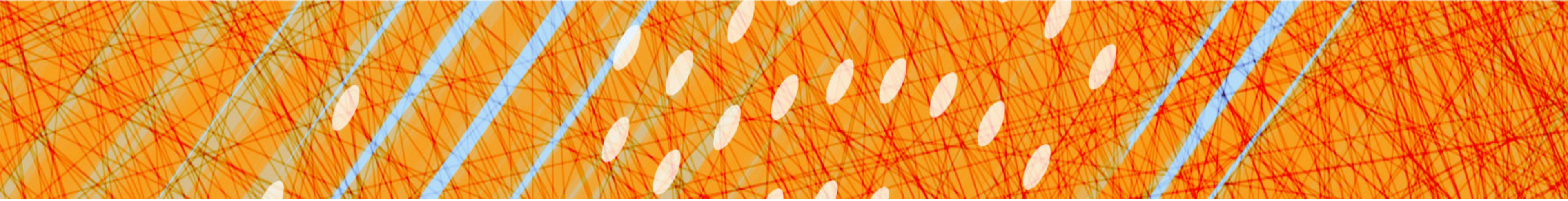


Main Memory: Address Translation

(Chapter 12-17)

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Can’t We All Just Get Along?

Physical Reality: different processes/threads share the same hardware → need to multiplex

- CPU cores (temporal)
- Memory and cache (spatial and temporal)
- Disk and devices (spatial and/or temporal)

Why worry about memory sharing?

- Complete working state of process and/or kernel is defined by its data (memory, registers, disk)
- Don’t want different processes to have access to each other’s memory (protection)

Aspects of Memory Multiplexing

Isolation

Don’t want distinct process states collided in physical memory (unintended overlap → chaos)

Sharing

Want option to overlap when desired (for efficiency and communication)

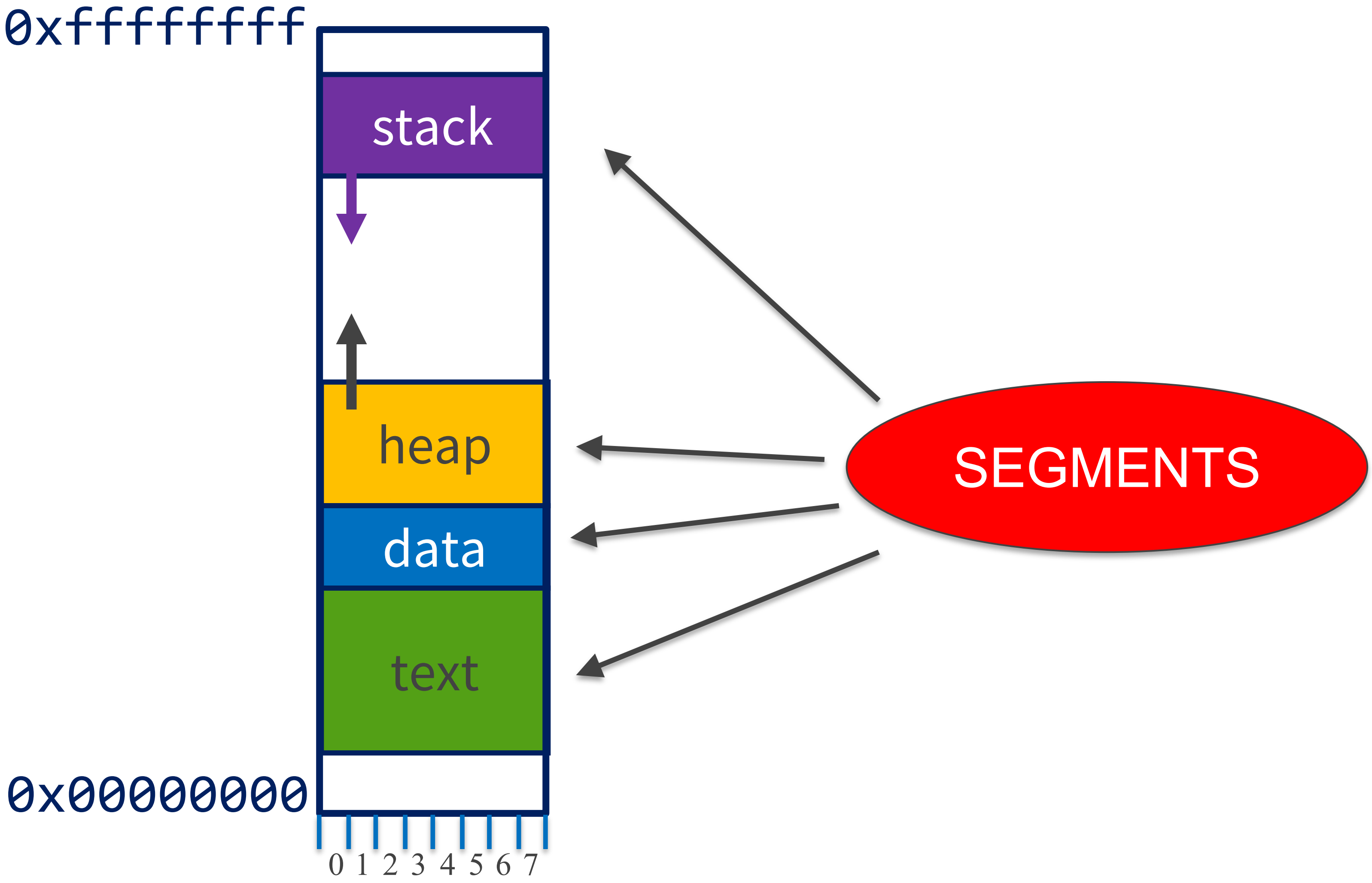
Virtualization

Want to create the illusion of more resources than exist in underlying physical system

Utilization

Want the best use of this limited resource

Virtual view of process memory



Where do we store virtual memory?

Need to find a place where the physical memory of the process lives

→Keep track of a “free list” of available memory blocks (so-called “holes”)

Dynamic Storage-Allocation Problem

- **First-fit**: Allocate *first* hole that is big enough
- **Next-fit**: Allocate *next* hole that is big enough
- **Best-fit**: Allocate *smallest* hole that is big enough; must search entire free list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate *largest* hole; must also search entire free list
  - Produces the largest leftover hole

Fragmentation

Internal Fragmentation

- allocated memory may be larger than requested memory; this size difference is memory internal to a partition, but not being used

External Fragmentation

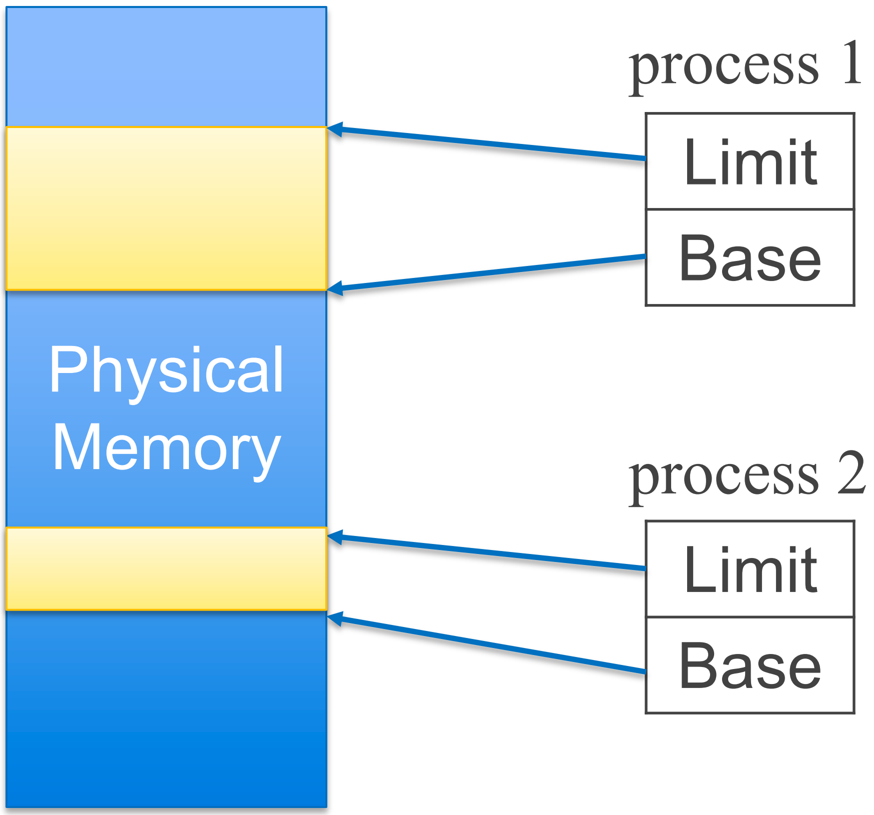
- total memory space exists to satisfy a request, but it is not contiguous

How do we map virtual → physical

- Having found the physical memory, how do we map virtual addresses to physical addresses?

Early Days: Base and Limit Registers

Base and **Limit** registers for each process

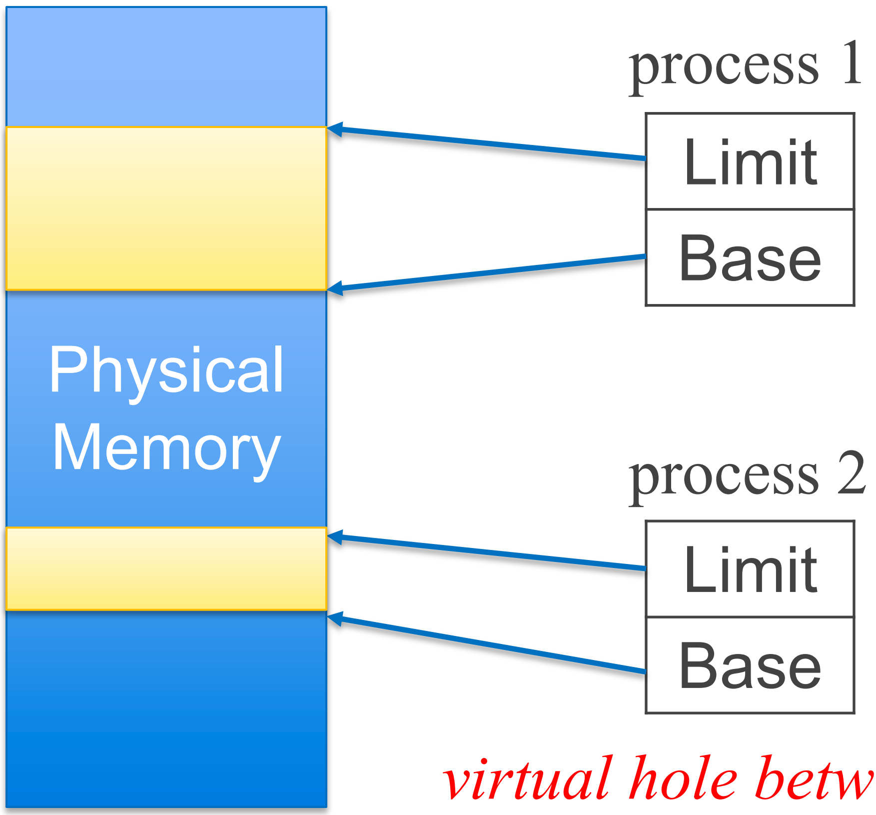


physical address =
virtual address + base

(segmentation fault if virtual address $\geq$ limit)

Early Days: Base and Limit Registers

Base and **Limit** registers for each process



Physical Memory

process 1

Limit

Base

process 2

Limit

Base

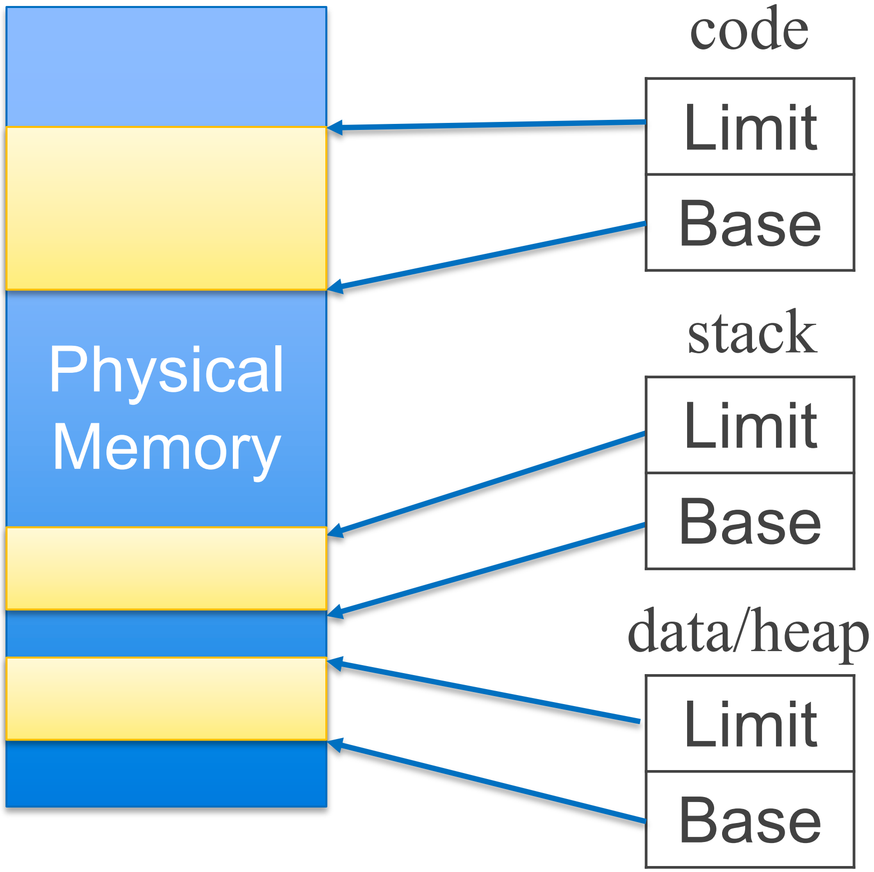
physical address =
virtual address + base

(segmentation fault if
virtual address ≥ limit)

virtual hole between heap and stack leads to significant **internal** *fragmentation*

Next: segmentation

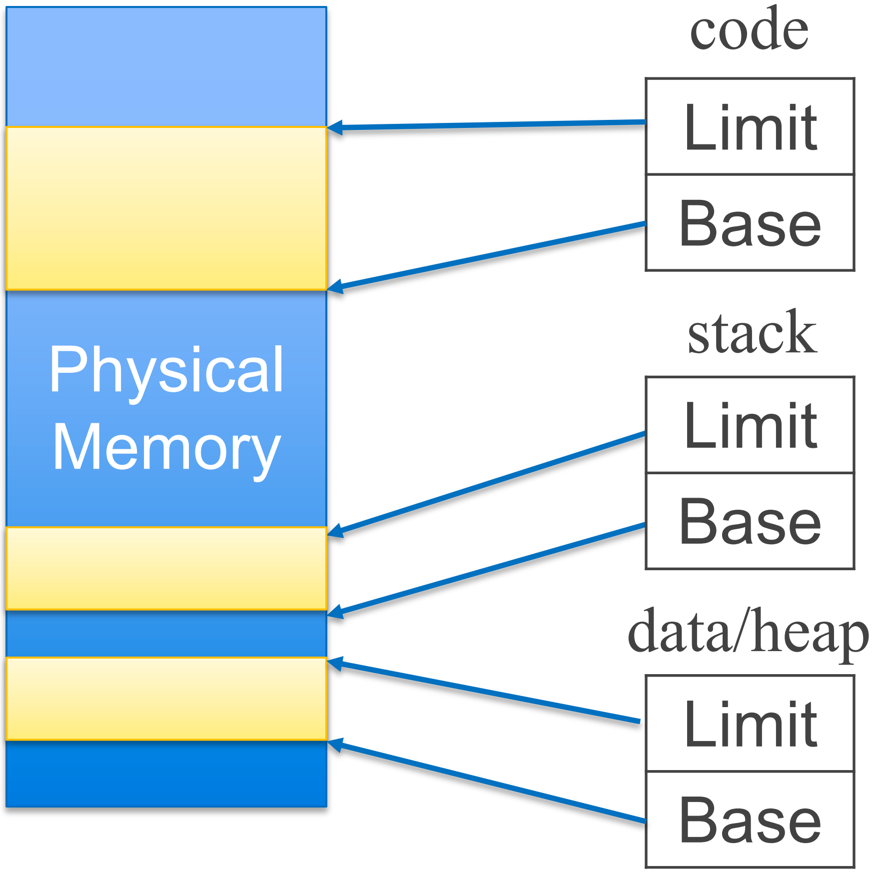
- **Base** and **Limit** register for each segment: code, data/heap, stack



physical address =
virtual address
− virtual start of segment
+ base

Next: segmentation

- **Base** and **Limit** register for each segment: code, data/heap, stack



Physical Memory

code

Limit

Base

stack

Limit

Base

data/heap

Limit

Base

physical address =
virtual address
− virtual start of segment
+ base

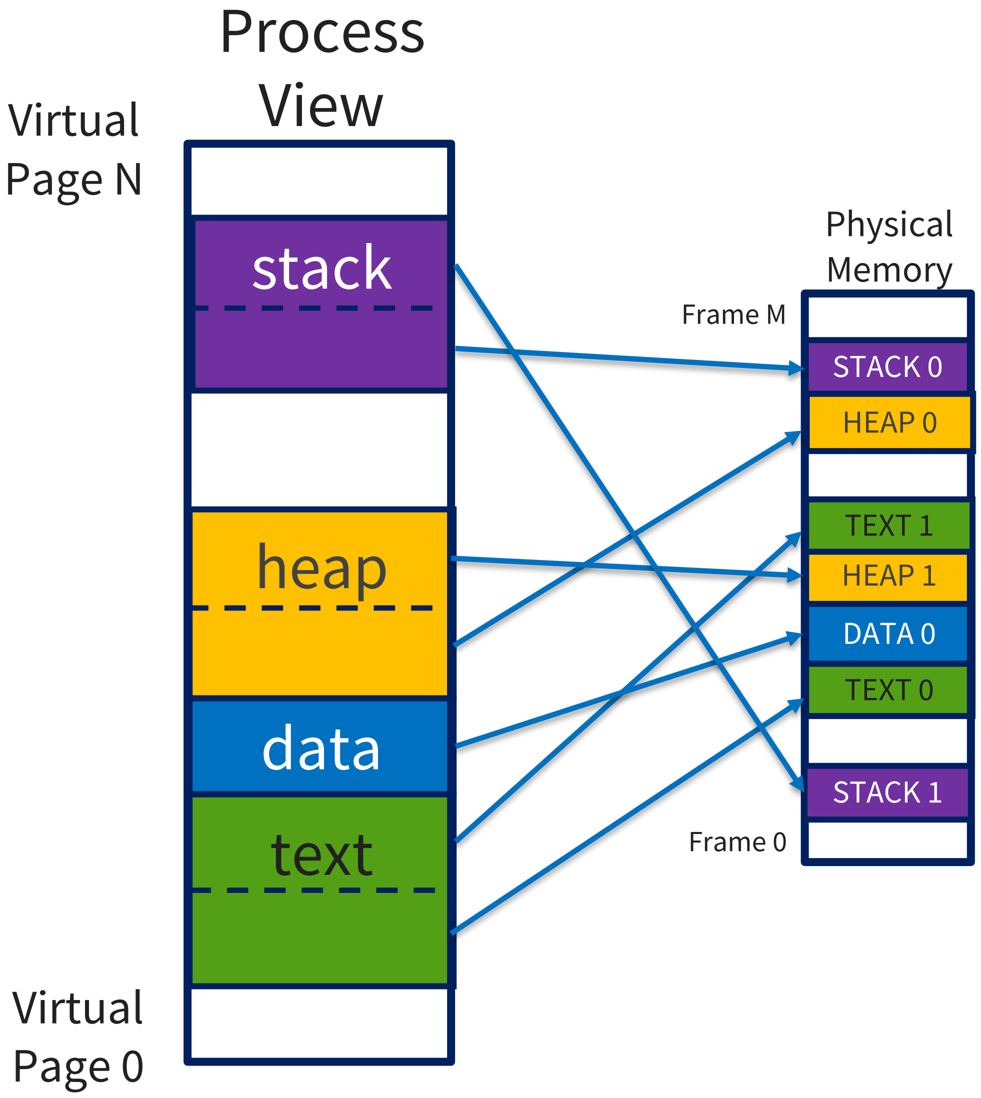
physical holes between segments leads to significant ***external*** *fragmentation*

Paged Translation

TERMINOLOGY ALERT:

Page: virtual

Frame: physical



Solves both internal and external fragmentation! (to a large extent)

Paging Overview

Divide:

- Physical memory into fixed-sized blocks called **frames**
- Virtual memory into blocks of same size called **pages**

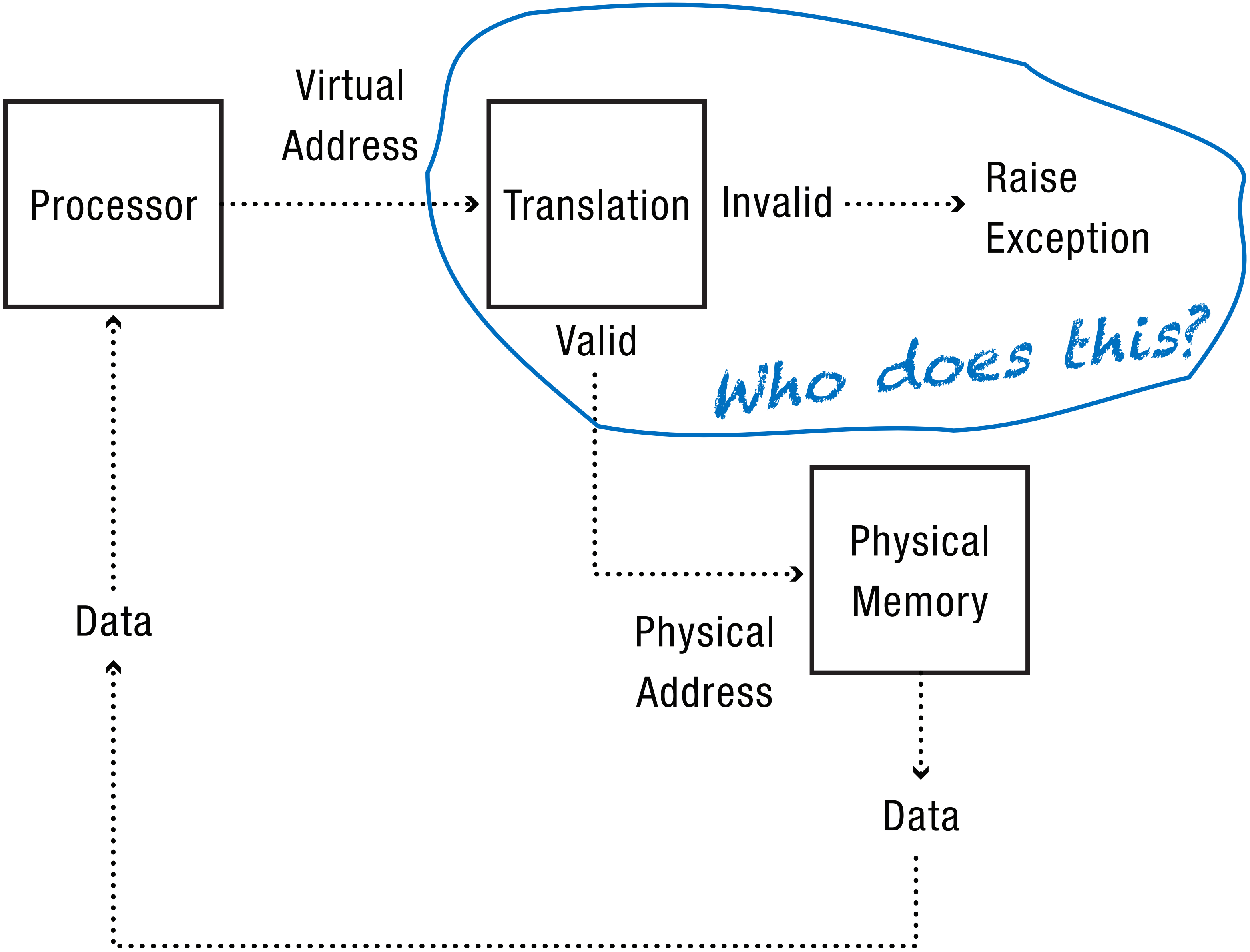
Management:

- Keep track of which pages are mapped to which frames
- Keep track of all free frames

Notice:

- Not all pages of a process may be mapped to frames

Address Translation, Conceptually



Memory Management Unit (MMU)

- Hardware device
- Maps virtual to physical address (used to access data)

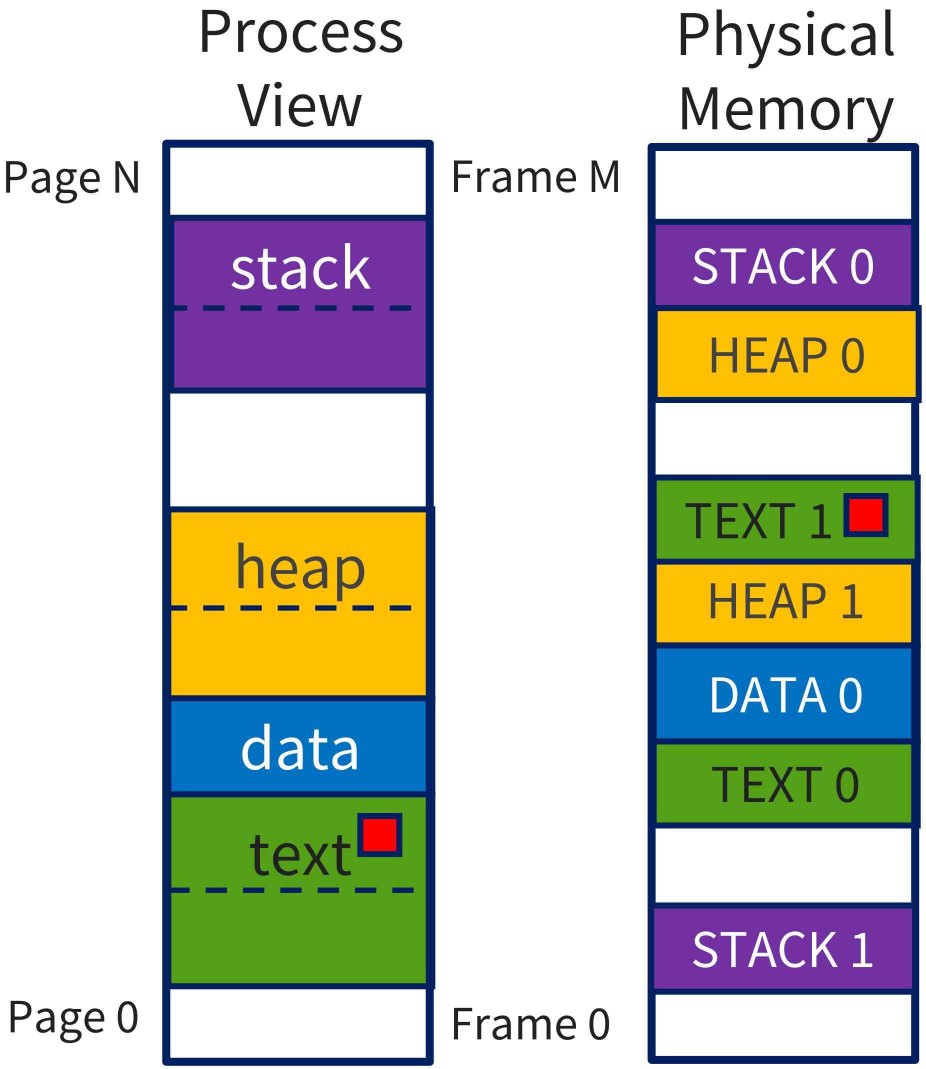
User Process:

- deals with *virtual* addresses
- Never sees the physical address

Physical Memory:

- deals with *physical* addresses
- Never sees the virtual address

High-Level Address Translation



🟥 red cube is 255th byte in page 2.

Where is the red cube in physical memory?

Virtual Address Components

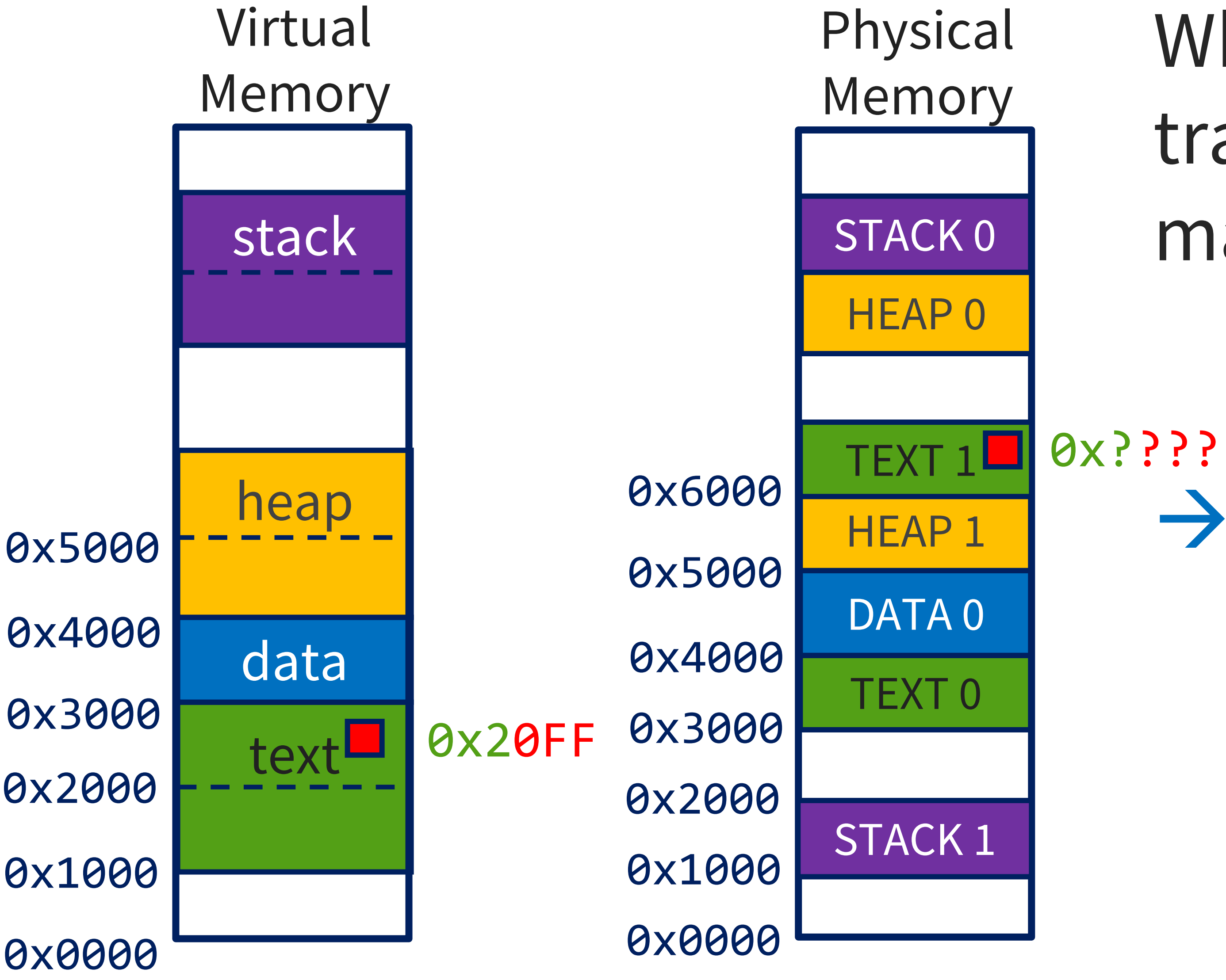
Page number – Upper bits (most significant bits)
- Must be translated into a physical frame number

Page offset – Lower bits (least significant bits)
- Does not change in translation

page number	page offset
$m - n$	n

For given logical address space 2^m *and page size* 2^n

High-Level Address Translation

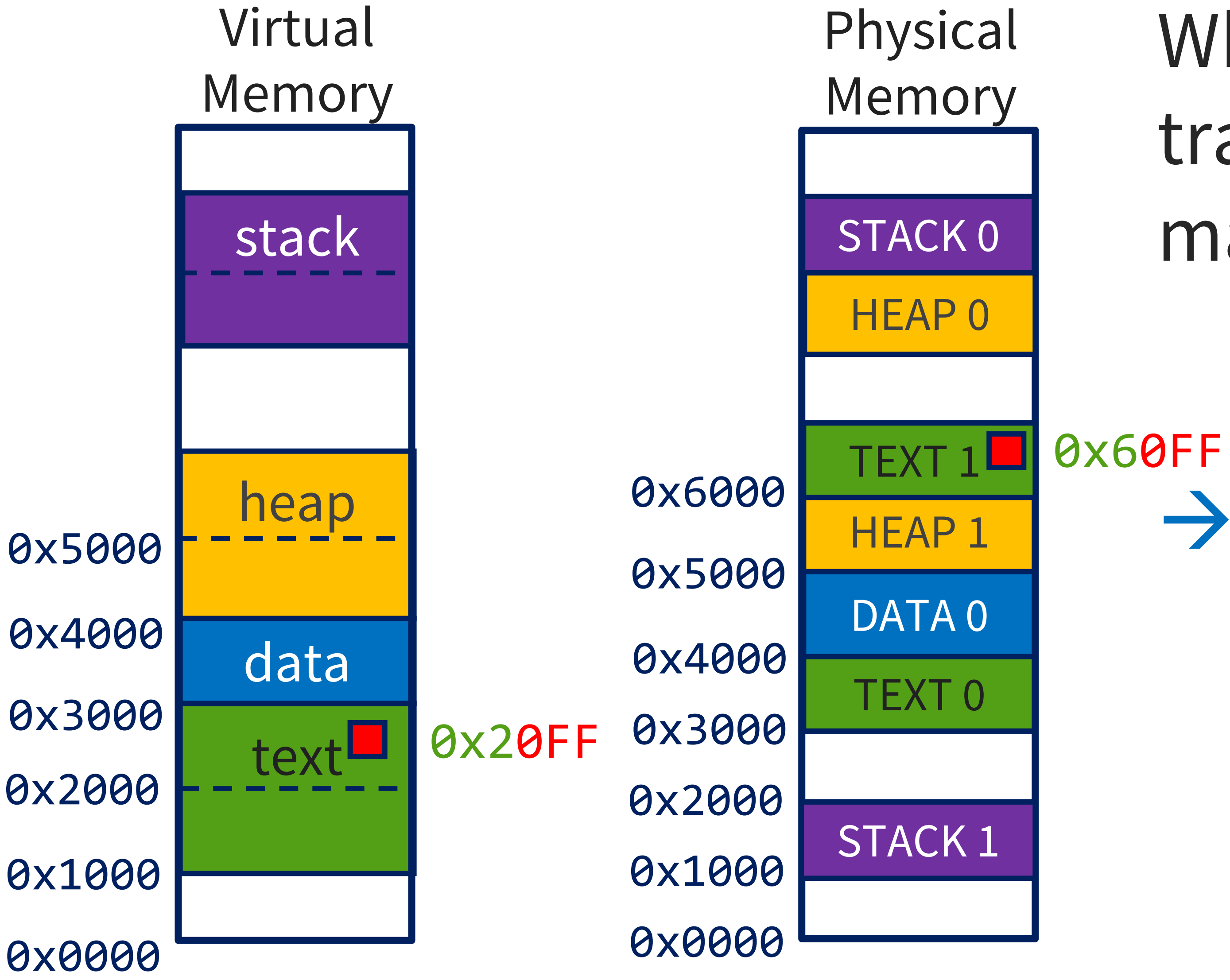


Who keeps track of the mapping?

→ Page Table

0	-
1	3
2	6
3	4
4	8
5…	5

High-Level Address Translation

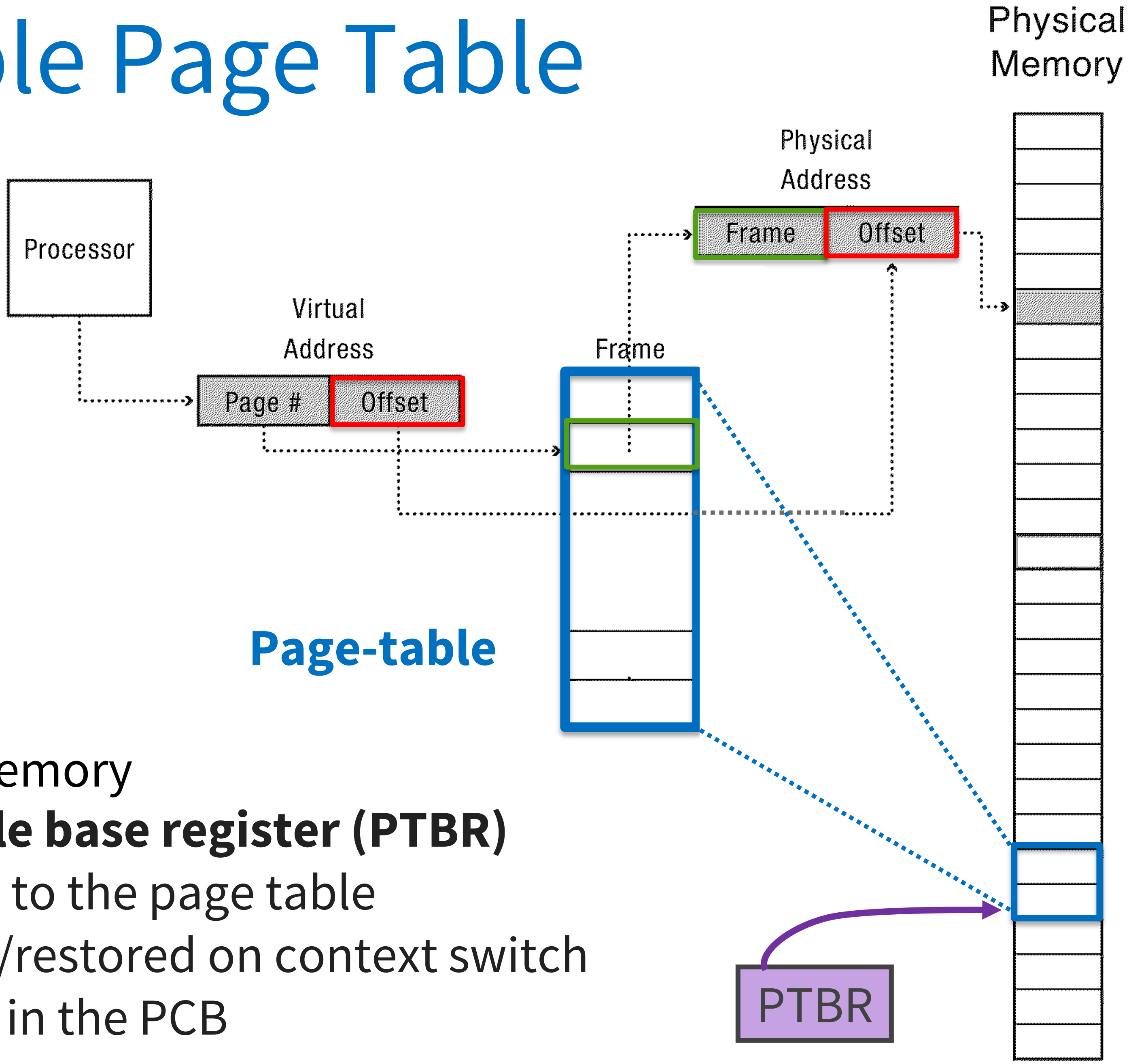


Who keeps track of the mapping?

→ Page Table

0	-
1	3
2	6
3	4
4	8
5…	5

Simple Page Table



Lives in Memory

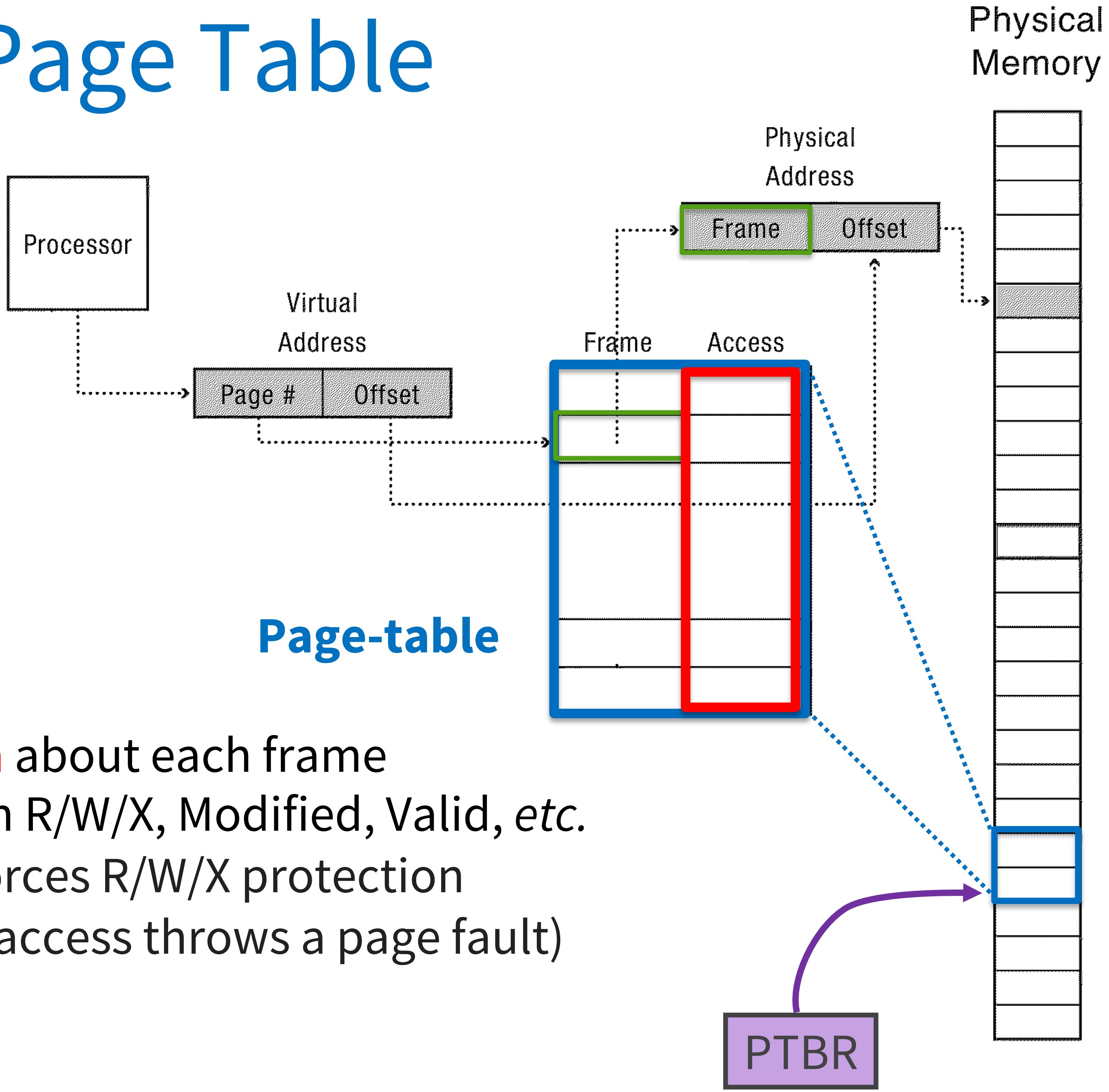
Page-table base register (PTBR)

- Points to the page table
- Saved/restored on context switch
- Saved in the PCB

Leveraging Paging

- Protection
- Demand Loading
- ~~Copy-On-Write~~

Full Page Table



Page-table

Meta Data about each frame

Protection R/W/X, Modified, Valid, *etc.*

MMU Enforces R/W/X protection
(illegal access throws a page fault)

Leveraging Paging

- Protection
- Demand Loading
- ~~Copy On Write~~

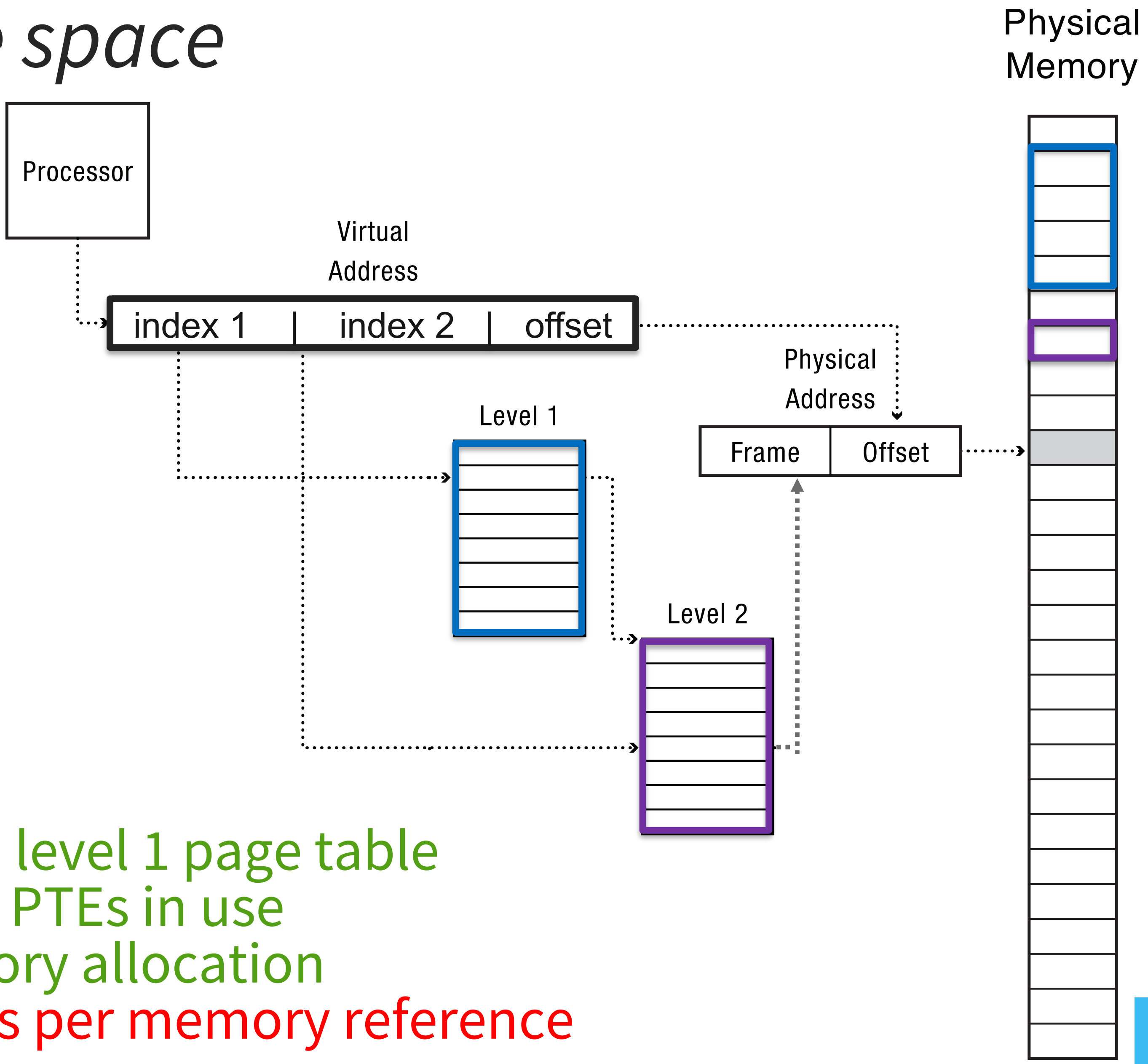
Demand Loading

- Page not mapped until it is used
- Requires free frame allocation
  - What if there is no free frame???
- May involve reading page contents from disk or over the network

Address Translation

- Paged Translation
- Efficient Address Translation
  - Multi-Level Page Tables
  - ~~Inverted Page Tables~~
  - TLBs

Multi-Level Page Tables to reduce page table space



PTBR points to level 1 page table
+ Allocate only PTEs in use
+ Simple memory allocation
− ***more*** lookups per memory reference

Two-Level Paging Example

32-bit machine, 1KB page size

- Logical address is divided into:
  - a page offset of 10 bits (1024 = 2^{10})
  - a page number of 22 bits (32-10)
- Since the page table is paged, the page number is further divided into (say):
  - a 12-bit first index
  - a 10-bit second index
- Thus, a logical address is as follows:

page number		page offset
index 1	index 2	offset
12	10	10

Another example

- Suppose 32-bit virtual address, 2-level page table
  - So, address is 10+10+12 bits
- Page Table Entry (PTE) is 32 bits (4 bytes)
  - convenient: PTE is the size of a word
- Frame number in PTE is 22 bits
- What is the page size?
  - Answer: $2^{12} = 4096$ bytes
- What is the frame size?
  - Answer: same
- How many pages in the virtual address space?
  - Answer: $2^{10} \times 2^{10} = 2^{20}$
- How many PTEs in the first-level page table?
  - Answer: $2^{10} = 1024$
- How many PTEs in the second-level page table?
  - Answer: same
- How large is a page table?
  - Answer: $2^{10} \times 4 = 4$ kilobytes
    - – conveniently fits in a frame, which simplifies allocation
- What is the maximal physical memory size?
  - Answer: $2^{22} \times 2^{12} = 2^{34} = 16$ gigabytes

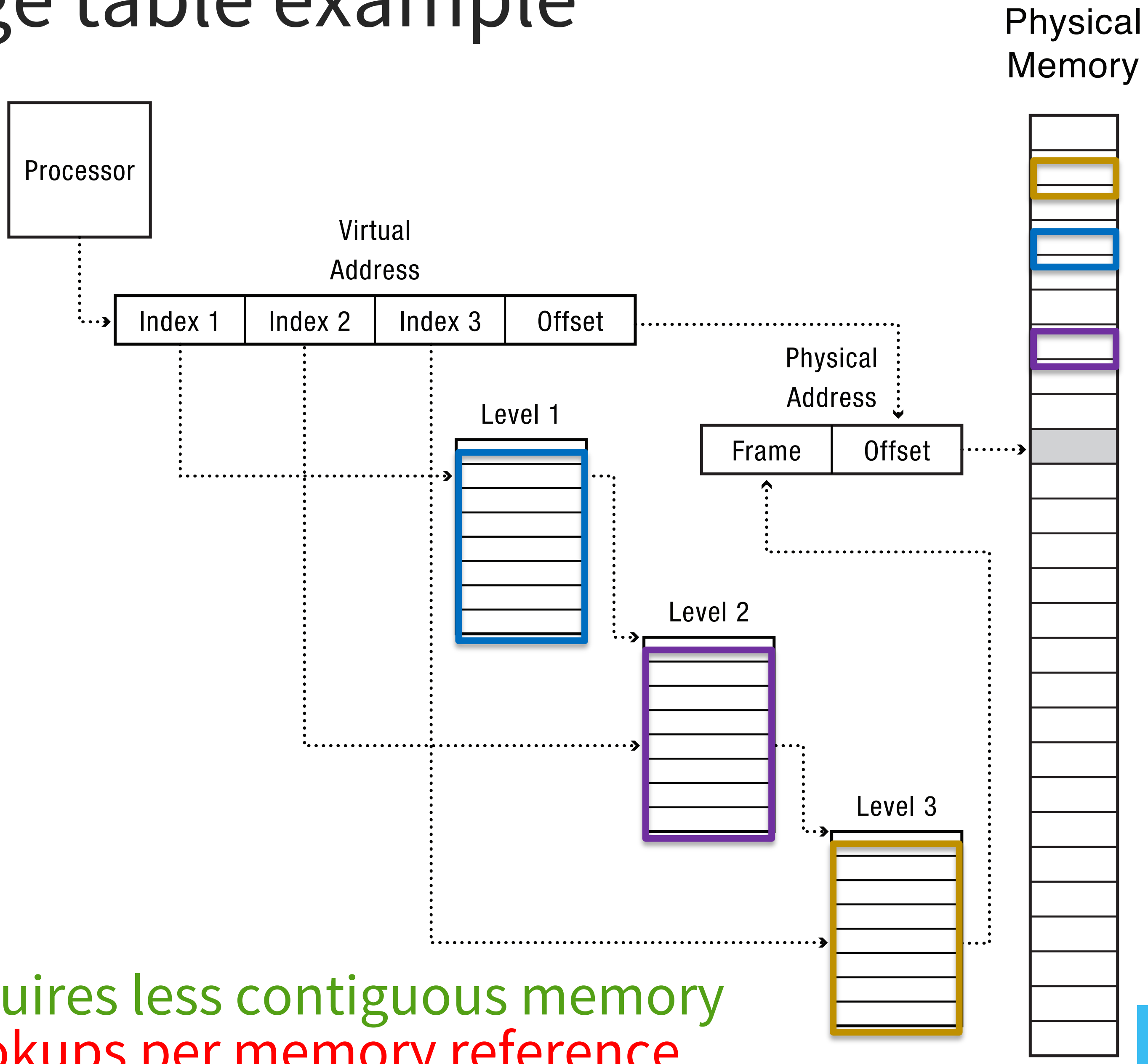
Another example, continued

- In our example, a page table fits in a frame
- Suppose now that we only need to map the following pages:
  - 1: text
  - 2: data + heap
  - $2^{20} - 1$: stack
- How many frames do we need to allocate?
  - Answer: 6
    - 1 for the first level page table
    - 1 second level page table for pages 1 and 2
    - 1 second level page table for page $2^{20} - 1$
    - 3 frames for each of the pages
- How many memory accesses are needed to read a word in virtual memory?
  - Answer: 3 (see next slide)

Another example, continued

- How to read word at address 0x12345678?
  - assuming this address is mapped
  - Offset is 0x678 (12 bits)
  - Page number is 0x12345 (20 bits)
  - Split into two 10 bit indices:
    - 0001 0010 0011 0100 0101 →
      - index1 = 0001001000 = 0x048
      - index2 = 1101000101 = 0x345
  - Load entry 0x048 in first-level page table:
    - @address PTBR + 0x048 → X (frame number of next PT)
  - Load entry 0x345 in second-level page table:
    - @address X + 0x345 → Y (frame number)
  - Load word @address Y + 0x678
  - Note: math didn't include some right shifts for readability

3 level page table example



+ First Level requires less contiguous memory

− ***even more*** lookups per memory reference

Complete Page Table Entry (PTE)

Present	Protection R/W/X	Ref	Dirty	Index

Index is an index into (depending on Present bit):

- frames
  - physical process memory or next level page table
- backing store
  - if page was swapped out

Synonyms:

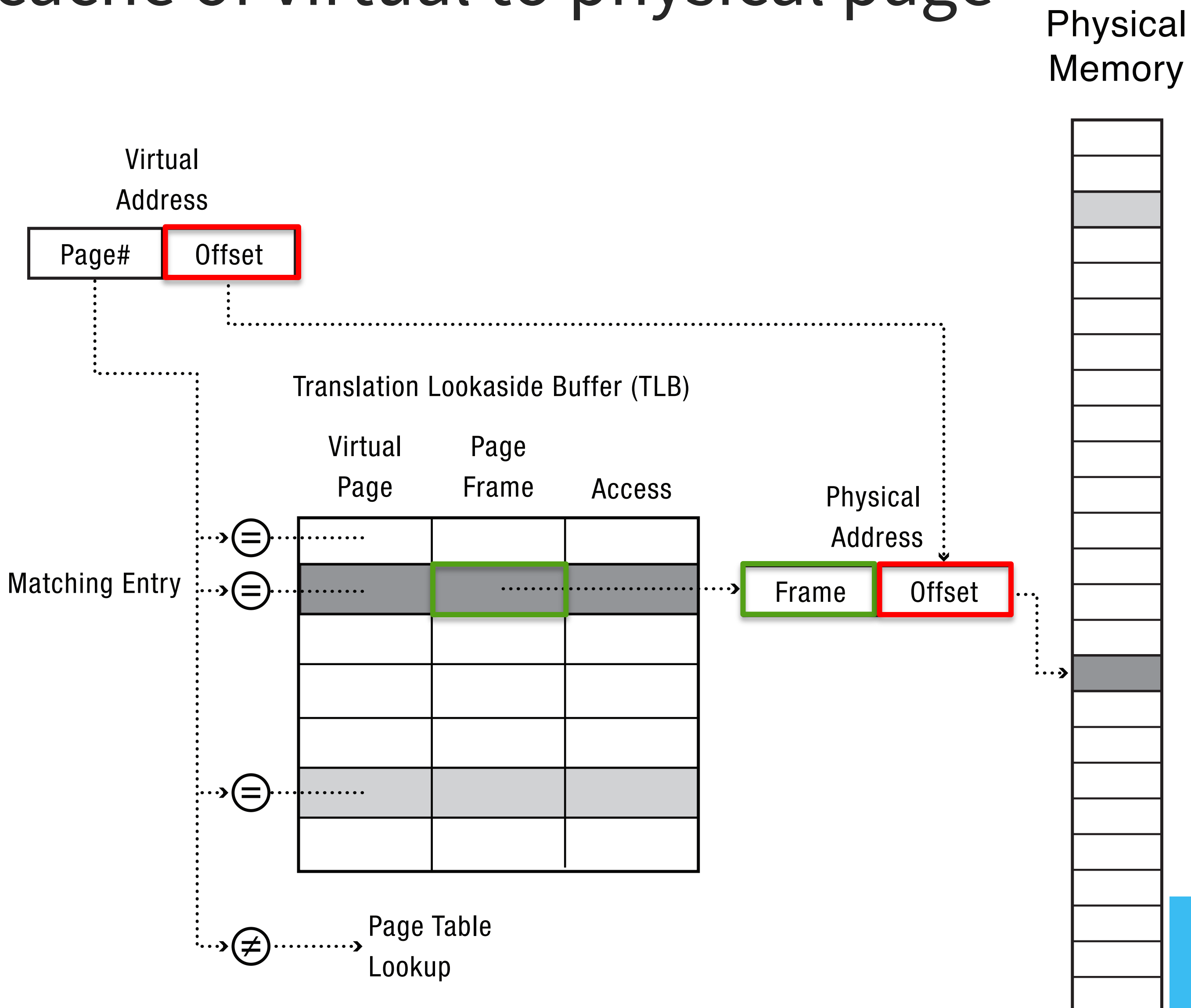
- Present bit == Valid bit
- Dirty bit == Modified bit
- Referenced bit == Accessed bit
- Index == offset

Address Translation

- Paged Translation
- Efficient Address Translation
  - Multi-Level Page Tables
  - ~~Inverted Page Tables~~
  - TLBs

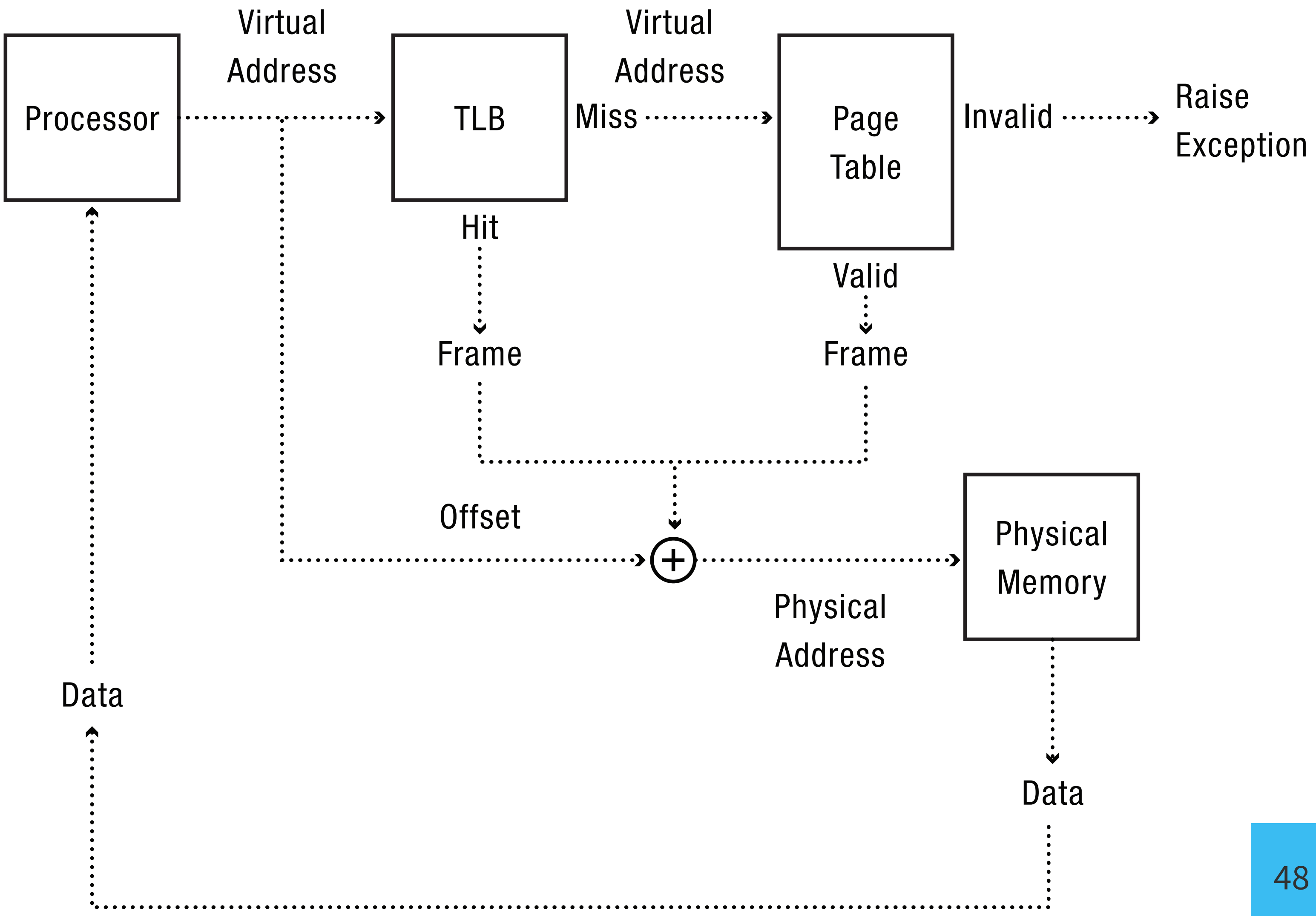
Translation Lookaside Buffer (TLB)

Associative cache of virtual to physical page translations



Address Translation with TLB

Access TLB before you access memory.



Why not just have a large TLB?

Why not just have a large TLB?

- TLBs are fast because they are small

Software vs. Hardware-Loaded TLB

- Software-loaded: TLB-miss → software handler
- Hardware-loaded: TLB-miss → hardware ”walks” page table itself
  - may lead to “page fault” if page is not in memory

Address Translation Uses!

Process isolation

- Keep a process from touching anyone else’s memory, or the kernel’s

Efficient inter-process communication

- Shared regions of memory between processes

Shared code segments

- common libraries used by many different programs

Program initialization

- Start running a program before it is entirely in memory

Dynamic memory allocation

- Allocate and initialize stack/heap pages on demand

MORE Address Translation Uses!

Program debugging

- Data breakpoints when address is accessed

Memory mapped files

- Access file data using load/store instructions

Demand-paged virtual memory

- Illusion of near-infinite memory, backed by disk or memory on other machines

Checkpointing/restart

- Transparently save a copy of a process, without stopping the program while the save happens

Distributed shared memory

- Illusion of memory that is shared between machines