Embedded Linux (PARRT 2)

- PAGING & SEGMENTATION
- VIRTUAL MEMORY UPON THE PAGING TECHNIQUES.

ENG.KEROLES SHENOUDA

Part 1 Summary ☺

Address Binding

Address binding is **the process of mapping** the program's logical or virtual addresses to corresponding physical or main memory addresses

Compile Time Binding

if memory **location** is **fixed** and **known** at compile time, absolute code with absolute address can be generated.

Load Time Binding

if Memory location in memory is **unknown** at compile time AND location is fixed.
The relocatable code with relative address can be generated.
Binding is done when the program is **loaded** into the memory

Execution Time Binding

if processes can be moved in memory during execution
Binding is delayed until **run time**
Requires hardware support! (**MMU**) for address mapping

MMU is the **Hardware device** that maps **virtual** to **physical** address.

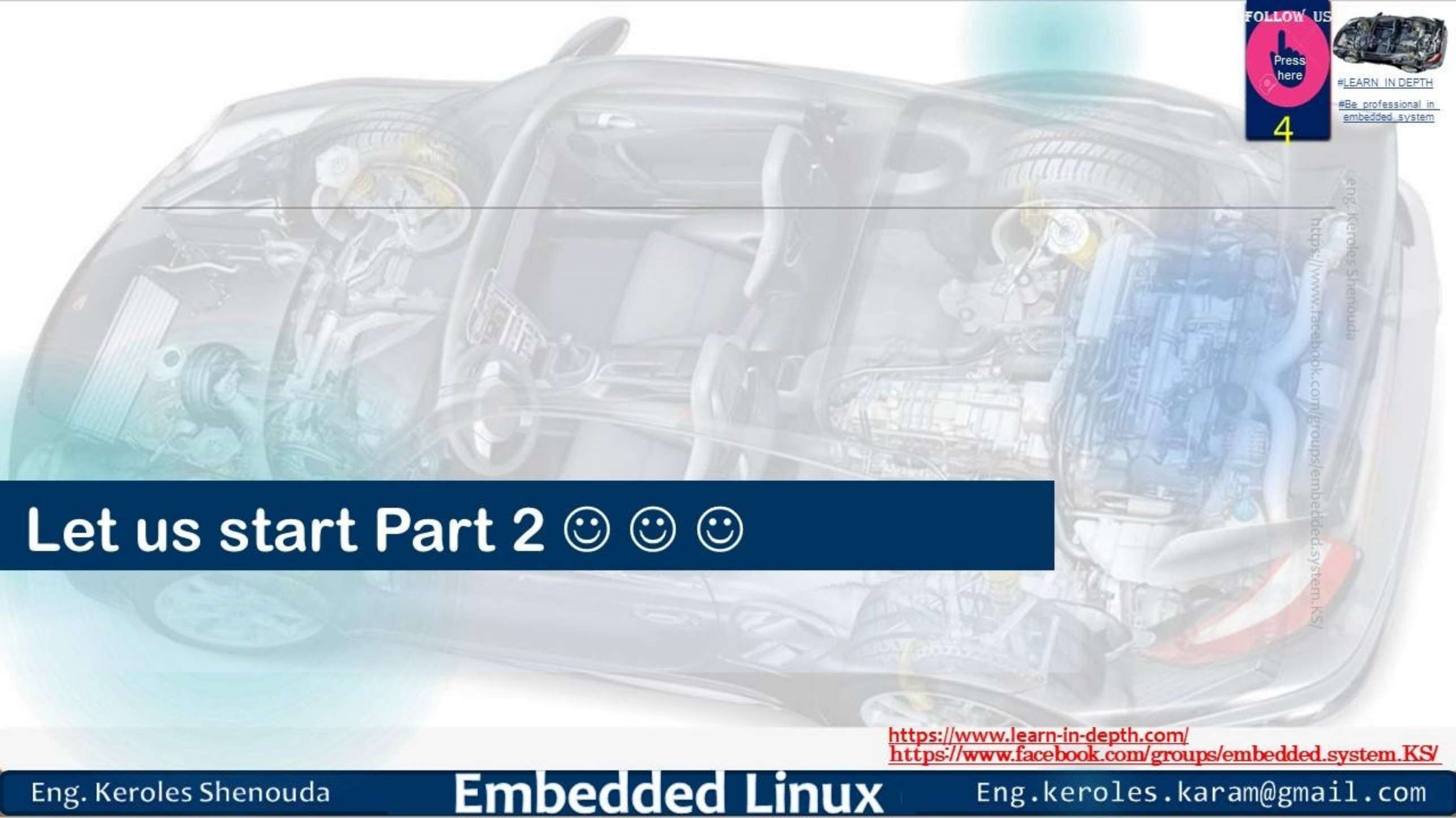
Dynamic Allocation

There are three algorithms:
First-fit, **Best-fit** and **Worst-fit**

Problem

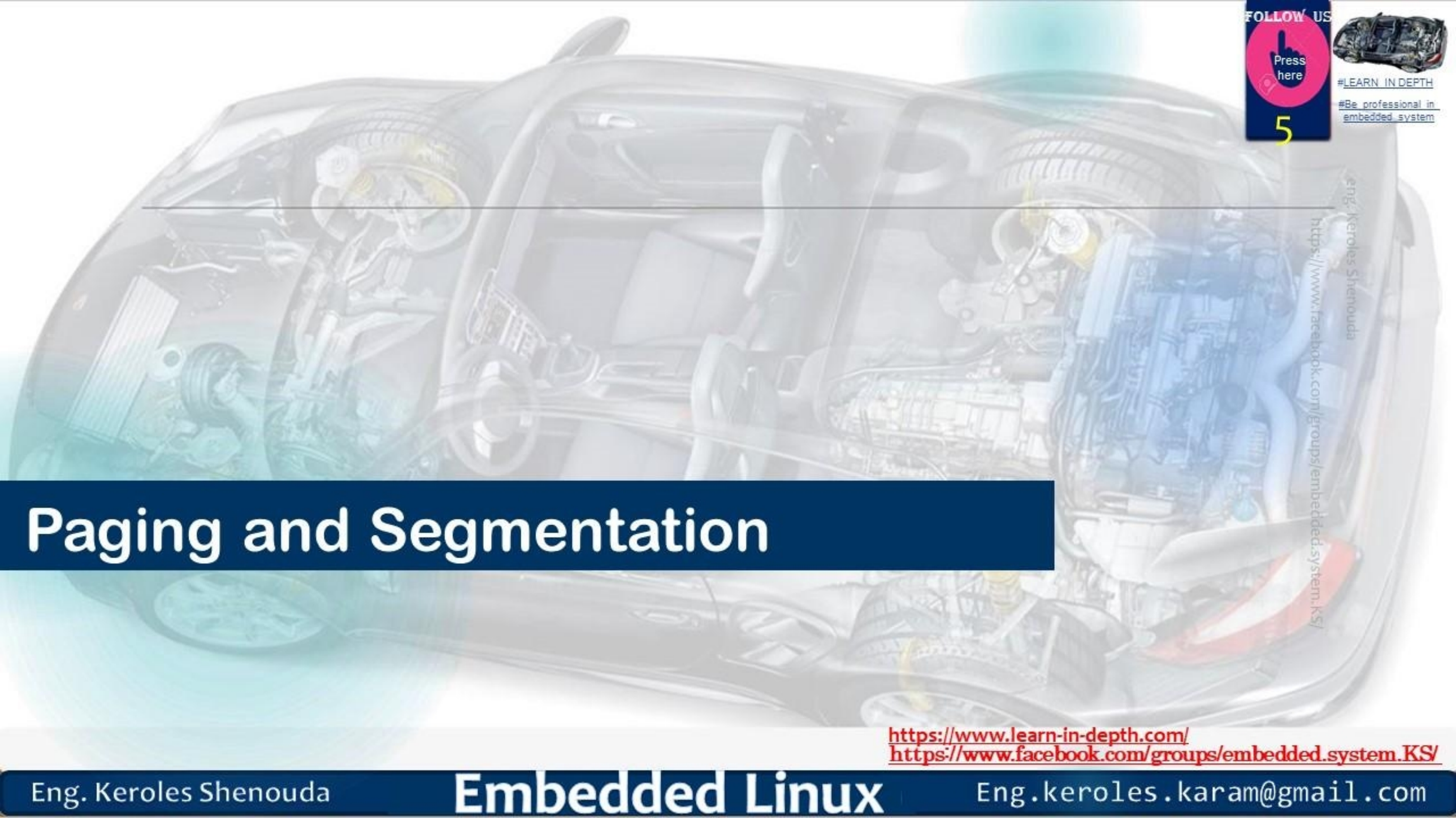
Internal Vs External Fragmentation

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



Let us start Part 2 ☺ ☺ ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



Paging and Segmentation

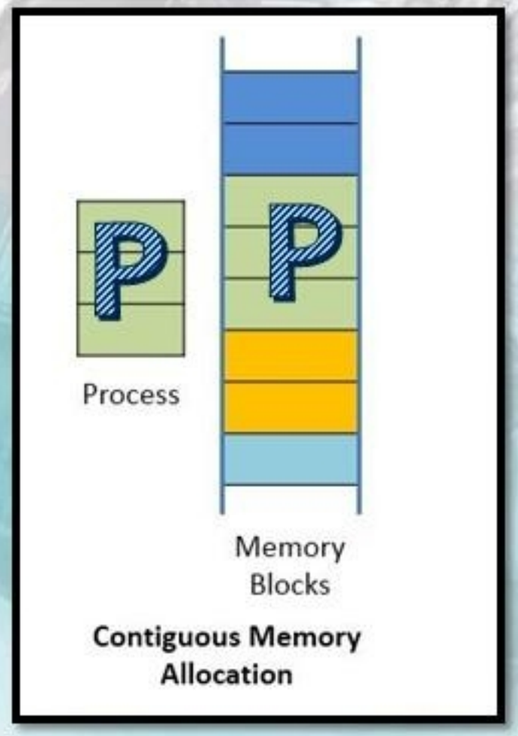
Difference Between Contiguous and Noncontiguous Memory Allocation

Contiguous storage allocation:-
Contiguous storage allocation implies that a **program's data and instructions** are assured to occupy **a single contiguous memory**

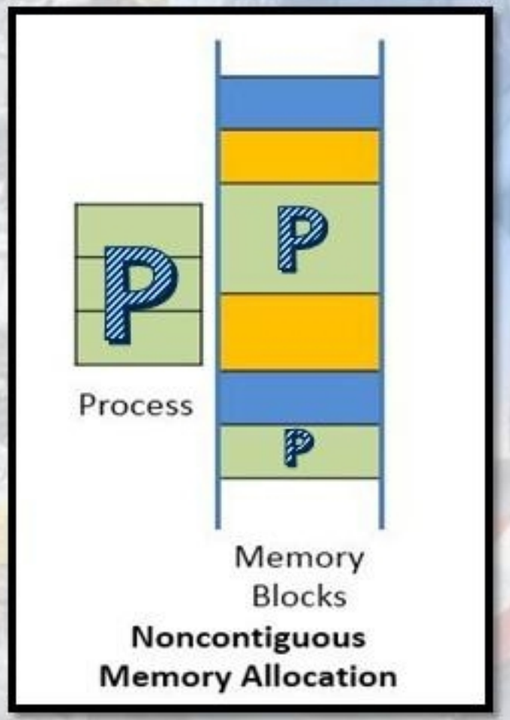
Non-contiguous storage allocation:-
In non-contiguous storage allocation, **a program's data and instructions** may occupy non-contiguous area of memory. **Examples** are:-
1.Paging.
2.Segmentation.

Difference Between Contiguous and Noncontiguous Memory Allocation

Contiguous Memory Allocation Vs Noncontiguous Memory Allocation



BASIS THE COMPARISON	CONTIGUOUS MEMORY ALLOCATION	NONCONTIGUOUS MEMORY ALLOCATION
Basic	Allocates **consecutive blocks of memory** to a **process.**	Allocates **separate blocks of memory** to a process.
Overheads	Contiguous memory allocation **does not have the overhead of** address translation while execution of a process.	Non-contiguous memory allocation **has overhead of address translation** while execution of a process.
Execution rate	**A process executes faster** in contiguous memory allocation	**A process executes quite slower** comparatively in noncontiguous memory allocation.

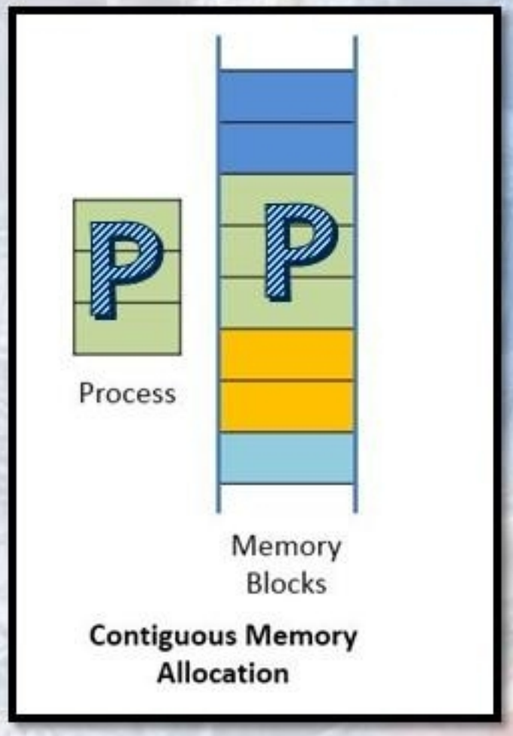


Problems with Contiguous Allocation

1. **Hard** to **share** programs
2. **Hard** to **grow** address space
   - Programs may want to grow during execution
     - More room for stack, heap allocation, etc
3. **Internal fragmentation** may happen
4. **External fragmentation** may happen

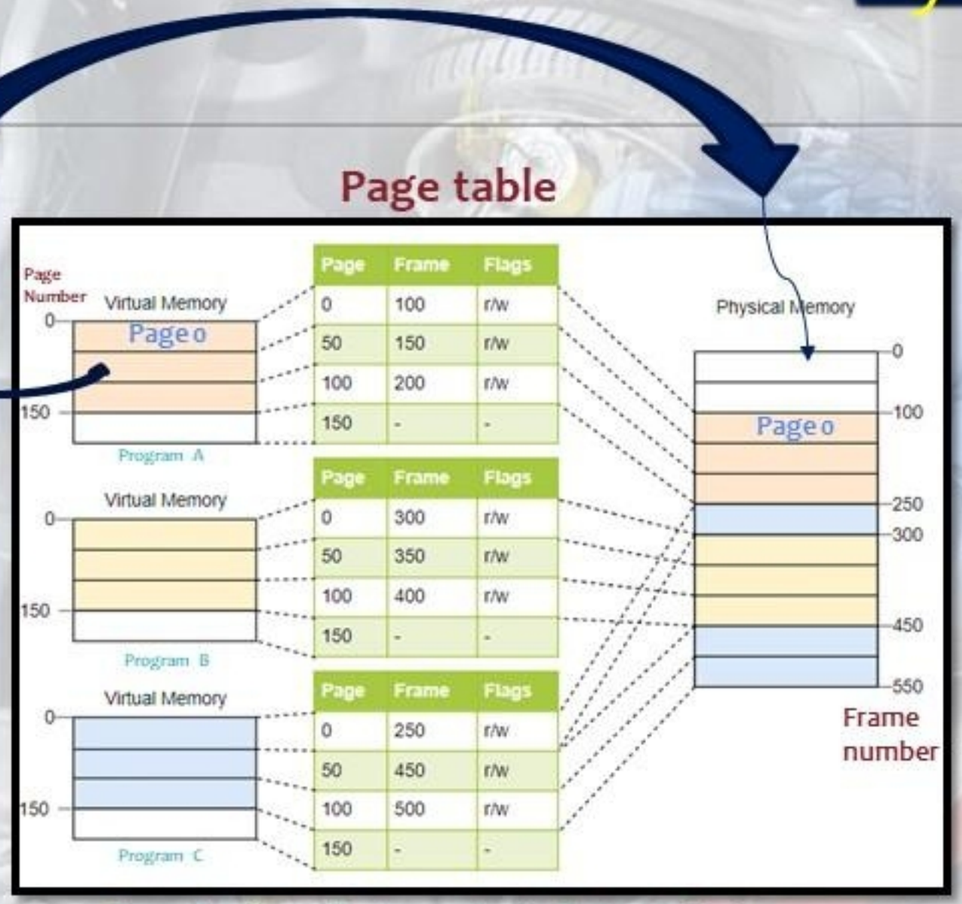
Soln:

- 1 &2 &4: Segmentation
- 1 & 3 &5: Paging
- 1-5: Segmentation + Paging



Paging

- Many operating systems have been using paging technique.
- Divide physical memory into **fixed-sized blocks** called **frames**
  - (size is power of 2, between 512 bytes and 8192 bytes "8KB").
- Divide logical memory into **blocks** of same size called **pages**.
- Keep track of all free frames.
  - To run a program of **size n pages**, need to find ***n* free frames** and load program.
  - Set up a page table to translate **logical** to **physical** addresses. (MMU looks up page table)
- Entire program image **resides on disk**.
  - Ex. Program have page **X** can be either
    - **Already loaded** in memory page frame Y, or
    - **Never been loaded before**, it is in disk
- When the program starts, **just load the first page only**.
- The rest of pages are loaded in memory *on-demand*
- Pages can be placed **anywhere** in memory.

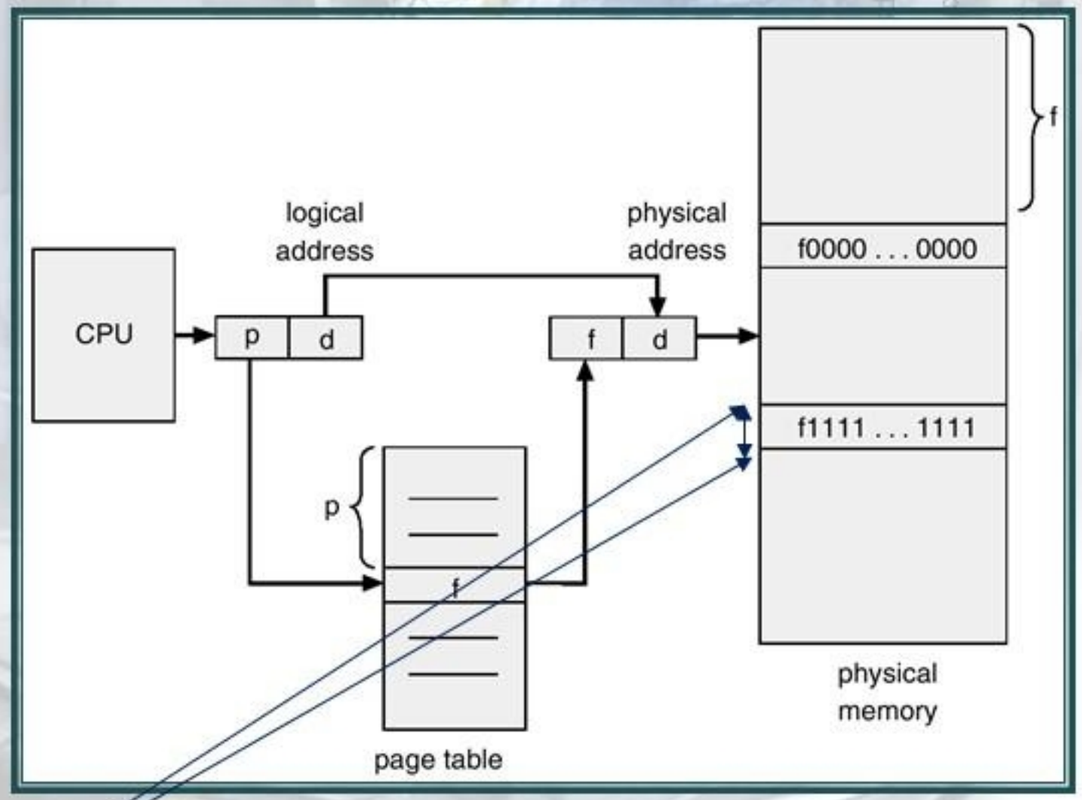


The advantage of using paging technique is that program image does not need to be loaded into memory continuously.

Address Translation in Paging

- The logical Address generated by CPU is divided into two parts:
  - *Page number (p)* - used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* - combined with base address to define the physical memory address that is sent to the memory unit.

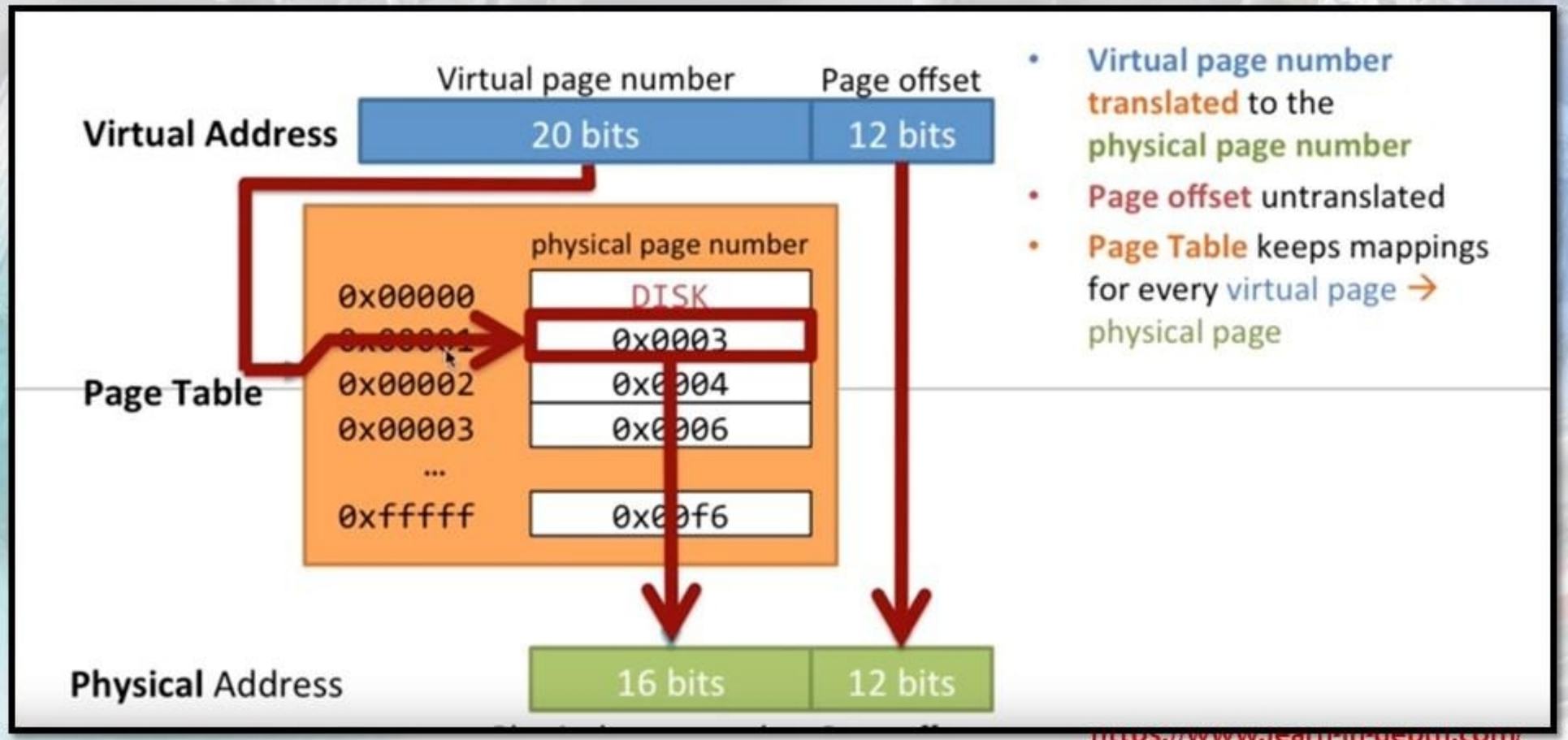
EX. Given address length 32 and page size 4KB



2^12 = 4096/1024 =4 KB

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

how to do a page table lookup

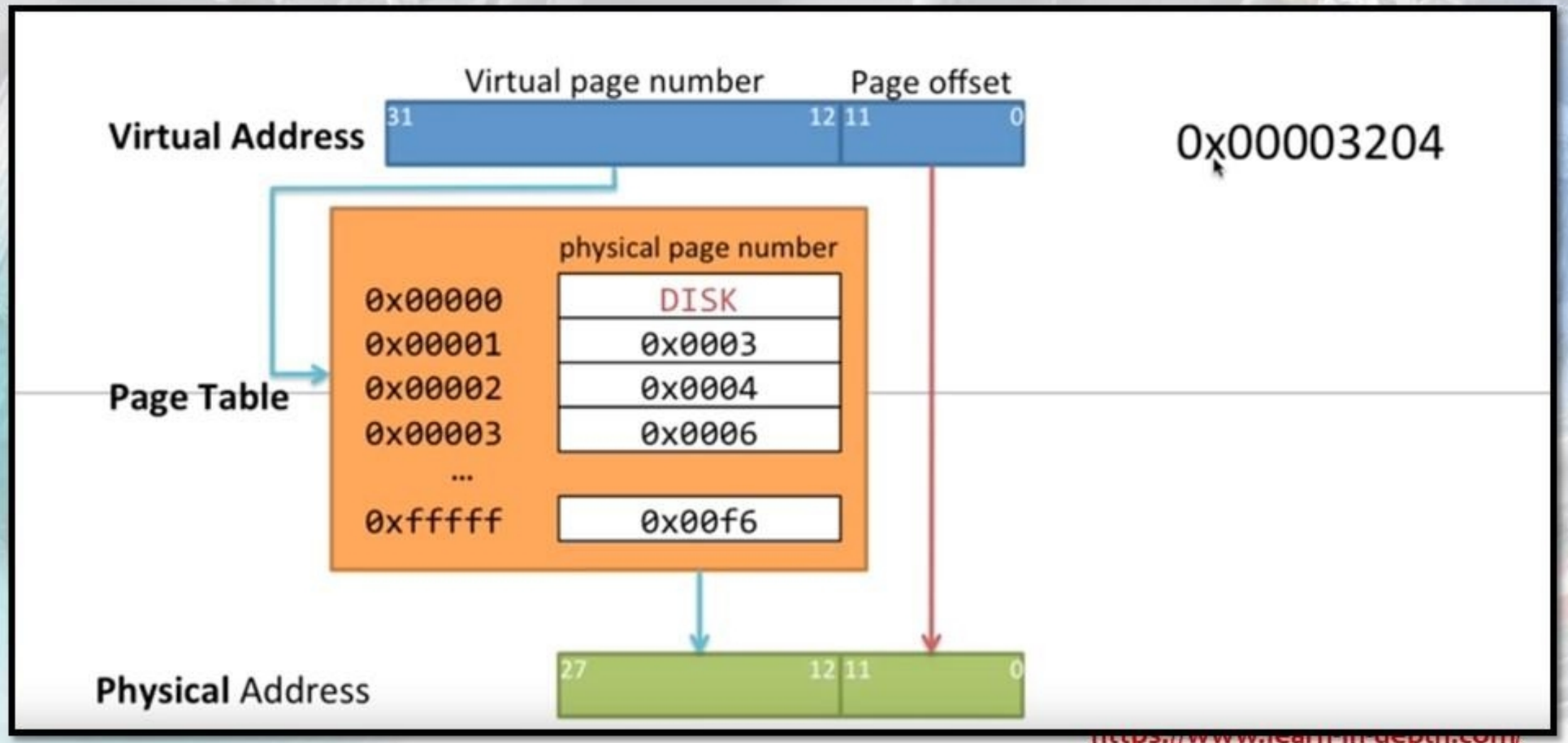


- Virtual page number translated to the physical page number
- Page offset untranslated
- Page Table keeps mappings for every virtual page → physical page

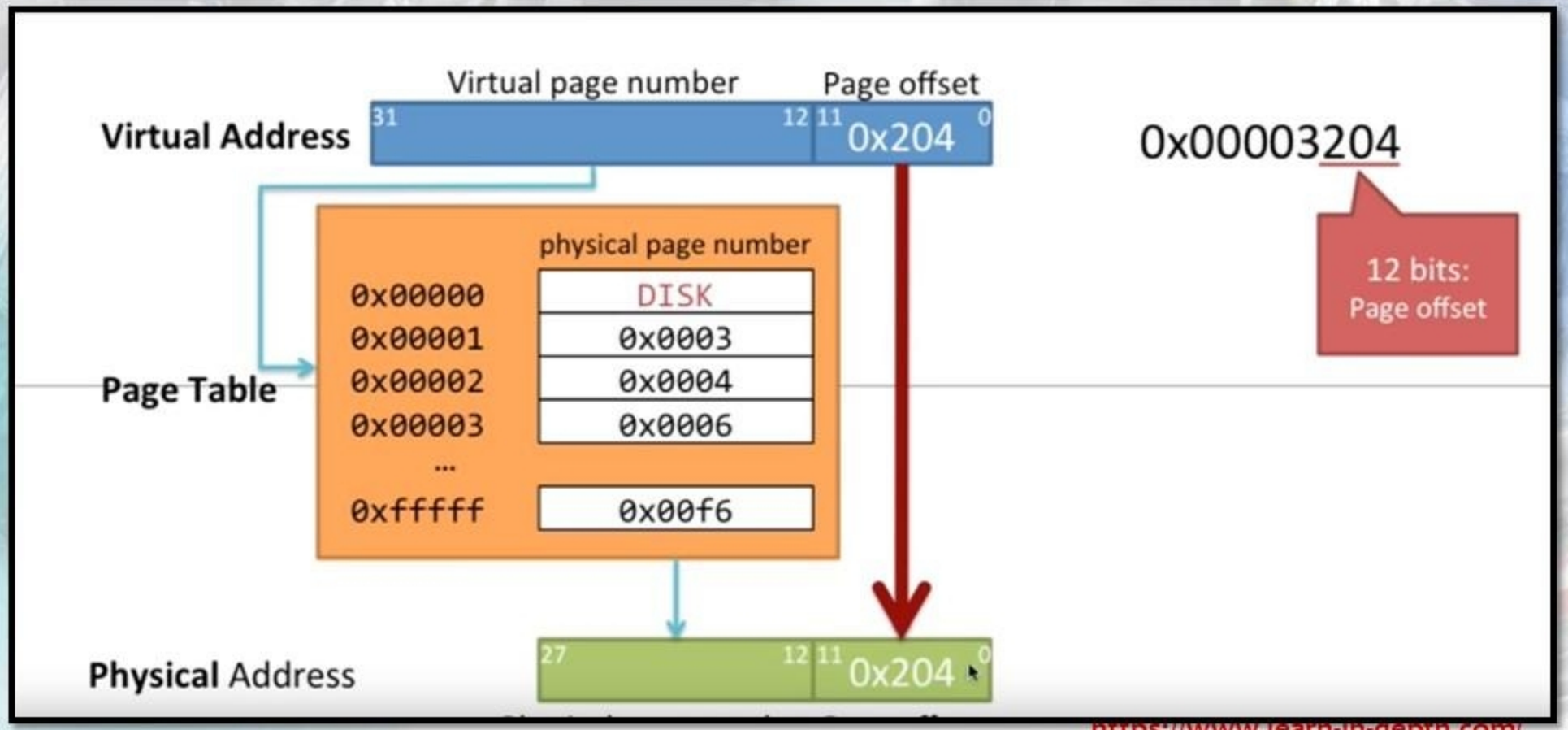
Example 1 Page block is located in memory

THINK ☺

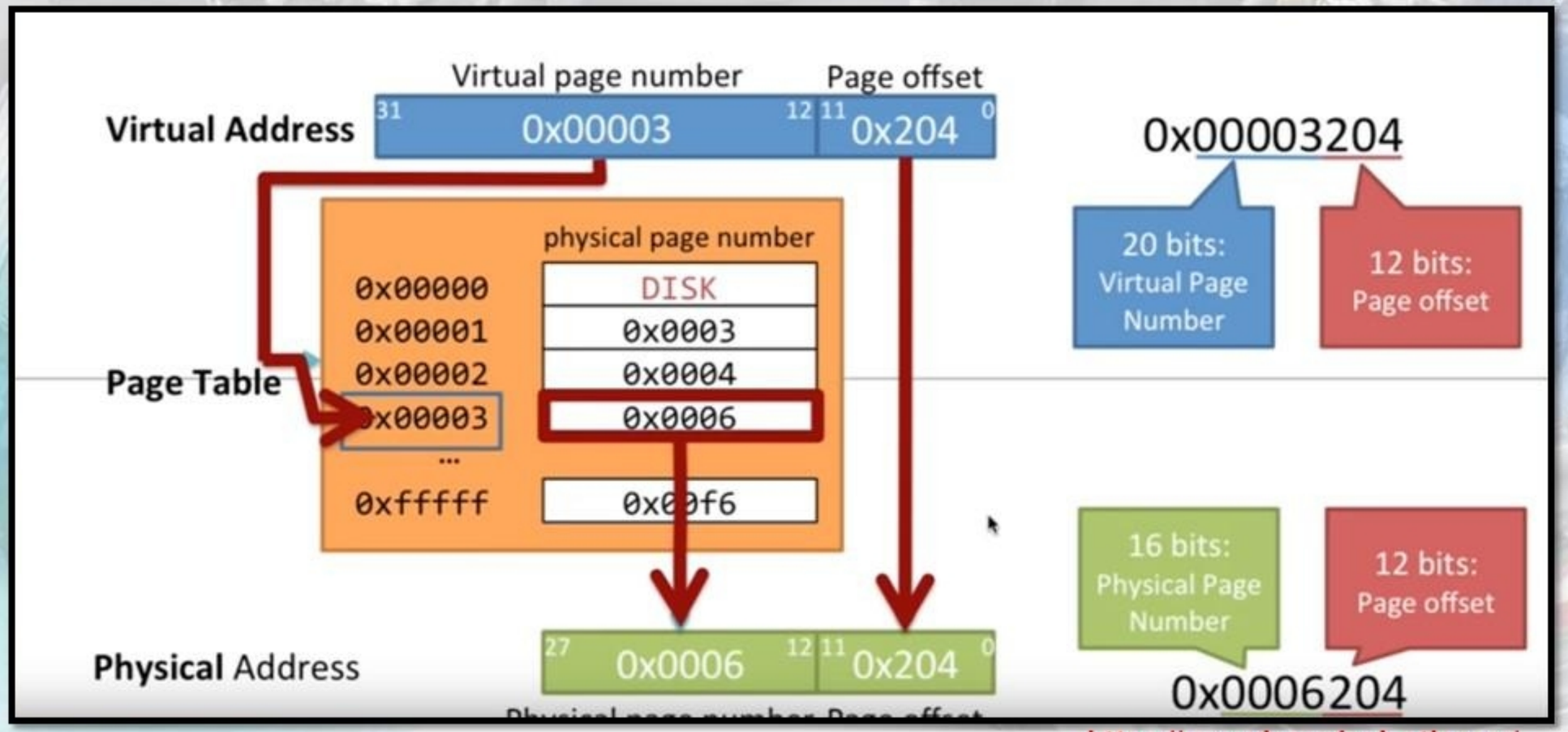
Example on Paging



Example on Paging Cont.



Example on Paging Cont.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

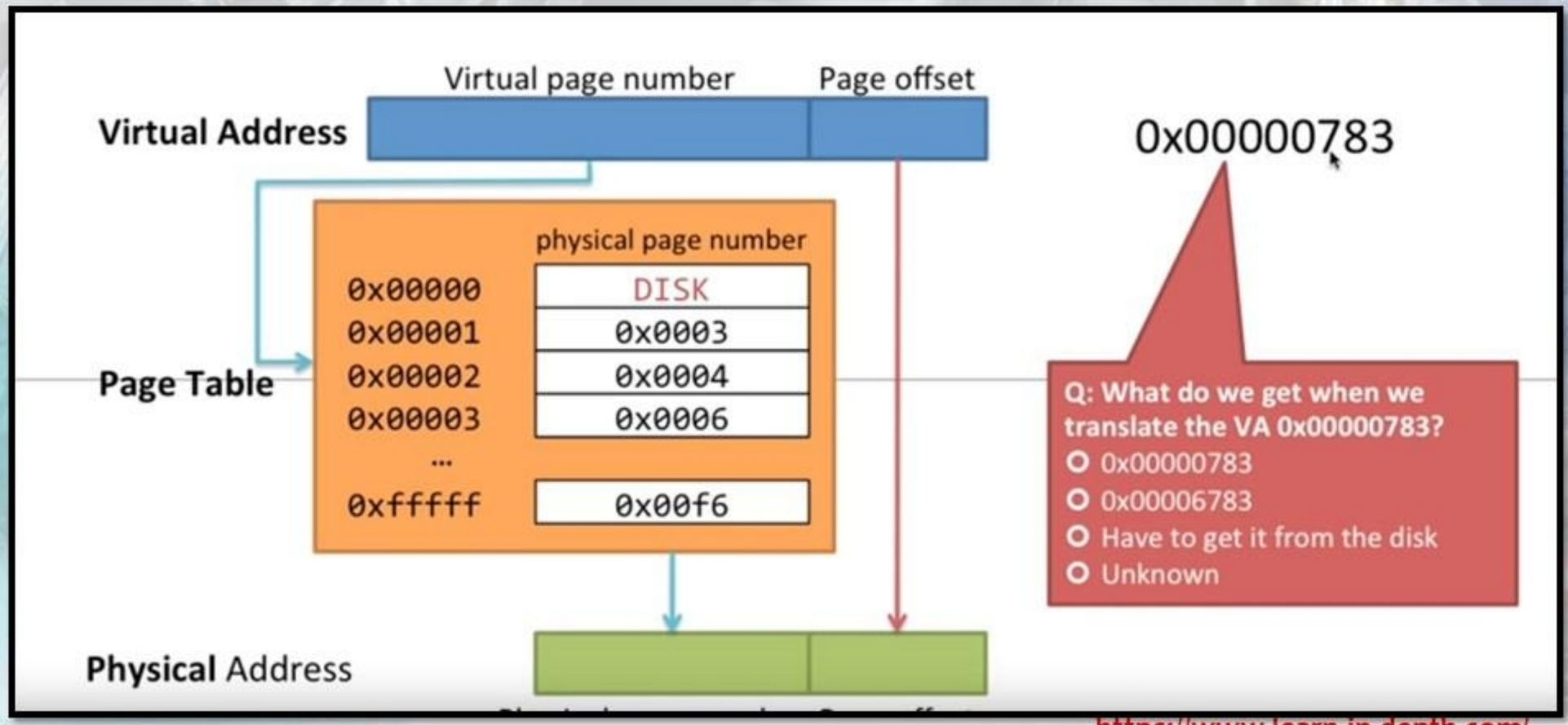
Learn In Depth

Example 2 Page block is located in DISK

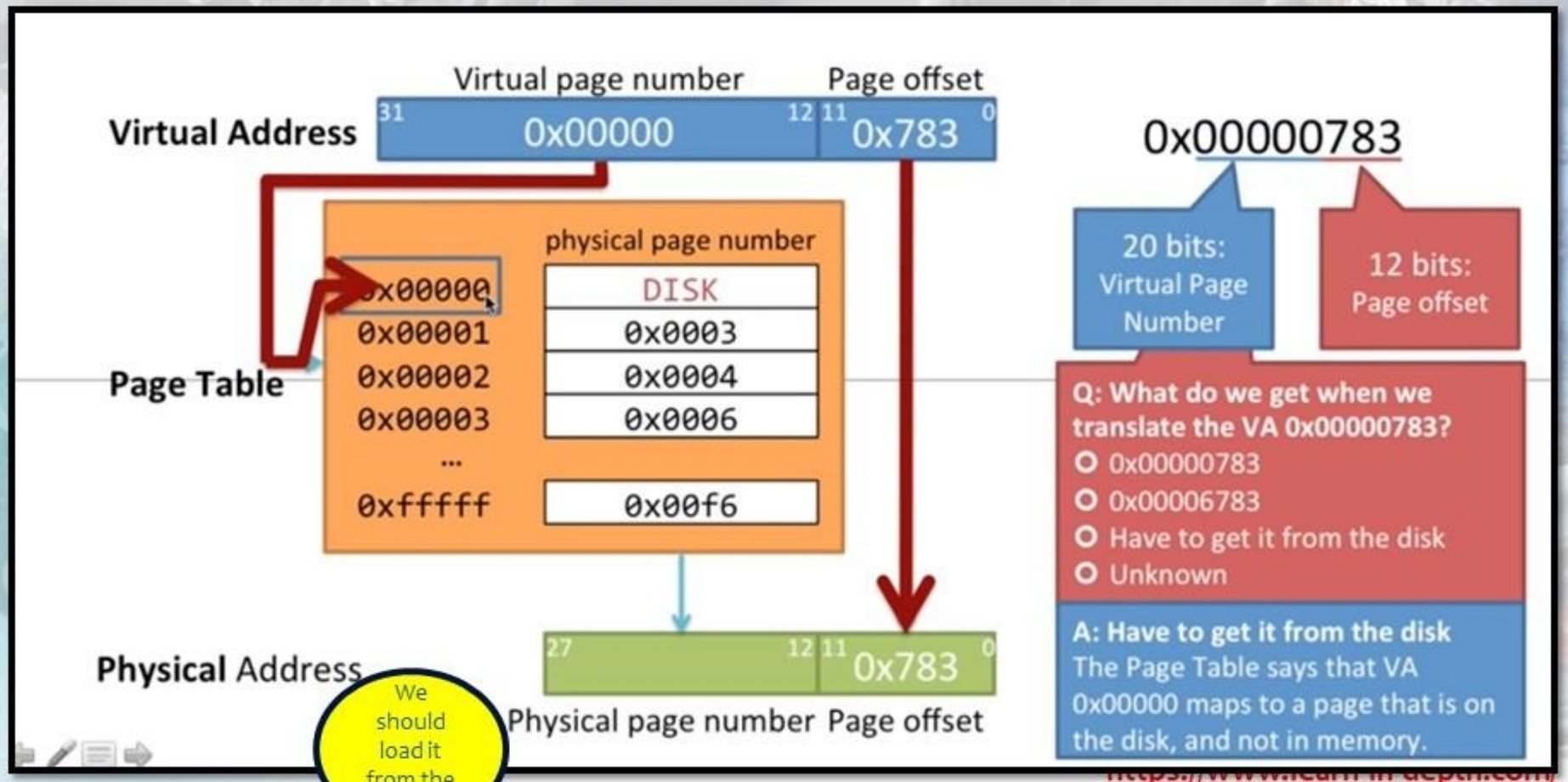
THINK ☺

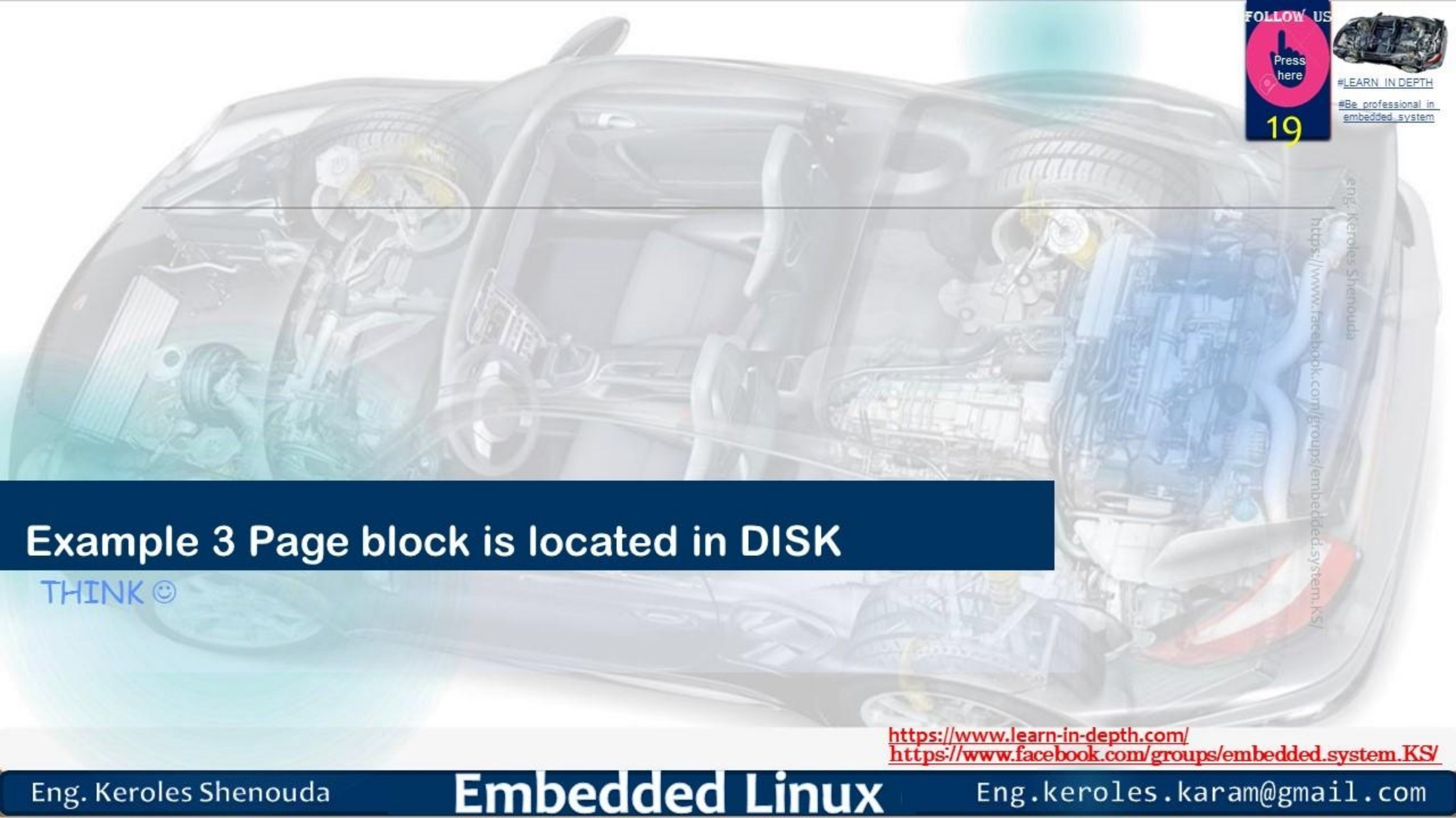
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Example on Paging



Example on Paging Cont.



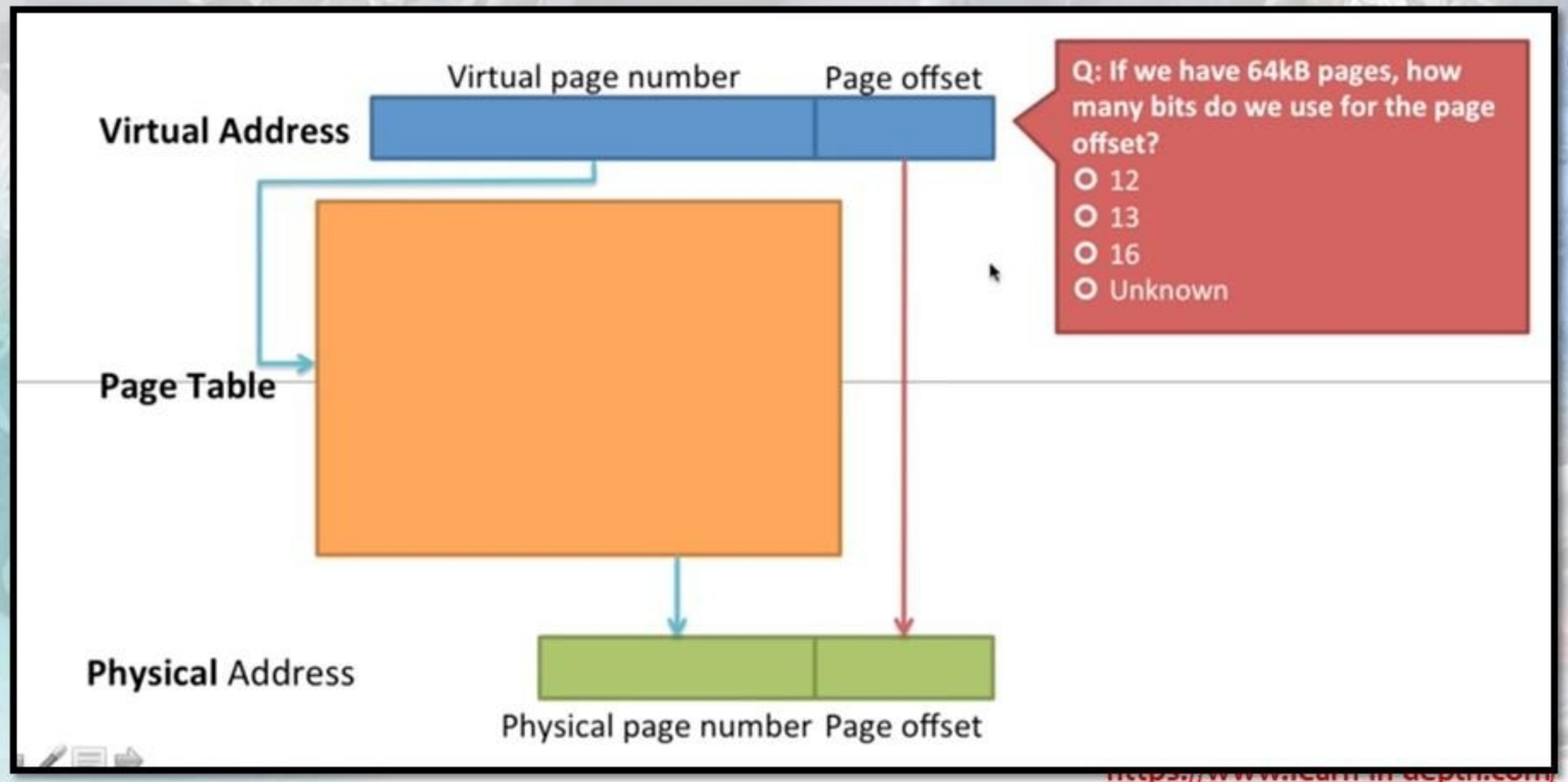


Example 3 Page block is located in DISK

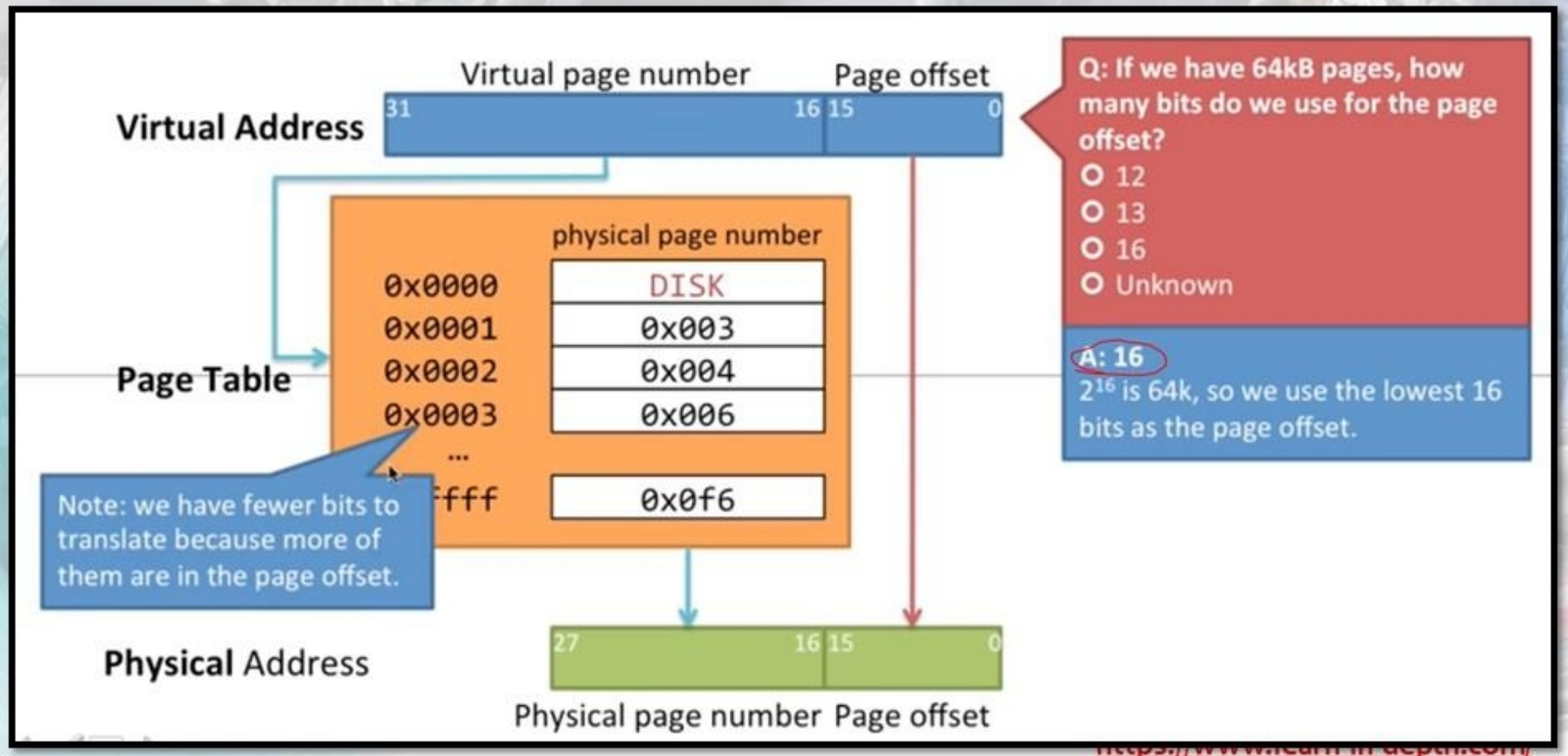
THINK ☺

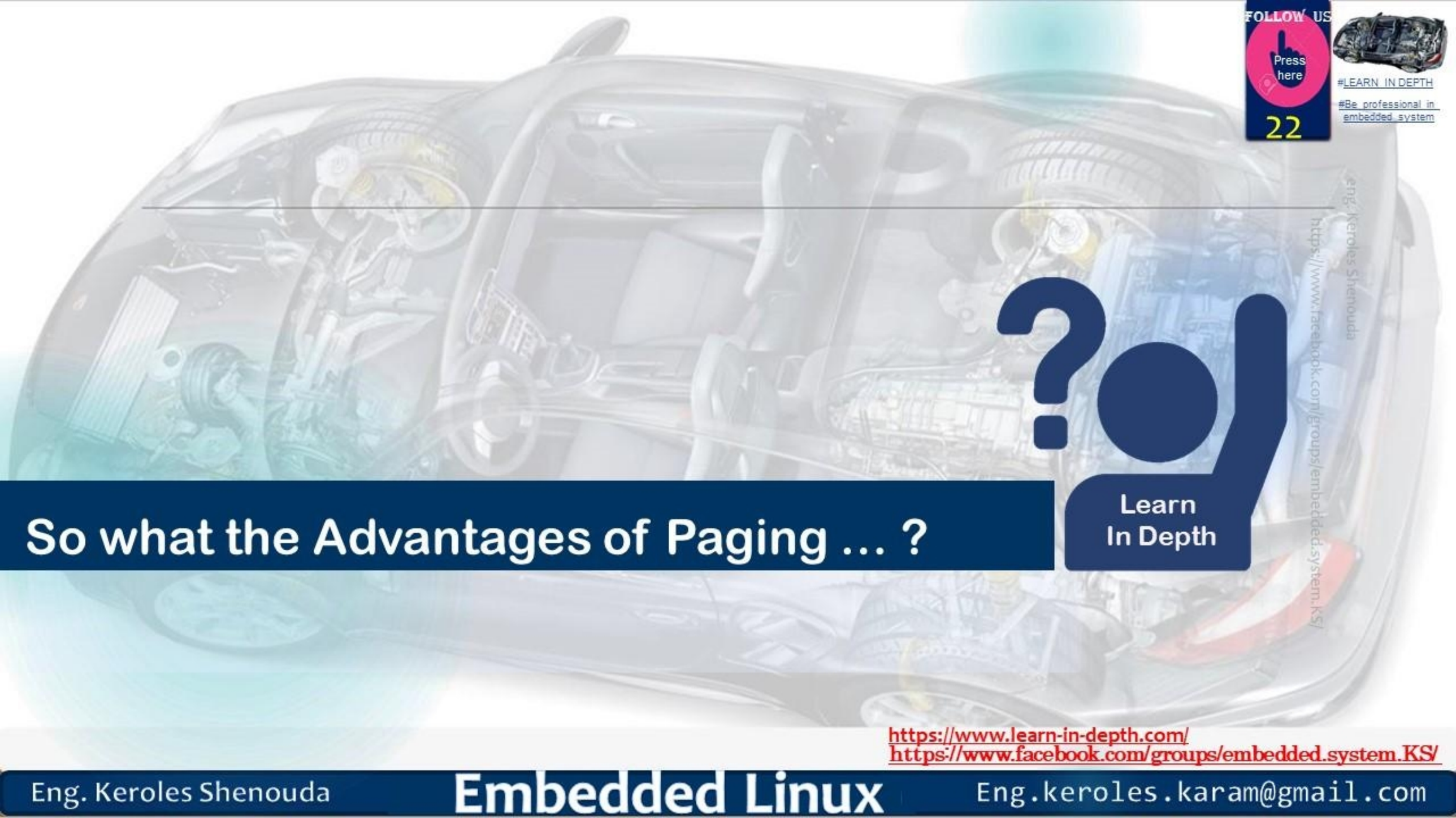
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Example on Paging



Example on Paging





So what the Advantages of Paging ... ?

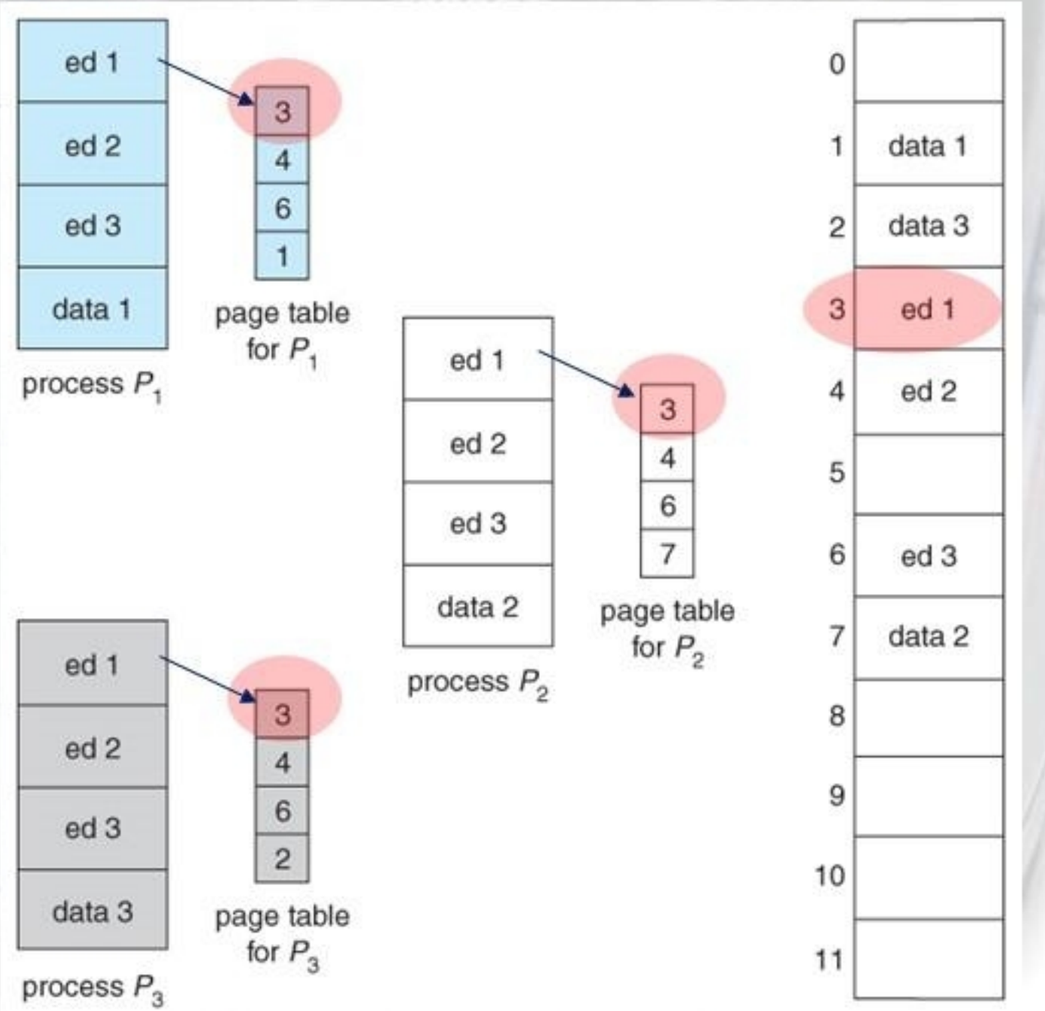
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

So the Advantages of Paging

- non-contiguous allocation
  - how do we associate page frames with processes?
    - Processes use virtual addresses
- Shared Pages

Shared Pages

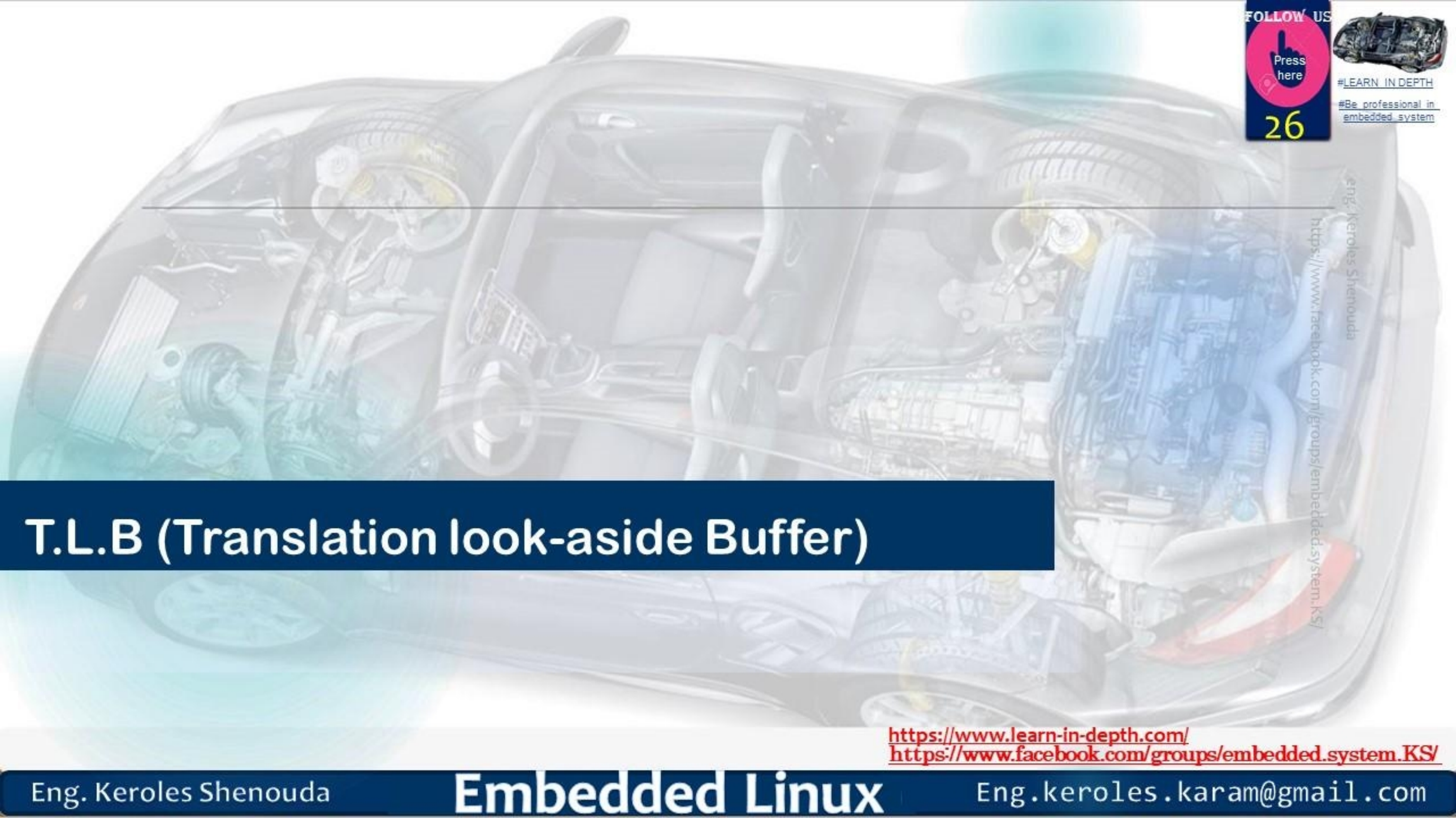
- **Paging systems** can make it very easy to **share blocks** of memory, **by simply duplicating page numbers in multiple page frames.**
  - This may be done with either code or data.
- If code is ***reentrant***, that means that it does not write to or change the code in any way (**it is non self-modifying**), and it is therefore safe to re-enter it.
  - More importantly, it **means the code can be shared by multiple processes**, so long as each has their own copy of the data and registers, including the instruction register.



Advantages and Disadvantages of paging

- Advantages:
  - Provides **memory space larger than physical memory**
    - it can run much larger program than size of physical memory.
    - For example in case of 32 bit address system it can run programs with size up to 4 gigabytes.
  - it does not need to load unnecessary pages because **page is loaded on demand.**
  - it does not care about **page placement** because all page frames have equal size.
  - It can easily **share** and **protect** memory among processes **via page table.**
- Disadvantages
  - there is **temporal overhead** for page table access.
    - this problem can be solved by the support of hardware called **T.L.B** (Translation look-aside Buffer)
  - page table can be relatively big.
    - this problem can be solved by **multi-level page structure.**

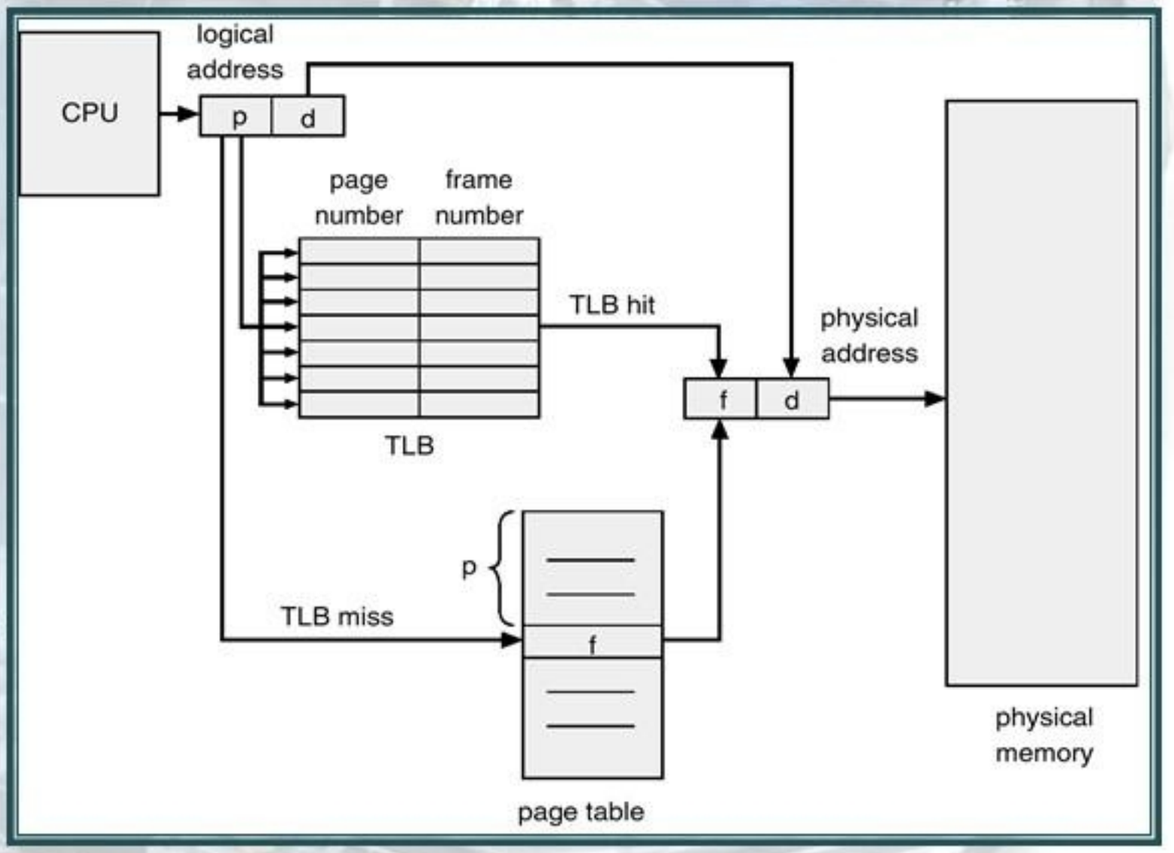
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



T.L.B (Translation look-aside Buffer)

Implementation of Page Table

- Page table **is kept** in **main memory**.
  - *Page-table base register* (PTBR) points to the page table.
  - *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every **data/instruction** access requires **two** memory accesses.
  - One for the page table and one for the data/instruction.
- ***Memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)***

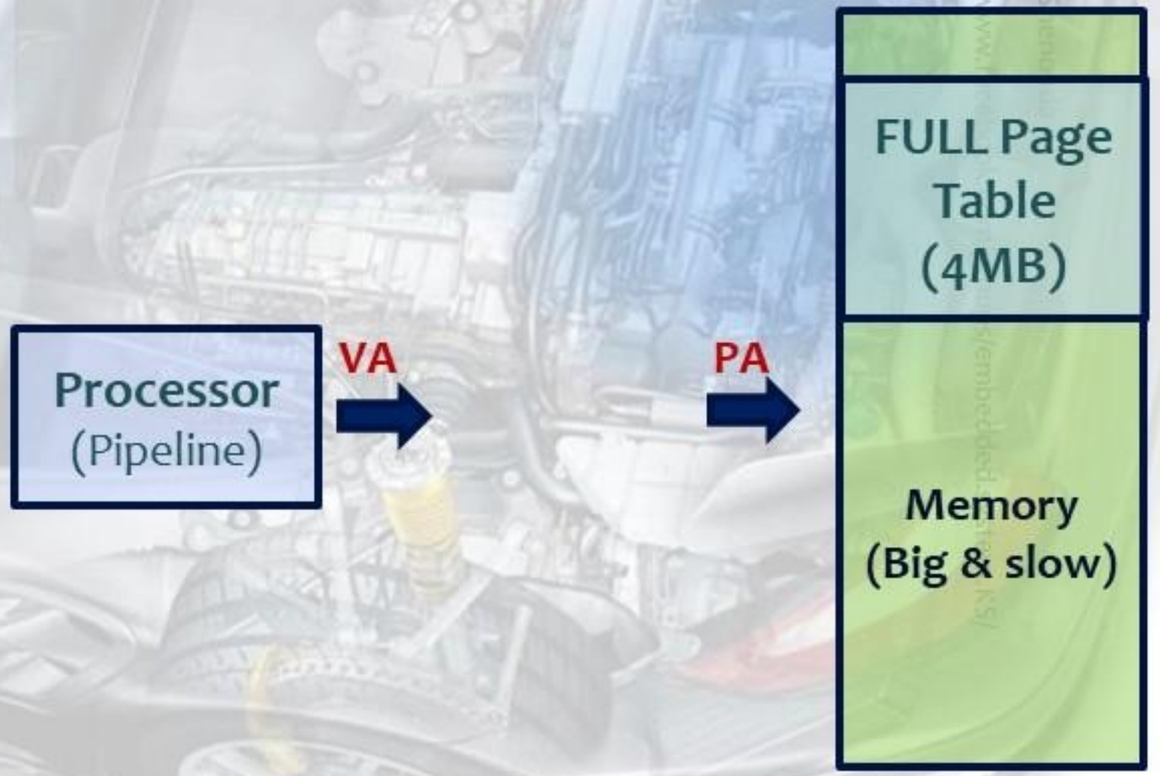


https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

translation look-aside buffers (TLBs)

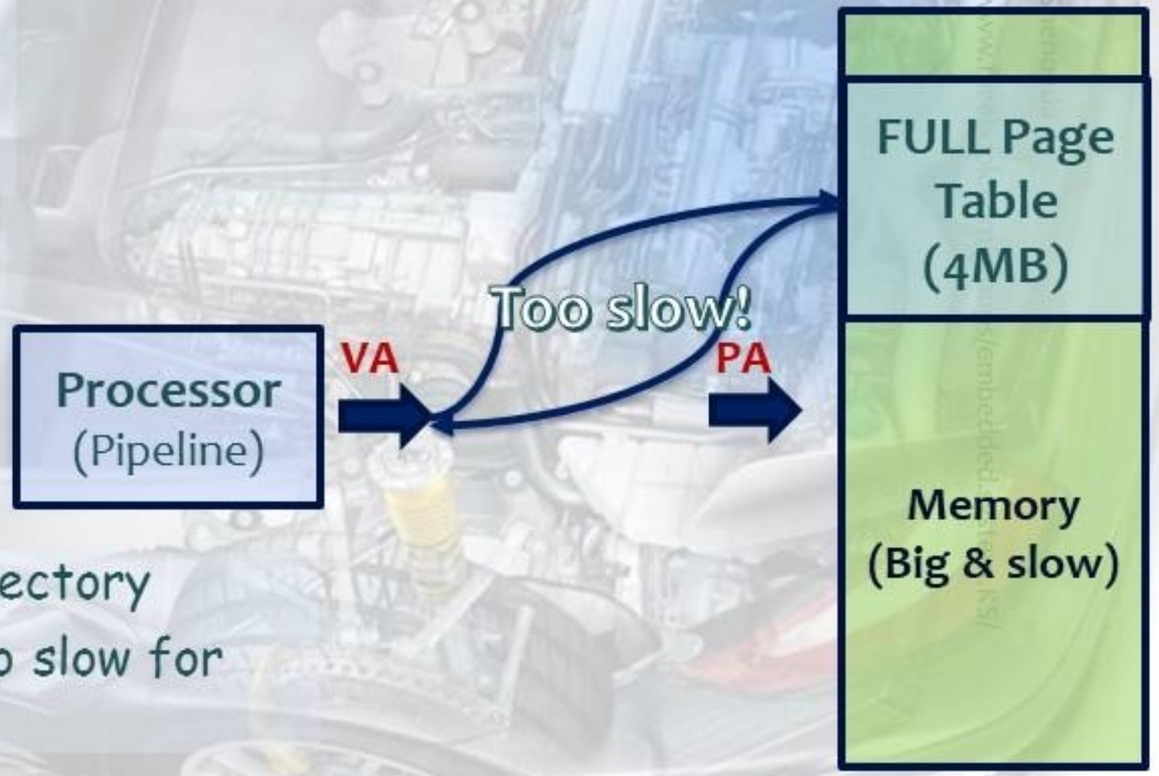
- To make virtual Memory VM **Fast** we add a special **Page Table Cache:**
  - The translation look-aside buffer (TLB)

- Processor is very fast
  - Issue transaction VA (virtual address)
- Memory is big and slow



translation look-aside buffers (TLBs)

- To make virtual Memory VM Fast we add a special **Page Table Cache:**
  - The translation look-aside buffer (TLB)
- Processor is very fast
  - Issue transaction VA (virtual address)
- Memory is big and slow

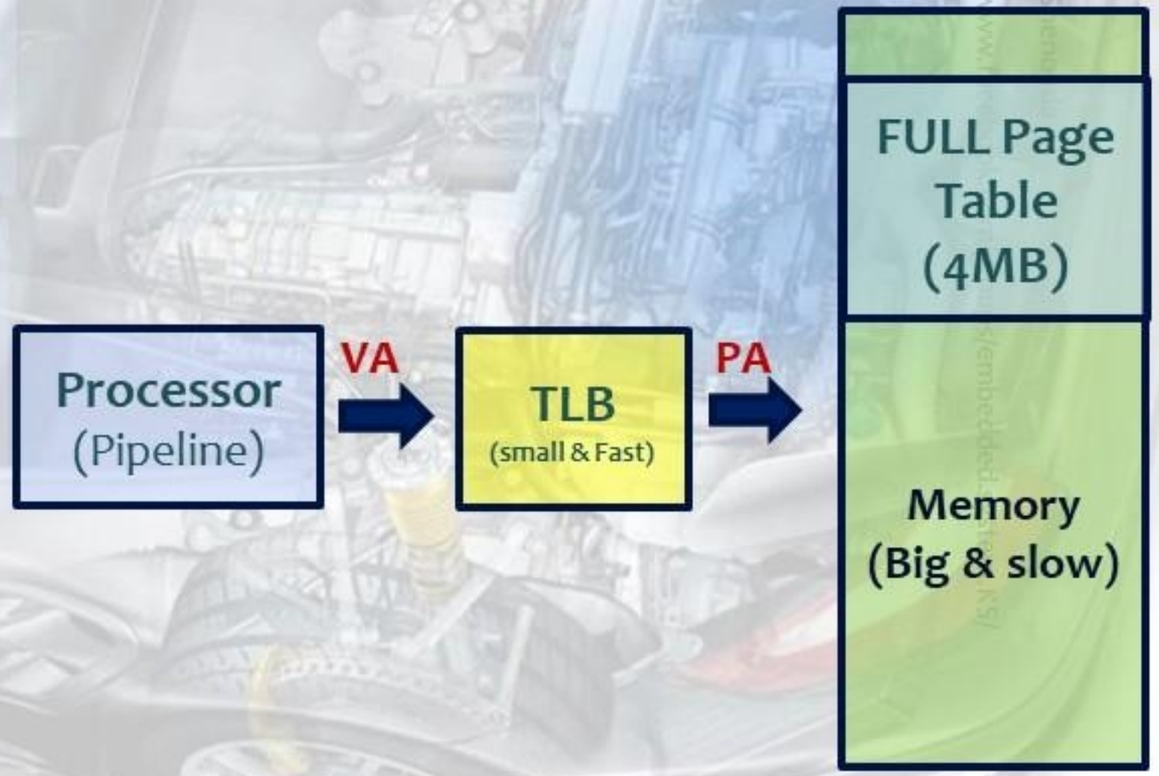


If the process access the page table directory

And get pack the translation It will be so slow for every single access

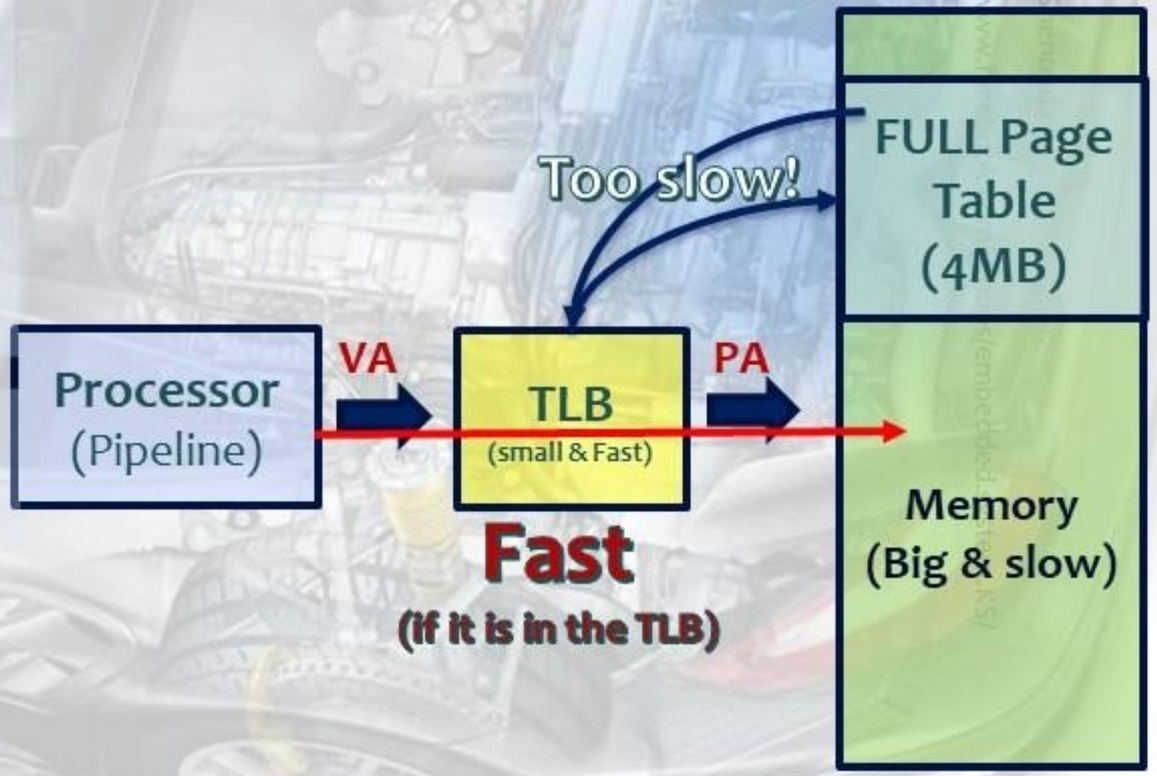
translation look-aside buffers (TLBs)

- To make virtual Memory VM Fast we add a special **Page Table Cache:**
  - The translation look-aside buffer (TLB)
- Fast: less than 1 Cycle (have to do it for every memory access)
- Very similar to a cache



translation look-aside buffers (TLBs)

- To make virtual Memory VM Fast we add a special **Page Table Cache:**
  - The translation look-aside buffer (TLB)
- Fast: less than 1 Cycle (have to do it for every memory access)
- Very similar to a cache
- TO Be Fast, TLBs must be small:
  - Separate TLBs for instructions (iTLB) and data (dTLB).



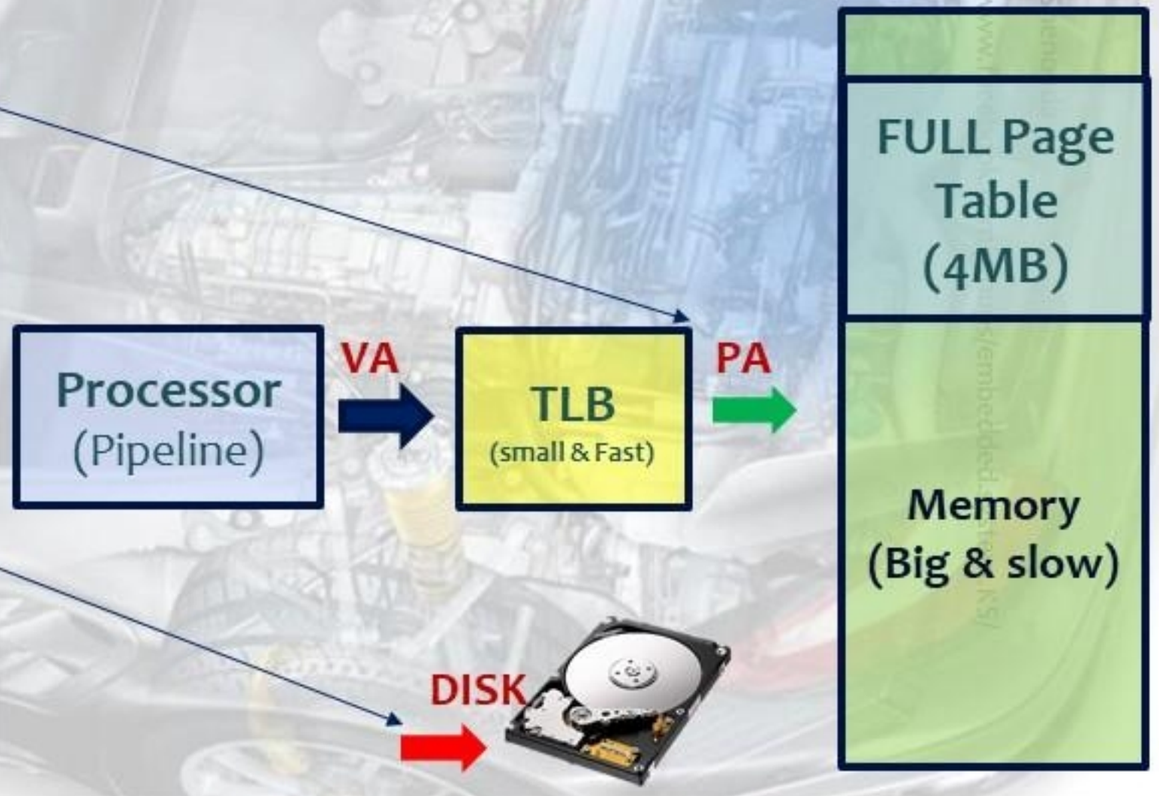
What can happen when we access memory (TLB exist) ?

Learn In Depth

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

What can happen when we access memory (TLB exist) ?

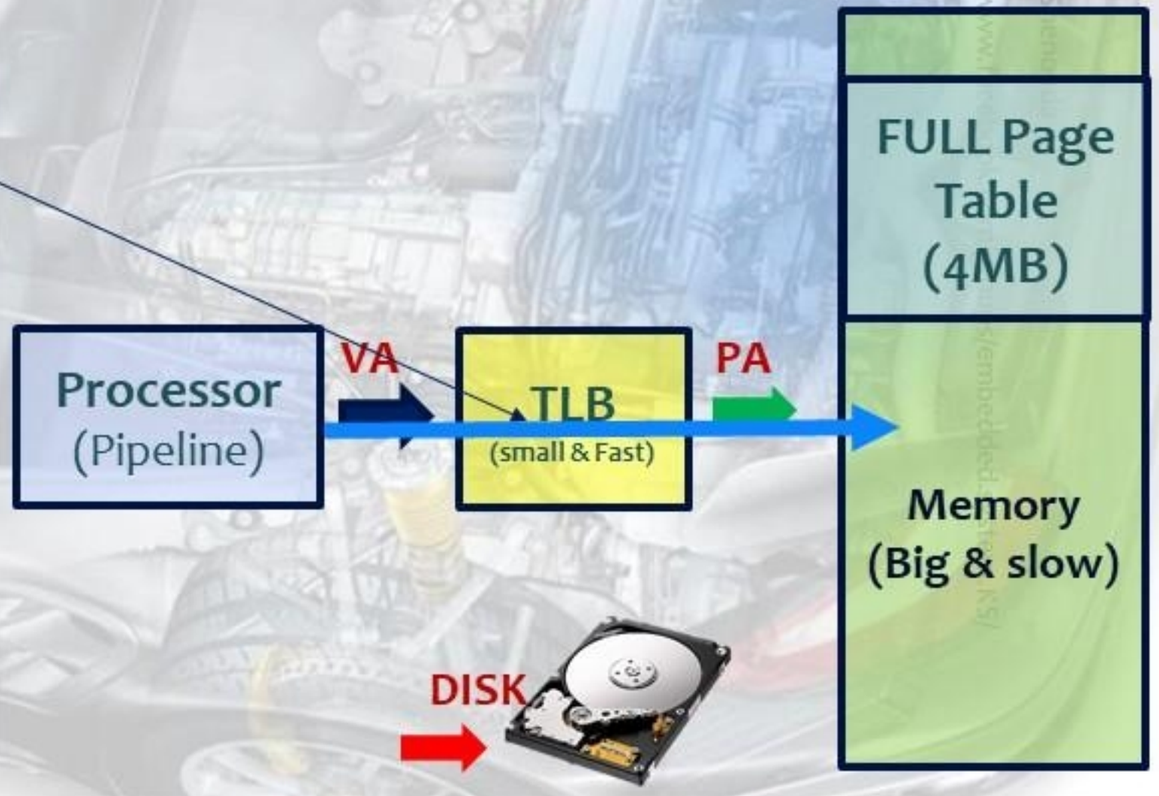
- Good scenario: Page in RAM
- Bad scenario: Page not in RAM
  - In DISK (that will be slowly)



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

What can happen when we access memory (TLB exist) ?

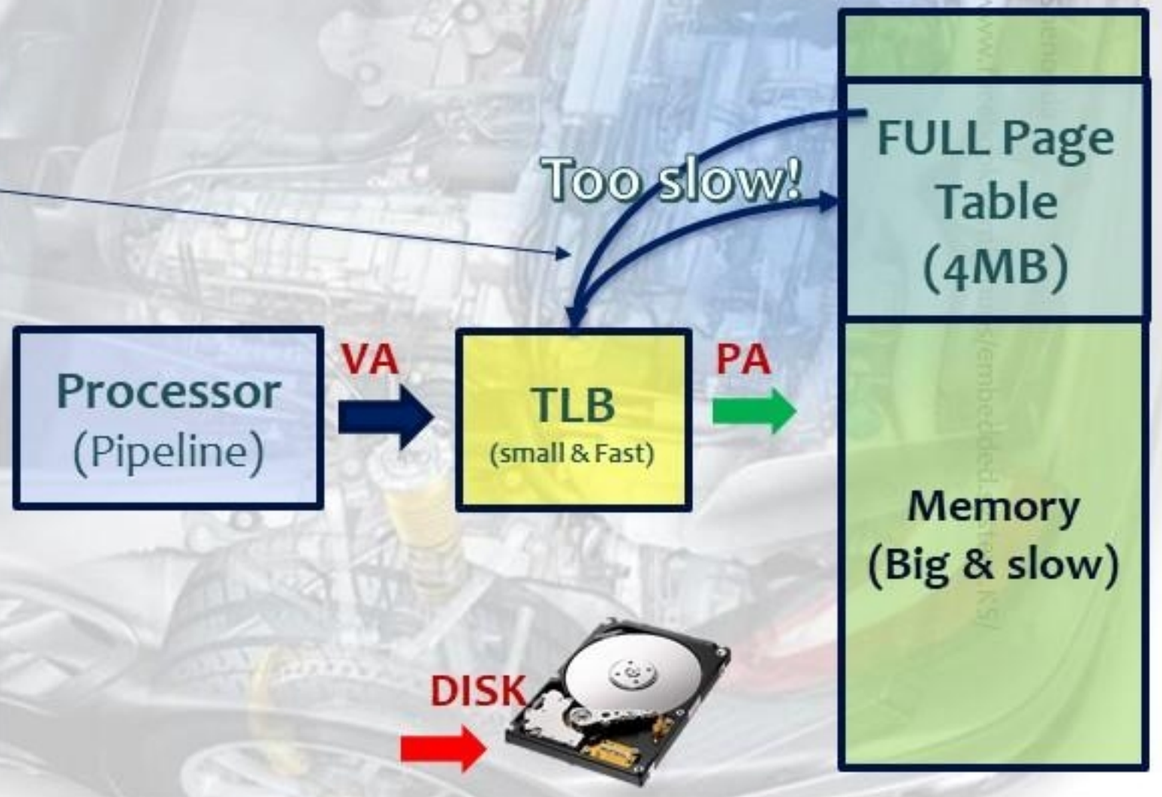
- **Good scenario: Page in RAM**
  - Very best scenario ☺ PTE "page table entry" in the TLB in this case we can access very quickly
    - <1 cycle> to translate, then go to ram or cache



- **Bad scenario: Page not in RAM**
  - In DISK (it will time to be accessed)

What can happen when we access memory (TLB exist) ?

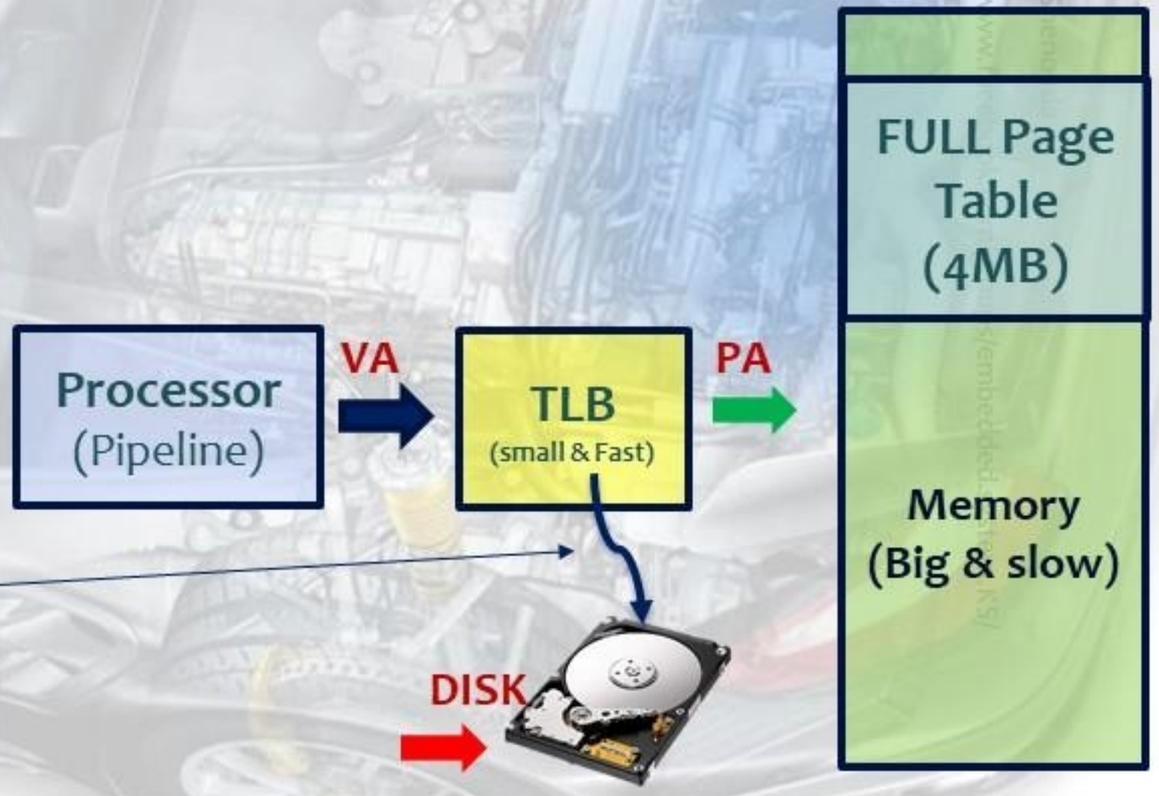
- **Good scenario: Page in RAM**
  - Very best scenario ☺ PTE "page table entry" in the TLB in this case we can access very quickly
    - <1 cycle> to translate, then go to ram or cache
  - **Not enough good scenario PTE not in TLB**
    - Poor performance, 20-1000 cycles to load PTE from RAM, then go to RAM.



- **Bad scenario: Page not in RAM**
  - In DISK (it will time to be accessed)

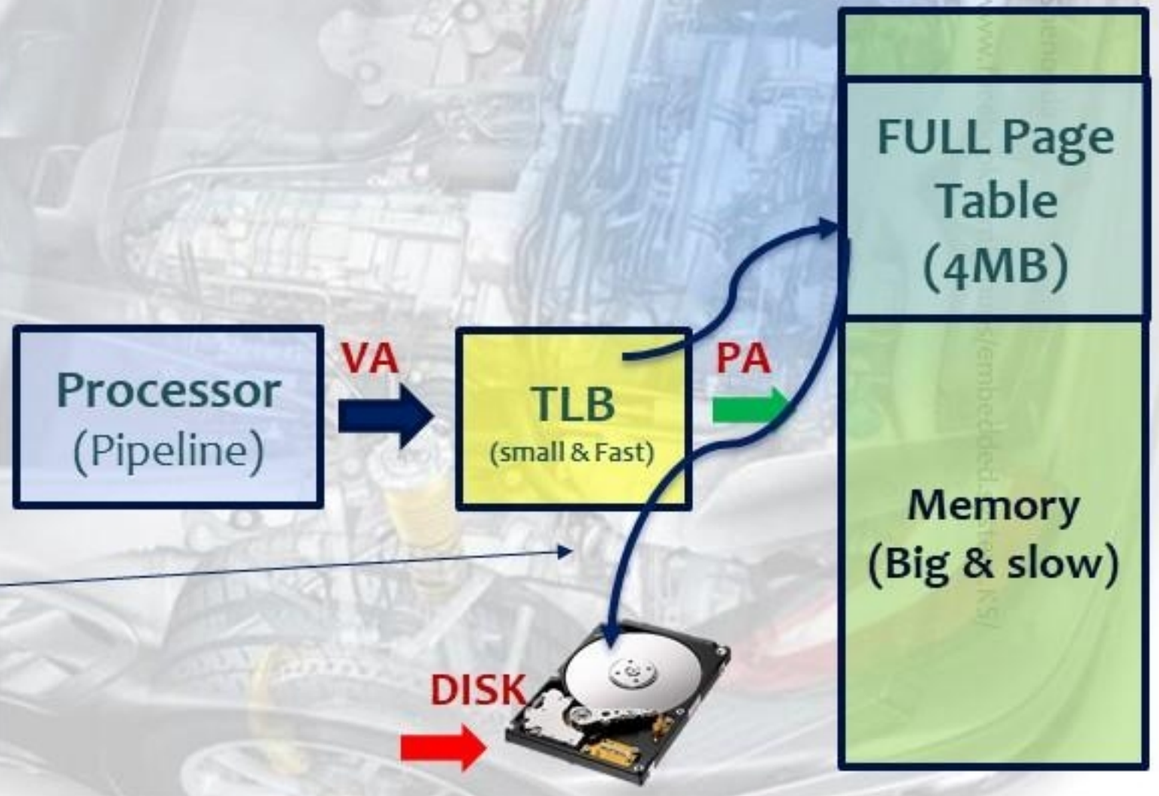
What can happen when we access memory (TLB exist) ?

- **Good scenario: Page in RAM**
  - Very best scenario ☺ PTE "page table entry" in the TLB in this case we can access very quickly
    - <1 cycle> to translate, then go to ram or cache
  - **Not enough good scenario** PTE not in TLB
    - Poor performance, 20-1000 cycles to load PTE from RAM, then go to RAM.
- **Bad scenario: Page not in RAM**
  - In DISK (it will time to be accessed)
  - PTE in the TLB
    - 1 cycle to know it's on disk
    - ~80M Cyles to get it from disk



What can happen when we access memory (TLB exist) ?

- Good scenario: Page in RAM
  - Very best scenario ☺ PTE "page table entry" in the TLB in this case we can access very quickly
    - <1 cycle> to translate, then go to ram or cache
  - Not enough good scenario PTE not in TLB
    - Poor performance, 20-1000 cycles to load PTE from RAM, then go to RAM.
- Bad scenario: Page not in RAM
  - In DISK (it will time to be accessed)
  - PTE in the TLB
    - 1 cycle to know it's on disk
    - ~80M Cyles to get it from disk
  - PTE not in the TLB
    - 20-1000 cycle to know it's on disk
    - ~80M Cyles to get it from disk



Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)

In your Opinion ☺

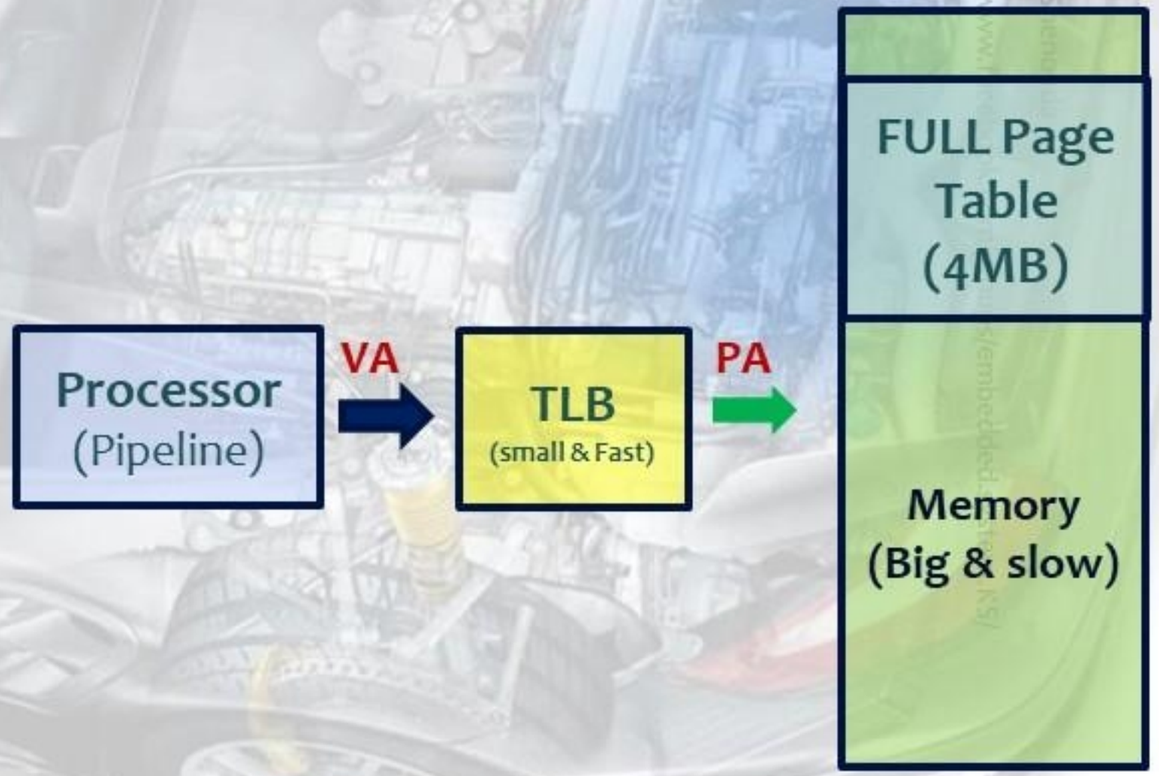
Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)

A:

1. Make pages larger

This increases the reach of the TLB because you need fewer pages to cover more data.



Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)

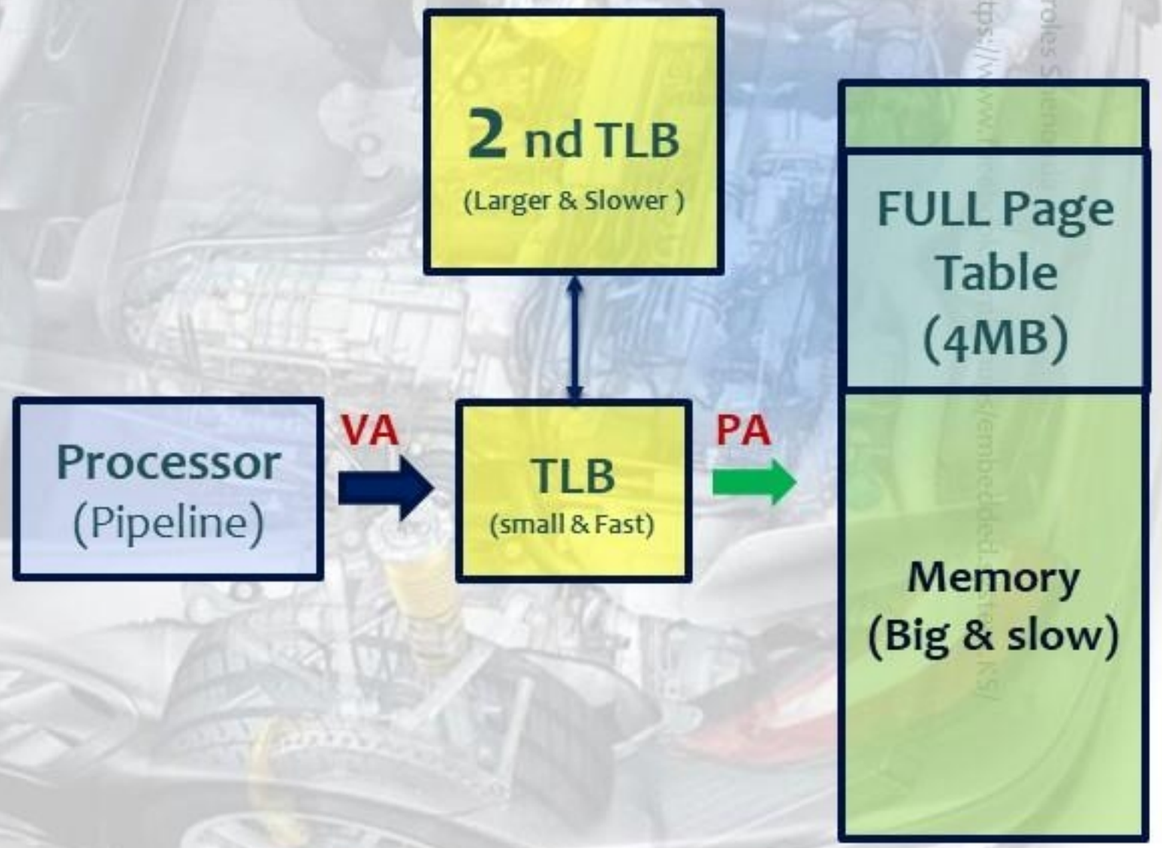
A:

1. Make pages larger

This increases the reach of the TLB because you need fewer pages to cover more data.

2. Add a second TLB that is larger, but a bit slower

Sure. Most processors have a level 2 TLB that is about 8x larger than the level 1 TLB, but also about twice as slow.



Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)

A:

1. Make pages larger

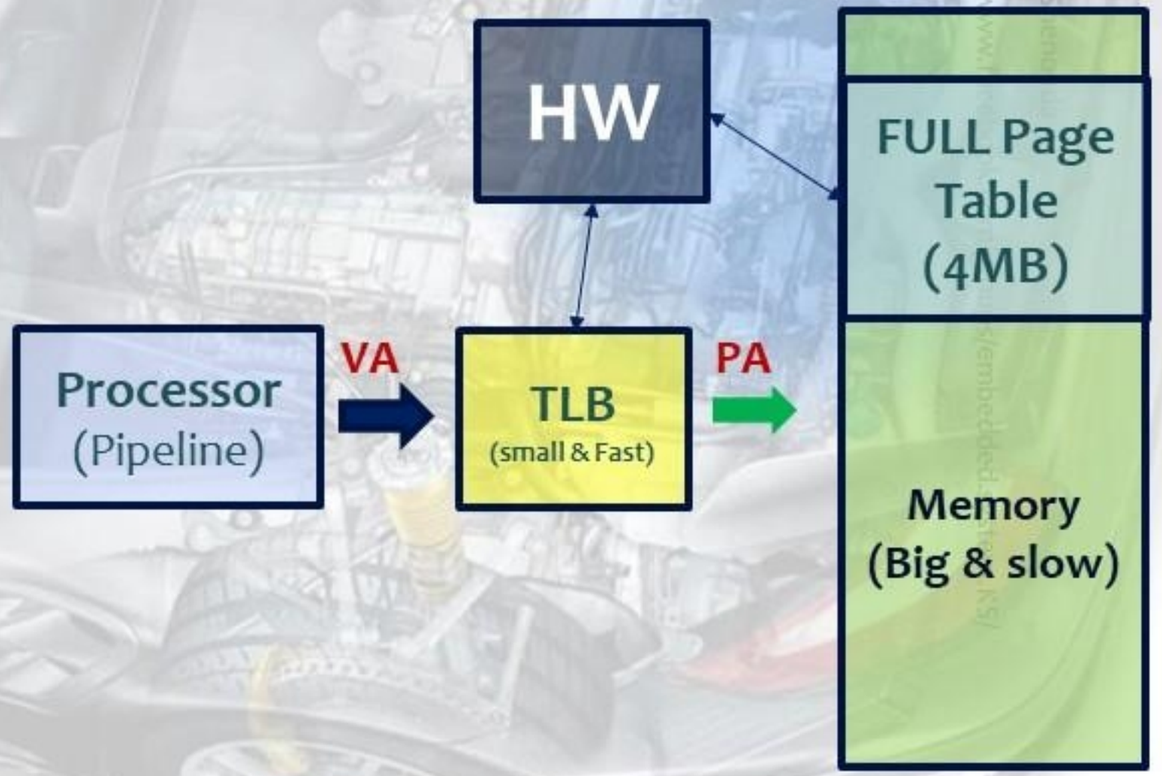
This increases the reach of the TLB because you need fewer pages to cover more data.

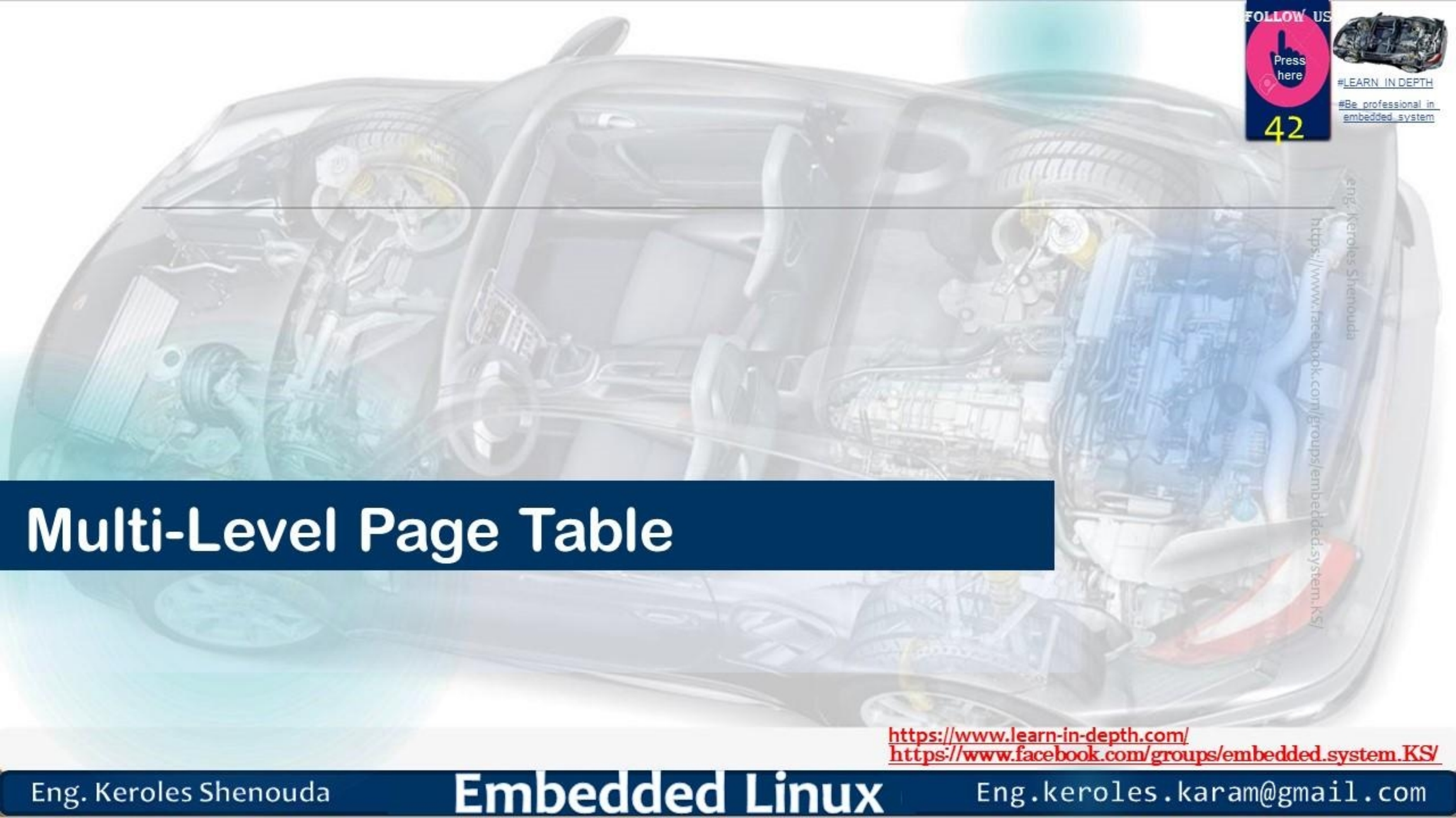
2. Add a second TLB that is larger, but a bit slower

Sure. Most processors have a level 2 TLB that is about 8x larger than the level 1 TLB, but also about twice as slow.

3. Have hardware to fill the TLB automatically

This is called a "hardware page table walk". Basically the hardware assumes the page table is in a special form in memory, and it can go get data from it on a TLB miss without having to go to the OS.



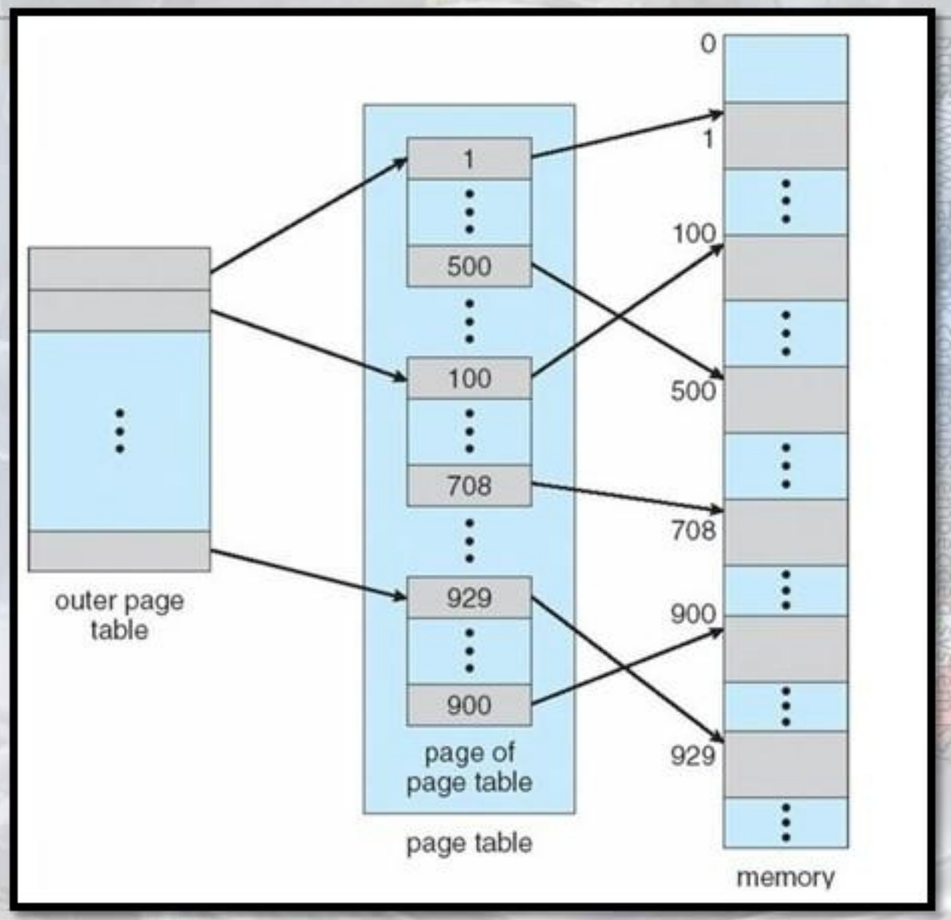


Multi-Level Page Table

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

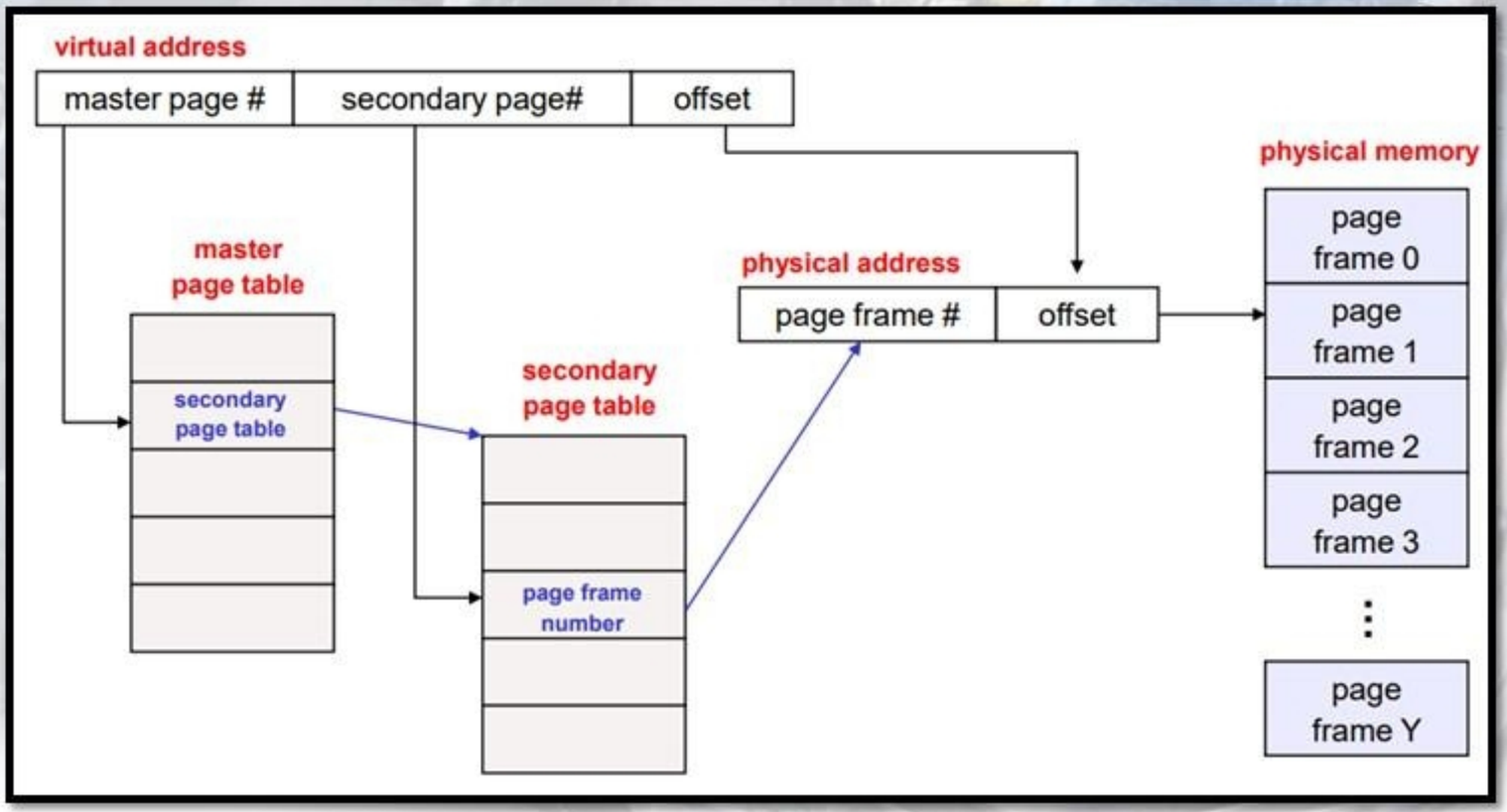
Multi-Level Page Table

- Additional memory space is required to store page table.
- **Program has a large address space**
  - If we use 32 bit address
    - Can address a program with 4GB size
- Example if we have two-level page table
  - The page table itself is also paged
    - Stores an entries page table in disks.
    - Load pages table's page on on-demand



Multi-Level Page Table

- Address translation with two-level page table



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Lab 1 Paging

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Lab1

- $gedit lab1.c &
- Compile it
  - $ gcc lab1.c -o lab1.elf
- Run it
  - ./lab1.elf
  - To terminate it press Ctrl+C

```
embedded_system_ks@embedded-KS: ~/labs/part2
embedded_system_ks@embedded-KS:~$ mkdir -p labs/part2
embedded_system_ks@embedded-KS:~$ cd labs/part2
embedded_system_ks@embedded-KS:~/labs/part2$ gedit lab1.c &
[1] 7952
embedded_system_ks@embedded-KS:~/labs/part2$ gcc lab1.c -o lab1.elf
embedded_system_ks@embedded-KS:~/labs/part2$ ./lab1.elf
[0] Learn In Depth Eng.Keroles Shenouda
[1] Learn In Depth Eng.Keroles Shenouda
[2] Learn In Depth Eng.Keroles Shenouda
[3] Learn In Depth Eng.Keroles Shenouda
[4] Learn In Depth Eng.Keroles Shenouda
[5] Learn In Depth Eng.Keroles Shenouda
```

Embedded Linux

ENG.KEROLES SHENOUDA

lab1.c (Embedded_Linux /media/embedded_system_ks/Embedded_Linux/images_preb

Open

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
/**
@Copyright www.learn-in-depth.com
Author: Keroles Shenouda
Learn In Depth
https://www.facebook.com/groups/embedded.system.KS
 */
int main(void)
{
    char *s;
    unsigned long int i;
s = (char*)malloc(100* sizeof(char));
s= "Learn In Depth Eng.Keroles Shenouda";
    if (s == NULL)
    {
        fprintf(stderr, "Can't allocate mem with malloc\n");
        return (EXIT_FAILURE);
    }
    i = 0;
    while (s)
    {
        printf("[%lu] %s\n", i, s);
        sleep(1);
        i++;
    }
    return (EXIT_SUCCESS);
}
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Lab1:check the virtual address

```
embedded_system_ks@embedded-KS: ~/labs/part2
embedded_system_ks@embedded-KS:~/labs/part2$ objdump -D lab1.elf > lab1.s
embedded_system_ks@embedded-KS:~/labs/part2$ gedit lab1.s &
[2] 8060
embedded_system_ks@embedded-KS:~/labs/part2$
```


lab1.s (~/labs/part2) - gedit

Open Save

```
394  80484b7:        74 f2                   je     80484ab <frame_dummy+0xb>
395  80484b9:        55                      push   %ebp
396  80484ba:        89 e5                   mov    %esp,%ebp
397  80484bc:        83 ec 14                sub    $0x14,%esp
398  80484bf:        50                      push   %eax
399  80484c0:        ff d2                   call   *%edx
400  80484c2:        83 c4 10                add    $0x10,%esp
401  80484c5:        c9                      leave
402  80484c6:        e9 75 ff ff ff          jmp    8048440 <register_tm_clones>
403
404 080484cb <main>:
405  80484cb:        8d 4c 24 04             lea    0x4(%esp),%ecx
406  80484cf:        83 e4 f0                and    $0xfffffff0,%esp
407  80484d2:        ff 71 fc                pushl  -0x4(%ecx)
408  80484d5:        55                      push   %ebp
409  80484d6:        89 e5                   mov    %esp,%ebp
410  80484d8:        51                      push   %ecx
411  80484d9:        83 ec 14                sub    $0x14,%esp
412  80484dc:        83 ec 0c                sub    $0xc,%esp
413  80484df:        6a 64                   push   $0x64
414  80484e1:        e8 ba fe ff ff          call   80483a0 <malloc@plt>
415  80484e6:        83 c4 10                add    $0x10,%esp
416  80484e9:        89 45 f4                mov    %eax,-0xc(%ebp)
417  80484ec:        c7 45 f4 e0 85 04 08    movl   $0x80485e0,-0xc(%ebp)
418  80484f3:        83 7d f4 00             cmpl   $0x0,-0xc(%ebp)
419  80484f7:        75 1e                   jne    8048517 <main+0x4c>
420  80484f9:        a1 28 a0 04 08          mov    0x804a028,%eax
421  80484fe:        50                      push   %eax
422  80484ff:        6a 1f                   push   $0x1f
423  8048501:        6a 01                   push   $0x1
424  8048503:        68 04 86 04 08          push   $0x8048604
425  8048508:        e8 83 fe ff ff          call   8048390 <fwrite@plt>
426  804850d:        83 c4 10                add    $0x10,%esp
```

Plain Text Tab Width: 9 Ln 404, Col 17 INS

All of those virtual address

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Check the memory mapes

- the Linux kernel maintains the processes page tables.
- Processes are only seeing virtual memory thru their address space. Processes use some syscalls, like e.g. mmap(2) or execve(2), to change their address space.
- /proc/<PID>/maps does not show anything about physical addresses, only about virtual one
- You have to get the process ID

We can also read the heap from /proc/<PID>/mem

- Step1 run the program
  - ./lab1.elf
- Get the pid address
  - ps -a
- Get the start end heap address
  - cat /proc/<pid>/maps
- Invoke the gdb
  - sudo gdb --pid <pid>
    - Then dump the memory
      - dump memory ~/labs/part2/heap_file 0x09262000 0x0
- Open the file
  - vi ~/labs/part2/heap_file

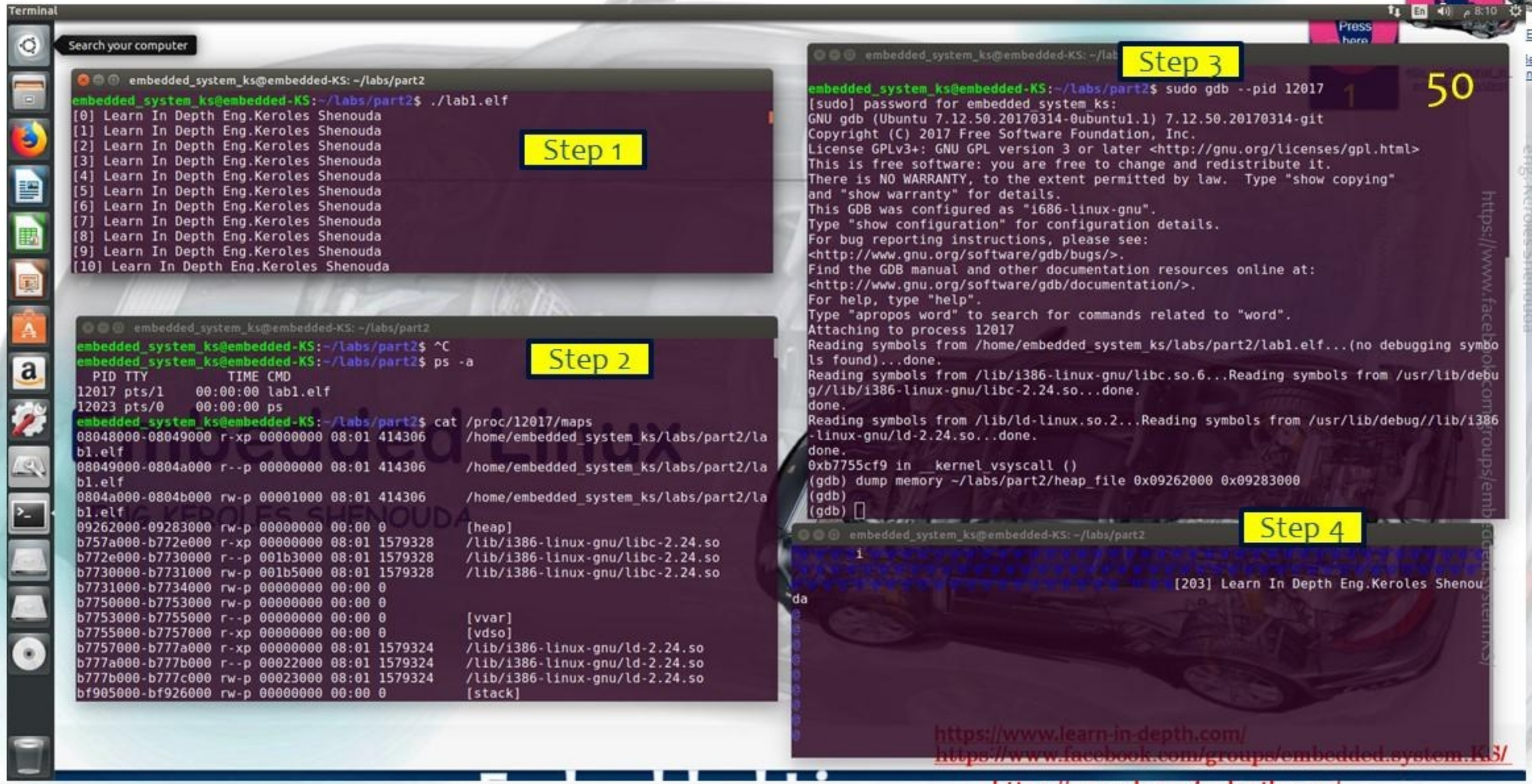
Memory Allocation (HEAP)

lab1.c (Embedded_Linux /media/embedded_system_ks/Embedded_Linux/images_preb

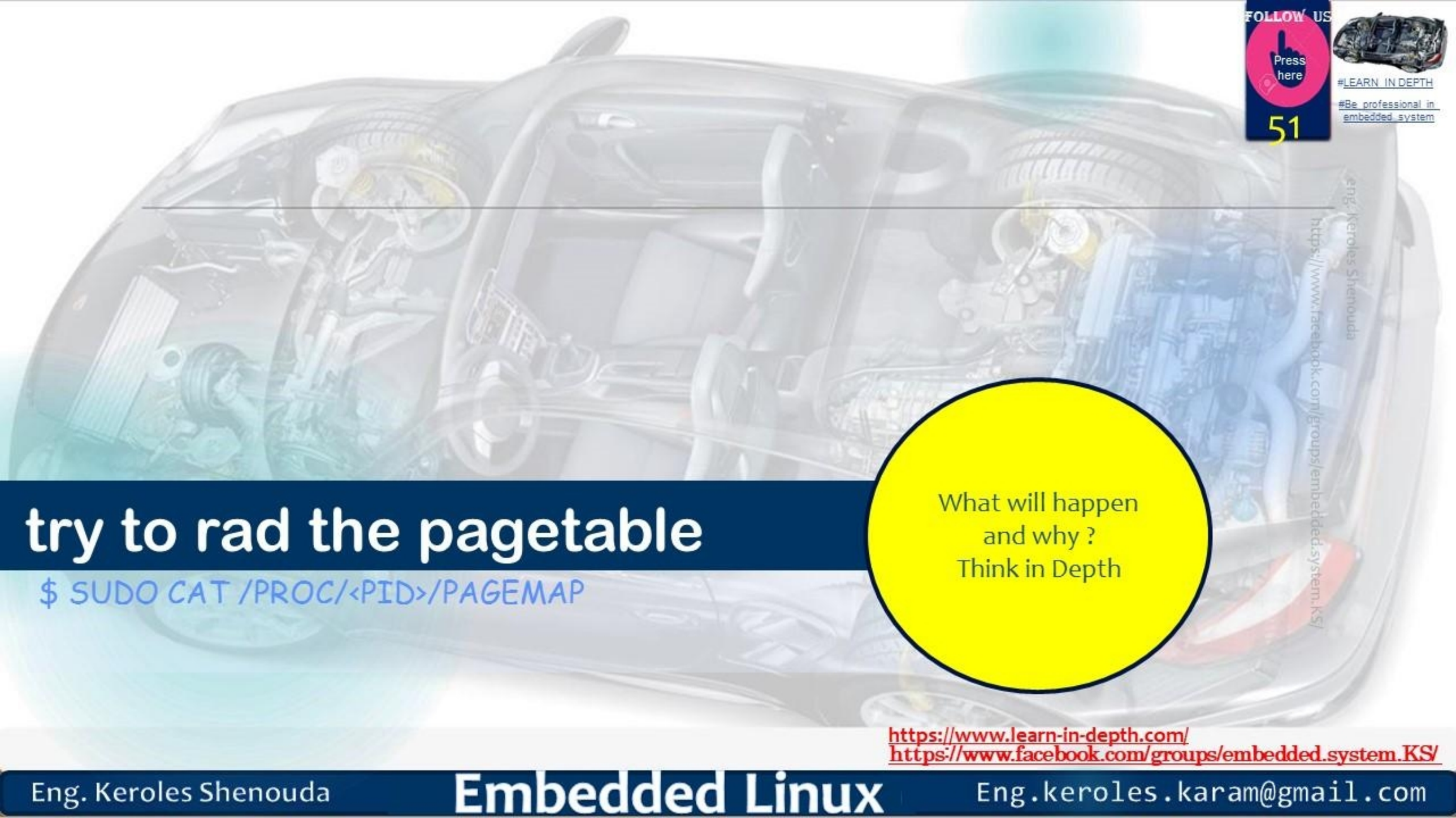
Open

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
/**
@Copyright www.learn-in-depth.com
Author: Keroles Shenouda
Learn In Depth
https://www.facebook.com/groups/embedded.system.KS
 */
int main(void)
{
	char *s;
	unsigned long int i;
s = (char*)malloc(100* sizeof(char));
s= "Learn In Depth Eng.Keroles Shenouda";
	if (s == NULL)
	{
		fprintf(stderr, "Can't allocate mem with malloc\n");
		return (EXIT_FAILURE);
	}
	i = 0;
	while (s)
	{
		printf("[%lu] %s\n", i, s);
		sleep(1);
		i++;
	}
	return (EXIT_SUCCESS);
}
```

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/



try to rad the pagetable

$ SUDO CAT /PROC/<PID>/PAGEMAP

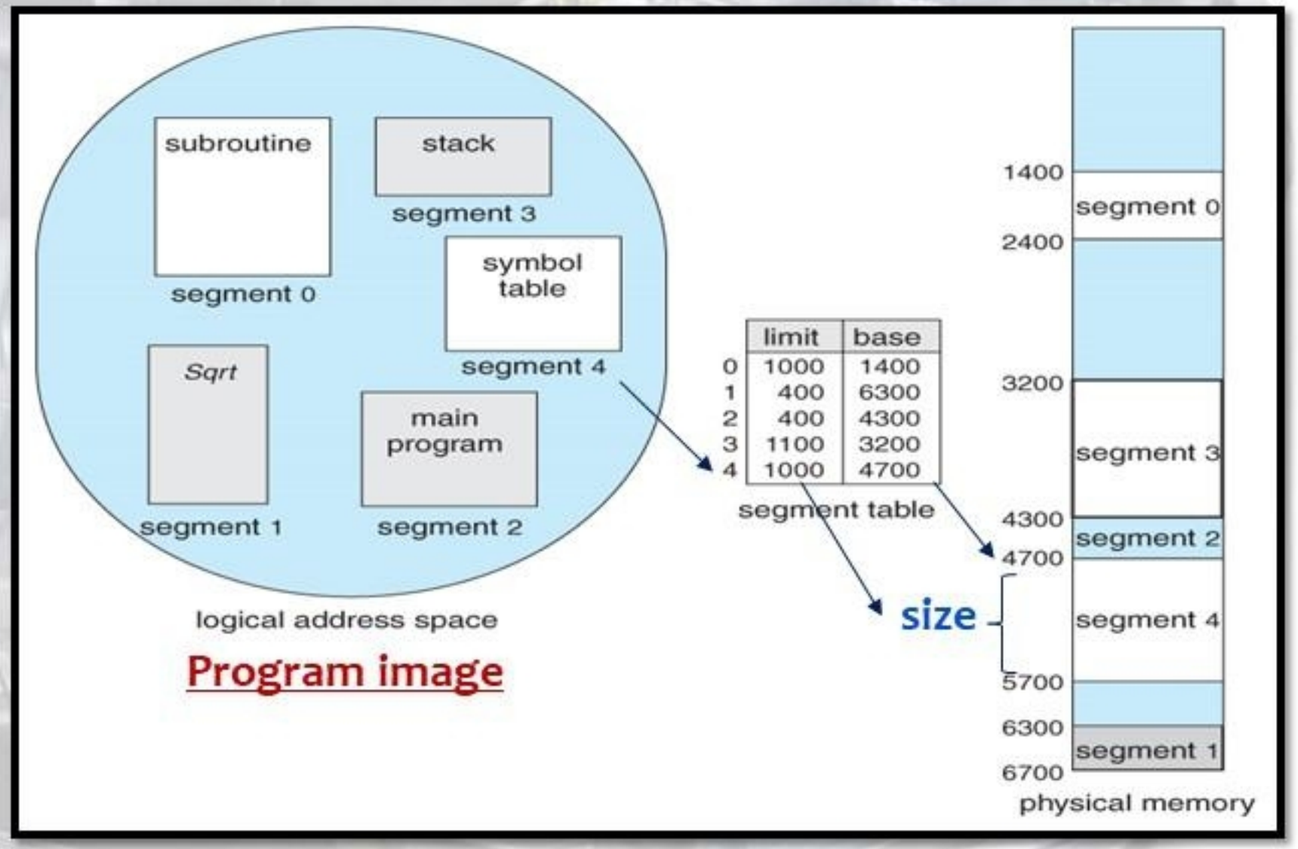
What will happen
and why ?
Think in Depth

Segmentation

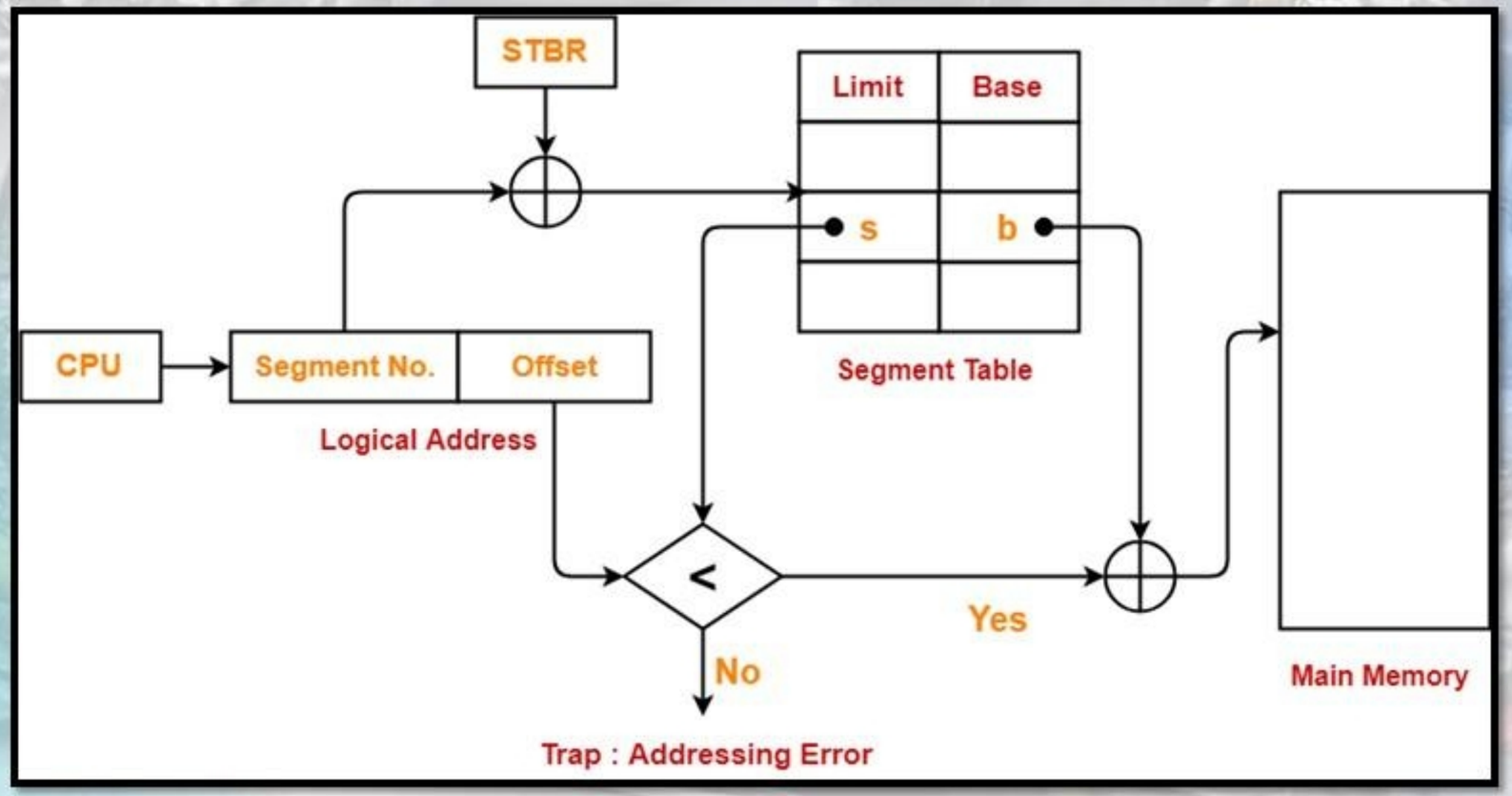
Segmentation

- Memory management scheme that supports **user view of memory**.
- A program is **a collection** of **segments**.
- A **segment** is a **logical unit** such as: subroutine, stack and main program.
- *Segment table* -each table entry has:
  - base - contains the starting physical address where the segments reside in memory.
  - *limit* - specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

segment number *s* is legal if *s* < STLR

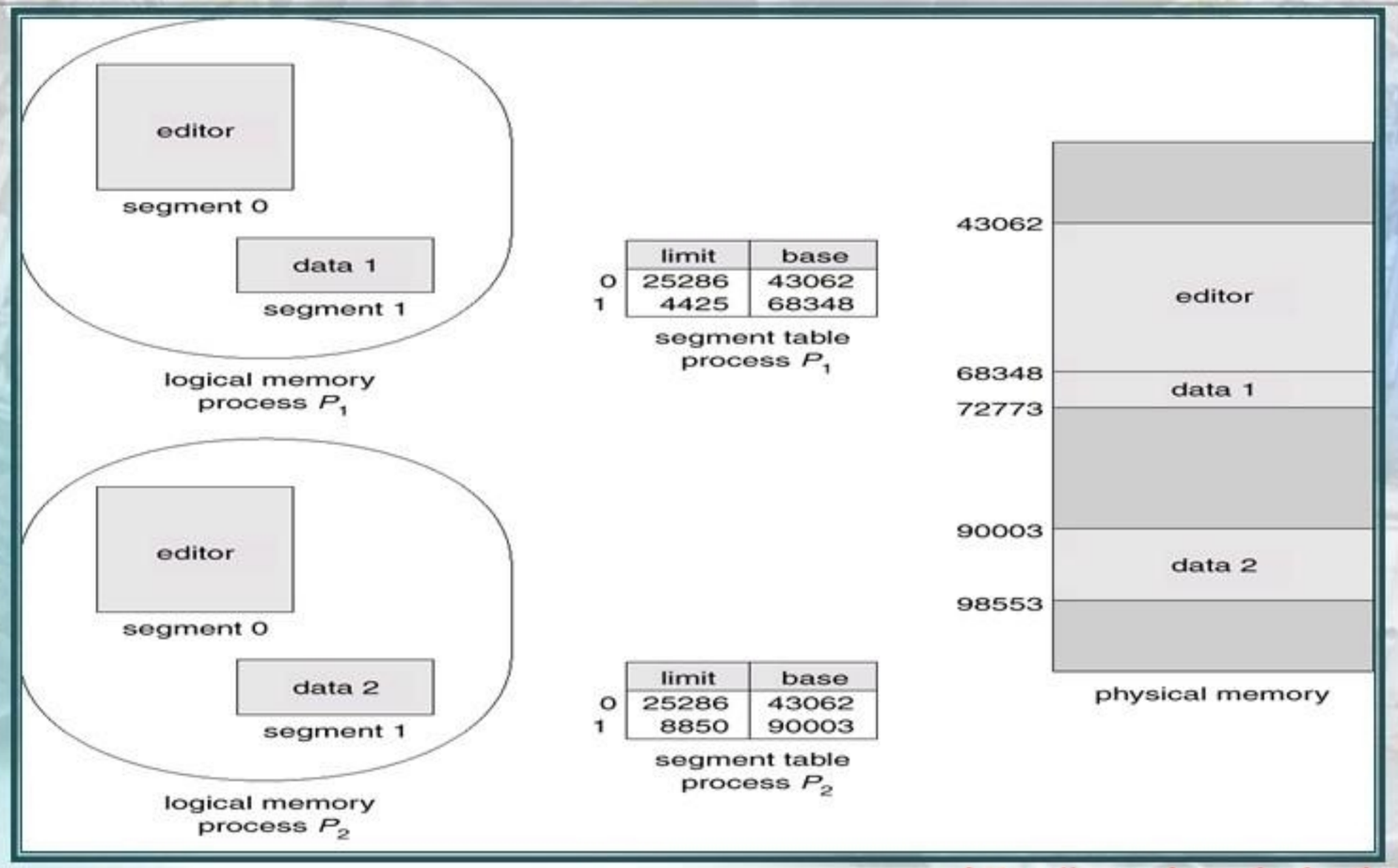


Address translation with segmentation architecture



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Sharing of Segments



Segmentation example

```
embedded_system_ks@embedded-KS: ~/labs/part2
embedded_system_ks@embedded-KS:~/labs/part2$ readelf lab1.elf  -a >> lab1.txt
embedded_system_ks@embedded-KS:~/labs/part2$ gedit lab1.txt &
[1] 13281
embedded_system_ks@embedded-KS:~/labs/part2$
```

```
 76  Section to Segment mapping:
 77   Segment Sections...
 78    00
 79    01     .interp
 80    02     .interp .note.ABI-tag .note.gnu.build-
   id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt
 81    03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
 82    04     .dynamic
 83    05     .note.ABI-tag .note.gnu.build-id
 84    06     .eh_frame_hdr
 85    07
 86    08     .init_array .fini_array .jcr .dynamic .got
 87
```

```
22 Section Headers:
23   [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
24   [ 0]                   NULL            00000000 000000 000000 00      0   0  0
25   [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
26   [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
27   [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
28   [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000024 04   A  5   0  4
29   [ 5] .dynsym           DYNSYM          080481d0 0001d0 000090 10   A  6   1  4
30   [ 6] .dynstr           STRTAB          08048260 000260 000067 00   A  0   0  1
31   [ 7] .gnu.version      VERSYM          080482c8 0002c8 000012 02   A  5   0  2
32   [ 8] .gnu.version_r    VERNEED         080482dc 0002dc 000020 00   A  6   1  4
33   [ 9] .rel.dyn          REL             080482fc 0002fc 000010 08   A  5   0  4
34   [10] .rel.plt          REL             0804830c 00030c 000028 08  AI  5  24  4
35   [11] .init             PROGBITS        08048334 000334 000023 00  AX  0   0  4
36   [12] .plt              PROGBITS        08048360 000360 000060 04  AX  0   0 16
37   [13] .plt.got          PROGBITS        080483c0 0003c0 000008 00  AX  0   0  8
38   [14] .text             PROGBITS        080483d0 0003d0 0001f2 00  AX  0   0 16
39   [15] .fini             PROGBITS        080485c4 0005c4 000014 00  AX  0   0  4
40   [16] .rodata           PROGBITS        080485d8 0005d8 000056 00   A  0   0  4
41   [17] .eh_frame_hdr     PROGBITS        08048630 000630 000034 00   A  0   0  4
42   [18] .eh_frame         PROGBITS        08048664 000664 0000e0 00   A  0   0  4
43   [19] .init_array       INIT_ARRAY      08049f08 000f08 000004 04  WA  0   0  4
44   [20] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 04  WA  0   0  4
45   [21] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
46   [22] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
47   [23] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
48   [24] .got.plt          PROGBITS        0804a000 001000 000020 04  WA  0   0  4
49   [25] .data             PROGBITS        0804a020 001020 000008 00  WA  0   0  4
50   [26] .bss              NOBITS          0804a028 001028 000008 00  WA  0   0  8
51   [27] .comment          PROGBITS        00000000 001028 00002d 01  MS  0   0  1
52   [28] .symtab           SYMTAB          00000000 001058 000460 10     29  47  4
53   [29] .strtab           STRTAB          00000000 0014b8 00022b 00      0   0  1
54   [30] .shstrtab         STRTAB          00000000 0016e3 00010a 00      0   0  1
```

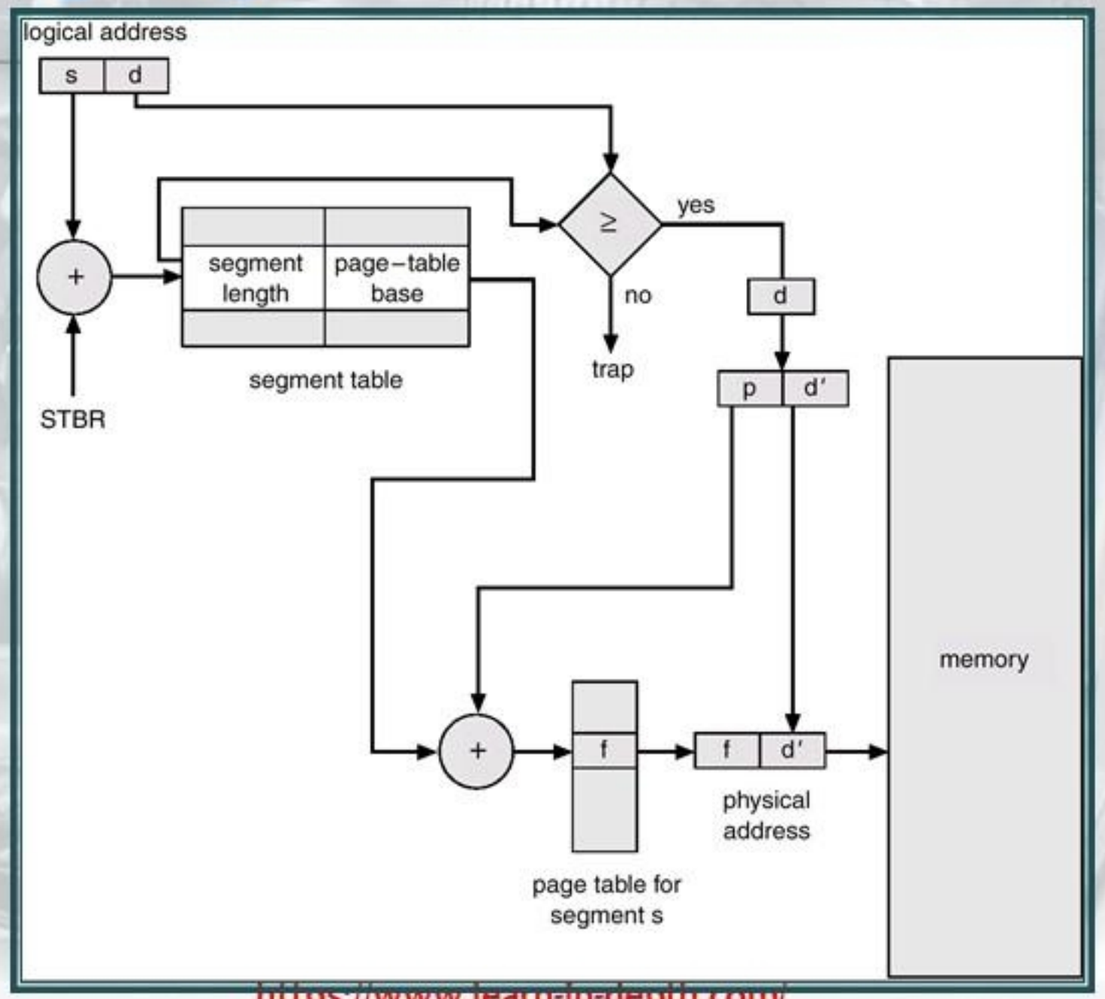
Imagine the mechanism for (Segmentation with Paging)

LEARN IN DEPTH ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Segmentation with Paging

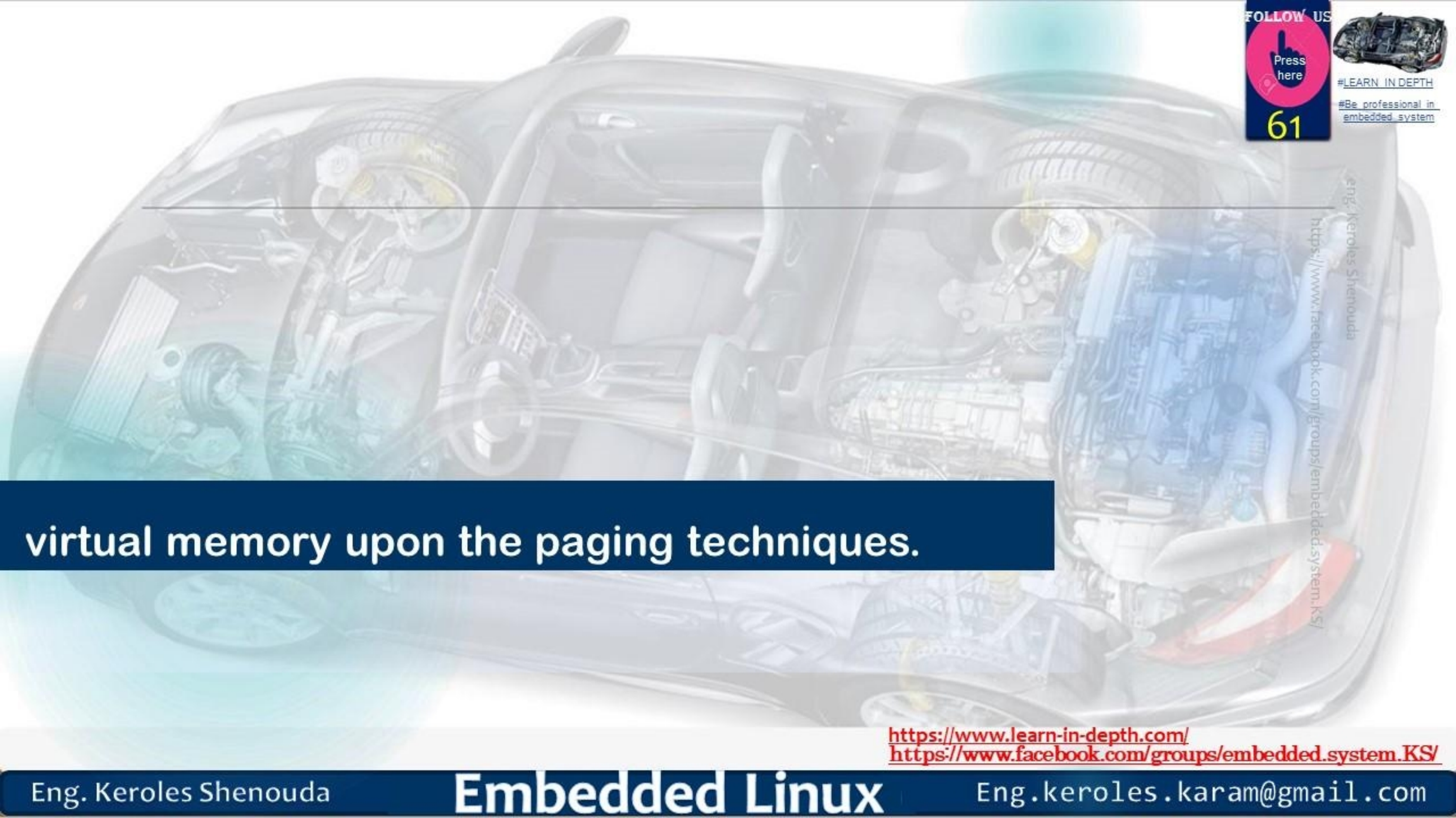
- Solve the problems of external fragmentation
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.



Summary ☺

Paging & segmentation

- In paging
  - Divide **physical memory** into **fixed-sized blocks** called **frames**
  - Divide **logical memory** into **blocks** of same size called **pages**.
  - Entire program image resides on disk. When the program starts, **only the first page is loaded. The reset of pages are loaded in memory on-demand.**
  - **Paging** enables the ***execution*** of program **larger** than physical memory.
  - In Paging, two memory access are required for execution.
    - One is for the page table and the other is for data/instruction.
  - In multi-level page table scheme, the page table itself is paged.
- In Segmentation
  - The program image can be loaded in segment level.

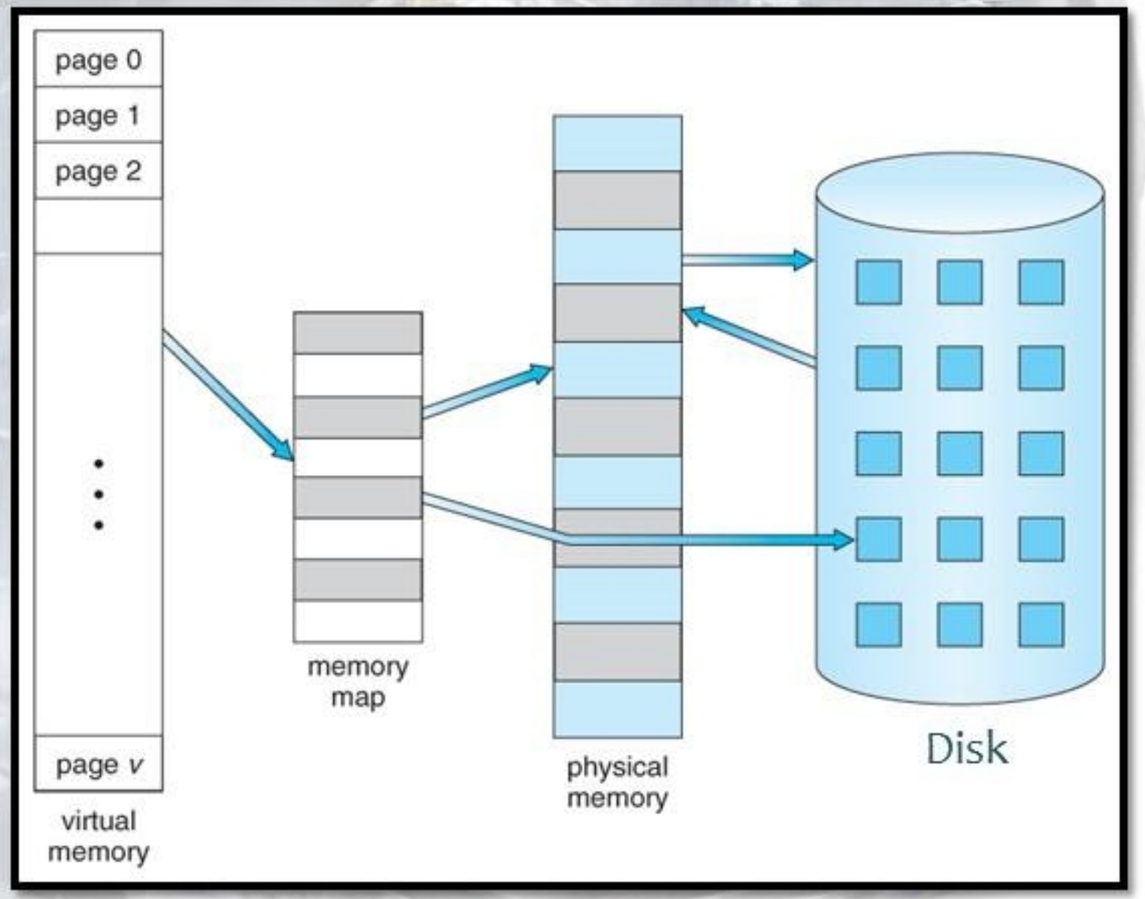


virtual memory upon the paging techniques.

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

virtual memory

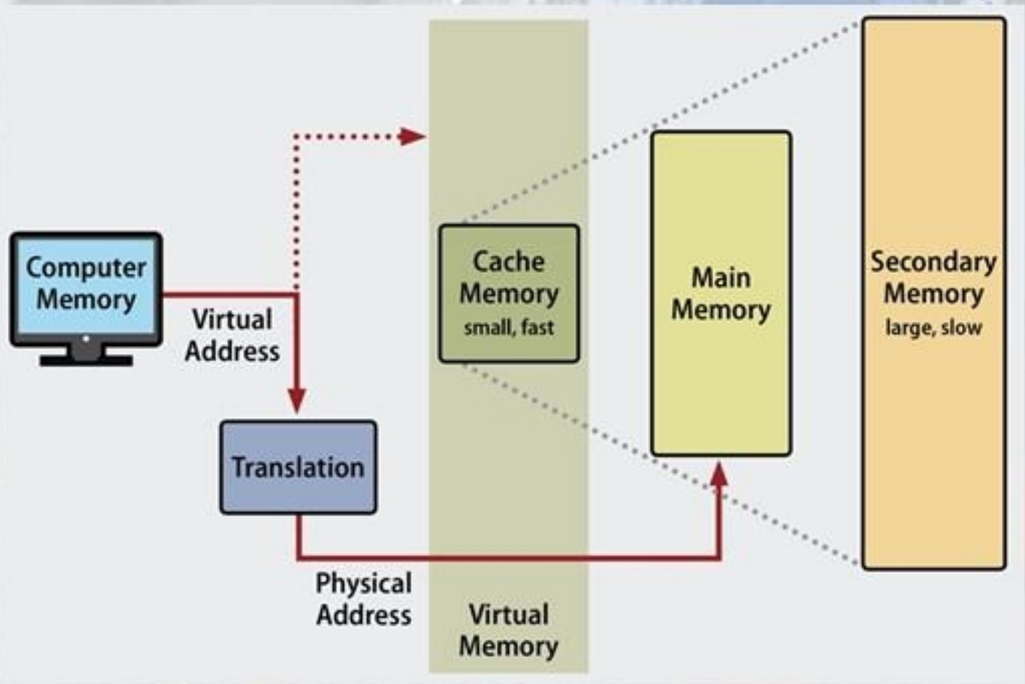
- Separation of ***user logical memory*** from ***physical memory.***
  - Only part of the program **needs to be in memory** for execution.
  - **Logical address space** can therefore be much **larger** than **physical address space.**
  - Allow **address spaces** to be **shared** by **several processes**.
  - Allows for more efficient process **creation**.
  - More programs running **concurrently**
- Virtual memory can be implemented Via:
  - Demand paging
  - Demand segmentation



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

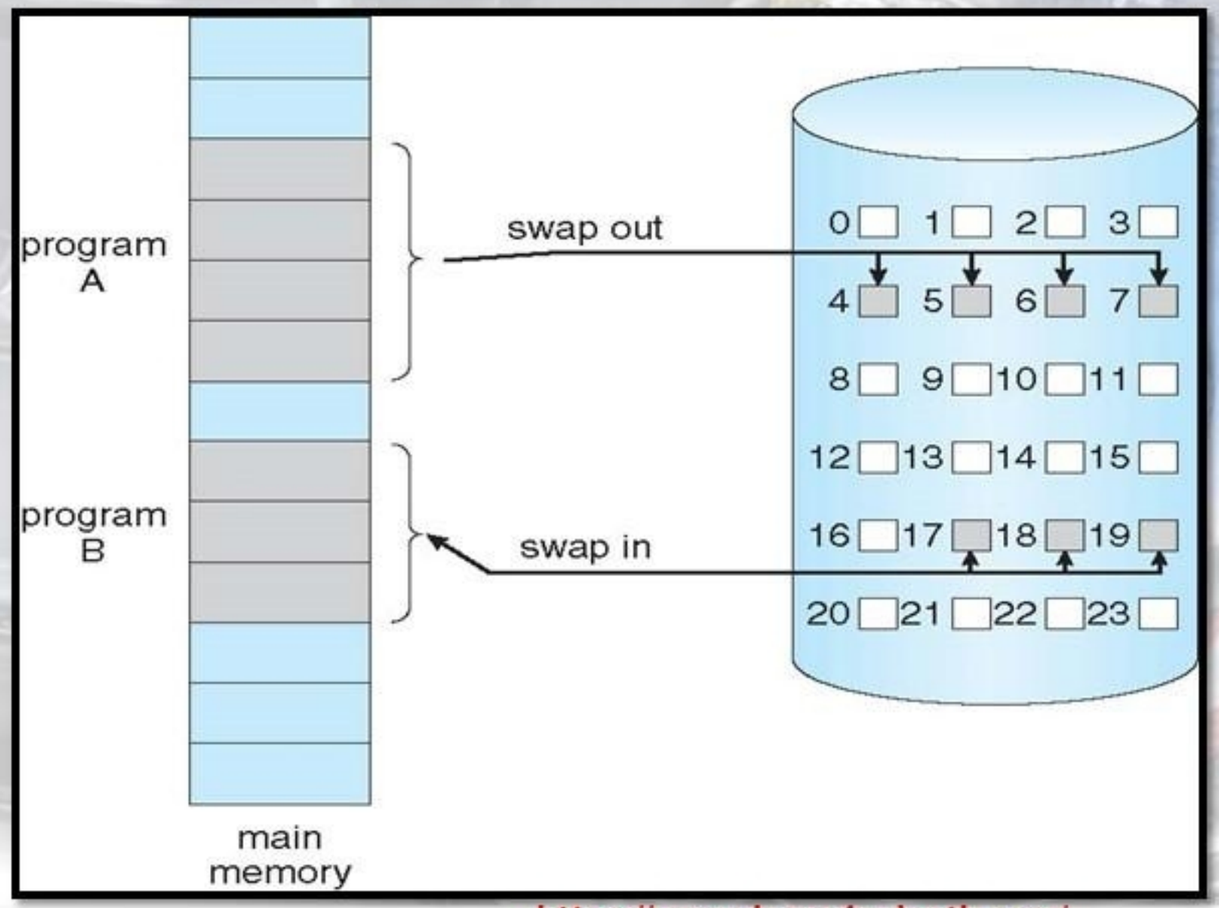
Virtual Memory Cont.

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical



Virtual memory can be implemented by Demand paging

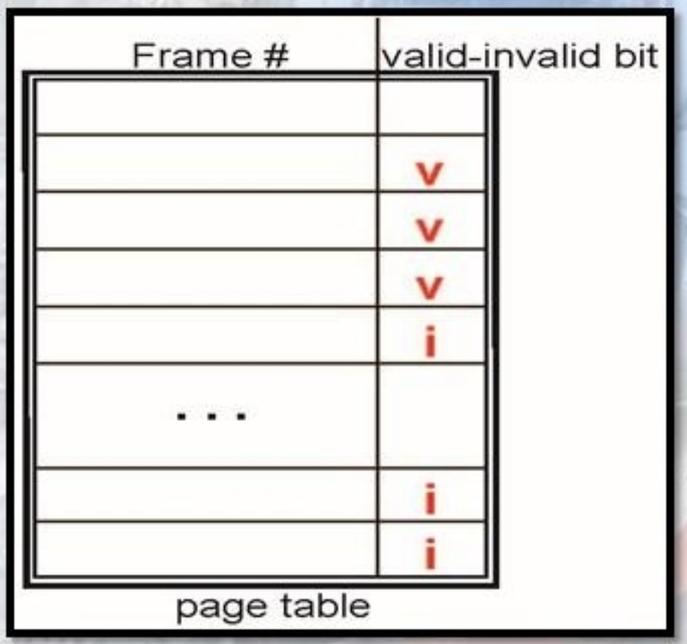
- bring a page into memory only when it is needed
  - Less memory needed
- Needs to allow pages to be **swapped** in and **out**.



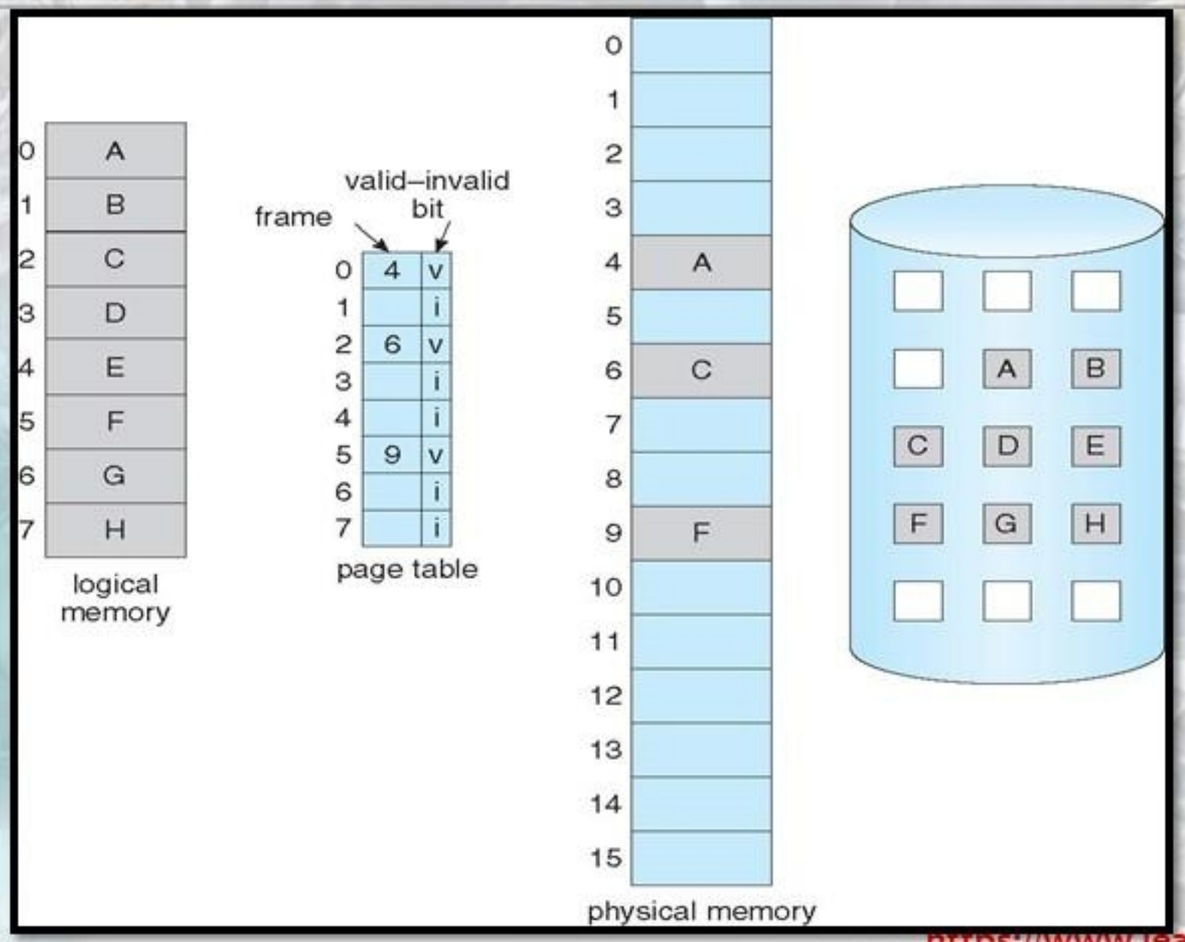
https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Demand paging Cont.

- When a page is referenced,
  - **Valid reference** -> reference it.
  - **Invalid reference** -> not-in-memory -> bring it into memory.
- Valid-invalid bit is associated with each page table entry.
- **V-> in-memory**
- **i->not-in-memory or not legal (bad address, protection violation)**
- **During address translation, if valid-invalid bit is I, page fault**

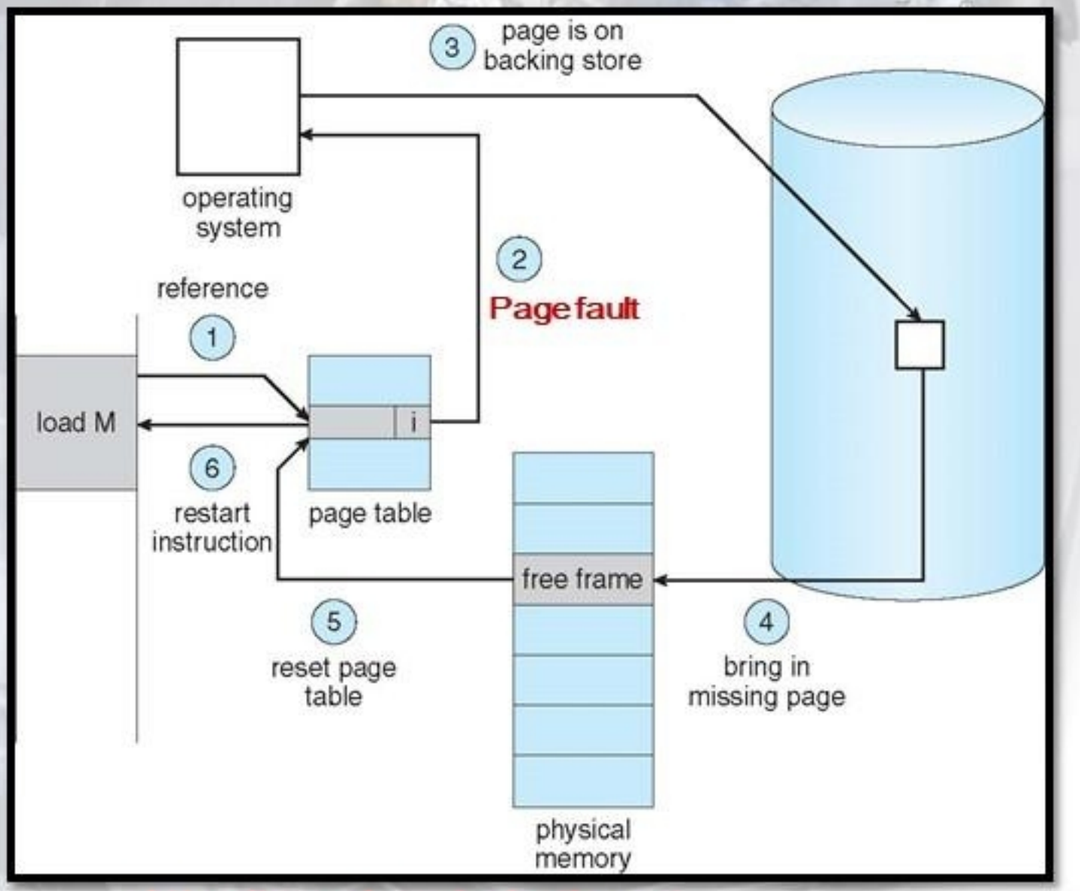


Page Table When Some Pages Are Not in Main Memory



Page Fault

- When CPU refers to page table to use page number **M**
- There is no page in memory since the bit of this page is **invalid**
- There is **page fault** when there is page fault it calls page fault handler in the operating system
- Since page **M** should be on **disk** it loads that page into an empty page frame.
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory
  Set validation bit = **v**
- Restart the instruction that caused the page fault



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

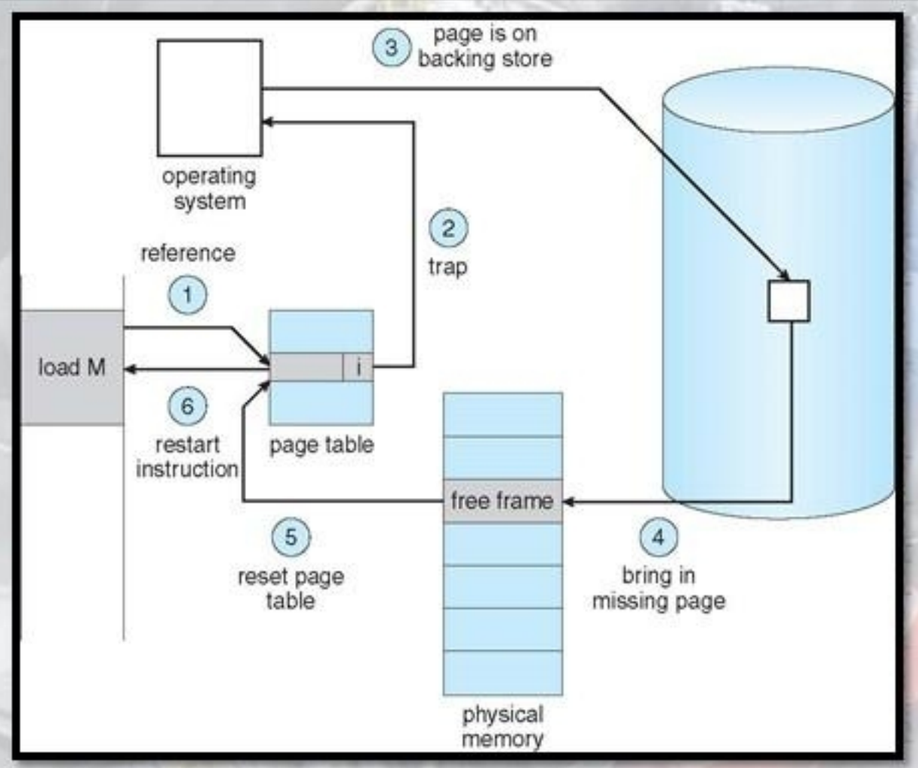
Aspects of Demand Paging

- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Performance of Demand Paging

- Assume the (worse case) page fault
- Three major activities
  - Service the interrupt - careful coding means just several hundred instructions needed
  - Read the page - lots of time
  - Restart the process - again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access} + p\ (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$



Demand Paging Example

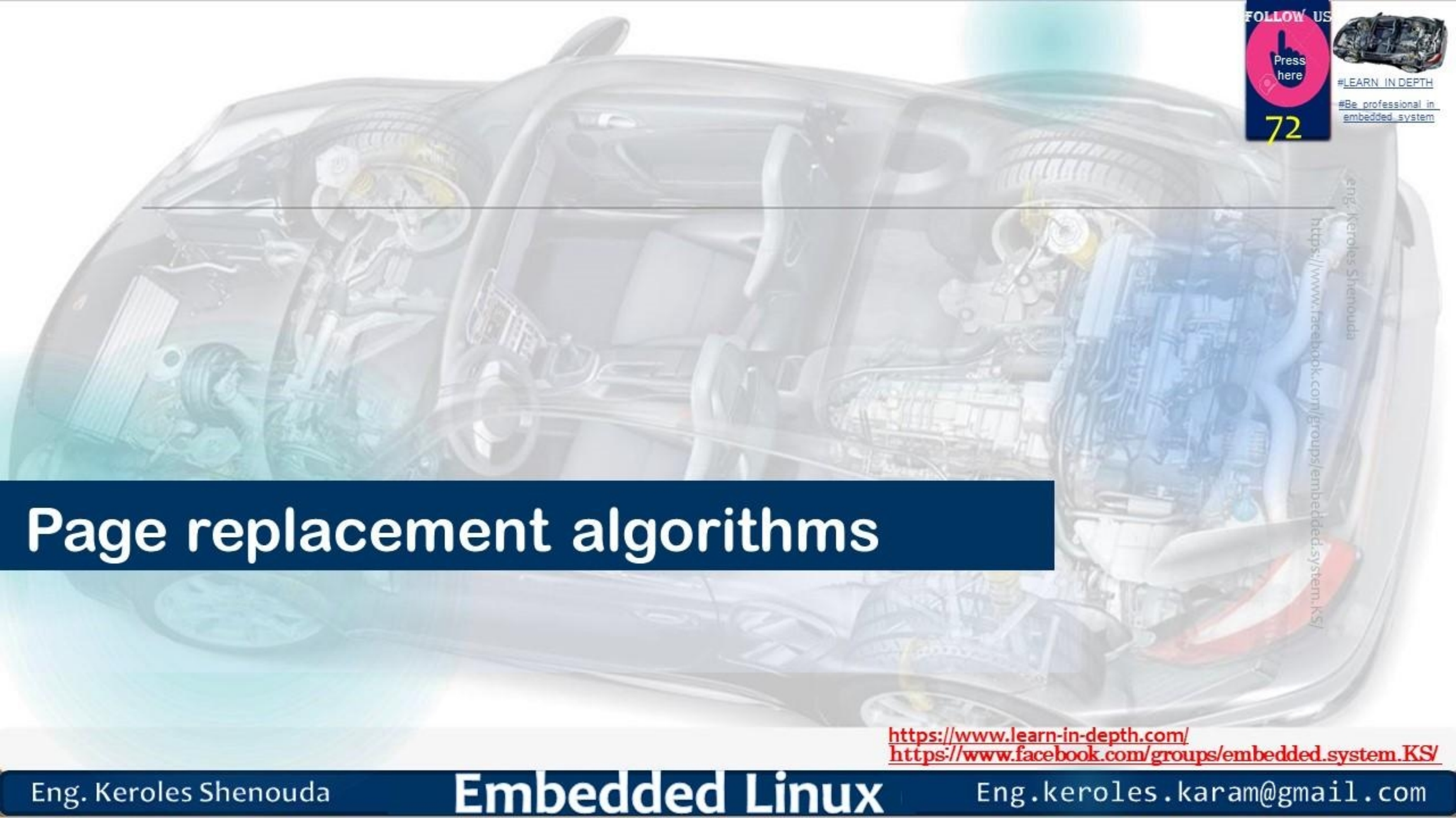
- Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - EAT = (1 – p) x 200 + p (8 milliseconds)
    = (1 – p) x 200 + p x 8,000,000
    = 200 + p x 7,999,800
  - If one access out of 1,000 causes a page fault, then
  - P=1/1000 = 0.001
    EAT = 8.2 microseconds.
- It is important to keep the **page fault rate low**
  - **Large** memory is required
  - Good **page replacement** policy is required

Use the ps command to view page faults for <PID>

- min_flt : Number of minor page faults.
- maj_flt : Number of major page faults.

```
embedded_system_ks@embedded-KS: ~/labs/part2
embedded_system_ks@embedded-KS:~/labs/part2$ ps -a
  PID TTY          TIME CMD
12652 pts/0    00:00:15 gedit
15111 pts/3    00:00:00 lab1.elf
15114 pts/1    00:00:00 ps
embedded_system_ks@embedded-KS:~/labs/part2$ ps -o min_flt,maj_flt,cmd 15111
 MINFL  MAJFL CMD
    72      0 ./lab1.elf
embedded_system_ks@embedded-KS:~/labs/part2$ ps -o min_flt,maj_flt,cmd 15111
 MINFL  MAJFL CMD
    72      0 ./lab1.elf
embedded_system_ks@embedded-KS:~/labs/part2$
```

```
embedded_system_ks@embedded-KS: ~/labs/part2
[60] Learn In Depth Eng.Keroles Shenouda
[61] Learn In Depth Eng.Keroles Shenouda
[62] Learn In Depth Eng.Keroles Shenouda
[63] Learn In Depth Eng.Keroles Shenouda
[64] Learn In Depth Eng.Keroles Shenouda
[65] Learn In Depth Eng.Keroles Shenouda
[66] Learn In Depth Eng.Keroles Shenouda
[67] Learn In Depth Eng.Keroles Shenouda
[68] Learn In Depth Eng.Keroles Shenouda
[69] Learn In Depth Eng.Keroles Shenouda
[70] Learn In Depth Eng.Keroles Shenouda
[71] Learn In Depth Eng.Keroles Shenouda
[72] Learn In Depth Eng.Keroles Shenouda
```



Page replacement algorithms

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Page replacement algorithms

- In **operating systems**, whenever a new page is referred and not present in memory, page fault occurs and **Operating System** **replaces** **one of the existing pages with newly needed page.**
- Different page **replacement algorithms** suggest different ways to decide which page to replace.
- **The target for all algorithms is to reduce number of page faults.**
- Page replacement
  - Find some page not really in use -> swap it out .
- Handling the loop
  - Same pages may be brought into memory several times.
- Algorithms examples
  - Optimal Page replacement
  - LRU (Least Recent Used)
  - LRU Approximation Algorithms

Optimal Page replacement

- Replace page that will not be used for longest period of time
- In this algorithm, OS replaces the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10
	2									

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2								
		3								

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3
Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2							
		3	3							
			4							

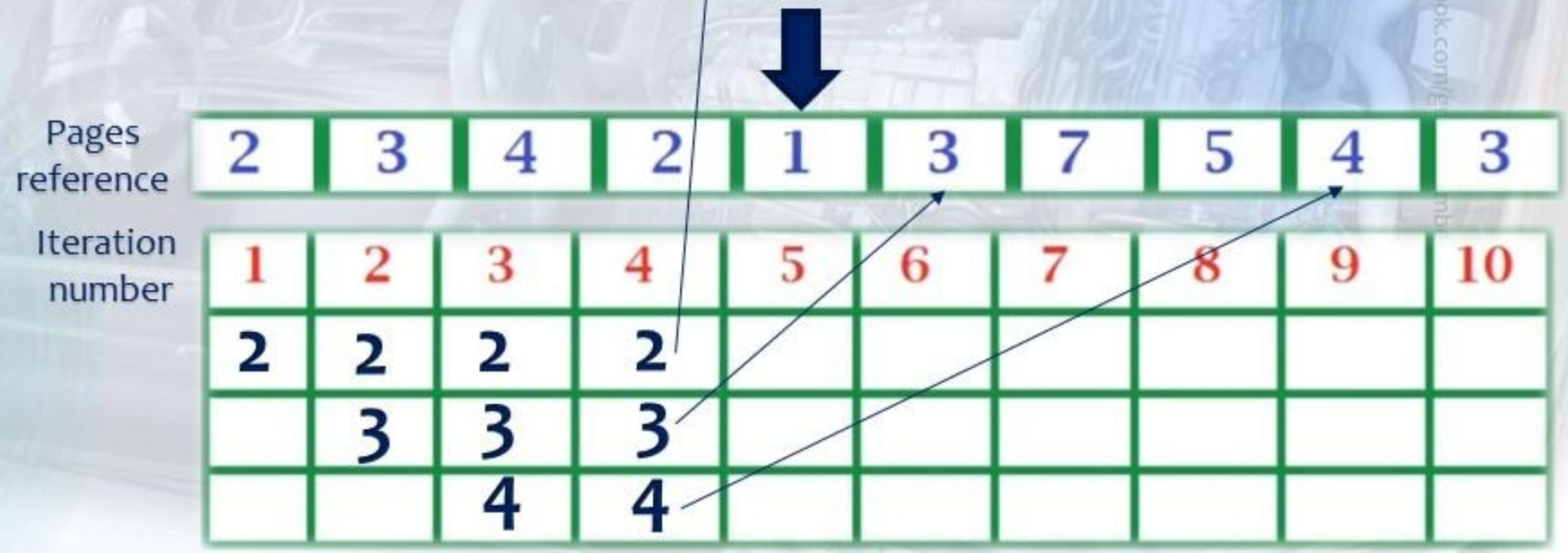
Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3
Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2	2						
		3	3	3						
			4	4						

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future



Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2	2	1					
		3	3	3	3					
			4	4	4					

Optimal Page replacement Cont.

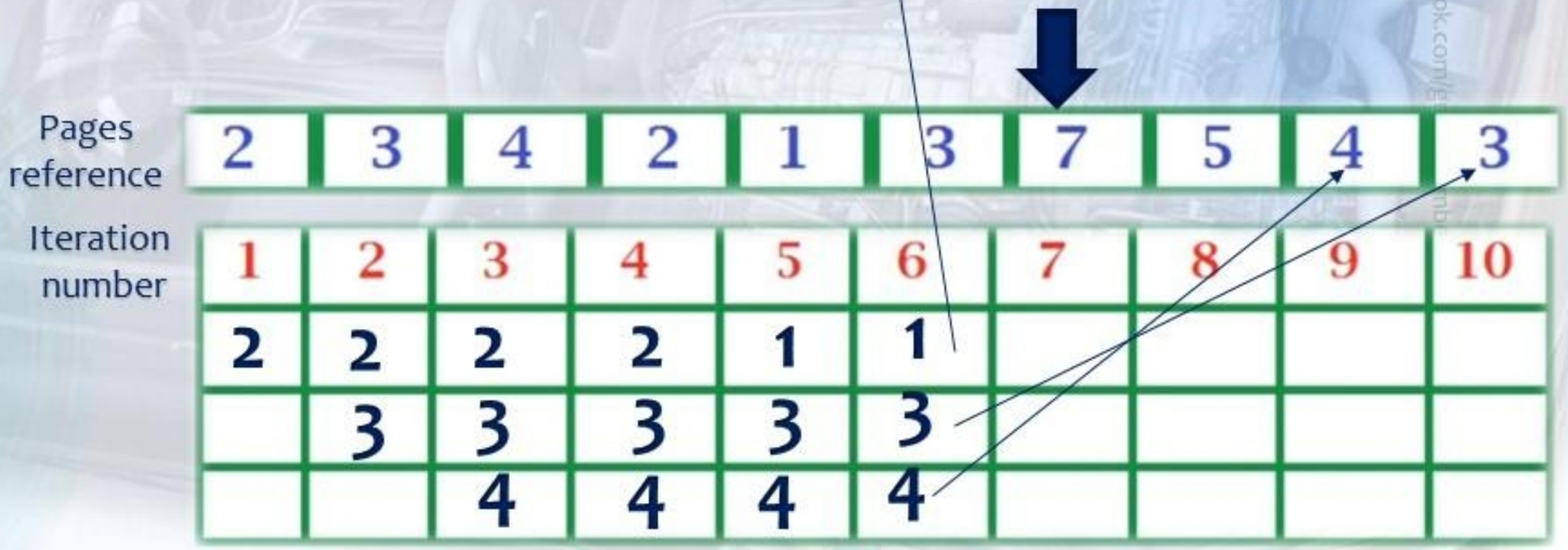
- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2	2	1	1				
		3	3	3	3	3				
			4	4	4	4				

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future



Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2	2	1	1	7			
		3	3	3	3	3	3			
			4	4	4	4	4			

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3

Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2	2	1	1	7	5		
		3	3	3	3	3	3	3		
			4	4	4	4	4	4		

Optimal Page replacement Cont.

- Replace page that will not be used for longest period of time
- In this algorithm, **OS replaces** the page that will not be used for the longest period of time in future

Pages reference	2	3	4	2	1	3	7	5	4	3
Iteration number	1	2	3	4	5	6	7	8	9	10
	2	2	2	2	1	1	7	5	5	5
		3	3	3	3	3	3	3	3	3
			4	4	4	4	4	4	4	4

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2

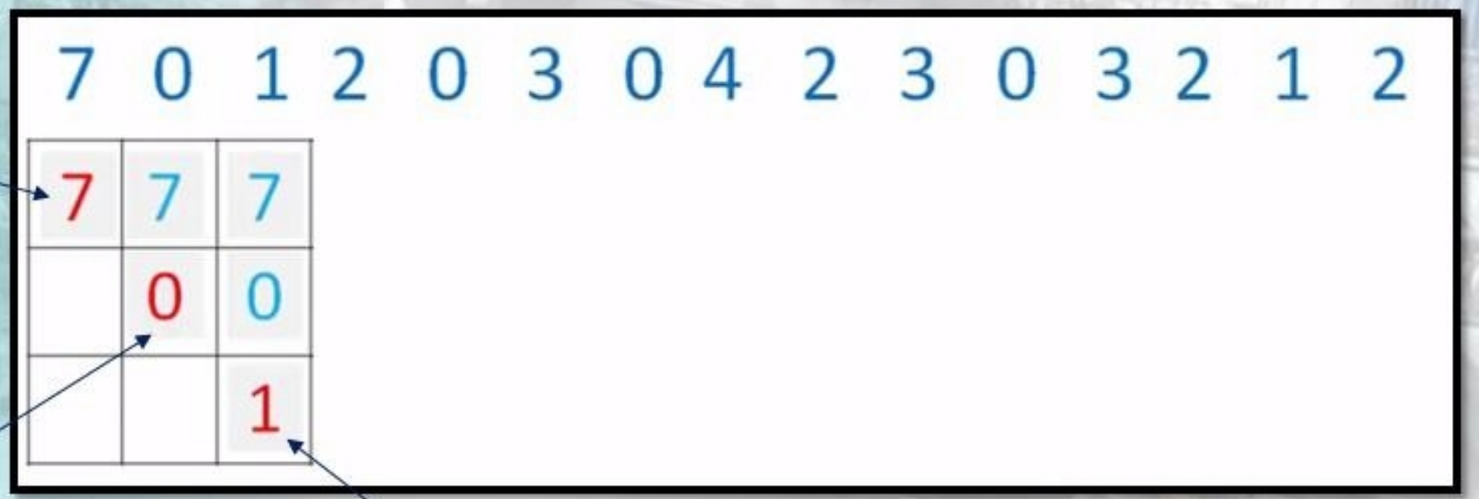
LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7												
	0	0												
		1												

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

Third recently used

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2											
	0	0	0											
		1	1											

Second recently used

First recently used

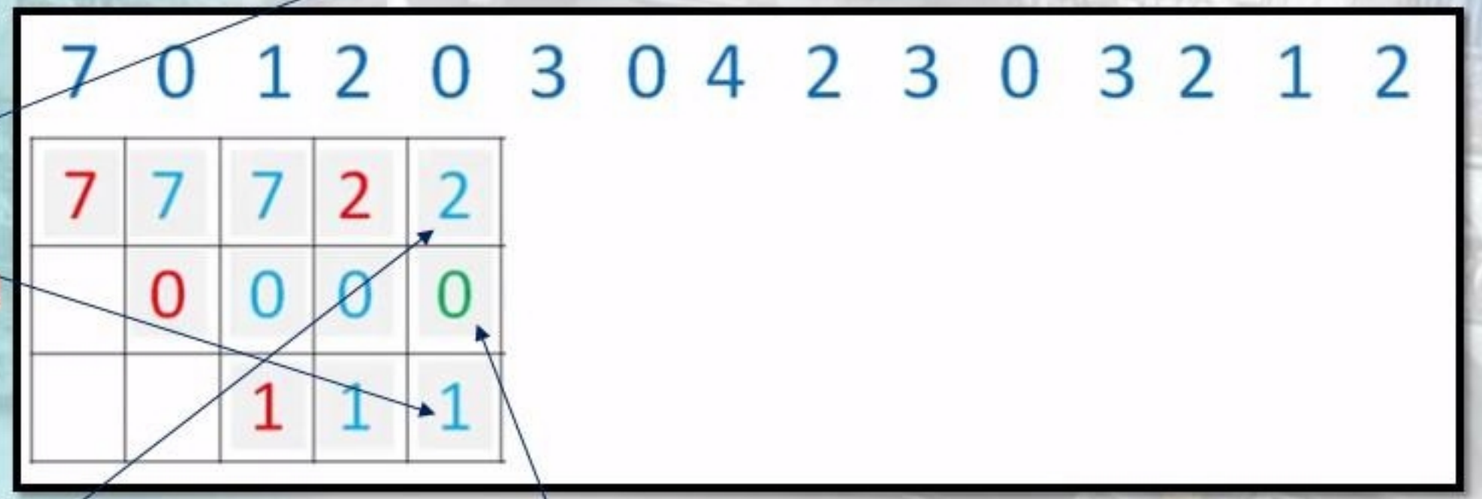
LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2										
	0	0	0	0										
		1	1	1										

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.



LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2									
	0	0	0	0	0									
		1	1	1	3									

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2								
	0	0	0	0	0	0								
		1	1	1	3	3								

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2								
	0	0	0	0	0	0								
		1	1	1	3	3								

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4							
	0	0	0	0	0	0	0							
		1	1	1	3	3	3							

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4						
	0	0	0	0	0	0	0	0						
		1	1	1	3	3	3	2						

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4					
	0	0	0	0	0	0	0	0	3					
		1	1	1	3	3	3	2	2					

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4	0				
	0	0	0	0	0	0	0	0	3	3				
		1	1	1	3	3	3	2	2	2				

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4	0	0			
	0	0	0	0	0	0	0	0	3	3	3			
		1	1	1	3	3	3	2	2	2	2			

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4	0	0	0		
	0	0	0	0	0	0	0	0	3	3	3	3		
		1	1	1	3	3	3	2	2	2	2	2		

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4	0	0	0	1	
	0	0	0	0	0	0	0	0	3	3	3	3	3	
		1	1	1	3	3	3	2	2	2	2	2	2	

LRU (Least Recent Used)

In this algorithm page will be replaced which is **least recently used**.

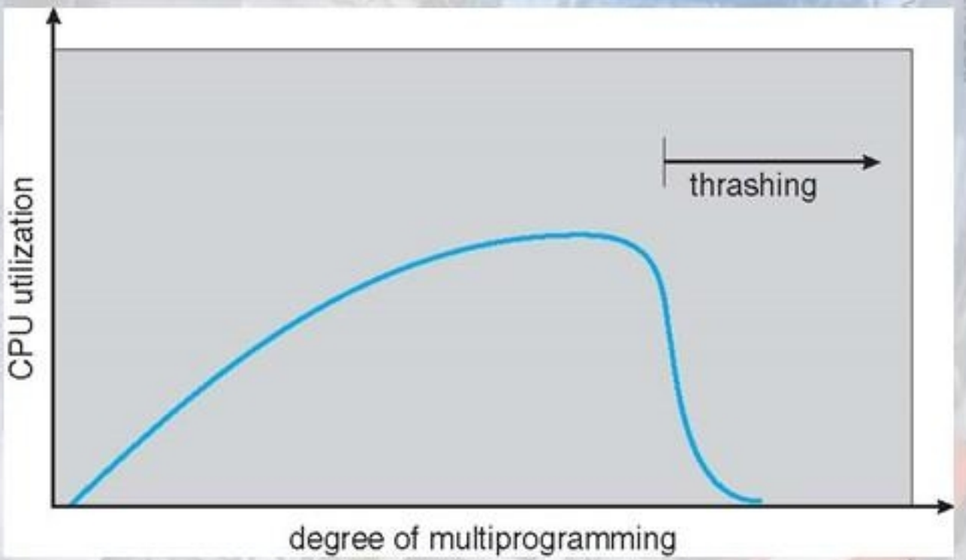
Notes:

- Reference times should be recorded for each page.
- Page with the oldest reference time should be found
- Too large space and time overhead.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1
	0	0	0	0	0	0	0	0	3	3	3	3	3	3
		1	1	1	3	3	3	2	2	2	2	2	2	2

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out



Summary ☺

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Virtual Memory

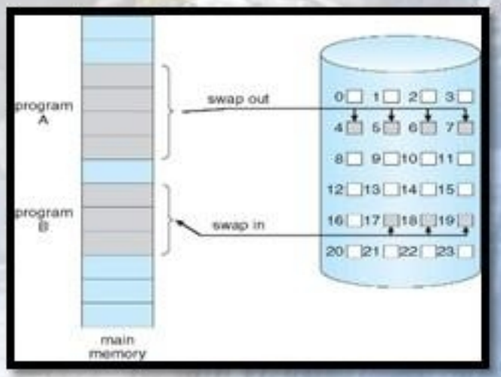
Virtual memory can be implemented Via:

- Separation of user logical memory from physical memory.
  - Only part of the program **needs** to be in memory for execution.
  - Logical address space can therefore be much **larger** than physical address space.
  - Allow address spaces to be **shared** by several processes.

Demand segmentation

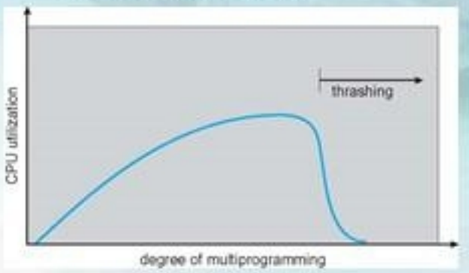
Demand paging

- bring a page into memory only when it is needed
  - Less memory needed
- Needs to allow pages to be **swapped** in and **out**.



Page Fault

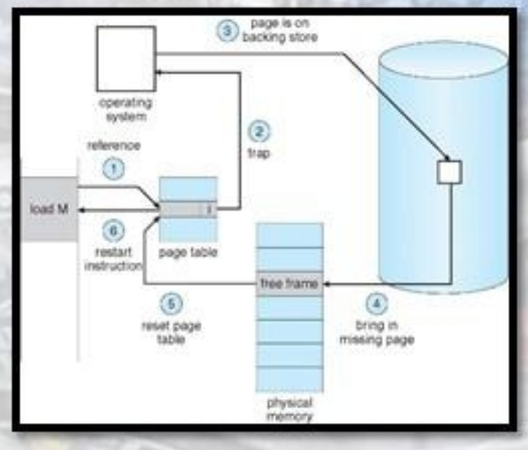
- **Thrashing** ≡ a process is busy swapping pages in and out



Page replacement algorithms

Optimal Page replacement

LRU (Least Recent Used)



https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

Learn-in-depth.com

References

- Embedded Linux training
  - https://bootlin.com/training/
- Linux kernel and driver development training
- Yocto Project and OpenEmbedded development training
- Buildroot development training
- Building Embedded Linux Systems, 2nd Edition
- Mastering Embedded Linux Programming - Second Edition
- Christopher Hallinan, **Embedded Linux Primer,** Prentice Hall, 2006.
- **Silberschatz**, Galvin, and Gagne's **Operating System Concepts**, **Eighth Edition**.
- Robert love, linux kernel develoepemnt 3 rd edition 2010
- **Advanced Linux Programming By Mark Mitchell, Jeffrey Oldham, Alex Samuel**
- **Linux OS in Embedded Systems & Linux Kernel Internals(2/2)**
  - Memory Management, Paging, Virtual Memory, File system and its implementation, Secondary Storage(HDD), I/O systems

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

- Memory Management article
- Introduction to Operating Systems Class 9 - Memory Management
- Virtual Memory lecture for Introduction to Computer Architecture at Uppsala University.
- OS Lecture Note 13 - Address translation, Caches and TLBs
- CSE 331 Operating Systems Design lectures
- CSE 331 Virtual Memory
- CSE 332 Computer Organization and Architecture
- File Systems: Fundamentals.

https://www.learn-in-depth.com/
https://www.facebook.com/groups/embedded.system.KS/

- **File Permissions in Linux/Unix with Example**
- Introduction to UNIX / Linux – 4
- **12.2 Basic I/O Concepts**
- Communicating with Hardware
- I/O Systems I/O Hardware Application I/O Interface
- COMPUTER ARCHITECTURE
- **OS**
- Linux Operating System
- W4118 Operating Systems, Instructor: Junfeng Yang
- CSNB334 Advanced Operating Systems 4. Process & Resource Management.
- Using the POSIX API
- **Linux Memory Management**
- Chapter 2: Operating-System Structures
- **POSIX Threads Programming**
- Pthreads: A shared memory programming model https://slideplayer.com/slide/8734550/