# Operating Systems

Eric Lo

## 13 - Virtual Memory

# Two **different processes**, use the **same address**?!

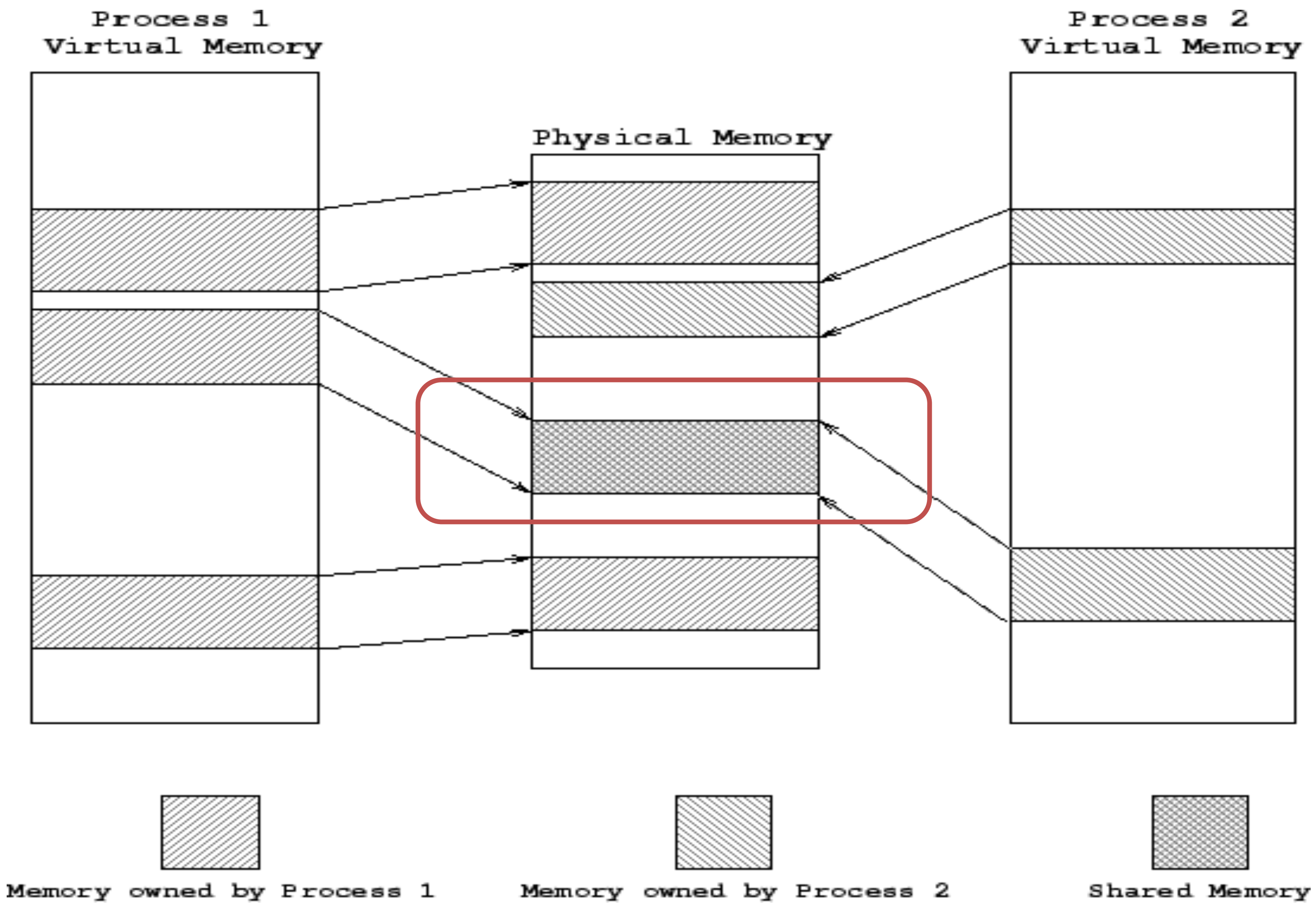
```
int main(void) {
    int pid;
    pid = fork();
    printf("PID %d: %p.\n", getpid(), &pid);
    if(pid)
        wait(NULL);
    return 0;
}
```

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

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

# Virtual memory

- This is also how threads/processes share memory



# Two **different processes**, use the **same address**?

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

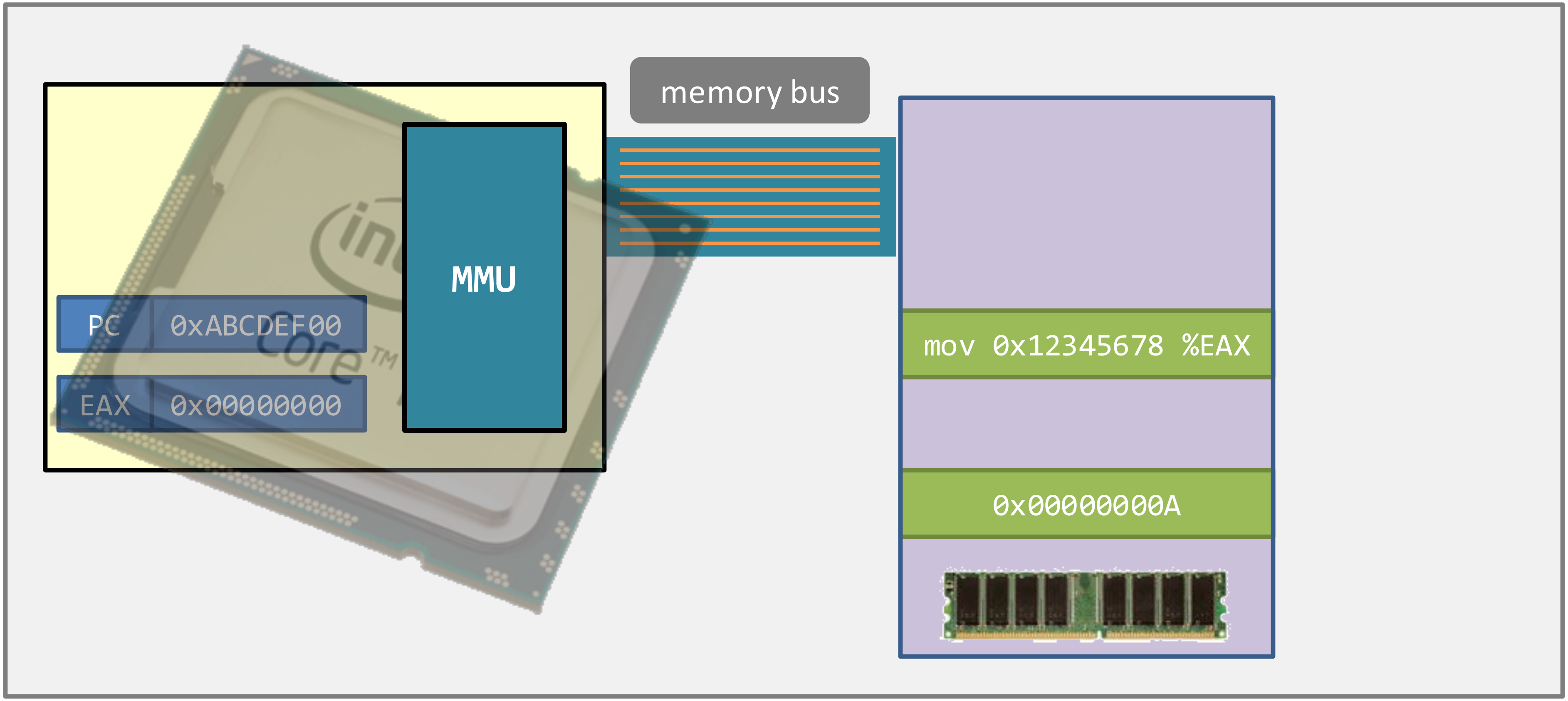
```
$ ./same_addr
PID 1234: 0xbfe85e0c.
PID 1235: 0xbfe85e0c.
$ _
```

- Different processes may use the same virtual addresses
  - Processes 1234 and 1235 after all are exactly the same program
    - So, local variables shall go to the same stack area
  - But they get **translated to different physical addresses**

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

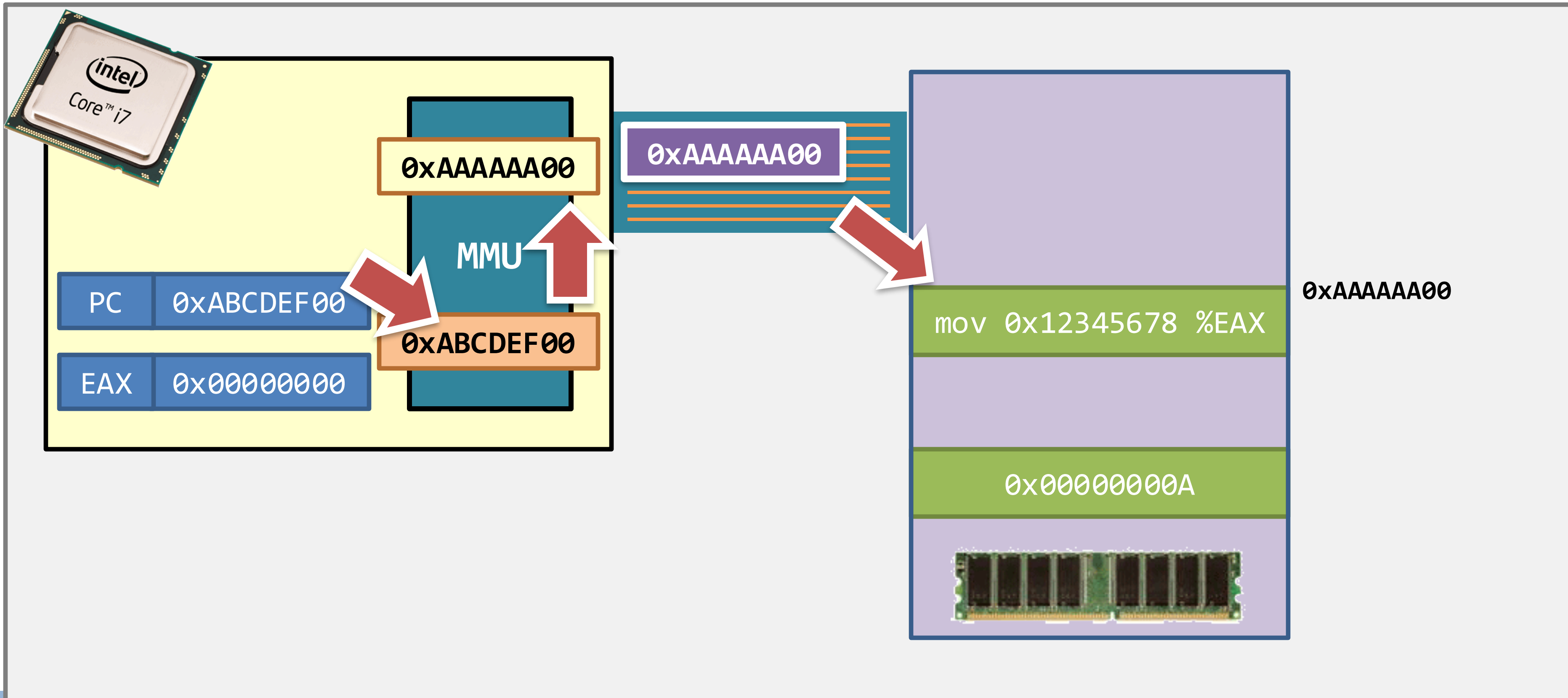
# Virtual memory support in modern CPUs

- The **MMU – memory management unit**
  - Usually on-chip (but some architecture may off-chip)



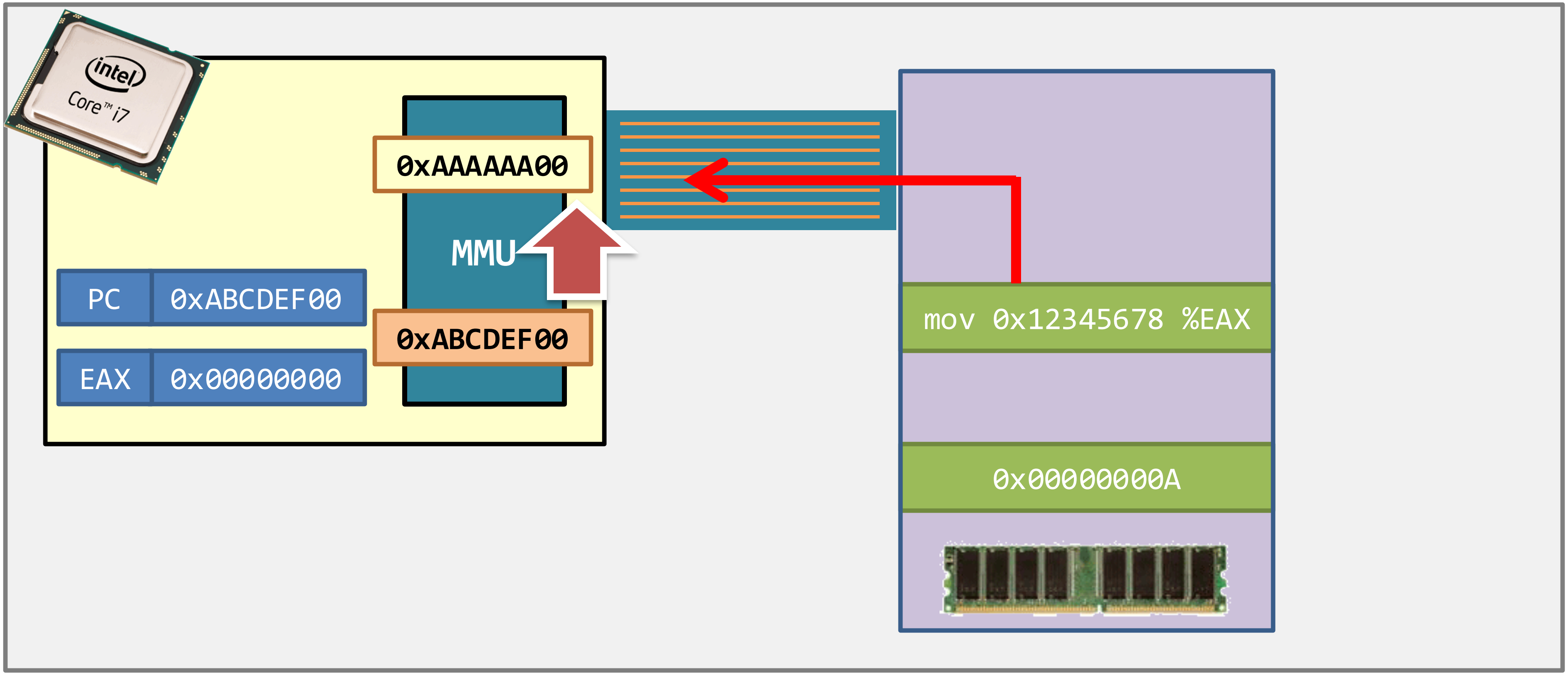
# Virtual memory – how does it work?

- Step 1. When CPU wants to fetch an instruction
  - the **virtual address** is sent to MMU and
  - MMU refers to a "*page table*" in the memory and translates it into a **physical address**.



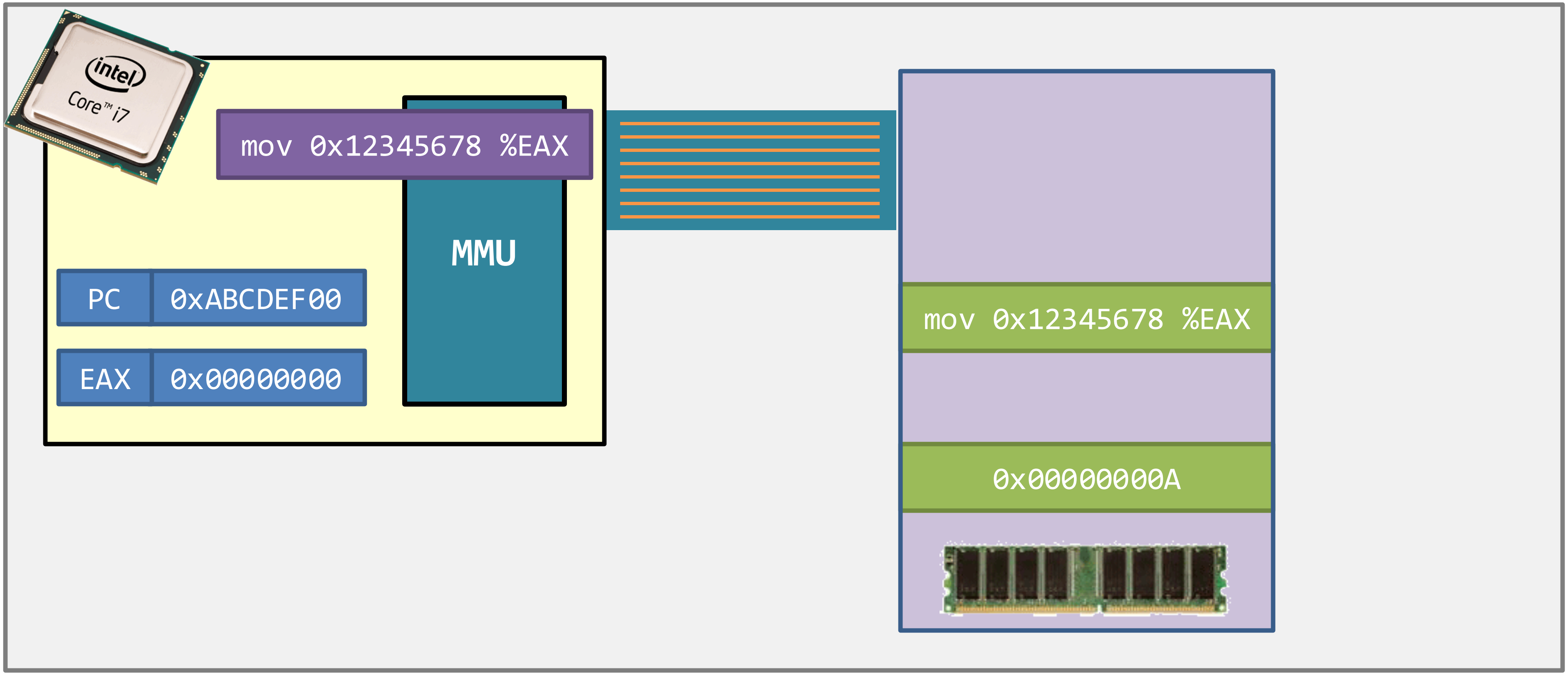
# Virtual memory – how does it work?

- Step 2. The memory returns the instruction



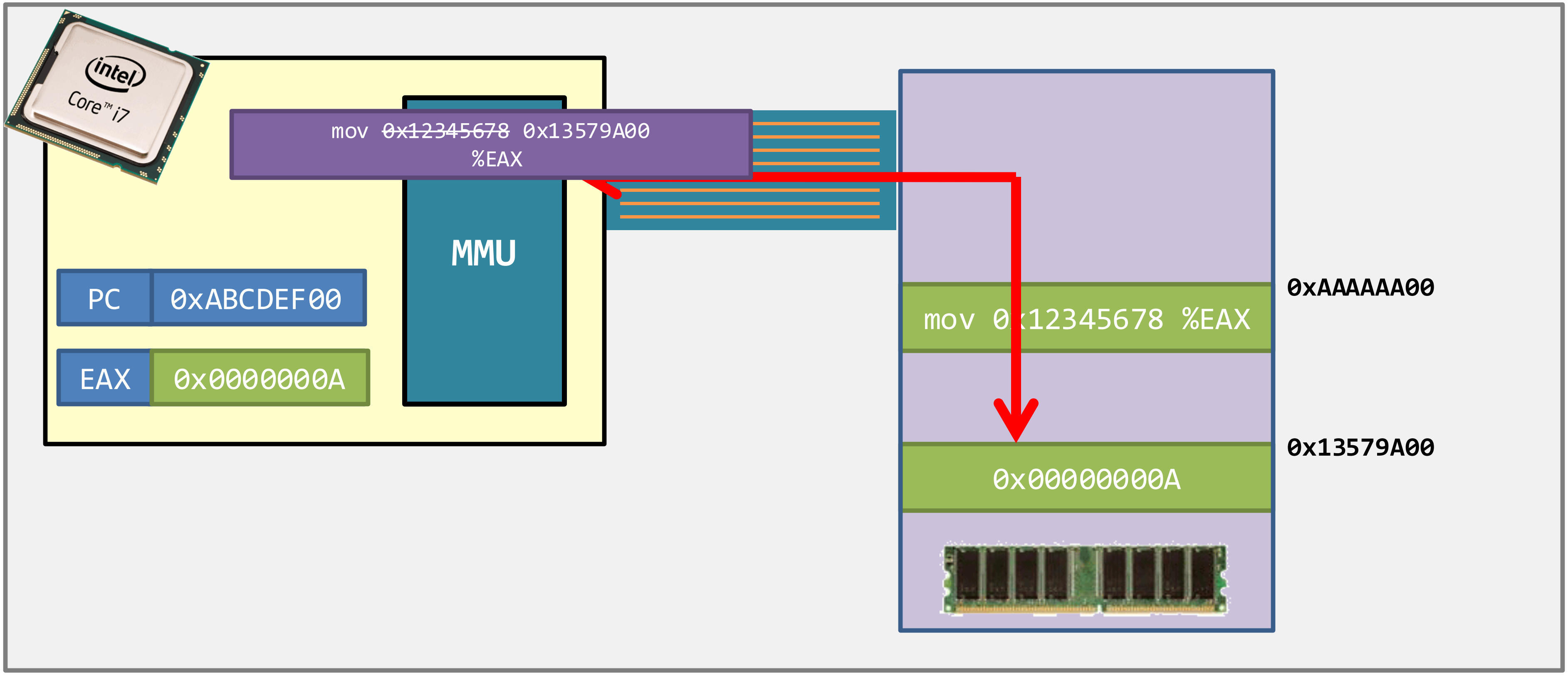
# Virtual memory – how does it work?

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

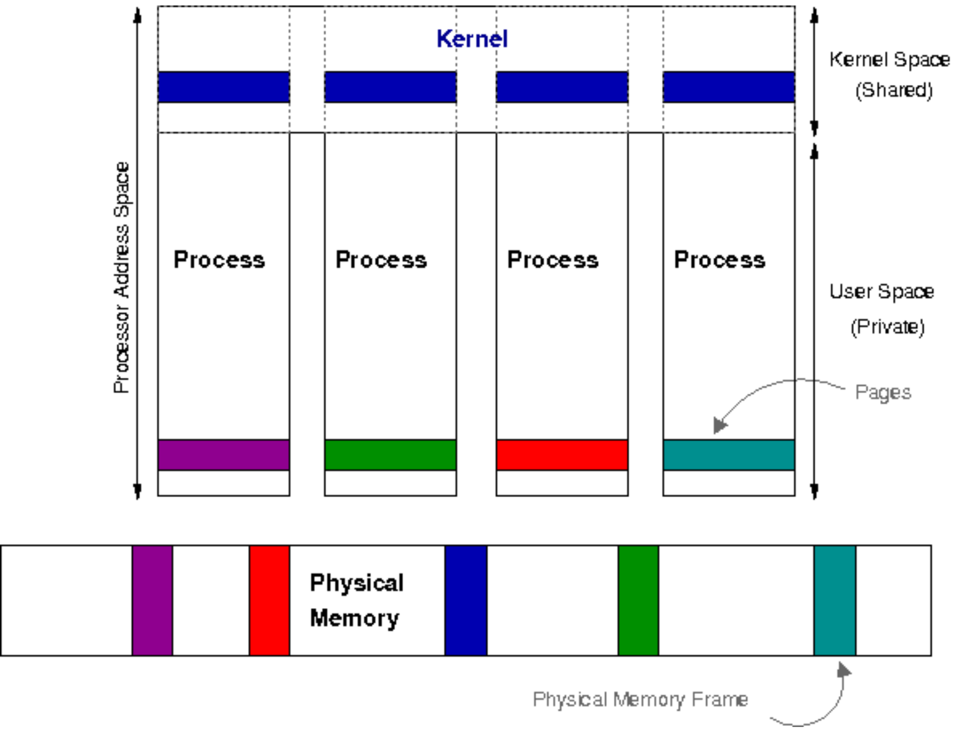


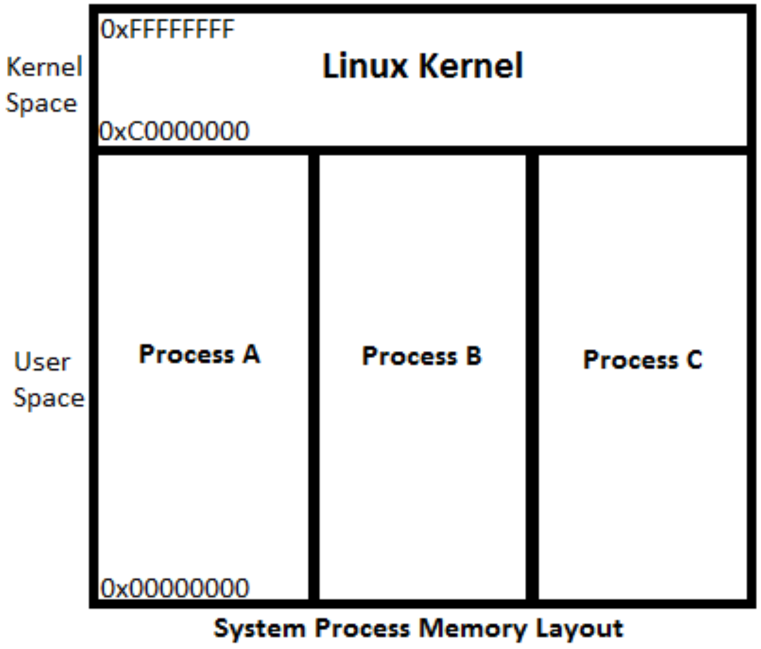
# Virtual memory – how does it work?

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



# Kernel space and user space



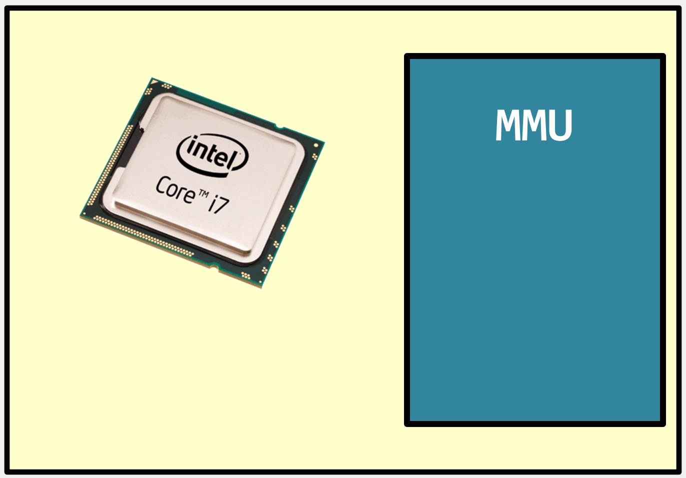



## Memory Management

- Virtual memory = CPU + MMU
- Paging


# Translation table?

- Translation is done by a **lookup table**?
  - Every process has **its own** lookup table in the kernel space
  - The MMU will lookup that table




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?

**32-bit CPU**
**How many addresses are there?** $2^{32}$

**How large is a 32-bit address?** **4 bytes**

**Only this column is stored.**

| Virtual address | Physical Address |
|---|---|
| 0x00000000 | 0x01234567 |
| 0x00000001 | 0x452796AB |
| …… | …… |
| 0xFFFFFFFF | 0x6714EFD8 |

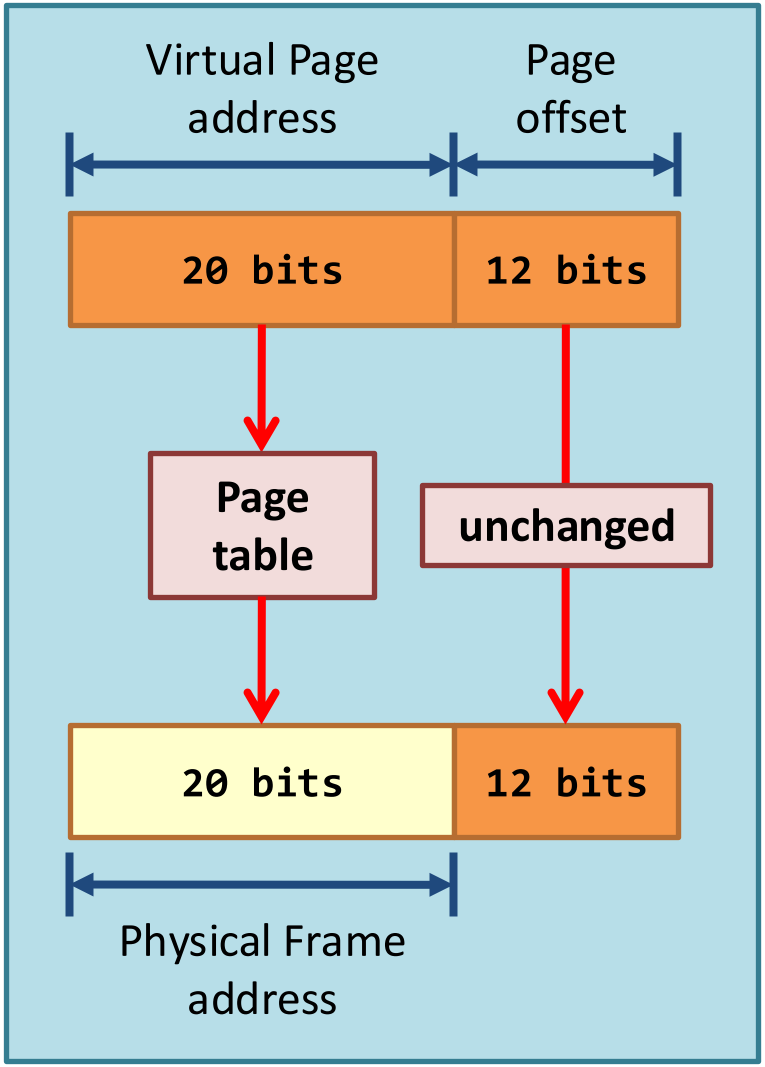
**Lookup Table internals**

```
Size of the lookup table =

        Number of addresses
   x    Size of an address
```

**$2^{32}$ x 4 bytes = 16 Gbytes**

# MMU implementation – paging

- Partitions the memory into fixed blocks called **frames**.
- The lookup table is now called the **page table**.

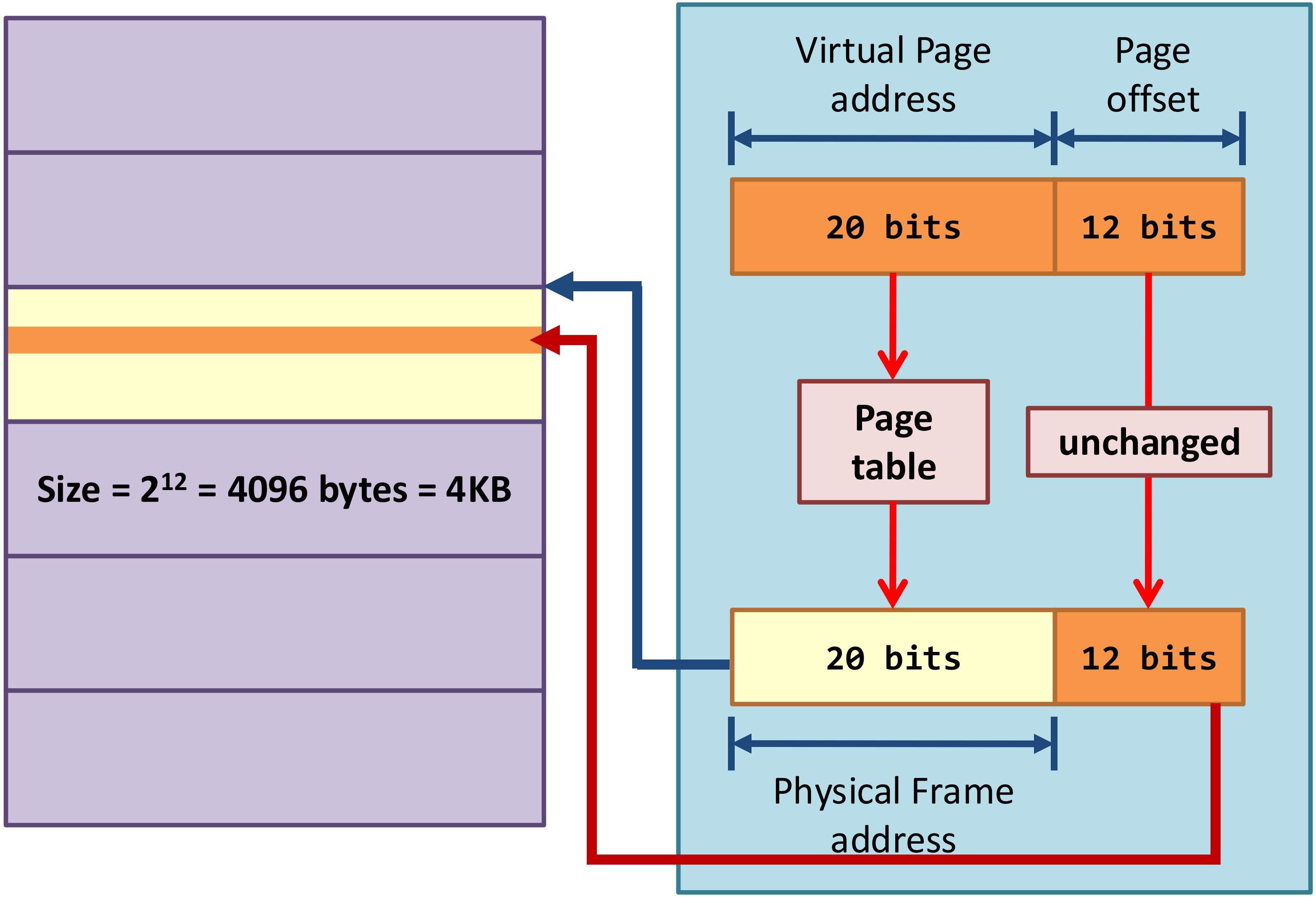


# Paging Analogy

- Virtual Address
  - SHB Room 107
- Physical Address
  - Ho Sin Hang Engineering Building Room 107

| 20-bit Virtual Page Address | 20-bit Physical Frame Address |
|---|---|
| SHB | Ho Sin Hang Engineering Building |
| ... | ... |

# Paging



Size = 2^12 = 4096 bytes = 4KB
Virtual Page address
Page offset
20 bits
12 bits
Page table
unchanged
20 bits
12 bits
Physical Frame address

# Page Table Size

Size of the page table =

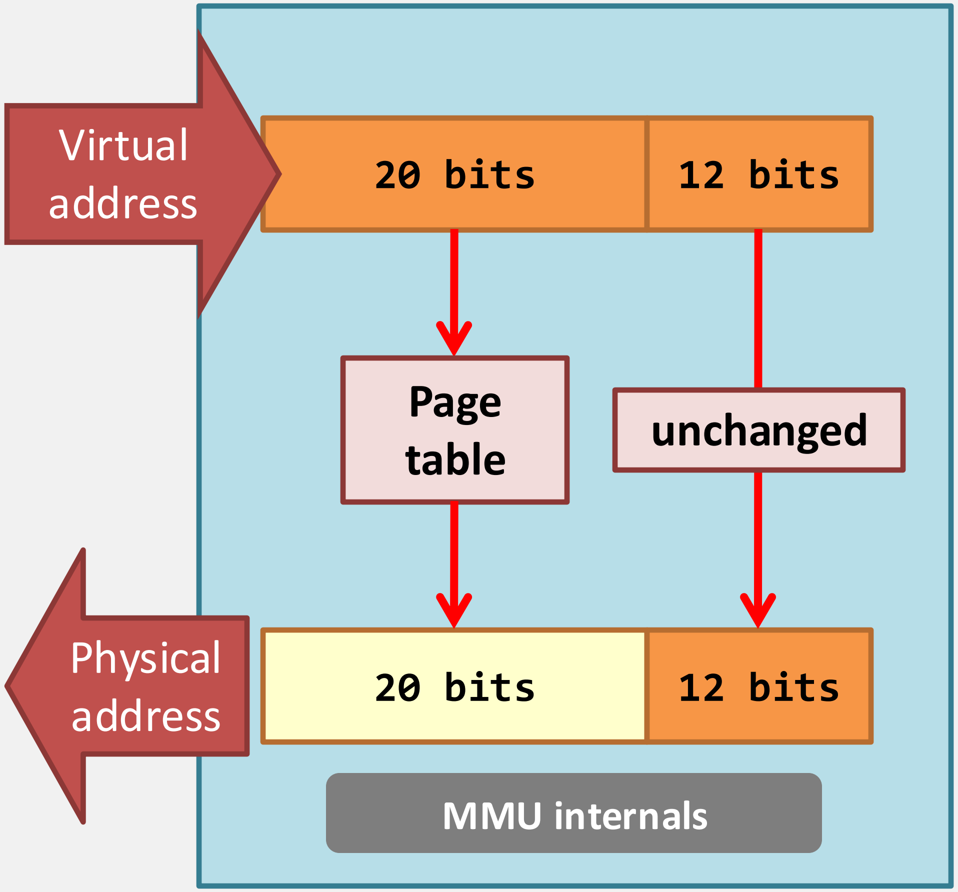
Number of addresses

x Size of a 32-bit entry

$2^{\cancel{32}20}$ x 4 bytes = 4 Mbytes

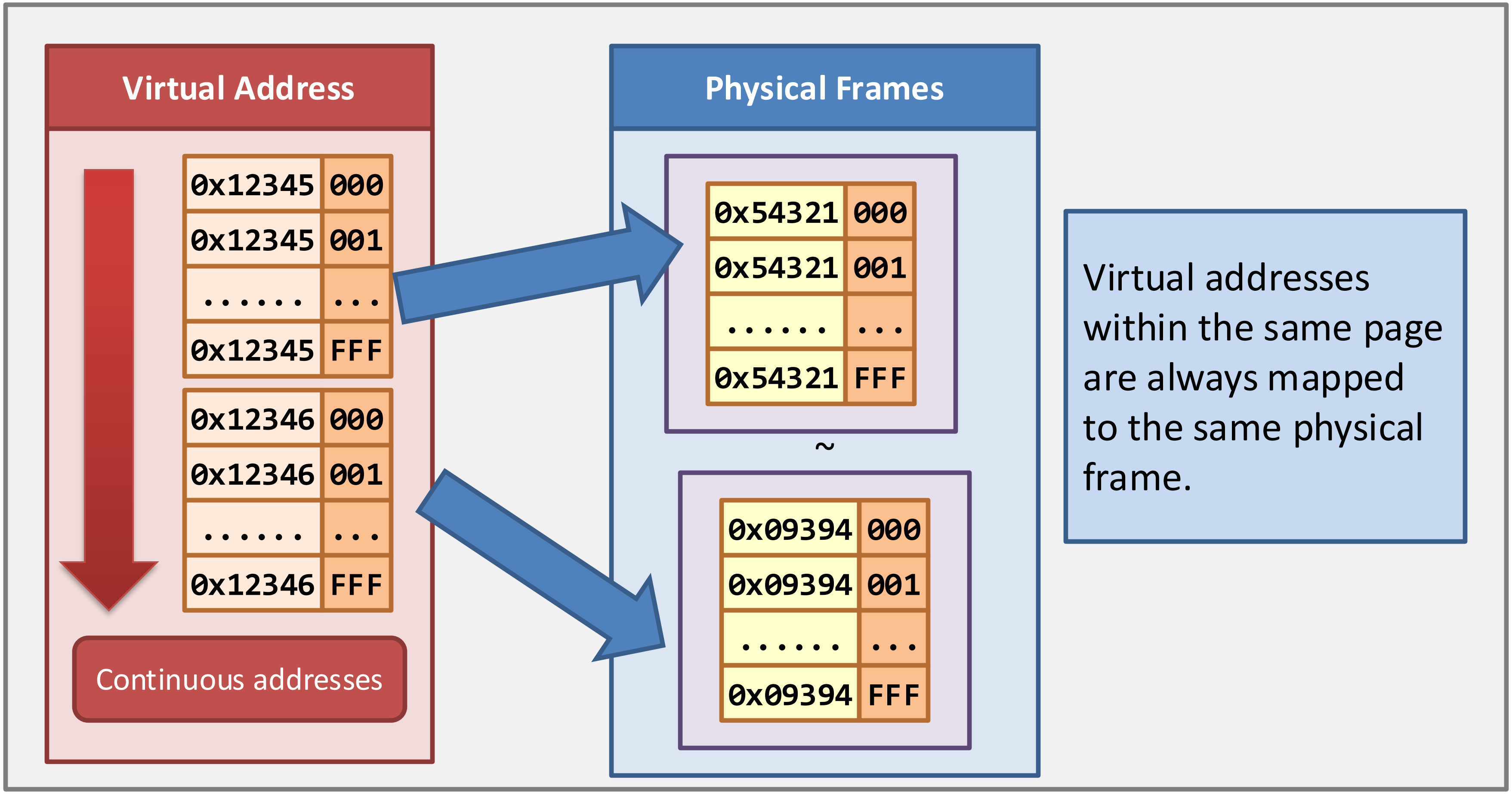
A page table 32-bit entry contains:

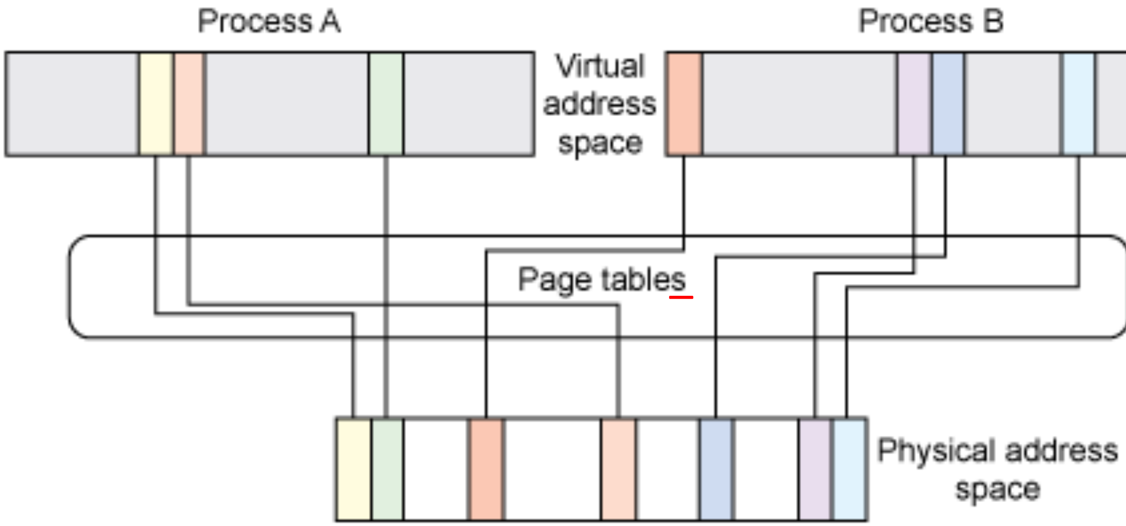
- 20-bit address
- Control Bits (e.g., Permission attributes, etc.)



# Paging - properties

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




Process A
Process B
Virtual address space
Page tables
Physical address space

# Paging – memory allocation

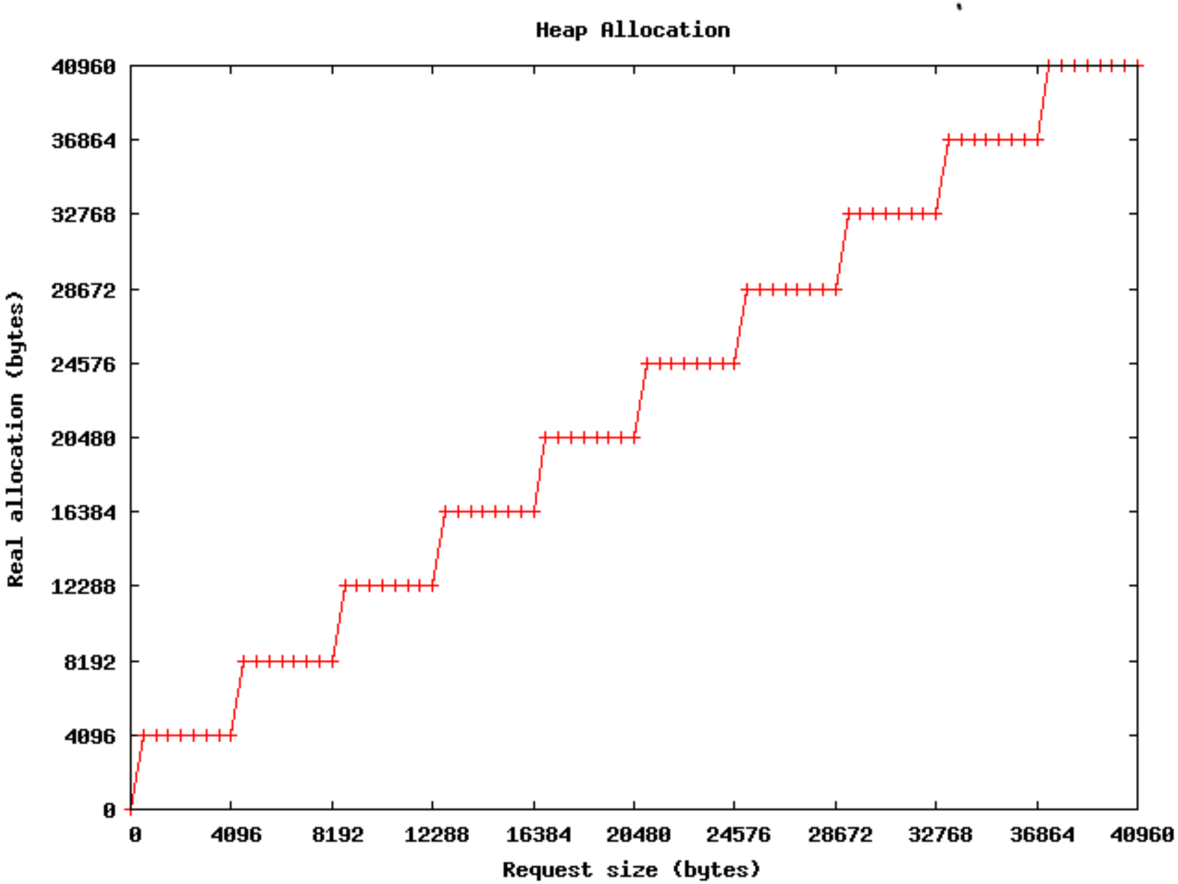
- A page is the basic unit of memory allocation

```c
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);// end address of the current heap.
    sbrk(1);                 // increase heap by 1 byte
    while(1)
        c = *(++ptr);
}
```

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

# Paging – memory allocation

- A page is the basic unit of memory allocation.




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

The same for the growth of the stack.

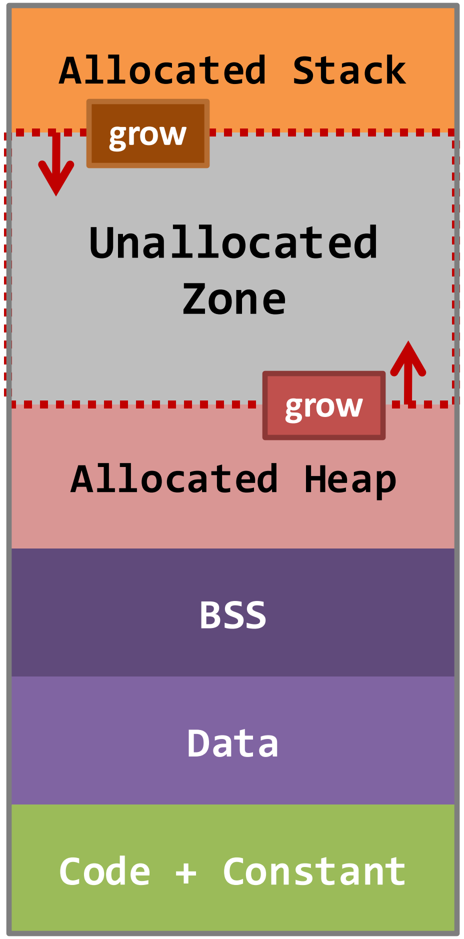
The whole page is belong to you

You malloc(4 bytes)

- By luck you can ptr++ beyond 4 bytes and got no segmentation fault until the page limit

# Paging – internal fragmentation

- Default page size is 4,096 bytes.
- Page size is able to set as 4MB under x86_64
  - "Huge Page"




**Internal fragmentation** means some space is wasted when allocation is done in a page-by-page manner.

# The Page Table

| Virtual Page # | Page Table of Process A | | |
|---|---|---|---|
| | **Permission** | **Valid-invalid bit** | **Frame #** |
| 0 | `rwx-` | 1 | 58 |
| 1 | `rwx-` | 0 | 66 |
| 2 | `r--s` | 1 | 72 |
| 3 | `NIL` | 0 | `NIL` |
| … | … | … | … |

The physical memory is organized as **an array of frames**. The size of a frame is the same as the page size

This row means the virtual page "0" is mapped to the physical frame "58".

# Paging – segmentation fault

| Virtual Page # | Page Table of Process A | | |
|---|---|---|---|
| | Permission | Valid-invalid bit | Frame # |
| 0 | rwx- | 1 | 58 |
| 1 | rwx- | 0 | 66 |
| 2 | r--s | 1 | 72 |
| 3 | NIL | 0 | NIL |
| ... | ... | ... | ... |

`s – means sharable.`

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

When a virtual address is translated to an unallocated frame…

When you write to read-only pages…

When you try to execute a non-executable pages…

# The Page Table

| Virtual Page # | Page Table of Process A | | |
|---|---|---|---|
| | Permission | Valid-invalid bit | Frame # |
| 0 | rwx- | 1 | 58 |
| 1 | rwx- | 0 | 66 |
| 2 | r--s | 1 | 72 |
| 3 | NIL | 0 | NIL |
| ... | ... | ... | ... |

This bit is to tell the CPU whether the frame is in memory or not. If it is 0

- Frame #66 **was** **paged-out** to the **swap area** of the disk
- If this virtual page #1 is re-accessed,
  - **page fault**!
- The content of virtual page #1 shall be **paged-in** from disk to the memory (possibly to another frame)

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

# Memory / allocation – demand paging

- 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.

```c
#define ONE_MEG (1024 * 1024)
#define COUNT   3072

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++) {
        memset(ptr[i], 0, ONE_MEG);
    }
}
```

`malloc()`:
- actually does **not** involve any memory access
- only involve **the allocation of the *virtual* address page.**

**<u>So, this loop is only for enlarging the virtual heap.</u>**

This statement really accesses the "*<u>allocated</u>*" memory.

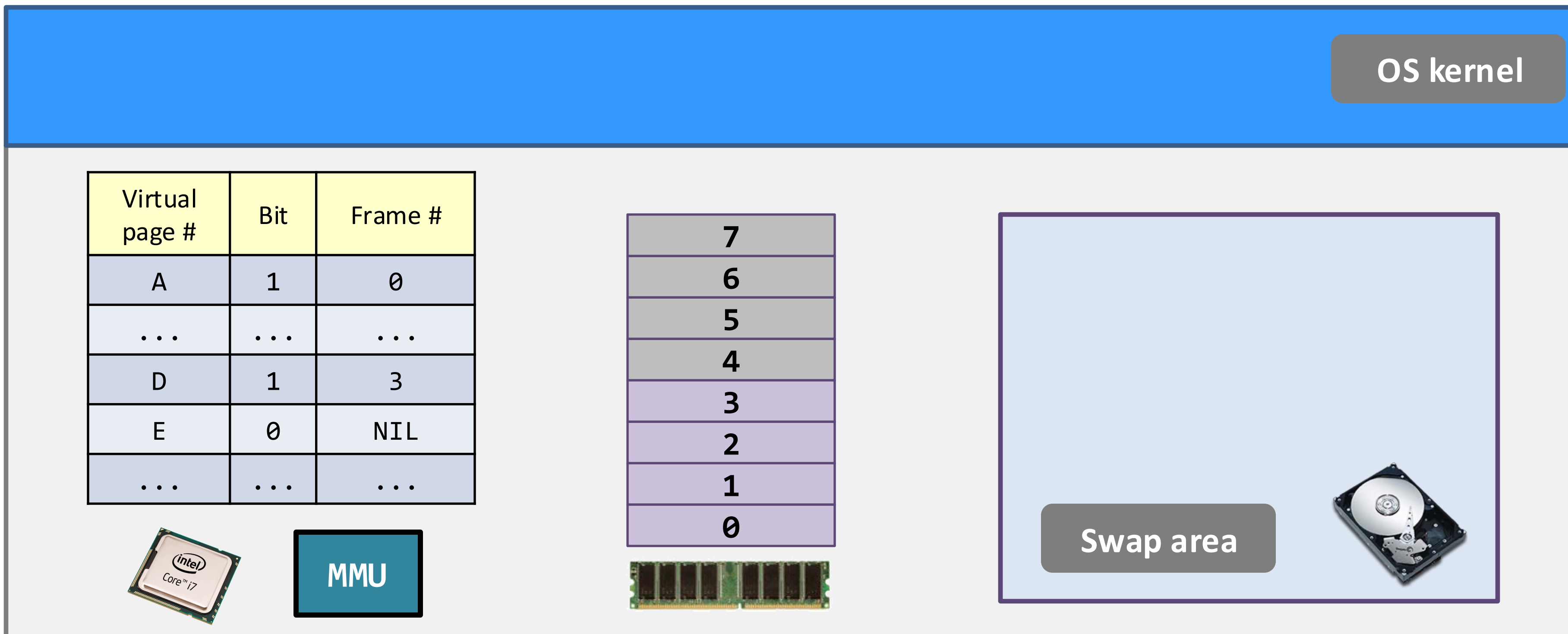
So, this statement really **accesses** the memory.

```
[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 - 12.

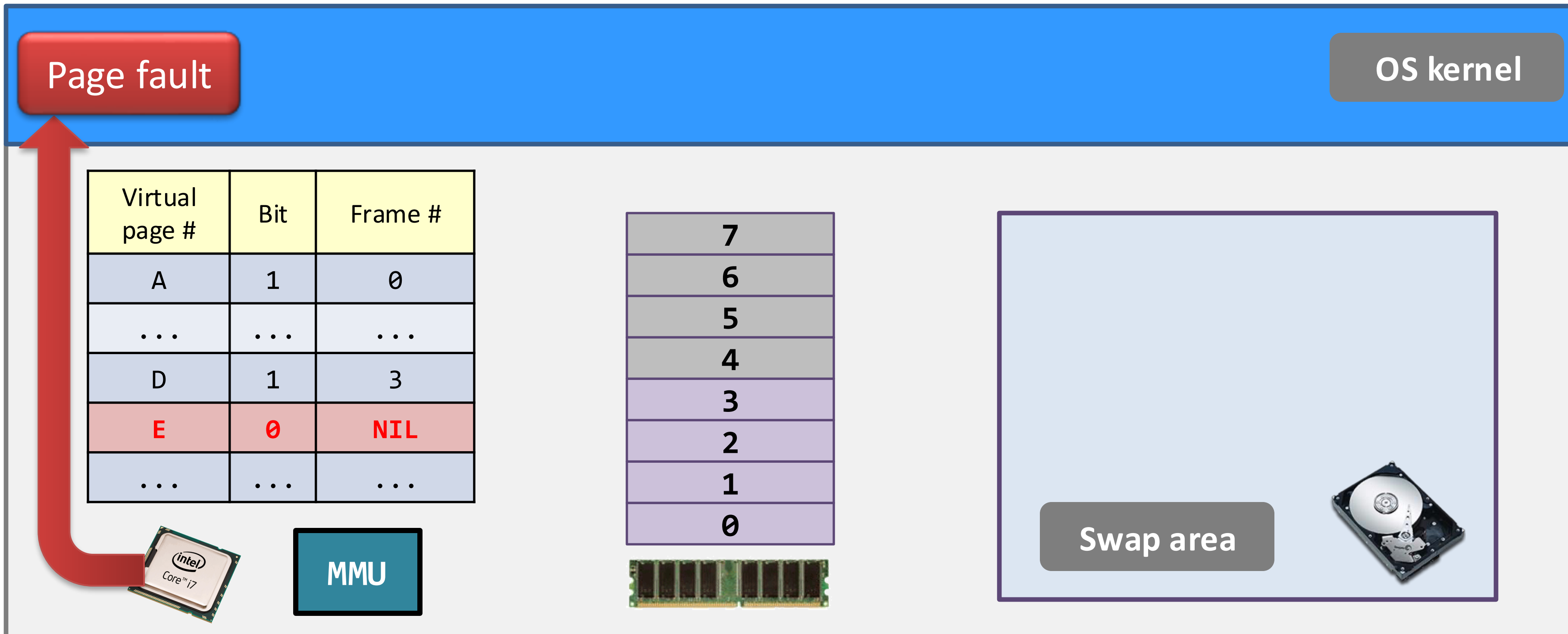



| Virtual page # | Bit | Frame # |
|---|---|---|
| A | 1 | 0 |
| ... | ... | ... |
| D | 1 | 3 |
| E | 0 | NIL |
| ... | ... | ... |

# 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 exception called **page fault**.

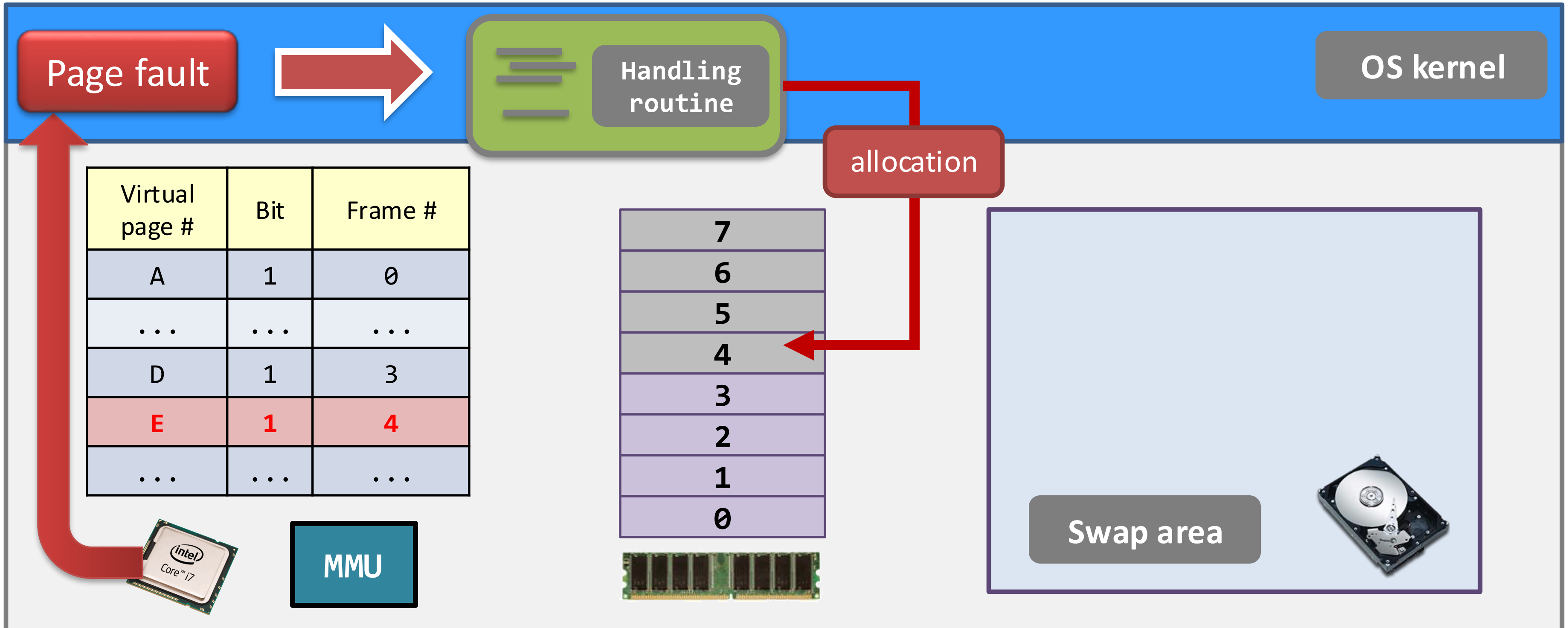



| Virtual page # | Bit | Frame # |
|---|---|---|
| A | 1 | 0 |
| ... | ... | ... |
| D | 1 | 3 |
| E | 0 | NIL |
| ... | ... | ... |

# Demand paging – illustration.

Assumption: 1 process only.

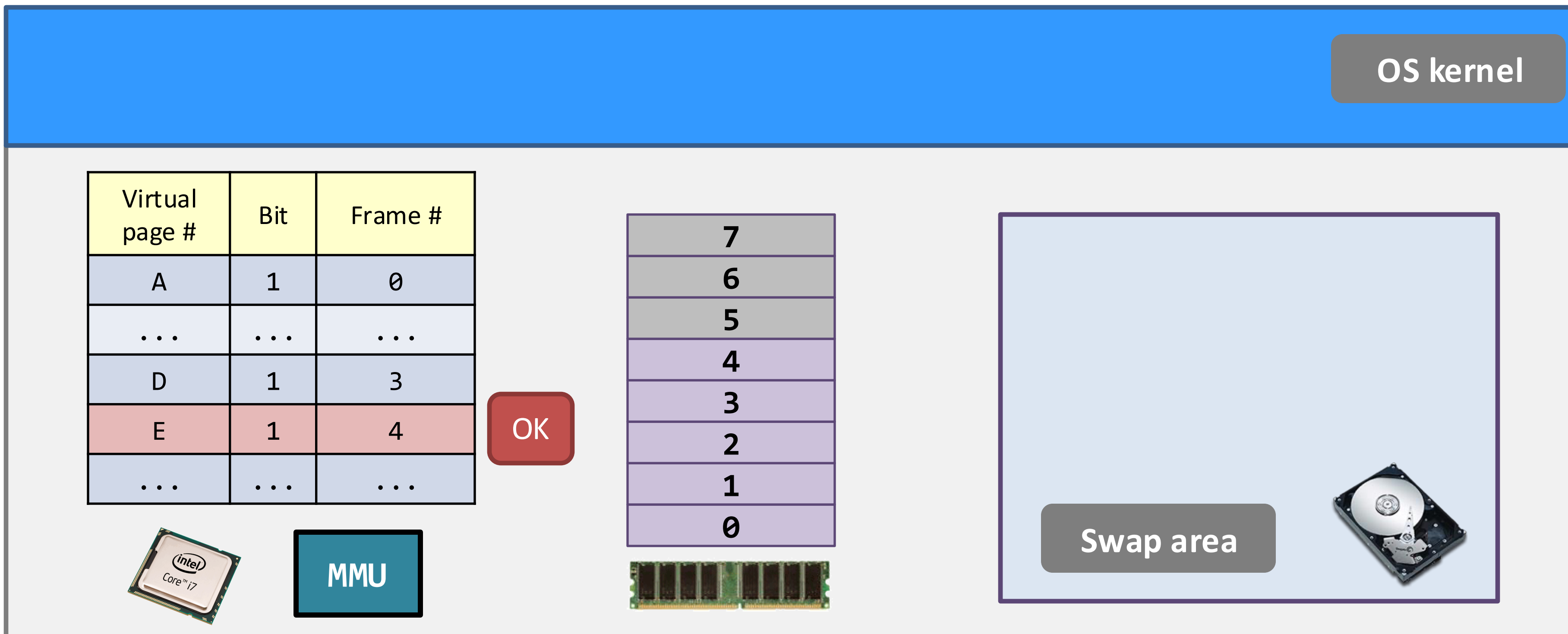
- The **page fault handling routine** is running:
  - The kernel knows the page allocation for all processes.
  - It allocates a memory frame for that request.
  - Last, the **page table entry** for Page E is updated.



# 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.

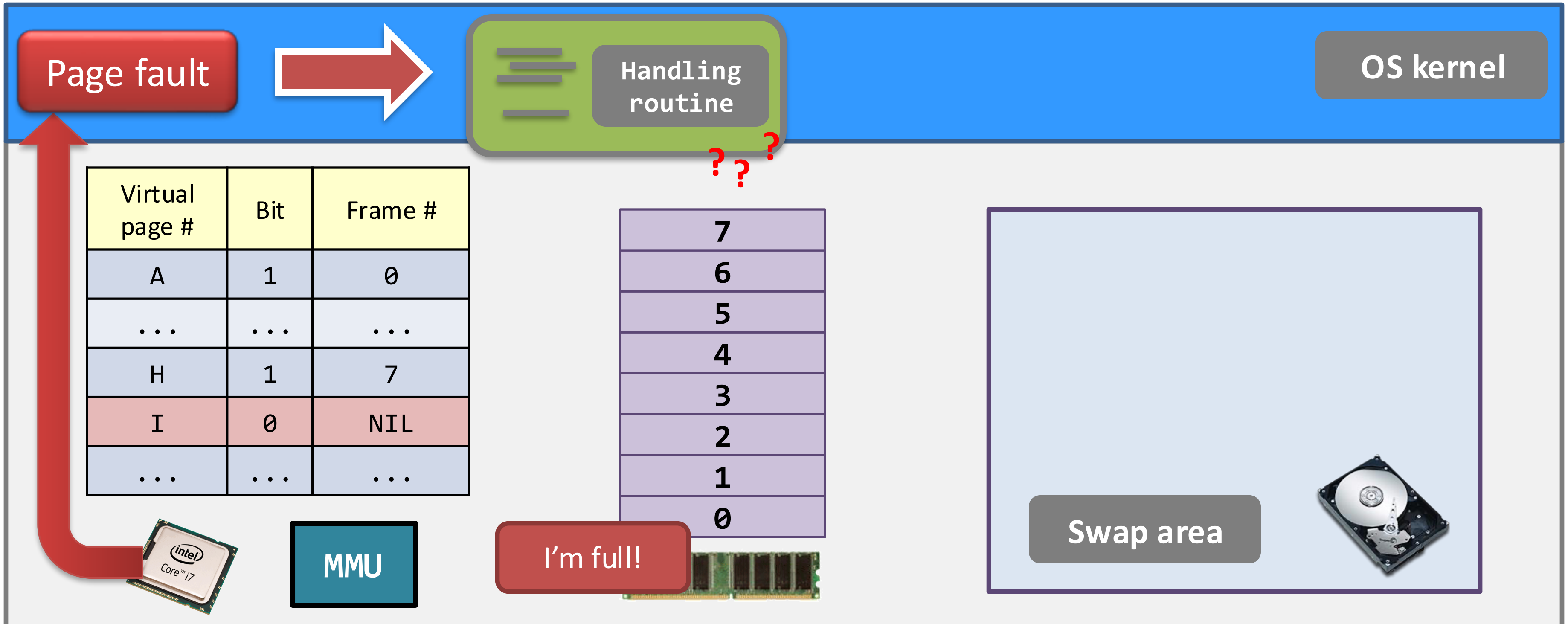



| Virtual page # | Bit | Frame # |
|---|---|---|
| A | 1 | 0 |
| ... | ... | ... |
| D | 1 | 3 |
| E | 1 | 4 |
| ... | ... | ... |

# 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**.

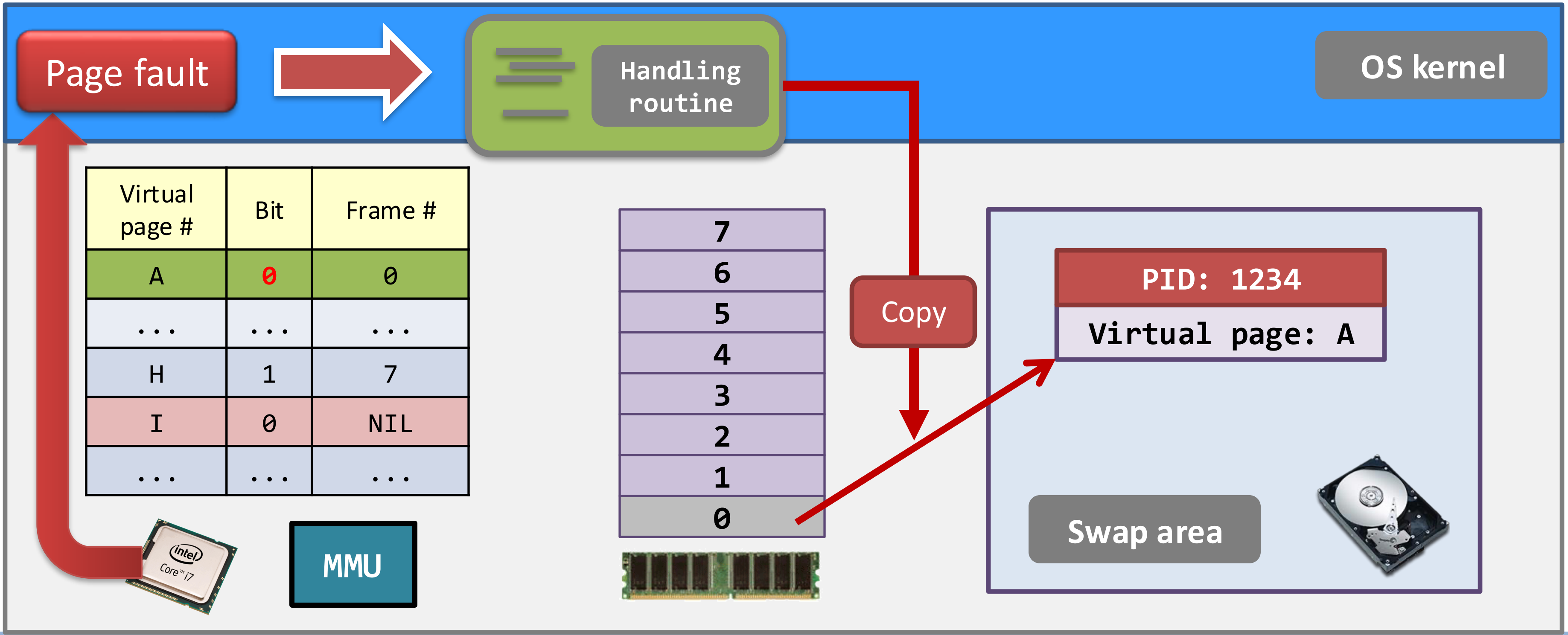



| Virtual page # | Bit | Frame # |
|---|---|---|
| A | 1 | 0 |
| ... | ... | ... |
| H | 1 | 7 |
| I | 0 | NIL |
| ... | ... | ... |

# Demand paging – illustration.

Assumption: 1 process only.

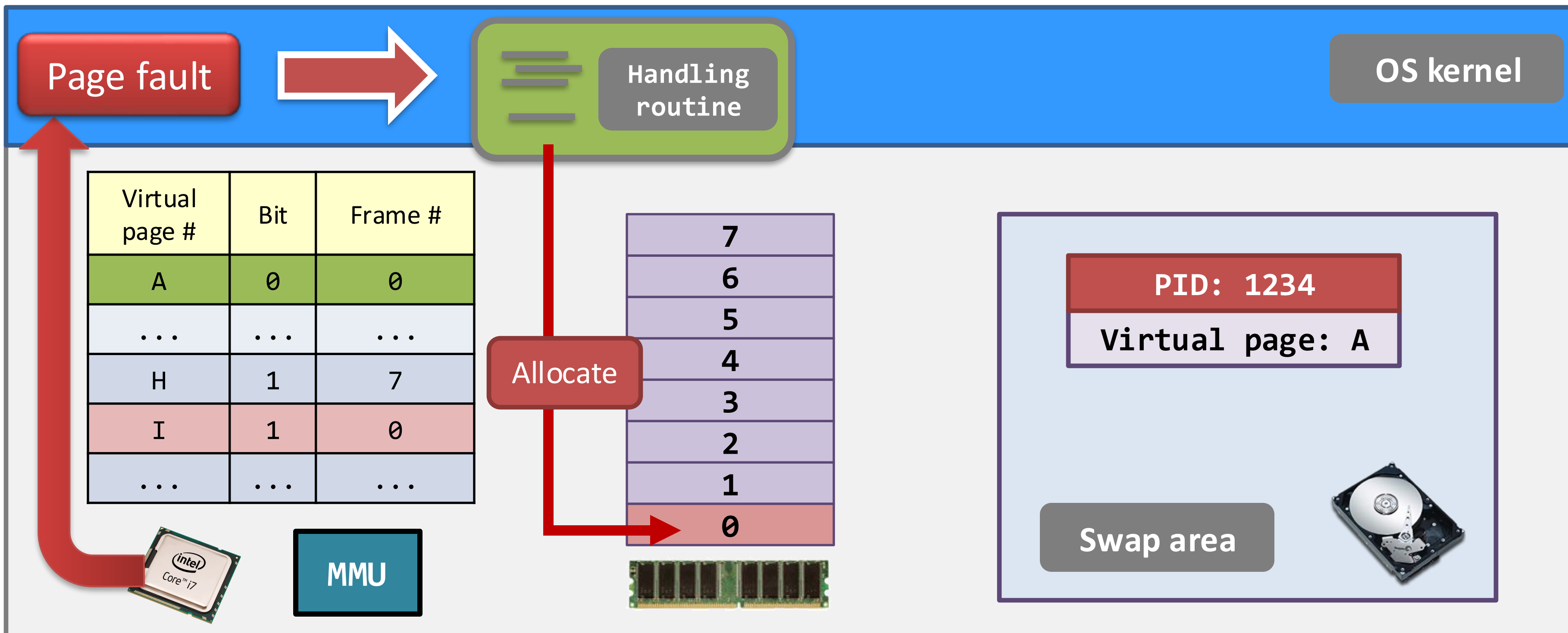
- Using the swap area:
  - Step (1) Select a **<u>victim virtual page</u>** and copy the victim to the swap area.
    - Now, <u>Frame 0 is a free frame</u> and <u>the bit for Page A is 0</u>.



# 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.

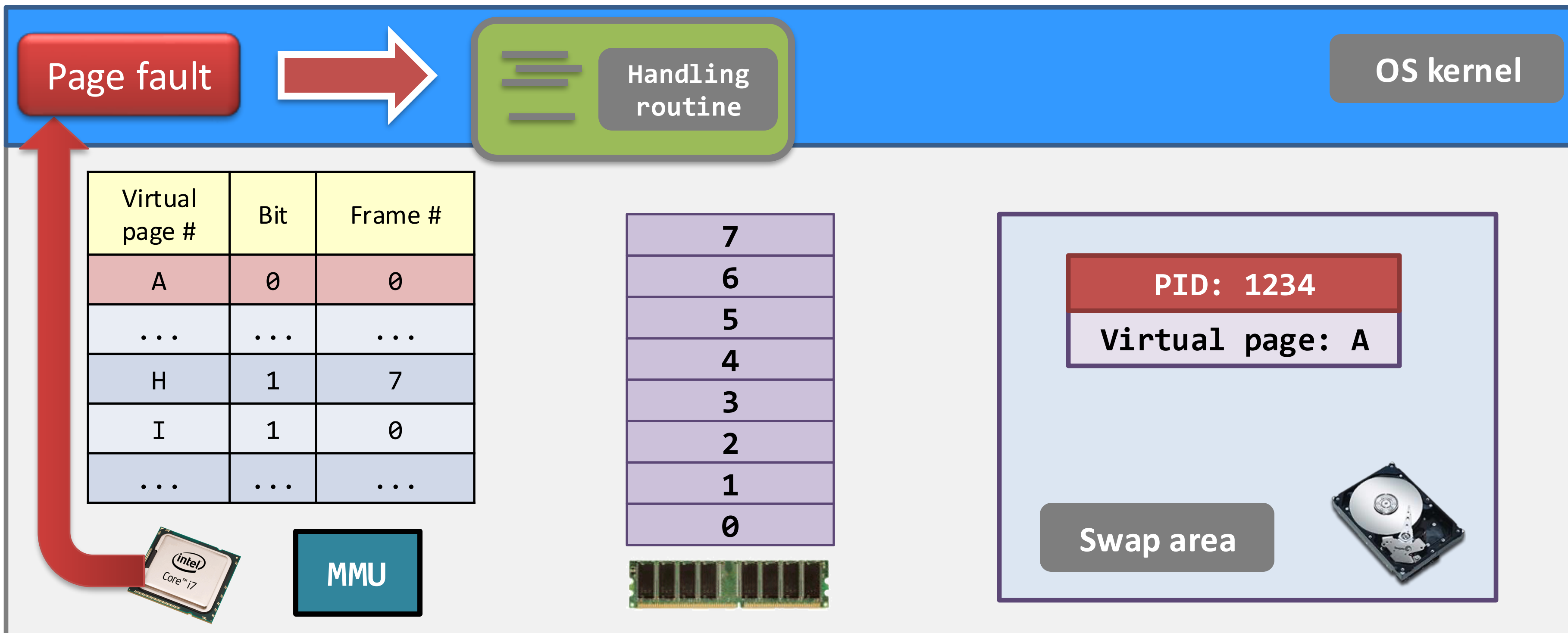



| Virtual page # | Bit | Frame # |
|---|---|---|
| A | 0 | 0 |
| ... | ... | ... |
| H | 1 | 7 |
| I | 1 | 0 |
| ... | ... | ... |

# Demand paging – illustration.

**Assumption: 1 process only.**

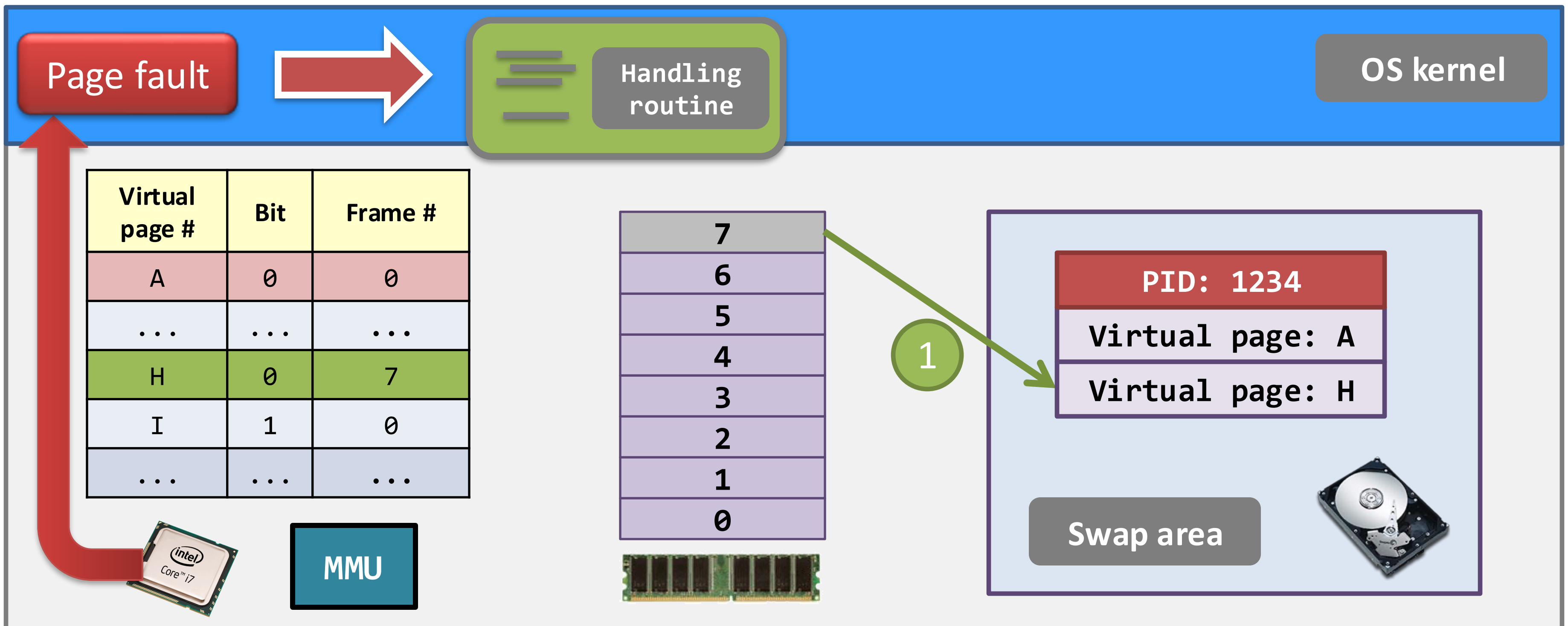
- How about **virtual page A** is accessed again?
  - Of course, a page fault is generated, and
  - steps similar to the previous case takes place.



# Demand paging – illustration.

**Assumption: 1 process only.**

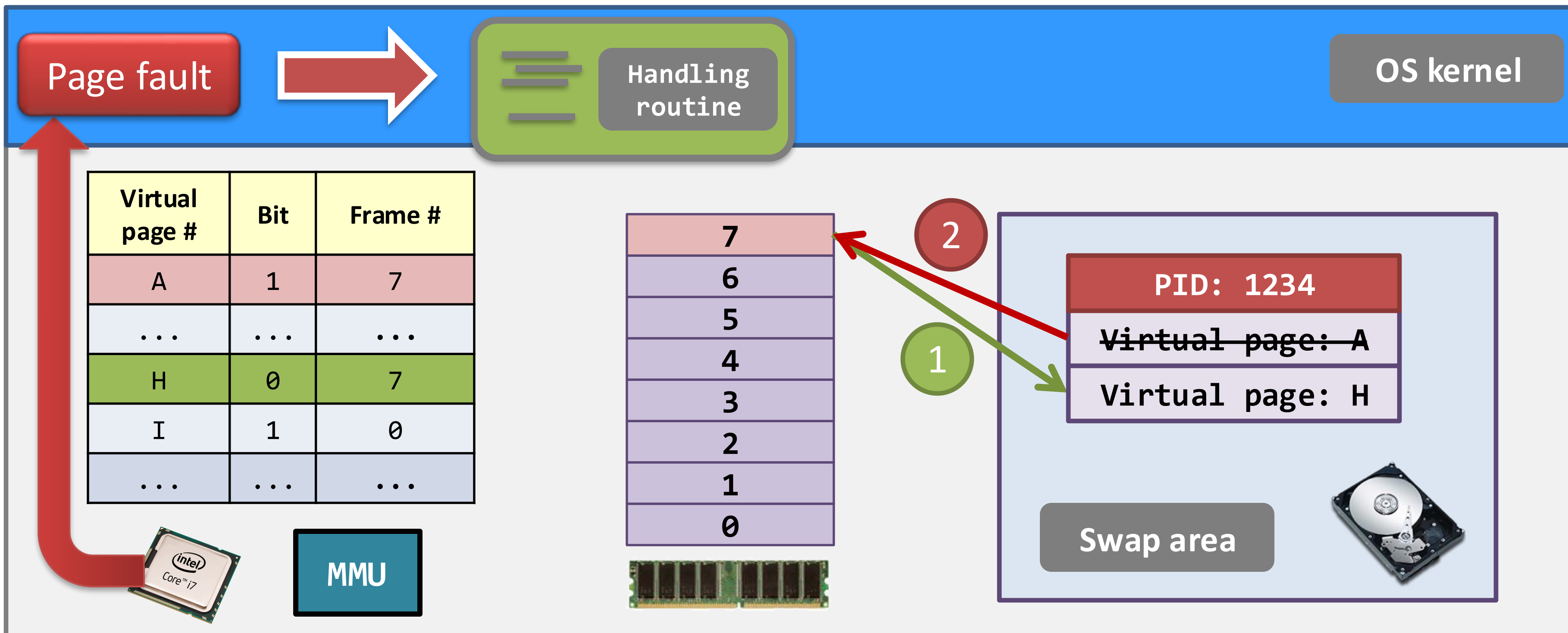
- Step (1) Select a victim virtual page and copy the victim to the swap area.
  - Now, <u>Frame 7 is a free frame</u> and <u>the bit for Page H is 0</u>.



# 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 |
| ... | ... | ... |

# Revisiting the Real OOM

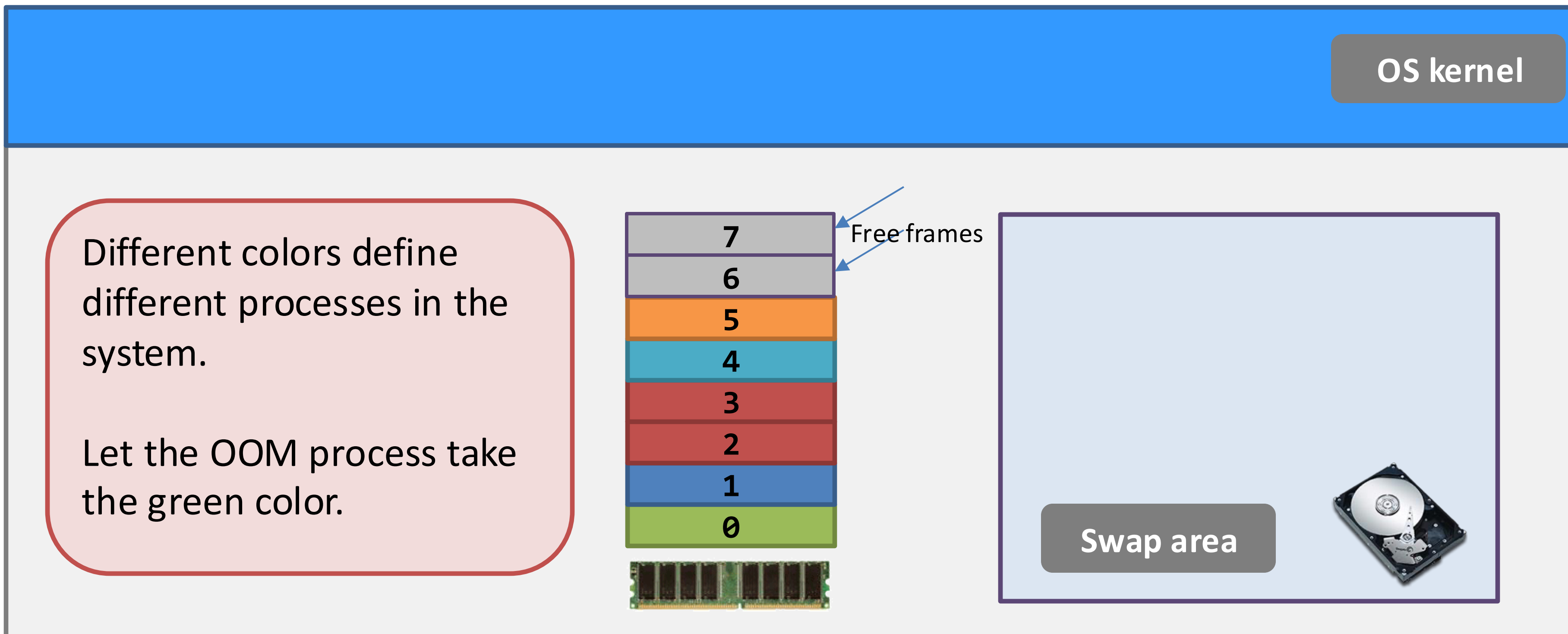
```
#define ONE_MEG  1024 * 1024

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.

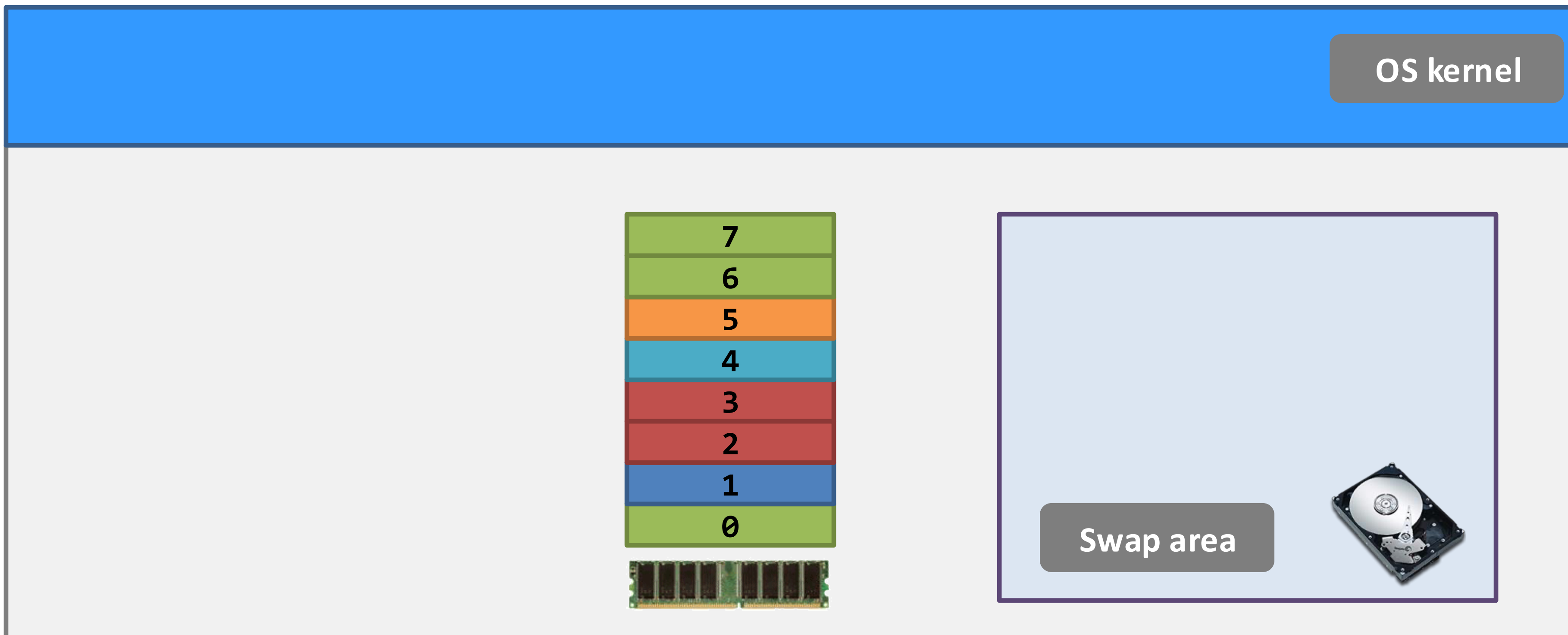
# Real OOM – illustration

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



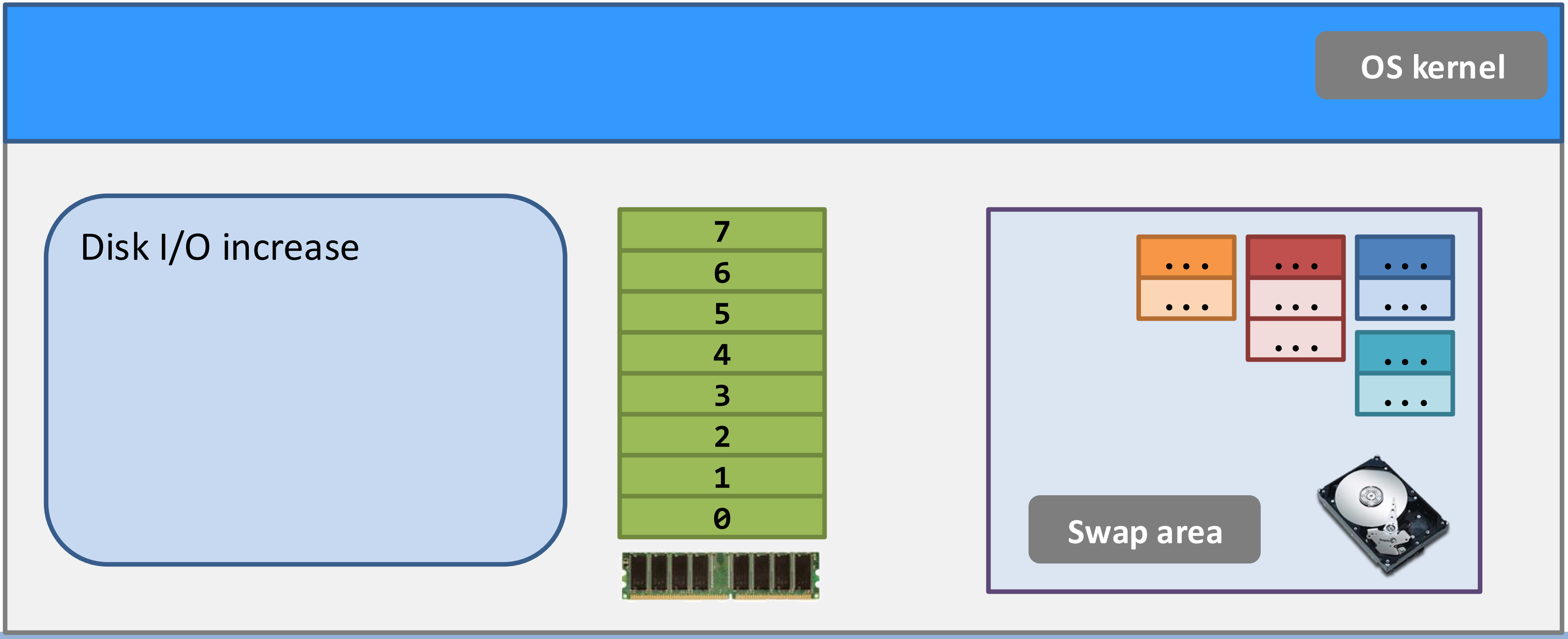
# Real OOM – illustration

- OOM is running…
  - **<u>1st stage</u>**. The **free memory frames** are the first "dim sum" of OOM
  - All other processes could hardly allocate pages.



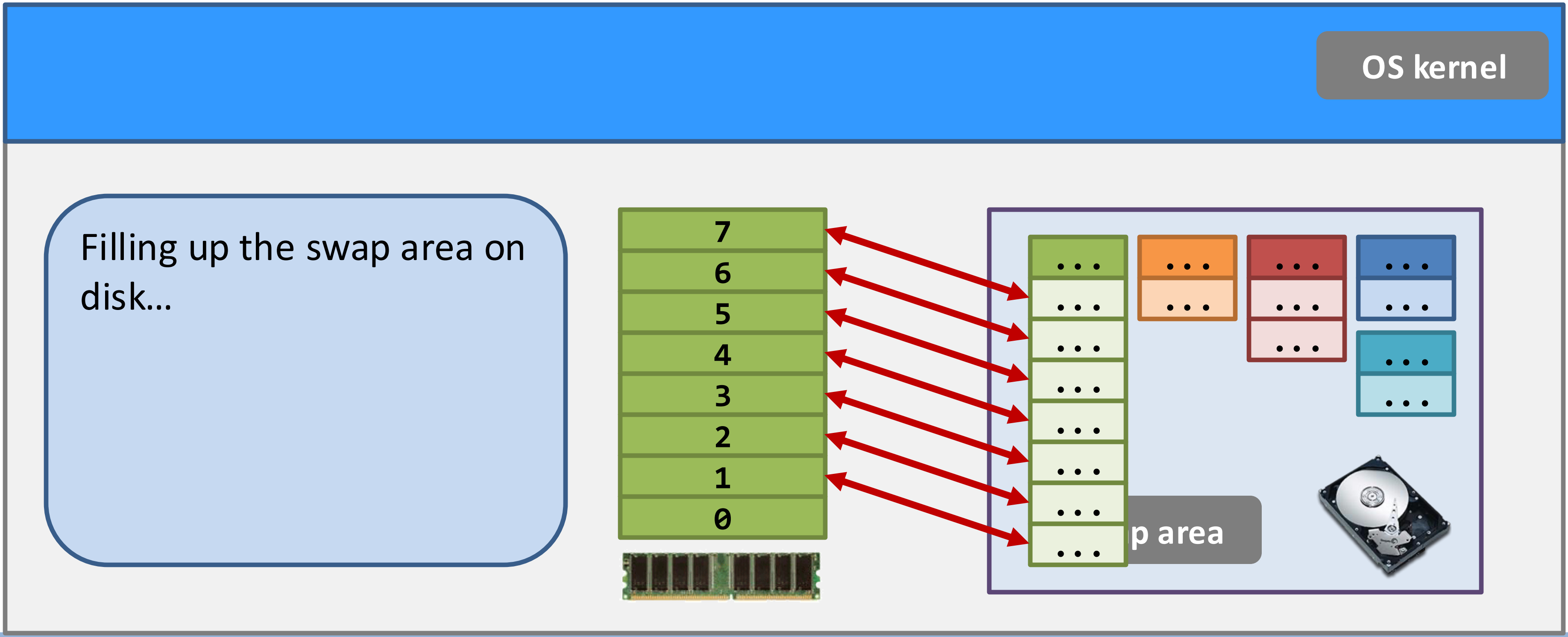
# Real OOM – illustration

- OOM is running…
  - **<u>2nd stage</u>**. Occupied memory frames are the next "dim sum" of OOM
  - **Swapping out others' frame to disk**



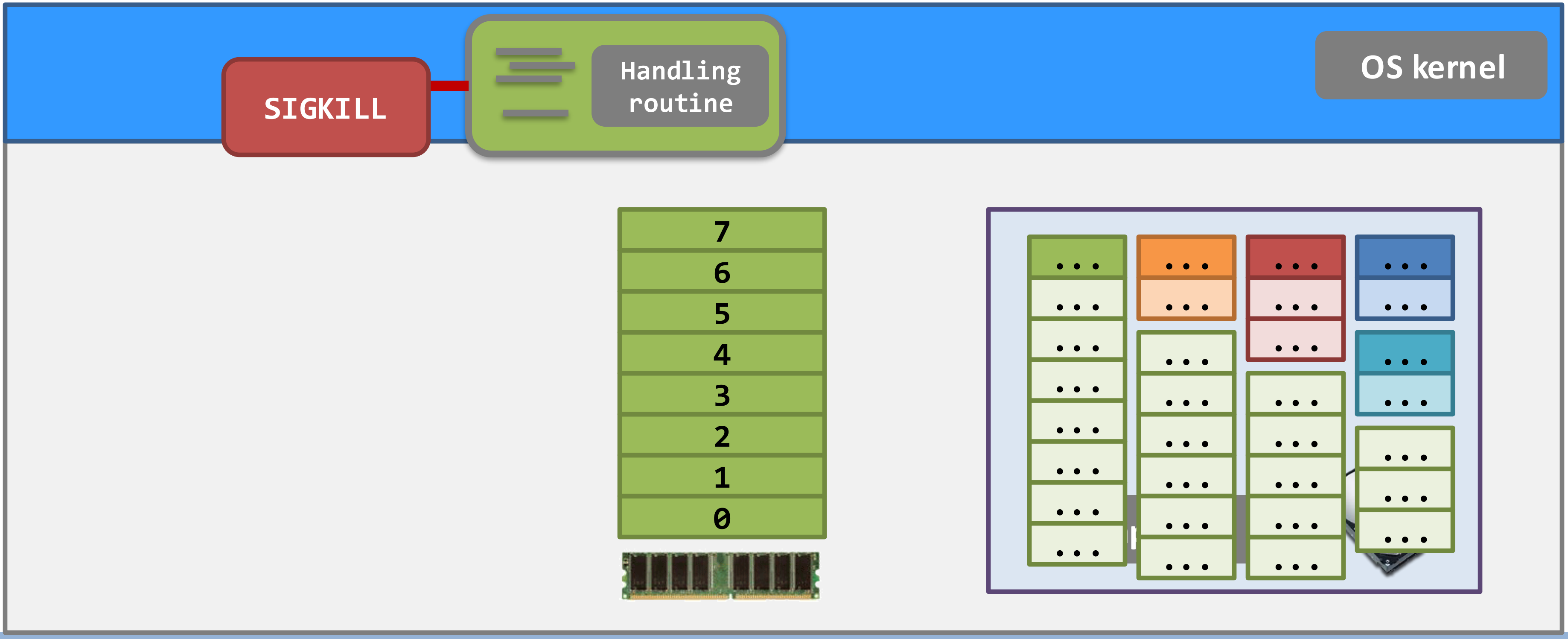
# Real OOM – illustration

- OOM is running… (more **new** frames are required)
  - **3rd stage**. Swapping out its own frames to disk
  - **Disk activity flies high!**



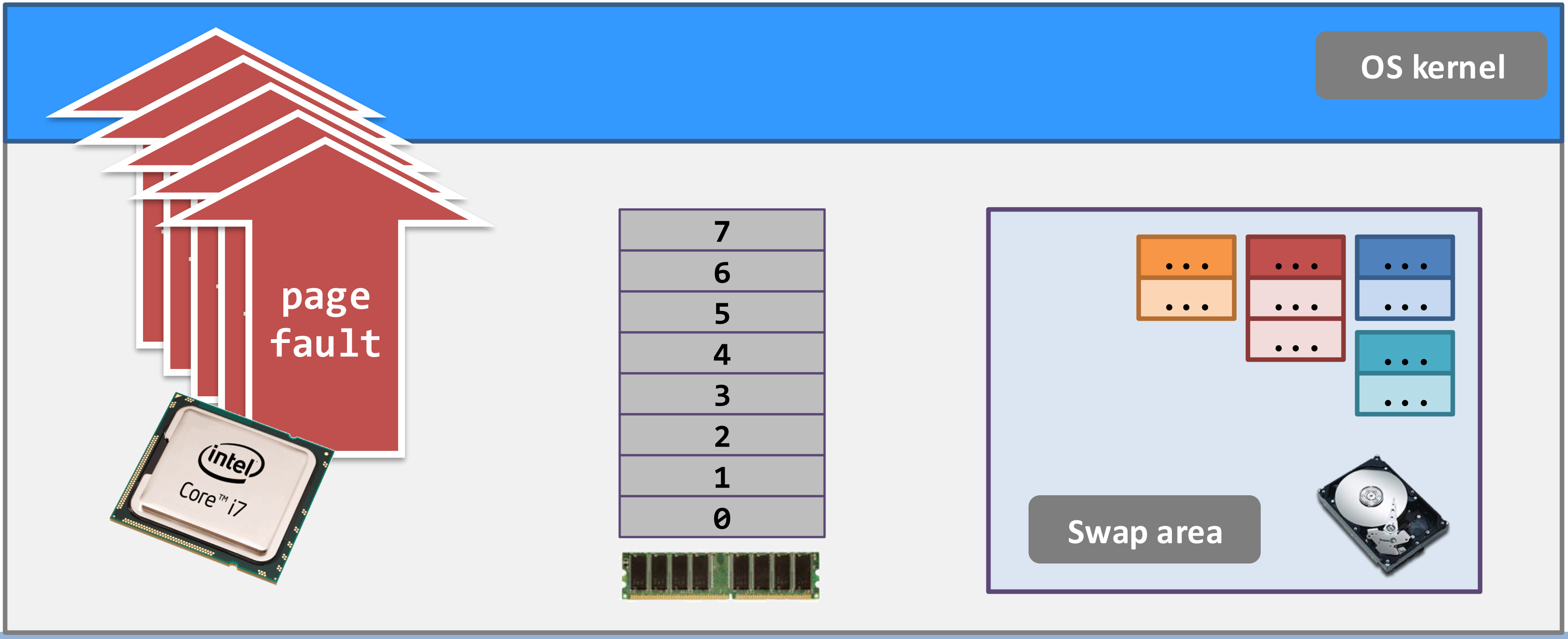
# Real OOM – illustration

- OOM is running…
  - **Final stage**. 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 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): _
```

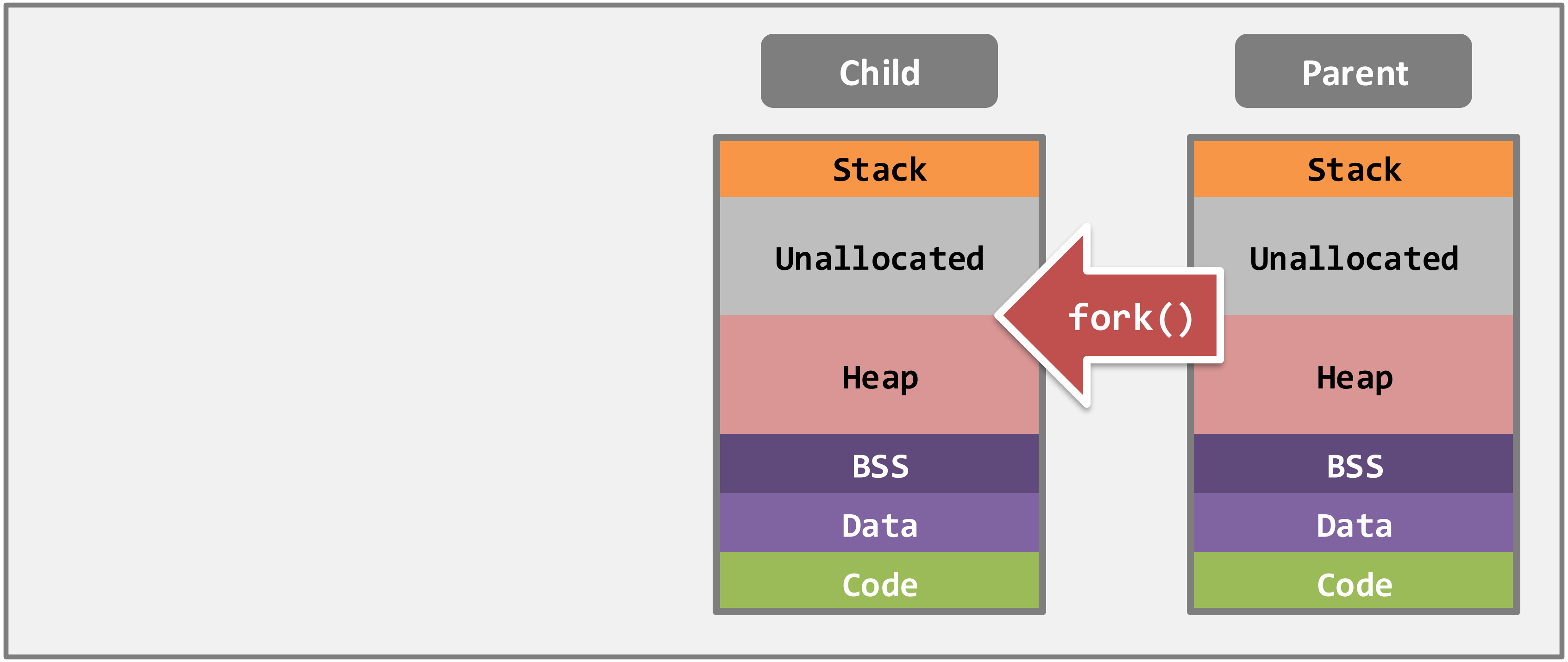

Windows hides a file "**pagefile.sys**", which is the swap area, in one of the drives.

# Demand Paging

- A disk page is copied to the memory only when
  - an attempt is made to access it and
  - that page is not already in the memory yet
    - i.e., a page fault
- So, a process begins execution with none of its page in the RAM
  - Incur many page faults
    - Until most of its **working set** is in the memory
- In other words, if every frame a program needed is not in the memory (e.g., because of an OOM is running)
  - → every frame a program needed causes a page fault
  - → every frame a program needed causes a disk access
  - We call this bad situation as **thrashing**

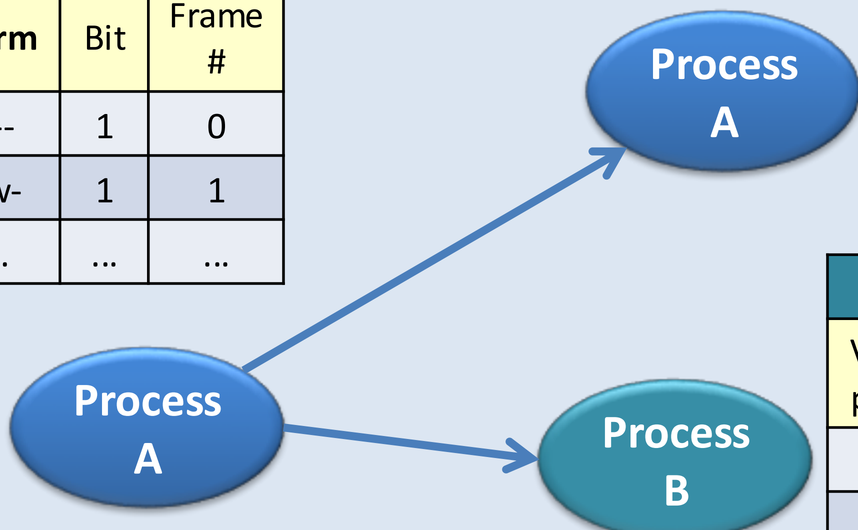
# fork() implementation

- The parent process and the child process **are identical** from the *userspace memory* point of view.
- But memory copying is heavyweight



# Copy-on-write (COW) technique.

– During **`fork()`**, copy the page table with special permission



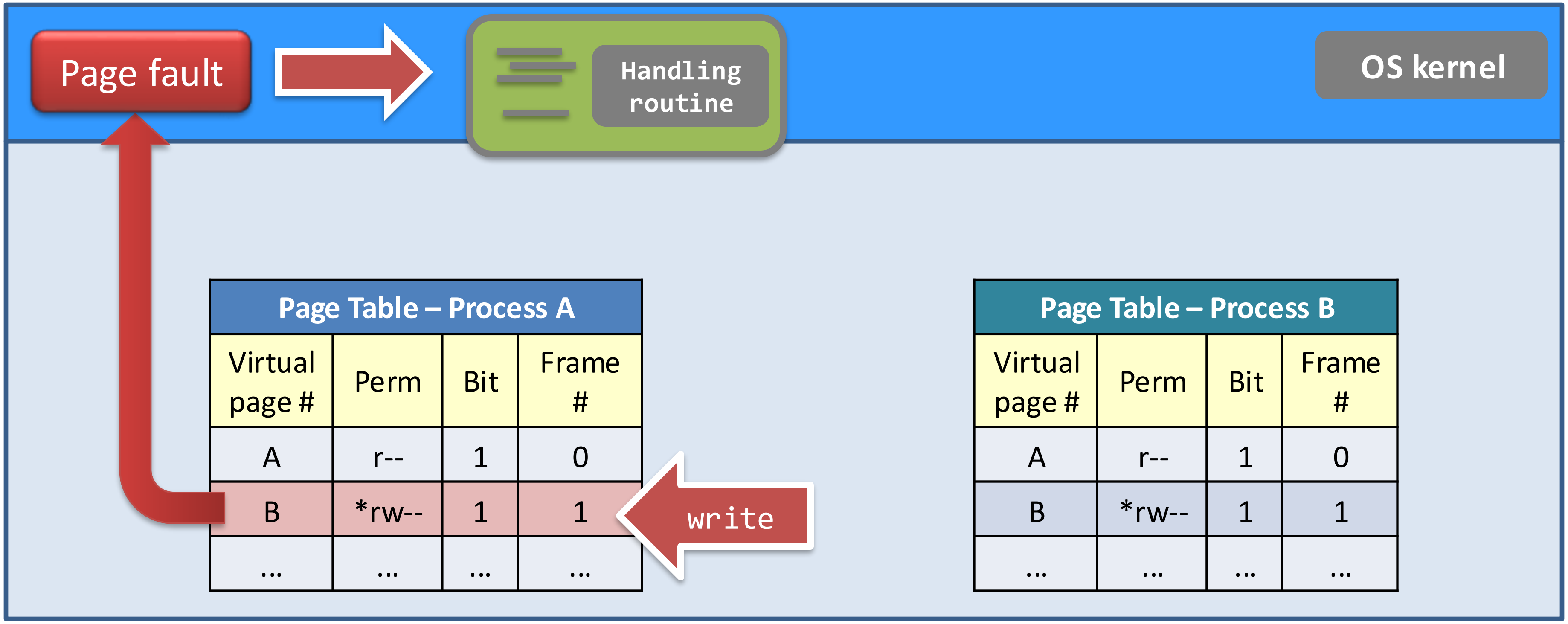

| Page Table – Process A | | | |
|---|---|---|---|
| Virtual page # | **Perm** | Bit | Frame # |
| A | r-- | 1 | 0 |
| B | rw- | 1 | 1 |
| … | … | … | … |

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

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

# Copy-on-write (COW) technique.

- When there is a write on a page, page fault is generated.

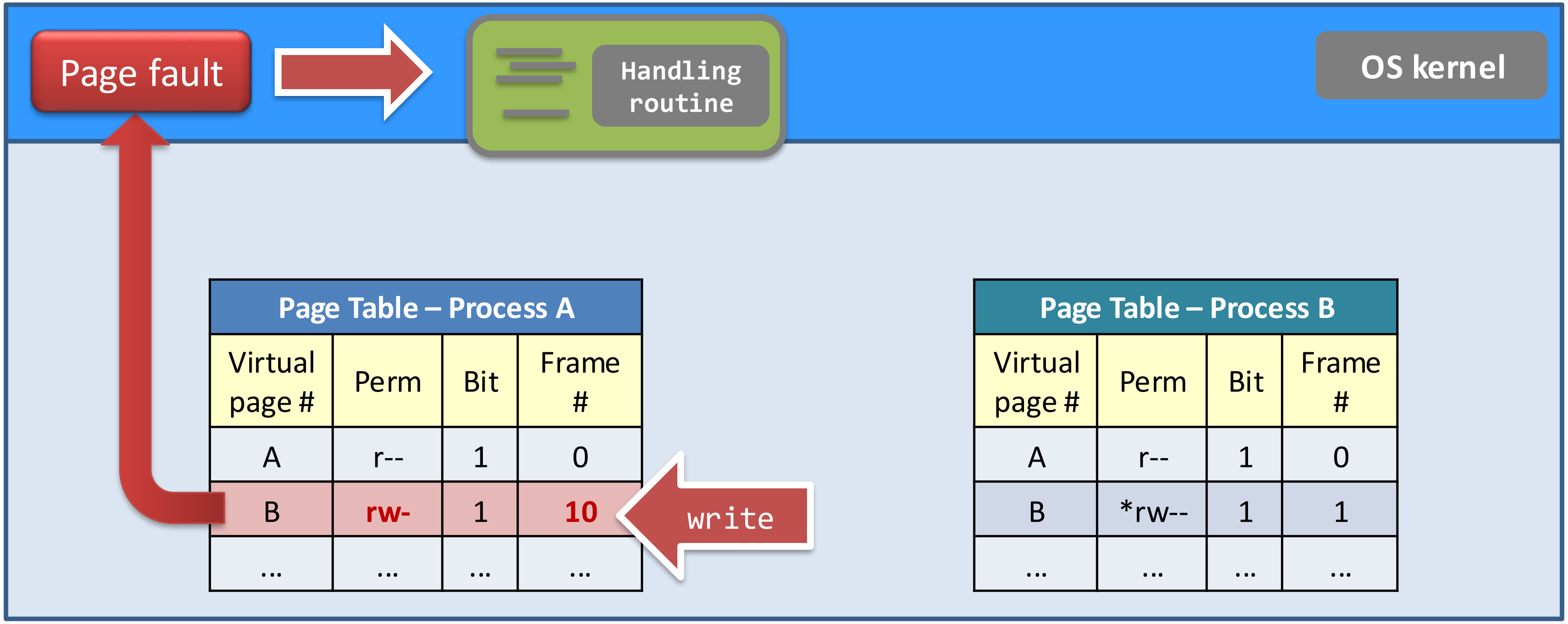



| Page Table – Process A | | | |
|---|---|---|---|
| Virtual page # | Perm | Bit | Frame # |
| A | r-- | 1 | 0 |
| B | *rw-- | 1 | 1 |
| ... | ... | ... | ... |

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

# Copy-on-write (COW) technique.

- When there is a write on a page, page fault is generated.
- The handler makes the real copy and writes on the real copy




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

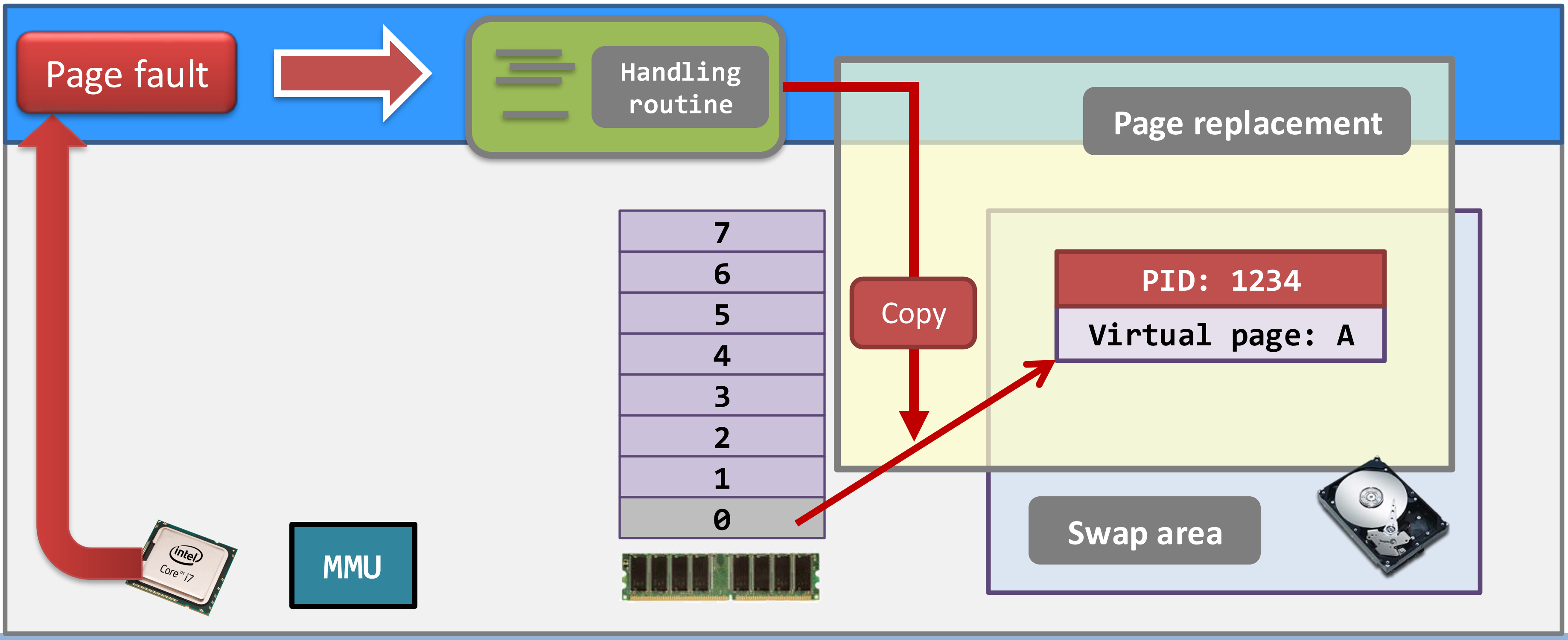
| Page Table – Process B | | | |
|---|---|---|---|
| Virtual page # | Perm | Bit | Frame # |
| A | r-- | 1 | 0 |
| B | *rw-- | 1 | 1 |
| … | … | … | … |

Memory Management

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


# Page replacement – introduction

- Remember the **<u>page replacement operation</u>**?
  - 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**!
  - Page replacement algorithms should **minimize further page faults**.

- Page replacement algorithms:
  - Optimal – if full reference string is given, e.g.,
    - 4 frames; Reference string = C B D E C A C B D C B D C B D
  - First-in first-out (FIFO);
  - Least recently used (LRU);
  - …

arriving

If the memory can house 4 frames only, it's now holding:
C B D E
Now, who shall be the victim if A needs to kick one out?

# Page replacement – LRU algorithm

- **Least-Recently-Used**

| | **Implementation 1** | **Implementation 2** | **…** |
| --- | --- | --- | --- |
| Main Idea | Every page frame has an age counter | Doubly linked list | … |
| When a page frame f is referenced | f.age = 0;<br>Otherframe.age++; | Move f to the list's head | … |
| When finding a victim | Full scan $\Theta(n)$ | Return list's tail | … |
| Storage: | n integers (where n=# of frames) | Storage: 2 pointers per frame | … |

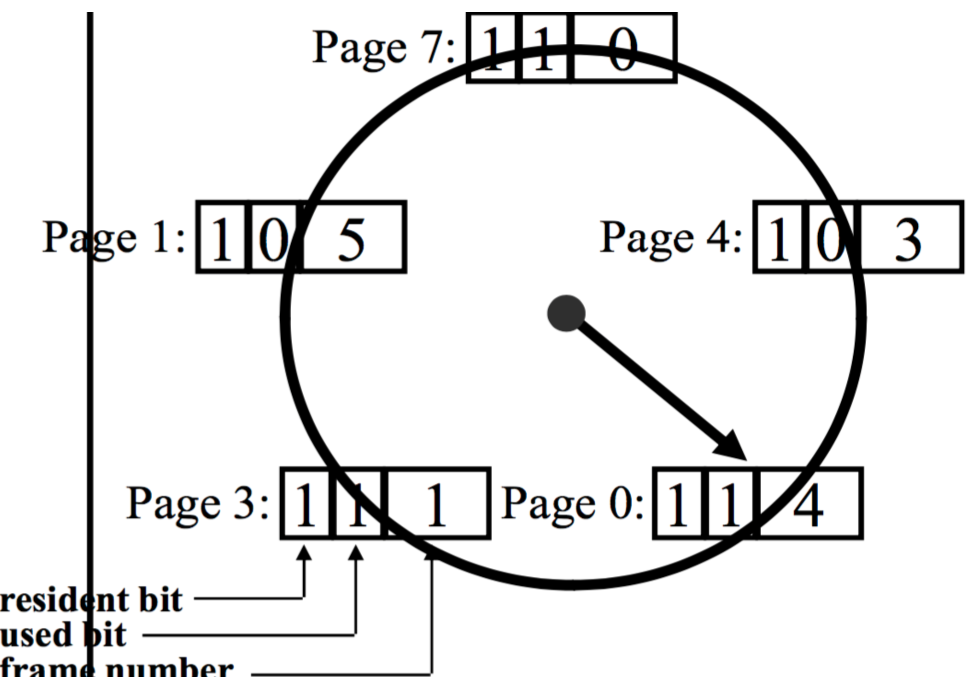
# Cache replacement policies

- https://en.wikipedia.org/wiki/Cache_replacement_policies
- What are the goals of a page replacement algorithm?
   A. Reduce the # of pages faults
   B. Minimize the overhead of the algorithm
   C. All of the above

# Clock Algorithm

- LRU is good but difficult to implement it efficiently in practice
- LRU is approximate (not optimal) anyway
- So, why not come up with an approximate algorithm that can be efficiently implemented?

| | Clock |
|---|---|
| Main Idea | Circular list<br>Each page gets a used **bit** (vs. age **integer** counter) |
| When a page frame p is referenced | p.used = 1; //means referenced |
| When finding a victim | Clock hand sweeps over pages and returns the first one with used = 0; //means that is a page that hasn't been referenced for 1 complete revolution of the clock; reset those used' bits to 0 |
| Analysis | Storage: n bits (where n = # of frames)<br>Finding Victim: O(n) |



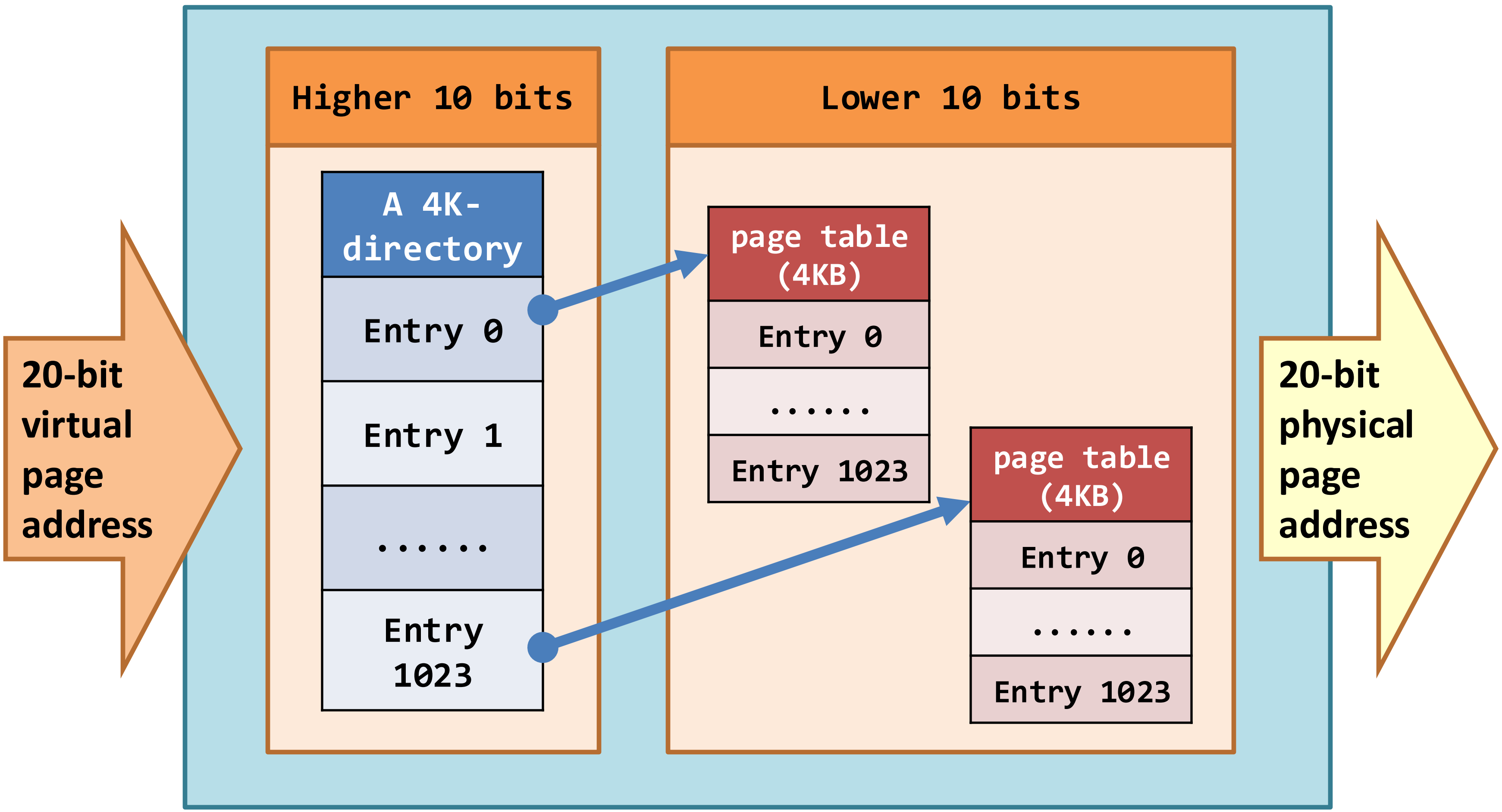

```
func Clock_Replacement
begin
  while (victim page not found) do
    if (used bit for current page = 0) then
      replace current page
    else
      reset used bit
    end if
    advance clock pointer
  end while
end Clock_Replacement
```

Ref: E. Witchel UT Austin CS 372

# Paging for Real

- The page table **is still 4MB large**!
  - 20-bit page addresses
  - Each page table entry is 4 bytes
  - $2^{20} * 4 = 4MB$
- Solution?
  - Multi-level page table with unused pages not stored.
    - It is the state-of-the-art design of a page table.

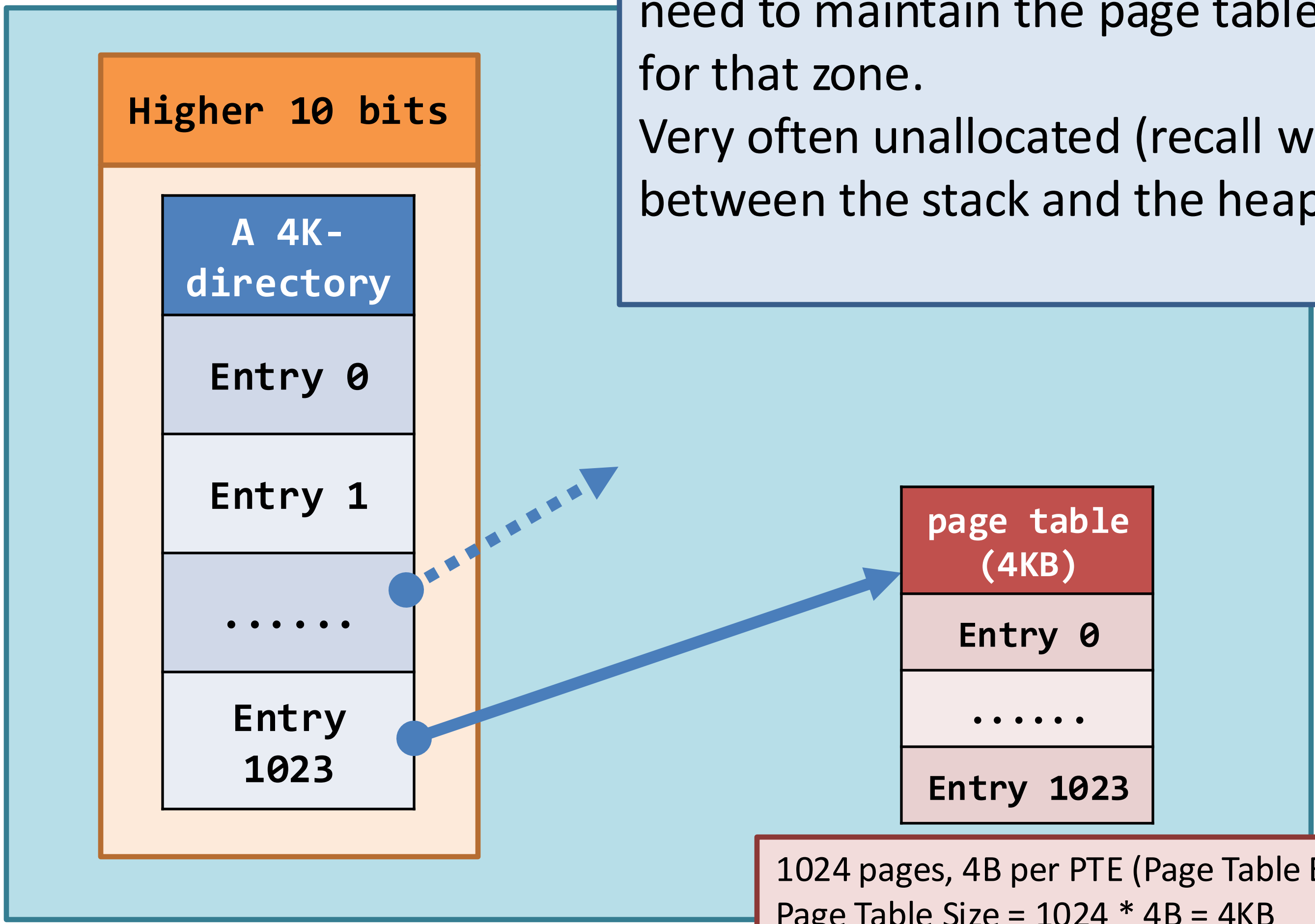
# Paging – multi-level page tables



# Paging – multi-level page tables

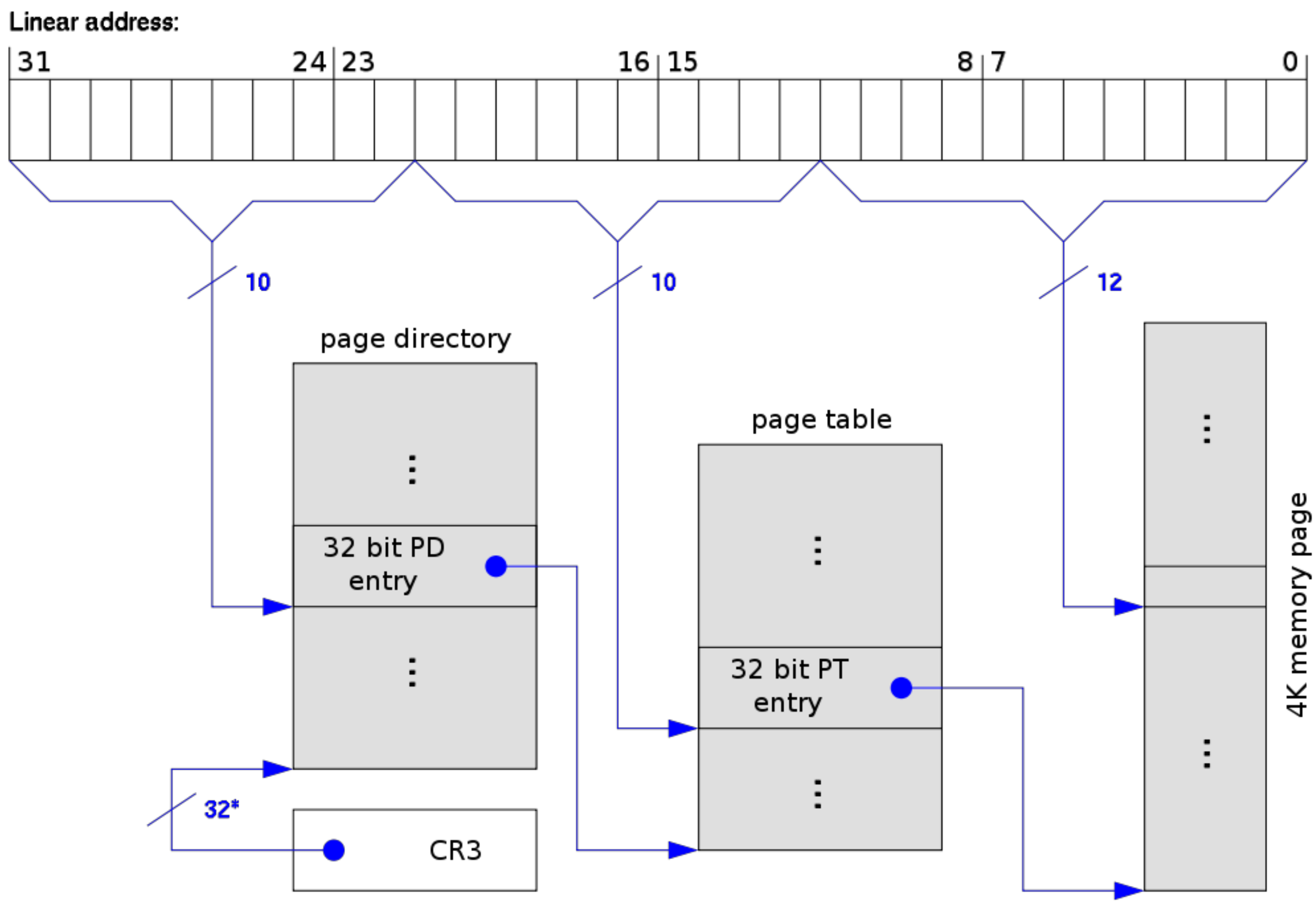
- The merit:

If a zone is not allocated, we don't need to maintain the page table entries for that zone.
Very often unallocated (recall what is between the stack and the heap?)




1024 pages, 4B per PTE (Page Table Entry)
Page Table Size = 1024 * 4B = 4KB

# Paging – multi-level page tables

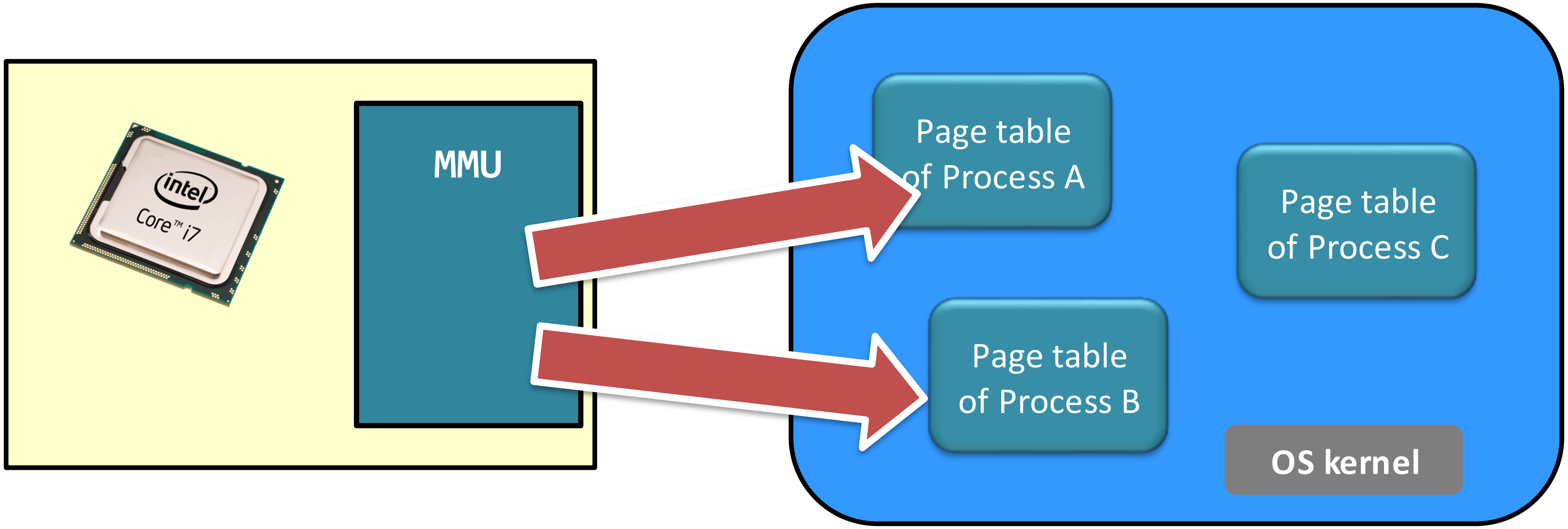



CR3 is a register that contains the physical address of the top-level of the page directory in the memory

So, technically, when **context switch** from P1 to P2, it is saving P1's register values (e.g., CR3) to a temporary place, and then popping the CR3 value of P2 from a kernel stack, and then push temp registers to the kernel stack

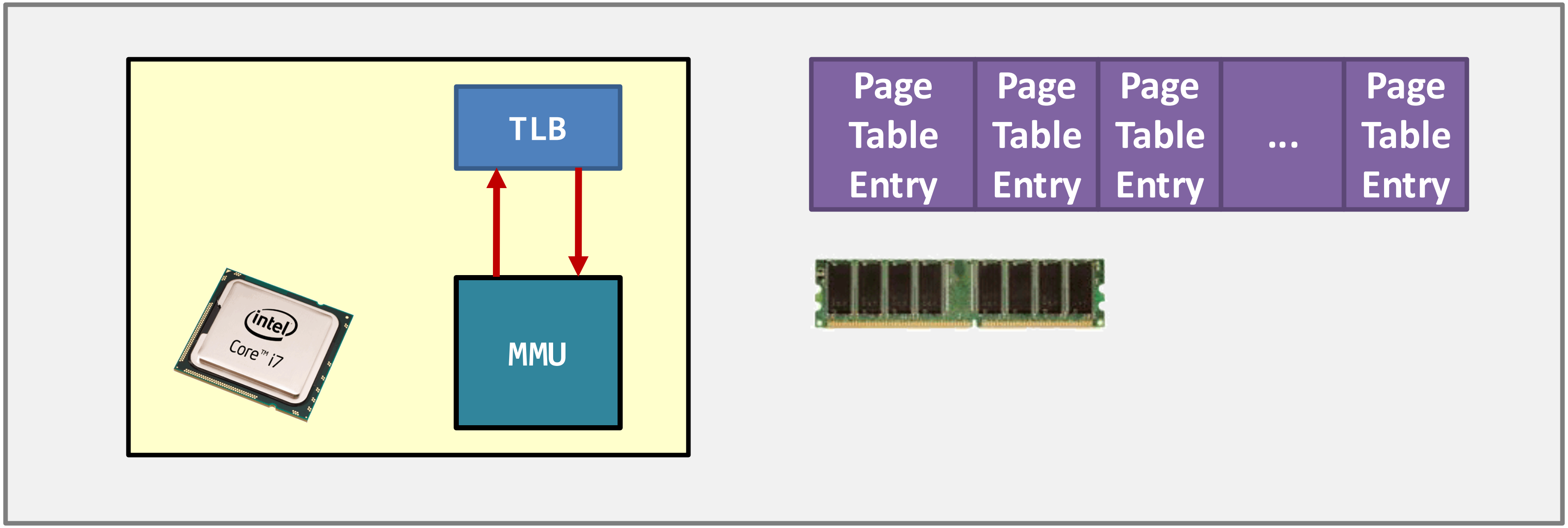
# Paging – Hardware Support

- Remember, what is context switching?
  - The page table is also switched during the context switching procedure.



# Paging – Translation Lookaside Buffer

- Cache recent translation of virtual page to physical frame
  - Sometimes implemented using content-addressable memory
    - Very expensive (So, only cache several hundreds of entries)
    - Very fast for content-based searching (not address-based searching)
    - In TLB, the mapping is like <virtual page number, page frame>
    - Why keep the virtual page number here this time?
      - » Because TLB has limited entries, can’t use implicit addressing.




- Some architecture cleans the TLB on process switches;
- Some keep the mapping as <processid+virtual-page-number, page frame>