3150 - Operating Systems

Dr. WONG Tsz Yeung

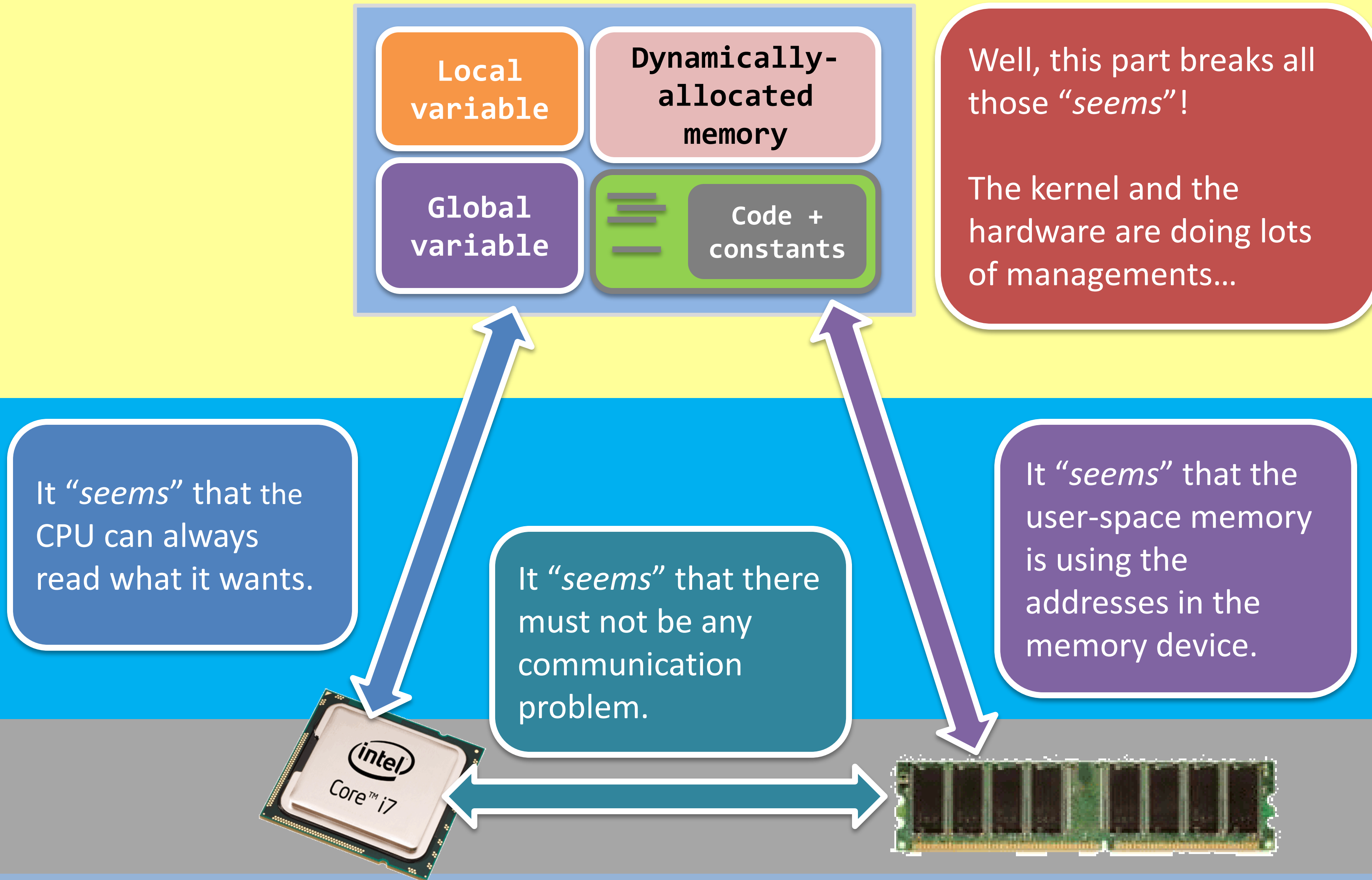
p.56-81 are not included in the final examination.

You can skip printing them and save paper.

Chapter 4, part 2 - Virtual Memory Support

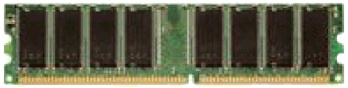
- Virtual memory: memory is a physical device that virtually stores things that you think they are real.

Outline


Local variable
Dynamically-allocated memory
Global variable
Code + constants
Well, this part breaks all those "*seems*"!
The kernel and the hardware are doing lots of managements…
It "*seems*" that the CPU can always read what it wants.
It "*seems*" that there must not be any communication problem.
It "*seems*" that the user-space memory is using the addresses in the memory device.
intel Core™ i7

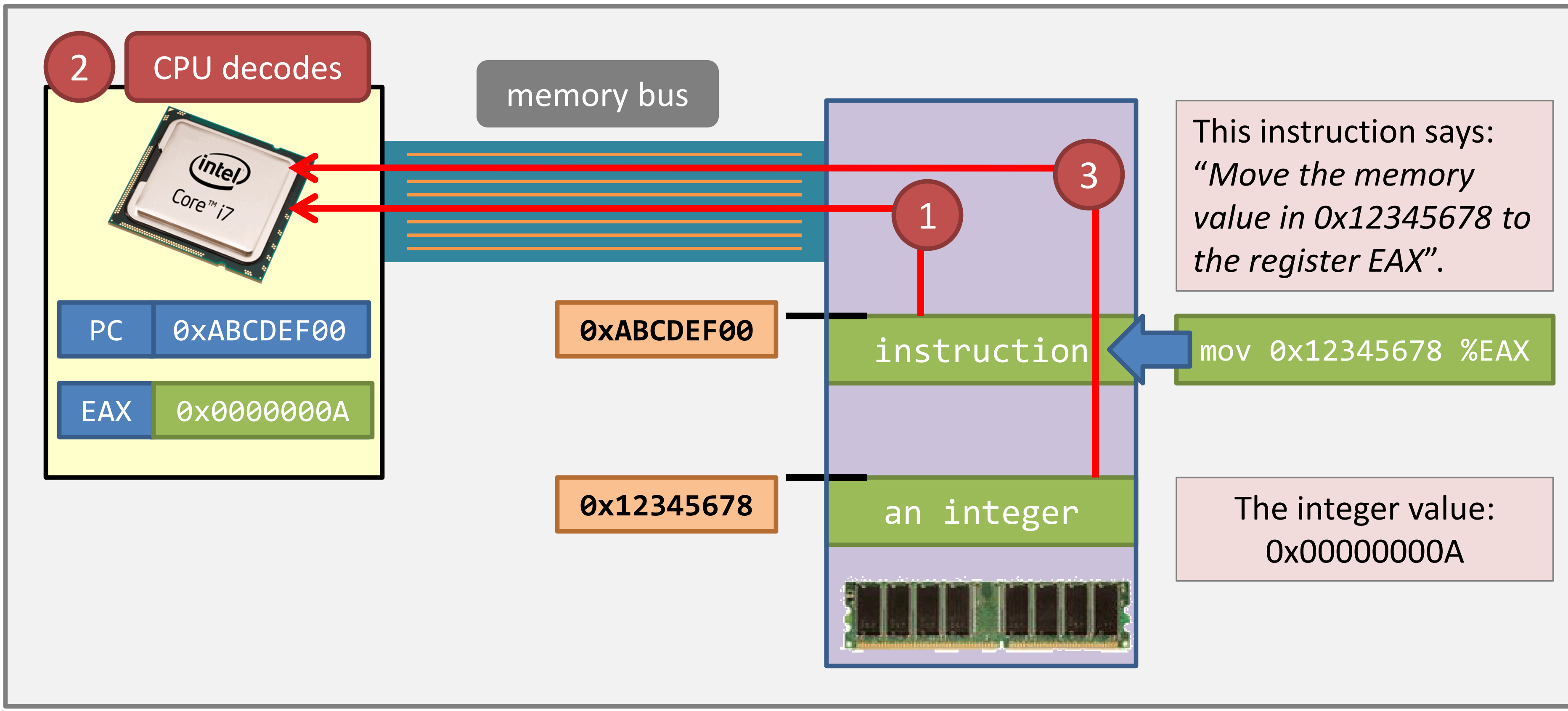
Memory Management

- Virtual memory = CPU + MMU;



CPU working – illustration that you may know

- Let's review the "**fetch-decode-execute**" cycle!



“You’ve been living in a dream world, Neo”

```
int main(void) {
    int pid;
    pid = fork();
    printf("PID %d: %p.\n", getpid(), &pid);
    if(pid)
        wait(NULL);
    return 0;
}
```

```
$ ./same_addr
PID 1234: 0xbfe85e0c.
PID 1235: 0xbfe85e0c.
$ _
```

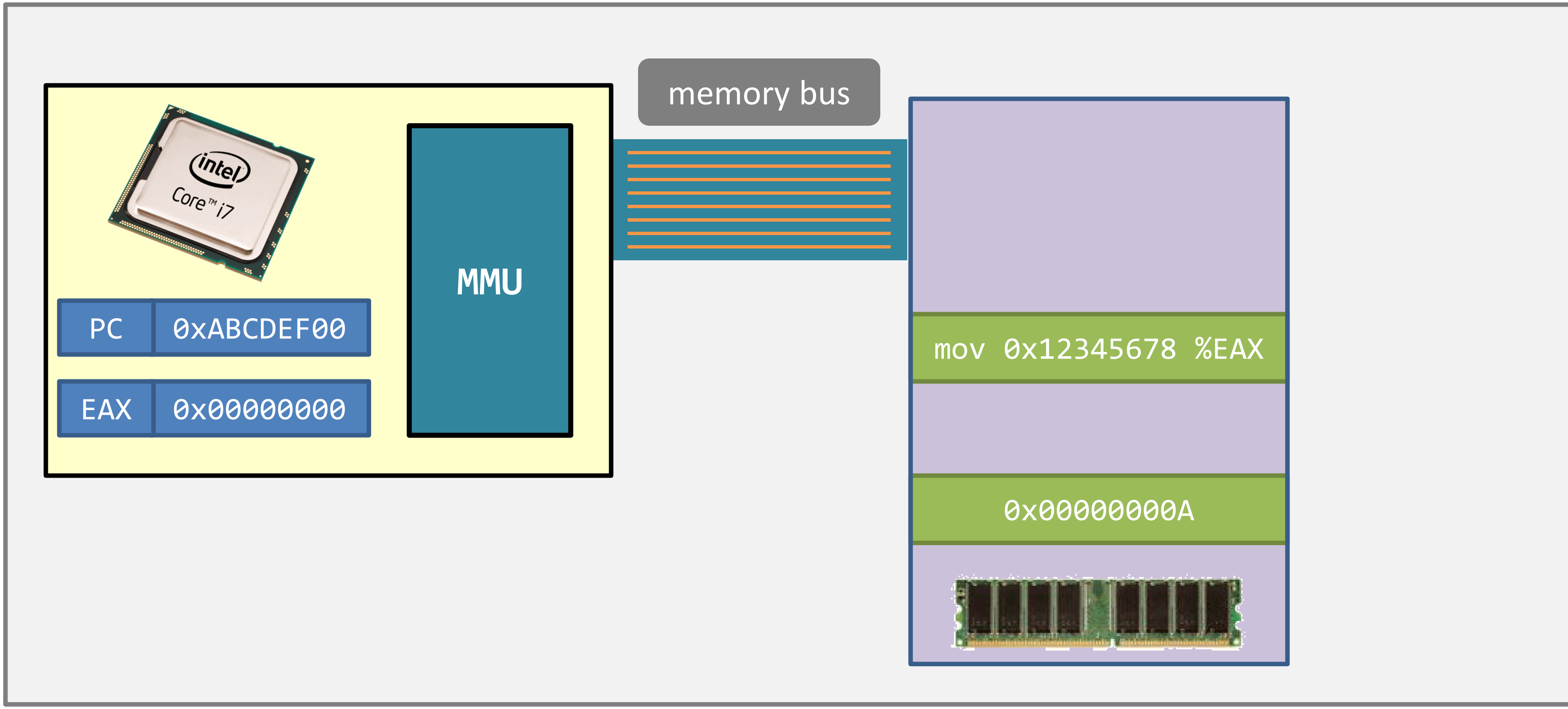
- What can you say about the result?
  - Two **different processes**, the **same variable name,** carry **different values**, use the **same address**!

    (What? How COME?!)
- Well, what is the <u>meaning</u> of a memory address?!

```
[examples@3150] cat same_addr.c
```

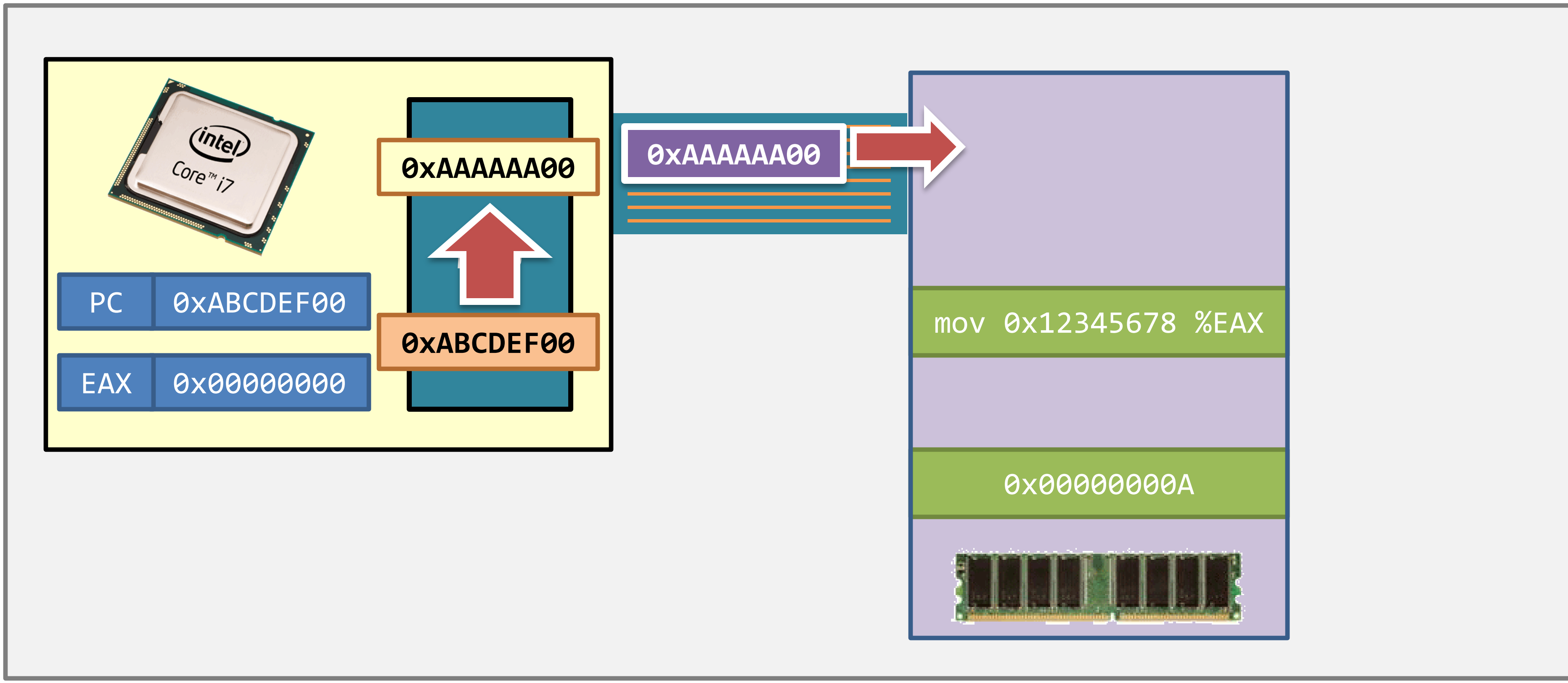
Virtual memory support in modern CPUs

- The **MMU – memory management unit** – is an on-CPU device.



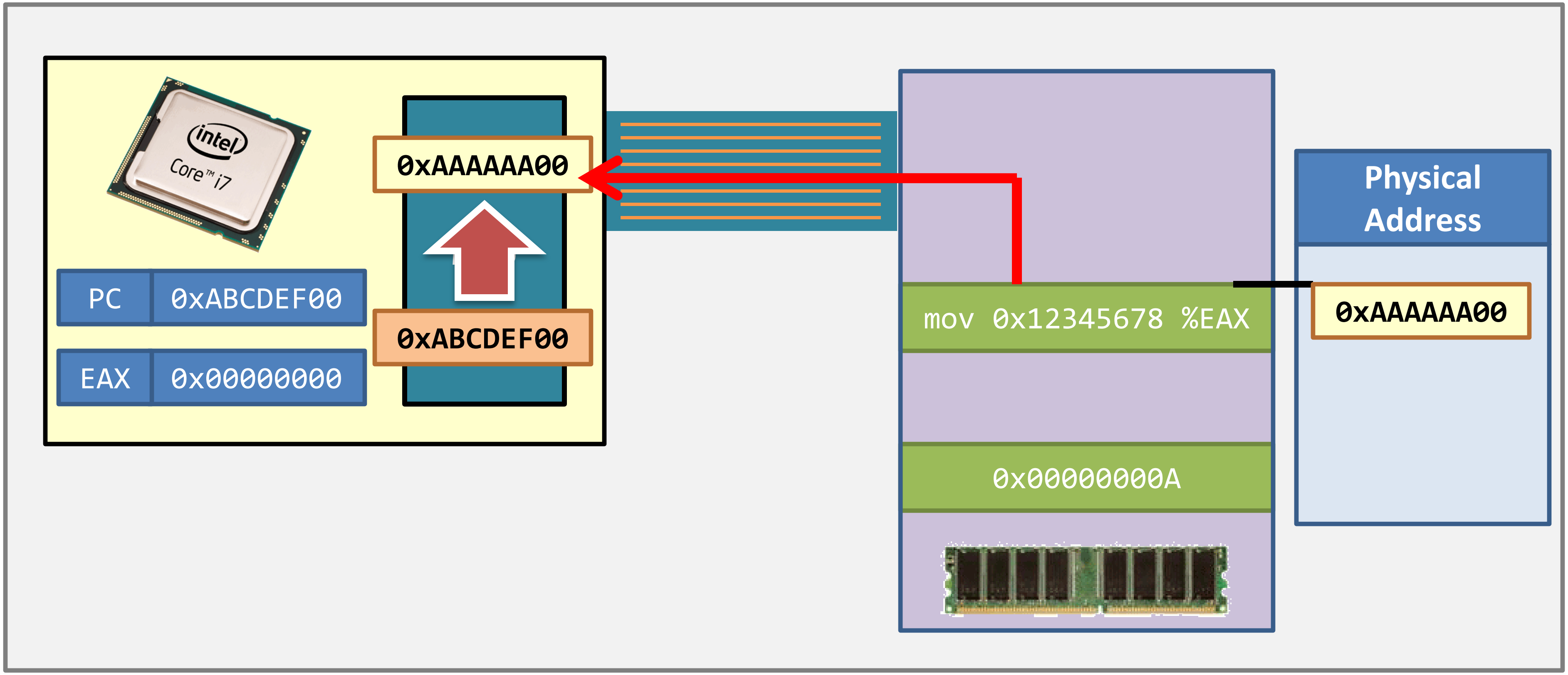
Virtual memory – how does it work?

- Step 1. When CPU wants to fetch an instruction, the **virtual address** is sent to MMU and is translated into a **physical address**.



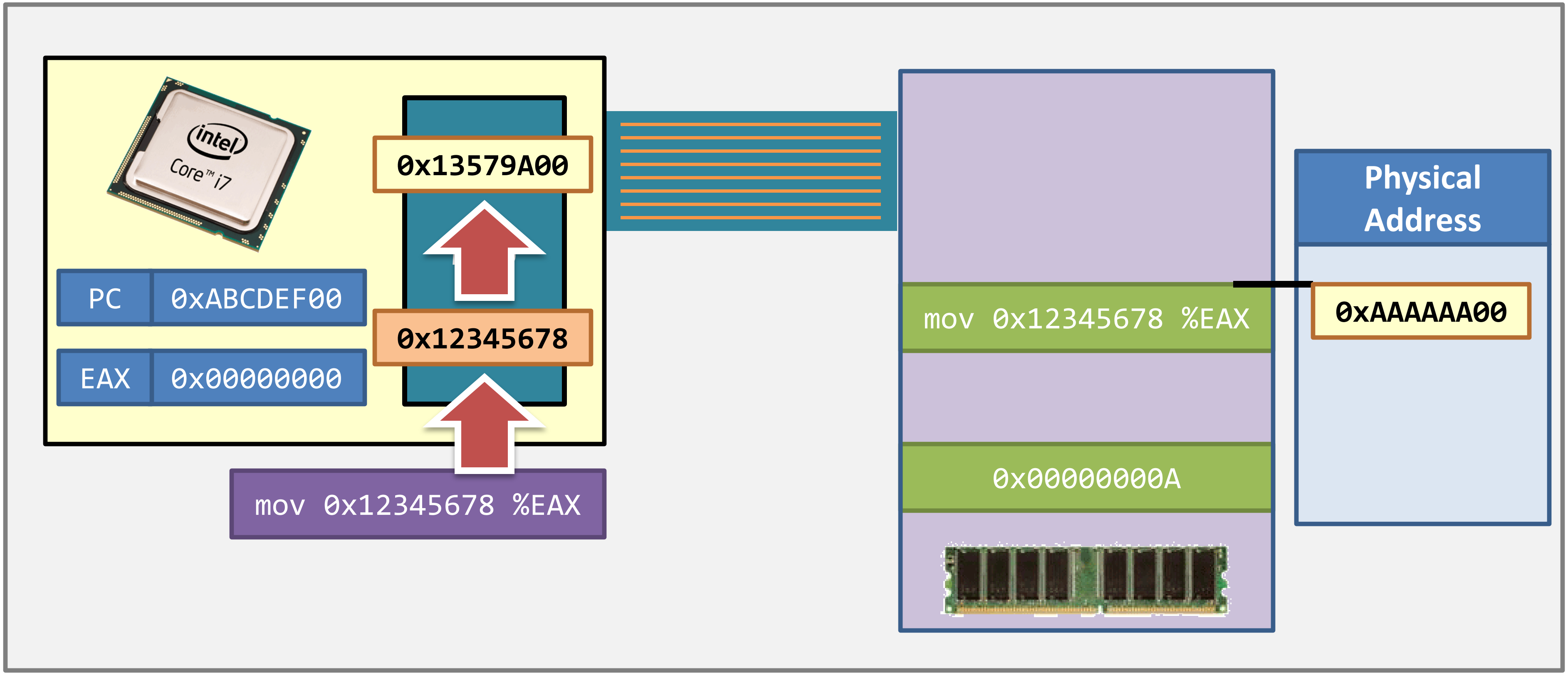
Virtual memory – how does it work?

- Step 2. The memory returns the instruction addressed in physical address.



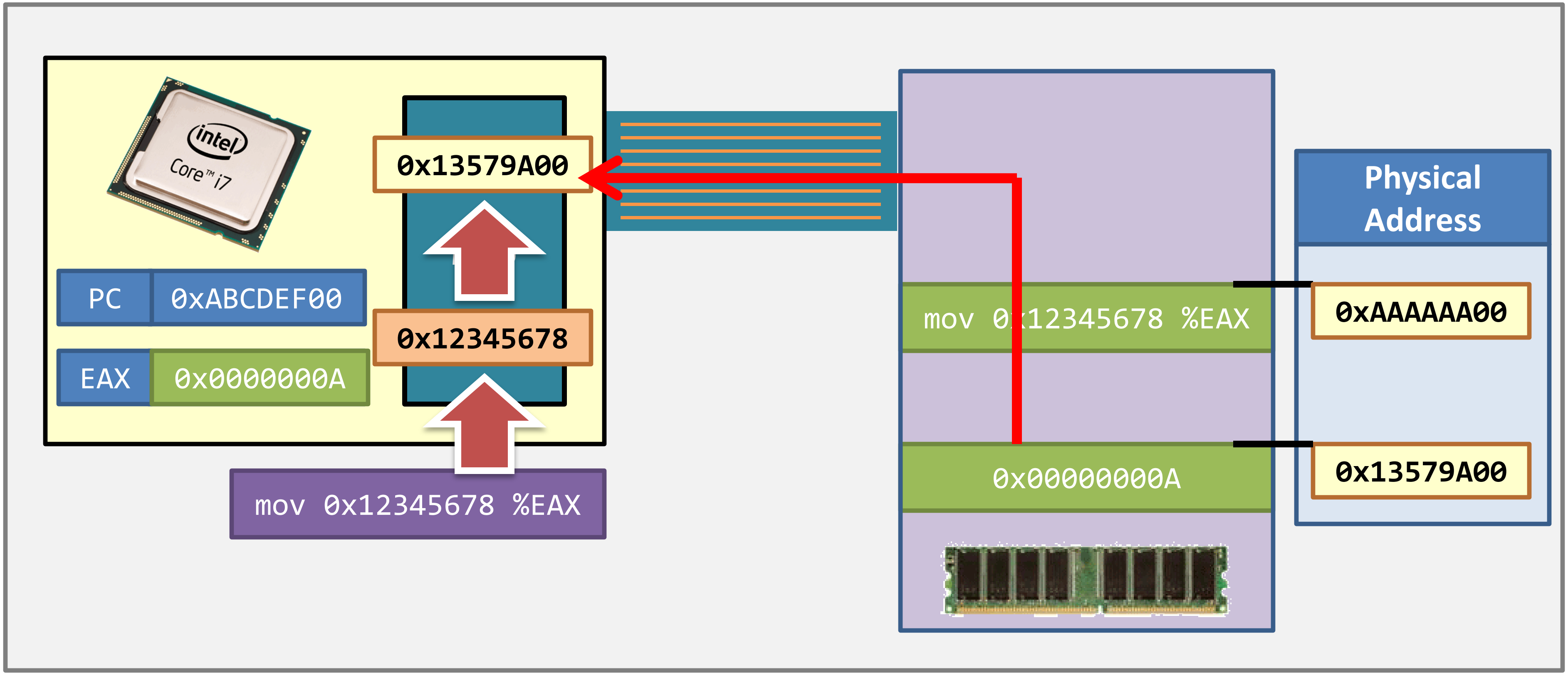
Virtual memory – how does it work?

- Step 3. The CPU decodes the instruction.
  - An instruction **always stores virtual addresses**, but not physical addresses.



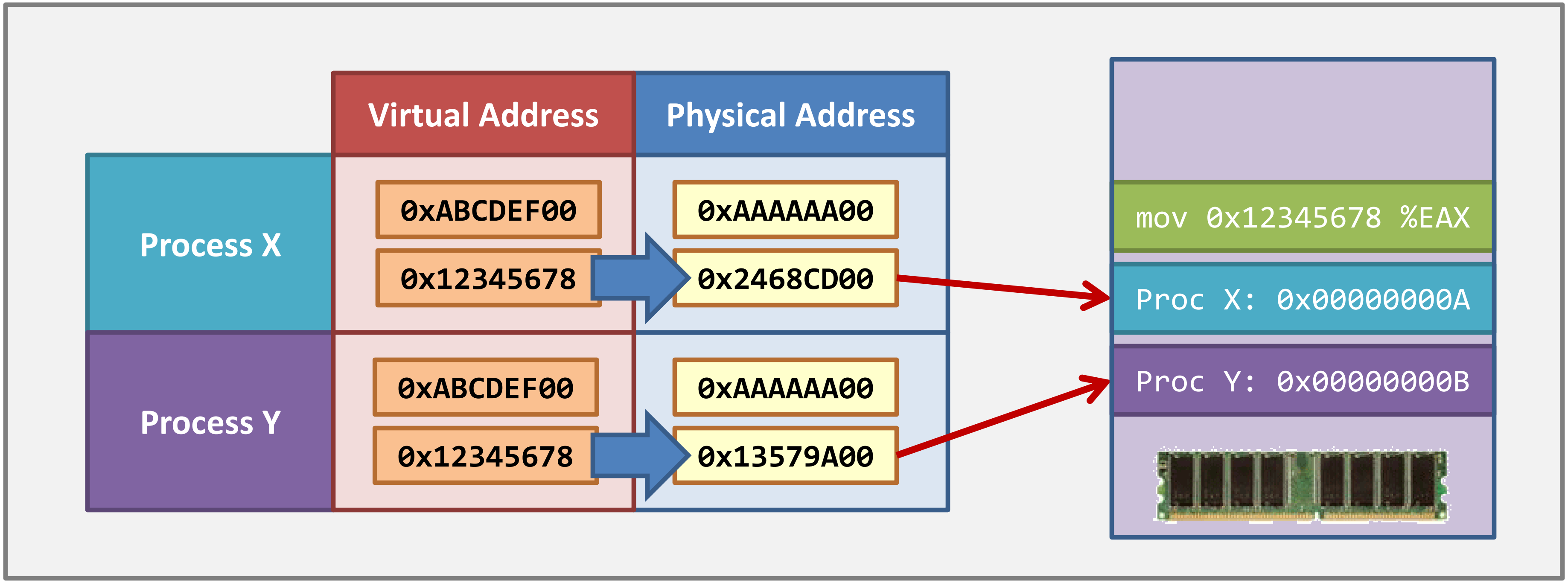
Virtual memory – how does it work?

- Step 4. With the help of the MMU, the target memory is retrieved.



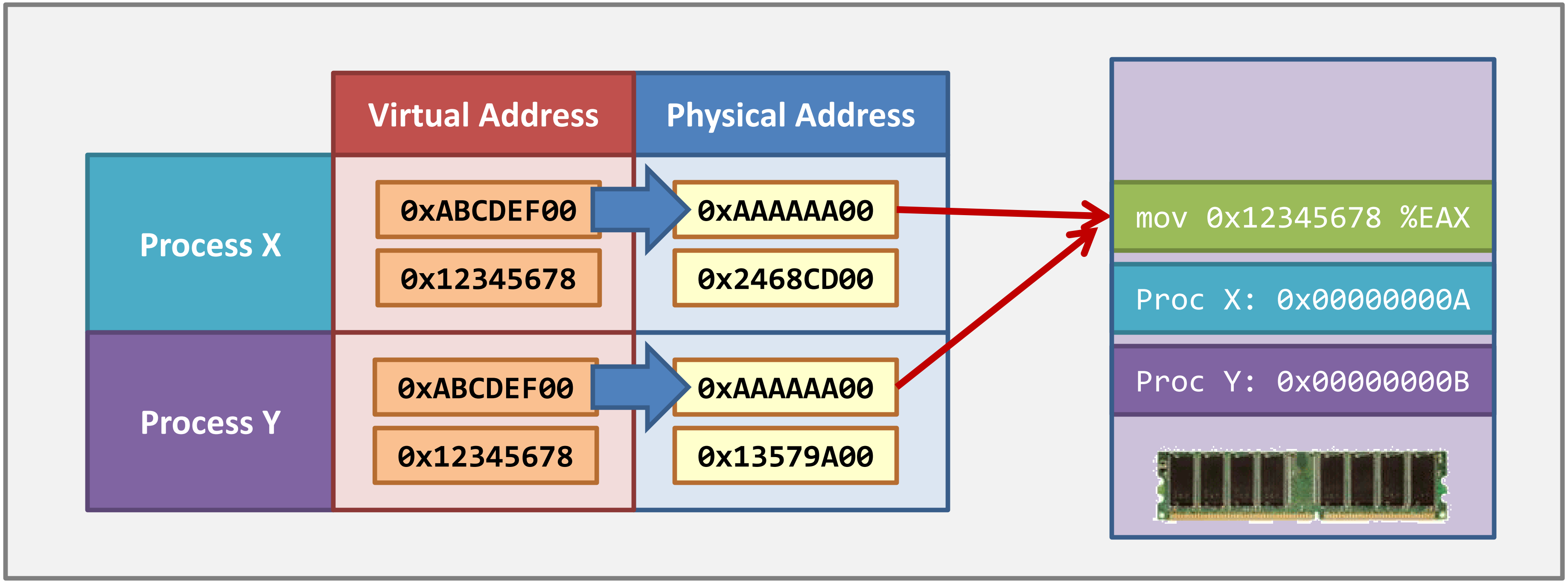
Virtual memory – What is the good?

- **<u>Merit 1.</u>** Although different processes use the same virtual addresses, they may be **translated to different physical addresses**.
  - Recall the "**`pid`**" variable in the example using **`fork()`**.



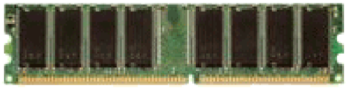
Virtual memory – What is the good?

- **<u>Merit 2.</u>** **Memory sharing** can be implemented!
  - This is how threads share memory!
  - This is how different processes share codes! (HOW?)



Memory Management

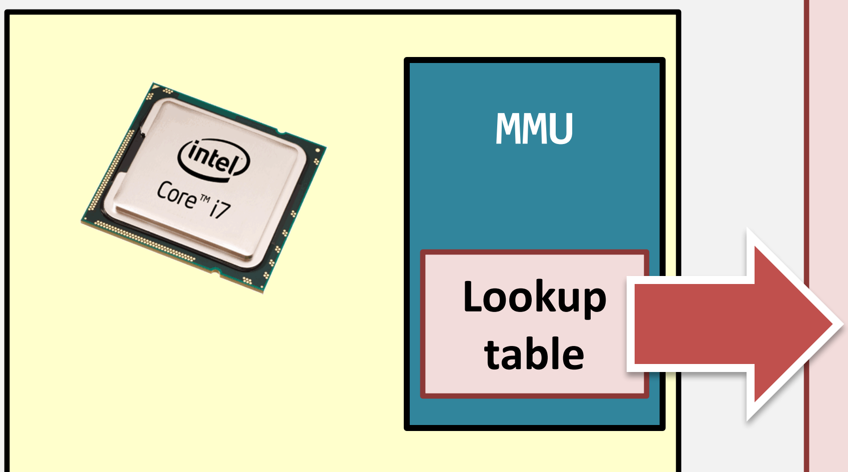
- Virtual memory = CPU + MMU;
- **MMU implementation & paging;**



MMU implementation – a translation table

- So, can translation be done by a **lookup table**?
  - Remember, every process has its own lookup table.

    (Do you remember the reason?)



translation

Virtual address	Physical Address
0x00000000	0x01234567
0x00000001	0x452796AB
....	......
0xFFFFFFFF	0x6714EFD8

Lookup Table internals

MMU implementation – a translation table

- Then, how large is the lookup table?

How many addresses are there? **2^{32}**

How large is an address? **4 bytes**

```
Size of the lookup table =

          Number of addresses
       x  Size of an address
```

2^{32} x 4 bytes = 16 Gbytes

Only this column is stored.

Virtual address	Physical Address
0x00000000	0x01234567
[illegible]	0x452796AB
......	......
0x[illegible]FFF	0x6714EFD8

Lookup Table internals

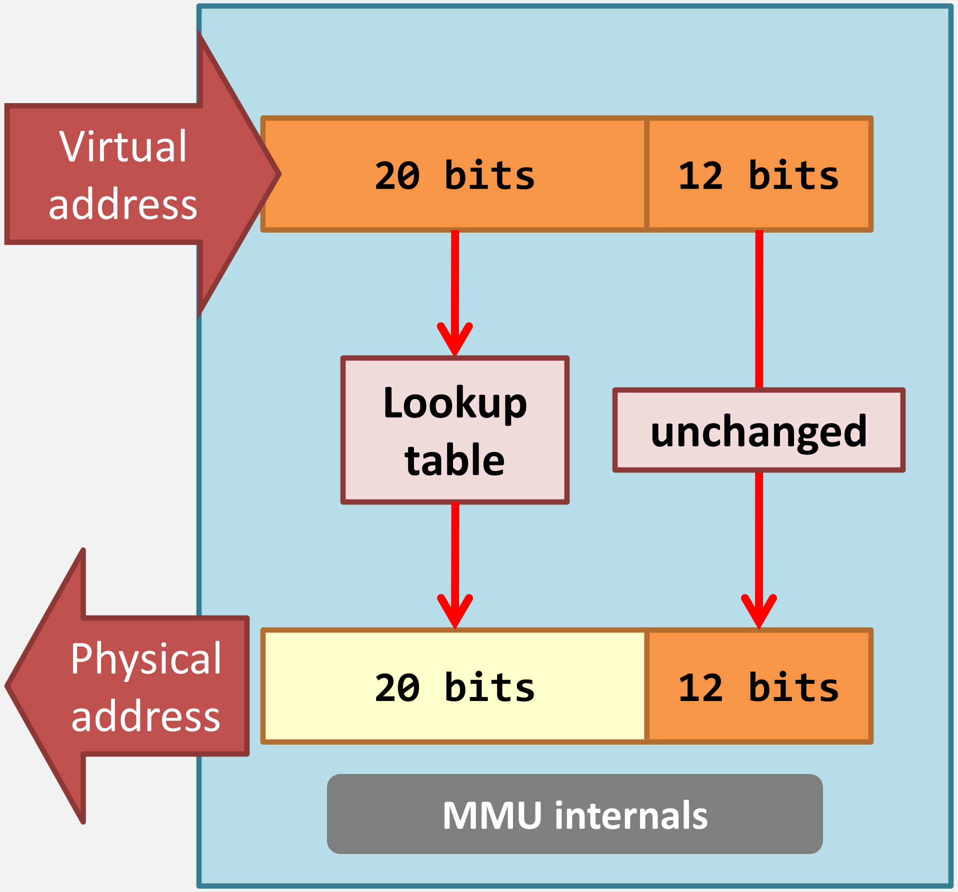
MMU implementation – a partial lookup table

Size of the lookup table =

Number of addresses
x Size of an address

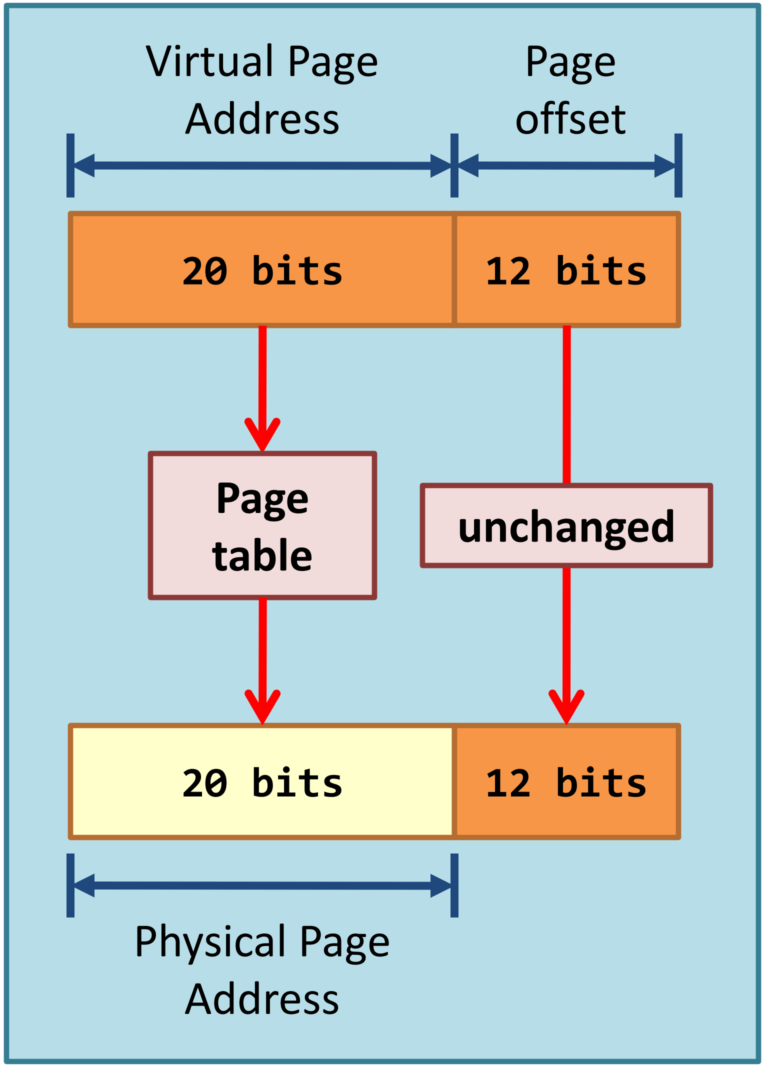
2^{20} x 4 bytes = 4 Mbytes

Note. Every address in a CPU is always of 4 bytes although you only use 20 bits.

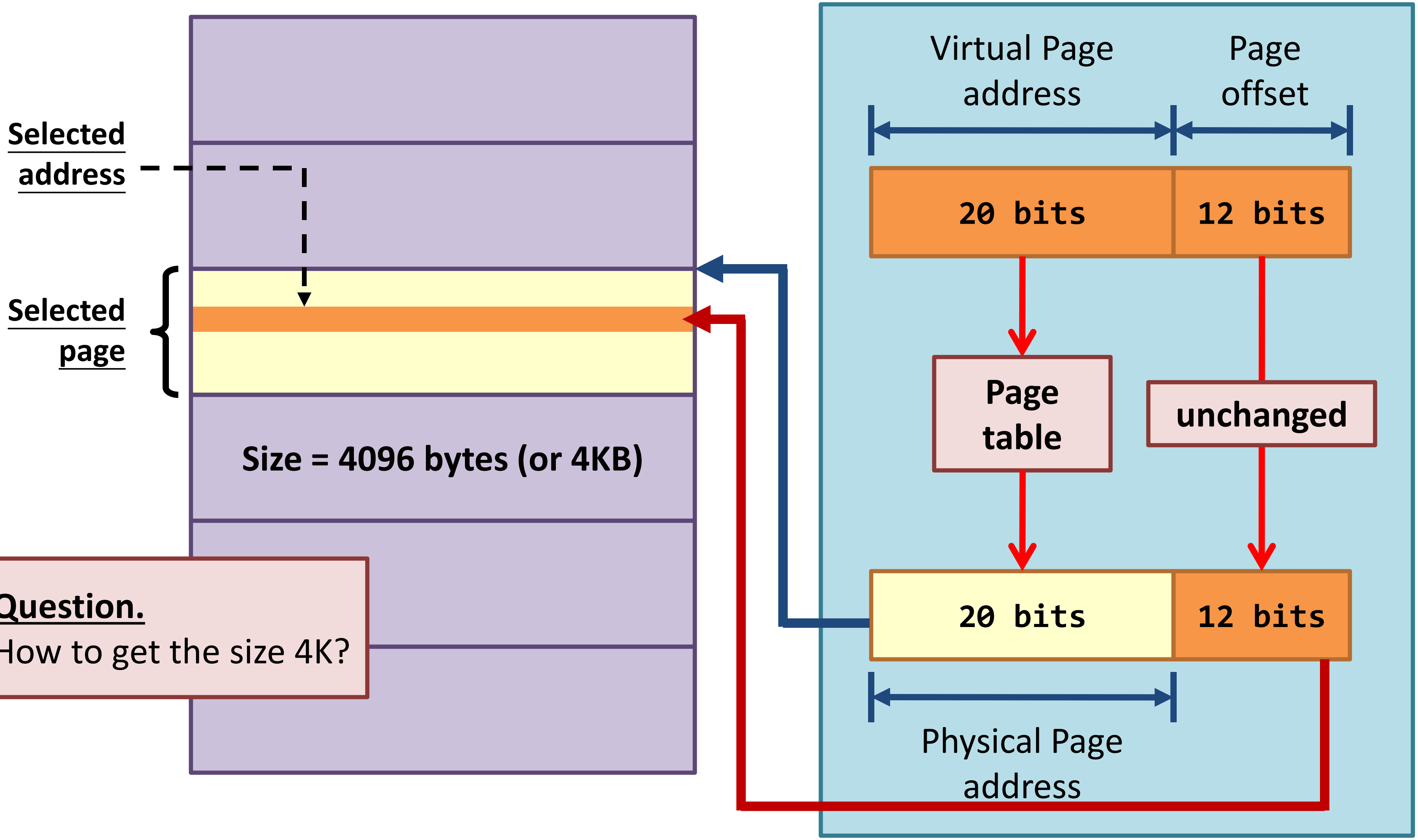


MMU implementation – paging

- This technique is called **paging**.
  - This partitions the memory into fixed blocks called **pages**.
  - The lookup table inside the MMU is now called the **page table**.



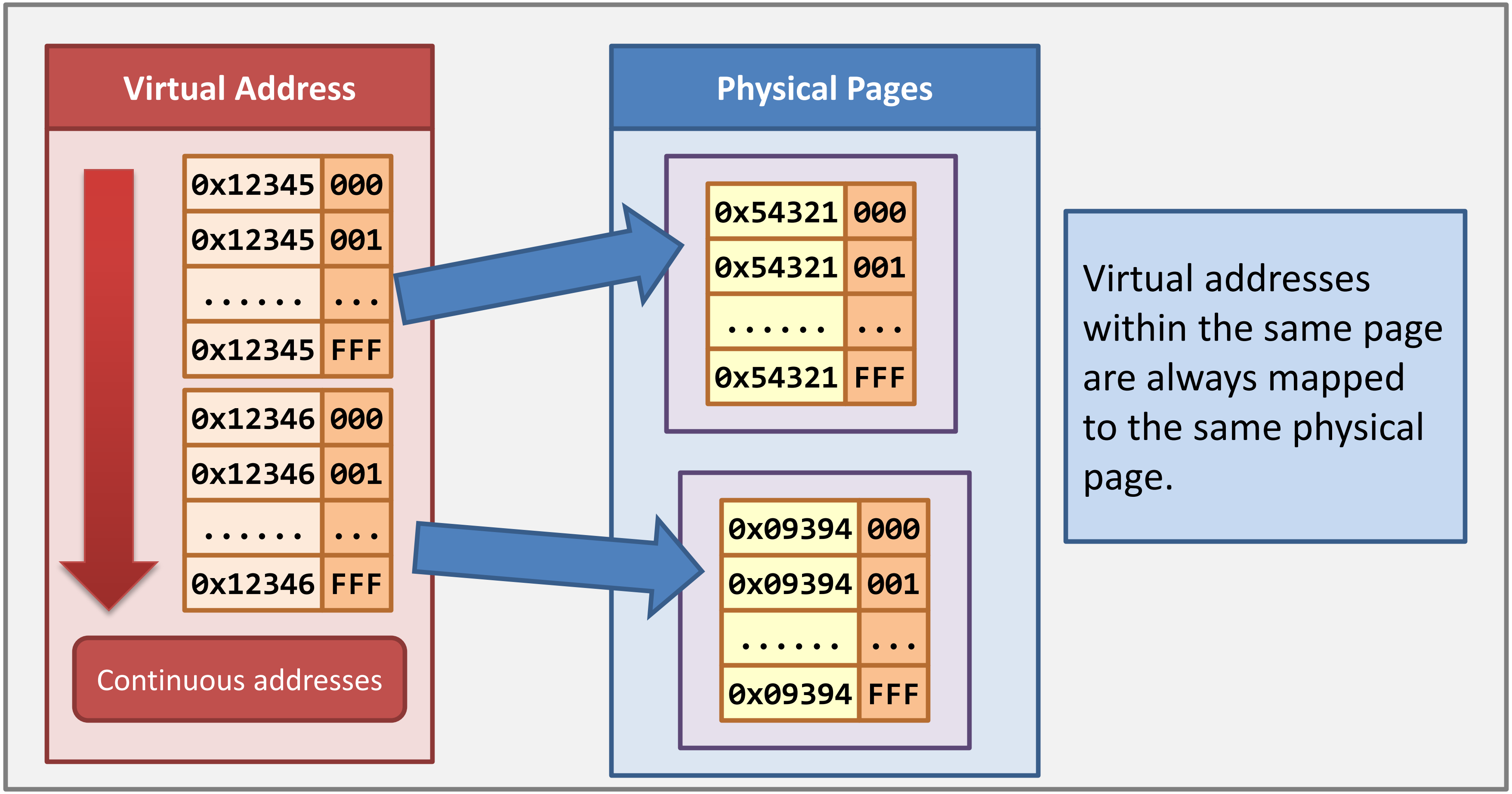
Paging - properties



Question.
How to get the size 4K?

Paging - properties

- Adjacent virtual pages are not guaranteed to be mapped to adjacent physical pages.



Paging – memory allocation

- Believe me or not, a page is the basic unit of memory allocation.

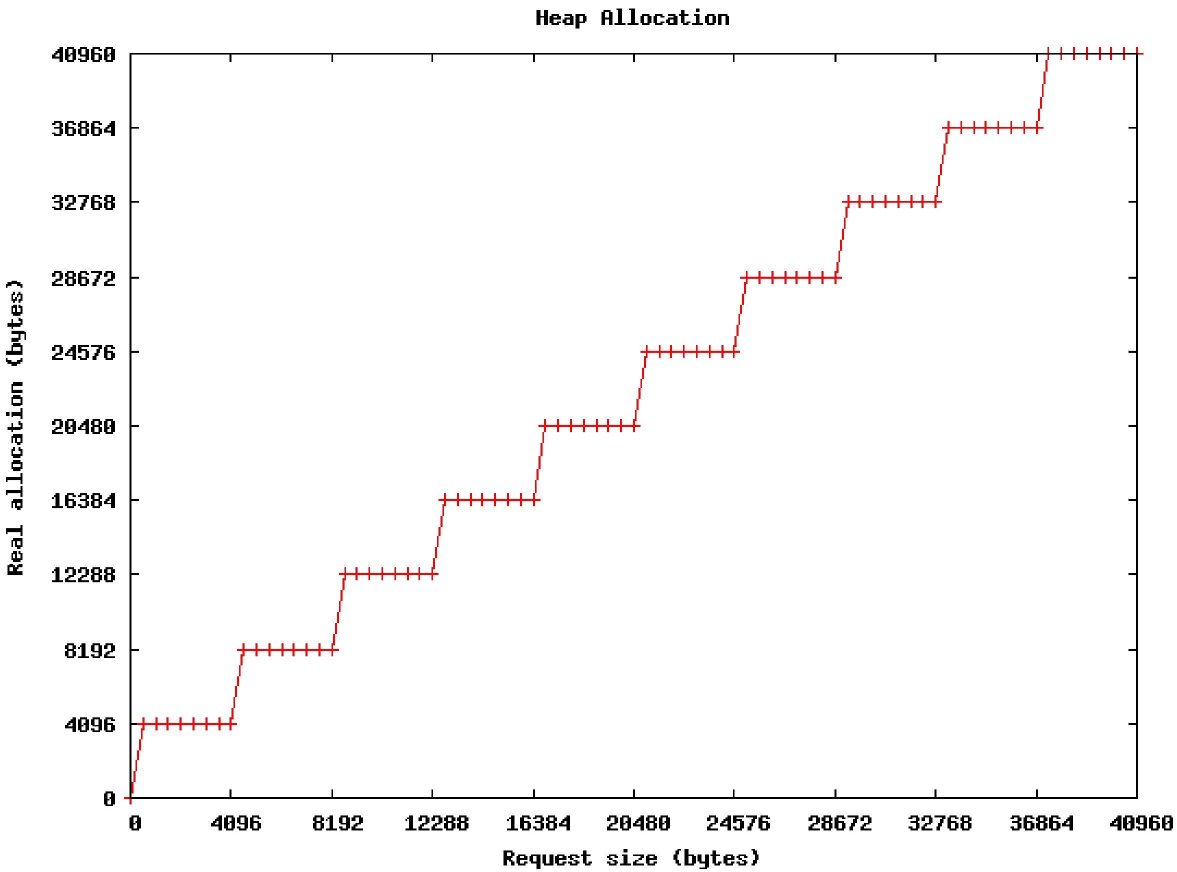
```
char *prev_ptr = NULL;
char *ptr = NULL;

void handler(int sig) {
    printf("Page size = %d bytes\n",
            (int) (ptr - prev_ptr));
    exit(0);
}
int main(int argc, char **argv) {
    char c;
    signal(SIGSEGV, handler);
    prev_ptr = ptr = sbrk(0);  // find the heap’s start.
    sbrk(1);                   // increase heap by 1 byte?
    while(1)
        c = *(++ptr);
}
```

```
[examples@3150] cat test_page_size.c
```

Paging – memory allocation

- Believe me or not, a page is the basic unit of memory allocation.



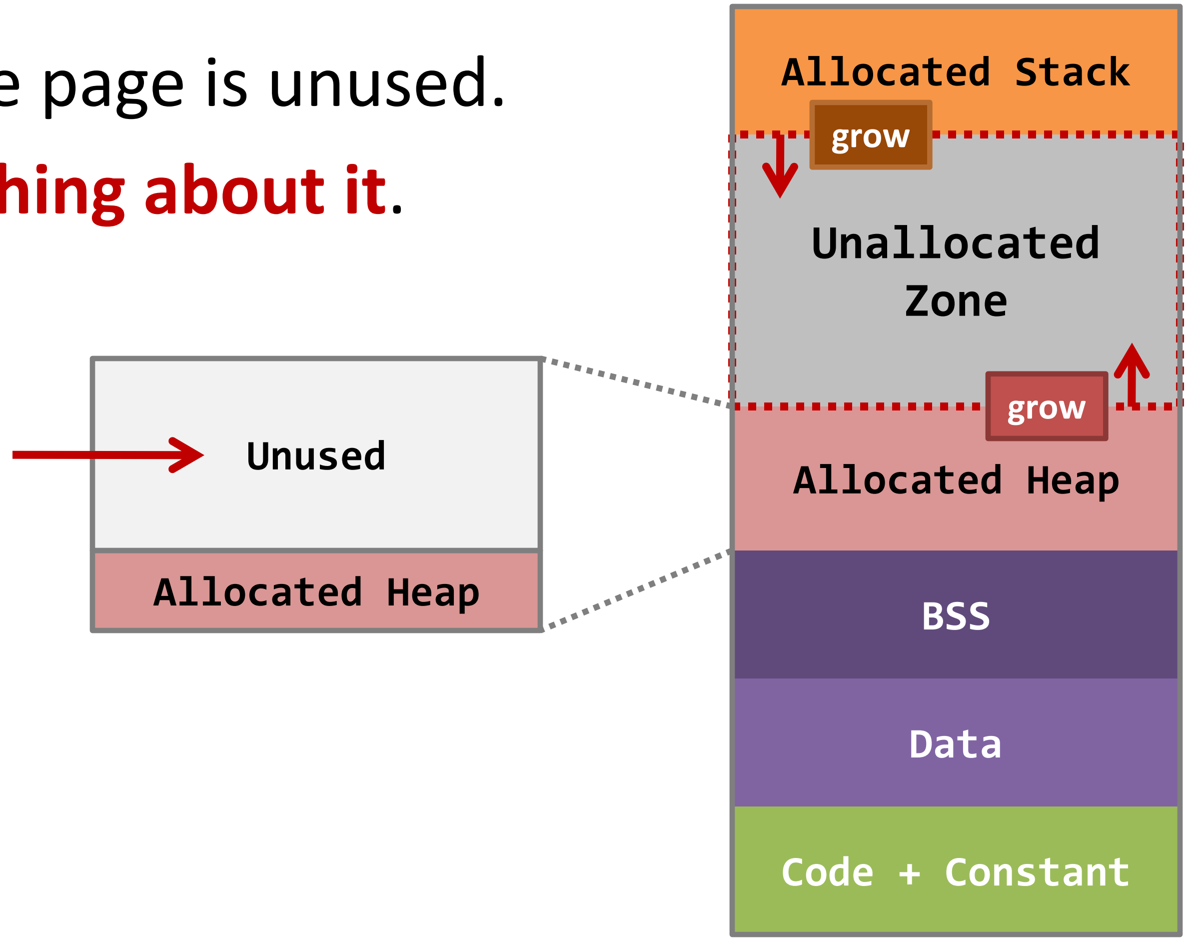
The allocation is in a **page-by-page** manner.

The same case for the growth of the stack.

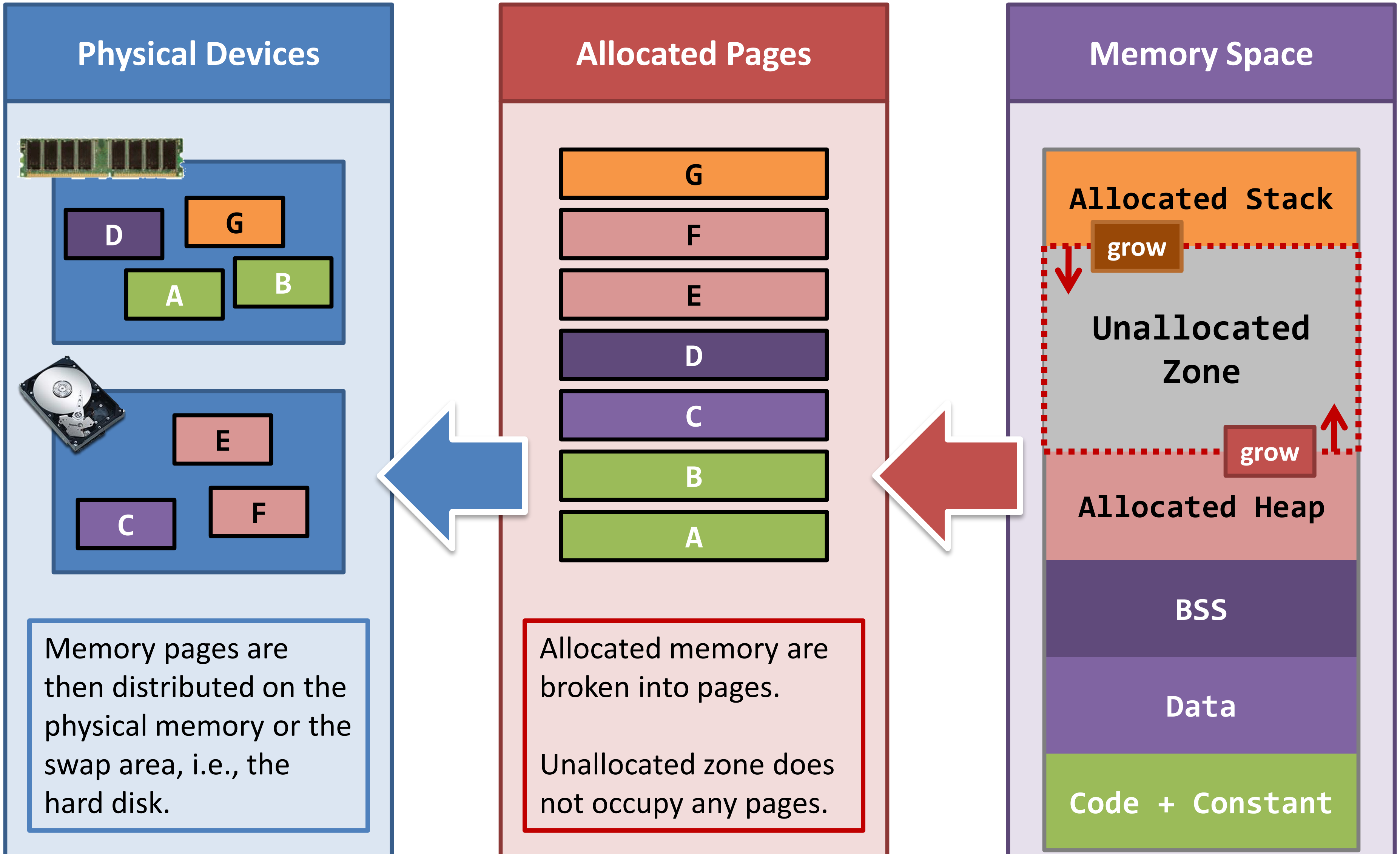
Paging – internal fragmentation

- The minimum allocation unit is 4,096 bytes.
  - But, the process cannot use that much.
  - So, the rest of the page is unused.
  - **We could do nothing about it.**

<u>Internal fragmentation</u> means space is avoidably wasted when allocation is done in a block-by-block manner.



Paging – putting it together



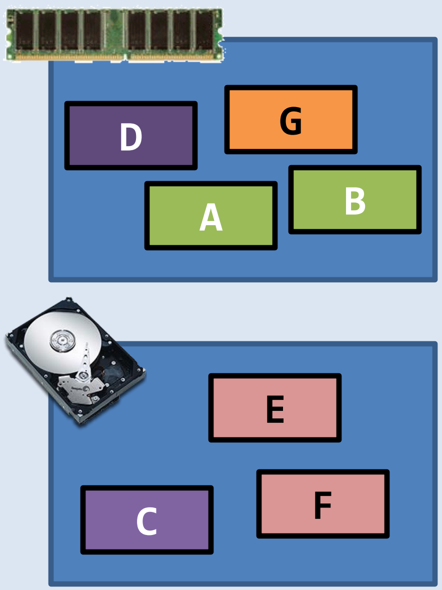
Memory pages are then distributed on the physical memory or the swap area, i.e., the hard disk.

Allocated memory are broken into pages.

Unallocated zone does not occupy any pages.

Paging – page table design

Physical Devices



Memory pages are then distributed on the physical memory or the swap area, i.e., the hard disk.

- So, next waves of questions are:
  - Who can tell which virtual page is allocated?
  - Who can tell which page is on which device?
- Those questions can be answered by the design of the page table.

Paging – page table design

Page Table of Process A			
Virtual Page #	**Permission**	**Valid-invalid bit**	**Frame #**
A	`rwx-`	1	0
B	NIL	0	NIL
C	`r--s`	1	2
D	NIL	0	NIL
...	...	...	...

The physical memory is just **an array of frames**. The size of a frame is 4KB.

This row means the virtual page "A" is mapped to the physical frame "0".

This row, with **NIL**, means the virtual page "D" is **not allocated**.

Remember, the entire 4G memory zone is usually not fully utilized.

For the sake of convenience, we don't use addresses here. Also, this column **is not stored in the page table**.

Paging – page table design

Page Table of Process A			
Virtual Page #	**Permission**	**Valid-invalid bit**	**Frame #**
A	`rwx-`	1	0
B	`NIL`	0	`NIL`
C	`r--s`	1	2
D	`NIL`	0	`NIL`
...	...	...	...

This bit is to tell the CPU whether **this row is valid or not**.

If the row is invalid, it means that **the virtual page is not in the memory**.

Note. This is not the same as an unallocated page.

```
1 – valid, in memory.
0 – invalid, not in memory.
```

Paging – page table design

Page Table of Process A

Virtual Page #	Permission	Valid-invalid bit	F
A	`rwx-`	1	
B	`NIL`	0	
C	`r--s`	1	
D	`NIL`	0	
...	...	...	

`s – means sharable.`

E.g., this is how the CPU check if you can write to a memory zone!

When a virtual address is translated to an **unallocated frame**…

OR

When you write to **read-only pages**…

OR

When you try to execute a non-executable pages…

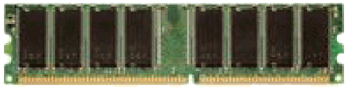
SEGMENATION FAULT!!

Paging – summary

- Virtual memory (VM) is just a table-lookup implementation. The specials about VM are:
  - The table-lookup process is implemented inside the CPU, i.e., a hardware solution.
  - **Each process should have its own page table.**
    - Do you think that the virtual address **0x12345678** of Process A and Process B should be pointing to the same page?
    - That's how the VM support is to <u>teach the CPU to understand processes</u>.

Memory Management

- Virtual memory = CPU + MMU;
- MMU implementation & paging;
- **Demand paging;**



Memory / page allocation?

- As a matter of fact, we haven’t covered how the memory of a process grows…
  - Well…only the stack and the heap will grow:
    - (1) calling **`brk()`**, i.e., the heap grows;
    - (2) calling **nested function calls**, i.e., the stack grows;

Memory / allocation – demand paging

- The reality is: allocation is done in a **lazy** way!
  - The system only **says** that the memory is allocated.
  - Yet, it is **not really allocated** until you access it.

```
#define BUF_SIZE  512 * 1024
void re() {
    char buf[BUF_SIZE];
    while( getchar() != '\n' );
    memset(buf, 0, sizeof(buf));
    while( getchar() != '\n' );
    re();
}

int main(void) {
    re();
    return 0;
}
```

This statement does not involve any memory access.

So, the virtual address space is allocated, but **the page is not allocated yet**.

This statement really accesses the "*allocated*" memory.

So, this statement really **asks the system** to allocate memory.

```
[examples@3150] cat grow_stack.c
```

Memory / allocation – demand paging

- How about the heap?
  - Again, experiment with the program "**top**".

```c
#define ONE_MEG (1024 * 1024)
#define COUNT   1024

int main(void) {
    int i;
    char *ptr[COUNT];
    for(i = 0; i < COUNT; i++)
        ptr[i] = malloc(ONE_MEG);

    for(i = 0; i < COUNT; i++) {
        while(getchar() != '\n');
        memset(ptr[i], 0, ONE_MEG);
    }
}
```

As a matter of fact, `malloc()` does not involve any memory allocation, only involving **the allocation of the virtual address page.**

So, this loop is only for enlarging the virtual page allocation.

This statement really accesses the "*allocated*" memory.

So, this statement really asks the **asks the system** to allocate memory.

```
[examples@3150] cat grow_heap.c
```

Demand paging – illustration.

Assumption: 1 process only.

- Suppose that a process initially has 4 page frames.
  - Let's consider the "**`grow_heap.c`**" example.
  - We are now in the **`memset()` for-loop** in Lines 10 - 13.

OS kernel

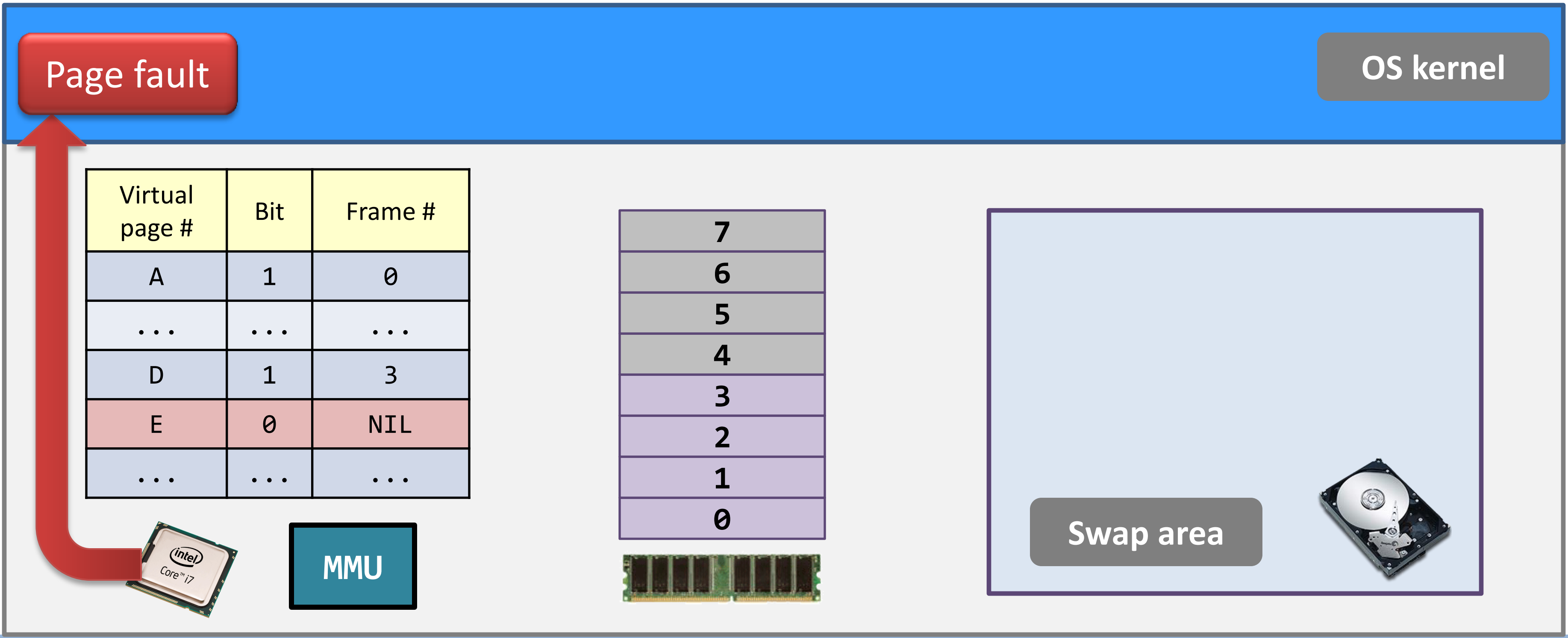
Virtual page #	Bit	Frame #
A	1	0
...	...	...
D	1	3
E	0	NIL
...	...	...

MMU

Demand paging – illustration.

Assumption: 1 process only.

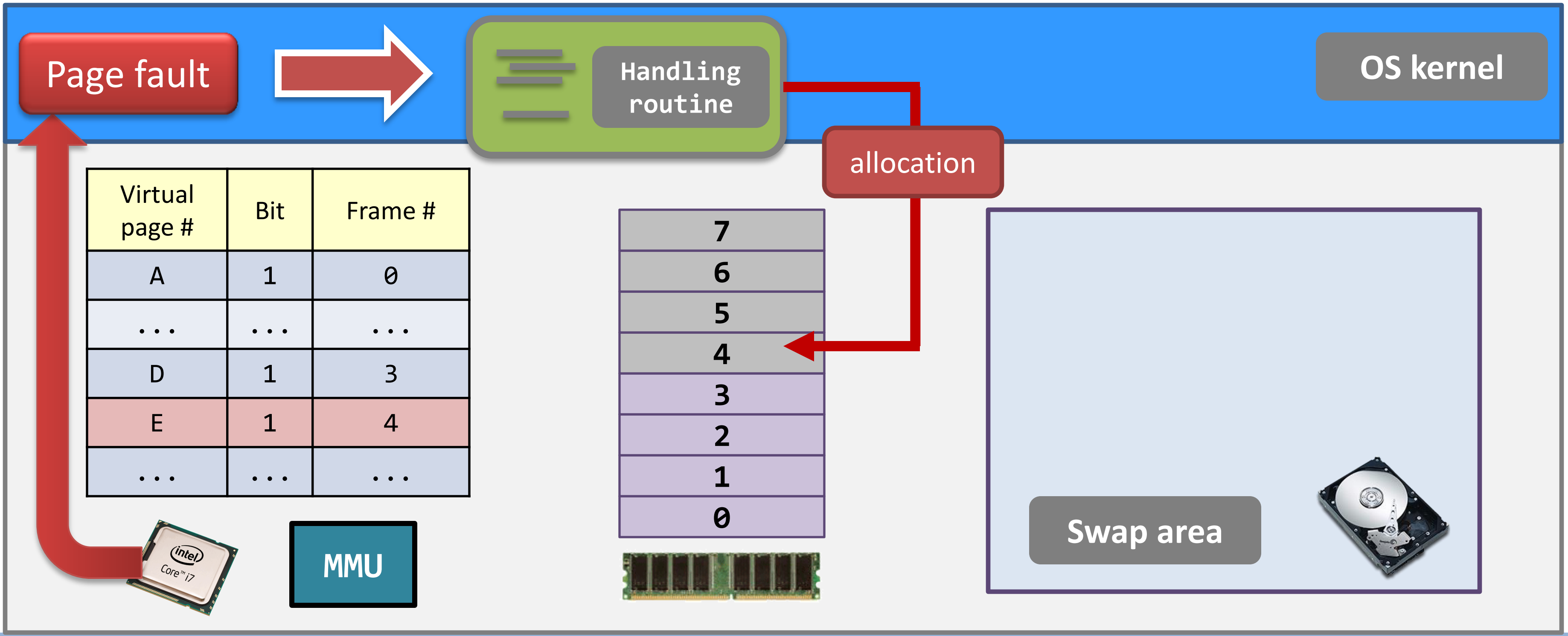
- When **`memset()`** runs,
  - the MMU finds that a **virtual page involved is invalid**,
  - the CPU then generates an interrupt called **page fault**.



Demand paging – illustration.

Assumption: 1 process only.

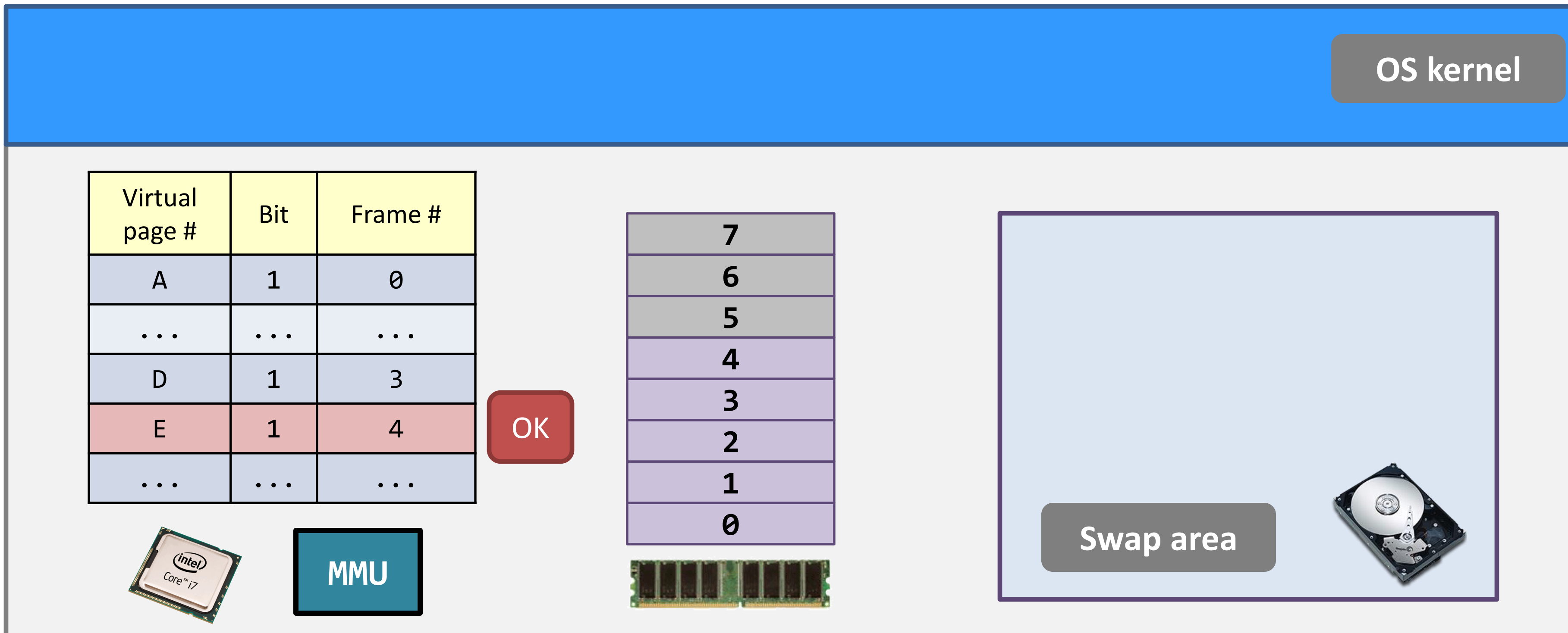
- The **page fault handling routine** is running:
  - The kernel knows the page allocation for all processes.
  - It allocates a memory page for that request.
  - Last, the **page table entry** for Page E is updated.



Demand paging – illustration.

Assumption: 1 process only.

- The routine finishes and the **`memset()`** statement **is restarted**.
  - Then, no page fault will be generated until the next unallocated page is encountered.

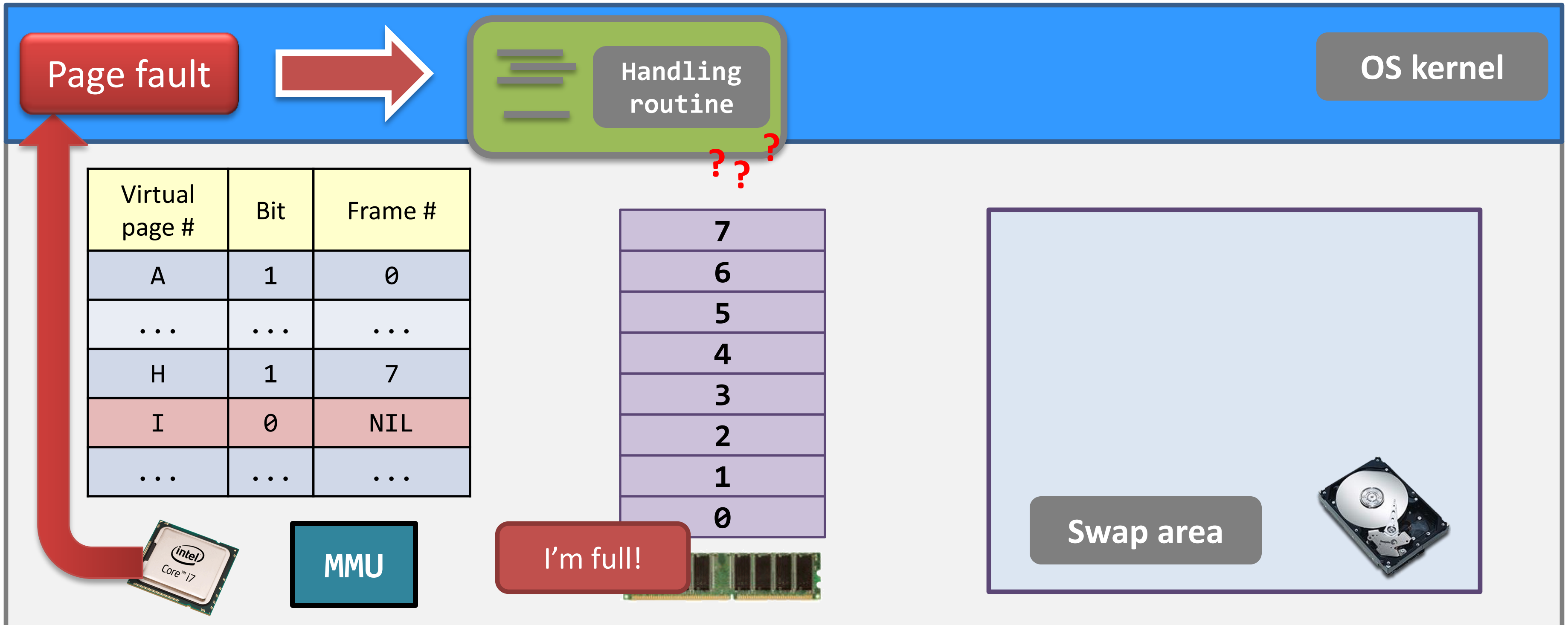


Virtual page #	Bit	Frame #
A	1	0
...	...	...
D	1	3
E	1	4
...	...	...

Demand paging – illustration.

Assumption: 1 process only.

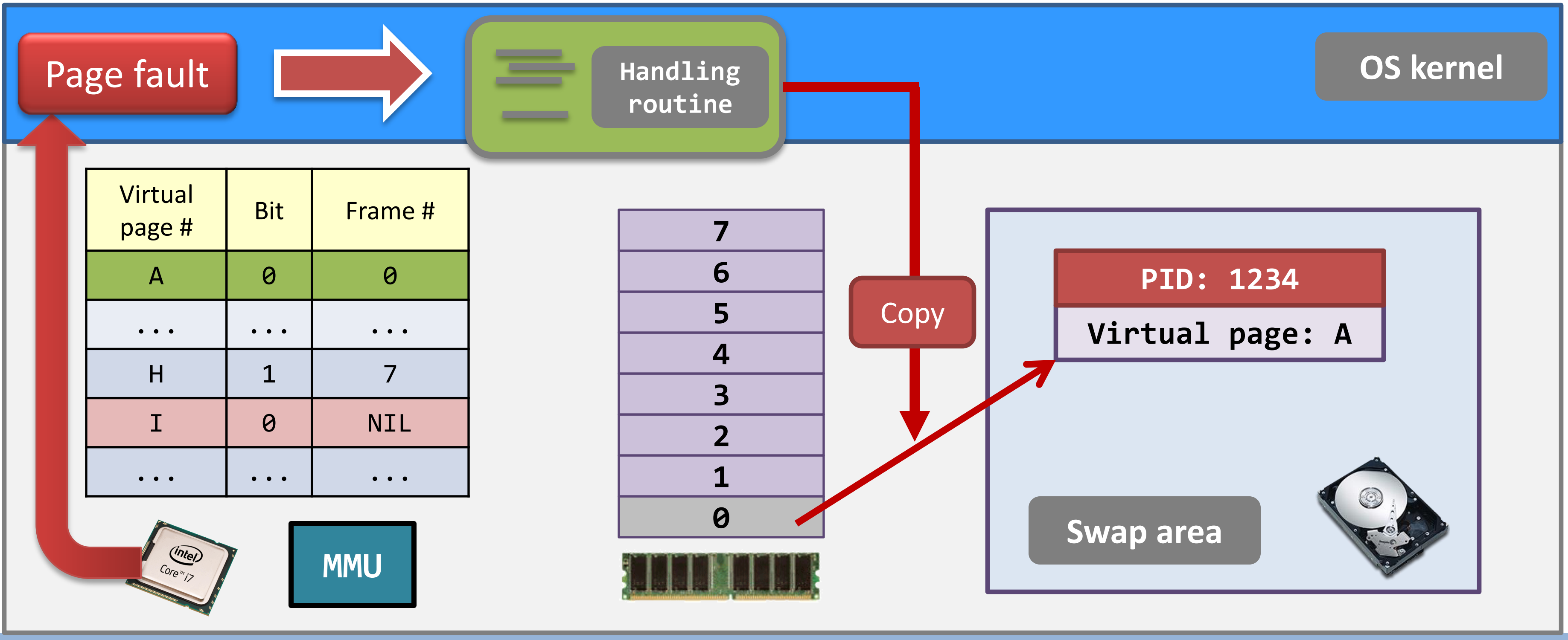
- So, how about the case when the routine finds that **all frames are allocated**?
  - Then, we need the help of the **swap area**.



Demand paging – illustration.

Assumption: 1 process only.

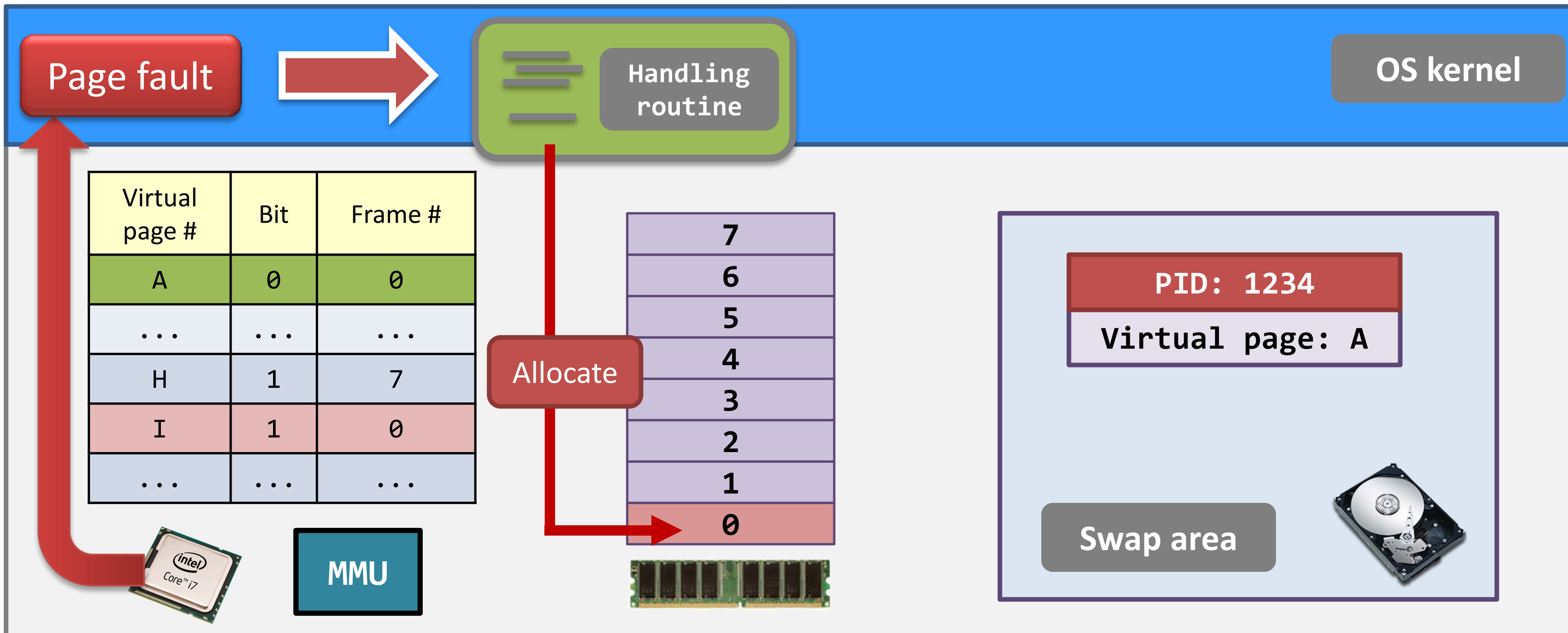
- Using the swap area:
  - Step (1) Select a <u>**victim virtual page**</u> and copy the victim to the swap area.
    - Now, <u>Frame 0 is a free frame</u> and <u>the bit for Page A is 0</u>.



Demand paging – illustration.

Assumption: 1 process only.

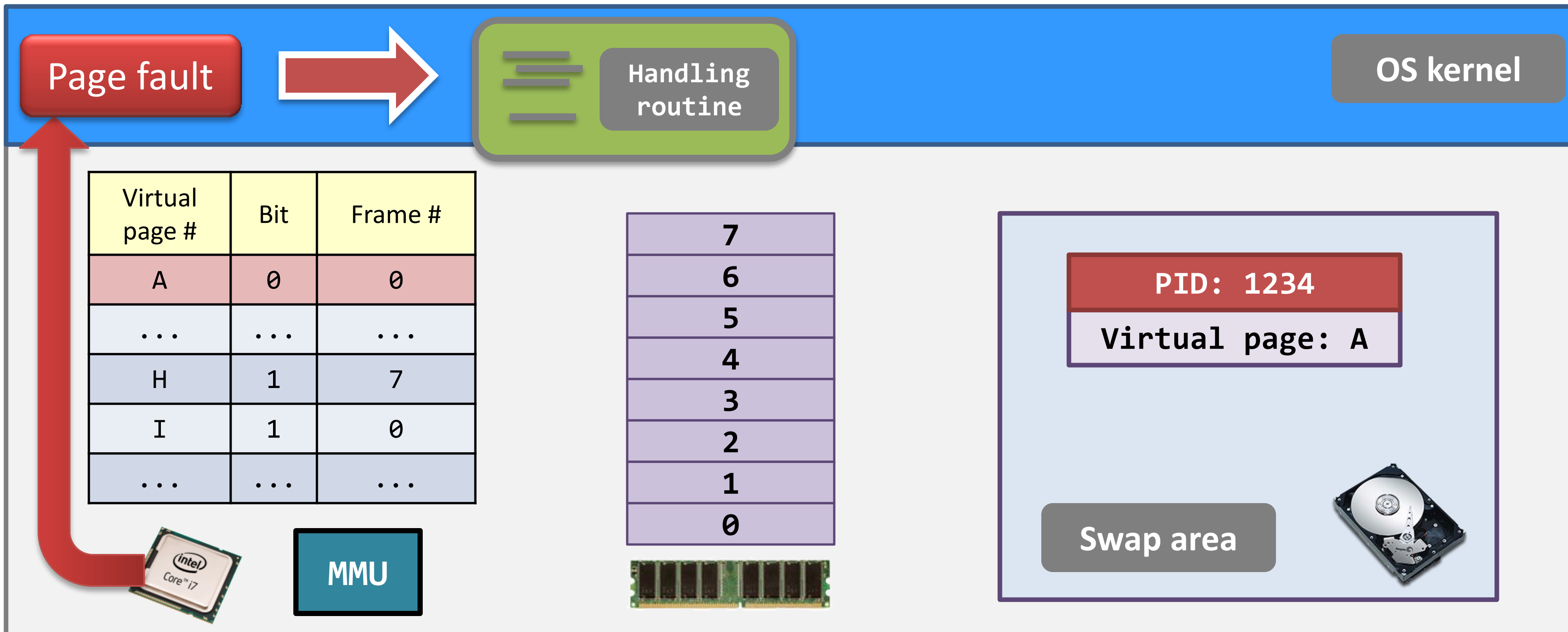
- Using the swap area:
  - Step (2) Allocate the free frame to the new frame allocation request.
    - Now, Page I takes Frame 0.



Demand paging – illustration.

Assumption: 1 process only.

- How about **virtual page A** is accessed again?
  - Of course, a page fault is generated, and
  - steps similar to the previous case takes place.

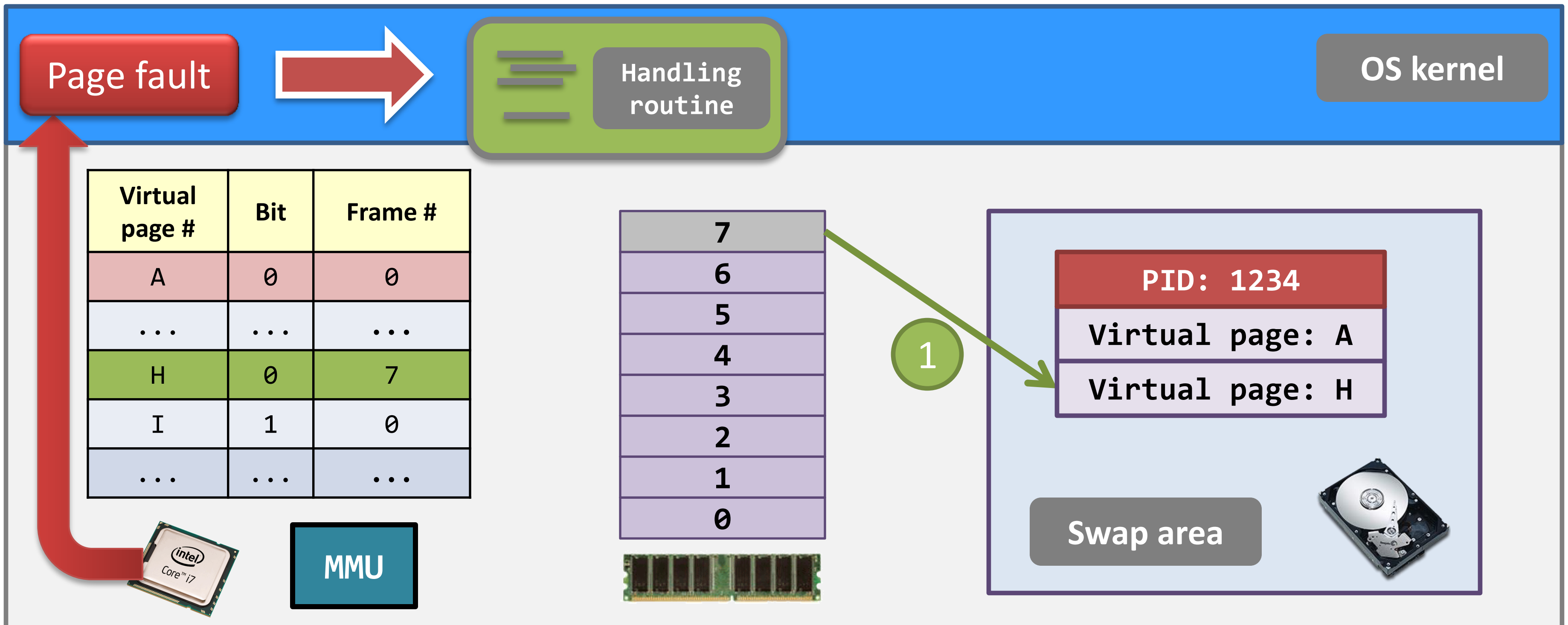


Virtual page #	Bit	Frame #
A	0	0
...	...	...
H	1	7
I	1	0
...	...	...

Demand paging – illustration.

Assumption: 1 process only.

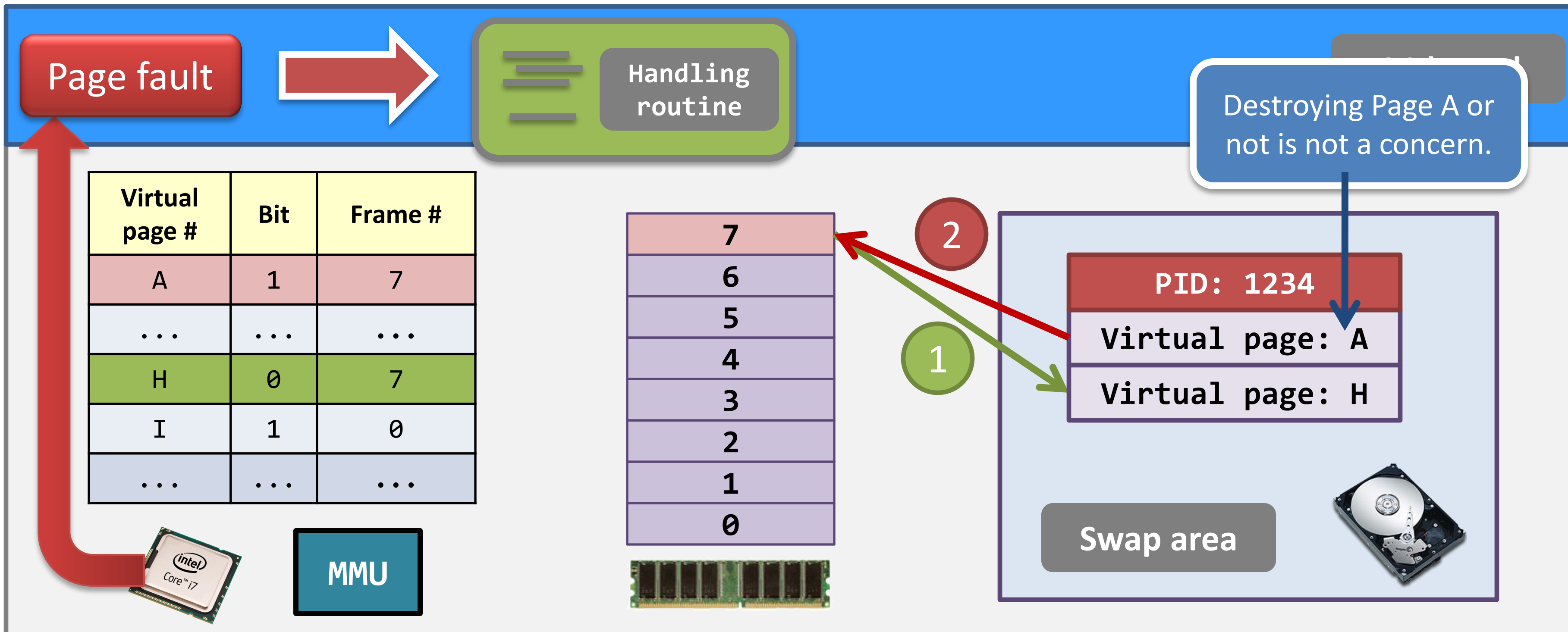
- Step (1) Select a victim virtual page and copy the victim to the swap area.
  - Now, <u>Frame 7 is a free frame</u> and <u>the bit for Page H is 0</u>.



Demand paging – illustration.

Assumption: 1 process only.

- Step (2) Allocate the free frame with the virtual page in the swap area.
  - Now, Page A takes Frame 7 and the bit for Page A is 1.



Virtual page #	Bit	Frame #
A	1	7
...	...	...
H	0	7
I	1	0
...	...	...

Real OOM – code

```
#define ONE_MEG  1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        memset(ptr, 0, ONE_MEG);
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

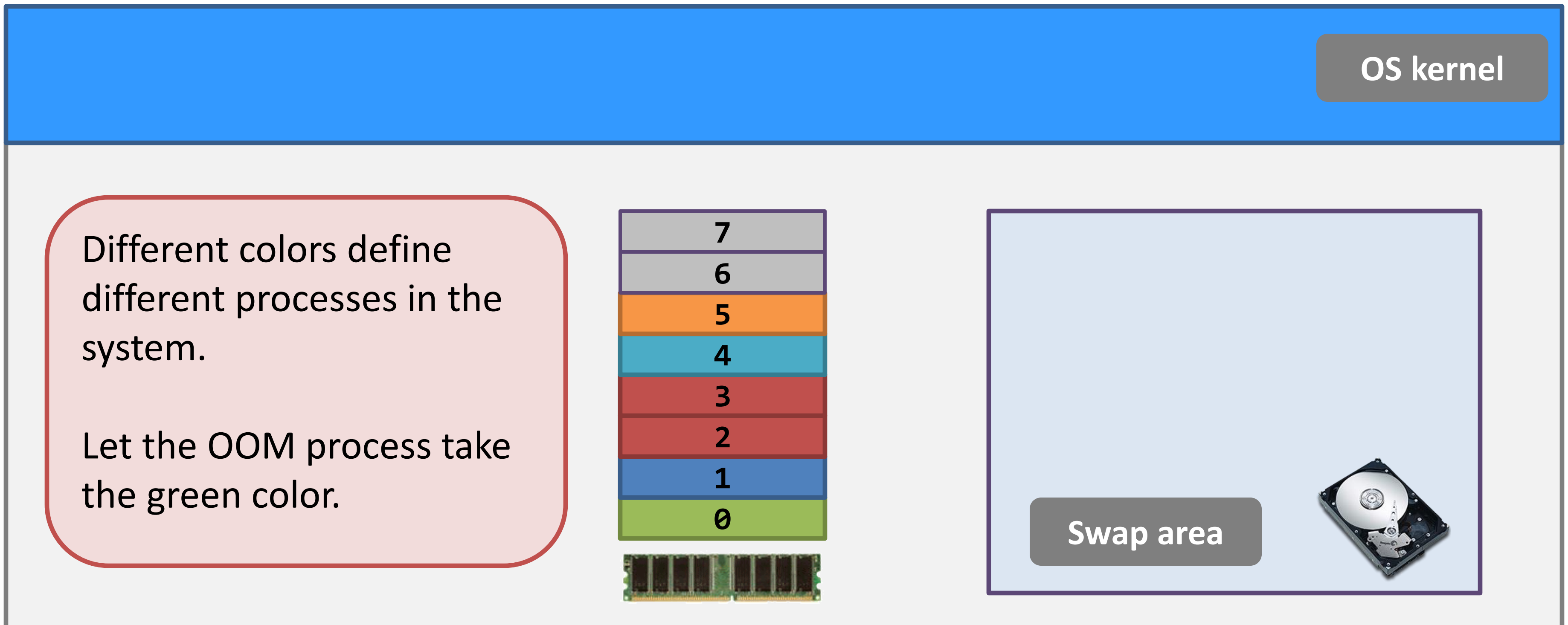
Warning #1. Don't run this program on any department's machines.

Warning #2. Don't run this program when you have important tasks running at the same time.

Real OOM – illustration

`oom_v3: running`

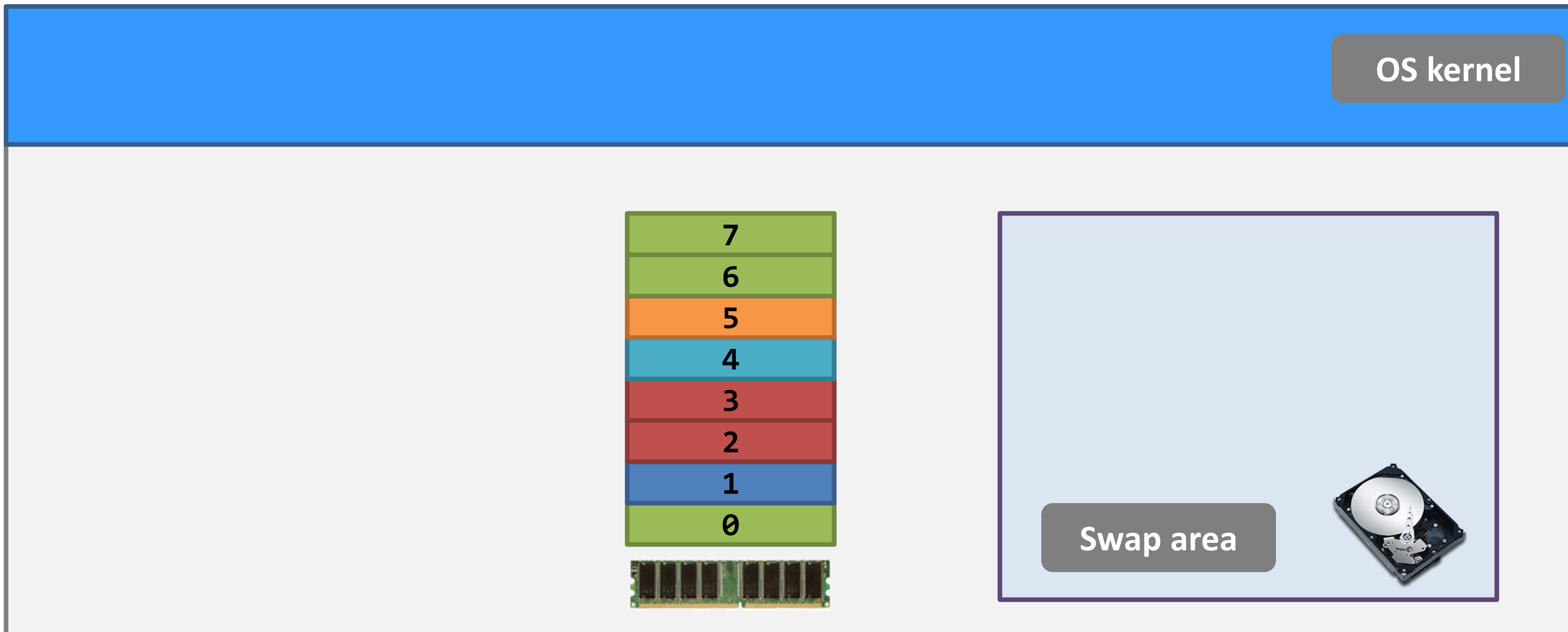
- So, what will happen when the real OOM program is running?
  - Suppose the OOM program has just started with **only one page allocated**. (Illustration only! *Not possible! Why?*)



Real OOM – illustration

`oom_v3: running`

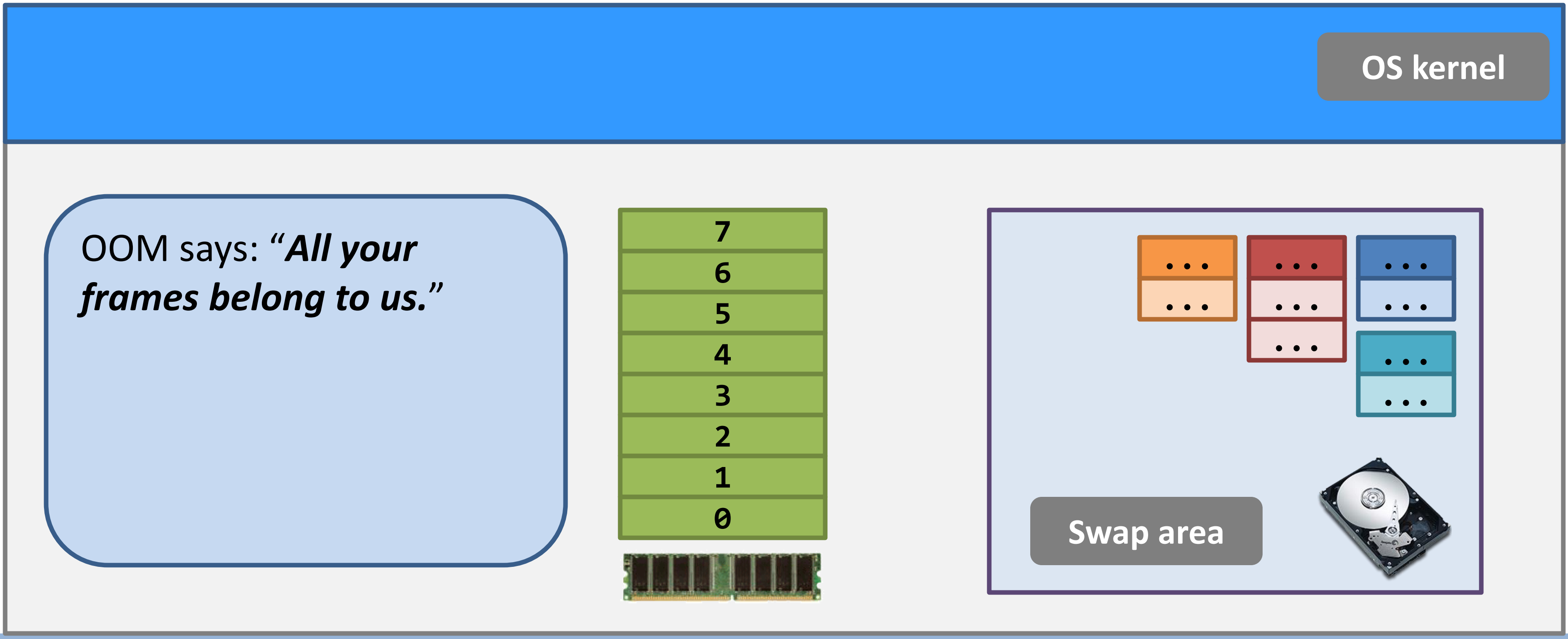
- OOM is running…
  - **1st stage**. The **free memory frames** are the first zone that the process has conquered.
  - All other processes could hardly allocate pages.



Real OOM – illustration

`oom_v3: running`

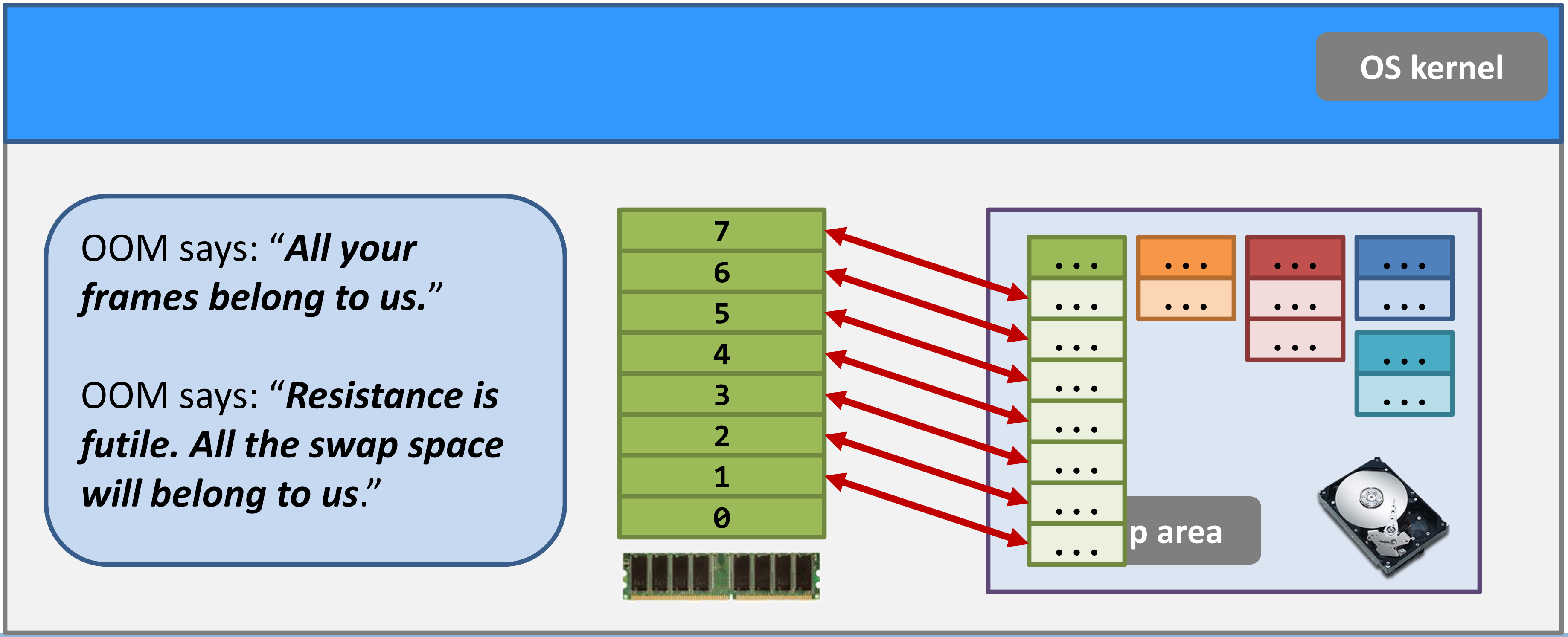
- OOM is running…
  - **2nd stage**. Occupied memory frames are the next zone that the process conquers.
  - **Disk activity flies high!**



Real OOM – illustration

`oom_v3: running`

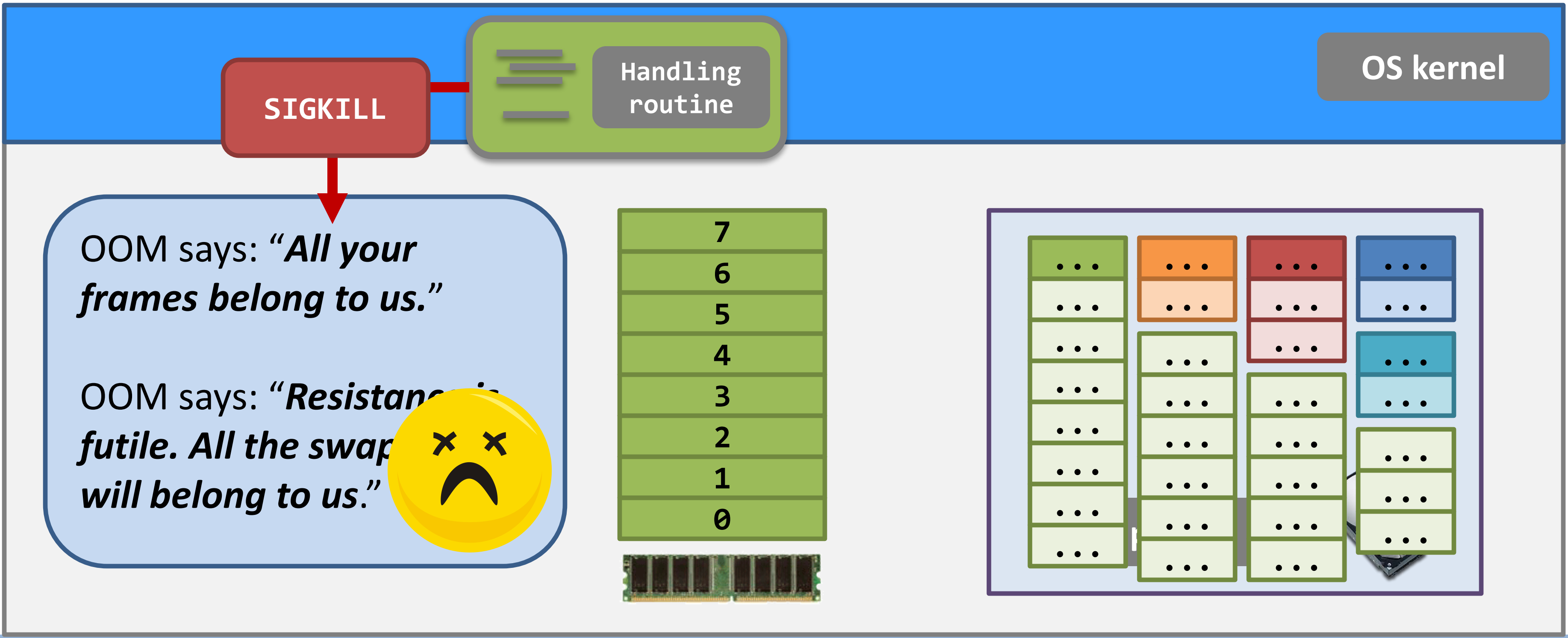
- OOM is running…
  - **<u>3rd stage</u>**. The previously-conquered frames are swapping to the swap area.
  - **Disk activity flies high!**



Real OOM – illustration

`oom_v3: running`

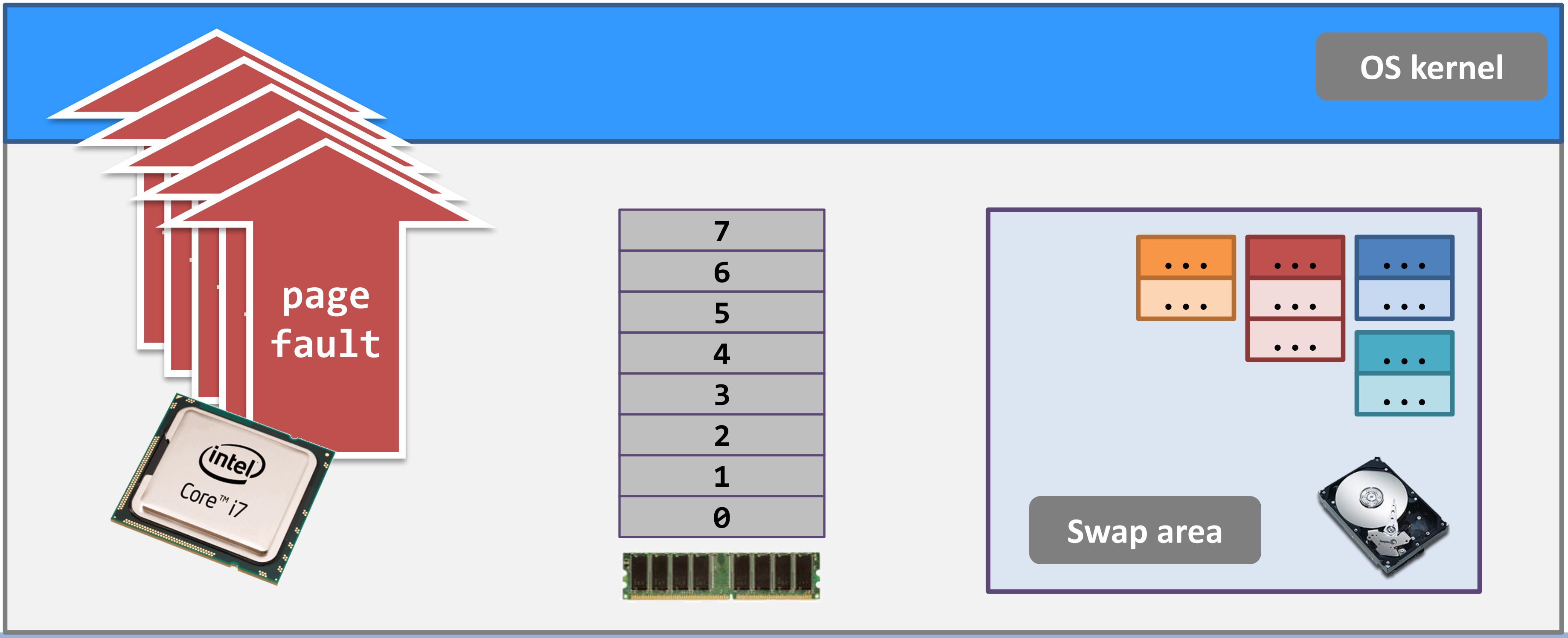
- OOM is running…
  - **<u>Final stage</u>**. The page fault handling routine finds that:
    - **No free space left in the swap area!**
    - **Decided to kill the OOM process!**



Real OOM – illustration

`oom_v3: killed`

- OOM has died, but…
  - **Painful aftermath**. **Lots of page faults!**
    - It is because other processes need to take back the frames!
    - **Disk activity flies high again,** but will go down eventually.

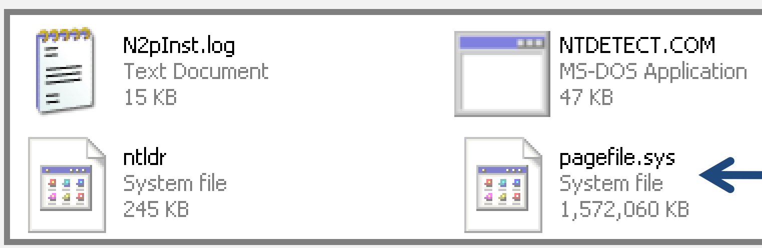


Swap area – location

- The swap area is usually **a space reserved** in a permanent storage device.

Linux needs a separate partition and it is called the **swap partition**.

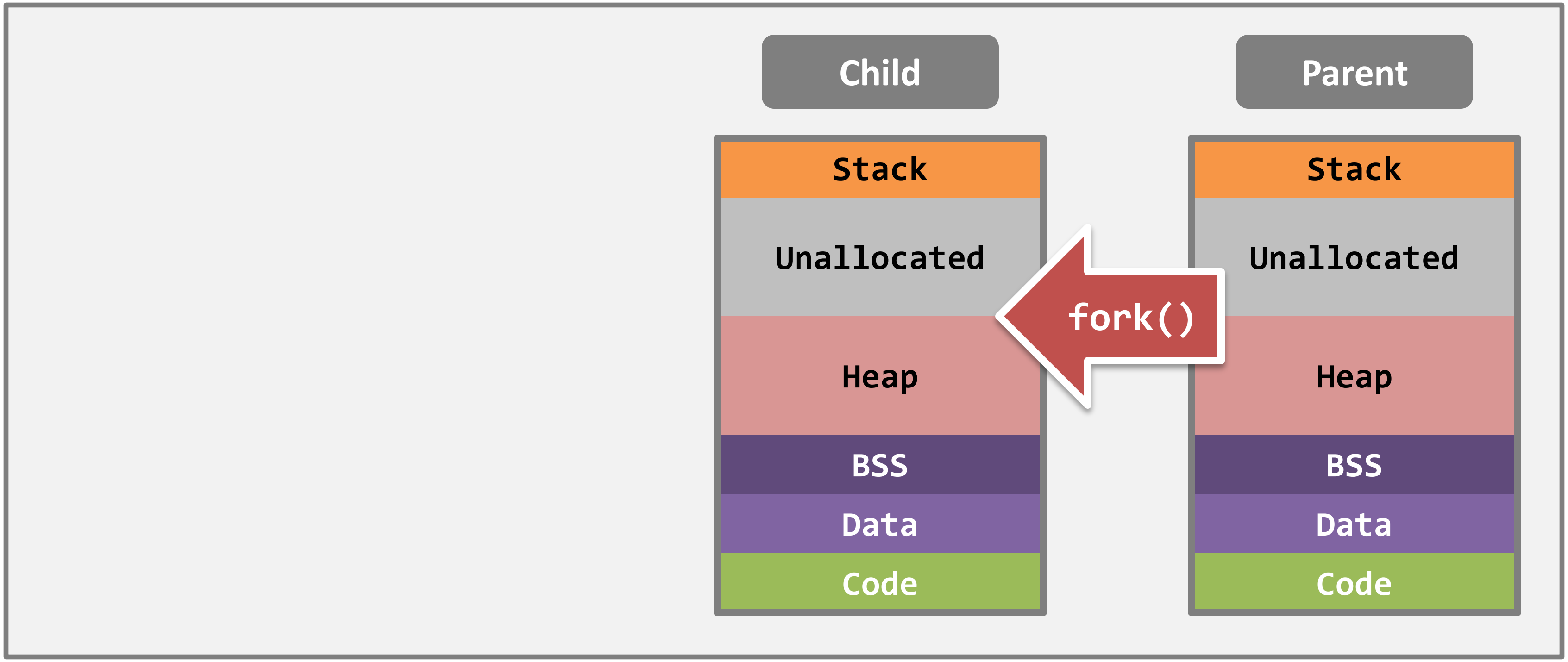
```
$ sudo fdisk /dev/sda
......
Command (m for help): p
......
/dev/sda1 ...... Linux
/dev/sda2 ...... Linux swap / Solaris
Command (m for help): _
```



Windows hides a file "**pagefile.sys**", which is the swap area, in one of the drives.

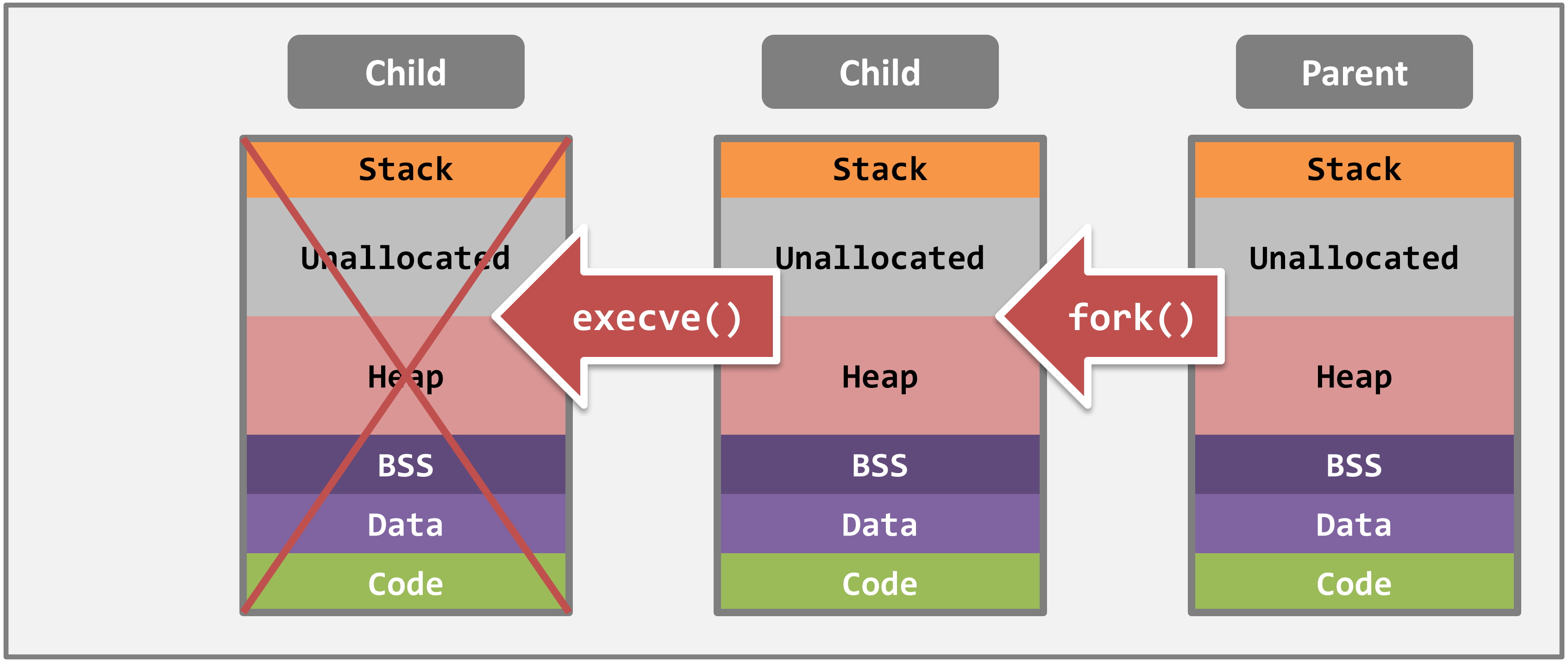
Demand paging – `fork()` & `execve()`

- What we have learned about the **`fork()`** system call is…**duplication**!
  - The parent process and the child process **are identical** from the *userspace memory* point of view.



Demand paging – `fork()` & `execve()`

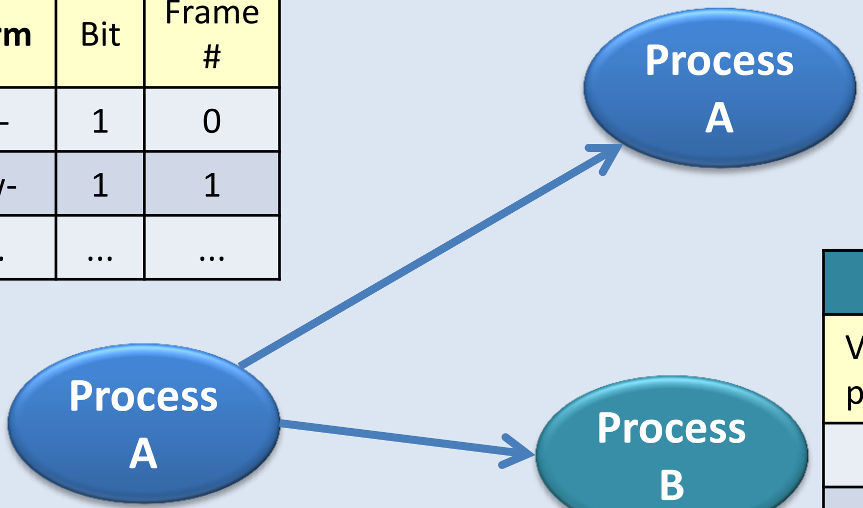
- Then, isn't it **stupid** to copy and then destroy?
  - Don't forget: **`fork()`** & **`execve()`** come in pair.
  - OMG…



Demand paging – **fork()** & **execve()**

- Copy-on-write (COW) technique.
  - During **fork()**, the permission in the page table will be changed.

Page Table – Process A			
Virtual page #	**Perm**	Bit	Frame #
A	r--	1	0
B	rw-	1	1
...	...	...	...

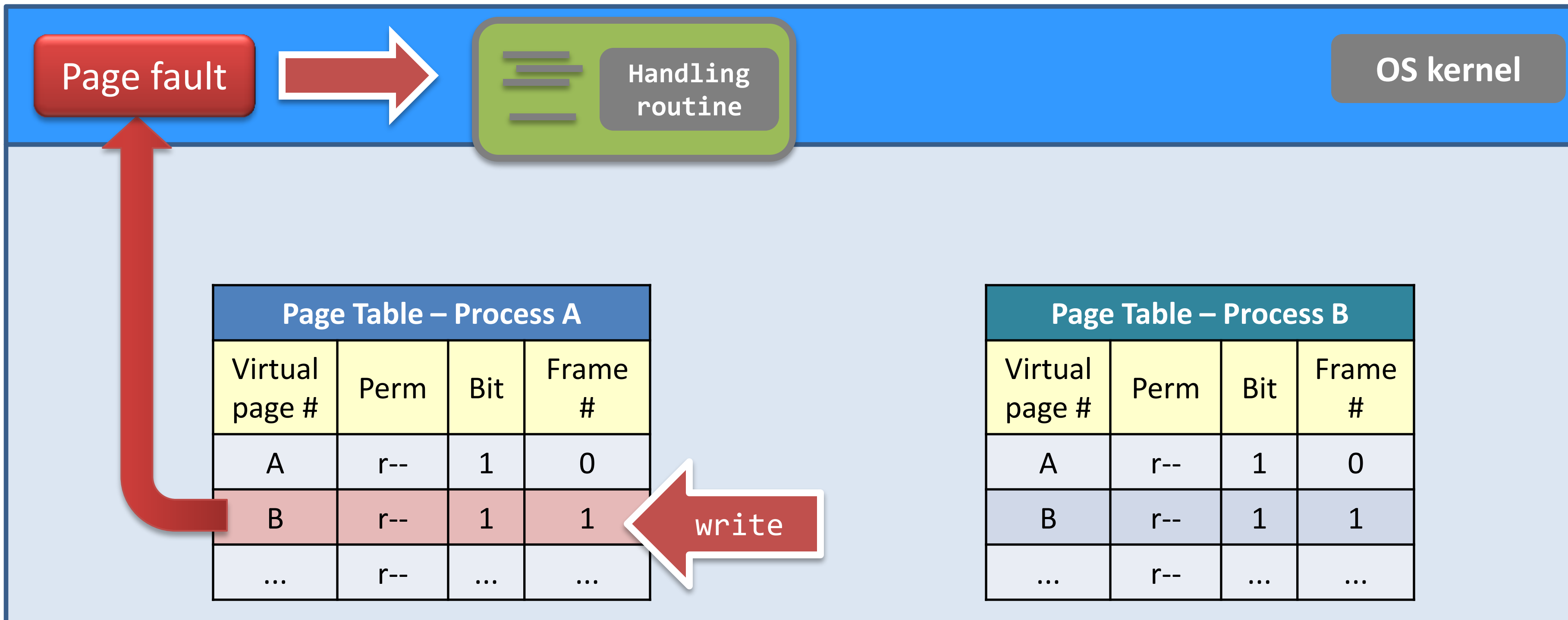


Page Table – Process A			
Virtual page #	**Perm**	Bit	Frame #
A	r--	1	0
B	r--	1	1
...	r--	...	...

Page Table – Process B			
Virtual page #	**Perm**	Bit	Frame #
A	r--	1	0
B	r--	1	1
...	r--	...	...

Demand paging – `fork()` & `execve()`

- Copy-on-write (COW) technique.
  - When there is a write on a page, page fault is generated.

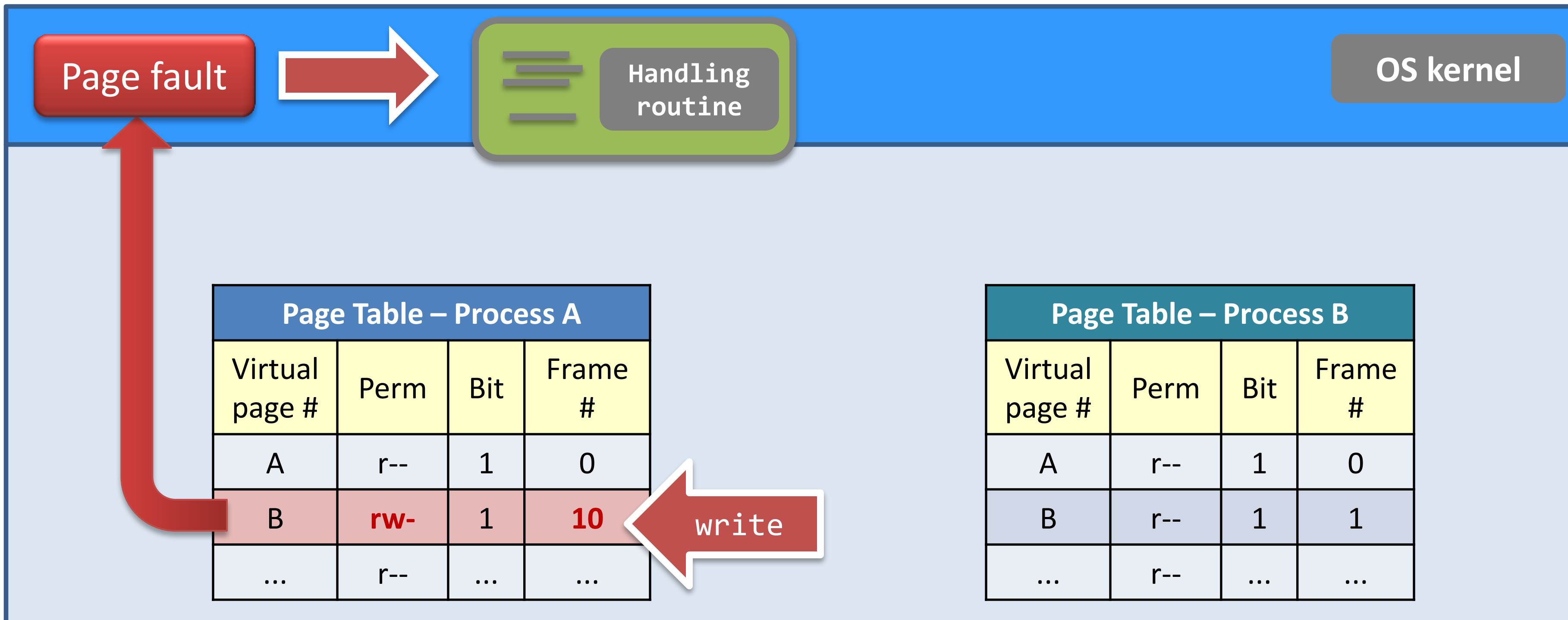


Page Table – Process A			
Virtual page #	Perm	Bit	Frame #
A	r--	1	0
B	r--	1	1
…	r--	…	…

Page Table – Process B			
Virtual page #	Perm	Bit	Frame #
A	r--	1	0
B	r--	1	1
…	r--	…	…

Demand paging – `fork()` & `execve()`

- Copy-on-write (COW) technique.
  - When there is a write on a page, page fault is generated.
  - The handler changes the page table and **copies a new page** for the process.



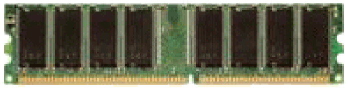
Page Table – Process A			
Virtual page #	Perm	Bit	Frame #
A	r--	1	0
B	**rw-**	1	**10**
…	r--	…	…

Page Table – Process B			
Virtual page #	Perm	Bit	Frame #
A	r--	1	0
B	r--	1	1
…	r--	…	…

EXTRA

Memory Management

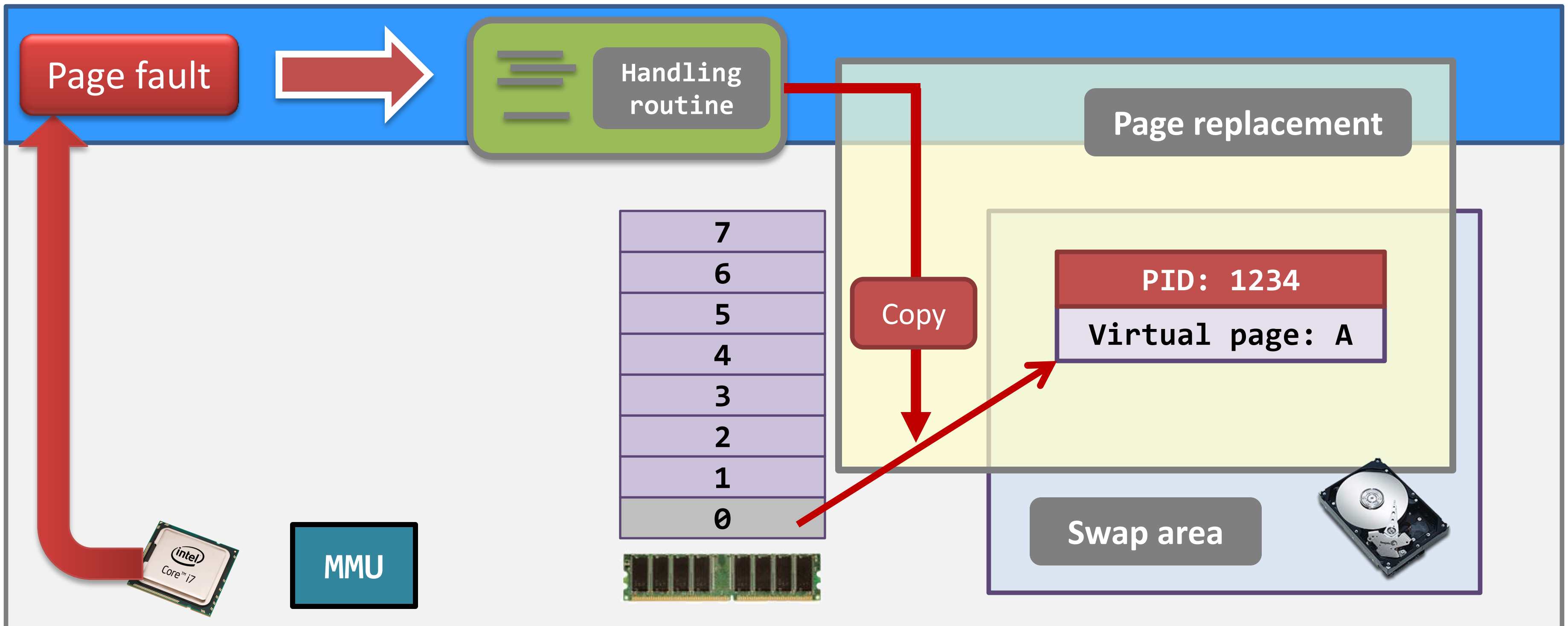
- Virtual memory = CPU + MMU;
- MMU implementation & paging;
- Demand paging;
- **Page replacement algorithms;**



Page replacement – introduction

EXTRA

- Remember the **page replacement operation**?
  - It is the job of the kernel to **find a victim page** in the physical memory, and...
  - write the victim page to the swap space.



Page replacement – introduction

EXTRA

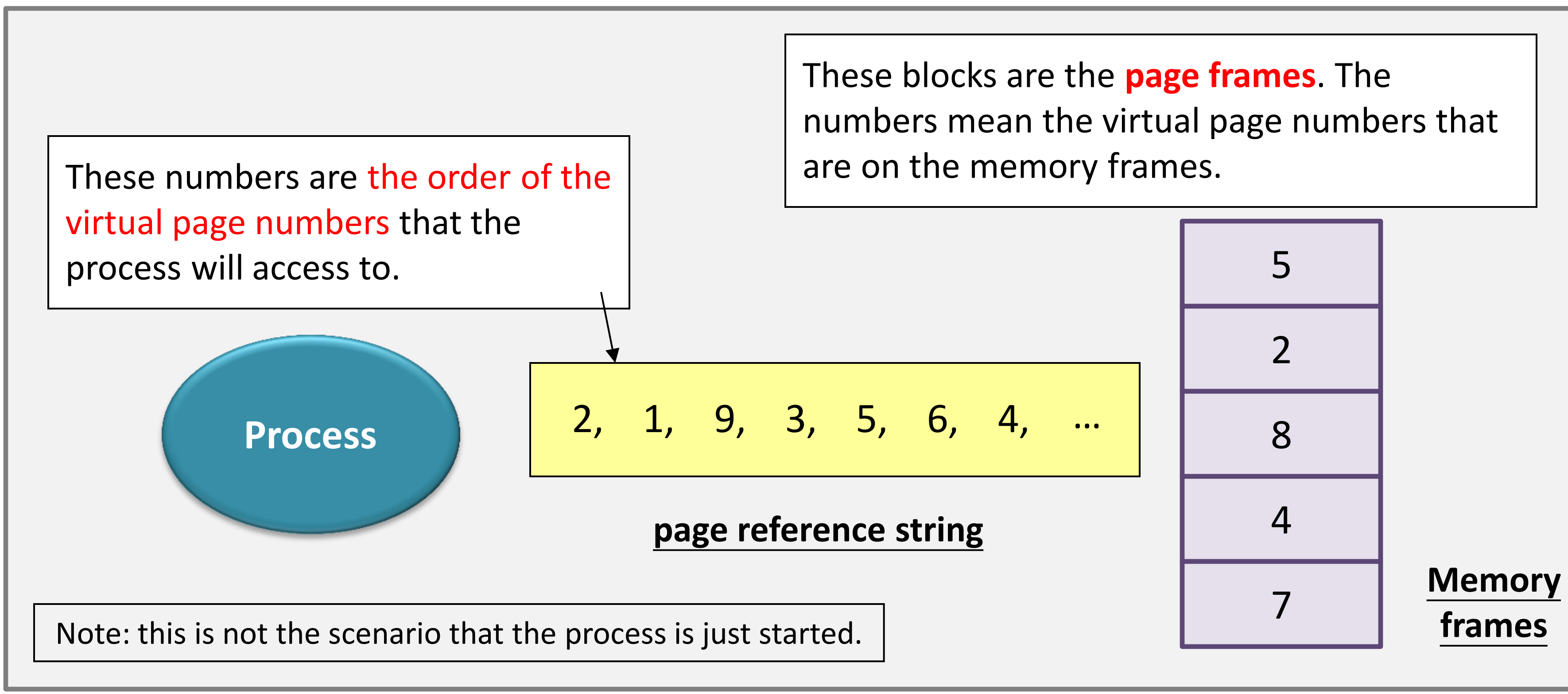
- Replacing a page involves disk accesses, therefore a page fault is **slow and expensive**!
  - Page replacement algorithms should **minimize further page faults**.

- In the following, we introduce three page replacement algorithms:
  - Optimal;
  - First-in first-out (FIFO);
  - Least recently used (LRU);

Page replacement – algorithm

EXTRA

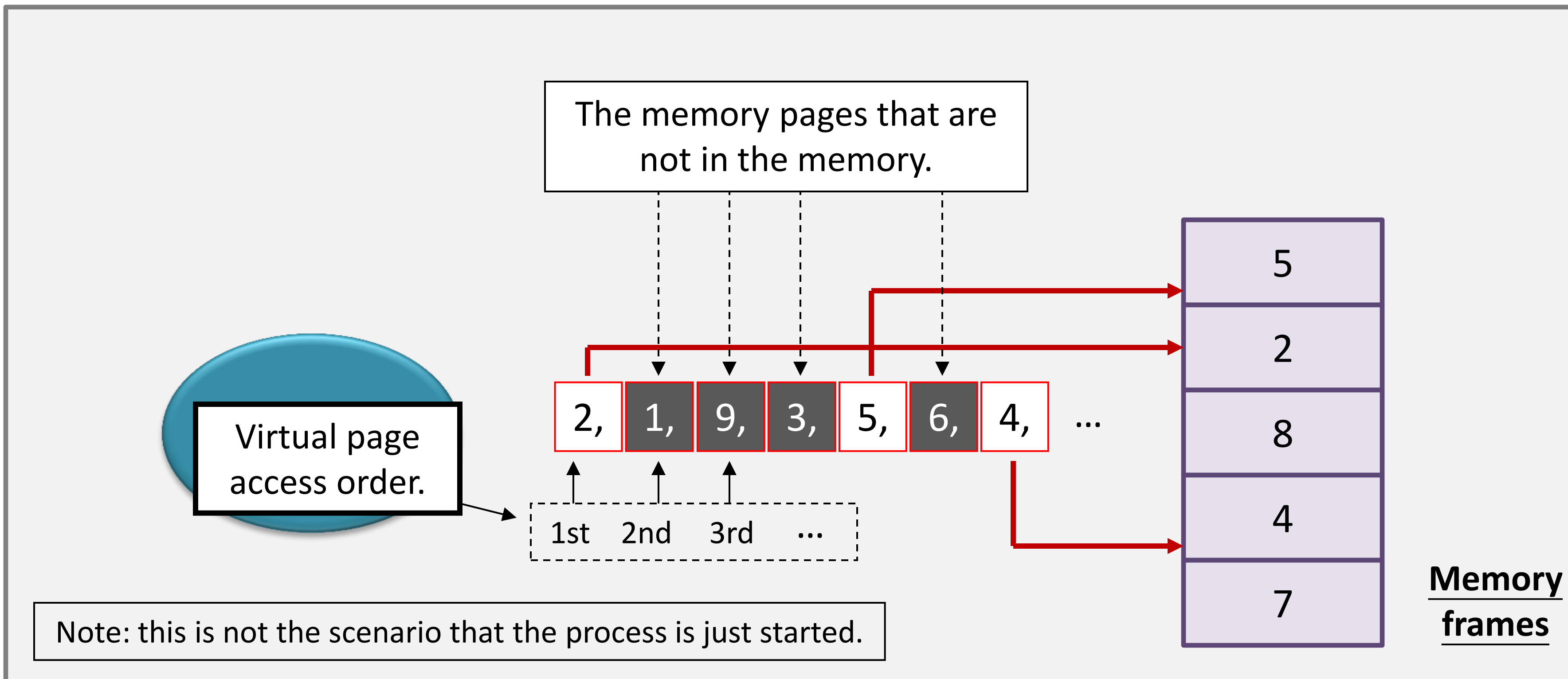
- Imagine that you are the kernel…
  - you have a process just started to run;
  - the process' memory is larger than the physical memory;
  - assume that all the pages are in the swap space.



Page replacement – algorithm

EXTRA

- Imagine that you are the kernel…
  - you have a process just started to run;
  - the process' memory is larger than the physical memory;
  - assume that all the pages are in the swap space.



Page replacement – when an algorithm starts

EXTRA

- Initially, let all the frames to be empty.
  - The first page request will cause a **page fault**.
  - Because there are free frames, no replacement is needed.

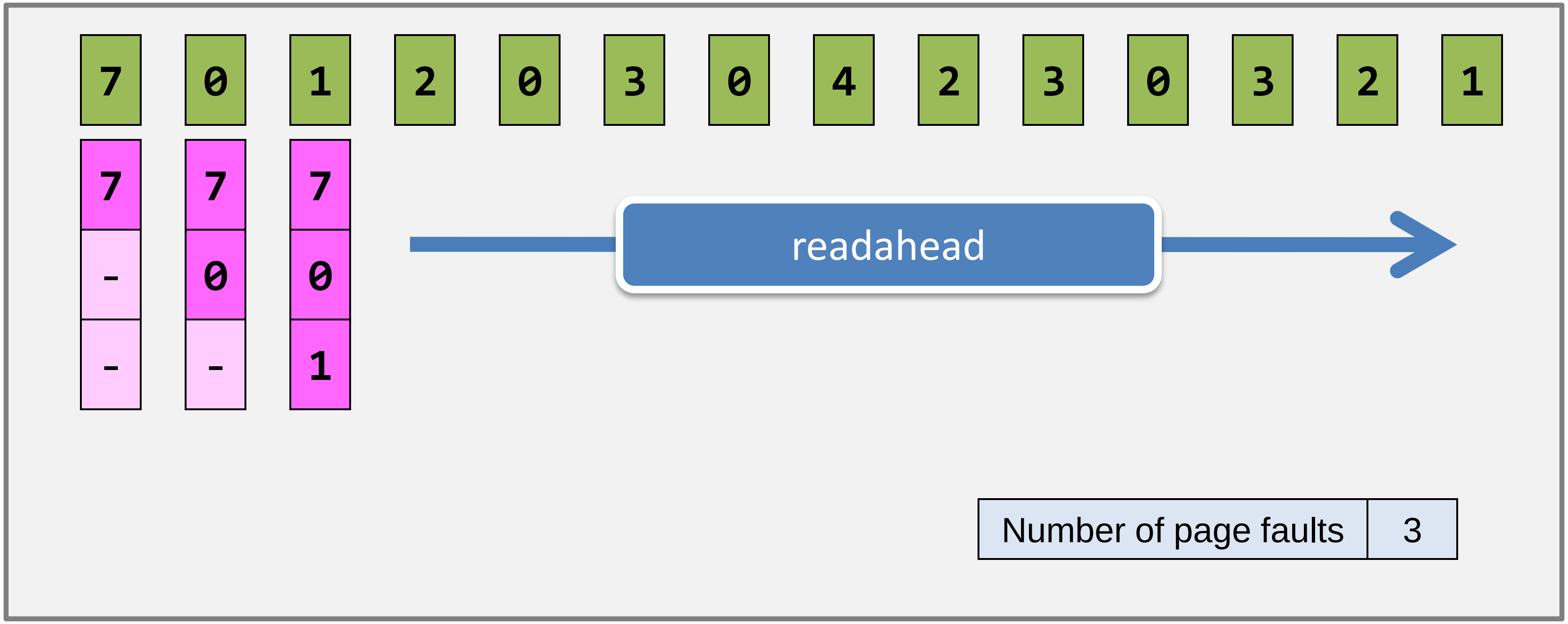
7	0	1	2	0	3	0	4	2	3	0	3	2	1
7													
-													
-													

Number of page faults	1

Page replacement – optimal algorithm

EXTRA

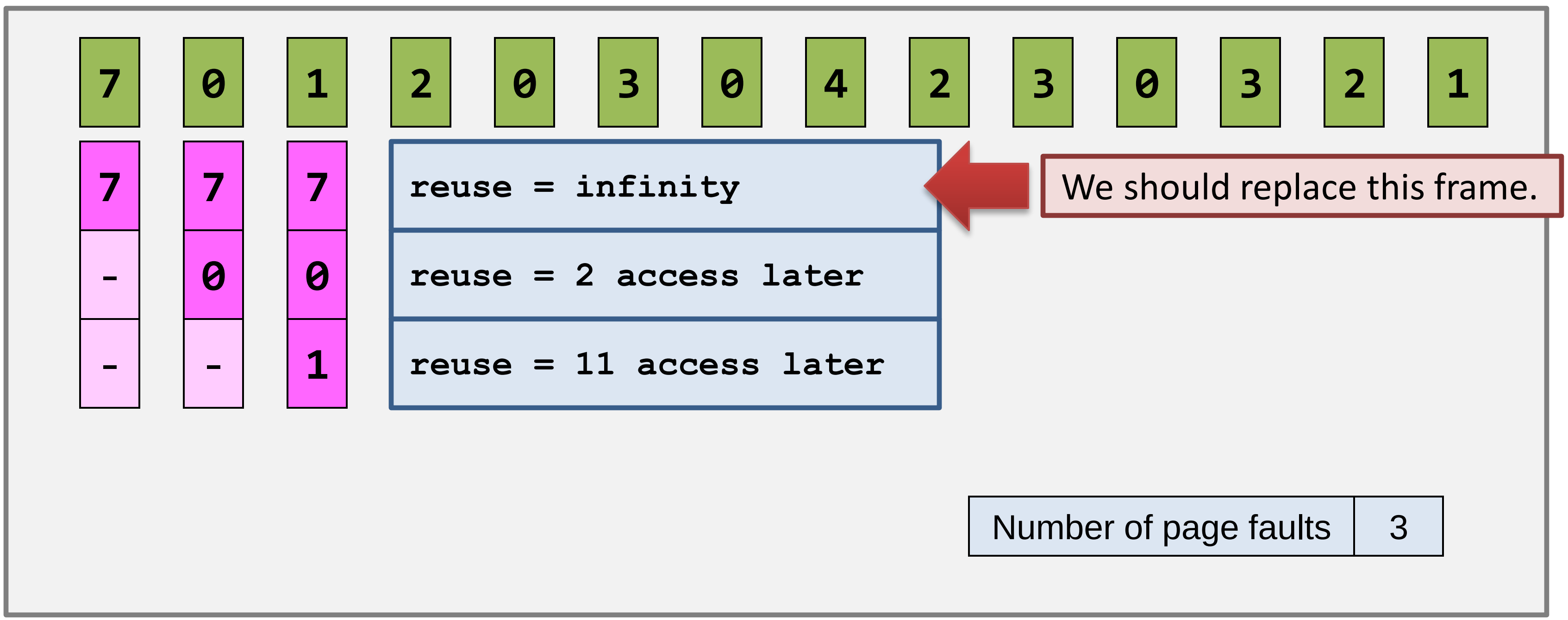
- If I **know the future**, then I know how to do better.
  - That means I can optimize the result *if the page reference string is given in advance*.
  - That's why the algorithm is called "optimal".



Page replacement – optimal algorithm

EXTRA

- Strategy:
  - To replace the page that **will not be used for the longest period of time**.



Page replacement – optimal algorithm

EXTRA

- The story goes on…
  - But, do you think that this is a **non-sense**?
  - Of course, this is to give you a sense that **how close** an algorithm is from the optimal.

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

Number of page faults	8

Page replacement – FIFO algorithm

EXTRA

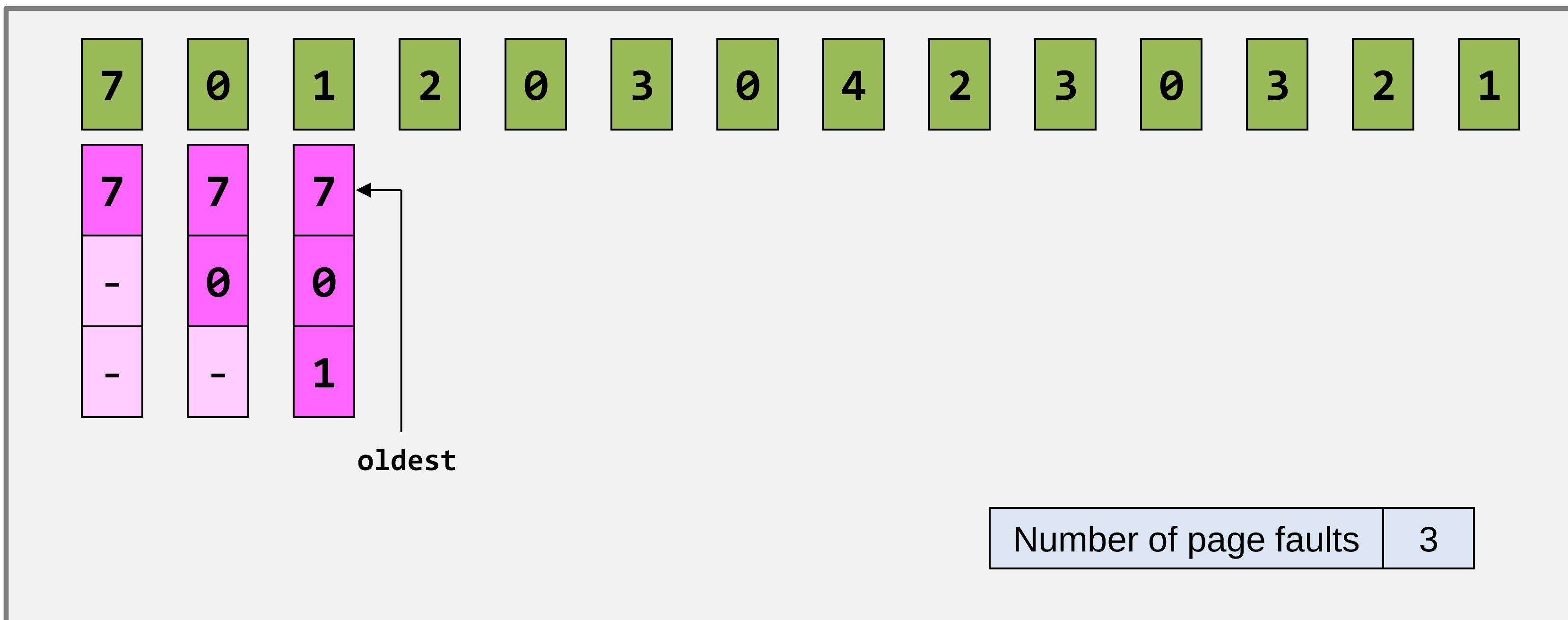
- Unfortunately, you never know the future…
  - Let us demonstrate an easy-to-implement algorithm.

- FIFO: the **first page being swapped into** the frames will be the **first page being swapped out** of the frames.
  - The victim page will always be the oldest page among all the frames.
  - The age of a page is counted by the time period that it is stored in the memory.

Page replacement – FIFO algorithm

EXTRA

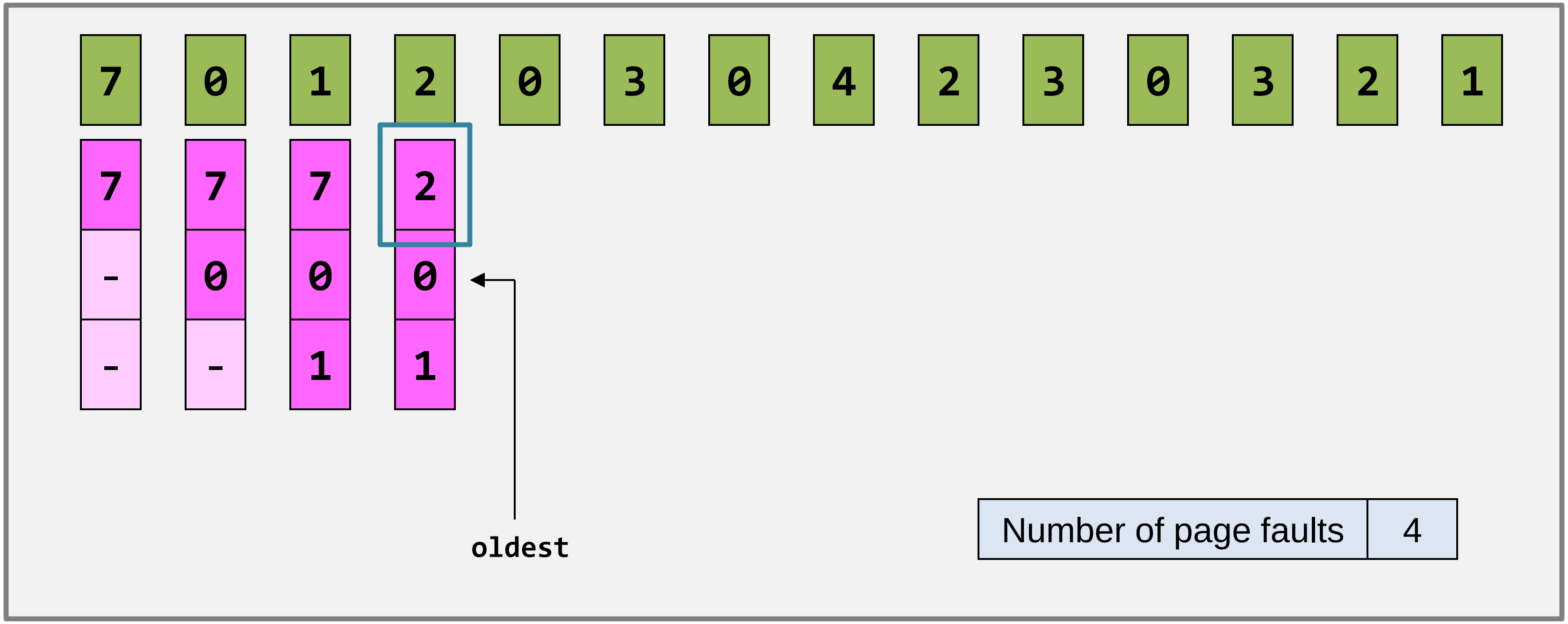
- When there is no free frames,
  - The **FIFO page replacement algorithm** will choose the oldest page to be the victim.



Page replacement – FIFO algorithm

EXTRA

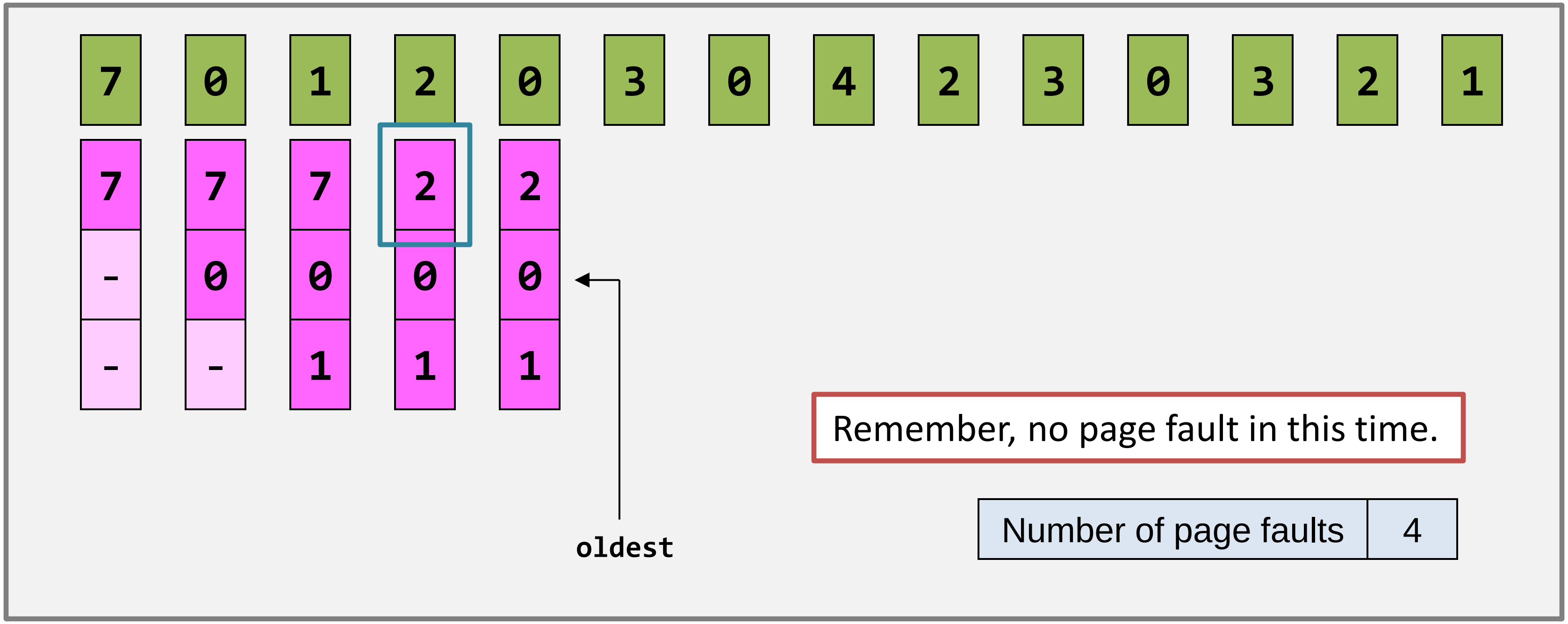
- When there is no free frames,
  - The **FIFO page replacement algorithm** will choose the oldest page to be the victim.
  - Of course, the oldest page changes.



Page replacement – FIFO algorithm

EXTRA

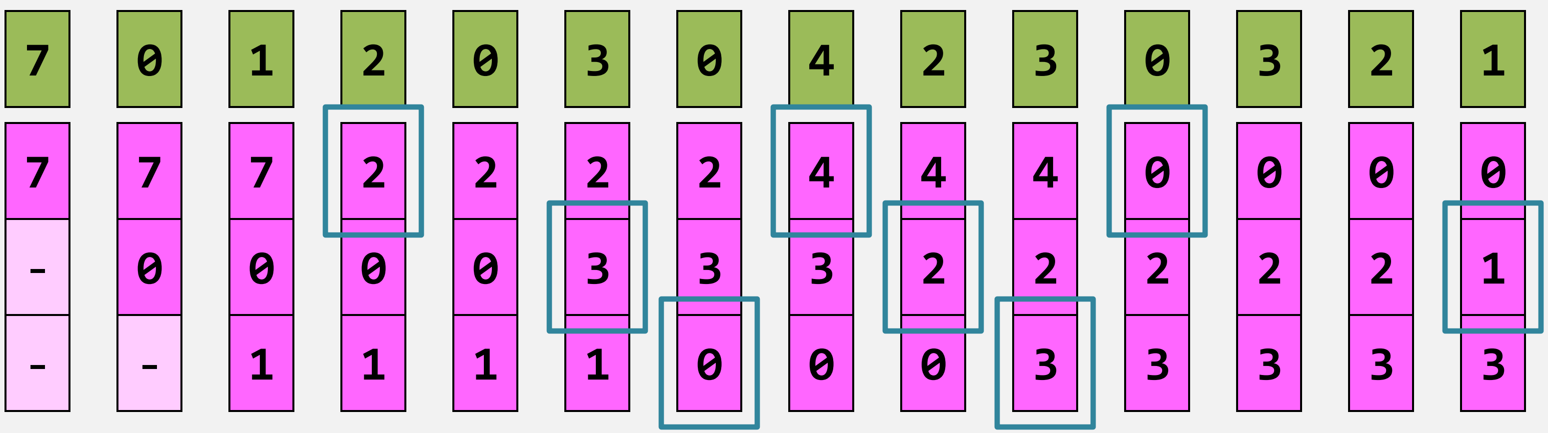
- When a memory reference can be found in the memory,
  - The age of that frame will not change.
    - The frame storing “page 0” is still the oldest frame.



Page replacement – FIFO algorithm

EXTRA

- The story goes on…
  - Seems that there is **no intelligence** in this method…
  - Can we do better than this?



Number of page faults	11
Number of page faults in the optimal algorithm	8

Page replacement – LRU algorithm

EXTRA

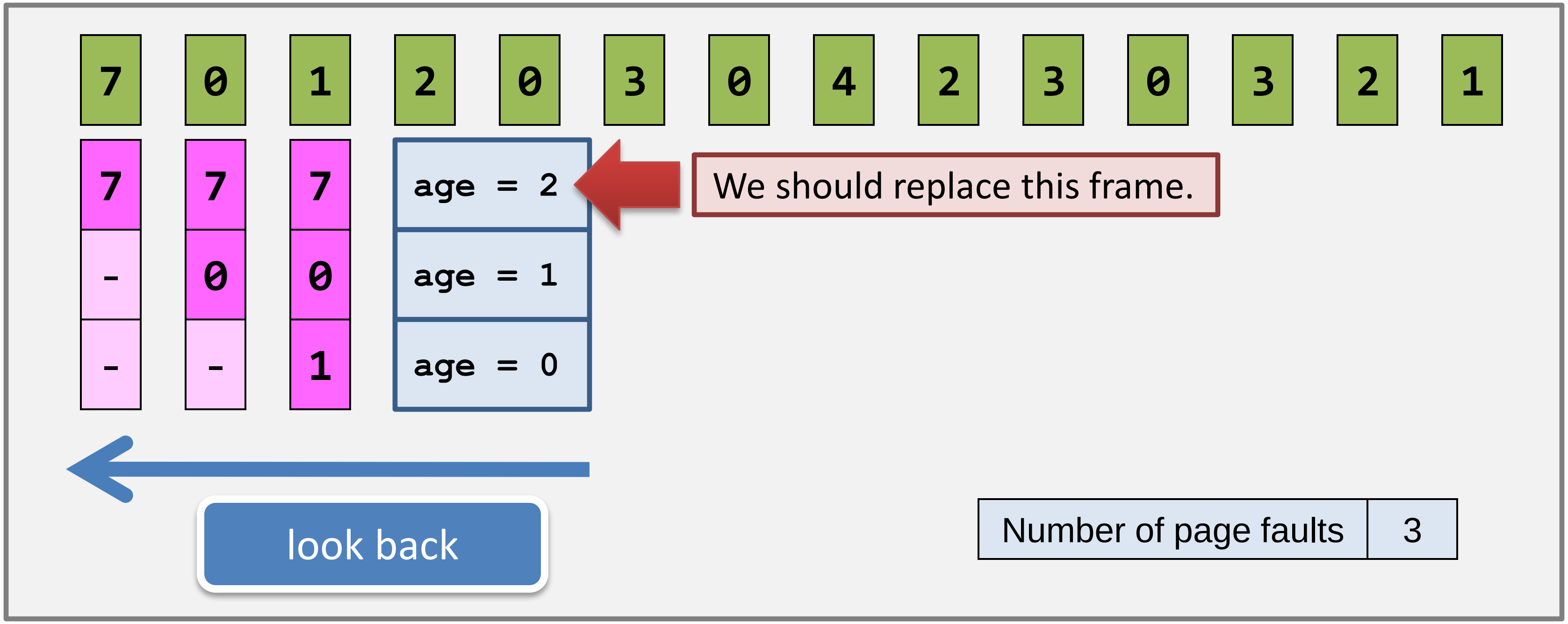
- Unfortunately, FIFO is very stupid…
  - A usually-chosen algorithm is the **least-recently-used** (LRU) page replacement.

- Methodology:
  - Attach every frame with an age, which is an integer.
  - When a page is just accessed,
    - no matter that page is originally on a frame or not, **set the frame's age to be 0**.
    - Other frames' ages are incremented by 1.

Page replacement – LRU algorithm

EXTRA

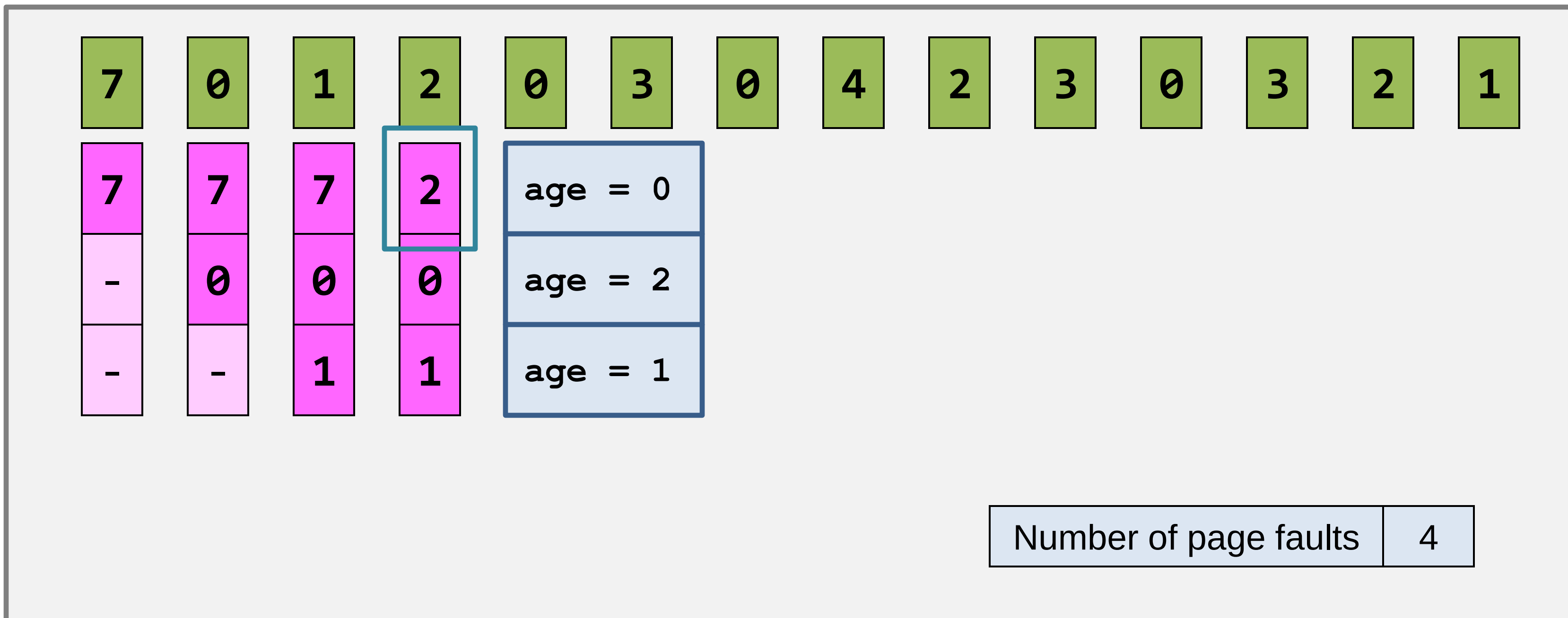
- Strategy:
  – To replace the page that is least-recently used, *not necessarily the oldest frame*.



Page replacement – LRU algorithm

EXTRA

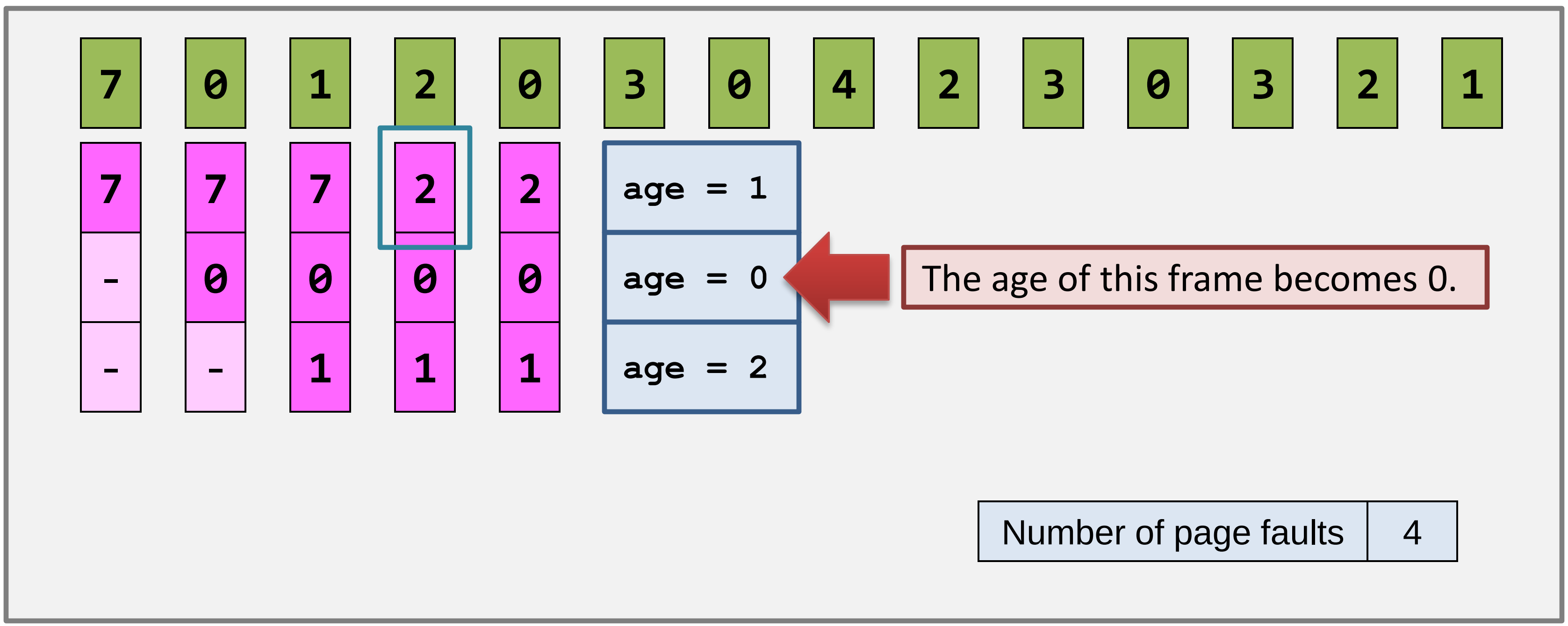
- Strategy:
  - To replace the page that is least-recently used, *not necessarily the oldest frame*.



Page replacement – LRU algorithm

EXTRA

- Strategy:
  - To replace the page that is least-recently used, *not necessarily the oldest frame*.



Page replacement – LRU algorithm

EXTRA

- Strategy:
  - To replace the page that is least-recently used, *not necessarily the oldest frame*.

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

Number of page faults	10

Page replacement – performance

EXTRA

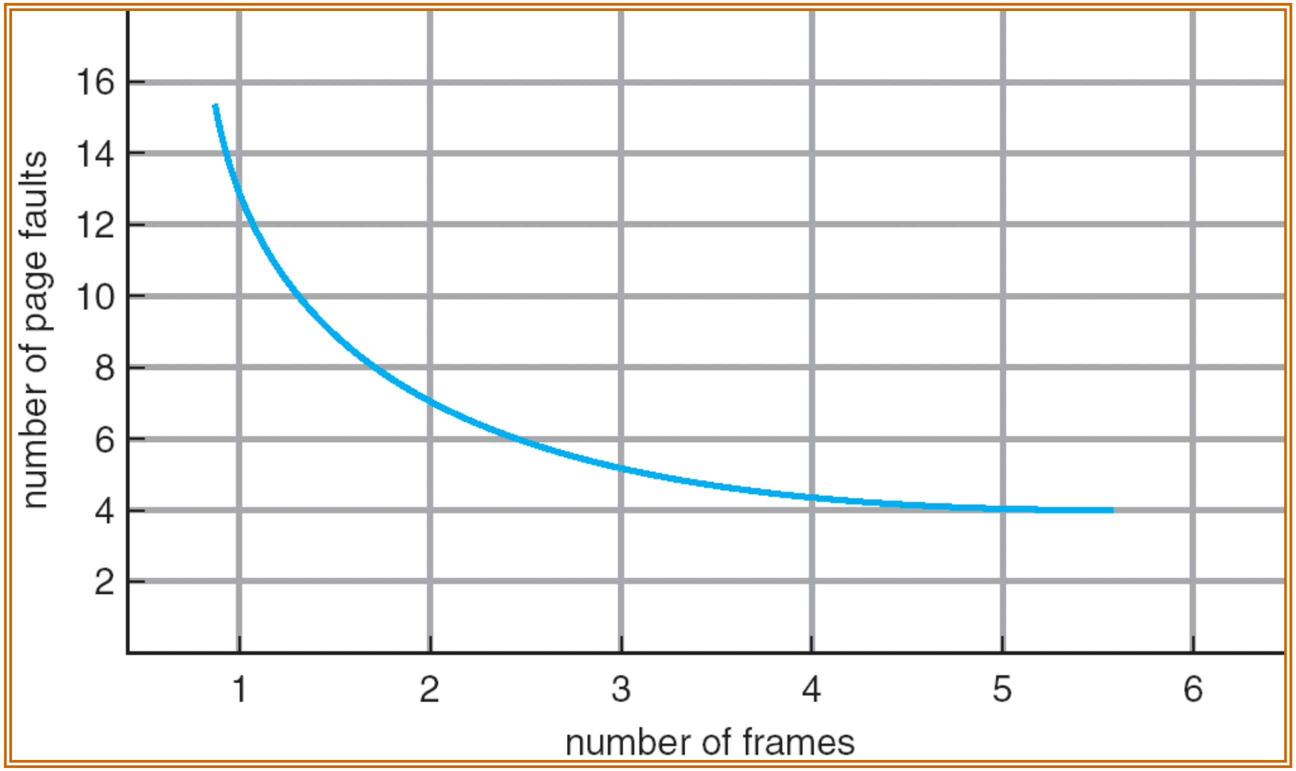
- Number of page frames VS Performance.
  - Increasing the number of page frames implies increasing the amount of the physical memory.

- So, it is natural to think that:
  - I have more memory…and more frames…
  - Then, my system **must be faster** than before!
  - Therefore, the number of **page faults must be fewer** than before, given the same page reference string.

Page replacement – performance

EXTRA

- Your expectation:

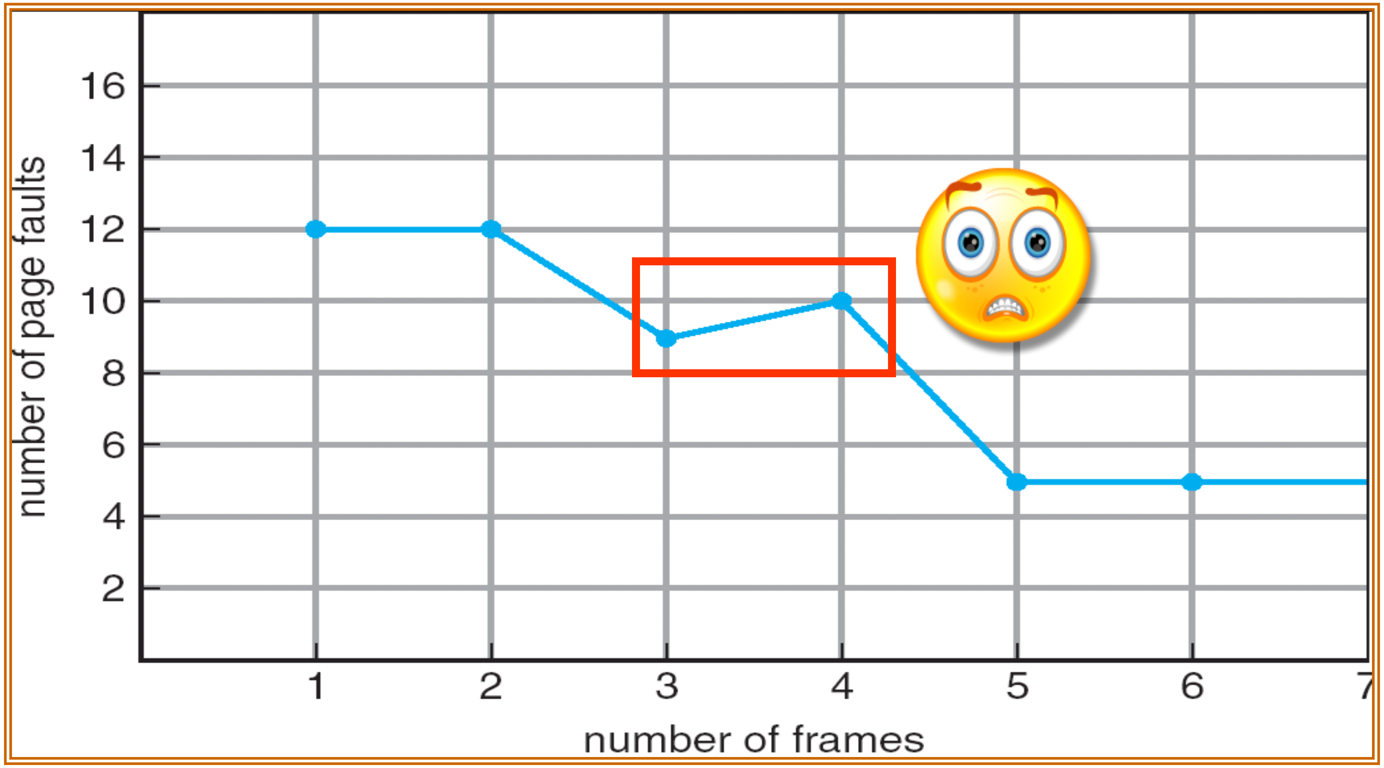


Page replacement – performance

EXTRA

- The reality may be:

This is called Belady's anomaly



Page replacement – performance

EXTRA

- Try the following:
  - all page frames are initially empty;
  - use FIFO page replacement algorithm;
  - use the number of frames: 3, 4, and 5.
  - The page reference string is:

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

- You'll result in the previous graph.

Conclusion

EXTRA

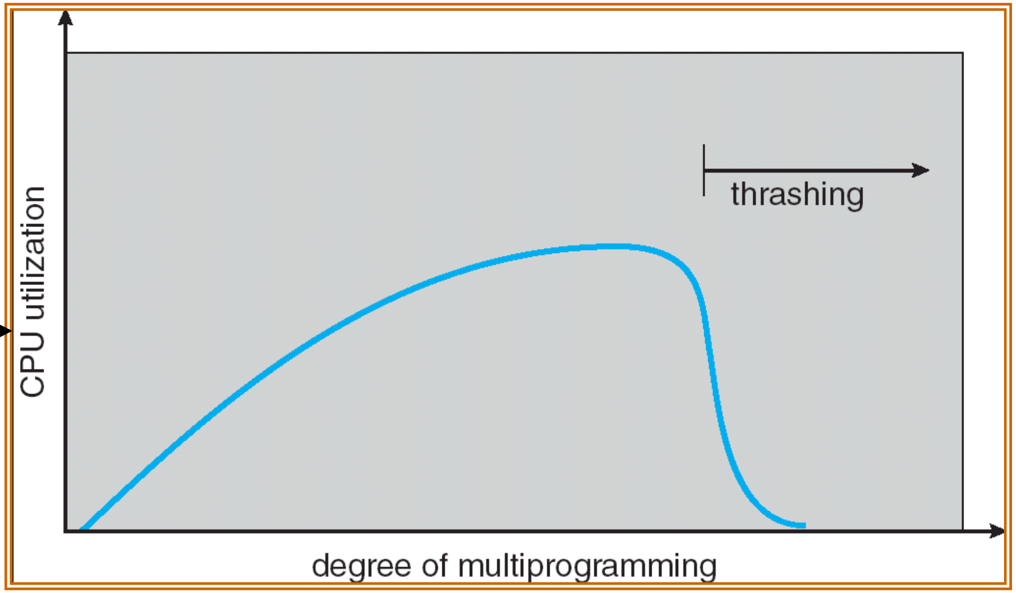
- Virtual memory is a huge topic which can span one semester time.
  - Only the basics of VM are covered.
  - We missed TLB:
    Translation Look-aside Buffer
  - Real implementation in CPU.

Conclusion

EXTRA

- With the VM support, the overall performance of an OS is about:
  - the number of processes and
  - the memory access pattern.

This means: $\dfrac{\text{User time}}{\text{System time}}$



(number of processes)

Hope you enjoyed CSCI 3150!