

Virtual Memory and Page Tables

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Sid Sannapareddy Maya Huizar

Logistics

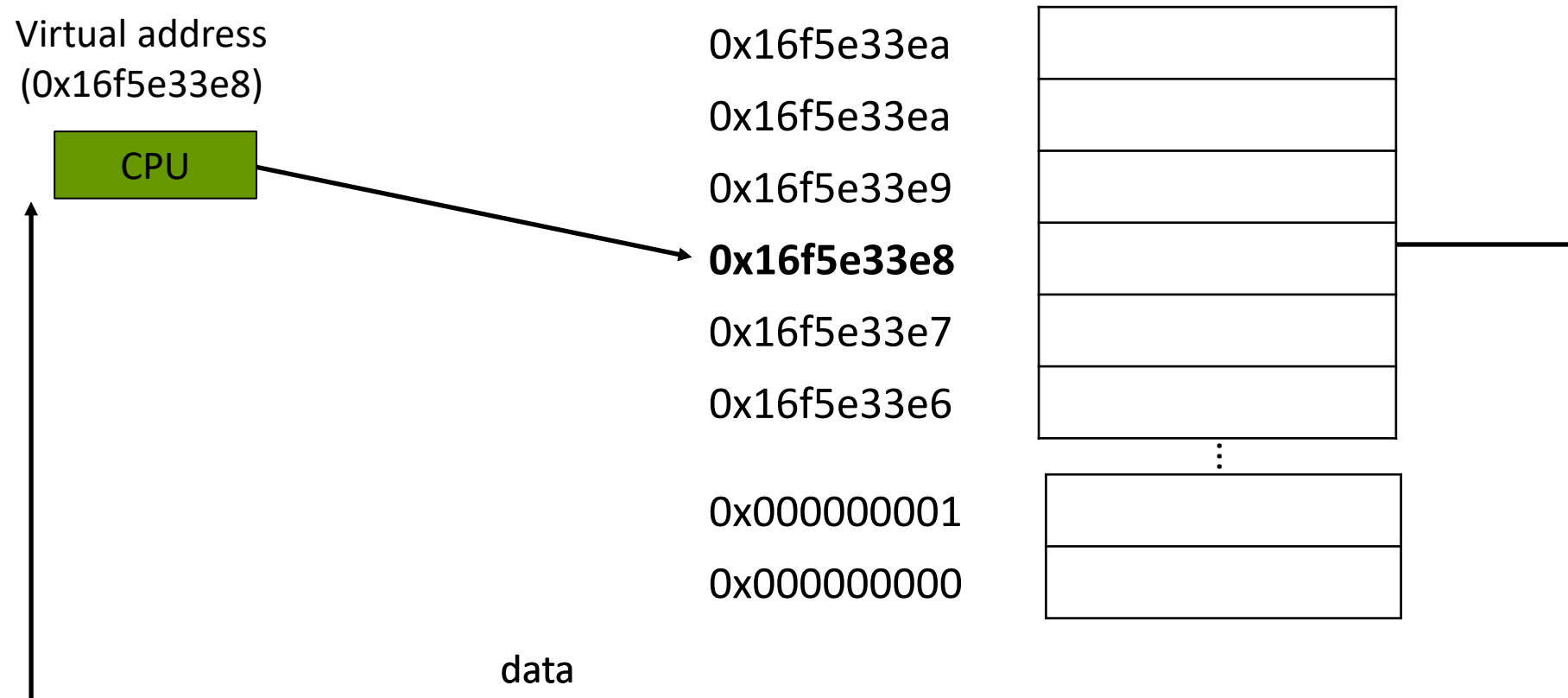
- ❖ **Meet with your TAs by the EOD Friday to present Milestone 1!**
 - Milestone 1 is not the end of the world – communicate with us if anything comes up.
- ❖ **Check-In Released Wednesday @ 1:30 PM, Due Monday @ 12 PM**

Lecture Outline

- ❖ **High Level Refresher**
- ❖ TLB
- ❖ Page Table Details
- ❖ Multi-Level Page Tables
- ❖ Inverted Page Tables

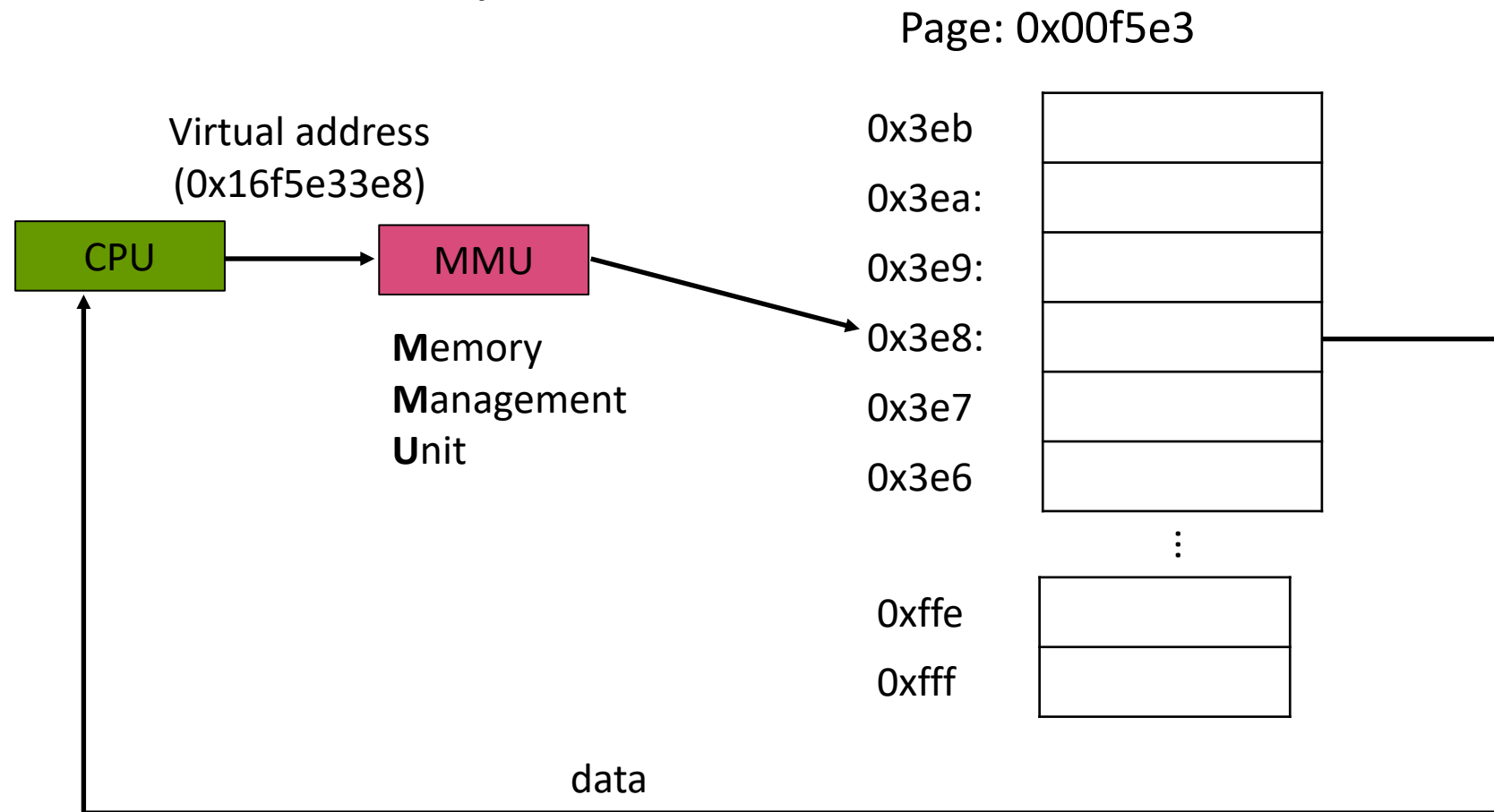
Direct Addressing

- ❖ The CPU directly uses an address to access a location in memory
 - Creates several different issues....



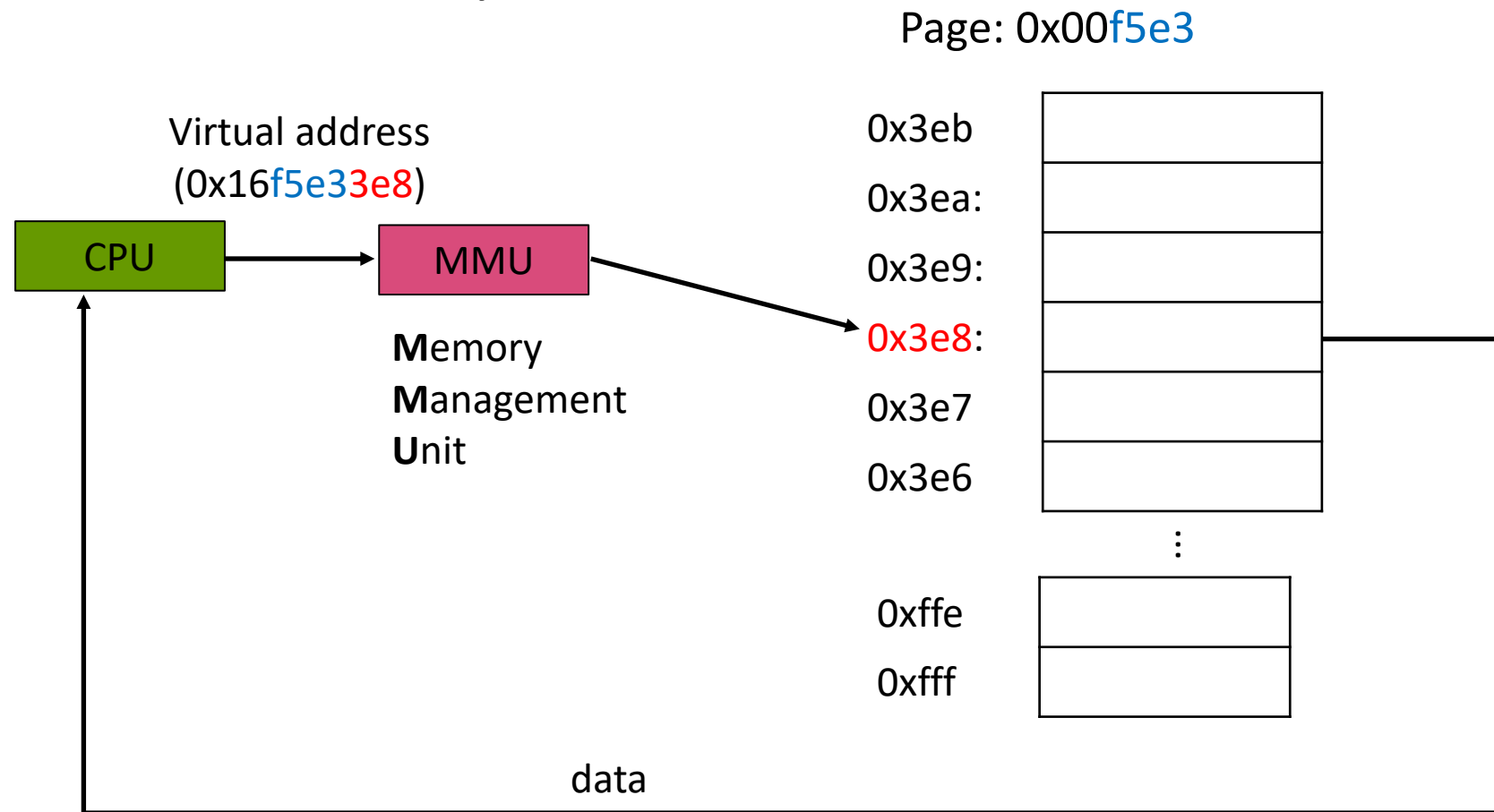
Virtual Memory: Translation

- ❖ Programs don't know about physical addresses; virtual addresses are translated into them by the MMU



Virtual Memory: Translation

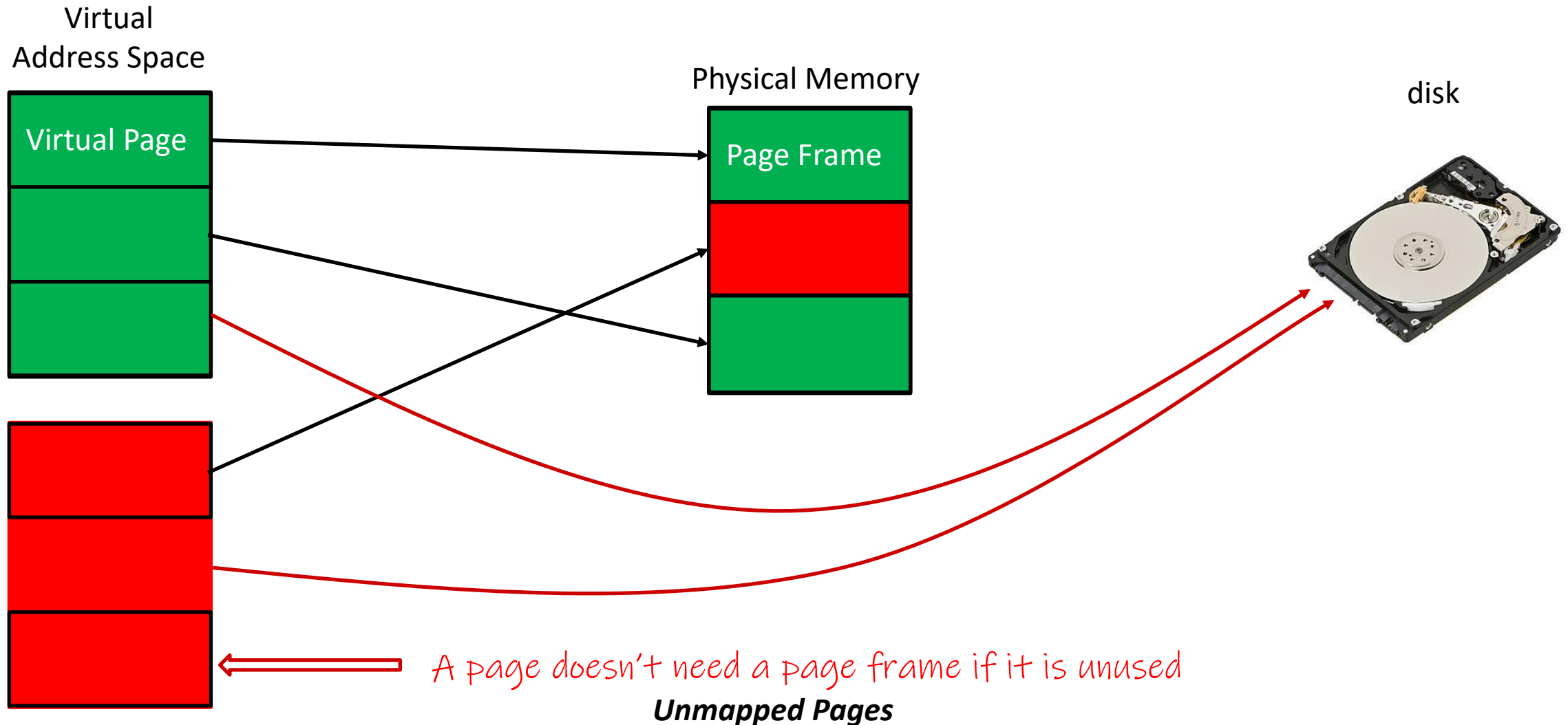
- ❖ Programs don't know about physical addresses; virtual addresses are translated into them by the MMU



The Virtual Address is manipulated to create the Physical Address.

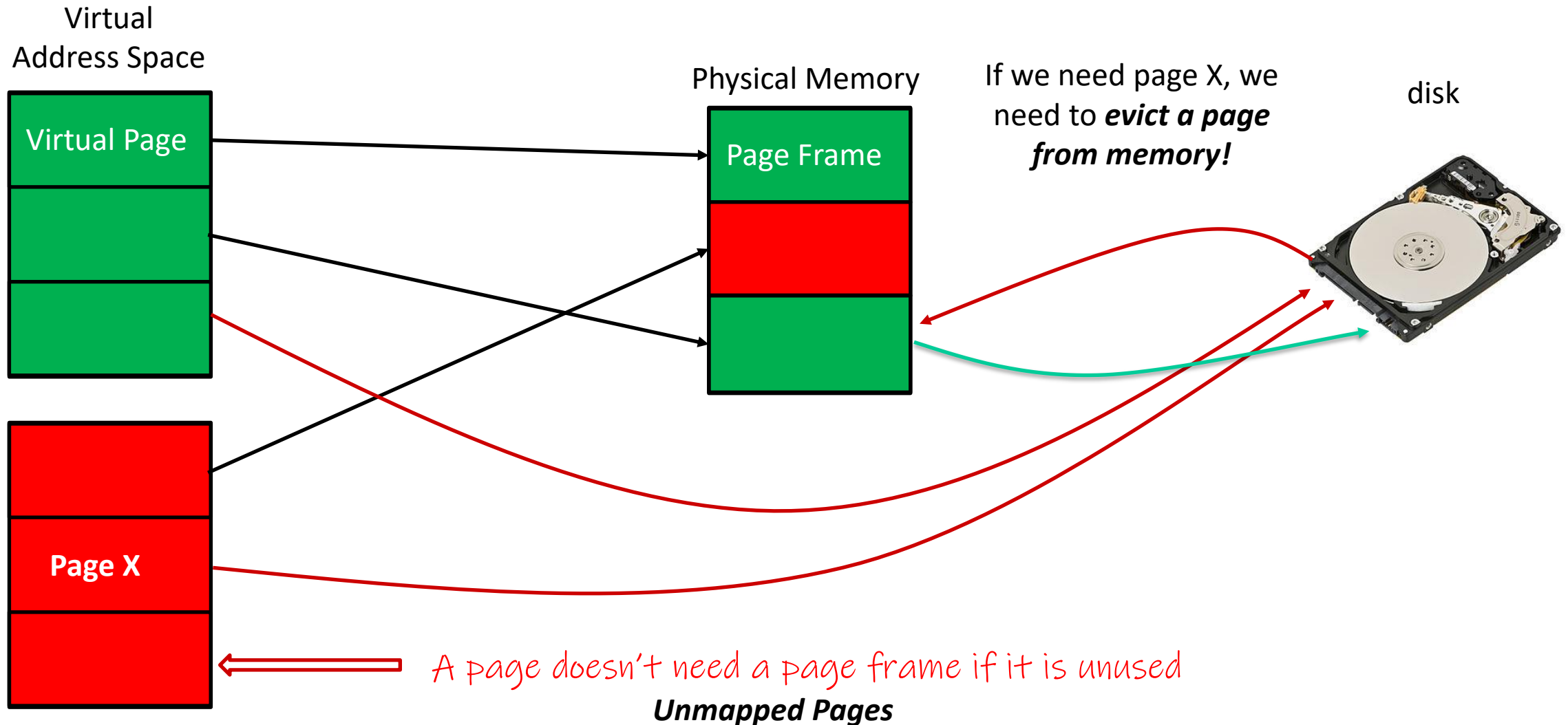
Virtual Pages and Physical Page Frames

- ❖ Memory is divided into fixed-size units called **pages**



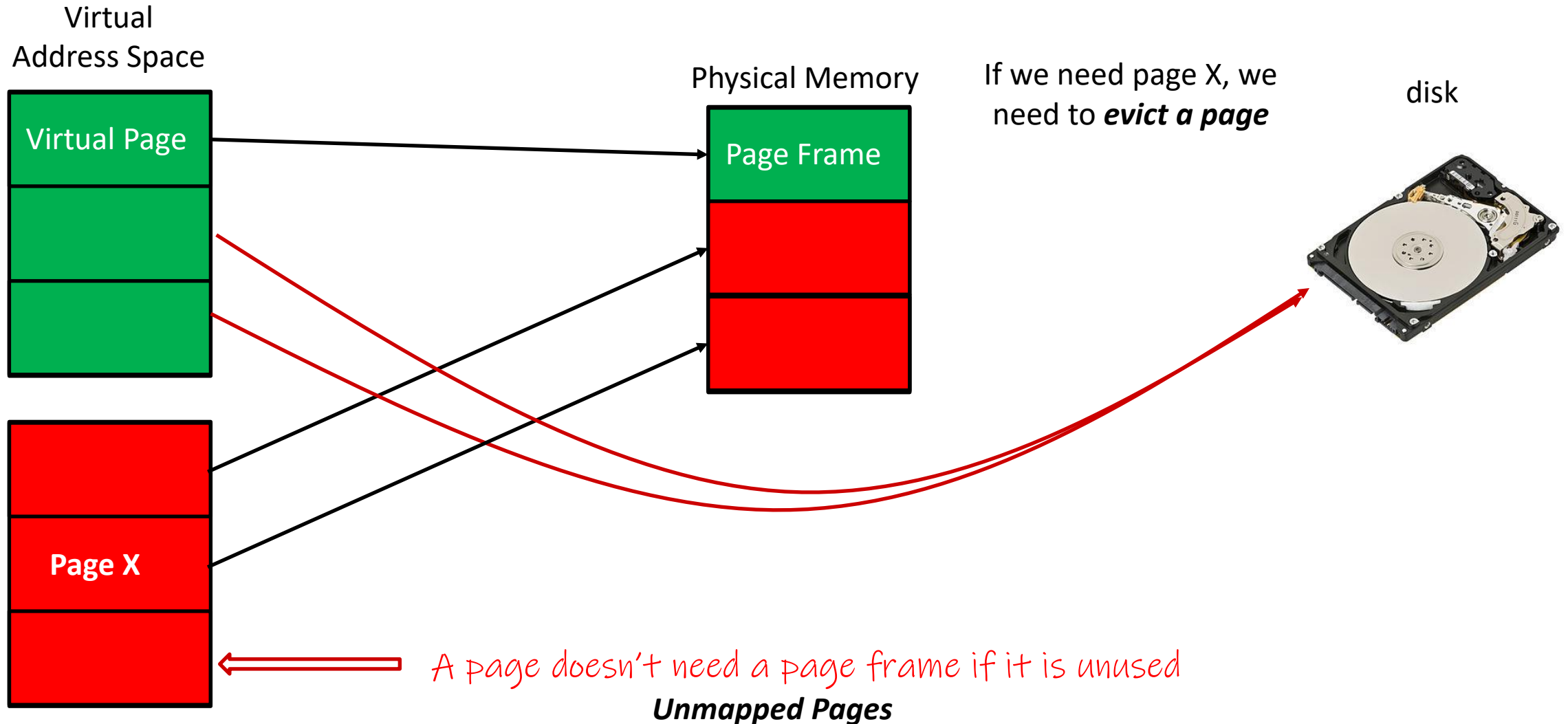
Virtual Pages and Physical Page Frames

- ❖ Memory is divided into fixed-size units called **pages**



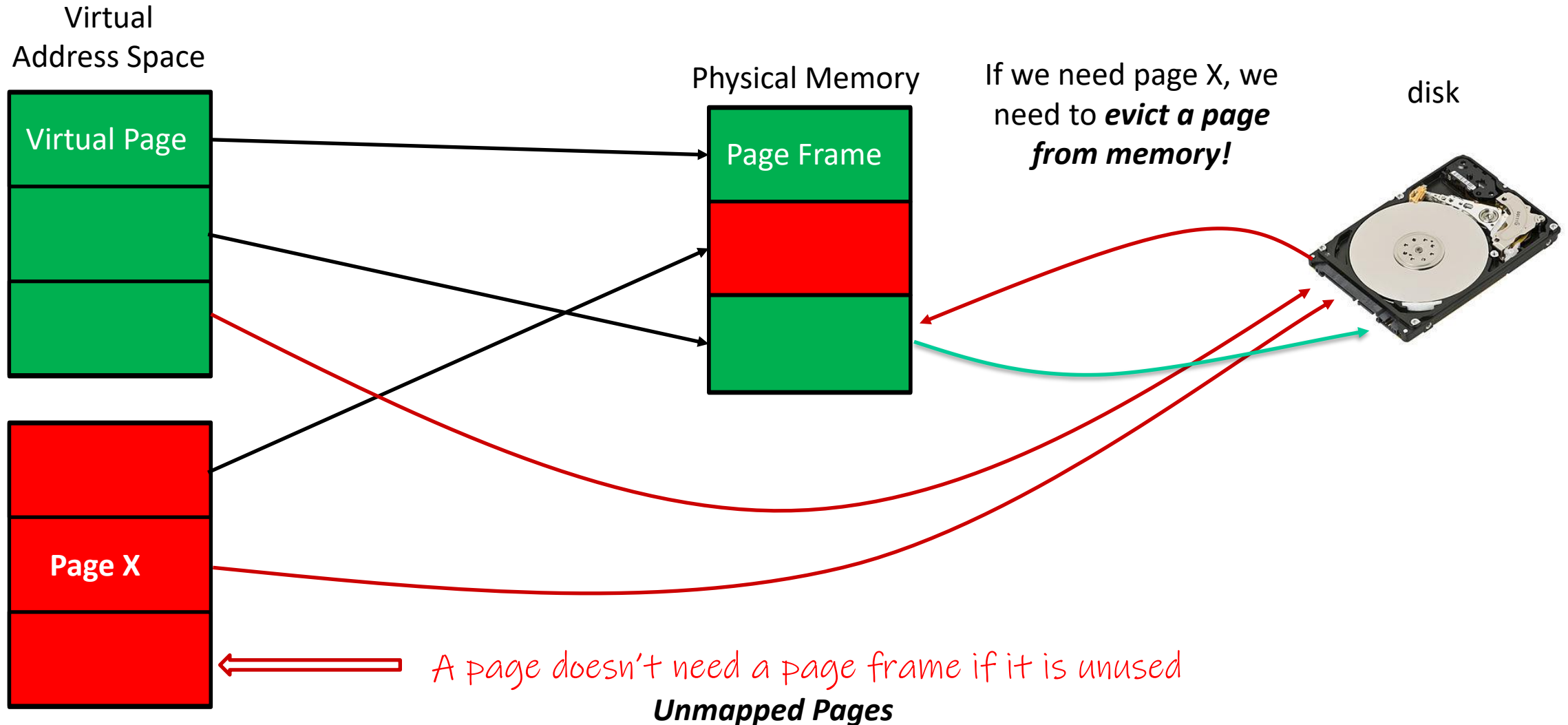
Virtual Pages and Physical Page Frames

- ❖ Memory is divided into fixed-size units called **pages**



Virtual Pages and Physical Page Frames

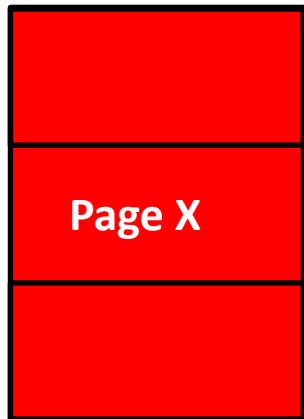
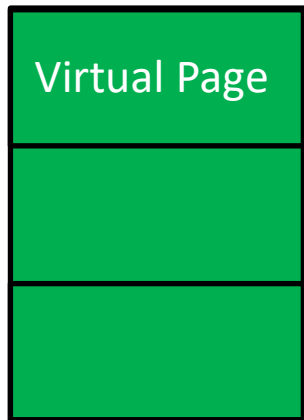
- ❖ Memory is divided into fixed-size units called **pages**



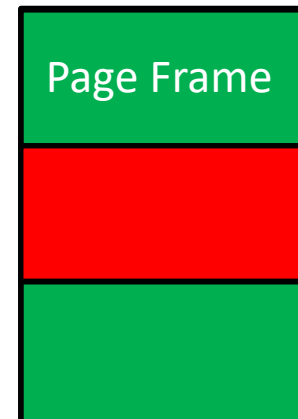
Virtual Pages and Physical Page Frames

- ❖ Memory is divided into fixed-size units called **pages**

Virtual
Address Space



Physical Memory



*Writing a Page to Disk
is an expensive task*

disk



💡 Only pages **with content**
are moved to disk (swap file).

*Why would you write
empty pages to disk?*

If the green process was allocated the page but never wrote to it,
do we still need to write the page to disk?

Page Tables

- ❖ Virtual addresses are converted into physical addresses via a page table.
- ❖ Each process ***needs its own mapping*** so there is a page table per process

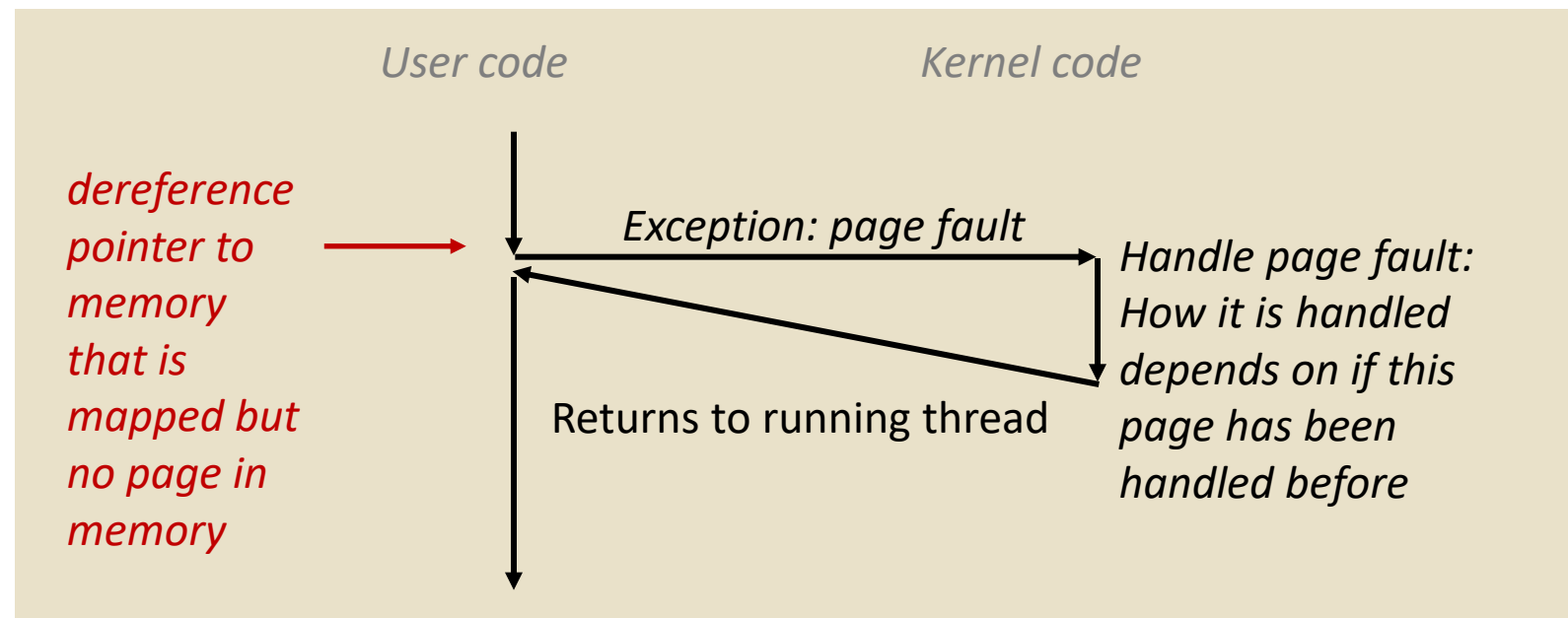
Virtual page #	Valid	Physical Page Frame
0x00f5e5	0	-----
0x00f5e4	1	0x00a1b2
0x00f5e3	1	0x00c3d4
0x00f5e2	0	0x00e5f6

Valid determines if the
page is in physical memory

If a page is on disk,
it will be fetched

Page Fault Exception

- ❖ An *Exception* is a transfer of control to the OS *kernel* in response to some *synchronous event* (*directly caused by what was just executed*)
- ❖ In this case, writing to a memory location that is not in physical memory currently



In this example, a variable that points to a virtual address that is mapped (but not in memory) will trigger a page fault.

If you want to know what virtual address is an invalid virtual address, 0x0 is a good example.

Types of Addresses

❖ Virtual Address:

- Used to refer to a location in a virtual address space
 - Think the typical Stack and Heap diagram.
- Used by CPU during pointer arithmetic and used by our programs.

❖ Physical Address

- Refers to a location in physical memory
- Virtual addresses are converted to physical addresses *when we perform accesses to memory.*

Page Offset

- ❖ **Typically, Pages are 4096 bytes in size (2^{12} bytes).**
 - In the real world, this size is configurable (e.g. 1MB Pages....)
- ❖ On most modern machines, memory is **byte-addressable**.
 - Each individual byte has a unique address.

Questions Worth Pondering

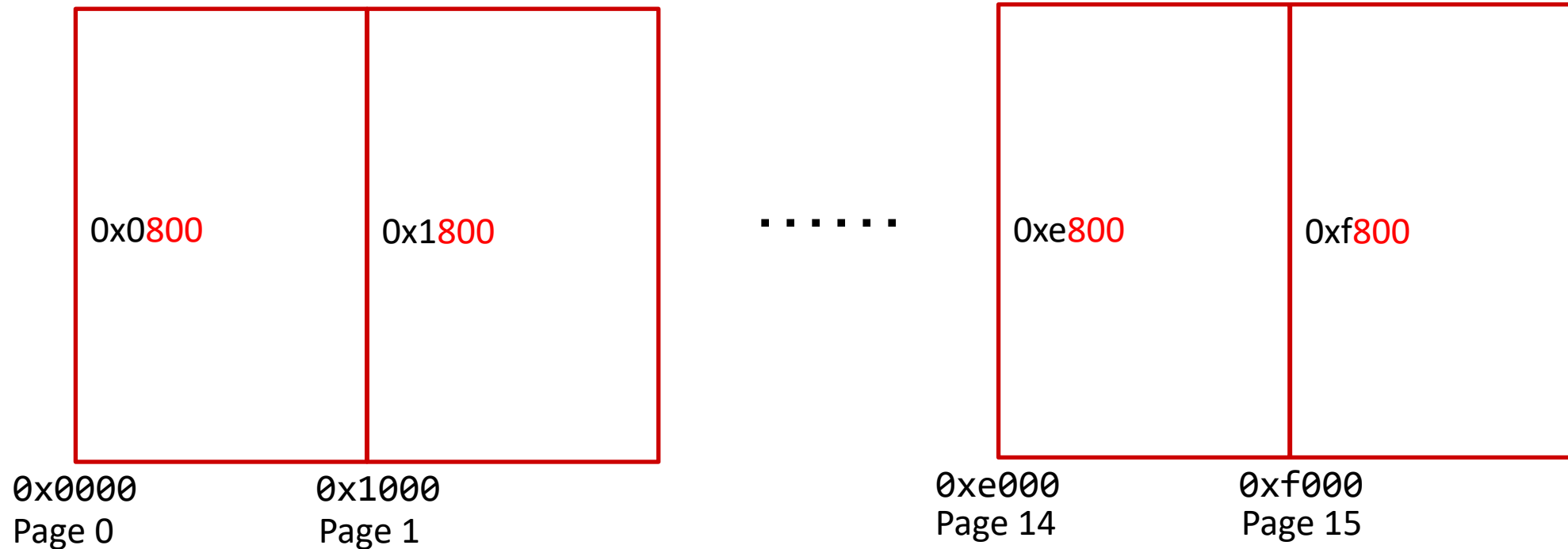
How many different addresses belong to the same page?
(i.e. how many values can we access within a page)

How many bits are needed to specify a location within a page?
(i.e. what's the lowest number of bits needed to encode an offset)

Physical Memory

❖ High level view:

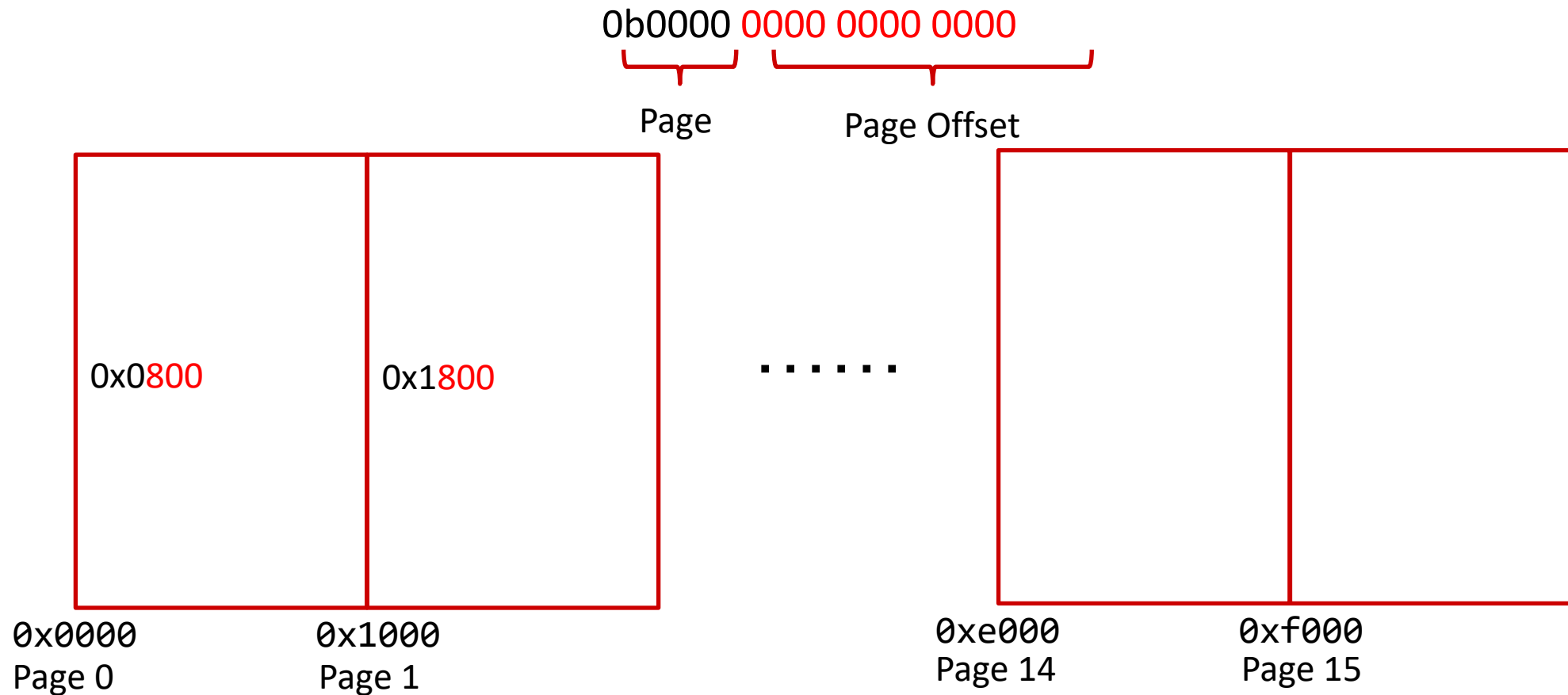
- Each page is aligned by a multiple of 4096 (0X1000)
- If we take an address and add 4096 (0x1000) we get the same offset but within the next page (see below, the red corresponds to the offset, the black with the page.)



Translating from Virtual to Physical

❖ Virtual Address: Simple Limitations

- A portion of the virtual address is used to calculate the physical frame and the value of the offset. Here's an example if addresses == sizeof(short)



Translating from Virtual to Physical

❖ Increasing the number of pages per process...

If we can have more virtual pages per process, we can dedicate more bits to those pages.

0b0000 0000 0000 0000 0000



Page

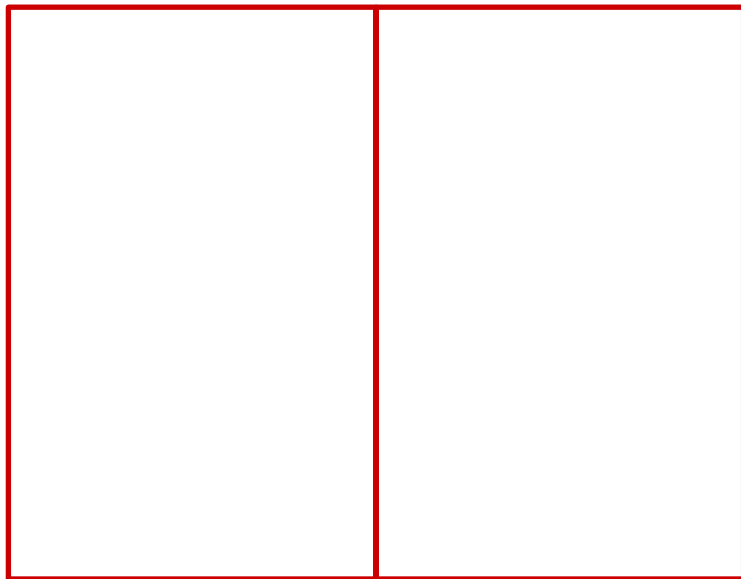


Page Offset

If pages stay the same size, we don't change the number of bits dedicated to the offset.



Think about how the range of pages changes



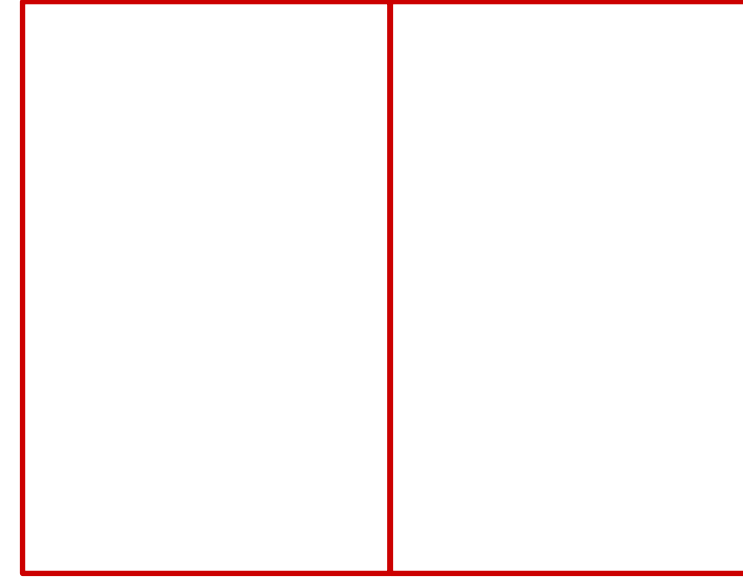
0x00000

Page 0

0x01000

Page 1

...



0xfe000

Page n-1

0xff000

Page n

pollev.com/cis5480

- ❖ In the previous example, we worked with a virtual address that dedicated 8 bits to the page number and 12 bits to the offset within the page.

Let's place some system limitations

- A page is 4096 bytes
- There are 128 virtual pages *max per process*

Is **0x81111** a valid virtual address? Why or why not?

pollev.com/cis5480

- ❖ In the previous example, we worked with a virtual address that dedicated 8 bits to the page number and 12 bits to the offset within the page.

Let's place some system limitations

- A page is 4096 bytes
- There are 128 virtual pages *max per process*

Is **0x81111** a valid virtual address? Why or why not?

- ❖ 0x111 is a valid offset!
- ❖ 0x81 represents value 129 (this would be page 130...)

💡 Remember: Just because you have n bits doesn't mean all 2^n possible values correspond to valid page numbers (or more precisely, to valid page table entries).

Address Translation: Lookup & Combining

- ❖ The extracted “**page number**” is *actually* used as an **index** into a table to look up the corresponding physical page frame (if it exists..)

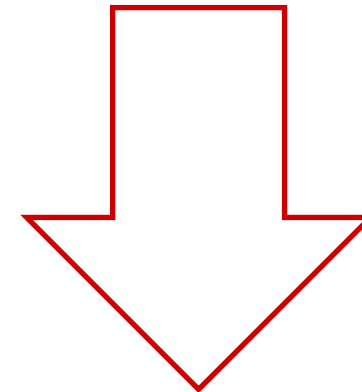
Virtual Address:

0b0111 1111 0000 0000 0001

Page Index

Page Offset

Virtual page #	Valid	Physical Page Number
...	0	null
0x7E	0	0xFF
0x7F	1	0xF0



Physical Address:

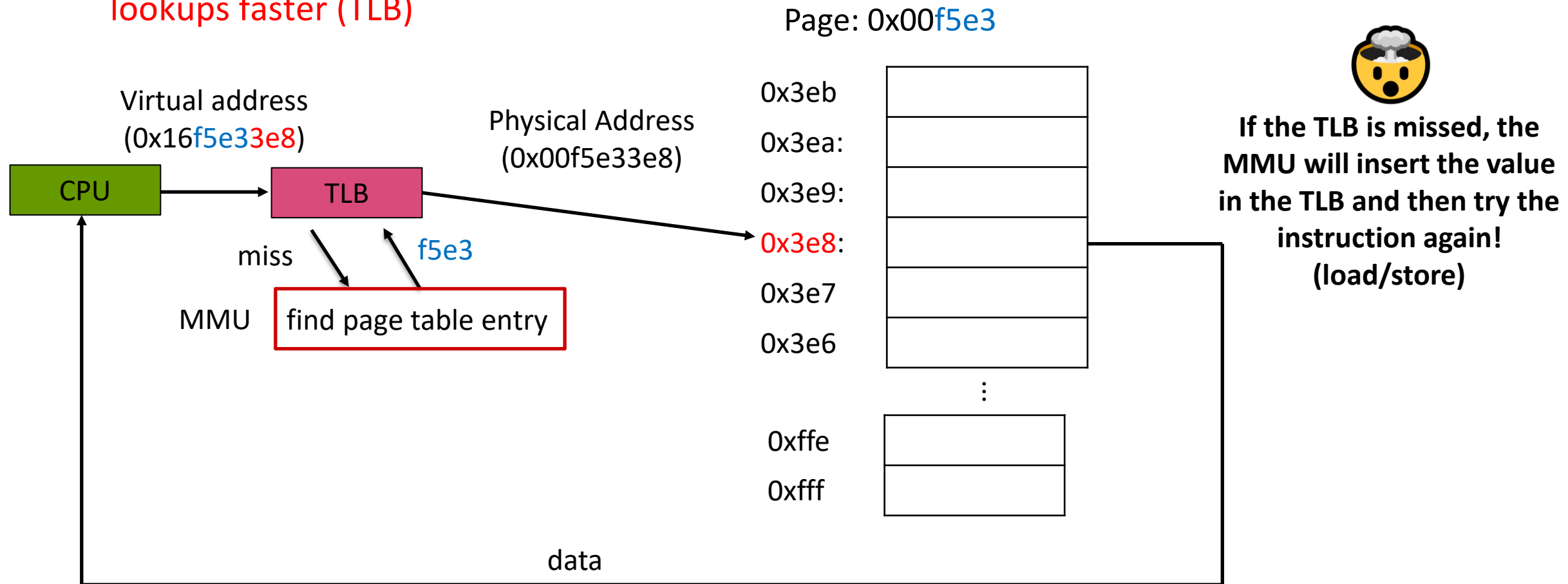
0b1111 0000 0000 0000 0001

Lecture Outline

- ❖ High Level Refresher
- ❖ **TLB**
- ❖ Page Table Details
- ❖ Multi-Level Page Tables
- ❖ Inverted Page Tables

MMU + TLB

- ❖ So, does the MMU access the page table for every memory access?
 - **No: Looking for values in the table is not as simple as it seems. A *dedicated cache* makes lookups faster (TLB)**



Transition Lookaside Buffer (TLB)

- ❖ A special piece of hardware memory that is quick to do lookups in. (cache)
- ❖ **Stores recent virtual page to physical frame translations.**
 - **Hardware for TLB is special, it can quickly check all entries to see if it contains a mapping.**
 - Hardware is expensive, so the TLB is kept relatively small usually (256 entries or so...)
- ❖ **TLB prevents MMU from having to read/walk the page table on each translation to find the mappings.**

This Example with the TLB

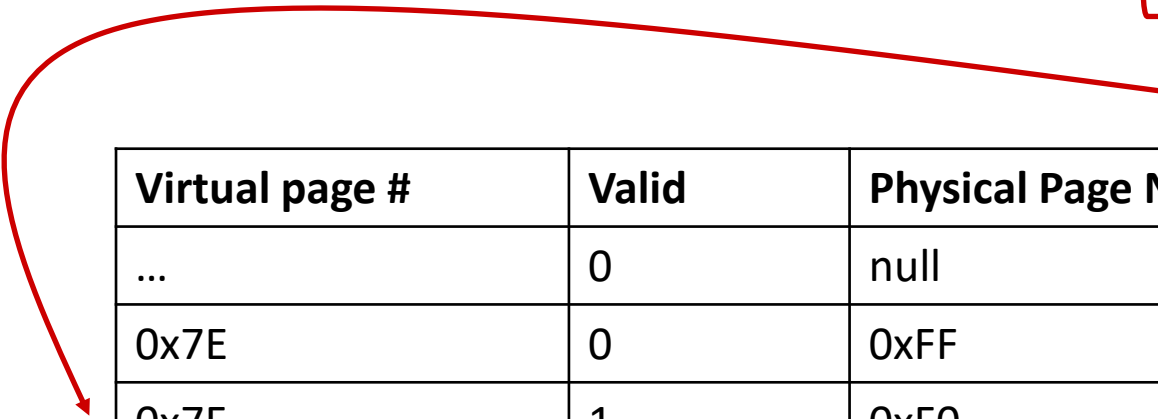
- ❖ If this mapping exists within the TLB, **this is not performed!**

Virtual Address:

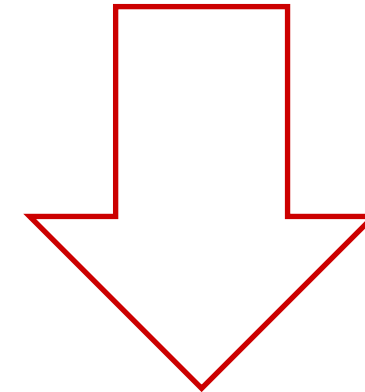
0b0111 1111 0000 0000 0001

Page Index

Page Offset



Virtual page #	Valid	Physical Page Number
...	0	null
0x7E	0	0xFF
0x7F	1	0xF0



Physical Address:

0b1111 0000 0000 0000 0001

This Example with the TLB

- ❖ If this mapping exists within the TLB, **this is not performed!**

Virtual Address:

0b0111 1111 0000 0000 0001

It starts to look more like this.

TLB

Physical Address:

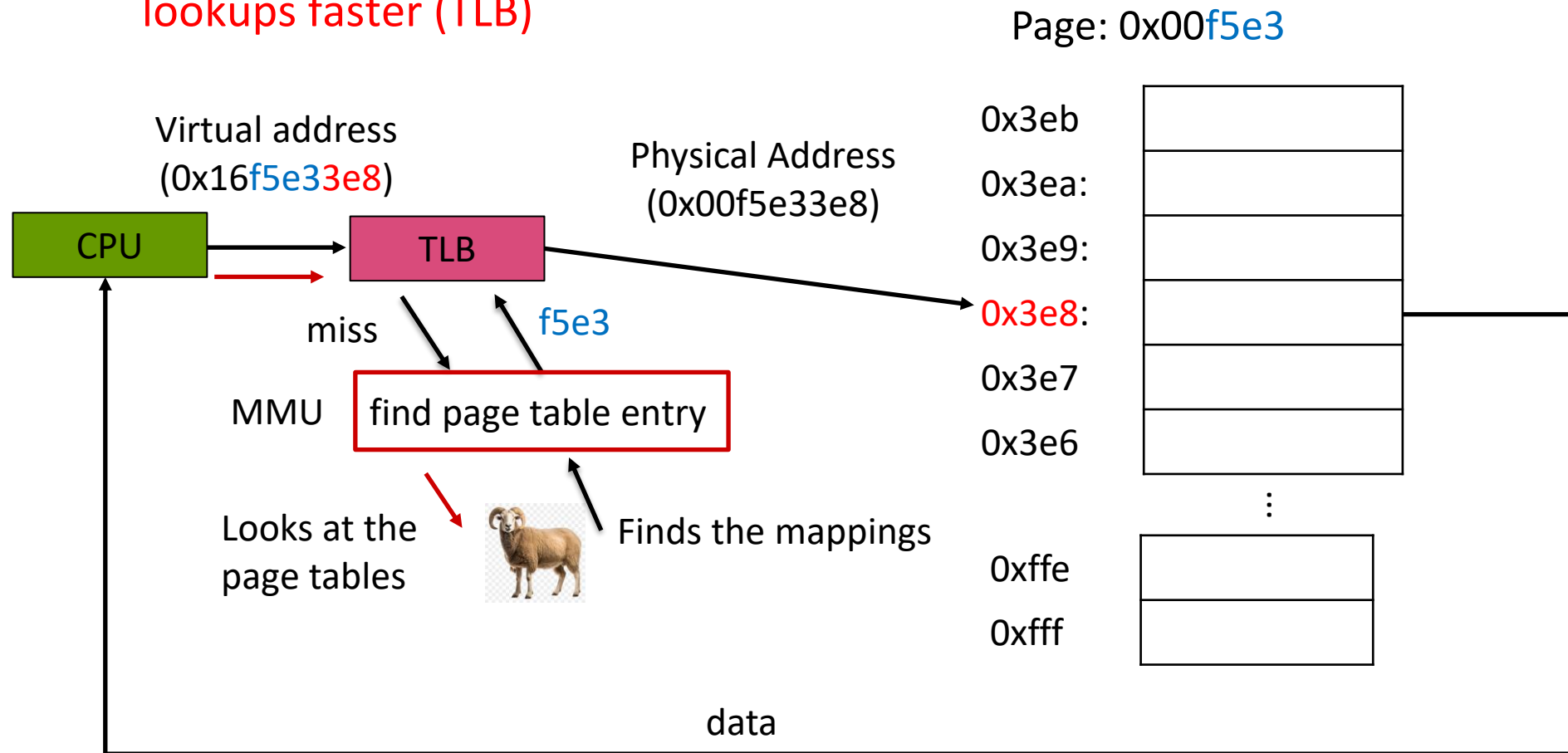
0b1111 0000 0000 0000 0001

TLB Locality

- ❖ Is limited in the number of page table entries it can cache
 - Dramatically smaller than you expect.
- ❖ TLB takes advantage of temporal locality to decide which pages should be stored inside of it
 - Pages that are accessed more often are more likely to be accessed soon in the future
 - The things you need more often you probably keep on your desk and closer to you...when's there's no more space you choose something to evict...or sometimes you just throw everything off your desk (The TLB's equivalent would be called a TLB Flush)

MMU + TLB

- ❖ So, does the MMU access the page table for every memory access?
 - **No: Looking for values in the table is not as simple as it seems. A *dedicated cache* makes lookups faster (TLB)**



If the TLB is missed, the MMU will insert the value in the TLB and then try the instruction again! (load/store)

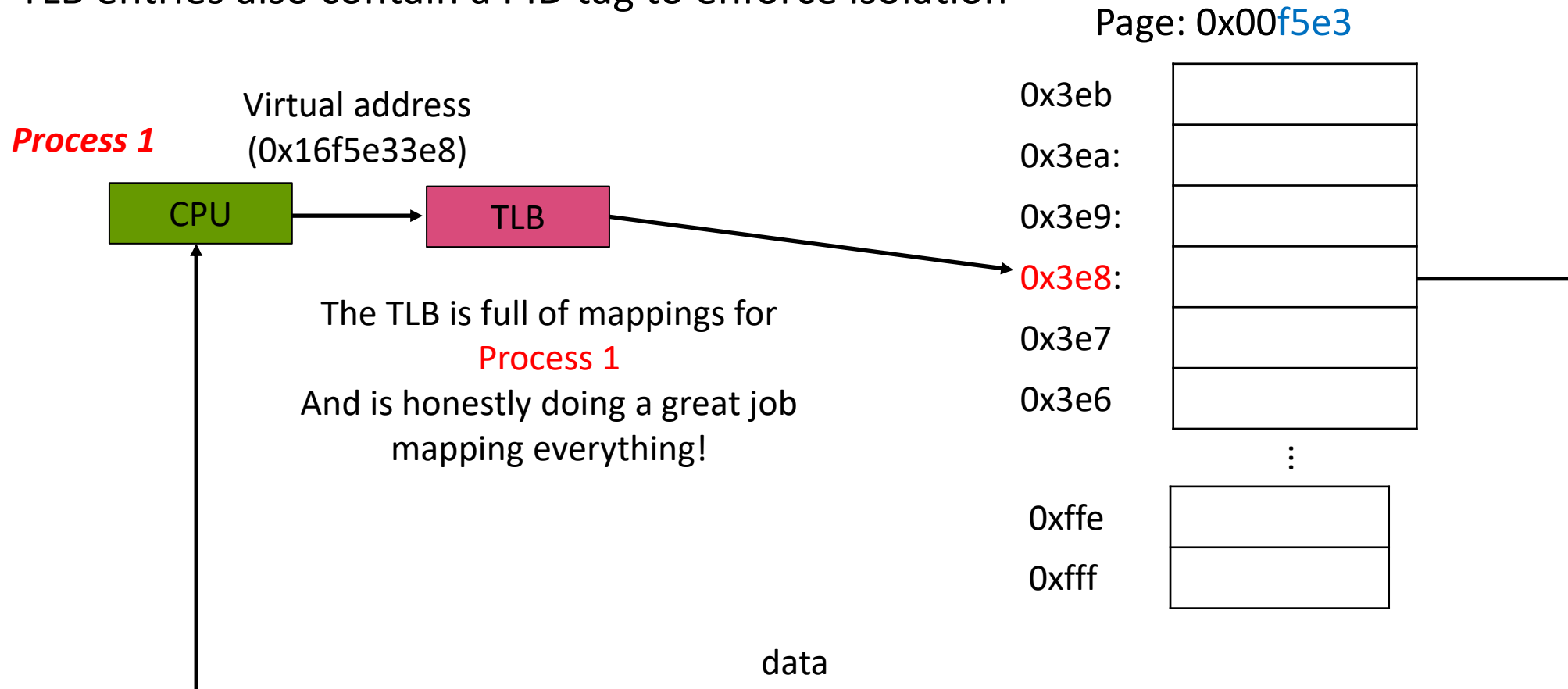
The address must be resolved by the TLB!

TLB: More Details

- ❖ Entries in the TLB need to store:
 - The virtual page -> physical frame mapping
 - *Dirty & Permission* bits
- ❖ TLB Entries need to be kept n'sync with the page table
 - If a TLB entry is updated, the page table must be synced to have the updated dirty bit value
 - If a page is evicted from the page table, but is in the TLB, then that entry must be removed from the TLB (***If not, it will access invalid memory because it will resolve the address***)
- ❖ To maintain process isolation, one of two things
 - When we switch executing processes, the TLB is cleared
 - TLB entries also contain a PID tag to enforce isolation

TLB: More Details

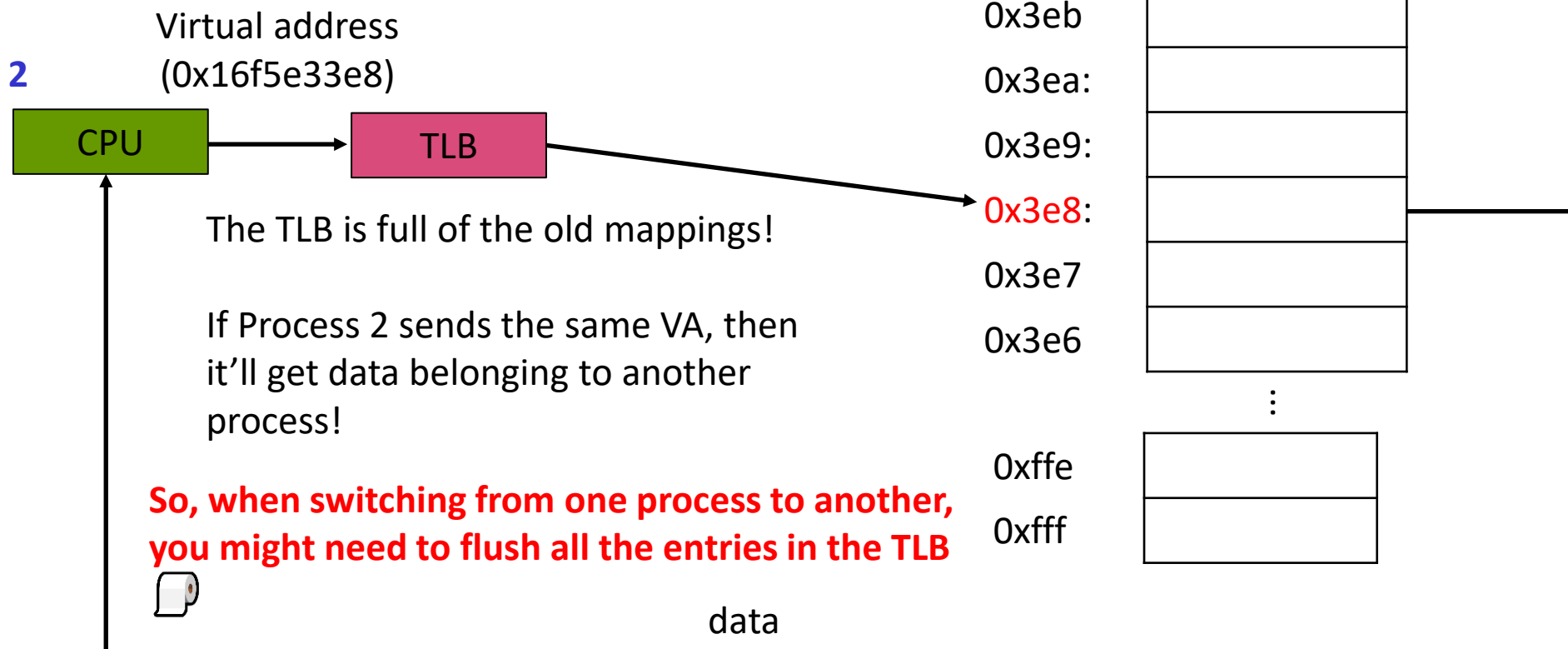
- ❖ To maintain process isolation, one of two things
 - When we switch executing processes, the TLB is cleared
 - TLB entries also contain a PID tag to enforce isolation



TLB: More Details

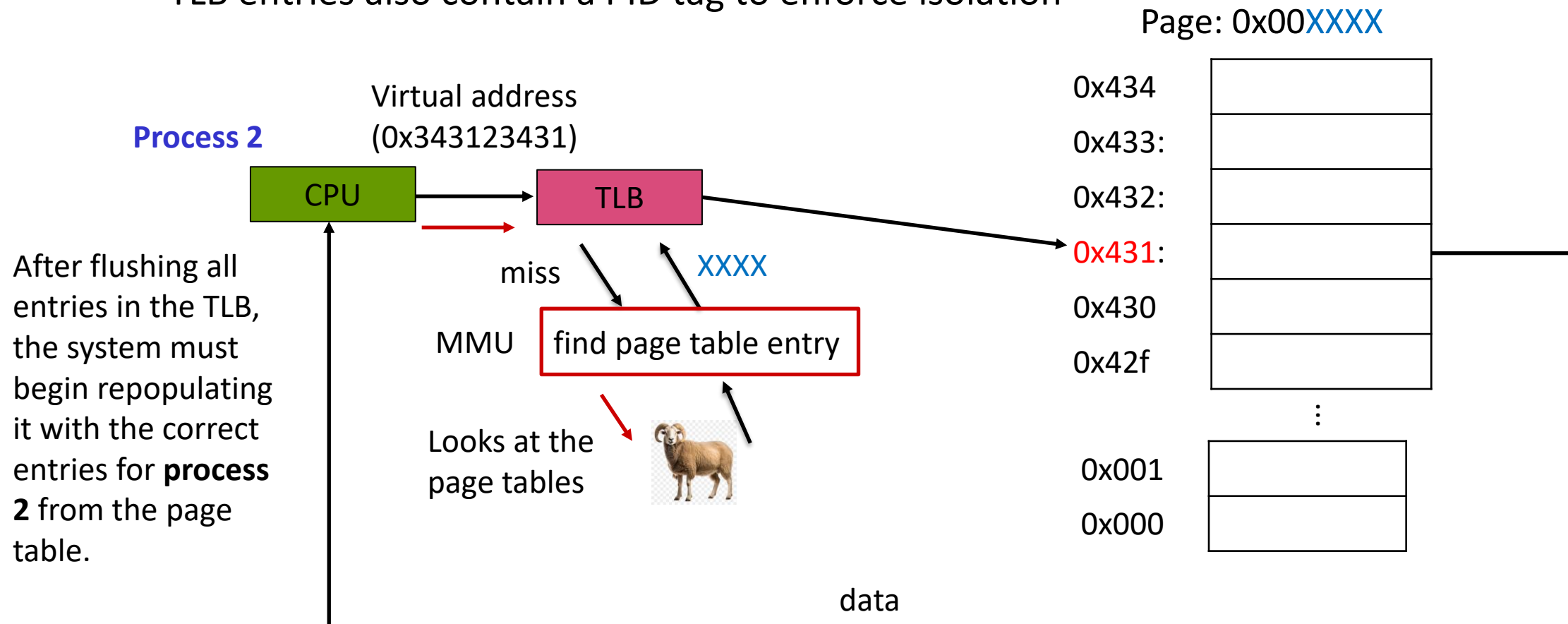
- ❖ To maintain process isolation, one of two things
 - When we switch executing processes, the TLB is cleared
 - TLB entries also contain a PID tag to enforce isolation

Process 2



TLB: More Details

- ❖ To maintain process isolation, one of two things
 - When we switch executing processes, the TLB is cleared
 - TLB entries also contain a PID tag to enforce isolation



TLB: More Details

- ❖ Entries in the TLB need to store:
 - The virtual page -> physical frame mapping
 - *Dirty & Permission* bits
- ❖ TLB Entries need to be kept n'sync with the page table
 - If a TLB entry is updated, the page table must be synced to have the updated dirty bit value
 - If a page is evicted from the page table, but is in the TLB, then that entry must be removed from the TLB (***If not, it will access invalid memory because it will resolve the address***)
- ❖ To maintain process isolation, one of two things
 - When we switch executing processes, the TLB is cleared
 - TLB entries also contain a PID tag to enforce isolation

TLB: More Details

- ❖ Like Caches, CPU's usually have more than 1 TLB.
- ❖ A Level 1 TLB
 - Faster (hardware can check all entries in parallel)
 - Smaller ~64 or 128 entries
 - Usually (nowadays) two Level 1 TLBs
 - One for data
 - One for instructions
- ❖ A Level 2 TLB
 - Faster than looking up in a Page Table but slower than a level 1 TLB lookup
 - ~512 entries
 - Usually contains addresses for both instructions & data.

Lecture Outline

- ❖ High Level & Address Translation Refresher
- ❖ TLB
- ❖ **Page Table Details**
- ❖ Multi-Level Page Tables
- ❖ Inverted Page Tables

Previous View of a page table

- ❖ One page table per process
- ❖ Is just a big array of page table entries
- ❖ One entry per page
 - on a modern 64-bit machine, that is 2^{52} (4,503,599,627,370,496) entries

Virtual page #	Valid	Physical Page Frame
0	0	---- //page hasn't been used yet
1	1	0
2	1	1
3	0	1

Page Table Entry: Valid Bit & Reference Bit

❖ Valid:

- 1 bit, True/False whether the page is in physical memory
- If bit is 0, then it is not present in memory and a page fault occurs

❖ Reference:

- Used by the MMU to determine eviction; helps measure the *age* of a page
- More on this next lecture!

Virtual page #	Valid	Frame #	Reference	dirty	permissions
0	0	----			
1	1	0	11	1	R/W
2	1	1	01	0	R/X
3	0	1			

Page Table Entry: Dirty & Permission Bits

❖ Dirty:

- 1 bit whether the page has been written to
- If page is dirty and needs to be evicted from physical memory, then the data **must** be written back to the swap file

❖ Permissions:

- At least three bits to determine permissions to that memory
- Can it be **R**ead, **W**ritten or e**X**ecuted?

Virtual page #	Valid	Frame #	dirty	permissions
0	0	----		
1	1	0	1	R/W
2	1	1	0	R/X
3	0	1		

Page Table Entry

- ❖ A page table entry stores more than a valid bit and the physical page number (and more than what I have here)
 - Valid: True/False whether the page is in physical memory
 - Frame #: the location of the page in physical memory iff it is in it
 - Dirty: whether the page was written to or not
 - Permissions: whether the page can be used for **R**eading, **W**riting or e**X**ecuting.
 - And much more...

Virtual page #	Valid	Frame #	dirty	permissions
0	0	----		
1	1	0	1	R/W
2	1	1	0	R/X
3	0	1		

A Big Array

- ❖ We can view the page table as being an array that we can index into using the Virtual page number
- ❖ With 2^{52} virtual pages per process, that is 2^{52} entries per page table... It would help to keep page table entries small

Ask Yourself

- ❖ Question: What could we remove to make the entries smaller?

Virtual page #	Valid	Frame #	Reference	dirty	permissions
0	0	----			
1	1	0	11	1	R/W
2	1	1	01	0	R/X
3	0	1			

Optimization: Remove Virtual Page

- ❖ The Virtual page # can be removed since it is implicitly the index into our Page Table

Virtual page #	Valid	Frame #	Reference	dirty	permissions
0	0	----			
1	1	0	11	1	R/W
2	1	1	01	0	R/X
3	0	1			

Page Tables in Reality

- ❖ Page Table Entries are simply numerical values, where specific bits encode information such as the physical address, access permissions, etc...

- ❖ Frame # – 8 Bits

- ❖ Valid – 1 bit

- ❖ Reference – 2 bits

- ❖ Dirty – 1 bit

- ❖ Permissions – 4 bits

'0b00000000000000000000'

Index N

⋮

'0b00000000000000000000'

Index 2

'0b00000000000000000000'

Index 1

'0b00000000000000000000'

Index 0

Table Address _____
(Where the table starts)

Arm v7 Page Table Entry

❖ Page Table Entry

- Arm calls them “Descriptors” (not sure why)



❖ Armv7 is a 32 bit architecture...

❖ “Small Page Base Address” is a 4KB page

- Bits 31–12, indicated the Physical Frame Number

❖ nG (Not Global)

- 2 bits
- Is this memory shared by everyone? Process Specific?

❖ AP (Access Permissions)

- 2 bits
- Read/Write?

Executable Page?

Can we execute instructions stored at this frame?

XN = Execute Never; 1 Bit

Still really big :(

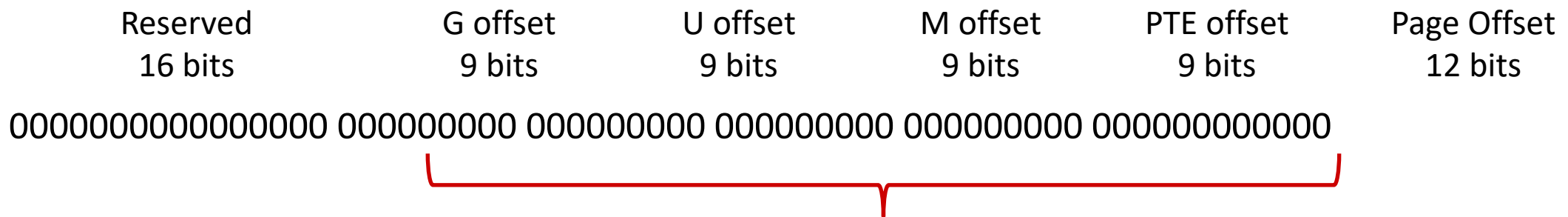
- ❖ Removing the page number saves us 52 bits from the input, but we still end up with ~30 bits (4 bytes) per entry
- ❖ One page table takes up $2^{52} * 4 = 2^{54}$ bytes ☹️
- ❖ How can we make this better?

Lecture Outline

- ❖ High Level & Address Translation Refresher
- ❖ TLB
- ❖ Page Table Details
- ❖ **Multi-Level Page Tables**
- ❖ Inverted Page Tables

Multi Level Page Table: x86 Linux Implementation

- ❖ On a 64-bit address, we keep the bottom 12 bits for the page offset, and the upper 52 for the page number.
- ❖ We can split the page number into 4 groups of 9 bits



Each of these groups of bits are an index into a table.

❖ High level view



Looking up an address

- ❖ First index into top level table using the top 9-bit chunk

Reserved
16 bits

G offset
9 bits

U offset
9 bits

M offset
9 bits

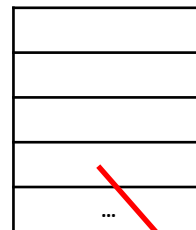
PTE offset
9 bits

Page Offset
12 bits

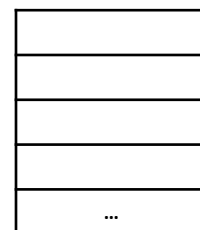
0000000000000000 101001010 100000010 111011010 001101011 000101011011

We already know the location of the top-level page table—so we use the **G Offset** to index directly into it.

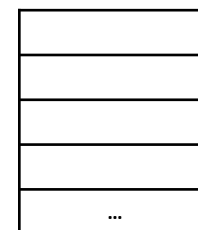
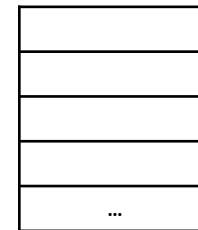
Top level table



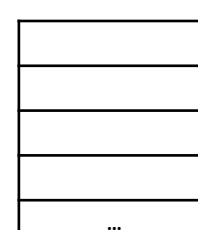
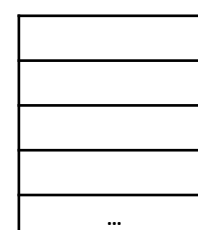
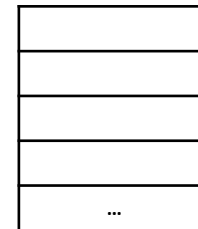
Third tables



Mid level tables



PTE's



Looking up an address

- ❖ First index into top level table using the top 9-bit chunk

Reserved
16 bits

G offset
9 bits

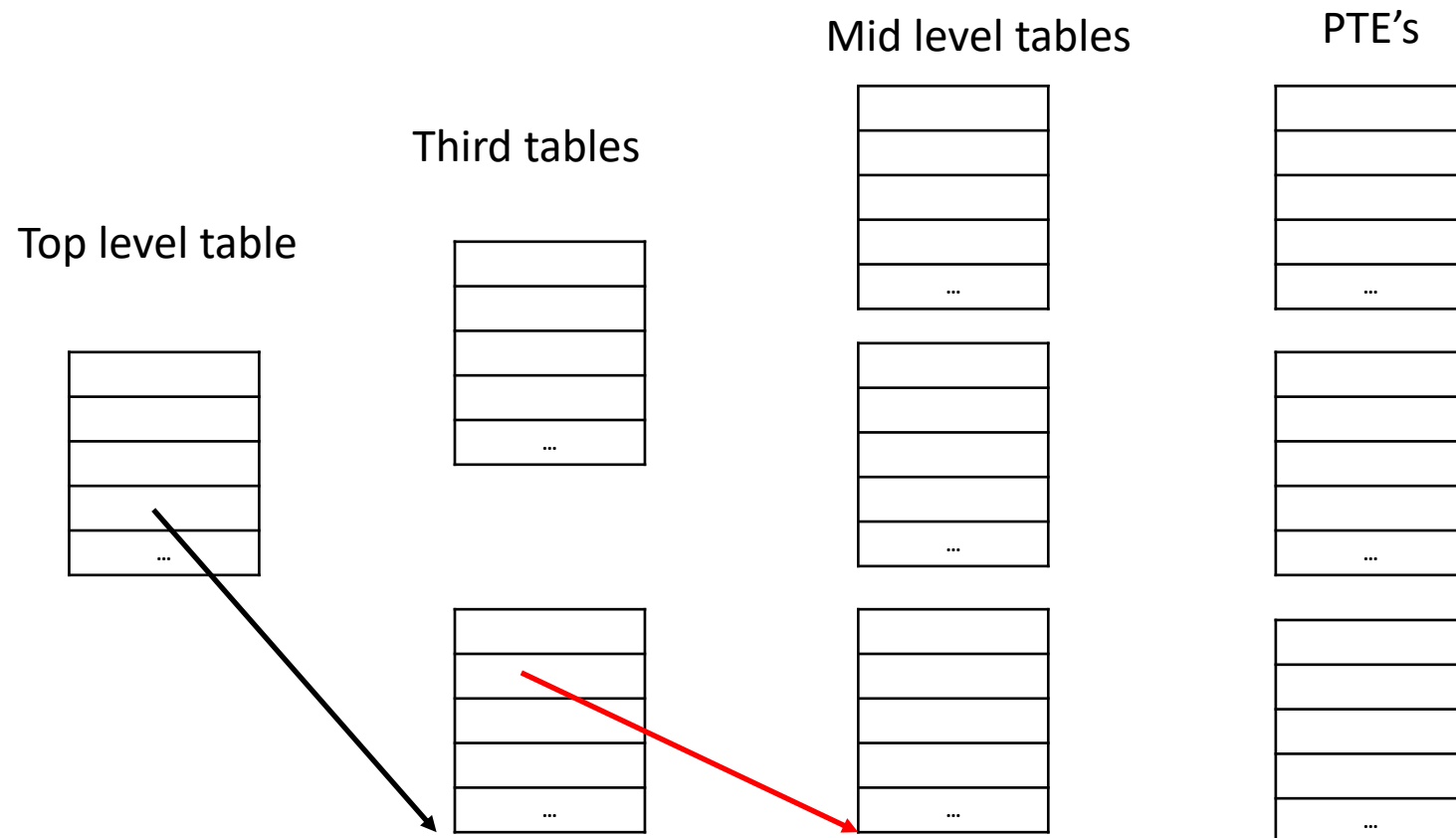
U offset
9 bits

M offset
9 bits

PTE offset
9 bits

Page Offset
12 bits

0000000000000000 101001010 100000010 111011010 001101011 000101011011



Looking up an address

- ❖ First index into top level table using the top 9-bit chunk

Reserved
16 bits

G offset
9 bits

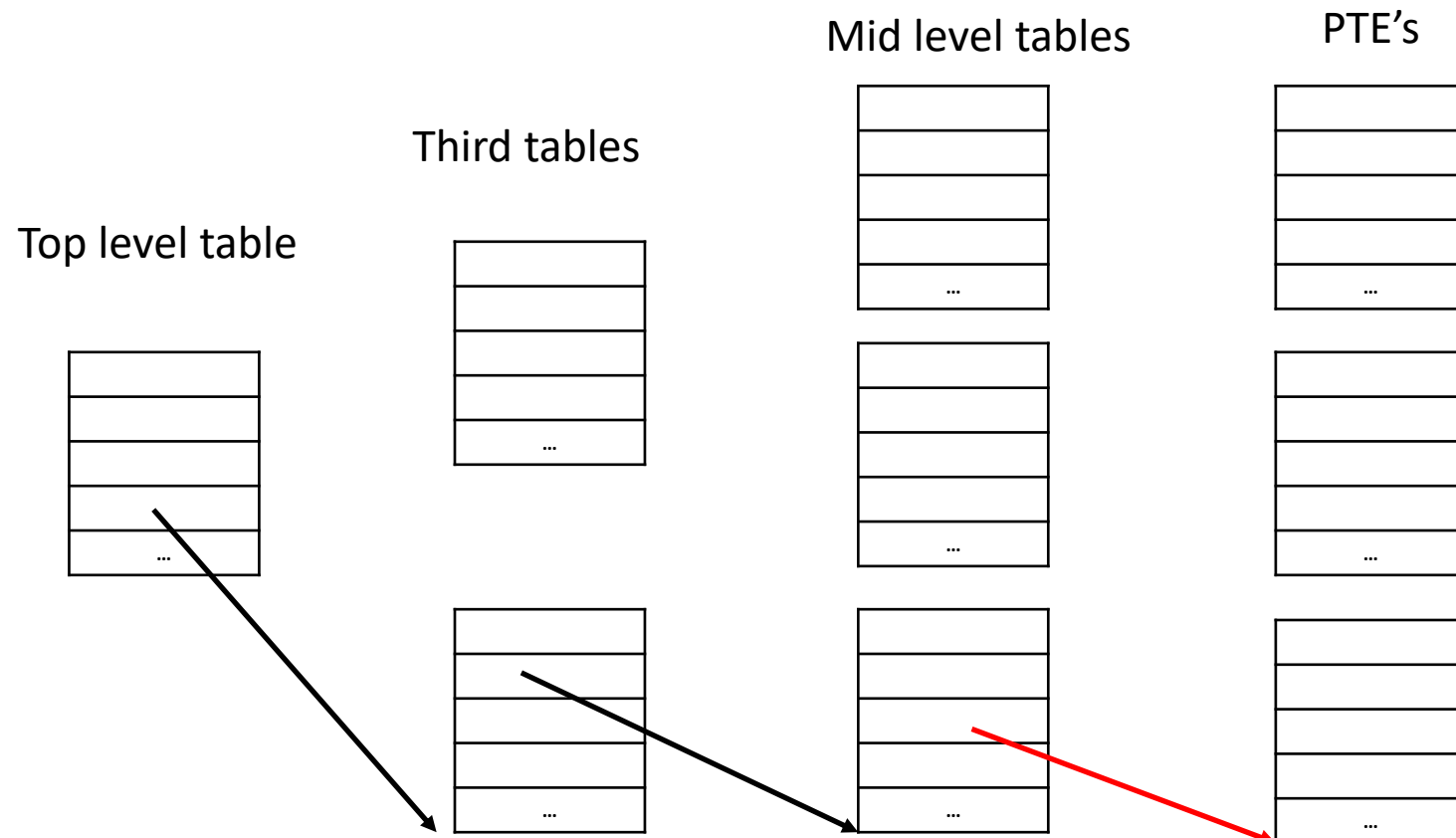
U offset
9 bits

M offset
9 bits

PTE offset
9 bits

Page Offset
12 bits

0000000000000000 101001010 100000010 **111011010** 001101011 000101011011



Looking up an address

- ❖ First index into top level table using the top 9-bit chunk

Reserved
16 bits

G offset
9 bits

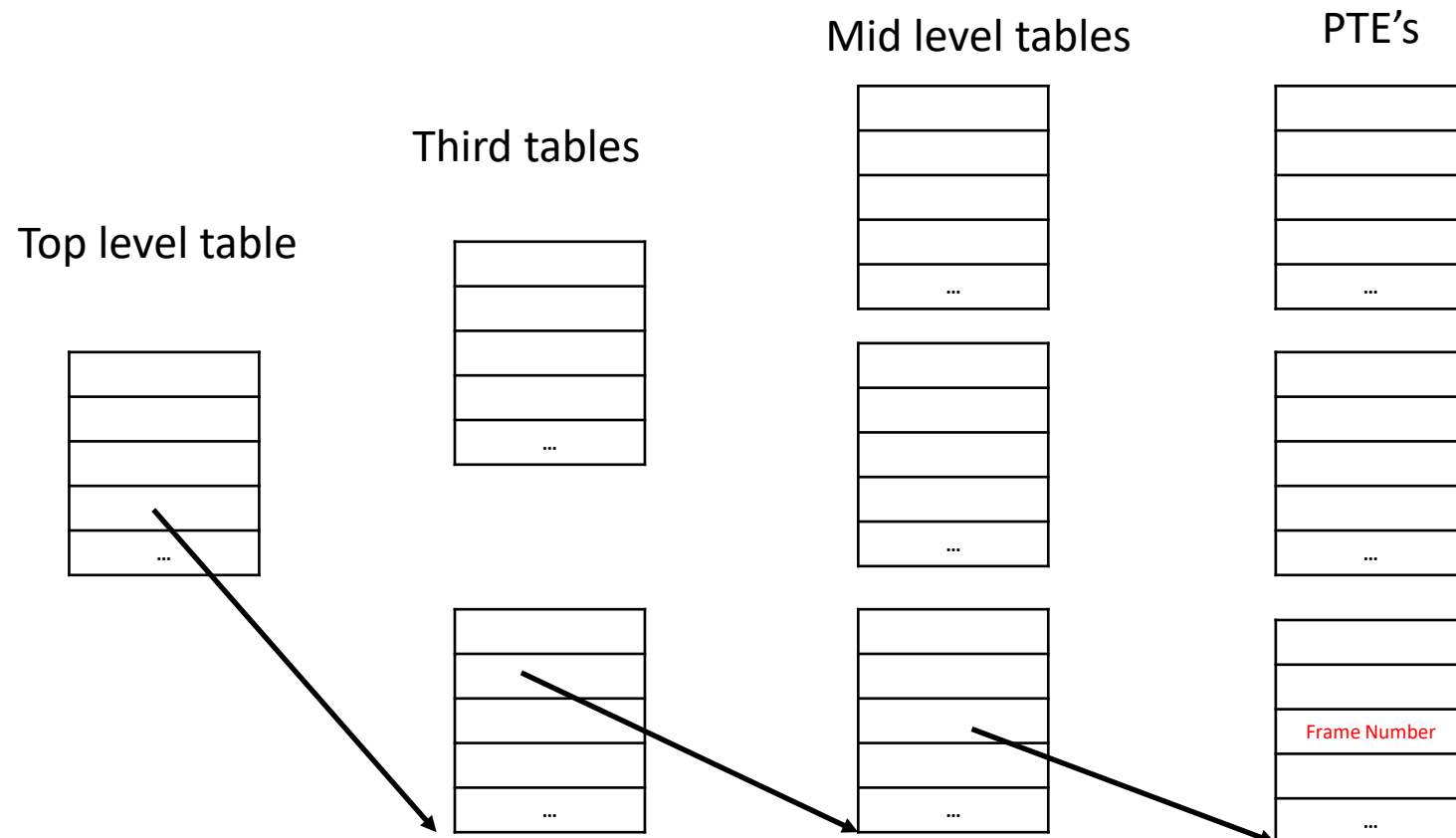
U offset
9 bits

M offset
9 bits

PTE offset
9 bits

Page Offset
12 bits

0000000000000000 101001010 100000010 111011010 001101011 000101011011



Why 9 bits?

- ❖ Why is each index into a level of the page table 9 bits?
 - $9 \text{ bits} = 2^9 = 512$ entries in each Intermediary Table
- ❖ Each entry is just a pointer to the next level table
 - A pointer on a 64-bit machine is 8 bytes
 - A page table entry is also at max 8 bytes
- ❖ 2^9 entries * 2^3 bytes per entry = 2^{12} bytes (size of a page!)
 - This means each level into the page table itself is the size of the page. Makes maintaining the page table itself convenient since the page table itself lies in memory.

Analysis

- ❖ Most of the pages that are theoretically available to a process go unused. Multi Level Page Tables take advantage of this, most pointers in the table are **NULL**
 - A lot less space needed than our first idea of a page table
- ❖ Lazily allocate page table entries for pages as they are needed
 - E.g. only allocate them once they are needed
- ❖ Take advantage of **temporal locality**: if a particular memory location is referenced, it is likely that it and nearby memory locations will be accessed soon
 - I'll revisit the idea of locality later

Analysis pt. 2

- ❖ Take advantage of **temporal locality**: if a particular memory location is referenced, it is likely that it and nearby memory locations will be accessed soon
 - If pages near each other in memory are accessed, they will in the same nodes in the tree!
Not every page access requires the creation of a mid-level node
 - I'll revisit the idea of locality later
- ❖ What was once just one memory access to lookup page frame is now four memory accesses 😞
 - This can be very expensive time-wise
 - There is hardware (TLB) that helps a lot with this 😊

Poll Yourself At Home



Reserved
16 bits

G offset
9 bits

U offset
9 bits

M offset
9 bits

PTE offset
9 bits

Page Offset
12 bits

0000000000000000 000000000 000000000 000000000 000000000 0000000000000

We know there is a single global G Table containing 2^9 entries.

Each entry in the G table points to the base address of a U Intermediary Table, and this structure continues down through additional levels.

If **every entry at every level** is valid, what is the **maximum amount of memory** that the page table will occupy?

$$\text{The total number of entries is } 8 * (2^9 + (2^9 * 2^9) + (2^9 * 2^9 * 2^9) + (2^9 * 2^9 * 2^9 * 2^9))$$

Bytes per entry
U Entries
PTE Entries

G Entries
M Entries

Poll Yourself At Home



Reserved
16 bits

G offset
9 bits

U offset
9 bits

M offset
9 bits

PTE offset
9 bits

Page Offset
12 bits

0000000000000000 000000000 000000000 000000000 000000000 000000000000

We know there is a single global G Table containing 2^9 entries.

Each entry in the G table points to the base address of a U Intermediary Table, and this structure continues down through additional levels.

If **every entry at every level** is valid, what is the **maximum amount of memory** that the page table will occupy?

The total number of entries is

~ 550831656960 Bytes = .5 Terabytes

Lecture Outline

- ❖ High Level & Address Translation Refresher
- ❖ TLB
- ❖ Page Table Details
- ❖ Multi-Level Page Tables
- ❖ More Next time! 😊