Page Tables

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Poll Everywhere

pollev.com/tqm

- How is proj1 milestone going?

Administrivia

- Peer Evaluation: out now, due **TONIGHT**
  Tuesday 9/26 @ 11:59 pm
  - Please do it, it shouldn't take long
  - Mostly completion, don't just say "this is fine" for everything

- Project 1 is out now
  - The milestone is due **TOMORROW** Wed 9/27 @ 11:59 pm
    late deadline: 11:59 pm on Sun, Oct 01

- Recitation 3 after lecture: some GDB and then open office hours
- Travis has office hours from 4:30 to 6:30 pm today

Poll Everywhere

pollev.com/tqm

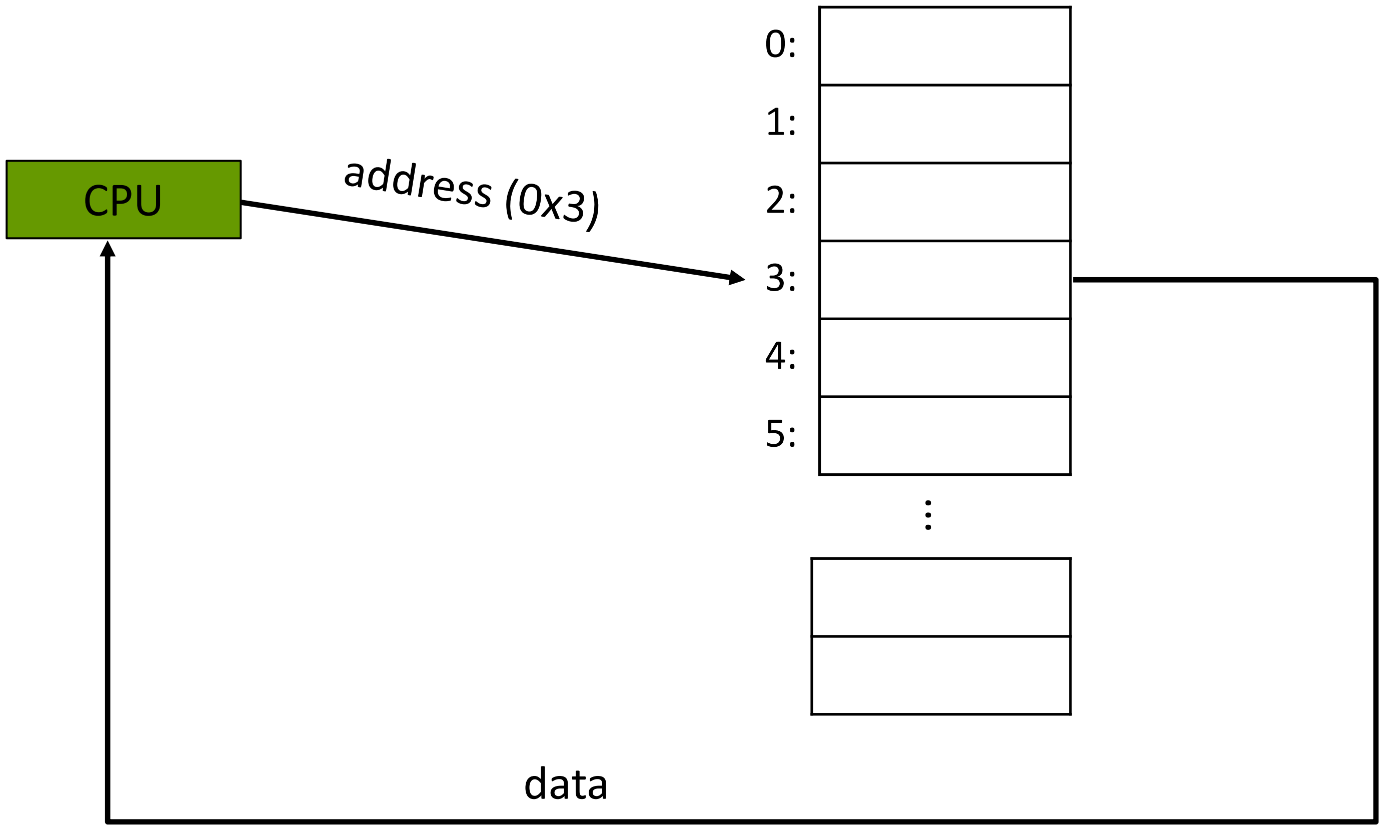
- Any questions, comments or concerns from last lecture?

Lecture Outline

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

This doesn’t work anymore

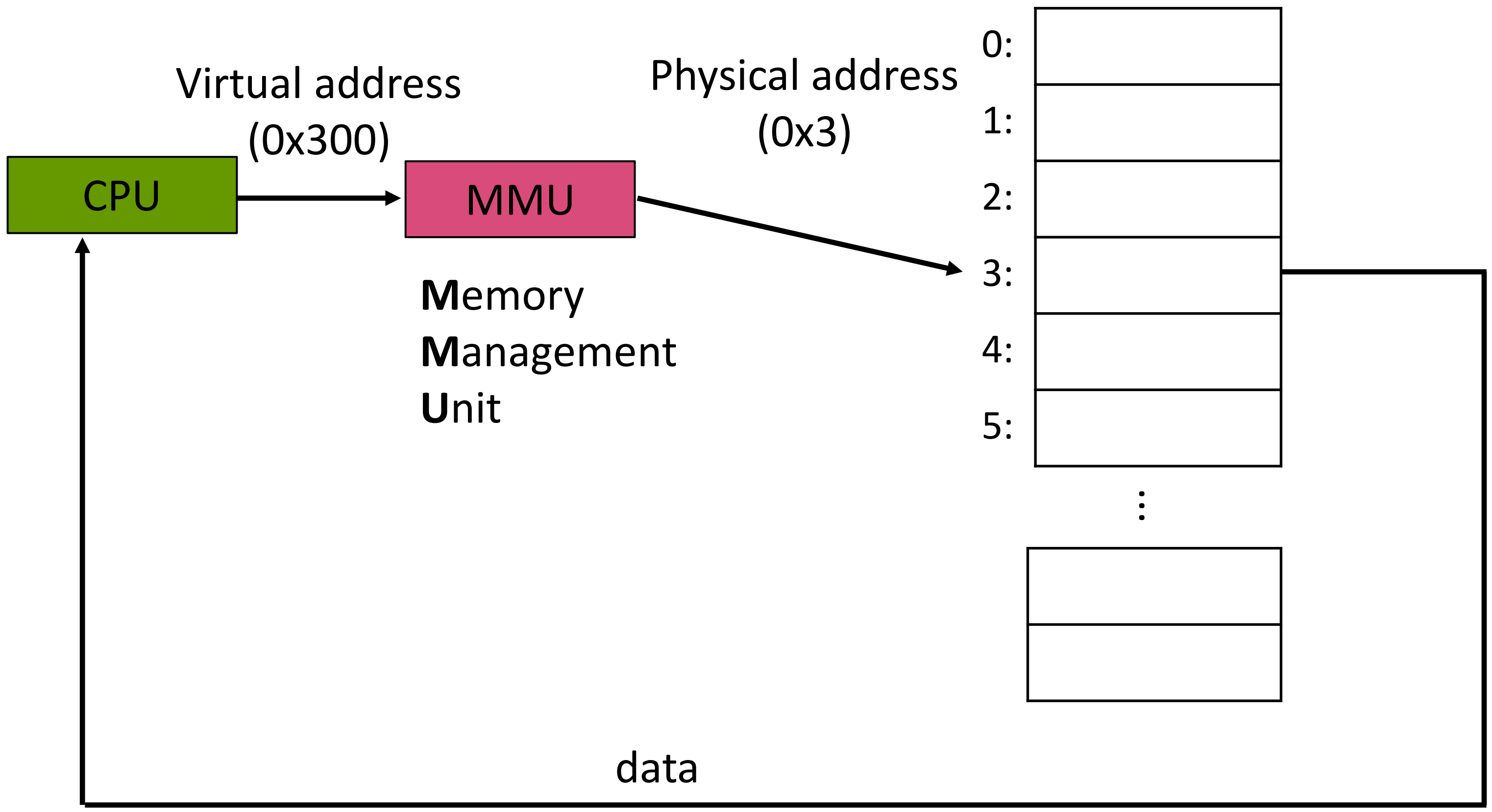
- The CPU directly uses an address to access a location in memory



Virtual Address Translation

THIS SLIDE IS KEY TO THE WHOLE IDEA

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

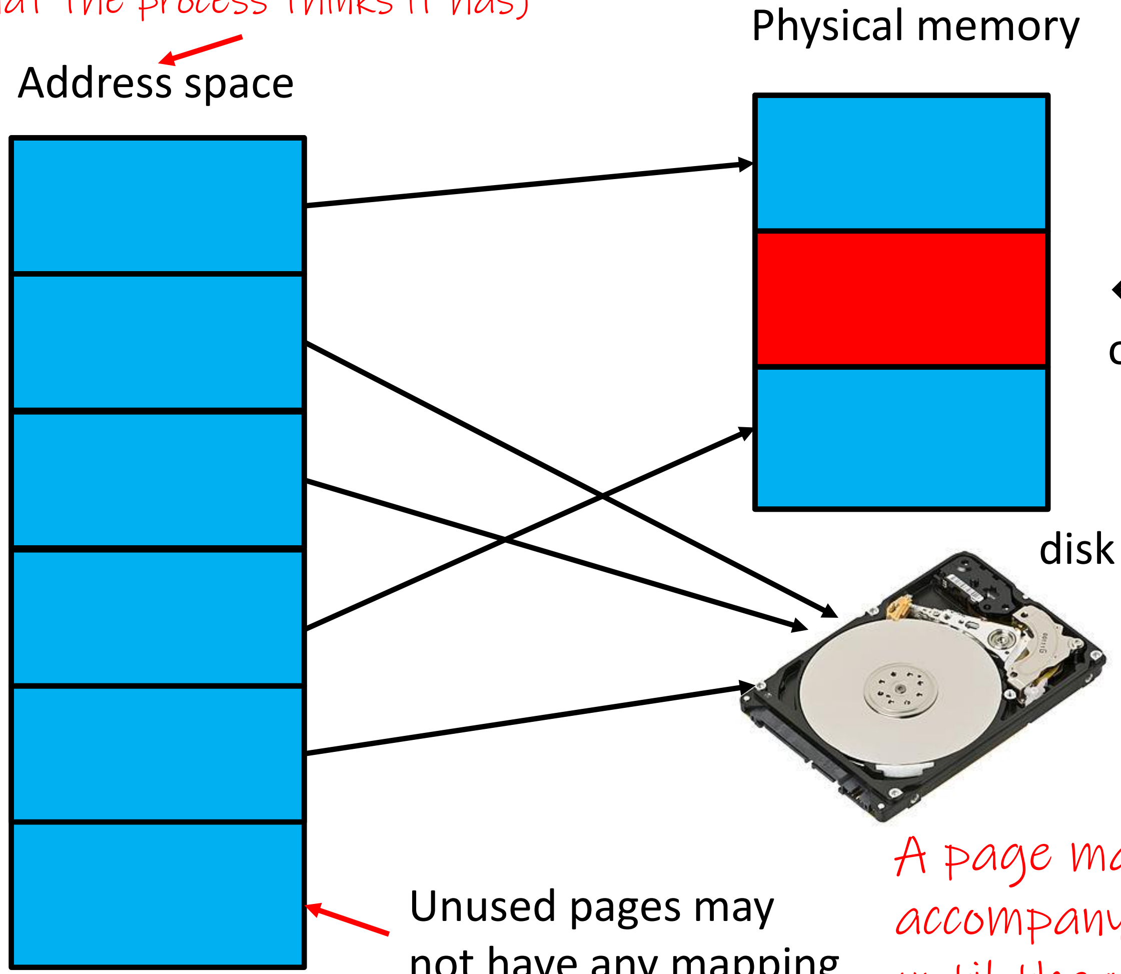


Pages

Pages are of fixed size ~4KB
4KB -> (4 * 1024 = 4096 bytes.)

- Memory can be split up into units called "pages"

(what the process thinks it has)



Pages currently in use are stored in physical memory (RAM)

← Ram may contain pages from other active processes

Pages in physical memory are called "Page frames"

Pages not currently in use (but were used in the past) are stored on disk

Unused pages may not have any mapping

A page may not have an accompanying page frame until the page is used

Page Tables

More details about translation later

- Virtual addresses can be converted into physical addresses via a page table.

- There is one page table per processes, managed by the MMU

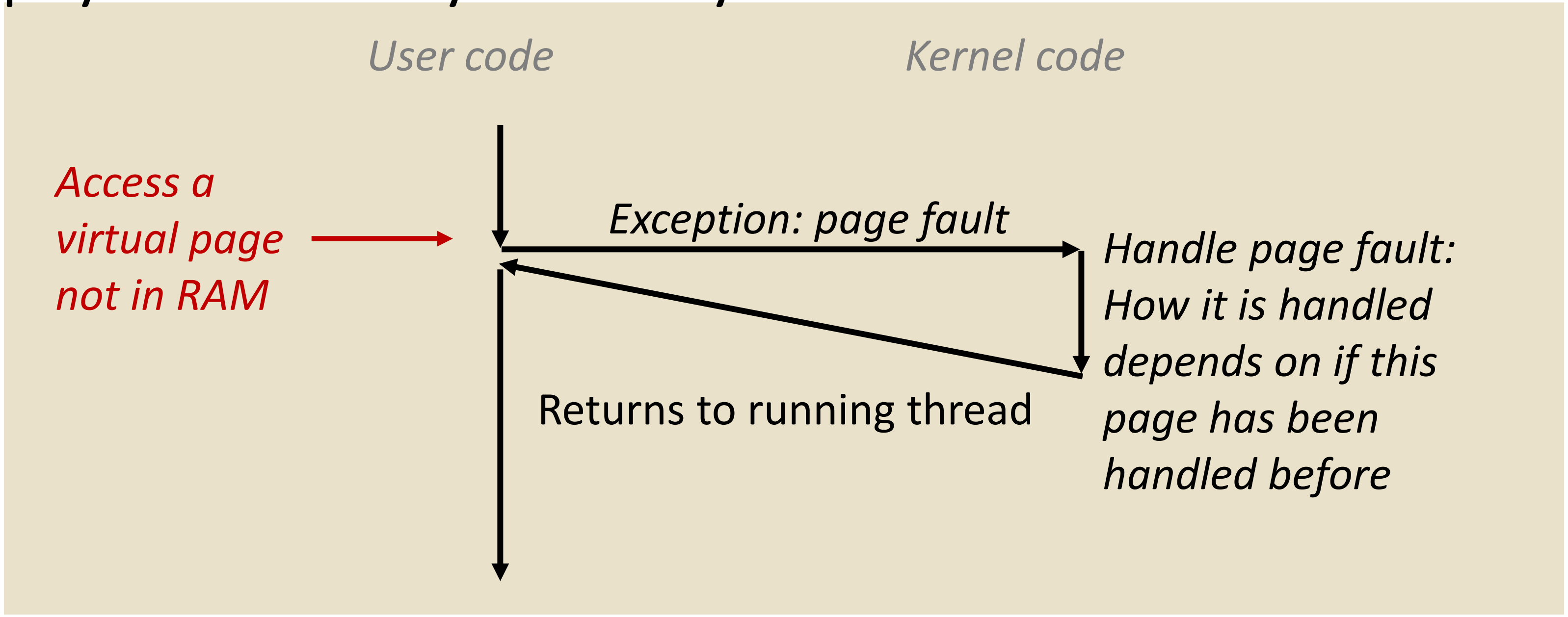
Virtual page #	Valid	Physical Page Frame
0	0	---- //page hasn't been used yet
1	1	0
2	1	1
3	0	1

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


User code
Kernel code
Access a virtual page not in RAM
Exception: page fault
Handle page fault: How it is handled depends on if this page has been handled before
Returns to running thread

Poll Everywhere

pollev.com/tqm

- What happens if this process tries to access an address in page 3?

Virtual page #	Valid	Physical Page Frame
0	0	3
1	1	0
2	1	1
3	0	1
…	…	…

Poll Everywhere

pollev.com/tqm

- What happens if this process tries to access an address in page 3?

We get a page fault,
the OS evicts a page
from a frame, loads in
new page into that frame

Virtual page #	Valid	Physical Page Frame
0	0	3
1	1	0
2	1	1
3	**0**	1
…	…	…

Poll Everywhere

pollev.com/tqm

- What happens if this process tries to access an address in page 1?

Virtual page #	Valid	Physical Page Frame
0	0	3
1	1	0
2	1	1
3	0	1
…	…	…

Poll Everywhere

pollev.com/tqm

- What happens if this process tries to access an address in page 1?

The MMU access the corresponding frame (frame 0)

Virtual page #	**Valid**	**Physical Page Frame**
0	0	3
1	**1**	**0**
2	1	1
3	0	1
…	…	…

Poll Everywhere

pollev.com/tqm

- A page is typically 4 KiB -> 2^{12} -> 4096 bytes

- If physical memory is 32 GiB (1 GiB = 2^{30} bytes), how many page frames are there?

  A. 2^{30} B. 2^{18} C. 2^{35} D. 2^{23} E. We're lost…

- If addressable memory for a single process consists of 2^{64} bytes, how many pages are there for one process?

  A. 2^{52} B. 2^{35} C. 2^{50} D. 2^{23} E. We're lost…

Poll Everywhere

pollev.com/tqm

- A page is typically 4 KiB -> 2^{12} -> 4096 bytes

- If physical memory is 32 GiB (1 GiB = 2^{30} bytes), how many page frames are there?

  A. 2^{30} B. 2^{18} C. 2^{35} D. 2^{23} E. We're lost…

32 GiB = $2^5 * 2^{30}$ B = 2^{35} bytes 2^{35} bytes / 2^{12} bytes = 2^{23} frames

- If addressable memory for a single process consists of 2^{64} bytes, how many pages are there for one process?

  A. 2^{52} B. 2^{35} C. 2^{50} D. 2^{23} E. We're lost…

2^{64} bytes / 2^{12} bytes = 2^{52} pages

Addresses

- Virtual Address:
  - Used to refer to a location in a virtual address space.
  - Generated by the CPU and used by our programs
- Physical Address
  - Refers to a location on physical memory
  - Virtual addresses are converted to physical addresses

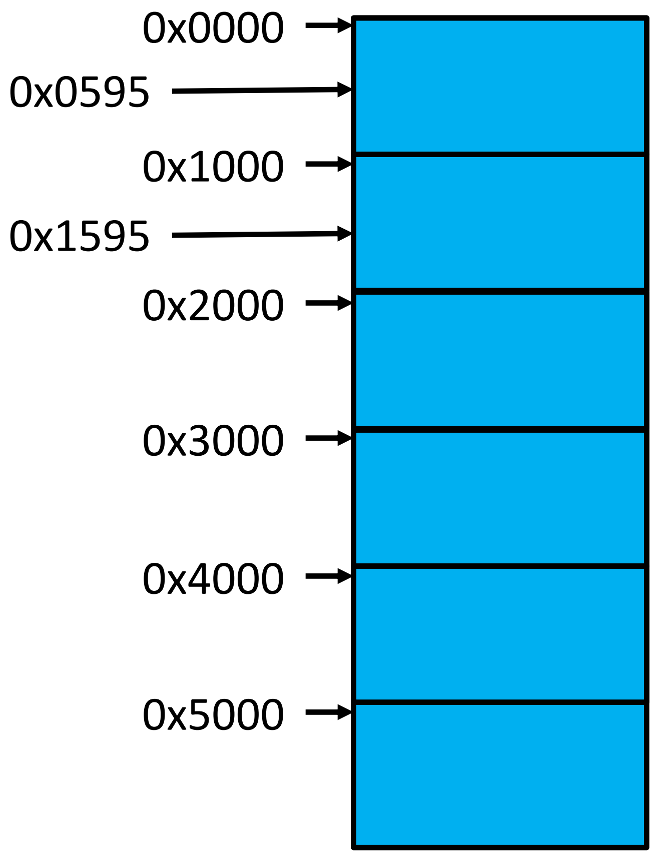
Page Offset

- This idea of Virtual Memory abstracts things on the level of Pages (4096 bytes == 2^{12} bytes)
- On almost every machine, memory is ***byte-addressable*** meaning that each byte in memory has its own address
- How many different addresses correspond to the same page? 4096 addresses to a single page
- How many bits are needed in an address to specify where in the page the address is referring to?

  12 bits

Virtual Address High Level View

- High level view:
  - Each page starts at a multiple of 4096 (0X1000)
  - If we take an address and add 4096 (0x1000) we get the same offset but into the next page



Steps For Translation

- Derive the virtual page number from a virtual address
- Look up the virtual page number in the page table
  - Handle the case where the virtual page doesn’t correspond to a physical page frame
- Construct the physical address

Address Translation: Virtual Page Number

- A virtual address is composed of two parts relevant for translating:

Virtual Page Number	**Page Offset**

  - Virtual Page Number length = bits to represent number of pages
  - Page offset length = bits to represent number of bytes in a page

- The virtual page number determines which page we want to access

- The page offset determines which location within a page we want to access.
  - Remember that a page is many bytes (~4KiB -> 4096 bytes)

Address Translation: Virtual Page Number

- A virtual address is composed of two parts relevant for translating:

Virtual Page Number	Page Offset

  - Virtual Page Number length = bits to represent number of pages
  - Page offset length = bits to represent number of bytes in a page

pollev.com/tqm

- Example address: 0x34805
  - What is the page number?
  - What is the offset?
  - For this problem: **there are 64 virtual pages, and a page is 4096 bytes**

Address Translation: Virtual Page Number

- A virtual address is composed of two parts relevant for translating:

Virtual Page Number	Page Offset

  - Virtual Page Number length = bits to represent number of pages
  - Page offset length = bits to represent number of bytes in a page

pollev.com/tqm

- Example address: 0x34805 `0011 0010 1000 0000 0101`
  - What is the page number? `0011 0010 -> 0x34`
  - What is the offset? `1000 0000 0101 -> 0x805`
  - For this problem: **there are 64 virtual pages, and a page is 4096 bytes**

Quick one: just discuss

- In the previous example, we worked with the virtual address 0x34805. **Why would the address 0x54805 not be a legal virtual address in that same system?**
  - A page is typically 4 KiB -> 2^{12} -> 4096 bytes
  - There were 64 virtual pages

Quick one: just discuss

- In the previous example, we worked with the virtual address 0x34805. **Why would the address 0x54805 not be a legal virtual address in that same system?**
  - A page is typically 4 KiB -> 2^{12} -> 4096 bytes
  - There were 64 virtual pages

0x54 translates to 84, which is outside the range of valid page numbers (0-63)

Alternative approach:
64 pages = 6 bits for the page number...
we need 7 bits to represent **0x54** -> **0b 0$\underline{\text{101 0100}}$**

Address Translation: Lookup & Combining

- Once we have the page number, we can look up in our page table to find the corresponding physical page number.
  - For now, we will assume there is an associate page frame

Virtual page #	Valid	Physical Page Number
…	0	null
0x34	1	0x2
…	…	…

- With the physical page number, combine it with the page offset to get the physical address

Physical Page Number	Page Offset

  - In our example, with 0x34805, our physical address is 0x2805

Translation Done!

Lecture Outline

- High Level & Address Translation Refresher
- **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

- 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
  - Reference: two bits used for page replacement policy
  - 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.

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: Valid Bit & Frame

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

- Frame #
  - #bits = $\log_2$(num_frames)
  - The corresponding frame number for that page, if it is in memory

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: Reference & Dirty Bits

- Reference:
  - 2 bits
  - Used to keep track of how recently a page was used. This information is used for page replacement policies
- 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

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: Permission Bits

- 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 #**	**Reference**	**dirty**	**permissions**
0	0	----			
1	1	0	11	1	**R/W**
2	1	1	01	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

pollev.com/tqm

- Question: is there something we can remove from this to make 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			

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
- Page Table Details
- **Multi-Level Page Tables**
- Inverted Page Tables

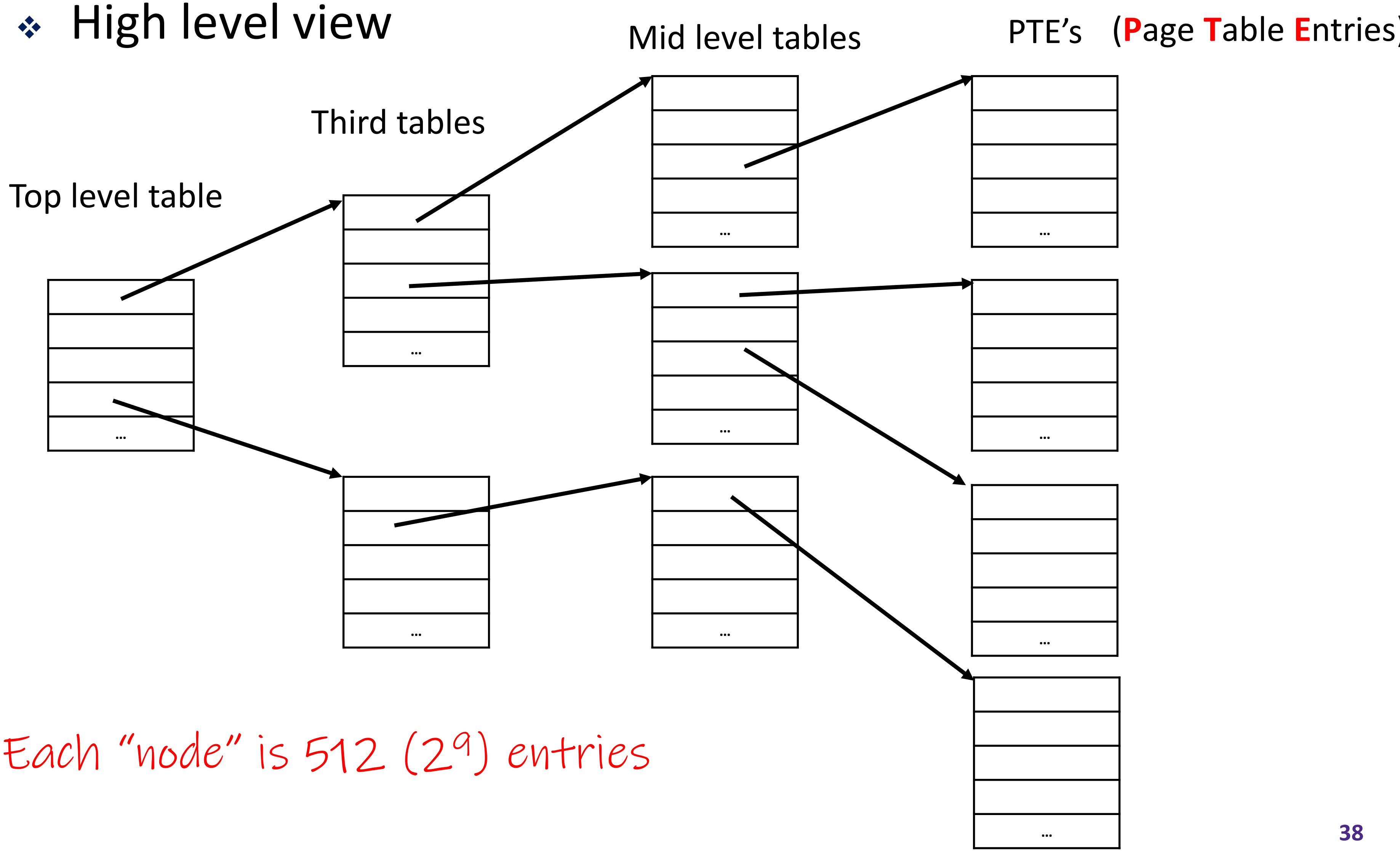
Multi Level Page Table

- If you've heard of a Trie or a prefix tree, then this is basically that
- 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 (ignore the remainder)

Ignored 16 bits	G offset 9 bits	U offset 9 bits	M offset 9 bits	PTE offset 9 bits	Page Offset 12 bits

Diagram

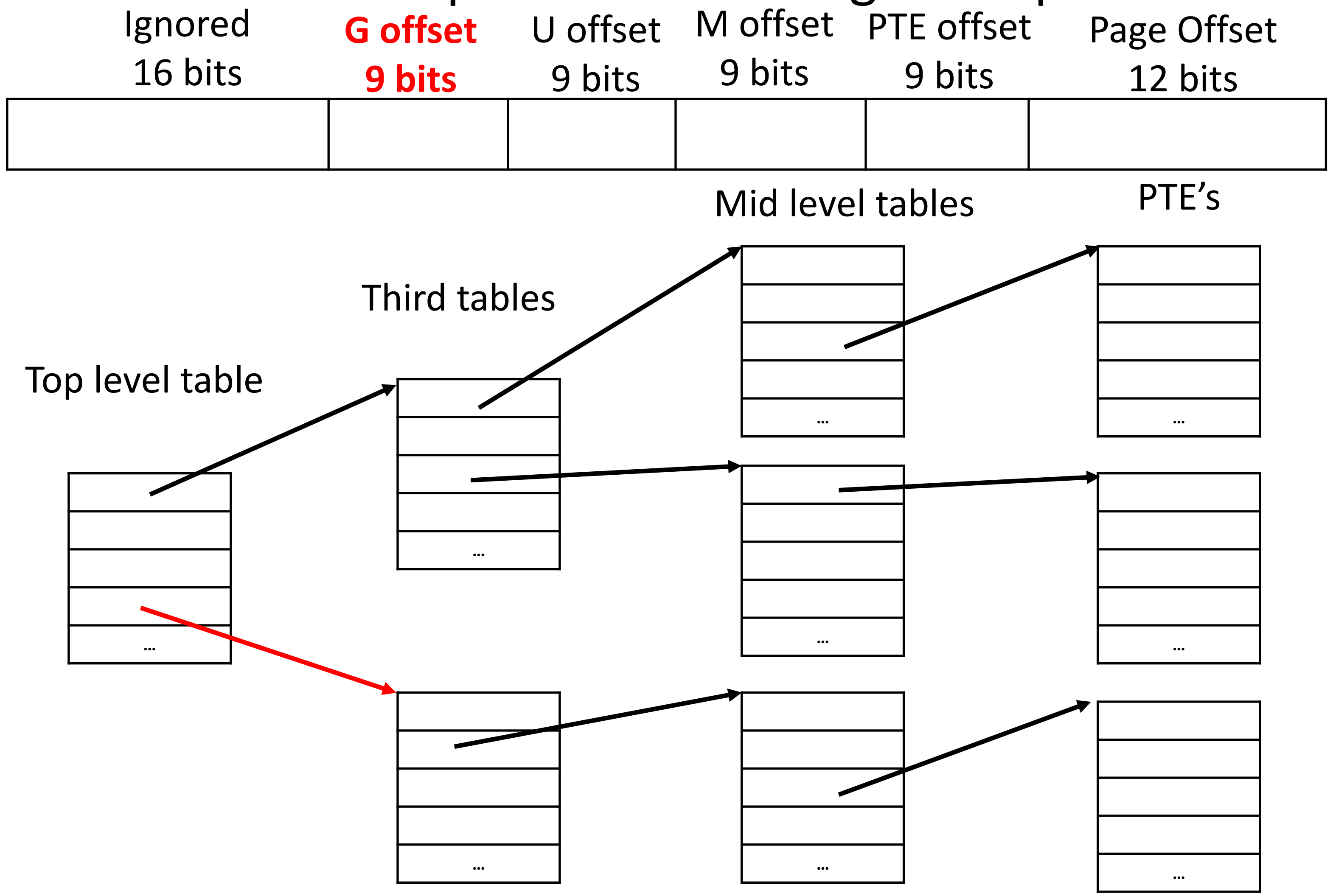
- High level view



Each "node" is 512 (2^9) entries

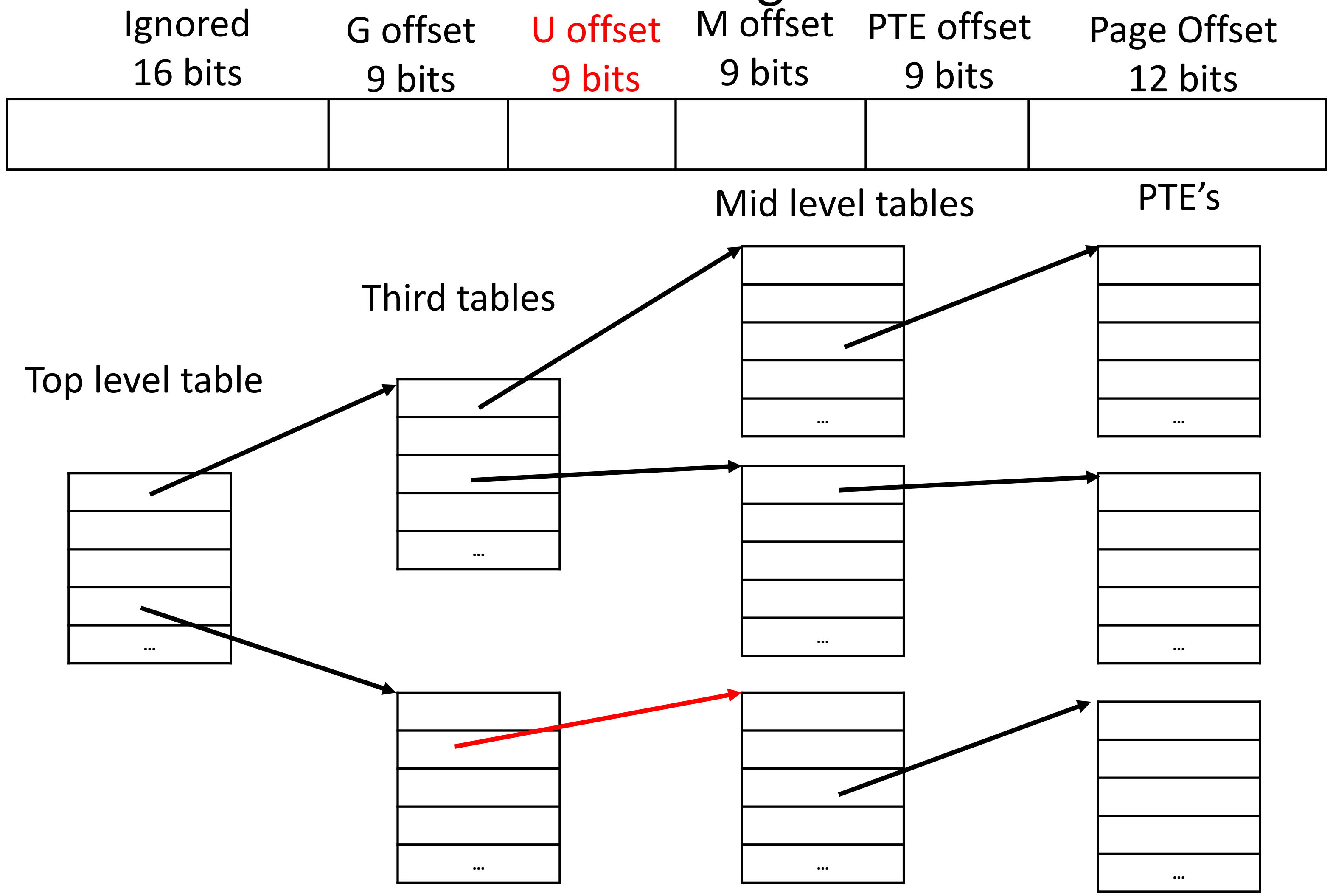
Looking up an address

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



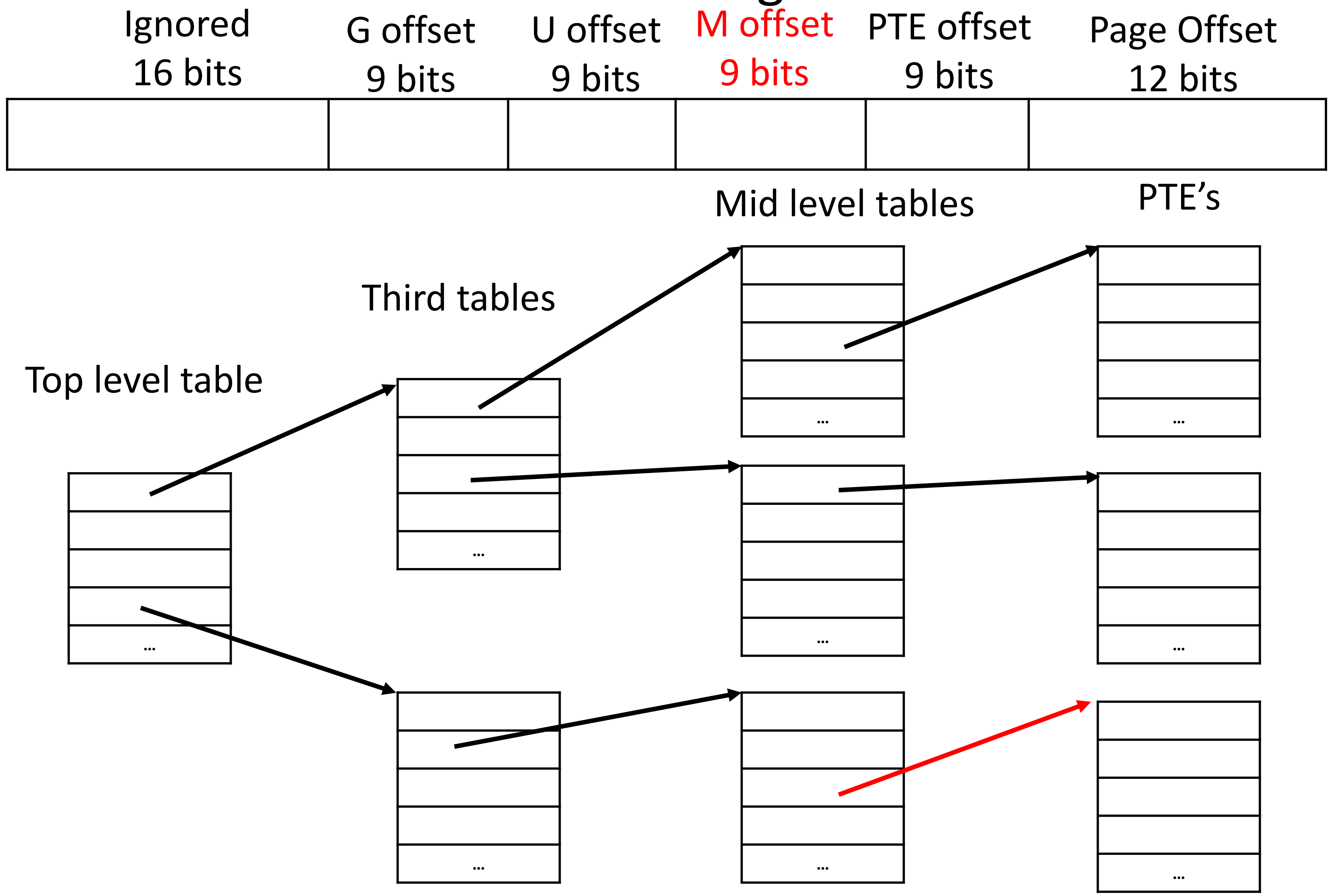
Looking up an address

- Index into next level table using the next 9-bit chunk



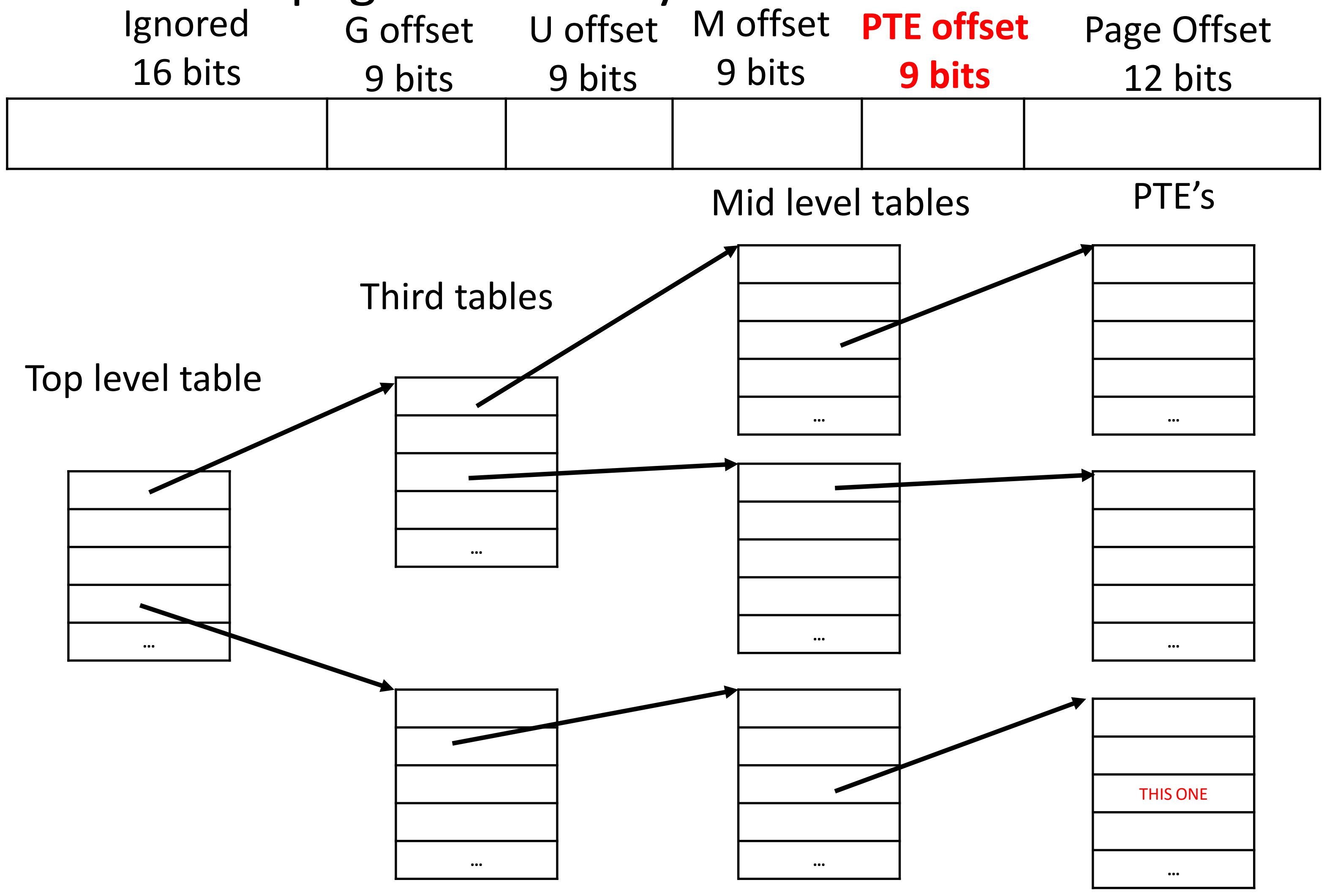
Looking up an address

- Index into next level table using the next 9-bit chunk



Looking up an address

- Access the page table entry based on the last 9 bits



Why 9 bits?

- Why is each index into a level of the page table 9 bits?
  - 9 bits = 2^9 = 512 entries into each "node"

- Each entry is just a pointer to the next level table
  - A pointer on a 64-bit machine is 8 (2^3) bytes
  - A page table entry is also at max 8 bytes

- Any guesses?

Why 9 bits?

- Why is each index into a level of the page table 9 bits?
  - 9 bits = 2^9 = 512 entries into each "node"

- Each entry is just a pointer to the next level table
  - A pointer on a 64-bit machine is 8 (2^3) 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 ☺ (more later)

Lecture Outline

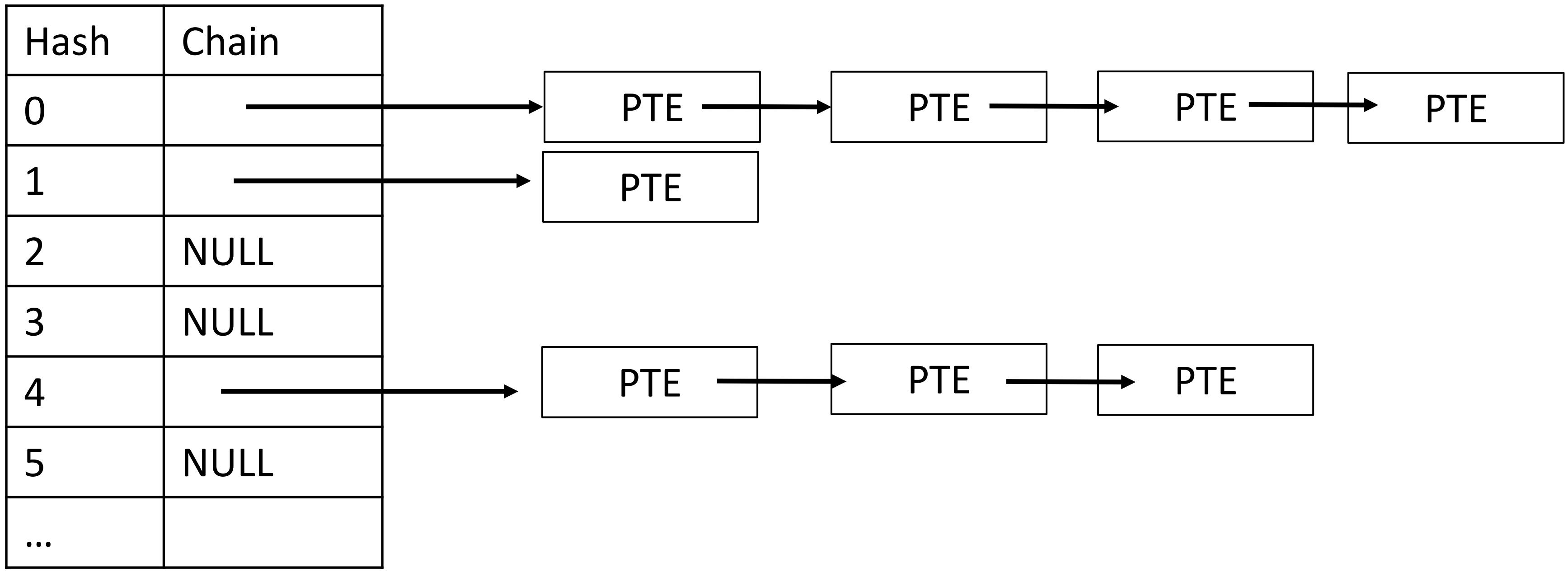
- High Level & Address Translation Refresher
- Page Table Details
- Multi-Level Page Tables
- **Inverted Page Tables**

Another way we can save space

- Idea: there are a lot more virtual pages than there are physical pages…
- Why not just have one entry per physical page?
- Would be one global page table since it is based on physical memory
  - Still need a way to enforce process isolation
- Implemented essentially as a changing hash table

Diagram

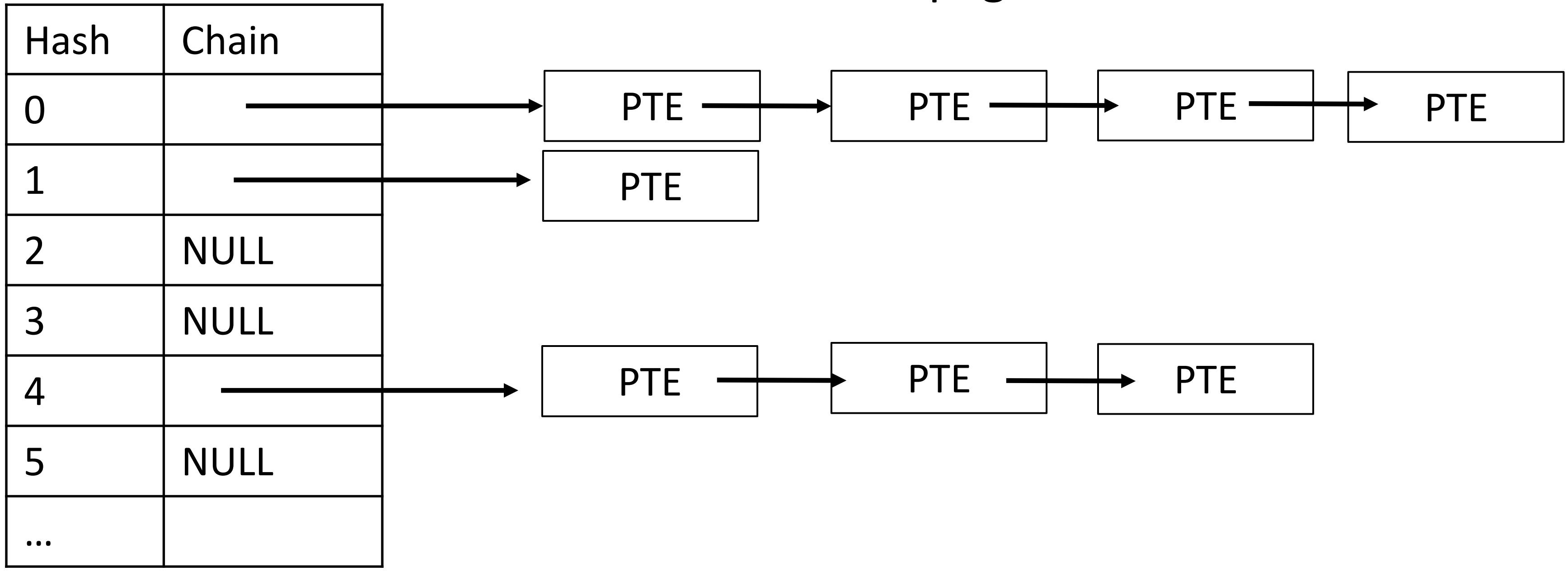
- Chaining Hash Table
  - Hash: the If a process wants to lookup to see if a page is in physical memory, it combines the target virtual page number and its process id to create the hash



Hash	Chain
0	
1	
2	NULL
3	NULL
4	
5	NULL
…	

Diagram

- Inspecting the chain
  - Once it find the corresponding chain, it iterates through the PTE's in the chain to see if any are for the corresponding virtual page
  - The PTE must store the virtual page num and the PID so it can be validated as the correct page



Analysis

- Potentially faster, potentially slower. Depends on how long the chains in the table get.

- Uses up a LOT less space, only 1 PTE per frame that is being used.