COMP 4735: Operating Systems Concepts

Lesson 11: Virtual Memory

Rob Neilson

rneilson@bcit.ca

Administrative stuff ...

- Reading: Virtual Memory is covered in chapters 3.3, 3.4
- Next quiz: Monday March 30th
  - covers chapters 3.1, 3.2, 3.3, 3.4 plus all stuff from lecture

Acknowledgements ...

- *some of the slides in this lesson are courtesy of Jonathan Walpole, who teaches at Portland State University*

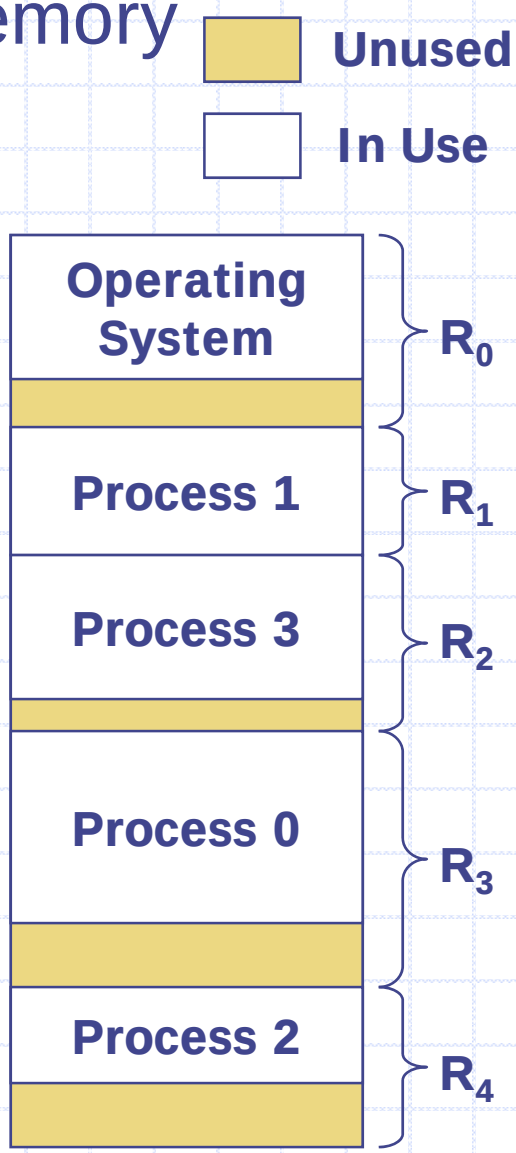
Today's Topics

- Introduction to Paging Systems
- Virtual Address Translation

Virtual Memory

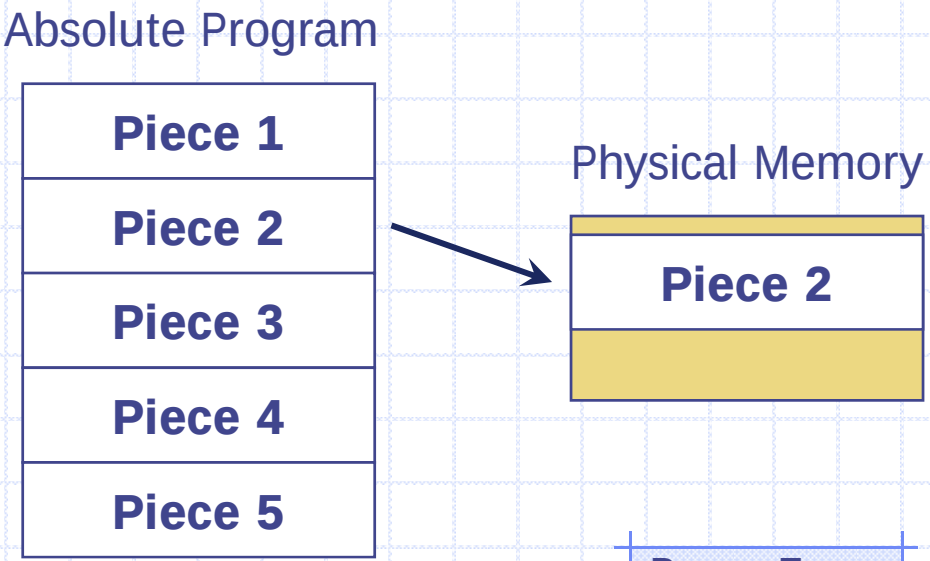
Multiprogramming in Partitioned Memory

- Recall the partitioned memory model from last lesson.
- Processes are mapped into partitions or free space in memory.
- This model assumes process size is smaller than partition size.
- What happens if process size >> partition size?
  - can't run
  - need to wait for another process to finish
- What happens if process size >> physical memory size?
  - can't run, ever
  - need to split the program and run each piece separately



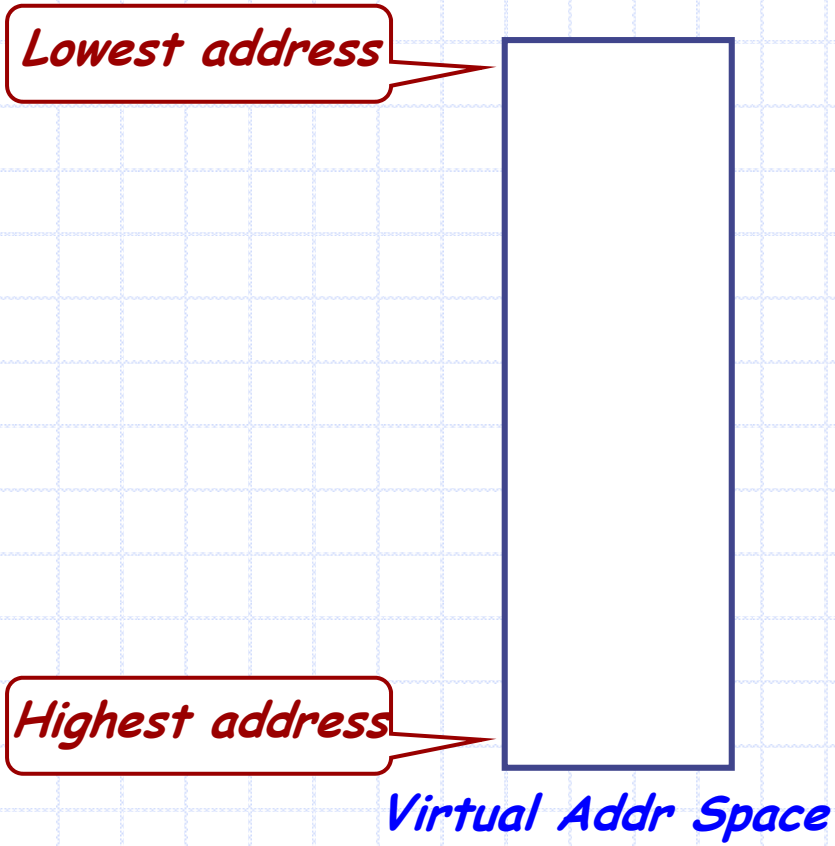
Virtual Memory (General Idea)

- Assume the process size is greater than memory size.
- This means we cannot keep the entire program in memory at once.
- We split the program into a bunch of pieces, and load the pieces on-demand (ie: when we need them).
- How can we do this?



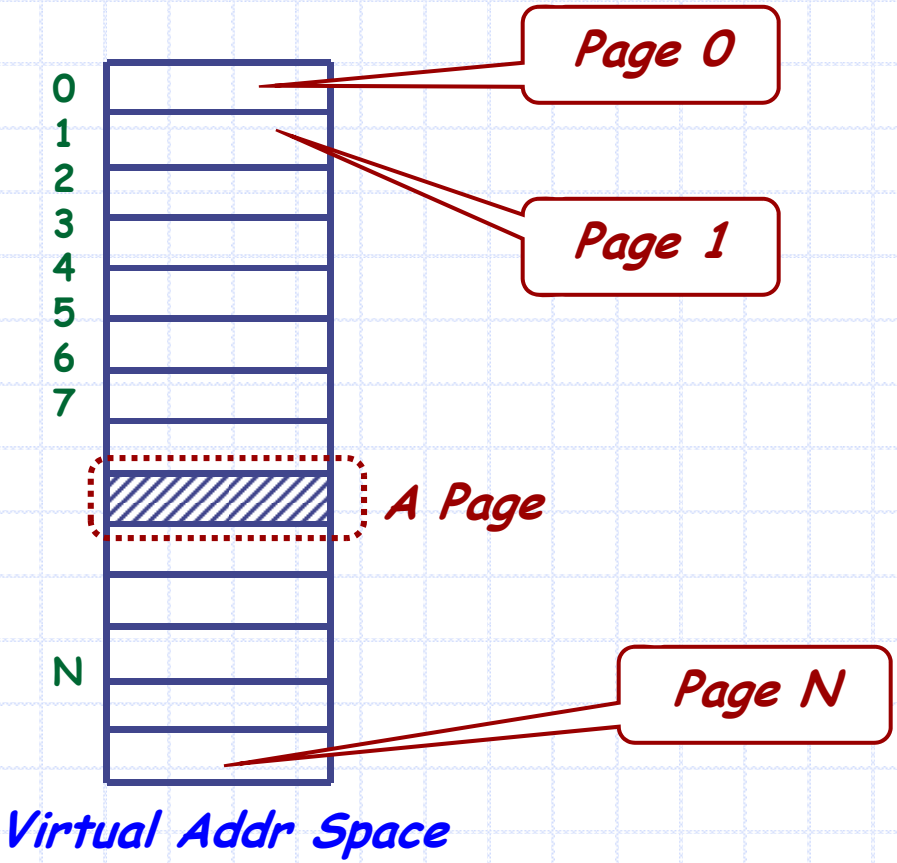
Virtual Address Spaces

- Here is the virtual address space
  - (as seen by the process)



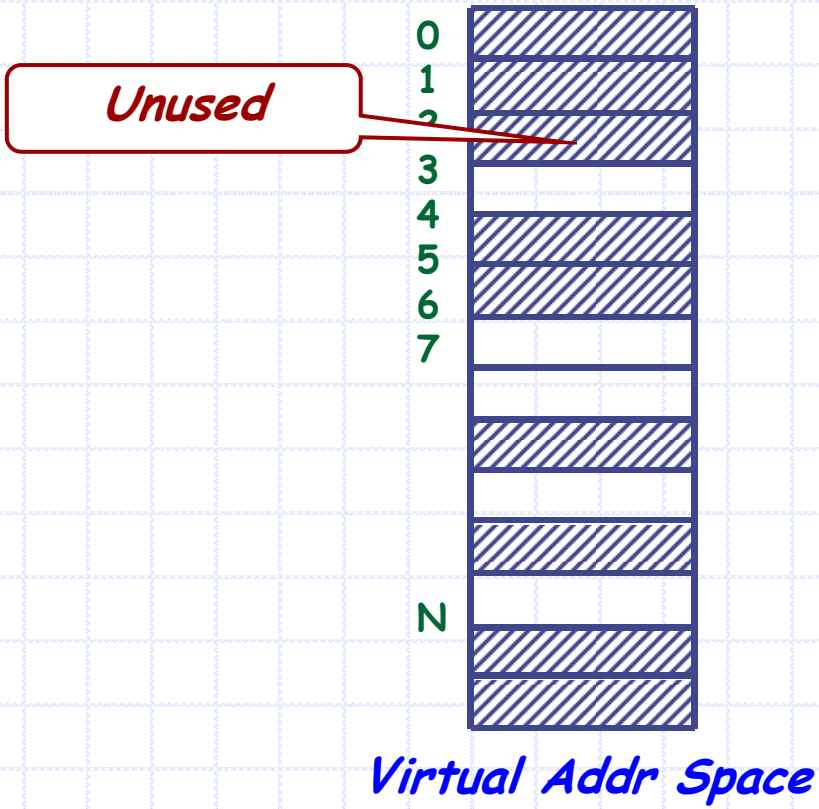
Virtual Address Spaces

- The address space is divided into “pages”
  - for example, on Pentium machines, pages are 4KB (2^{12})



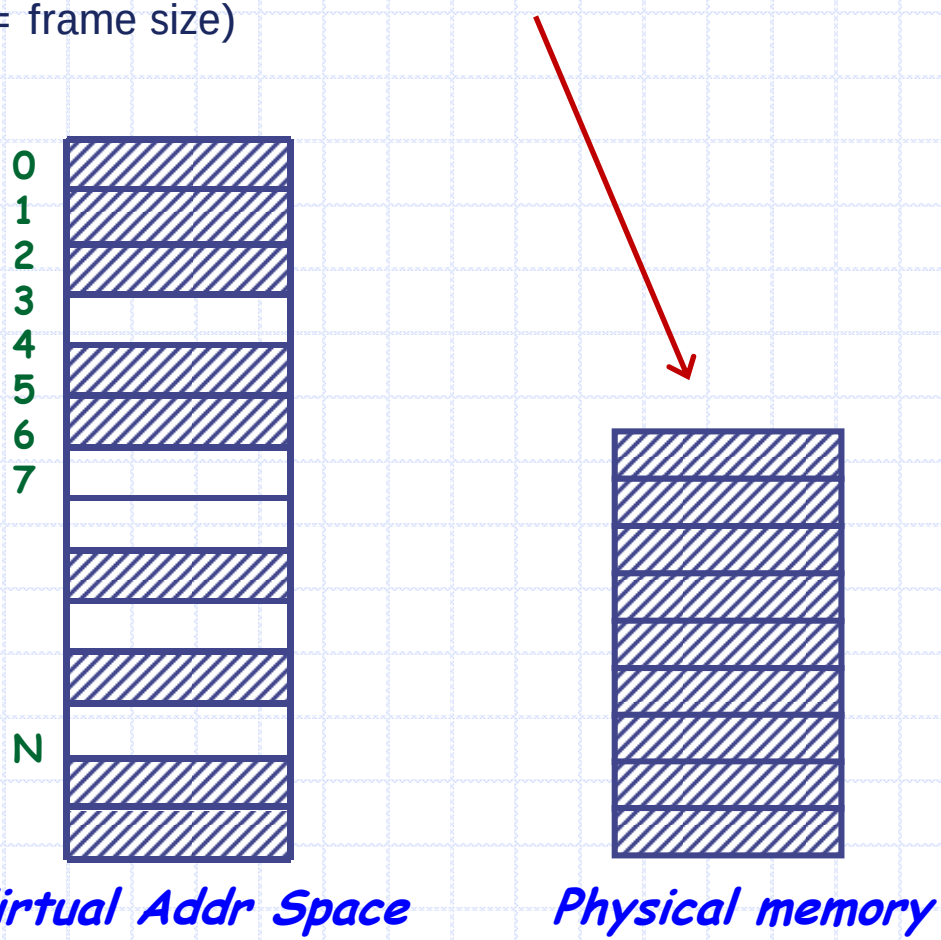
Virtual Address Spaces

- In reality, only some of the pages are used (as needed to load the text, data, stack segments for the process)



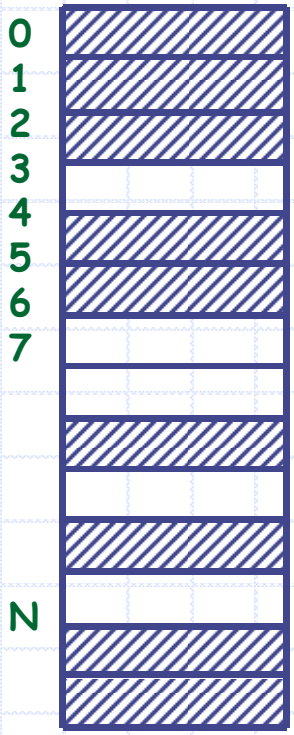
Physical Memory

- Physical memory is divided into "*page frames*"
  - (Page size == frame size)

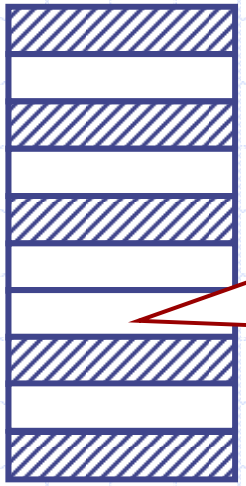


Virtual and Physical Address Spaces

- *Some page frames* are used to hold the pages of this process



Virtual Addr Space

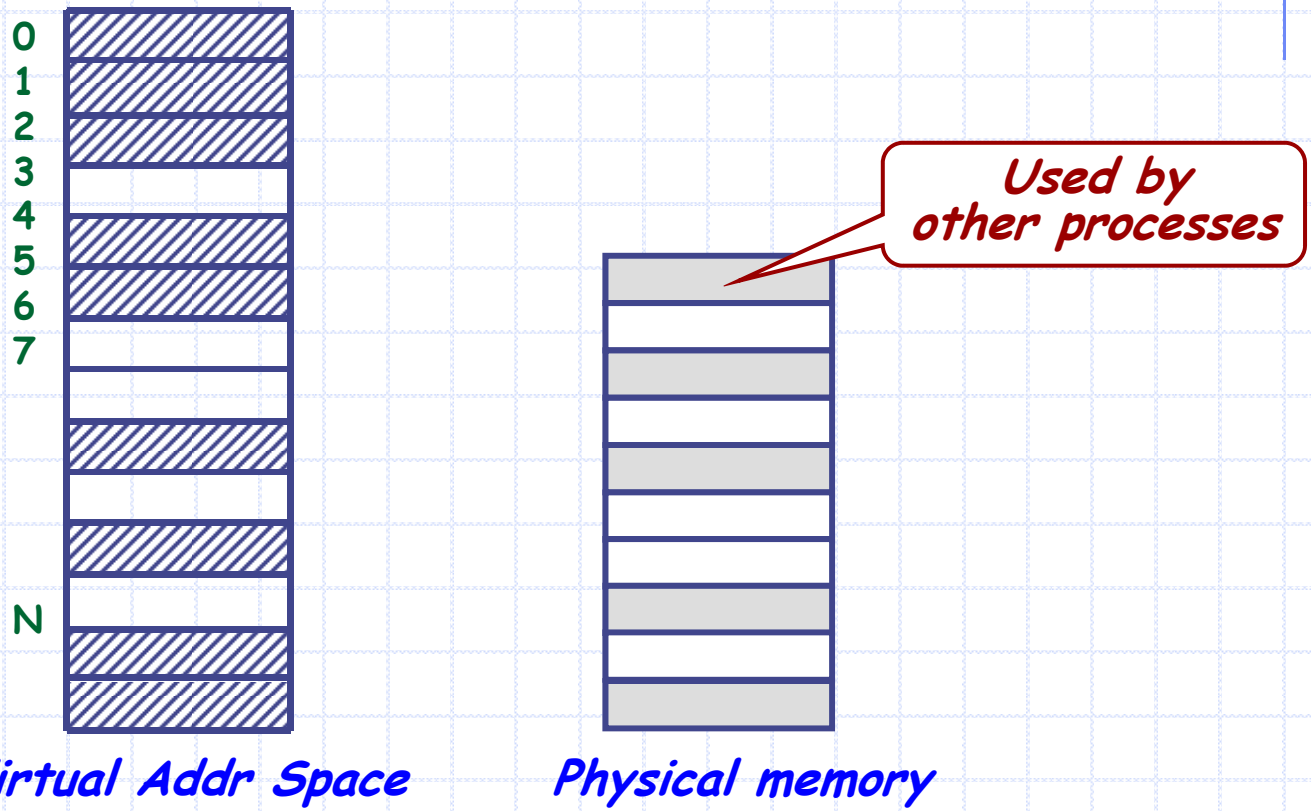


Physical memory

Assume these frames are used by this process

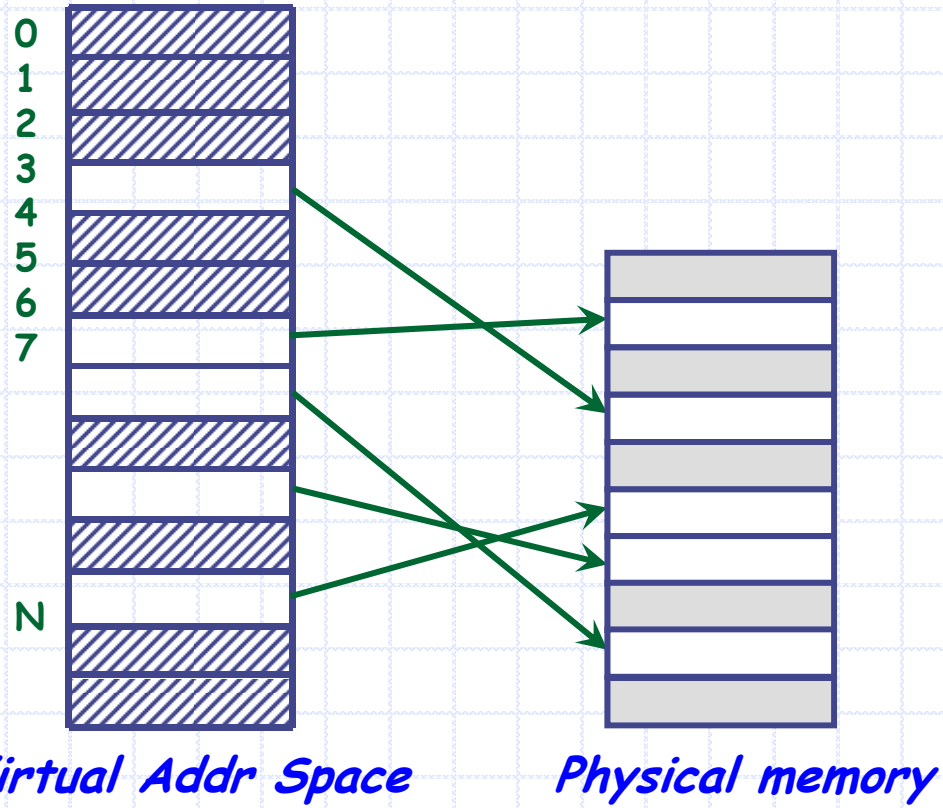
Virtual and Physical Address Spaces

- Some page frames are used for other processes



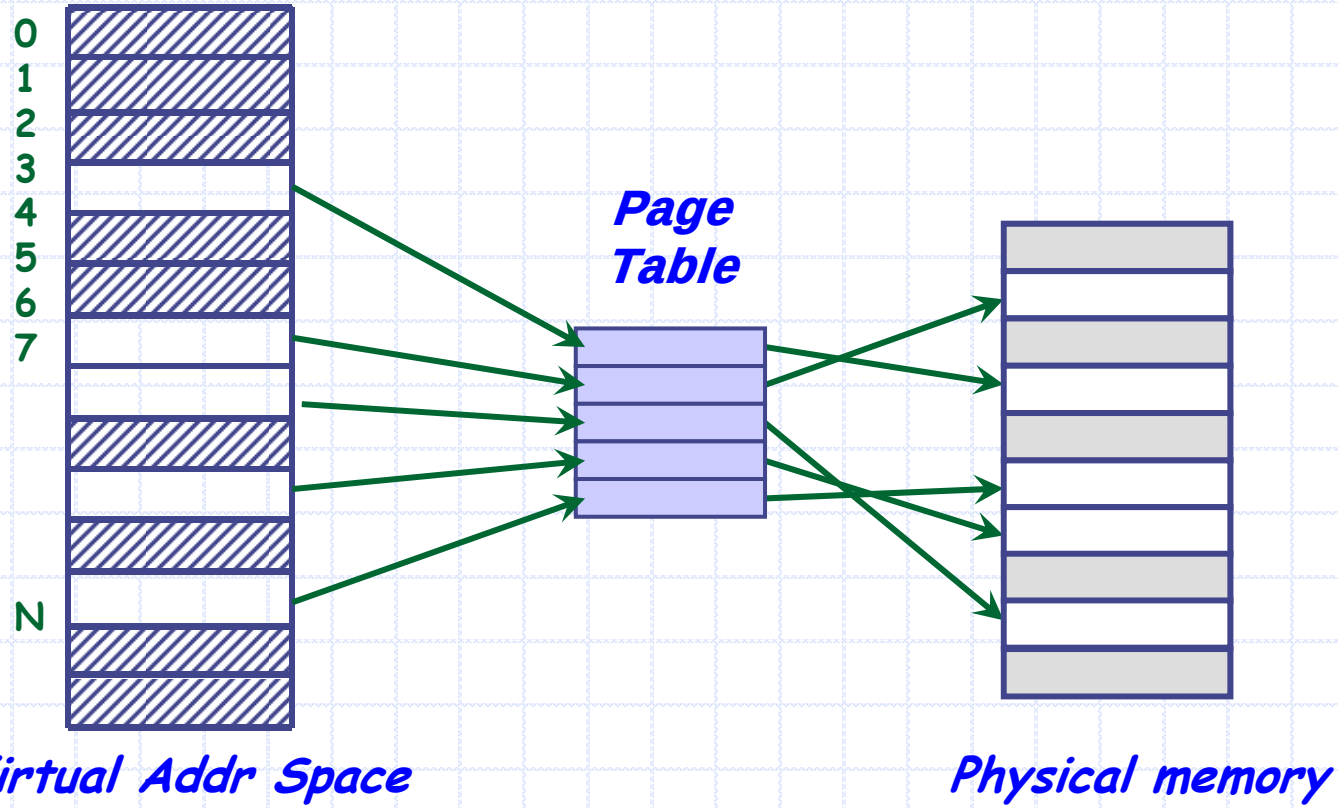
Virtual Address Spaces

- Address mappings tell the computer which frame has which page



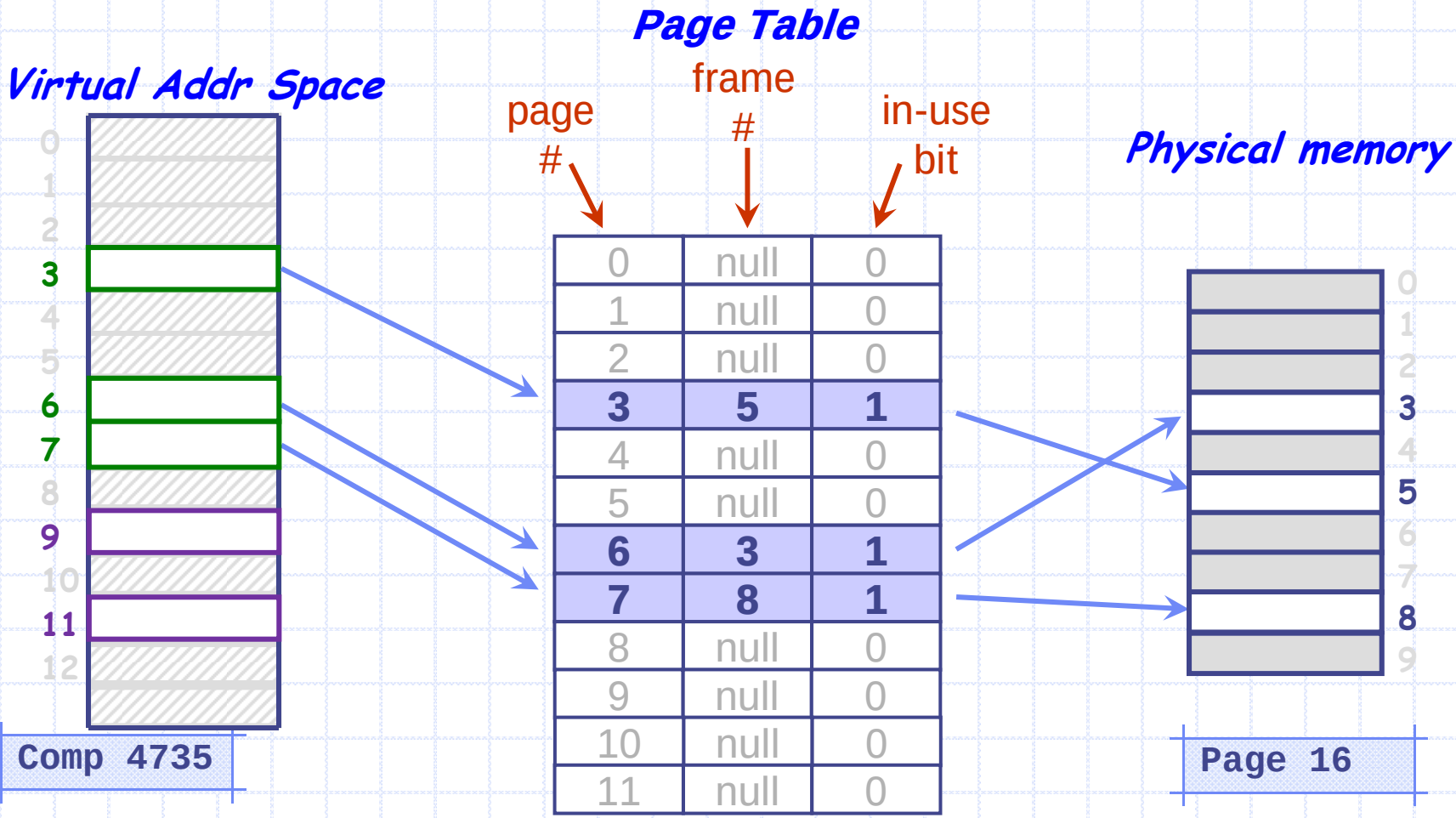
Page Table (1)

- Address mappings are actually stored in a *page table* in memory
- One page table entry per page...



Page Table (2)

- a page table has one reference for every page needed by the process
- some entries point to allocated frames in physical memory
- other entries are empty until they are referenced



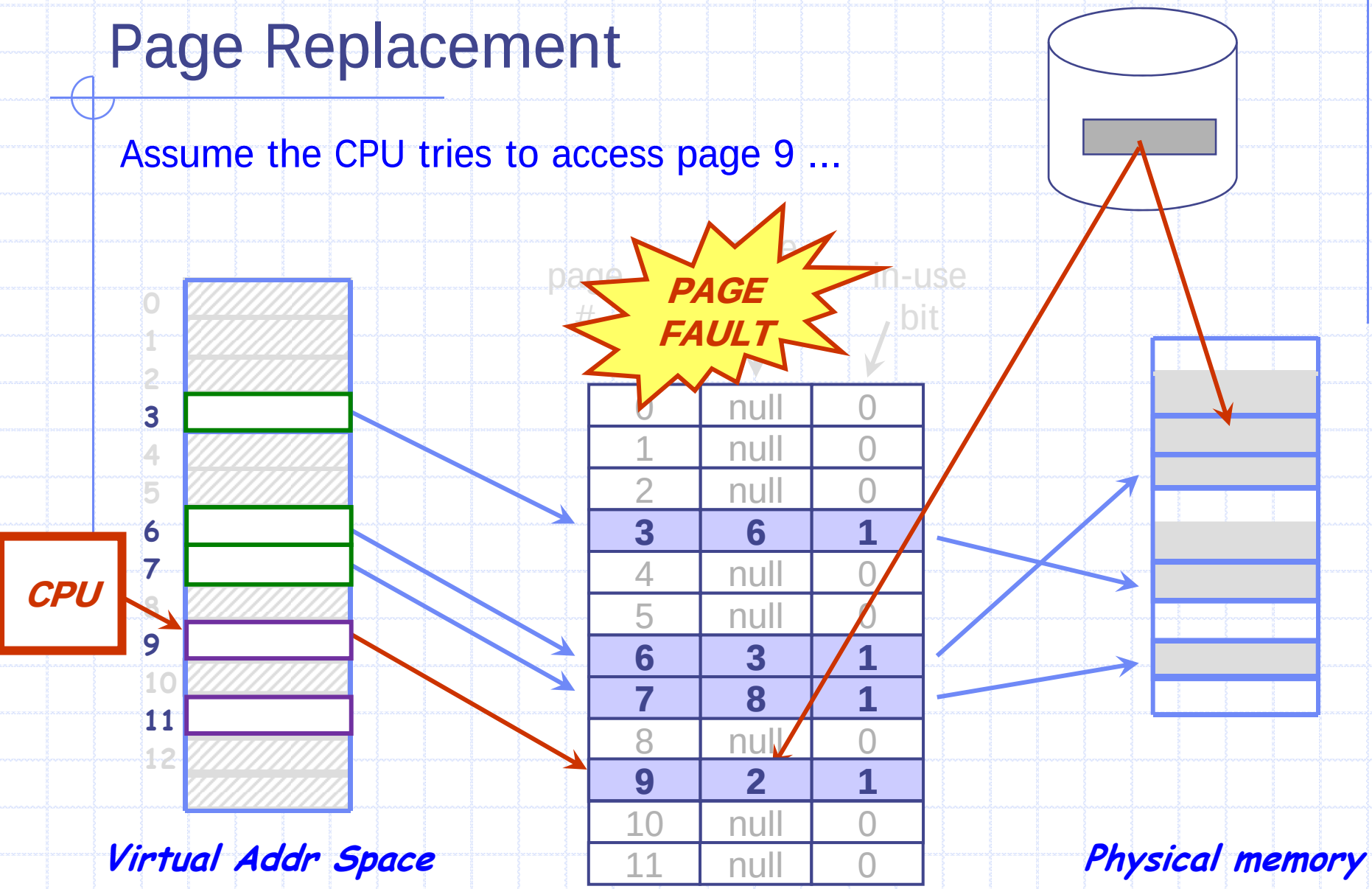
Page Faults (summary)

- What happens if a program references a memory location that is on a page that is not loaded?

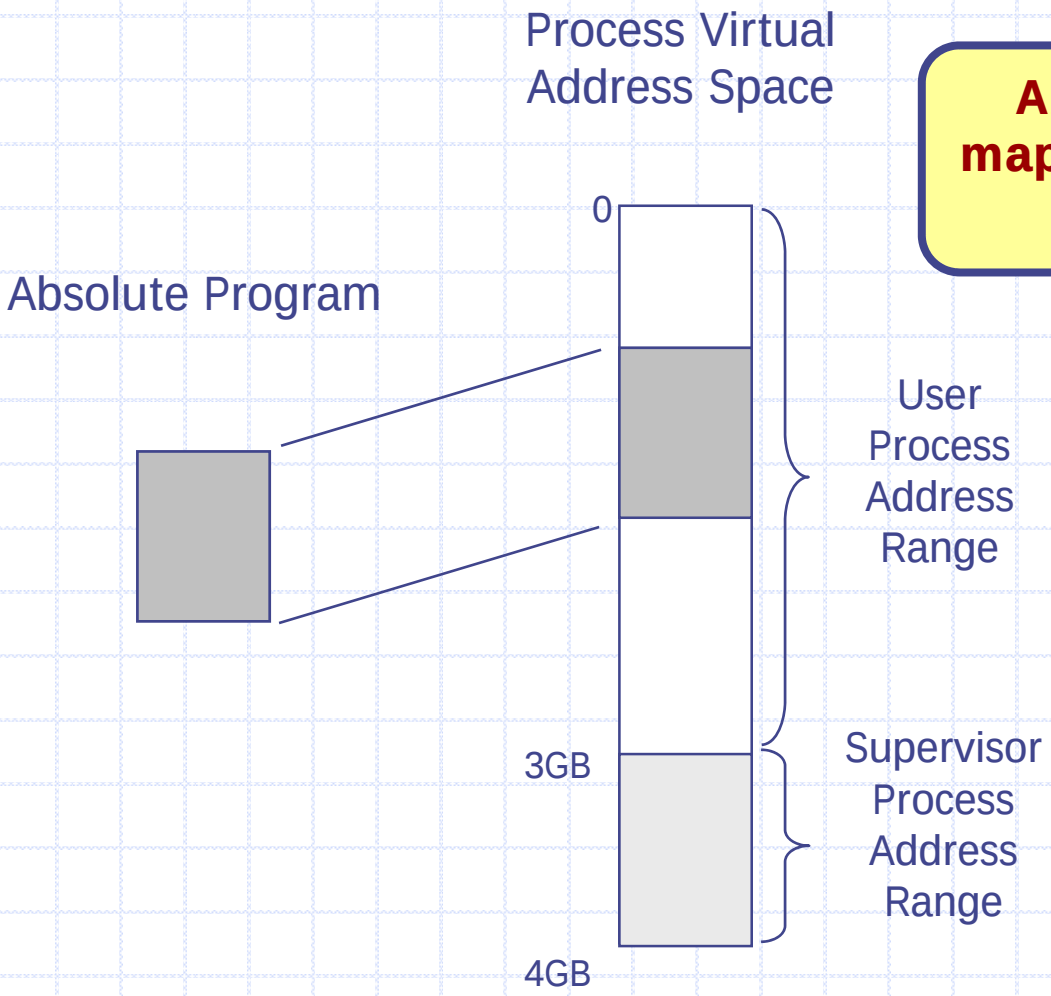
1. a "page fault" is generated, and the program traps to OS code
2. the process requesting the missing page is suspended
3. the memory manager locates the missing page on disk
4. the page is loaded into primary memory
   - if all pages in primary memory are full, a *page replacement algorithm* is run to decide which page should be moved out to disk
5. the page table in memory is updated
6. process execution is resumed
   - note: this may require restarting an instruction

Page Replacement

Assume the CPU tries to access page 9 ...

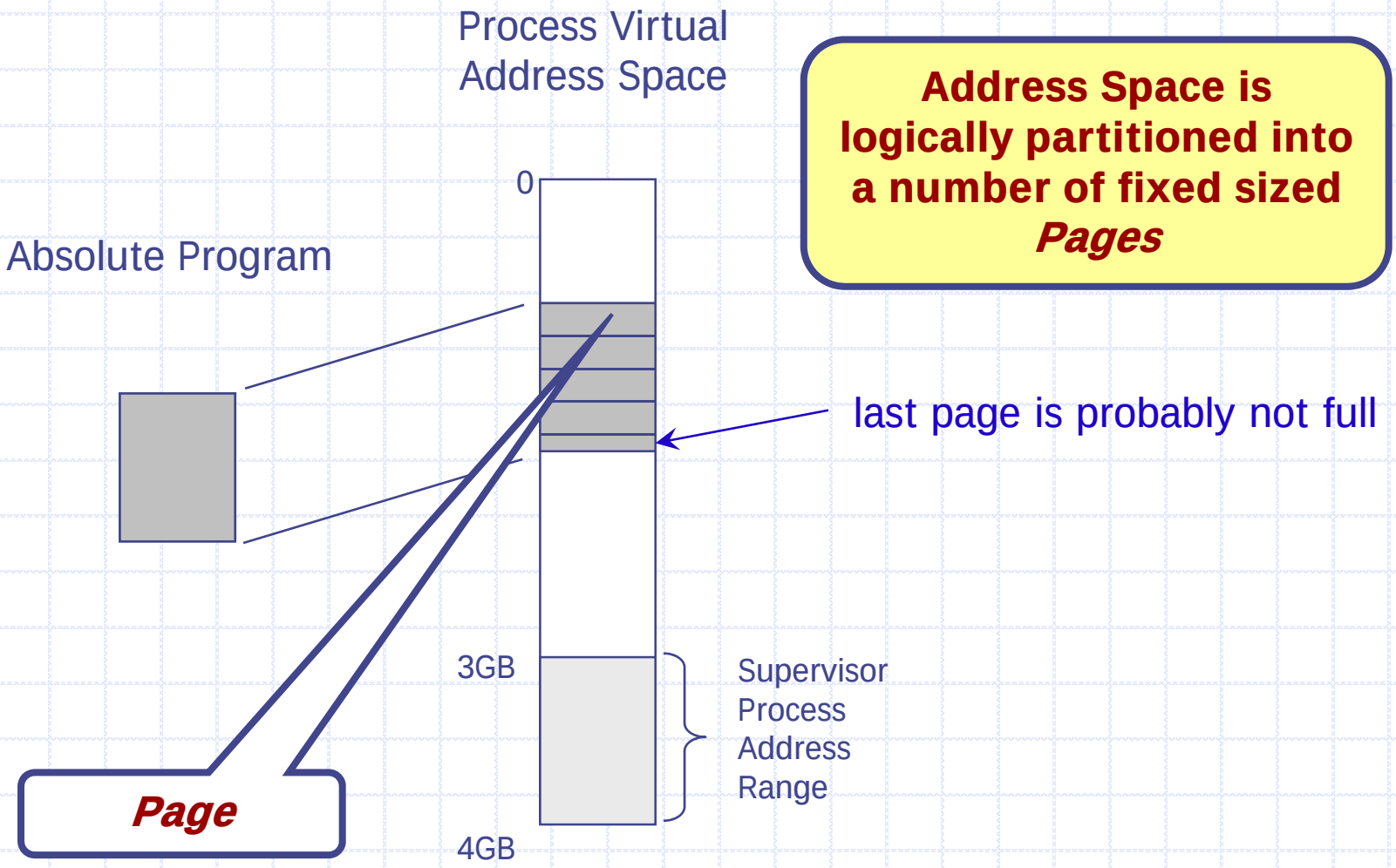


Virtual Address Space (revisited)



Absolute program is mapped into the address space

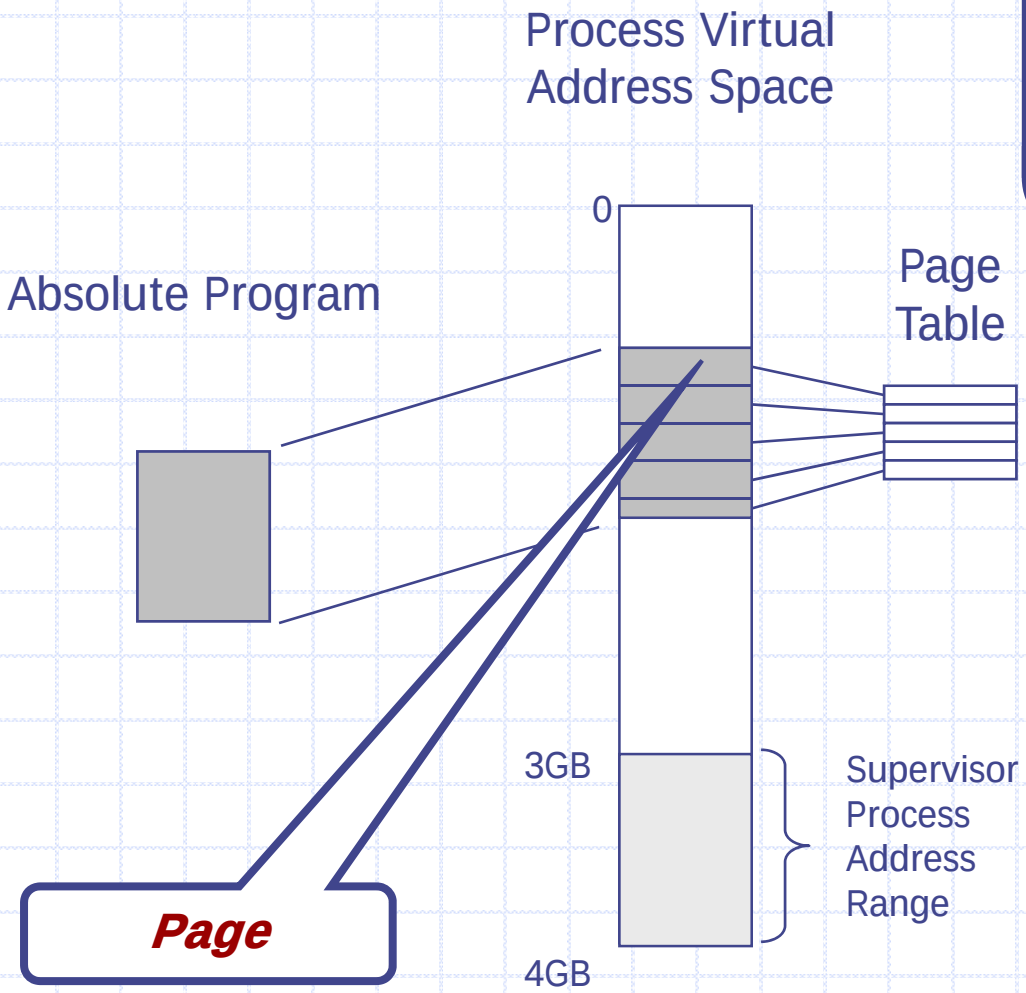
Virtual Address Space (2)



Address Space is logically partitioned into a number of fixed sized *Pages*

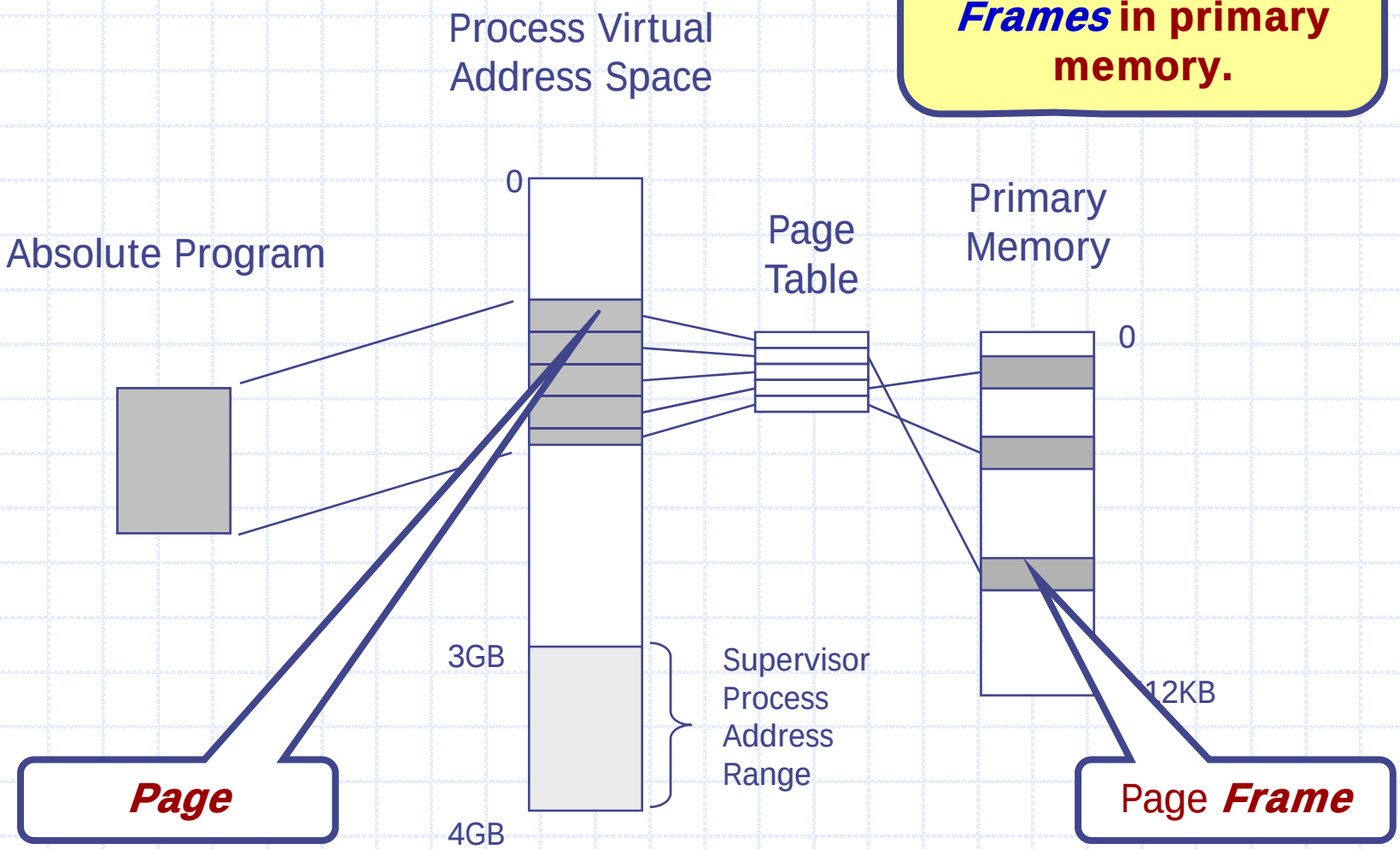
Virtual Address Space (3)

A *page table* is created for the process. It keeps track of the pages in the virtual address space.



Virtual Address Space (4)

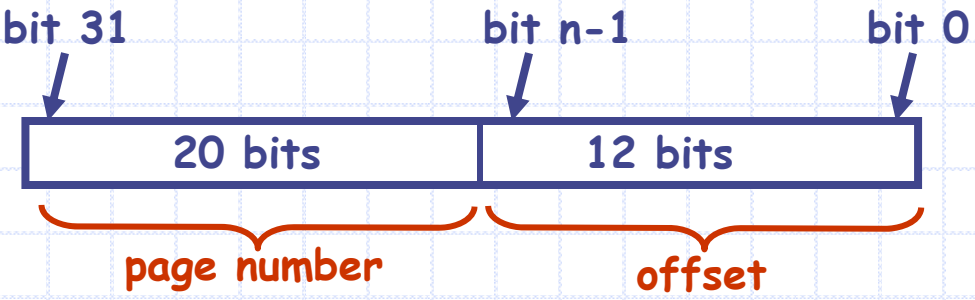
The page table maps the pages to *Page Frames* in primary memory.



Virtual Addresses

We need a way to create and address the pages in the virtual address space

- Virtual memory addresses (what the process uses)
  - Page number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page number



Example: 32 bit virtual address

Page size = 2^{12} = 4KB

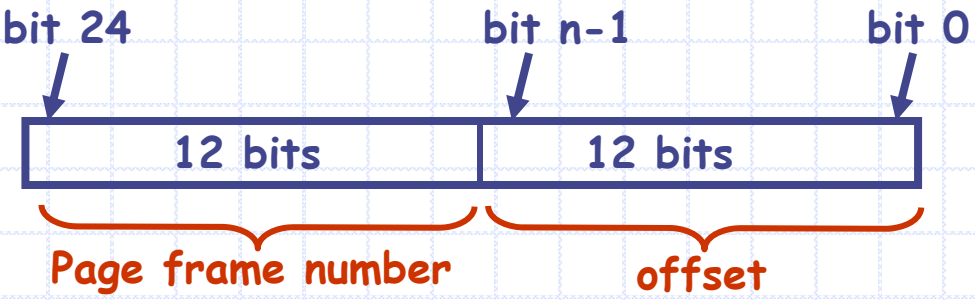
Address space size = 2^{32} bytes = 4GB

Number of pages in address space = 2^{20} = 1MB (1,048,576) pages

Physical Addresses

Similarly, we need a way to address the frames in physical memory

- Physical memory addresses (what the CPU uses)
  - Page frame number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page frame number



Example: 24 bit physical address

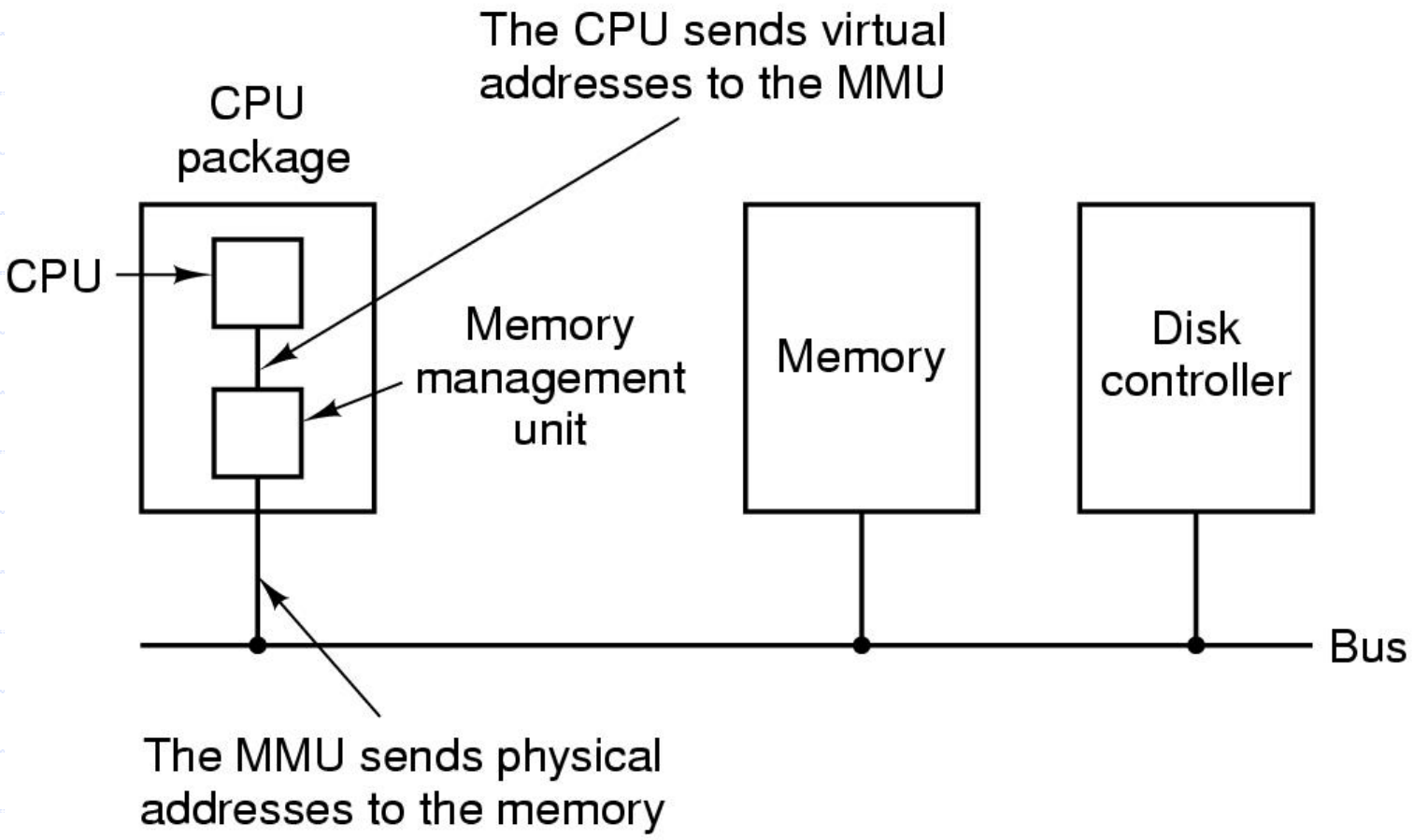
Page frame size = 2^{12} = 4KB

Max physical memory size = 2^{24} bytes = 16MB

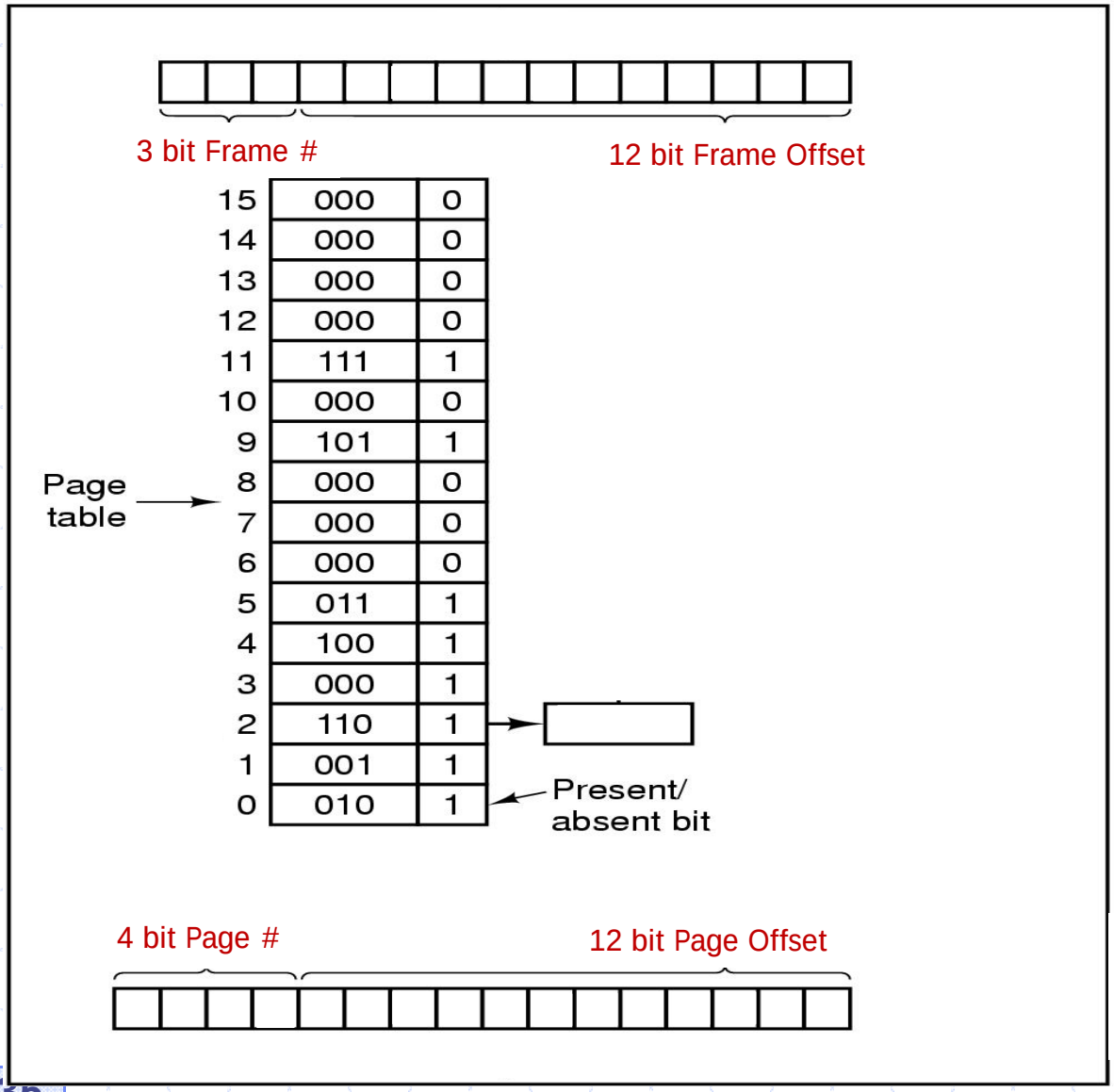
Number of page frames in physical mem = 2^{12} = 4KB (4096) frames

Virtual Address Translation

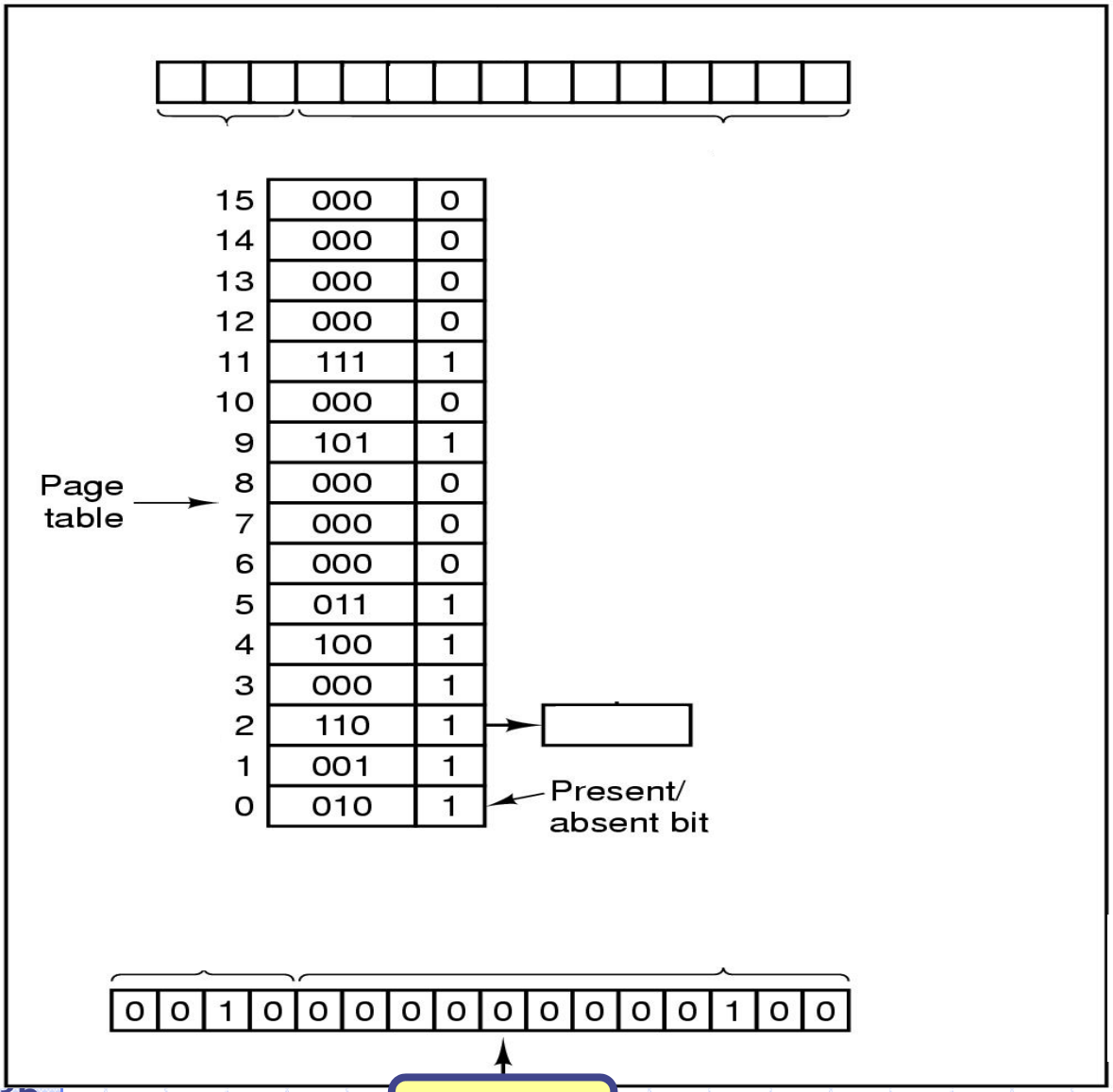
- Hardware (MMU) maps page numbers to page frame numbers



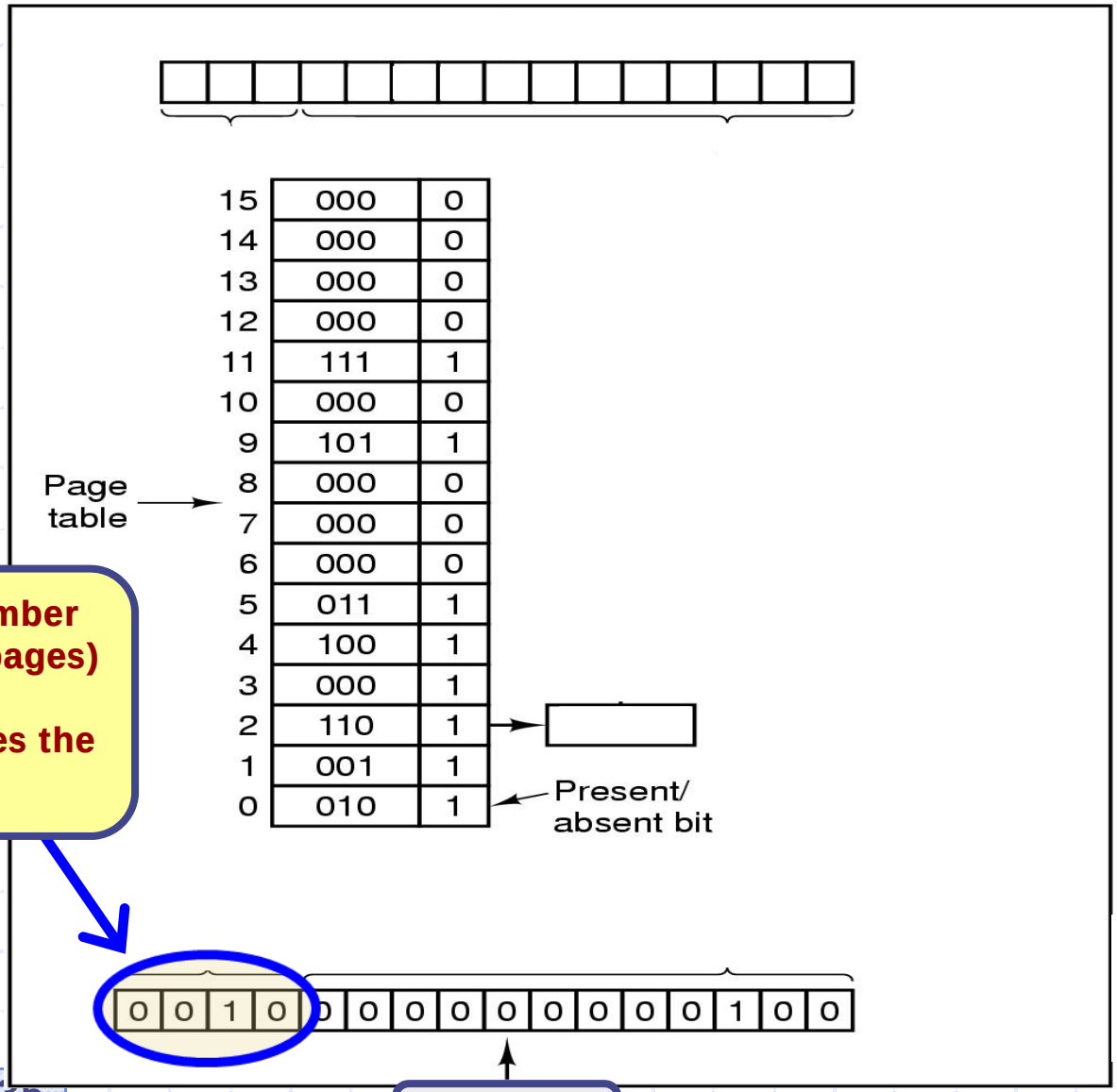
Example MMU Operation (1)



Example MMU Operation (2)

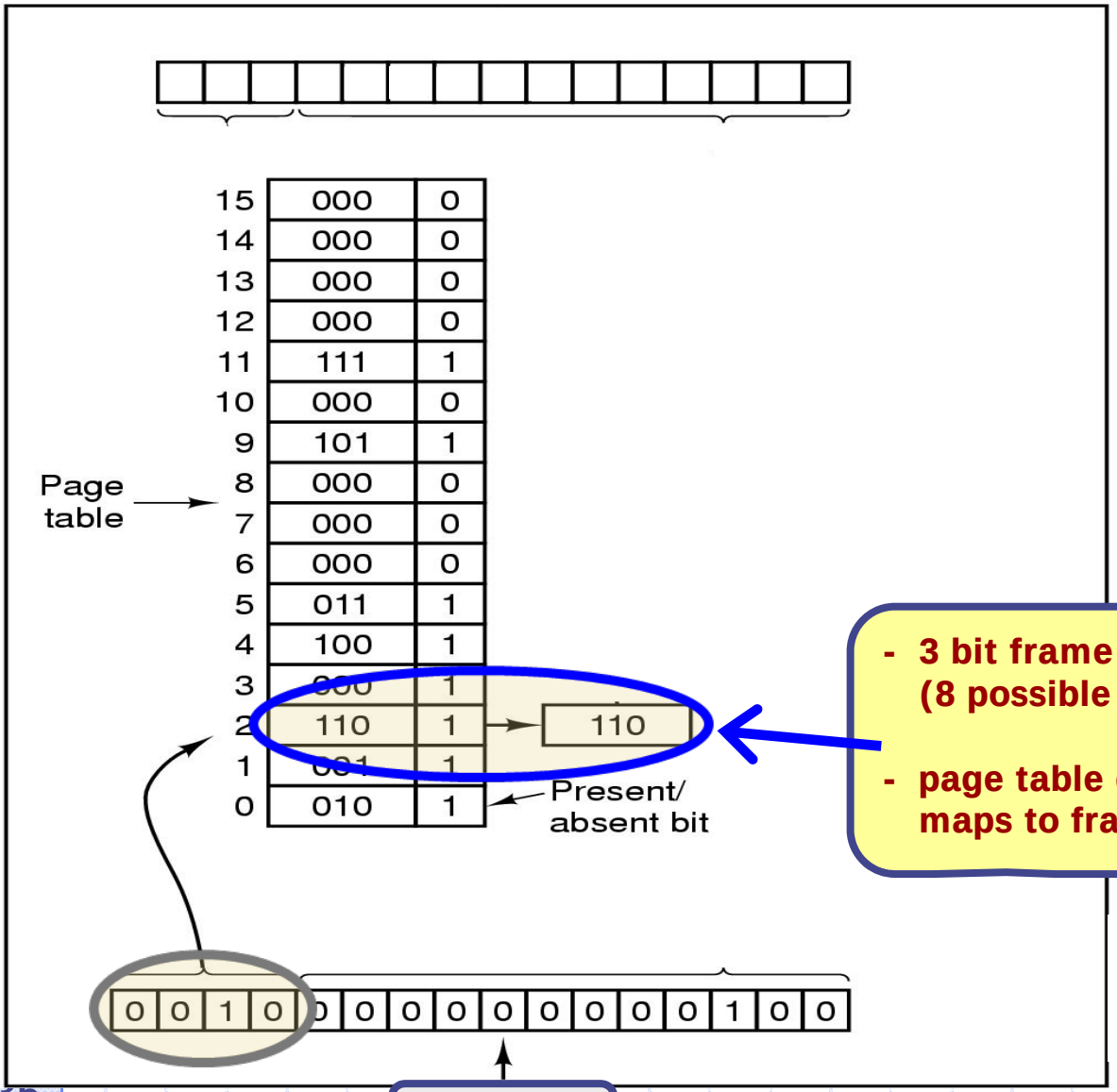


Example MMU Operation (3)



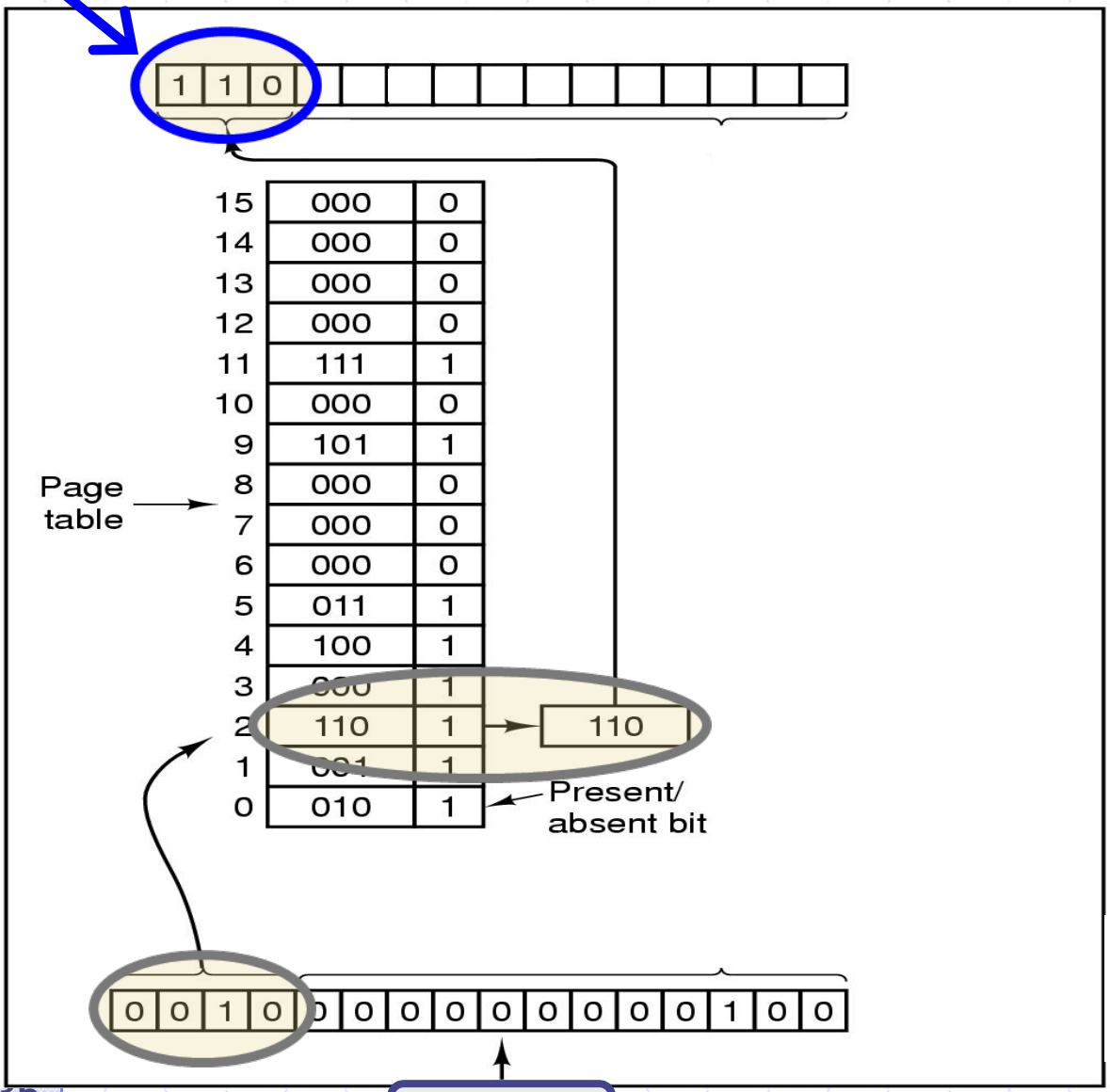
- 4 bit page number (16 possible pages)
- page # indexes the page table

Example MMU Operation (4)

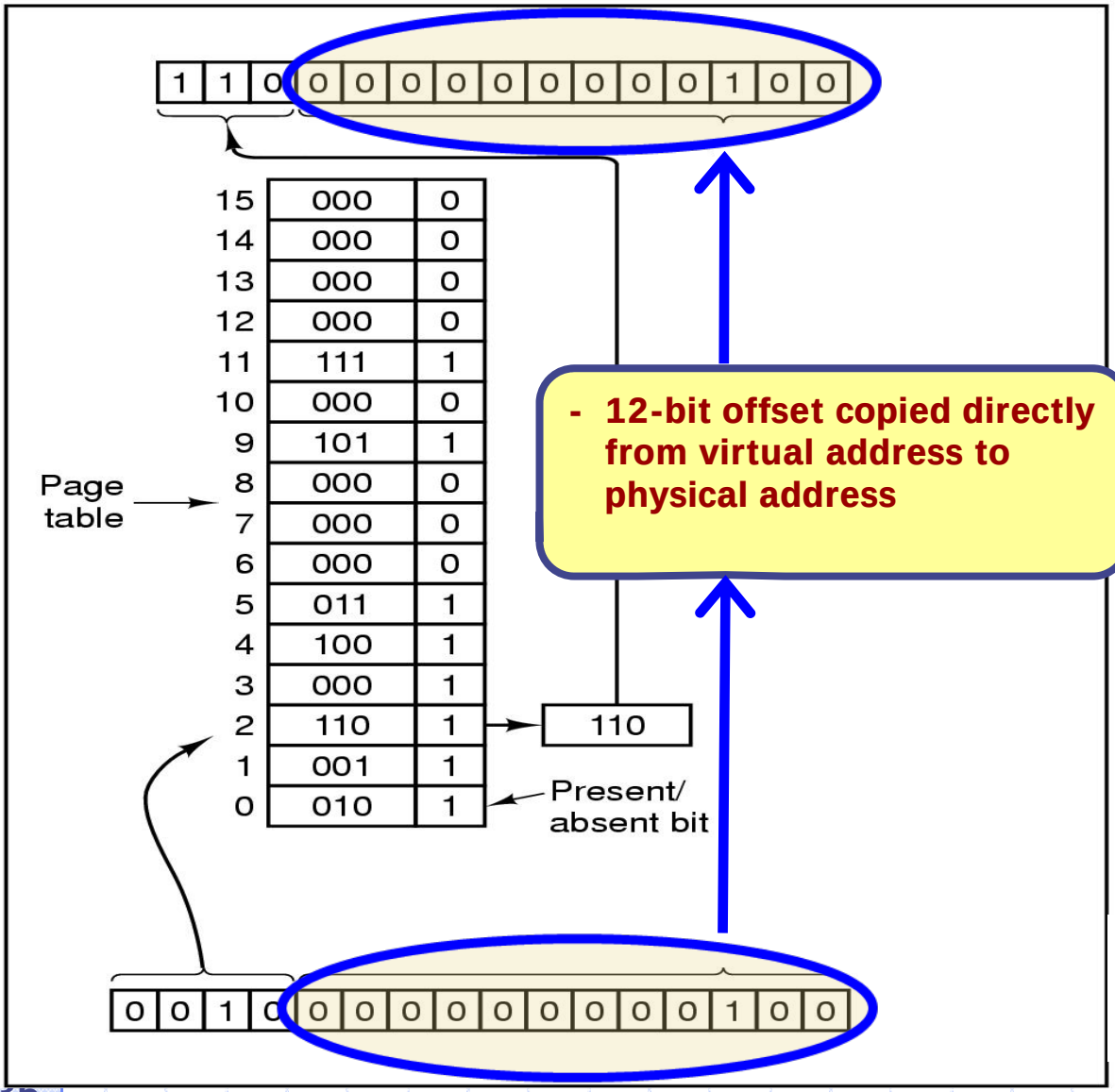


- page frame loaded into physical address

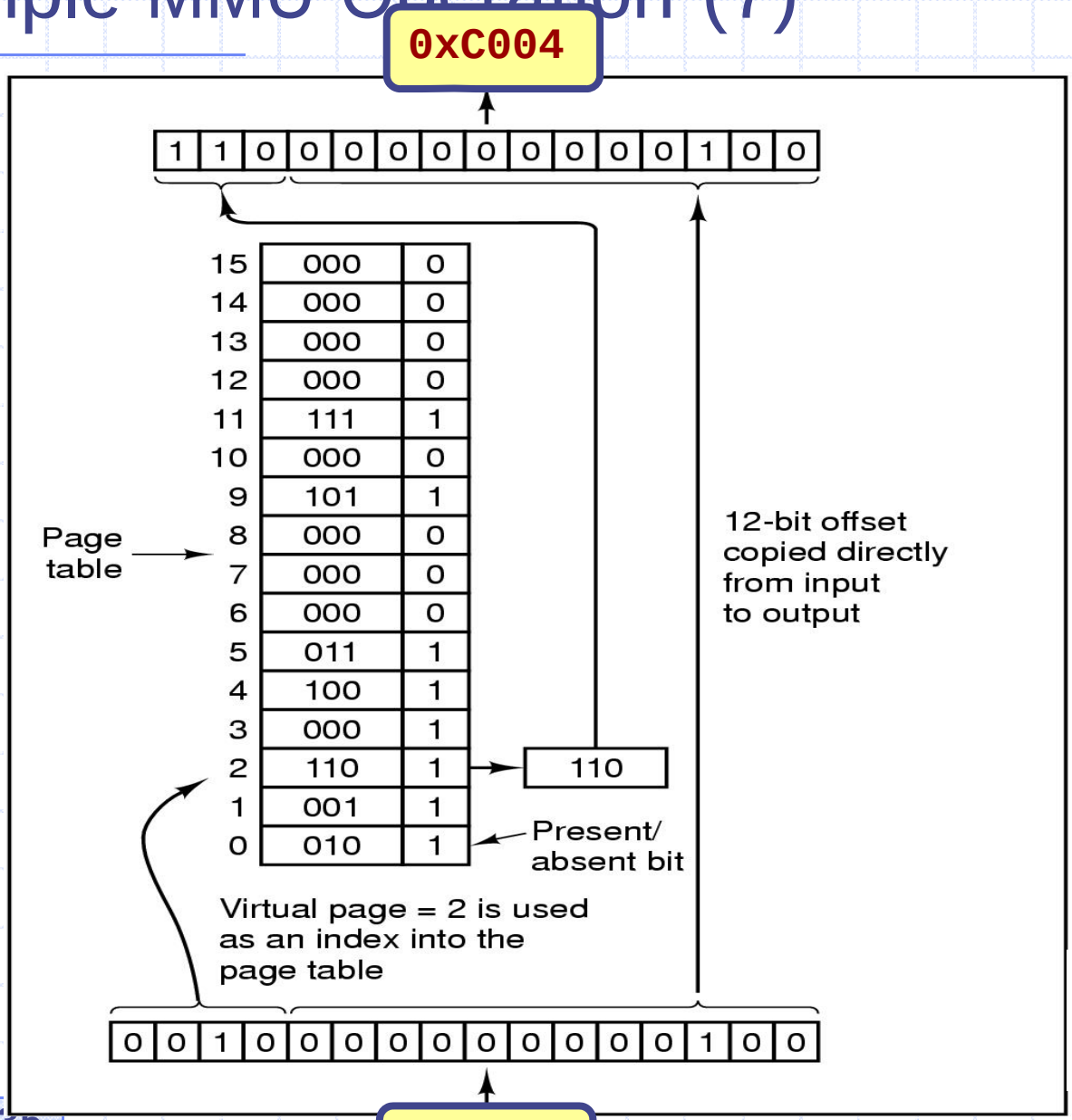
MMU Operation (5)



Example MMU Operation (6)



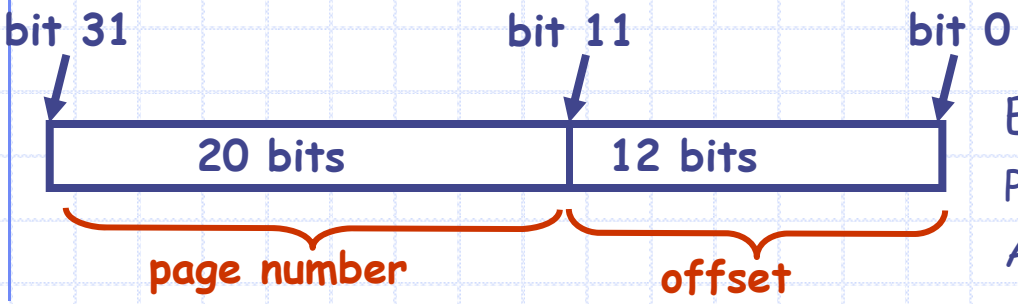
Example MMU Operation (7)



Address Translation (again... to make sure you get it!)

Problem: we have a page in the address space (virtual memory) that we want to locate in physical memory

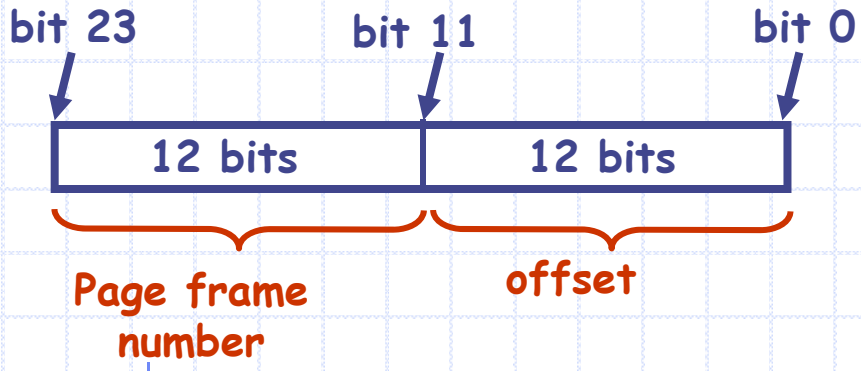
Solution: use a table lookup to map the virtual page number to the physical frame number



Example: 32 bit virtual address

Page size = 2^{12} = 4KB

Address space size = 2^{32} bytes = 4GB



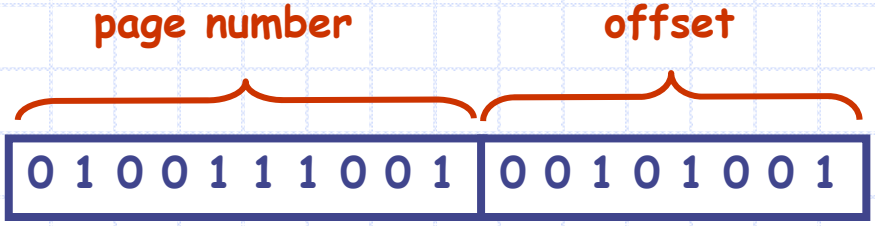
Example: 24 bit physical address

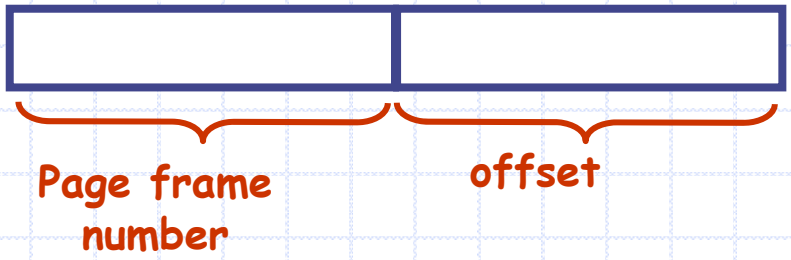
Frame size = 2^{12} = 4KB

Physical mem size = 2^{24} bytes = 16MB

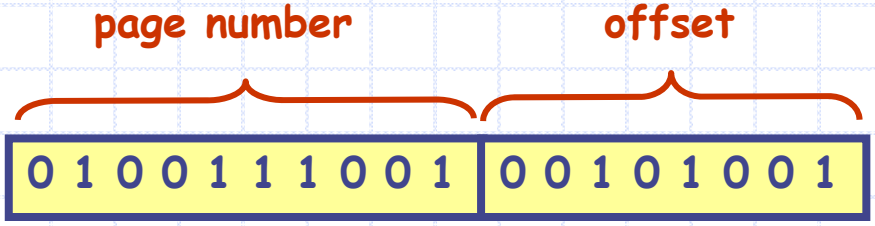
Address Translation Example -1

Example:



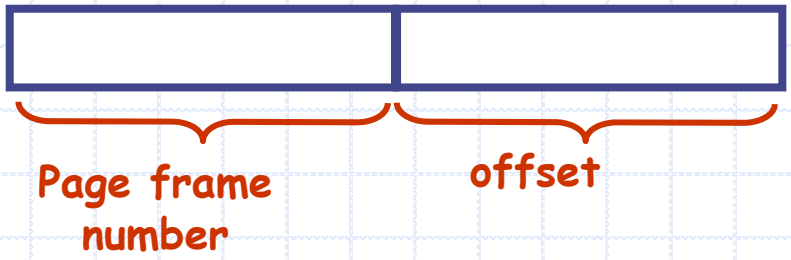


Address Translation Example -2

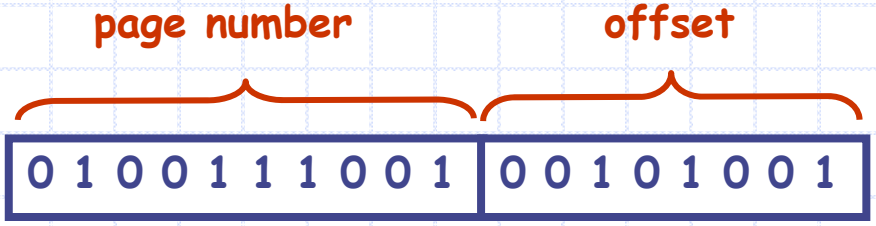


Example, assume...

- 18 bit virtual address

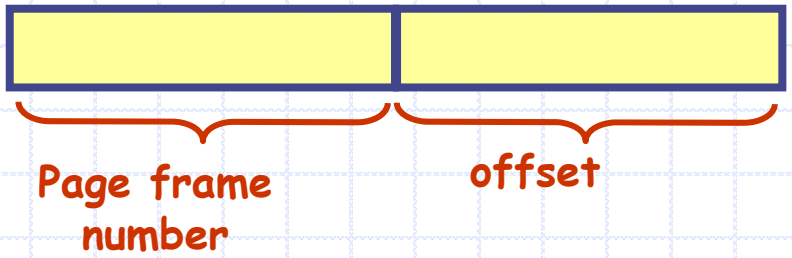


Address Translation Example -3

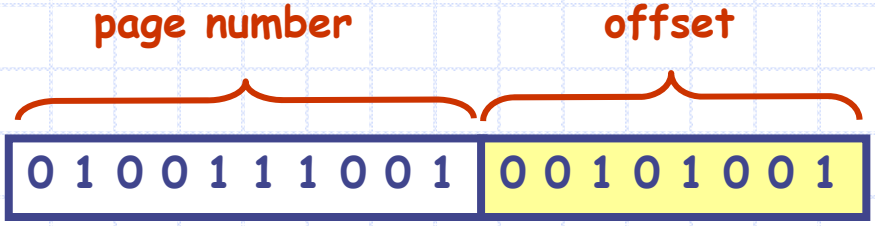


Example, assume...

- 18 bit virtual address
- 16 bit physical address

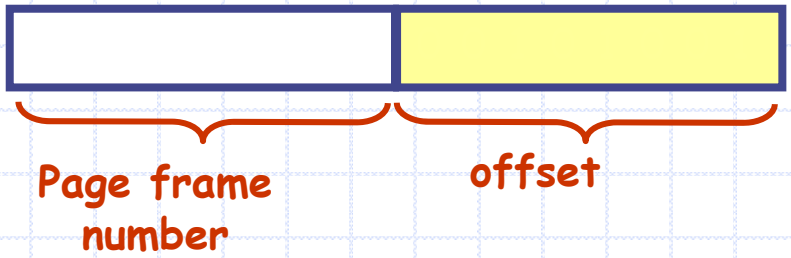


Address Translation Example -4

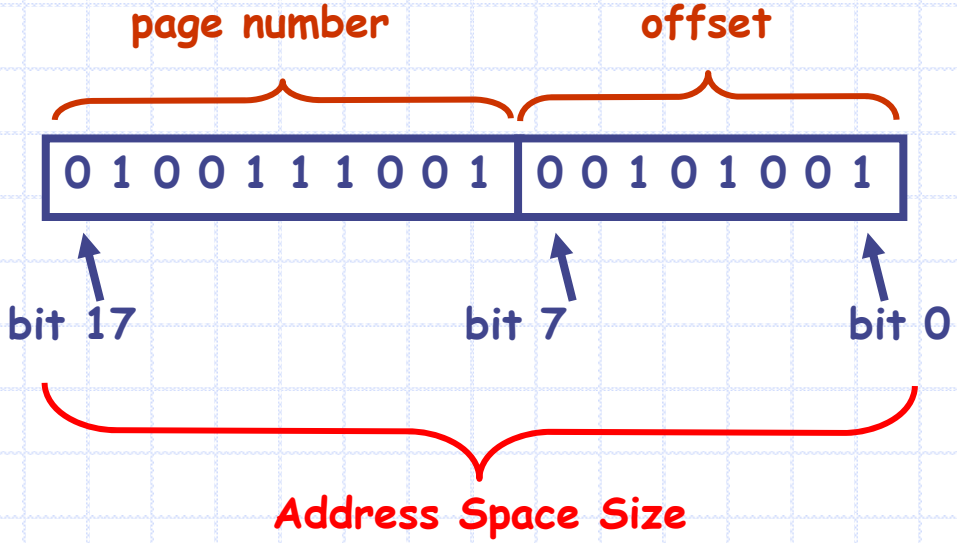


Example:

- 18 bit virtual address
- 16 bit physical address
- page size = 2^8 = 256 bytes

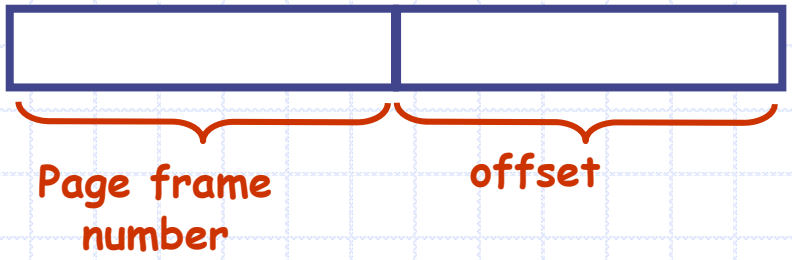


Address Translation Example -5

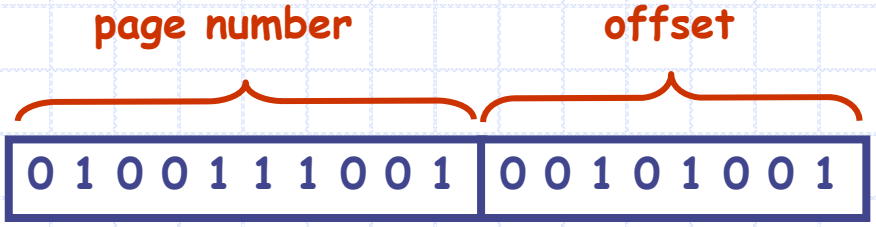


Example:

- 18 bit virtual address
- 16 bit physical address
- page size = 2^8 = 256 bytes
- address space size = 2^{18}
  = 256KB

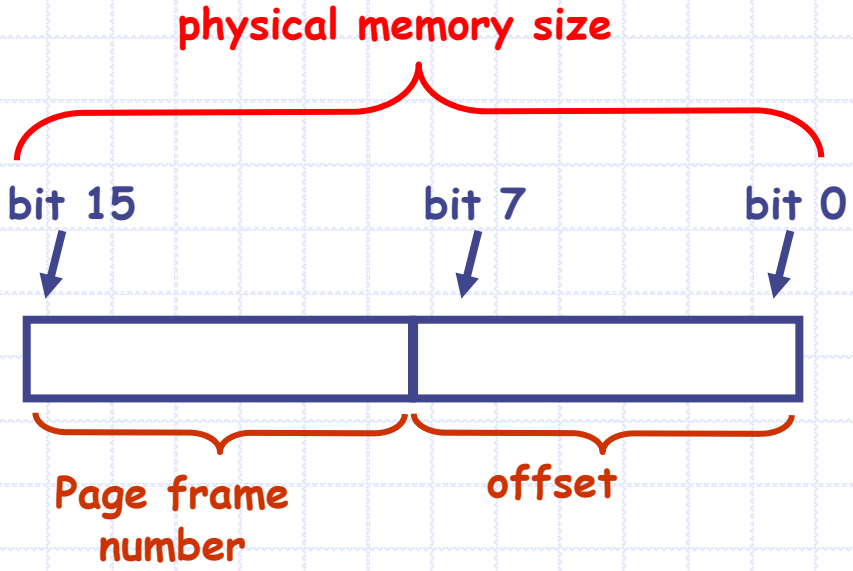


Address Translation Example -6

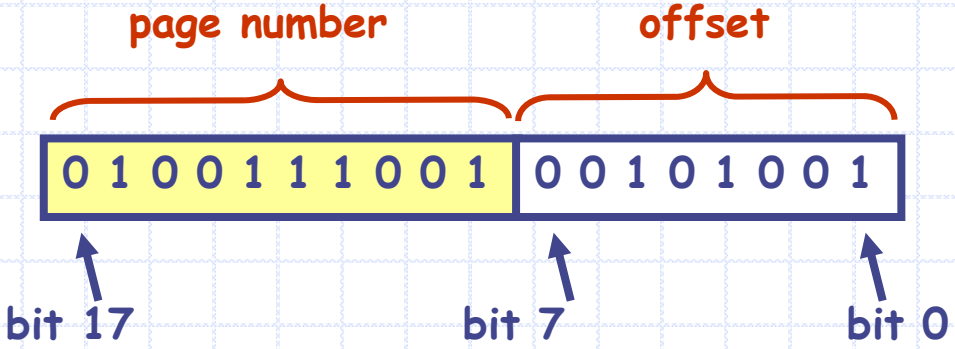


Example:

- 18 bit virtual address
- 16 bit physical address
- page size = 2^8 = 256 bytes
- address space size = 2^{18}
  = 256KB
- physical mem size = 2^{16}
  = 64KB

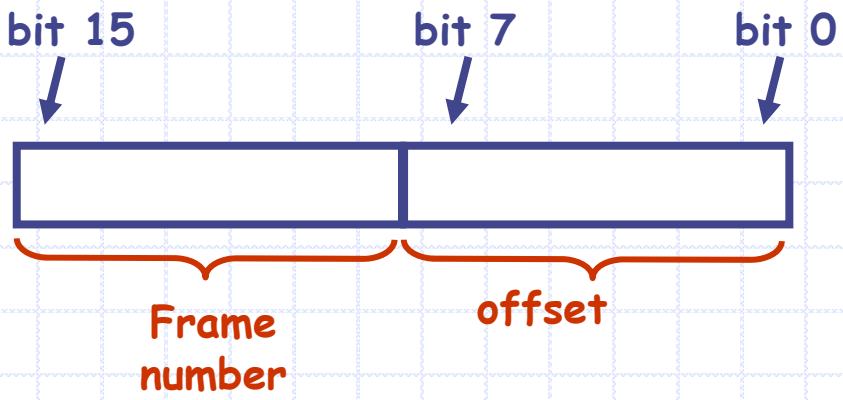


Address Translation Example -7

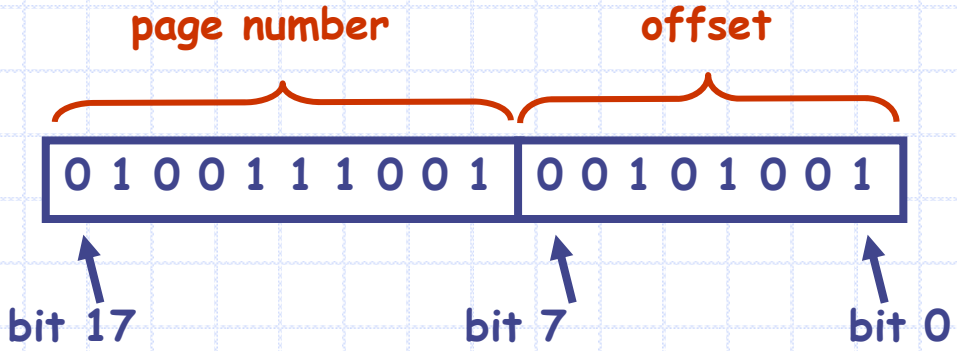


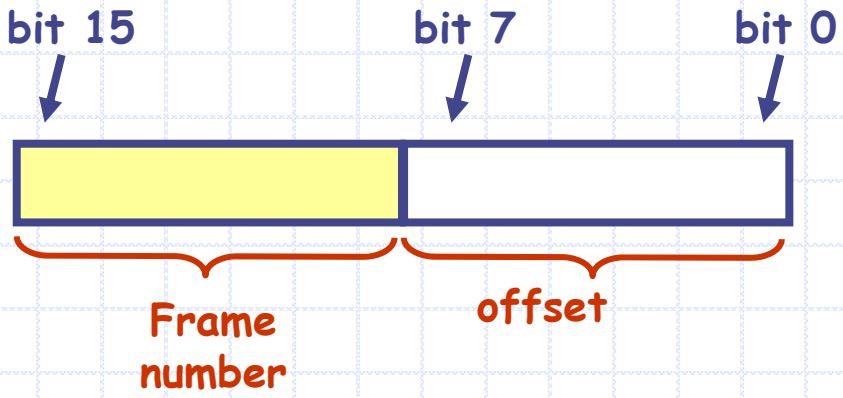
Example:

- 18 bit virtual address
- 16 bit physical address
- page size = 2^8 = 256 bytes
- address space size = 2^{18} = 256KB
- physical mem size = 2^{16} = 64KB
- number pages = 2^{10} = 1024



Address Translation Example -8

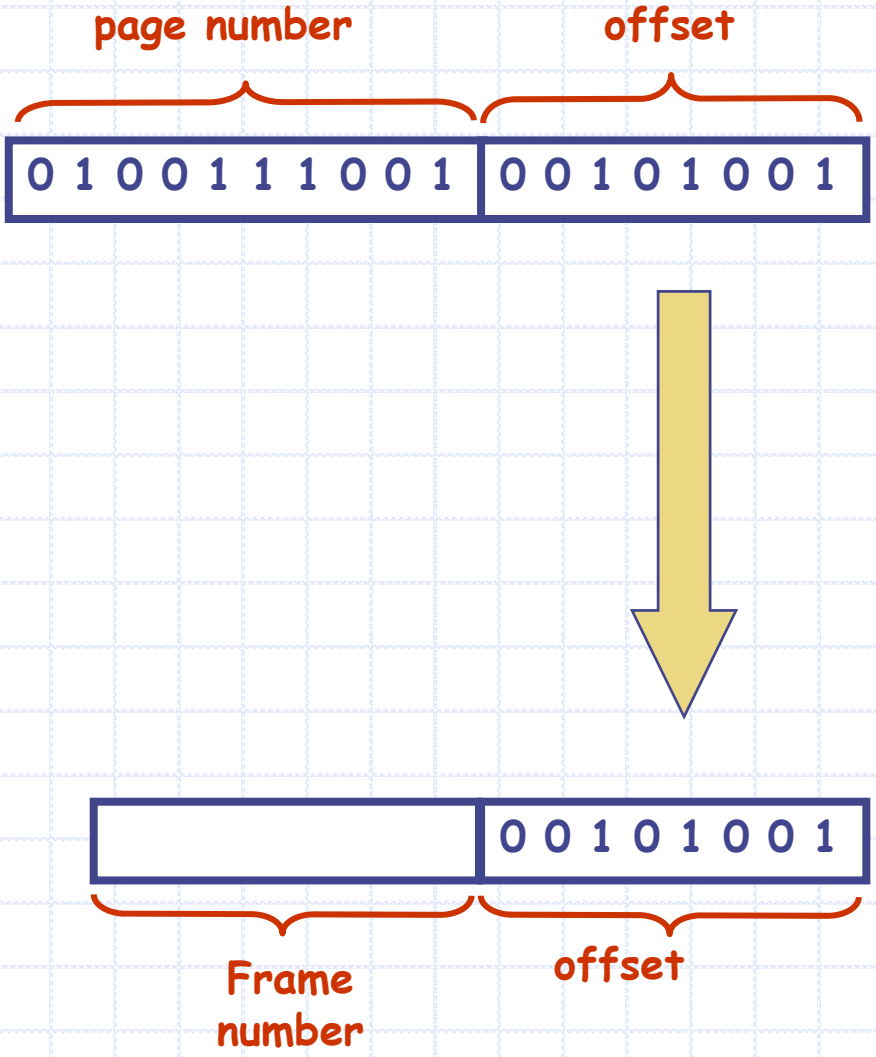




Example:

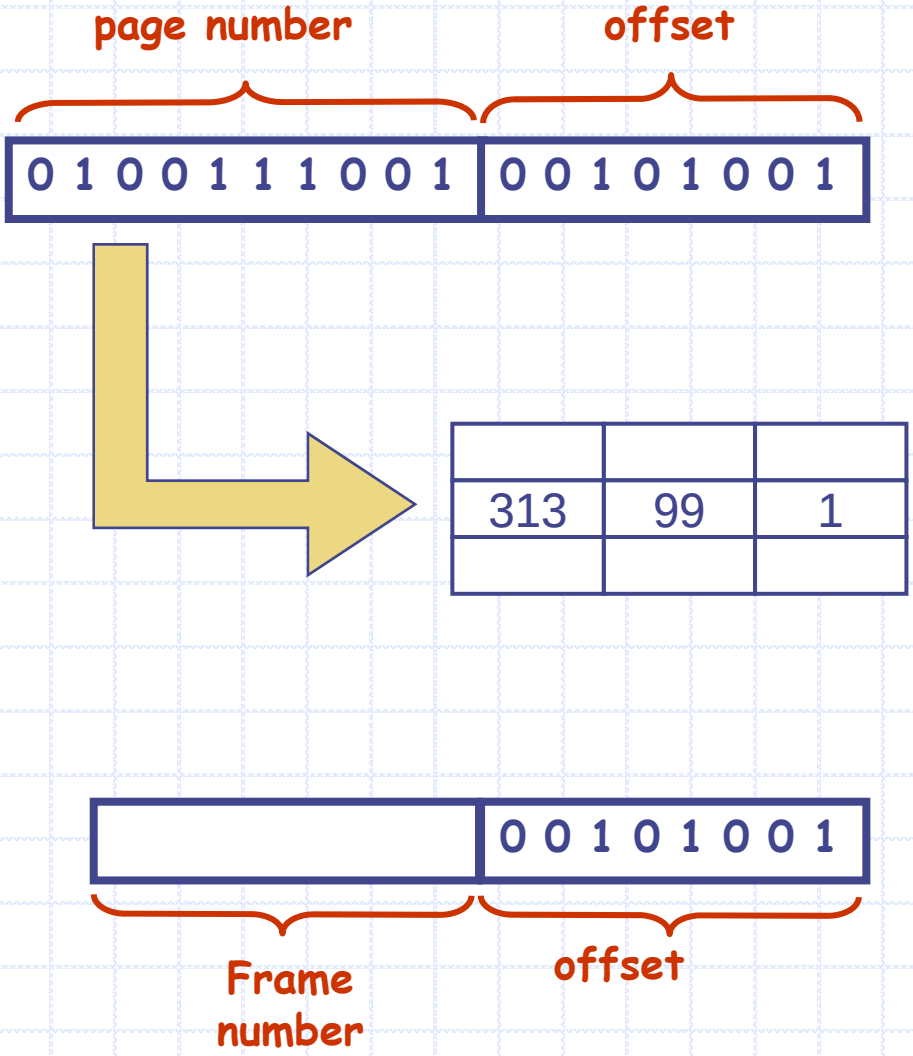
- 18 bit virtual address
- 16 bit physical address
- page size = 2^8 = 256 bytes
- address space size = 2^{18} = 256KB
- physical mem size = 2^{16} = 64KB
- number pages = 2^{10} = 1024
- number frames = 2^8 = 256

Address Translation Example -9



Step 1: the offset is shifted directly into from virtual to physical address

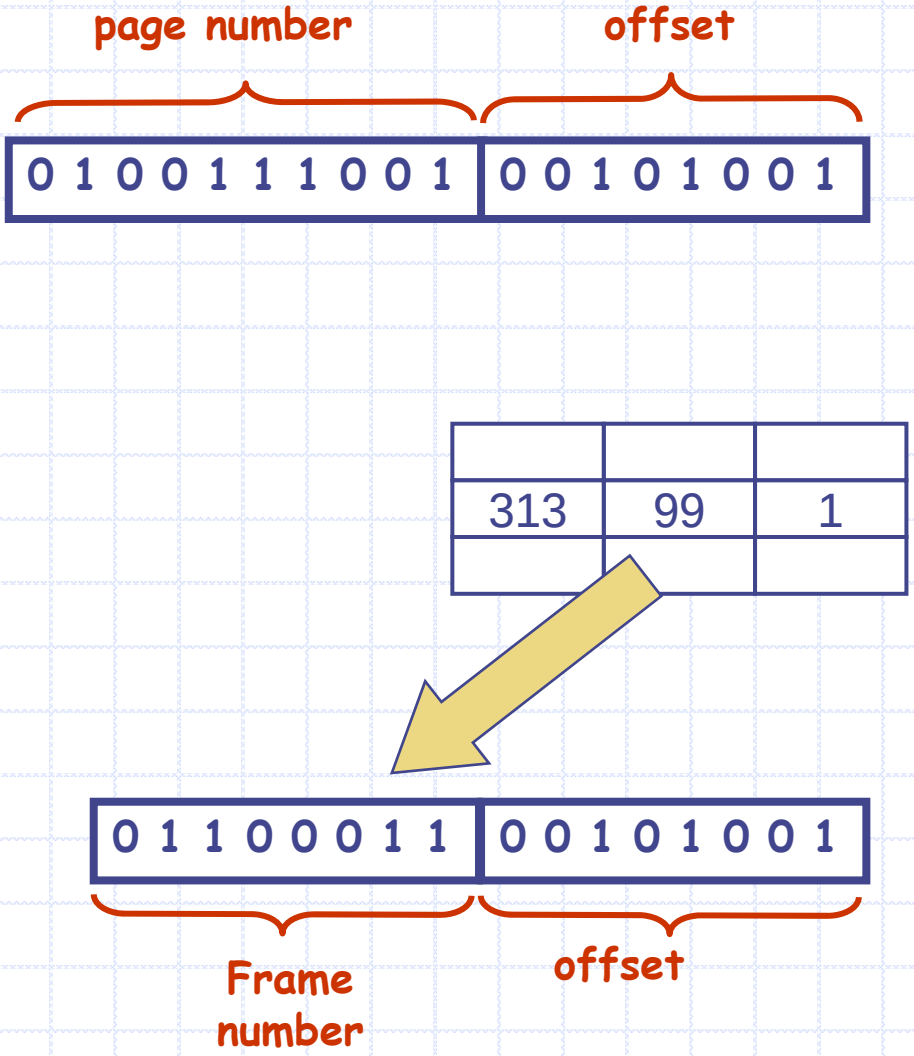
Address Translation Example -10



Step 1: the offset is shifted directly into from virtual to physical address

Step 2: page number is the index to a page table entry

Address Translation Example -11



Step 1: the offset is shifted directly into from virtual to physical address

Step 2: page number is the index to a page table entry

Step 3: frame number is taken from the page table and placed in physical address

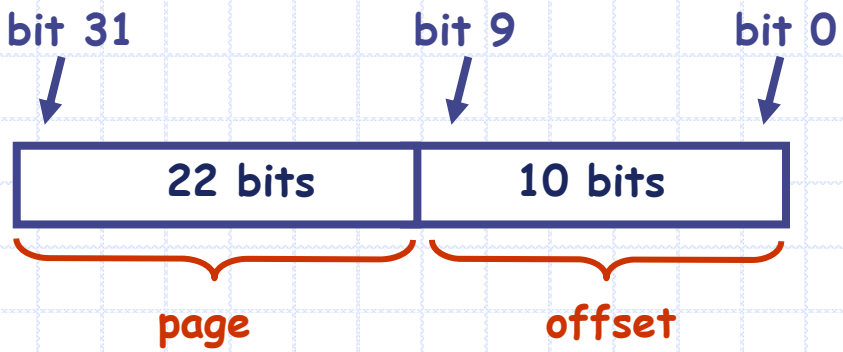
Exercise (a)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - How many pages can fit in memory at one time?

physical memory size = 2^{22} = 4MB

page size = 1KB = 2^{10}

4MB / 1KB = 2^{22} / 2^{10} = 2^{12} = 4KB = 4096 pages

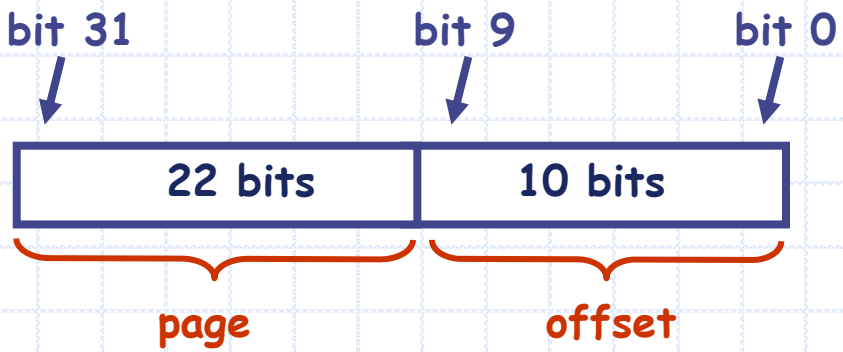


Exercise (b)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - How many bits are used to represent the offset?

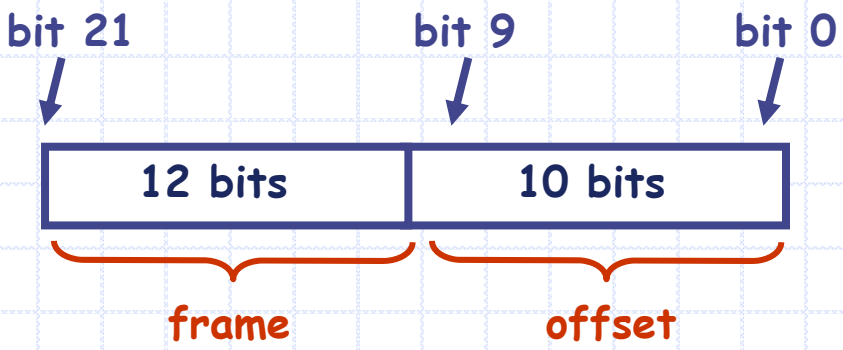
offset is the page size = 1KB = 2^{10}

→ 10 bit offset



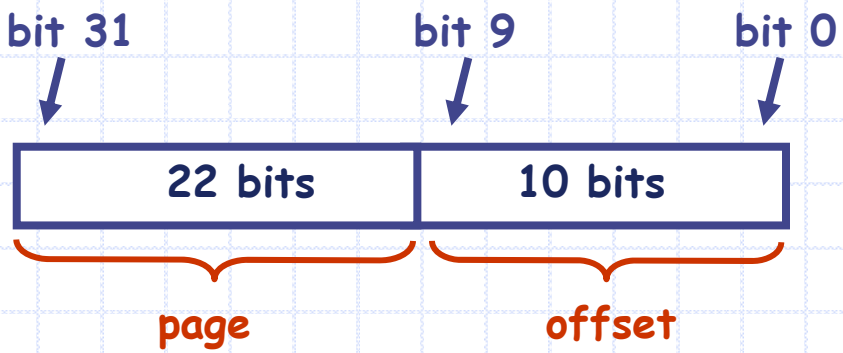
Exercise (c)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - How many bits are used to represent the frame number?
    - offset needs 10 bits
    - 4MB = 2^{22} = 22 bits of addressable RAM
    - 22 bits = 10 bits = 12 bits for the frame number



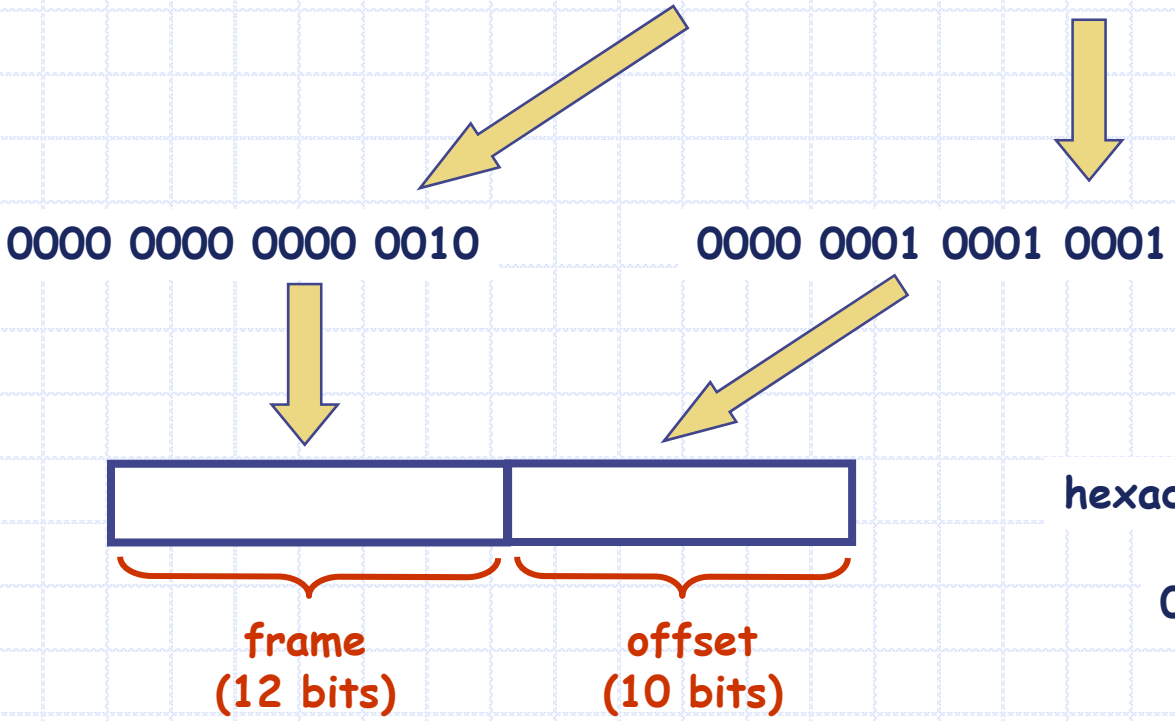
Exercise (d)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - How many bits are used to represent the page number?
    - offset needs 10 bits
    - 32 bits of addressable virtual memory
    - 32 bits = 10 bits = 22 bits for the page number



Exercise (e)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - What is the hexadecimal address of frame 0x0002, offset 0x0111?

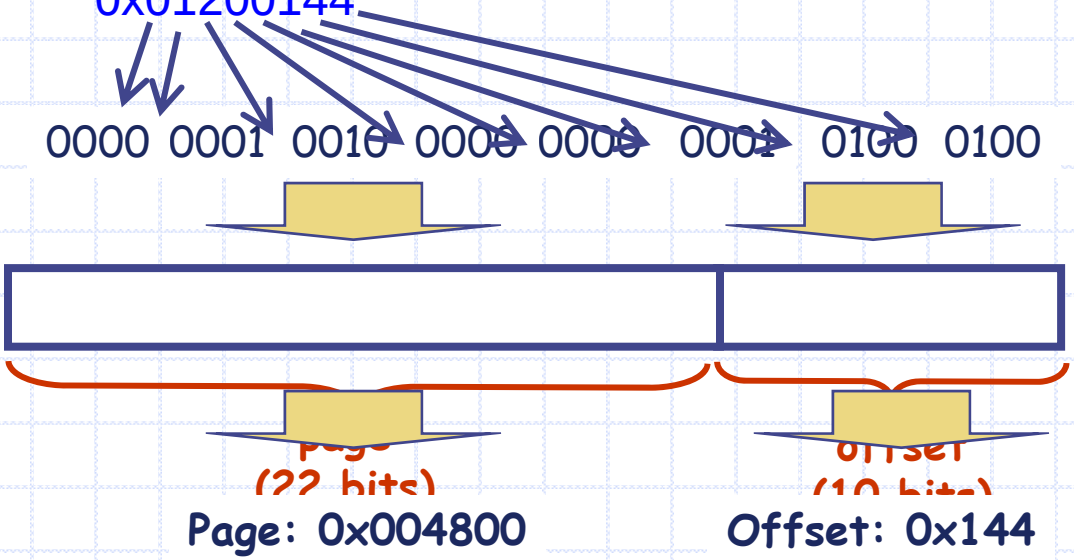


hexadecimal address

0x0911 hex

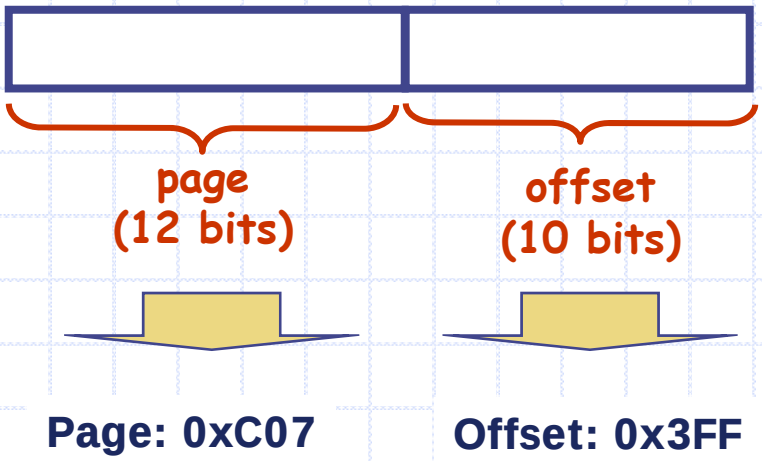
Exercise (f)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - What are the page number and offset (in hex) of the virtual address 0x01200144



Exercise (g)

- Suppose a machine has 4MB of RAM, uses a page size of 1KB, and uses 32 bit addresses.
  - What are the frame number and offset (in hex) for the physical address 0x301FFF



Address Translations (implementation)

Address Translations

- this address translation needs to happen for every instruction
  - *therefore it needs to be really fast!*
- luckily this can mostly be done in hardware, using SHIFT, AND, and OR instructions
- let's look at how this translation is implemented in a bit more detail ...

Address Mappings and Translation (1)

- Address mappings are stored in a page table in memory
  - Typically one page table for each process
- Address translation is done by hardware (ie the MMU)
- How does the MMU get the address mappings?
  - Should the MMU do a memory lookup (recall, the page table is in memory) for each memory access?
    - *No! This would be way too expensive (in memory access time).*
  - Should the MMU hold the entire page table
    - *No! This would also be too expensive (to implement in hardware).*
  - We need to design the MMU to hold only a portion of the page table
    - MMU caches page table entries
    - called a translation look-aside buffer (TLB)

Address Mappings and Translation (2)

- What if the TLB needs a mapping it doesn't have?
- Software managed TLB
  - it generates a TLB-miss fault which is handled by the operating system (like interrupt or trap handling)
  - The operating system looks in the page tables, gets the mapping from the right entry, and puts it in the TLB
- Hardware managed TLB
  - it looks in a pre-specified memory location for the appropriate entry in the page table
  - The hardware architecture defines where page tables must be stored in memory

Page Tables (revisited)

- When and why do we access a page table?
  - We do not need page table lookups on every instruction …
    … because many of these translations are in the TLB.

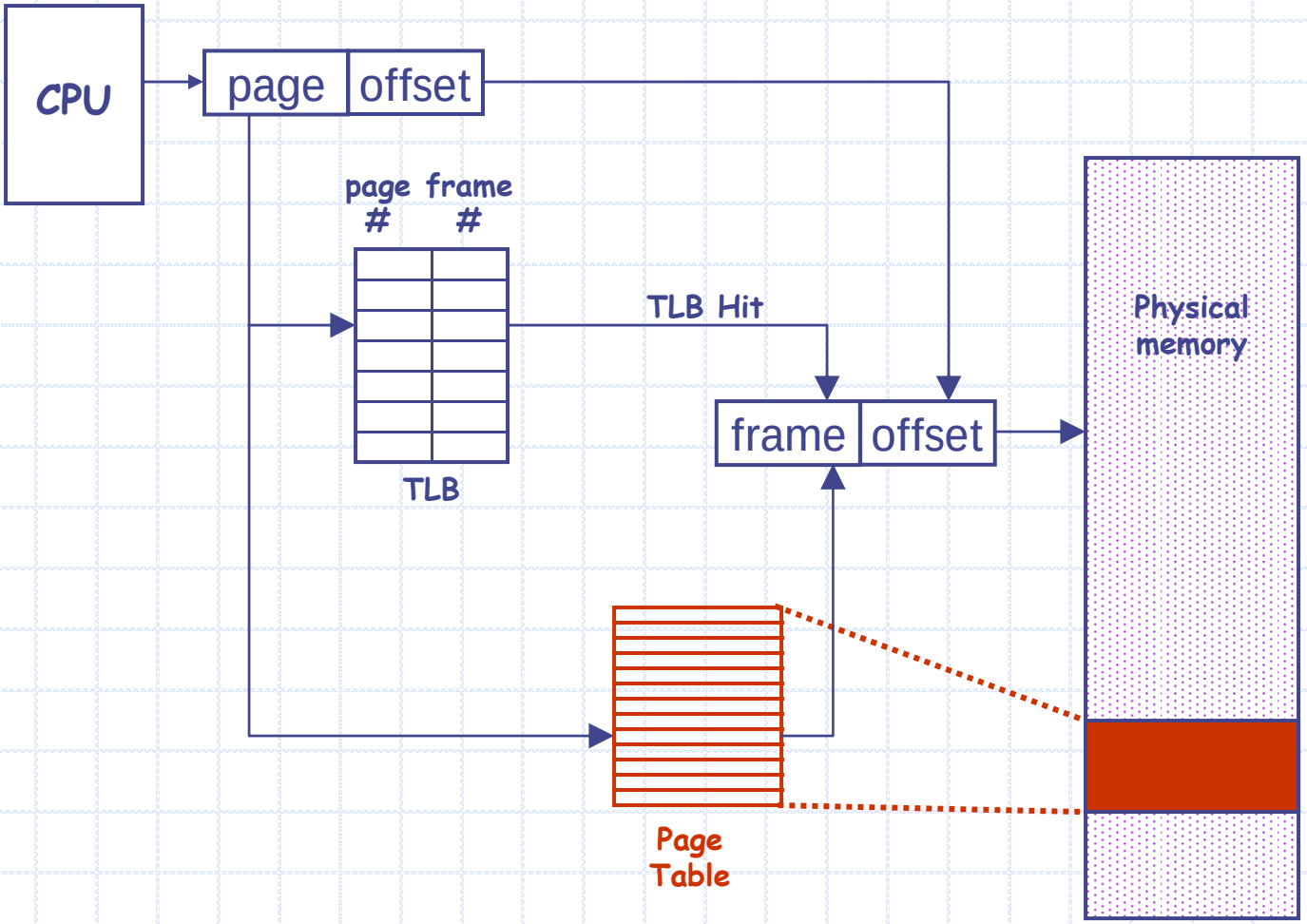
1. We access the page table on TLB miss faults
   - to refill the TLB with new entries from the page table

2. We access the page table during process creation and destruction
   - like to create/destroy it, initialize mapping, load TLB

3. We access the page table during process execution
   - when a process starts executing we need to flush the TLB, and load it with entries for the new process

4. We access the page table when a process allocates or frees memory
   - ie: if it allocates more than is currently available we need a new page, so we need to update the page table

Translation Look-aside Buffer



Page Fault Handling (details) - 1

Previously, a summary of page fault handling was presented.

Now we will look at it in detail.

Assume that a process is executing when it causes a page fault ….

(ie: it tries to reference a frame that is not loaded in memory)

1. The hardware traps to the kernel, saving the PC on the stack
   - note: most machines also save info (state) for the current instruction in reserved CPU registers
2. An OS routine (similar to an interrupt handler) saves the general CPU registers and other volatile information. This routine then calls the page fault handler in the OS.
3. The page fault handler needs to discover which page caused the fault.
   - this could be in one of the hardware registers, or
   - the OS could retrieve the PC, fetch the instruction that was executing, and parse it to figure out what page is needed

Page Fault Handling (details) - 2

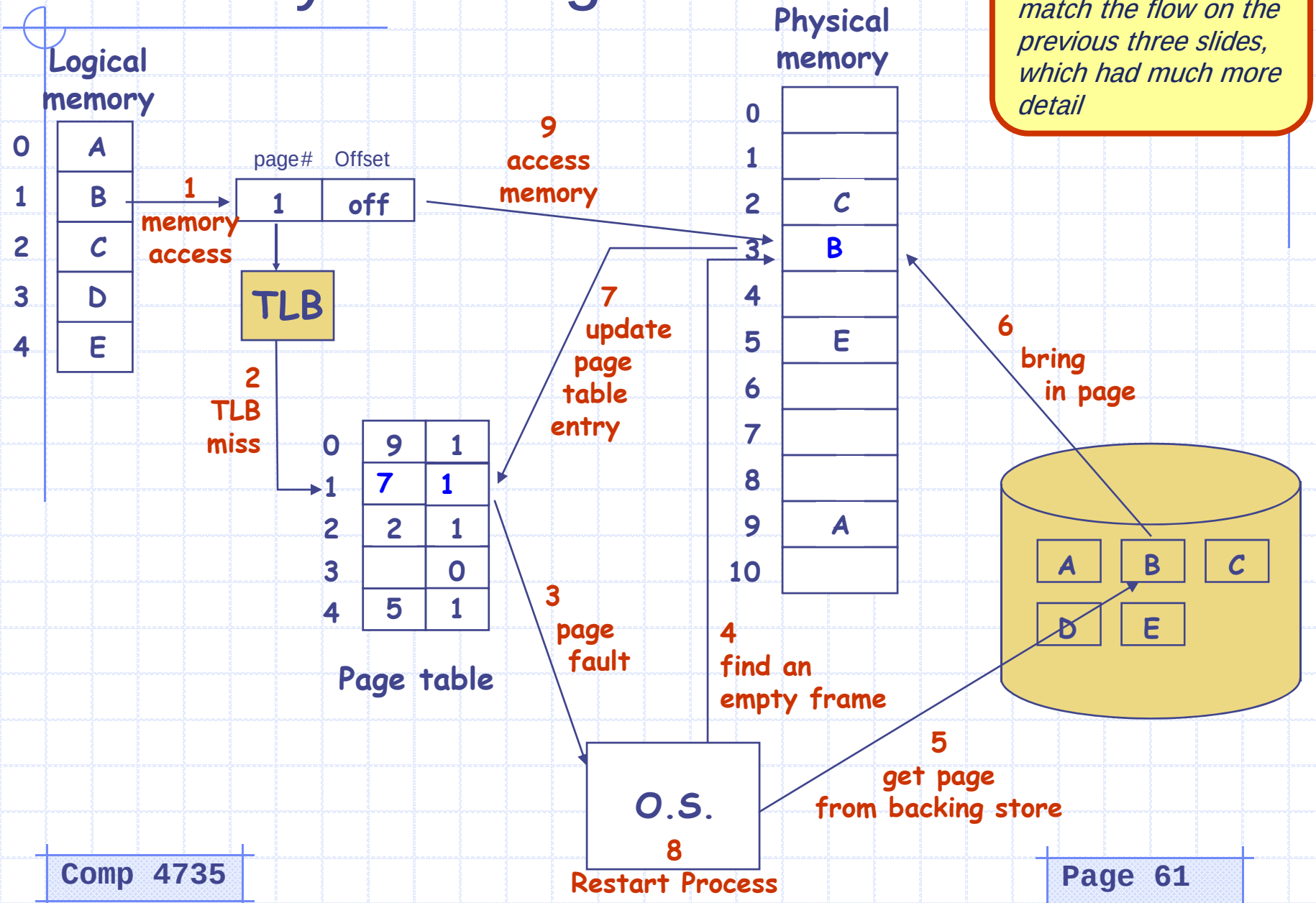
4. The OS checks the virtual address to make sure it is valid.
   - if not, it signals the process or throws an exception
5. The OS checks to see if there is an empty page frame, and reserves it
   - if there is no empty frame, the page replacement algorithm is run to select a candidate to be paged to disk
6. If the frame is “dirty” (ie: it is in use by another process), the OS starts copying it to disk, and performs a context switch (so that other processes can run)
7. When the frame is “clean” the OS finds the location of the needed page (on disk), and initiates disk IO to load the page into the frame
   - a frame is considered as “clean” once it has been written to disk … or immediately if it was empty to begin with
   - note that the faulting process is still suspended while all this is happening

Page Fault Handling (details) - 3

8. When the disk interrupt occurs to indicate that the page has been transferred, the OS updates the page table to show the frame location and status.
9. The faulting instruction is rolled back to the state it had before the instruction started, and the PC is reset accordingly.
10. The faulting process is scheduled, and the OS returns control to the assembly routine that called it.
11. The assembly routine restores the registers it backed up, and returns to user mode where execution continues.

Anatomy of a Page Fault

fine print: the numbers on this slide do not match the flow on the previous three slides, which had much more detail



The End