INF113: Paging

Kirill Simonov

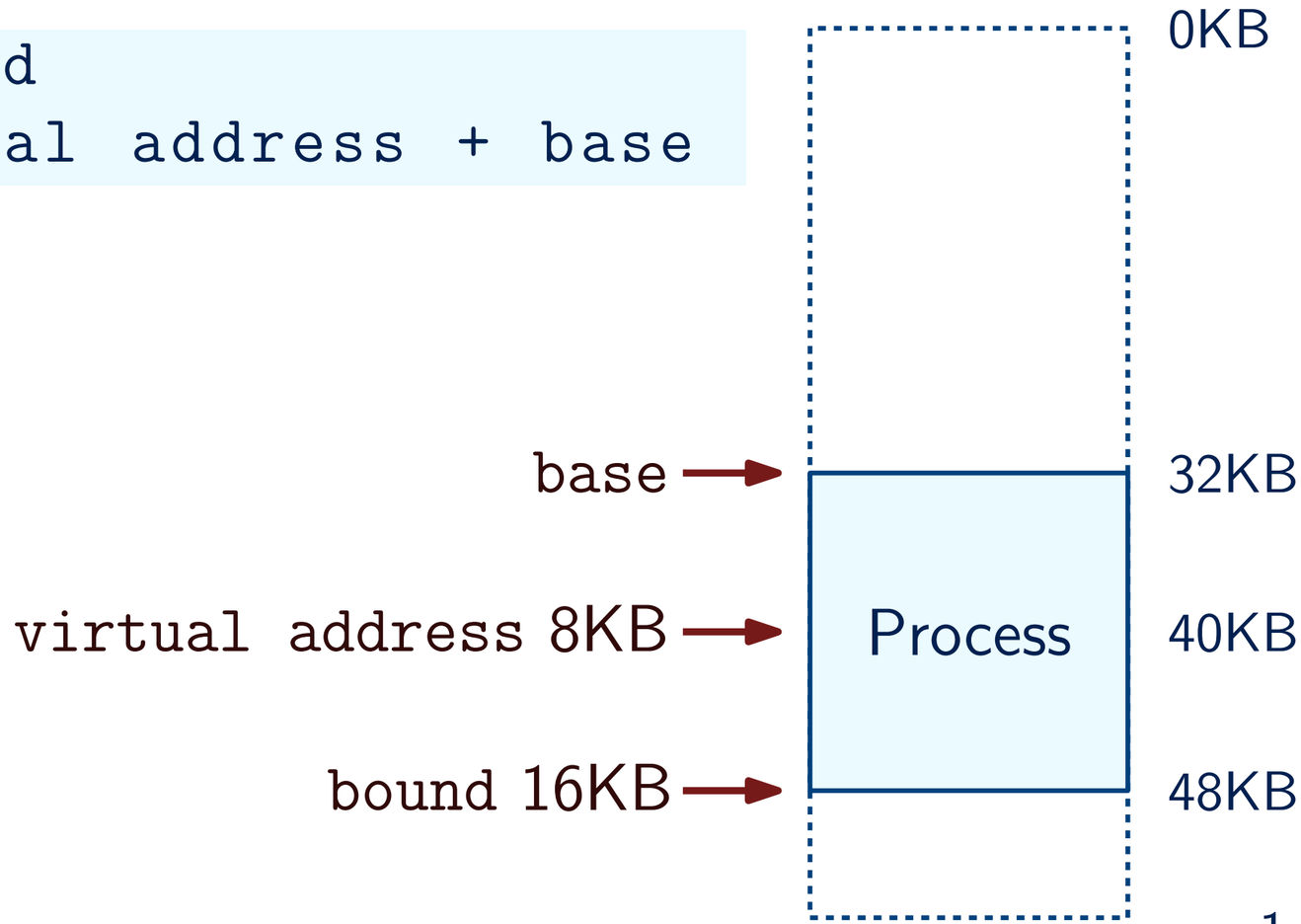
08.10.2025

Base and bound: Reminder

- Process takes a contiguous chunk of memory, start and end stored in registers
- Translating addresses is easy:

```
if virtual address >= bound
  physical address = virtual address + base
```

- **Main issue:** Internal fragmentation
  - Each process needs lots of "spare" memory within its address space
- Address space may never be as large as physical memory

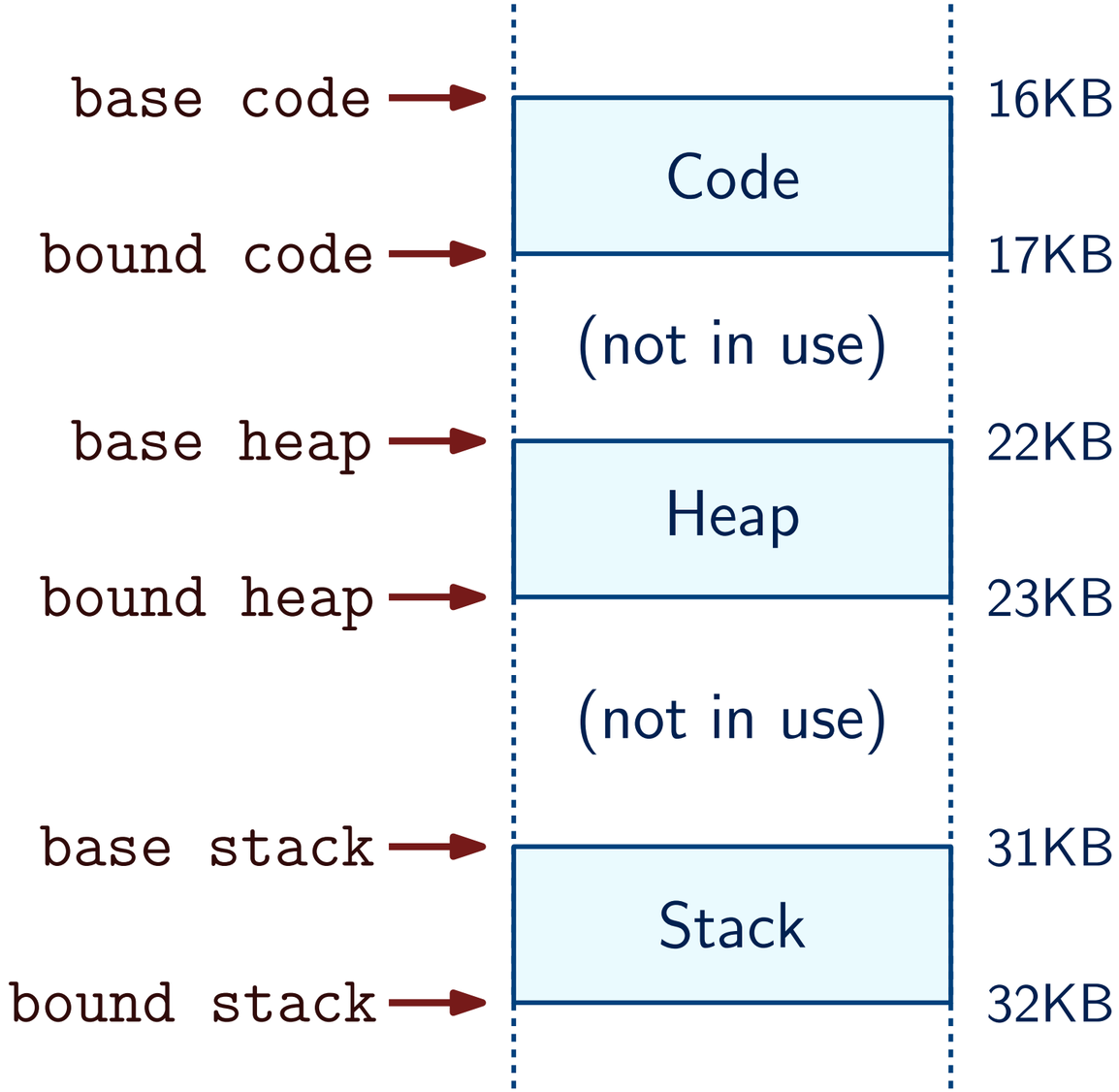


Segmentation: Reminder

- Each logical segment takes a contiguous chunk of memory
- For address translation, look up the segment table

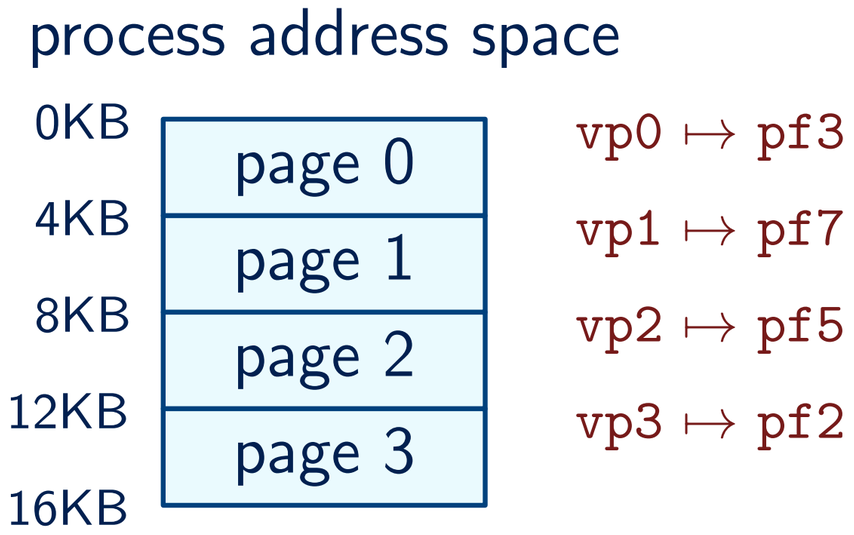
	base	bound	dir	rights
code	16KB	1KB	1	r-x
heap	22KB	1KB	1	rw-
stack	32KB	1KB	0	rw-

- **Main issue:** External fragmentation
  - Hard to arrange all process segments without gaps



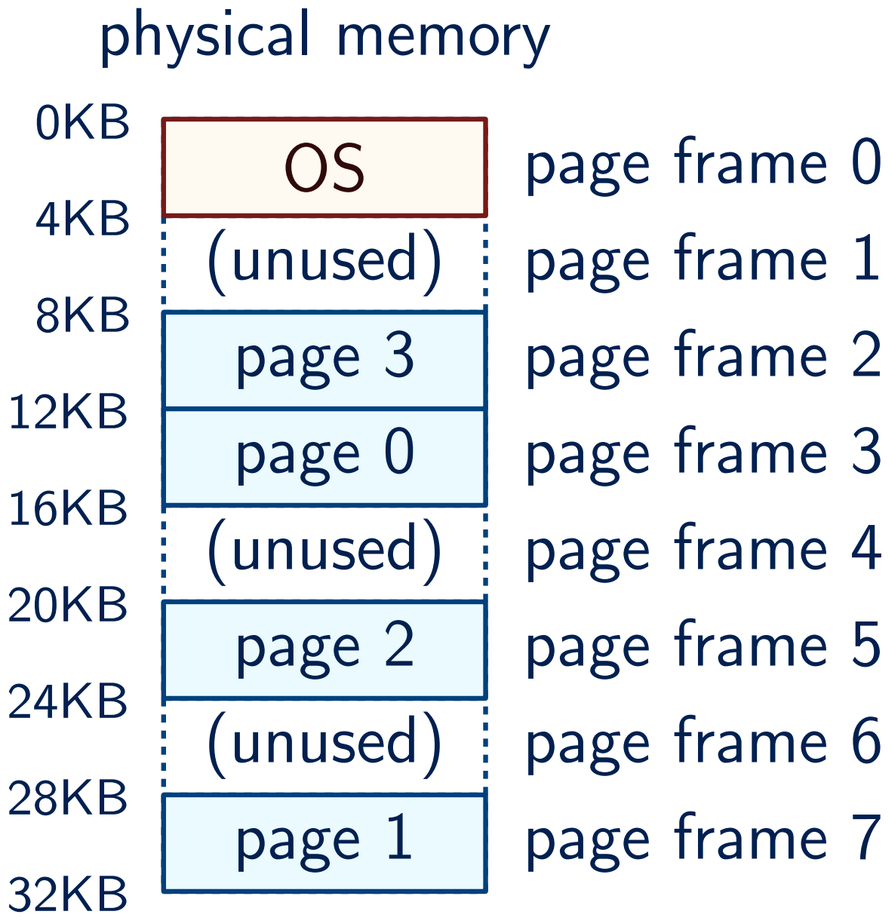
Paging

- Partition address space into small same-sized **pages**, e.g., 4KB each



- Each page is stored at some page frame

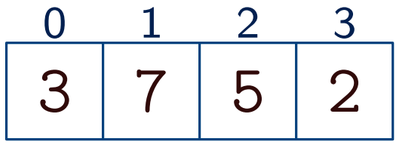
- Partition physical memory into **page frames** of the same size



Page table

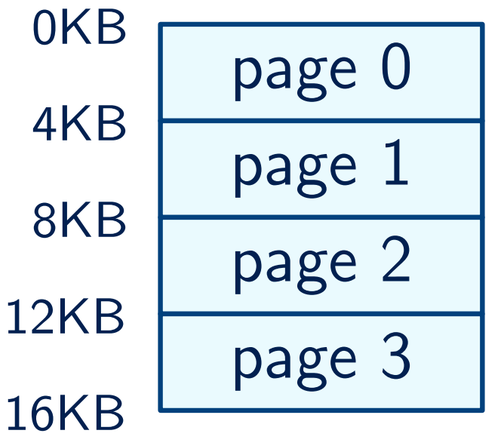
- OS stores the **page table**, mapping virtual pages to physical page frames

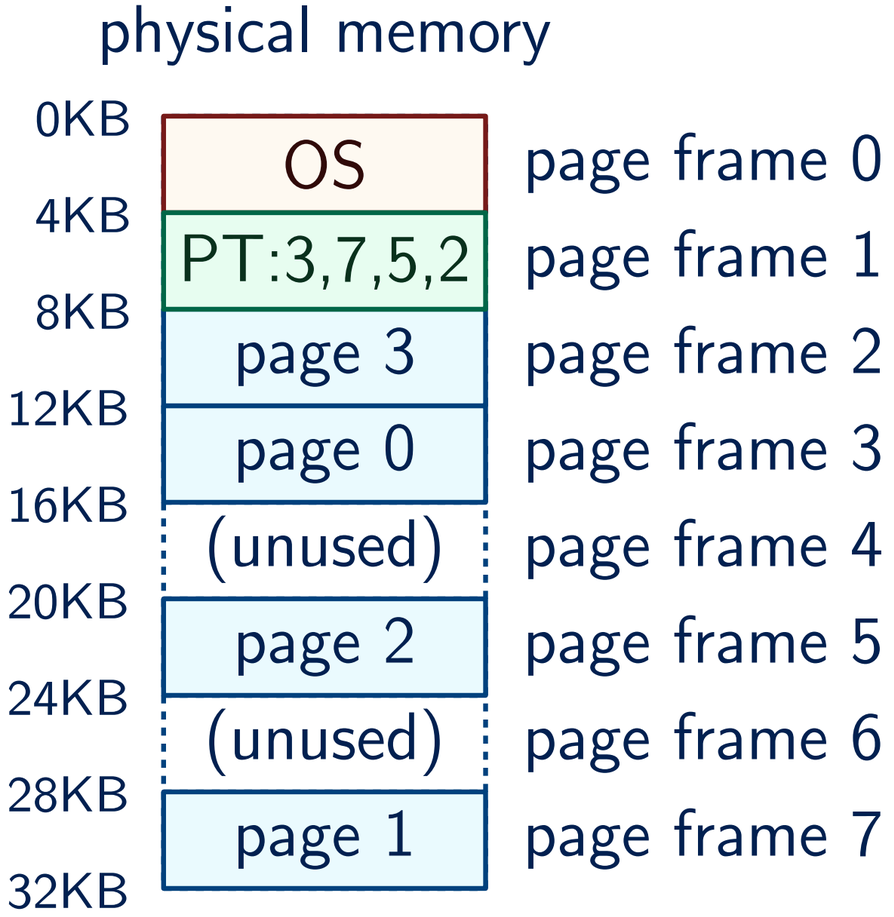
virtual page number	VPN	PFN	physical frame number
	0	3	
	1	7	
	2	5	
	3	2	



page table
as an array

- Page table is stored in the memory

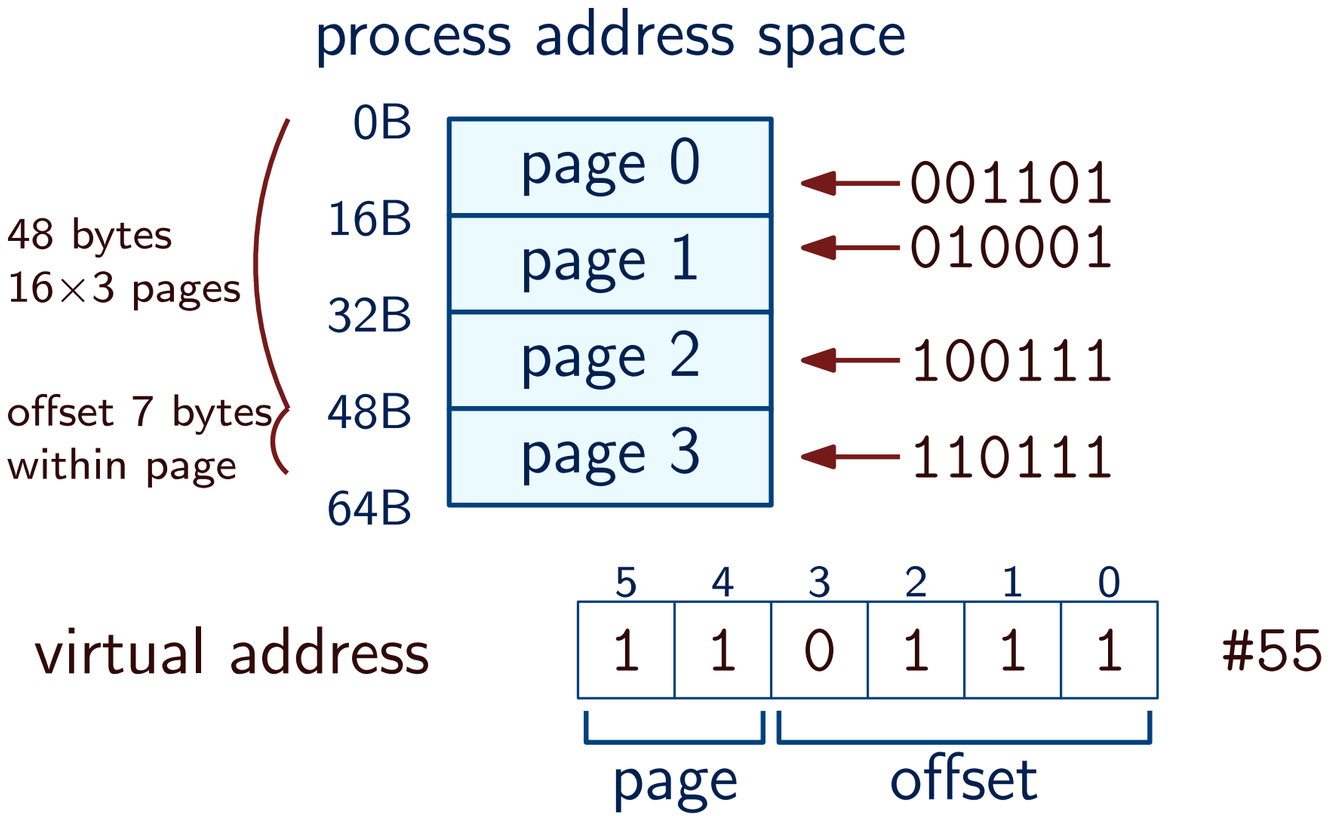


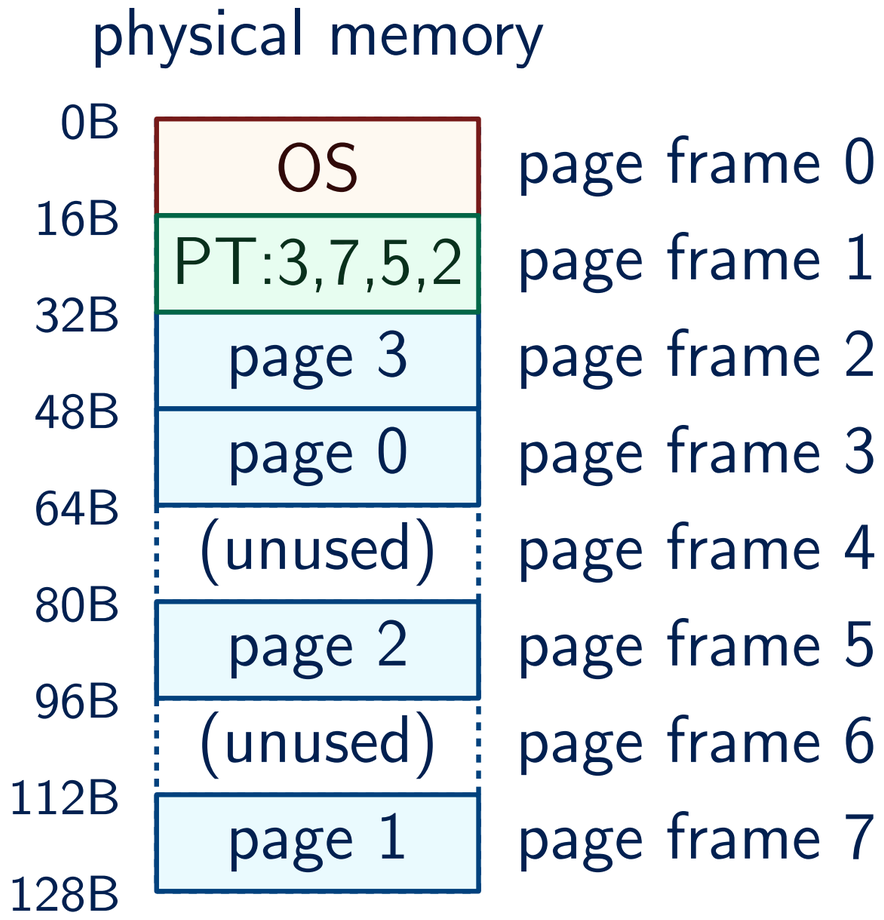


Address translation

assume page is 16 bytes

- When a process requests memory by the virtual address, MMU converts it to the physical address via the page table

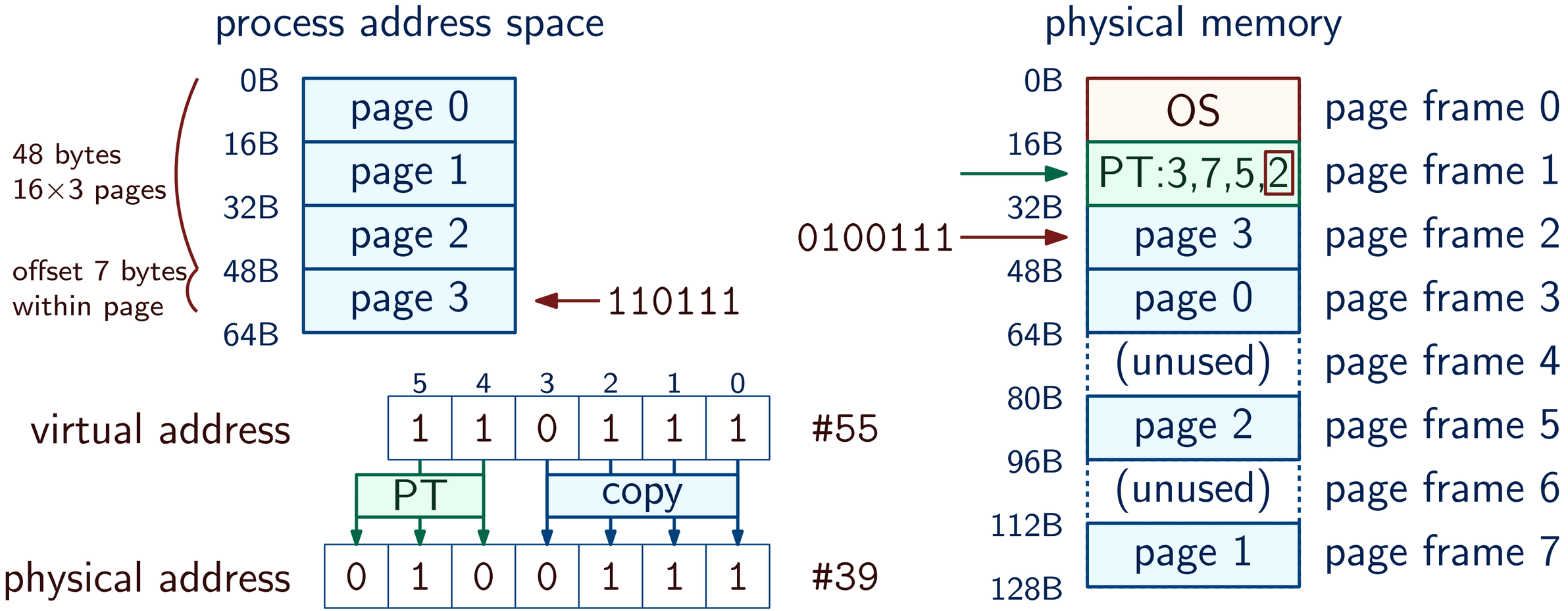




Address translation

assume page is 16 bytes

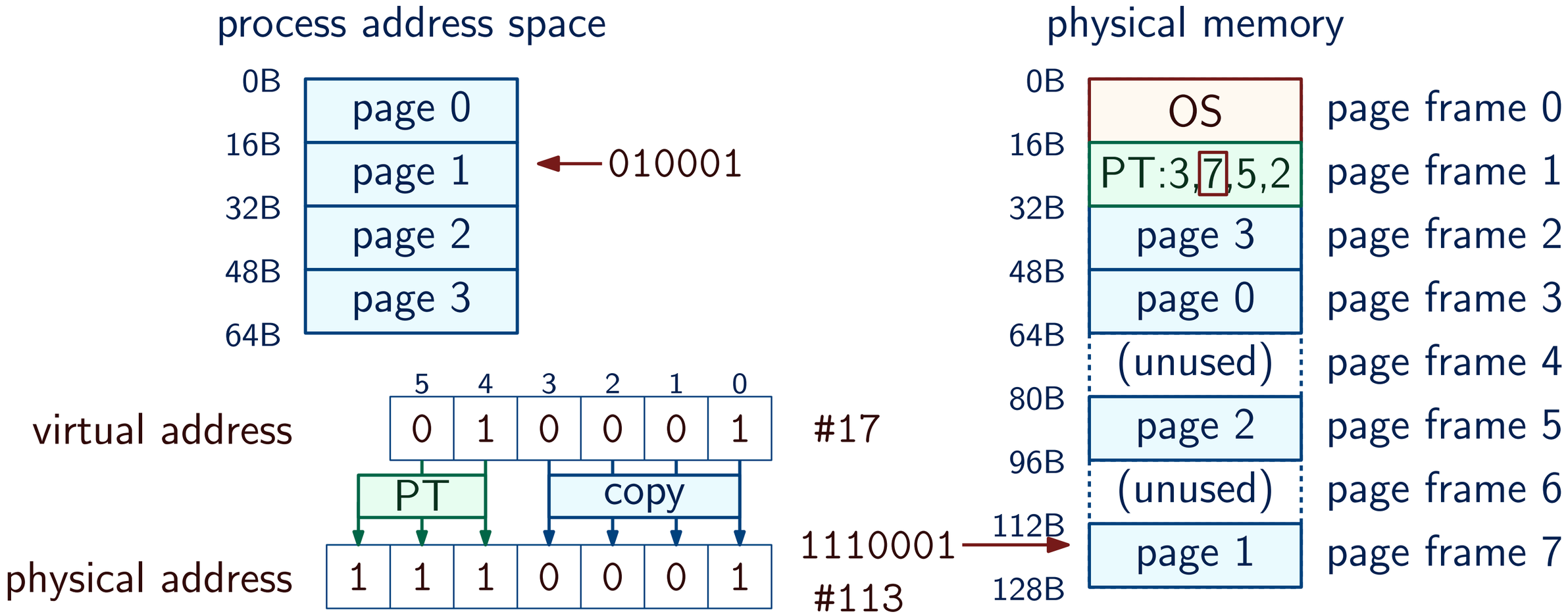
- When a process requests memory by the virtual address, MMU converts it to the physical address via the page table



Address translation

assume page is 16 bytes

- When a process requests memory by the virtual address, MMU converts it to the physical address via the page table



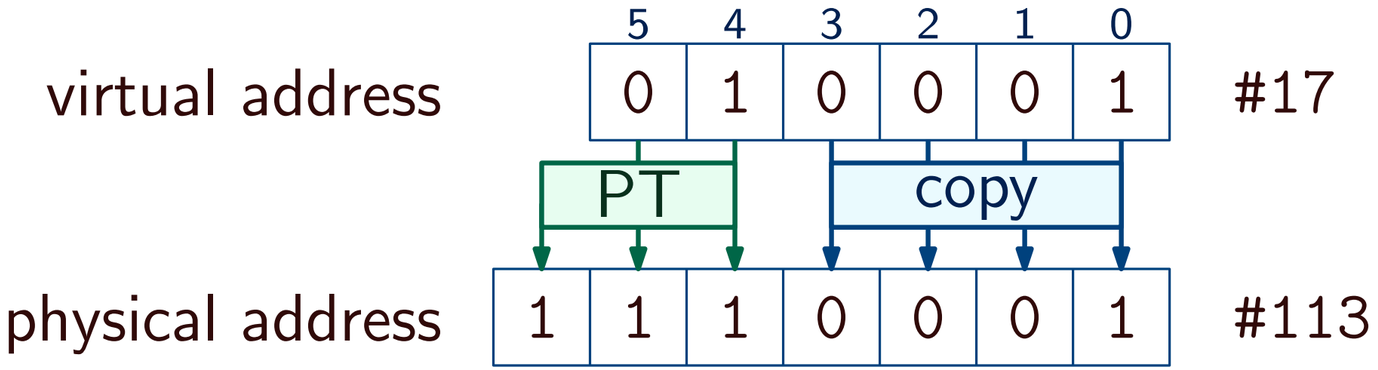
Address translation

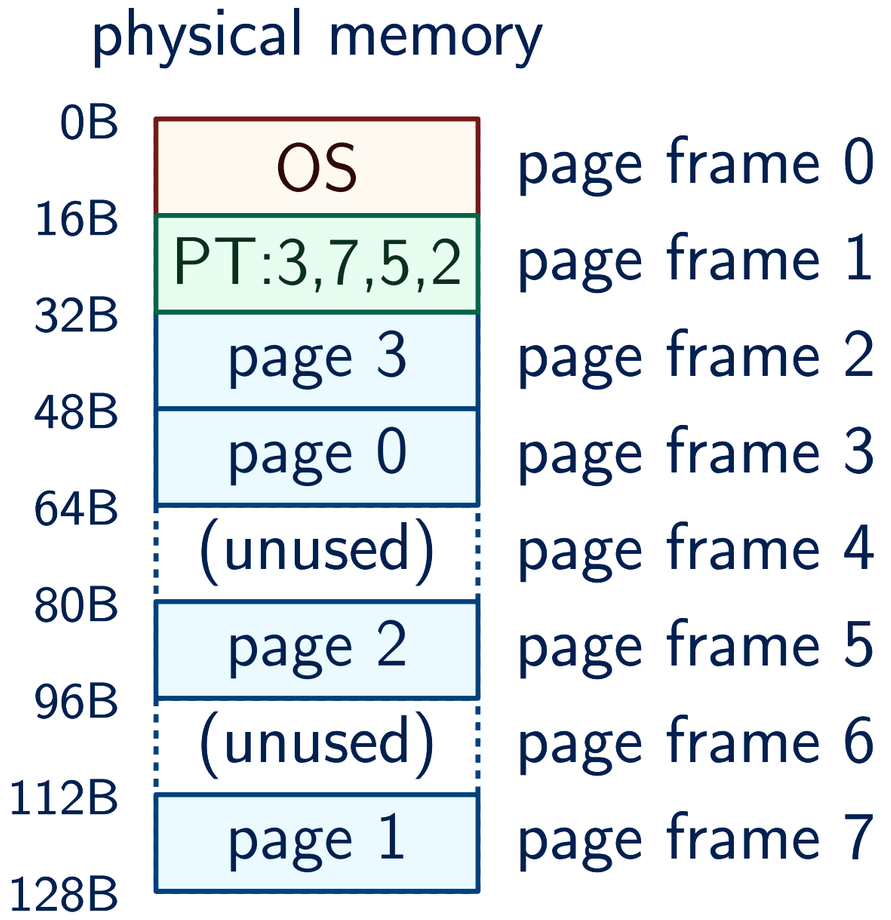
- If the process performs

```
movl virt_addr, %eax
```

- MMU translates the address as follows

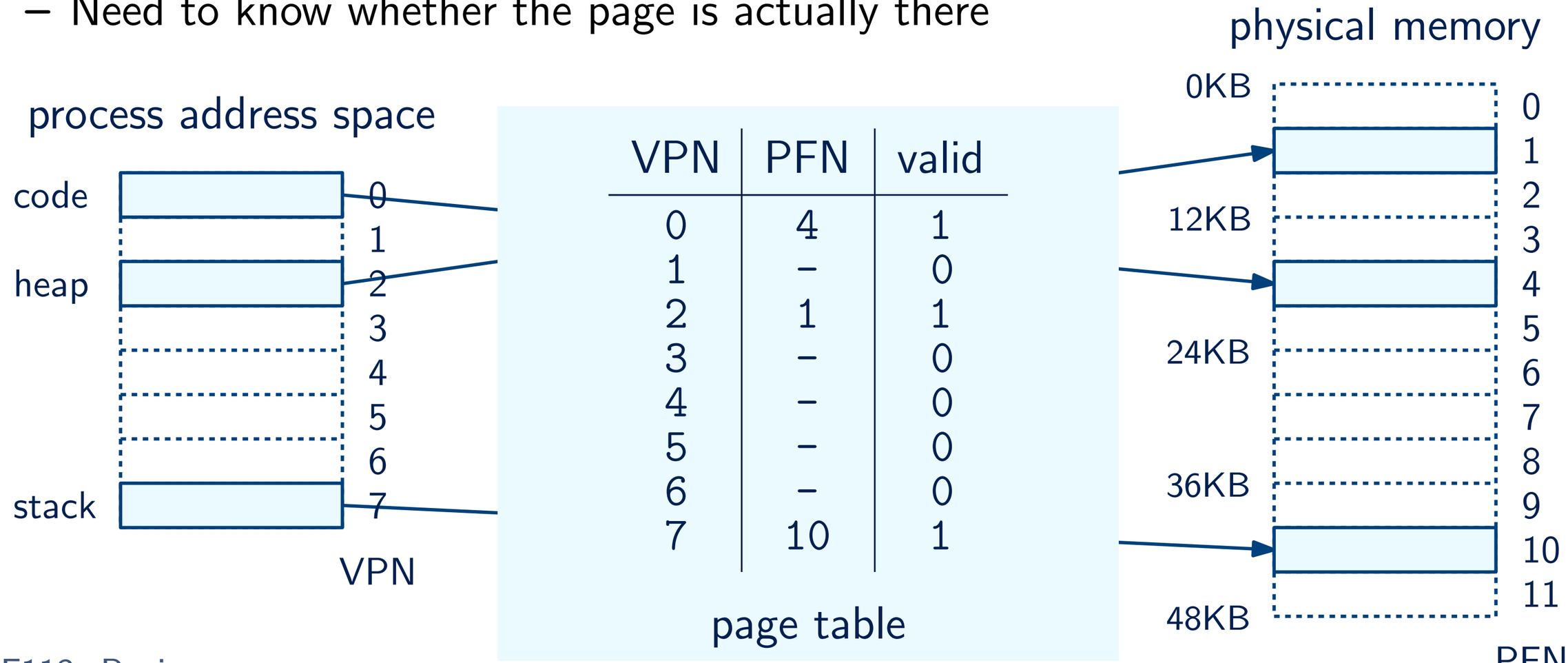
```
offset = virt_addr & 0x3f
VPN = (virt_addr & 0xc0) >> 6
phys_addr = (PT[VPN] << 6) | offset
```





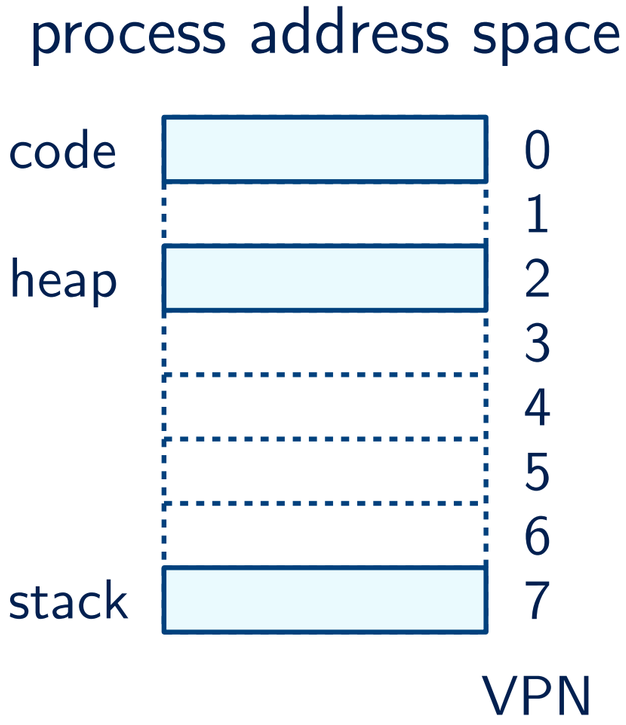
Larger address space

- In practice, processes will have huge address spaces
  - Most pages will not be used
  - Need to know whether the page is actually there



Page Table Entry (PTE)

- OS stores a PTE record for each page
  - PFN, page frame number
  - validity bit—is page allocated?
  - present bit—is page in RAM?
  - protection bits
  - ...

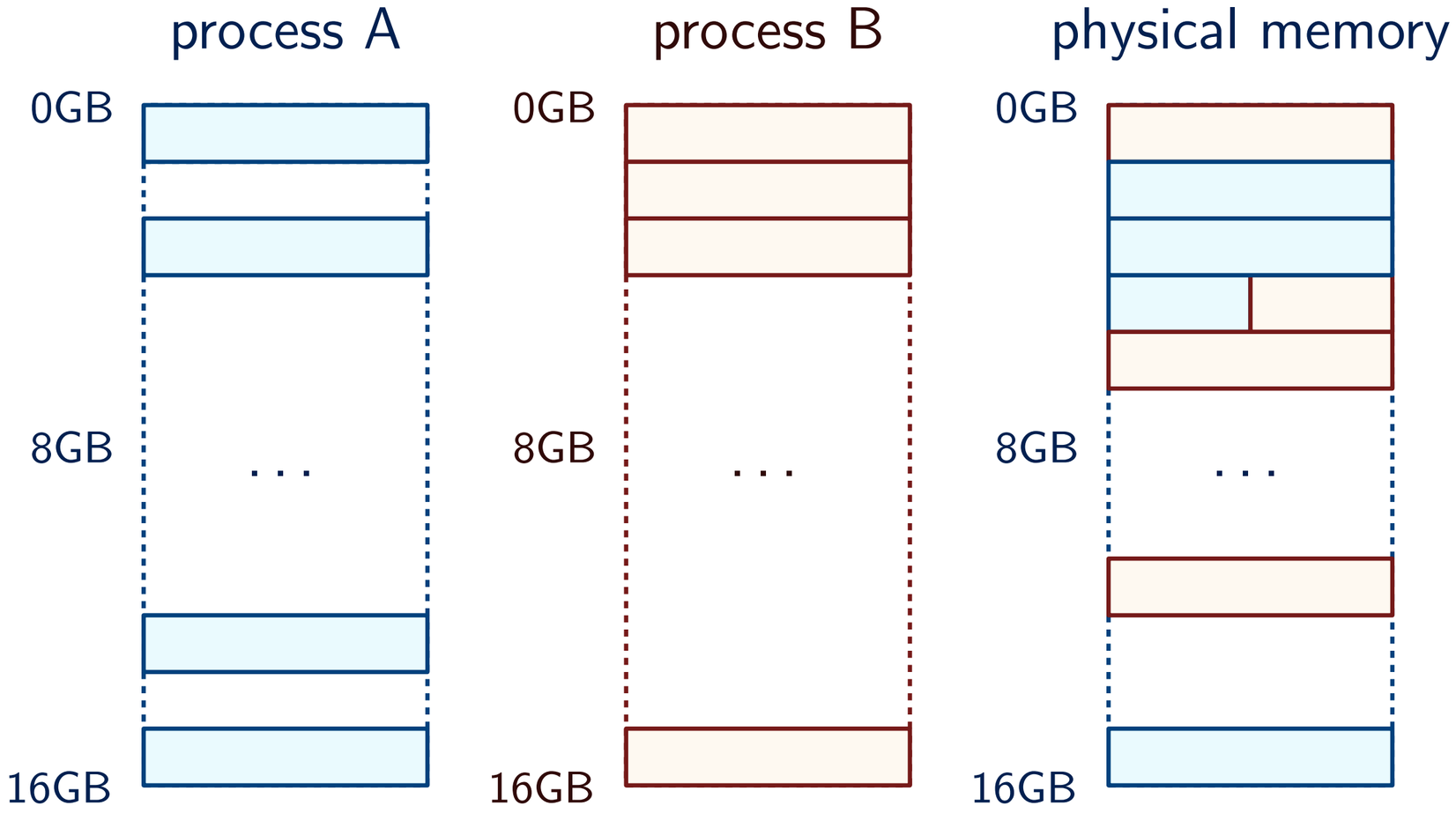


VPN	PFN	valid	pr.	prot.
0	4	1	1	r-x
1	-	0	0	---
2	1	1	1	rw-
3	-	0	0	---
4	-	0	0	---
5	-	0	0	---
6	-	0	0	---
7	10	1	1	rw-

page table

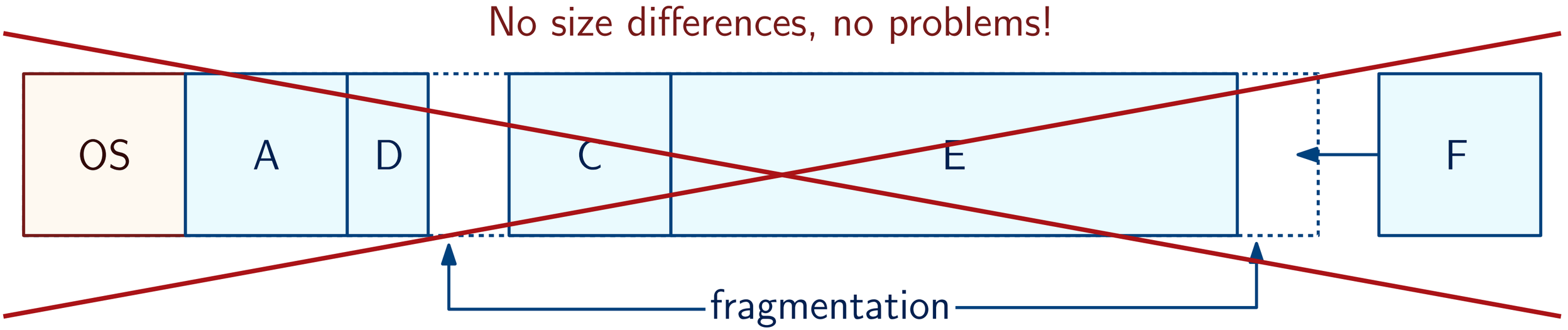
Benefits of paging

- Great abstraction
  - Each process has huge and (seemingly) contiguous address space
  - But OS can arrange individual pages very flexibly on the RAM



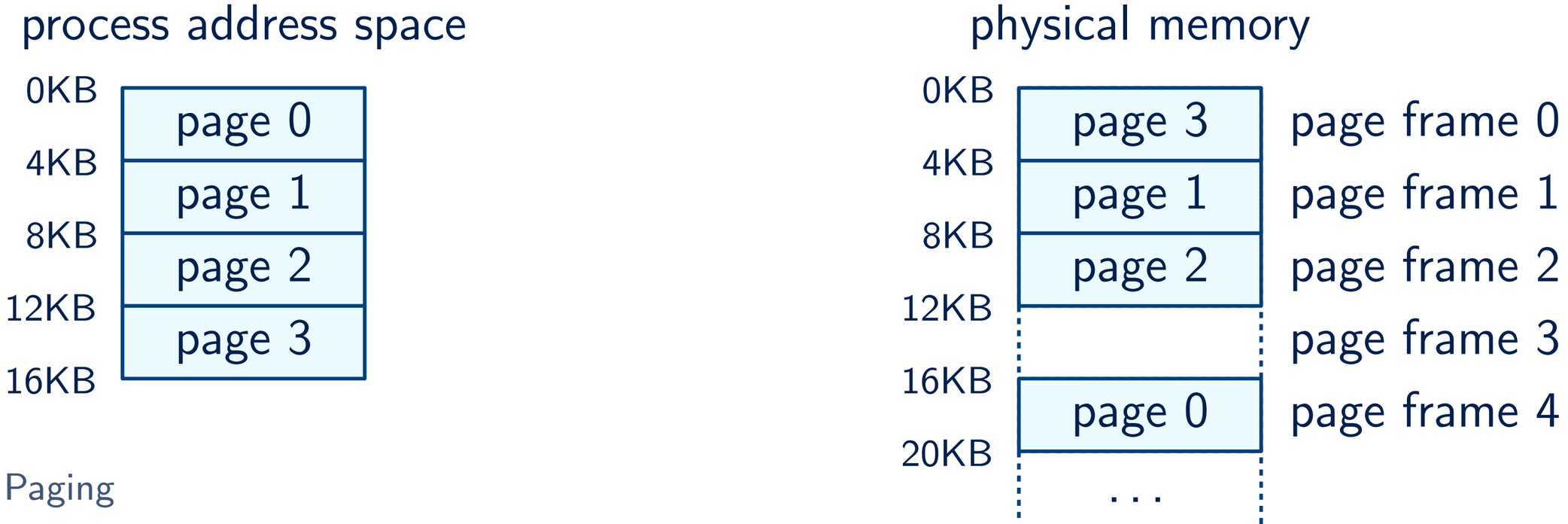
Benefits of paging

- Great abstraction
  - Each process has huge and (seemingly) contiguous address space
  - But OS can arrange individual pages very flexibly on the RAM
- Easy free space management for OS
  - Any frame fits any page, so a page can be placed in any free slot



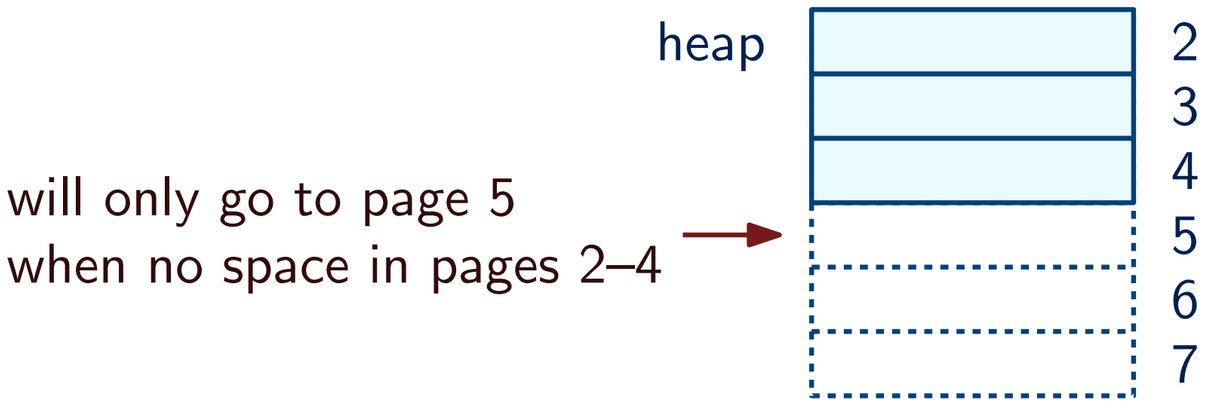
Benefits of paging

- Great abstraction
  - Each process has huge and (seemingly) contiguous address space
  - But OS can arrange individual pages very flexibly on the RAM
- Easy free space management for OS
  - Any frame fits any page, so a page can be placed in any free slot



Benefits of paging

- Great abstraction
  - Each process has huge and (seemingly) contiguous address space
  - But OS can arrange individual pages very flexibly on the RAM
- Easy free space management for OS
  - Any frame fits any page, so a page can be placed in any free slot
- Almost no fragmentation
  - As soon as a page frame is free, more pages can be allocated
  - Only the remainder of the last page in each logical segment has to be wasted; just a few KBs per process



Benefits of paging

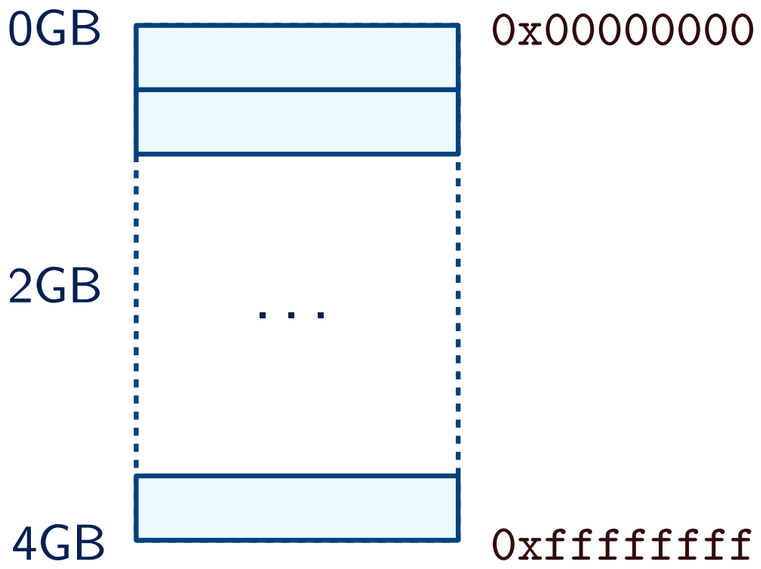
- Great abstraction
  - Each process has huge and (seemingly) contiguous address space
  - But OS can arrange individual pages very flexibly on the RAM
- Easy free space management for OS
  - Any frame fits any page, so a page can be placed in any free slot
- Almost no fragmentation
  - As soon as a page frame is free, more pages can be allocated
  - Only the remainder of the last page in each logical segment has to be wasted; just a few KBs per process
- However, there is a price...
  - Memory for storing page tables
  - Each memory access has an overhead of reading the page table

Storing page tables

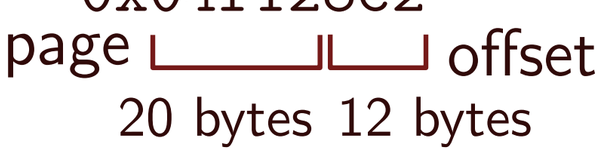
- Larger address space means larger page table
  - 32-bit addresses—at most 4GB—one million 4KB pages
  - For each page, store at least the PFN—4 bytes
  - At least 4MB for the page table
- Each process has its own page table—400MB for 100 processes!

need to be smarter about storing the PT—will see the solution next time

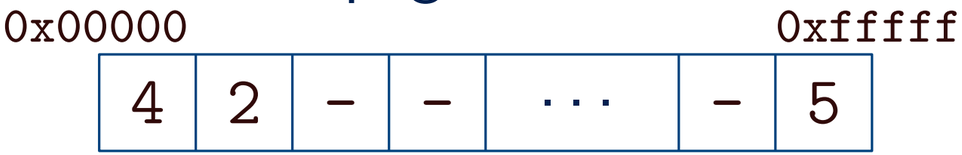
32-bit address space



single 32-bit address



page table



one million page translations in the array

Memory access overhead

- MMU translates the address as follows

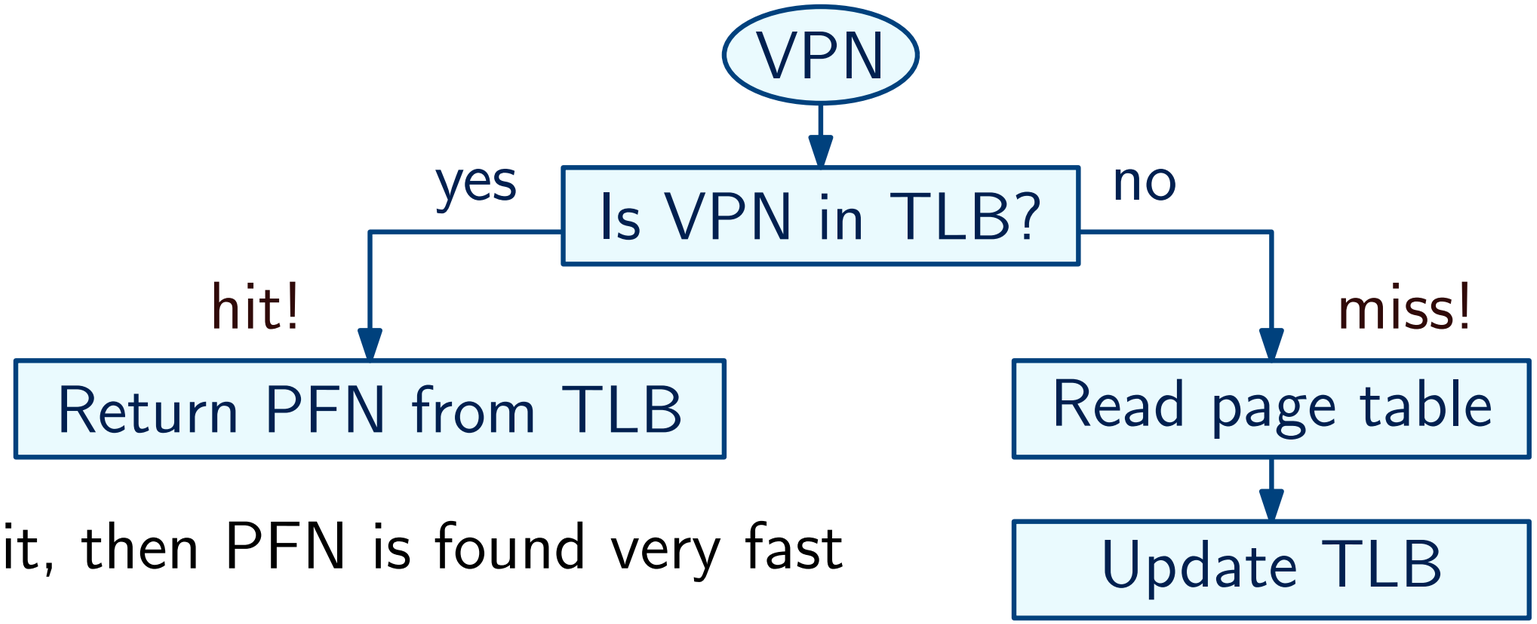
```
offset = virt_addr & OFFSET_MASK
VPN = (virt_addr & PAGE_MASK) >> PAGE_SHIFT
phys_addr = (PT[VPN] << PAGE_SHIFT) | offset
```

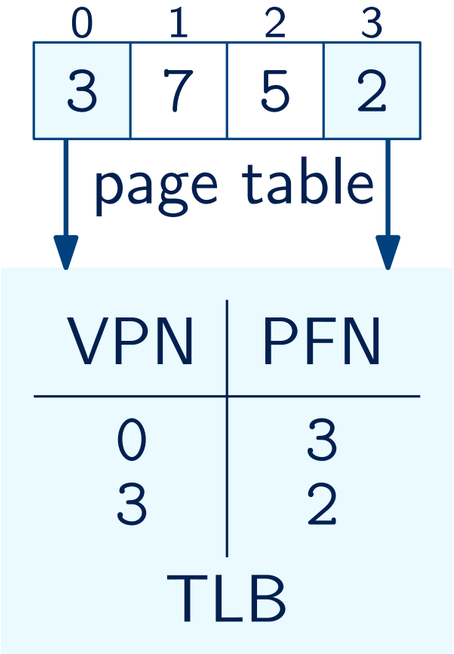
looking up PFN of the given page

- Every memory access now spawns another memory access—to the page table
- Memory access is already 20x–200x slower than referencing caches/registers, now another factor of 2!
- Too many pages to fit into cache/registers
- Maybe at least cache some VPN-PFN pairs?

Translation-lookaside buffer (TLB)

- MMU keeps a small cache of page-table lookups
- To get the PFN from a given VPN:





- If hit, then PFN is found very fast
- If miss, we actually pay the cost of extra memory access
  But next operations with the same VPN will be hits

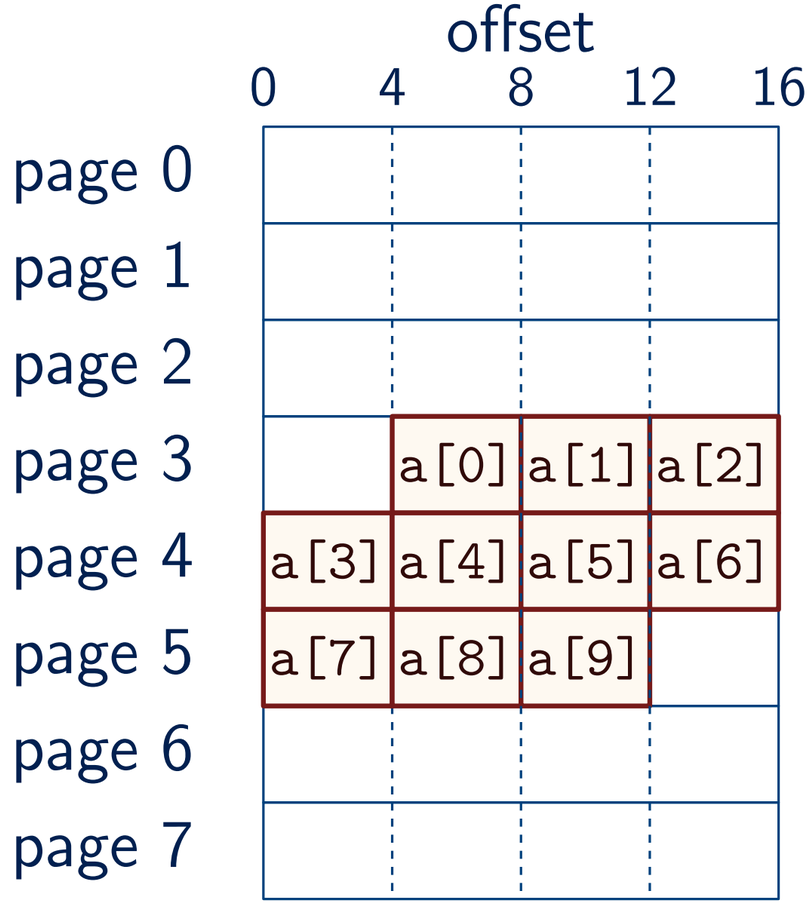
Why TLB helps

- Example: 16-byte pages

```
int i, sum = 0;
for (i = 0; i < 10; i++) {
  sum += a[i];
}
```

VPN	PFN
3	7
4	9
5	8

TLB



i = 0: address 0110100 ↦ page 3 ↦ miss
i = 1: address 0111000 ↦ page 3 ↦ hit
i = 2: address 0111100 ↦ page 3 ↦ hit
i = 3: address 1000000 ↦ page 4 ↦ miss
i = 4: address 1000100 ↦ page 4 ↦ hit
i = 5: address 1001000 ↦ page 4 ↦ hit
...

- 3 misses, 7 hits: 30% miss ratio
- Much better in real life: traversing an array with 4KB page → 1 miss for 1023 hits!

Typical TLB

- A few thousand entries
  - Each entry covers one 4KB page
  - So millions of addresses are cached at all times
- Extremely fast VPN lookup
  - Hit: 0.5–1 cycles
  - Miss: 10–100 cycles
- Small miss rate
  - 0.01%–1% "on average"
  - 20%–40% for sparse applications

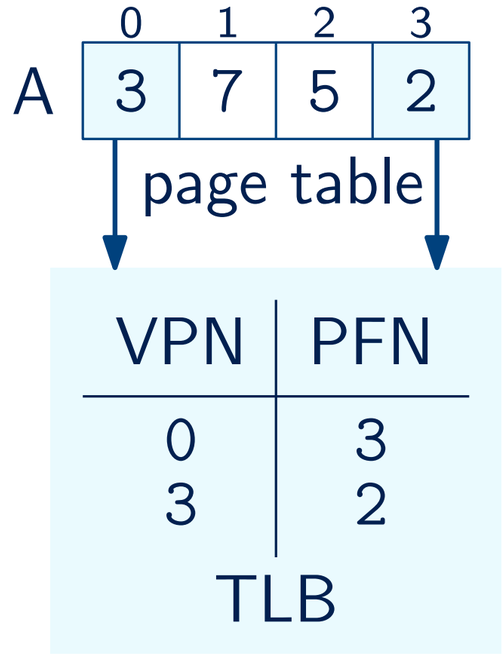
Example:
Assume hit is 1 cycle,
miss is 30 cycles,
miss rate is 1%.
Average TLB overhead is then
$1 + 30 \cdot 0.01 = 1.30$ cycles per query

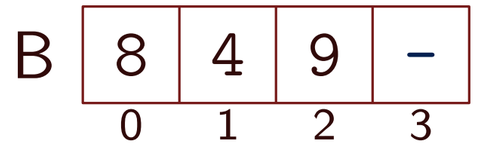
- Better locality in your program—fewer misses!
  - Program only uses few MBs: all its pages numbers will be cached
  - Accessing an array consecutively—always great
  - Array by random indices—also perfect hits if fits in a few MBs

TLBs and context switches

- When another process takes over, old TLB is not helpful

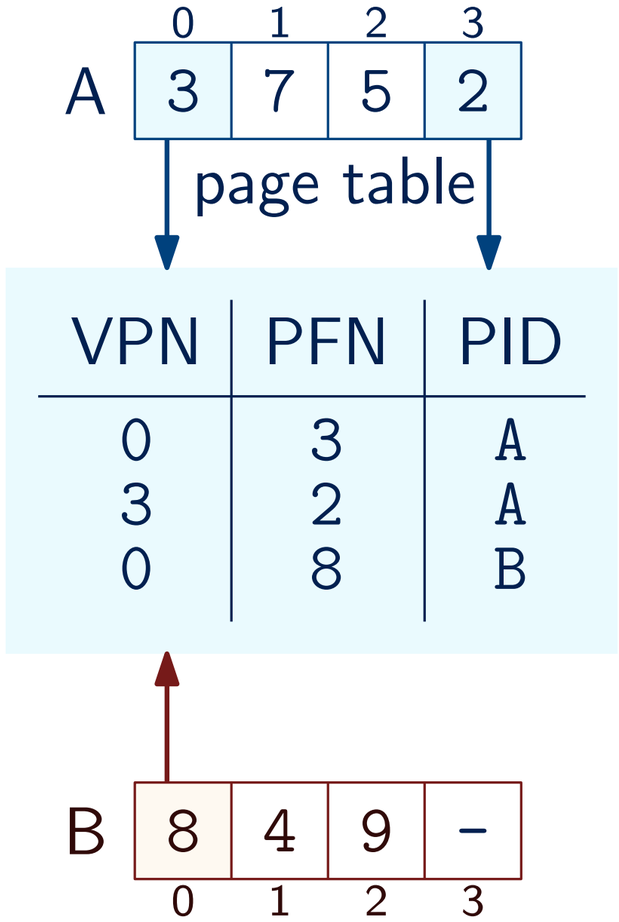
  B accesses a value on page 0
  → page 0 is converted by TLB to frame 3
  → frame 3 is out-of-bounds for B
  → crash!





TLBs and context switches

- When another process takes over, old TLB is not helpful
- It makes little sense to save TLB and then restore it
  - Just letting it fill naturally will have the same cost
- Simple solution is to clear TLB on every context switch
  - Nanoseconds vs mIlliseconds, so not too bad
- Many systems support process ID in TLB
  - Then entries for A can be safely left
  - Either they will be eventually overwritten
  - Or process A is restored quickly and most of its TLB survives—win!



Which pages to cache?

When a new entry comes, how to decide which old one to overwrite?

- Has to be a simple strategy
  - Fast
  - No information about the future
  - Very limited space to store past data
- **Example:** Least recently used—kick the entry that stayed unused for too long
- **Example:** Random—kick a random entry
- Similar question for swapping policies
  Which memory pages can be safely stowed on the disk?

Summary

- Paging discretizes the address space into small same-sized chunks
- This achieves great flexibility in memory usage
  $\rightarrow$ and little fragmentation
- Also greatly simplifies free space management for the OS
- The overhead of paging has to be dealt with
  - Extra memory accesses—solved by TLB, caching translations
  - Too much memory for the page table—will see better ways of organizing the page table next time
- Take a look at homework after Chapters 18 and 19
- Mandatory Assignment 2 starts Friday!