Memory Management

Alex Pajuelo and Juan Jose Costa

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

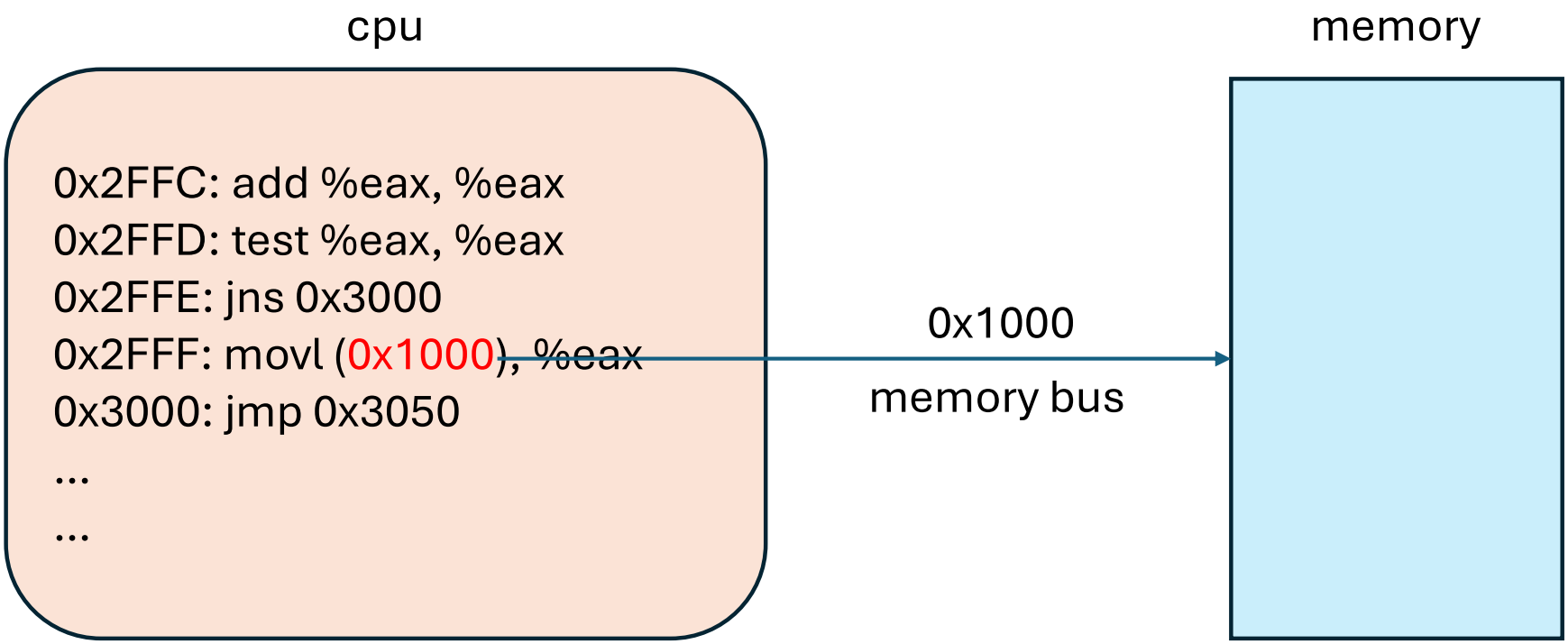
Memory management

- The memory is a critical resource to ensure performance
- Even if there is a lot of memory available it is not unlimited
- Wrong memory management could lead to slowdowns and performance degradation
- The OS is responsible to distribute memory among processes

Architecture fundamentals

First memory system

- The first memory system was way too simple due to hardware limitations
- Code was compiled setting the starting physical address and, in addition, all addresses of code and data were generated as physical



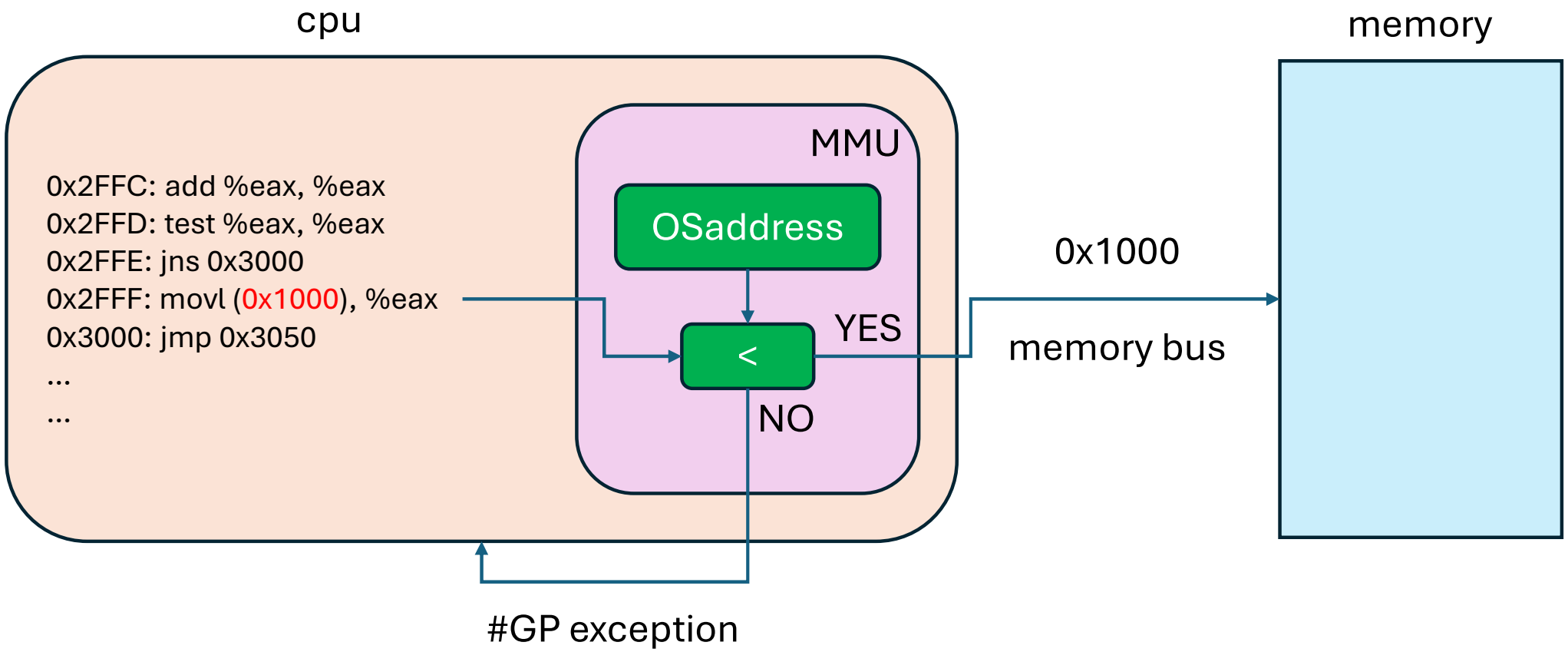
First memory system Pros and Cons

- Pros:
  - Hardware is simple
  - No memory management by the OS
- Cons:
  - User code can access OS data
- Improvement: Memory Management Unit (MMU)

Memory management unit

- New component of the processor that monitors every access before they are sent to memory
- In this case the MMU checks if a user instruction tries to access the OS memory
- If it does, raise a #GP exception
- For that, the MMU incorporates a register (OSaddress) to hold the first physical address of the OS and a comparator
- OSaddress is set up during OS boot

First memory system + MMU



If (0x1000 < OSaddress) send address 0x1000 through the memory bus
If not, raise a #GP exception

First memory system + MMU Pros and Cons

- Pros:
  - Easy hardware implementation
  - Security
- Cons:
  - Only one process can be executed
  - Since code is compiled with physical addresses, there is high probability of overlapping when two or more processes are allocated in memory
- Improvement: logical addresses

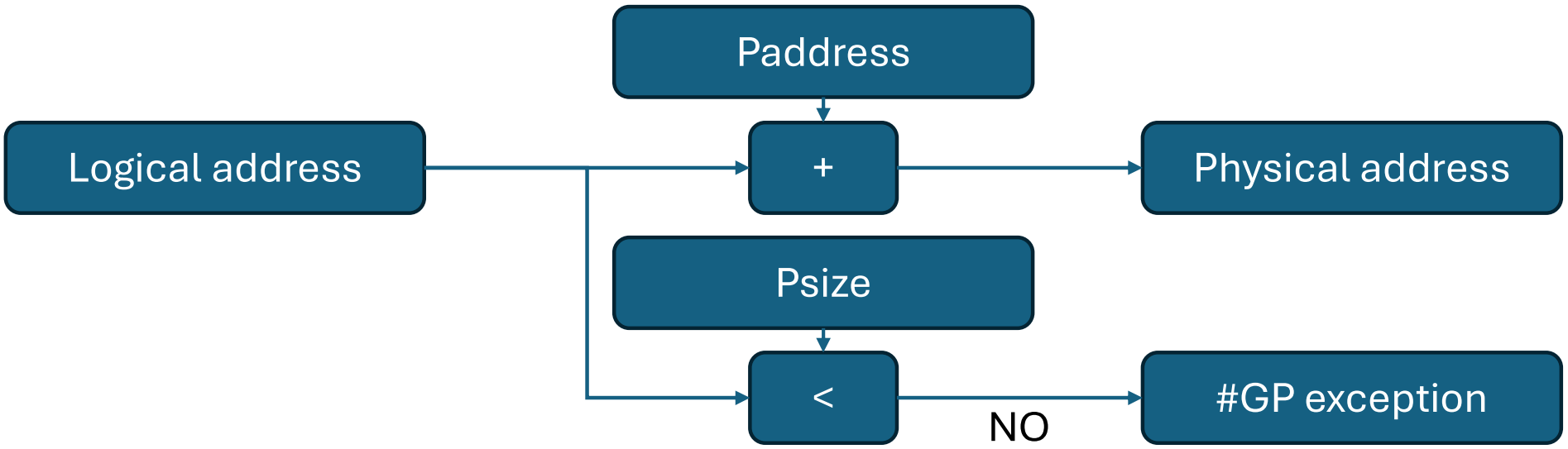
Second memory system: logical addresses

- When the code is compiled, all code and data addresses are relative to 0
  - They can be seen as offsets from the beginning of the executable
  - This is why they are called logical
- When the OS loads the program, the first memory address used is stored (Paddress)
- A translation is needed for every logical address (offset) to finally access the physical memory:

Physical address = Paddress + offset

Second memory system MMU

- The MMU now has 2 registers, and adder and a comparator:
  - Register Paddress: stores the first physical address of the process in memory (from where the OS has load the program)
  - Register Psize: amount of memory the process is using
- If logical address >= Psize raise #GP exception



Second memory system MMU and multiple processes

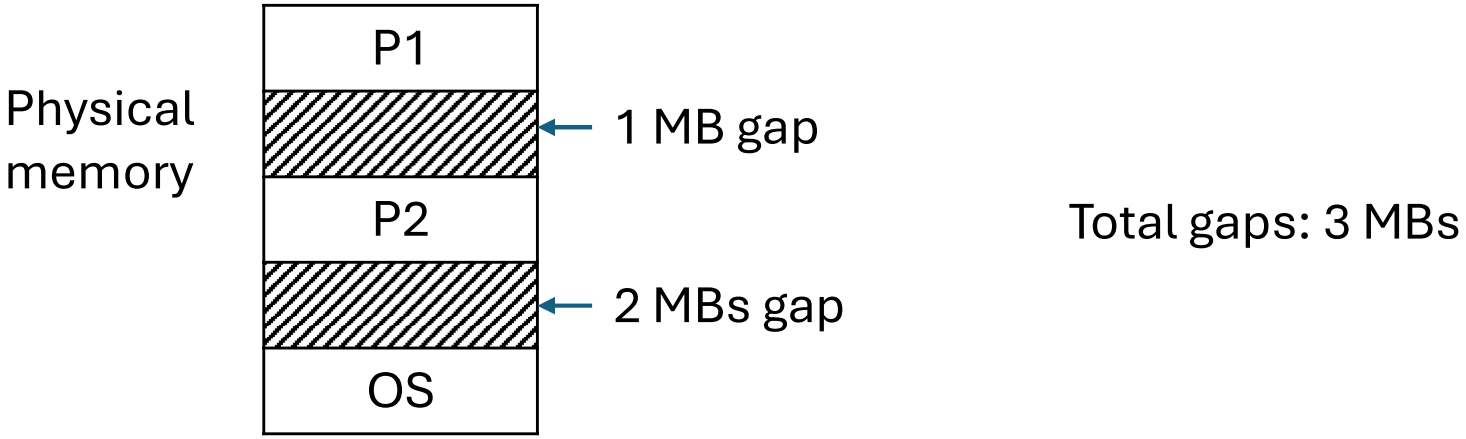
- Every process has its own couple of values for the registers Paddress and Psize stored in the PCB
- When the OS changes the process that is currently executing, it changes the contents of the Paddress and Psize MMU registers with the values in the PCB
- We will see later when and how

Second memory system Pros and Cons

- Pros:
  - Easy to implement
  - Few modifications of the OS
  - Address translation is fast
- Cons:
  - Fragmentation

Fragmentation

- There exist gaps in memory that cannot be used
- Example: having 2 processes in memory (P1 and P2)

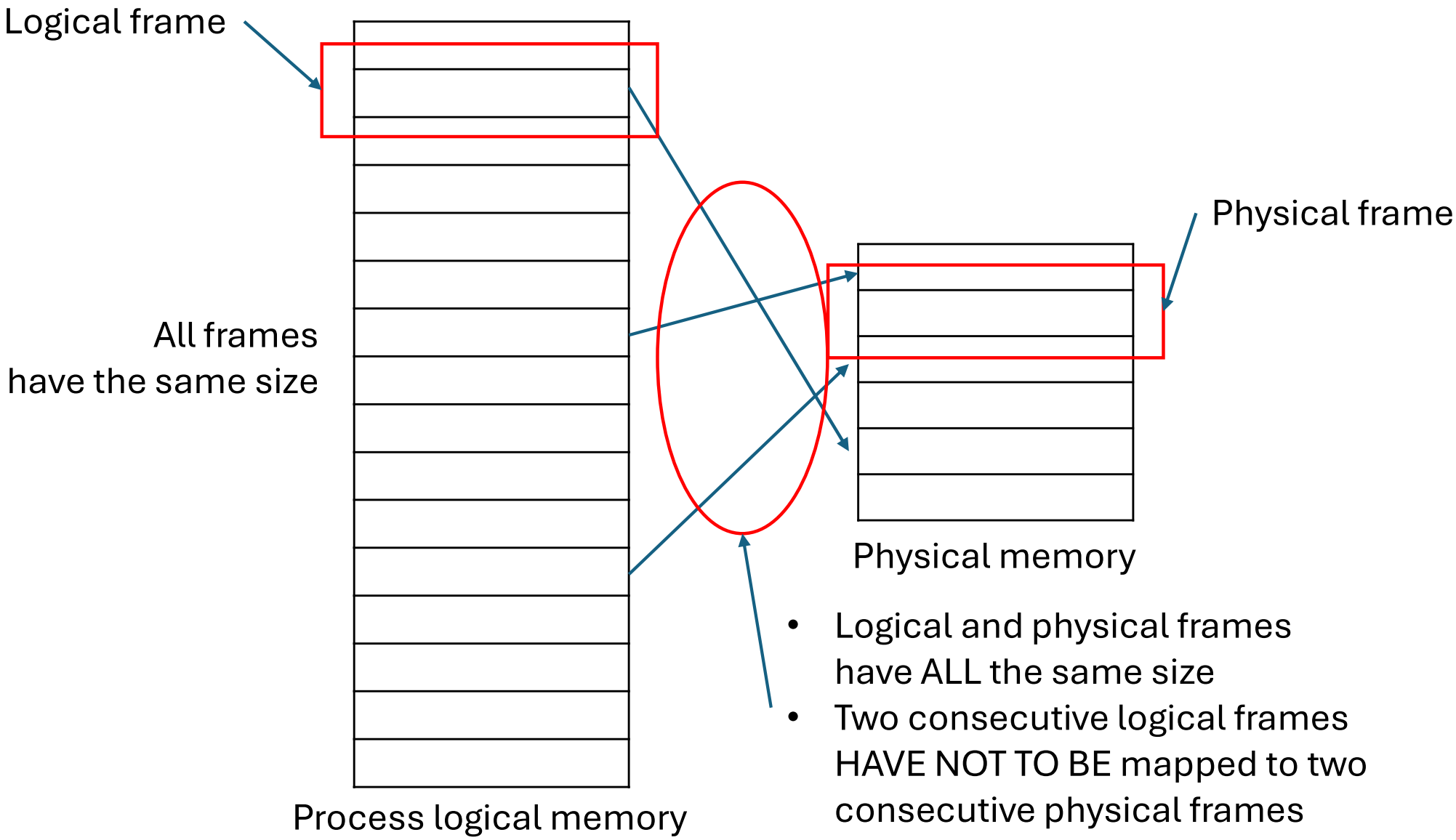


- If you want to execute a 3MBs process, you cannot because a process needs to be consecutive in memory
- Solution: Pagination

Pagination

- Now, processes do not have to be consecutive in physical memory
- Trick: split both the process memory and the physical memory in ranges of addresses. All those ranges have the same size
- When talking about process memory, one of those ranges is called Logical Frame or Logical Page
- When talking about physical memory, one of those ranges is called Physical Frame, Physical Page or just Frame
- Pagination maps one Logical Frame to one Physical Frame
- Two consecutive logical frames have not to be necessarily mapped to two consecutive physical frames

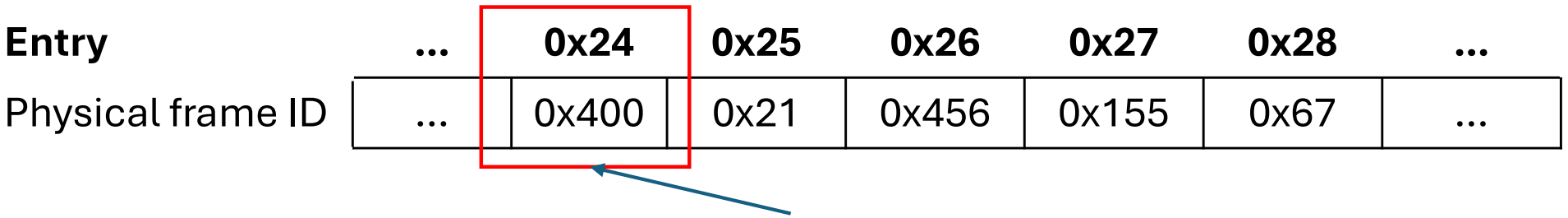
Pagination scheme



- Logical and physical frames have ALL the same size
- Two consecutive logical frames HAVE NOT TO BE mapped to two consecutive physical frames

Page table

- The maps between logical and physical are stored in the Page Table (PT)
- Every process has its own PT, and two or more processes do not share the same PT
- The page table is an array in memory with as many entries as logical frames a process can have and, in every entry, among others, it stores the frame ID of the physical frame that logical page is mapped to



Logical frame 0x24 is mapped to physical frame 0x400

Page table (and II)

- In a x86, with address of 32 bits and a page size of 4KBs, the number of entries in the PT is:

$$\frac{2^{32}\ total\ logical\ addresses}{4KBs\ per\ page} = 2^{20}\ entries$$

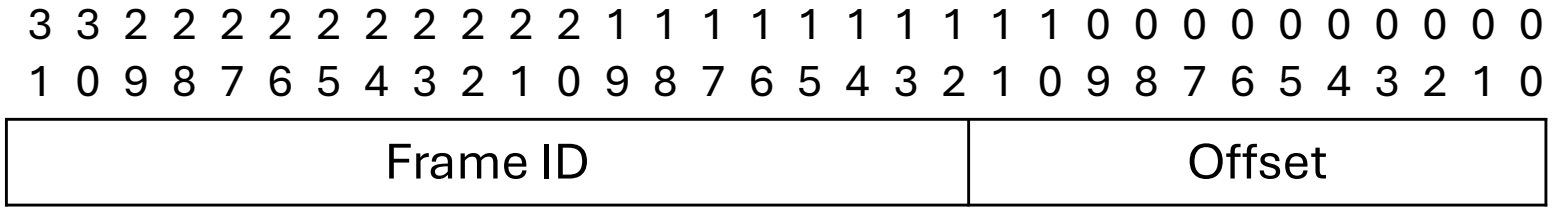
- A pointer to the PT is stored in the PCB of the process

Frame IDs

- Most of the time the OS does not work with a frame but with a frame ID
- A frame ID is a 20-bit number
- Two logical frames cannot have the same ID
- Two physical frames cannot have the same ID
- Remember that we are working in x86 in which addresses are 32-bit long

Memory addresses and frame IDs

- From now on, we set the frame size to be 4KBs (4096 bytes)
- A 32-bit memory address is composed by a frame ID plus and offset inside that frame
- Since a frame is 4KBs, 12 bits are used as offset inside the frame



Memory address

From memory addresses to frame IDs

- To calculate the frame a memory address belongs to, the offset must be removed from the address

```
Int frameID = memory_address >> 12;
```

- Remember that the 20 most significant bits are the frame ID!!!

From frame IDs to memory addresses

- To calculate the range of addresses that belongs to a frame given its frame ID, shift left the frame ID and add 0 to 0xfff for the whole range

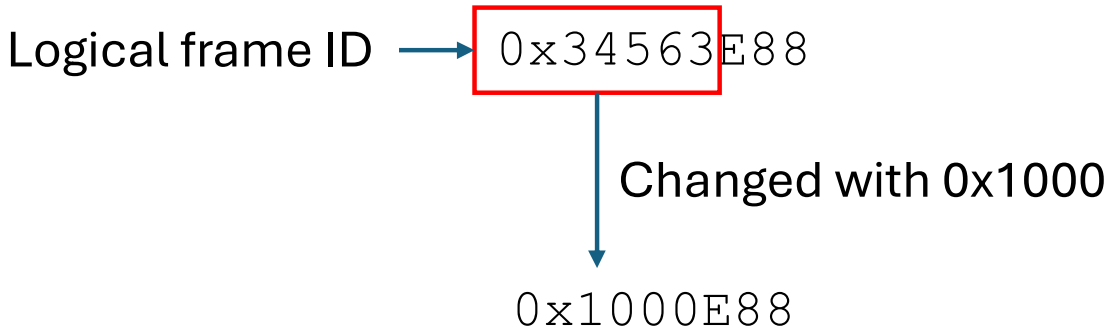
```
Start_address = frameID << 12;

End_address = frameID << 12 + 0xFFF;
```

- Remember that in an address, the 20 most significant bits are the frame ID

Logical to physical address translation

- The translation from logical address to a physical address is performed by changing the logical frame ID of the logical address by the physical frame ID the logical frame is mapped to
- Example: translate to physical address the logical address 0x34563E88 knowing the logical page 0x34563 is mapped to physical frame 0x1000



Address translation with pagination

- Address translation must be performed by the processor for every logical address
- Accessing the PT to find the physical frame ID per every memory access introduces overhead
- In addition, the PT size is 2^20 entries x 32 bits per entry = 4MBs!!!
- Problem: The PT is way too big to be stored in the CPU and its access introduces overhead
- Solution: Translation Lookaside Buffer

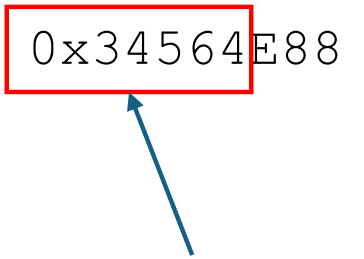
Translation Lookaside Buffer (TLB)

- TLB is a processor structure that stores a subset of entries of the PT of the process that is currently executing
- It is used by the CPU to find the physical frame ID a logical frame ID is mapped to, for every memory address
- In every entry of the TLB, among others, a couple of <logical frame ID, physical frame ID> is stored

Accessing the TLB

- The TLB is accessed for every memory address. For that, the CPU first obtains the logical frame ID of a logical address

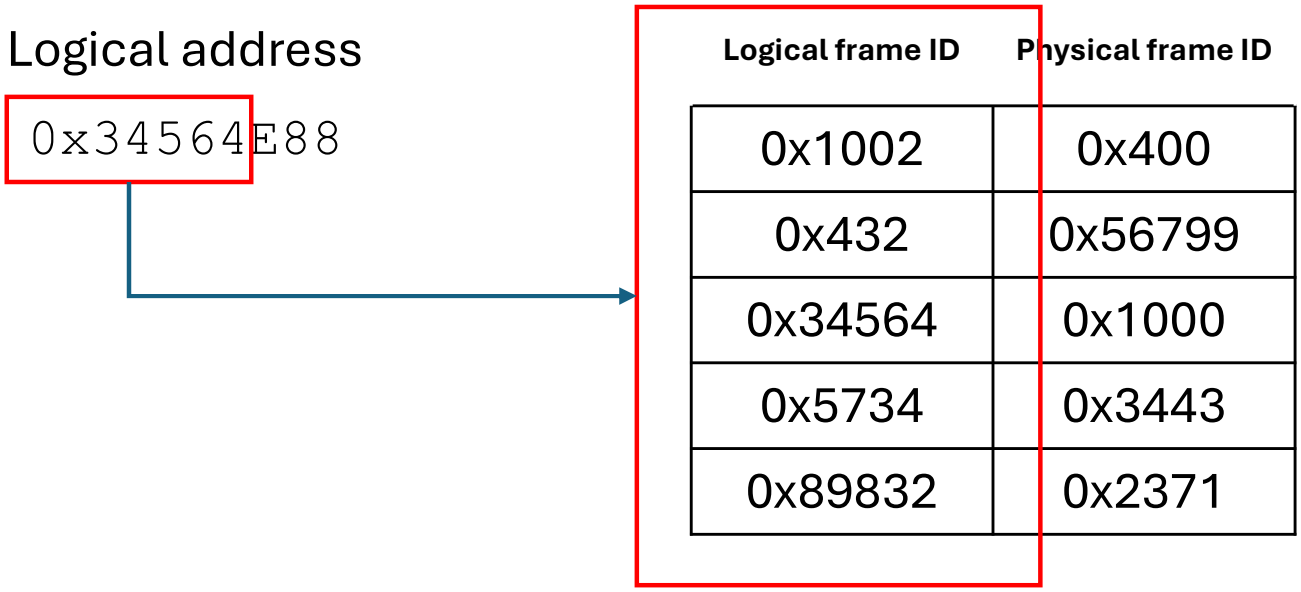
Logical address



Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

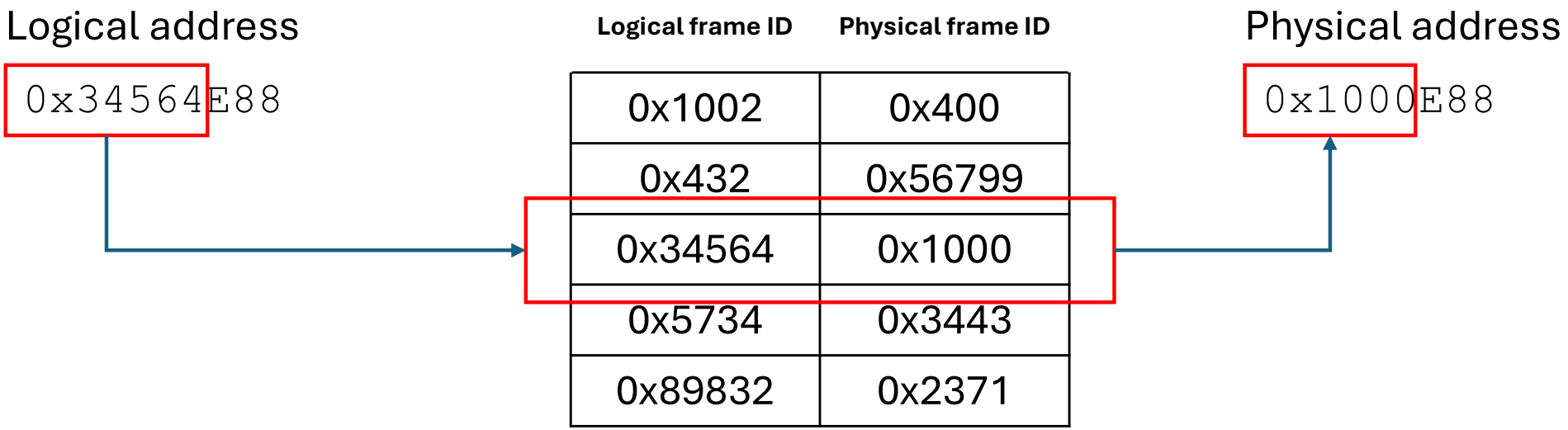
Accessing the TLB

- The frame ID is compared to all logical frame IDs in the TLB in parallel



Accessing the TLB

- If found, the corresponding physical frame ID is used to create the physical memory address



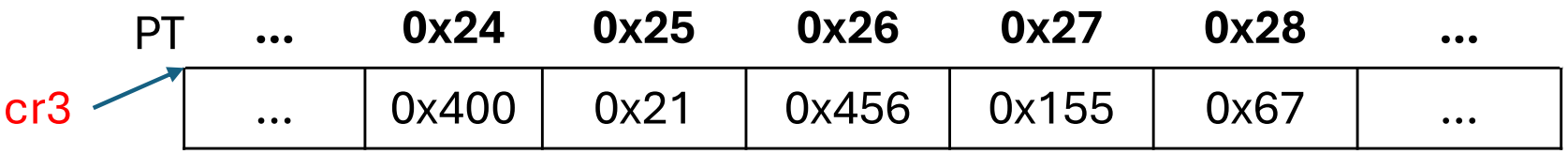
This is a TLB HIT

Loading the TLB

- If the logical frame ID is found in the TLB, it is a TLB Hit
- Remember that the TLB holds a subset of the entries of the PT, so, not all logical frames ID are at the TLB
- If the logical frame ID is not found in the TLB, it is a TLB miss
- When a TLB miss happens, the CPU accesses the PT to find the map of the logical frame ID and stores this logical frame ID together with the corresponding physical frame ID in an entry of the TLB
- Usually, a pseudo LRU algorithm is used to evict entries of the TLB

TLB miss

- The register cr3 is a CPU register that holds the physical address of the PT of the current process



Logical address

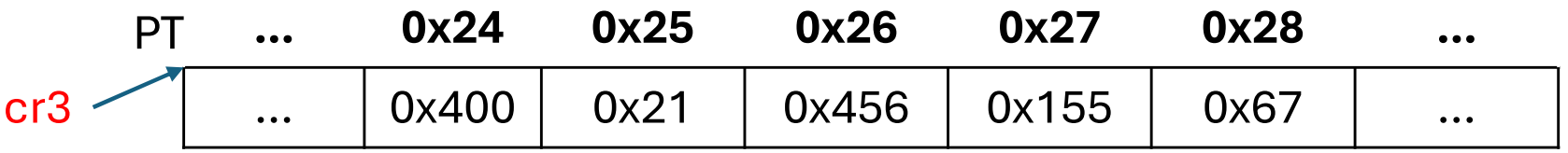
0x00025E88

Logical frame ID

Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

TLB miss

- As before, the logical frame ID is checked against the TLB



Logical address

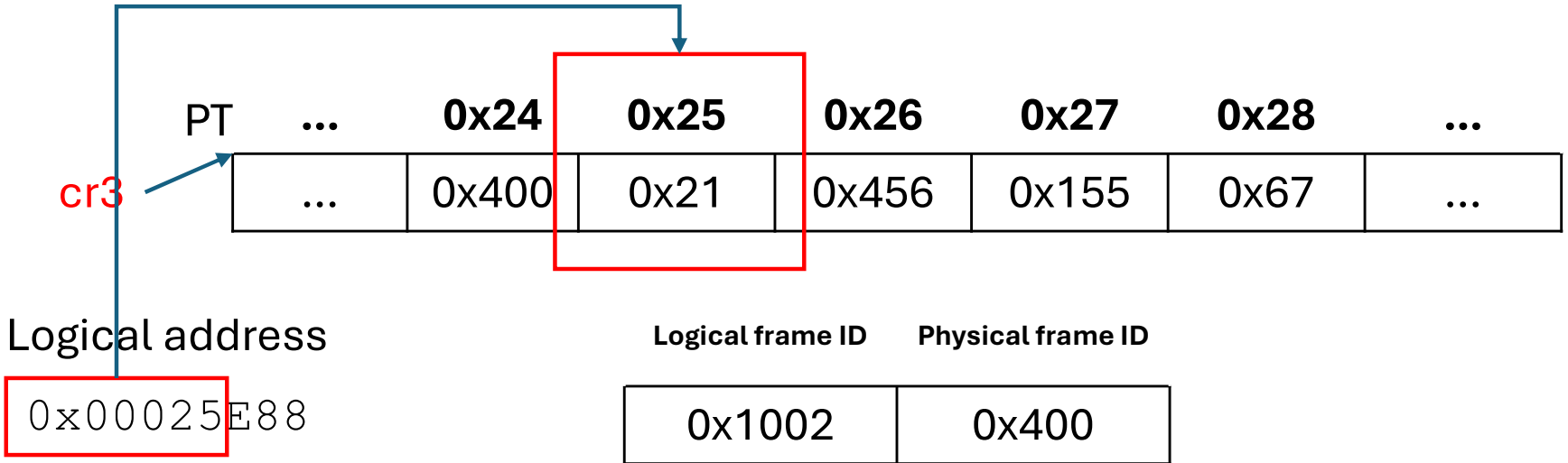
0x00025E88

Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

Not found!

TLB miss

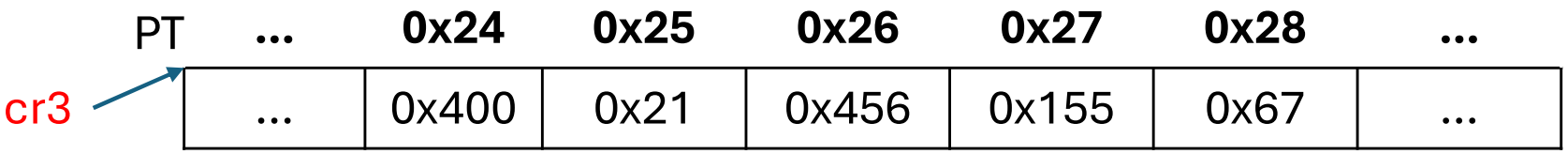
- If the logical frame ID is not found in the TLB, the CPU accesses via cr3 the corresponding entry of the PT



Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

TLB miss

- An entry (victim) of the TLB is substituted with the map and creates the physical address as before



Logical address

0x00025E88

Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x25	0x21
0x89832	0x2371

Physical address

0x00021E88

CR3 register

- The CR3 register is a special CPU registers that holds the physical address of the PT of the current process
- We will see later how the CR3 register is updated
- Since the PT structure is defined by the processor, the size of an entry is always 4 bytes.
- So, the access of an entry of the PT through the CR3 register is performed as:

Physical frame ID = *cr3 + logical frame ID * 4

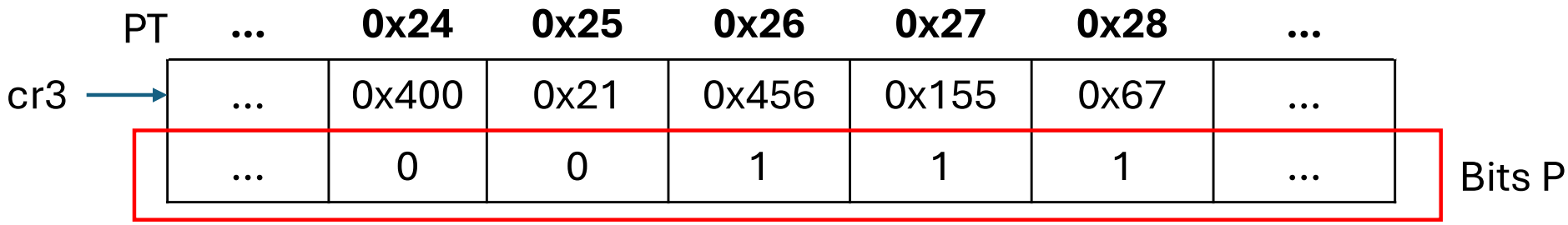
- A TLB miss incurs into a performance penalty since memory must be accessed to find the physical frame ID

Bit P in the PT

- Usually not all logical frames are mapped to physical frames
- To know if a logical frame is mapped to a physical frame, every entry in the PT is extended with a bit called Present (or bit P)
- This bits indicates in the physical frame ID of an entry is valid or not
- If the bit P is equal to 0 for an entry, it means that that logical frame is not mapped to a physical frame. So, no translation is possible
- When this happens, the CPU raises a Page Fault exception to invoke the OS
- Having a P bit in the PT changes how the TLB is accessed

TLB miss with the P bit

- Every entry in the PT has its own P bits. A value of 0 indicates that the physical frame ID is not valid.



Logical address

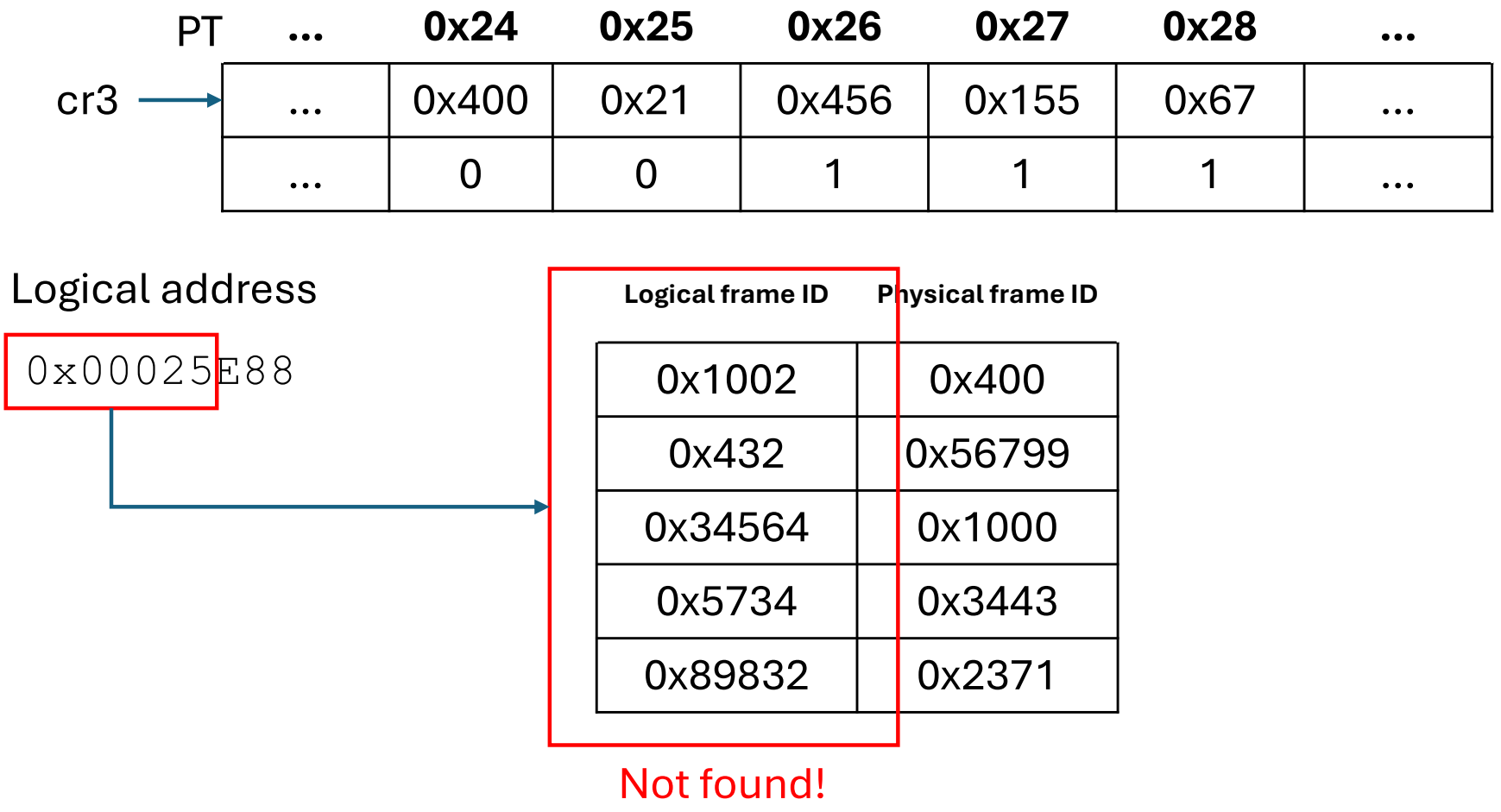
0x00025E88

Logical frame ID

Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

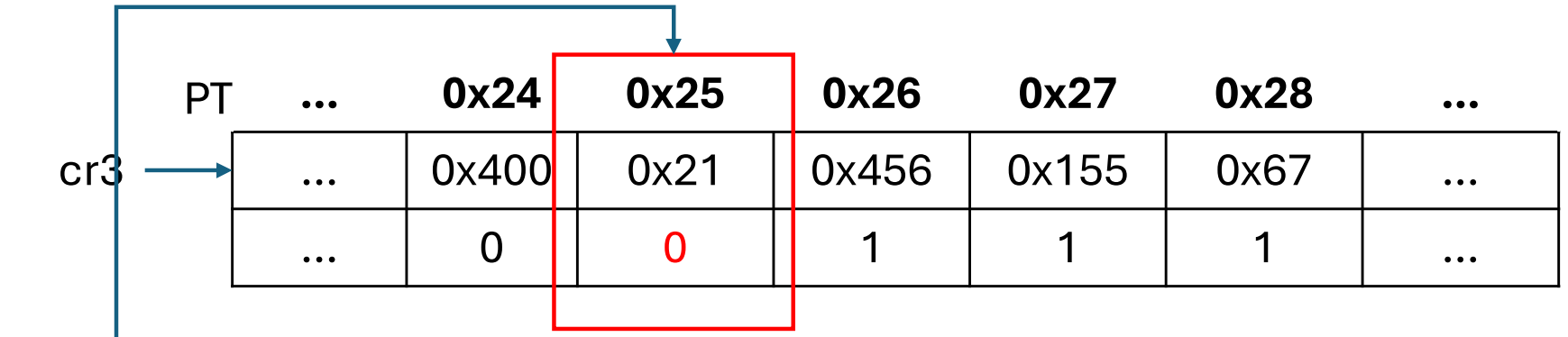
TLB miss with the P bit

- As before, TLB miss



TLB miss with the P bit

- When accessing the PT, first, check the value of P



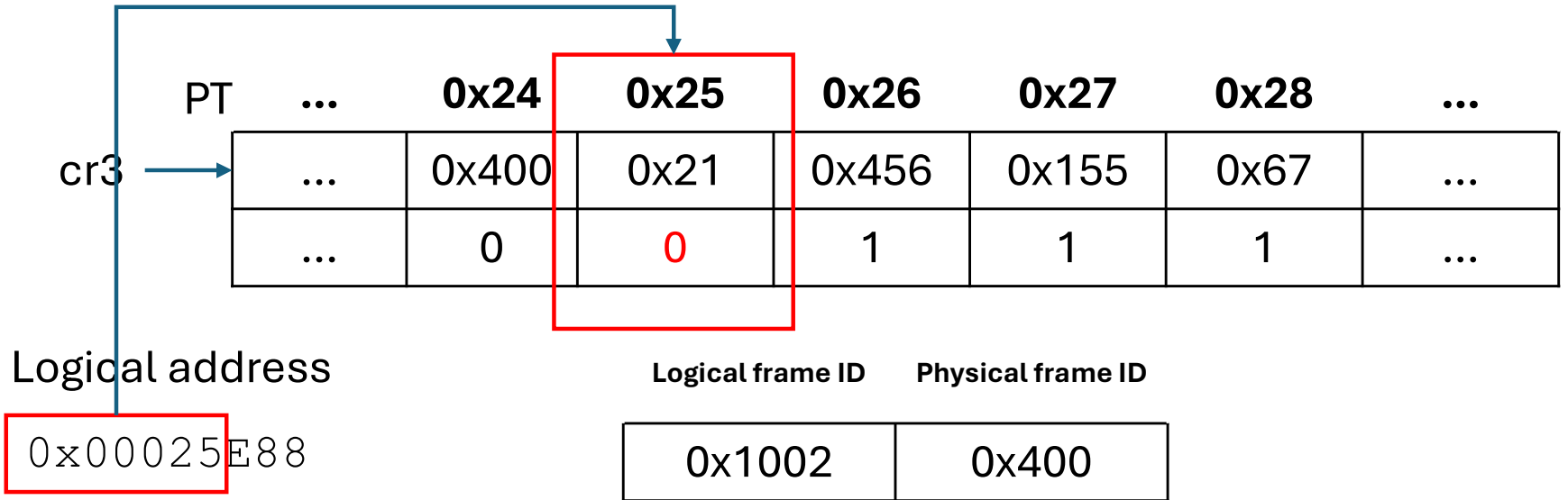
Logical address

0x00025E88

Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

TLB miss with the P bit

- And if it is 0, raise a Page Fault exception and call the OS

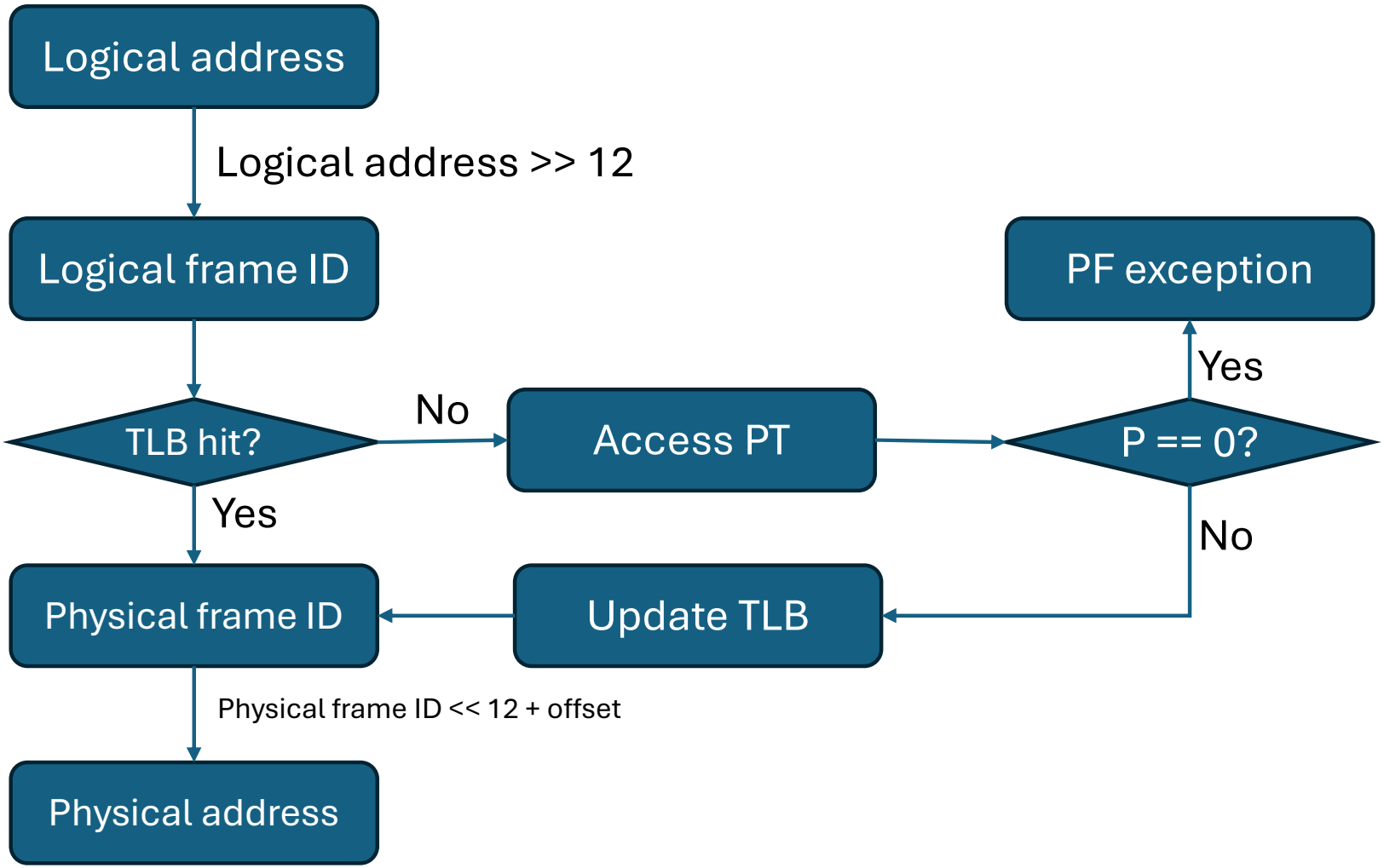


Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x5734	0x3443
0x89832	0x2371

Page Fault exception (PF)

- When the TLB cannot translate a logical address, it raises a Page Fault exception to invoke the OS
- This only happens when the CPU accesses an entry of the PT of the current process, under a TLB miss, and the value of the P bit of that entry is 0
- Under a PF, the OS must decide, depending on its features, what to do

Complete TLB access diagram



Full contents of a PT entry

```
struct {
    unsigned int present  : 1; Present bit (P)
    unsigned int execute  : 1; Execution permission
    unsigned int rw       : 1; Read/write permission
    unsigned int user     : 1; user/supervisor level
    unsigned int write_t  : 1; write through/write_back (linux write back)
    unsigned int cache_d  : 1; caching is enabled (linux sets this)
    unsigned int accessed : 1; (reset by OS)
    unsigned int dirty    : 1; (set by OS)
    unsigned int ps_pat   : 1; (page_size: normal or big)
    unsigned int global   : 1; (if set, tlb is not flushed after modifying CR3)
    unsigned int avail    : 2; (not in use)
    unsigned int pbase_addr : 20; Physical frame ID
  } page_table_entry;
```

RW bits

- RW bit is used to know if a logical frame is read only (0) or read&write(1)
- When the TLB is accessed, this bit is also checked to validate the type of access (read or write)
- In a TLB miss, the RW bit is copied in the TLB entry from the PT entry as well as the frame ID
- A load instruction will always access the memory (frames cannot be marked as no readable)
- A store to a frame with the RW set to 0 raises a #GP exception

Access type check (loads)

- A load instructions is always successful when checking the access

PT

cr3

...	0x24	0x25	0x26	0x27	0x28	...
...	0x400	0x21	0x456	0x155	0x67	...
...	0	0	1	1	1	...
	1	1	0	0	0	

Bits P

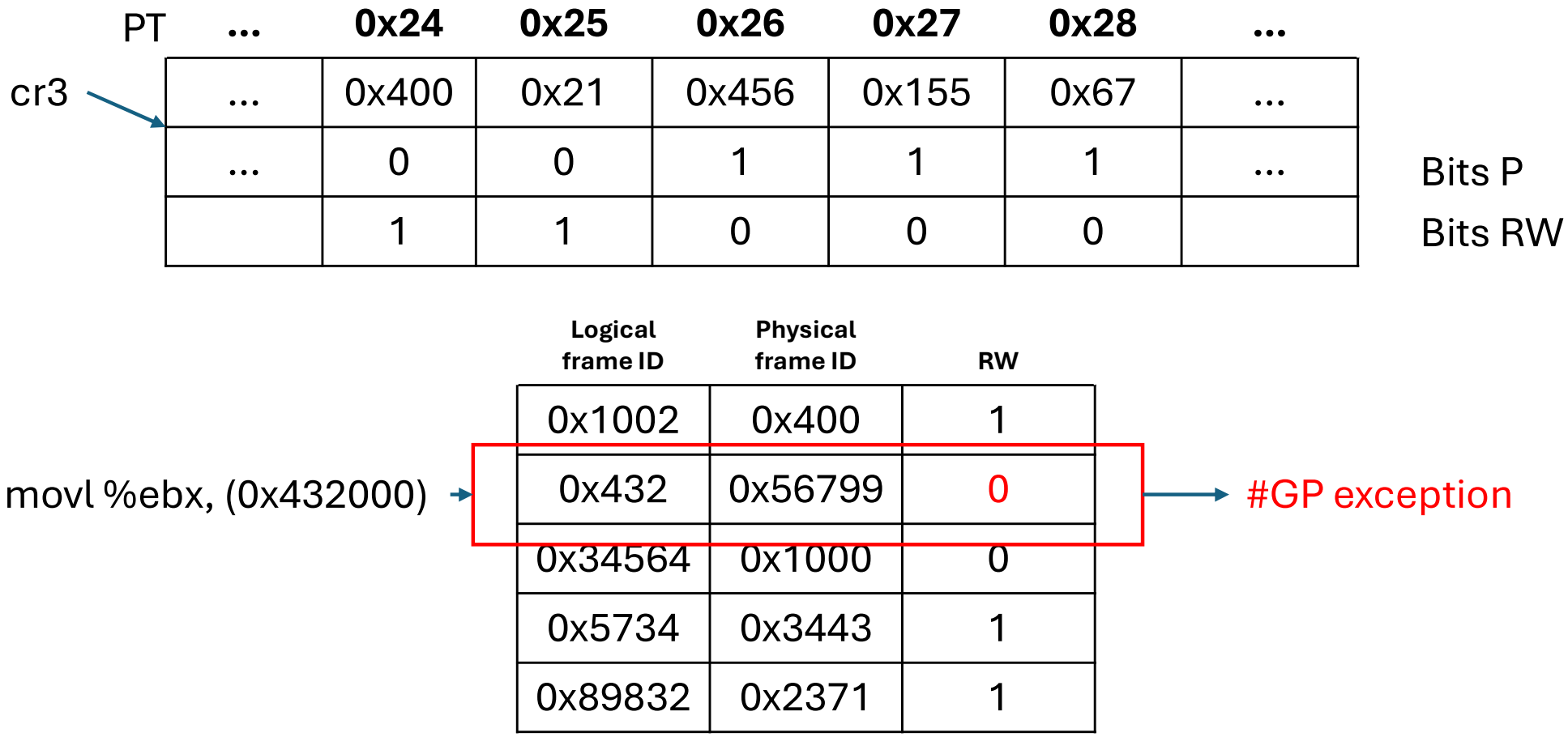
Bits RW

movl (0x432000), %ebx

Logical frame ID	Physical frame ID	RW
0x1002	0x400	1
0x432	0x56799	0
0x34564	0x1000	0
0x5734	0x3443	1
0x89832	0x2371	1

Access type check (stores)

- A store instruction can only be valid if RW bit is set to 1. If not raise #GP exception



Execute bit

- Follows the same behavior as the RW bit but for marking that a section contains code
- Before the CPU fetches an instruction, it checks if the logical address pointed by EIP is marked as executable (Execute bit set to 1)
- If it is not, raise #GP exception
- This is called Data Execution Prevention
  - Prevents Data Execution attacks used for code injection in processes
  - Does not allow ROP execution

User bit

- Access to a logical frame can also be limited depending on the privilege level
- User bit:
  - If set to 0, allow access to the logical frame from any privilege level
  - If set to 1, allow access to the logical frame only from Ring 0 (OS mode)
- The access to a logical frame marked with the User bit (set to 1) from user mode raises a #GP exception
- The User bit is copied in the TLB entry under a TLB miss

Building the OS

Memory management of the OS

- The OS is responsible of all the memory management
- This includes:
  - Enabling pagination
  - Tracking of free physical memory
  - Assignment of physical memory to processes
  - Security
- Remember that the OS always works with frame IDs
- So, whenever we are referring to a logical or physical frame, we are talking about its frame ID

Memory during boot

- When the OS boots, the CPU is in real mode
  - All the memory addresses are physical
  - No address translation is performed
- When the OS has setup all the needed structures, it changes to protected mode
  - From that point onwards all memory addresses are logical
  - Adress translation is enabled
- So, a part of the OS must be compiled to be executed in real mode **and** protected mode
- Solution: make logical and physical address coincide through identity mapping

Tracking of free physical memory

- In addition, in boot time, the OS creates a list of free physical frames
- A physical frame ID that it is not in that list means that
  - It has been assigned
  - It does not exist
- Useful when the OS needs to allocate memory
- Management functions:

```
// Allocates one physical page and removes it from the list of free physical frames
// Return the allocated physical frame ID
int alloc_frame();

// Marks one physical frame as available and inserts it in the list of free physical frames
void free_frame(int frameID);
```

Assignment of physical memory

- The OS manages the assignment of physical memory to processes
- For that, the OS modifies the PT of the process to map/unmap a physical frame in a given entry
- To map a physical frame to a logical frame in the PT of a process, the following function is used:

```
// PT:          page table to modify
// logicalID:   logical frame (or PT entry) being mapped
// physicalID: physical frame mapped
int set_ss_pag(page_table *PT, int logicalID, int physicalID);
```

Unmapping of physical memory

- Physical frames can be unmapped
- This means that the physical memory assigned to the process decreases
- The function to unmap a physical frame is

```
// PT:          page table to modify
// logicalID:   logical frame (or PT entry) being unmapped
int del_ss_pag(page_table *PT, int logicalID);
```

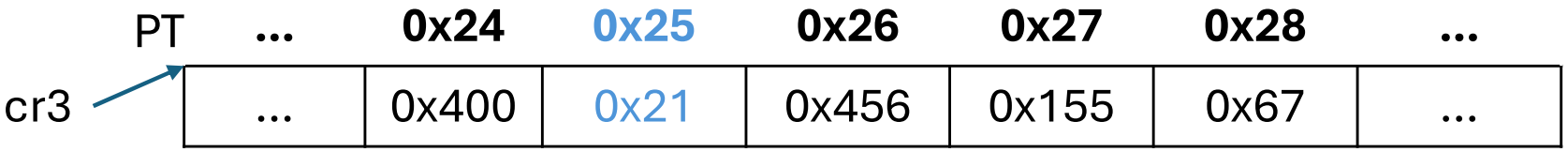
- This function sets the P bit to 0 in the logicalID entry of the PT
- The physical frame is not automatically freed. Free_frame must be used instead.

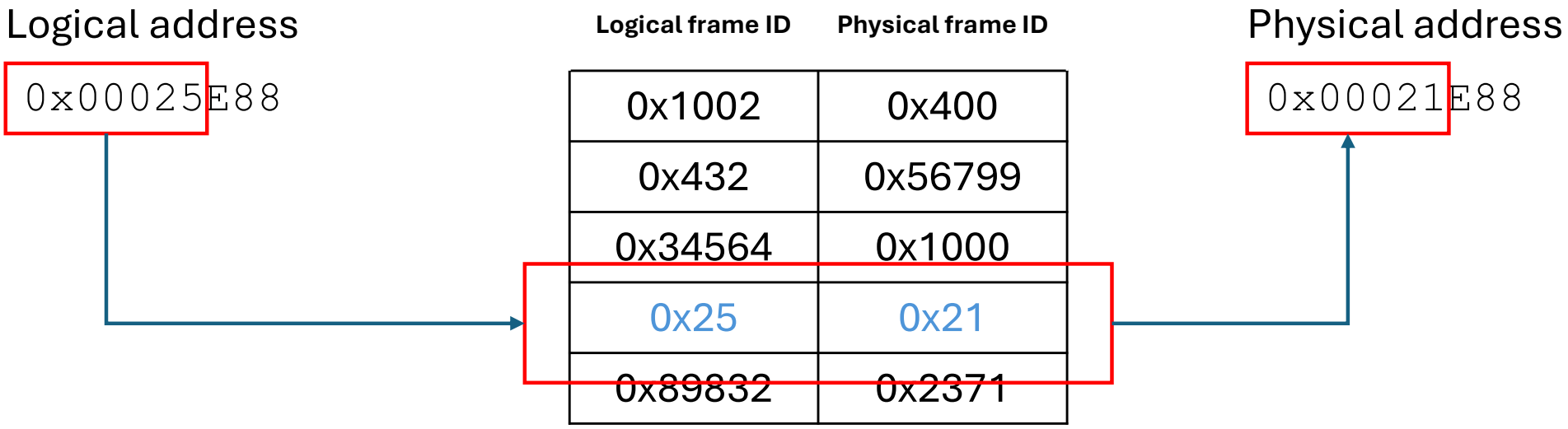
PT and TLB synchronization

- Remember that the TLB miss is the mechanism to update the TLB contents
- Modifying the PT of a process, does not modify the TLB
- So, it is possible, that for a given logical frame ID, the map in the PT and in the TLB do not coincide
- To prevent this, whenever the PT of a process has been modified, the TLB must be flushed
- To flush the TLB: write the contents of the CR3 CPU register

TLB flush

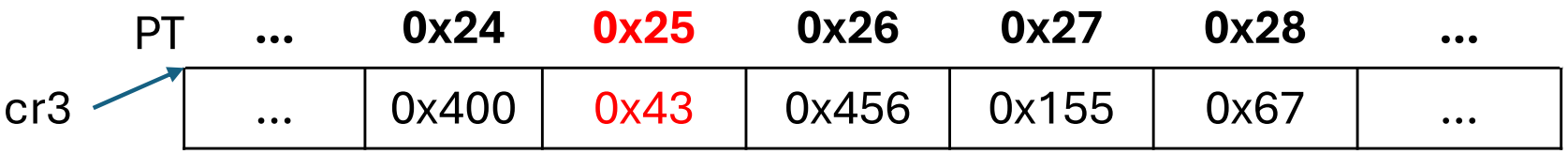
- As before, the TLB translates addresses. The logical frame ID 0x25 is in the TLB and its corresponding physical frame ID coincides with that of the PT





TLB flush

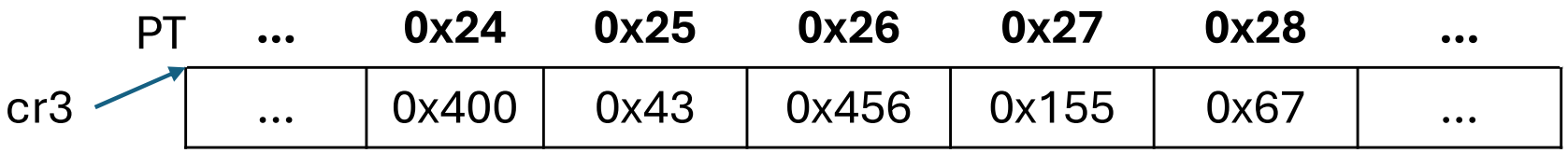
- After executing set_ss_pag(PT, 0x25, 0x43) the new map is updated in the PT but not in the TLB

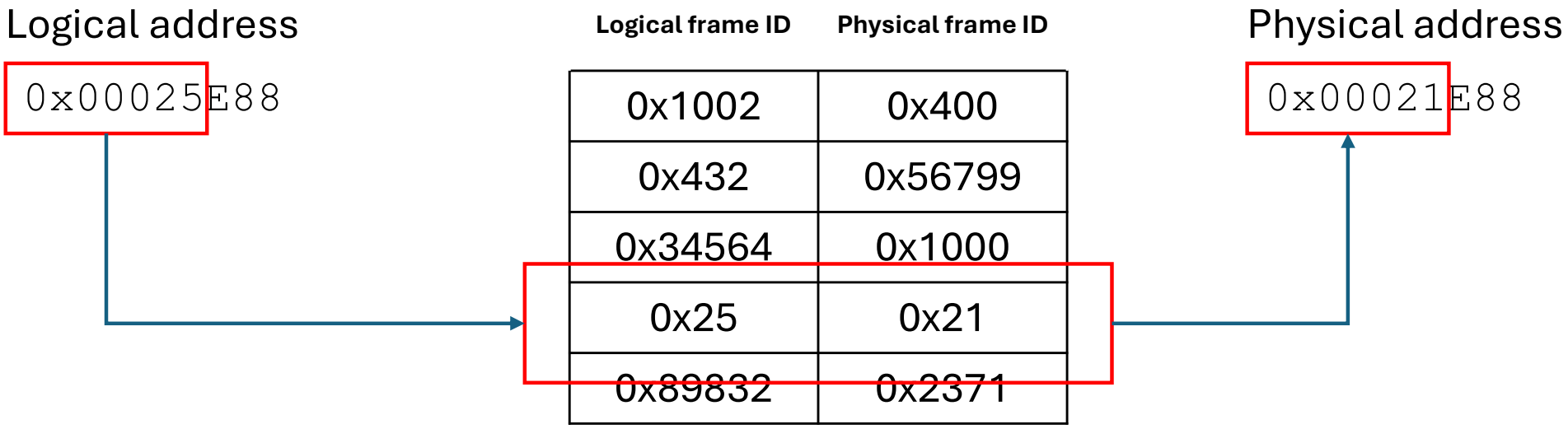


Logical frame ID	Physical frame ID
0x1002	0x400
0x432	0x56799
0x34564	0x1000
0x25	0x21
0x89832	0x2371

TLB flush

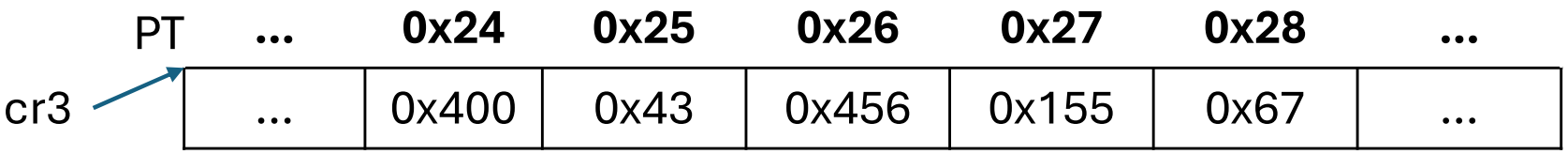
- The logical frame ID 0x25 will translate to 0x21 in the TLB since it is a TLB hit but it should have been to 0x43





TLB flush

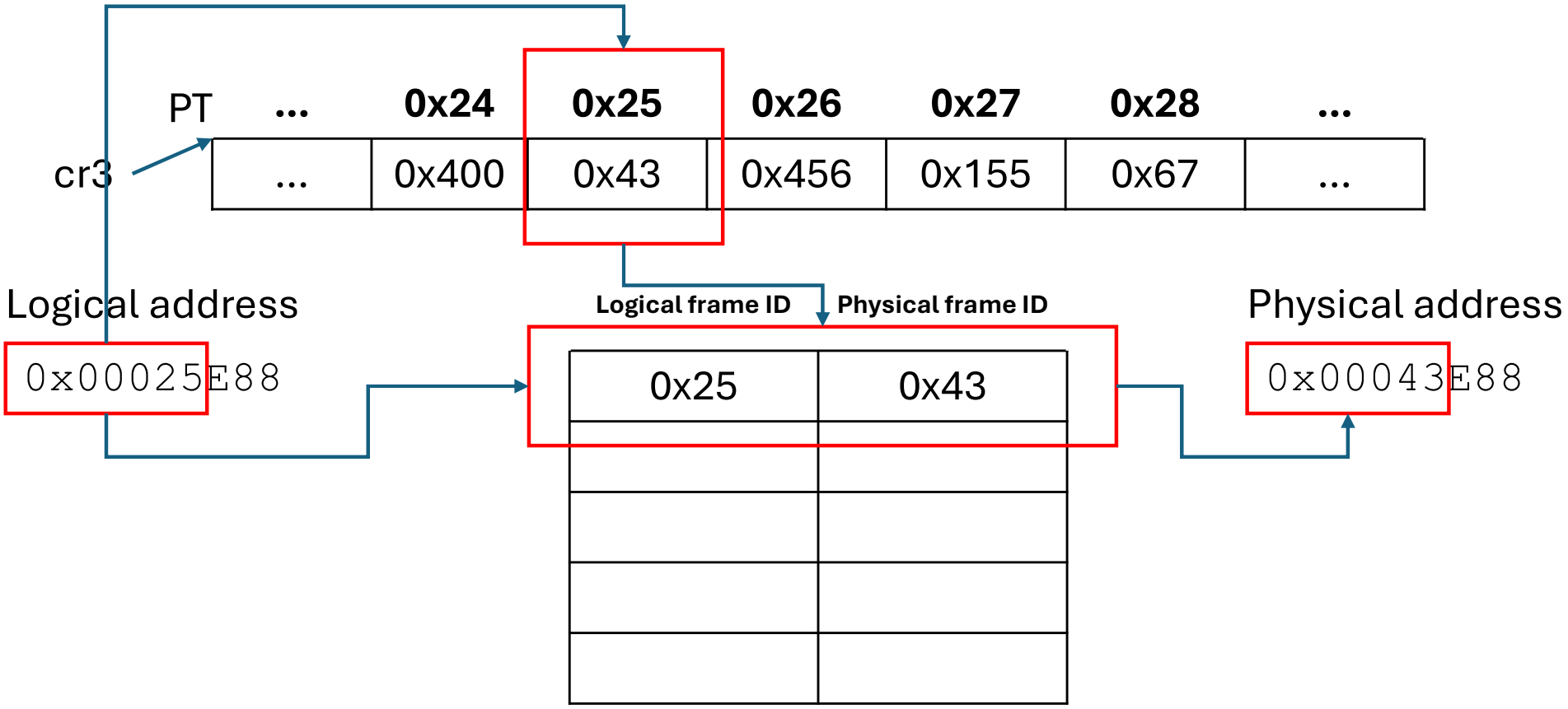
- TLB flush: set_cr3(PT);



Logical frame ID	Physical frame ID

TLB flush

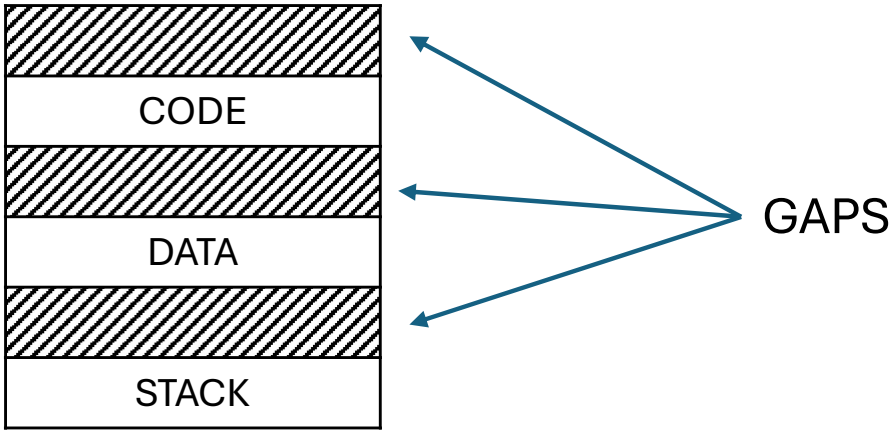
- After flushing the TLB, the new access will cause a TLB miss and the TLB will be updated with the correct value



Process structure

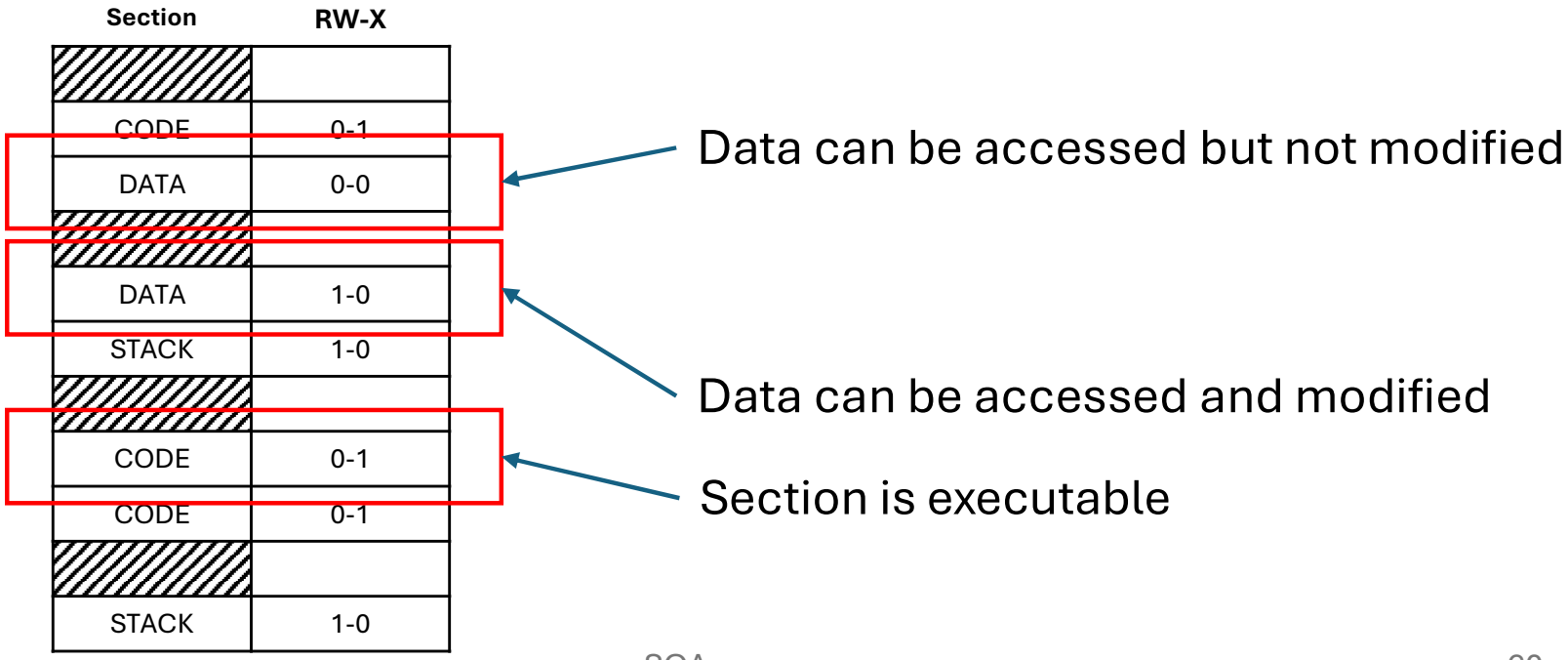
- The OS programmer decides how the logical address space of a processor will be used
- The code, data and stack are distributed into one or more differentiated sections
- Gaps between sections are deliberately left
- There can be one or more sections for code, data and stacks

Image of a Process in memory



Process sections

- A section groups information of the same type inside the processor
- Can be more than one section of the same kind
- To prevent attacks and wrong accesses to sections, they are protected using the RW and X bits in the PT

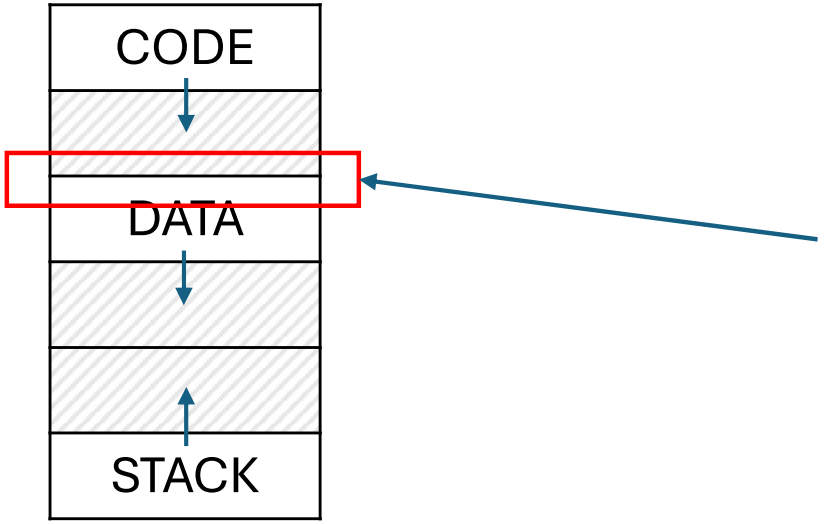


Memory gaps

- Gaps are intentionally left in the image in memory of a process
- They are implemented setting the P bit to 0 in logical pages
- This allow sections to grow dynamically:
  - Code: loading dynamic link libraries
  - Data: through memory allocation (sbrk)
  - Stack: automatically grows when the top is reached
- Accessing a gap will raise a Page Fault exception that the OS must process
  - i.e. generating a segmentation fault and killing the process

Section growing

- To make a section grow, the OS maps new physical frames at the end of the section
- The OS must check that section do not collide with the following one by checking the P bit of the PT
- If two sections will collide, return error



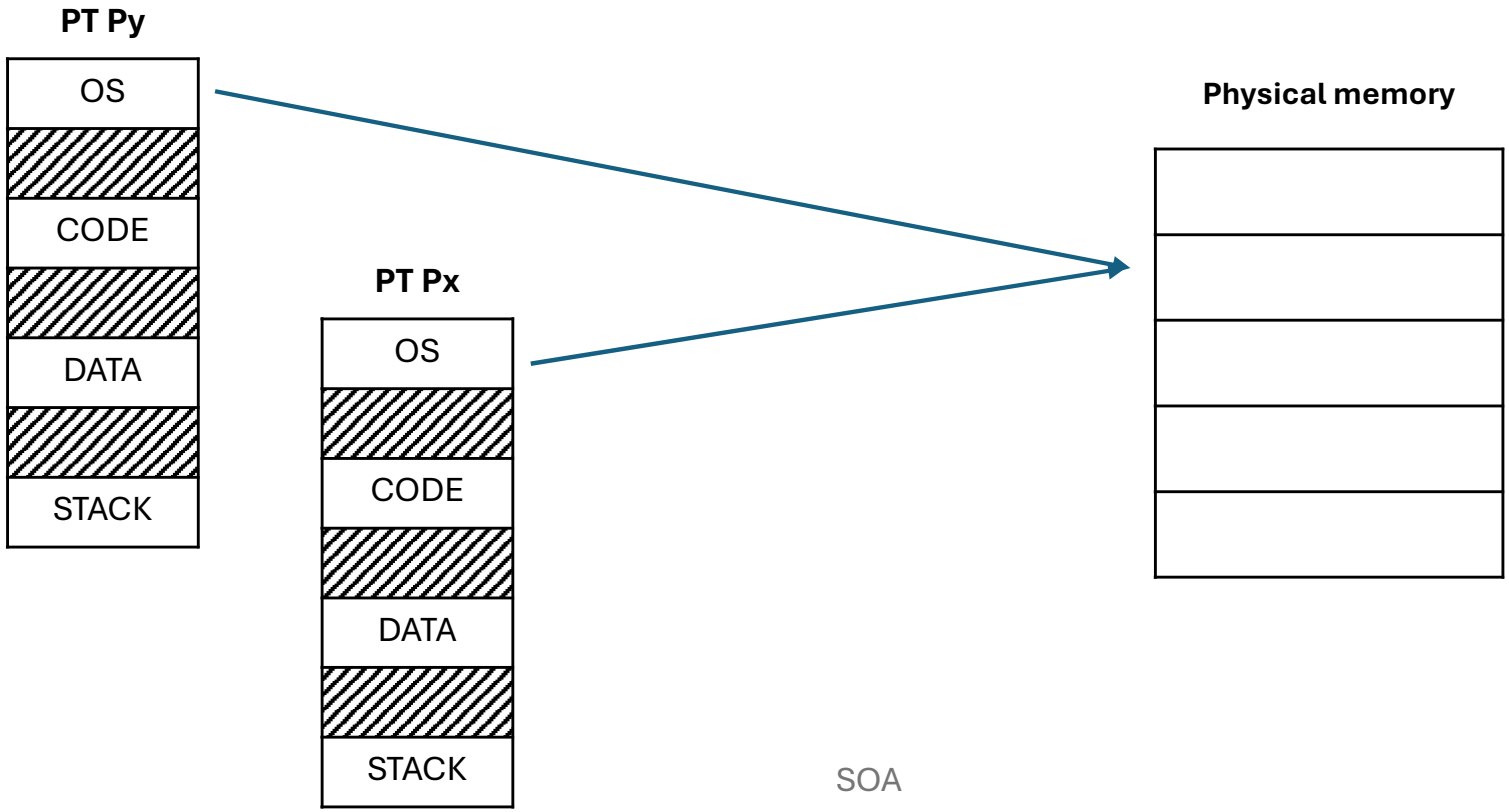
Possible collision:
Check the P bit of the PT to validate that the next logical frame is not mapped to a physical frame

OS Memory

- Remember that after enabling Protected Mode, all addresses are logical address and must be translated
- This means that even in OS mode privilege level, memory addresses are logical
- So, when the OS is executing, the TLB must be used to translate logical addresses
- Problem: where the maps between logical and physical frames of the OS are stored?
- Solution: map the OS addresses in the PT of the processes

Kernel memory

- The physical memory storing the OS code and data, is mapped in the same place in the PT of all processes
- This makes that OS code and structures are shared among all processes



OS Memory protection

- Problem: since the OS memory is mapped in the PT of the process, OS memory could be accessed in user mode.
- Solution: use the USER bit in the PT to mark OS memory