CS3.301 Operating Systems and Networks

Memory Virtualization - Paging: Smaller Page Tables

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Software Engineering Research Centre

Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems: In three easy pieces, by Remzi et al.
- Lectures on Operating Systems by Youjip Won, Hanyang University

Illustrative Example: Array Access

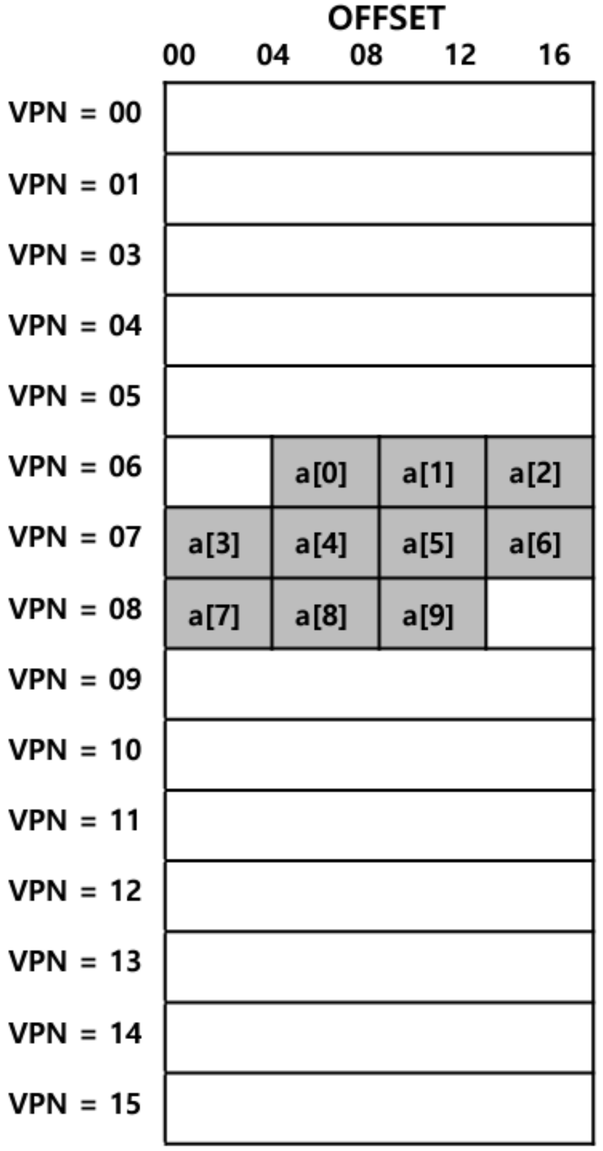
- Assume an array of 10 4-byte integers starting at VA 100. Assume an 8-bit VA space with 16 byte pages
  - This implies 4 bits for offset and 4 bits for VPN
  - Each page is 2^4 = 16 byte size
- How shall the virtual address space be organized?

Array access program in C

```c
int main(int argc, char* argv[])
{
  int sum = 0;
  for (int i = 0; i < 10; i++)
  {
    sum = sum + a[i];
  }
  return 0;
}
```

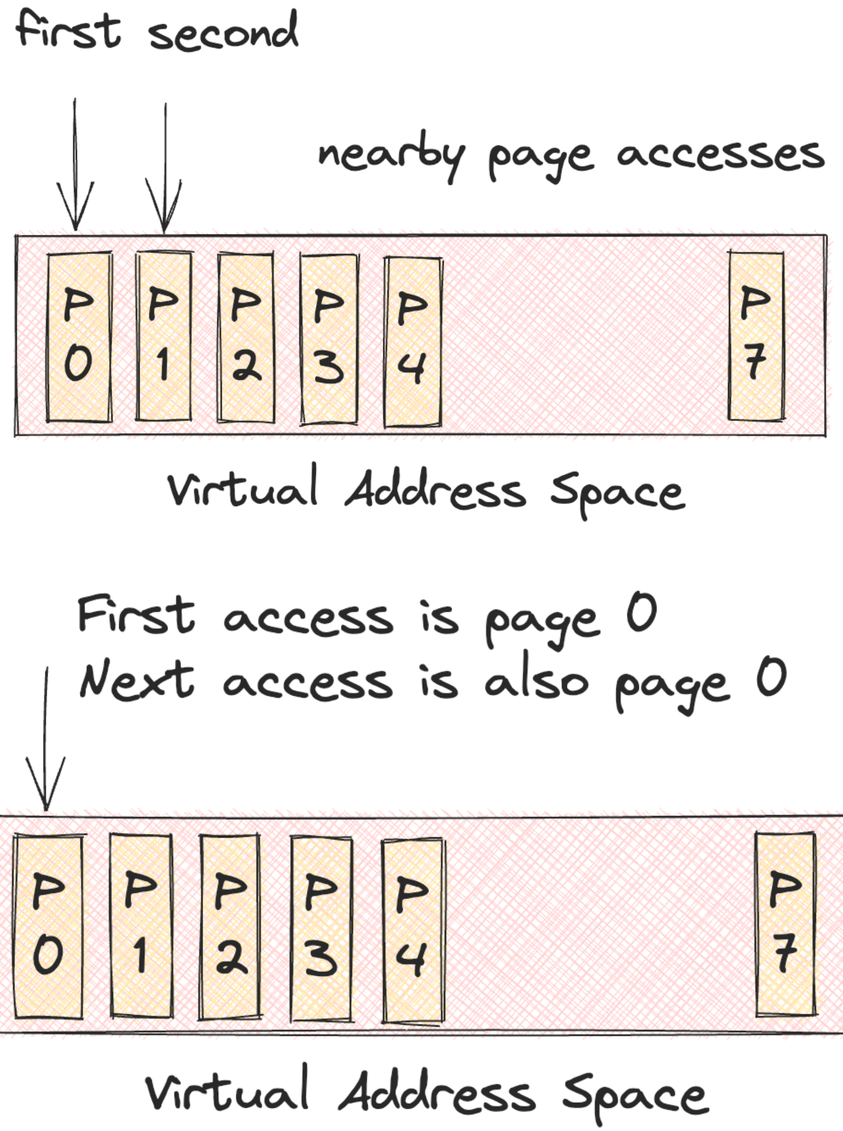
Illustrative Example: Array Access

- Access to a[0] with VA 100 - How?
  - It will be a TLB miss!
- Access to a[1] will be a TLB hit!
- Access to a[3] will be a TLB miss!
- Access to a[4] to a[6] will be a hit!
- Access to a[7] will be a miss then a[8] and a[9] will be hit
- miss, hit, hit, miss, hit, hit, hit, miss, hit, hit
- Hit rate: 70% - Not bad!



Locality as a factor

- **Spatial Locality**
  - If a program accesses memory at x, it will likely soon access memory near x.
  - What if page size was 32 bytes in previous example?
- **Temporal Locality**
  - An instruction or data that has been recently accessed will likely be re-accessed soon in the future



Handling TLB Miss?

Hardware Handler

- Hardware handles the TLB miss entirely on CISC
  - Older systems based on Intel x86
  - Hardware has to know where the page tables are stored
  - The base address is stored in Page Table Base Register
  - Hardware would walk the entire page table in parallel to find the correct PTE and **extract** the translation, **update** and **retry** instruction

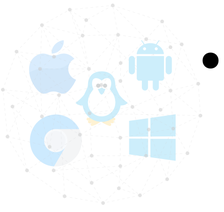
Handling TLB Miss?

Software/OS Handler

- Software handles the TLB miss entirely on RISC
  - Hardware simply raises an exception on a miss (trap handler)
  - Trap handler for handling TLB miss is invoked
  - The code will look up the page table for translation, updates TLB with privileged instruction and return from trap
  - At this point hardware retries the instruction
    - **There is a subtlety to context switch here - Why?**

Some Caution!!

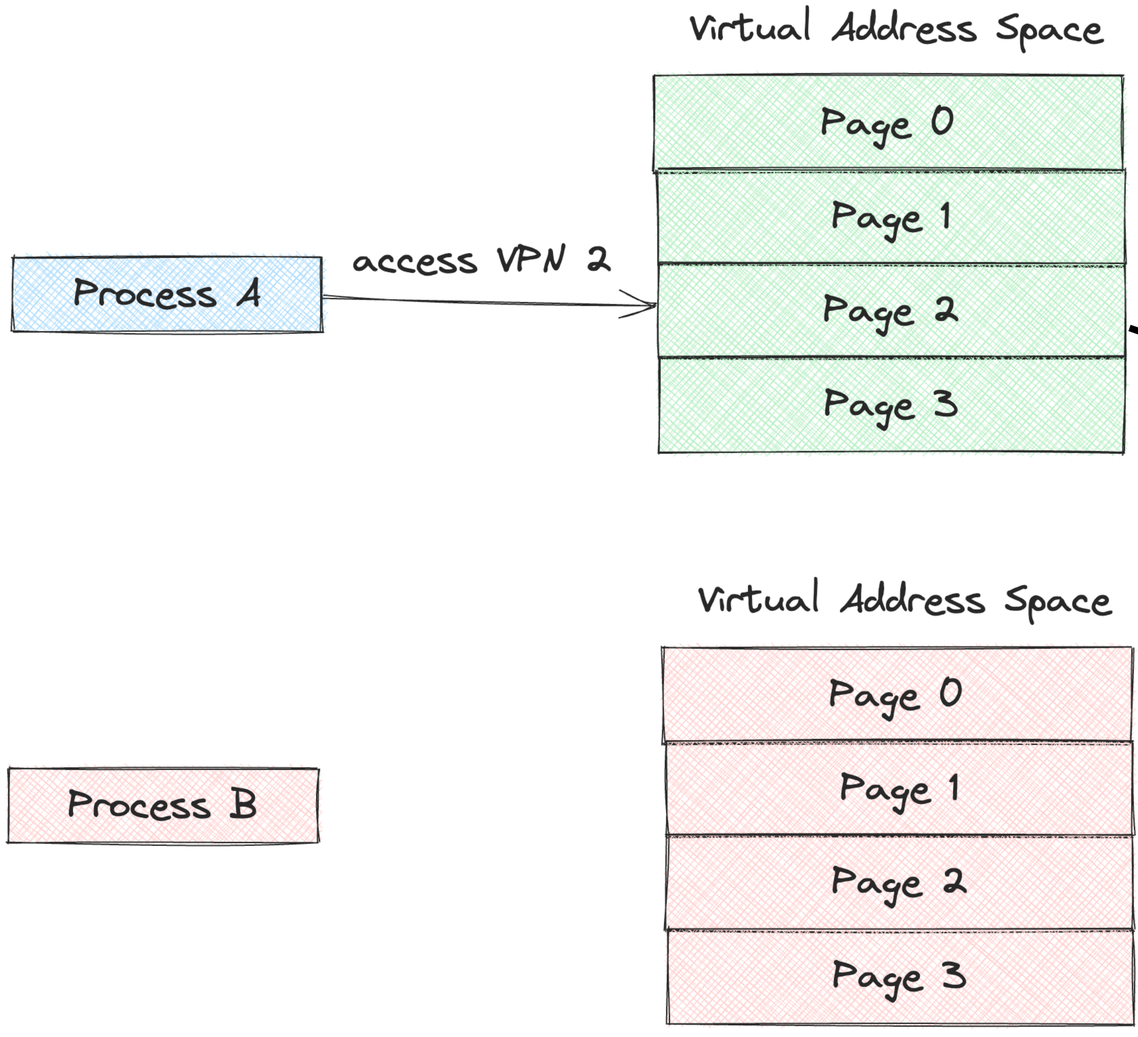
- When returning from TLB miss
  - Hardware must save the PC of current instruction and again execute the same instruction for **retry** resulting in hit
  - In usual scenario the return-from-trap goes to next instruction
- OS should not go into infinite chain of TLB misses
  - TLB miss handlers are also code that requires address translation
  - Keep TLB miss handlers in physical memory
  - Use some permanent translation slots for handler code



Inside TLB

- TLB is managed by fully associative method
  - A typical TLB have 32, 64 or 128 entries
  - Hardware searches the entire TLB in parallel to find the translation
  - Other bits
    - **Valid:** Enty has a valid translation
    - **Protection**: how a page can be accessed (read only, read, etc)
    - Other bits: **address space identifier**, **dirty bit**, etc.

What about Context Switch?

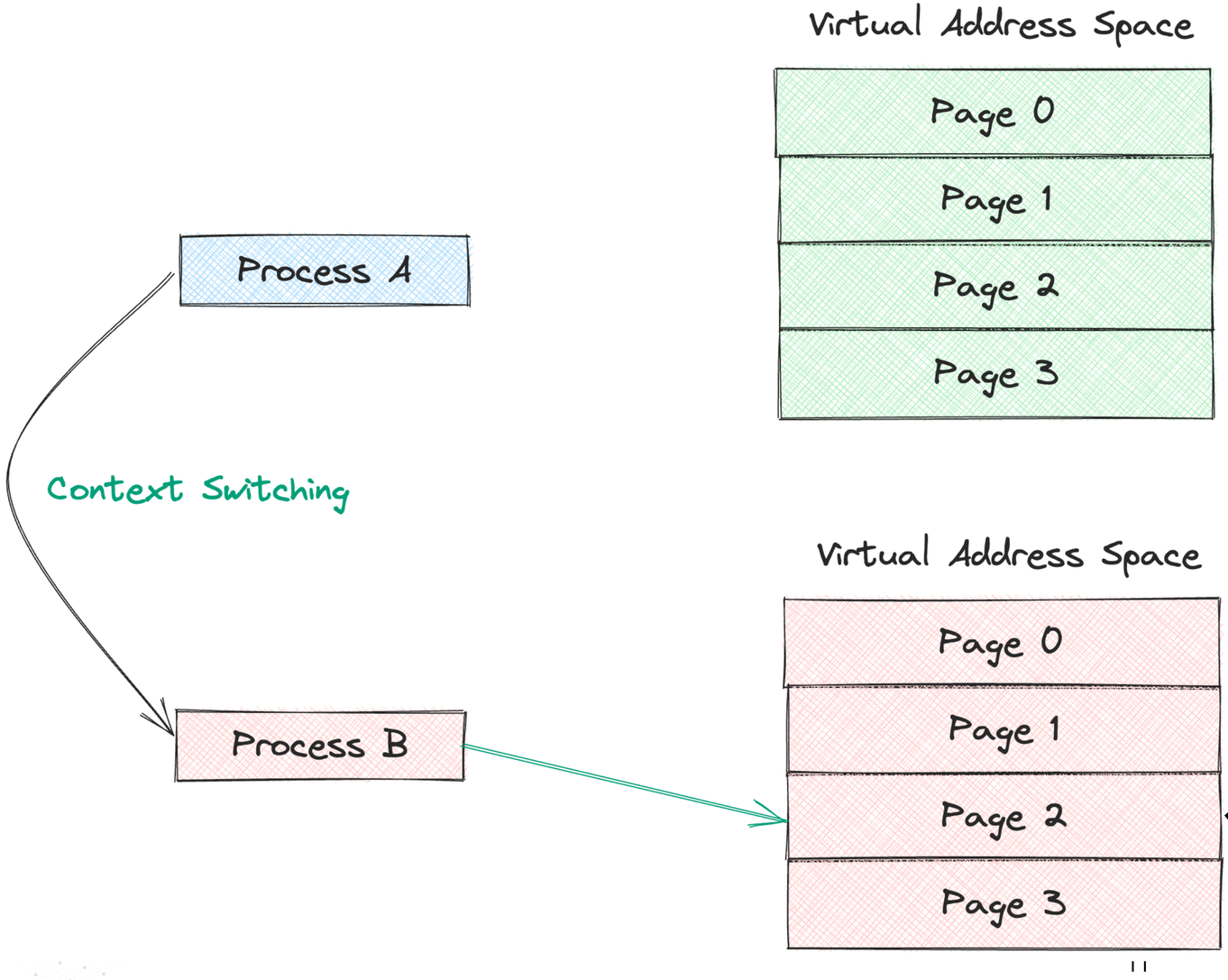


Insert TLB Entry

TLB Table

VPN	PFN	Valid	Prot
2	100	1	rwx
...	...	...	...

Context Switching and TLB



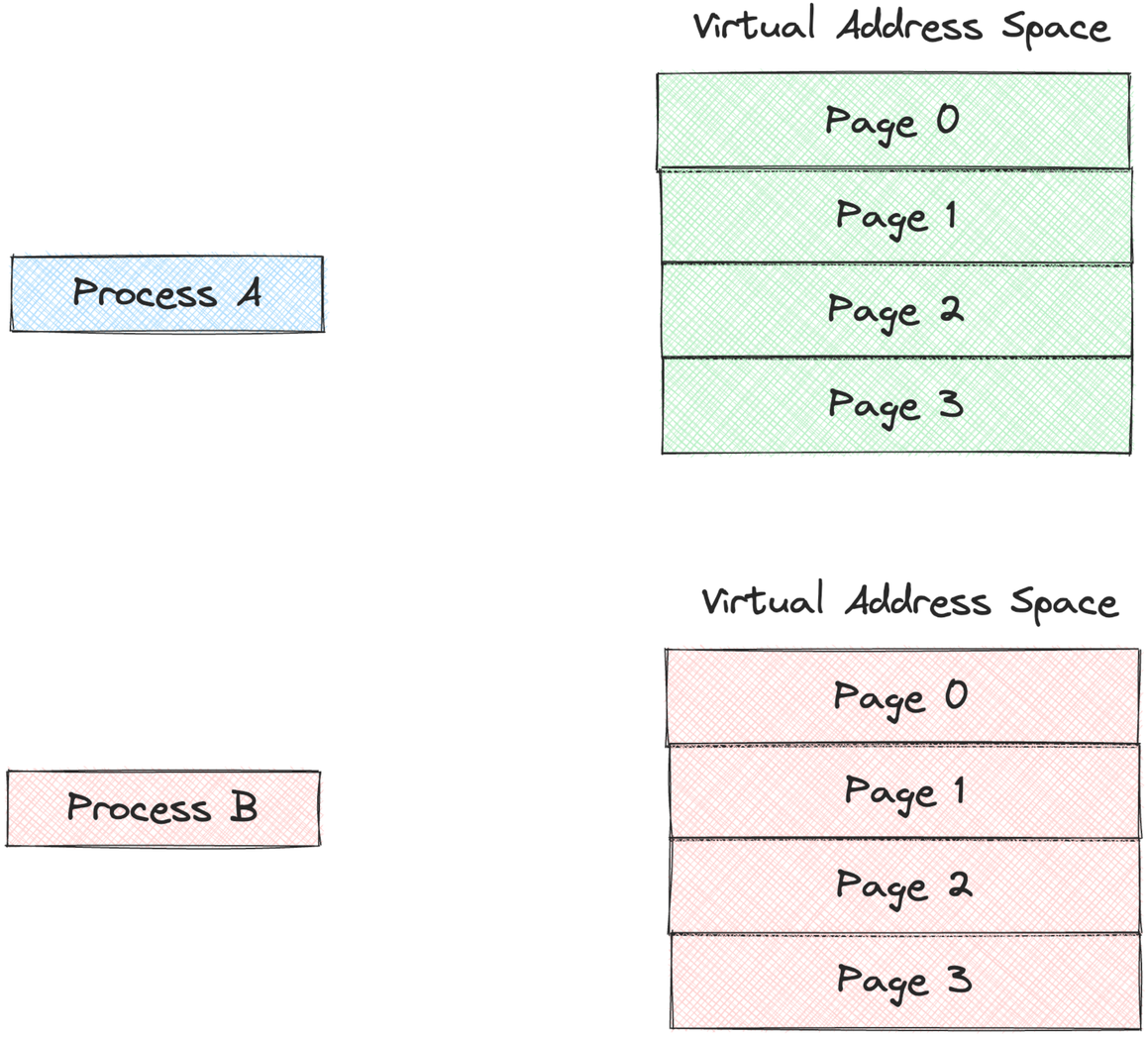
Flushing can be an option but… TLB miss?

TLB Table

VPN	PFN	Valid	Prot
2	100	1	rwx
…	…	…	…
2	180	1	rwx

Insert TLB Entry

Address Space Identifier



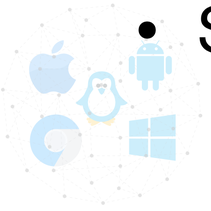
VPN	PFN	Valid	Prot	ASID
2	100	1	rwx	1
...	...	...	...	
2	180	1	rwx	2

Address Space Identifier (ASID) field to distinguish (8 bits)

Processes can Share a Page

VPN	PFN	Valid	Prot	ASID
2	100	1	r-x	1
...	...	...	...	
10	100	1	r-x	2

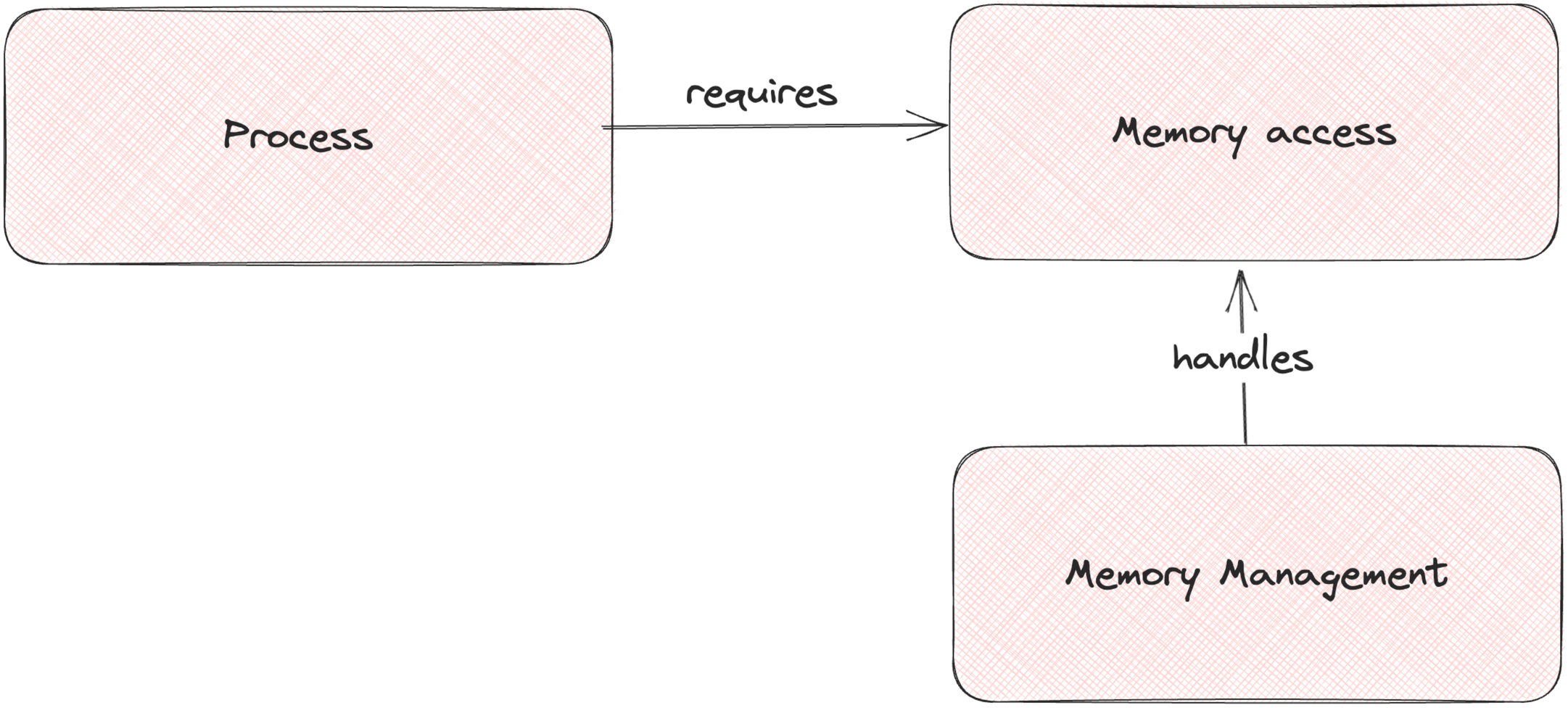
- Sharing can be useful, it reduces the number of physical frames



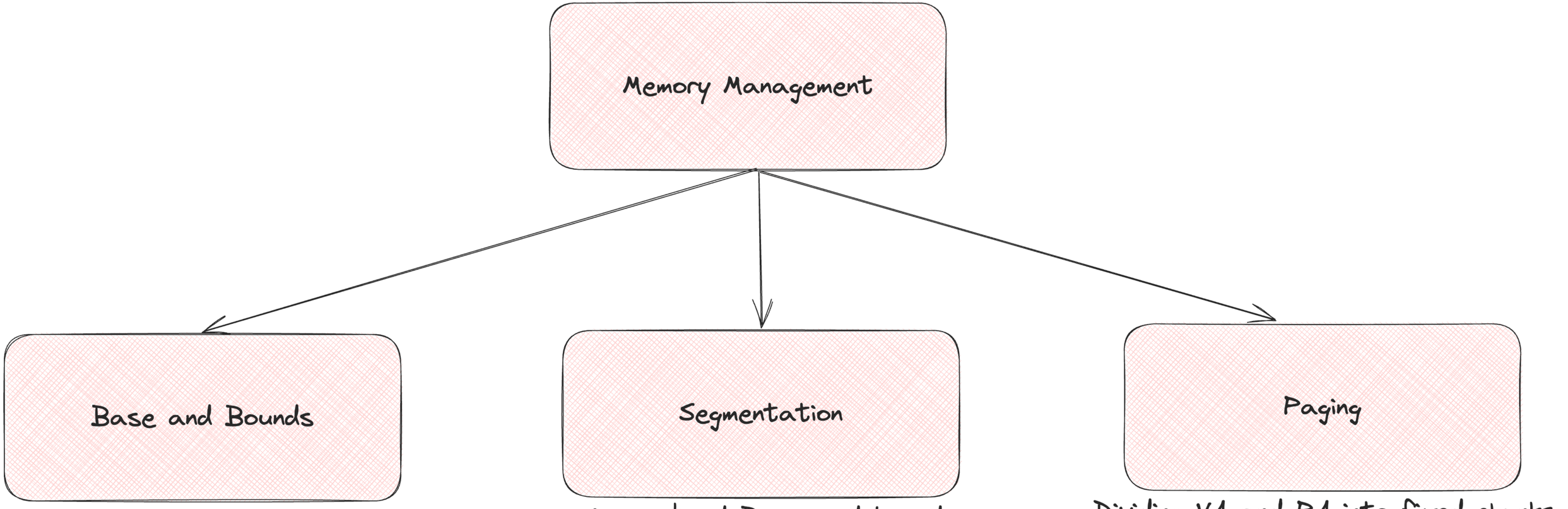
Still some issues!

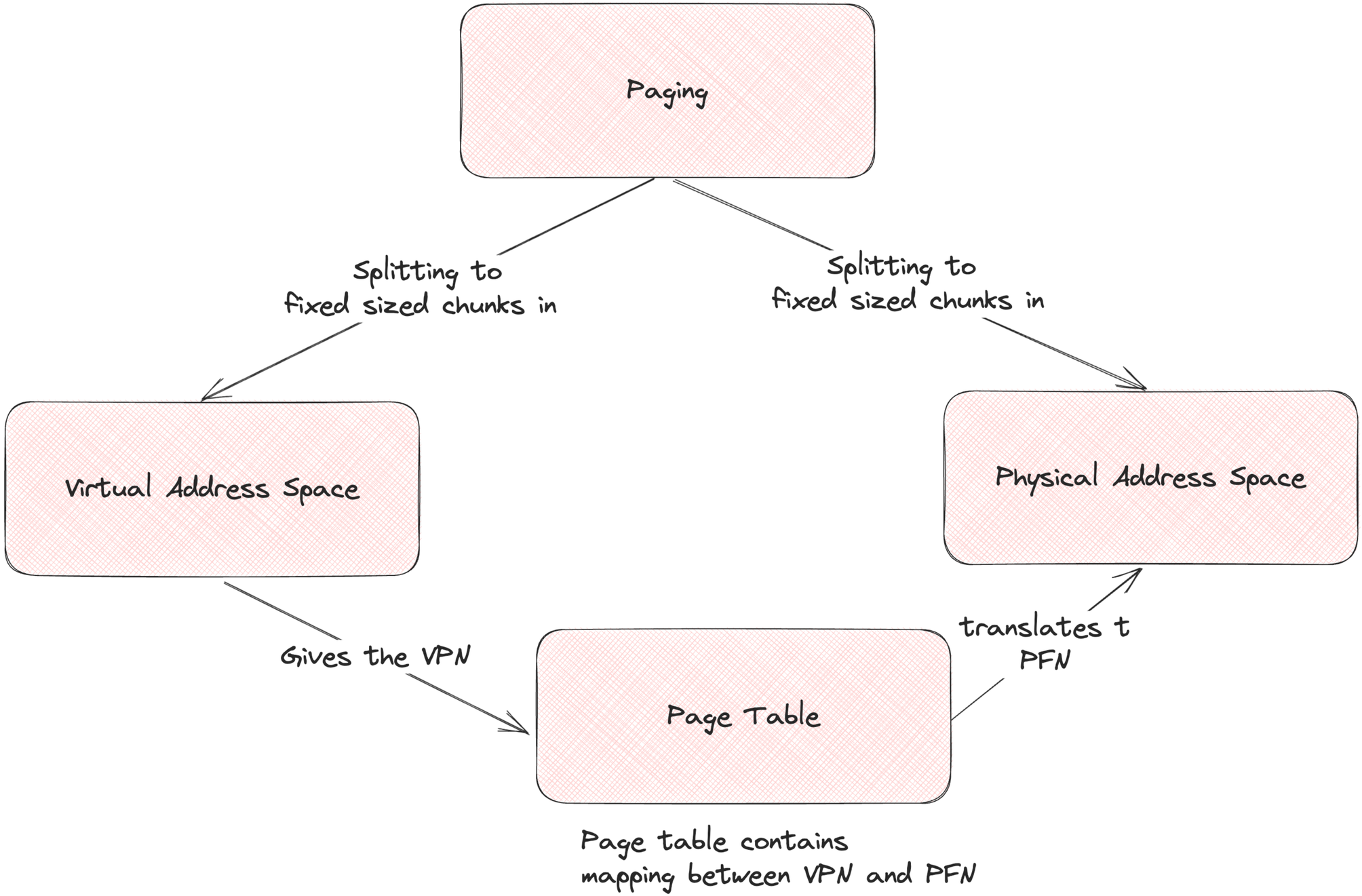
- TLB has limited size
  - Which entry to replace when adding new entries?
- What about the size of the page table?
  - Are there ways to minimise them?
  - Where will they be stored?

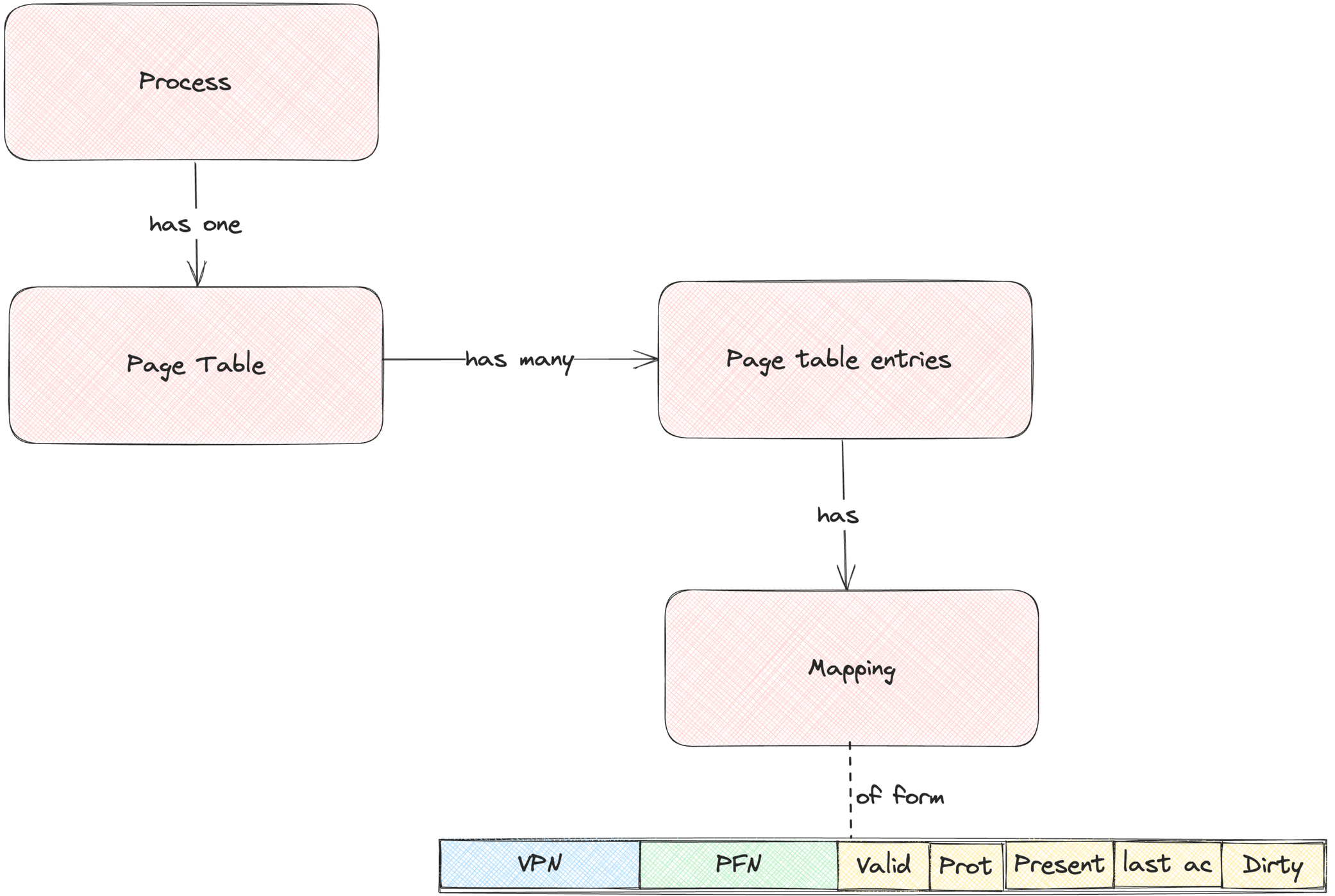
Memory Virtualization: High level view

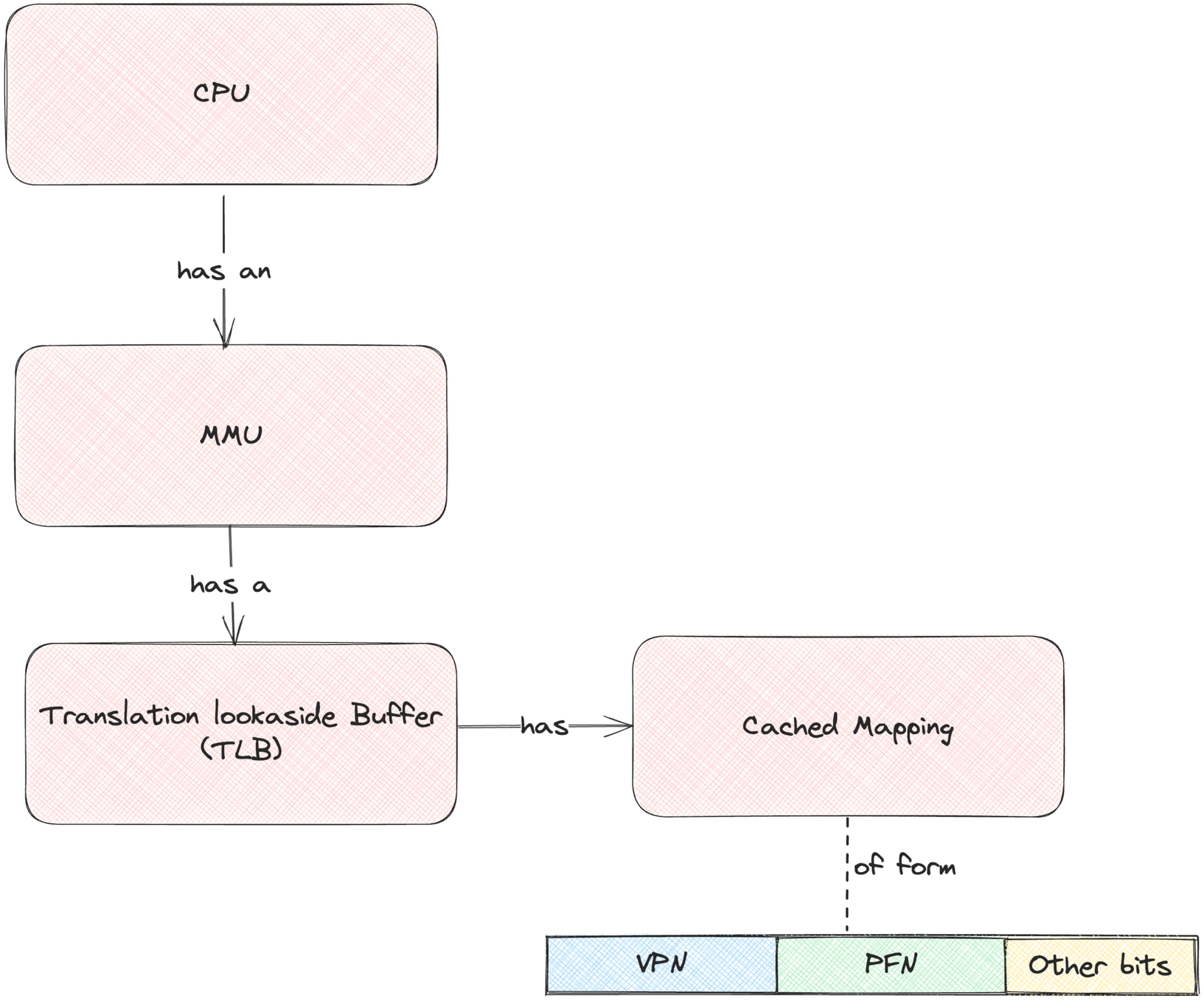


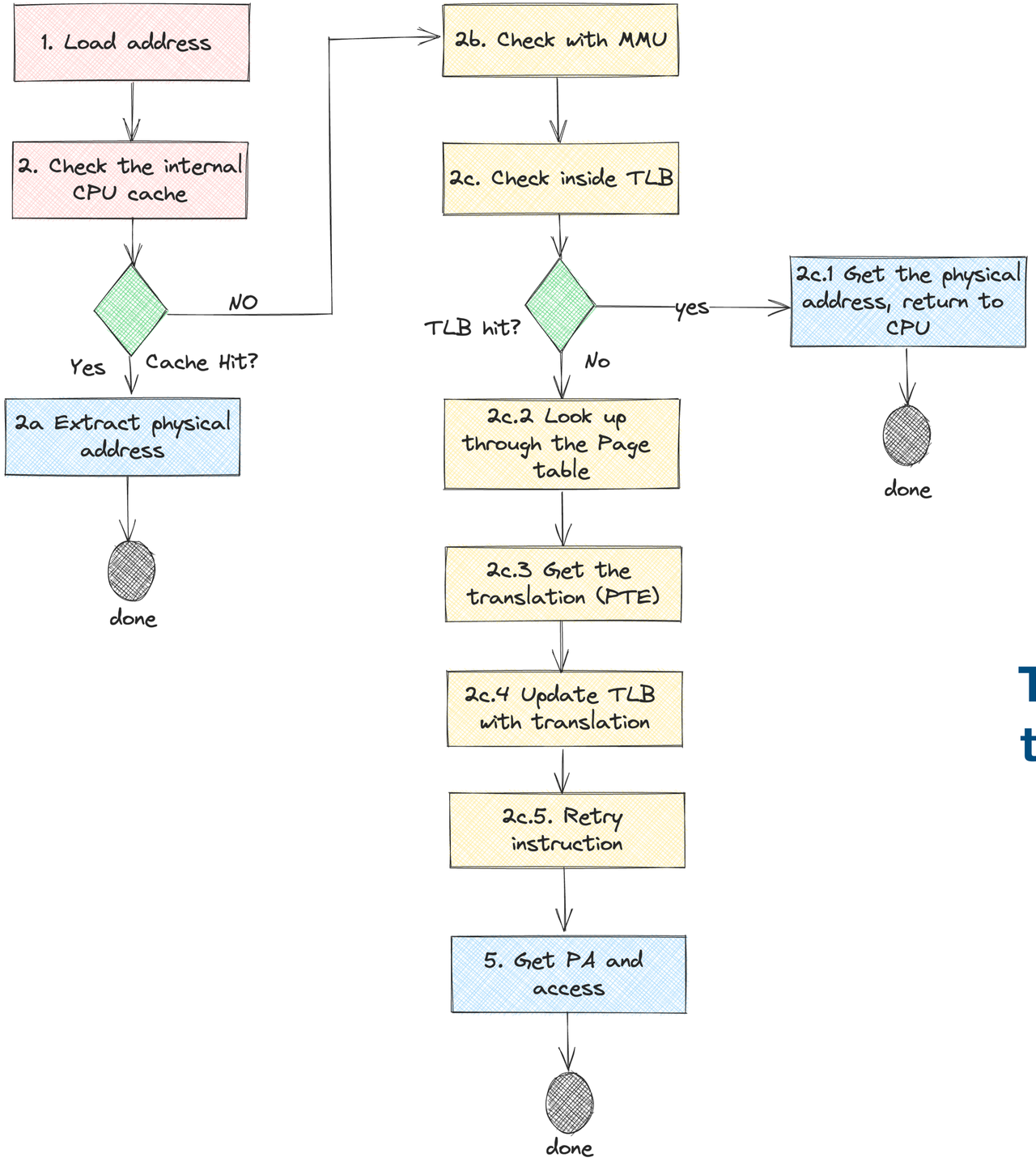
Memory Management: A Quick Recap




Paging
Splitting to
fixed sized chunks in
Splitting to
fixed sized chunks in
Virtual Address Space
Physical Address Space
Gives the VPN
translates t
PFN
Page Table
Page table contains
mapping between VPN and PFN


Process
has one
Page Table
has many
Page table entries
has
Mapping
of form
VPN
PFN
Valid
Prot
Present
last ac
Dirty


CPU
has an
MMU
has a
Translation lookaside Buffer (TLB)
has
Cached Mapping
of form
VPN
PFN
Other bits



The overall address translation Process

Size of Page Tables

- Let us take a guess about the size of a page table
  - Consider a 32 bit address space with 4 KB pages. What will be the number of bits for offset?
    - 12 (4 KB = 2^{12})
    - VPN: 20 bits, think about possible translations?
    - 2^{20} translations ~ **a million mappings**
    - If each mapping is 4 bytes => total 2^{20} X 4 = 4 MB per process per page table
    - What if there are 100 processes => **400 MB just for address translations!**

What can be done to manage size?

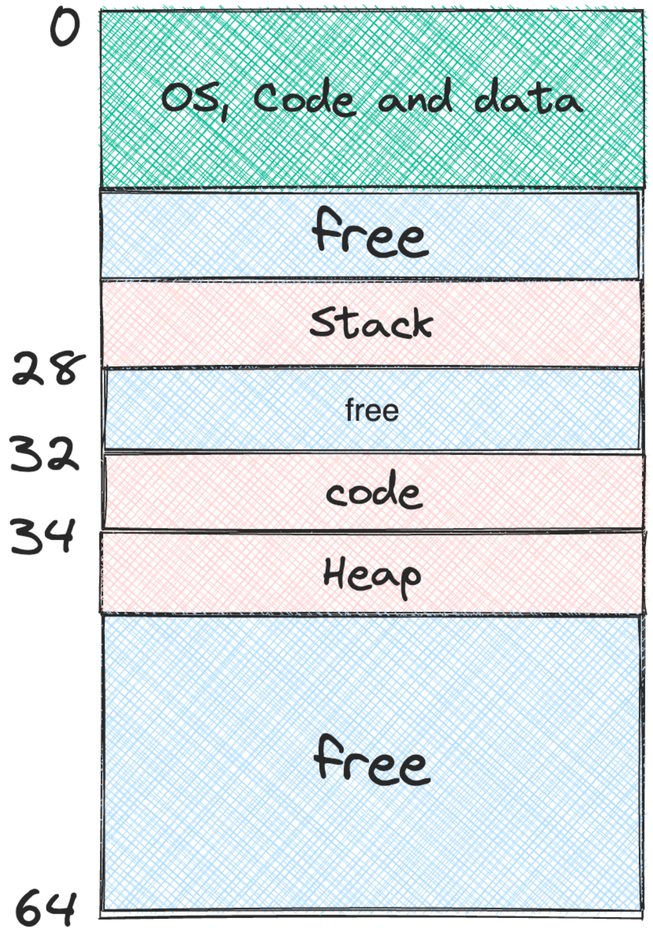
- Can we come up with smaller page tables?
  - Linear page tables did help but they can be really big!!
- Simple solution: Why not larger pages?
  - Assume a 32 bit address space but instead of 4 KB pages how about 16 KB pages
  - 16 KB = 2^{14} (Offset -> 14 bits, VPN -> 18 bits)
  - 2^{18} entries in the page table ~ **1 MB per page table** (assuming 4 bytes per entry) - Almost 1/4 reduction!!

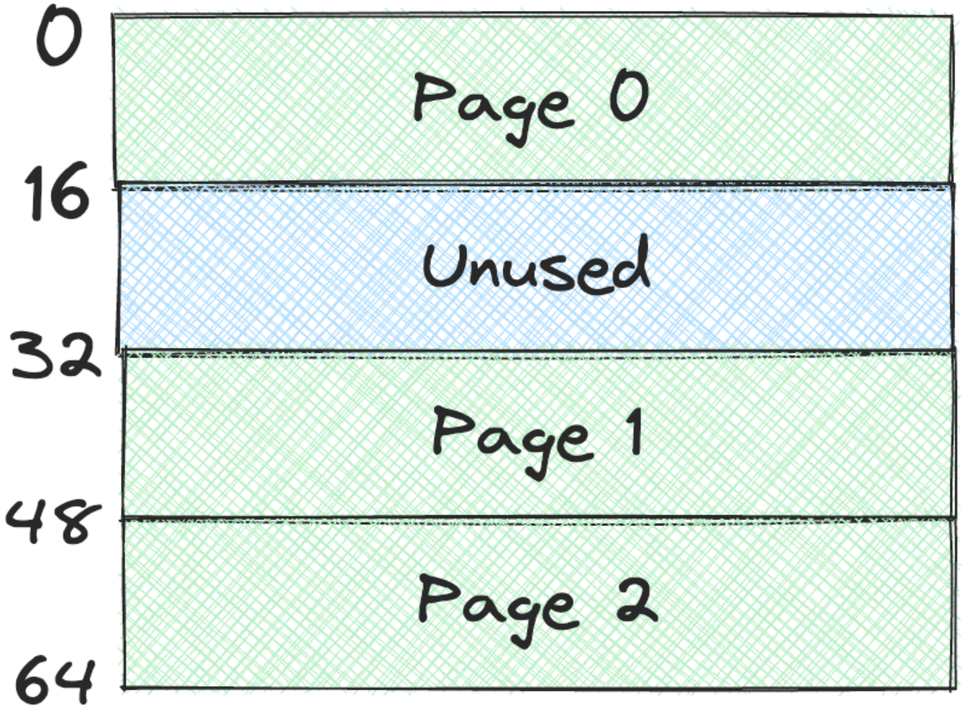
Increasing Page Size: How good is it?

- Larger page sizes can suffer more issues with internal fragmentation
- Applications overly gets filled up with large pages and there will be empty sizes
- Due to this issue, most system relatively small page size
  - Common size: 4 KB or 8 KB
- Can we think of something else?
  - How about thinking in the lines of segmentation and paging?

Hybrid approach: Paging and Segmentation

Can we get the best of two worlds?



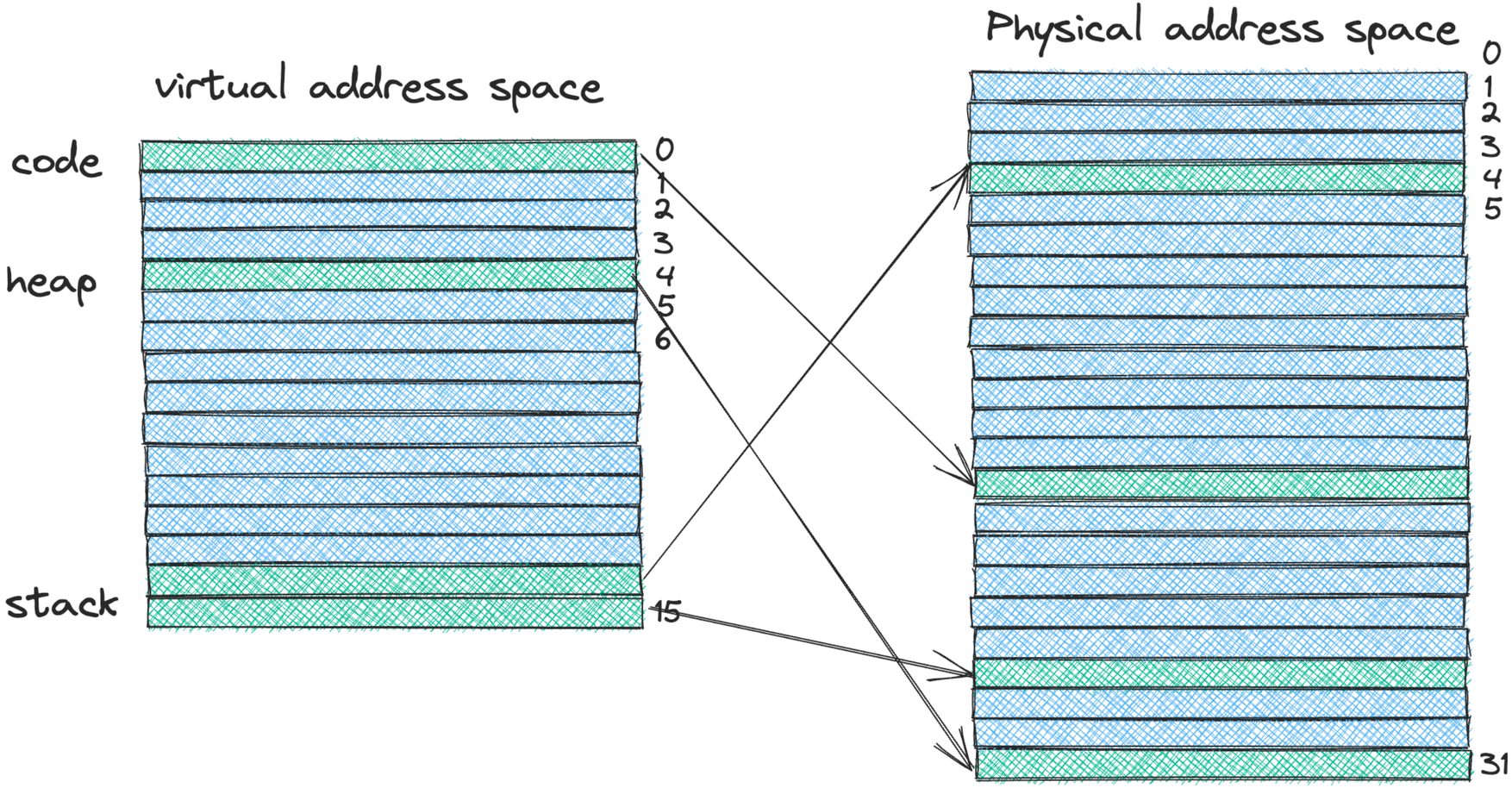


Hybrid Approach: Paging and Segmentation

- Jack Dennis, the creator of Multics had such an idea
  - Construction of Multics virtual memory system
  - Combine Paging and Segmentation was the key aspect
- This can have a good reduction in the size of page table
  - What are the key segments in a process?
  - What was the approach used in Paging?
- Let us revisit linear page table!

Revisiting Linear Page table

- Consider a tiny 16 KB address space with 1 KB pages

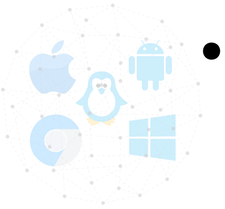


Most of the entries will be invalid

Lots of space Will be wasted!!

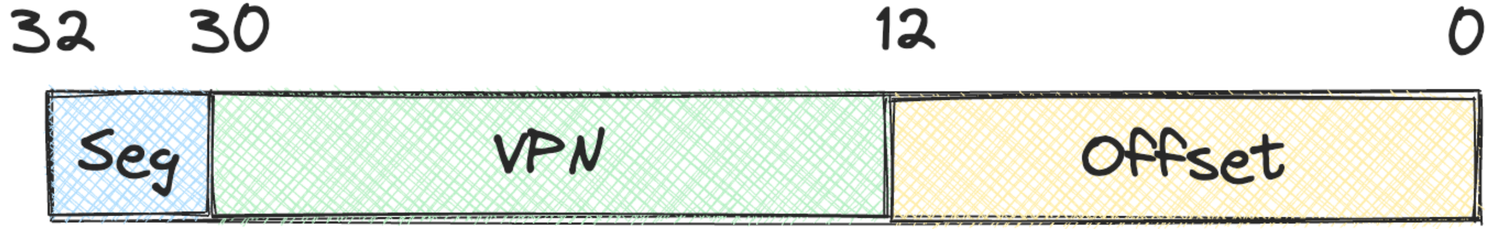
Hybrid Approach: Paging and Segmentation

- Instead of having one page table per process?
  - Can we have it for every segment?
  - There will be total of 3 pages
  - With segmentation
    - It was one base register
    - One bound register
  - In hybrid, there will be those registers in the MMU - What would be this?



What about registers?

- Base register and bounds register
- Base register is about base of the page table that correspond to the segment
- Bound indicates the end of the page table
- Assume 32 bit VA space with 4 KB pages and address space split into three segments
- How to do addressing?



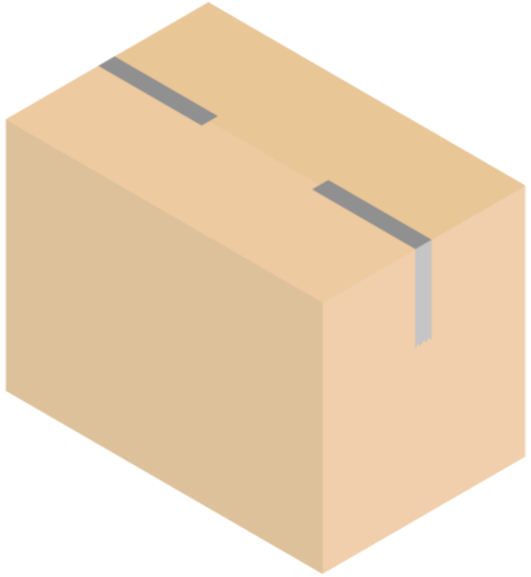
How good is the Hybrid Approach

- The good parts
  - Ensures significant memory savings compared to linear page table
    - Any reason?
    - Unallocated size between stack and heap no longer takes up space in page table?
- Some issues
  - Segmentation is used - Not quite flexible as paging
  - The problem of external fragmentation comes back - **Page tables** can be of arbitrary size. Finding memory for them can be difficult

Going back to the Analogy

Over the idea of categories

Use fixed sized shelves



Each box can contain some items
The boxes are placed inside fixed sized shelves

Some mapping between product, box and shelves

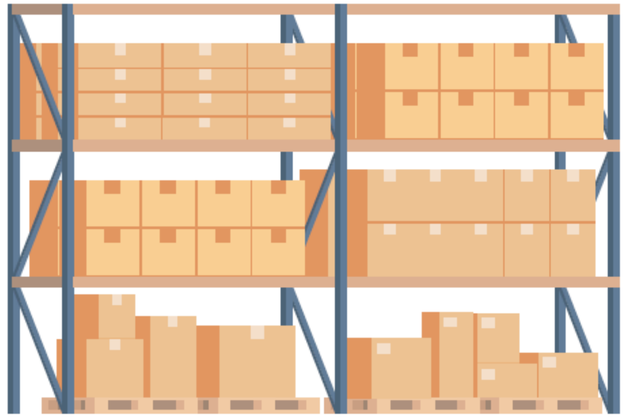
Going back to the Analogy

Warehouse

Location List

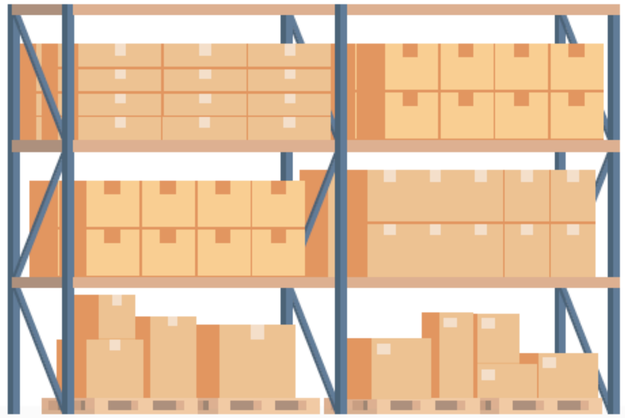
Index

TV



Index

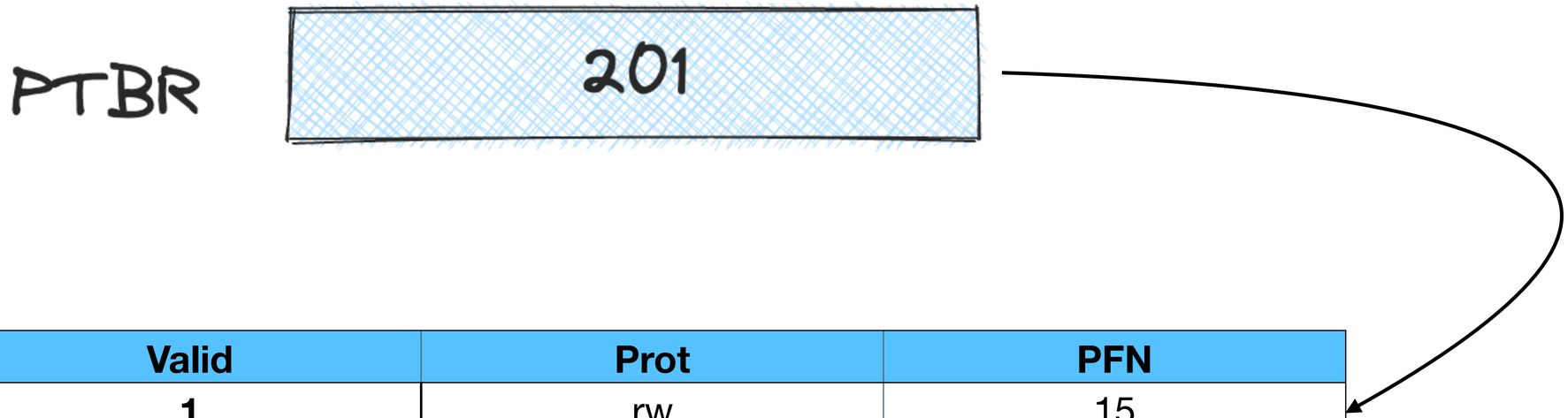
Living Room Furniture



Introducing Multi-level Page Tables

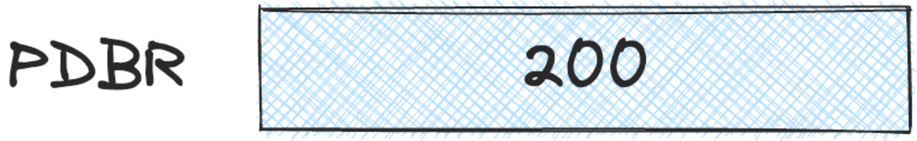
- In simple terms - Turn the page table into a tree like structure
- Chop up the page table into page sized chunks
  - If an entire page of page table is full of invalid entries, don’t allocate that page of the page table at all.
  - To track, use a simple data structure, **Page Directory**
- Page directory can specify where the Page of page table is located
  - It allows to ensure that a part of the page table contains no valid pages

Lets revisit Linear Page Table in Action



Valid	Prot	PFN
1	rw	15
1	rwx	16
0	-	-
0	-	-
0	-	-
1	rx	100
1	rwx	25

Multiple levels of Mappings are Required



Valid	PFN
1	201
0	-
0	-
1	204

The Page Directory (PFN: 200)

Page 1 and 2 are not allocated - PDBR?

Page 0 of PT (PFN 201)

Valid	Prot	PFN
1	rw	15
0	rwx	16
0	-	-
0	-	-

Page 3 of PT (PFN 204)

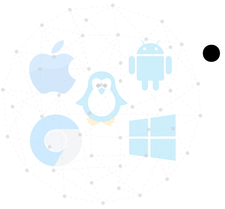
Valid	Prot	PFN
1	rx	100
0	rwx	25
0	-	-
0	-	-

Multi-Level Page Tables has Advantages

- Allocate page table space promotional to the address space being used
- Each portion of the page table fits neatly within a page
  - Easier to manage memory
  - OS can simply grab the next free page when it needs to grow the page table
- Compare with linear page table indexed by VPN requires contiguous memory
  - If page table is large, finding such block of memory may be hard!
- Multi-level page table also comes with a cost - Any guess?

Multi-Level Page Table comes with Overhead

- TLB Miss Scenario
  - There will be two loads from the memory
  - One for the Page directory and one for PTE itself
  - **Trade-off** between time and space
  - Smaller page tables, yes but not free! - Lookups are costly
- Another issue is complexity
  - Building the multi-level tables and complicated lookups



An Illustrative Example

- Consider an address space of 16 KB with 64-byte pages
  - 16 KB: 2^{14} as bits
  - 14 bit virtual address space
  - 64 byte implies - 6 bits for offset
  - 8 bits remaining for VPN
  - Assume that 0 and 1 are for code, 4 and 5 for stack, 254 and 255 for heap
- If it was full table then 256 entries - Assume each entry 4 bytes ~ 1 KB table

An Illustrative Example

- Can be divided into 16, 64 byte pages
- Each page can hold 16 PTE (16 *4 = 64)
- 256 entries spread over 16 pages
- Number of entries in the Page Directory: 16 How?
- Number of bits required to index into directory: 4 How?
- Number of bits required to index into PTE: 4 How?

Representation

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

An Illustrative Example

- Eg: Address translation
  - 11 1111 1000 0000
  - 8 bits VPN: 4 bits for PDR, 4 bits for getting to PTE
  - Last 6 bits offset
  - Try to work it out!

Solution

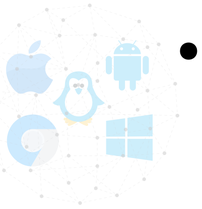
- 1111 points to 101
  - PT corresponding to PFN 101
- Next step is to get the PTE
  - VPN (1110) corresponds to 14th entry
- PFN of the 14th entry is 55!
  - Add offset to 55 PFN to get the exact physical address!

More than Two Levels

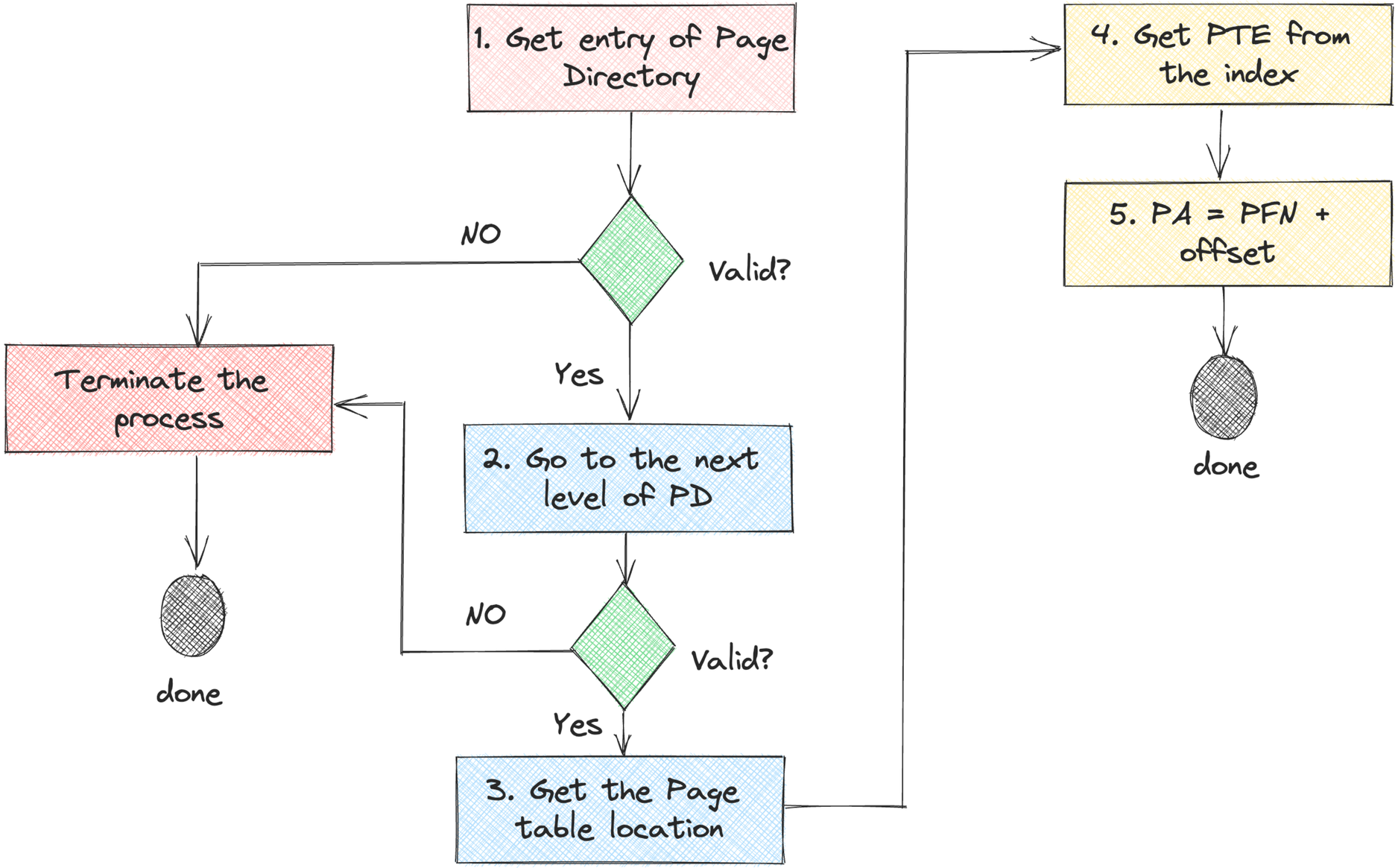
- Its not about just two levels in **Multi-level page tables**
  - Deeper tree is possible
  - Why is it required?
- Assume a 30 bit Virtual address space and 512 byte pages
  - Virtual address has 21 bits for VPN and 9 bits for offset
  - Assuming PTE is 4 bytes, 128 PTE can fit on single page
  - 7 bits required to index into PT to get the PTE
  - Remaining 14 bits for mapping to get PT from PDR!!

Multi-level page tables

- Page directory itself is divided into multiple pages
  - One additional meta directory on top of all directories
  - The top most one points to the right directory
  - The directory further points to the correct Page table to fetch PTE
  - The PTE and offset is combined to generate physical address
- In case of TLB miss, the memory look up will be multi access
  - In two level itself it can be two additional access

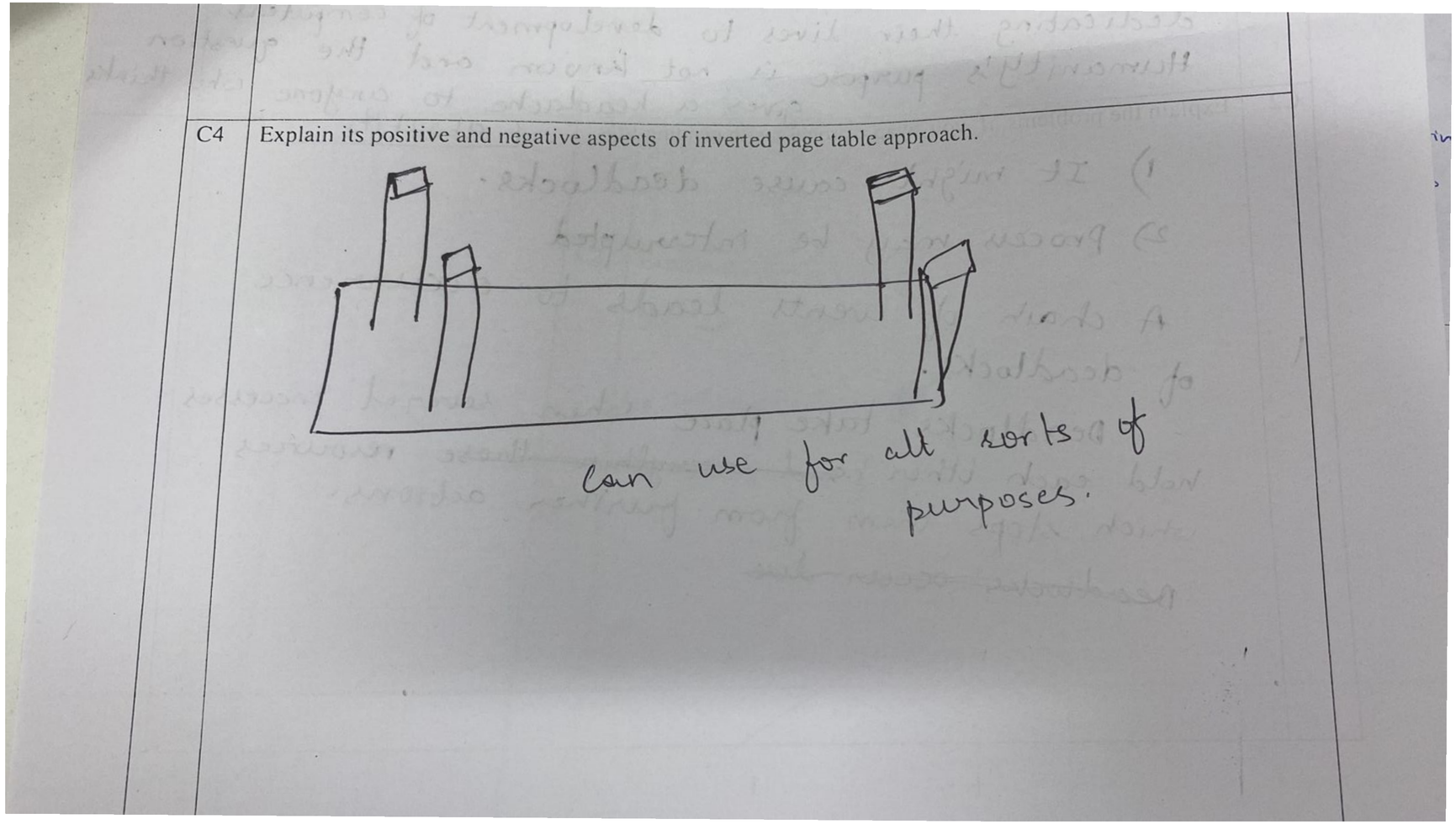


The overall flow



Inverted Page Tables

Not this one!



Inverted Page Tables

- Instead of having one page table per process
  - Have one single page table for all the processes
  - Searching for an entry is like searching through entire table
  - Linear scan can be very expensive!!
  - **Remember:** Page tables are just data structures
    - All crazy combinations can be though off! (Hash tables)

Some more questions needs answers!

- What if the size of the address space is larger than the physical space?
- Are all pages of all active processes always in main memory?
  - All of them may not fit in the main memory - so how does it work?
- OS uses a part of the disk (swap space) to store pages that are not active in use - How does that work?

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere