13

Virtual Memory

Introduction to
Systems Software

wilkie

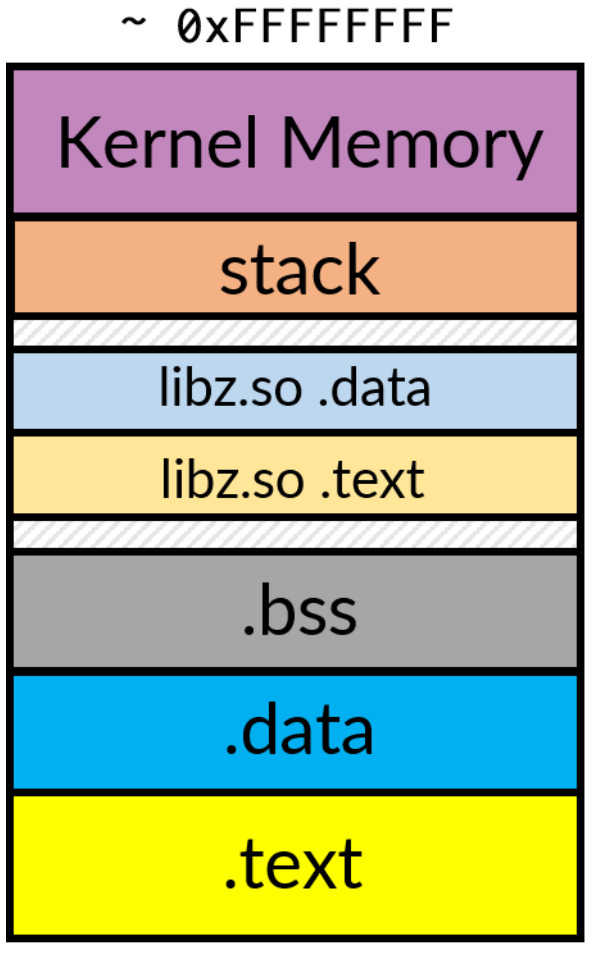
Spring 2019/2020

The Virtual

I just want to use this time to point out that “Virtual Reality” is an oxymoron.

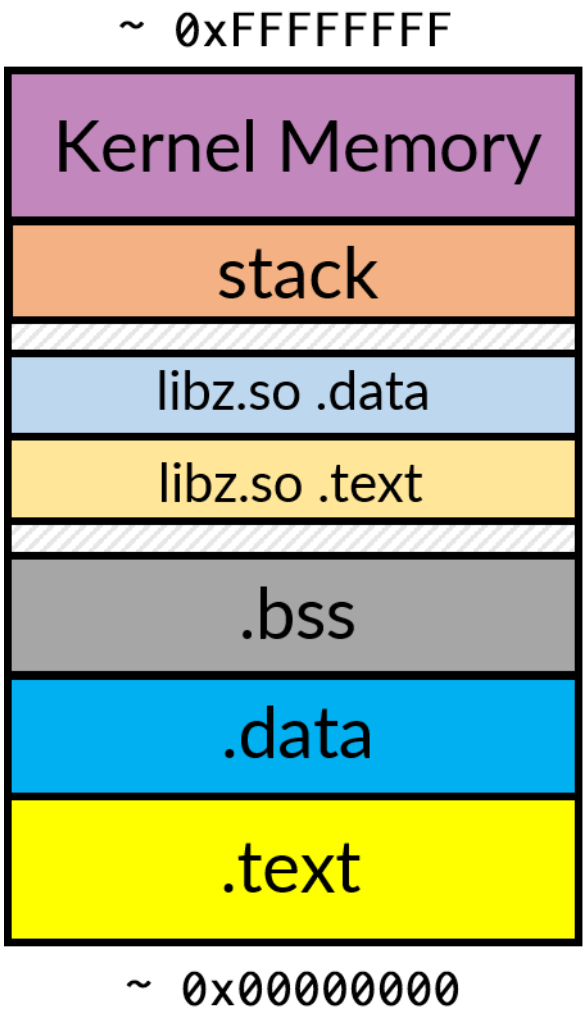
Our protagonist's journey so far…

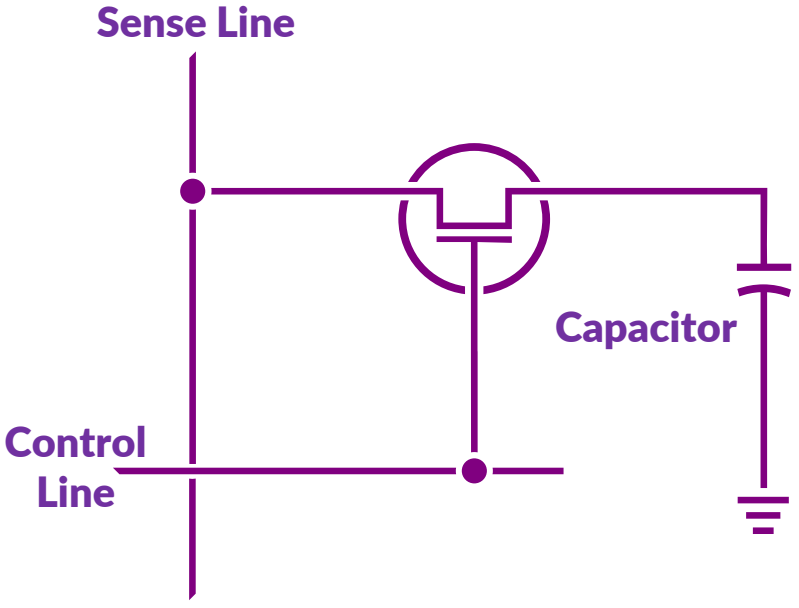
- Processes have an addressable memory space.
  - We call this an **address space**.
- Now, we know it has some obvious things…
  - The code (.text)
  - The data (.data)
  - A stack and some available space that can be allocated as we need it.
- We added the "Kernel" memory to our diagram.
  - This is the OS code and data.
  - For system processes to run, they need to be resident in memory as well.

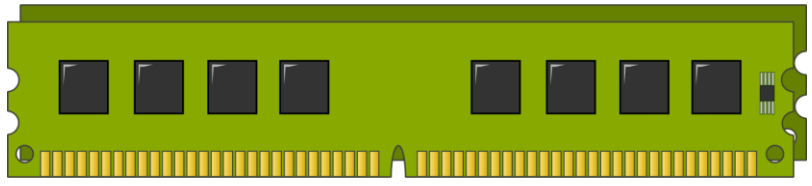


Random Access Memory

- Memory is a physical device.
- It is a **random access** device.
  - Allows the machine to access data at any point.
- As opposed to *sequential access*.



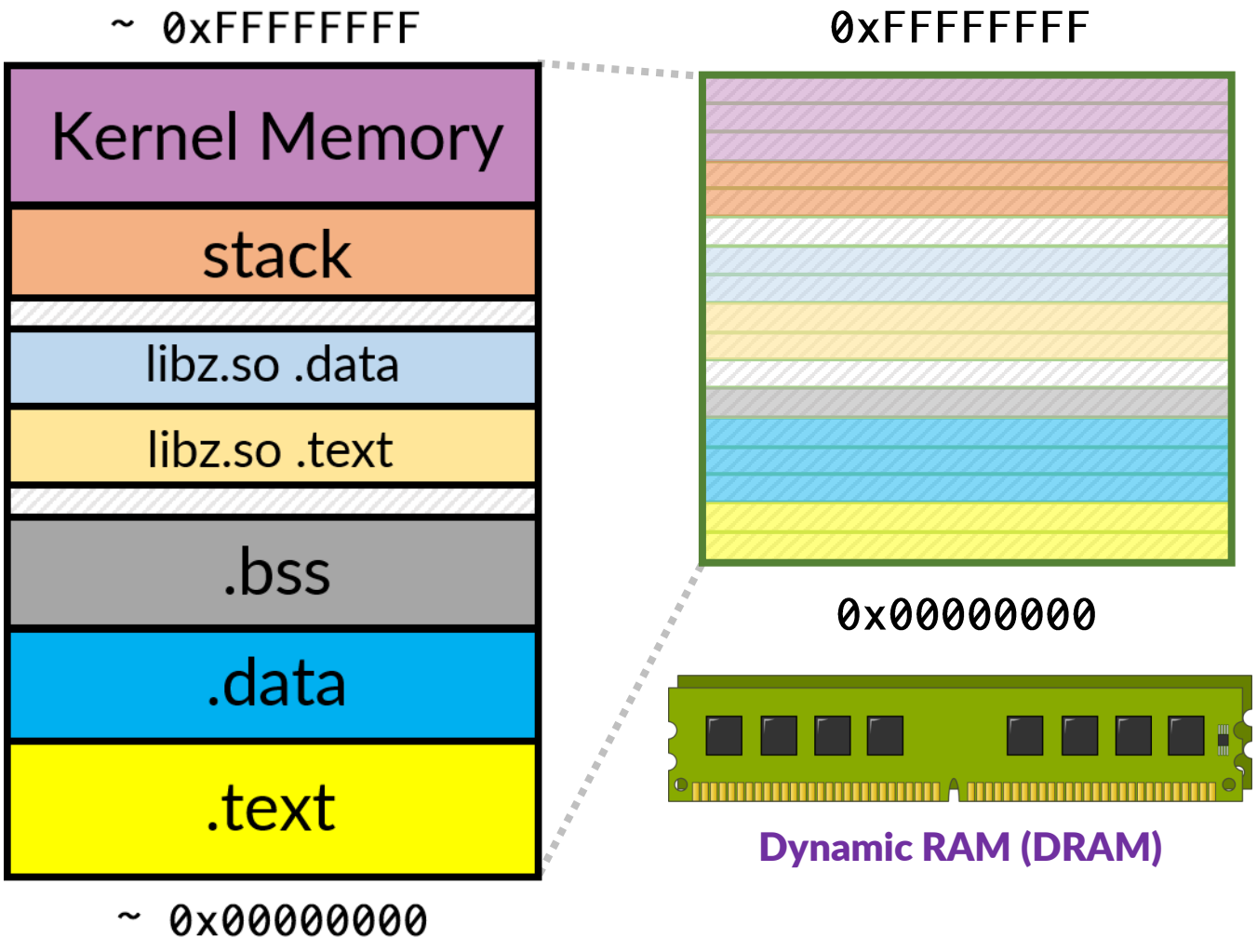




Dynamic RAM (DRAM)

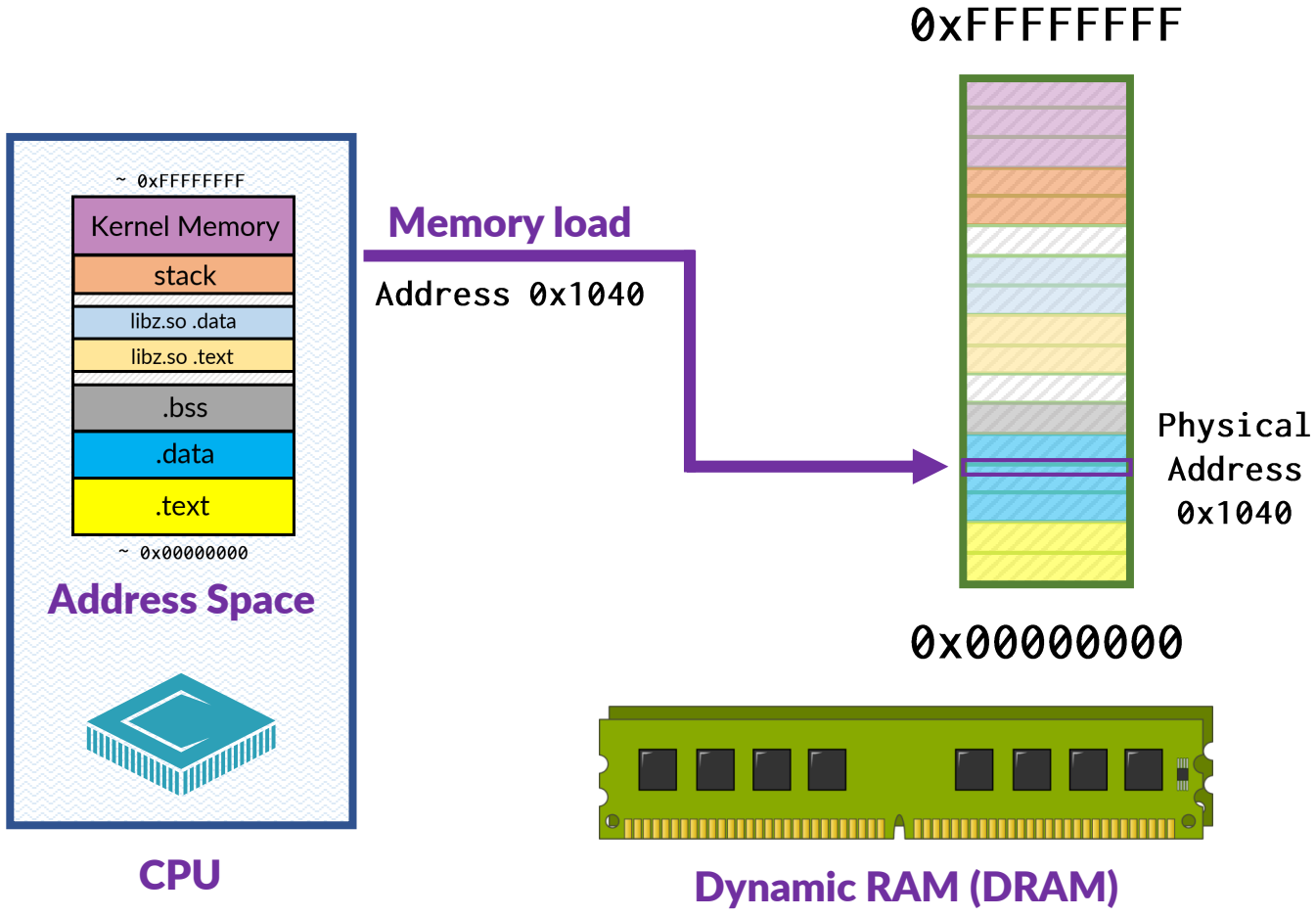
Let's get physical

- **Physical addressing** is when hardware relates a program address directly to the memory hardware.
- The program addresses are the exact physical locations of data.



Let's get physical

- Physical addressing is useful when you have only a single, simple process.
- Embedded devices (small, specific uses)
- Think of your toaster… or a thermostat.



The problem

- However, we don't always have such simple cases.
  - A general-purpose system, like our phones and desktop machines, can run a variety of programs.
  - We often have *multitasking operating systems* running many programs at the same time.

- When we have several processes, how to we manage memory?
  - How to present a consistent address space?
  - How to prevent other processes from interfering?
  - How to prevent address space fragmentation?

- The solution, as usual: *indirection*: Virtual Memory.

Virtual vs. Reality

- Just like “virtual reality,” we create a world that resembles reality, but it really is a facsimile.
- We can provide a scheme, backed by hardware, that allows memory addresses to be seen by programs in a specific place…
- Yet, those addresses differ from the actual physical memory location.

Angela Lansbury as Jessica Fletcher, Director Lee Smith
Murder, She Wrote; “A Virtual Murder”
Universal Studios, Universal Television, 1993

Consistency, dear Watson.

- When you write a program, do you write it deliberately for the memory layout of your OS?
  - No!
- The OS loads executables to specific places in memory.
  - The program expects data to be in a specific place.
  - The program expects memory to be “large enough.”
- So, we will look at one strategy to define well-known stretches of memory (virtually) that are mapped to physical memory (in reality.)

Segmentation

It's not just a type of fault.

Segmentation

- **Segmentation** is a virtual memory system where spans of physical memory called a **segment** are given a physical base address.

- The application refers to the virtual address by its segment index which is translated by hardware into the real address behind-the-scenes.

- Here, we have a **segment table** which defines two segments.
  - The first segment defines a range of addresses from `0x7fde0000` to `0x7fde2fff`.
  - Second is `0x23fa0000` to `0x23fa3fff`

Segment Table

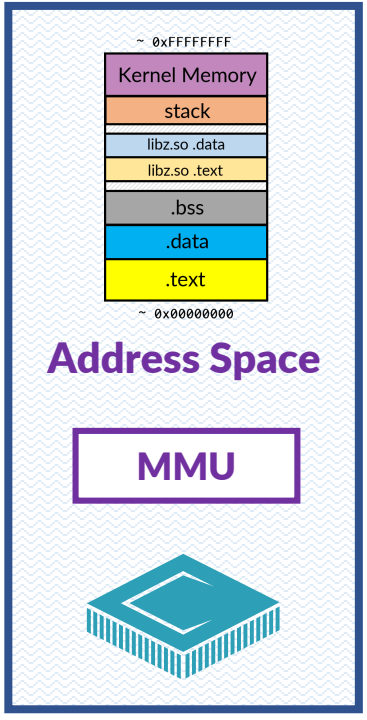
Index	Physical Base	Size
00	0x7fde0000	0x3000
01	0x23fa0000	0x4000
02	NULL	0

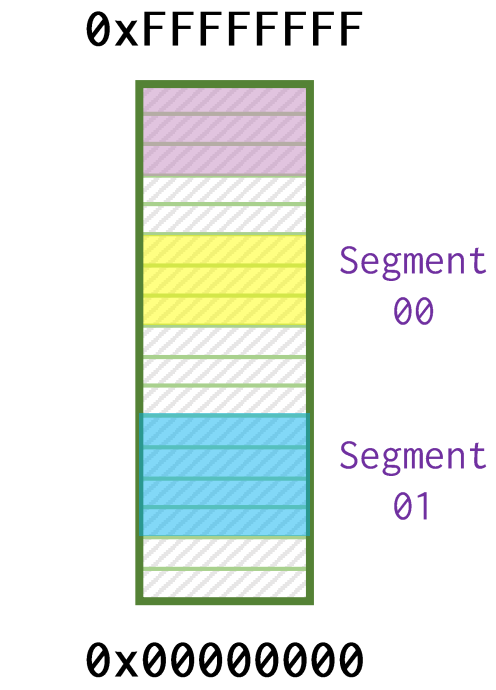
Address Translation

- The **Memory Management Unit** (MMU) is a hardware component of your CPU that translates virtual addresses to physical addresses.
- Here, it translates based on the segment table.

Segment Table

Index	Physical Base	Size
00	0x7fde0000	0x3000
01	0x23fa0000	0x4000
02	NULL	0



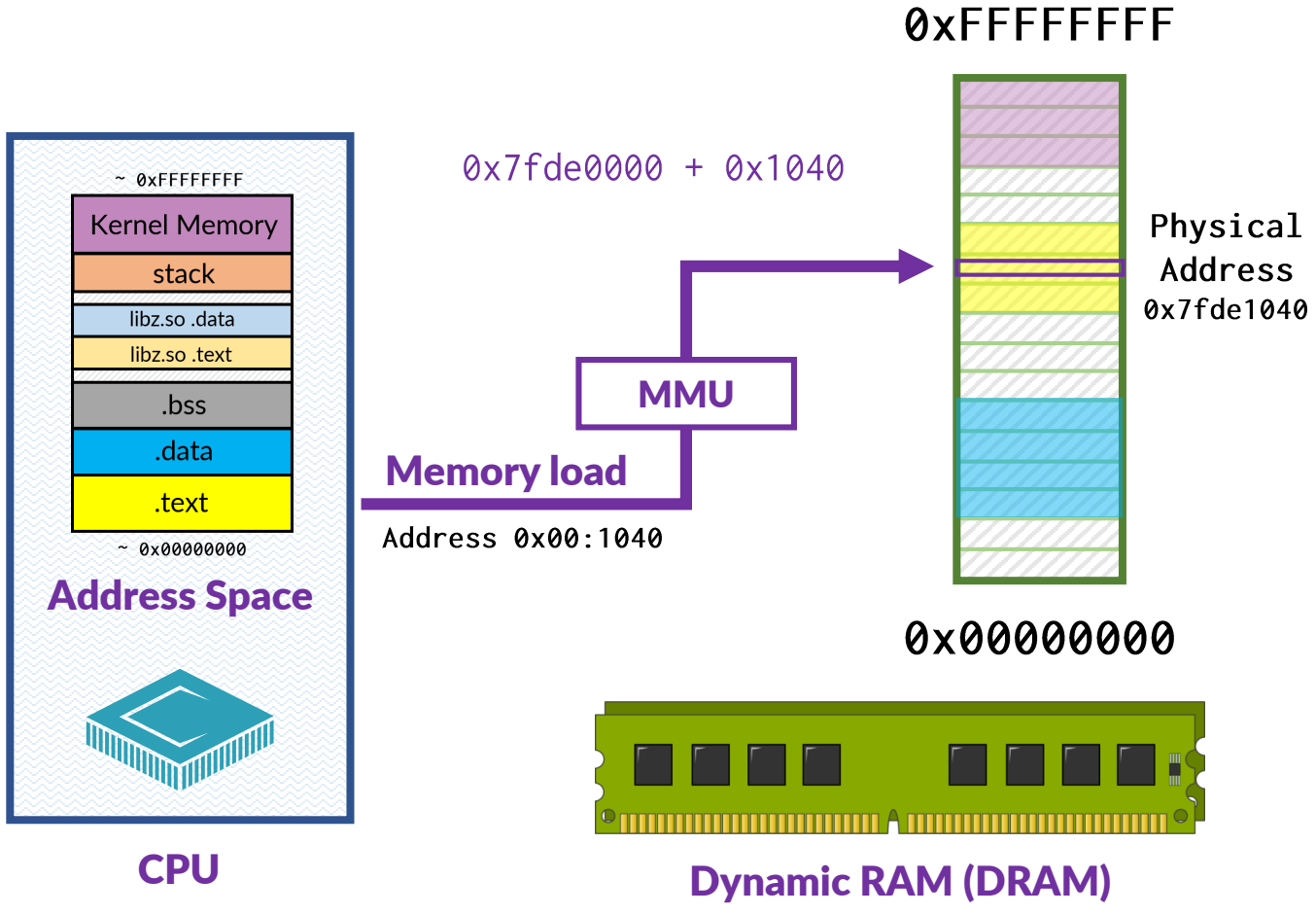


Addressing the Code Segment

- The MMU translates addresses by looking up the segment index and adding the base to the given offset.

Segment Table

Index	Physical Base	Size
00	0x7fde0000	0x3000
01	0x23fa0000	0x4000
02	NULL	0

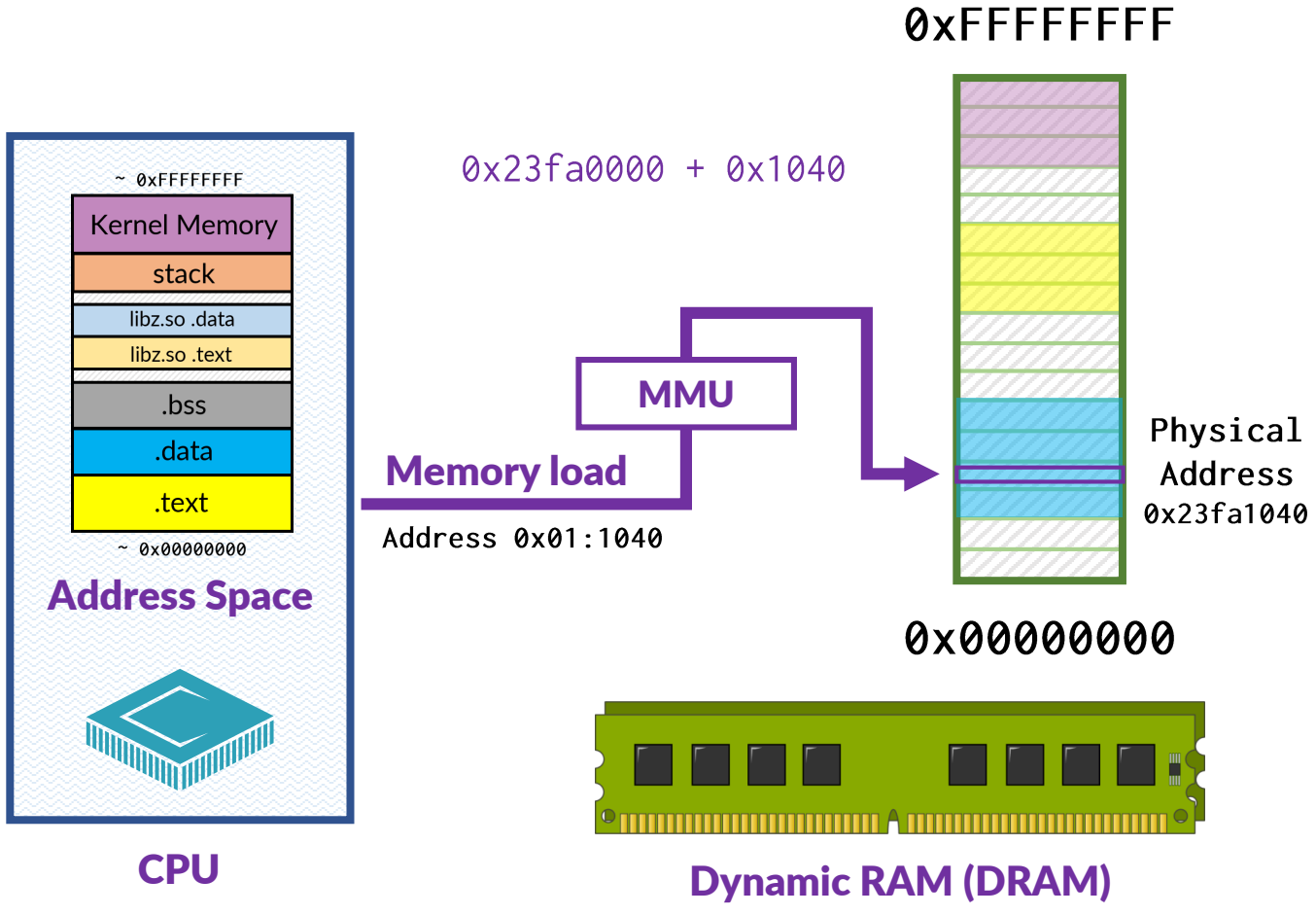


Addressing the Data Segment

- The same offset might refer to a different physical address depending on the index and the table.

Segment Table

Index	Physical Base	Size
00	0x7fde0000	0x3000
01	0x23fa0000	0x4000
02	NULL	0

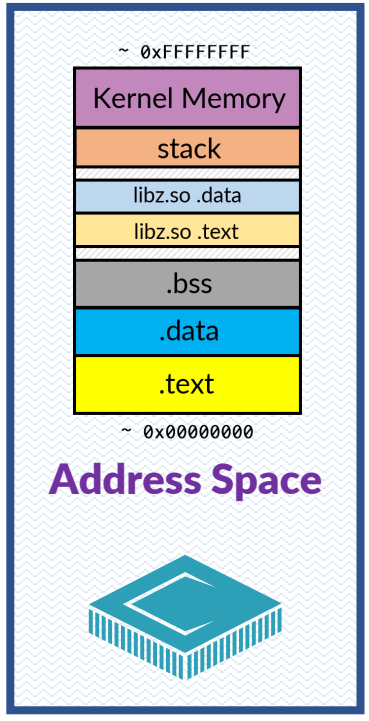


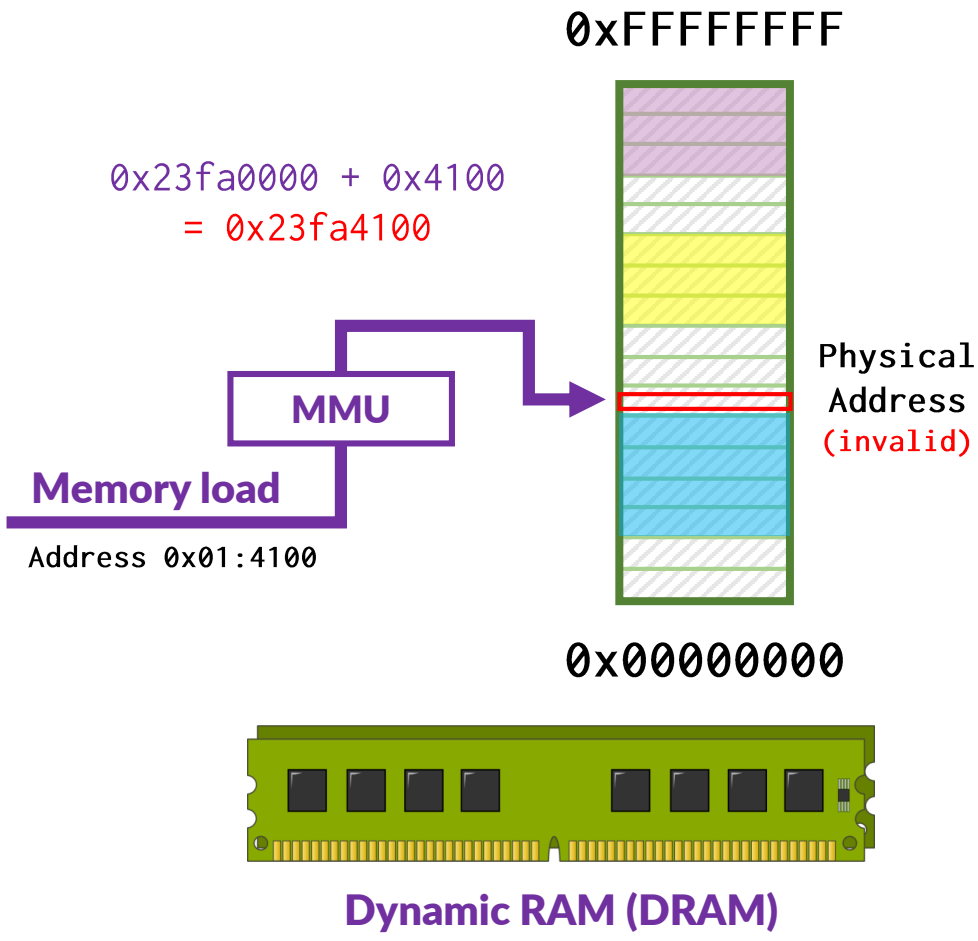
Addressing… nothing

- However, if the MMU cannot translate an address, it will fault.
- This is a **segmentation fault**.

Segment Table

Index	Physical Base	Size
00	0x7fde0000	0x3000
01	0x23fa0000	0x4000
02	NULL	0



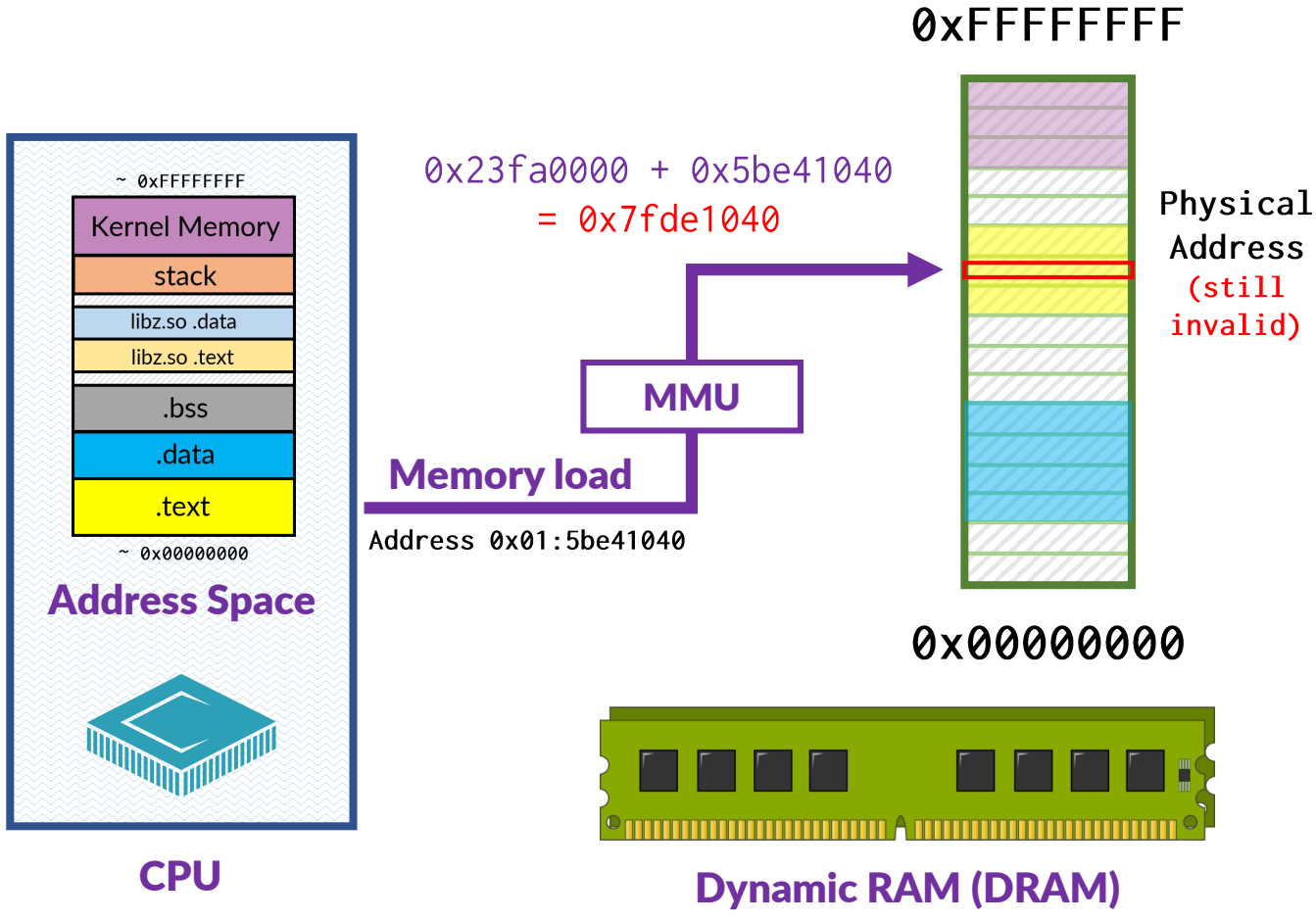


Addressing… out of range

- This is true even if the address calculation results in an address that is allocated to another segment.
- Fault: It goes beyond the size of the segment.

Segment Table

Index	Physical Base	Size
00	0x7fde0000	0x3000
01	0x23fa0000	0x4000
02	NULL	0



It's for your pwn protection

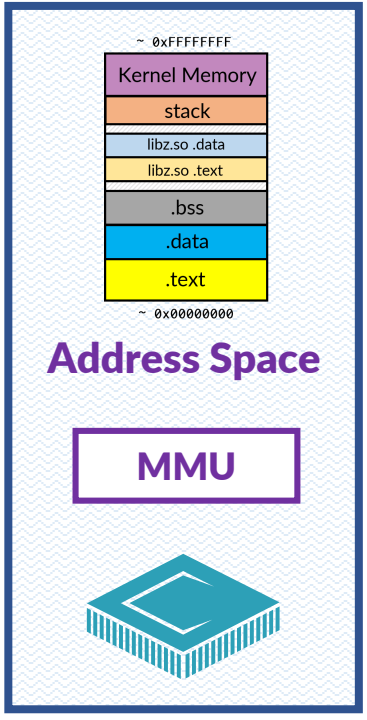
- **The lie only operates if processes cannot see each other.**
  - It's not just about address spaces not overlapping…
  - It is also for security purposes!
- **You don't want your login process to be snooped on by another.**
- **Yet, also, you don't want your own program to do ridiculous things it should not do!**
  - Should your program be able to write to constant values?
  - Should your program be able to execute instructions in the `.data` segment?
- **Virtual memory generally also has ways to arbitrate access.**

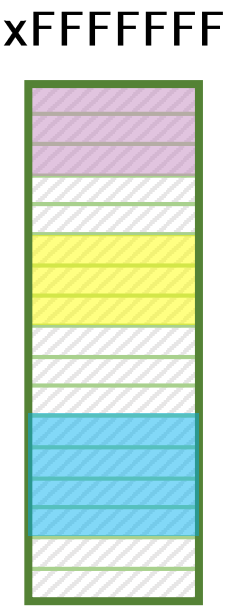
Amending to add access control

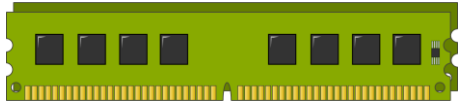
- The MMU can also arbitrate access to the segments by adding access control to the segment table.
- Here, a `1` in the table denotes the action is allowed.
  - `W`: Writes allowed.
  - `X`: Can be executed.

Segment Table

I	Physical Base	Size	W	X
00	0x7fde0000	0x3000	0	1
01	0x23fa0000	0x4000	1	0
02	NULL	0	0	0





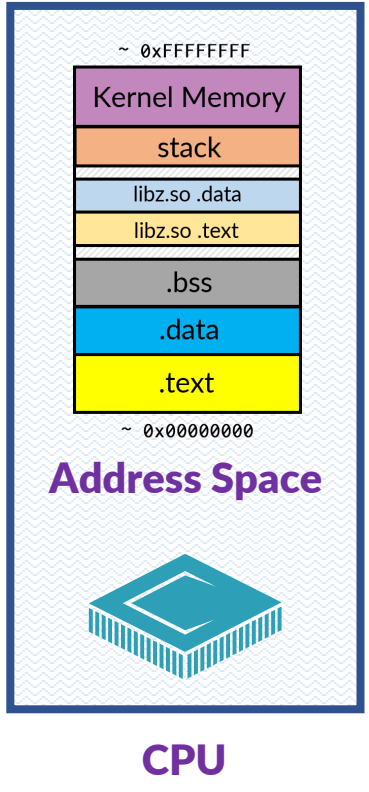


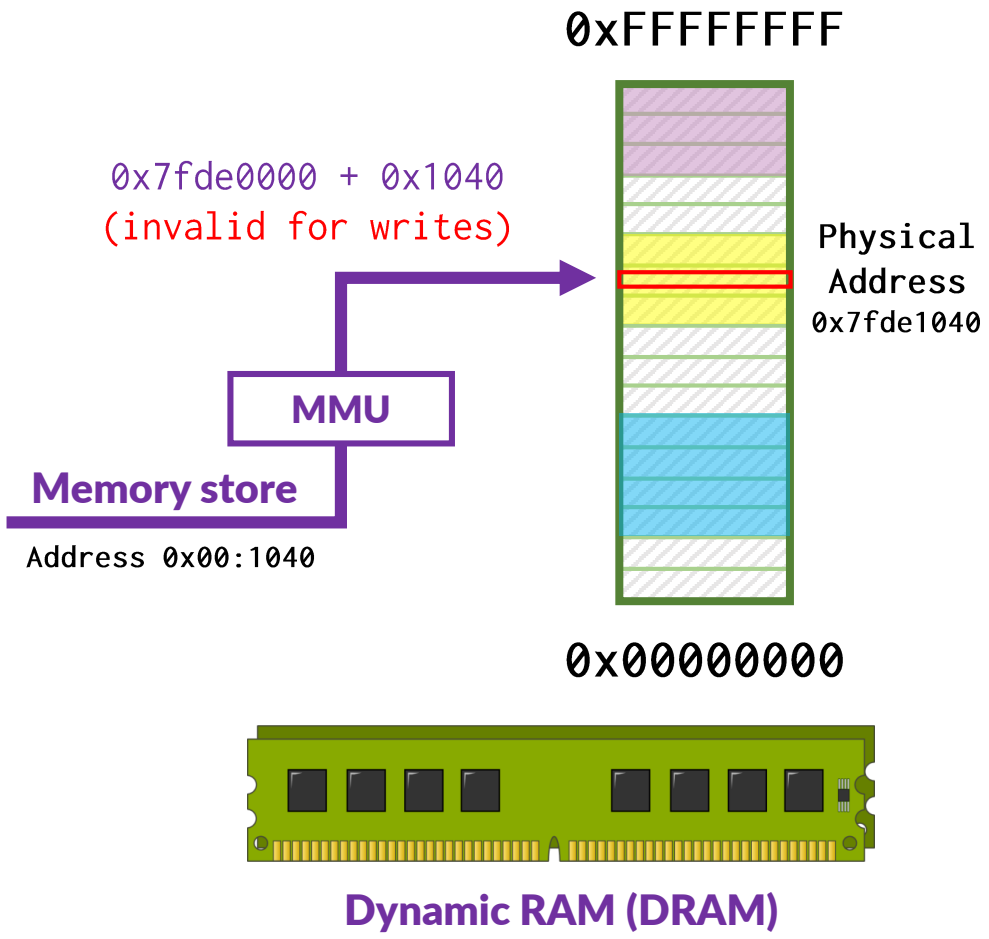
Writing to the Code Segment? NO!!

- When the MMU decides if an action is allowed, it looks at the access control bits in the table.

Segment Table

I	Physical Base	Size	W	X
00	0x7fde0000	0x3000	0	1
01	0x23fa0000	0x4000	1	0
02	NULL	0	0	0



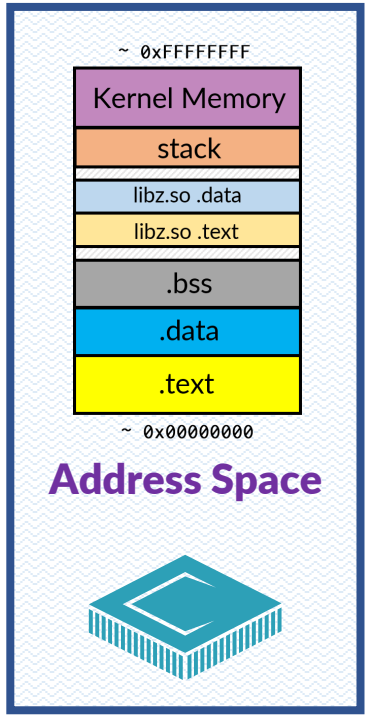


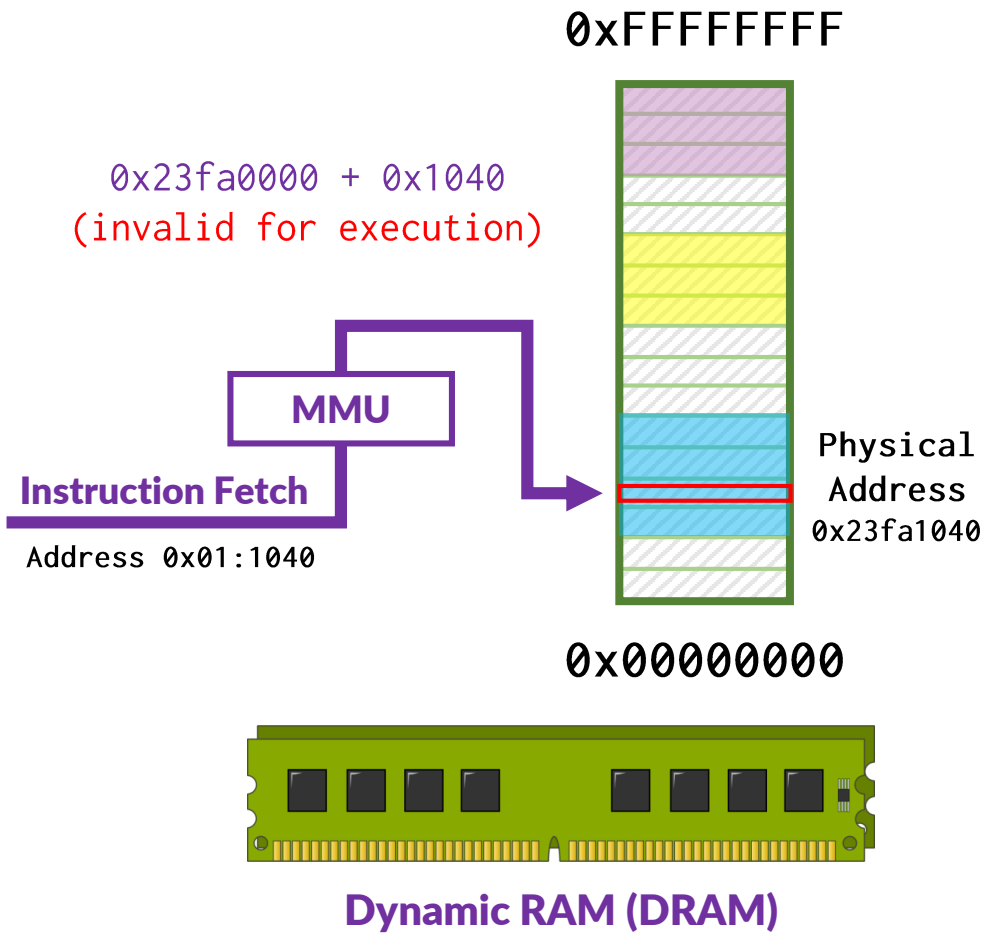
Executing the Data Segment? ABSOLUTELY NOT!!

- This feature can be used to effectively prevent many buffer overflow attacks.
- Here, you can't execute application data.

Segment Table

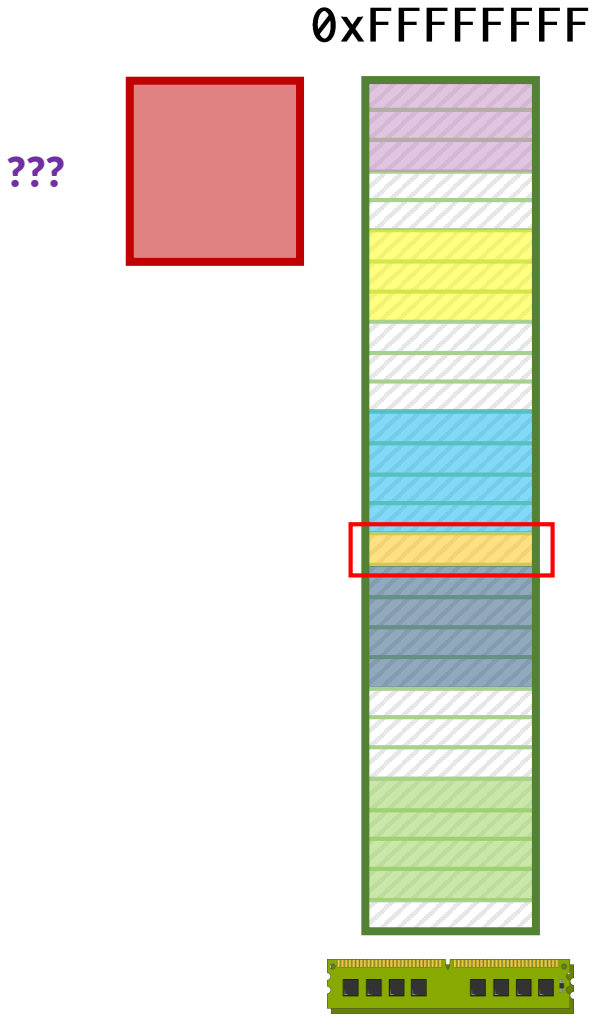
I	Physical Base	Size	W	X
00	0x7fde0000	0x3000	0	1
01	0x23fa0000	0x4000	1	0
02	NULL	0	0	0





A Problem Remains: Fragmentation

- In a purely segmented system, you can map regions of physical memory.
- However, the segments of virtual memory are continuous in physical memory and cannot overlap other physical regions on the system.
- We may run out of room as we run more processes…
- … and as processes finish, they may leave awkward gaps in memory. (external fragmentation)

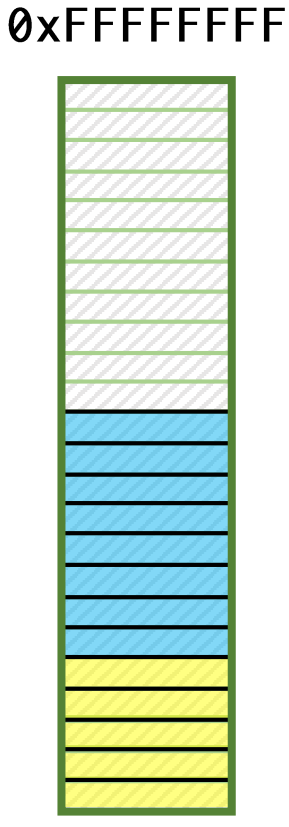


Paging

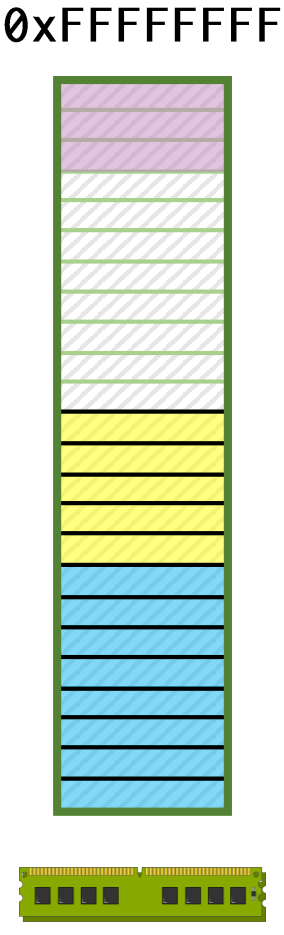
This won't give you paper cuts... I don't think.

Making things… smaller.

- So, segmentation helped us isolate processes by allowing a virtual address space where large spans of memory were mapped continuously to a physical address range.
- Since segments are large, managing that space is difficult.
- So why don't we make the segments… small? And use more of them?
- Welcome to the wonderful world of pages!

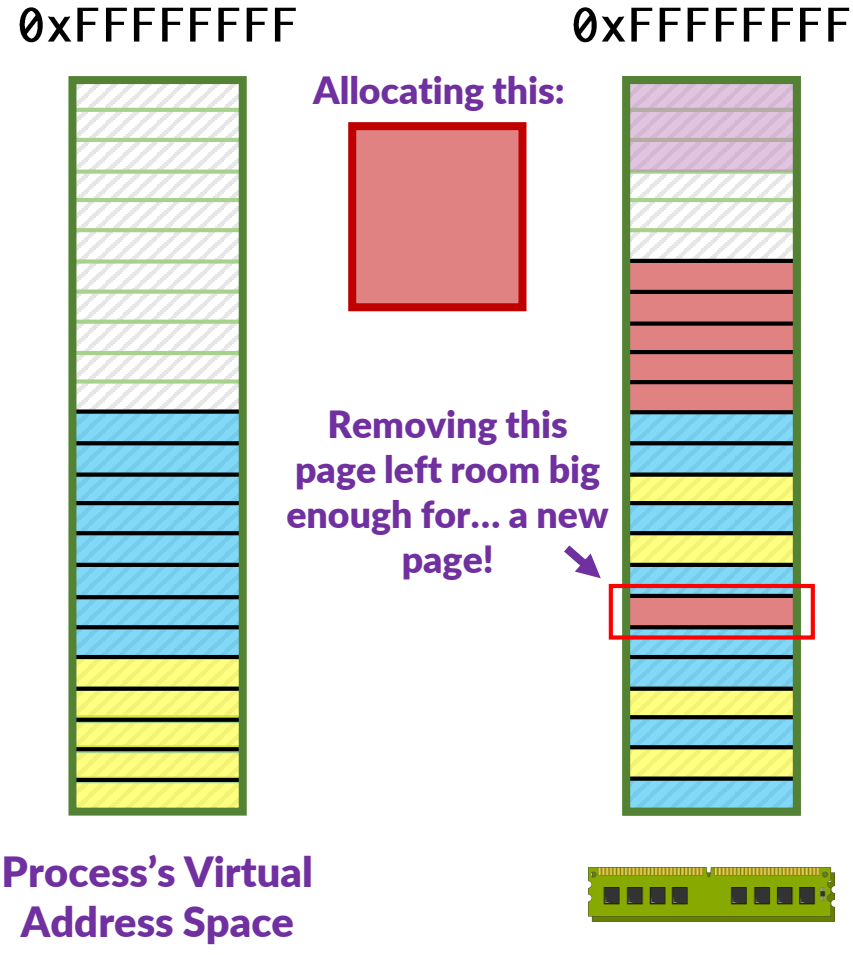


Process's Virtual Address Space



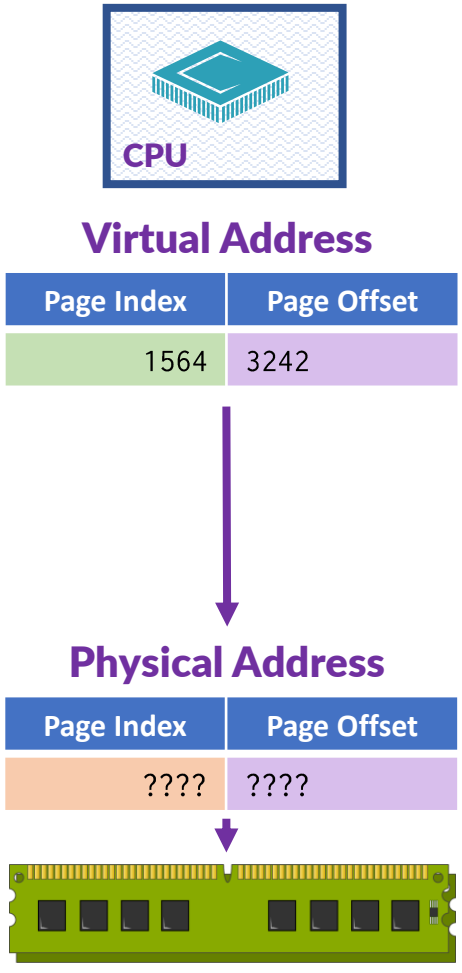
Paging Mr. Herman… Mr. P. W. Herman…

- Each segment is itself divided into smaller pieces called a **page**.
- This allows us to even interleave the different pages that make up a section of memory.
- Because of this interleaving and that every page is the same exact size, removing a page leaves room for exactly one page… no fragmentation.
  - At the cost of over-allocating, if we need less space than a single page.



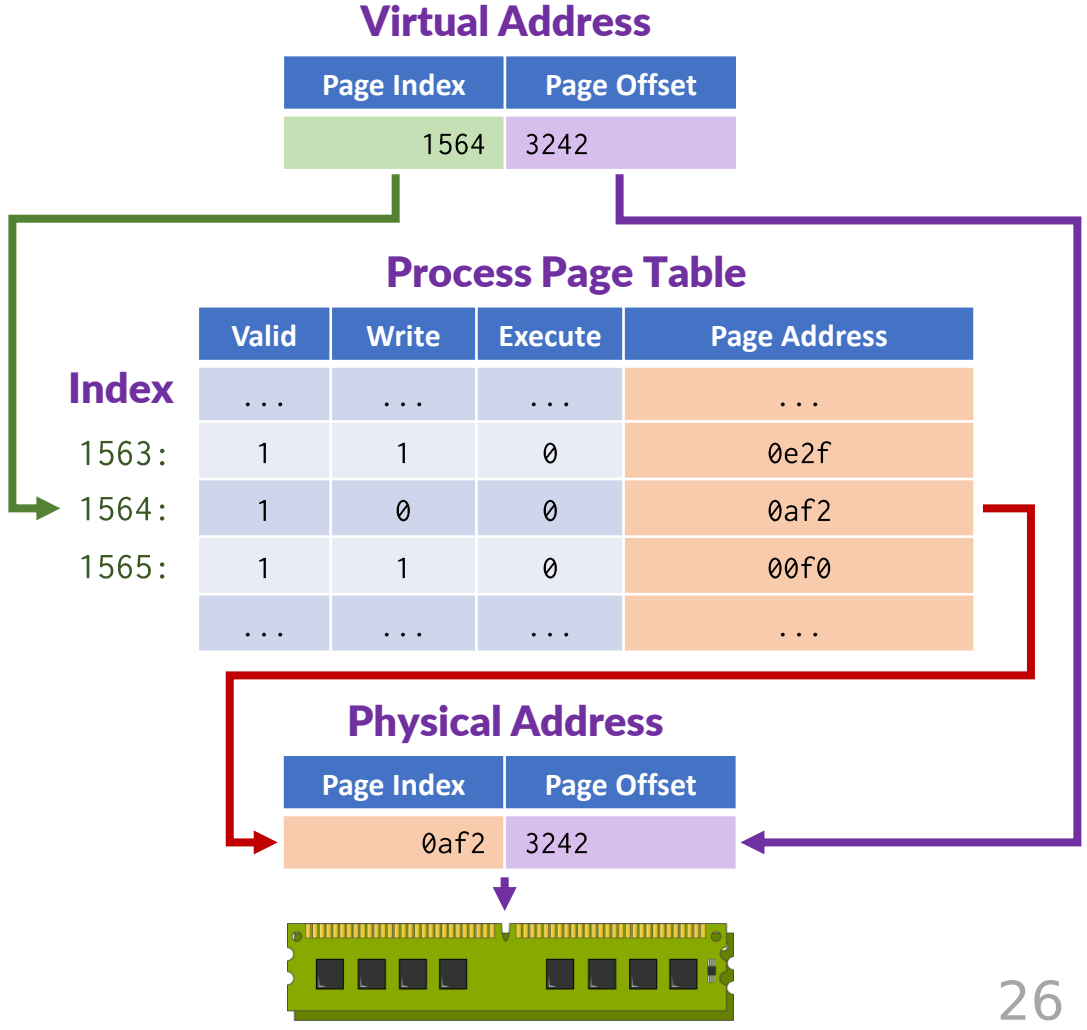
Page Tables

- There are many strategies for maintaining the metadata that maps virtual addresses to physical addresses.
- The first we will look at is the simple **page table**.
- In this strategy, we will maintain a data structure that maps virtual addresses to physical addresses.



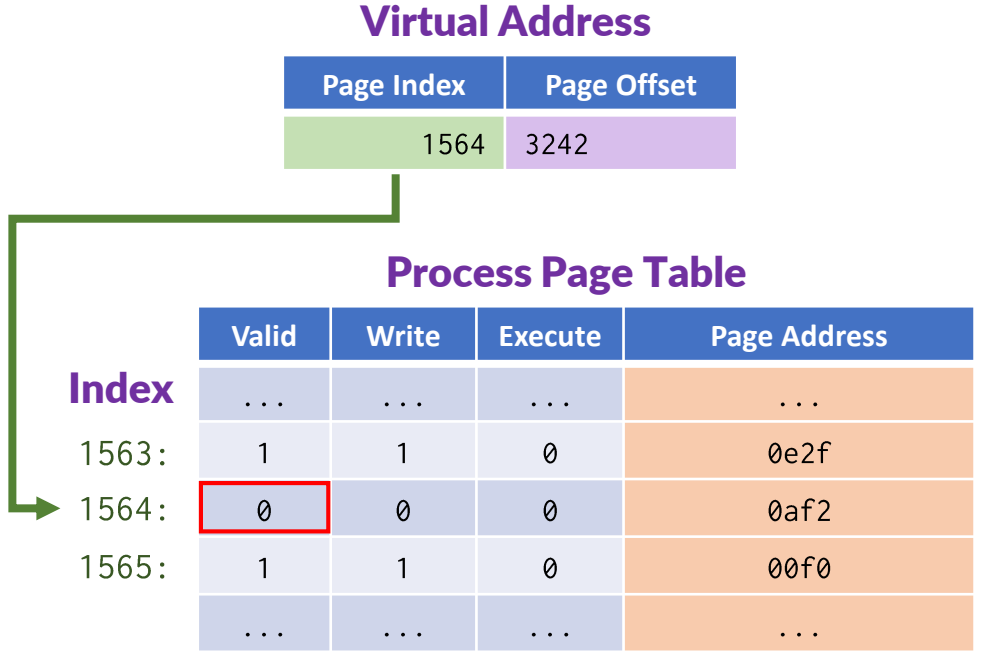
Address Fields

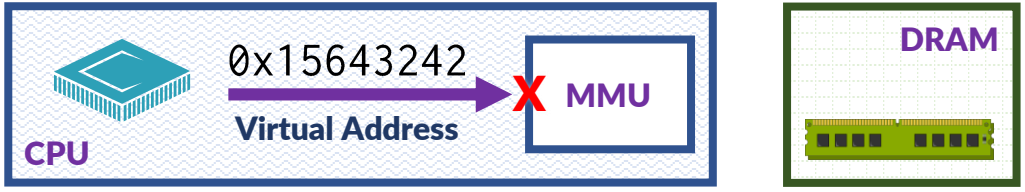
- First, you need to set a static page size.
  - Every page is the same size.
- Part of the virtual address is the offset, which is retained when the MMU translates the physical address.
  - This is determined by the page size.
- The remainder is used to determine the entry in the table.



It's not your fault…

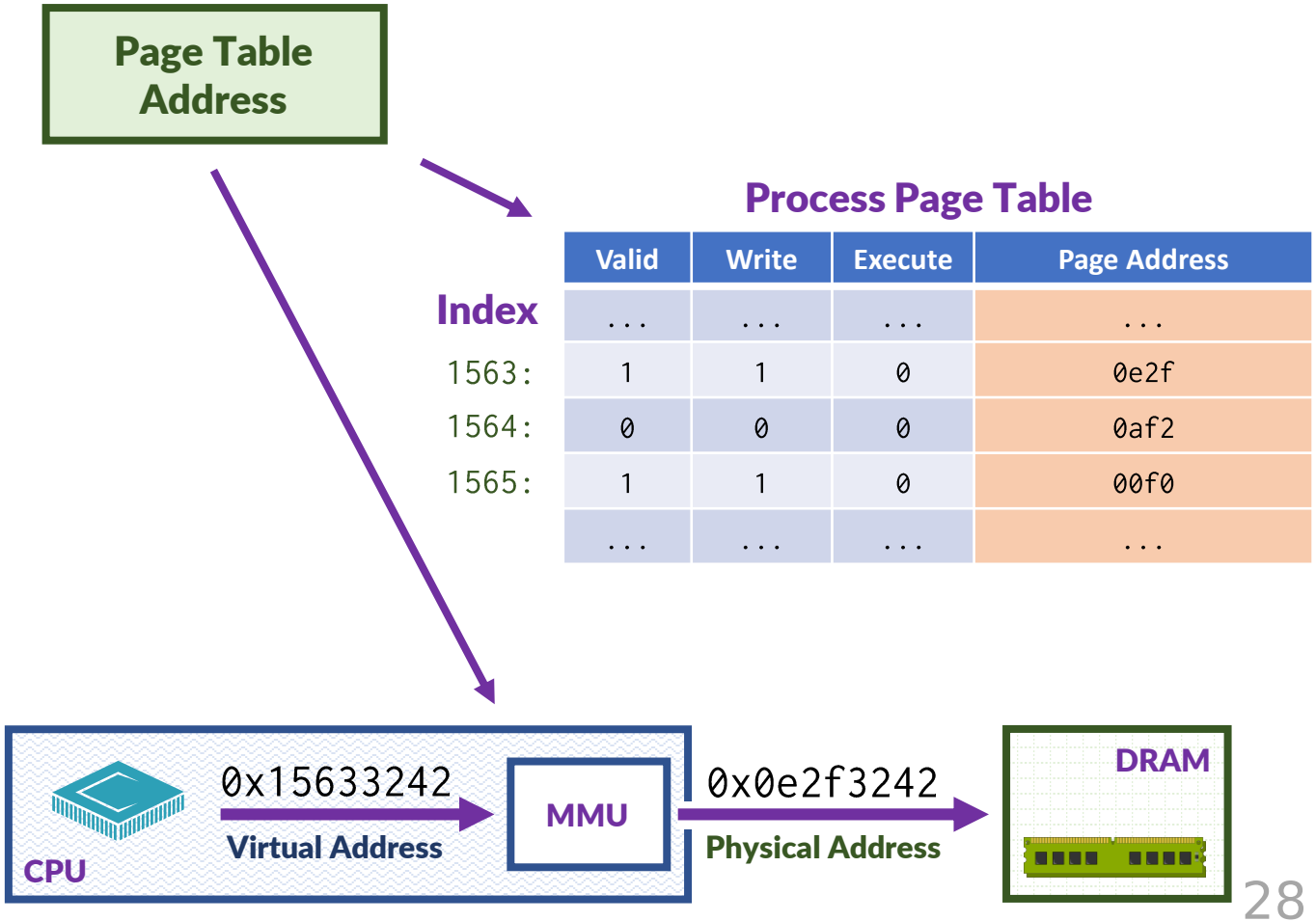
- If there is no entry for the given page or the entry isn't valid…
  - Or if an operation is not allowed.
- This signals a **page fault**.
  - Similar to a segmentation fault.
  - In fact, many OSes retain that term to this day, even when it is a page fault, technically.
- This is a generic error that is triggered by the MMU on such invalid accesses.





Process Isolation

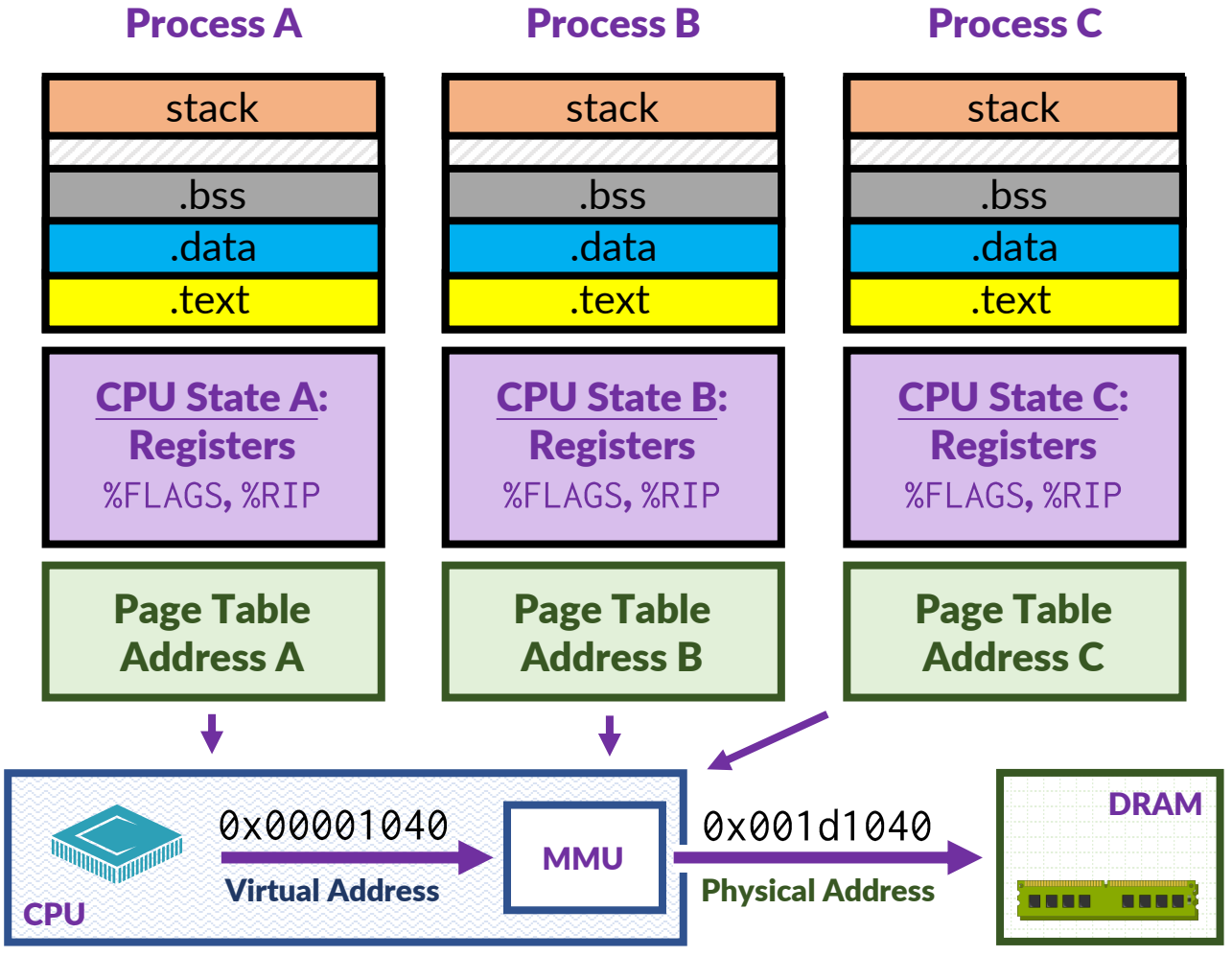
- To give each process its own virtual address space, each process gets its own page table.
- The CPU keeps track of which page table is active.



Context Switching: Getting to the root of it.

- When an Operating System goes from one process to another, it performs a **context switch**.

1. Store registers (including stack pointer and program counter) to memory.
2. Determine next process to run.
3. Load those registers from memory. Switch address space.
4. Jump to old program counter. Go!



Addressing the granularity issue

- The table size has to do with how big you make each page.
  - The bigger the page, the less entries you'll need for your process.
  - However, the more internal fragmentation if you do not need some of that space!

- For a page size of K
  - Page offset will have $\log_2 K$ bits.
  - Page index will be the remaining bits.

- For **32-bit** address spaces:
  - Assuming table entries are also 32-bits.

Virtual Address

Page Index	Page Offset
0001010101100100	0011001001000010

$K = 2^{16}KiB = 64KiB$

Mapping 2MiB takes 32 pages.

Page table size: $2^{16} * 4B = 256KiB$

Page Index	Page Offset
00010101011001000011	001001000010

$K = 2^{12}KiB = 4KiB$

Mapping 2MiB takes 512 pages.

Page table size: $2^{20} * 4B = 4MiB$

Page Index	Page Offset
00010101011001000110010	01000010

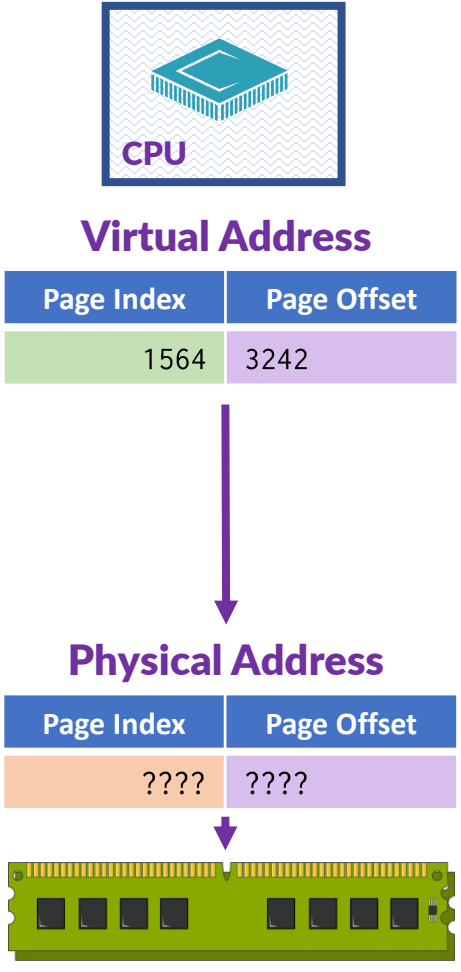
$K = 2^{8}B = 256B$

Mapping 2MiB takes 8192 pages.

Page table size: $2^{24} * 4B = 64MiB$

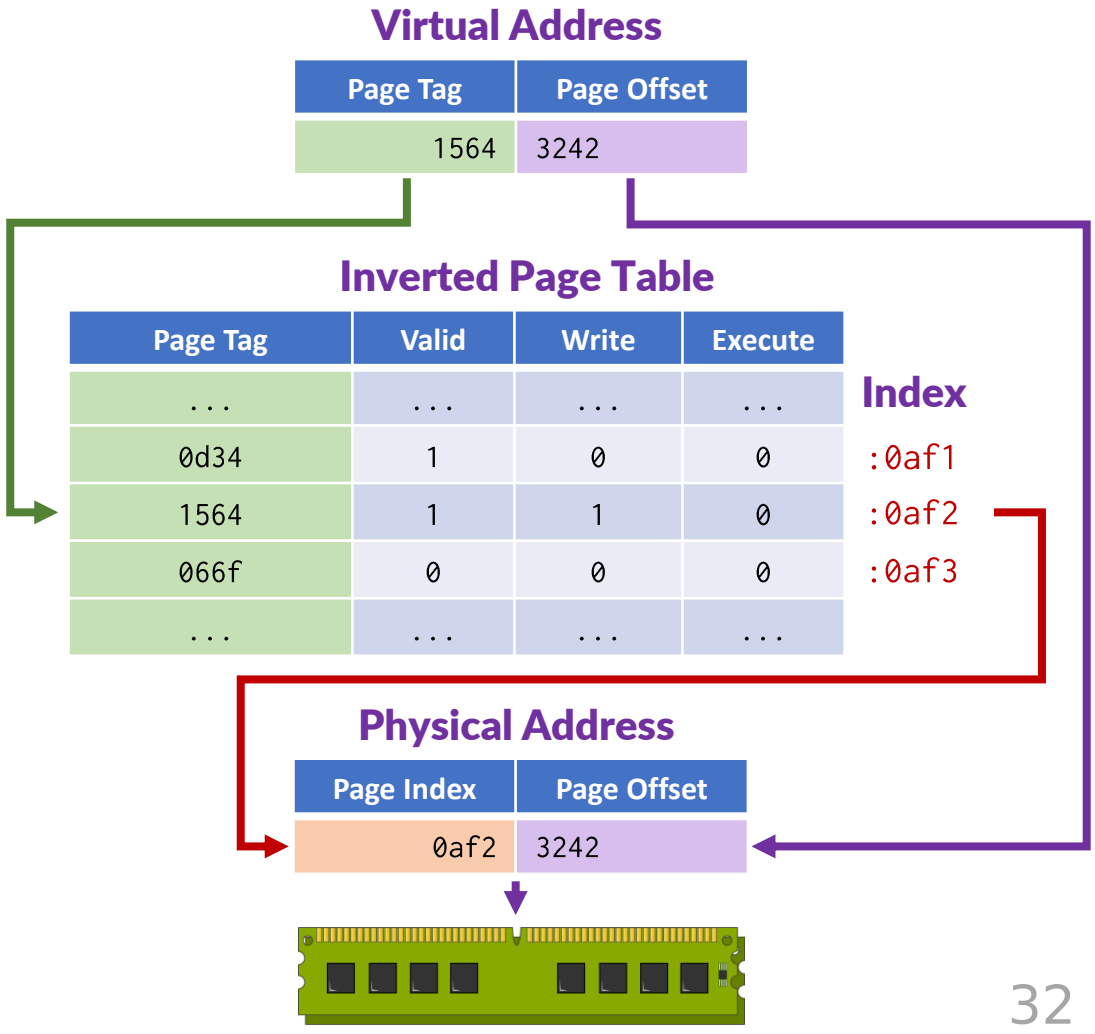
Inverted Page Tables

- There are many strategies for maintaining the metadata that maps virtual addresses to physical addresses.
- Now we will look at the **inverted page table**.
- In this strategy, we switch things around: we have just one table for the whole system and an entry for every possible physical page.



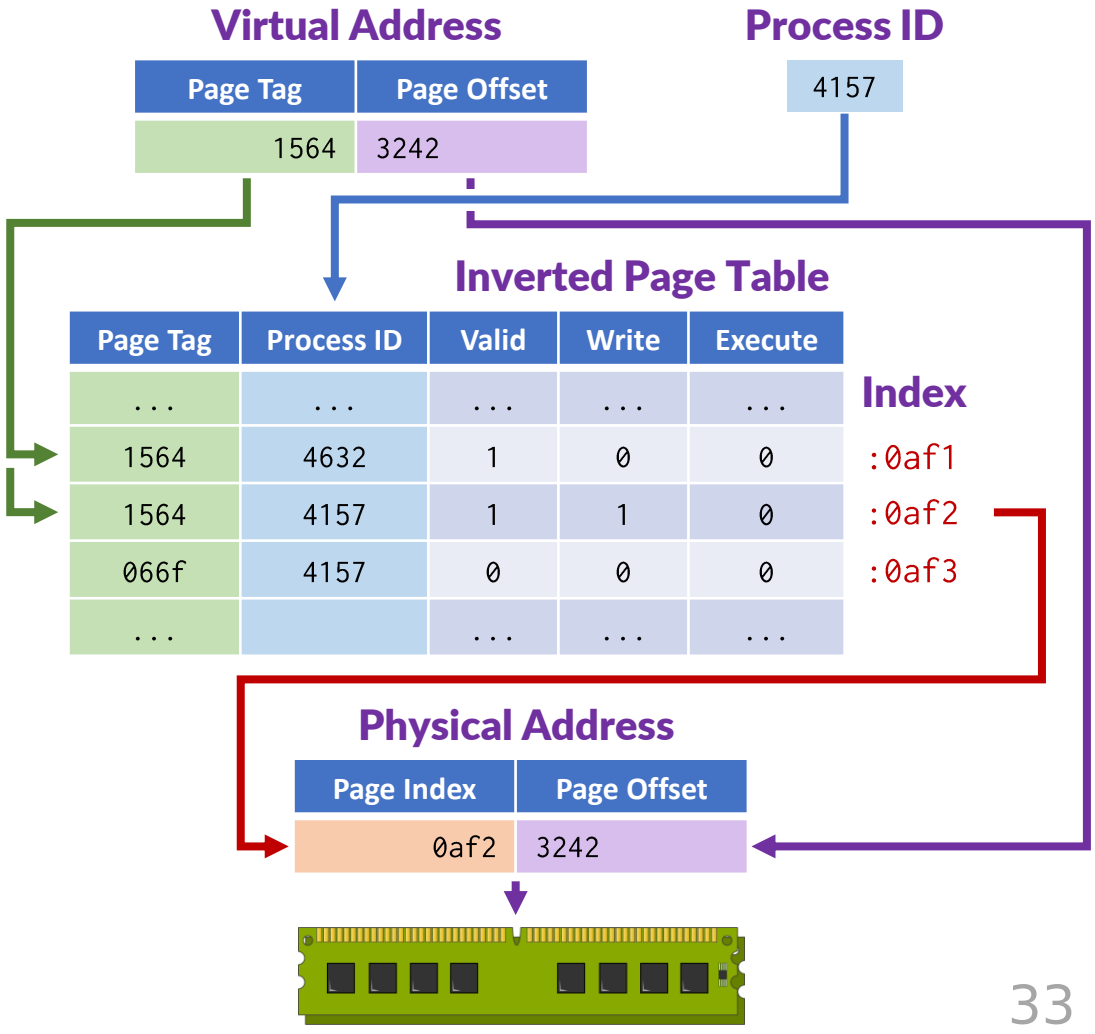
Address Fields

- In this case, you have a single table for the entire system.
- When translating, you scan the table to find an entry that contains the page index.
  - This may be intensive!
- When you do, and it is valid, make a note of the index of the entry. That is the physical page index.



Process Isolation

- Many processes exist, and each may use the same virtual address.
  - And expect a different physical page!
- Since there is only one table on the entire system, we have to disambiguate.
- Therefore, we also tag by the process identifier.



What's the size??

- One nice feature of an inverted page table is the size is bound.
- Since an inverted page table has an entry for every possible physical page…
  - You can simply allocate the table of a fixed size big enough to represent all of physical memory.
- The size of the table is the product of the entry size and the number of physical pages.

Inverted Page Table

Page Tag	Process ID	Valid	Write	Execute
...	...	...	...	...
1564	4632	1	0	0
1564	4157	1	1	0
066f	4157	0	0	0
...		...	...	...

20 bits 9 bits

If the page size (K) is **4KiB** and our system has **16GiB** of RAM, how big is the inverted page table?

$$16GiB \; / \; 4KiB \; = \frac{2^{34}}{2^{12}} = \; 2^{22} \; pages$$

$$2^{22} \; pages \; * 32 \; bits = \; 2^{22} \; pages \; * 4B$$

$$2^{24}B = 2^4 2^{20}B = 16MiB$$

Trade-offs. Trade-offs everywhere!

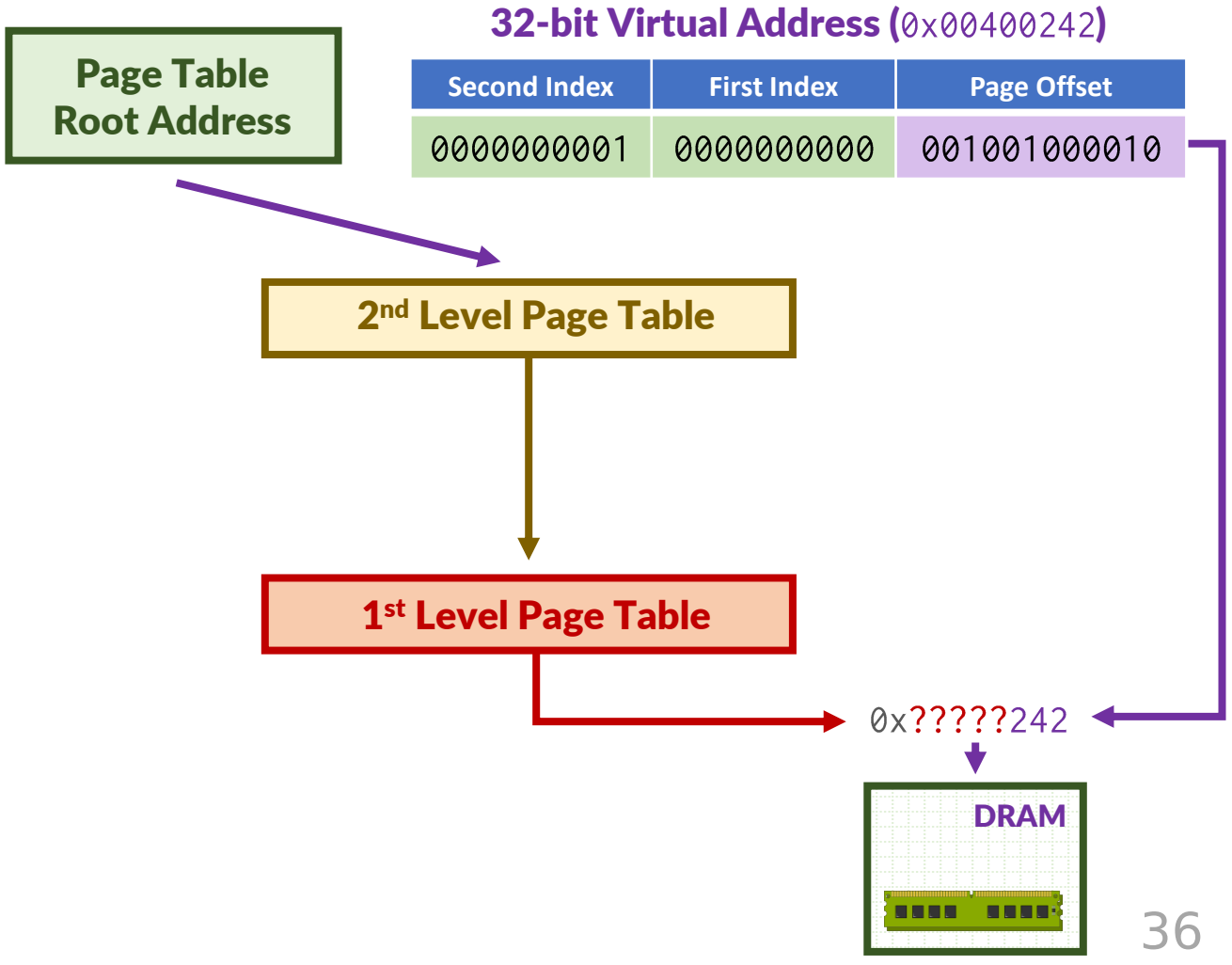
- **What is best? … Who even knows.**

- **Inverted page tables are very space efficient since entries are ordered by physical page.**
  - However, translations mean scanning the table for entries… a time-consuming task. $O(n)$ (Can implement with a hashing function, see your OS course.)
  - Normal page tables are a constant time lookup, $O(1)$

- **Since they are indexed by virtual address, normal page tables require ordered virtual memory to be space efficient.**
  - Gaps in virtual memory mean lots of page table entries going unused.
  - Perhaps we can solve this problem…

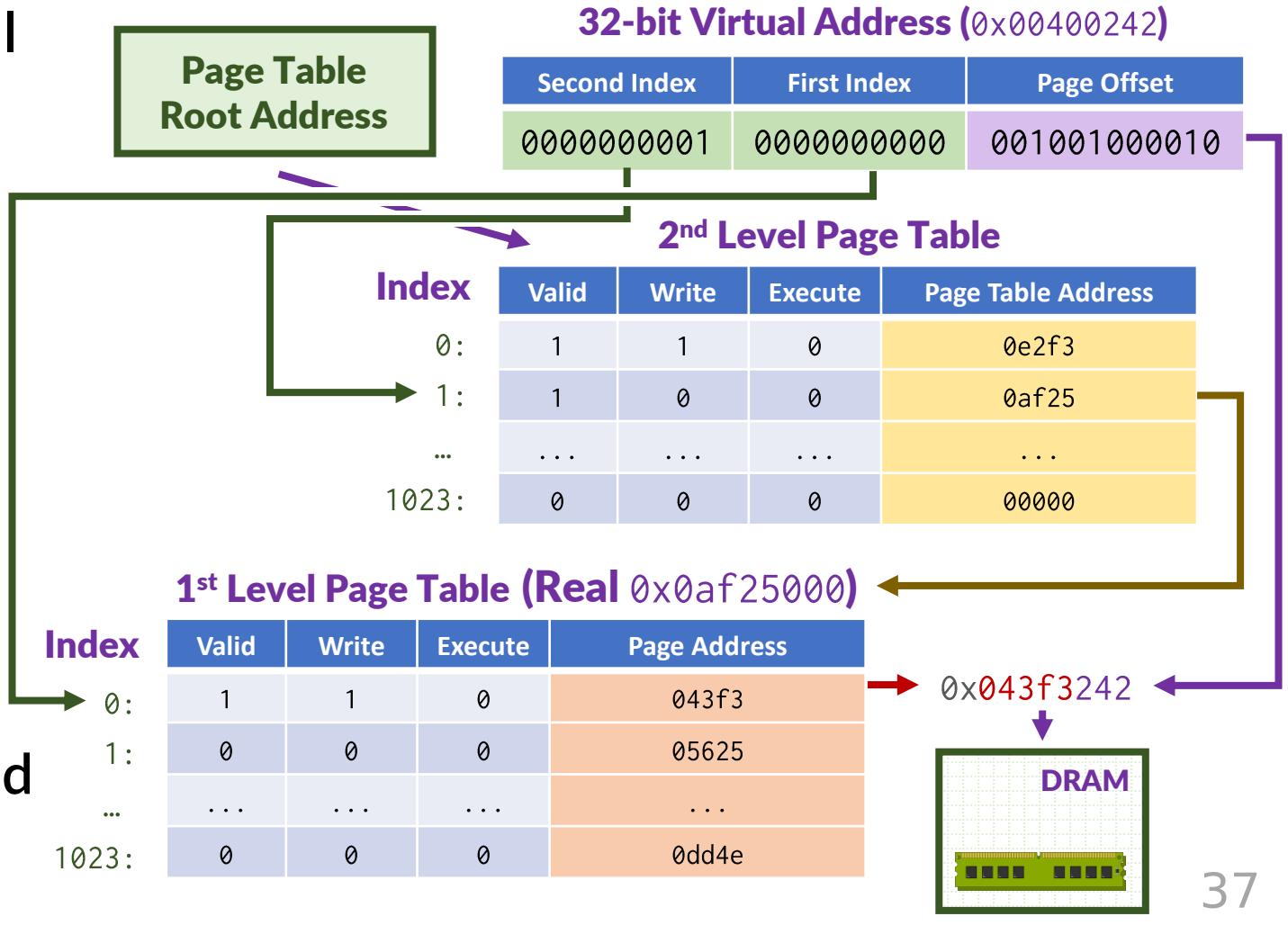
Multi-level Page Tables (Not a pyramid scheme)

- Perhaps we can allow gaps in virtual memory if we use MORE INDIRECTION!
- The use of multiple levels of indirection gives a lot of flexibility in defining the virtual address space.
- Each page table is the size of a page. (4KiB)



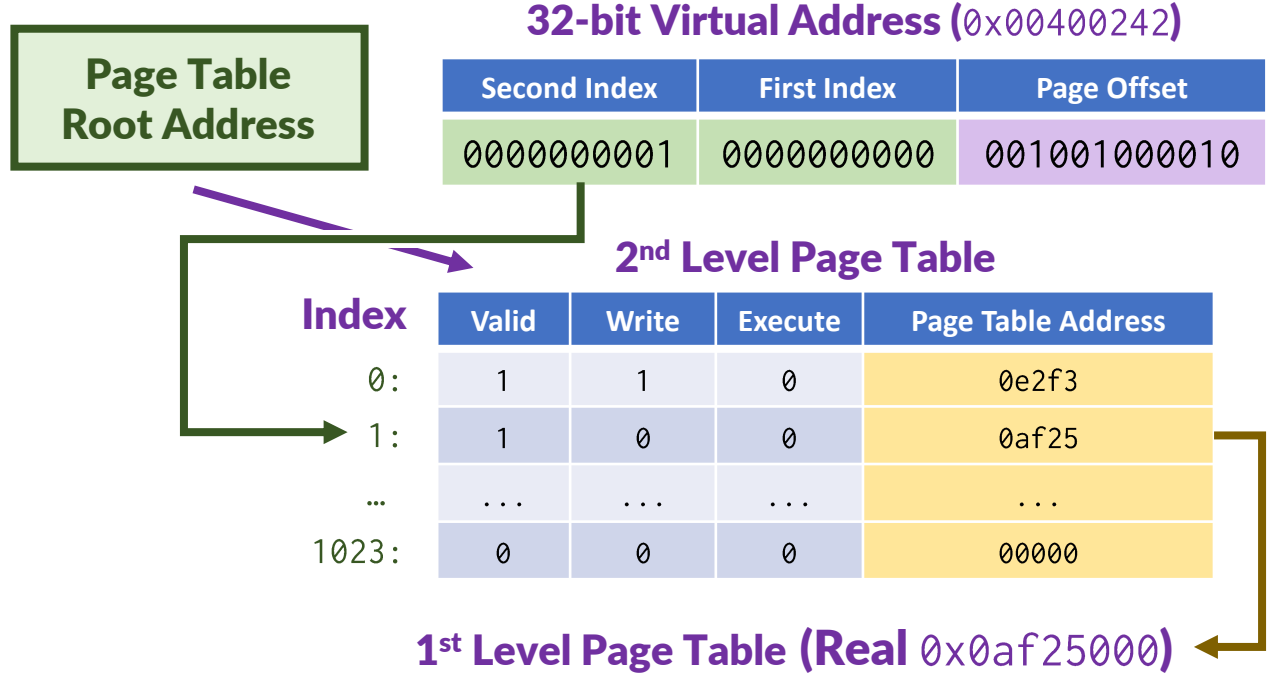
Indirection times two

- We split up the virtual address into further index fields.
- The top-level index yields the real physical address of the page containing the next page table.
- This table is used to determine the referred physical page.



Home, home on the [memory] range

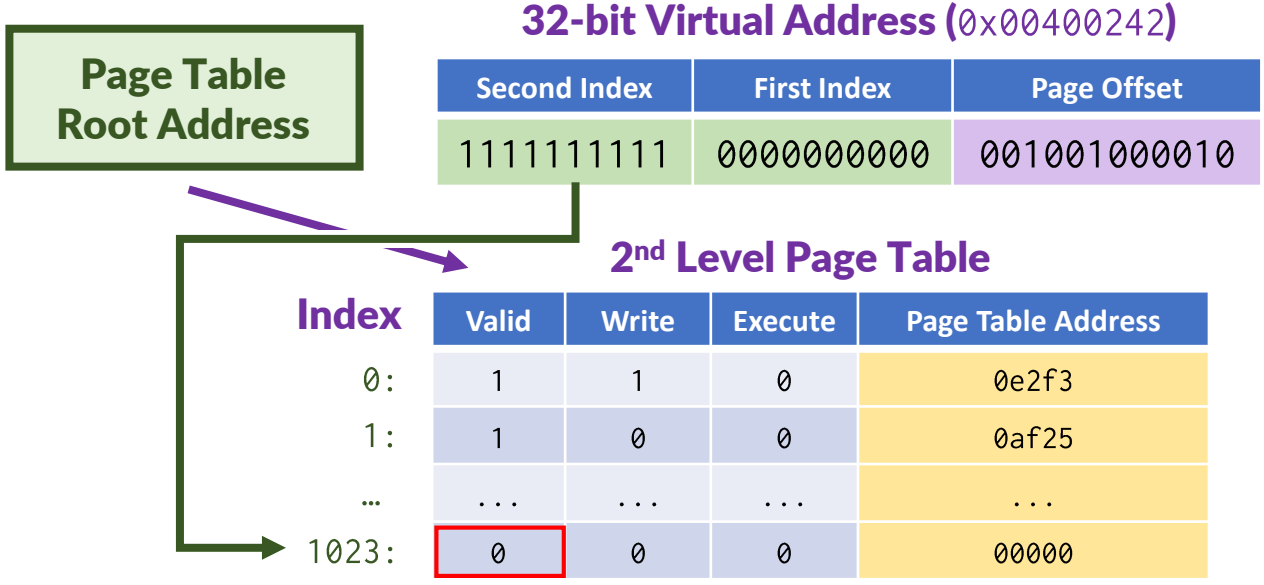
- Each entry in the top-level page table represents an entire range of memory.
- Here, the 2nd level index is `0x1`. This represents all virtual memory addresses with the most significant binary digits: `0000000001`



Maps 4KiB virtual page starting at 0x00400000
Maps 4KiB virtual page starting at 0x00401000
...
Maps 4KiB virtual page starting at 0x007ff000

It's a sparse world, after all

- By marking entries invalid in the top-level page table, this invalidates the entire memory range.
- Attempting to access such a virtual address would immediately page fault.



No 1st level table.

Therefore: all virtual addresses between `0xffc00000` and `0xffffffff` are not mapped (and are not referenceable.)

A got a sparsity jacket, but it was just the sleeves.

- Given a 32-bit virtual address.
  - And multi-level paging with two levels, each index 10 bits.

- What is the page size?
  - $32 - 10 - 10 = 12\ bits$ for offset
  - $2^{12}B = 4KiB$ (4 Kibibytes)

- Given the root page table here, and assuming unknown entries are invalid, what virtual address ranges are potentially used?
  - Let's find out…

32-bit Virtual Address

Second Index	First Index	Page Offset
10 bits	10 bits	??????

2nd Level Page Table

Index	Valid	Write	Execute	Page Table Address
0:	1	0	1	0e2f3
1:	0	0	0	00000
…	...	...	...	...
255:	0	0	0	00000
256:	1	1	0	0b3d8
257:	0	0	0	00000
…	...	...	...	...
1022:	0	0	0	00000
1023:	1	0	0	0af3d

Continuing: Filling in the blanks

- Given the root page table here, and assuming unknown entries are invalid, what virtual address ranges are potentially used?

32-bit Virtual Address

Second Index	First Index	Page Offset
10 bits	10 bits	12 bits

0000000000 00000000000000000000 to
0000000000 11111111111111111111
Maps virtual pages from `0x00000000` **to** `0x003fffff` →

0100000000 00000000000000000000 to
0100000000 11111111111111111111
Maps virtual pages from `0x40000000` **to** `0x403fffff` →

1111111111 00000000000000000000 to
1111111111 11111111111111111111
Maps virtual pages from `0xffc00000` **to** `0xffffffff` →

2nd Level Page Table

Index	Valid	Write	Execute	Page Table Address
0:	1	0	1	0e2f3
1:	0	0	0	00000
…	...	...	...	...
255:	0	0	0	00000
256:	1	1	0	0b3d8
257:	0	0	0	00000
…	...	...	...	...
1022:	0	0	0	00000
1023:	1	0	0	0af3d

Best of both worlds.

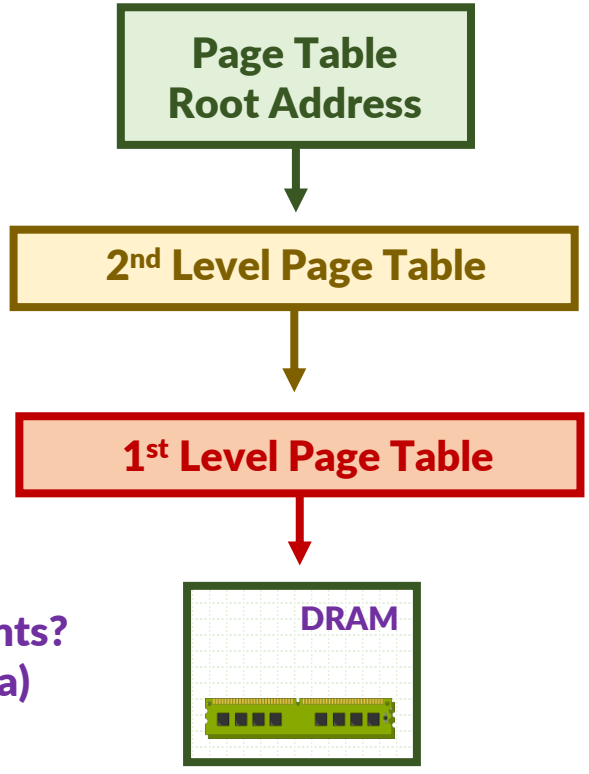
- With multi-level page tables, we can represent large ranges of memory with gaps, much like segments!
  - All the while, we can satisfy each individual page in this "segment" by interleaving them throughout physical ram. (flexibility, no external fragmentation)
- Modern architectures often use multi-level page tables.
  - x86-64 uses a 4 level page table! 😱

Maps virtual pages from `0x00000000` **to** `0x003fffff`
Maps virtual pages from `0x40000000` **to** `0x403fffff`
Maps virtual pages from `0xffc00000` **to** `0xffffffff`

Segments? (kinda)

32-bit Virtual Address

Second Index	First Index	Page Offset
10 bits	10 bits	12 bits



Summary

So, we have many complex processes running at the same time.

- How to present a consistent address space?
  - Indirection using segments or page tables.
  - We translate virtual addresses to physical addresses.
- How to prevent other processes from interfering?
  - We can mark segments or individual pages with access controls. (Read-only, non-execute, etc.)
- How to prevent address space fragmentation?
  - We give each process its own address space.
  - When we context switch, we switch address spaces.