CS3.301 Operating Systems and Networks

Memory Virtualization - Introduction to Paging

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:

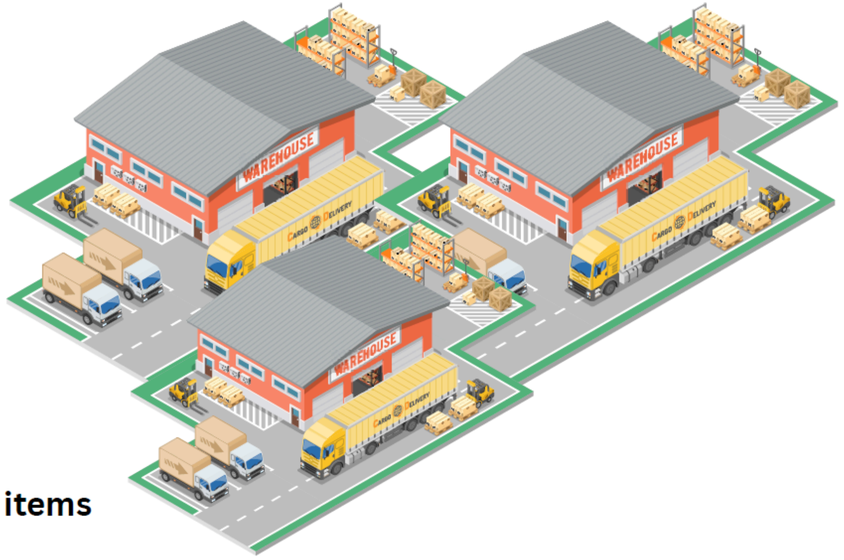
- Operating Systems: In three easy pieces, by Remzi et al.
- Lectures on Operating Systems by Youjip Won, Hanyang University

Memory Virtualization: An Analogy

Onsite Shopping

Every users have access to different items but to a limited set

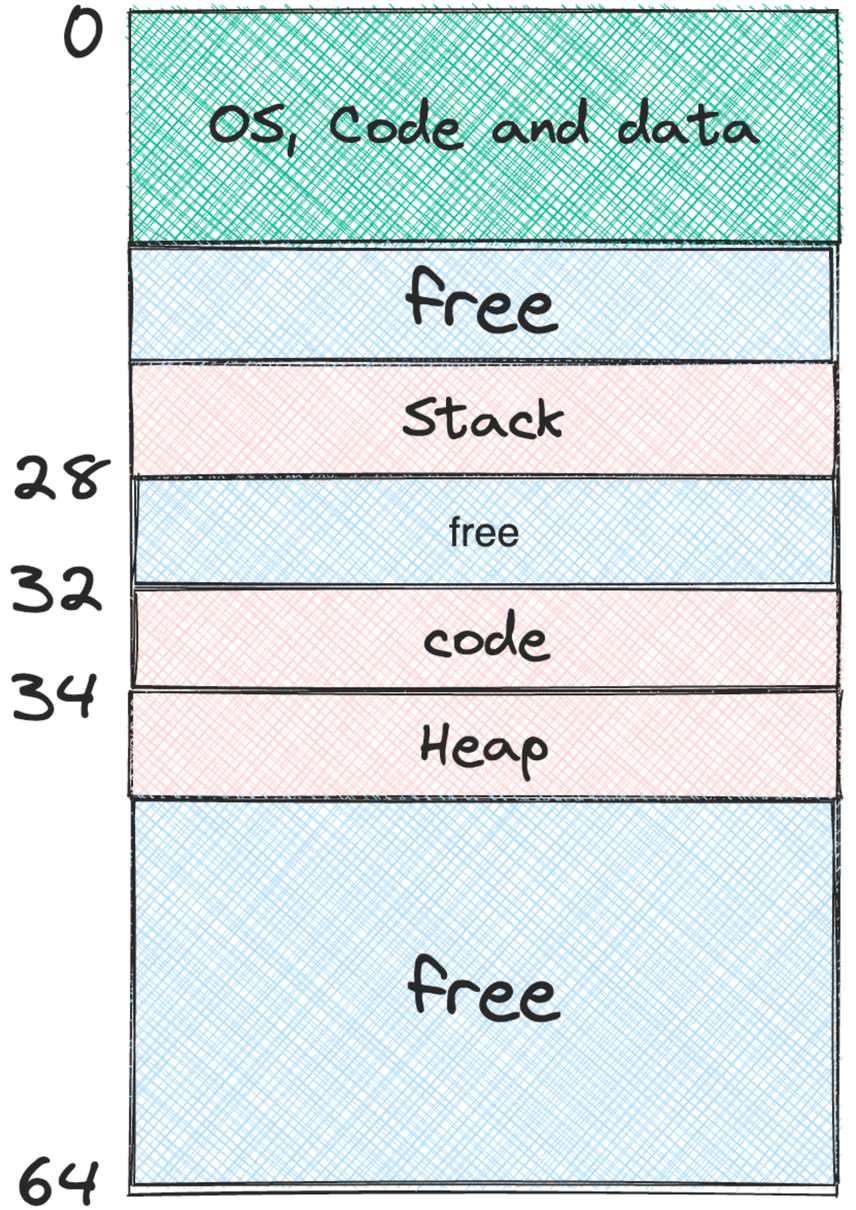
Online Shopping



Every Users feel that they have access to infinite items

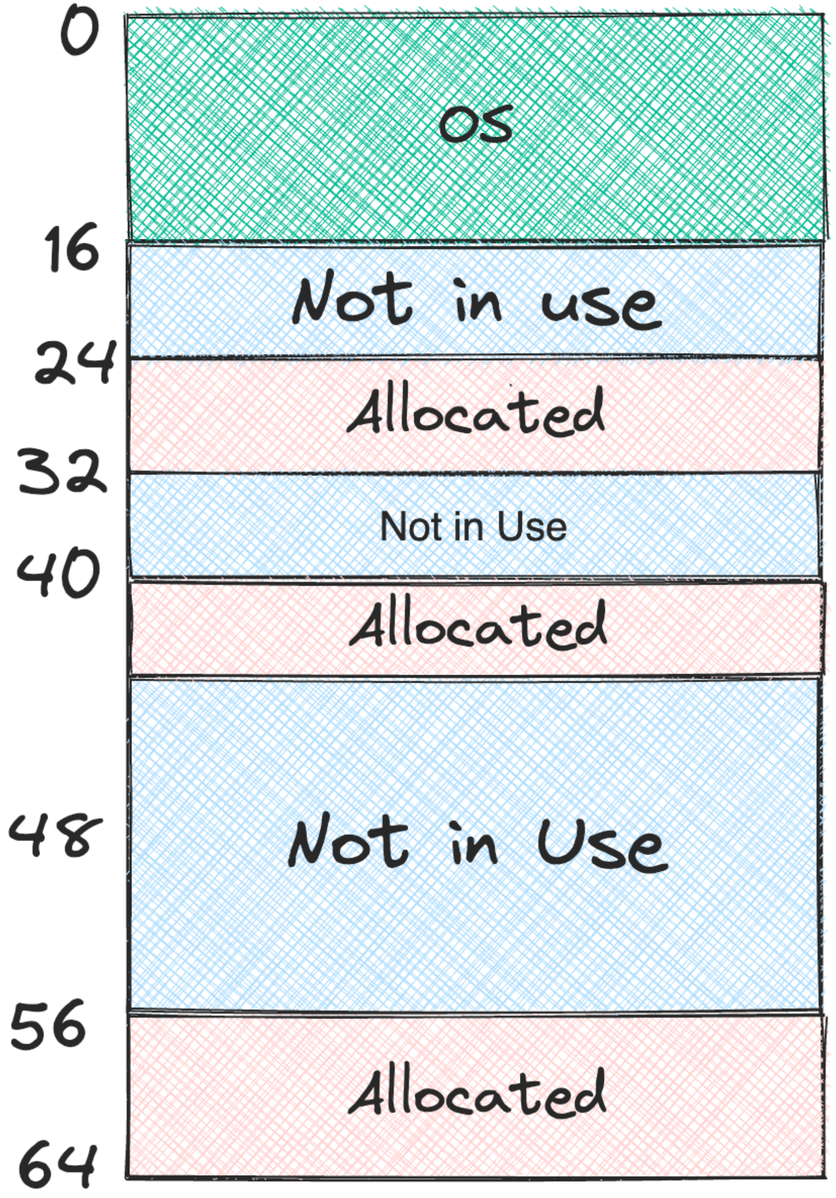
Hardware support (Registers)

Segment	Base	Size (Max 4K)
Code (00)	32K	2K
Heap (01)	34K	2K
Stack (11)	28K	2K



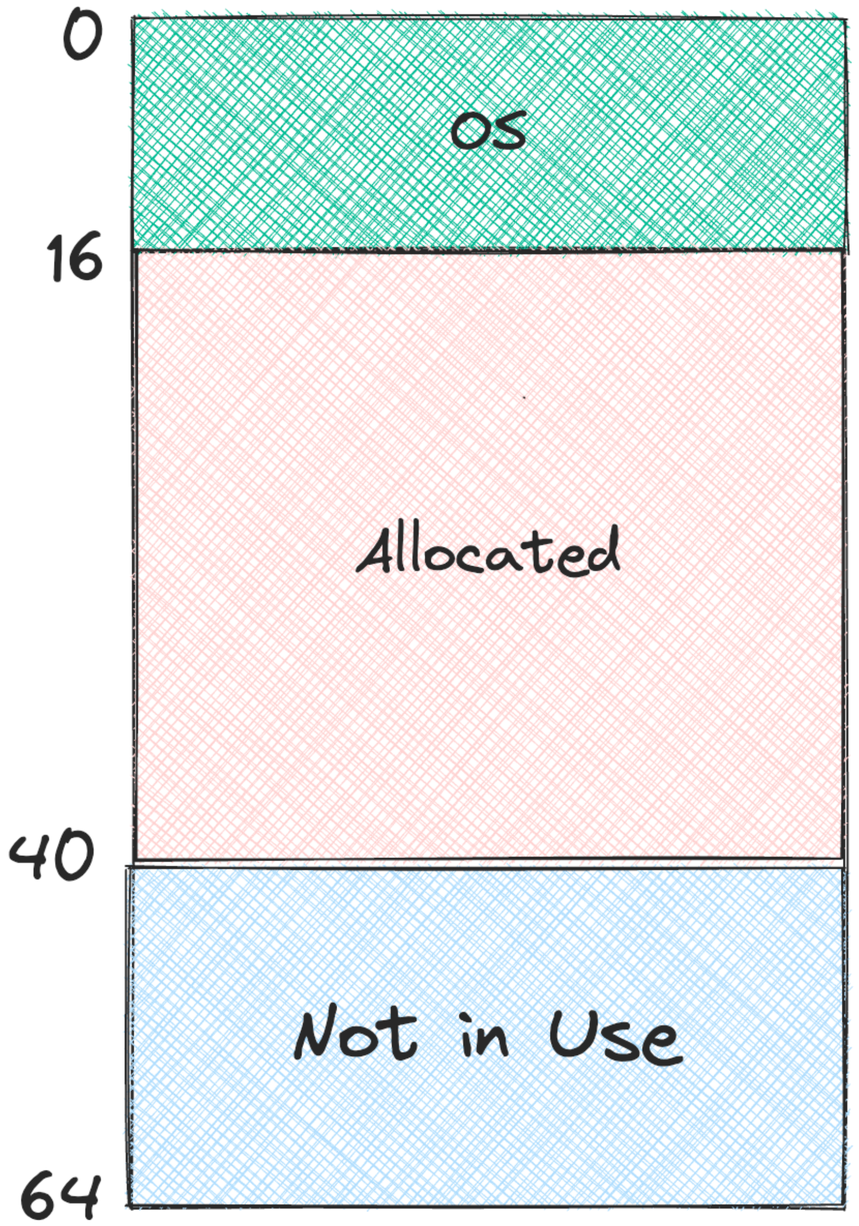
External Fragmentation

- Physical memory quickly becomes full of little holes
- Hard to allocate new segments
- Consider process wishes to allocate a 20 KB segment - 24 KB is free but not in a contiguous space!!
  - Can we come up with a compact version of this?



Compacted Version

- Seems like a more easy solution - OS could stop the running process
  - Copy data into a contiguous region
  - Change segment values to point to new region
  - Now there is larger memory
- Process is very **memory intensive!**



Algorithmic Approaches can be used

- Free-list management algorithm
  - OS can keep track of available memory
  - Different algorithms can be used to intelligently allocate
    - Best-fit: returns one closest in size that satisfies the request
    - Then other algorithms like first-fit, worst-fit, buddy algorithm, etc.
  - Good algorithm can only minimise external fragmentation but not avoid it
  - The only possible solution - **Why to allocate variable sized segments?**

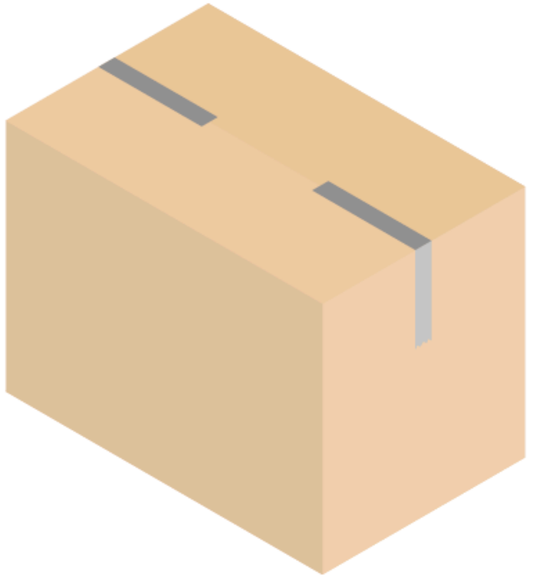
What can be done differently?

- Can we divide the physical memory into fixed sized lengths?
  - External fragmentation can be avoided
  - It will still result in internal fragmentation - This may be better!
  - This is what **Paging** does!

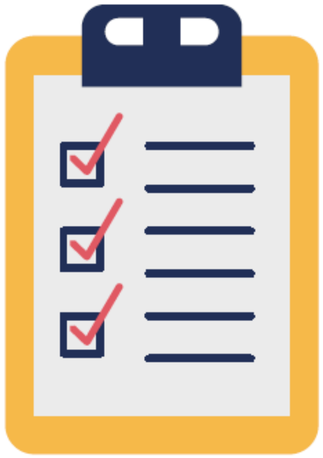
Going back to the Analogy

Over the idea of categories

Use fixed sized shelves



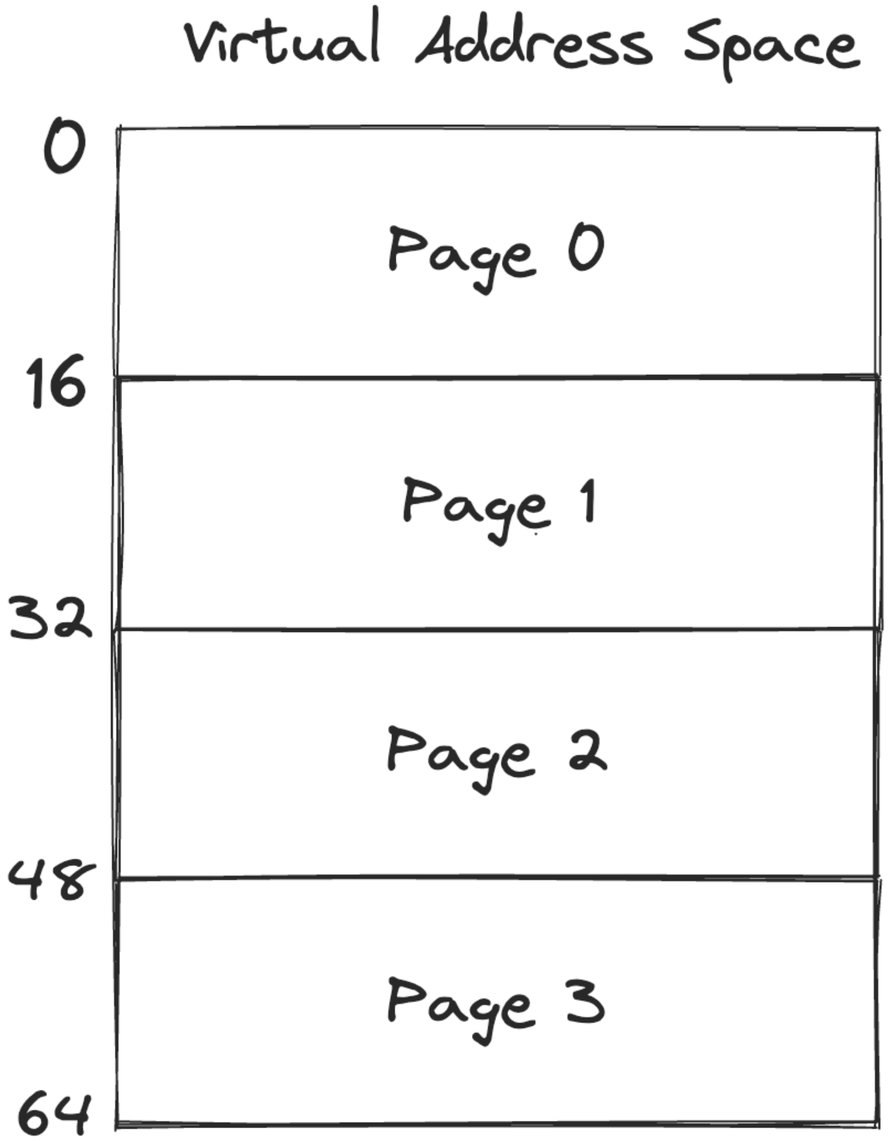
Each box can contain some items
The boxes are placed inside fixed sized shelves



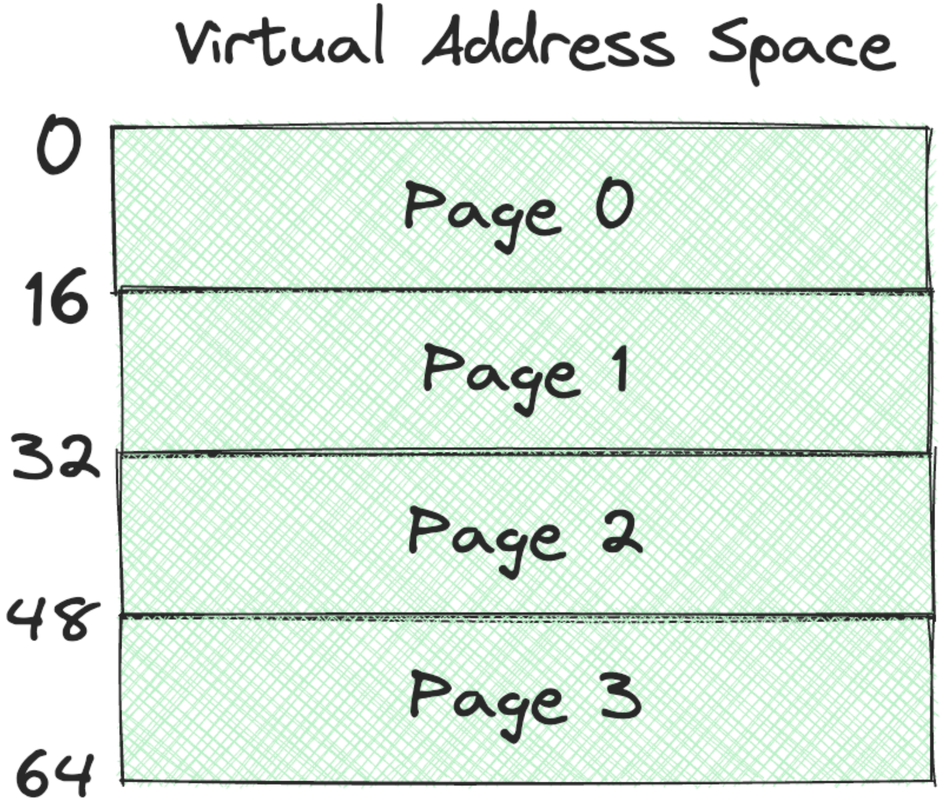
Some mapping between product, box and shelves

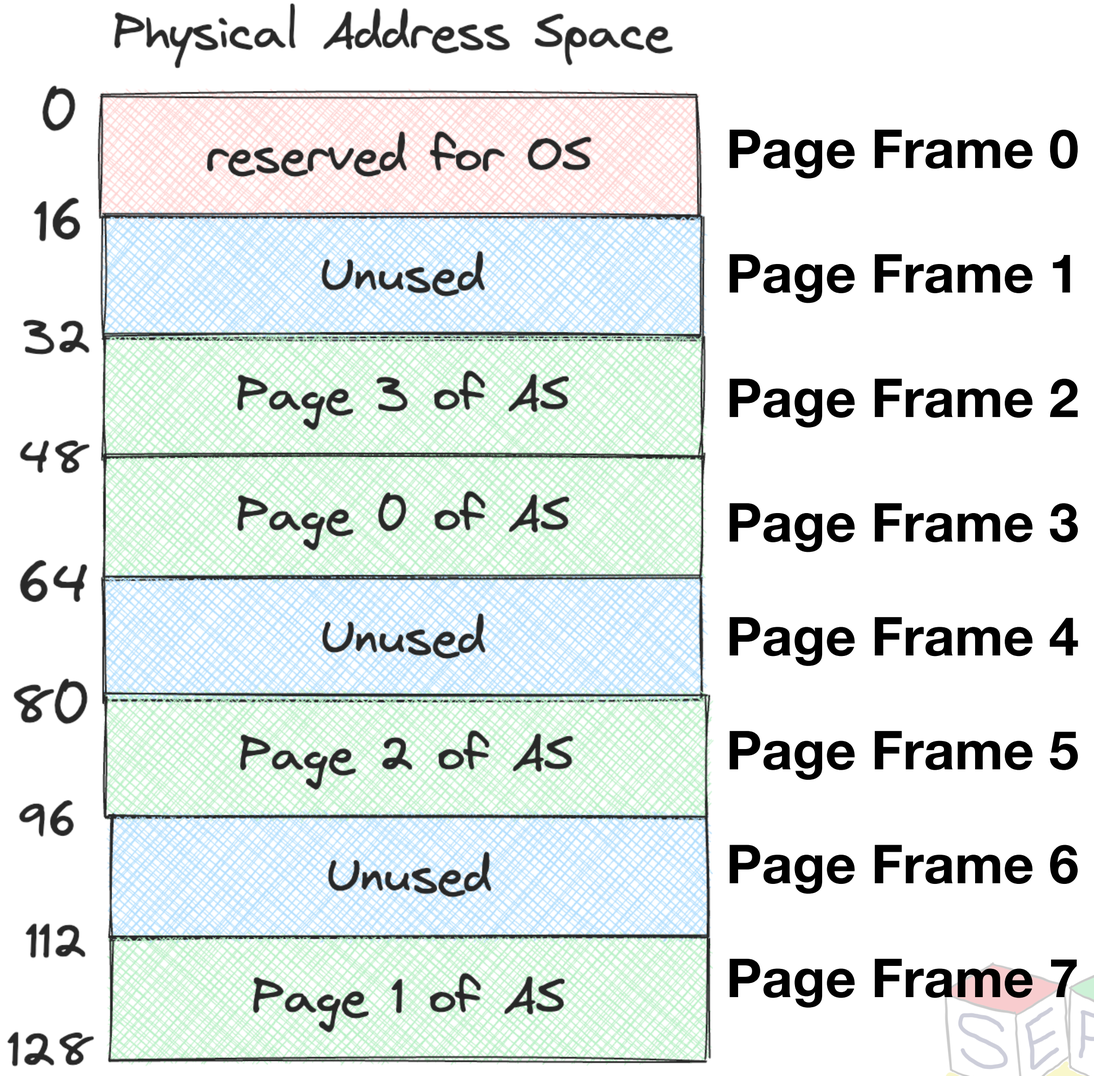
Introduction to Paging

- Idea goes back to the earlier systems - Altas
- Instead of splitting address space into three logical segments (variable size)
  - Split the address space into fixed sized units - **Pages**
  - Virtual address space and Physical address space are split into fixed sized pages
  - Its all about mapping **page in VA** to **page frame in PA**
- Eg: Virtual Address -> 64 bytes



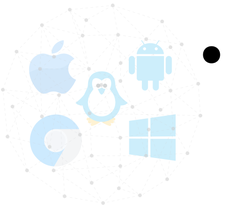
Virtual Address and Physical Address





Advantages of Paging

- **Flexibility**
  - Easy support for abstraction of physical space
  - No need to make assumption on the heap and stack space
- **Simplicity**
  - All that the OS needs to do is to find some free pages and load the process
- The OS makes use of per-process data structure - **Page table**
  - To record the mapping between page number and page frame



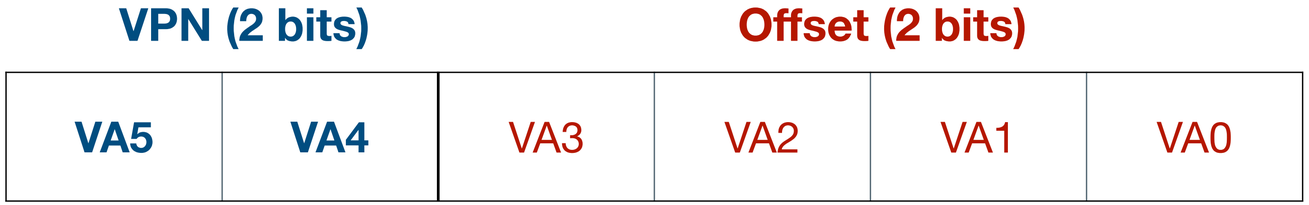
Page Table

- Role of page table is for address translation
- Page table is per-process data structure
- If another process runs, it will have its own page table
- The VPN maps to different PFNs as long as they don't share!!
- How can this be implemented?

Virtual Address Space	Physical Address Space
VPN 0	PFN 3
VPN 1	PFN 7
VPN 2	PFN 5
VPN 3	PFN 2

Process of Translation

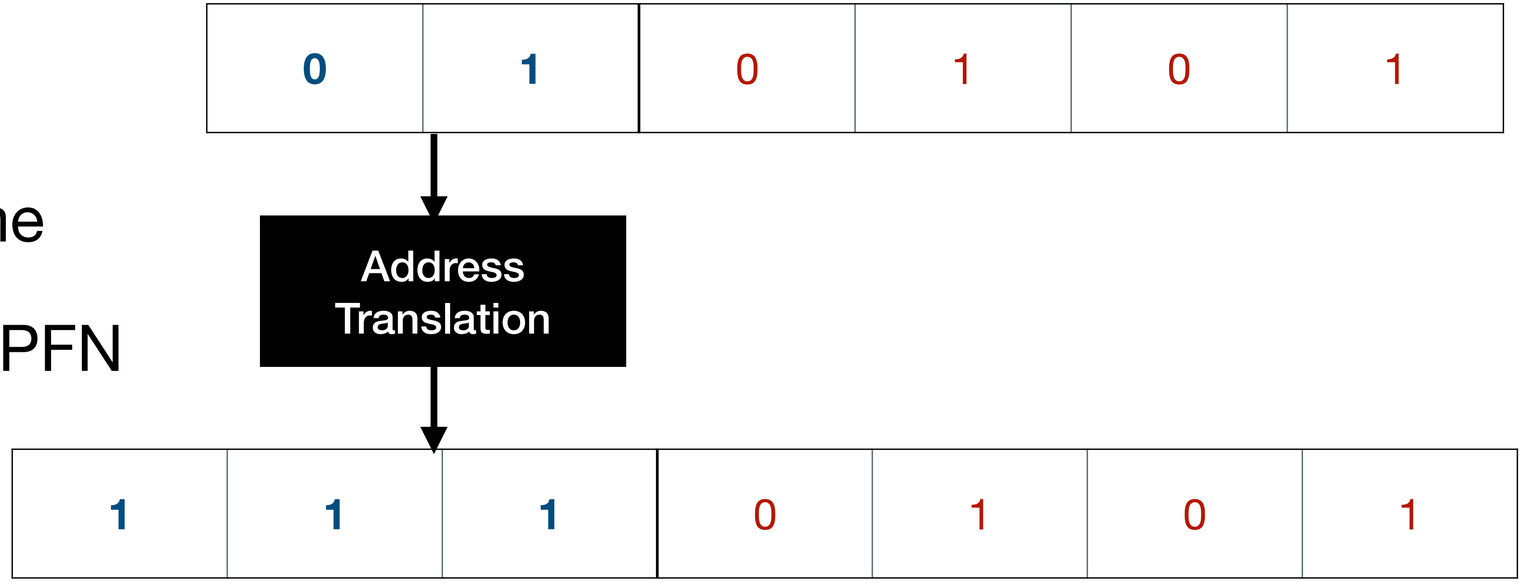
- Eg: Translation **movl <virtual address> %eax**
  - Split them into two components
    - Virtual Page Number (VPN)
    - Offset
    - VPN can be determined by number of pages
    - 64 bytes = 2^6 => 6 bits
    - Each page, 16 bytes = 2^4 => **4 bits (offset)**
    - **Remaining 2 bits for VPN**



Simple Example

Example instruction: mov 21 %eax

- Convert 21 into binary:



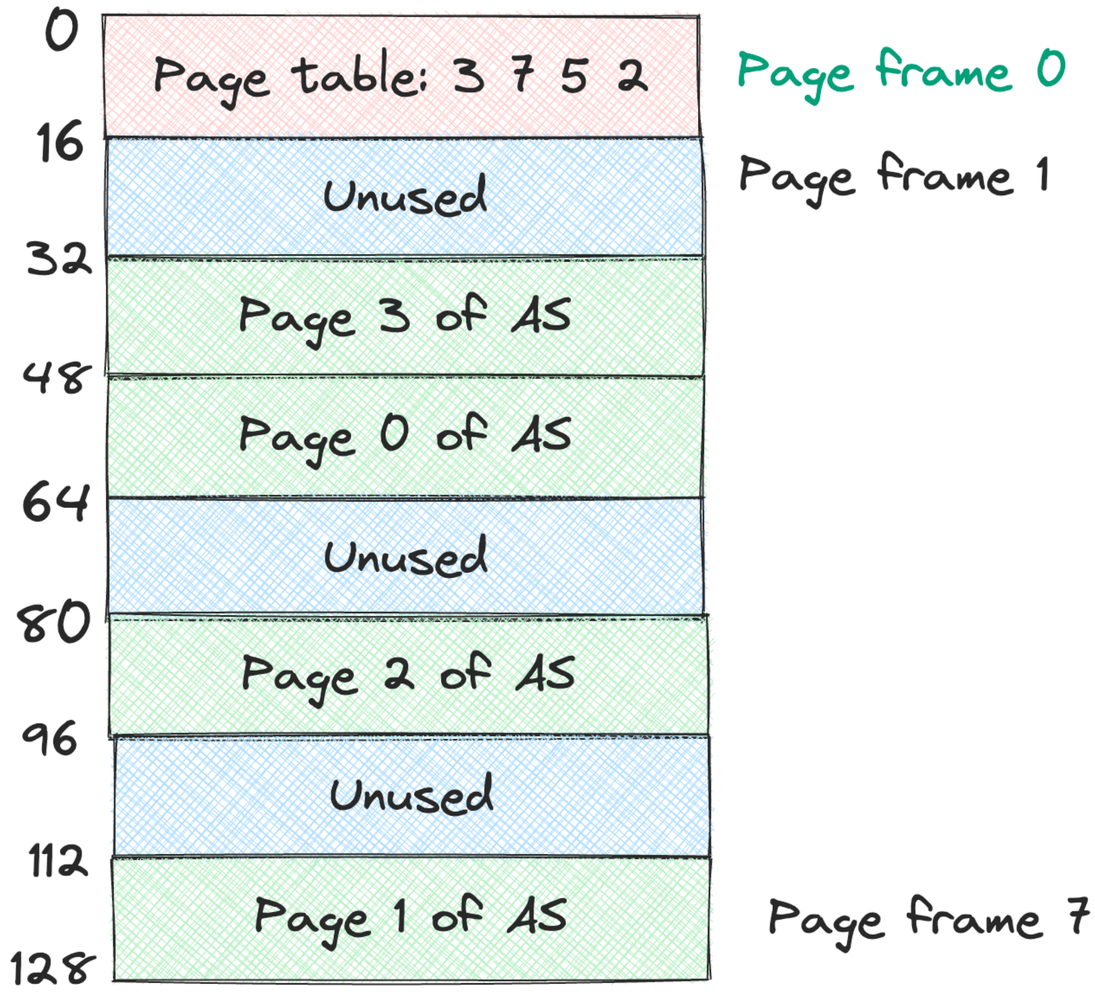
- Offset stays the same
- Its all about VPN -> PFN
- Page table helps!

Location 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!**

Storing Page Table

- Page tables can be huge
- Storing in MMU or any hardware register may be hard
- Instead, stored in-memory
- Assume that page table lives in Physical memory that OS handles



Contents of Page Table

- Page Table: data structure to map VA to PA
  - Each entry in a page table: **Page Table Entry (PTE)**
  - Simplest form is a linear page table: Just an array
  - Array indexed with VPN
  - Identify PTE at a given index to get to the PFN
  - Each PTE consists of a number of bits

Page Table Entry

- Valid bit: Whether page is valid (unused pages will be marked invalid)
  - Any access to invalid will result in trap -> termination
- Protection bit: Whether page can be read from, written to, executed from
  - Wrong access traps into the OS
- Present bit: Indicates whether page is present in physical memory
- Dirty bit: Whether page has been modified since brought into memory
- Reference bit (accessed bit): Whether page has been accessed (recently used!)

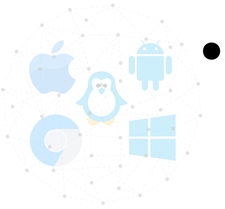
VPN	PFN	V	P	Present	Dirty	Ref

What about Efficiency of Paging

- Hardware must know where the page table is for a process
- To reach correct location => Identify VPN and offset
    1. Index into the array of PTE as pointed out by PTBR
    2. Get PTE from memory, extract PFN
    3. Add offset to PFN to get to final address
    4. Finally hardware fetches data and adds inside tax

What about Efficiency of Paging

- Page tables in memory can be too big!
  - They can also slow down things
  - mov 21 %eax
  - Translate 21 into VA into PA (117 for example)
  - Before loading, system must fetch PTE from page table
  - Perform translation and get the translated address and get desired data
  - Any issues so far?



What’s the problem?

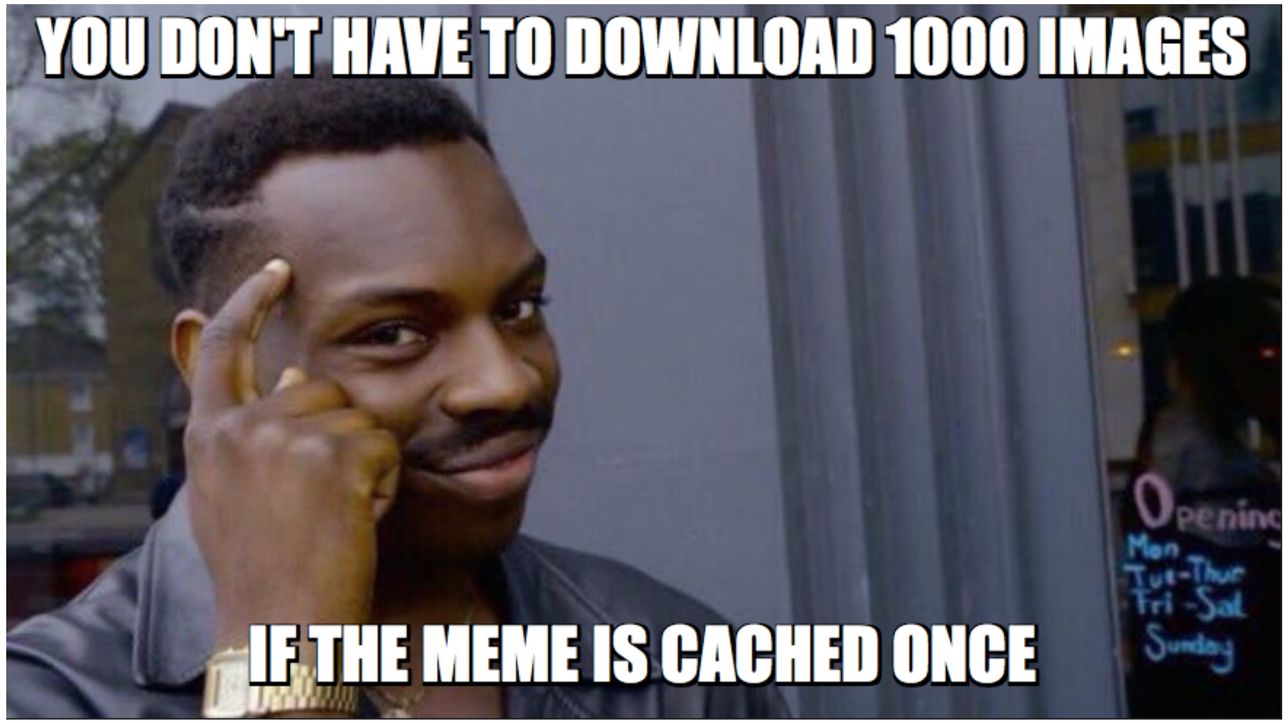
- There is one extra memory reference here - How?
- This can lead to a big slowdown
- Two main issues needs to be addressed:
  - Size of the page table is very big (Memory Overhead)
  - They can also be too slow to access (Time Overhead)
- Paging does help in solving **external fragmentation!**

Improving the speed of translation

- How to speed up translation and avoid the extra reference that paging requires?
  - What hardware support is required?
  - What OS involvement is needed?

- Let us think for a brief moment. Imagine that you are a designer! How to about this?

Why don’t we cache it?



Source: medium.com

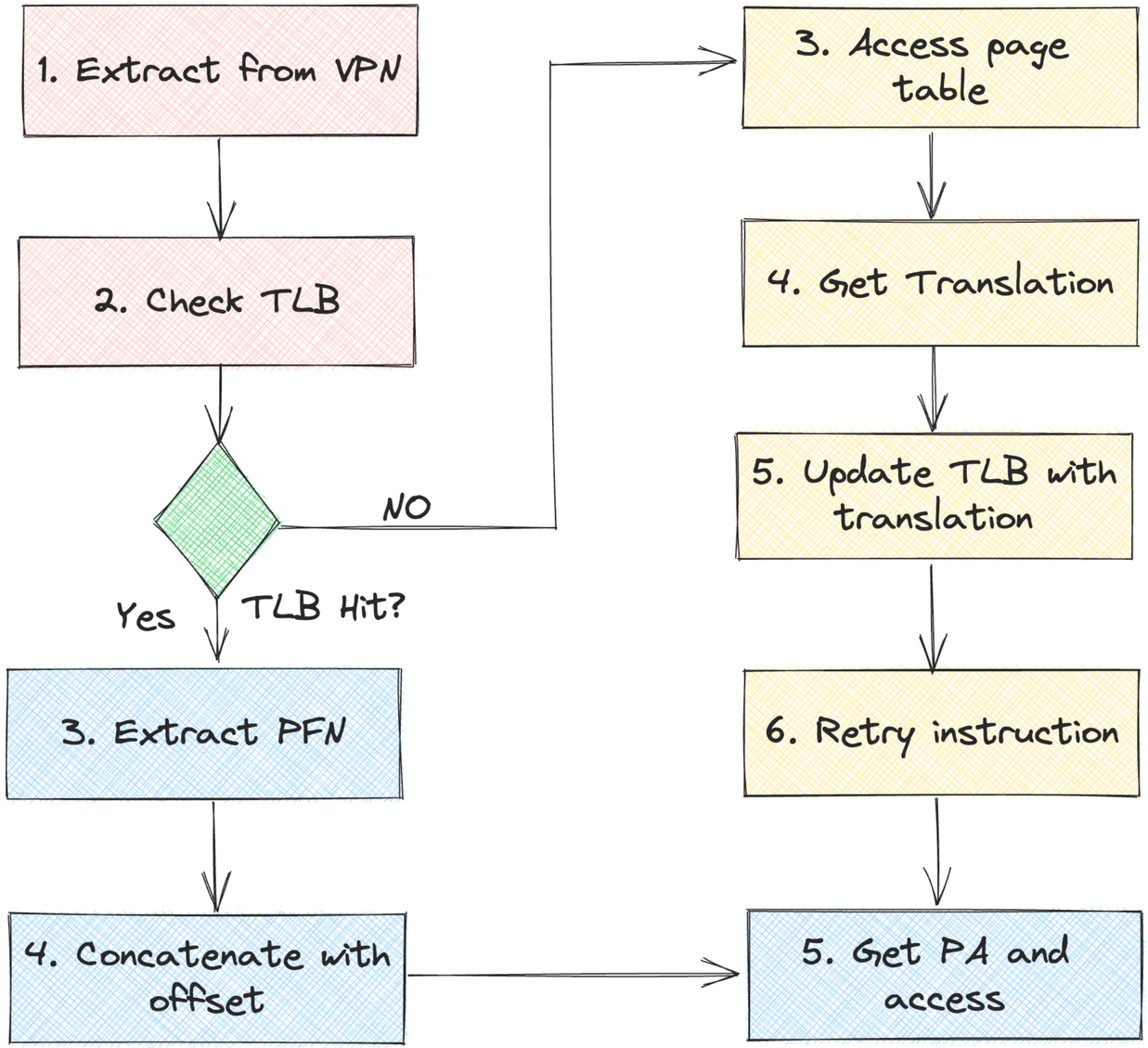
Put Hardware in action

Comes Translation Lookaside Buffer (TLB)

- Introduce translation Lookaside buffer to MMU
- To speed up address translations
  - Add TLB
  - This is similar to a hardware cache of popular VA to PA transitions
    - Address Translation Cache
  - For each translation
    - Check TLB if the page has already been accessed
    - If no, get it. If yes, use the address

TLB Control Flow Algorithm

- TLB is like all caches
- If translation found
  - Little overhead
- When there is miss
  - Cost of memory access!
- **Goal:** reduce TLB miss!



Illustrative Example: Array Access

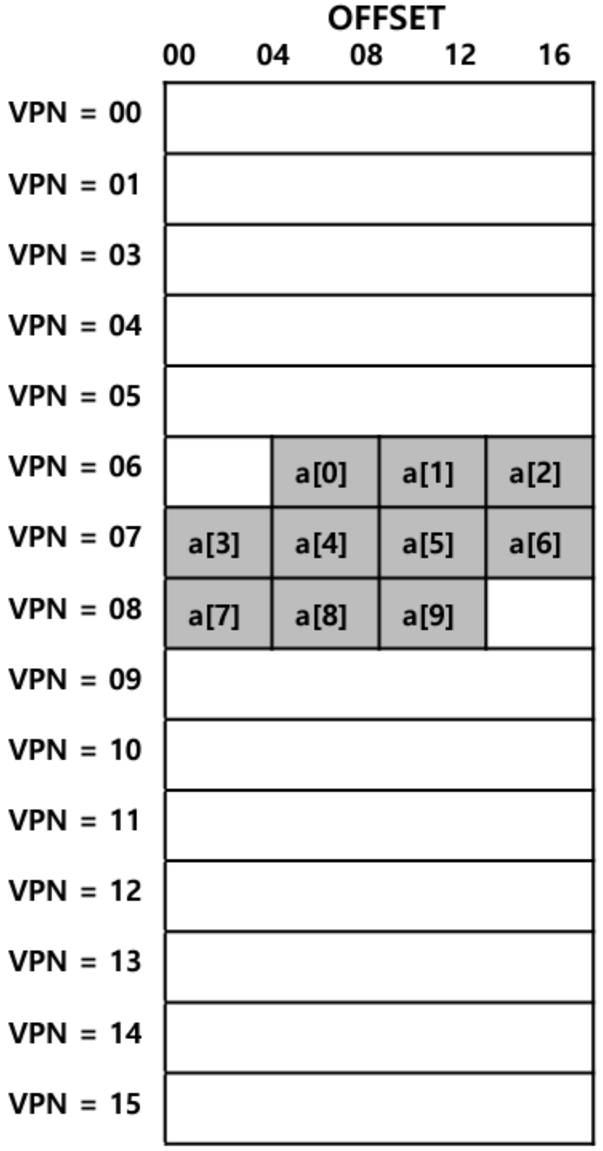
- Assume an array of 10 4-byte integers starting at VA 100. Assume an 8-bit VA space with 16 byte pages
  - This implies 4 bits for offset and 4 bits for VPN
  - Each page is 2^4 = 16 byte size
- How shall the virtual address space be organized?

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;
}
```

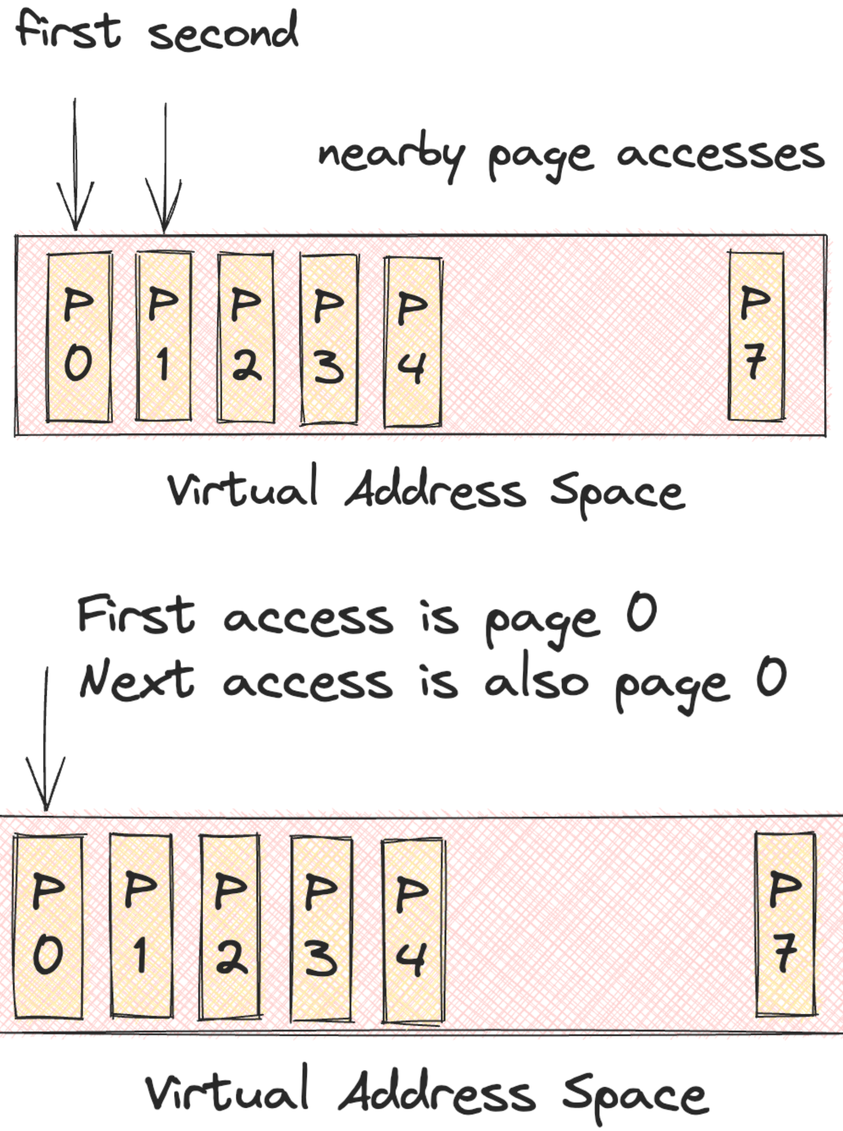
Illustrative Example: Array Access

- 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

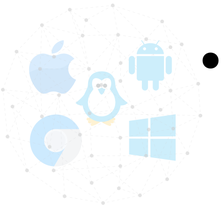
Handling TLB Miss?

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!!

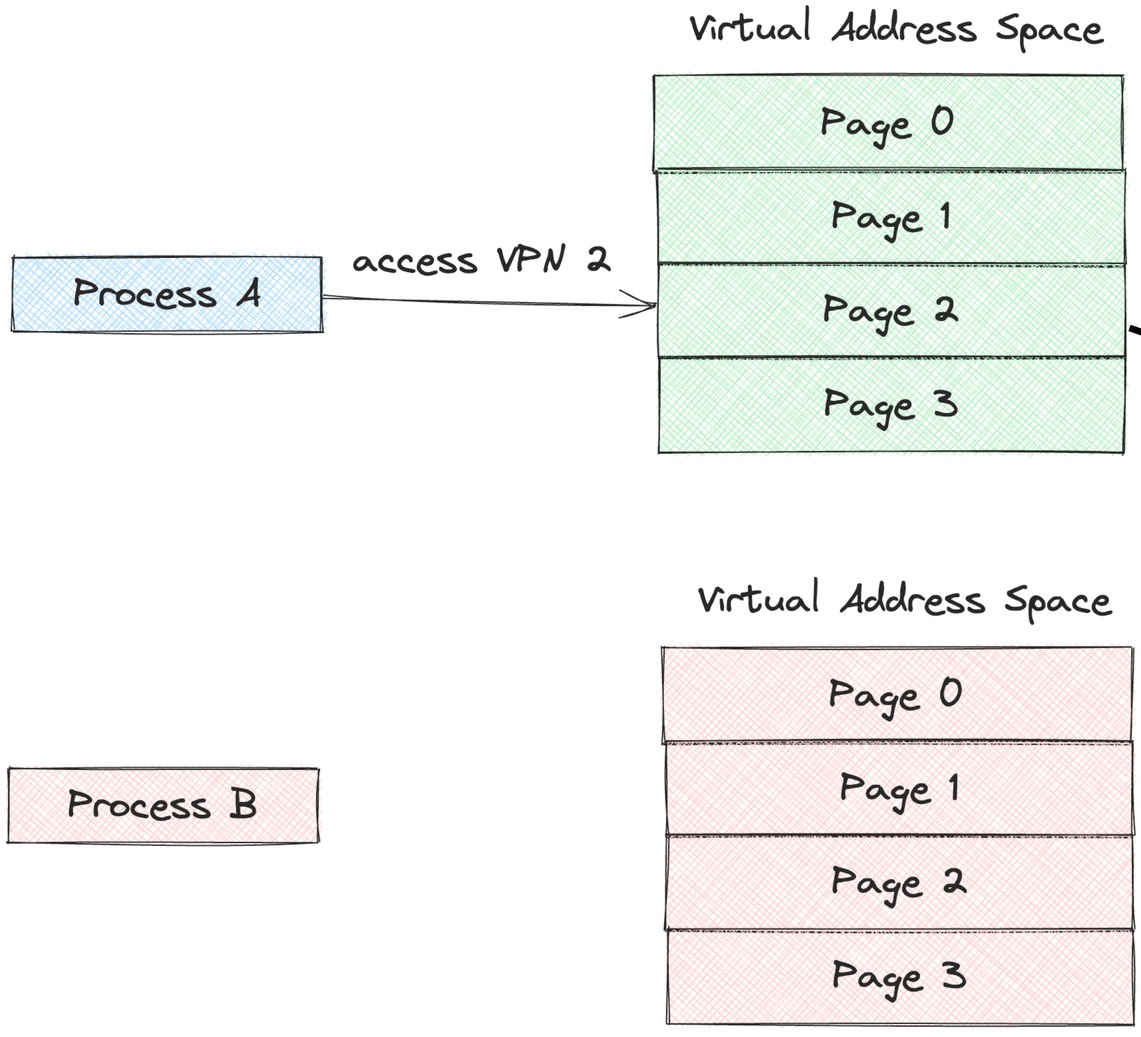
- When returning from TLB miss
  - Hardware must save the PC of current instruction and again execute the same instruction for **retry** resulting in hit
  - In usual scenario the return-from-trap goes to next instruction
- OS should not go into infinite chain of TLB misses
  - TLB miss handlers are 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?

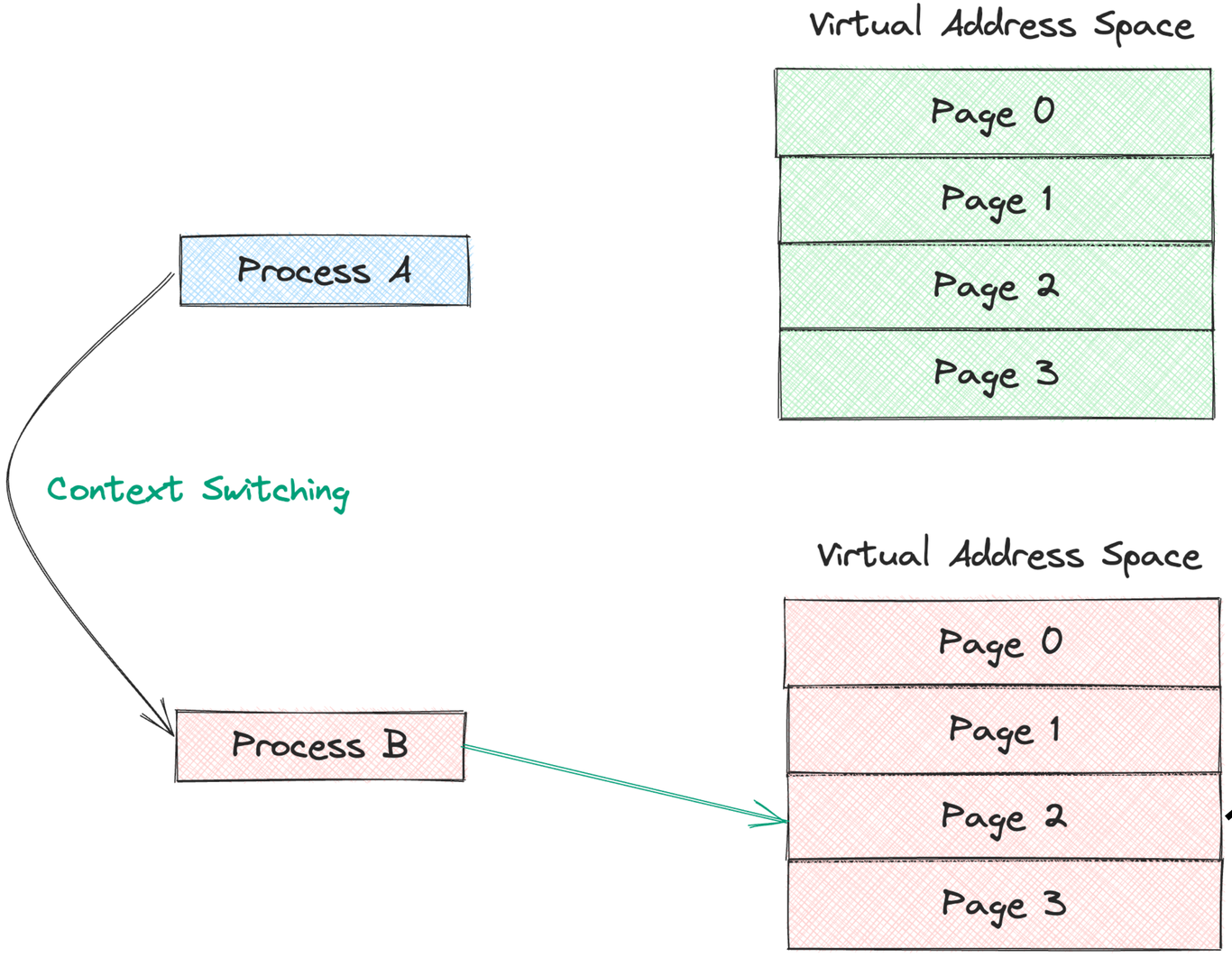


Insert TLB Entry

TLB Table

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

Context Switching and TLB



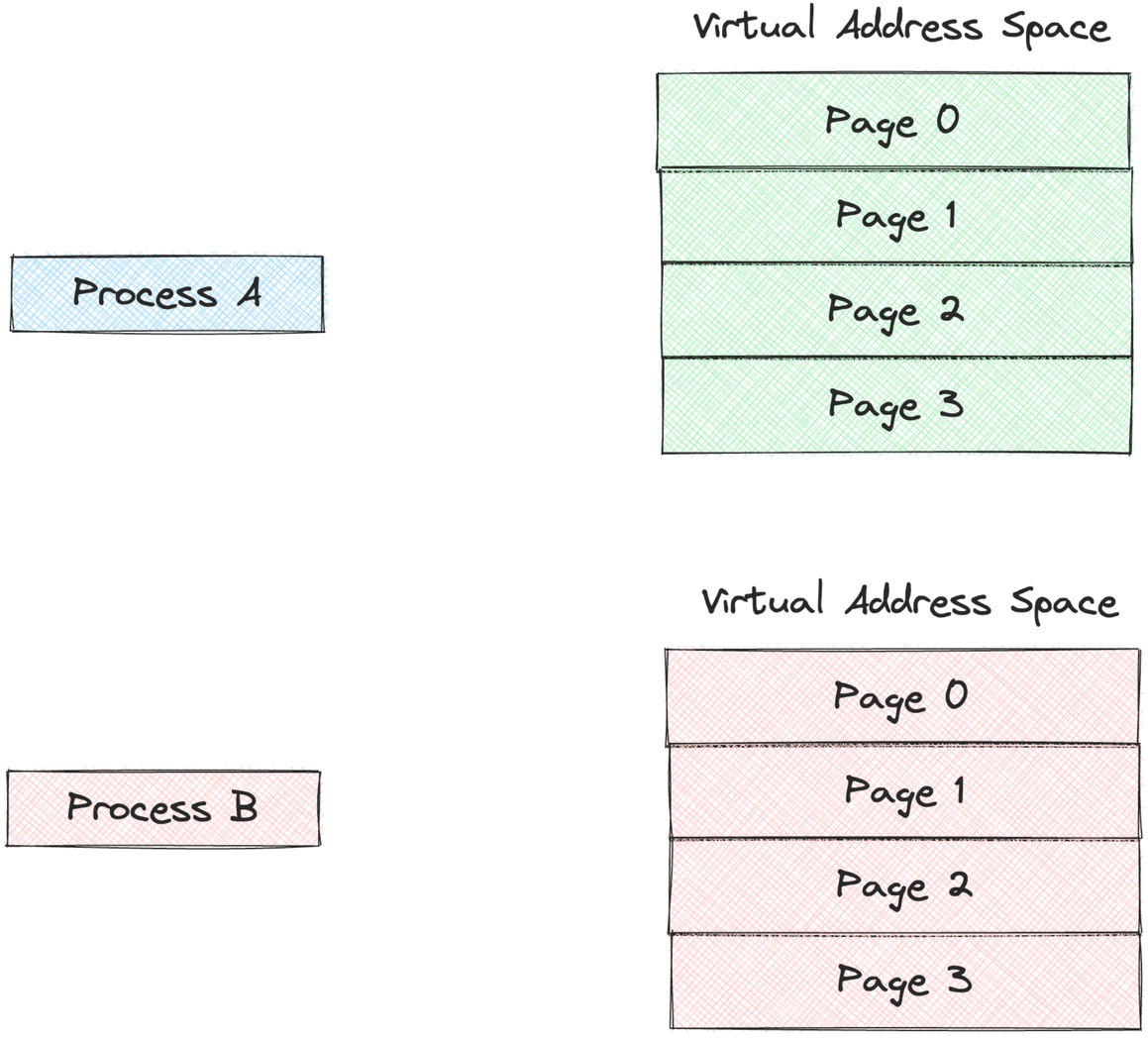
Flushing can be an option but… TLB miss?

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?

Thank you

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