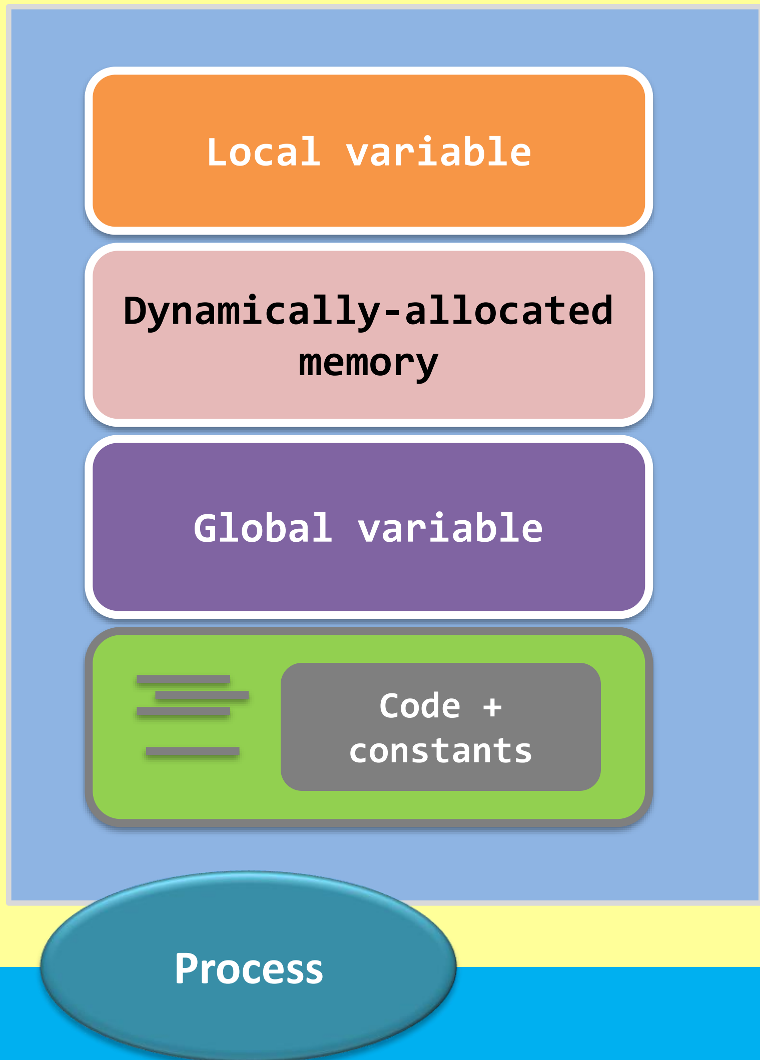
Operating Systems

Prof. Yongkun Li
中科大-计算机学院 特任教授
http://staff.ustc.edu.cn/~ykli

Ch7, part 2

Memory Management from the Kernel's Perspective: Virtual Memory Support

Memory management



How to use the addresses to access the memory device?

How do multiple process share the same physical memory device?

How to support large process?

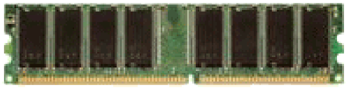
How does the CPU read what it wants from the memory device?

……

The kernel and the hardware are doing lots of managements…

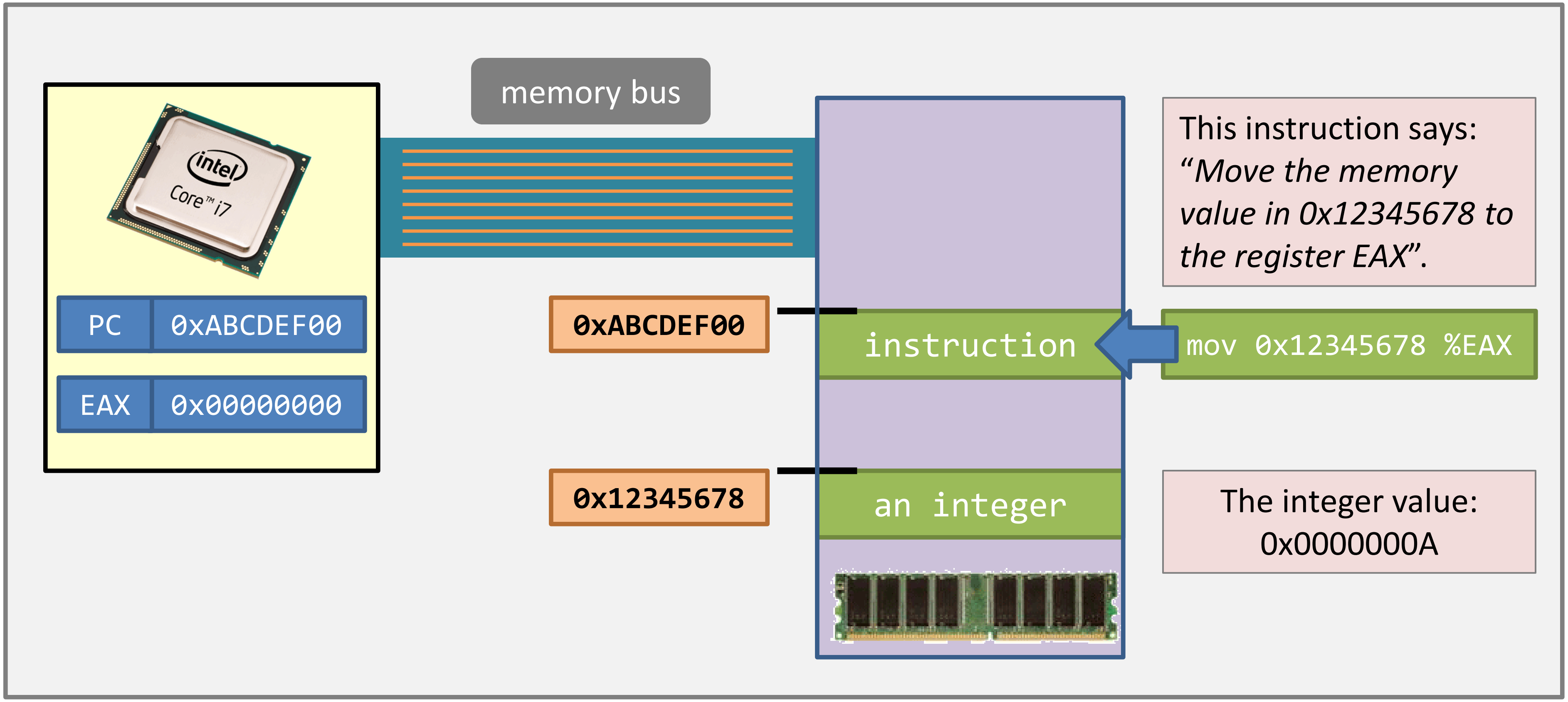
Memory Management

- Virtual memory;
- MMU implementation & paging;
- Demand paging;
- Page replacement algorithms;
- Allocation of frames;



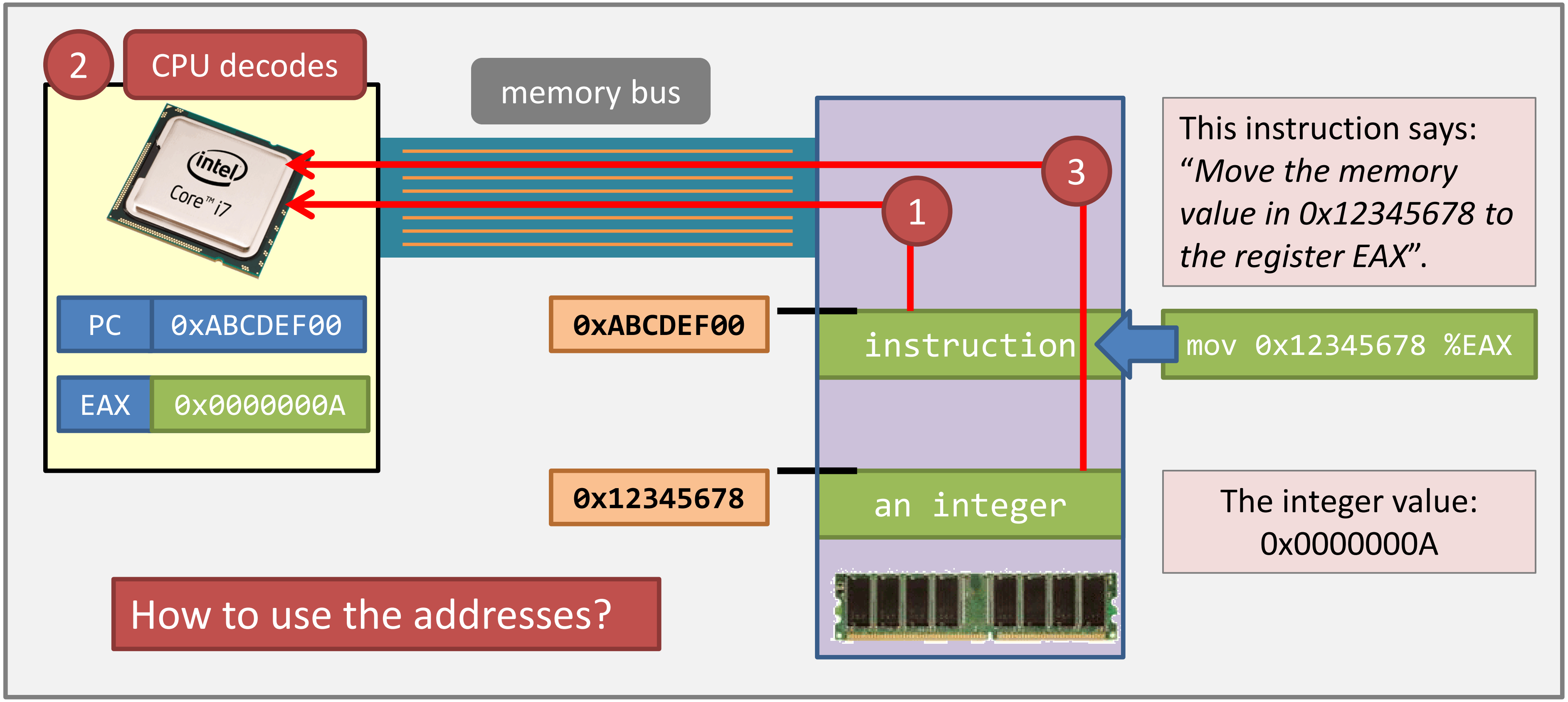
CPU working – illustration that you may know

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



CPU working – illustration that you may know

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



"You've been living in a dream world, Neo"

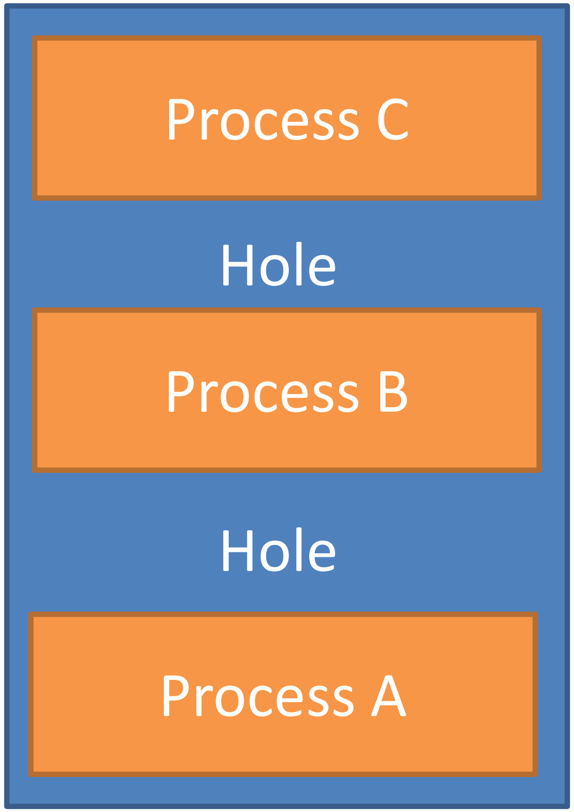
```c
int main(void) {
    int pid;
    pid = fork();
    printf("PID %d: %p.\n", getpid(), &pid);
    if(pid)
        wait(NULL);
    return 0;
}
```

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

- Can you guess 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?!
  - Logical address: **virtual memory**
  - Address translation needed (logical/virtual->physical)
  - Why we use virtual memory??

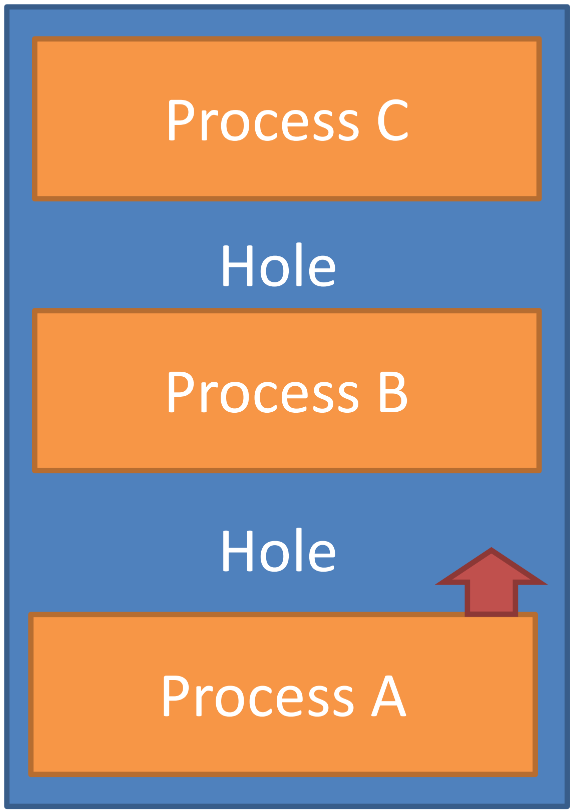
CPU working … contiguous allocation?

- Each process is contained in a single section of mem



CPU working … contiguous allocation?

- Problem #1…



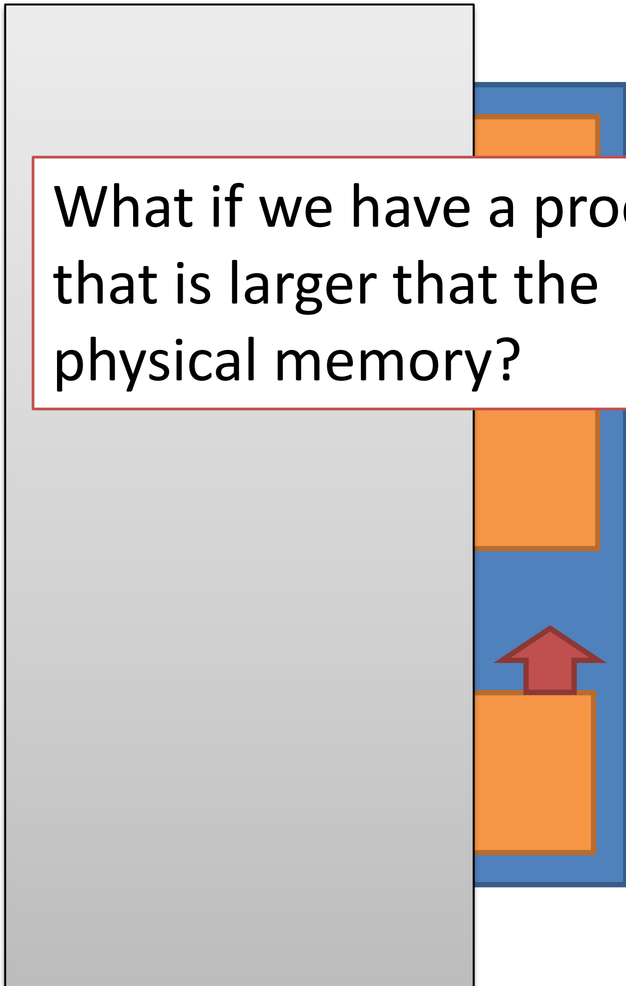
We also know that a process' memory can grow.

So, does a process **always** have a chance to grow to reach its need?

memory growth
e.g., because of **brk()** calls

CPU working … contiguous allocation?

- Problem #2…



What if we have a process that is larger that the physical memory?

We are not talking about the program's size, but the process' size!

What the CPU (or OS) can do is to give up running …

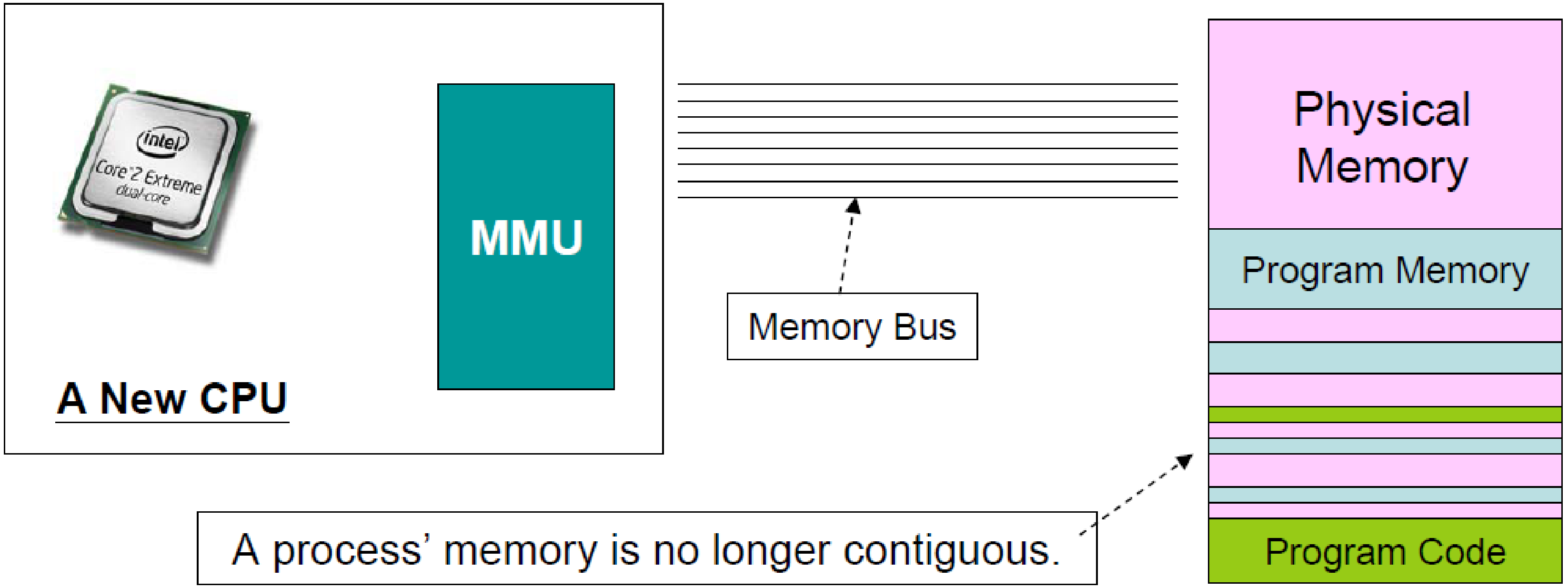
So, we need to have the CPU design that can understand processes so that:

(1) the address space is no longer required to be contiguous.

(2) it allows a process to have a size beyond the physical memory.

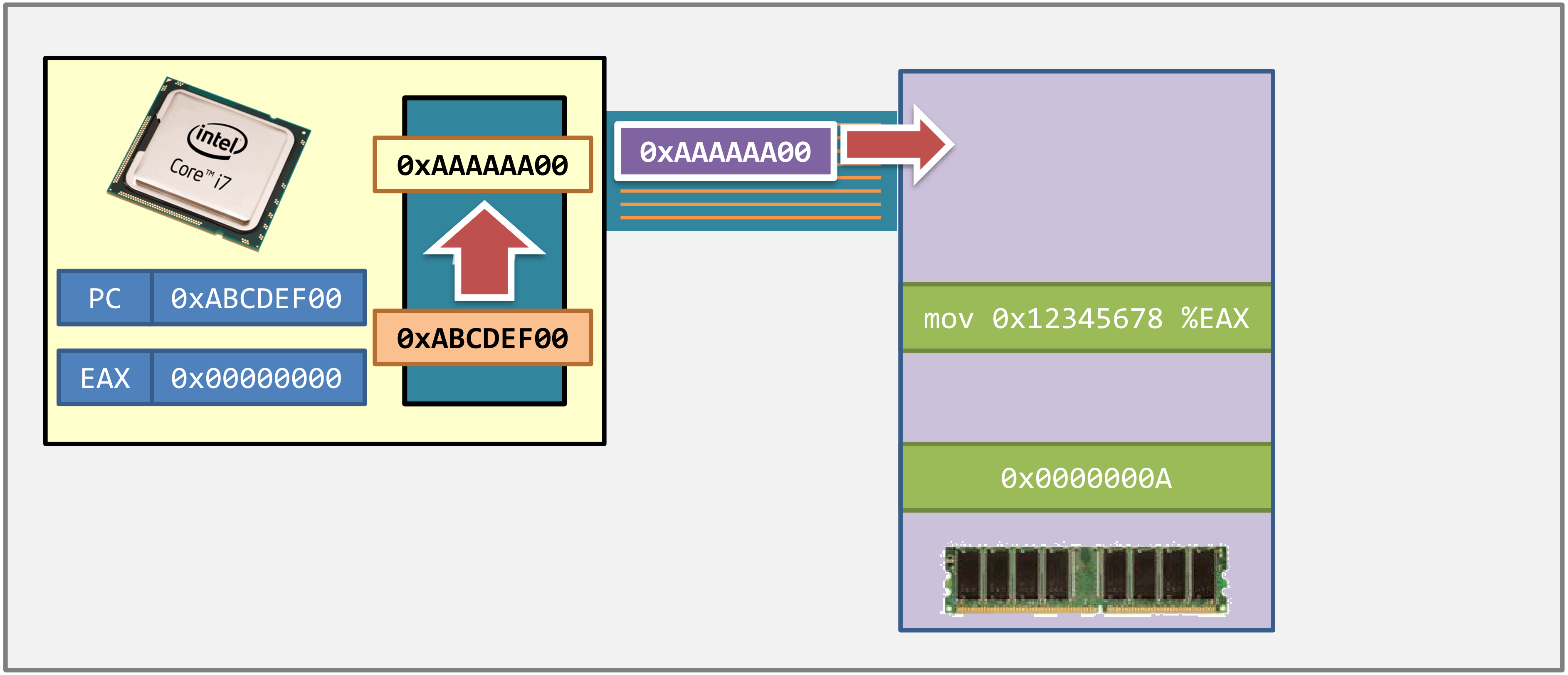
Virtual memory support in modern CPUs

- The new design of the CPU includes a new module: the **memory management unit (MMU).**
  - MMU is designed to perform address translation.
  - The **MMU** is an **on-CPU device**.



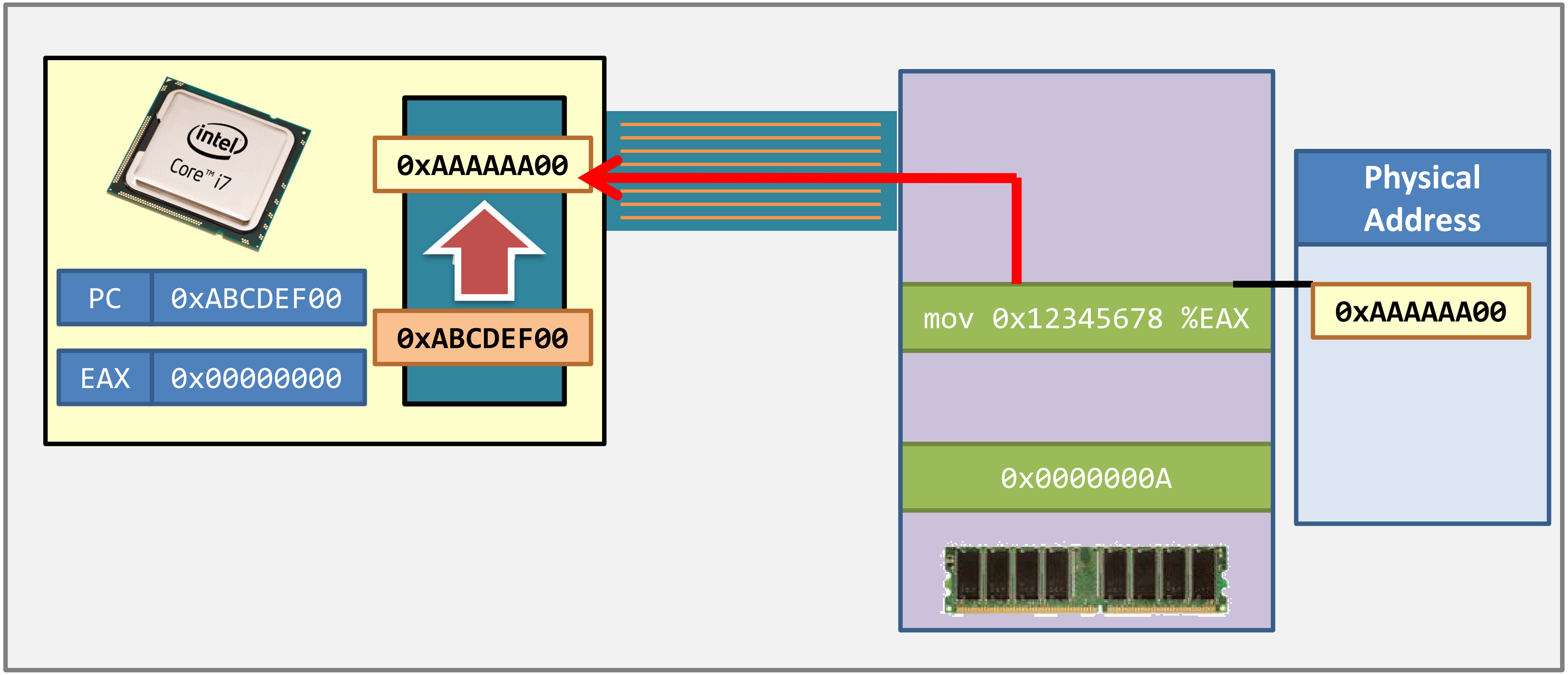
Virtual memory – how does it work?

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



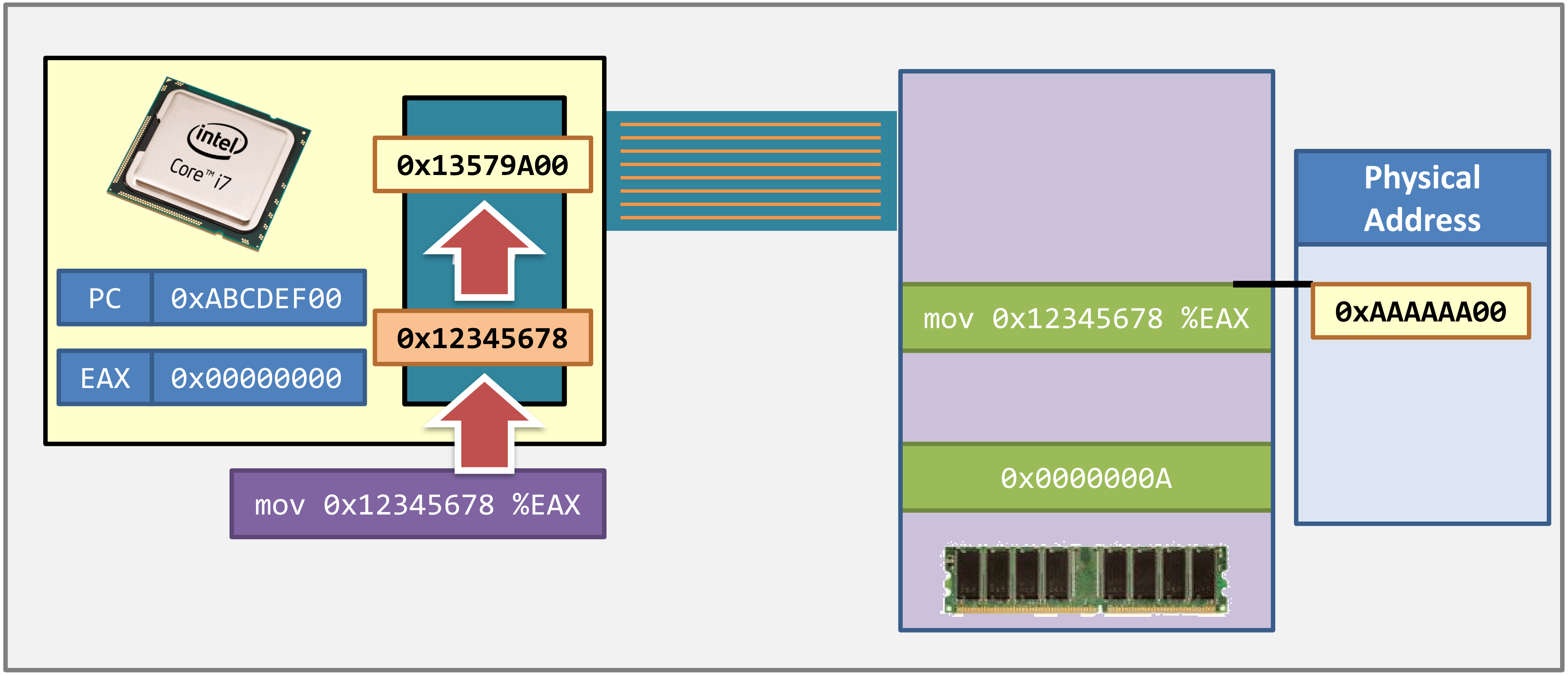
Virtual memory – how does it work?

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



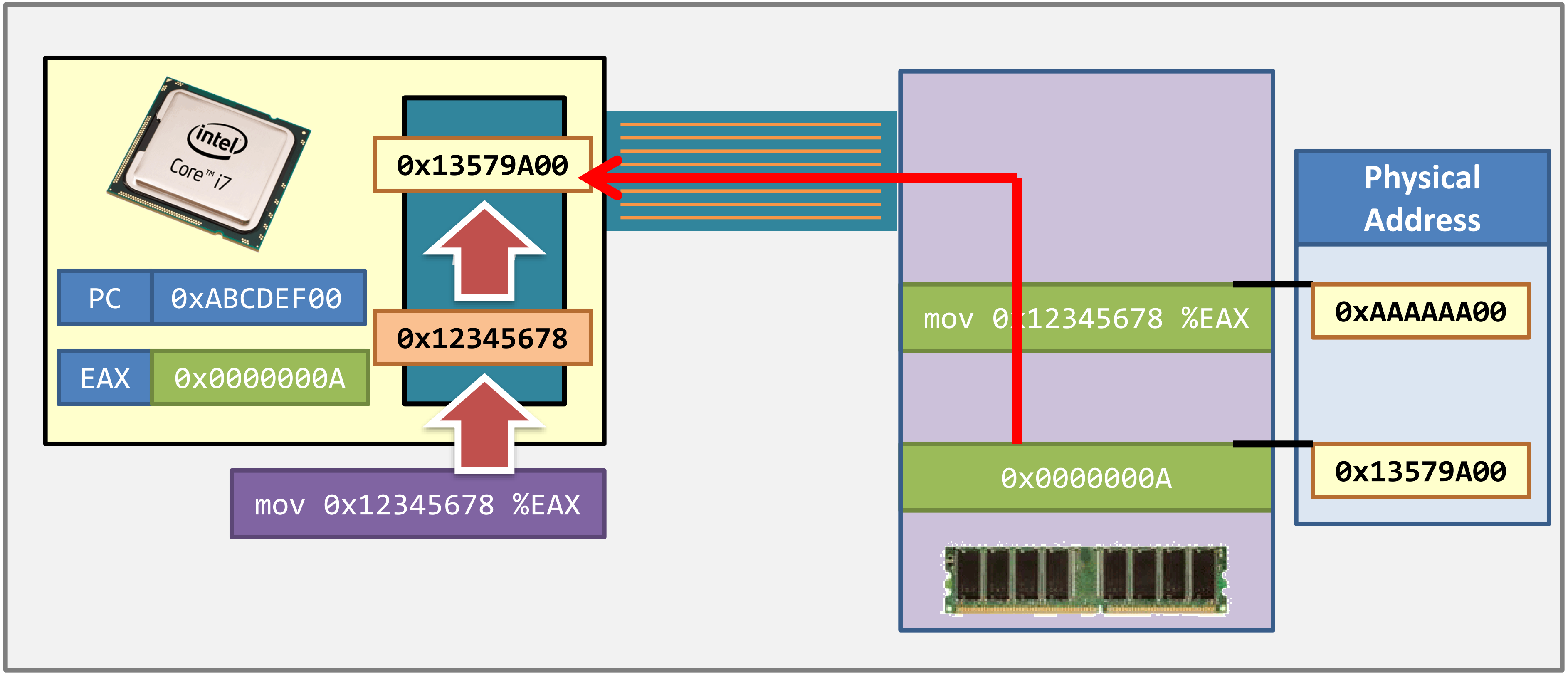
Virtual memory – how does it work?

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



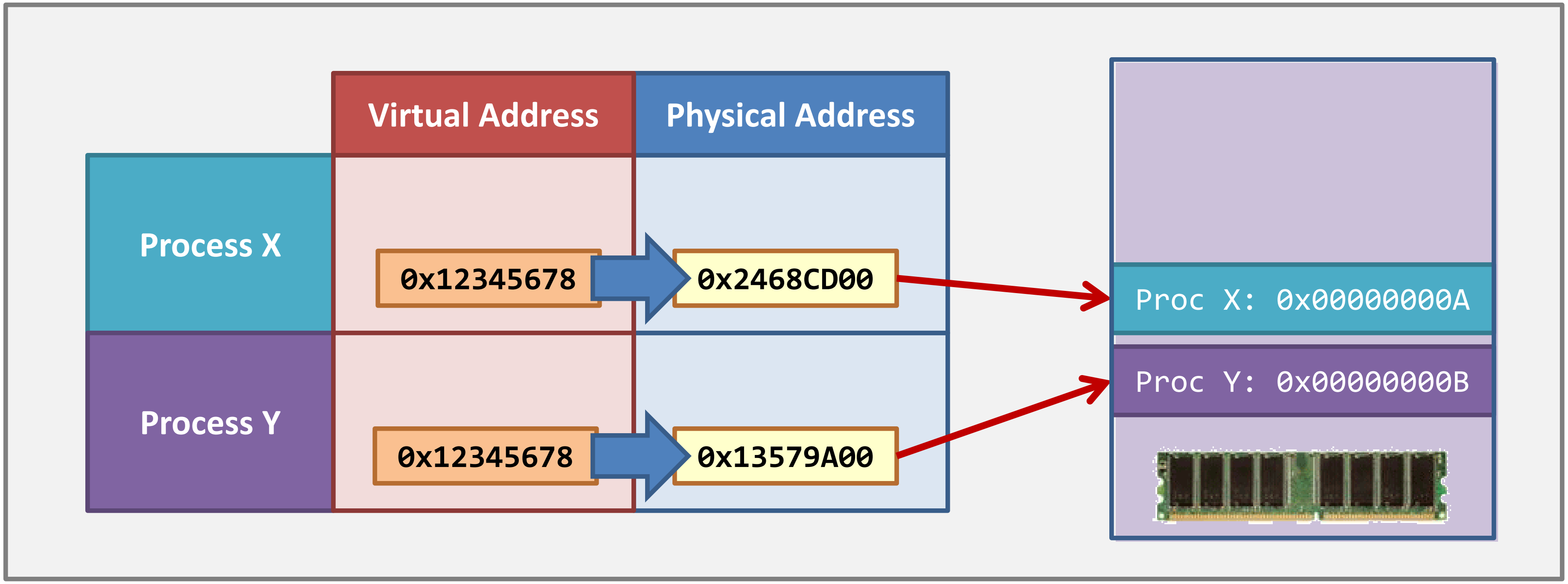
Virtual memory – how does it work?

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



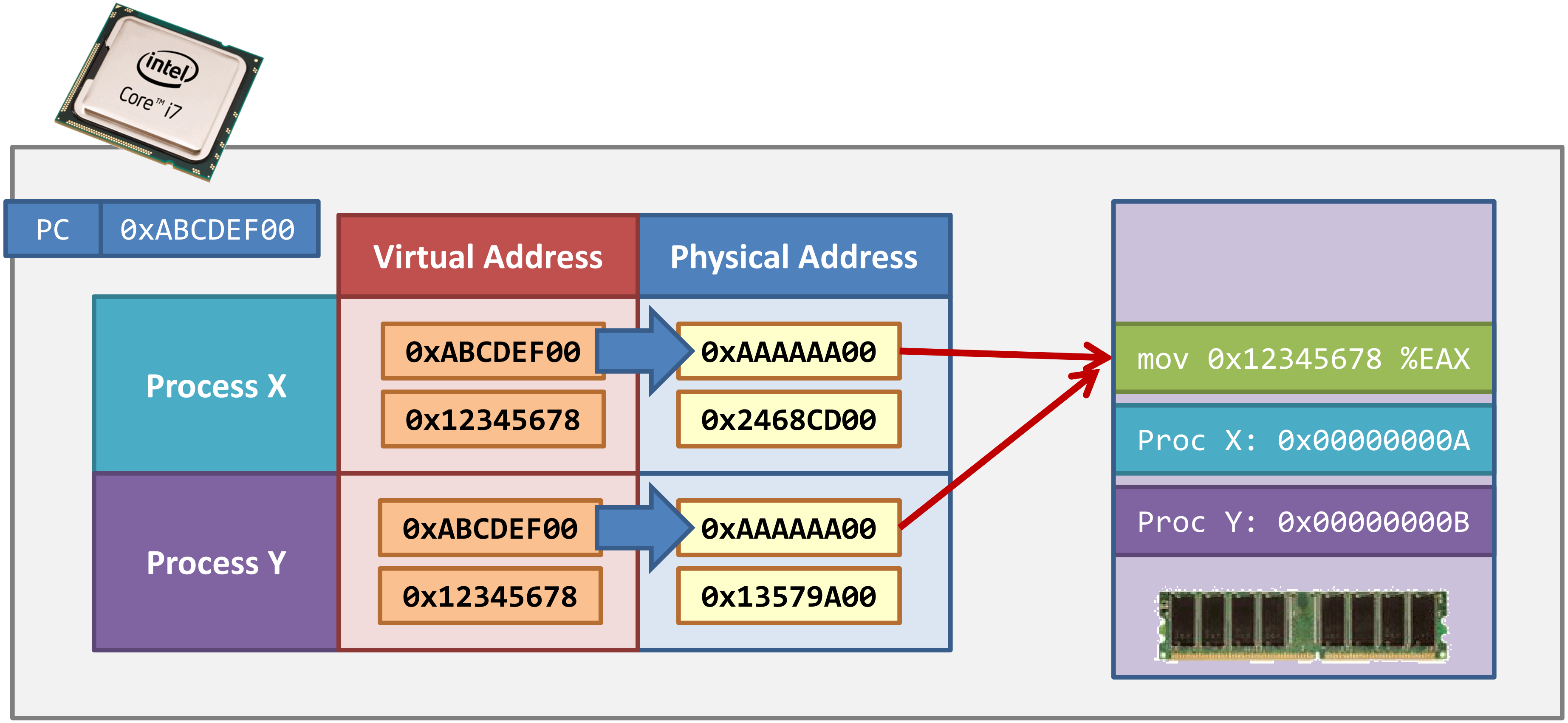
Virtual memory – What is the good?

- **Merit 1.** Different processes use the same virtual addresses, they may be **translated to different physical addresses**.
  - Recall the "**`pid`**" variable in the example using **`fork()`**.
  - The address translation helps the CPU to **retrieve data in a non-contiguous layout** (the process address space is contiguous).



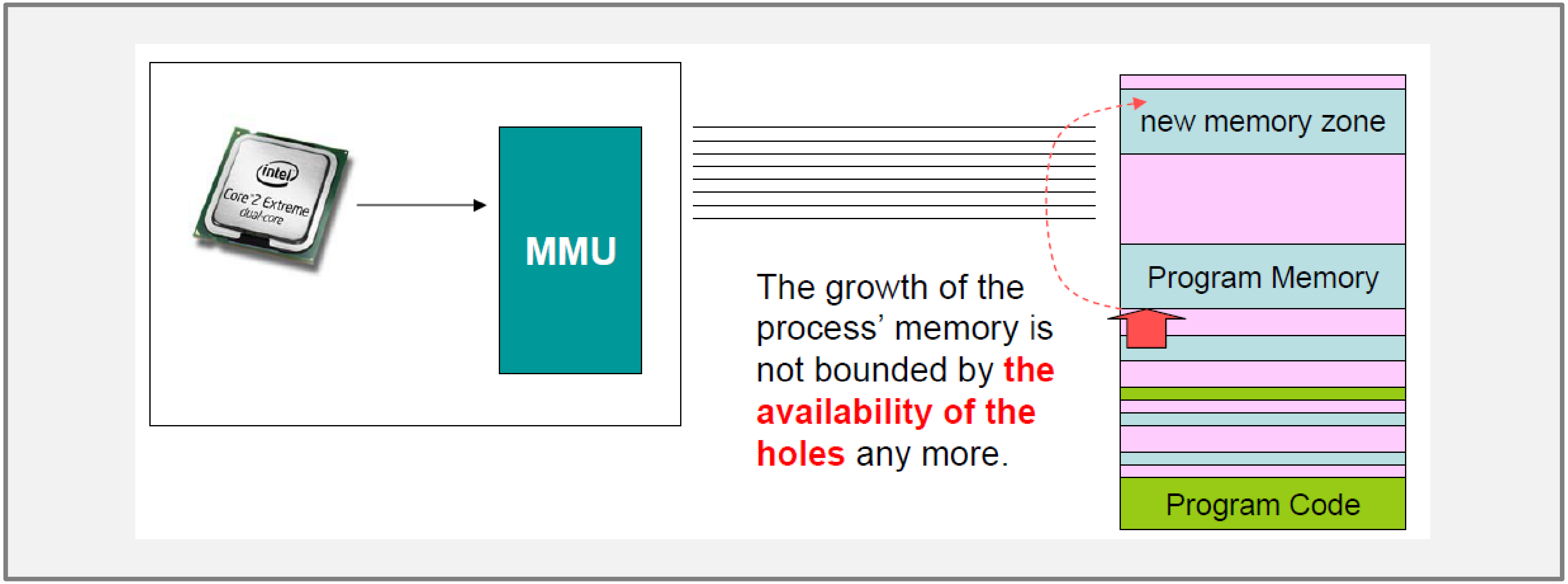
Virtual memory – What is the good?

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



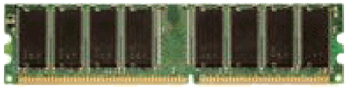
Virtual memory – What is the good?

- **Merit 3.** **Memory growth** can be implemented!
  - When the memory of a process grows, the newly-allocated memory is not required to be contiguous



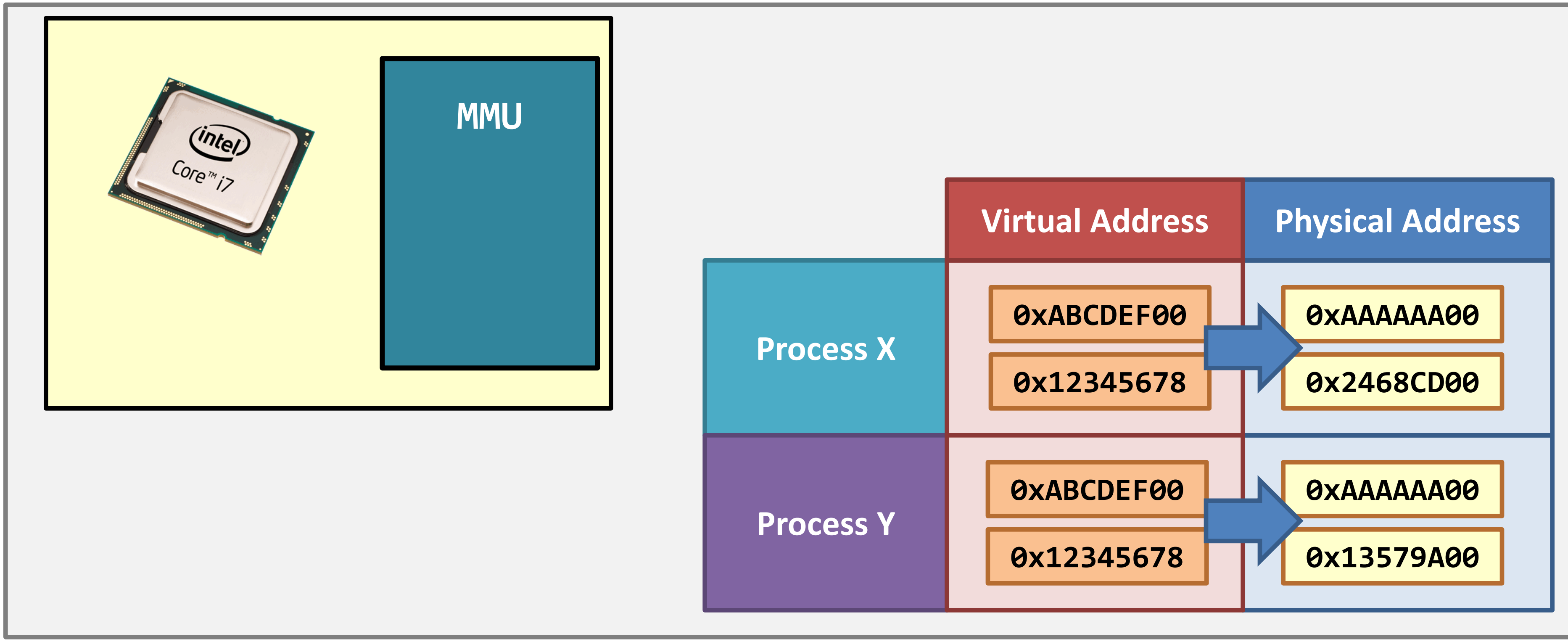
Memory Management

- Virtual memory;
- **MMU implementation & paging;**
- Demand paging;
- Page replacement algorithms;
- Allocation of frames;



MMU implementation

- How to implement the MMU?
  - How to efficiently translate from virtual address to physical address?
  - Translation is needed for every process



MMU implementation – a translation table

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

    (Do you remember the reason?)

MMU

Lookup table

translation

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

Lookup Table internals

What is the problem with this method?

MMU implementation – a translation table

- Then, how large is the lookup table?

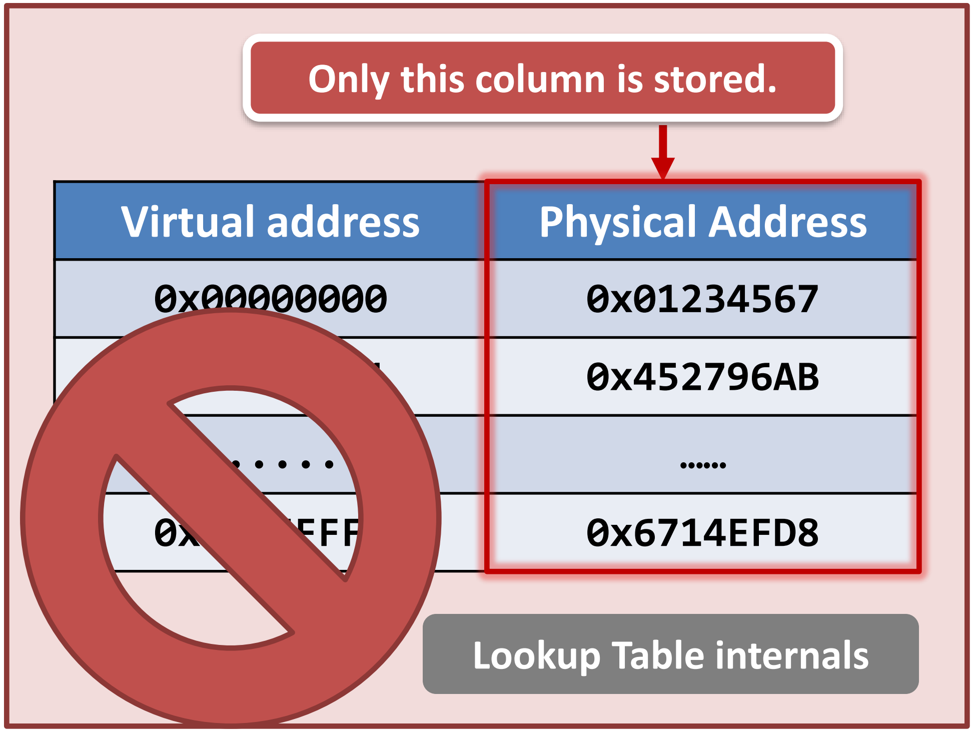
How many addresses are there? **2^{32}**

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



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

Can we reduce the table size?

Note. Every address in a CPU is always of 4 bytes.

The only choice is to reduce the number of addresses

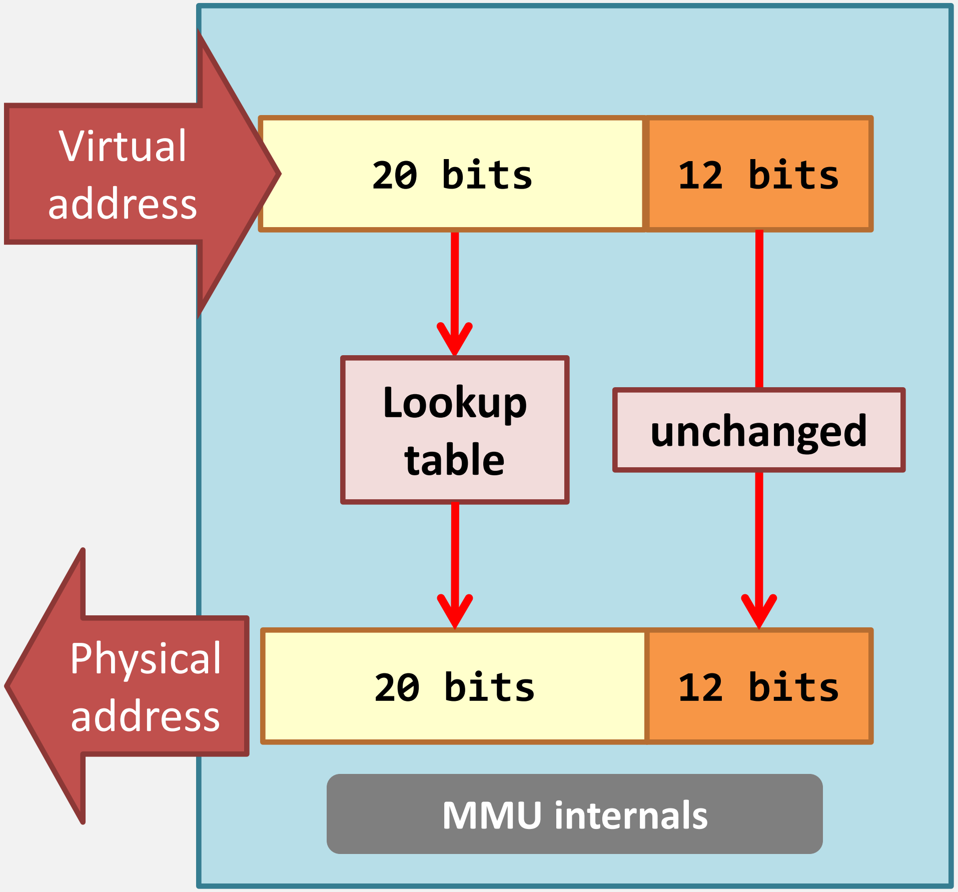
MMU implementation – a partial lookup table

```
Size of the lookup table =

        Number of addresses
      x  Size of an address
```

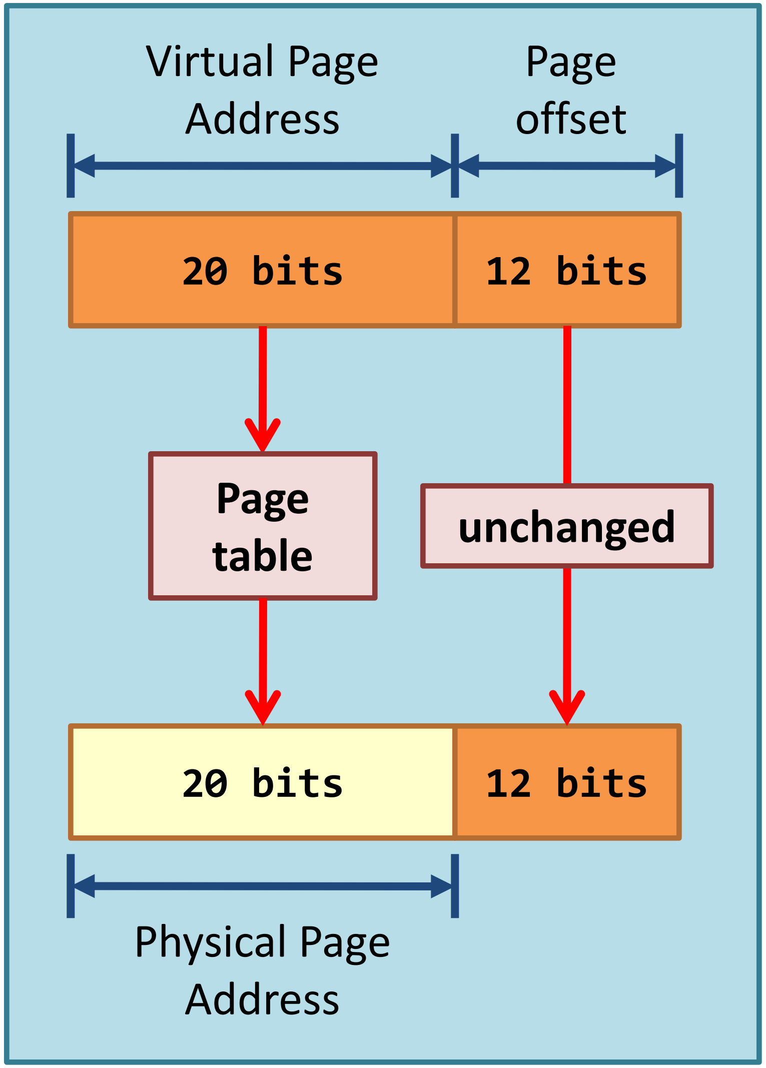
2^{20} x 4 bytes = 4 Mbytes

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

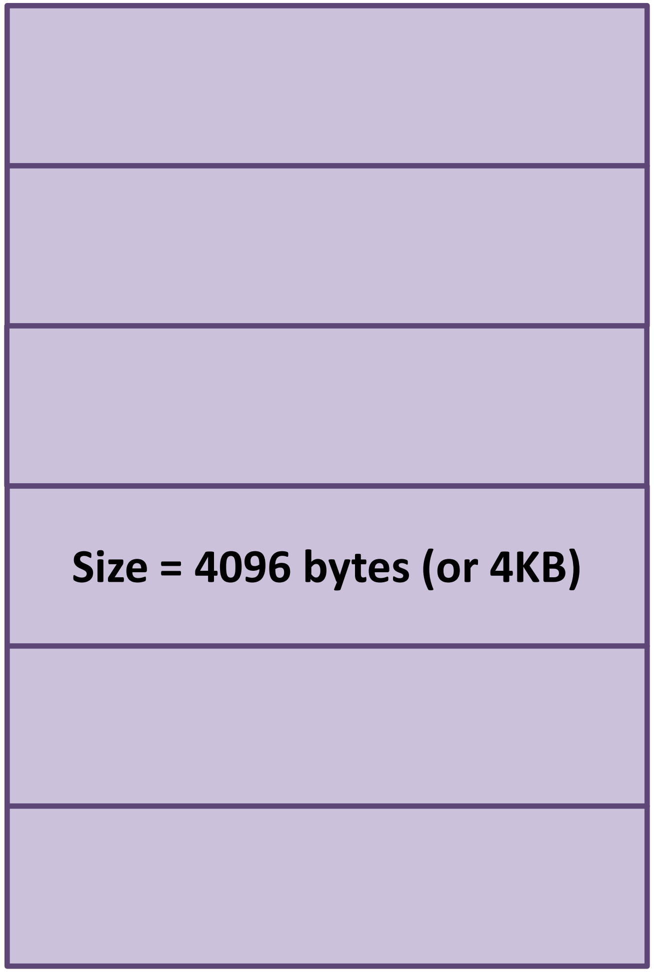


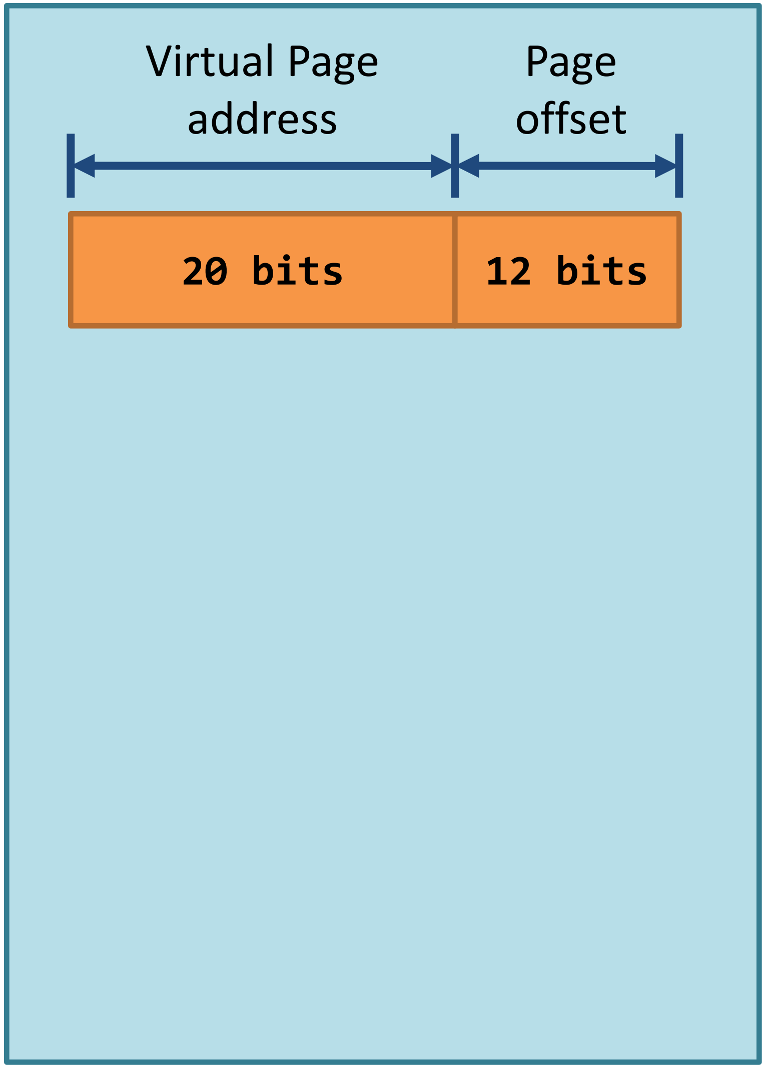
MMU implementation – paging

- 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**.

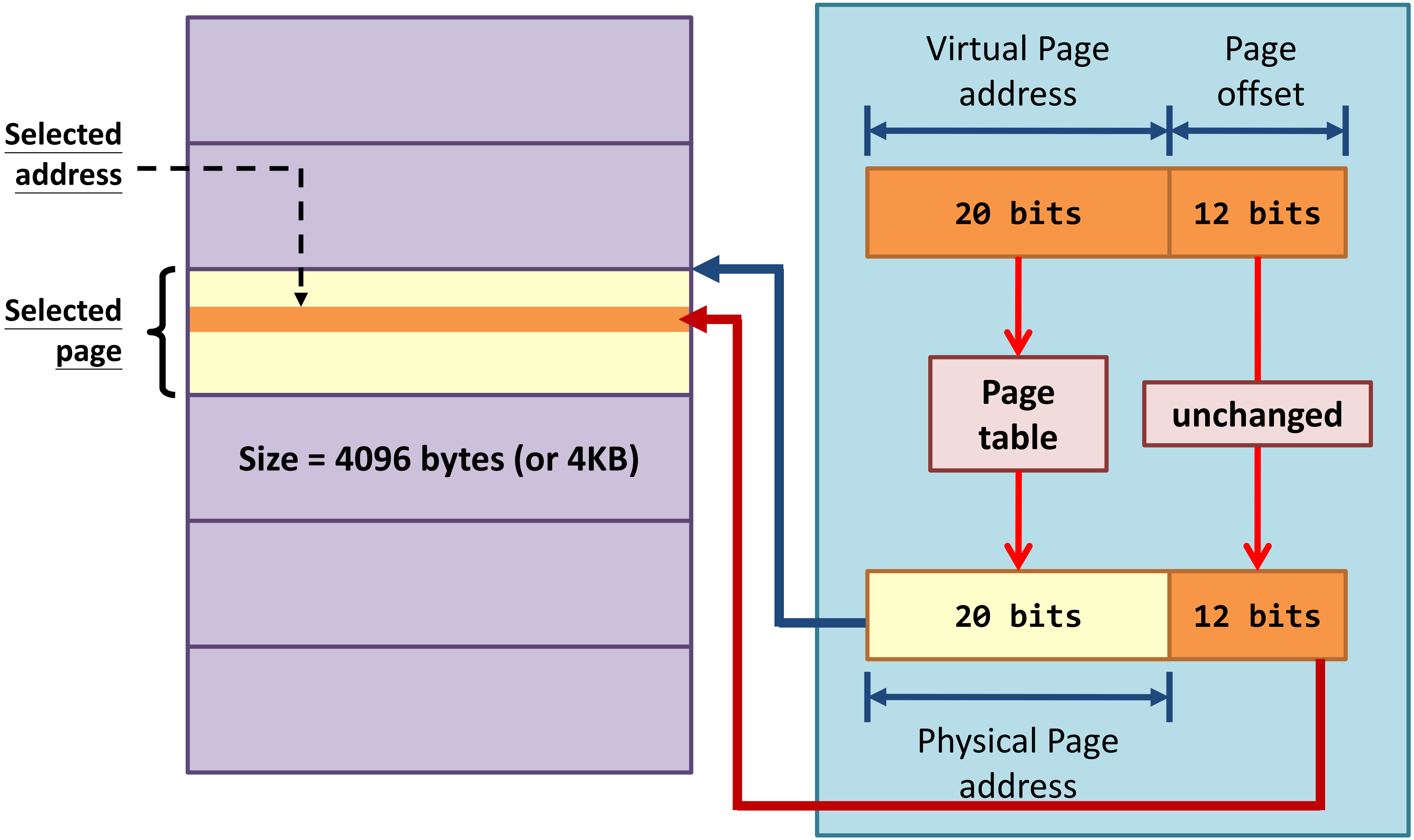


Paging - properties



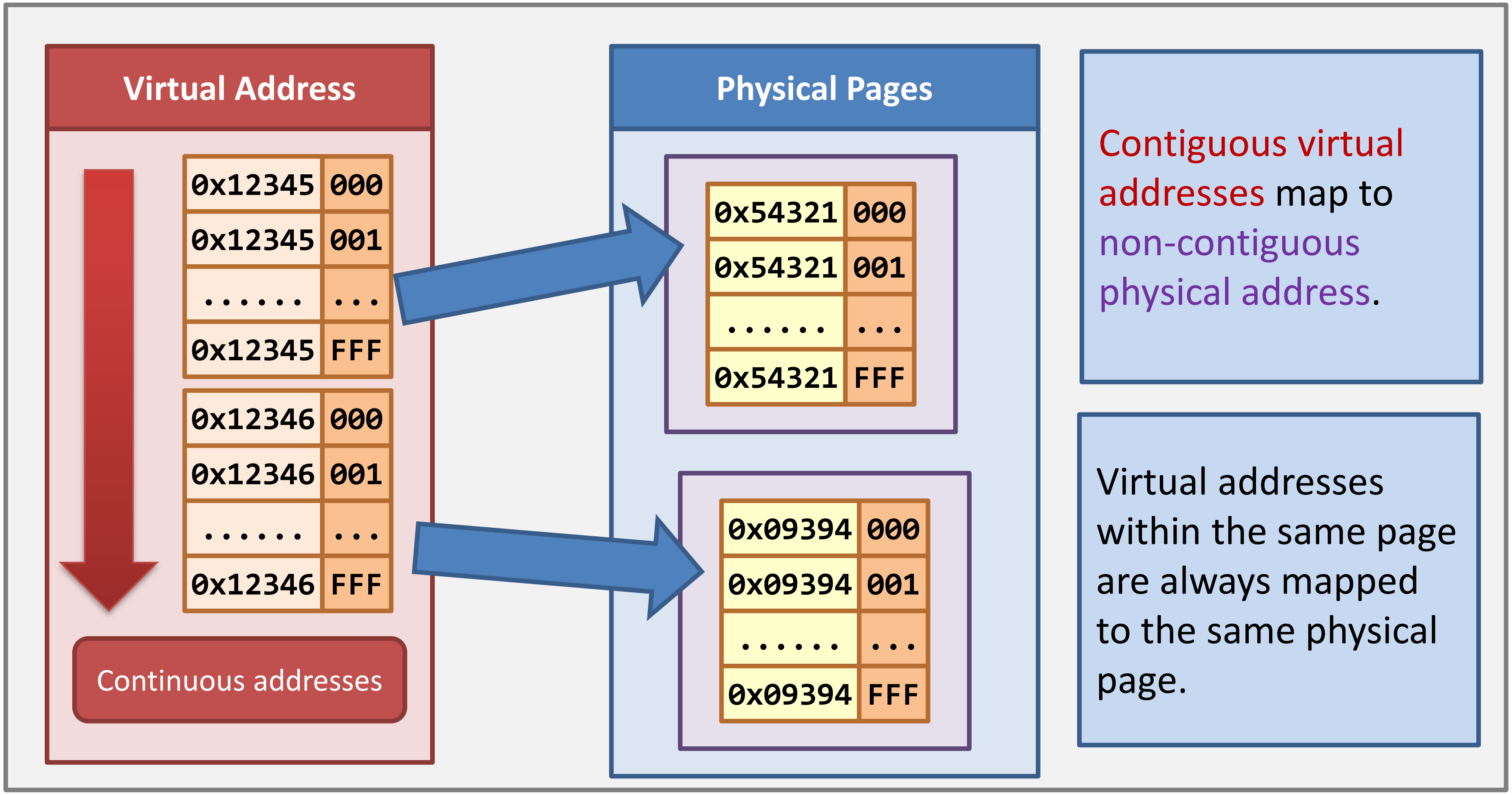


Paging - properties



Paging - properties

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



Paging – memory allocation

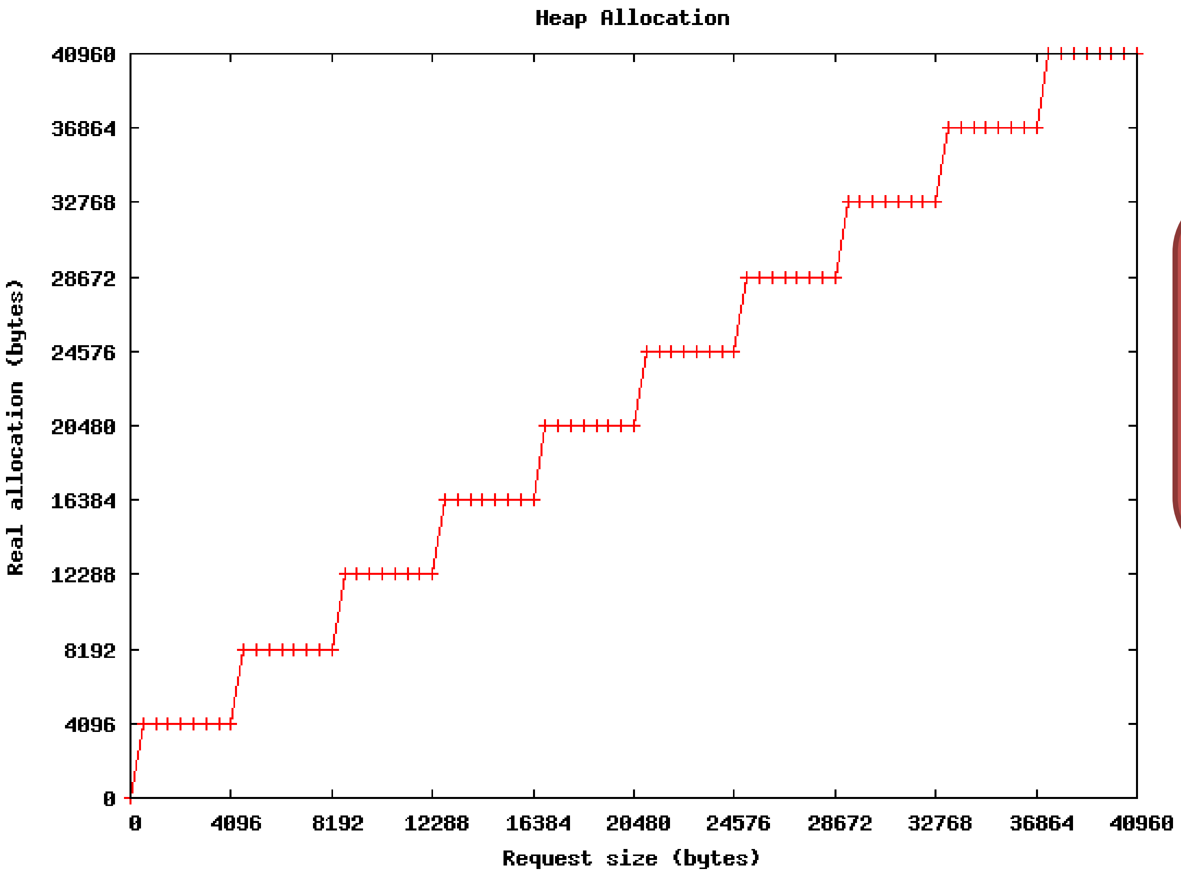
- How to do memory allocation with paging

```
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);
}
```

Paging – memory allocation

- A page is the basic unit of memory allocation.



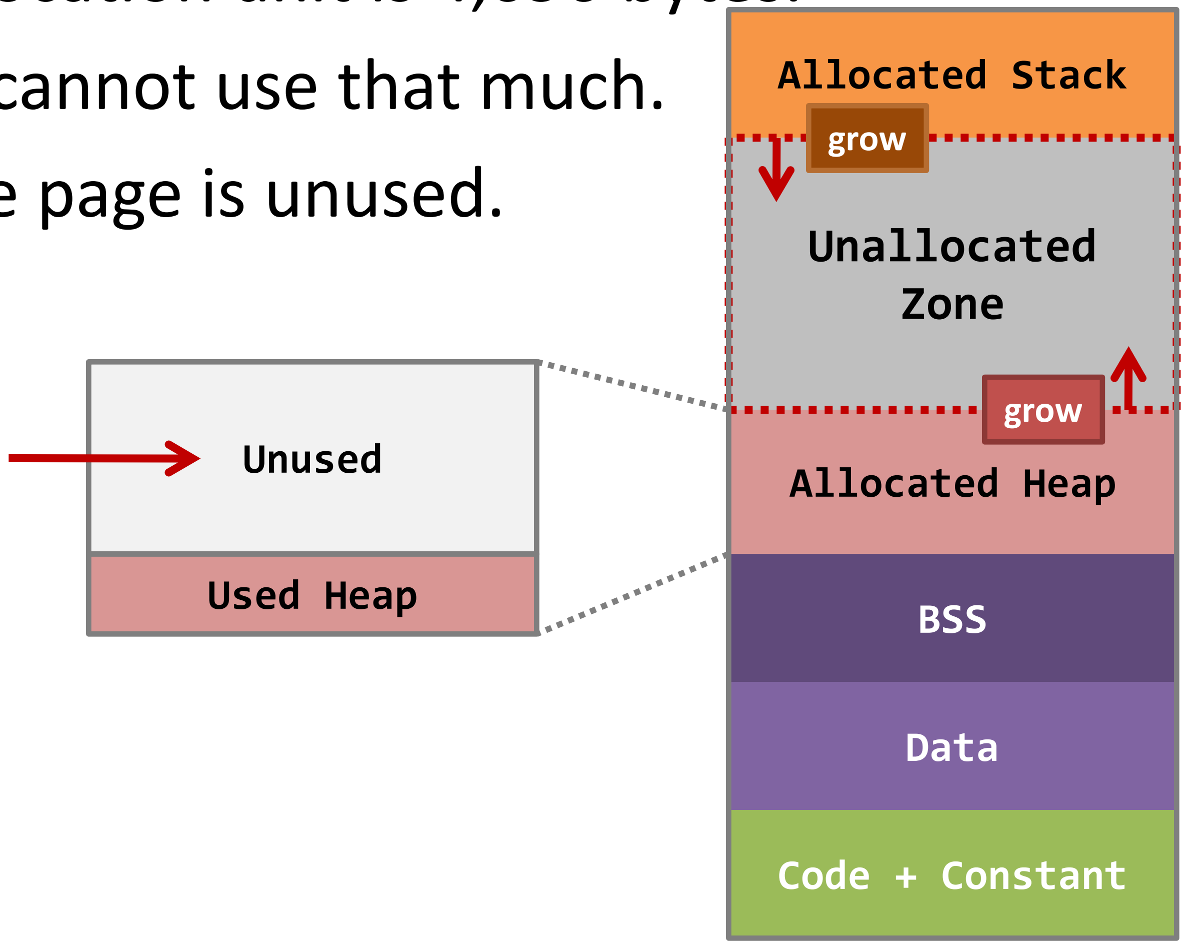
The allocation is in a **page-by-page** manner.

The same case for the growth of the stack.

Paging – memory allocation

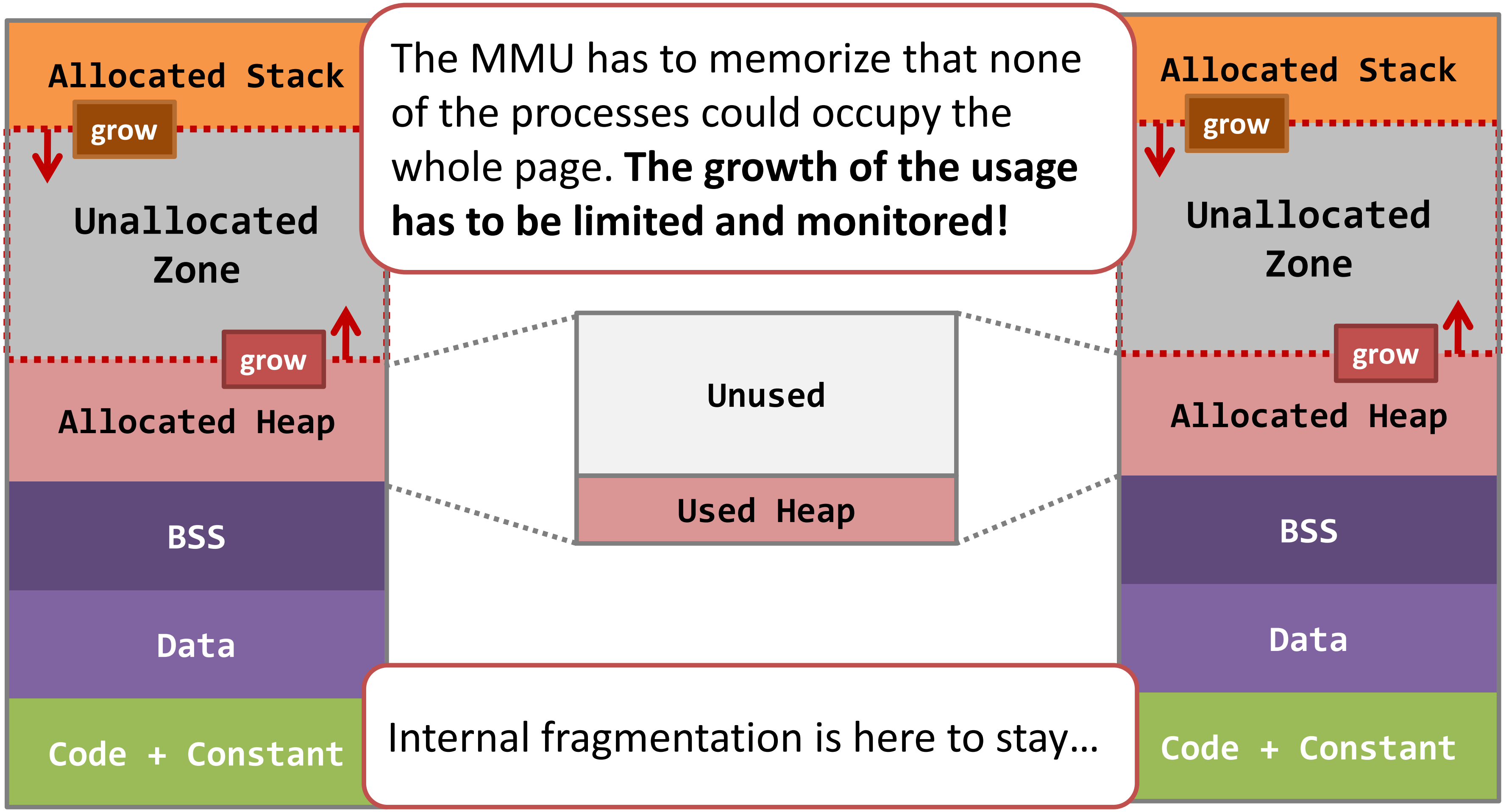
- Problem???
  - The minimum allocation unit is 4,096 bytes.
  - But, the process cannot use that much.
  - So, the rest of the page is unused.

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



Paging – internal fragmentation

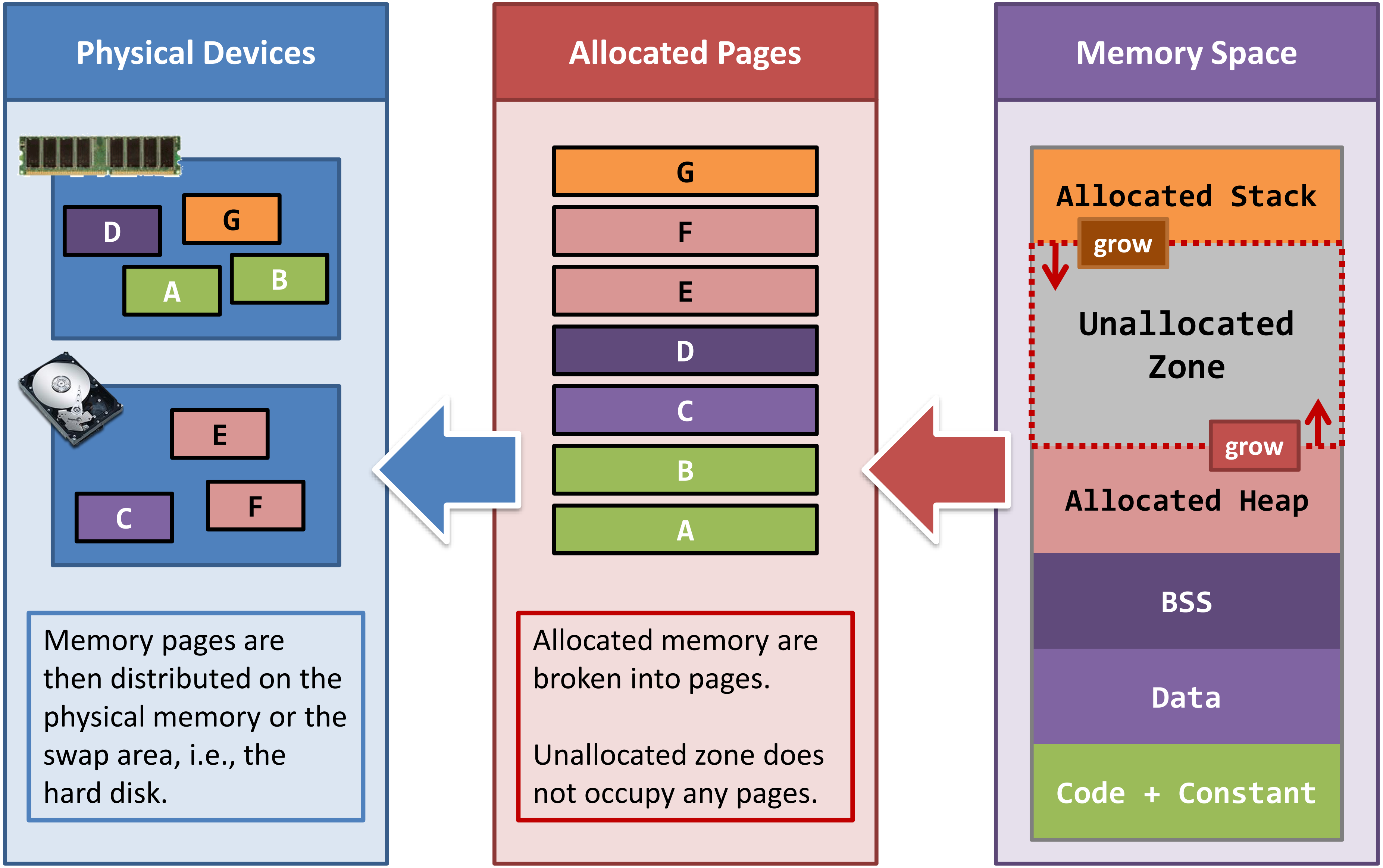
How about letting another process to use the "unused space"?



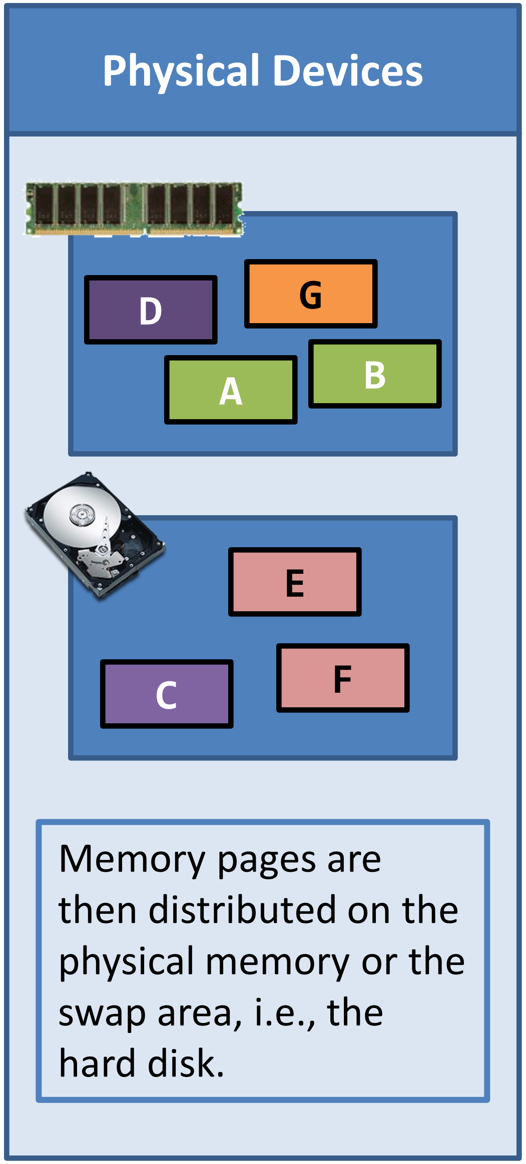
The MMU has to memorize that none of the processes could occupy the whole page. **The growth of the usage has to be limited and monitored!**

Internal fragmentation is here to stay…

Paging – putting it together



Paging – page table design



- 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

- How to design the page table?
  - First of all, which information need to be maintained?
    - Mapping from virtual pages to physical pages (called frames)
    - Permission information
    - Where is the page (in memory or not)
  - Second…
    - Each process needs one 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.`

How does 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 – page table design

- Other design issues

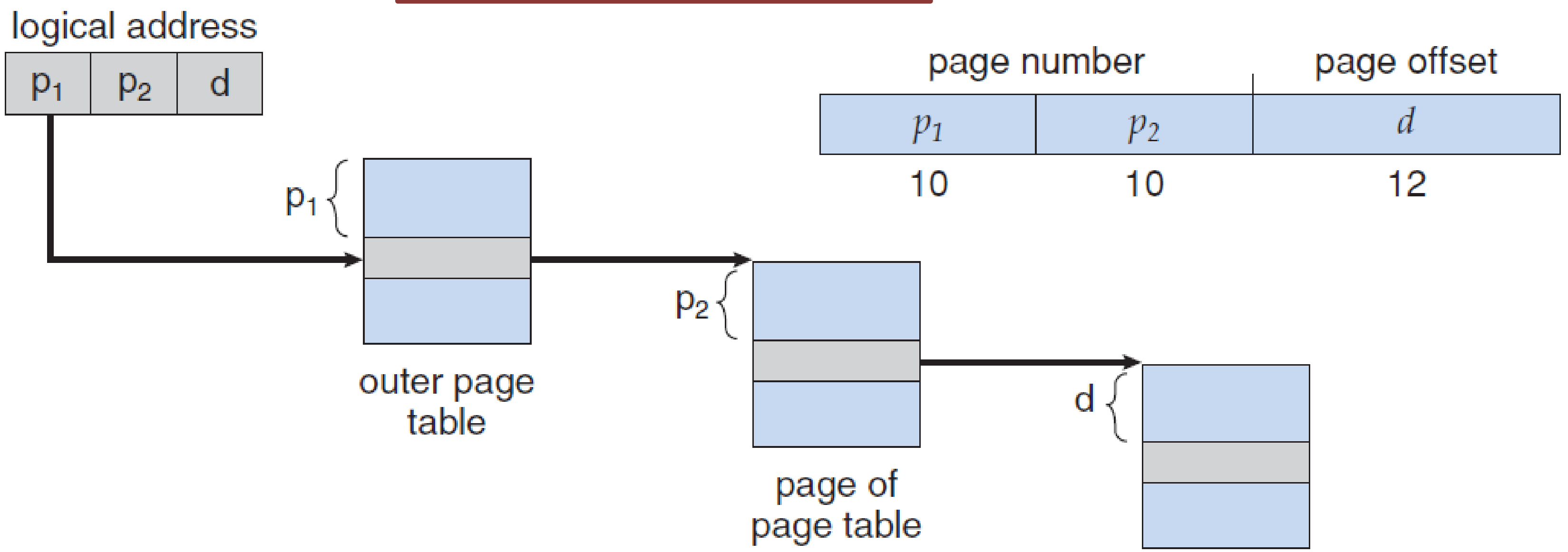
How to store the page table if it is large (structure of page table)?

How to improve memory access performance (page table look incurs large overhead)?
Caching: Translation lookaside buffer (TLB)

Paging – page table structure

- The page table may be large…multiple MBs
  - We would not want to allocate the page table contiguously in memory, how?
  - Divide the page table into pieces

Two-level page table



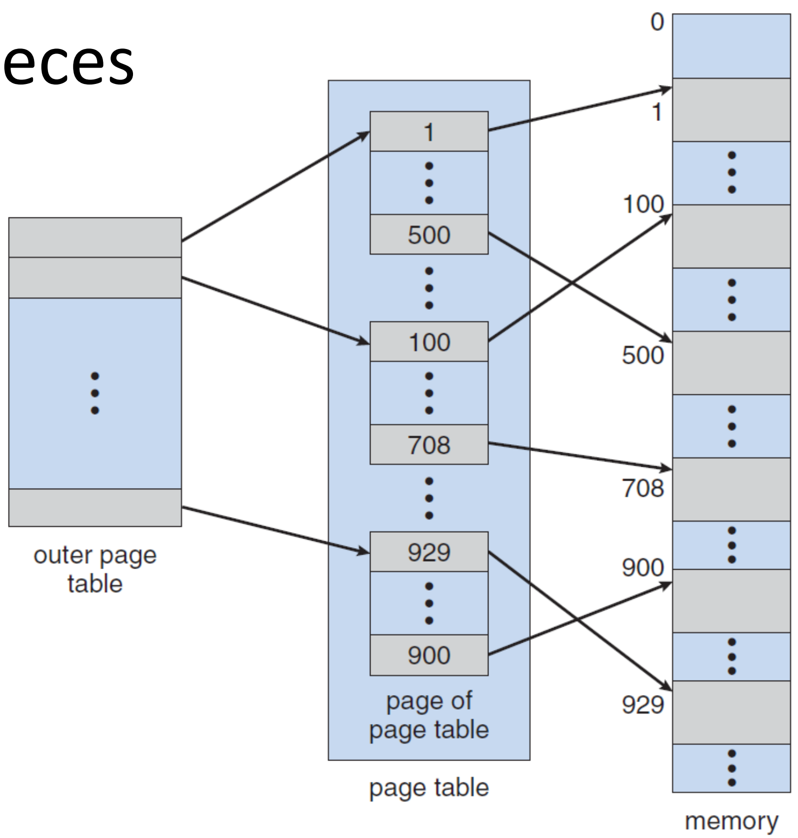
Paging – page table structure

- The page table may be large…multiple MBs
  - We would not want to allocate the page table contiguously in memory, how?
  - Divide the page table into pieces

Besides hierarchical paging, we can also use

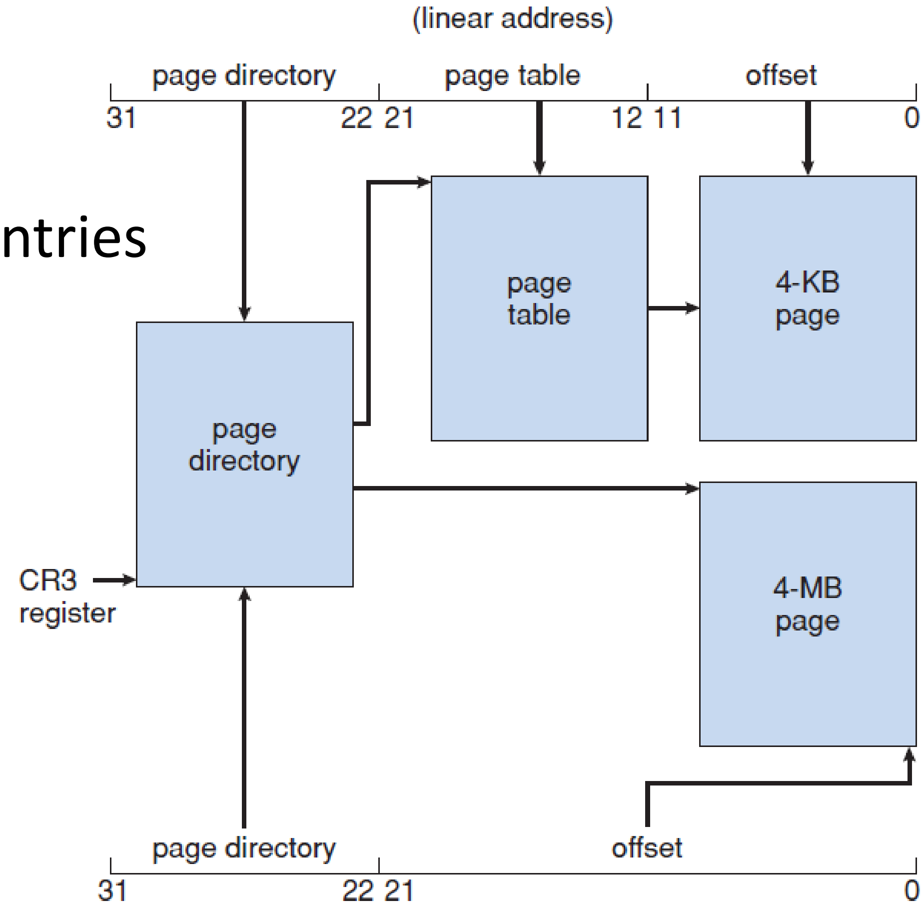
Hashed page tables

Inverted page tables



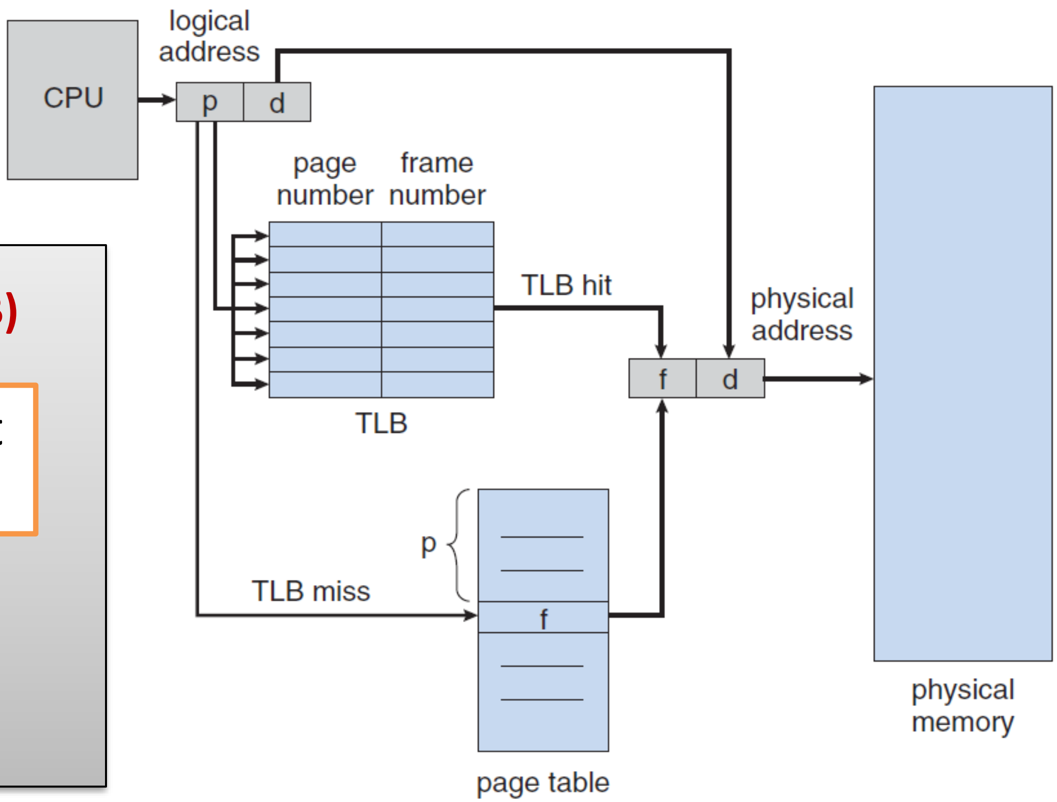
Paging – Performance Boost

- Memory access requires to look up page table
  - This overhead is even larger with multi-level page tables
  - Any solution?
  - **(1) large pages**
    - Reduce the page table entries
    - Cons?
      - Internal fragmentation
      - Deduplication



Paging – Performance Boost

- Memory access requires to look up page table
  - This overhead is even larger with multi-level page tables
  - Any solution?
  - **(2) Caching**



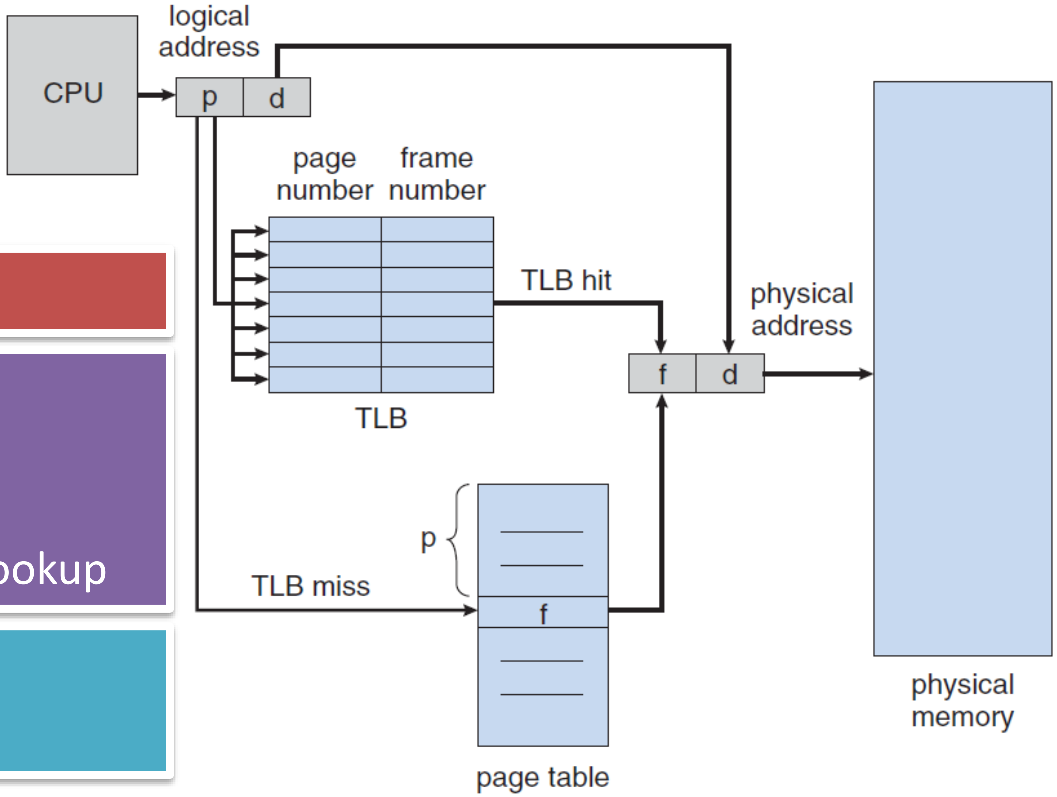
Translation lookaside buffer (TLB)

The search in TLB is fast: Part of the instruction pipeline

The size of TLB is small: e.g., 32-1024 entries

Paging – Performance Boost

- Memory access requires to look up page table
  - This overhead is even larger with multi-level page tables
  - Any solution?
  - **(2) Caching**



Effective memory-access time

Example：

- Hit ratio: 80%
- Mem access time: 100 ns
- One mem access for page table lookup

Effective mem-access time is
0.8*100+0.2*(100+100)=120ns

Paging – summary

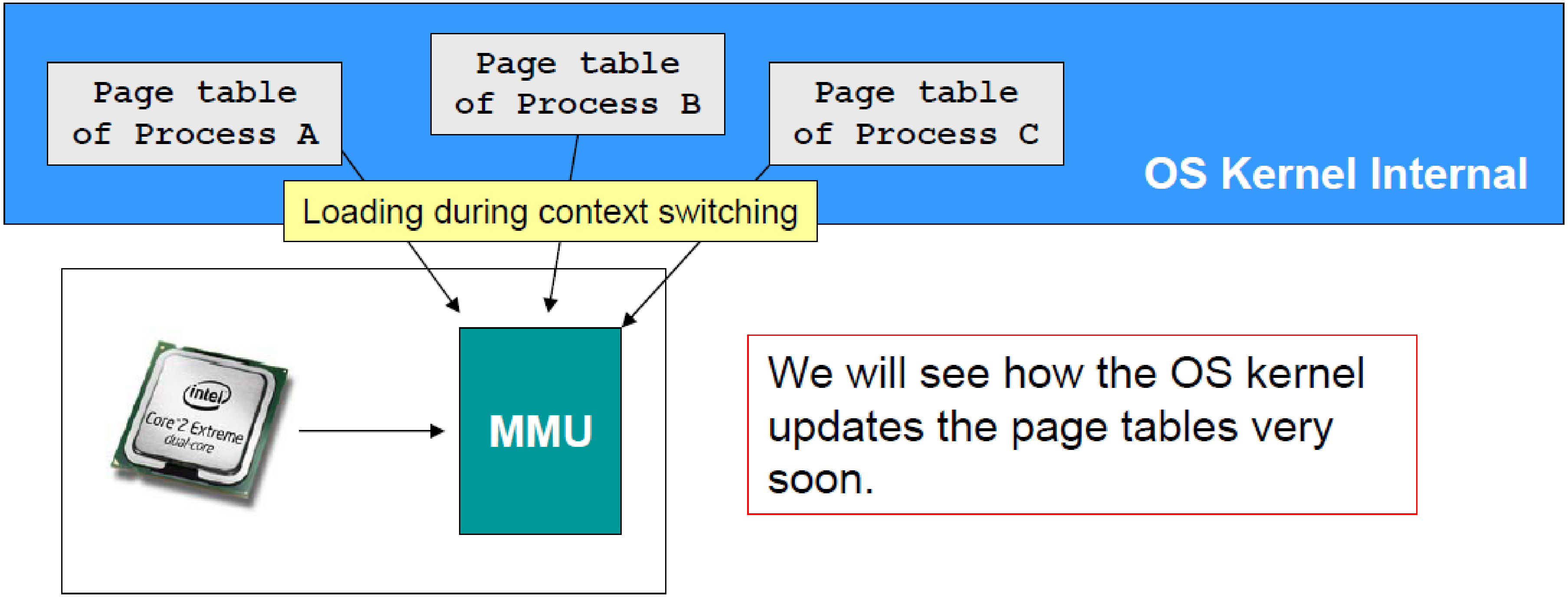
- Virtual memory (VM) is just a table-lookup implementation. The specials about VM are:

  – The table-lookup is implemented inside the CPU, i.e., a hardware solution.

  – **Each process should have its own page table**.

Paging – summary

- How about the OS?
  - The OS stores and manages the page tables of all processes.



We will see how the OS kernel updates the page tables very soon.

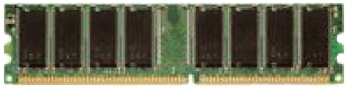
Paging – summary

- We talked about **segmentation** in part 1…
  - Address mapping can also be done in segments
    - Also permits physical address space of a process to be non-contiguous
    - But usually incurs severe **fragmentation** in both memory and backing store

- **Paging is used in most operating systems**
  - Hybrid scheme is also possible

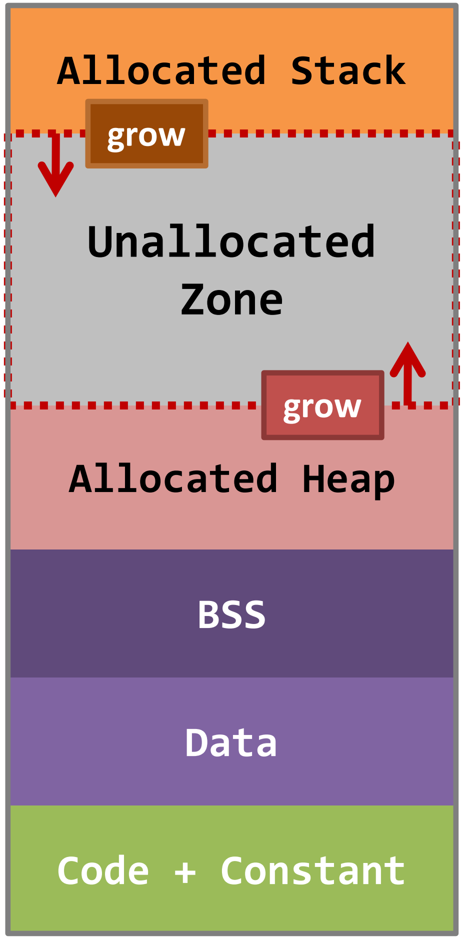
Memory Management

- Virtual memory;
- MMU implementation & paging;
- **Demand paging;**
- Page replacement algorithms;
- Allocation of frames;



Memory / page allocation?

- 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;
- The question is…
  - When will the memory be allocated for you when you call **malloc()**?



Remember the OOM generator?

```
#define ONE_MEG  1024 * 1024

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

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

    return 0;
}
```

```
Allocated 3052 MB
Allocated 3053 MB
Allocated 3054 MB
Allocated 3055 MB
linux2:/uac/rshr/ykli>
```

This program runs very fast, why?

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.

Memory / allocation – demand paging

- How about the heap?

```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);
    }
}
```

grow_heap.c

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.

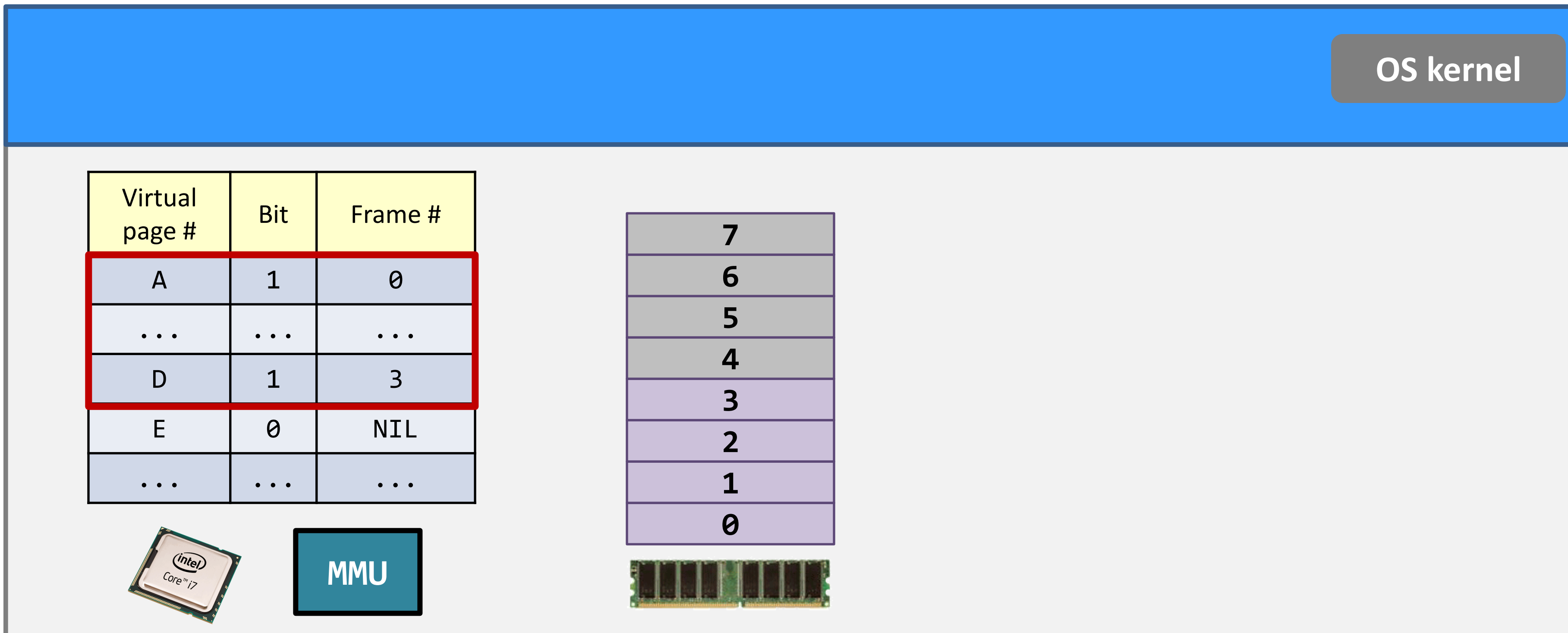
So, this statement really **asks the system** to allocate memory.

This lazy way is called **demand paging**, but how does it work?

Demand paging – illustration.

Assumption: 1 process only.

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

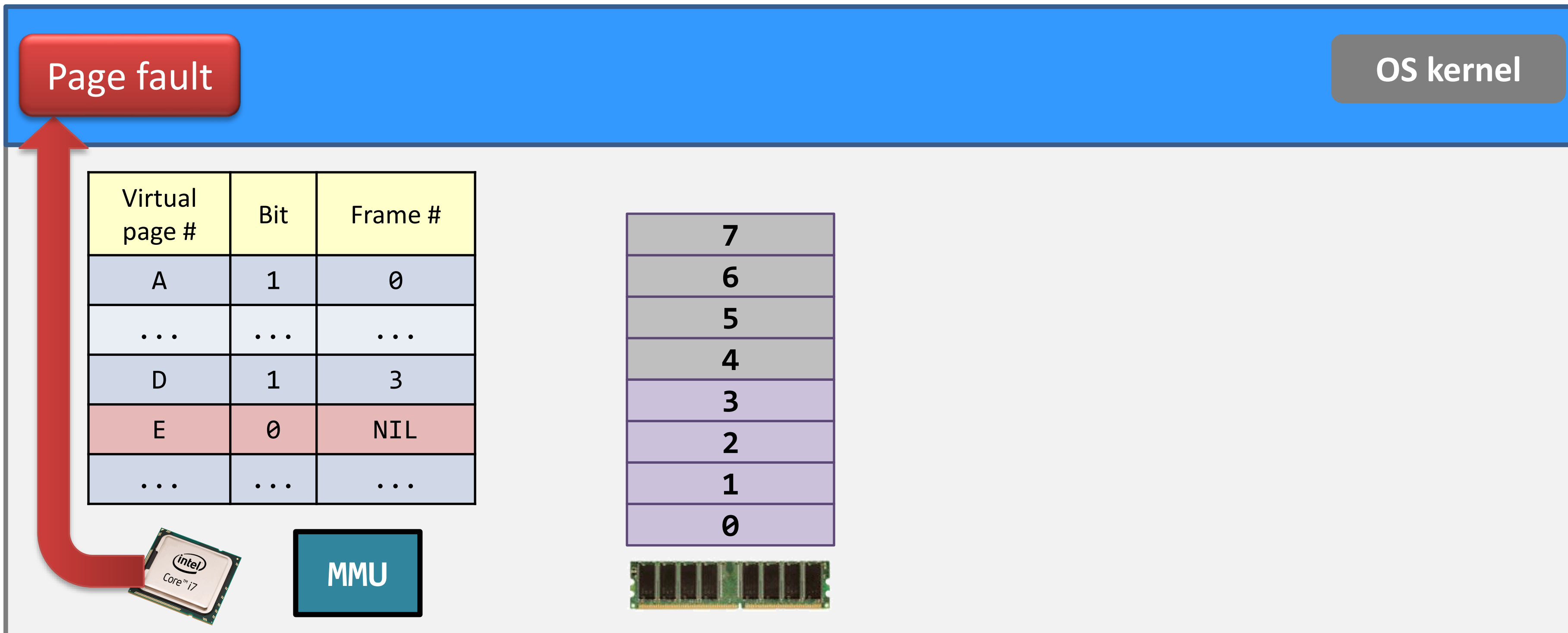


Virtual page #	Bit	Frame #
A	1	0
...	...	...
D	1	3
E	0	NIL
...	...	...

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**.

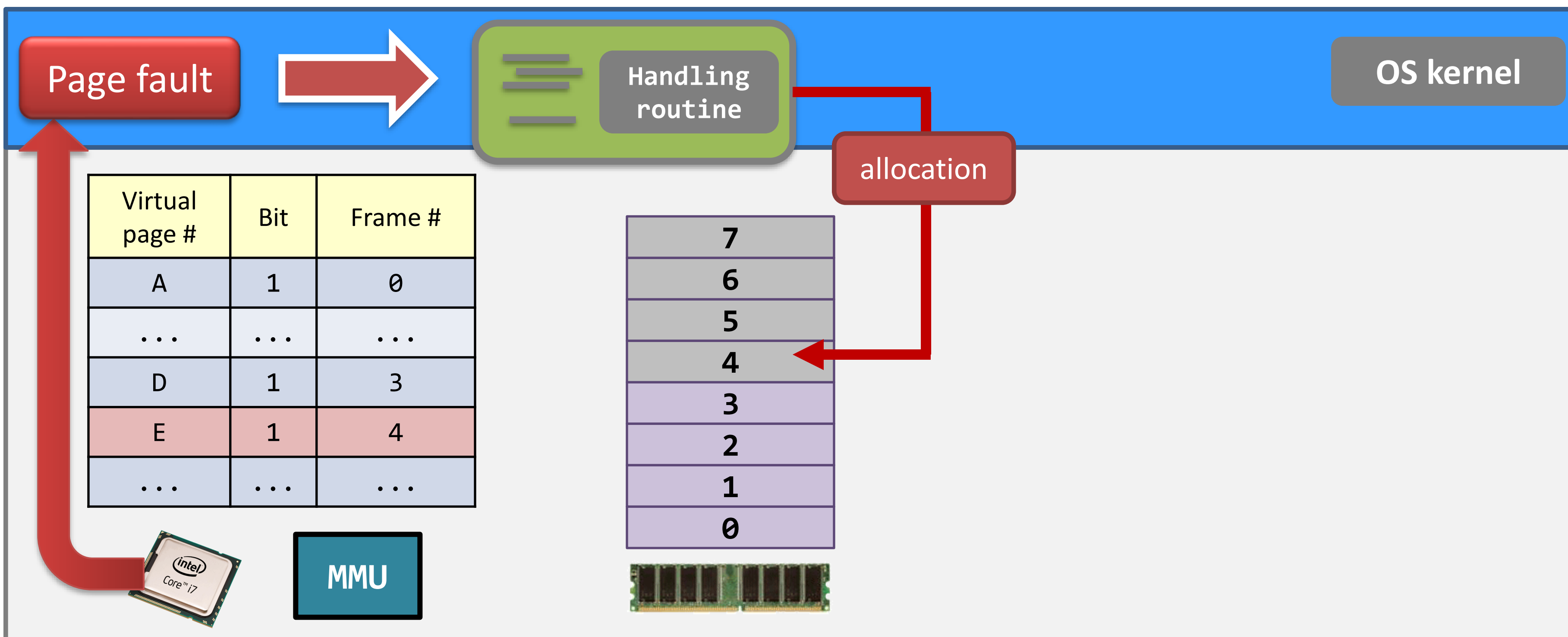


Virtual page #	Bit	Frame #
A	1	0
...	...	...
D	1	3
E	0	NIL
...	...	...

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.



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

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.

OS kernel

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

OK

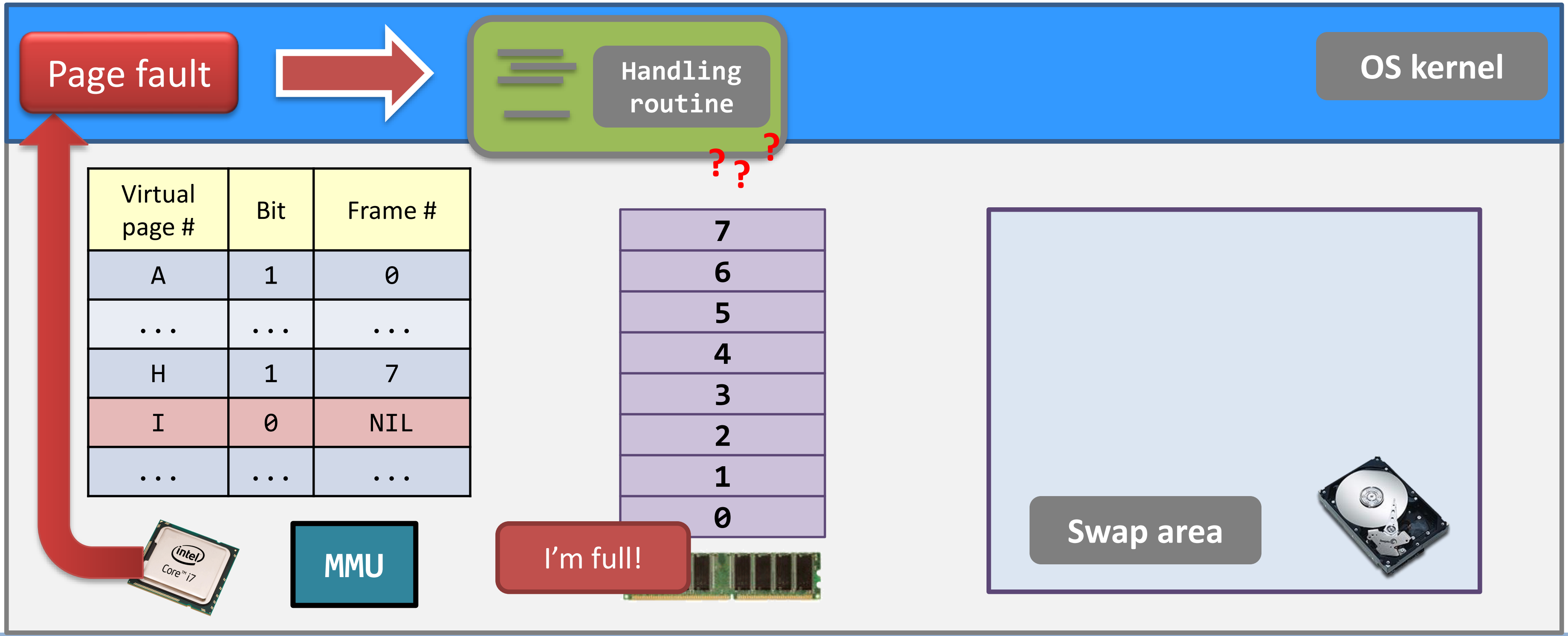
7
6
5
4
3
2
1
0

MMU

Demand paging – illustration.

Assumption: 1 process only.

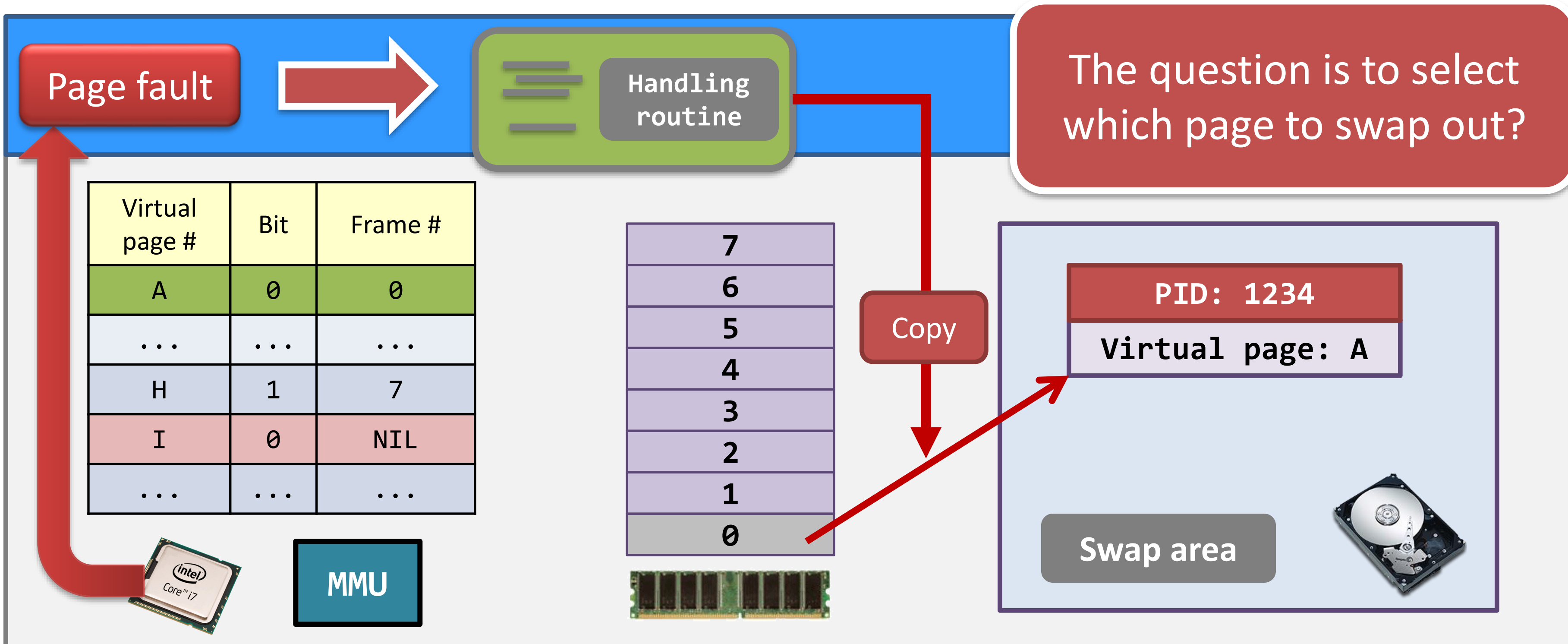
- So, how about the case when the routine finds that **all frames are allocated**?
  - Then, we need the help of the **swap area**.



Demand paging – illustration.

Assumption: 1 process only.

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

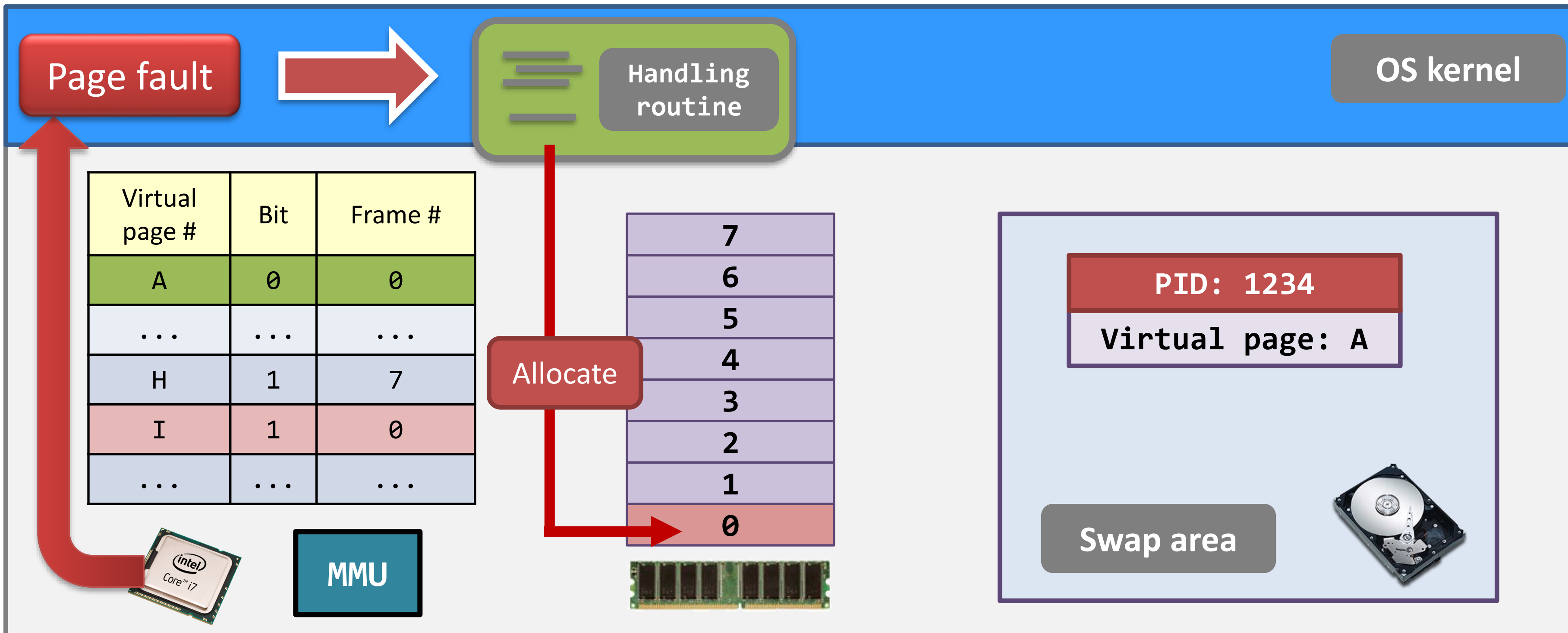


Virtual page #	Bit	Frame #
A	0	0
...	...	...
H	1	7
I	0	NIL
...	...	...

Demand paging – illustration.

Assumption: 1 process only.

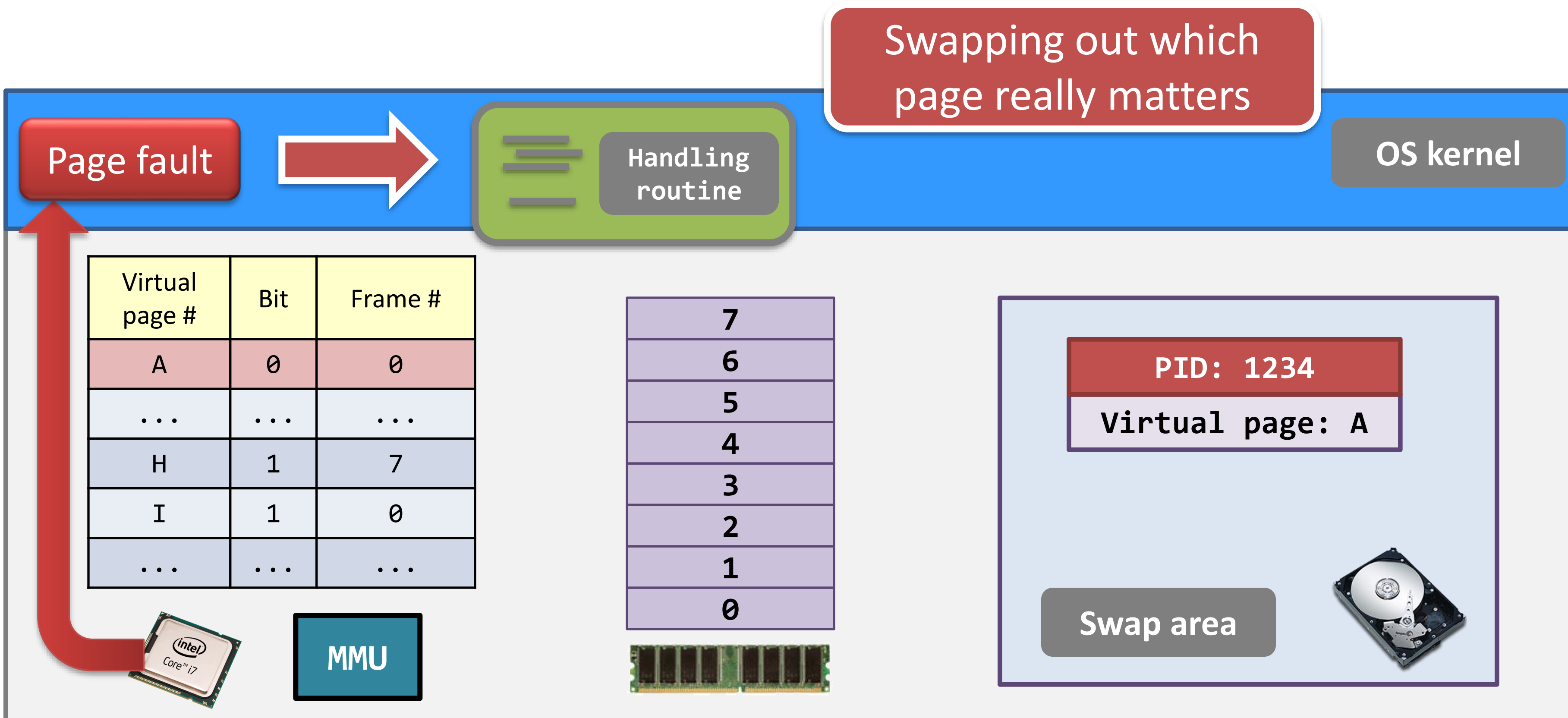
- Using the swap area:
  - Step (2) Allocate the free frame to the new frame allocation request.
    - Now, Page I takes Frame 0.



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.



OOM generator

- Now, you should understand why this OOM generator run very fast.

```
#define ONE_MEG  1024 * 1024

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

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

    return 0;
}
```

The memory page frames are not really allocated (demand paging).

It is only for enlarging the virtual page allocation.

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.

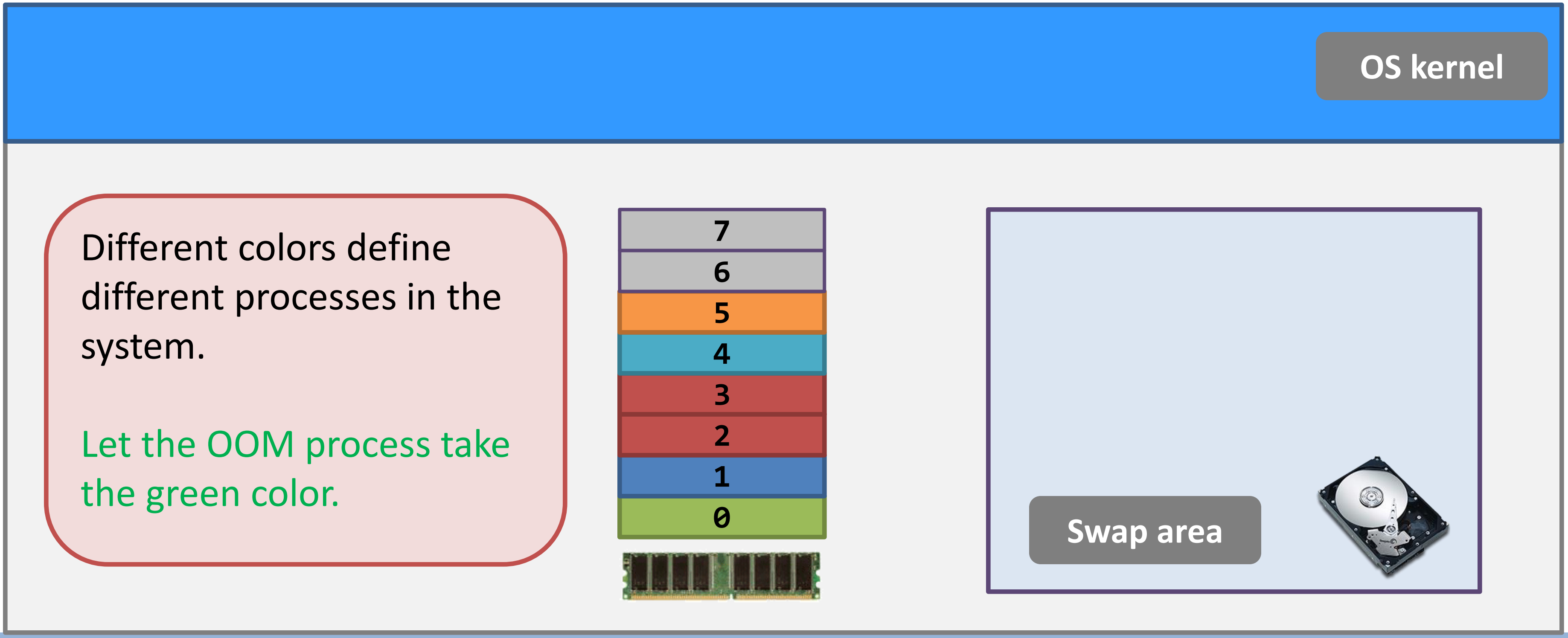
How does this program "eat" your memory?

What is the consequence after running this program?

Real OOM – illustration

`Real oom: running`

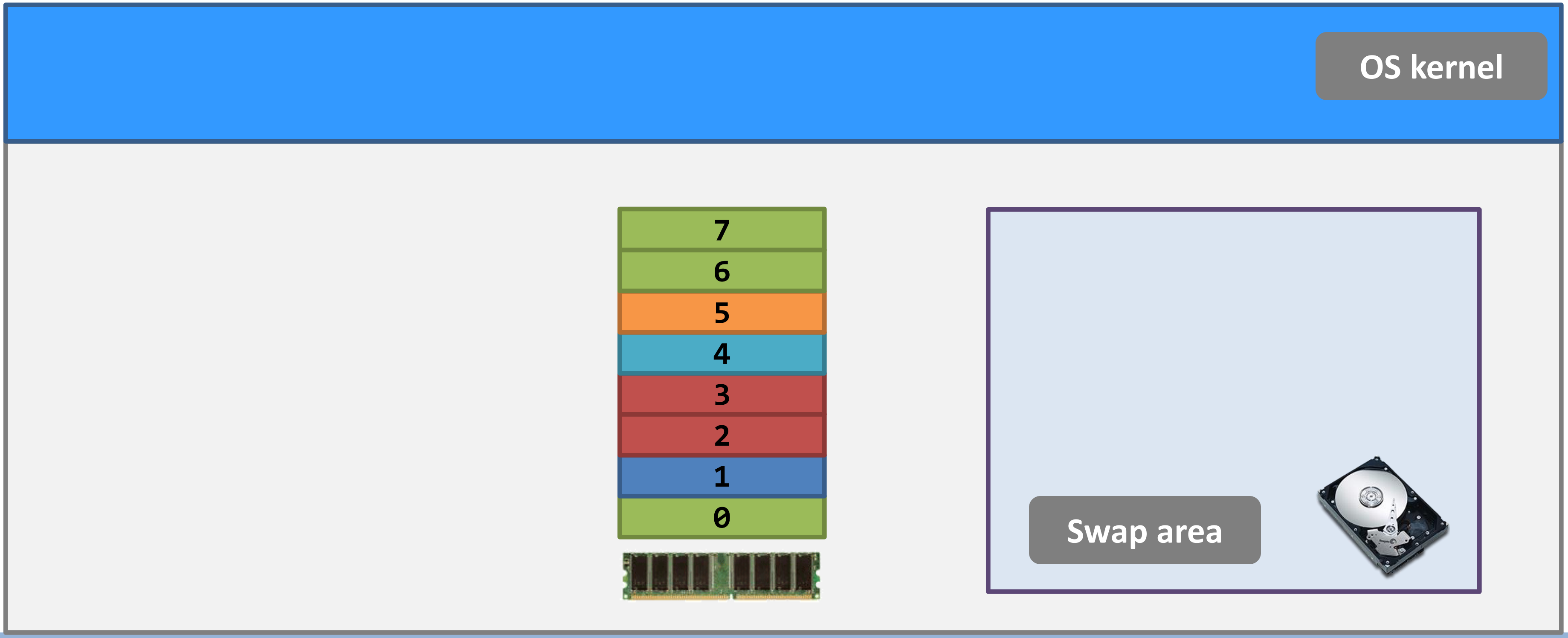
- So, what will happen when the real OOM program is running?
  - Suppose the OOM program has just started with **only one page allocated**. (For illustration only!)



Real OOM – illustration

Real oom: running

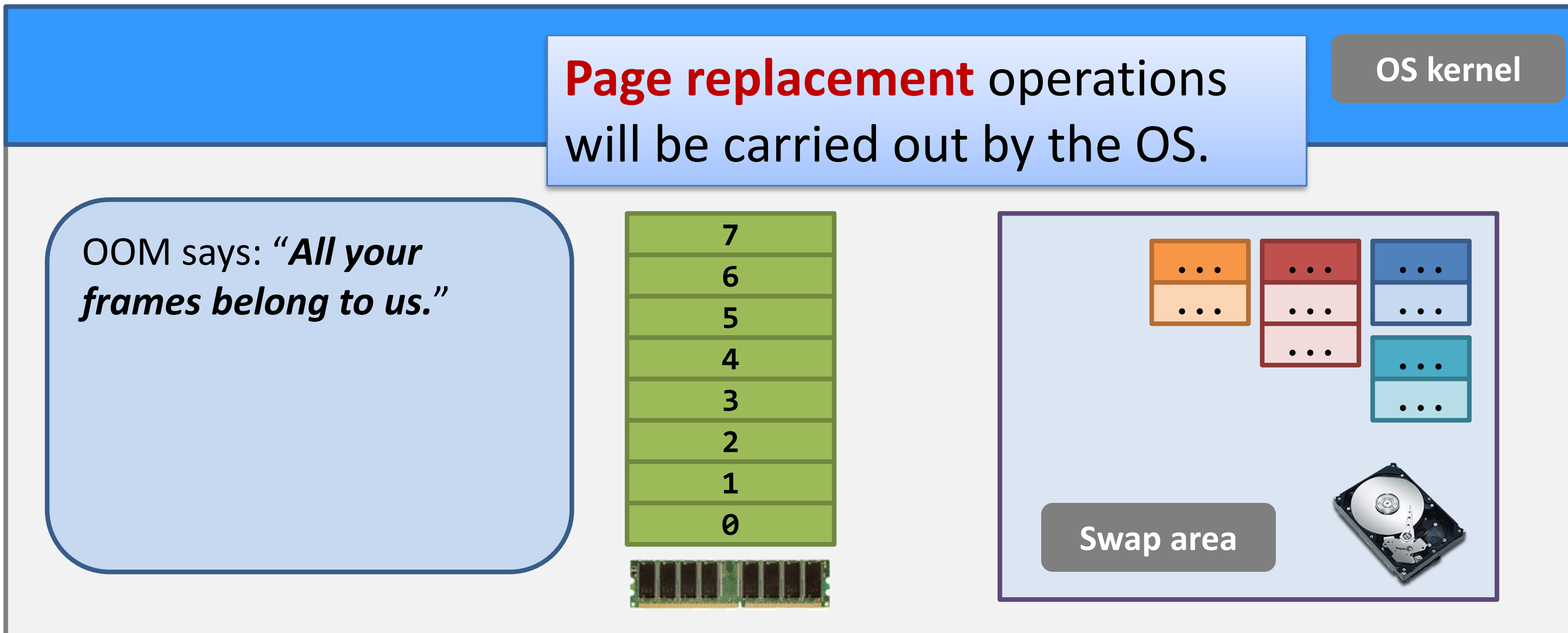
- OOM is running…**<u>1st stage</u>**.
  - The **free memory frames** are the first zone that the process has conquered.
  - All other processes could hardly allocate pages.



Real OOM – illustration

`Real oom: running`

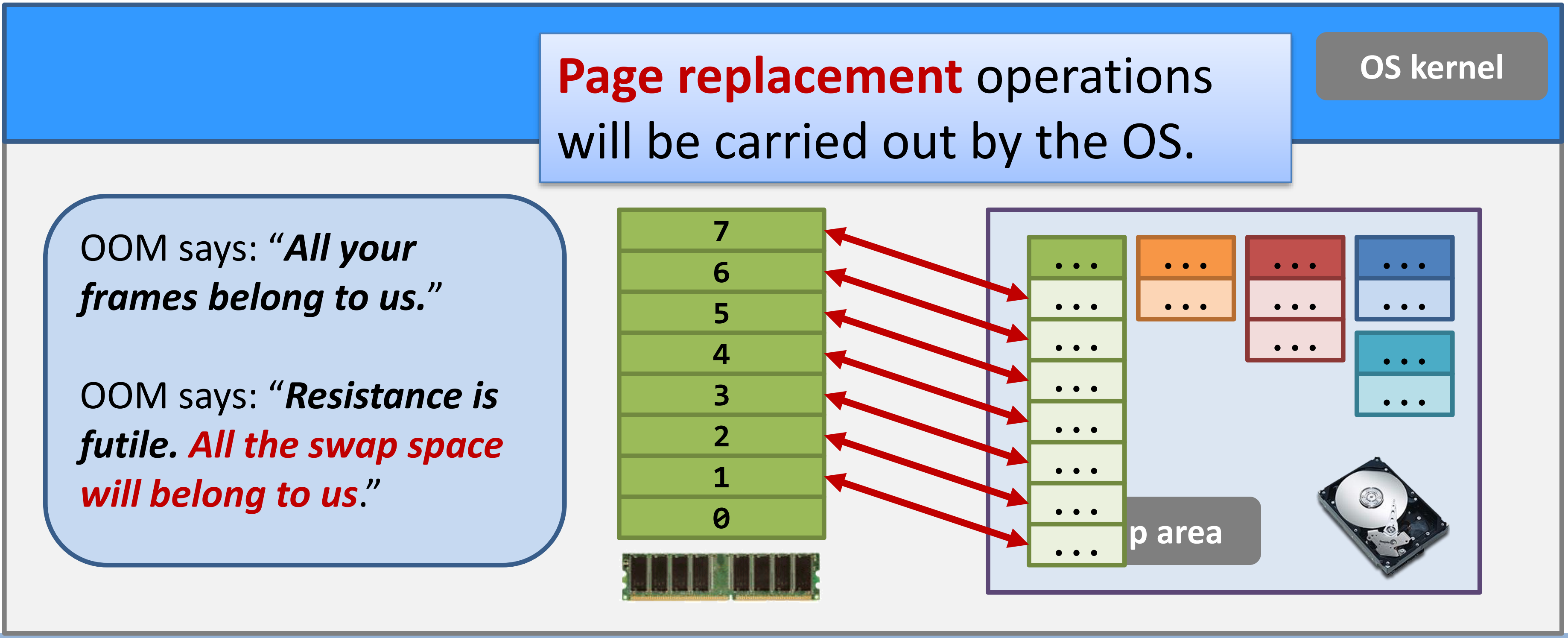
- OOM is running…**<u>2nd stage</u>**.
  - Occupied memory frames are the next zone that the process conquers (no unused frames).
  - **Disk activity flies high!**



Real OOM – illustration

`Real oom: running`

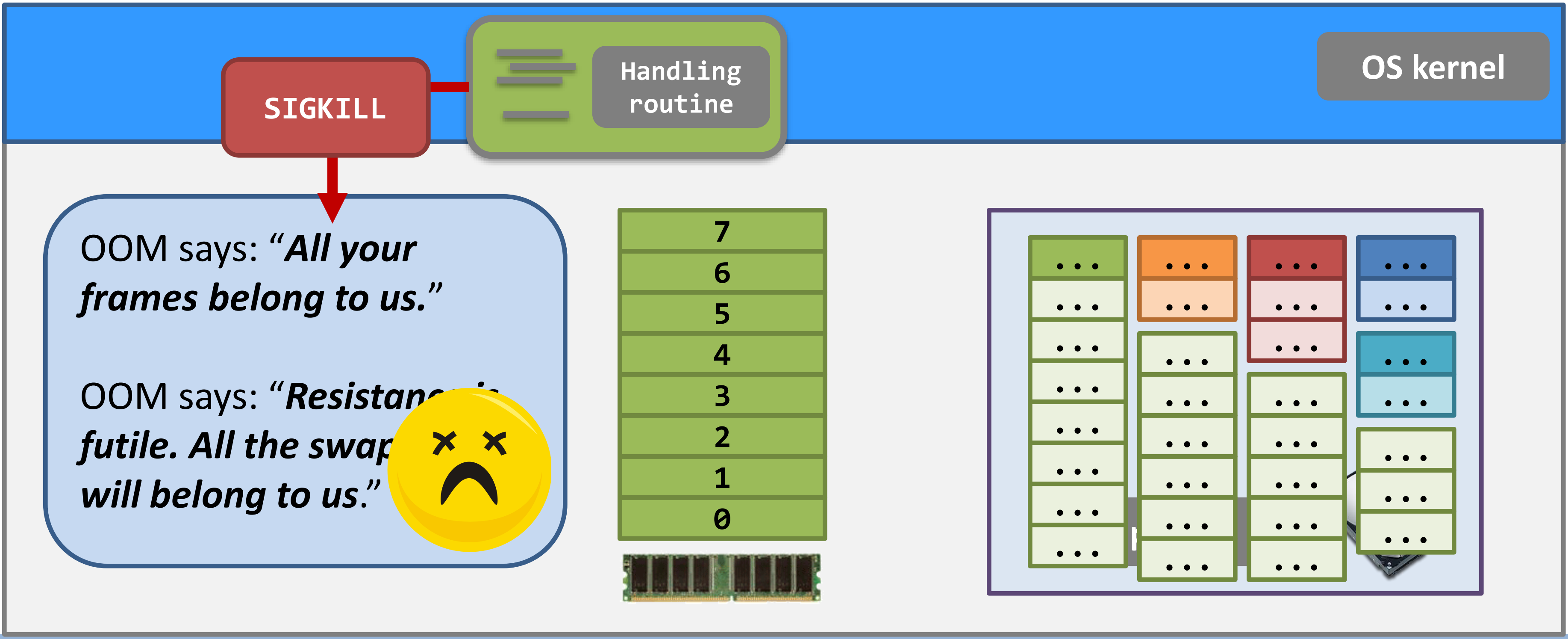
- OOM is running…**<u>3rd stage</u>**.
  - The previously-conquered frames are swapping to the swap area.
  - **Disk activity flies high!**



Real OOM – illustration

Real oom: 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

Real oom: killed

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

Demand paging - Issues

- Swap area
  – Where is it?
  – How large is it?

- Can we run a really large process (e.g., bigger than physical memory)?
  – How large is it at most?

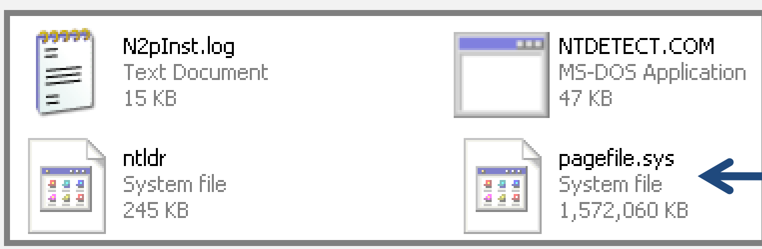
- How about fork() and exec*()?
  – Can they be clever?

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.

Swap area - size

- How large should the swap space be?
  - It should be at least the **same** as the size of the physical memory, so that …
    - when a really large process wants to take all the memory…
    - all the pages on the physical memory can find a place to hide.

  - An old rule said that “***swap should be twice the size of the physical memory***”.
    - But, I can’t find the reasons anymore, and this rule does not hold nowadays because we now have too much RAM!

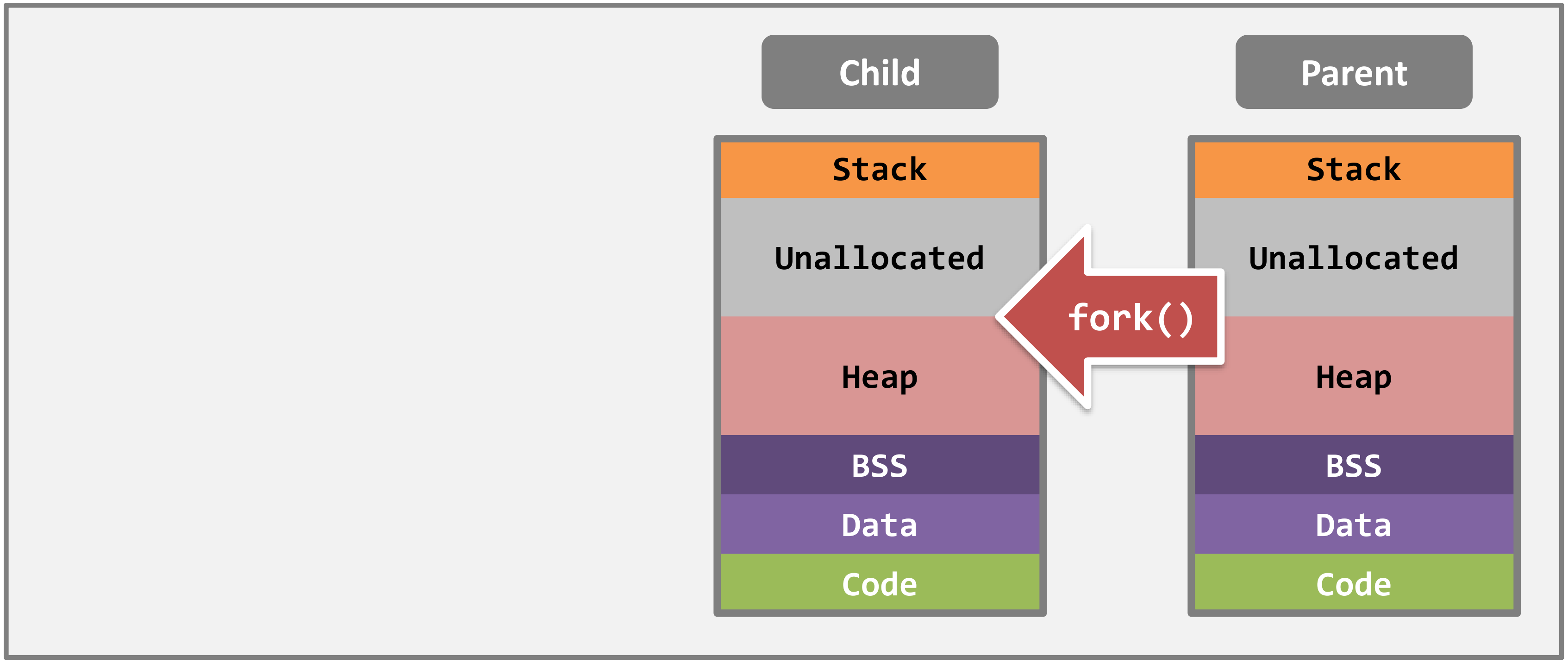
How about running large programs

- When a process is larger than the physical memory, is it able to run?
  - No need to load all data in memory…Demand paging
    - Generates **page fault** to allocate physical page frames
    - Trigger **page replacement** if there is no unused frames
- How large is a process that a system can support

Max. process size = Physical memory size + Available space in the swap partition (file) - Kernel memory size

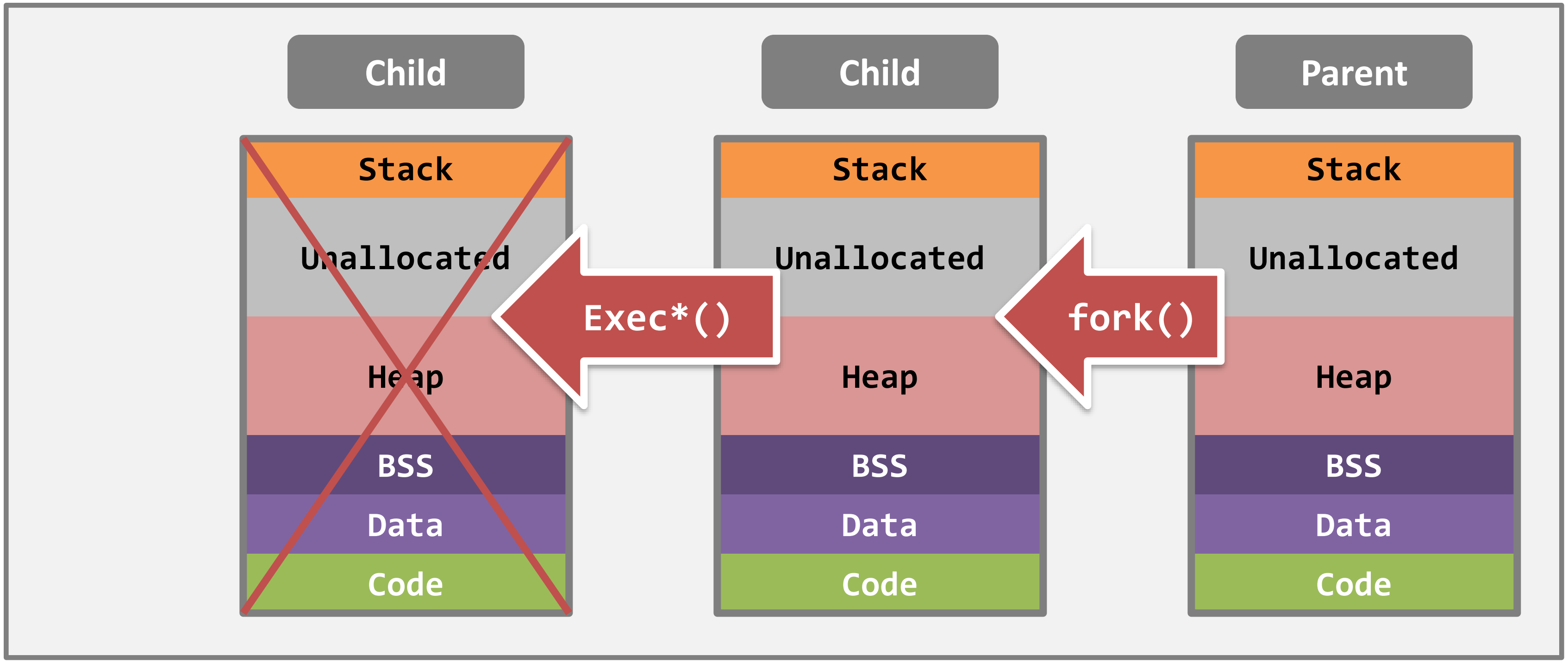
How about **`fork()` & `exec()`**

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



How about **fork() & exec()**

- What does duplication mean? Allocate new pages for the child process?
  - If yes…then consider **exec*()** system call as well…
  - Isn't it stupid?



How about `fork()` & `exec()`

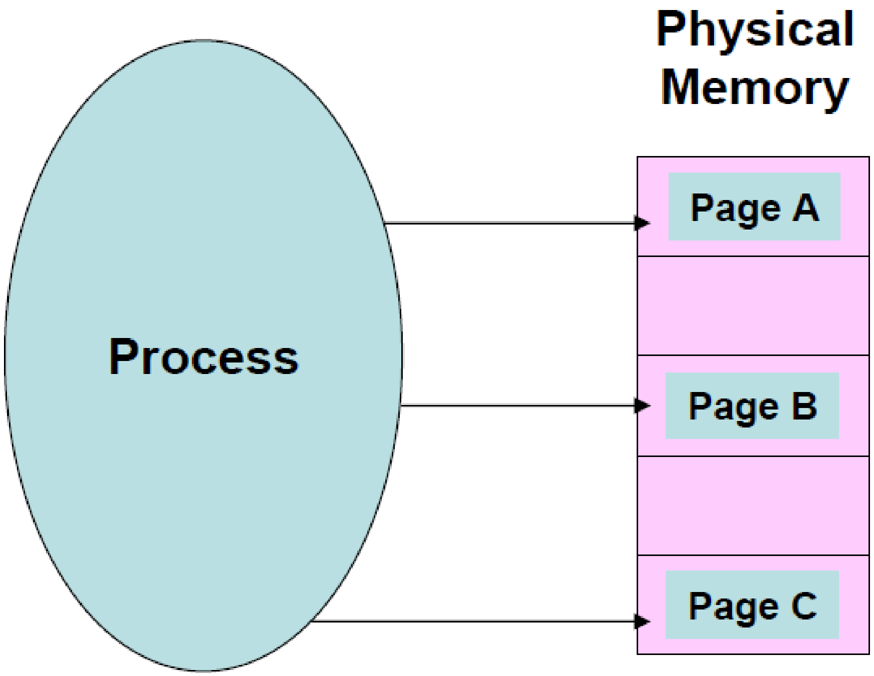
- Can we have a clever design with demand paging?
  - A technique called **copy-on-write** is implemented

Copy-on-write technique allows the parent and the child processes to **share** pages after the **`fork()`** system call is invoked.

A new, separated page will be **copied and modified** only when one of the processes **wants to write** on a shared page.

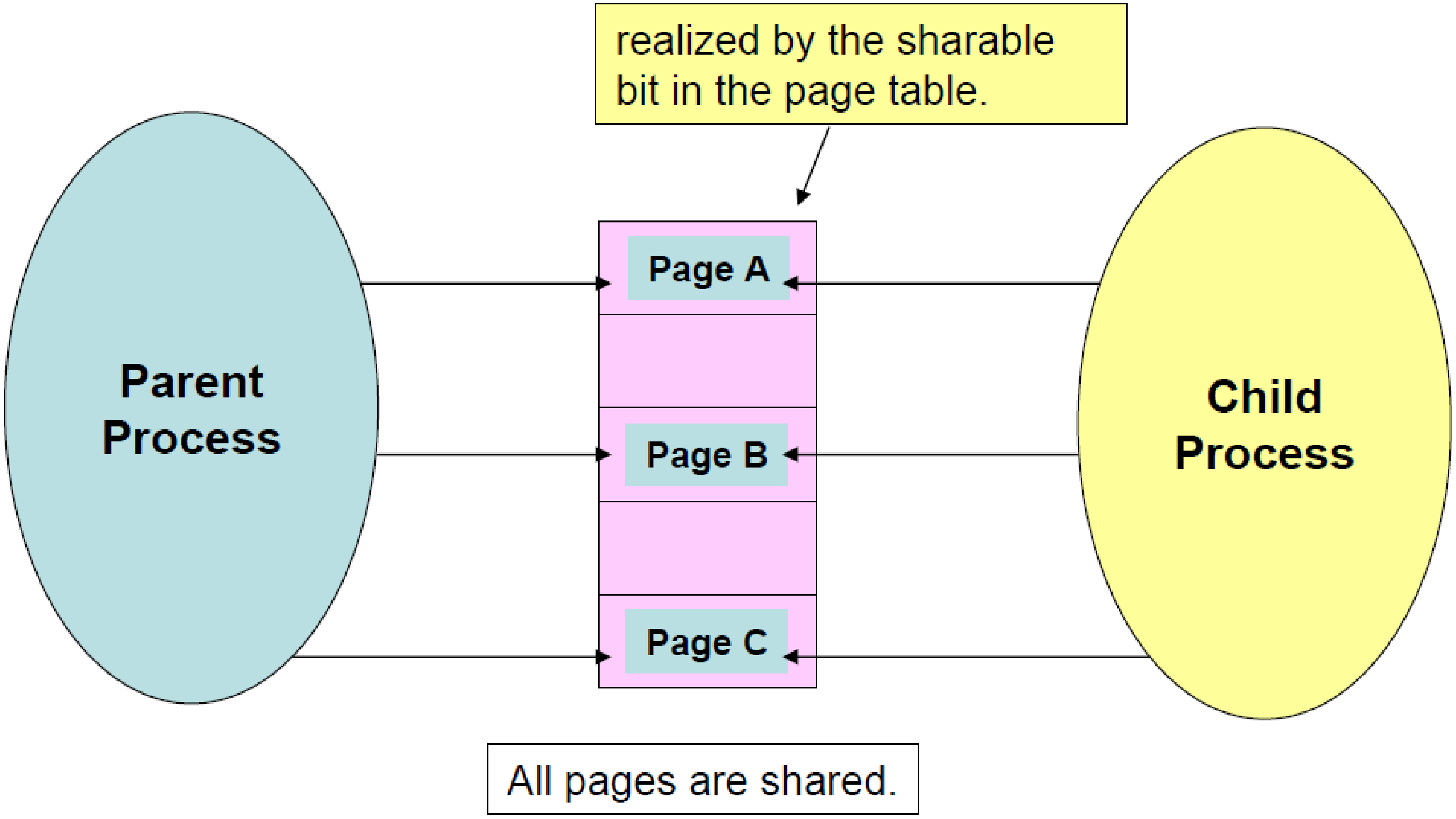
Copy-on-Write (COW)

- Before **`fork()`** …



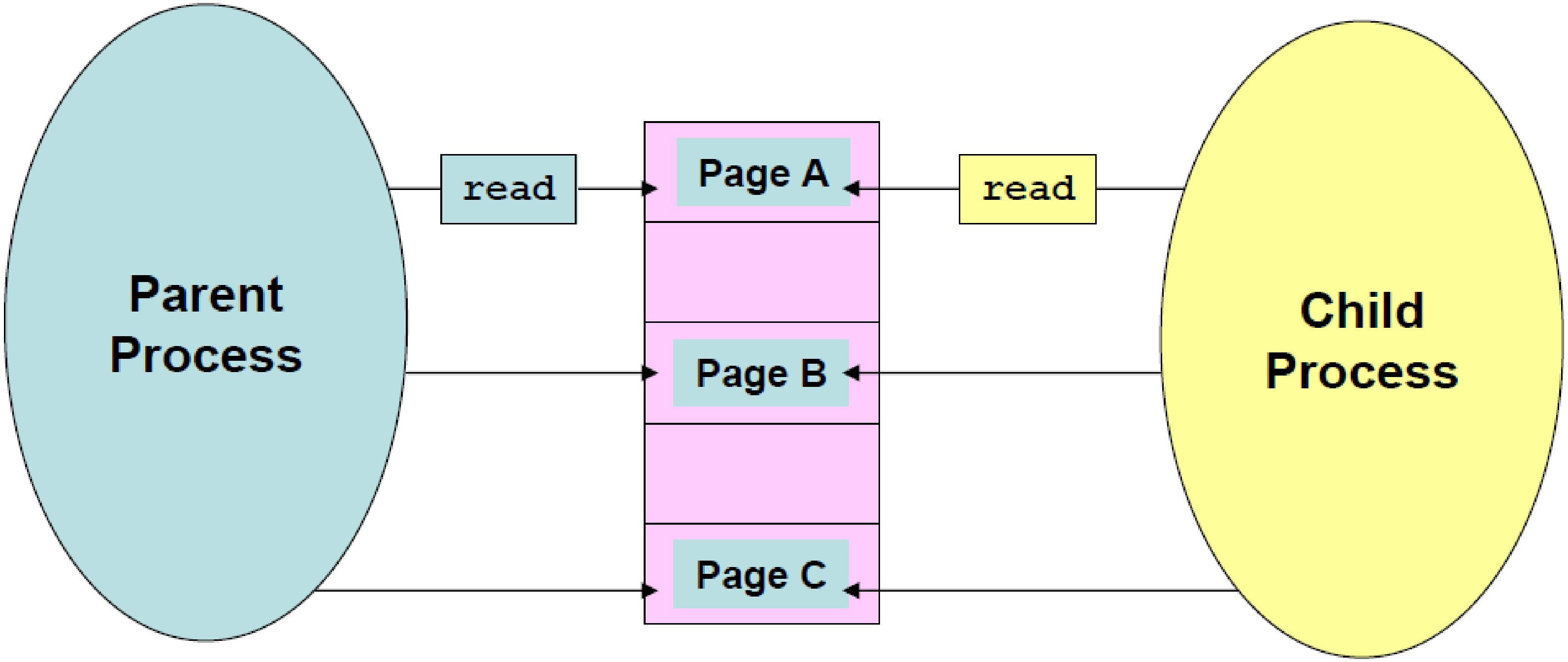
Copy-on-Write (COW)

- Right after **`fork()`** in invoked …



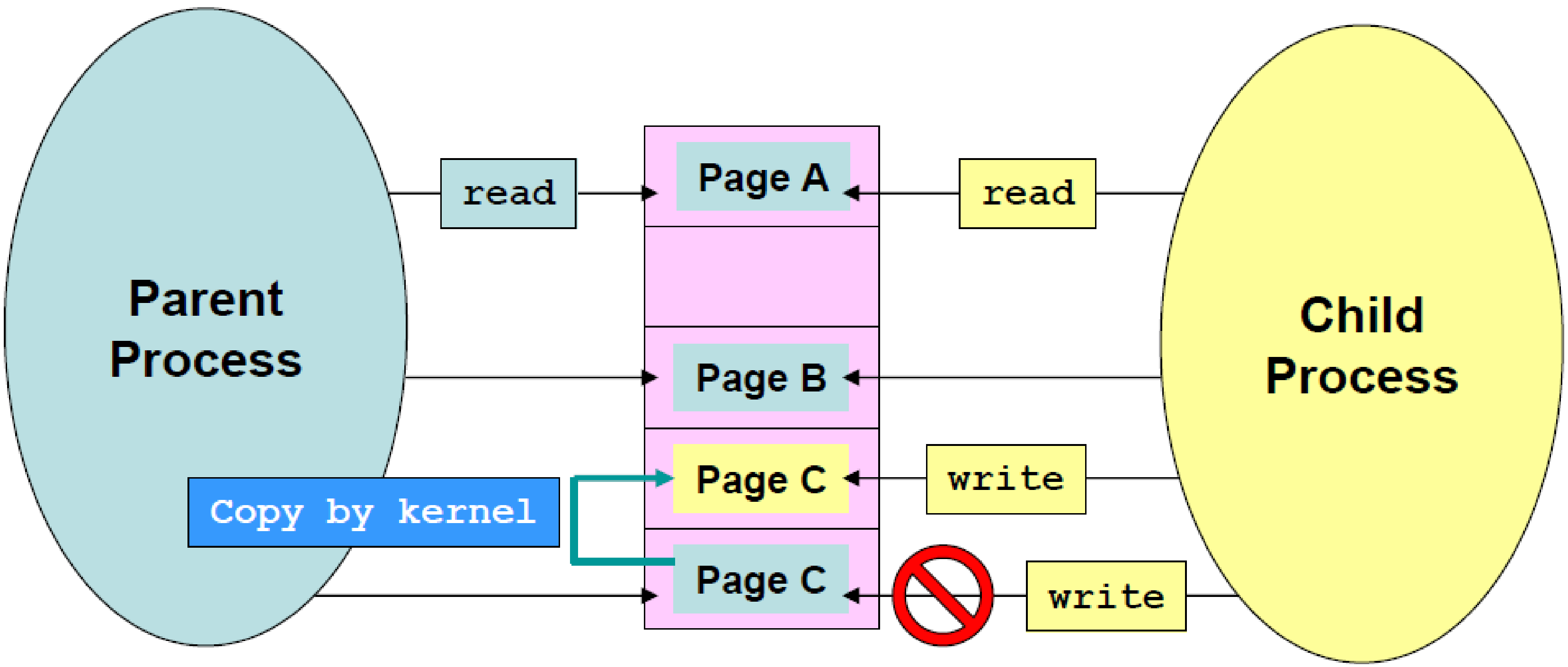
Copy-on-Write (COW)

- When both processes read the pages…



Copy-on-Write (COW)

- When one of the processes write to a shared page…



Demand paging - performance

- Demand paging can significantly affect performance
  - Service the page fault interrupt
  - Read in the page
  - Restart the instruction/process
- How to characterize?
  - Effective access time
  - $(1 - p) \times ma \ + \ \ p \times page\ fault\ time$
    - ma: memory access time (10-200ns)
    - p: prob. of a page fault
    - $page\ fault\ time$: ms

Example

- ma: 200ns, $page\ fault\ time$: 8ms
- 1/1000 page fault probability
  - Effective access time: $(1 - p)200ns + p \times 8\text{ms} = 8.2\mu s$
- To allow 10% performance degradation only
  - $(1 - p)200ns + p \times 8\text{ms} < 220\text{ns}$
  - $p < 0.0000025$
- Thus, page fault rate must be low

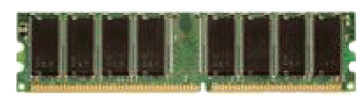
Summary of demand paging

- Demand paging enables **over-commitment**
  - Large process can be supported
  - Concurrent running of multiple processes is also supported

- One key issue is…
  - How to select victim pages to swap out?
  - Page-replacement algorithm

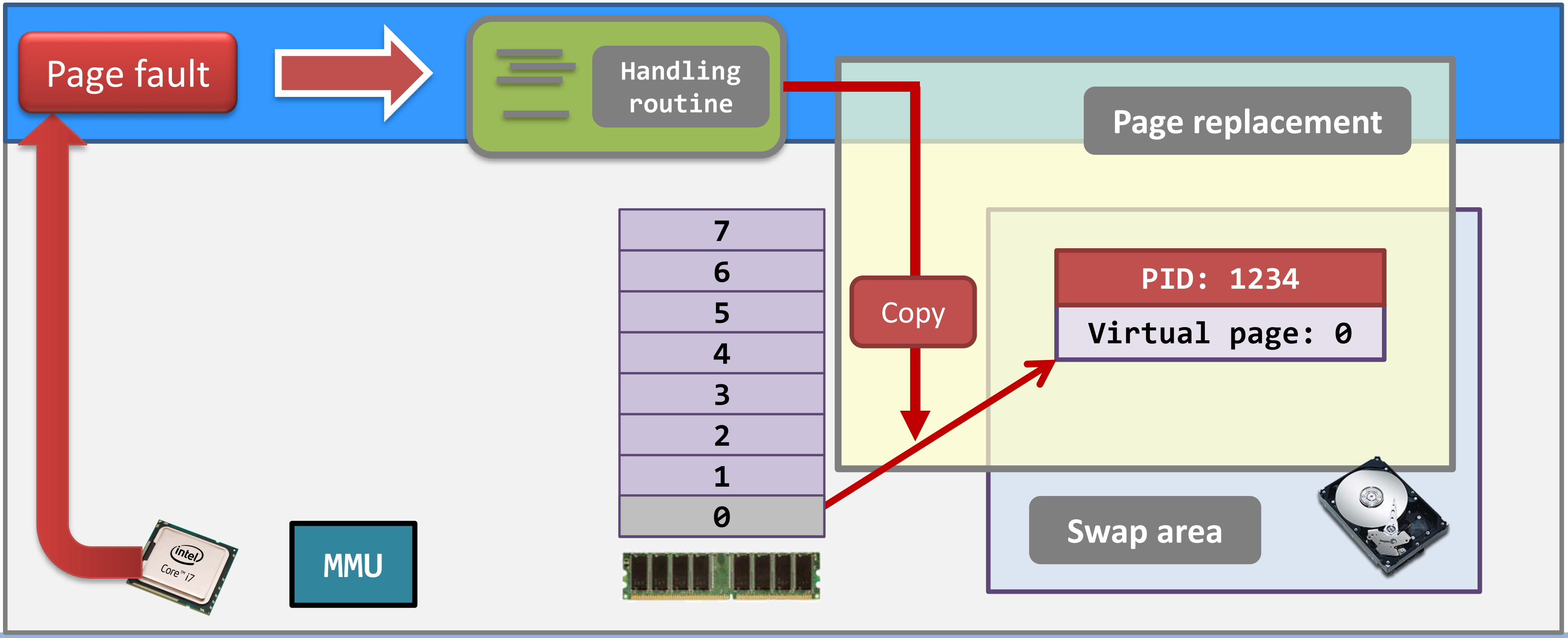
Memory Management

- Virtual memory;
- MMU implementation & paging;
- Demand paging;
- **Page replacement algorithms;**
- Allocation of frames;



Page replacement – introduction

- 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.



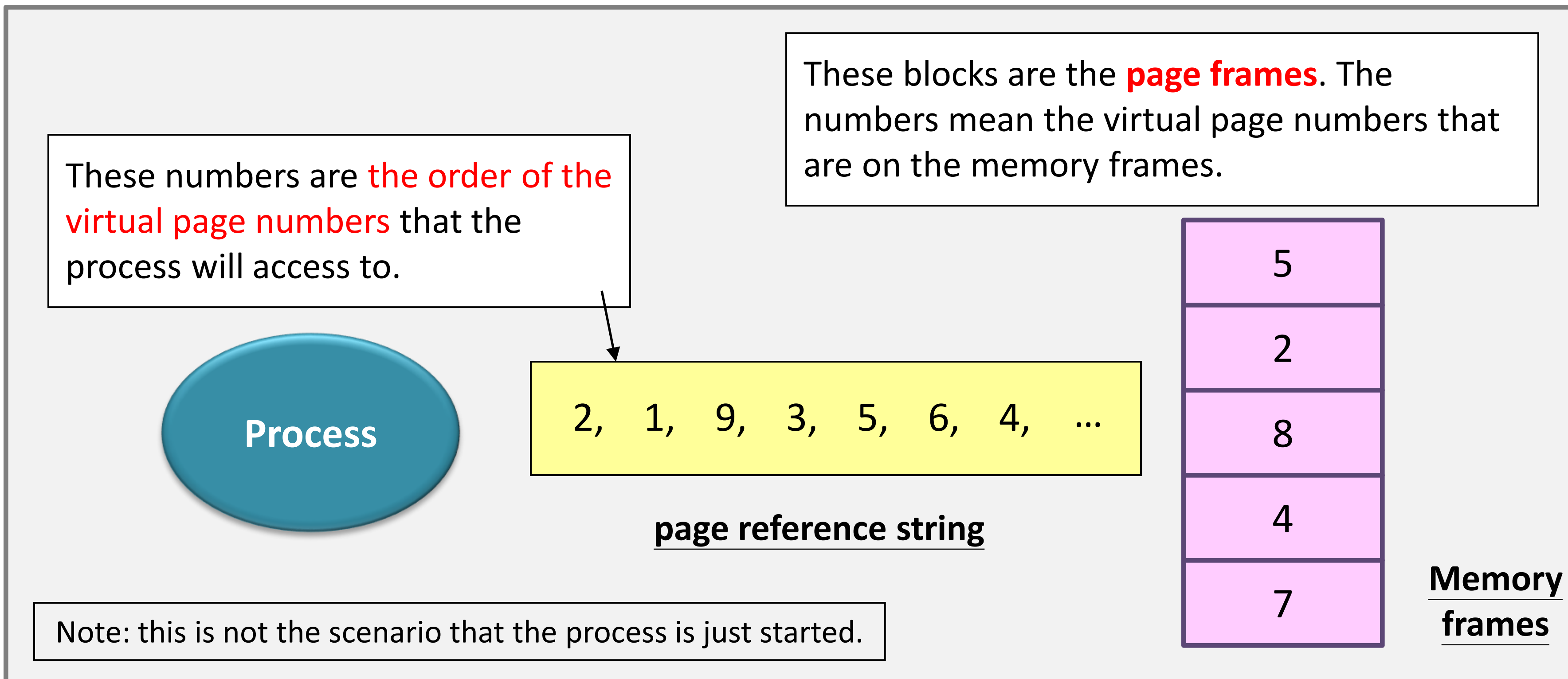
Page replacement – introduction

- Replacing a page involves disk accesses, therefore a page fault is **slow and expensive**!
  - Key issue: which page should be swapped out?
  - Page replacement algorithms should **minimize further page faults**.

- In the following, we introduce four algorithms:
  - Optimal;
  - First-in first-out (FIFO);
  - Least recently used (LRU);
  - Second-chance algorithm

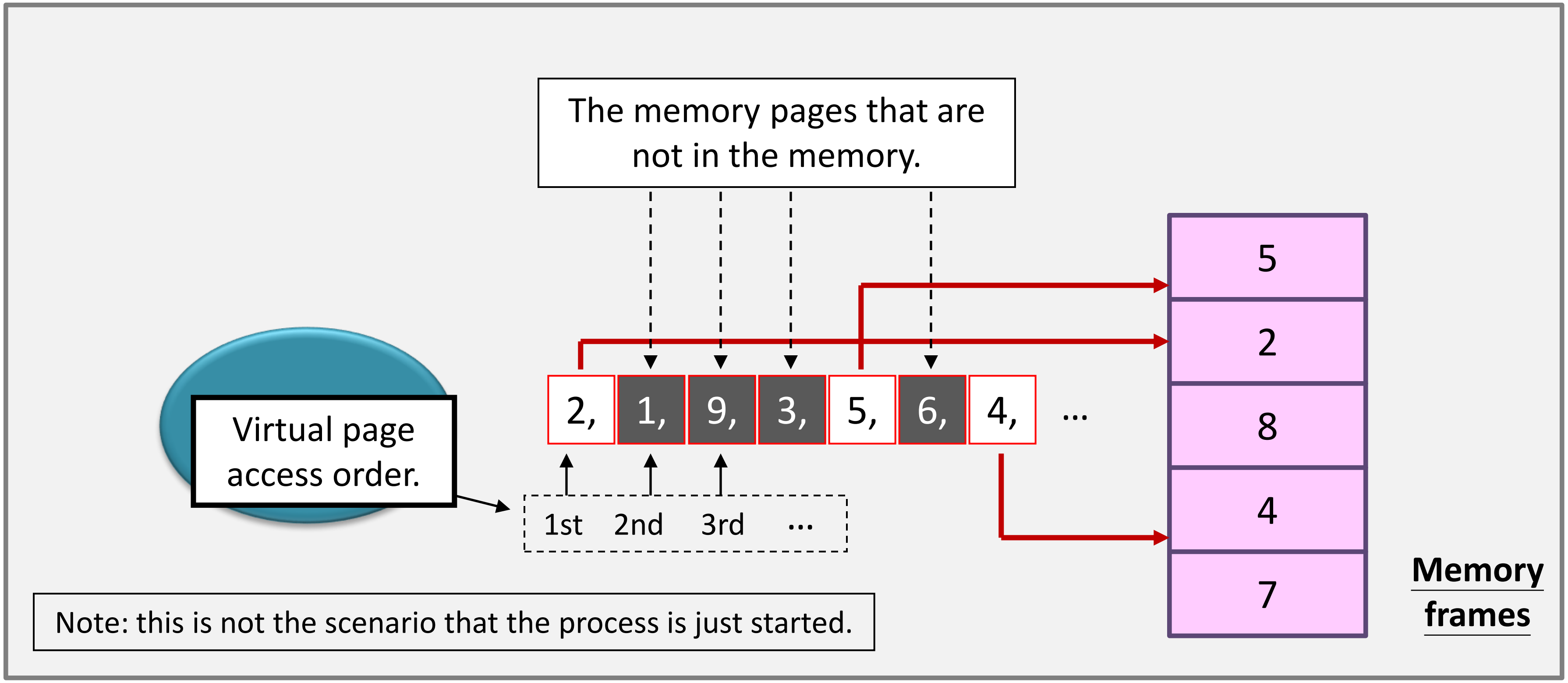
Page replacement – algorithm

- 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.



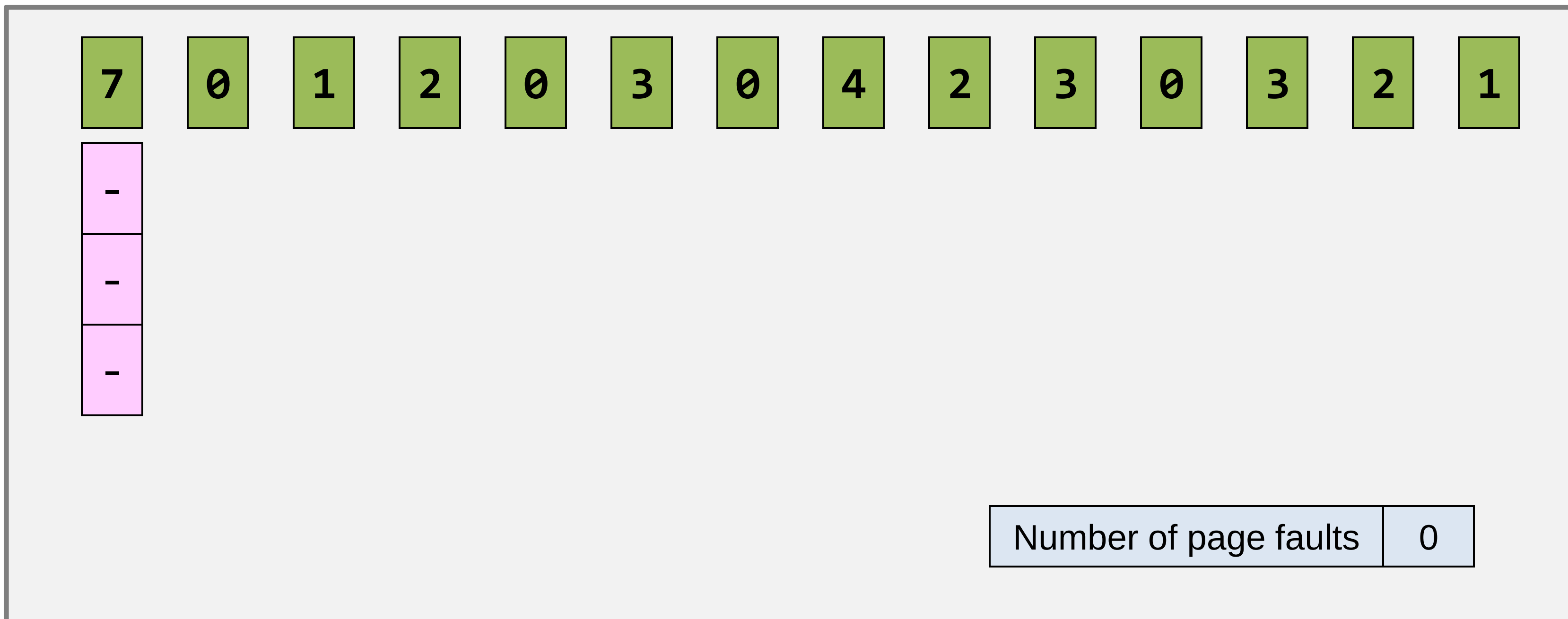
Page replacement – algorithm

- 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.



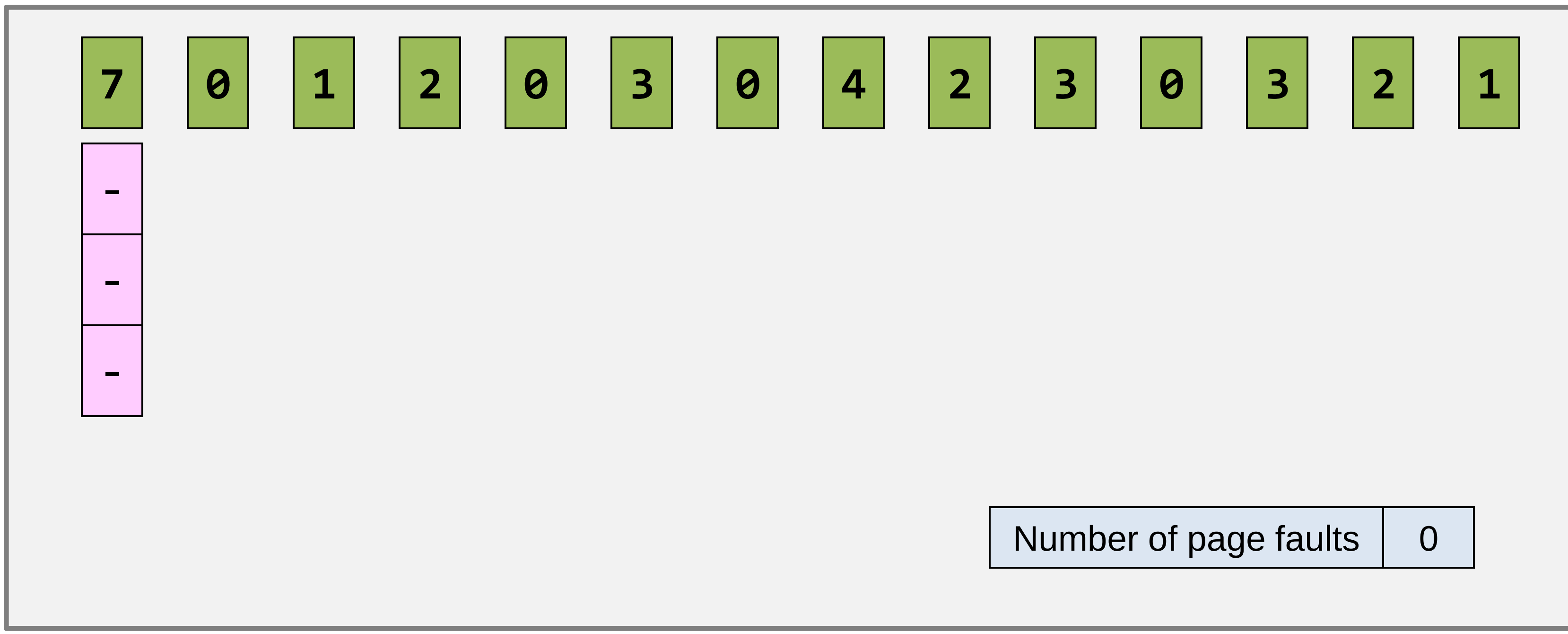
Page replacement – when an algorithm starts

- Initial condition
  - Let all the frames be empty.



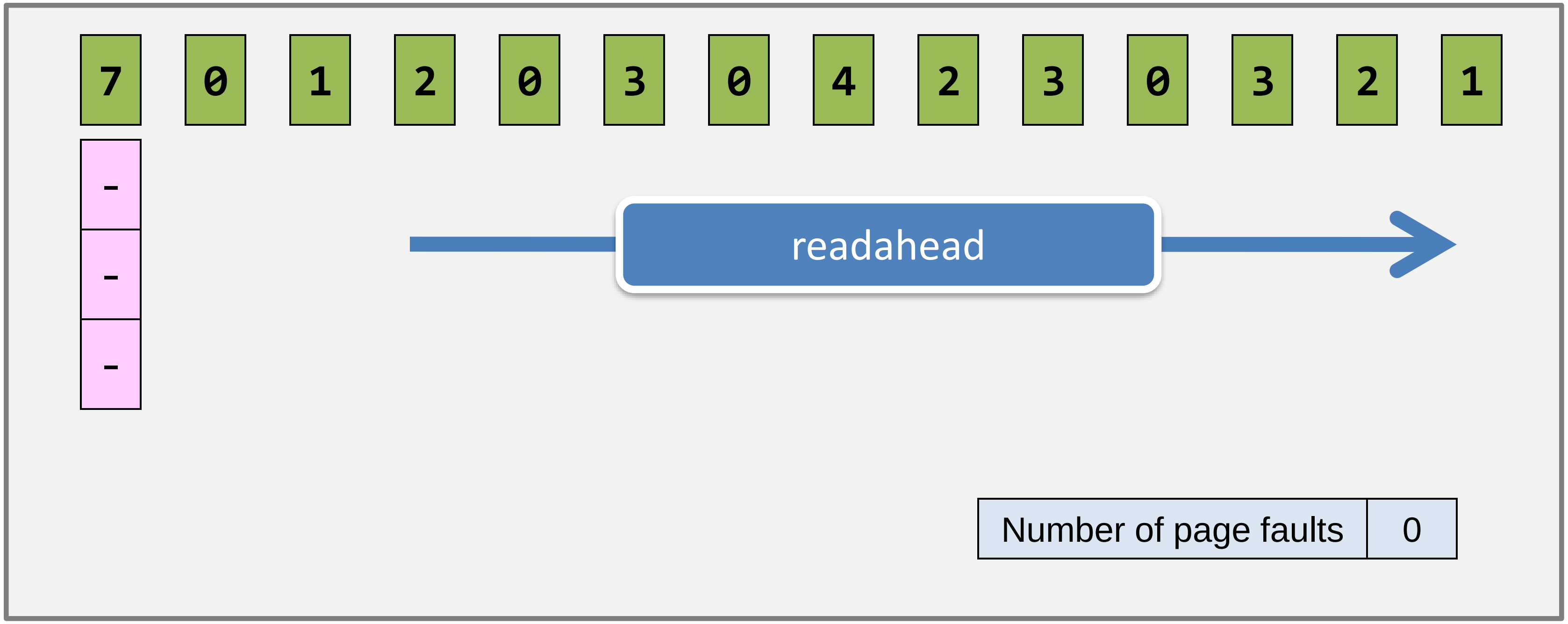
Page replacement – optimal algorithm

- What is the best algorithm?
  - Do not worry about the implementation at this moment.



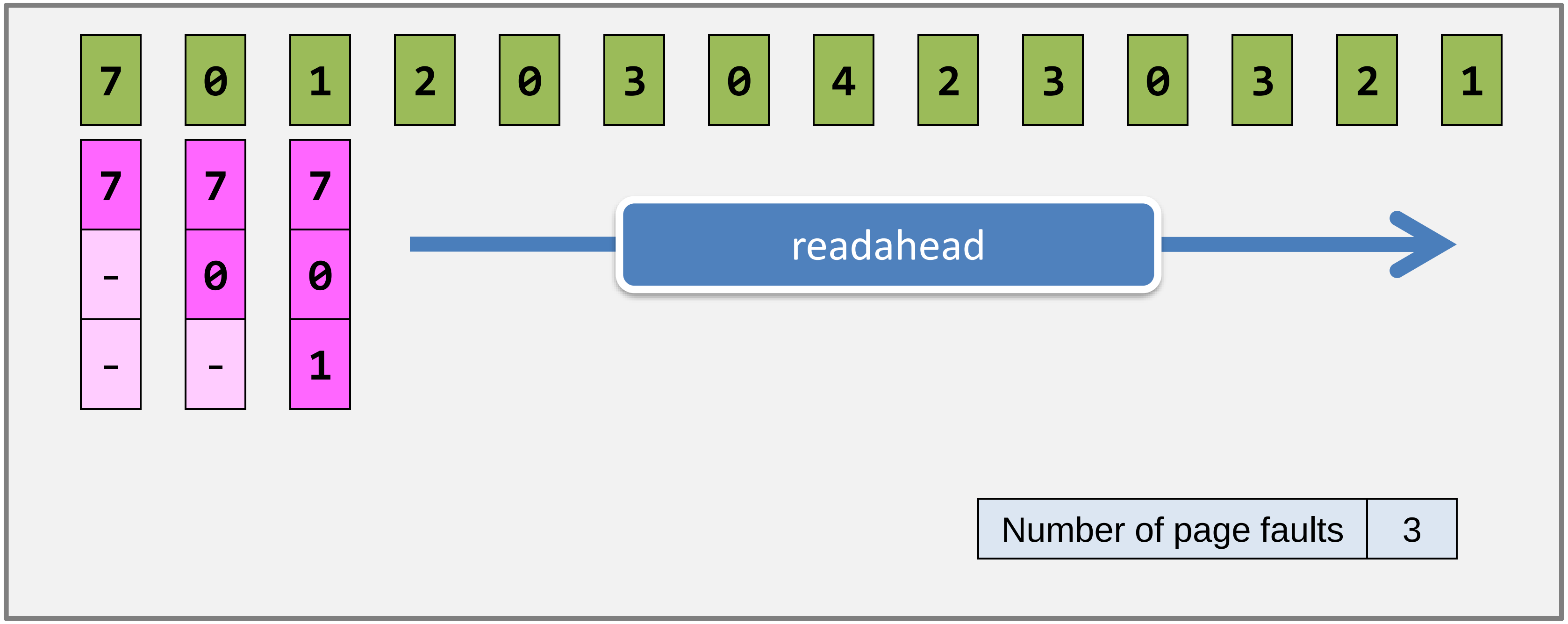
Page replacement – optimal algorithm

- 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".



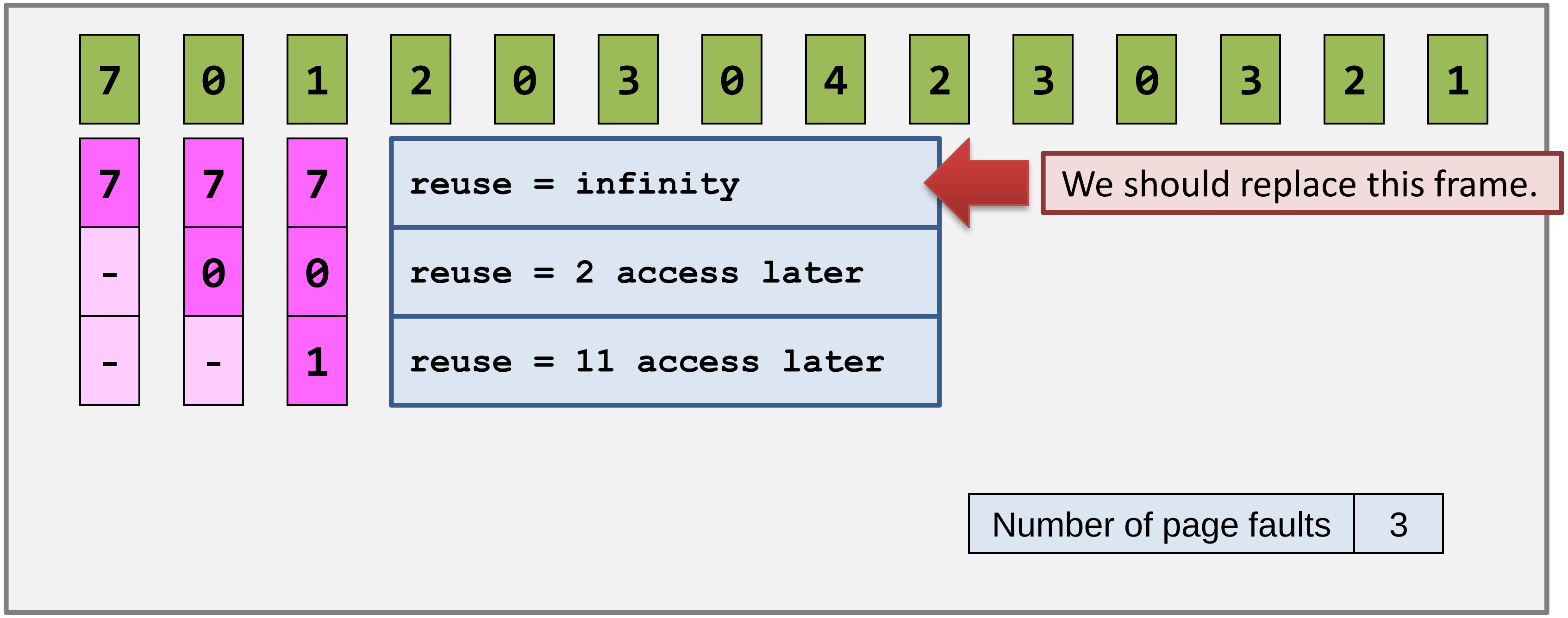
Page replacement – optimal algorithm

- If I **know the future**, then I know how to do better.
  - The first page request will cause a **page fault**.
    - Because there are free frames, no replacement is needed.



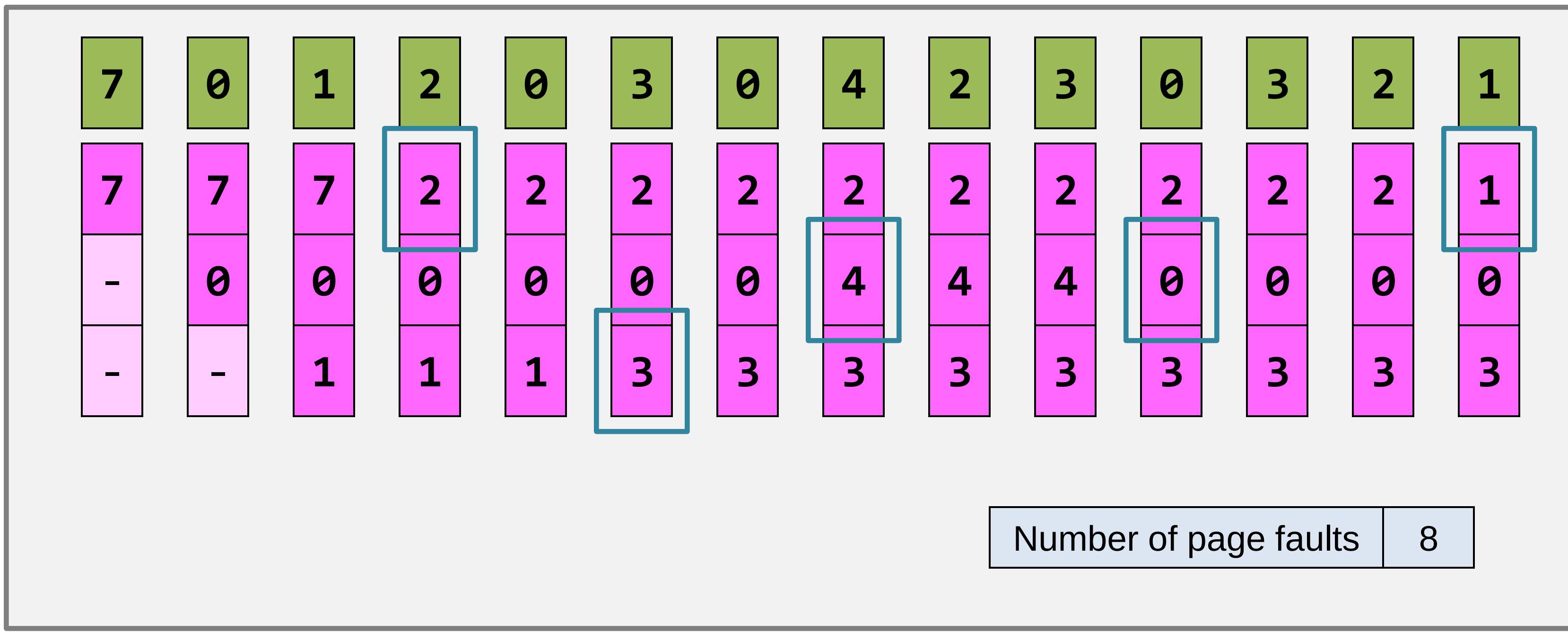
Page replacement – optimal algorithm

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



Page replacement – optimal algorithm

- 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.



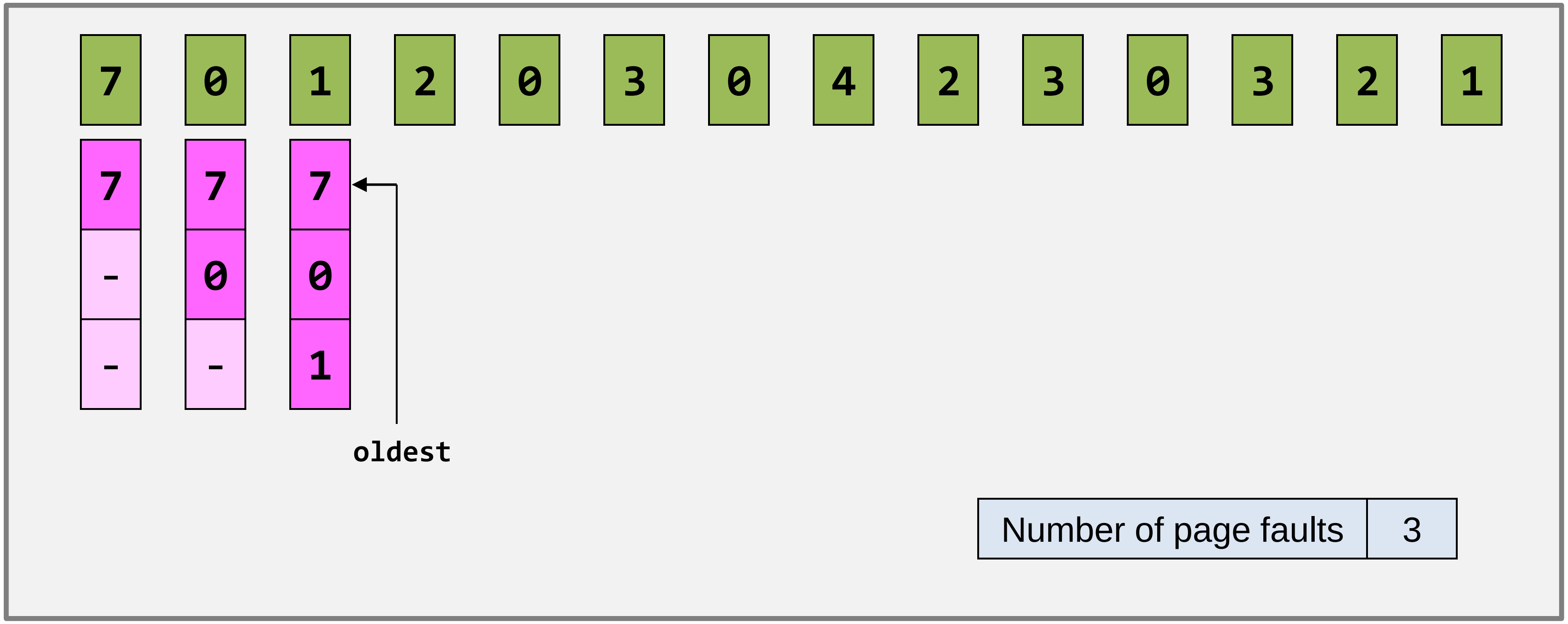
Page replacement – Problem of the optimal algorithm

- Unfortunately, you never know the future…
  - It is not practical to implement such an algorithm
  - Is there any easy-to-implement algorithm?
    - You have already learnt process scheduling

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

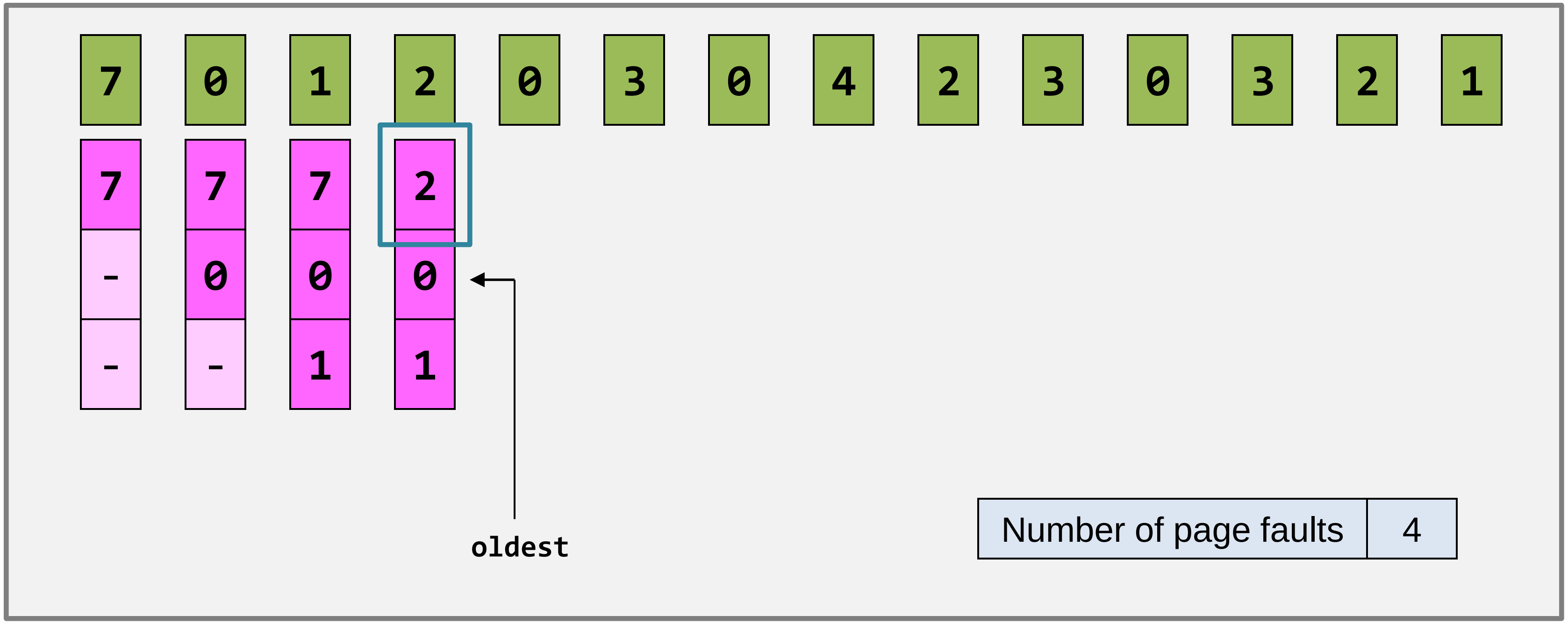
Page replacement – FIFO algorithm

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



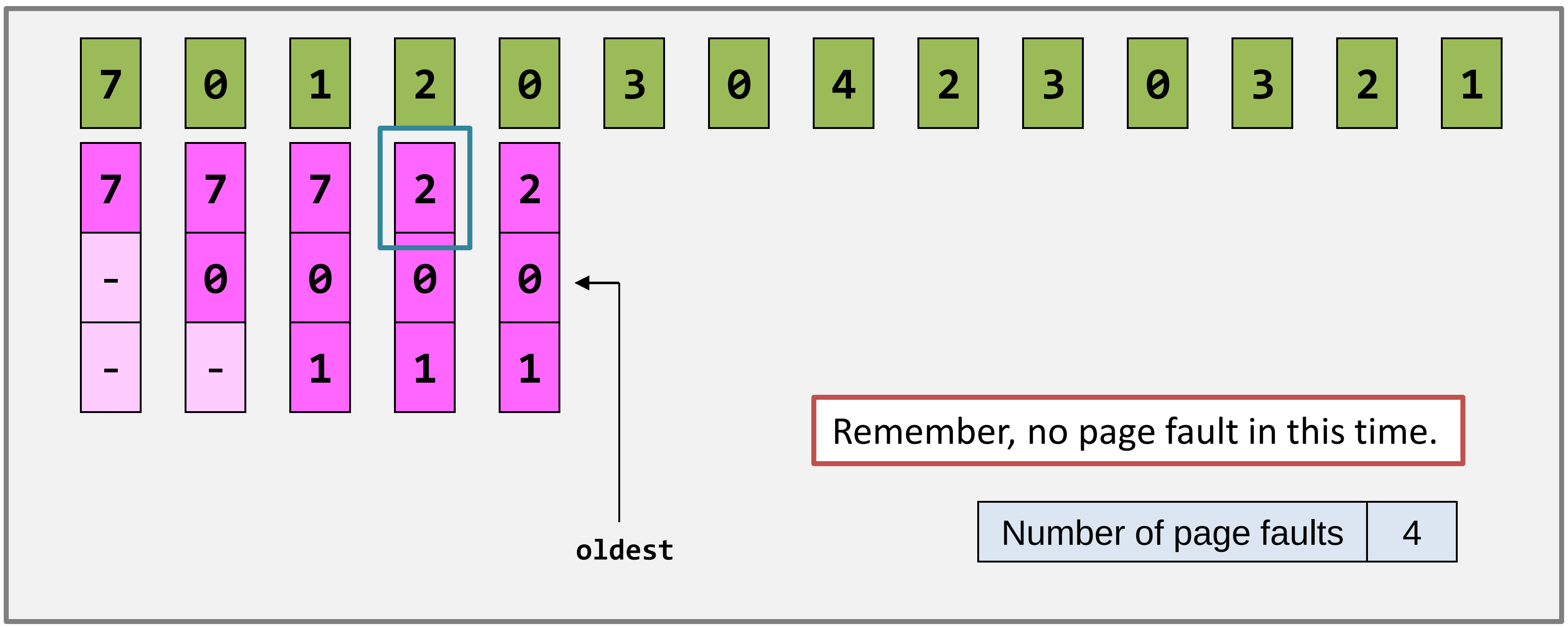
Page replacement – FIFO algorithm

- 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.



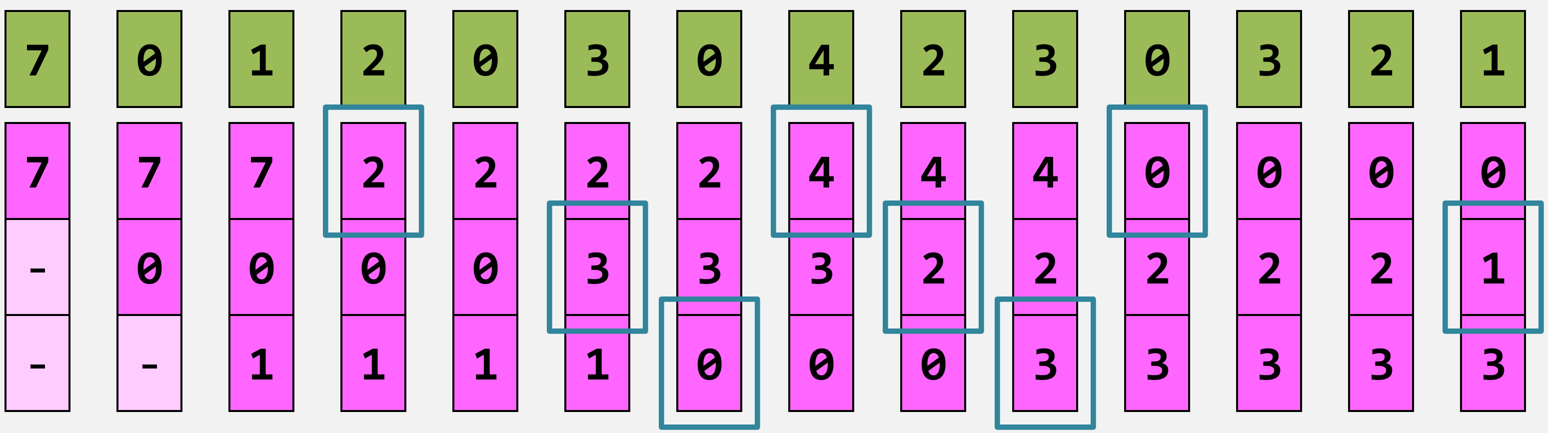
Page replacement – FIFO algorithm

- When a memory reference can be found in the memory, will the age of that frame be changed?
  - NO! The frame storing "page 0" is still the oldest frame.



Page replacement – FIFO algorithm

- The story goes on…
  - Seems that there is **no intelligence** in this method…
  - Pages which will be accessed again are swapped out



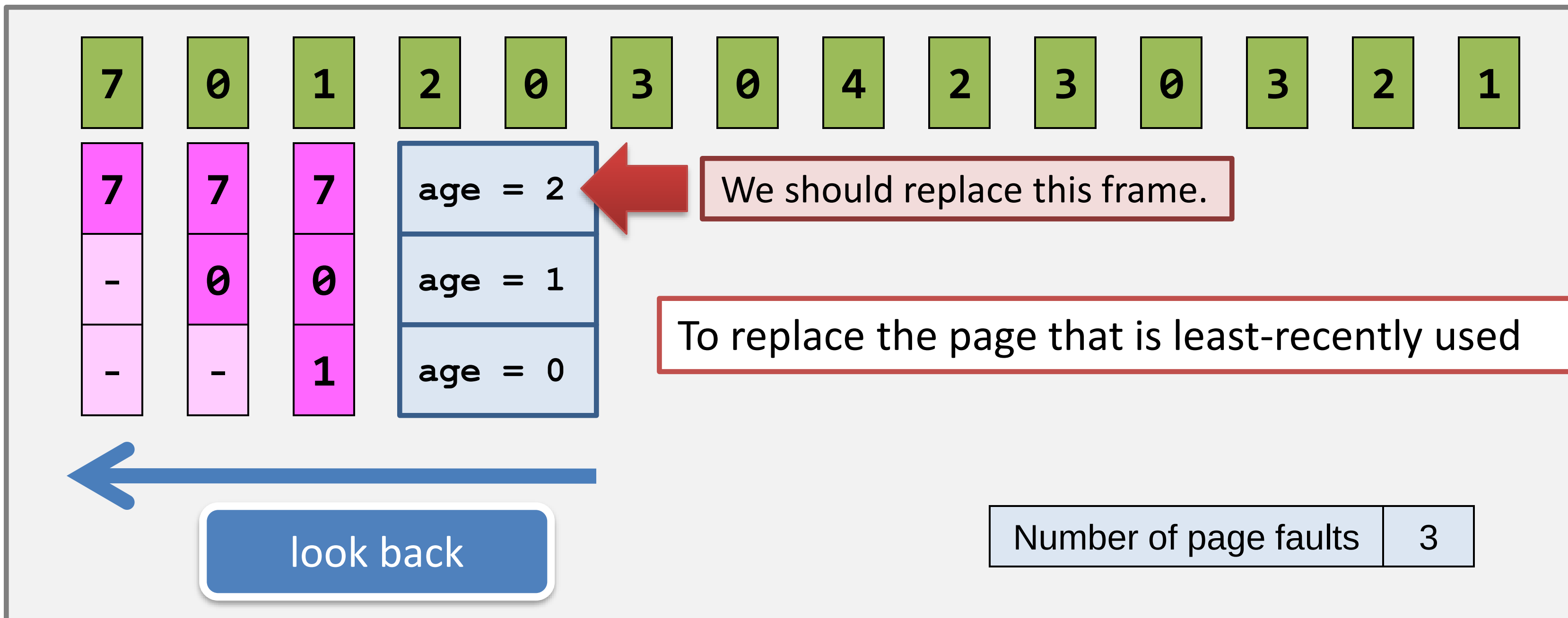
Number of page faults	11
Number of page faults in the optimal algorithm	8

Page replacement – LRU algorithm

- Can we do better?
  - Still remember the locality rule?
    - Recently accessed pages may be accessed again in near future
  - Why not swap out the pages which are not accessed recently
    - This is the **least-recently-used** (LRU) page replacement.

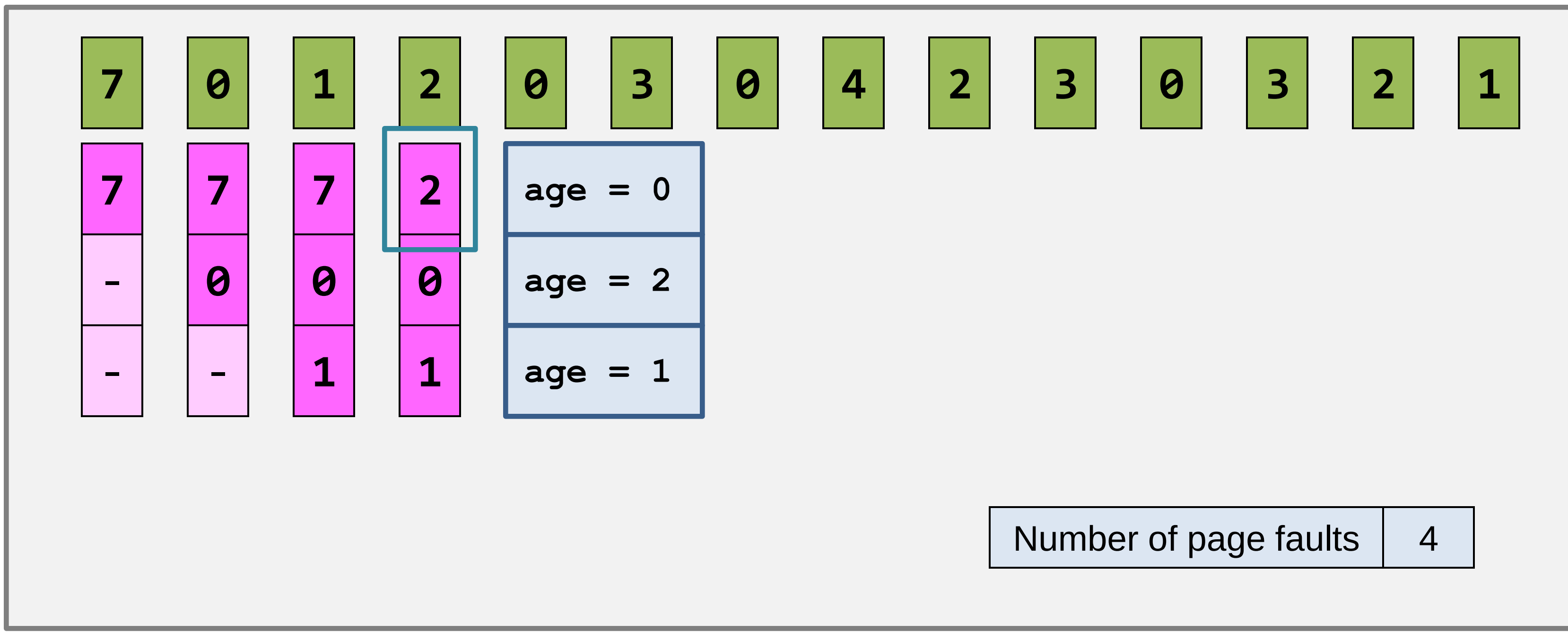
Page replacement – LRU algorithm

- Strategy:
  - 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 its age to be 0**.
    - Other frames' ages are incremented by 1.



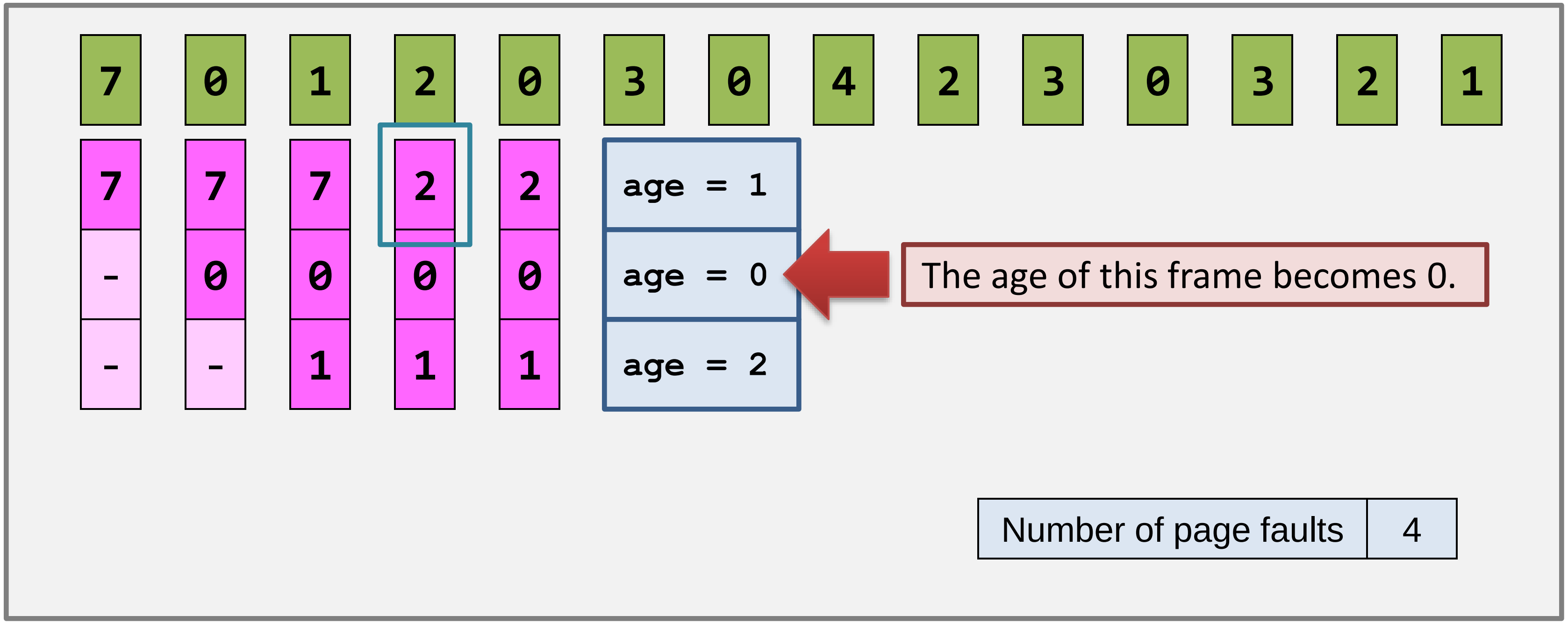
Page replacement – LRU algorithm

- Strategy:
  - 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 its age to be 0**.
    - Other frames' ages are incremented by 1.



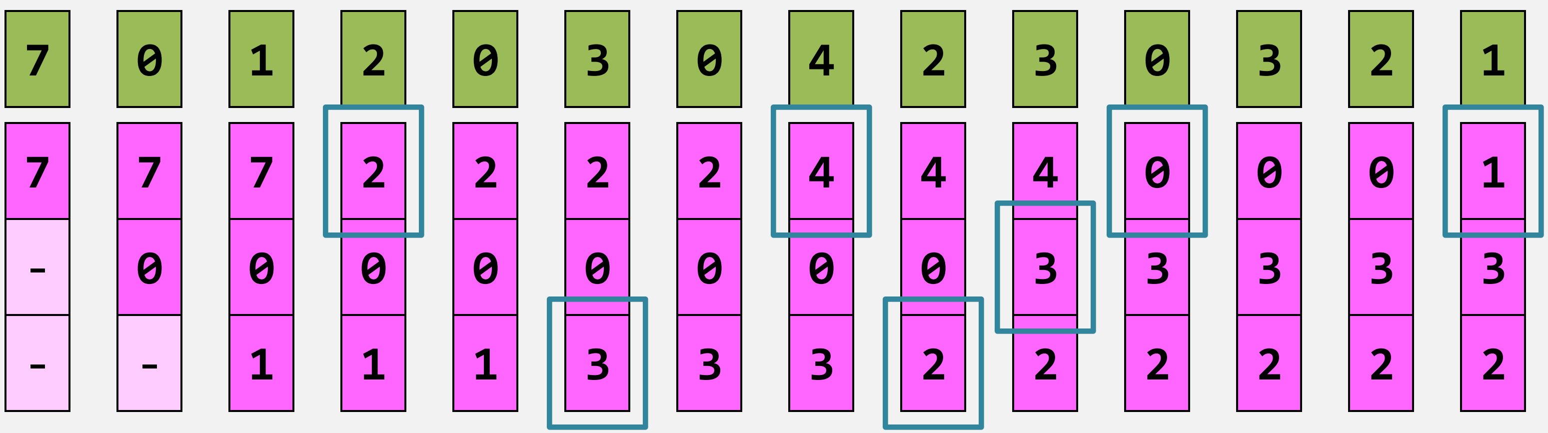
Page replacement – LRU algorithm

- Strategy:
  - 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 its age to be 0**.
    - Other frames' ages are incremented by 1.



Page replacement – LRU algorithm

- Strategy:
  - 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 its age to be 0**.
    - Other frames' ages are incremented by 1.



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

Number of page faults	10

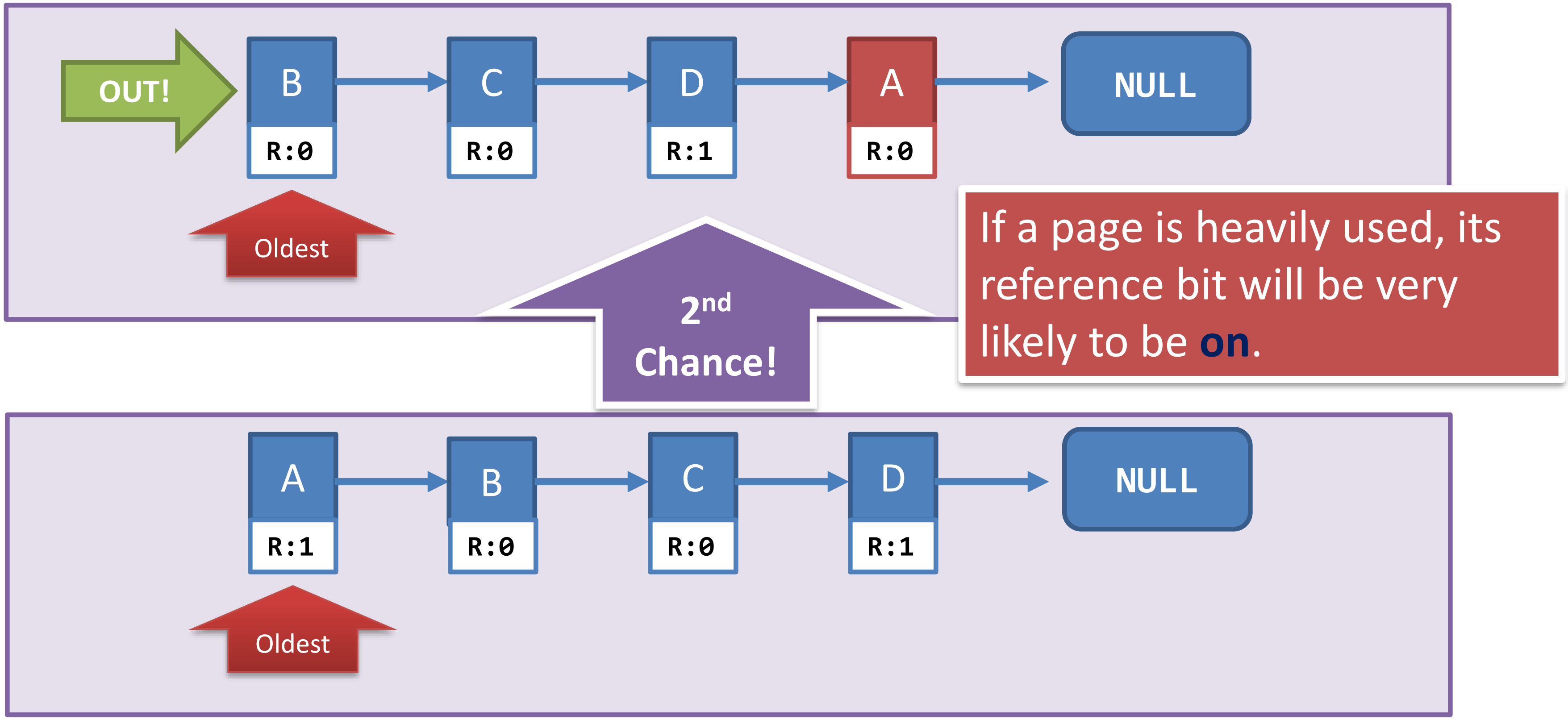
Page replacement – LRU algorithm

- The performance of LRU is considered to be good, but **how to implement** the LRU algorithm efficiently
  - Counters: requires to update counter and search the table to find the page to evict
  - Stack: implement with doubly linked list (pointer update)

- Common case in many systems
  - A reference bit for each page (set by hardware)
  - LRU approximation: **Second-chance algorithm**

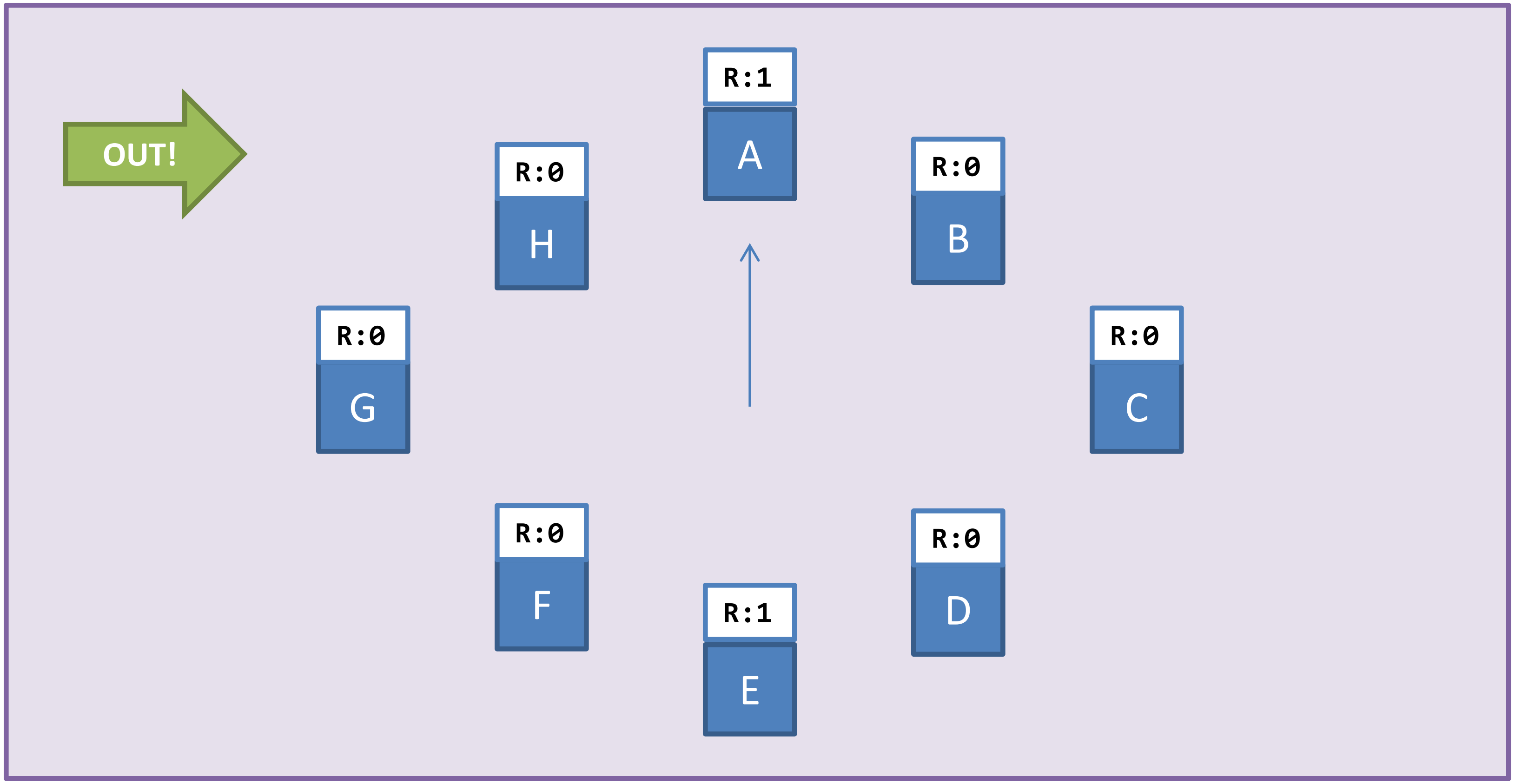
Page replacement – LRU approximation

- Second-chance algorithm
  - Basic: FIFO
  - Give the page a second chance if its reference bit is on



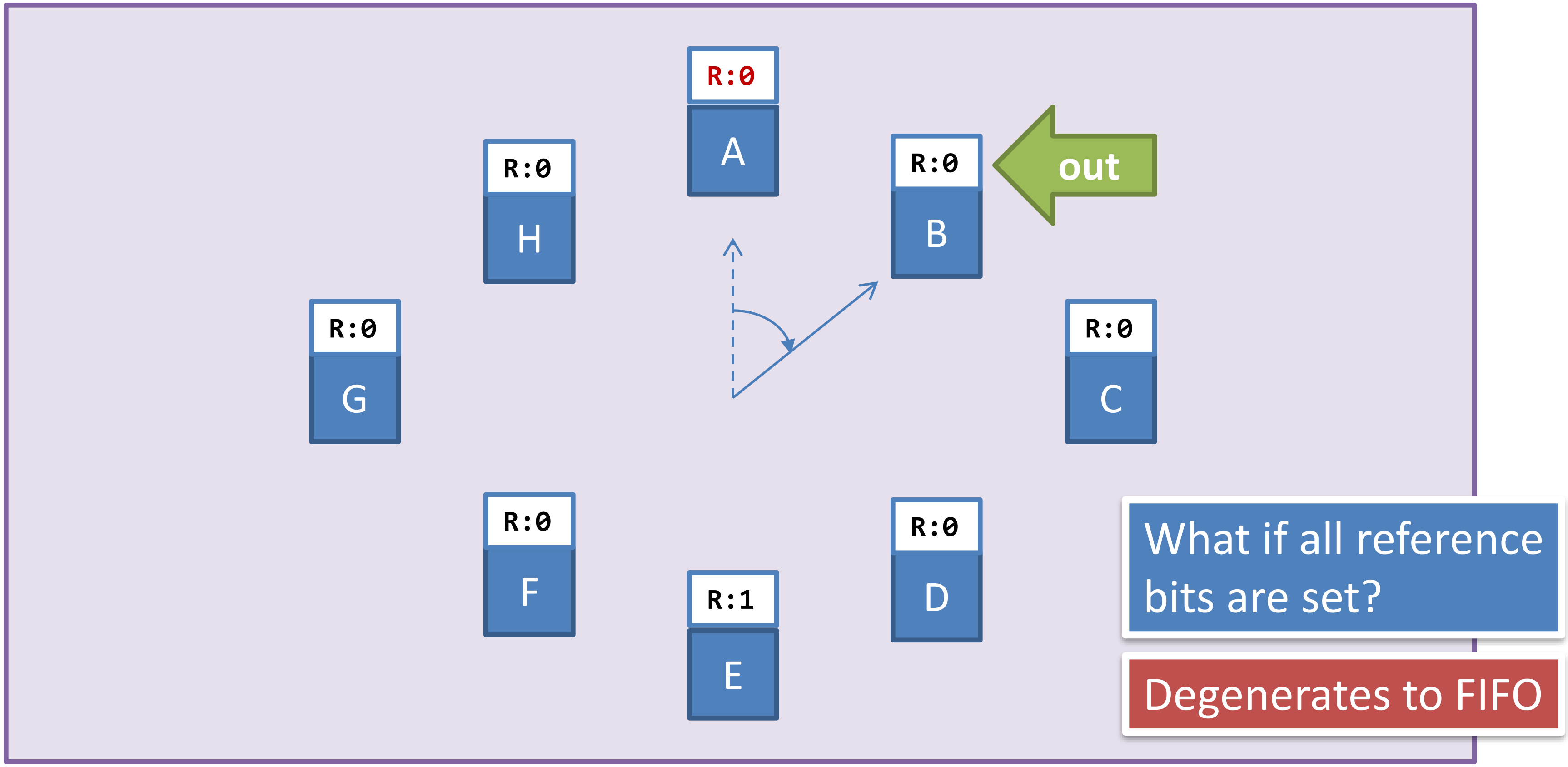
Page replacement – LRU approximation

- Clock is the efficient implementation of the 2nd chance algorithm (**circular queue**).



Page replacement – LRU approximation

- Clock is the efficient implementation of the 2nd chance algorithm (**circular queue**).



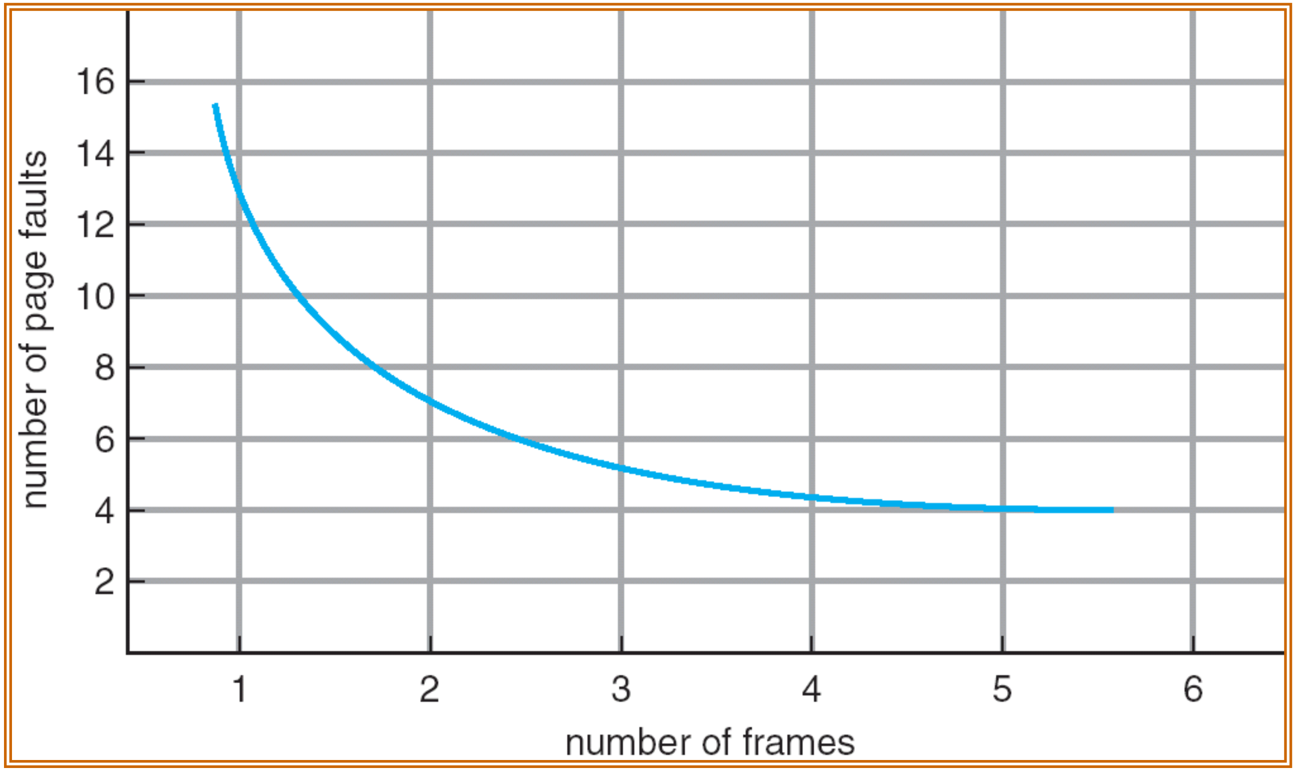
Page replacement – performance

- 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

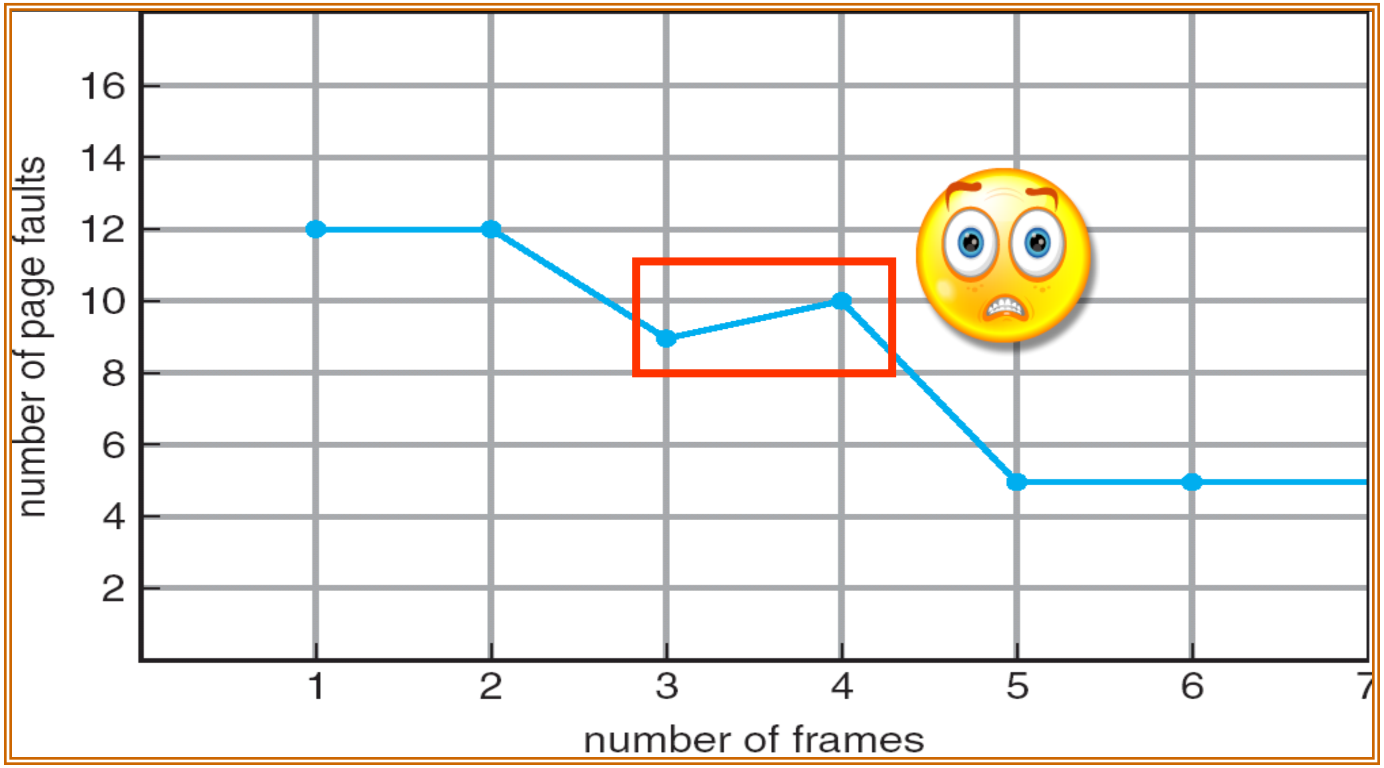
- Your expectation:



Page replacement – performance

- The reality may be:

This is called Belady's anomaly



Page replacement – performance

- 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
```

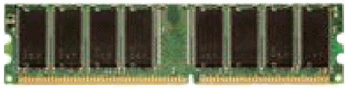
Page replacement – performance

- Belady's anomaly exists for some algorithms
  - Both optimal and LRU do not suffer from it

- **Stack algorithms:** never exhibit Belady's anomaly
  - Feature: The set of pages in memory for n frames is always a **subset** of the set of pages in memory for $n + 1$ frames
  - Example: LRU
    - The n most recently referenced pages will still be the most recently referenced pages when the number of frames increases

Memory Management

- Virtual memory;
- MMU implementation & paging;
- Demand paging;
- Page replacement algorithms;
- **Allocation of frames;**



Allocation for user processes

- Free-frame list
  - Demand paging and page replacement

- Constrains
  - Limit on number of frames
    - Upper bound: total available frames
    - Lower bound: has a minimum number
      - Performance consideration (limit page-fault rate)
      - Defined by computer architecture (instructions)
      - Process will be suspended if the number of allocated frames falls below the minimum requirement

Allocation algorithm

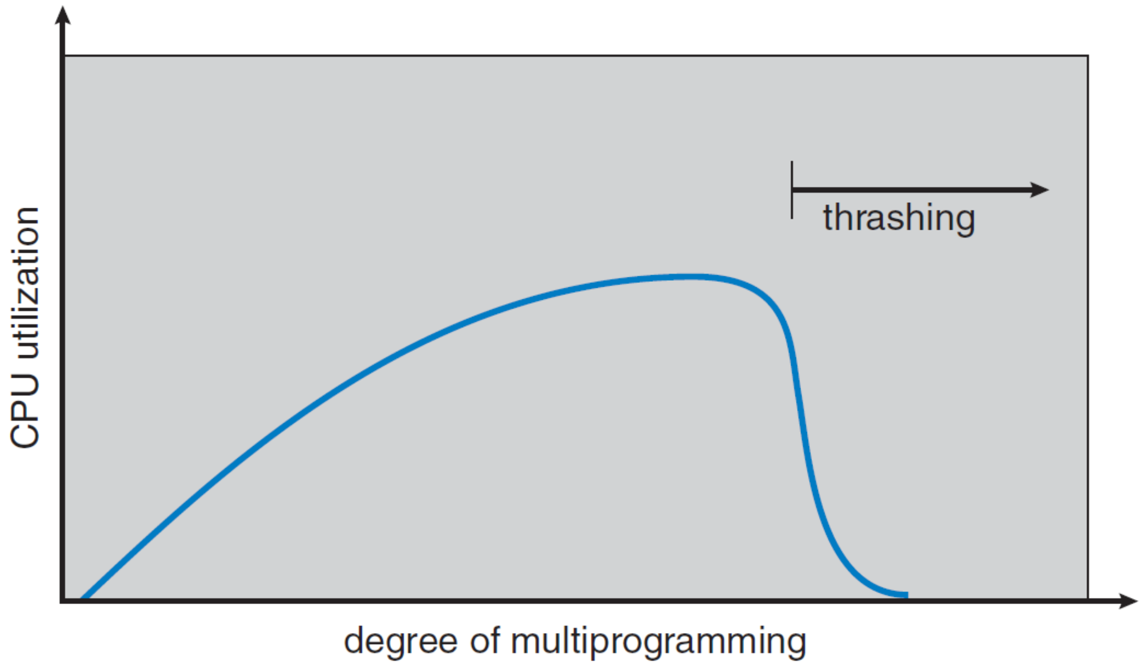
- Global / local allocation (replacement)
- Equal allocation
  - m frames among n processes
    - $\frac{m}{n}$ frames for each process
  - Memory waste
- Proportional allocation
  - Size of process p_i is s_i, then allocate
  - $a_i = \frac{s_i}{\sum s_i} \times m$
- Priority-based scheme
  - Ratio depends on both process size and priority

Issues - Thrashing

- If a process does not have enough frames – number of frames required to support pages in active use
  - Frequent page fault
    - Replace a page that will be needed again right away
  - This is called thrashing
    - Spend more time paging than executing

Issues - Thrashing

- Example: Multiprogramming + global page replacement
  - Increase CPU utilization (increase degree of multiprogramming)
  - Frequent page fault (queue up for paging, reduce CPU utilization, increase degree of multiprogramming)

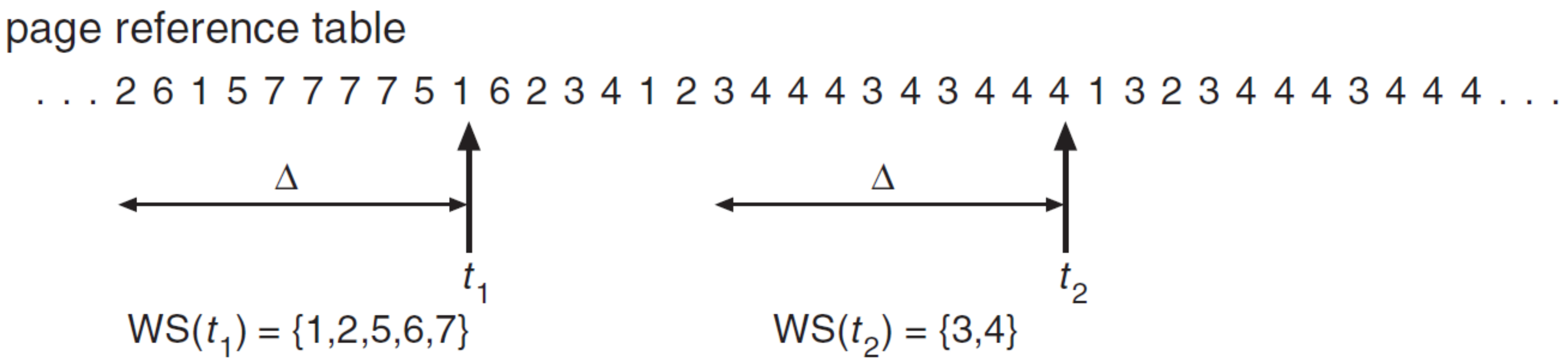


Issues - Thrashing

- How to address?
  - Local replacement/priority replacement
    - Will not cause other processes to thrash
    - Still not fully solve this problem
      - Increase average time for a page fault
      - longer queue for the paging device
      - longer effective access time even for non-thrashing processes

Issues - Thrashing

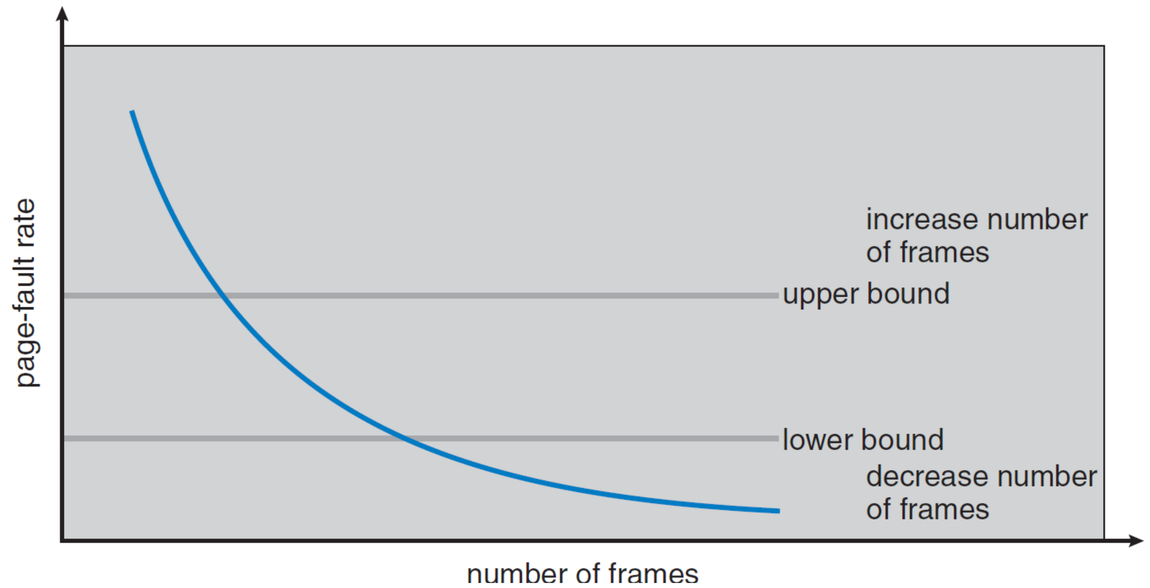
- How to address?
  - Provide as many frames as needed
    - Use working-set strategy to estimate needed frames
      - Working set: the set of pages in the recent Δ page references



$\sum WSS_i > \mathrm{m}$: thrashing may occur

Issues - Thrashing

- How to address?
  - Provide as many frames as needed
    - Use working-set strategy to estimate needed frames
      - Working set: the set of pages in the recent Δ page references
    - Use page-fault frequency



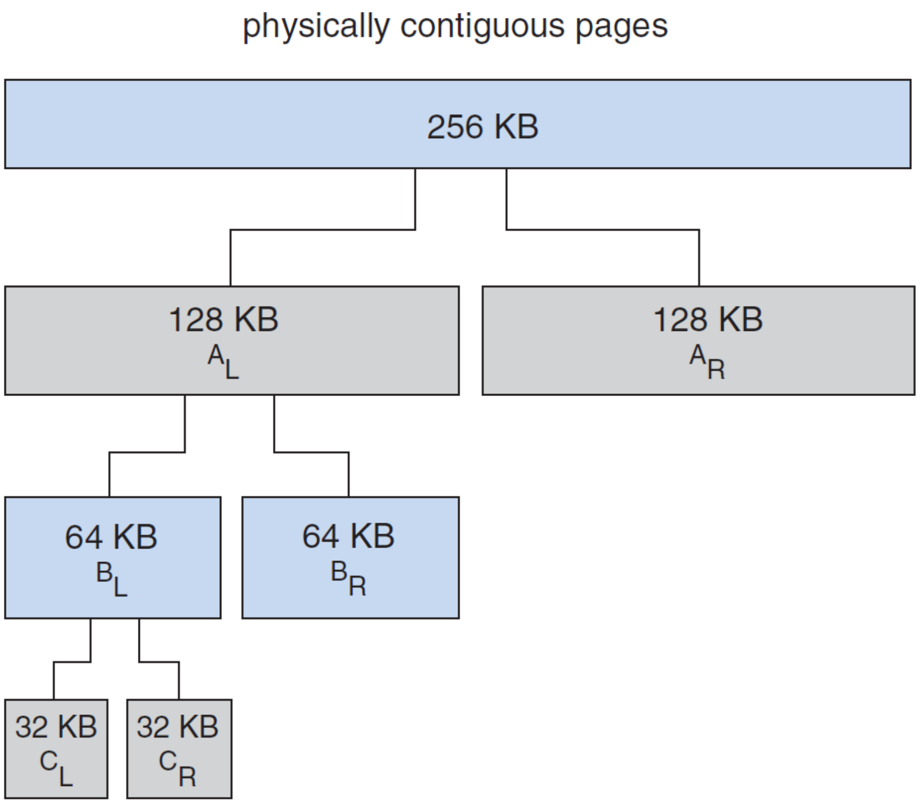
Allocation for kernel memory

- Kernel memory allocation requirement
  - Features
    - Varying (small) size requirement: different data structures
    - Contiguous requirement (certain hardware devices interact with physical memory)
  - Paging: Internal fragmentation

- Buddy system + Slab allocation

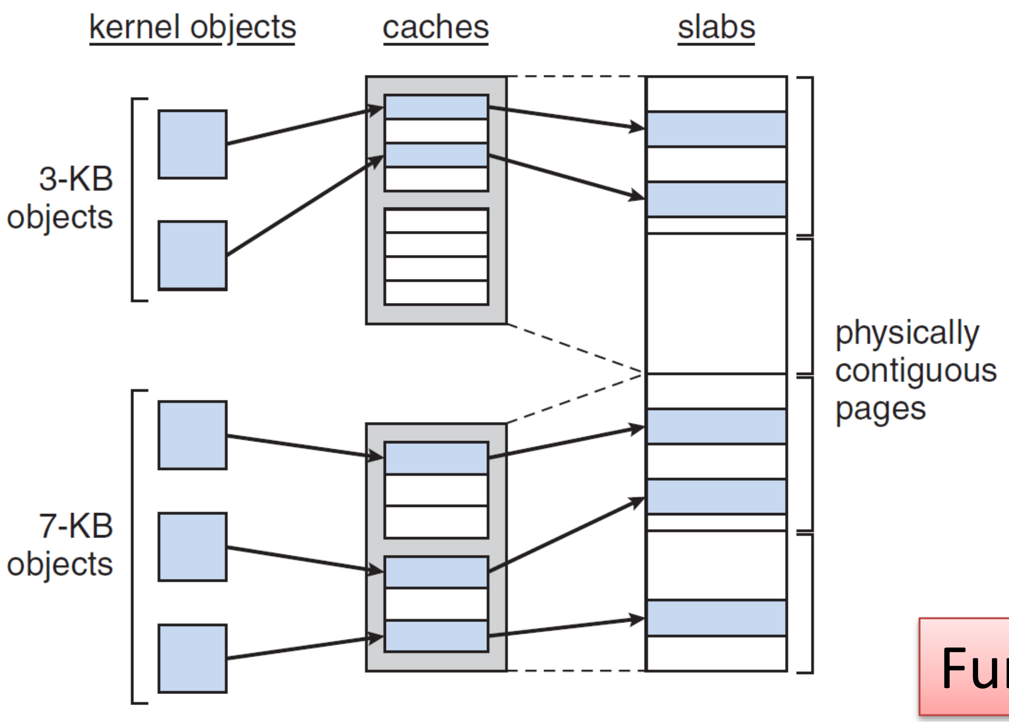
Buddy system

- Allocate memory from a fixed-size segment
  - Power-of-2 allocator (11 orders)
  - Advantage: coalescing



Slab allocation

- Allocate memory for small objects (limit fragmentation)
  - Slab: one/more contiguous pages
  - Cache: one/more slabs
    - A separate cache for each unique kernel data structure



Reduce fragmentation

Fast allocation (caching benefit)

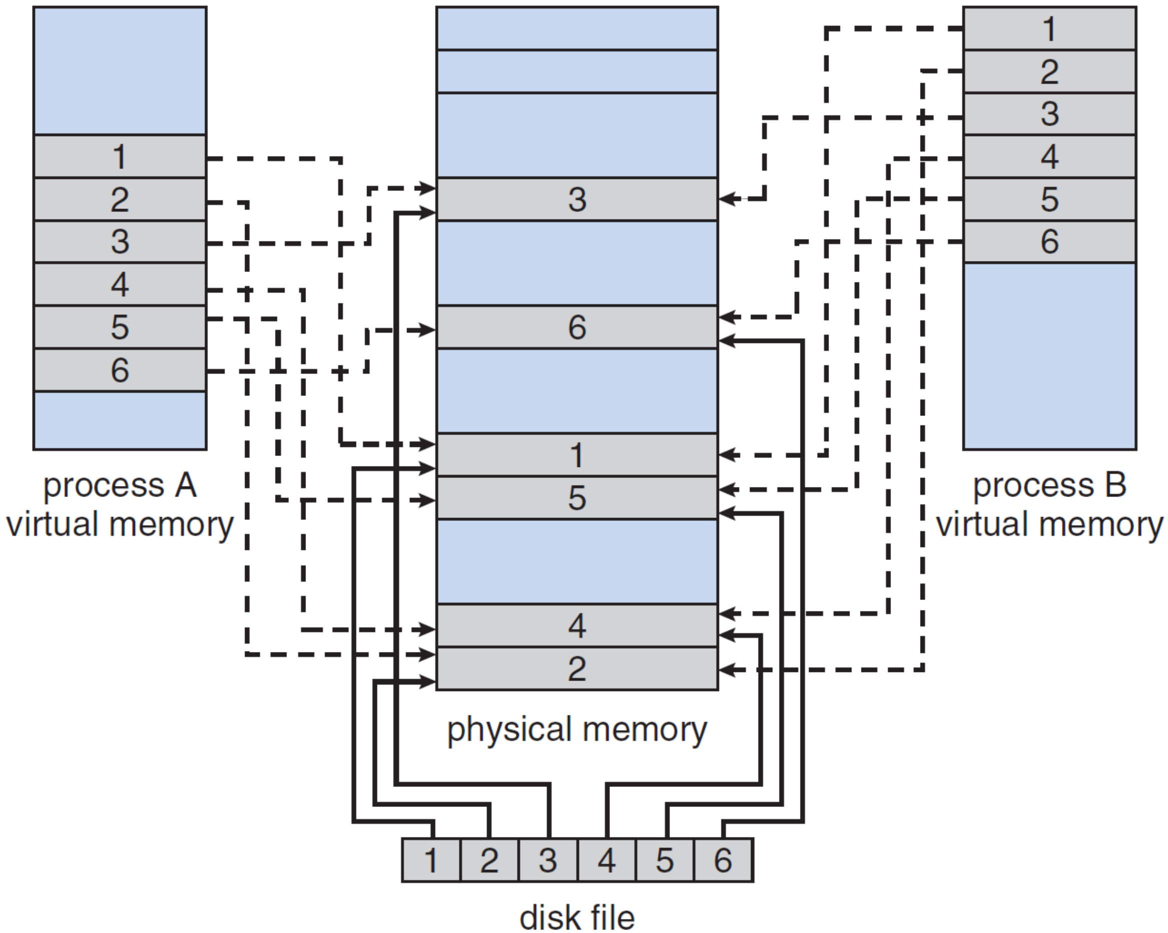
Further reading: SLOB/SLUB

Memory mapped file

- Ordinary file access
  - open(), read(), write()
  - System call + disk access
- Memory mapped file
  - Memory mapping a file: associate a part of the virtual address space with the file
  - File access
    - Initial access to file: demand paging
    - Subsequent reads/writes: routine memory accesses
    - Improves performance
  - Refer to mmap(2) system call

Memory mapped file

- Also allow multiple processes to map the same file



Summary

- We have introduced…
  - Segmentation
  - Paging + page table
  - Demand paging + COW + page replacement algorithms
  - Allocation of frames
    - User process
    - Thrashing
    - Kernel memory (buddy + slab)
  - Memory-mapped file
- More…
  - **`malloc()`** is not that simple: refer to "glibc malloc"
  - Other page-replacement algorithms

Hope you enjoyed the OS course!