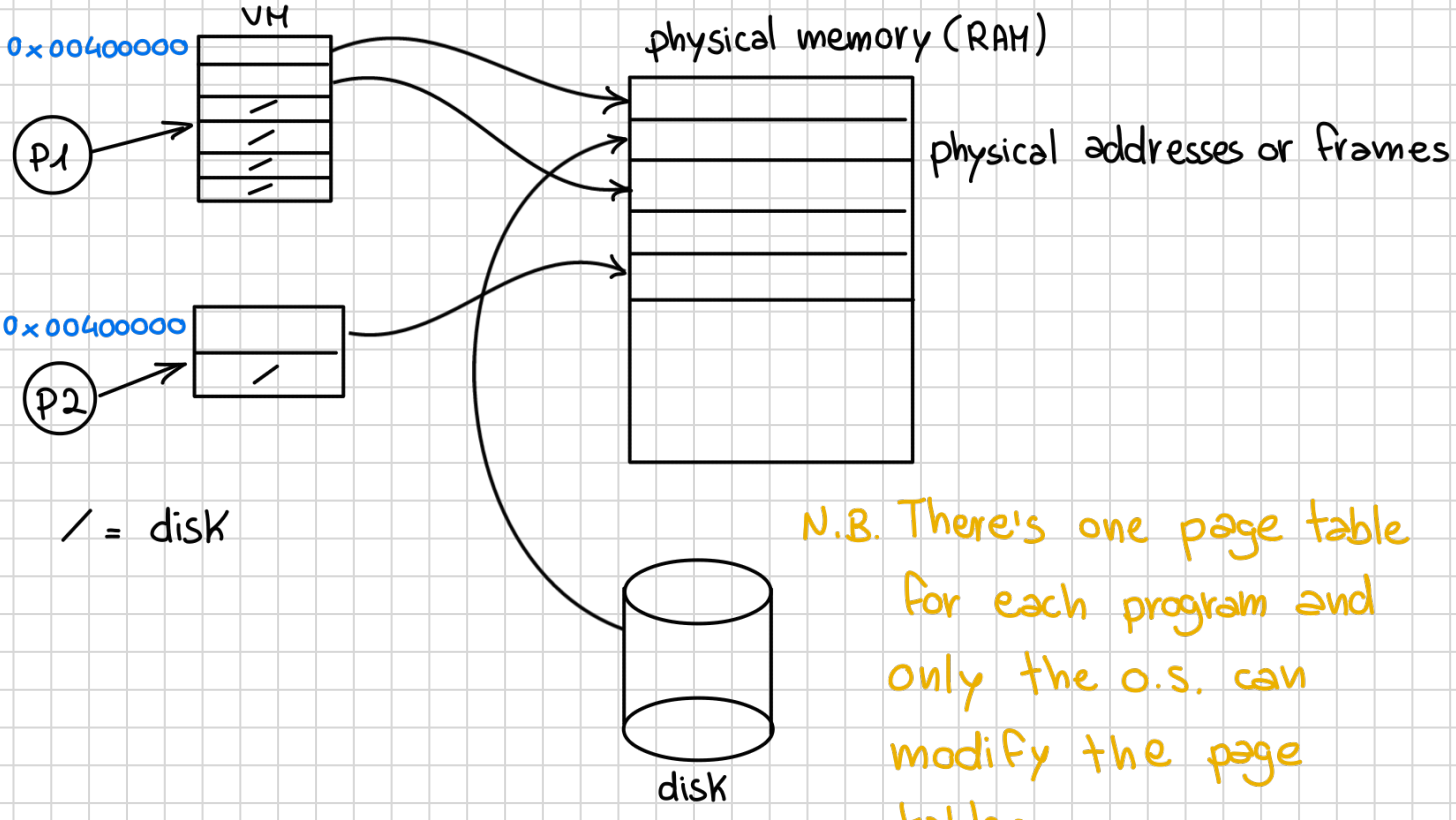
0x00400000

page # | offset

To perform the translation we must know where a virtual address

is actually located into the physical memory.
To know this, we make use of a page table.





N.B. There's one page table for each program and only the o.s. can modify the page tables.

Both programs P1 and P2 start at the same virtual address, but after the translation, it's drawn that they correspond to different

physical addresses.

A virtual memory is as big as needed (but that's not always true: consider the Apple's o.s. preceding the MACOS X).

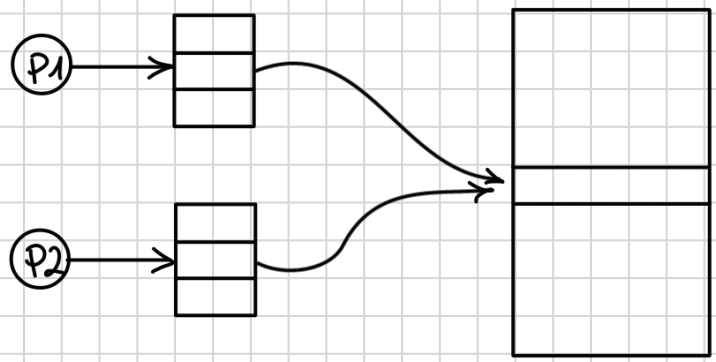
When a program hits a virtual page that redirects to a page of the disk (/), then, that page will be loaded from the disk into a free page of the RAM.

It may happen that the page table is full, so we need to remove one of the pages (maybe another program's page) multiple times. ⟶ Thrashing.

* The o.s. establishes which of the pages should be replaced. *

Nowadays, we use the virtual memory system to keep the various active processes isolated into the memory.

In order to make all this process as fast as possible, a kind of cache is used: TLB (translation lookaside buffer). The system basically caches the physical pages into this kind of cache. If the number of misses in this cache is very low, then the VM isn't so expensive.



Two programs sharing the same page. (handled by the o.s.). It's useful when you share common libraries, but DON'T

use absolute addresses because they create problems.

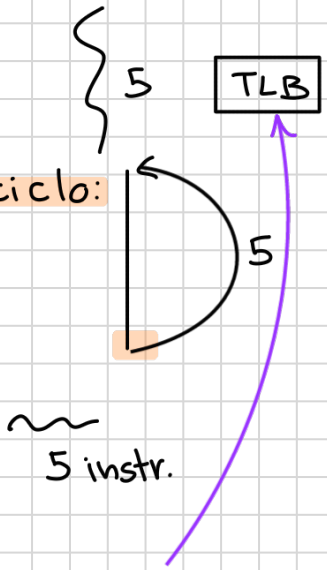
How many pages are used by this program? 6

```
.data 0x10010000
    .word ____________
    .word 4096
```

5 pages for the data because a word is 4 bytes and there are 4.096 items, so $4 \cdot 4.096 = 16kB$, but considering that there are 2 words, we need one xtra page.

```
.text 0x00400000
ciclo:
    5 instr.
```

1 page is starting at this address and it's 4kB big. In this case, 1 page is more than enough to store the program.



The TLB is a cache that's able to cache the map between the virtual pages and the physical pages.

How many page faults for the .data? 5

1st : the size of the array

the other 4 page faults occur while loading and reading the elements of the array. 1 single page covers 4.000 bytes, so every 4.000 bytes a page fault occur.

How many page faults for the .text? 1

How many mappings?

6 (one for each page)

If the TLB is big enough to store 6 pages, then the n. of misses on the TLB is gonna be 6.

TLB miss rate (.text): $\frac{1}{20.000}$ ← one miss because the .text fits in one page.

20k accesses on the .text

TLB miss rate (.data): $\frac{5}{4.096}$

Anyways the miss rate is extremely low on the TLB, so the VM system is very efficient, because the accesses to the page table are really rare.

The virtual memory translation can be done before or after the cache.

before: the cache has physical addresses.

↑ when we share memory, this is a good one.

If the cache has virtual addresses, and several programs use the same cache, then the same address can be translated into 2 different physical addresses but the cache doesn't know it and so we get an error. ??? :"

in this case we have to do the translations everytime.

after: " " virtual " .

↑ better

because in this case we have to do the translations only due to the cache misses.