ECS518U - Operating Systems
Week 6

Memory Management: Address Spaces & Paging

Tassos Tombros

Outline

- **The problem** that memory management tries to solve
  - Code relocation
  - Efficient memory use
- **Solutions**
  - Segments
  - Pages
- **Virtual memory**
- **Locality and page faults**
- OS **design** issues
  - Page replacement, …
- **Reading:**
  - **Stalling:** Chapter 7 & Chapter 8 (8.1, 8.2)
  - **Tanenbaum:** Chapter 3

Today

Week 8

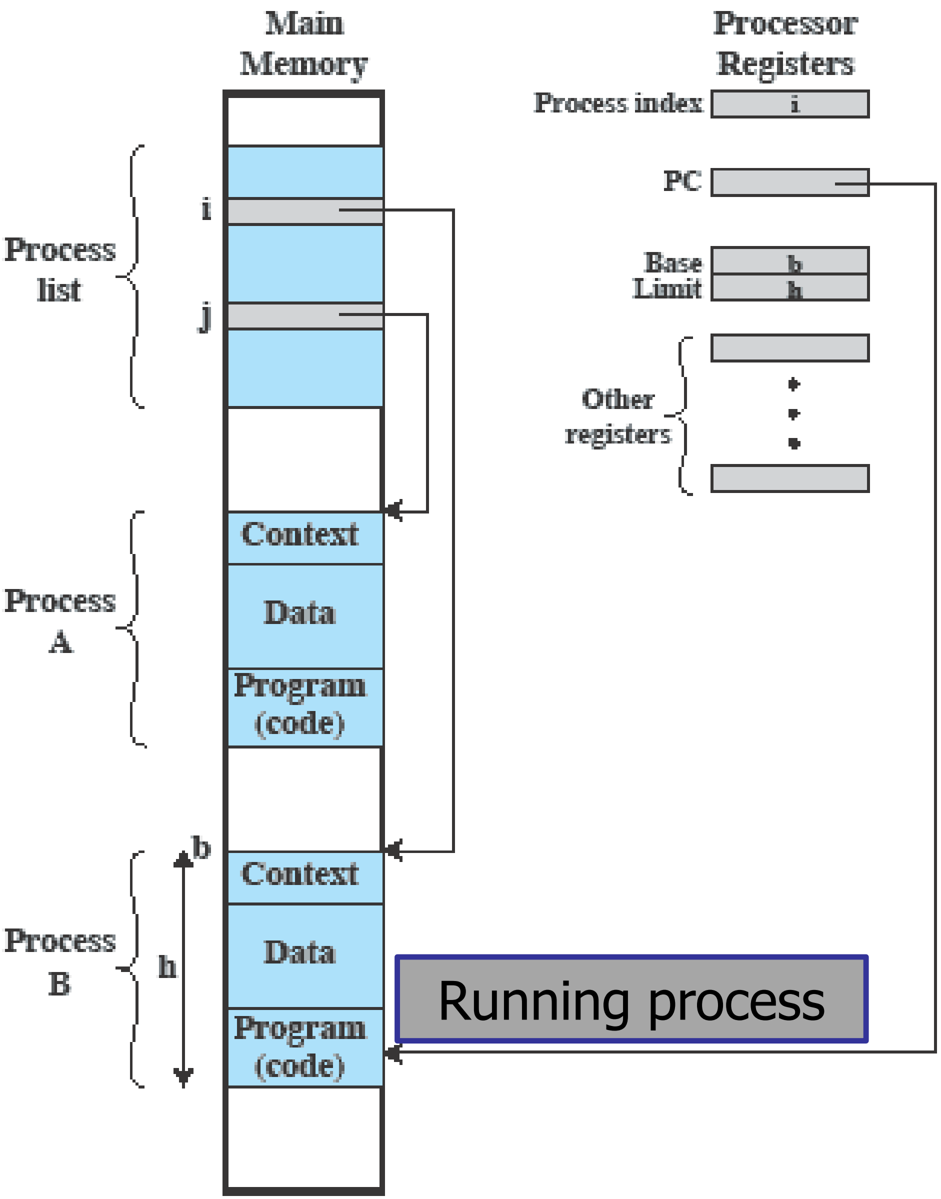
Things you will learn today

- Why is Memory Management necessary, **what problems does it solve?**
- Different address spaces
  - The logical address space
  - The physical address space
- **Solutions** to the Memory Management problem:
  - Fixed & Dynamic **Partitioning**
  - **Segmentation**
  - **Paging**

IMPORTANT: LOOKING AHEAD

- **WEEK 7**
  - **1ST MCQ Test will be released**
  - WILL OPEN **SATURDAY 24/02, morning**
  - Open for 36 hours
  - Make use of the non-assessed ones that I put up every week for practice
    - Also to know how to save, restart, submit, etc.
  - **MATERIAL:** All we have covered till end of Week 6
- **LECTURE WEEK 7**
  - **Exam preparation (well, sort of)**
  - **Anatomy of a good answer…**
  - Look at exam questions, look at what good and poor answers look like

From Week 2



- Many processes share the memory
- But **where** in memory are they?
- Are they actually nicely arranged in contiguous blocks like in the picture?
- The **compiler** we use for our programs needs to know where in memory to put program data
  - But it does not know, the picture proves it – why?
  - Because it can not possibly know how many other ‘things’ are running and where in memory they are

Memory Management Requirements

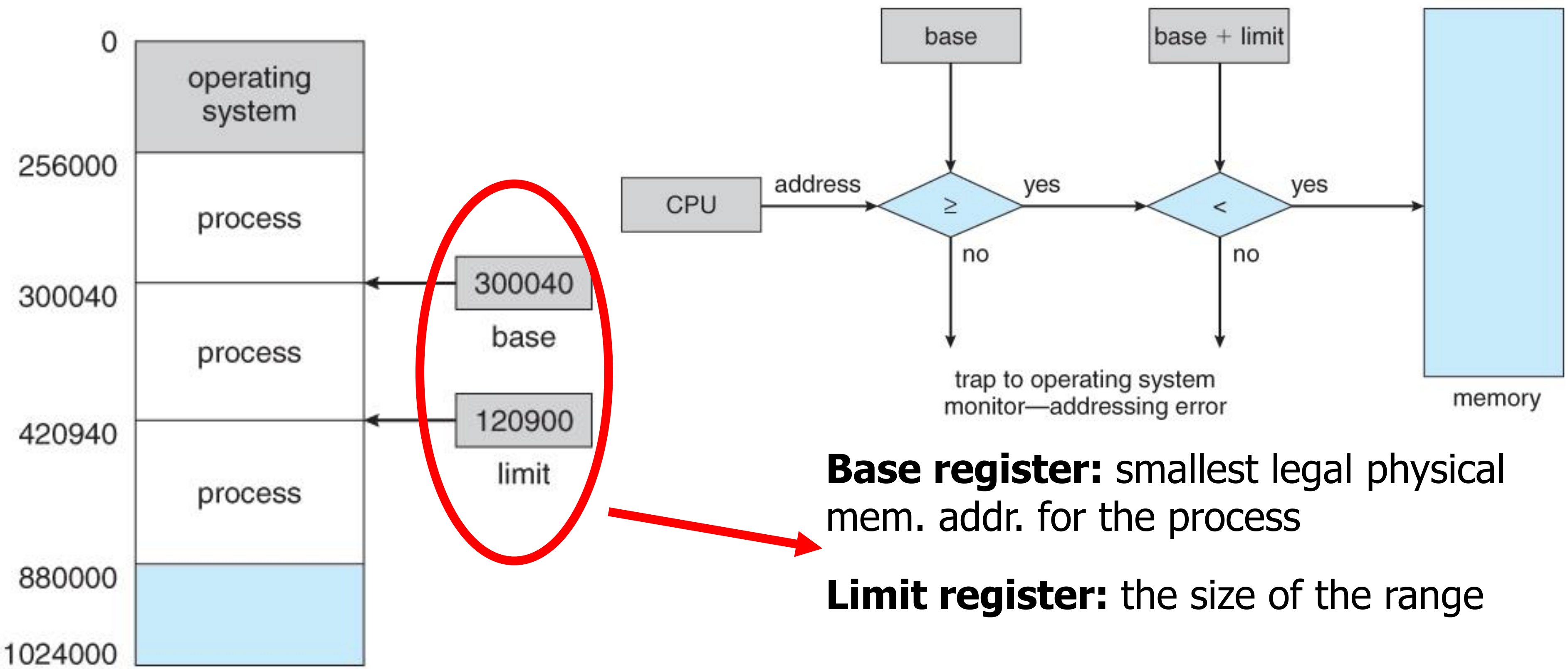
- **Why is memory management necessary**?
  - The computer memory is shared by (many) processes
  - But the compiler (and us) must produce code disregarding this fact – we can not know how many other processes are and where they are in memory
- Main **requirements** for memory management
  - **Relocation** (the ability to move the program from one place in memory to another)
  - **Protection** & **sharing**
  - Each process working in its own **address space** - (illusion)

Req. 1: Relocation

- **Where** is the program in memory?
  - the programmer does not know, the compiler does not know
  - it may move around after it is loaded, will def. move around if you stop it and run it later
  - what if you run two instances of the same program? where will they be?
- We need to **be able to relocate a program in memory**
- **What** really are variables?
  - References to memory locations
- **How** is this simple instruction executed x = x + 1?
  - In a series of LOAD and STORE assembly instructions

Req. 2: Protection and Sharing

- No use of memory locations in another process
  - Accessing memory that the program is not allowed to causes a **trap** (check the term from week 2) – switch into kernel mode for OS to act
- But there are cases where we may want to allow several processes to access the same portion of memory
  - Inter-Process Communication (IPC), shared libraries, …



Base register: smallest legal physical mem. addr. for the process

Limit register: the size of the range

Addresses: 2 types (for now)

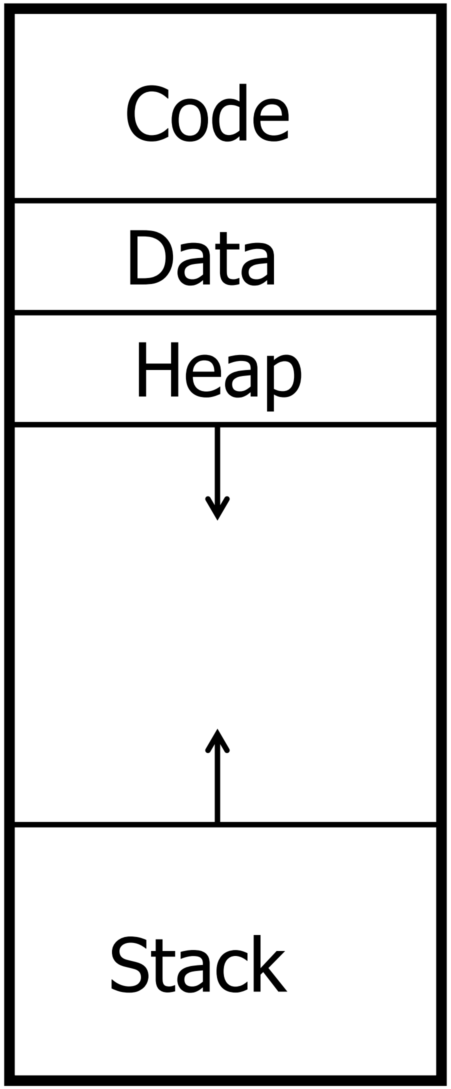
- **Logical**
  - Reference to a memory location independent of the current use of memory
  - Instructions issued by CPU reference logical addresses (pointers, arguments to LOAD/STORE, etc.)
- **Physical** or Absolute
  - The actual location in memory
  - Used by CPU
  - Easy to remember: physical memory

the memory you paid for!!!

- Address translation
  - **Change the address :** logical $\rightarrow$ physical

Logical Address Space

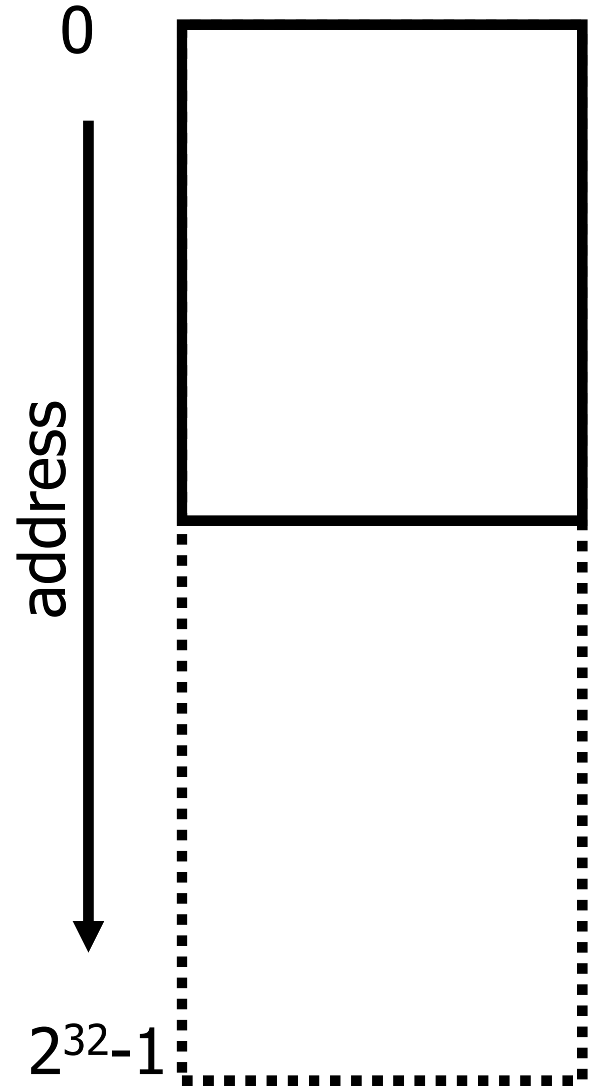
- **Logical Address space** is the set of addresses available to a program
- Possible addresses
  - 16 bits = Not enough any more
  - 32-bits = 4GBytes
  - 64-bits = (A LOT!!!)
- Recall these different segments from Week 2?



Physical (Real) Memory

- **The memory we paid for**
- Real memory is always less than logical address space
  - **Memory mapped I/O**
  - **OS data**
  - **Simply can not afford 2^{64} memory**

- **Address space abstraction:**
  - Process has its own address space
  - Can you imagine how it would be without this abstraction?



e.g.: 4GB of physical memory

An old Solution – Partition Memory

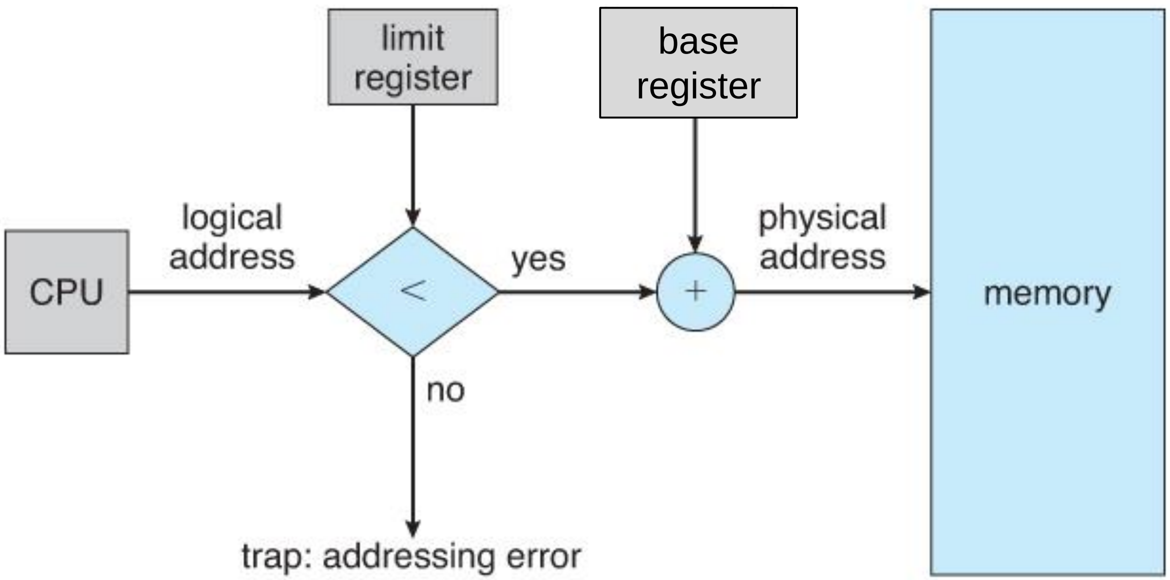
- **Fixed Partitioning**
  - Divide memory into contiguous ‘blocks’ at boot time
  - Partitions don’t change – what happens if your program grows? (you run out of space)
  - **Internal fragmentation** (we have reserved some memory for a process which we are not fully using)
  - Hardware requirement: **base** register (loaded by OS at context switch)
    - **physical address = logical address + base register**
- **Dynamic Partitioning**
  - Create partitions as programs are loaded
  - **External fragmentation** (unused memory broken-up into small regions that are unusable), but no internal fragmentation
  - Hardware requirement: **base and limit register**
    - **physical address = logical address + base register**
    - **protection:** if (physical address > (base+limit)) then **TRAP**!!!!

Dynamic Partitioning Example

- **External Fragmentation**
- Memory external to all processes is fragmented
  - In this scenario, a new process requiring 7M of memory would not fit, even if the total free memory available is much over 7M
  - We would need to compact the memory to make the free space contiguous – this has a high overhead

Concept link: (fragmentation in disks)

Protection and relocation in dynamic partitioning



- **protection** against user programs accessing areas that they should not
- programs can be **relocated** to different memory addresses as needed

Uses base and limit registers

- **base:** the ‘starting point’ of the process in physical memory
- **limit**: the ‘size’ of the process

Key Concepts - Checkpoint

1. **Relocation**
   - Code can be moved from one location in memory to another

2. **Fragmentation**
   - A problem to avoid: unusable chucks of memory
   - Internal and external

3. **Logical versus physical addresses**
   - Logical: from the compiler, as seen by the CPU
   - Physical: the memory we paid for

4. **Address translation**
   - Change logical addresses to physical
   - One method of relocation

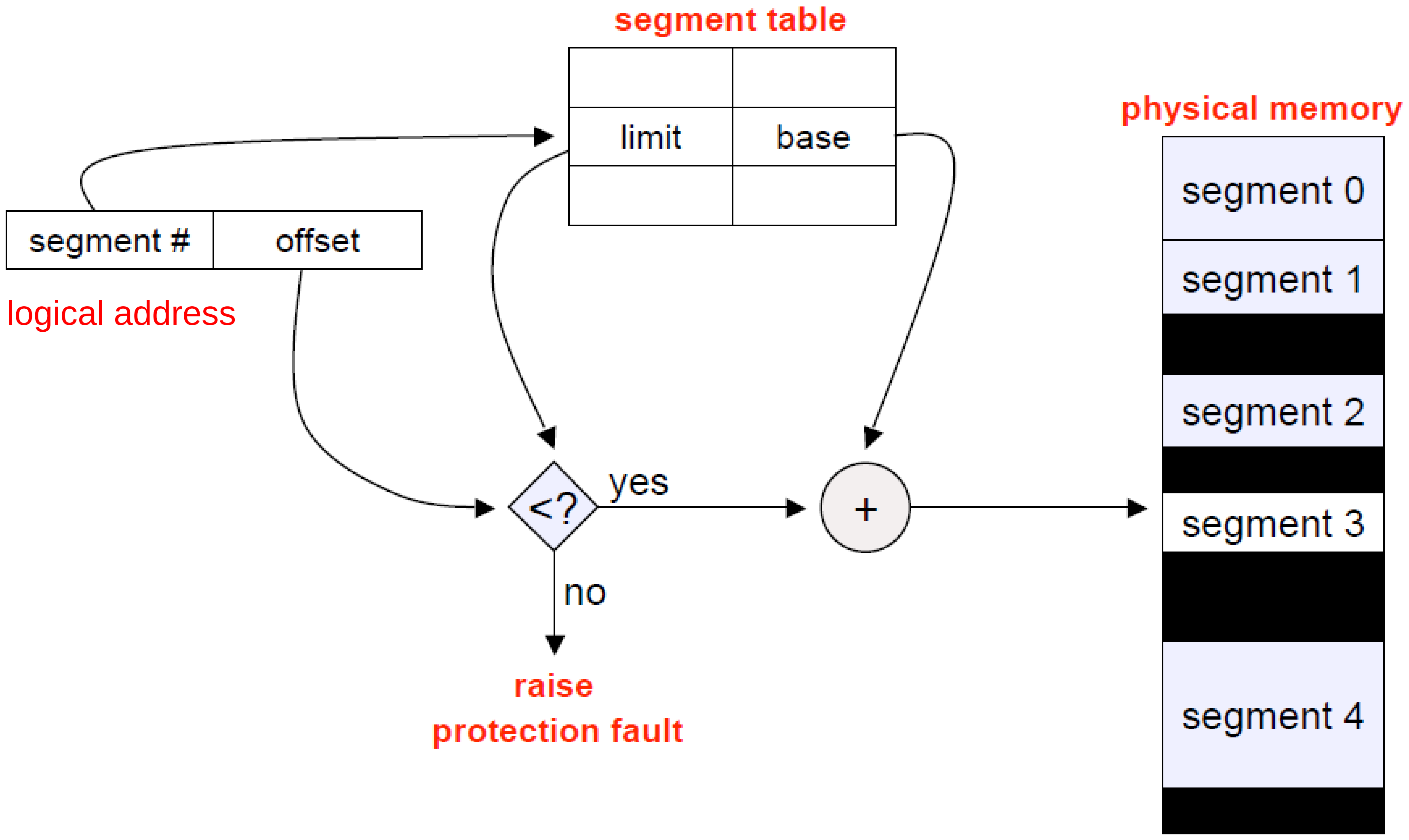
Better Solutions: Pages and Segments

- **Segmentation**
  - Divide program into segments
    - Each segment is contiguous
    - Some external fragmentation
- **Paging**
  - Divide memory into equal-size pages
  - Load program into available pages
    - Pages do not need to be consecutive
  - Some internal fragmentation
- **Virtual memory** (Week 8)
  - "when our programs need more memory than we paid for…"

Segmentation

- **A program can be subdivided into segments**
  - Segments may vary in length
  - There is a maximum segment length
- Why do we do that?
  - e.g. we want to separate the code from the data
  - **Code is read-only, data can be r+w**
  - Because code is read-only we can also safely share it if we want amongst many processes
- Addressing consists of two parts
  - a segment number and
  - an offset
- Segment table for a process

Segment look-ups



Segmentation Pros and Cons

- **Pros**
  - Protection, as we can set segments for
    - Data: read and write
    - Code: read only
    - Shared
  - No internal fragmentation
- **Cons**
  - Segment contiguous in memory (not flexible)
  - External fragmentation

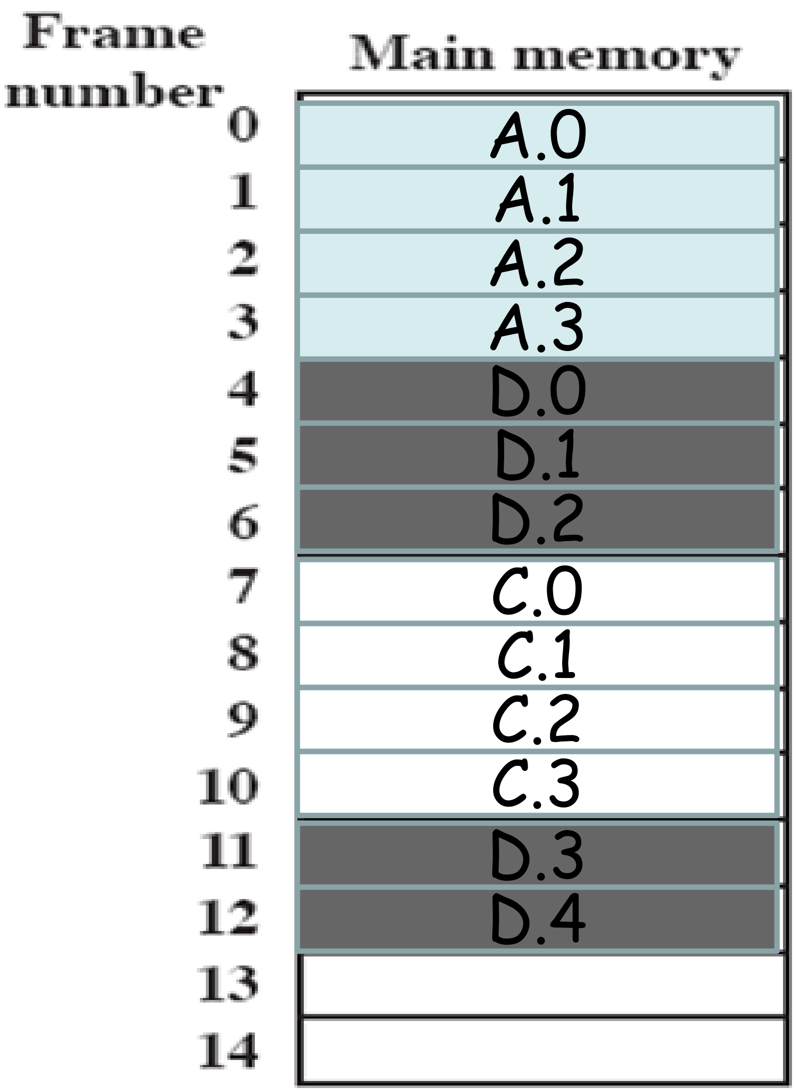
Looking ahead: Labs 6 & 7

- The next 2 labs use **SystemTap**
- **No more PHP ☺ … but still vminstance ☹**
- Allows to look into the kernel, we need to be 'root' so we run them using vminstance in ITL
- **Lab 6 (week 8):** Not assessed, looking at CPU burst lengths (scheduling)
  - Helps you learn to use system tap and prepare for the assessed lab
- **Lab 7 (week 9):** Assessed, looking at memory management (paging, page faults, etc.)
- Good (?) news: not a single line of code to write
- It requires (allows) you to **link what you observe in the lab with what we say in the lectures**

Paging: Frames and Pages

- Fixed size memory **frames** (refers to **physical address**)
- Process has memory **pages** (**same size as frames**) – refers to **logical address**
- We need to **allocate** a frame to each page that our process is using
  - any page (from any process) can be placed into any available frame
- Allows processes' physical memory to be **discontinuous**
- **Advantage** (in terms of efficient use of memory)
  - No external fragmentation
  - Internal fragmentation ok if page size (and so also frame size) is small

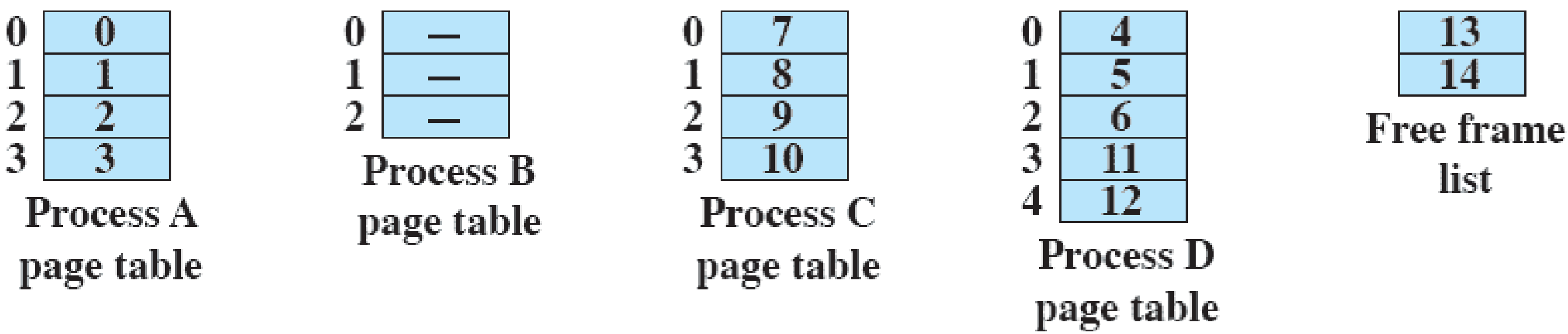
Page size trade-off



- The example shows that we avoid external fragmentation and that we allow for processes to grow (in pages)
- **Page size is a trade-off**
  - **Too large:** we risk to have a fair amount of internal fragmentation
  - **Too small:** We end up with too many pages and we need to store data in memory for them (page table)
  - **Current typical size: 4KB** but trend is to increase it as memory becomes cheaper (we can afford to store more stuff)

The Page Table

- Each process needs a map to show where its pages are (i.e. in which frame)
  - map **logical → physical address**
- We use a structure called a **Page Table for each process**
  - it is indexed by the page number



- **Protection:** each process only has access to its allocated frames
- **Sharing:** we can allow sharing on a page level

Page Table Entries

- Each process has a page table:

Page Number		
0	Page flags	Frame number
1		
2		
3		
4		
5		
6		

:

- **Flags:**
  - Is page in memory? (or it is on disk?)
  - Has it changed?
  - Protection: read only?
  - Shared?

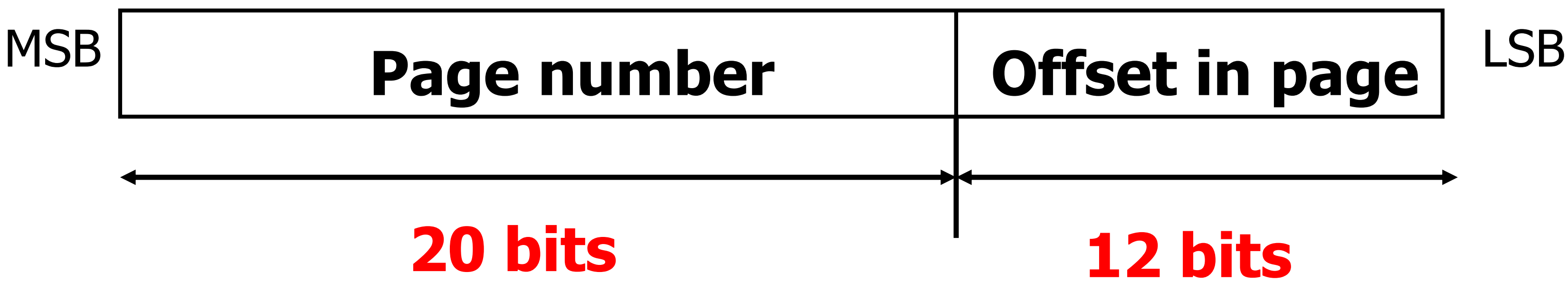
Page Table Entries: Example (Intel x86)

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as “valid” bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1$\Rightarrow$4MB page (directory only, for 2-level page tables).
Bottom 22 bits of logical address serve as offset

Page Sizes and Logical Addresses

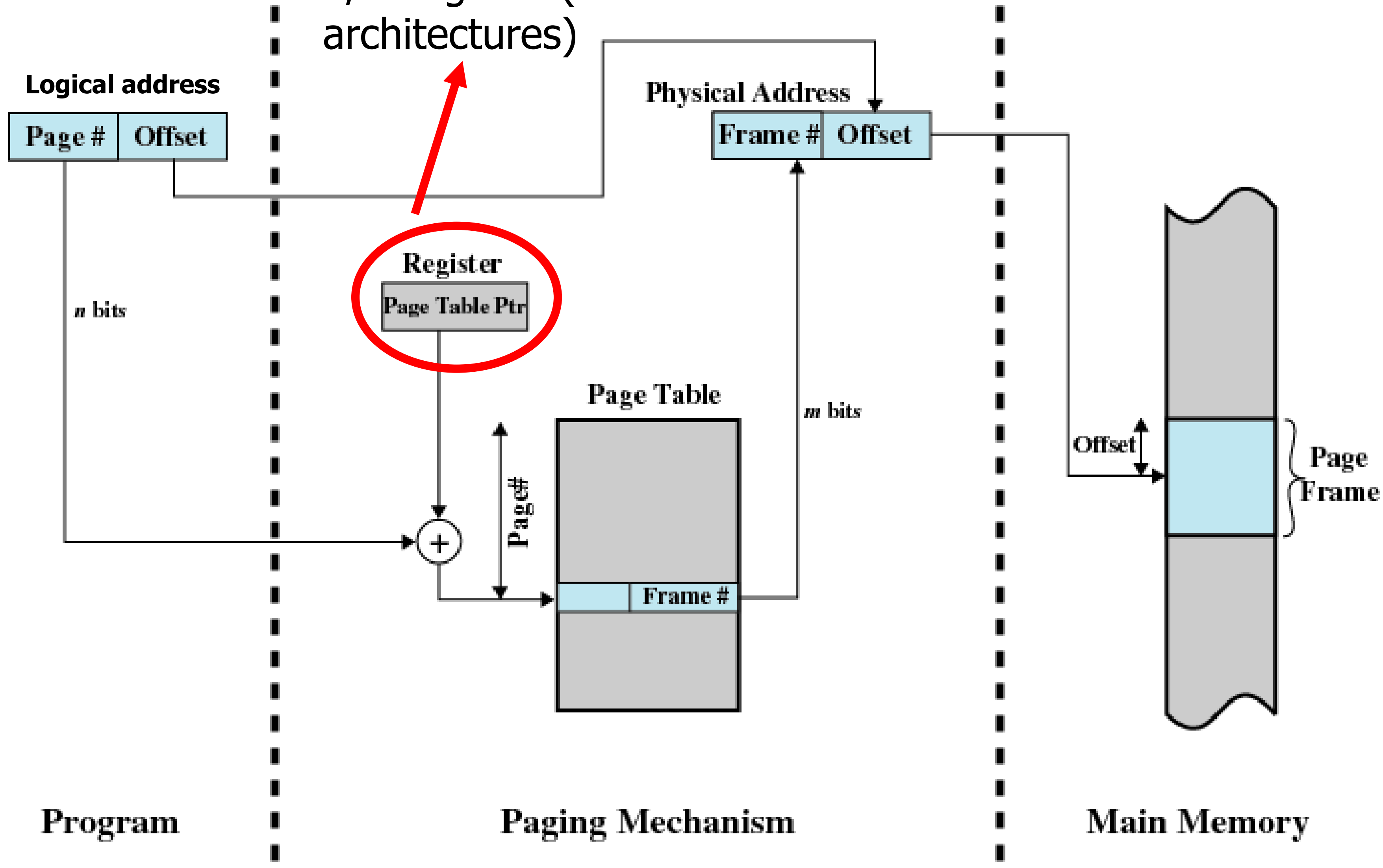
- A logical address for paging is split into:
  - Page number – **most significant bits**
  - Offset – **address within page**
- The actual split (e.g. 20/12) is architecture dependent



- From this we know:
  - **Page size: 2^{12} = 4KB (same as frame size)**
  - **Number of pages: 2^{20} = 1 million pages**
  - **Size of page table for each process, for 4bytes (=32 bits, as shown in previous slide) per Page Table Entry? 4MB**

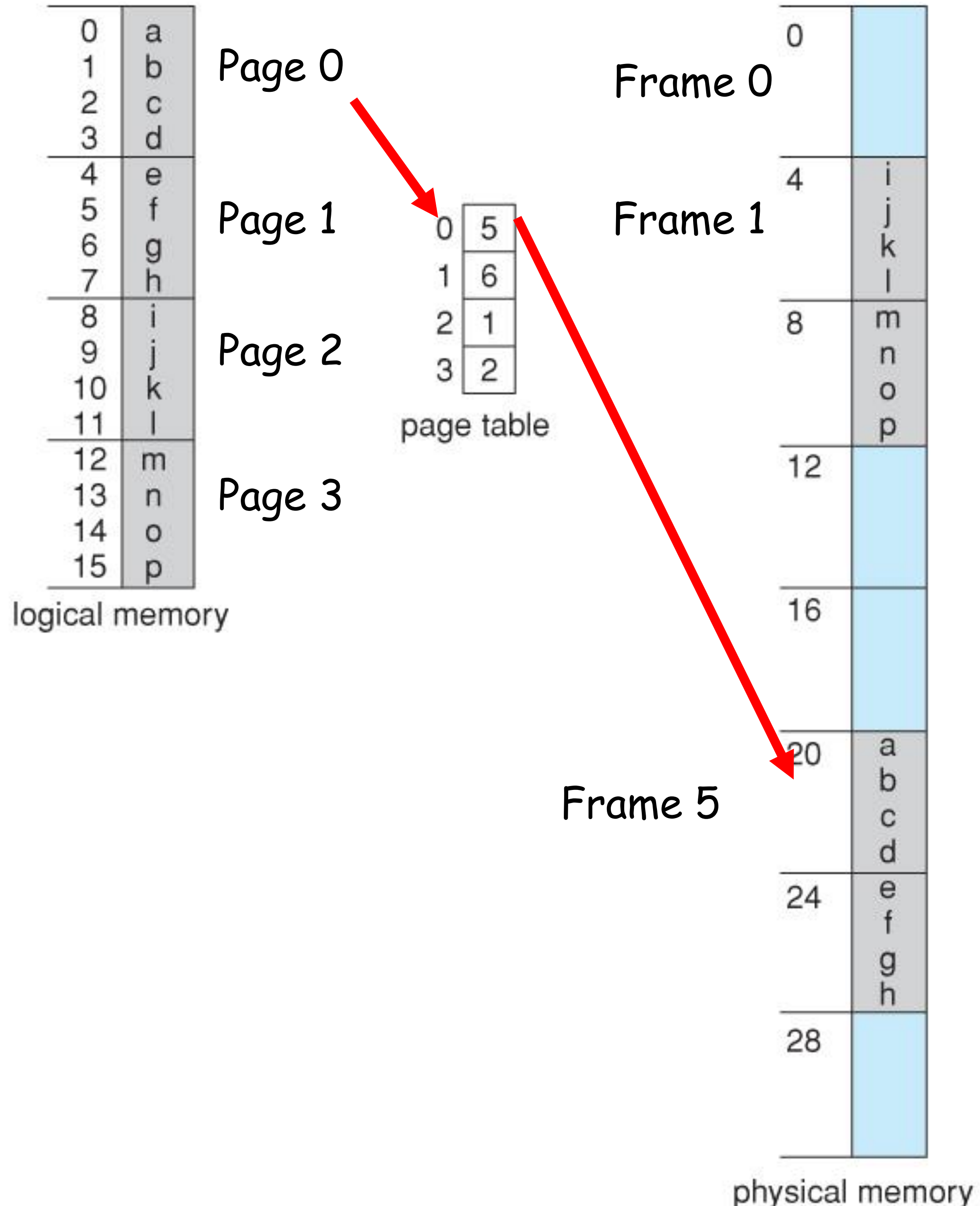
Address Translation I

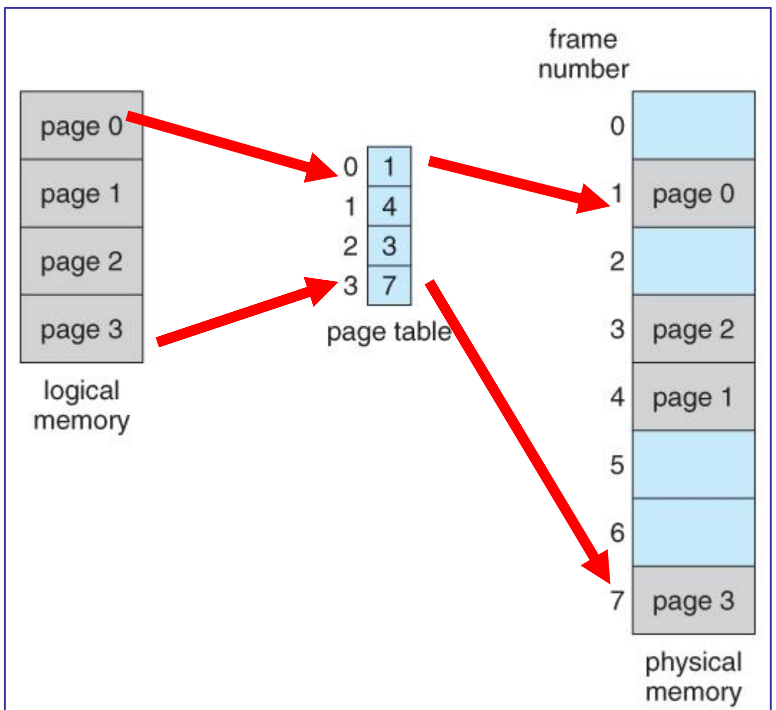
h/w register (CR3 in x86 architectures)



Examples

Can you figure out this example?

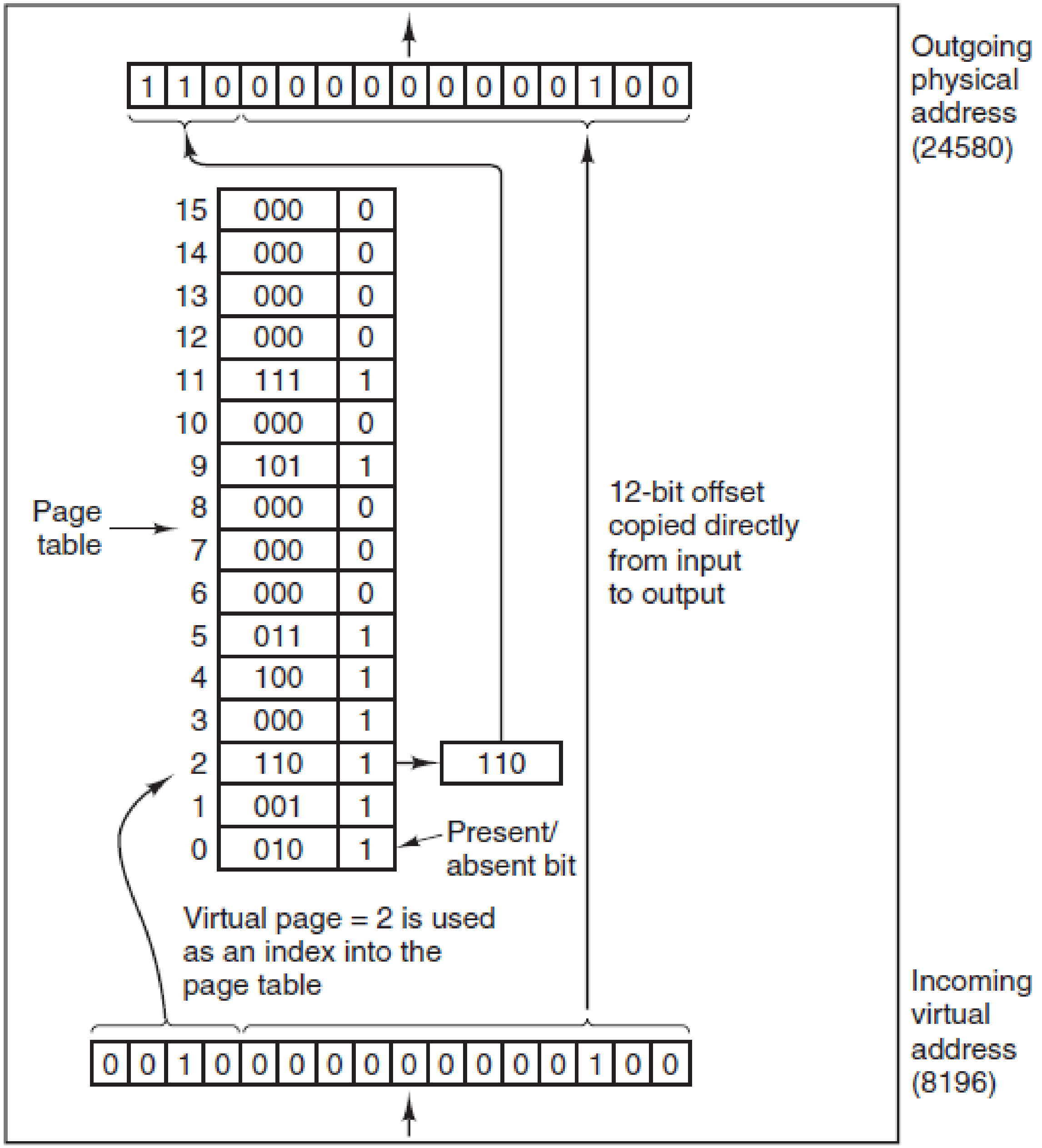




Example

What is great with 16-bit processors?

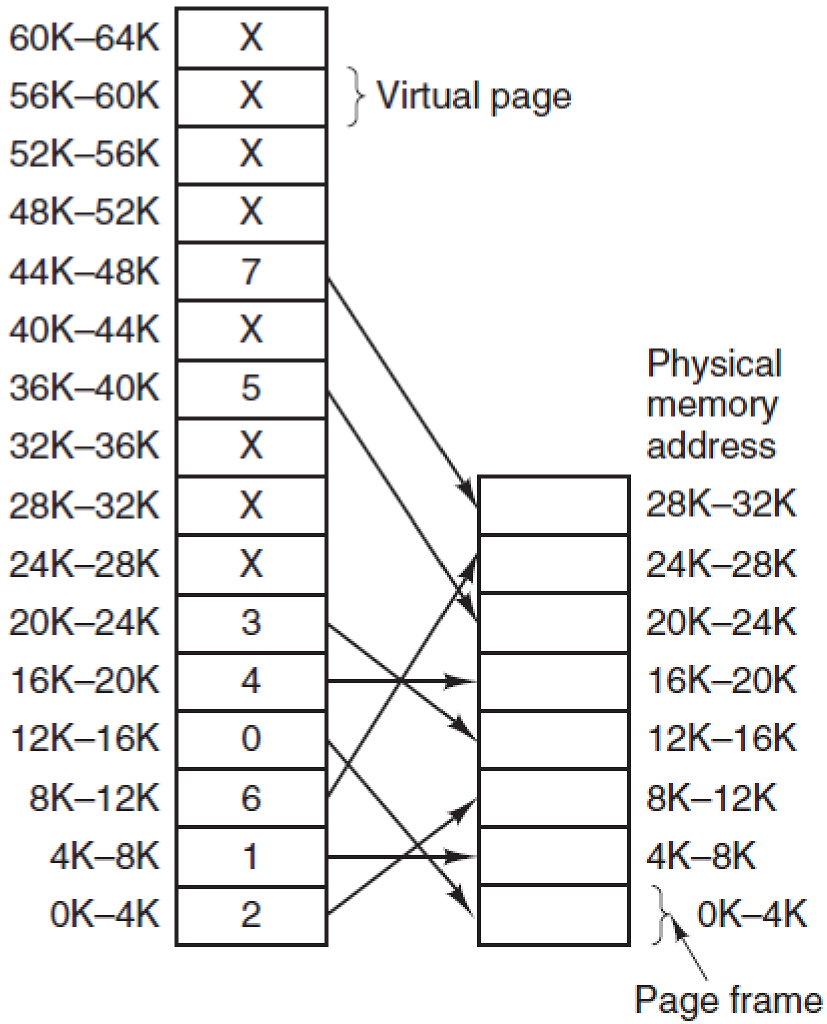
It's easy to follow translation examples in binary!!!



Paging Advantages

- Eliminates **external fragmentation**
- Easy to **implement**
- Easy to model **protection**
- Easy to model **sharing**
- **Easy to allocate** physical memory
  - Allocate a frame from list of free frames
- Leads naturally to **Virtual Memory**
  - It is not necessary to have the whole program ‘memory resident’
  - We can take pages that we don’t need off main memory to a page file somewhere (on disk)

Virtual Memory and Page Faults (link to next lecture)



- The **P** bit in the PTE of x86 keeps track of which pages are physically present in memory
- Pages marked with **X** are not memory resident, but kept on virtual memory (disk) – they have the **P** bit unset
- Requesting an unmapped page will cause the CPU to **trap** to the OS
- This trap is called a **Page Fault**
- **Week 8:** How we deal with Virtual Memory, page faults, strategies for caching, replacing pages, etc.

Some problems with Paging

- **P1: Page table is large**
  - 32 bit addresses & 4 KByte pages → 4 Mbyte per page table (assuming 4 bytes / entry)
  - **For Each Process!**
  - **Much worse for 64bit addresses**
  - But… are all 1 million Page Table Entries (PTEs) needed?
    - **No, sparse tables**
- **P2: Memory access is slow** (in CPU terms…)
  - TWO memory look-ups for each memory access
  - One look-up into the page table, one more to access the data in memory
- **Solutions** to these & more, Week 8

Summary

- CPU generates instructions in **logical space**
- Hardware sees **physical address space** (the memory we paid for)
- OS provides the illusion (abstraction) of an **address space owned by a process**
- **Memory Management** provides support for translation, relocation, protection, sharing, etc.
- Solutions need to consider issues such as efficient use of memory (**fragmentation**), **speed** (access to main memory is slow compared to CPU speed), …