CPSC313: Computer Hardware and Operating Systems

Unit 5: Virtual Memory

(5.3) Virtual Memory: How does it actually work?

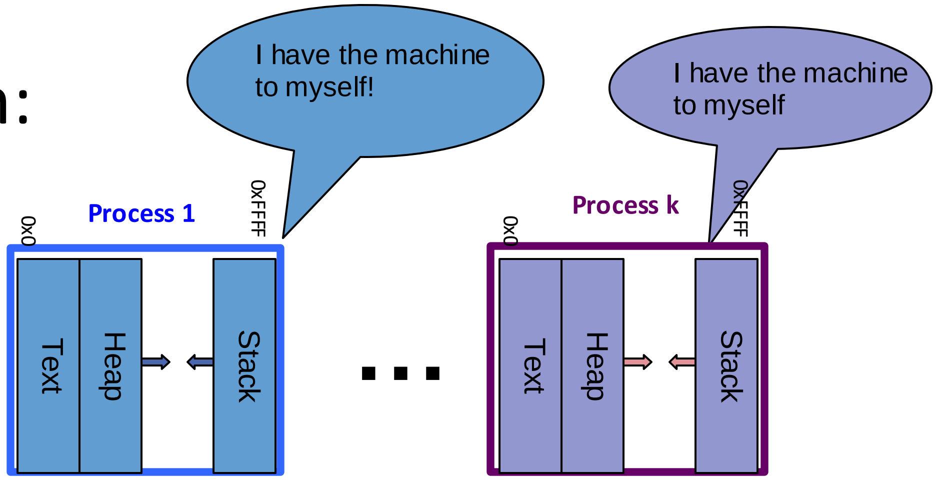
Administrivia

- Quiz 4 retakes finishing this week
- Quiz 5 coming next week
- Lab 10 due Sunday
- Exam period office hours schedule coming soon!

- **As always: check Canvas for all details!**

Context

We want the fiction of process isolation:



How do we get there?

- Virtual memory enforces isolation; translation/faulting supports virtual addressing
- We need to trust a resource "manager"– the OS:
  - Privileged (kernel/supervisor privilege, vs. user privilege)
  - Need to safely escalate privilege

Today

- Roadmap:
  - **Virtual memory (VM)** as a case study of how the OS and hardware work together and how/when control transfers.
  - To do that, we're going to first dive into how VM works.
  - We'll figure out what problem we are trying to solve and approaches to solving it.
  - Then we'll look at existing real world solutions.
- Learning Outcomes
  - Explain why we organize memory in pages
  - Define: Translation Lookaside Buffer (TLB)
  - Figure out how many bits are necessary to represent virtual and physical page numbers

Recall: (You’ve seen this!)

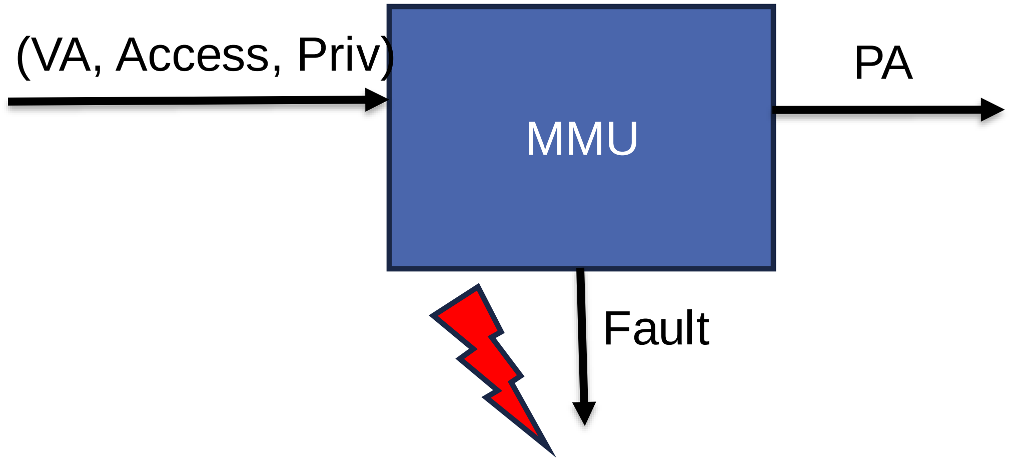
- We ask the **hardware** to map a triple of:
  - Virtual address (VA)
  - Access mode (read, write, execute)
  - **Privilege** level

  to a physical address

Recall: (You’ve seen this!)

- We ask the **hardware** to map a triple of:
  - Virtual address (VA)
  - Access mode (read, write, execute)
  - **Privilege** level

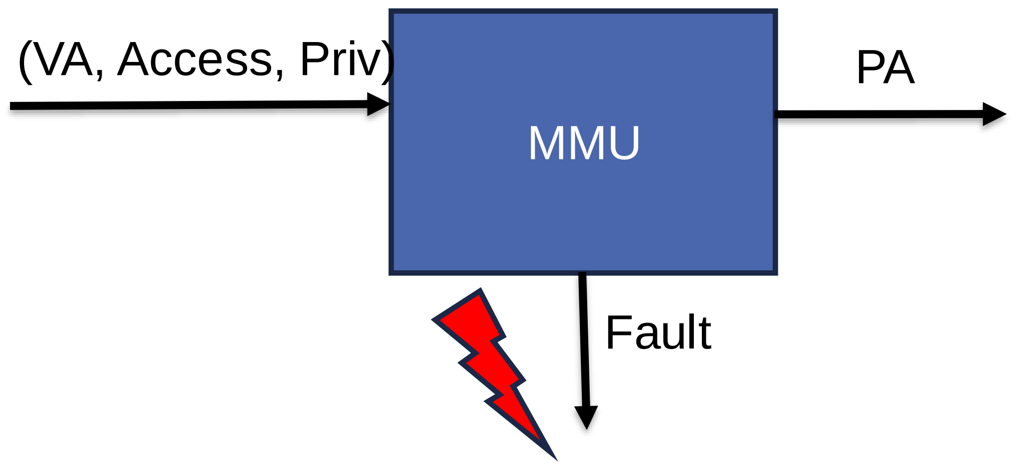
  to a physical address
- The hardware will either:
  - Produce a **physical address**
  - **Fault**, due to:
    - The address is invalid
    - The type of access is not allowed to the memory requested
    - The process requesting access does not have the appropriate privilege level to access the memory
    - The data is not in memory (it is on disk and hasn’t been loaded yet)



Recall: (You've seen this!)

- We ask the **hardware** to map a triple of:
  - Virtual address (VA)
  - Access mode (read, write, execute)
  - **Privilege** level

  to a physical address
- The hardware will either:
  - Produce a **physical address**
  - **Fault**, due to:
    - The address is invalid
    - The type of access is not allowed to the memory requested
    - The process requesting access does not have the appropriate privilege level to access the memory
    - The data is not in memory (it is on disk and hasn't been loaded yet)
- When the hardware faults: software (the OS) takes over
  - pre-class material explains how



PA: Physical Address
VA: Virtual Address

VM Mapping:

- Goal: We need a map from triples to physical address:
  - (VA, access, priv) => PA | FAULT
  - Physical addresses are not necessarily contiguous or ascending
  - We would like to avoid fragmentation.
- How might you implement this?
  Discuss with your neighbor!

PA: Physical Address
VA: Virtual Address

VM Mapping: The Dumbest Idea Ever

- Goal: We need a map from triples to physical address:
  - (VA, access, priv) => PA | FAULT
  - Physical addresses are not necessarily contiguous or ascending
  - We would like to avoid fragmentation.
- How might you implement this?
  - Store a mapping for every byte in the virtual address space
  - Critique:

PA: Physical Address
VA: Virtual Address

VM Mapping: The Dumbest Idea Ever

- Goal: We need a map from triples to physical address:
  - (VA, access, priv) => PA | FAULT
  - Physical addresses are not necessarily contiguous or ascending
  - We would like to avoid fragmentation.
- How might you implement this?
  - Store a mapping for every byte in the virtual address space
  - Critique:
    - The mapping table would be huge!
    - And wait -- we know about caching -- could you have a different mapping for every element of a cache line??? What a terrible idea.
    - And you've taught us about file systems where blocks are larger than cache lines. You probably don't even want a mapping for every cache line!

PA: Physical Address
VA: Virtual Address

VM Mapping: A Better Idea

- Goal: We need a map from triples to physical address:
  - (VA, access, priv) => PA | FAULT
  - Physical addresses are not necessarily contiguous or ascending
  - We would like to avoid fragmentation.
- How might you implement this?
  - Just like we divide files into blocks, let's divide both virtual address spaces and memory (DRAM) into blocks. We shall call them pages.
  - Store one mapping for each page.
  - Critique:

PA: Physical Address
VA: Virtual Address

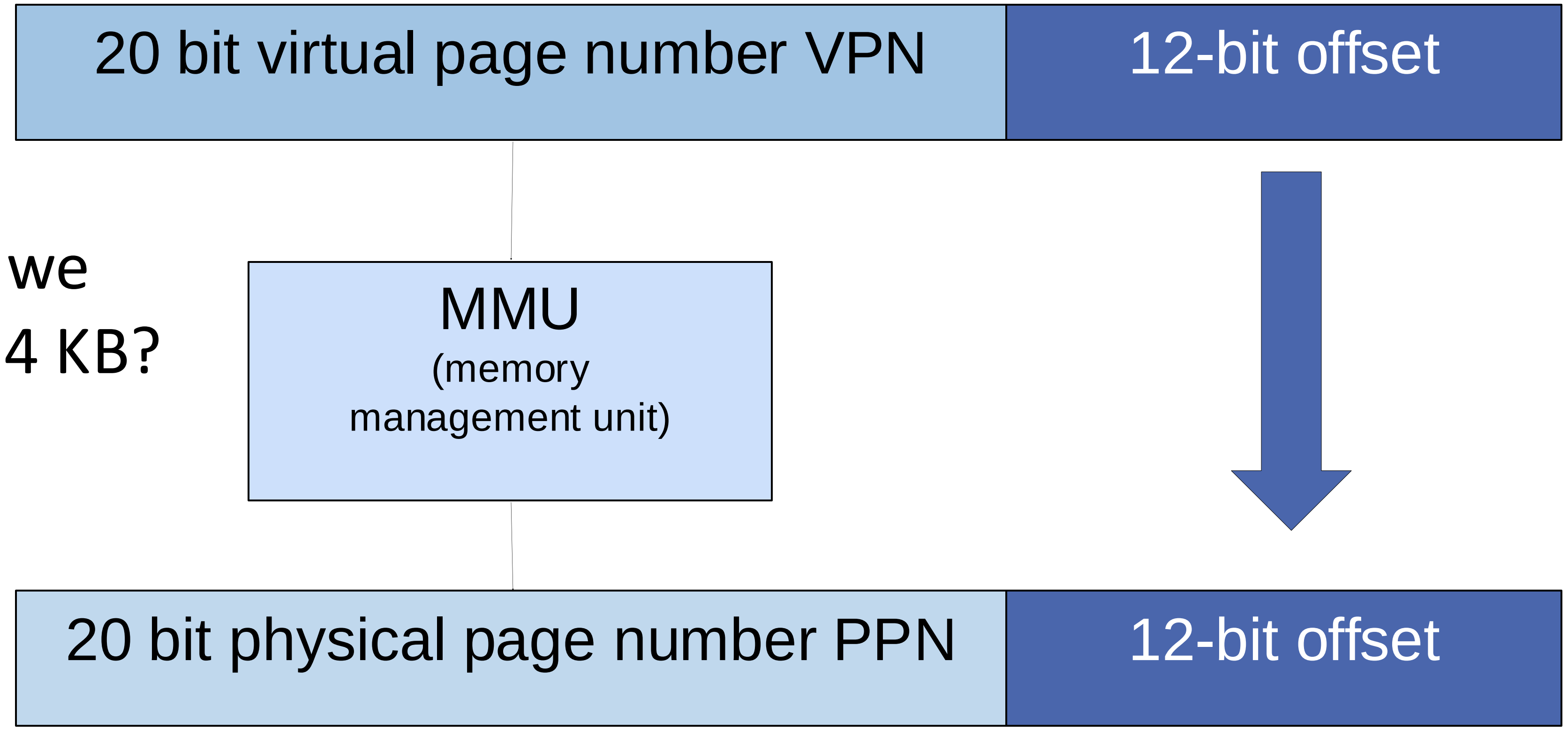
VM Mapping: A Better Idea

- Goal: We need a map from triples to physical address:
  - (VA, access, priv) => PA | FAULT
  - Physical addresses are not necessarily contiguous or ascending
  - We would like to avoid fragmentation.
- How might you implement this?
  - Just like we divide files into blocks, let's divide both virtual address spaces and memory (DRAM) into blocks. We shall call them pages.
  - Store one mapping for each page.
  - Critique: That seems far more sensible.
    - How large are pages?
    - How do you implement this map?

Dividing the bits in an address: x86-32

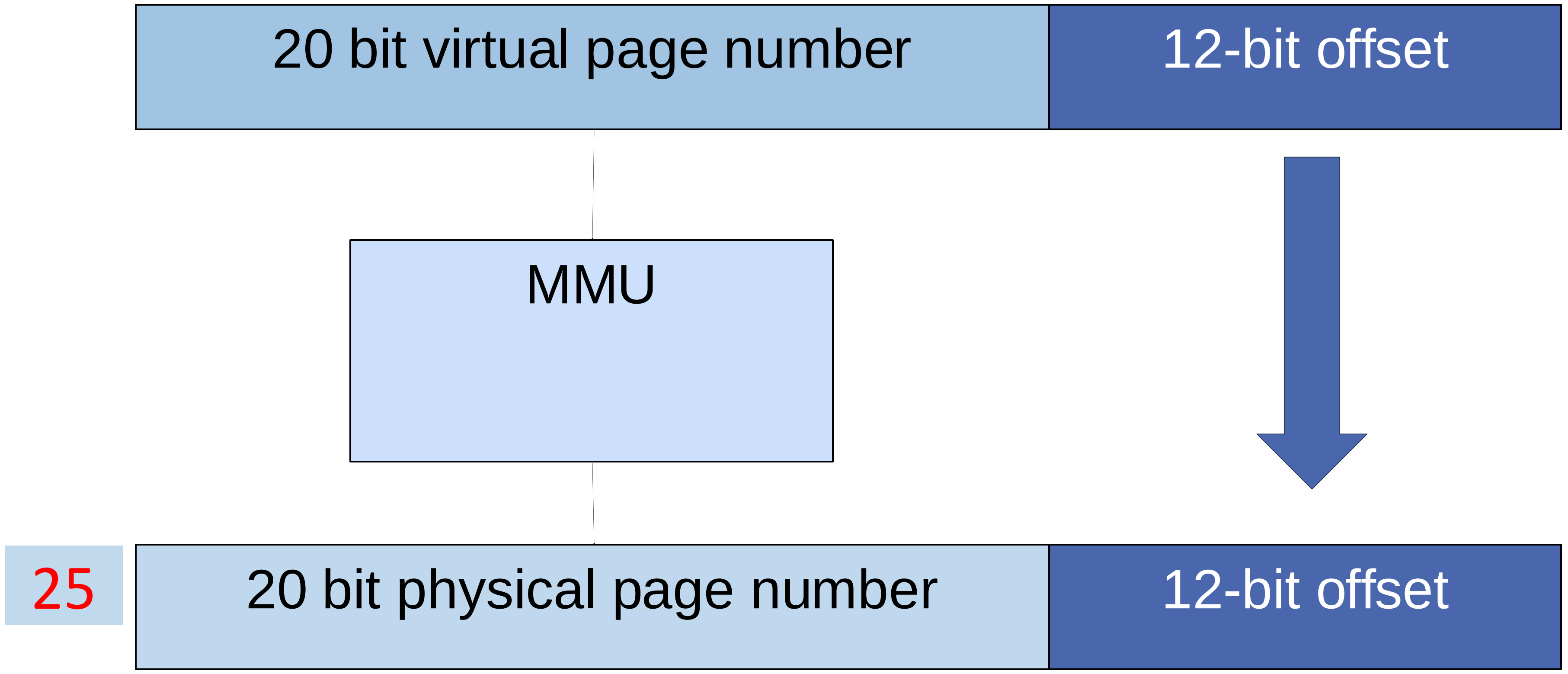
On x86:

- Pages are 4 KB
  - i.e., 2^{12} B
- How many bits do we need to represent 4 KB? 12


20 bit virtual page number VPN
12-bit offset
MMU
(memory management unit)
20 bit physical page number PPN
12-bit offset

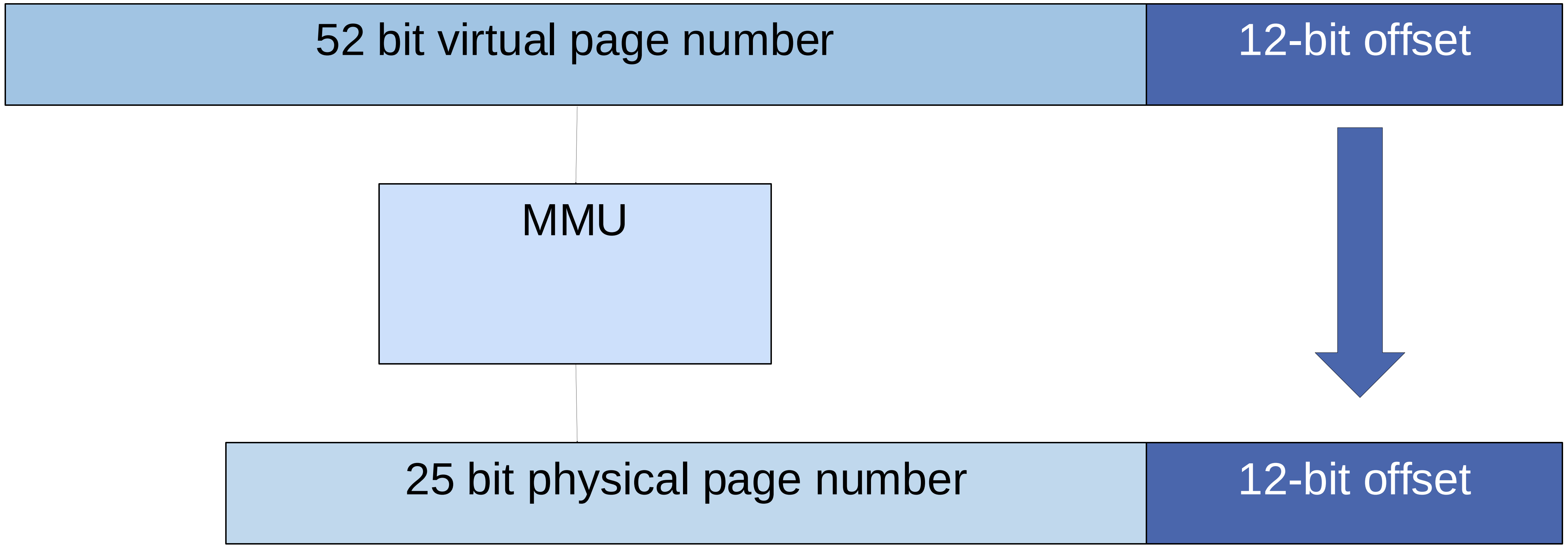
But wait …

- What if we have a 32-bit VAS, but our machine has more than 4 GB of memory?



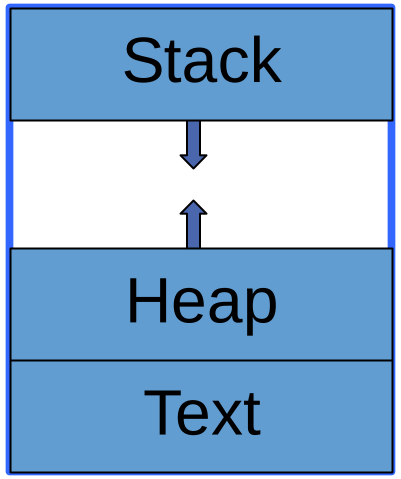
But wait – 2 …

- What if we have a 64-bit VAS, while our machine has much less main/physical memory DRAM?



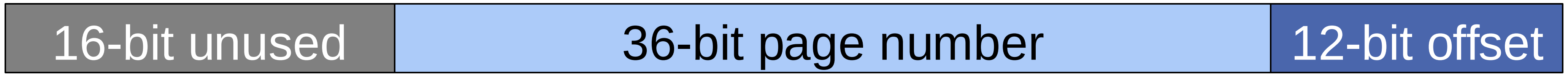
But wait – 2 …

- What if we have a 64-bit VAS, while our machine has much less than that much main/physical memory DRAM?
  - That's fine!
  - We run in that regime all the time: the Intel x86-64 runs with a 48-bit address space, and pretty much none of us have 256 TB machines, and it all works.
  - How?
    - Our processes don't use the entire 48-bit address space.
    - Sometimes you **want** "holes" in memory.
    - You can run a process without having all its pages in memory.



Example: x86-64

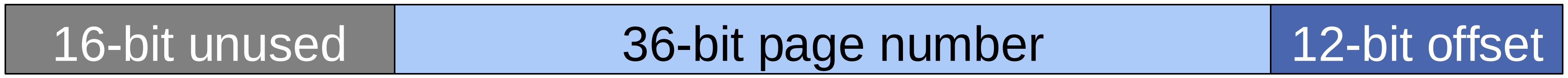
- An instructor machine:
  - 3.3 GHz Dual-Core Intel Core i5
  - 64-bit processor (Kaby Lake)
    - 4 KB page size
    - Of the 64 bits in the virtual address space:
      - 12 bits are page offset
      - 16 bits are unused
      - **36 bits are the virtual page number**



  - 16 GB 2133 MHz LPDDR3 DRAM
    - How many bits of physical page number do we need for this machine?

Example: x86-64

- An instructor machine:
    - 3.3 GHz Dual-Core Intel Core i5
    - 64-bit processor (Kaby Lake)
        - 4 KB page size
        - Of the 64 bits in the virtual address space:
            - 12 bits are page offset
            - 16 bits are unused
            - **36 bits are the virtual page number**

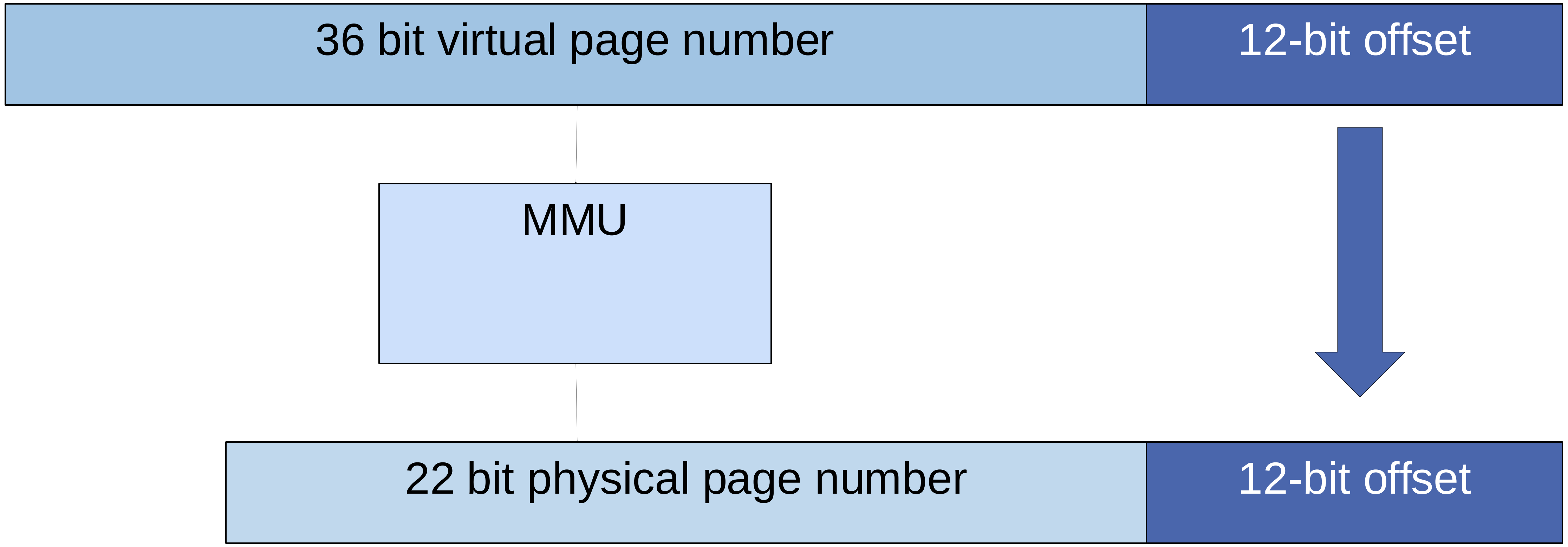


- 16 GB 2133 MHz LPDDR3 DRAM
    - How many bits of physical page number do we need for this machine?
      16GB needs 30+4 bits for addressing. 34 total bits – 12 offset bits = 22 phys. page bits

*) Note: we ignore devices for now and assume the physical address space is just DRAM...

A typical 2023 machine

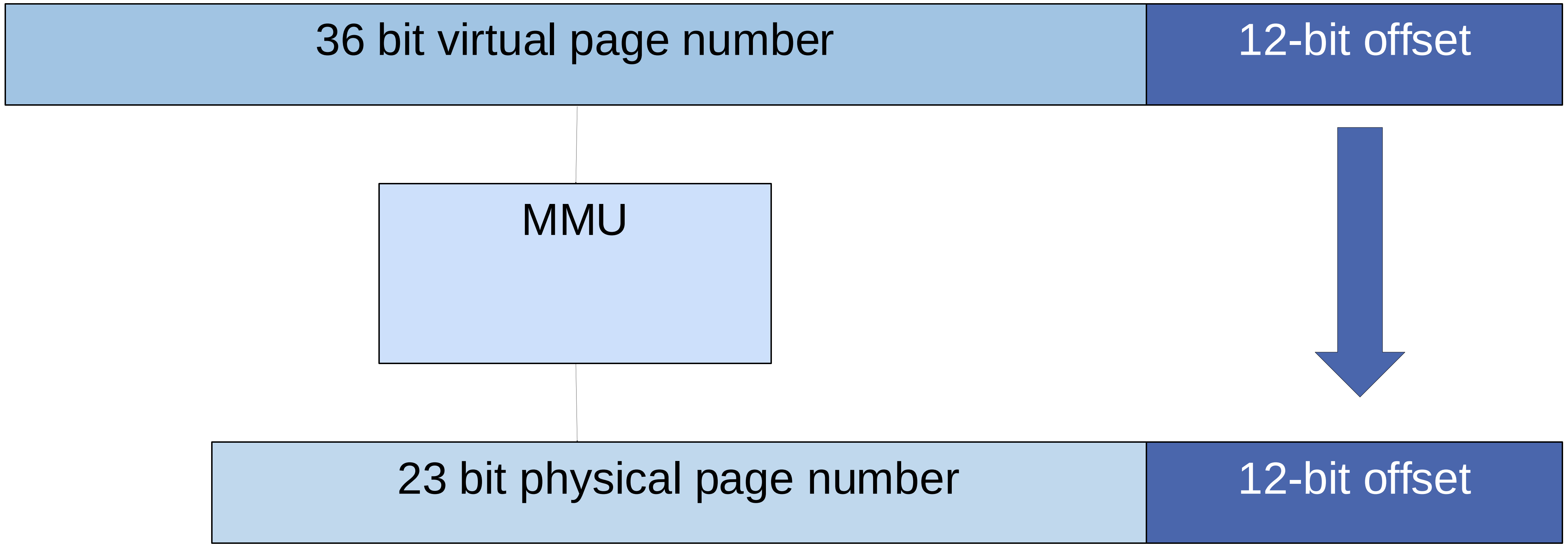
- 48-bit Virtual Address Space (256 TB*)*
- 34-bit Physical Address Space* (16GB)



*) Note: we ignore devices for now and assume the physical address space is just DRAM...

What if I had more memory?

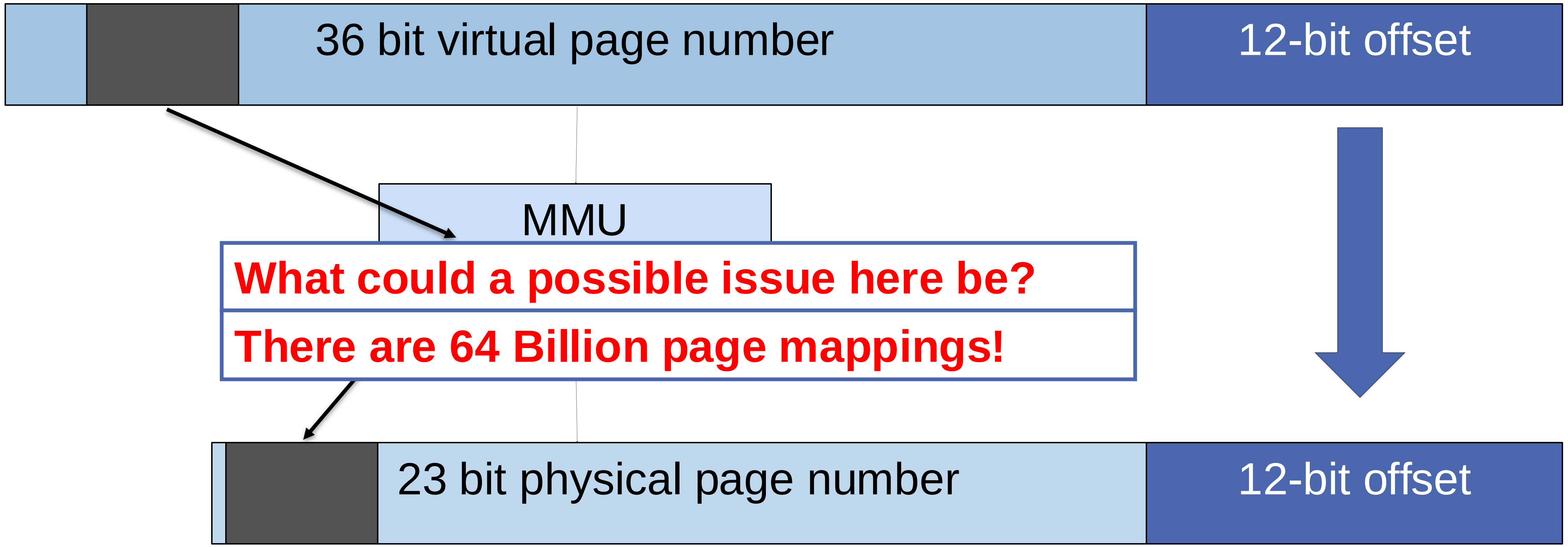
- 48-bit Virtual Address Space (256 TB)
- 35-bit Physical Address Space* (32GB)



*) Note: we ignore devices for now and assume the physical address space is just DRAM...

What if I had more memory?

- 48-bit Virtual Address Space (256 TB)
- 35-bit Physical Address Space* (32GB)



What could a possible issue here be?

There are 64 Billion page mappings!

What Does our MMU do?

- Let’s start with simple HW to go with our simple map.
- Rather than ask the MMU to store all the mappings, we can have the MMU cache some mappings.
- Why might this be OK?

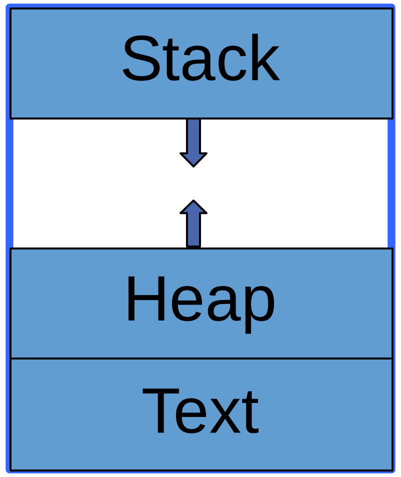
What Does our MMU do?

- Let’s start with simple HW to go with our simple map.
- Rather than ask the MMU to store all the mappings, we can have the MMU cache some mappings.
- Why might this be OK?
  - Instructions: These are pretty tiny and a lot of them fit in a single page, so a single mapping will likely get a lot of hits.
  - Data: Caching is premised on locality
    - If locality makes caching work, it ought to make caching translations work, too

Translation Lookaside Buffers (TLB)

- The simplest form of MMU is basically just a cache of translations (and permissions and protections).

Code (i.e., instructions)



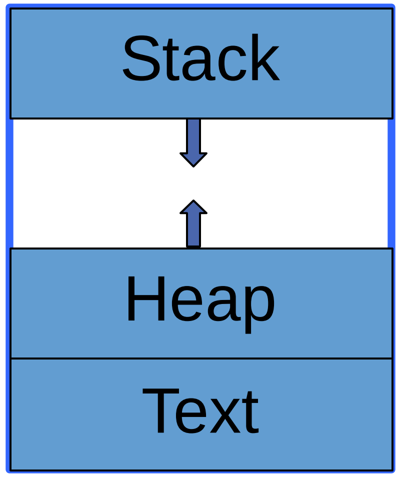
Virtual Page Number	Physical Page Number	Permissions	Mode
`0x000000000000`	`0x10001`	R/X	Supervisor
`0x000000010003`	`0x0CAFE`	R/X	User
`0x000FFF000000`	`0x10011`	R/W	User
`0x000000010005`	`0xFA0CE`	R/W	User
`0x000000010004`	`0x54321`	R	User

aka Kernel

Trying to **write** to this page will cause a FAULT (bad permissions)

Translation Lookaside Buffers (TLB)

- The simplest form of MMU is basically just a cache of translations (and permissions and protections).



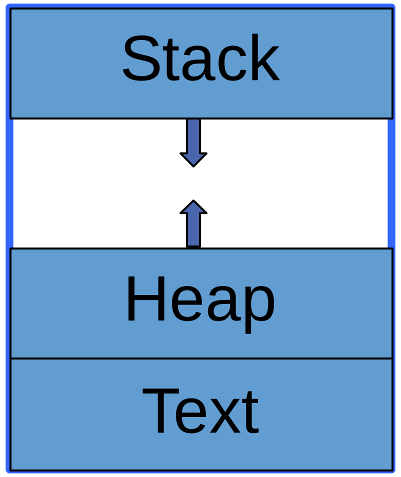
Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User

Stack

Trying to **execute** data on this page will cause a FAULT (bad permissions)

Translation Lookaside Buffers (TLB)

- The simplest form of MMU is basically just a cache of translations (and permissions and protections).



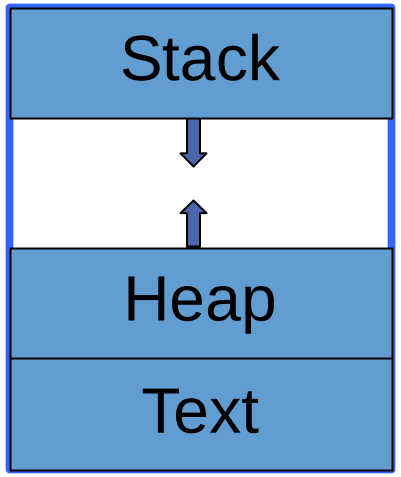
Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User

Heap

Trying to **execute** data on this page will cause a FAULT (bad permissions)

Translation Lookaside Buffers (TLB)

- The simplest form of MMU is basically just a cache of translations (and permissions and protections).



Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User

Read only data

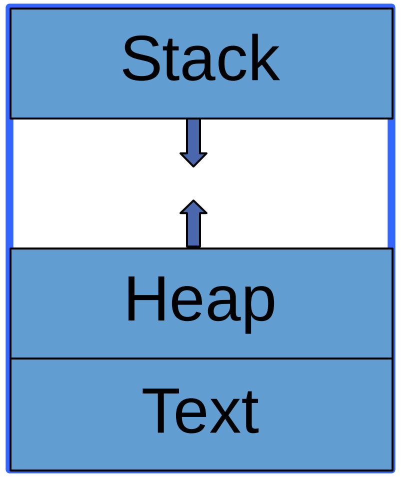
Trying to **write** or **execute** data on this page will cause a FAULT (bad permissions)

Translation Lookaside Buffers (TLB)

- The simplest form of MMU is basically just a cache of translations (and permissions and protections).

Operating System (code)

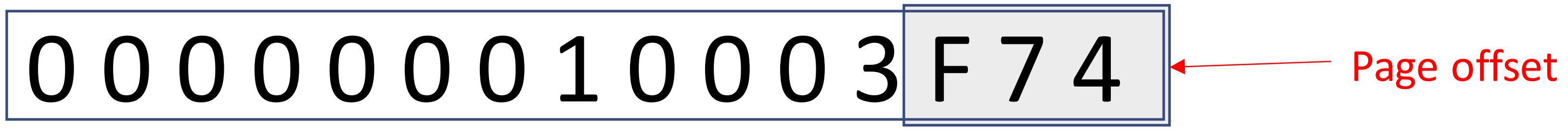
Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User



If a **user program** tries to access this page, it will produce a FAULT.

Translating a Virtual Address (1)

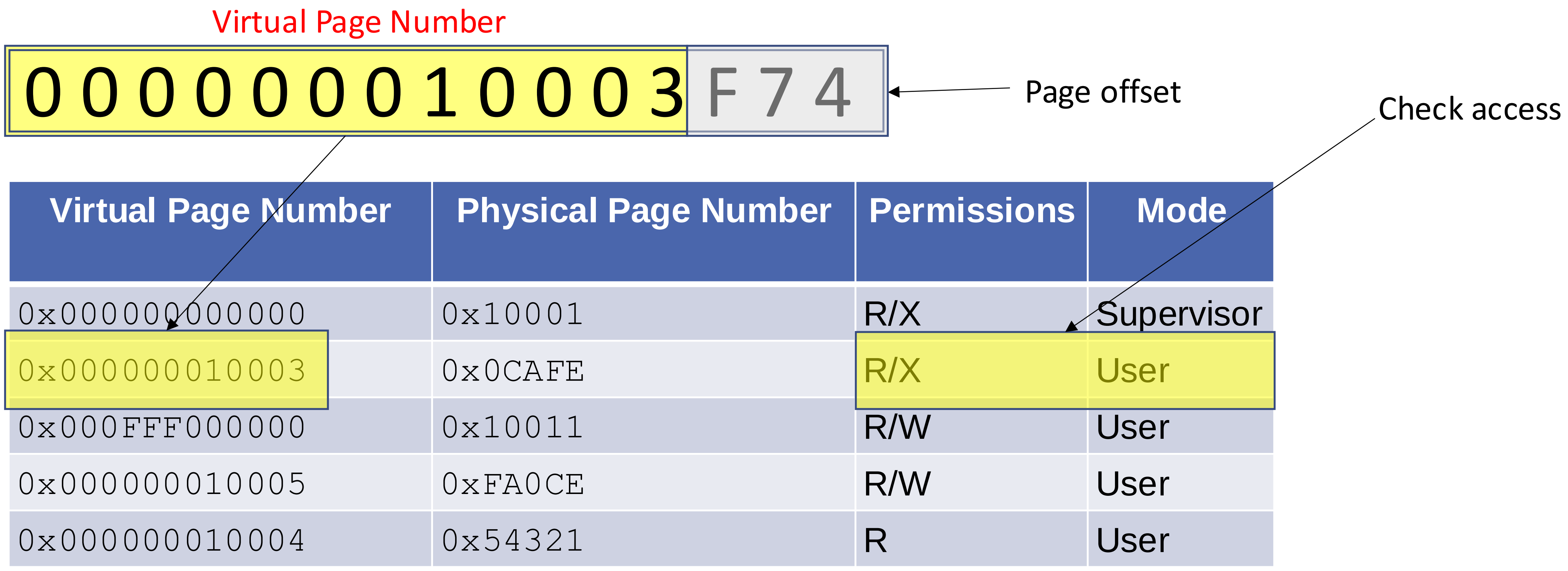
```
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x000000010003f74 goodbye`main(…)
* This is an execute access from user mode
```



Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User

Translating a Virtual Address (2)

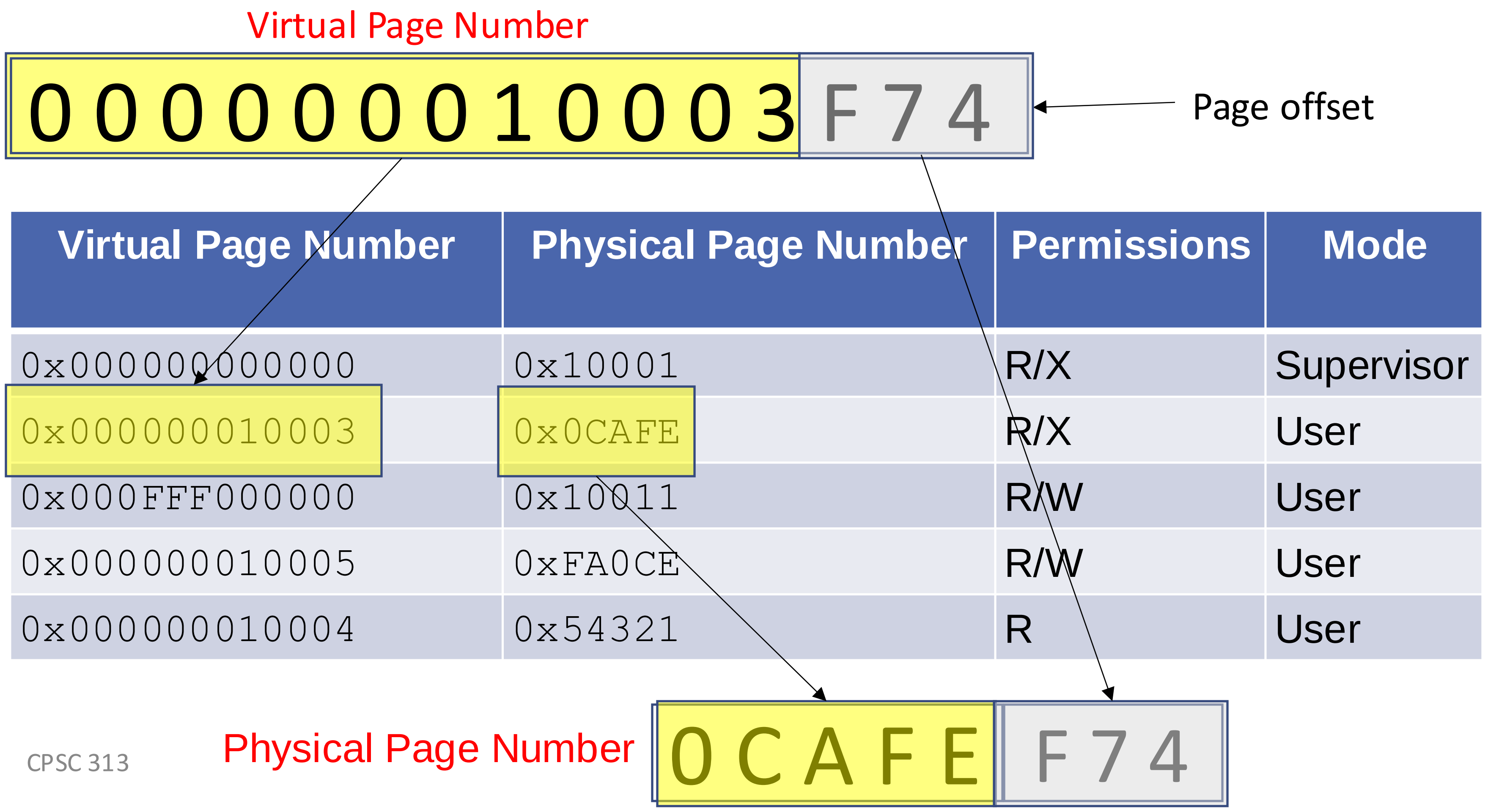
```
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000010003f74 goodbye`main(…)
* This is an execute access from user mode
```



Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User

Translating a Virtual Address (3)

```
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000010003f74 goodbye`main(…)
* This is an execute access from user mode
```

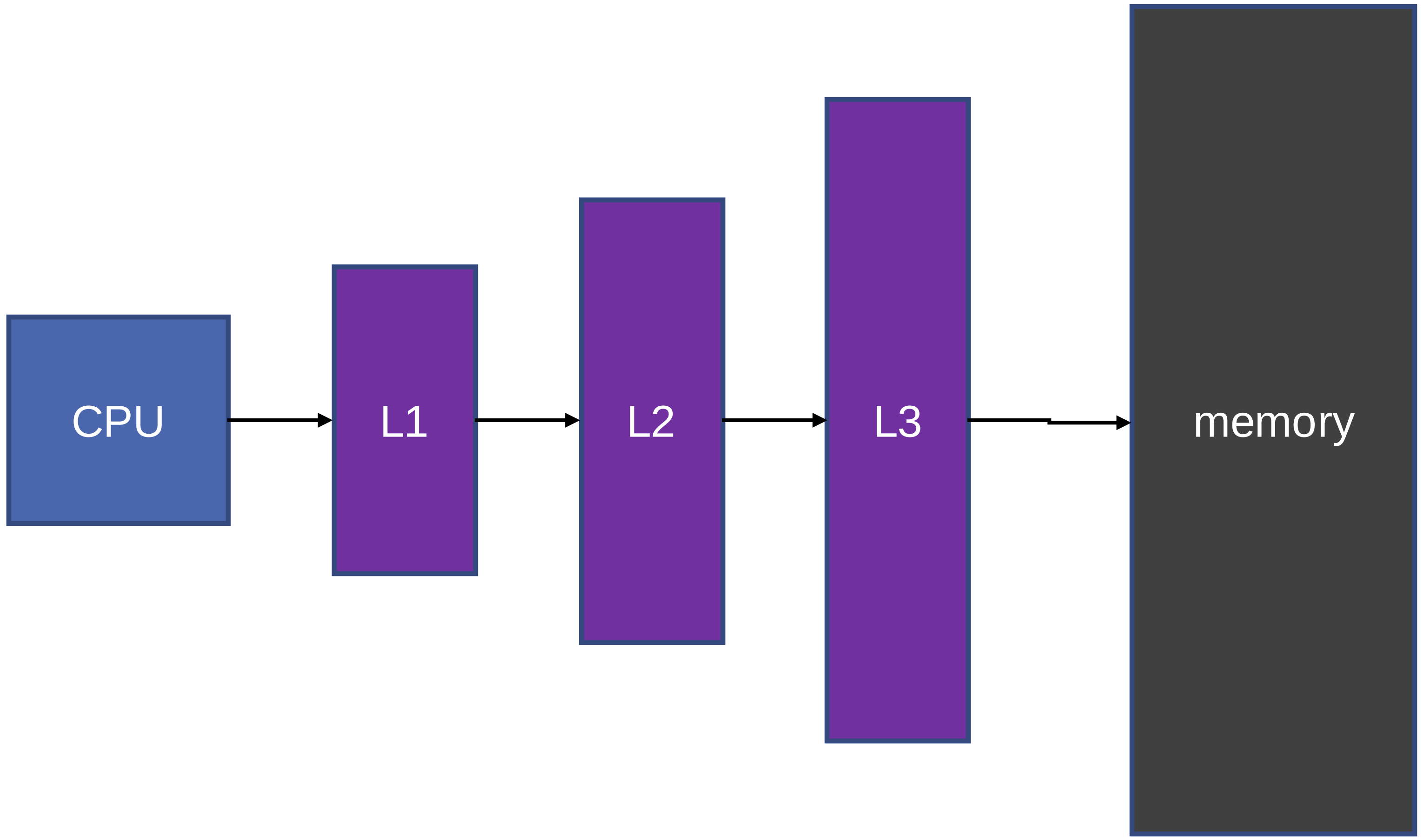


Virtual Page Number	Physical Page Number	Permissions	Mode
0x000000000000	0x10001	R/X	Supervisor
0x000000010003	0x0CAFE	R/X	User
0x000FFF000000	0x10011	R/W	User
0x000000010005	0xFA0CE	R/W	User
0x000000010004	0x54321	R	User

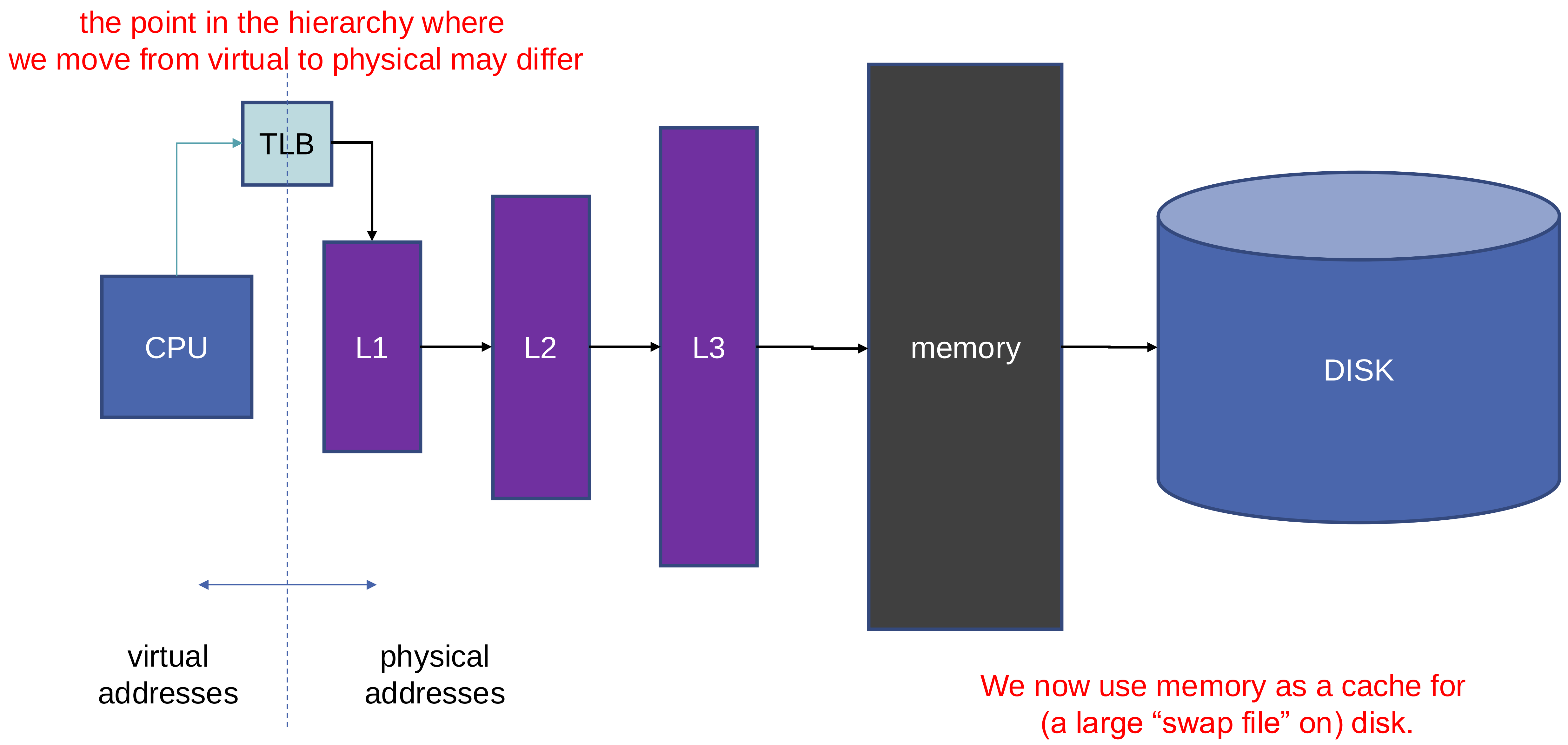
Takeaway

- Maximum virtual page number and physical page number can be different.
    - i.e., The virtual address space and physical address space can be different sizes.
- Virtual pages and physical pages are the same size.
    - The virtual page size is set by the hardware.
- The (maximum) size of the virtual address space is also determined by the hardware.
    - Therefore the number of bits in a virtual page number is also determined by the hardware.
- The (maximum) number of physical pages is also determined by the hardware
    - Therefore the number of bits in a physical page number is also determined by the hardware.
- The actual number of physical pages is specific to a particular machine.
- A TLB is a fast, hardware cache that maps virtual addresses to physical addresses (by storing mappings from virtual page numbers to physical page numbers).

What things looked like before:



What things look like now (maybe):


the point in the hierarchy where
we move from virtual to physical may differ
TLB
CPU
L1
L2
L3
memory
DISK
virtual
addresses
physical
addresses

We now use memory as a cache for
(a large “swap file” on) disk.

In class activity – becoming the TLB!

Think Ahead to Next Time

Our full page table maps what the process thinks are its page numbers to the **real** page numbers in memory.

There are a **lot** of (potential) pages in our virtual address space.

That means the page table itself will be **big**.

Worse, there are tons of holes/gaps in our virtual address space, and we don’t want to waste memory storing those.

I wish we had a data structure that was good for this sort of mapping!!