

NEU CS 3650 Computer Systems

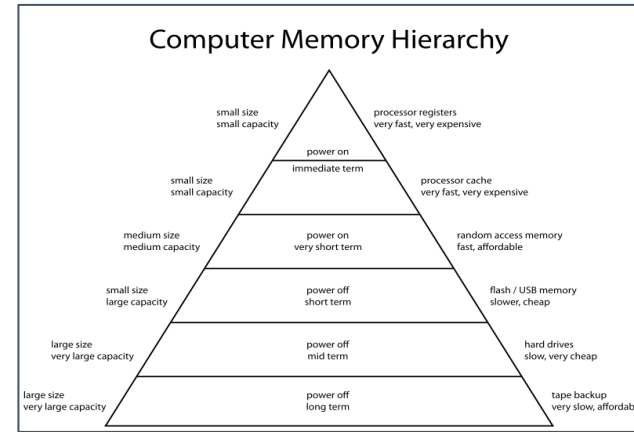
Instructor: Dr. Ziming Zhao

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Alden Jackson, Ben Weintraub, Gene Cooperman, Peter Desnoyers' lecture slides for the same course.

An Introduction to Caches

Cache

- Cache
 - A smaller, faster storage device than the layer below
 - A staging area for a subset of the data in a larger, slower device
- For each level in the memory hierarchy K
 - K serves as a cache for the larger slower device at level K+1
- A memory hierarchy works because of locality
 - Programs access data at level K more often than data at K+1
 - With this, we can hold a lot of data at lower levels, and still access data at high speeds using higher level caches



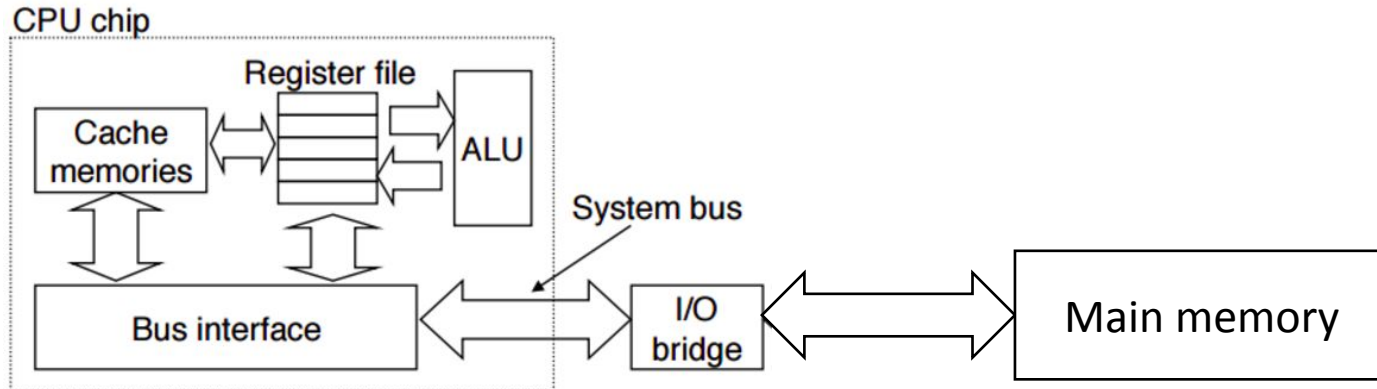
Caches are everywhere!

- Registers (Instruction Cache)
- L1 cache
- L2 cache
- Translation Lookaside Buffer (TLB)
- **Virtual Memory**
- Disk Cache
- Network buffer cache
- Browser cache
- Web Cache, CDNs, ...

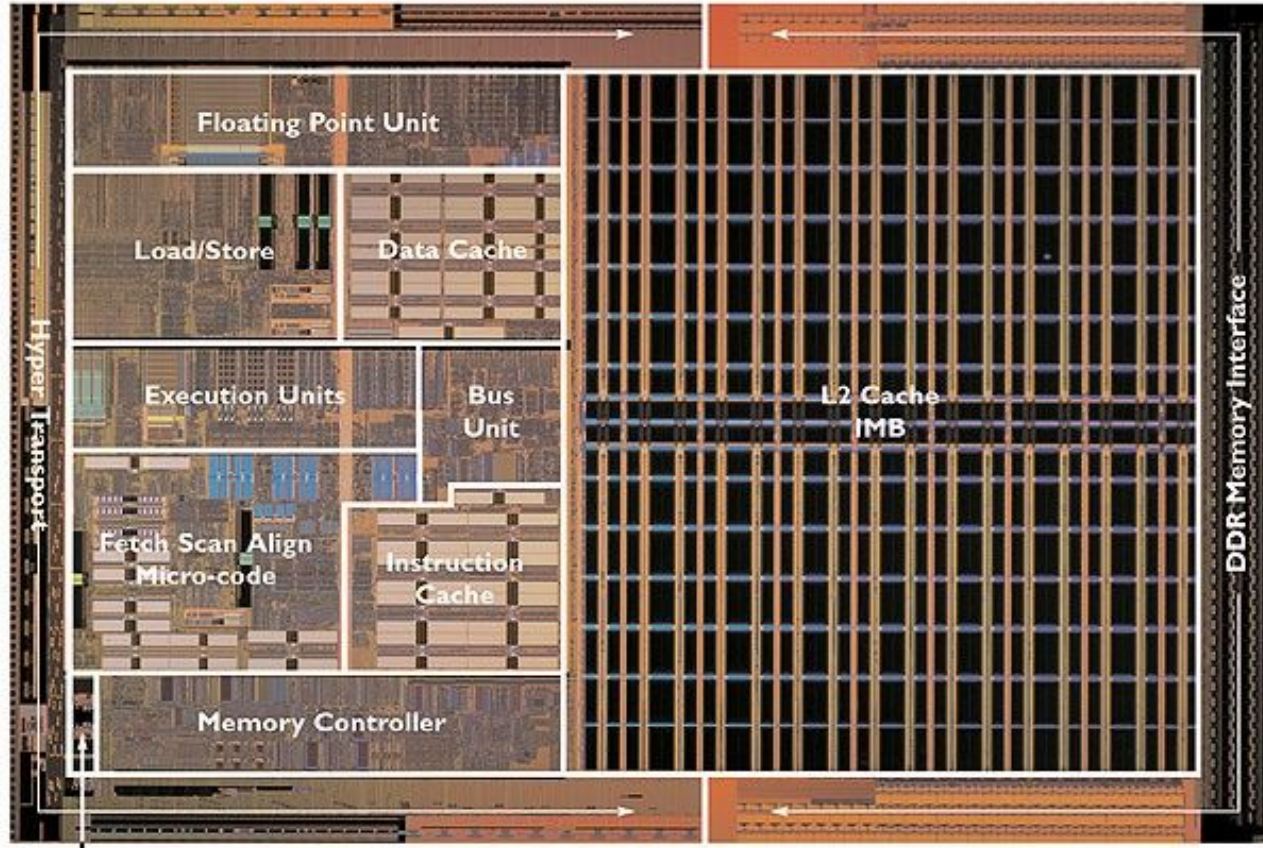
Cache on Hardware

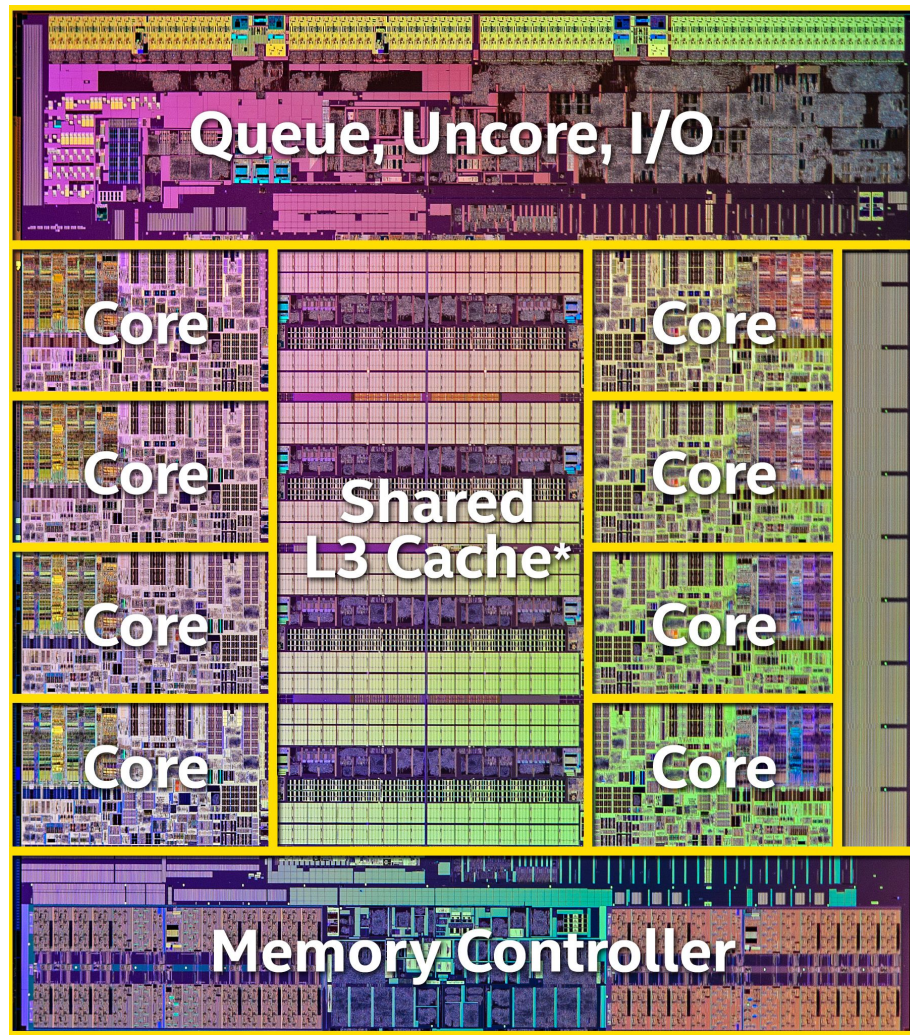
- CPU will look for data in Cache first
 - Attempt to load into registers
 - If not found, then will travel on System Bus -> I/O Bridge -> then to main memory

```
mov rax, [rbx + 0x10]
```



Cache Hierarchy

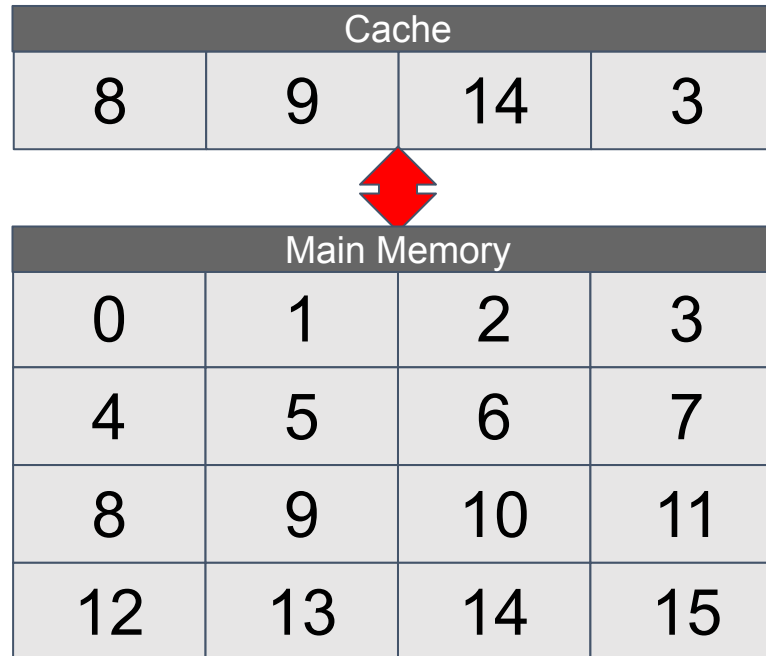




General Cache Concepts

Small Example

- Cache keeps a copy of data from main memory



Cache hit and misses

- Cache Hit
 - Data is requested and it is in the cache
- Cache Miss
 - Data is not in the cache and must be fetched from main memory
- So ideally, we want lots of cache hits!
 - We want to take advantage of these faster memory accesses!
 - This may also be a good metric to quantify locality of our programs.

Cache Hit

8

Load 8 - 8 is in the cache this is good!

Cache

8

9

14

3



Main Memory

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

Cache Miss

0

Load 0 - 0 is not in the cache!

Cache

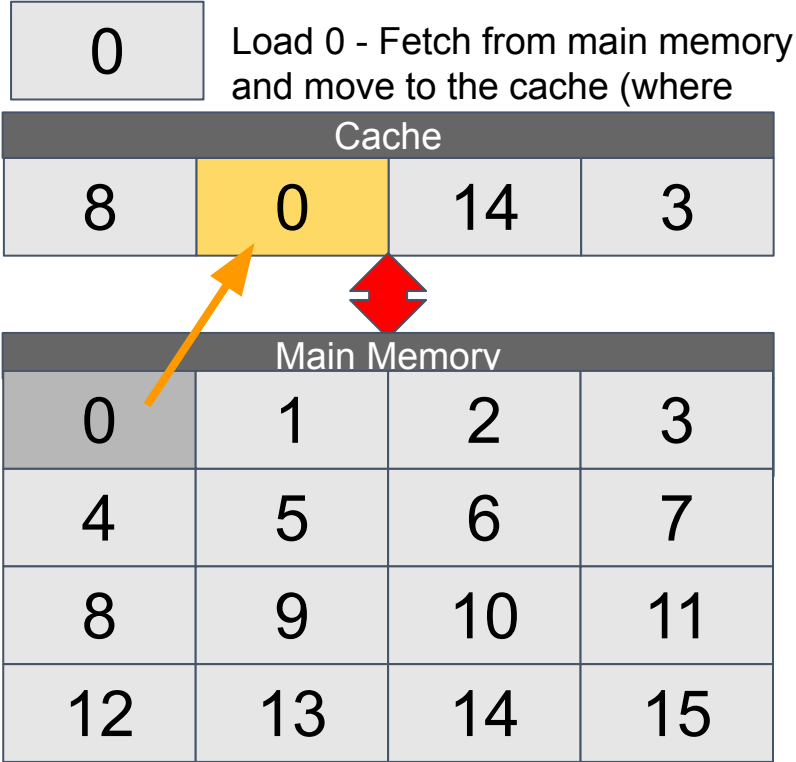
8	9	14	3
---	---	----	---



Main Memory

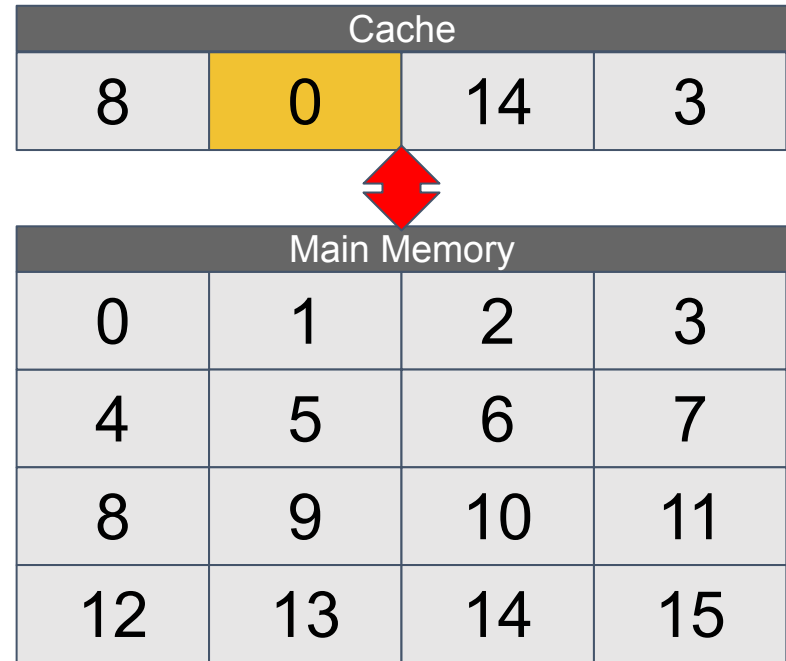
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Cache Miss



Note on Fetching

- It is almost always worthwhile to put data from memory into cache
- Memory access latency $\gg$ cache access latency
 - Memory access is over 10X slower
- The exact algorithm on how to replace and remove items depends on your policy.



Policies

- Now how I choose where to put that block is based on:
 - Placement Policy
 - Determine where blocks of memory go in the cache
 - Replacement Policy
 - Determines which block gets evicted when we run out of room.
- These policies in general are very simple! We usually do not want a complicated scheme that takes more processing power!

Sample Replacement Policies

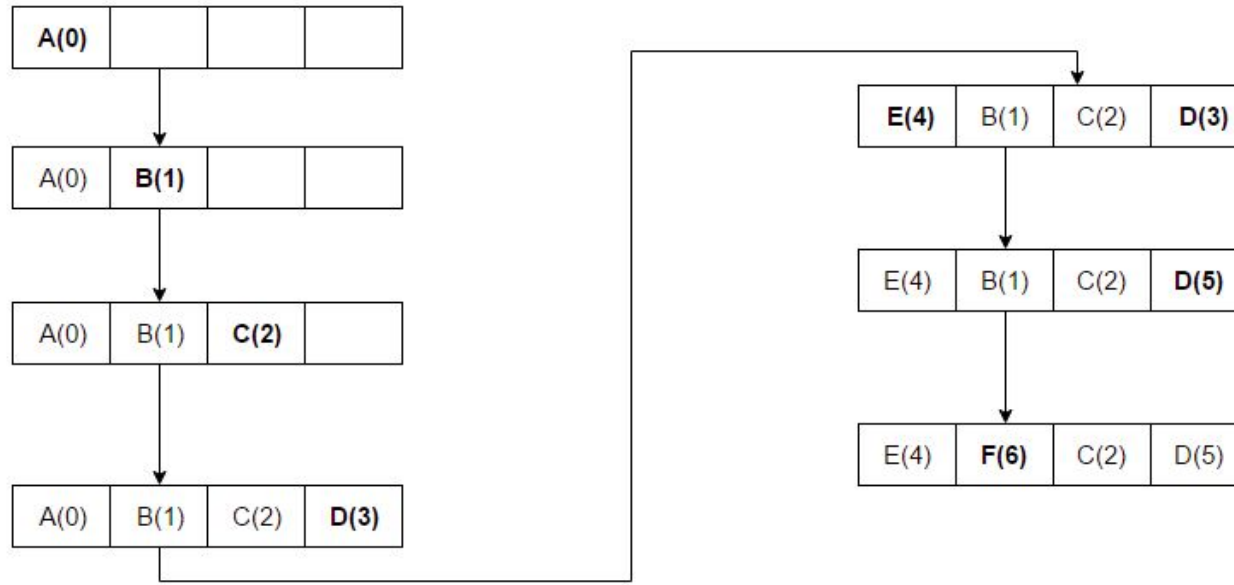
- Random - Just randomly remove something
- Least Recently Used (LRU) - Move out the youngest item.
- Here are some more:
 - https://en.wikipedia.org/wiki/Cache_replacement_policies

2 Policies

- 2.1 Bélády's Algorithm
- 2.2 First In First Out (FIFO)
- 2.3 Last In First Out (LIFO)
- 2.4 Least Recently Used (LRU)
- 2.5 Time aware Least Recently Used (TLRU)^[5]
- 2.6 Most Recently Used (MRU)
- 2.7 Pseudo-LRU (PLRU)
- 2.8 Random Replacement (RR)
- 2.9 Segmented LRU (SLRU)
- 2.10 Least-Frequently Used (LFU)
- 2.11 Least Frequent Recently Used (LFRU) ^[11]
- 2.12 LFU with Dynamic Aging (LFUDA)
- 2.13 Low Inter-reference Recency Set (LIRS)
- 2.14 Adaptive Replacement Cache (ARC)
- 2.15 Clock with Adaptive Replacement (CAR)
- 2.16 Multi Queue (MQ) caching algorithm|Multi Queue (MQ)
- 2.17 Pannier: Container-based caching algorithm for compound objects

LRU Example | A-D added, ()'s represent age bit

LRU = Least Recently Used (Item with youngest age)



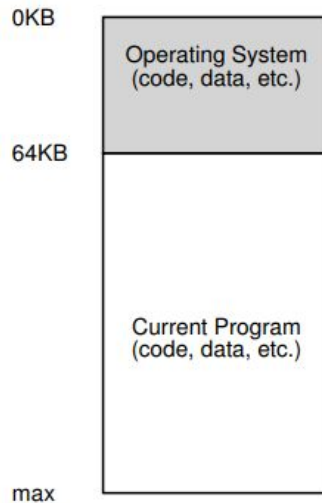
Cache Misses

- Cold (Compulsory) Miss
 - First time you access a cache (e.g., when you start a program)
- Capacity Miss
 - Set of the things you want to keep is larger than the cache size
- Conflict Miss
 - Cache is large enough, but multiple data map to the same block.
 - E.g., placement/alignment of data prevents different data to coexist

Virtual Memory and Memory Management Unit (MMU)

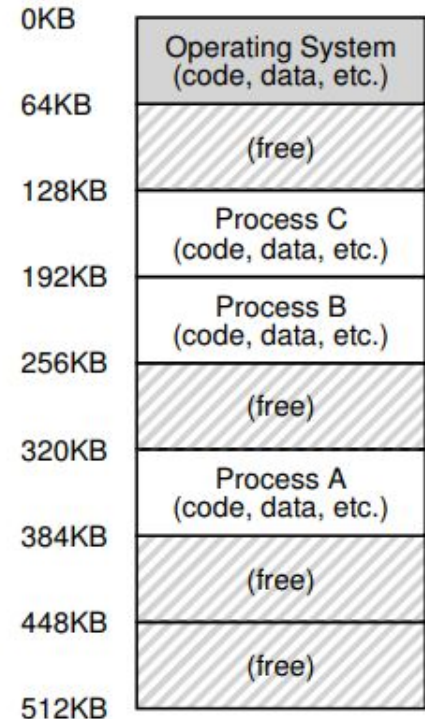
Early Computers

- Computers historically were really good at just doing one thing
- So a computer's memory stored the operating system and whatever program was currently running in memory



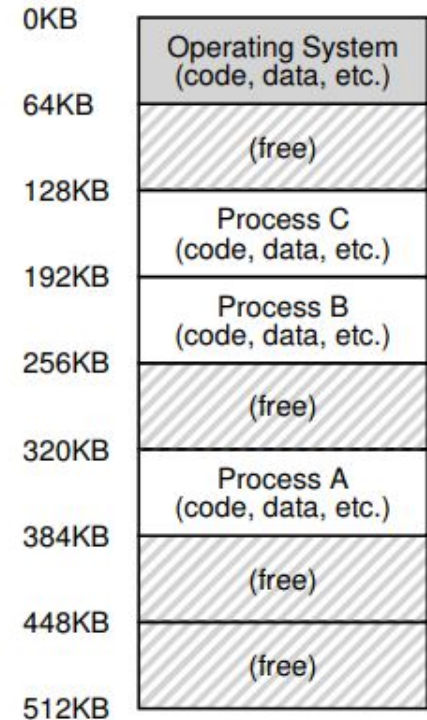
Sharing Physical Memory

- Later computer operators wanted to run more than one program at a time
- So as memory expanded, multiple processes could be loaded into fixed size chunks to run.
- And we have talked about how processes context switch and make this possible.



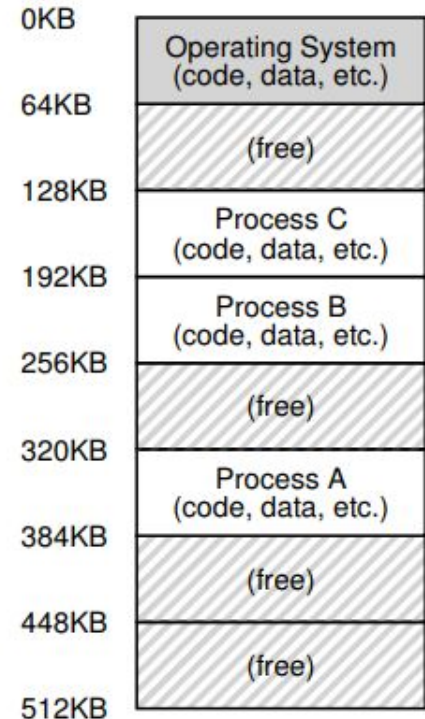
More efficient memory

- Eventually, programmers did not want to have a “fixed” size memory block.
 - Maybe one process needed more or less memory than the other
- Also memory size was limited but wanted to run more programs



More efficient memory

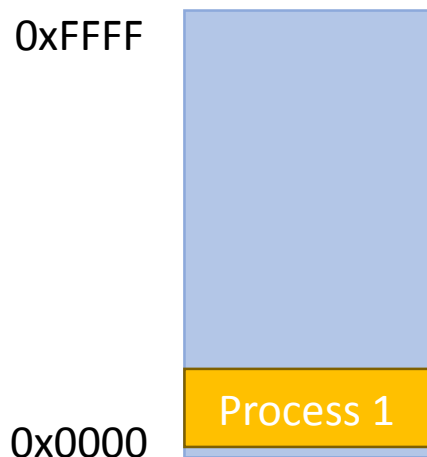
- Eventually, programmers did not want to have a “fixed” size memory block.
 - Maybe one process needed more or less memory than the other
- Also memory size was limited but wanted to run more programs
- How can we enable flexibly-sized processes?
 - Virtual memory could be the solution



Virtual Memory concept

- What do we mean by **virtual memory**?
 - Processes use **virtual** (or **logical**) addresses
 - Virtual addresses are translated to physical addresses

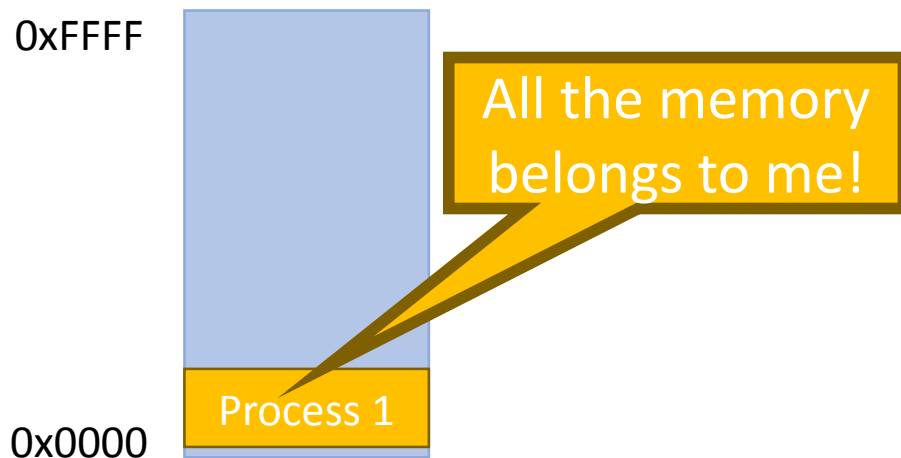
Process' View of Virtual Memory



Virtual Memory concept

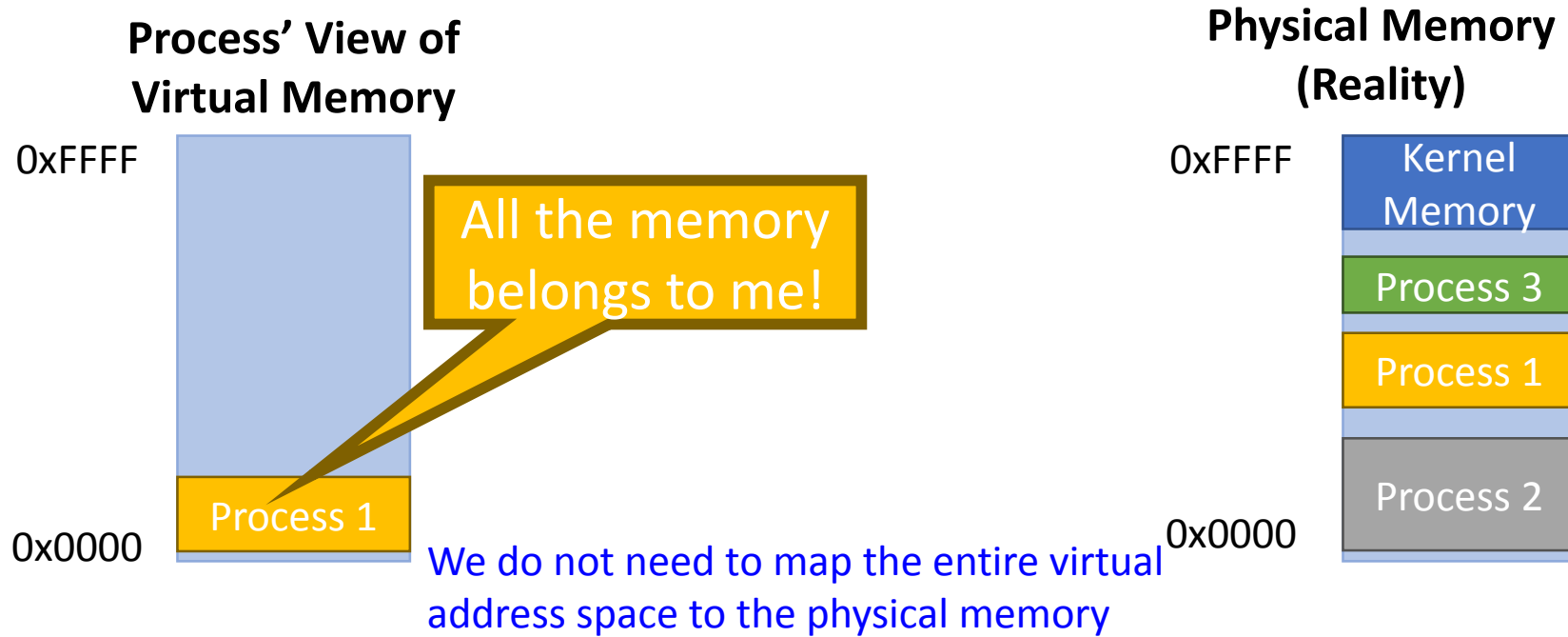
- What do we mean by **virtual memory**?
 - Processes use **virtual** (or **logical**) addresses
 - Virtual addresses are translated to physical addresses

Process' View of Virtual Memory



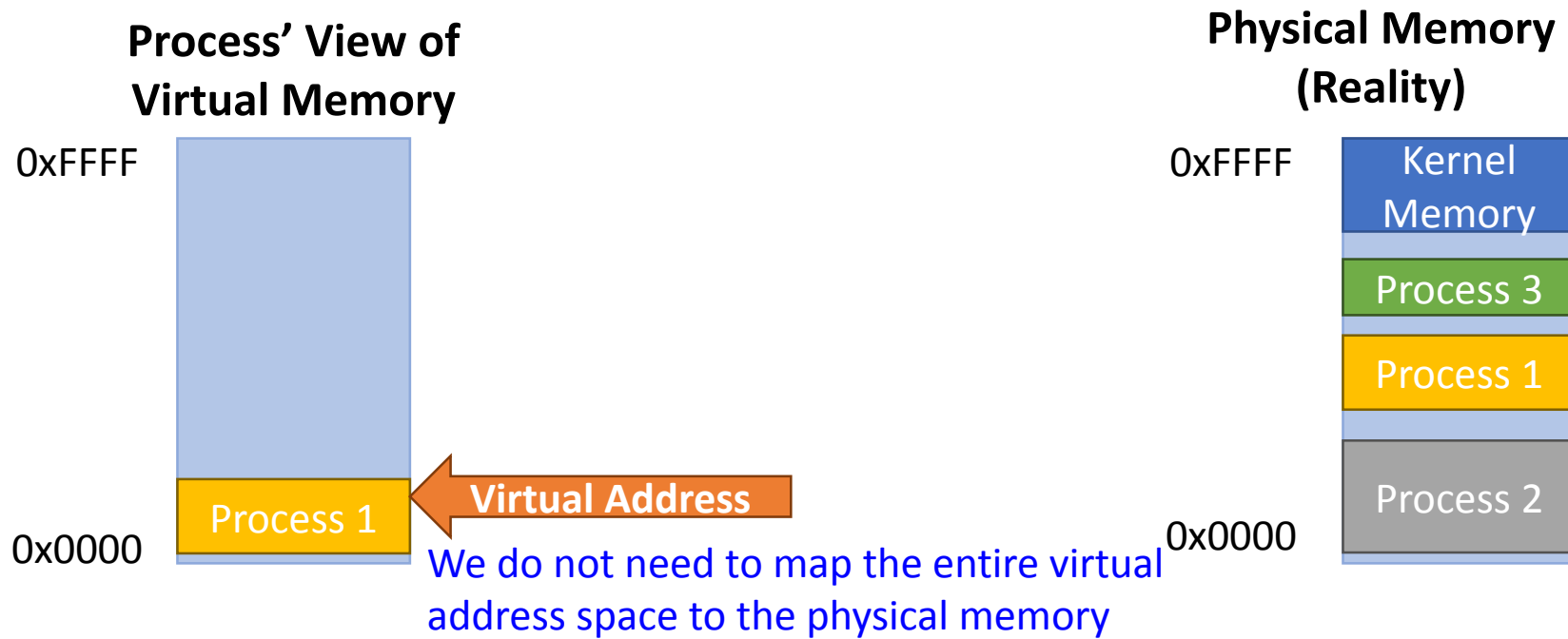
Virtual Memory concept

- What do we mean by **virtual memory**?
 - Processes use **virtual** (or **logical**) addresses
 - Virtual addresses are translated to physical addresses



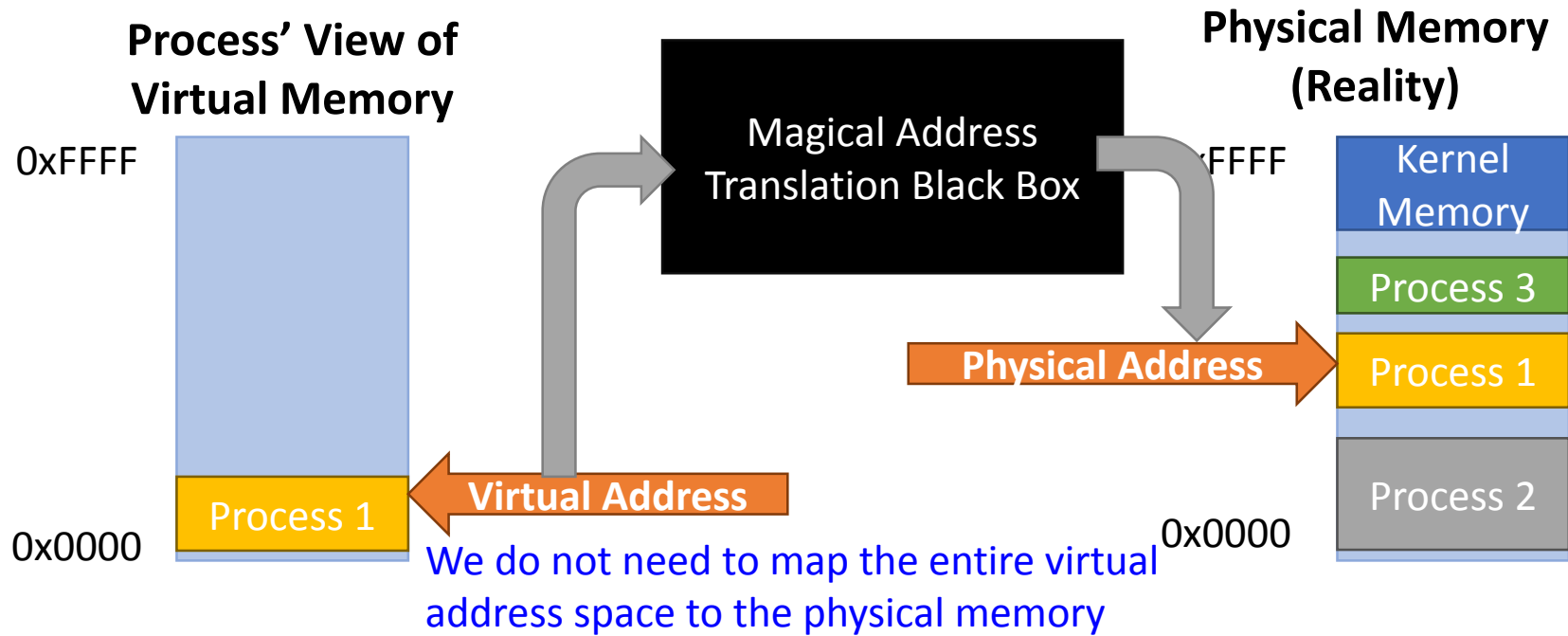
Virtual Memory concept

- What do we mean by **virtual memory**?
 - Processes use **virtual** (or **logical**) addresses
 - Virtual addresses are translated to physical addresses



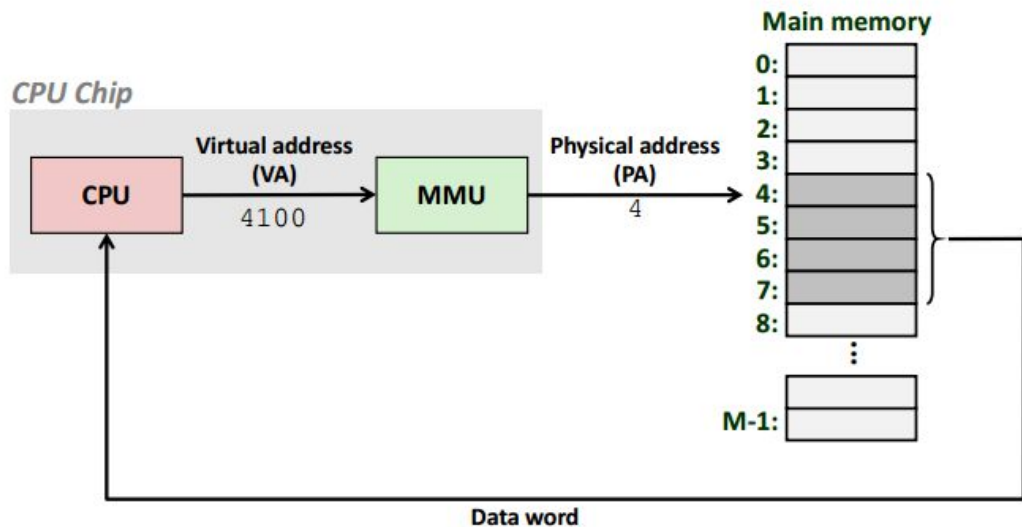
Virtual Memory concept

- What do we mean by **virtual memory**?
 - Processes use **virtual** (or **logical**) addresses
 - Virtual addresses are translated to physical addresses



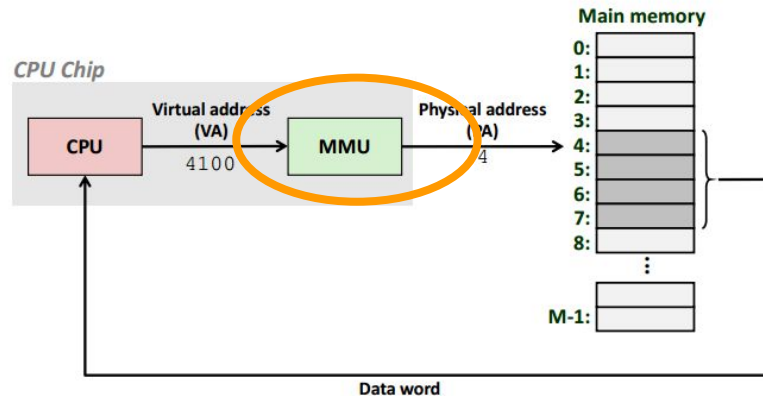
Introducing the Memory Management Unit (MMU)

- We still retrieve memory from main memory
- BUT, there is an additional translation step that occurs in the Memory Management Unit (MMU)

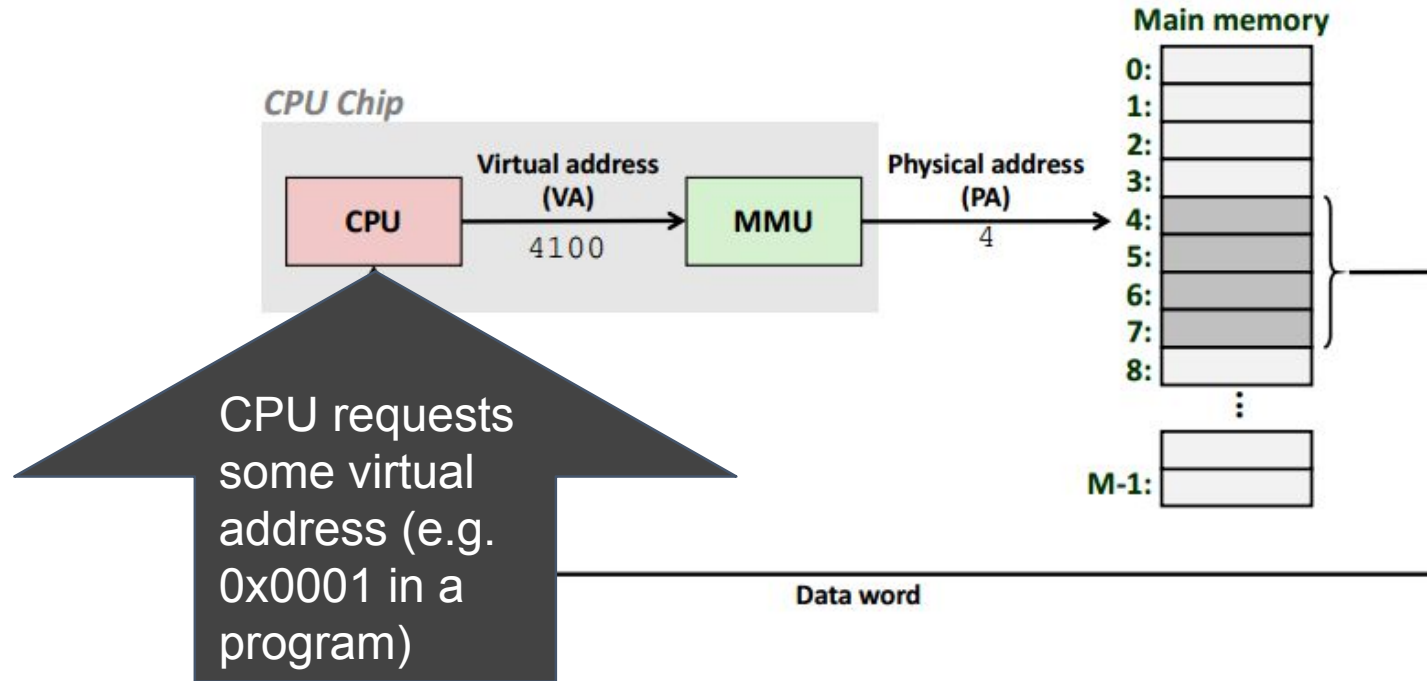


Memory Management Unit (MMU)

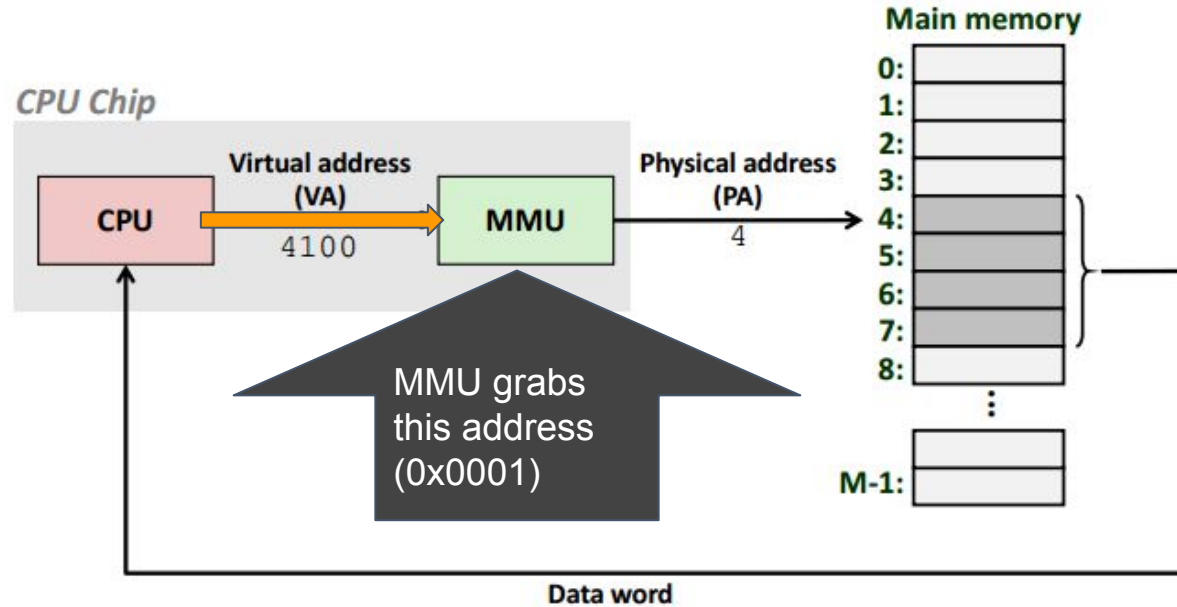
- MMU's job is to figure out (i.e. translate) the mappings from virtual memory address to physical memory address
- MMU moves memory in units called 'pages'
 - A page size varies by architecture and configuration settings
 - A common page size 4096 bytes (i.e. 4KB)



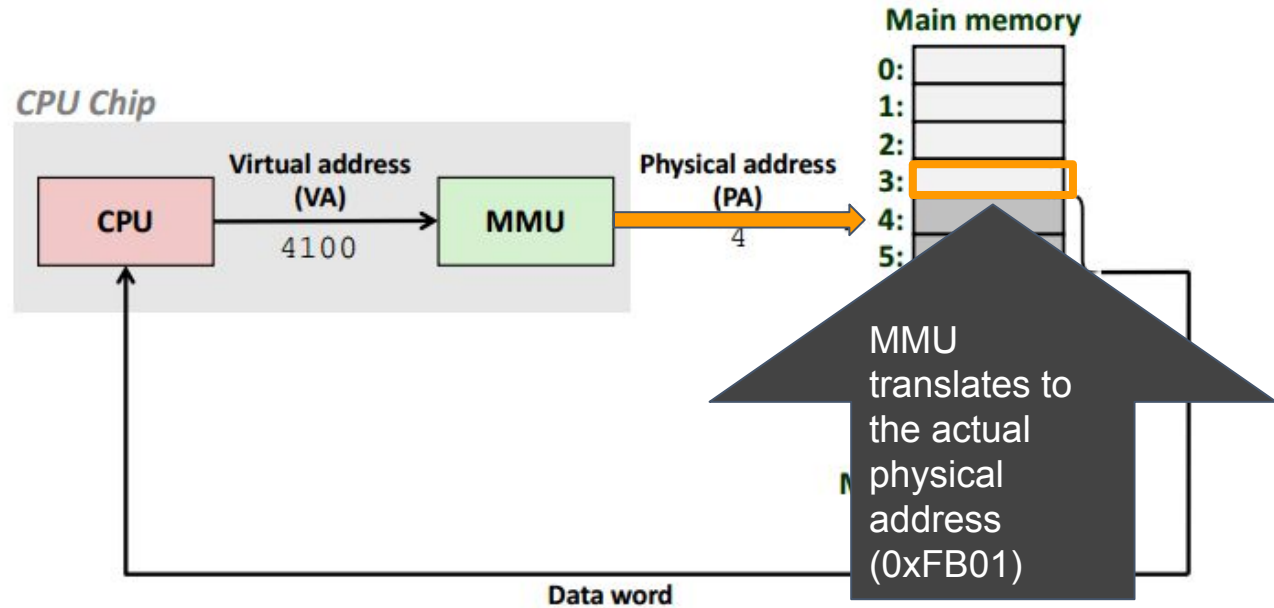
Memory Management Unit (MMU)



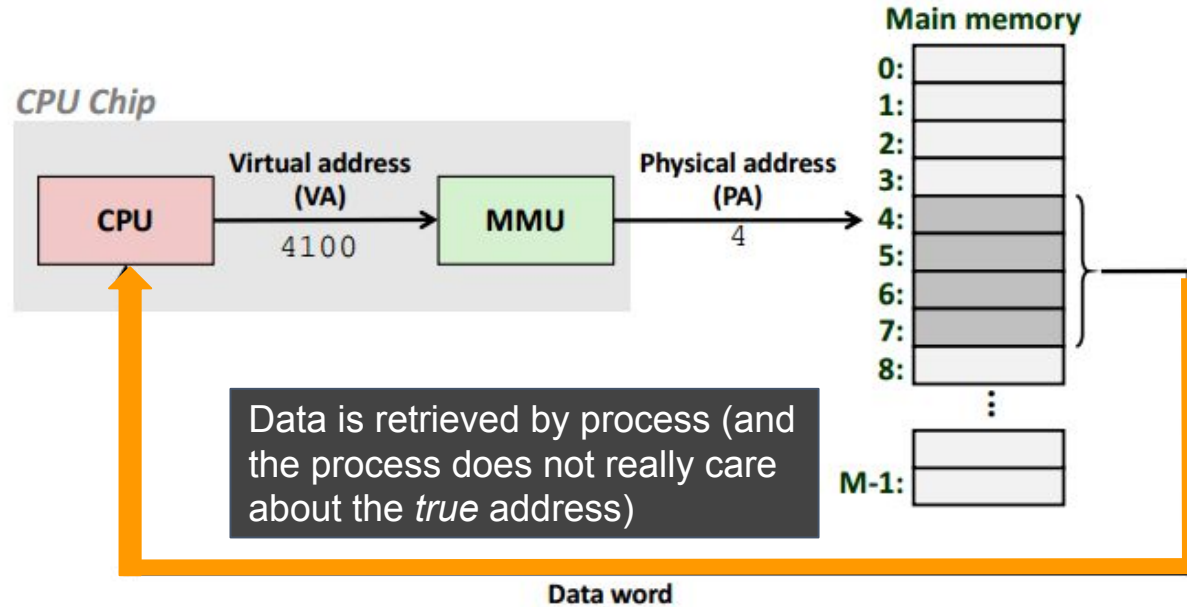
Memory Management Unit (MMU)



Memory Management Unit (MMU)



Memory Management Unit (MMU)



Virtual Memory

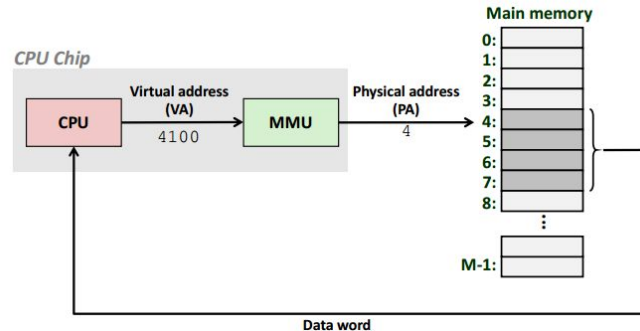
Three Virtual Memory Advantages

1. Uses main memory efficiently
2. Simplifies memory management (for application developers)
3. Isolates address spaces

Why Virtual Memory (1/3)

1. Uses main memory efficiently

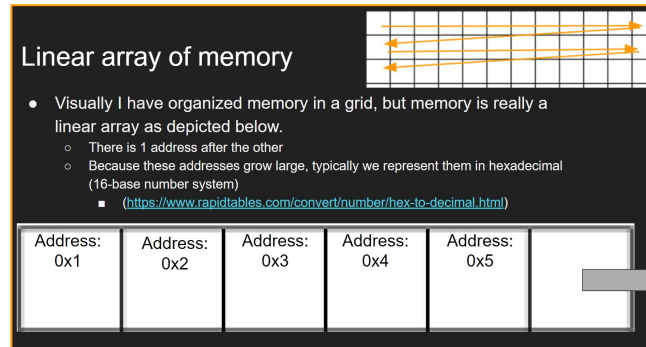
- Use physical memory as a “cache” for parts of a virtual address space
- Not all data in the virtual address space may be mapped to physical memory and some may be in disk



Why Virtual Memory (2/3)

2. Simplifies memory management (for application developers)

- Each process gets the same linear address space
 - This is how we have always thought of memory at this point
 - Our programs each have a simple linear address space
 - This is also (arguably) easier for the Operating System to manage



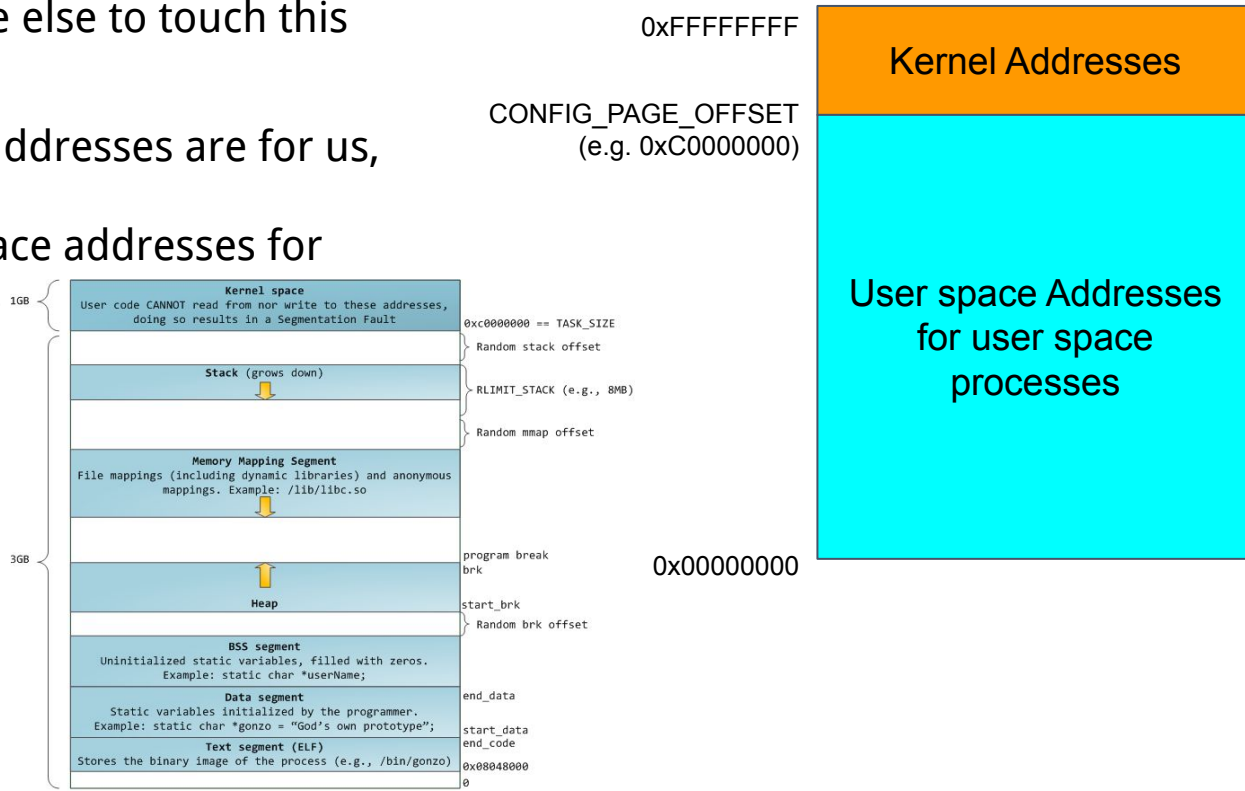
Why Virtual Memory (3/3)

3. Isolates Address Spaces

- A process is sandboxed in the virtual address space
 - One process cannot interfere with another
 - User's program cannot access kernel information and code.
- We do not need to memorize specific addresses
 - (e.g. where some device that is plugged in is located versus some other memory)

So here's another high level view

- The kernel gets a large chunk of memory
 - Roughly the top 1-2 GB of virtual address space for Linux.
 - We don't want anyone else to touch this space.
- But the rest of the virtual addresses are for us, the users.
 - We call these user space addresses for user space processes.



#1 Use Main Memory efficiently

Some terminology for Address Spaces (1/2)

- We refer to a Linear Address Space as
 - Order of contiguous non-negative integer addresses
 - $\{0,1,2,3,\dots\}$
- A 'page' of memory is some fixed size
 - Typically 4096 bytes (4KB)

Some terminology for Address Spaces (2/2)

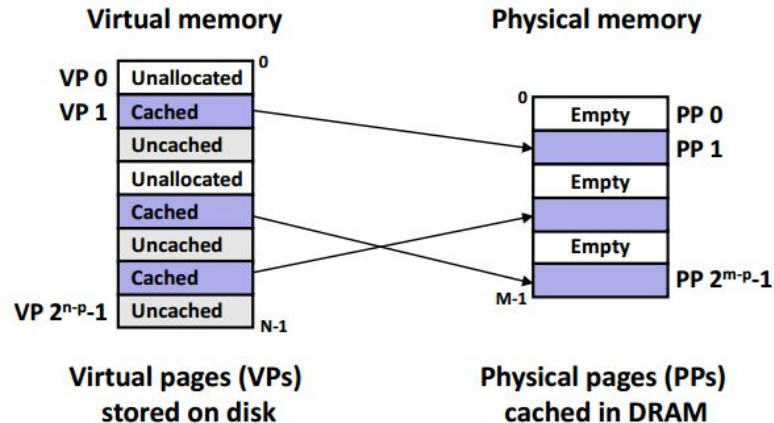
- Virtual address space:
 - Set of $N = 2^n$ virtual addresses
 - $\{0, 1, 2, 3, \dots, N-1\}$
- Physical Address Space
 - Set of $M = 2^m$ physical addresses
 - $\{0, 1, 2, 3, \dots, M-1\}$
- Okay, so this means we really have 2 memory addresses spaces to keep track of: Virtual and Physical

Two Address Spaces

- Physical Address Space
 - Is used by the hardware
- Virtual Addresses Space
 - Used by the software
 - Again, this is what we are familiar with
 - The exact translation happens in hardware for us by the MMU

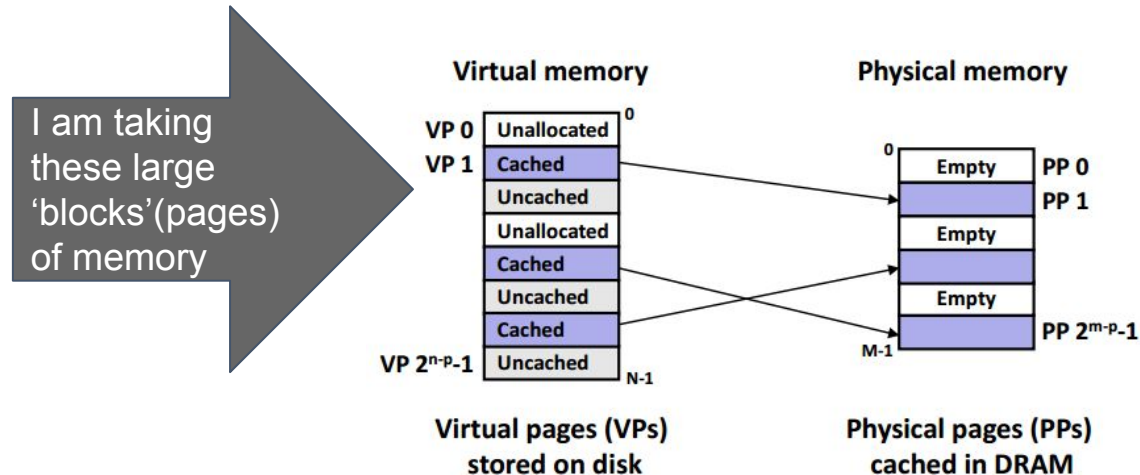
Virtual Memory to assist with caching (1/5)

- **Conceptually**, virtual memory is an array of contiguous bytes stored on disk (and memory pages indeed gets swapped out to disk)
- The contents of these arrays are cached in physical memory



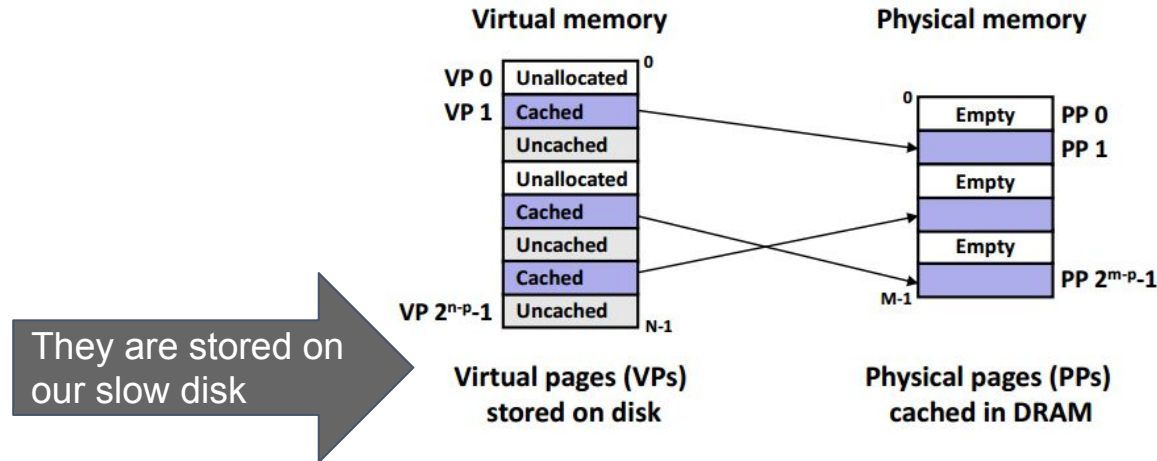
Virtual Memory to assist with caching (2/5)

- **Conceptually**, virtual memory is an array of contiguous bytes stored on disk (and memory pages indeed gets swapped out to disk)
- The contents of these arrays are cached in physical memory



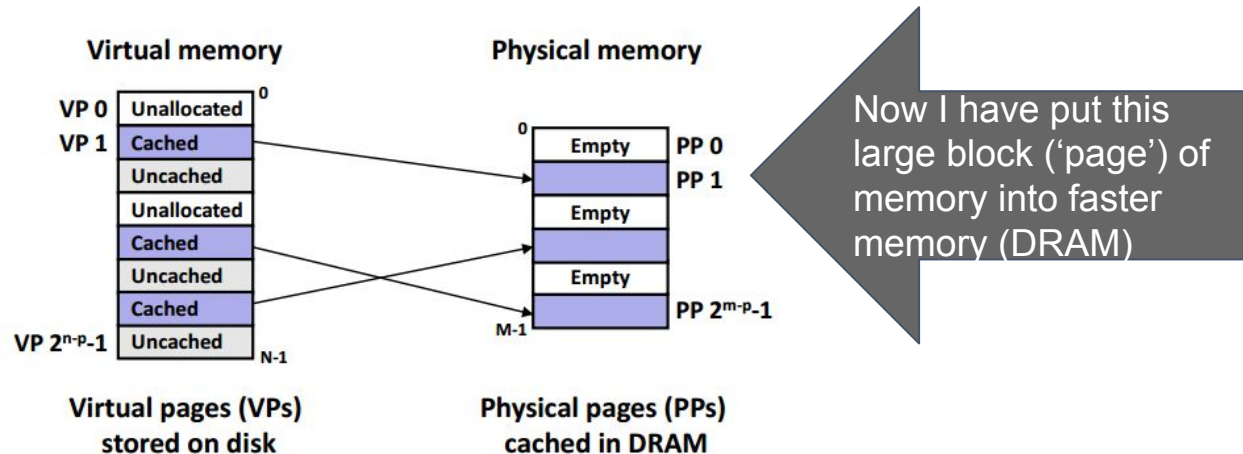
Virtual Memory to assist with caching (3/5)

- **Conceptually**, virtual memory is an array of contiguous bytes stored on disk (and memory pages indeed gets swapped out to disk)
- The contents of these arrays are cached in physical memory



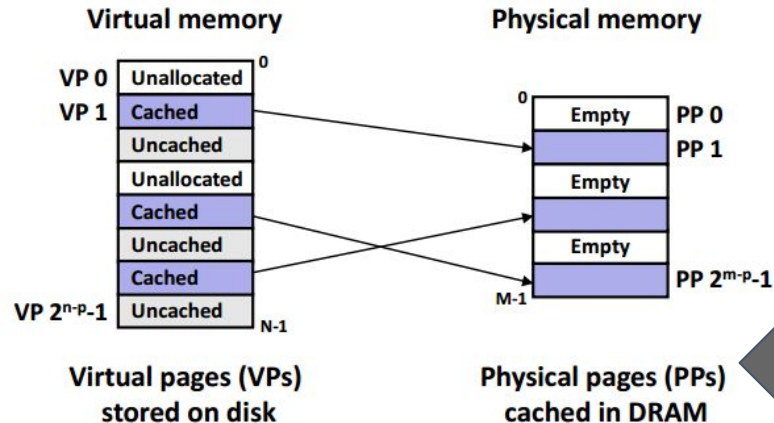
Virtual Memory to assist with caching (4/5)

- **Conceptually**, virtual memory is an array of contiguous bytes stored on disk (and memory pages indeed gets swapped out to disk)
- The contents of these arrays are cached in physical memory



Virtual Memory to assist with caching (5/5)

- **Conceptually**, virtual memory is an array of contiguous bytes stored on disk (and memory pages indeed gets swapped out to disk)
- The contents of these arrays are cached in physical memory



Our DRAM is faster than disk

Swap Space

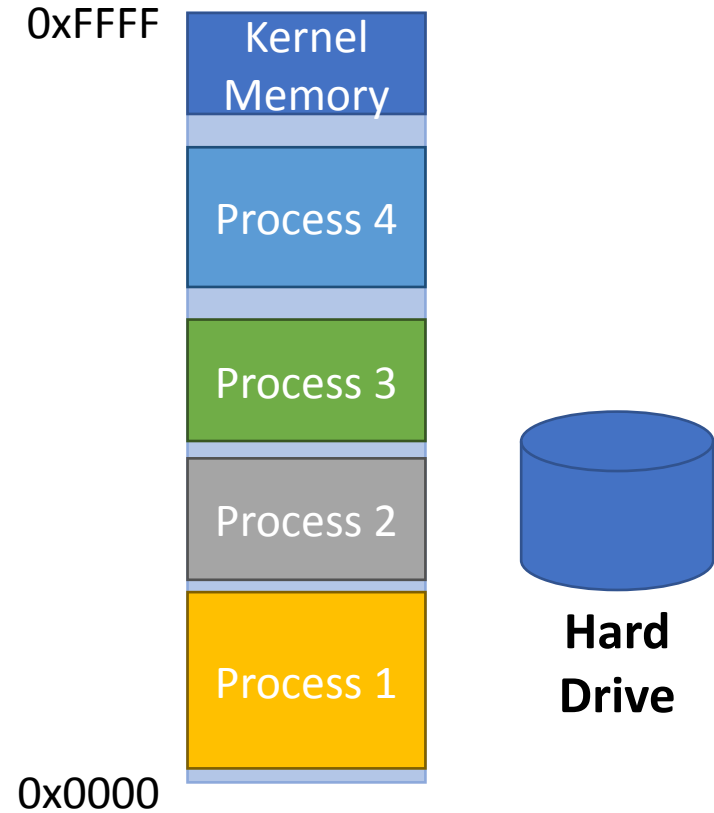
- Key idea:

Take frames from physical memory and swap (write) them to disk

- This frees up space for other code and data
-
- Load data from swap back into memory on-demand
 - If a process attempts to access a page that has been swapped out...
 - A page-fault occurs and the instruction pauses
 - The OS can swap the frame back in, insert it into the page table, and restart the instruction

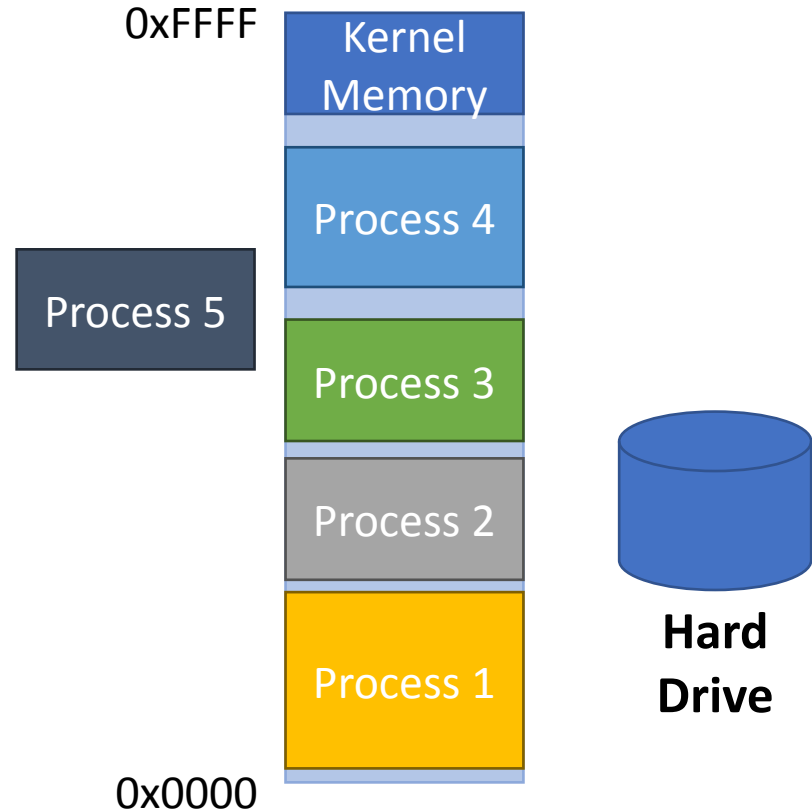
Swapping Example

- Suppose memory is full



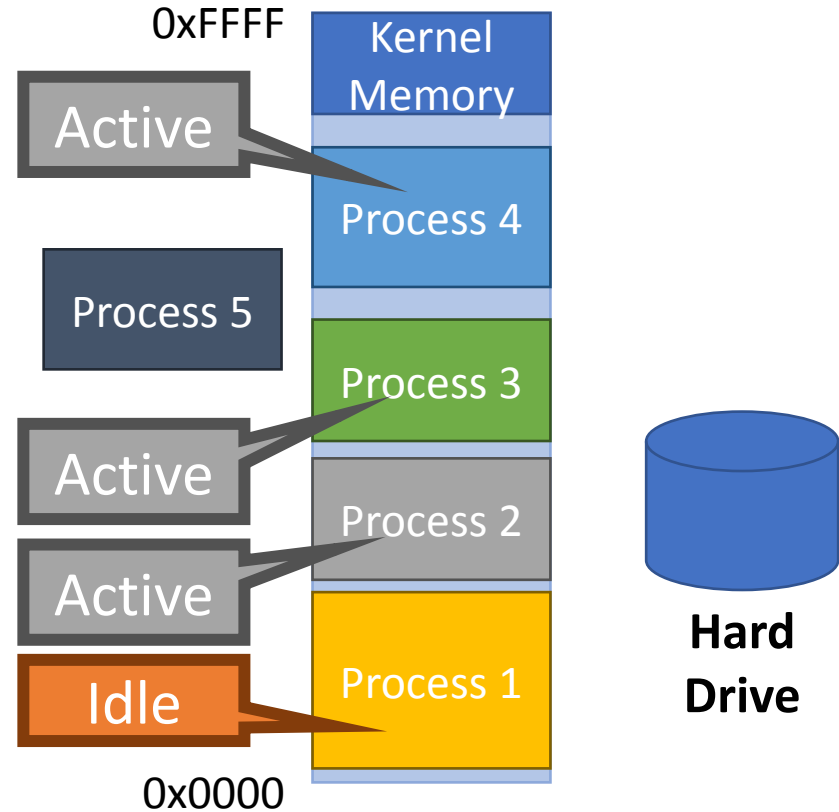
Swapping Example

- Suppose memory is full
- The user opens a new program



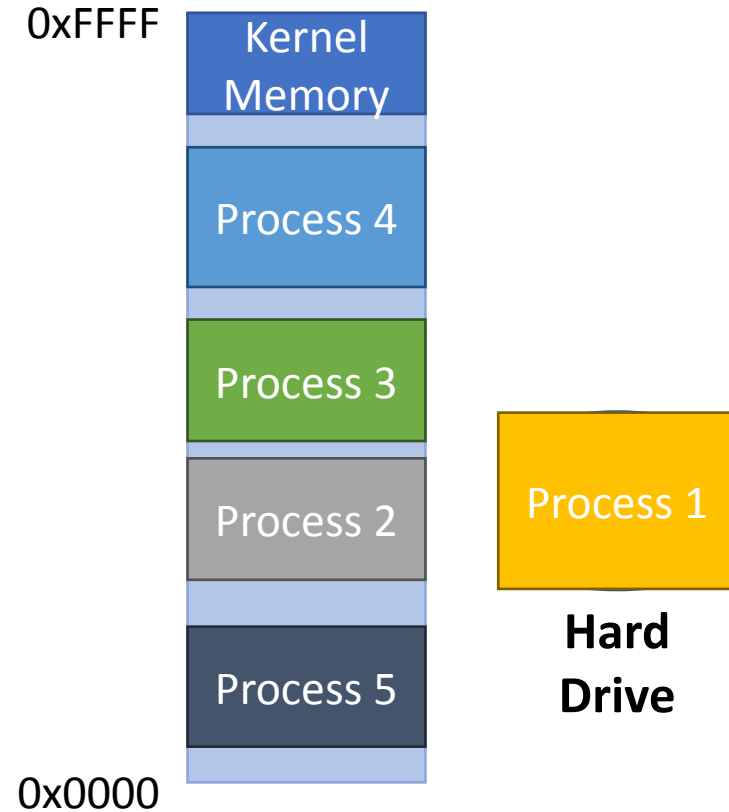
Swapping Example

- Suppose memory is full
- The user opens a new program



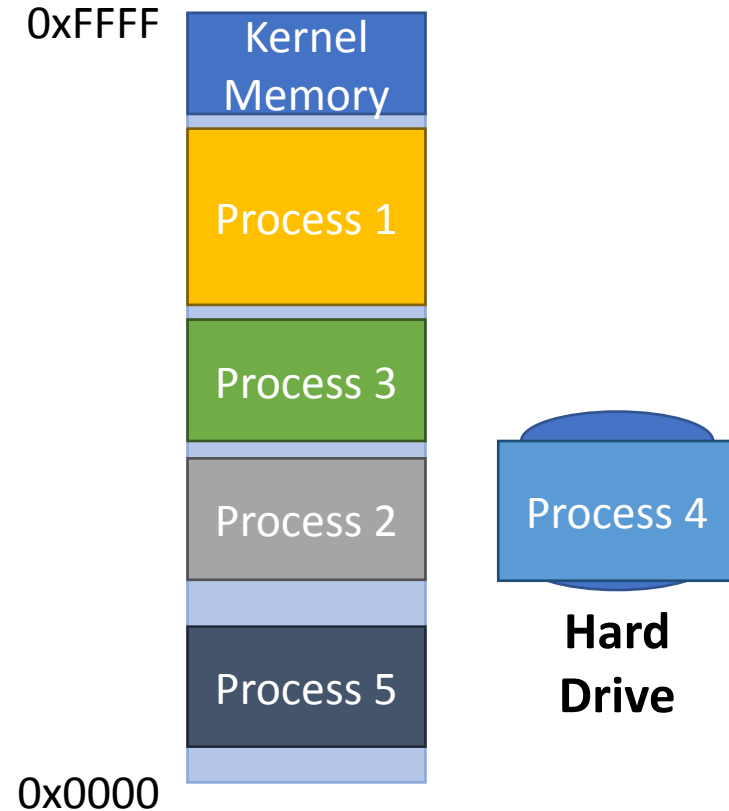
Swapping Example

- Suppose memory is full
- The user opens a new program
- Swap out idle pages to disk



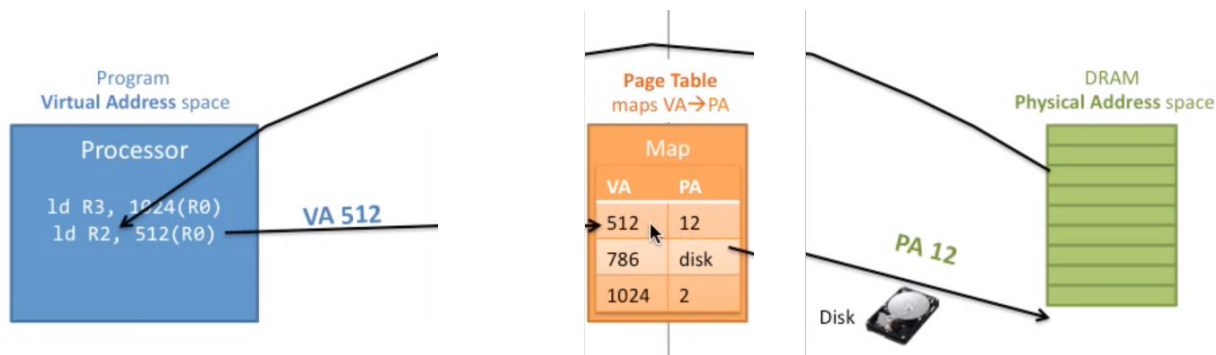
Swapping Example

- Suppose memory is full
- The user opens a new program
- Swap out idle pages to disk
- If the idle pages are accessed, page them back in



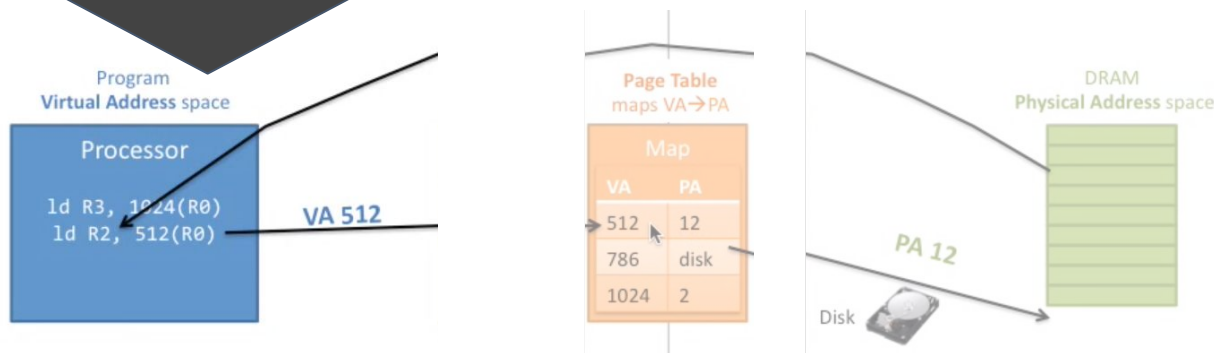
Introducing the **Page Table**!

- A page table keeps track of the mapping between virtual and physical memory addresses.
- Page table exists per process



Introducing the Page Table!

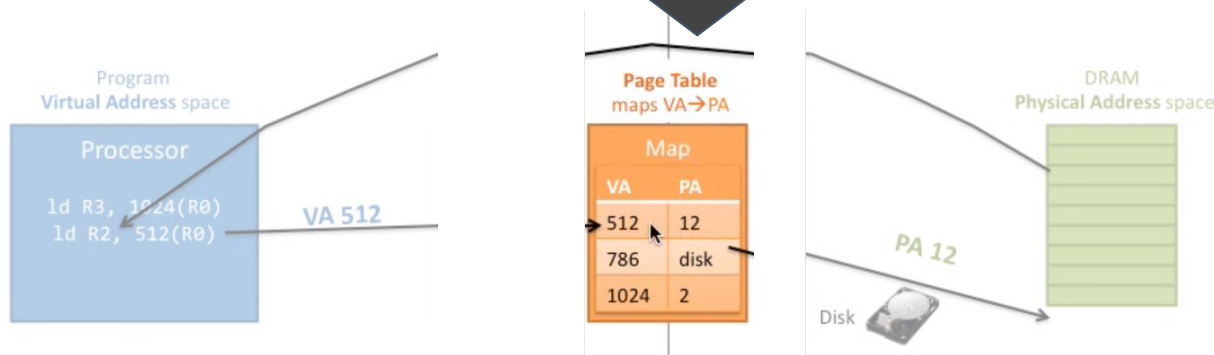
- A page table keeps track of the mapping between virtual and physical memory
 - Page table is a data structure maintained by the process
- Our process requests some address (which is actually a virtual address)



Introducing the Page Table!

- A page table keeps track of the mapping between virtual and physical memory addresses.
- Page table exists per process.

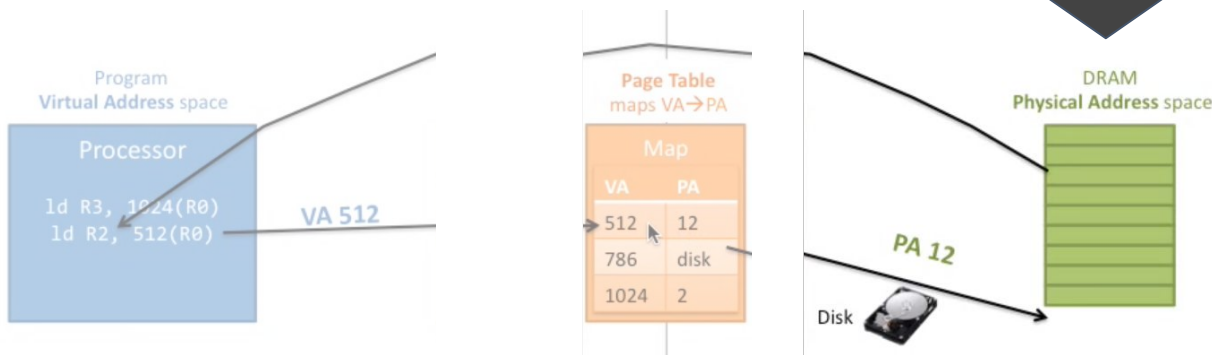
The Page Table maps us to the real physical address in DRAM



Introducing the Page Table!

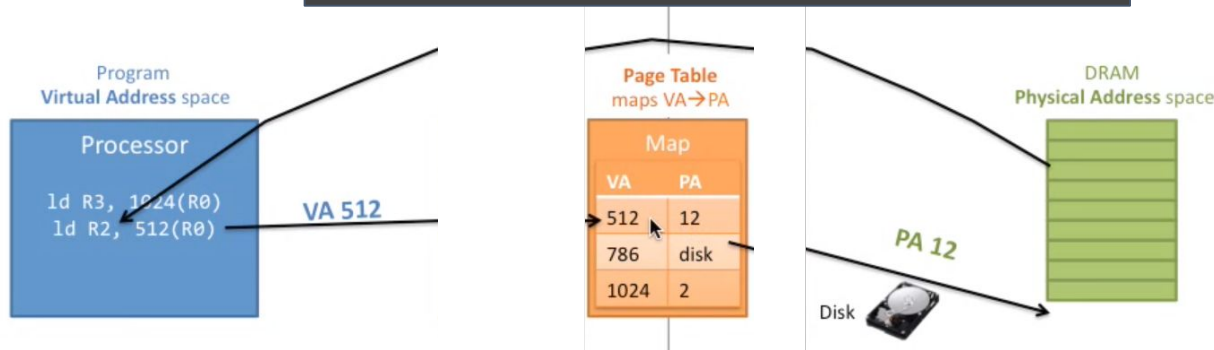
- A page table keeps track of the mapping between virtual and physical memory addresses.
- Page table exists per process

And we retrieve the actual data we need from DRAM.



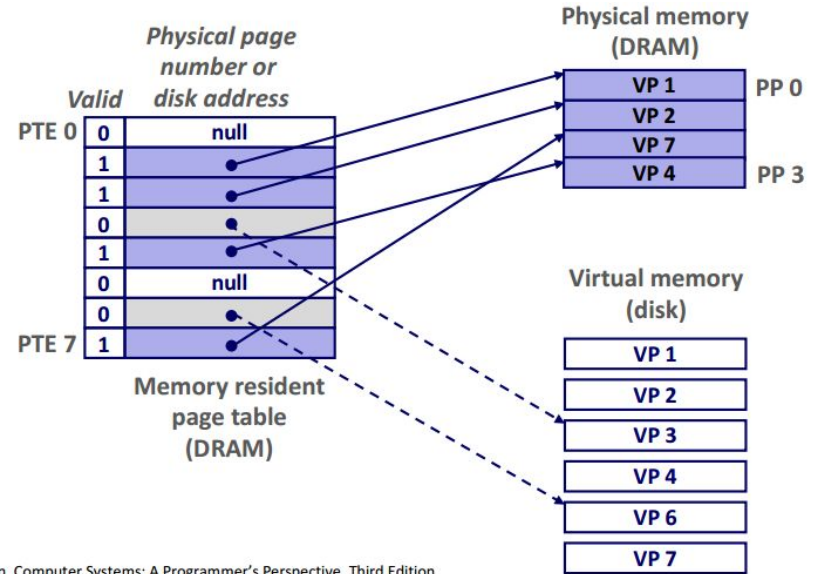
Introducing the Page Table!

- A page table keeps looking up 'pages'. (Otherwise we would have lots of 1 byte entries--which would make our page table huge!)
 - Page table exists p
- virtual and physical



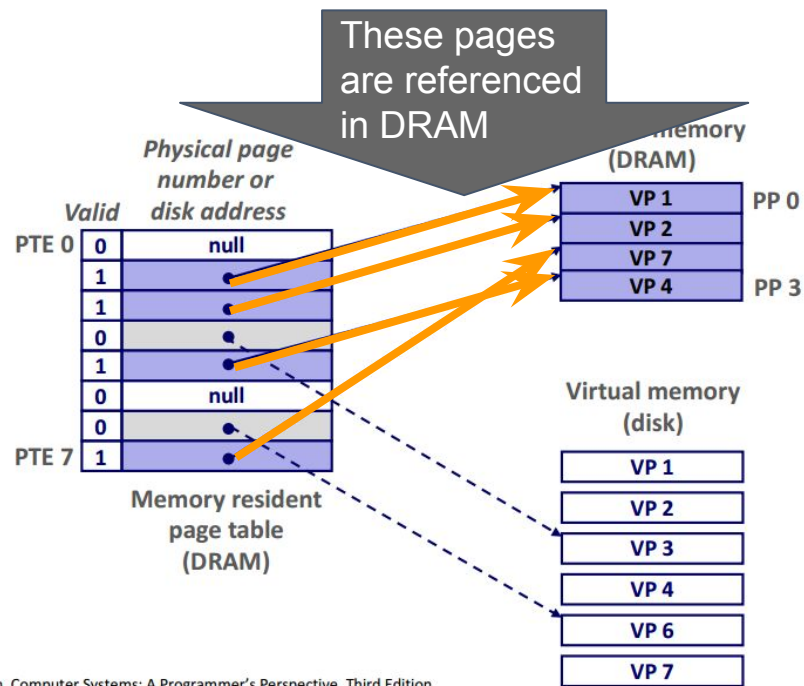
(Again) Enabling Data Structure: Page Table

- We divide memory into pages
 - Typically 4 KB for 1 page
- A page table then stores the mappings from a virtual page to its physical page address



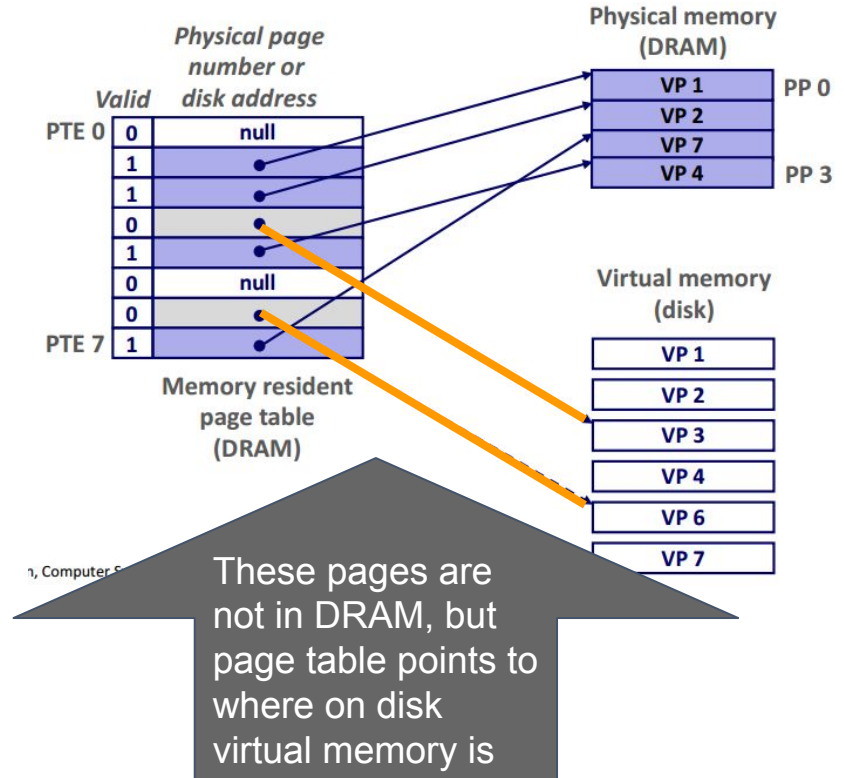
Enabling Data Structure: Page Table

- We divide memory into pages
 - Typically 4 KB for 1 page
- A page table then stores the mappings from a virtual page to its physical page address



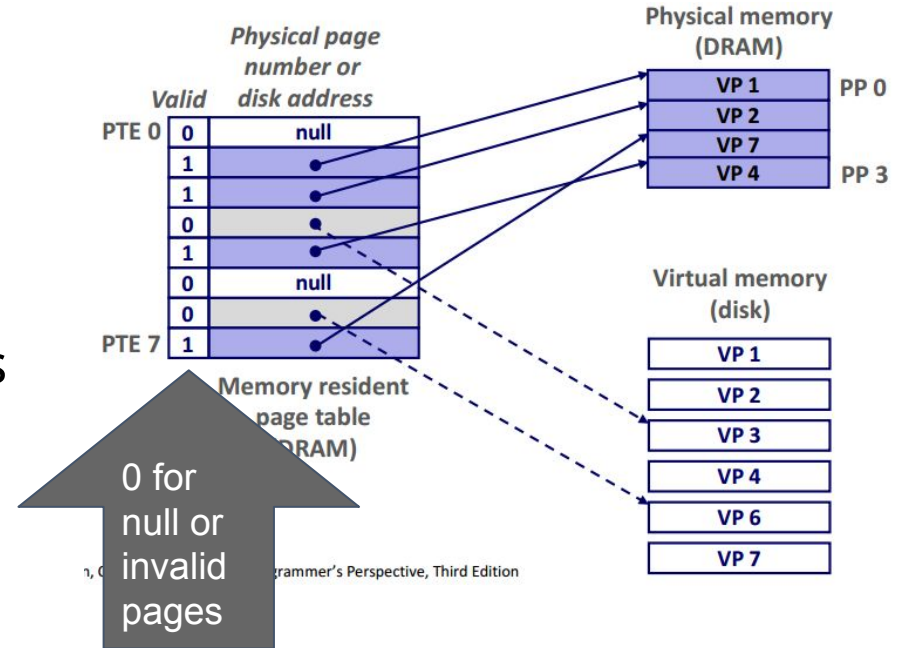
Enabling Data Structure: Page Table

- We divide memory into pages
 - Typically 4 KB for 1 page
- A page table then stores the mappings from a virtual page to its physical page address



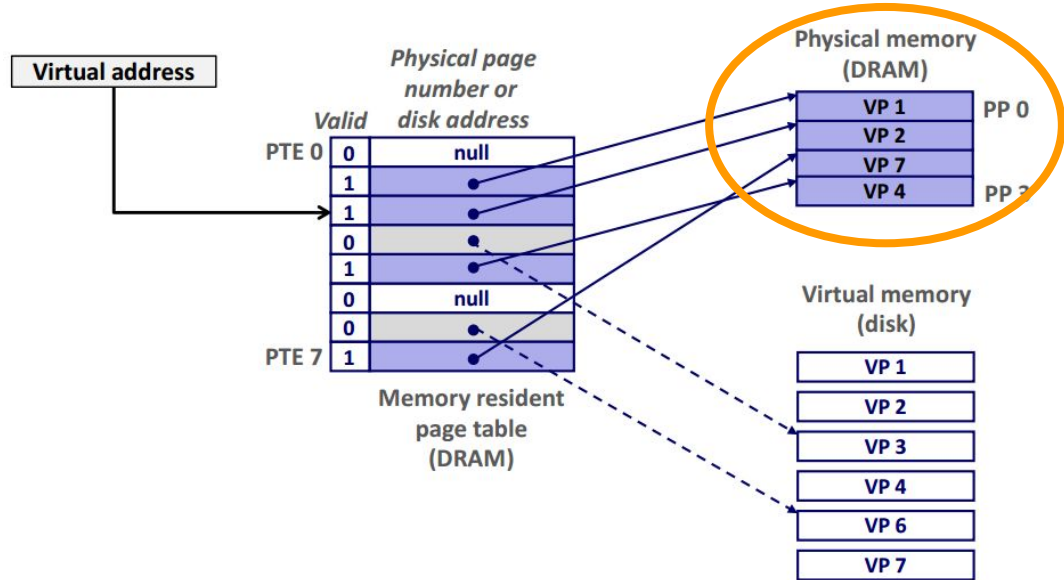
Enabling Data Structure: Page Table

- We divide memory into pages
 - Typically 4 KB for 1 page
- A page table then stores the mappings from a virtual page to its physical page address



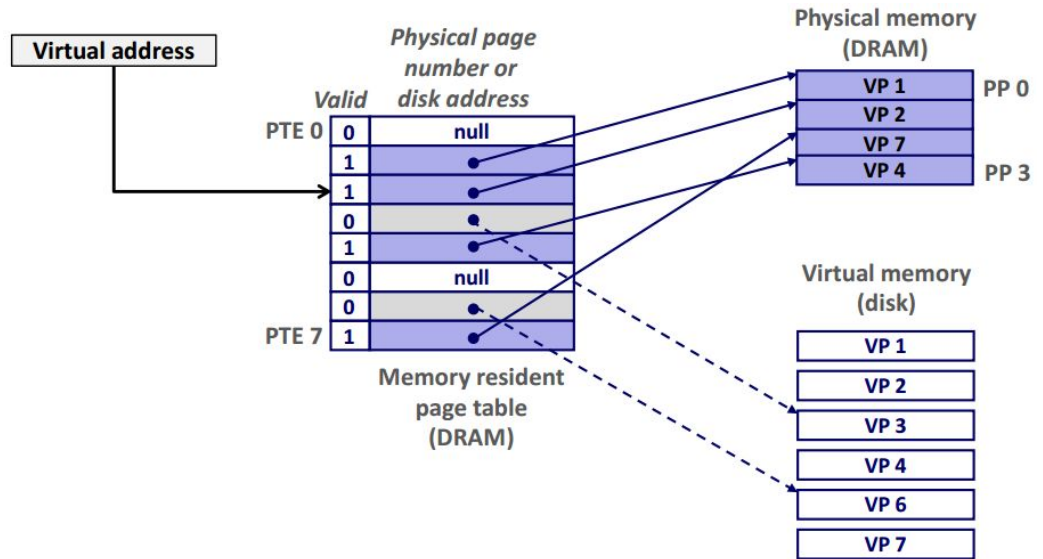
Page Hit

- Just like a cache hit, we see if our page is in DRAM



Page miss causes a Page Fault

- If our page is not in memory, then we get a page fault.
 - (VP 6 for example is not in our DRAM, but 1,2,7, and 4 are)



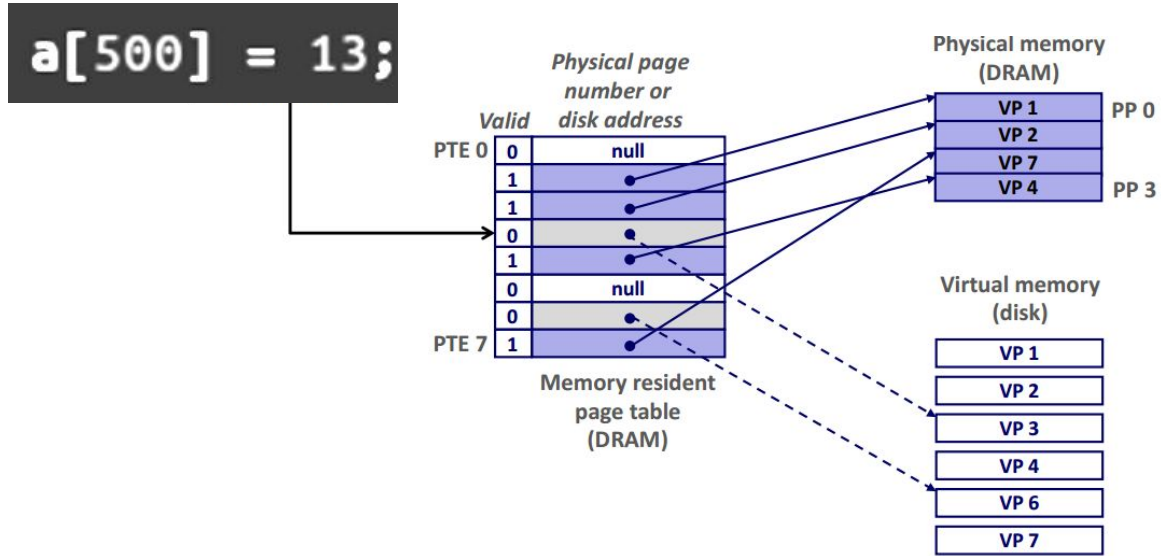
Page Fault Example

- User attempts to write to memory location

```
1 int a[1000];  
2  
3 main(){  
4  
5     a[500] = 13;  
6  
7 }
```

- OS may recognize this particular address is invalid.
 - Invalid in the sense of the OS noticing
“hey, this page is not in our page table”
- The proper behavior is for the OS to do something (i.e. handle this exception)
 - This involves evicting some page we do not need (some victim)
 - The instruction that caused the fault is then restarted
 - We get a page hit and move on.

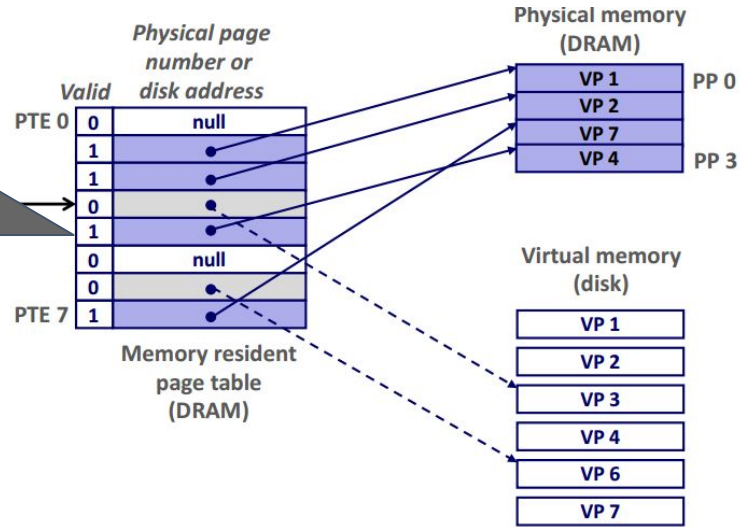
A walkthrough



A walkthrough

`a[500] = 13;`

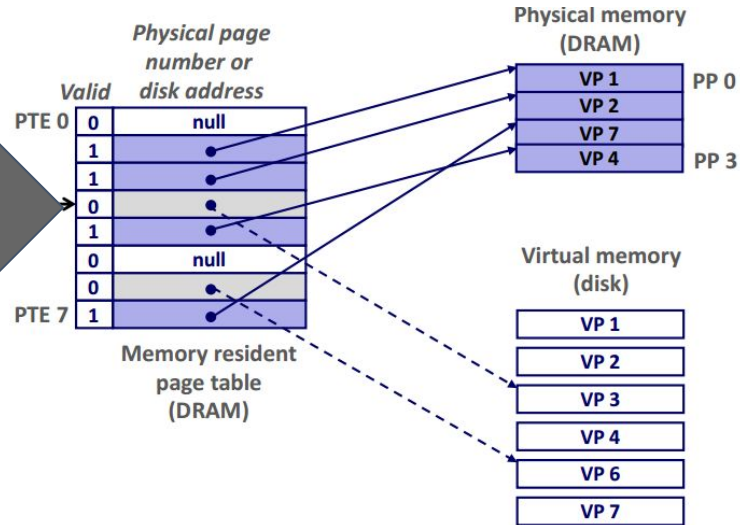
We try to
access/write
some data



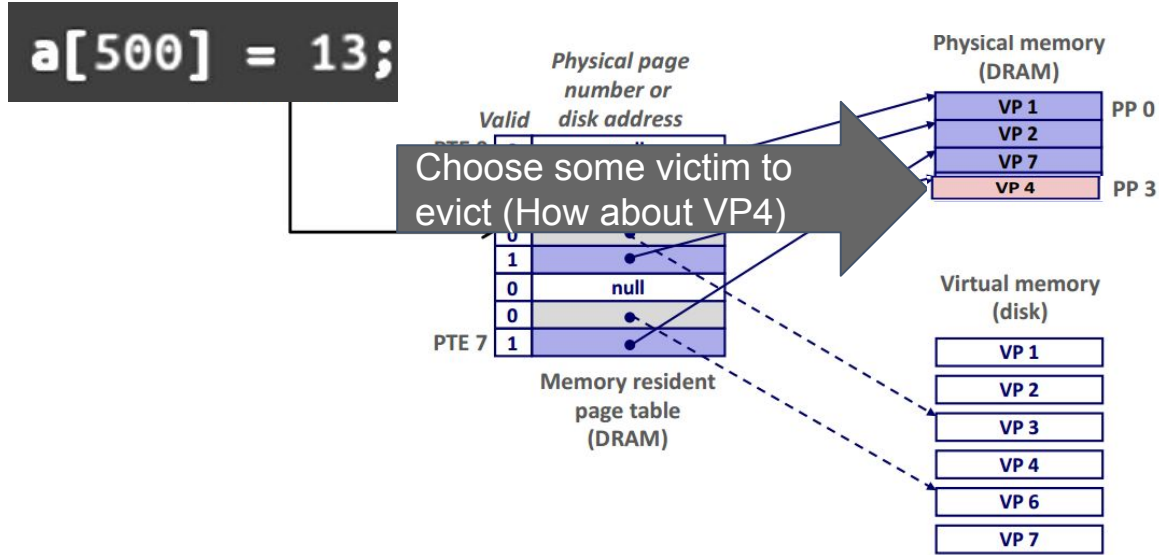
A walkthrough

a[500] = 13;

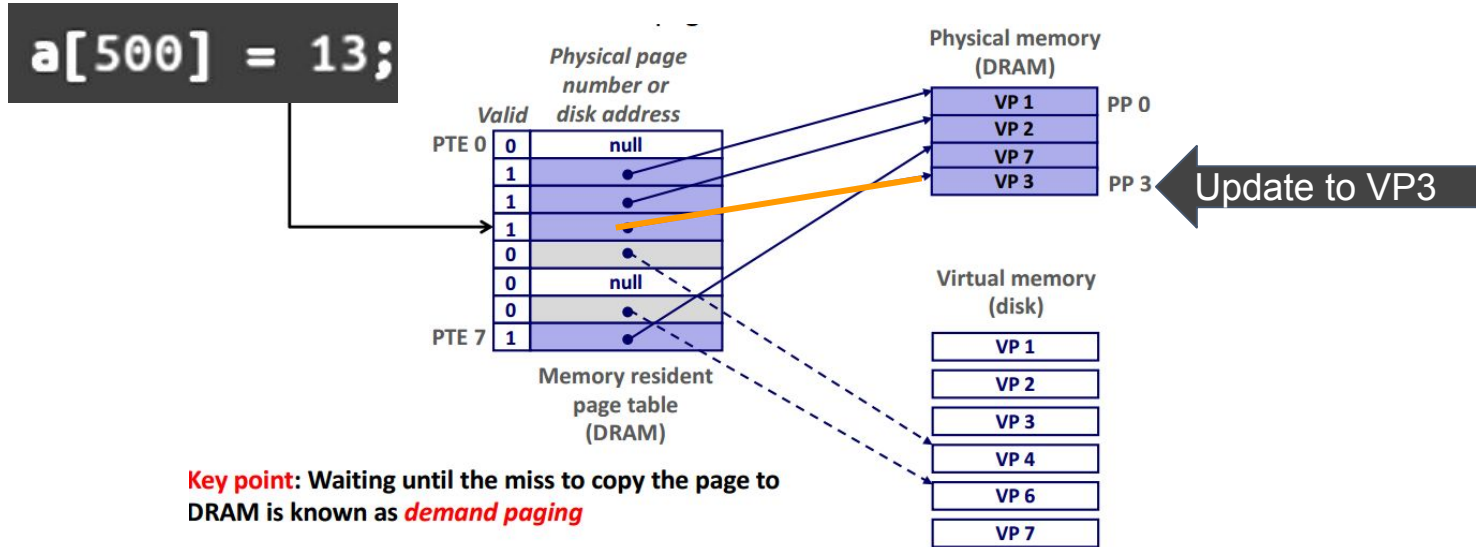
The page however is invalid (See the '0'), so now OS has to handle our page fault



A walkthrough



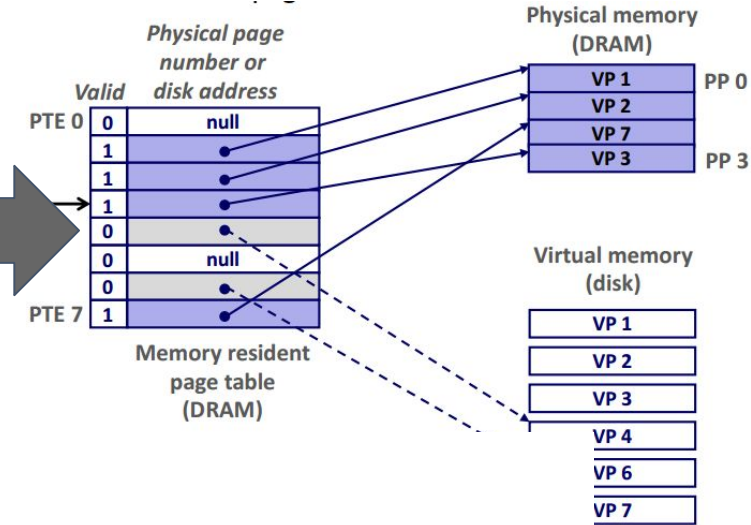
A walkthrough



A walkthrough

a[500] = 13;

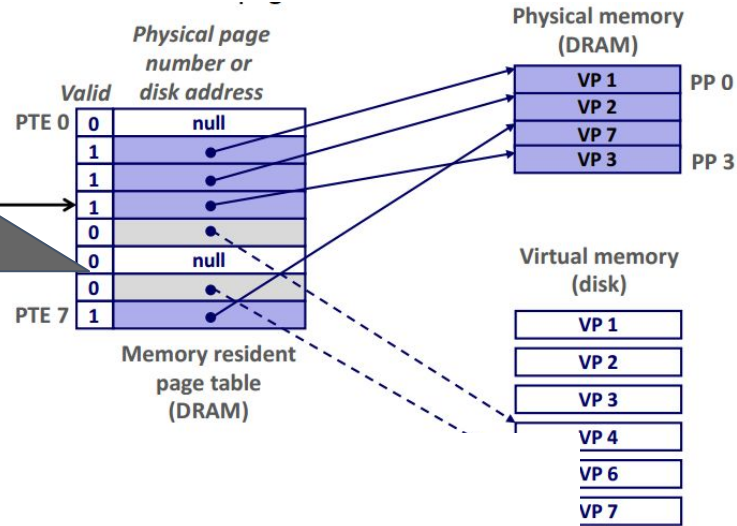
VP4 as a result is evicted



A walkthrough

a[500] = 13;

We execute where we left off and now see we have a valid page. a[500] is now 13.



Question: Page Faults

- When your program executes, do you get a lot of page faults?

Question: Page Faults

- When your program executes, do you get a lot of page faults?
- Use "perf stat ./myProgram"
 - Observe the different counts of the page-faults and context-switches

Answer and New Question

- When your program executes, do you get a lot of page faults?
 - Typically yes!
 - But this is okay because a lot of the nitty gritty is handled for us.
 - Generally we do not try to predict the access patterns of page accesses

Answer and New Question

- When your program executes, do you get a lot of page faults?
 - Typically yes!
 - But this is okay because a lot of the nitty gritty is handled for us.
 - Generally we do not try to predict the access patterns of page accesses
 - After our compulsory misses, we generally do pretty well. Why?
 - Locality to the rescue!
 - If we have a page of memory in our DRAM Cache, typically where we are working (our *working set*) only on a small piece of data at a time in our programs.
 - If the data we are working on is larger than our main memory size, then we get thrashing!
 - i.e. lots and lots of page swaps!

Quick Summary of Virtual Memory so far

- We found we could access our memory and organize them into 4096 byte pages
 - (Again, usually 4096 bytes per page, but this can vary by OS)
- We could then access these pages by looking in a page table
- These individual pages can be cached in the DRAM
 - This is a trend in computer science (i.e., we've seen this a couple of times), figure out how to cache things and speed up lookup times

Three Virtual Memory Advantages

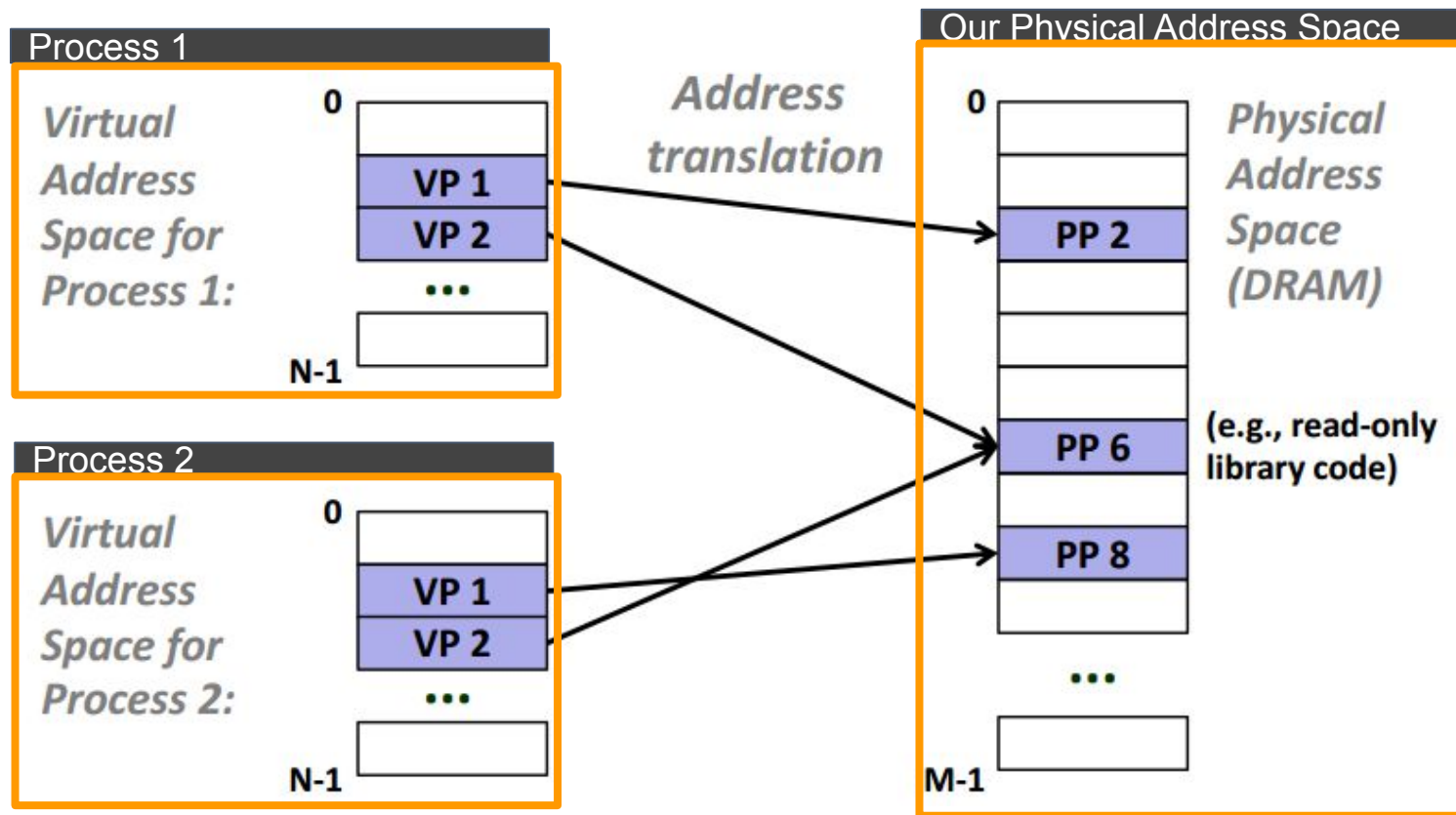
1. Use Main memory efficiently
- 2. Simplifies memory management (for application developers)**
- 3. Isolates Address Spaces**

**#2 Simplifies memory management
(for application developers)**

Virtual Memory for Memory Management

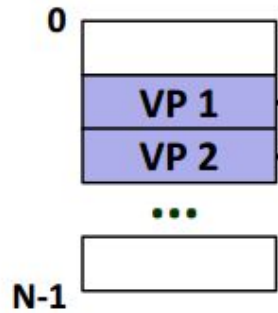
- Each process has its own virtual address space
 - This means we can view (within a process), memory as a linear array.
 - In reality, we know we have many pages scattered around.
 - (This could cause locality issues...so the OS needs to choose good mappings)

Example of page mappings



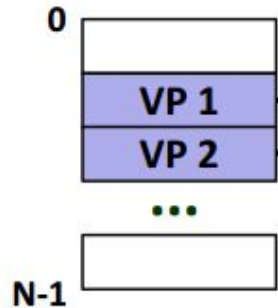
All of my
physical
memory is
here

*Virtual
Address
Space for
Process 1:*



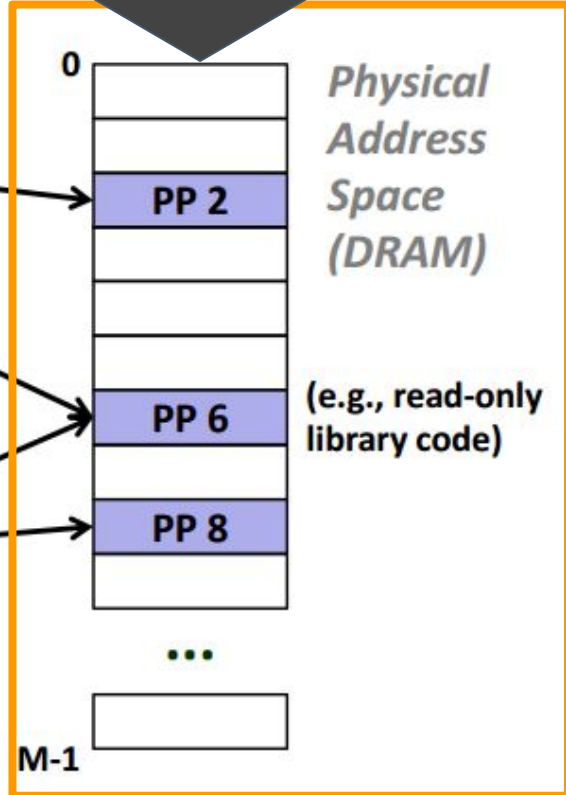
*Address
translation*

*Virtual
Address
Space for
Process 2:*

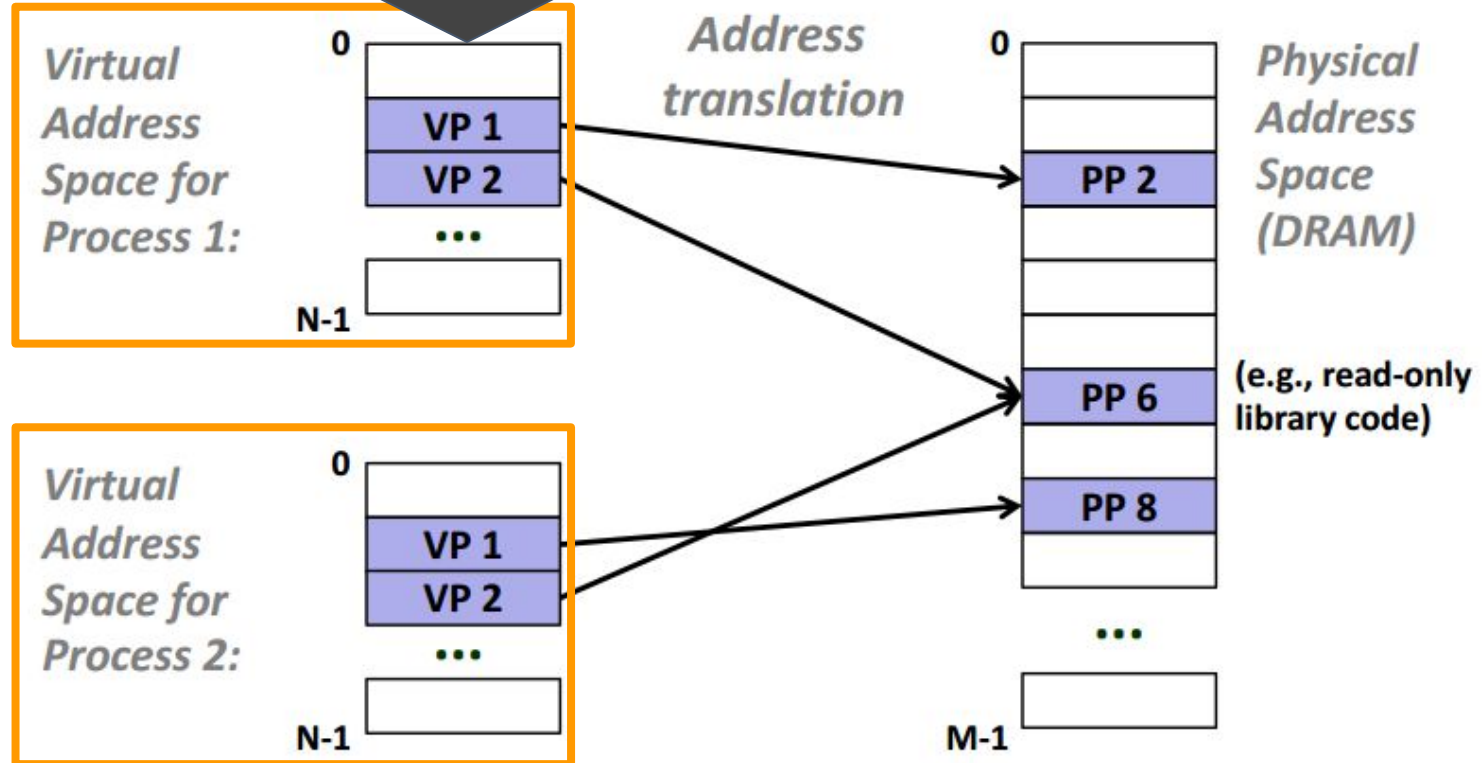


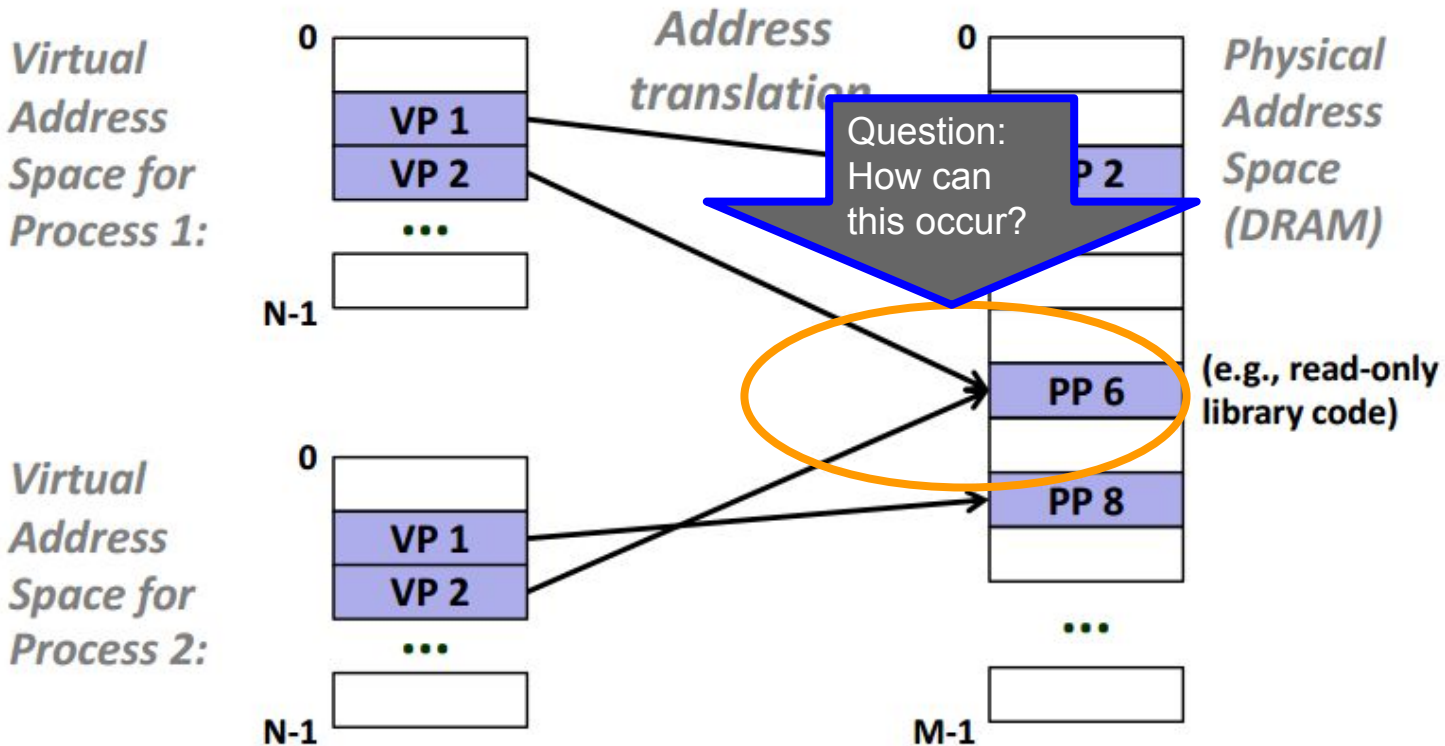
*Physical
Address
Space
(DRAM)*

(e.g., read-only
library code)

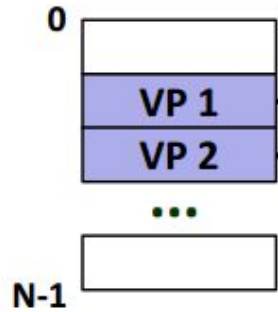


And our
process sees
its memory
stored linearly
here

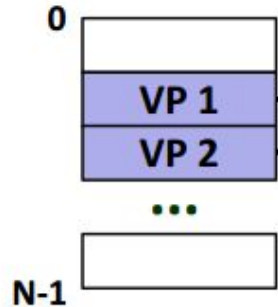




*Virtual
Address
Space for
Process 1:*



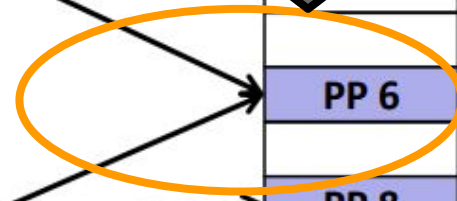
*Virtual
Address
Space for
Process 2:*



*Address
transl*

Answer: Assume
this is a fork(). As
long as the data
does not change
(.rodata or
library), no need
to map to different
data

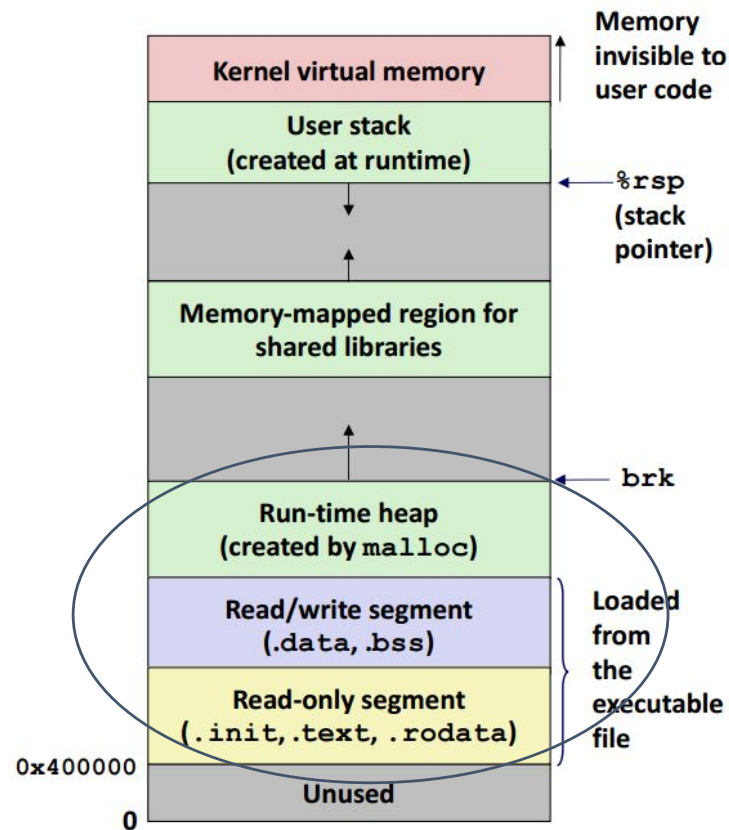
*Physical
Address
Space
(DRAM)*



(e.g., read-only
library code)

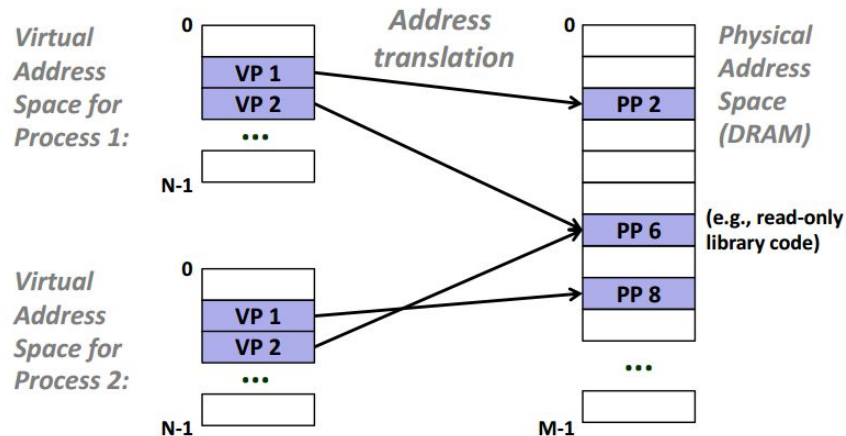
Virtual Memory supports Linking and Loading

- To our program, the virtual address space is roughly the same
 - code, data, and heap sections start at same address



Virtual Memory as Memory Manager Summary

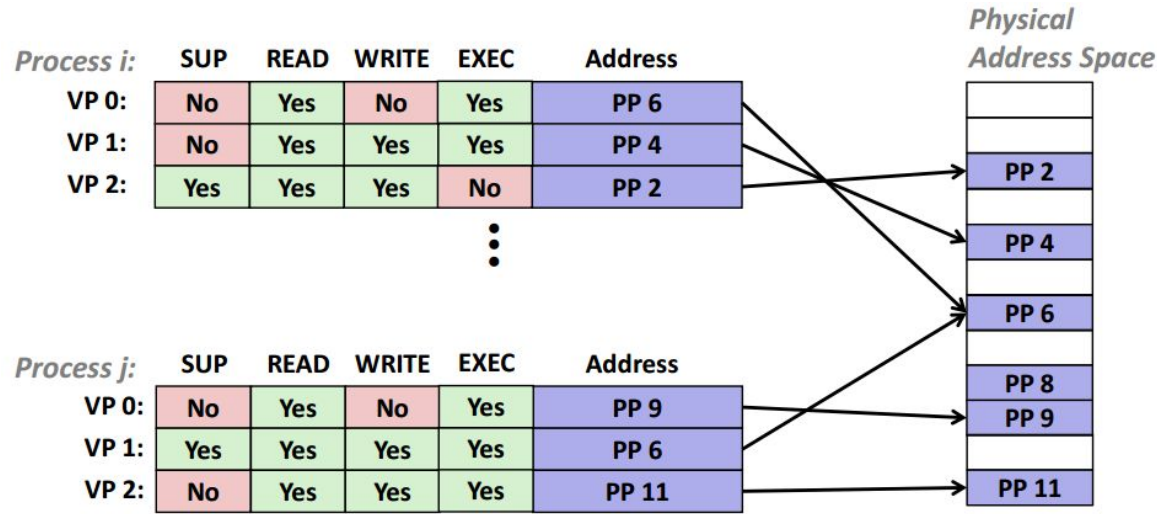
- So for each of these virtual pages, they map to a physical page (PP)
- Processes store any number of virtual pages at a given time.
 - And sometimes these virtual pages (VP) are shared if read-only code (e.g. a library of code--which will not change!)



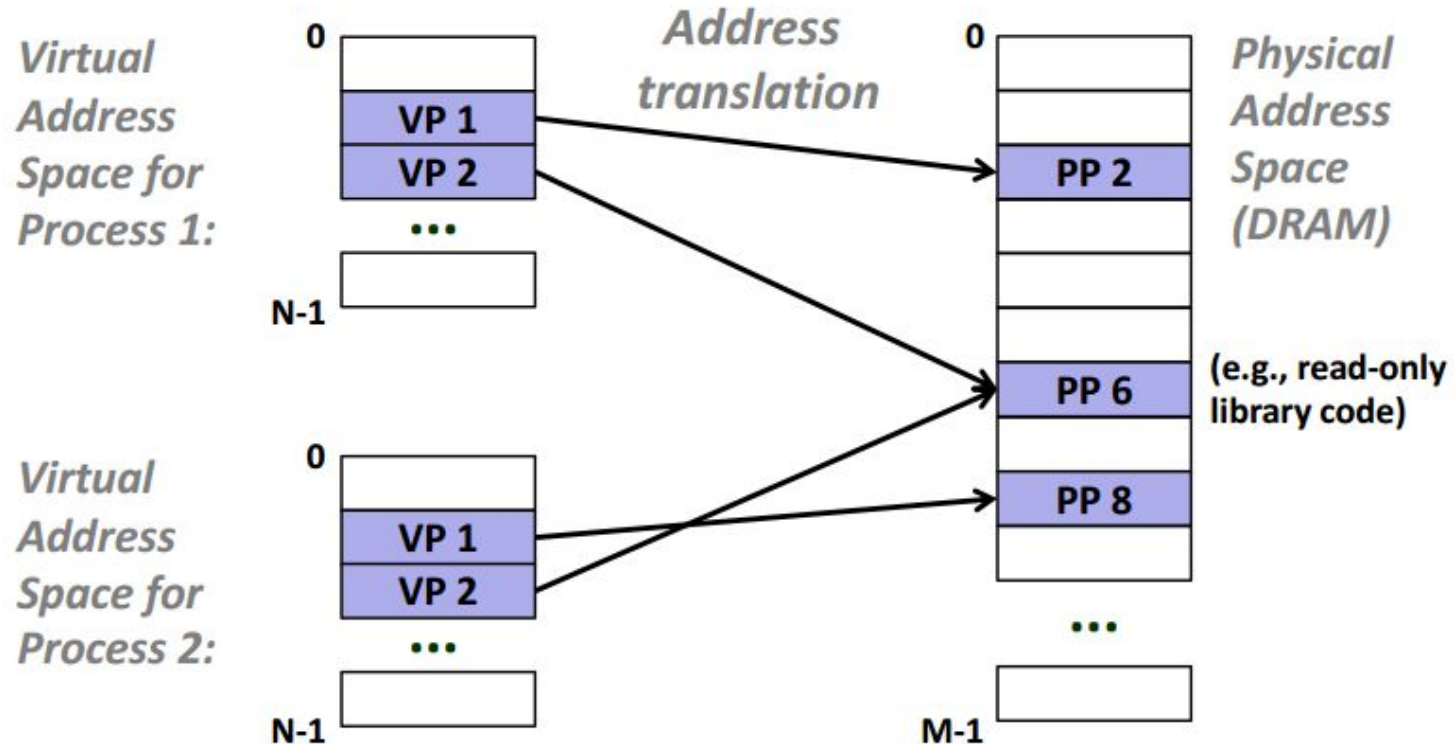
#3 Isolates Address Spaces

Virtual Memory protection

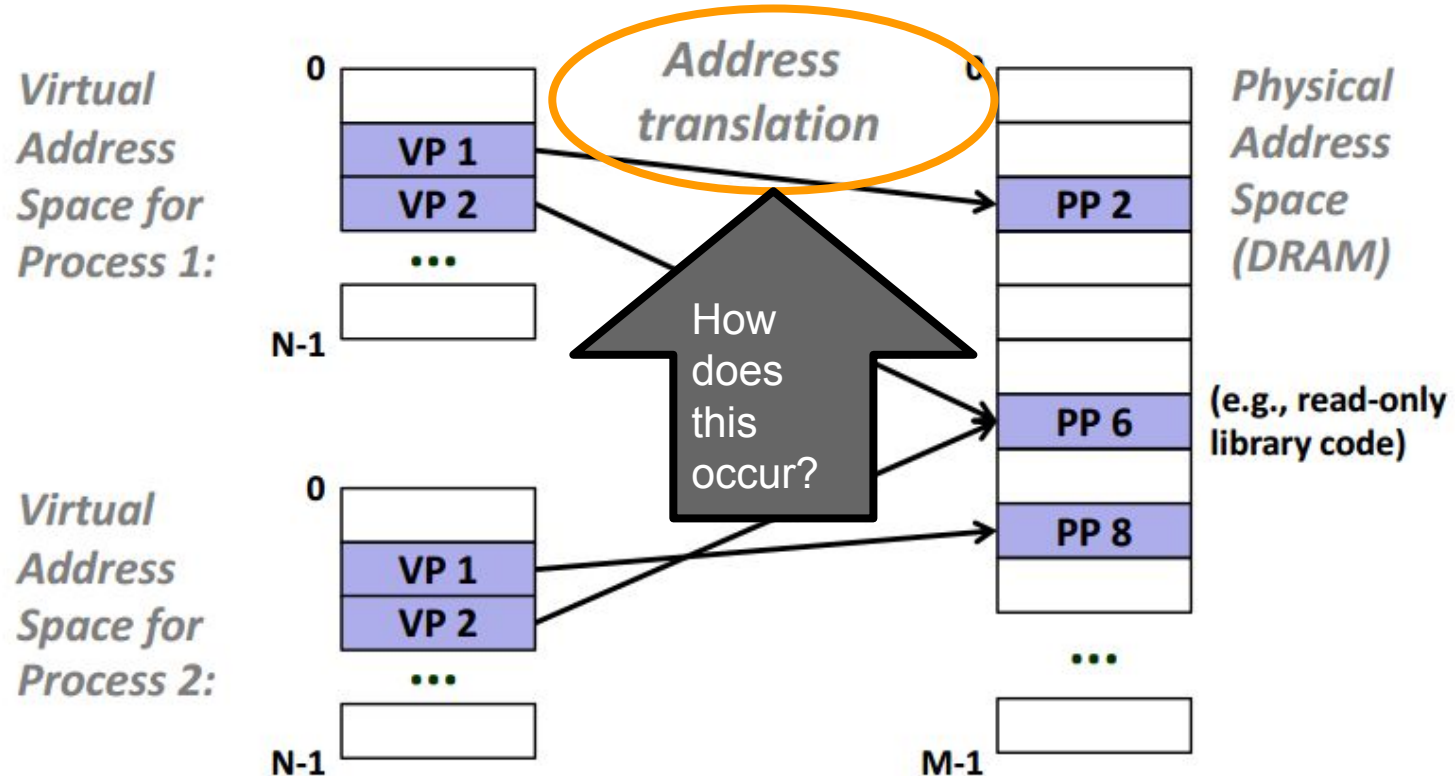
- Depending on the access, the MMU (Memory Management Unit) determines which pages can be executed.



Revisiting our picture - One missing component



Revisiting our picture - One missing component



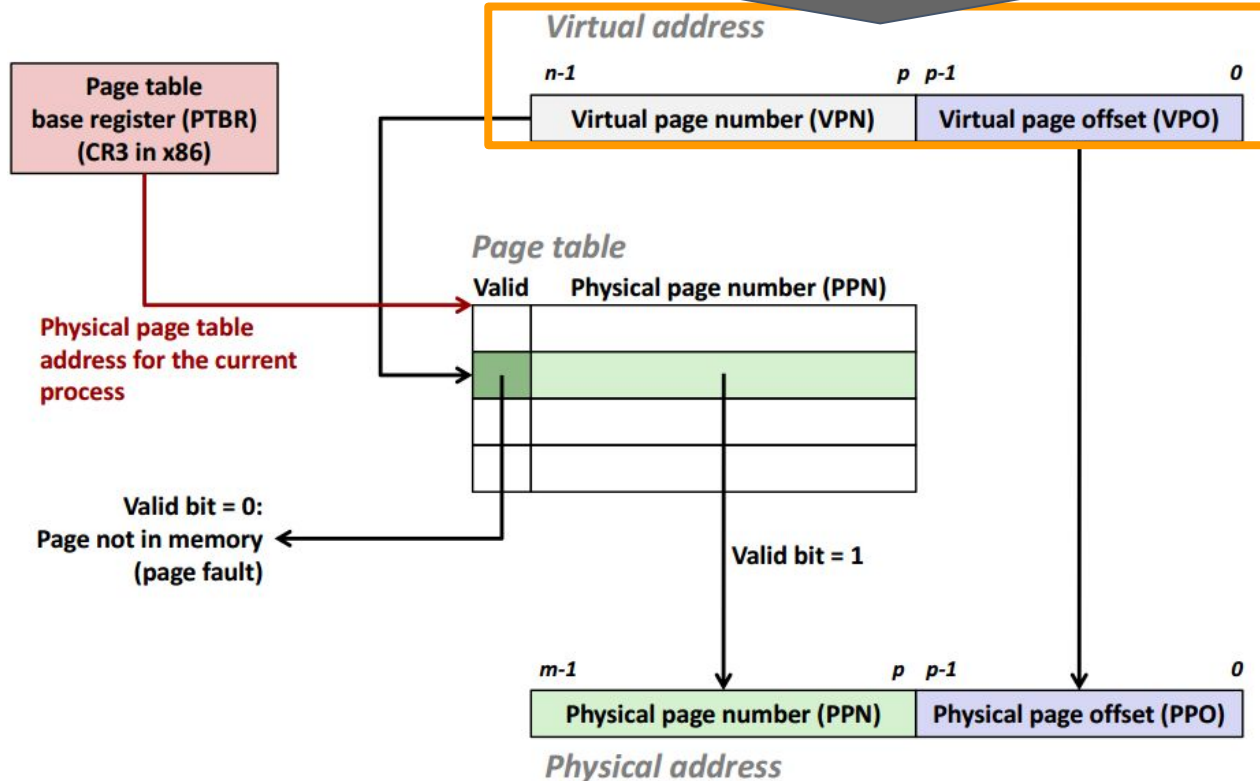
Address Translation Example

Address Translation - Notation

- Basic Parameters
 - $N=2^n$: Number of addresses in virtual address space
 - $M=2^m$: Number of addresses in physical address space
 - $P=2^p$: Page size (bytes)
- Components of virtual address (VA)
 - VPO: Virtual page offset
 - VPN: Virtual page number [what we are looking for]
- Components of physical address (PA)
 - PPO: Physical page offset (same as VPO)
 - PPN: Physical page number

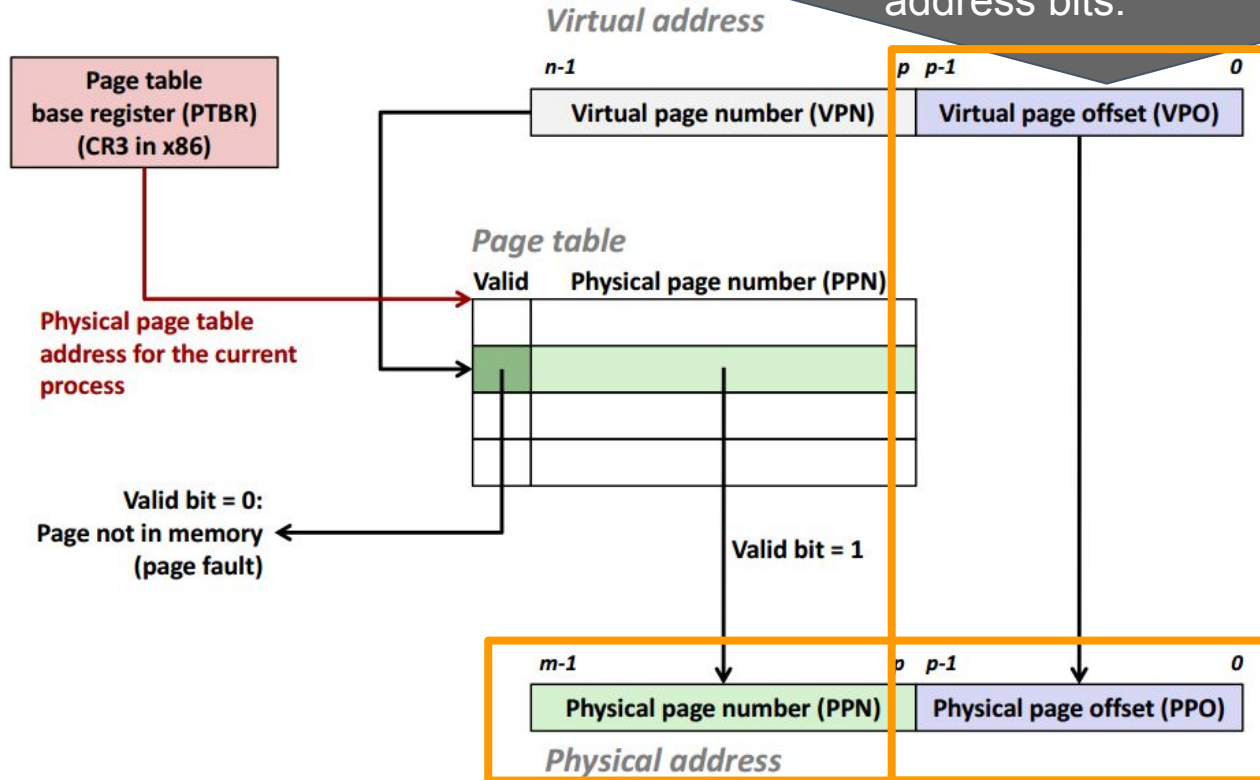
Address Translation with Page Table

Here's a virtual address I want to translate to its physical address

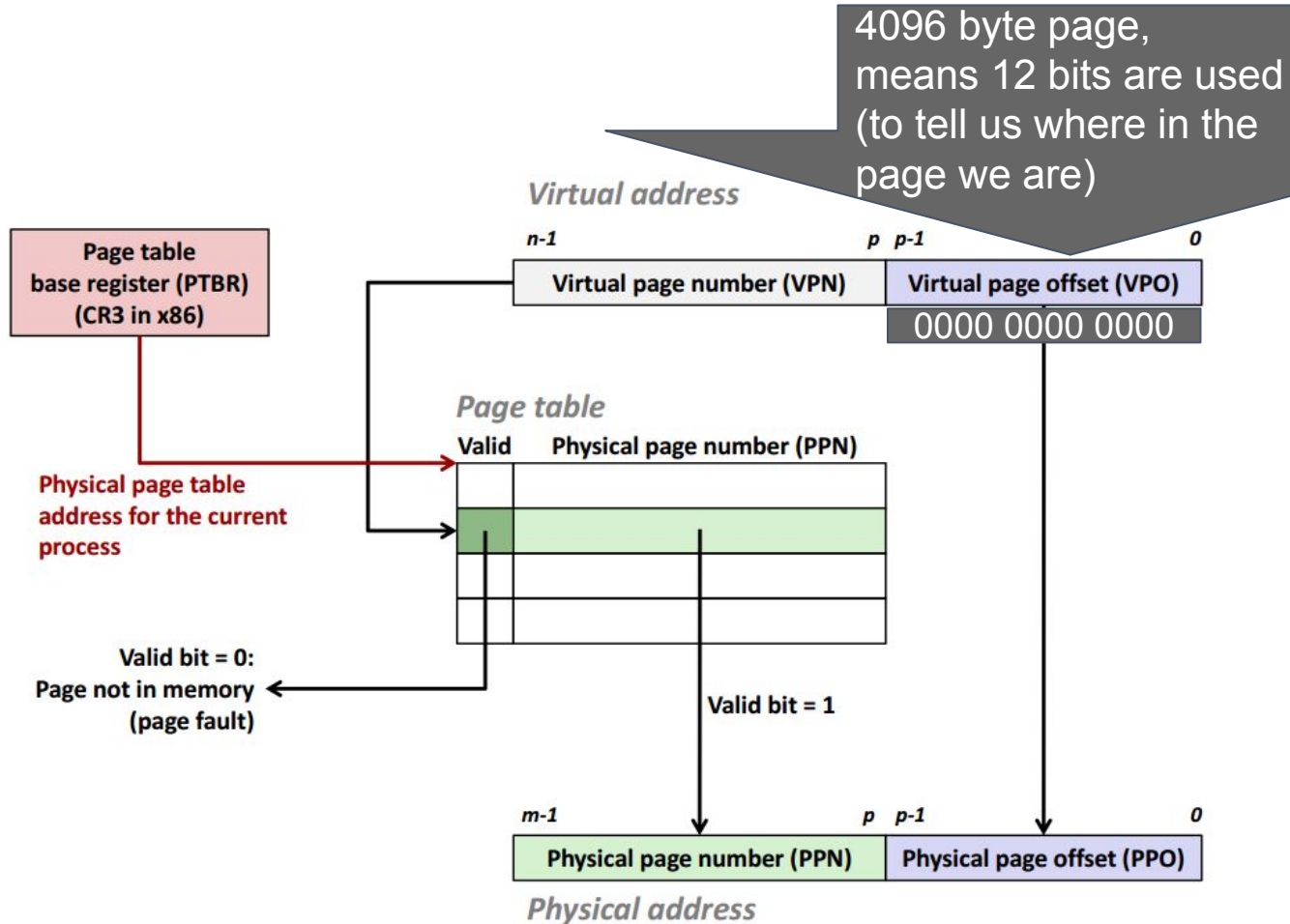


Address Translation with Page Table

This same lower order index of bits, will map to the same physical address bits.

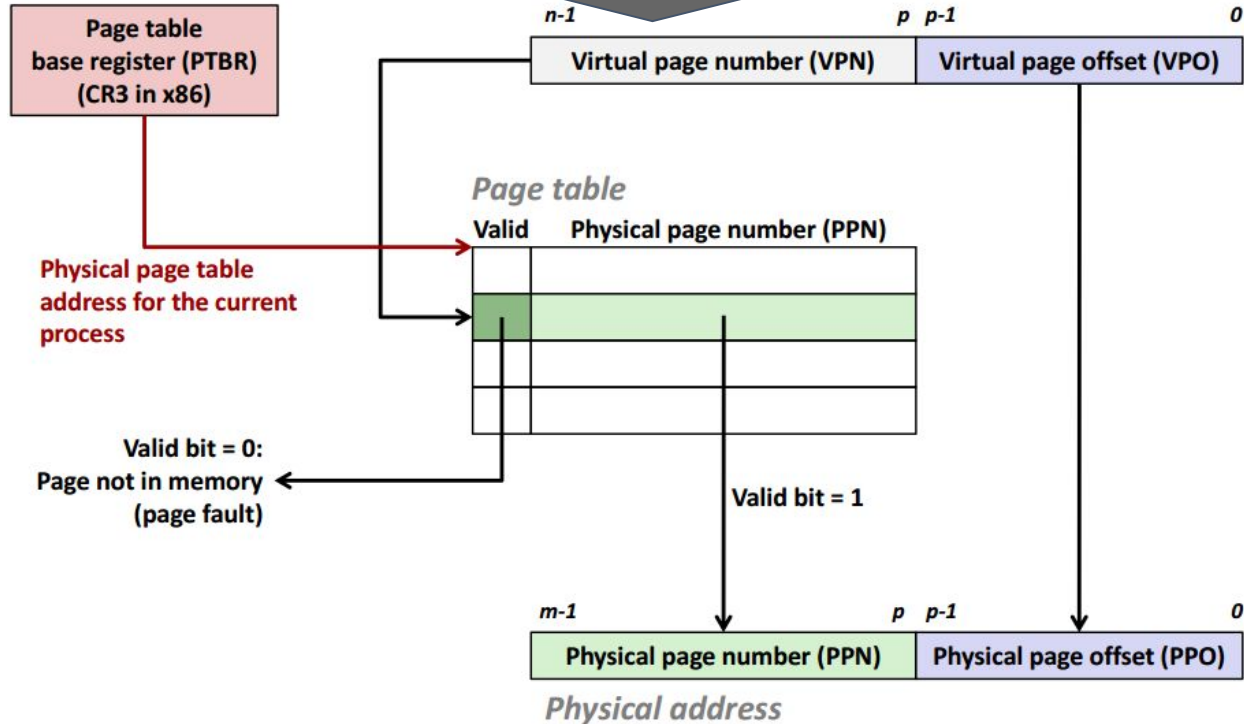


Address Translation with Page Table

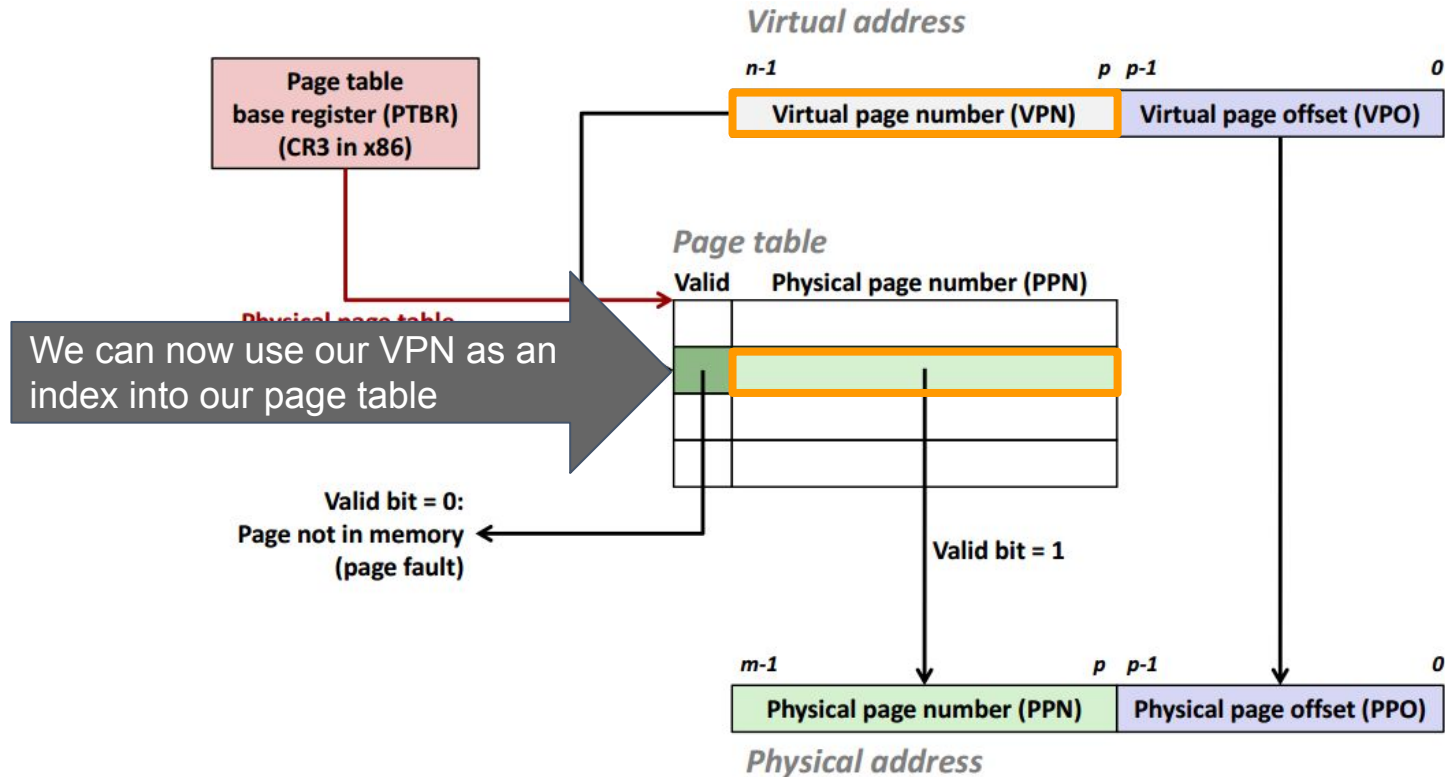


Address Translation with Page Table

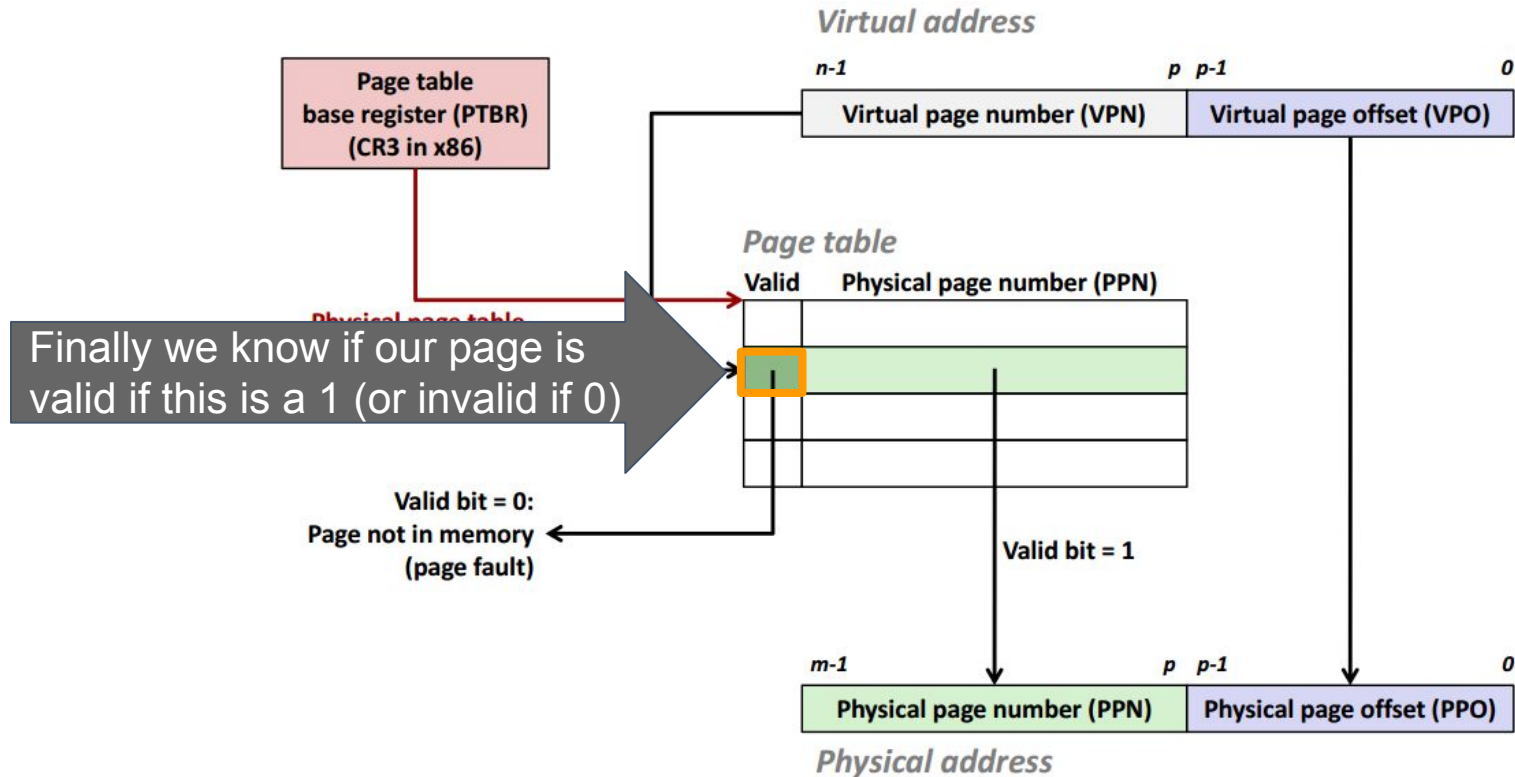
So now we translate our
virtual page number(VPN)
to physical page
number(PPN)



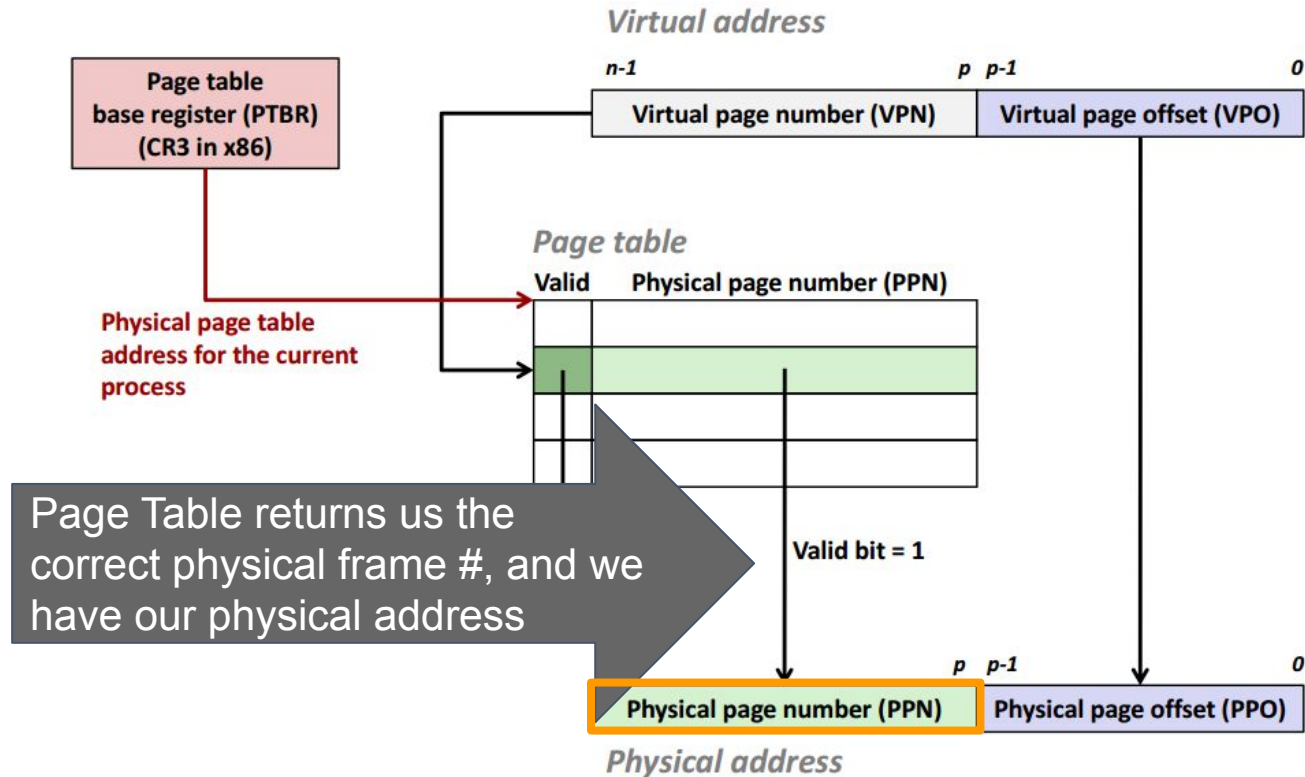
Address Translation with Page Table



Address Translation with Page Table

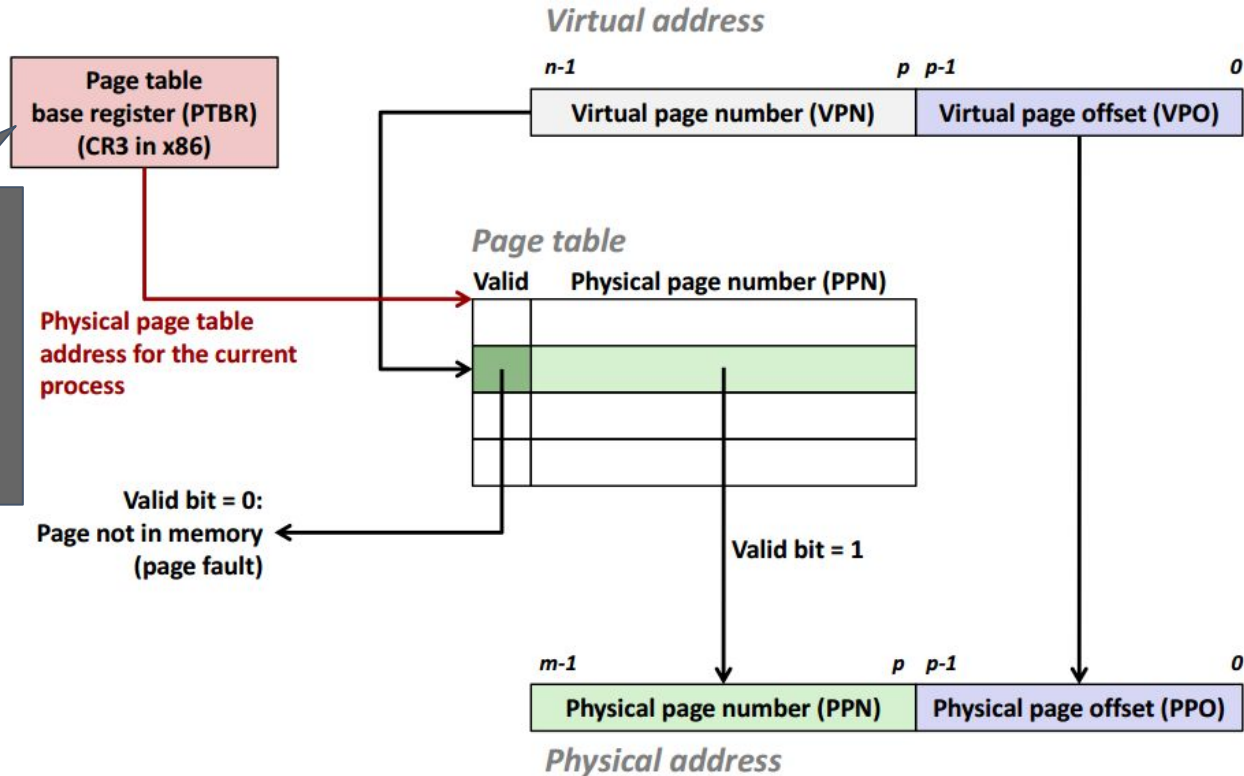


Address Translation with Page Table



Address Translation with Page Table

Note: A special register stores a pointer to the actual page table.

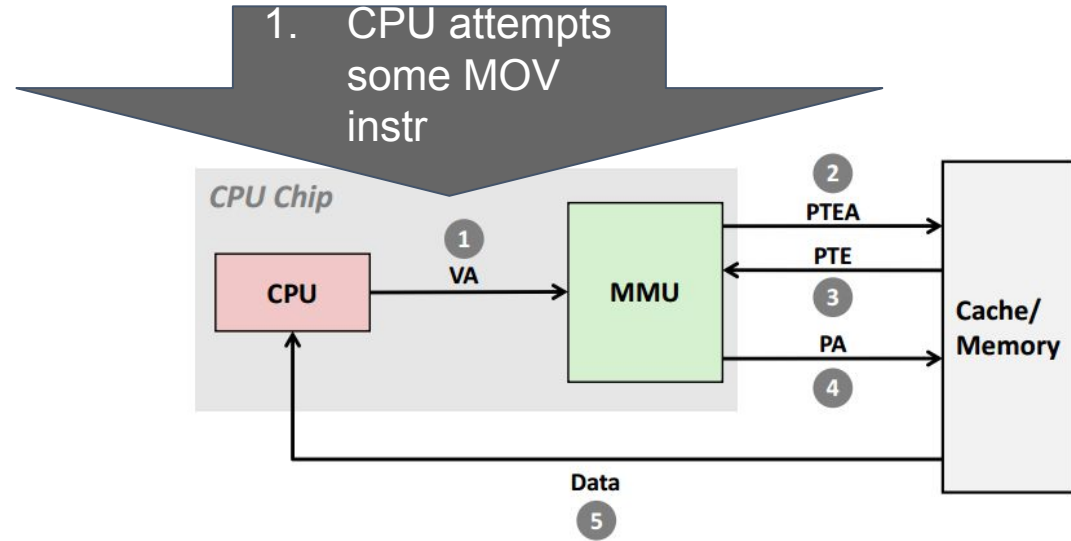


This looks like a LOT of work!

- There is a bit going on--remember what our goals are though
- We want our operating system to have the ability to handout more memory as needed.
 - And often this memory is not in nice sequential order
- And often when there is a lot of work to be done, we have special hardware for it
 - Let us take a look at the Memory Management Unit (MMU)!

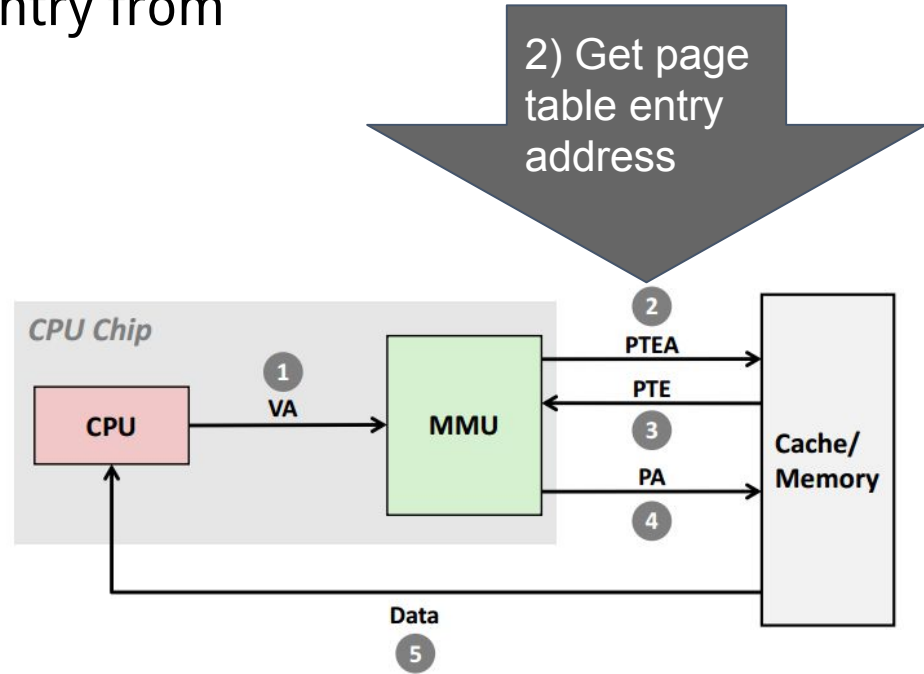
Address Translation: Page Hit

- 1) Processor sends virtual address to MMU
word to processor



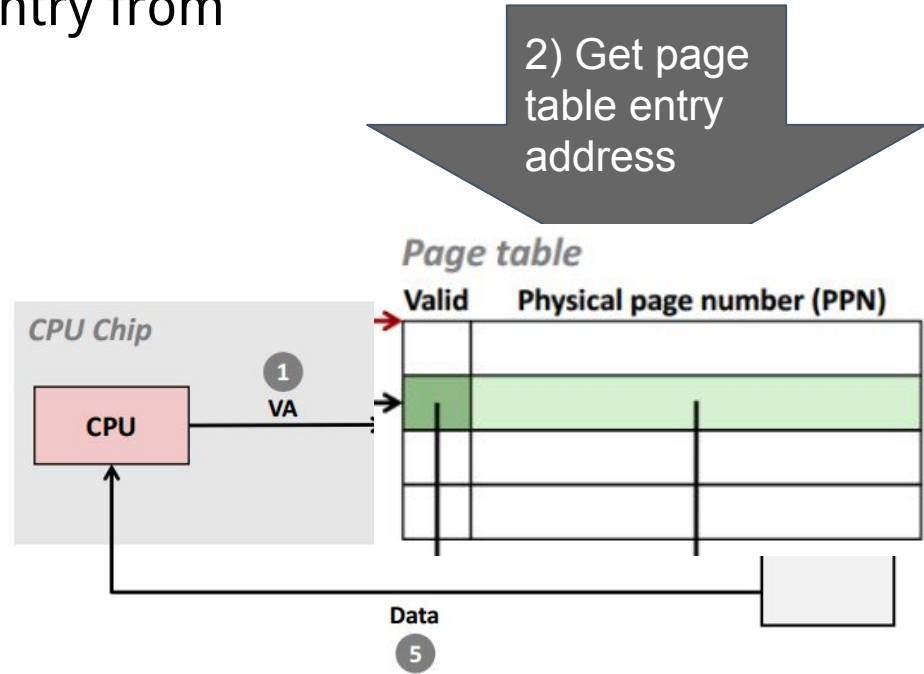
Address Translation: Page Hit

2, 3) MMU Fetches Page Table Entry from page table in memory



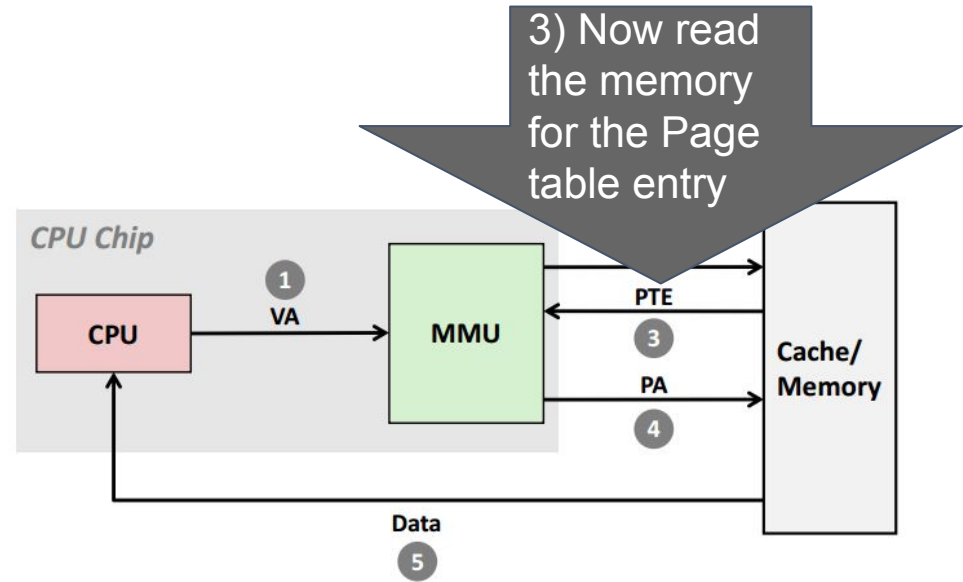
Address Translation: Page Hit

2, 3) MMU Fetches Page Table Entry from page table in memory



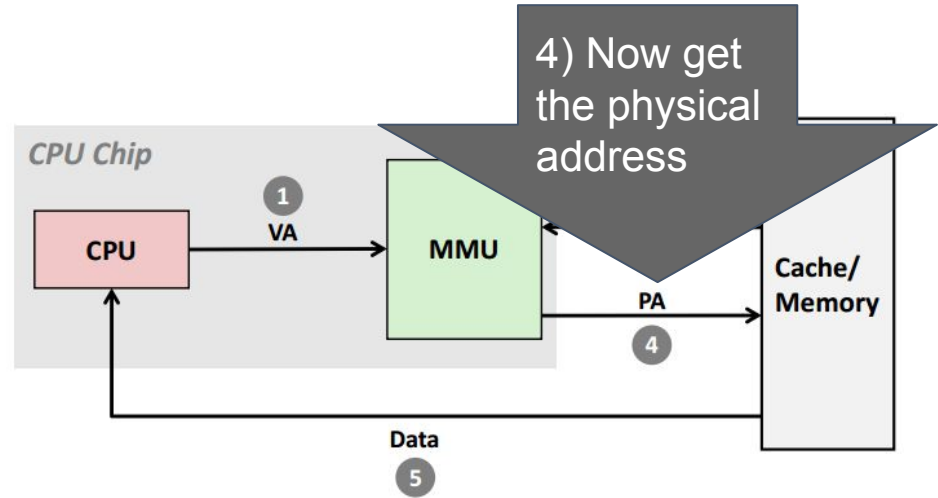
Address Translation: Page Hit

2, 3) MMU Fetches Page Table Entry from page table in memory



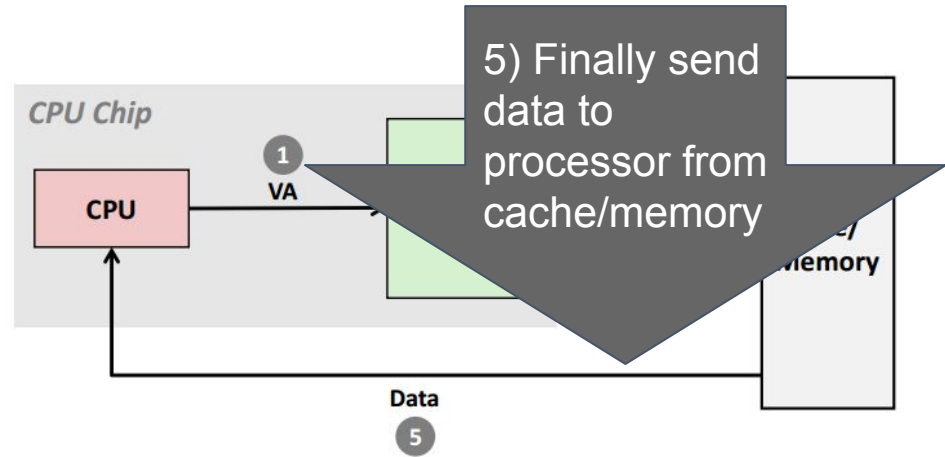
Address Translation: Page Hit

4) MMU Sends physical address to cache/memory

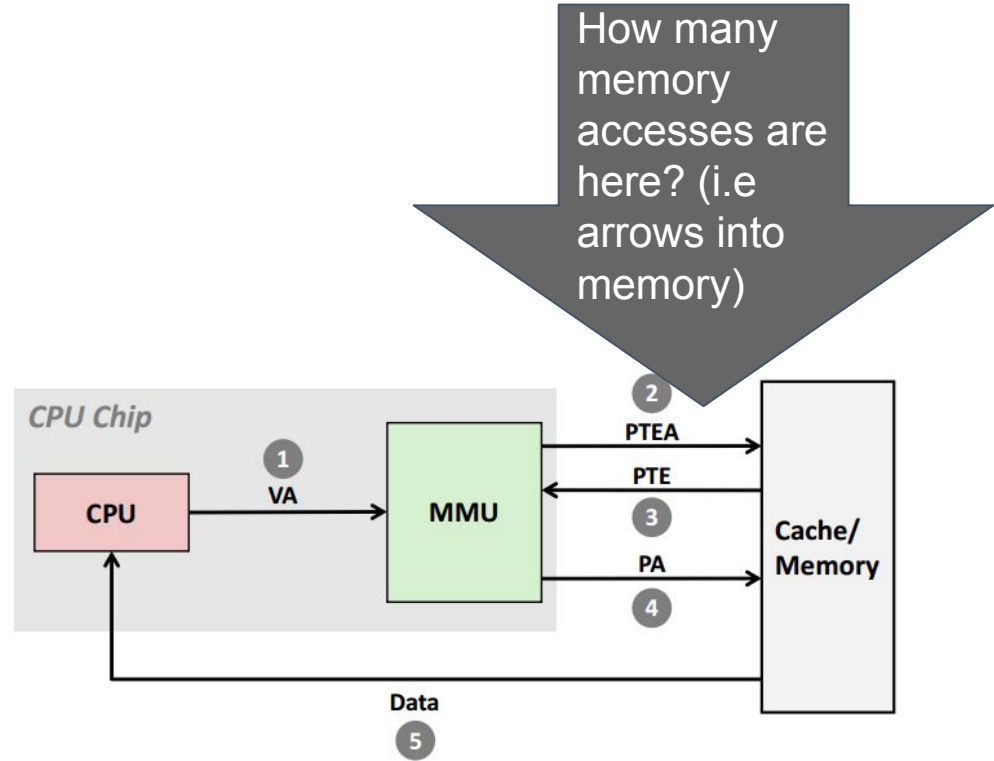


Address Translation: Page Hit

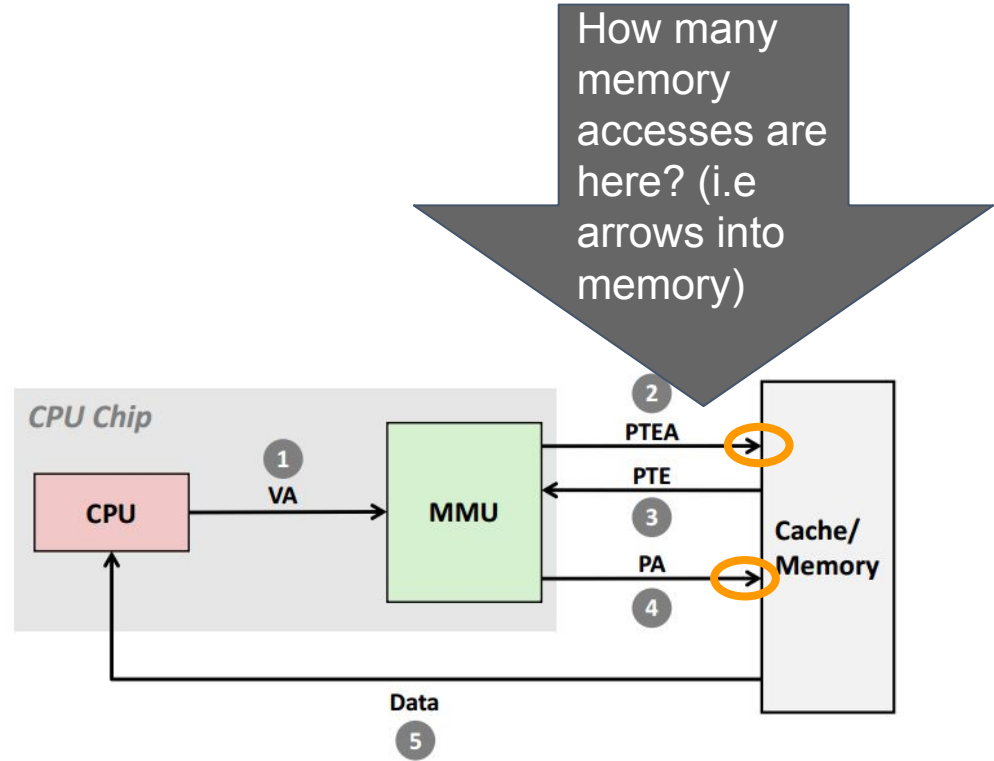
5) Cache/memory sends data word to processor



Address Translation: Page Hit



Address Translation: Page Hit



Address Translation: Page Fault

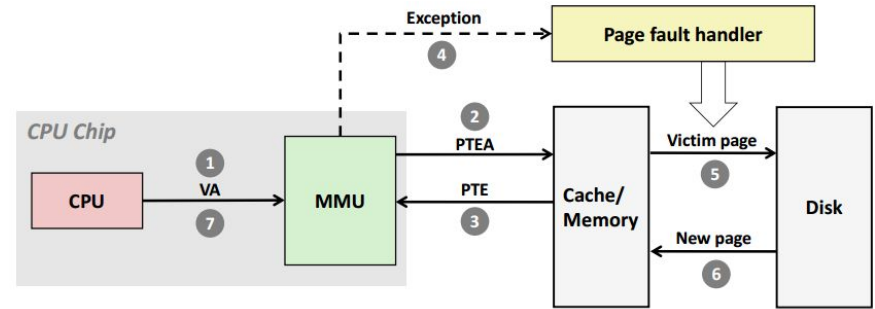
1) Processor sends virtual address to MMU
2-3) MMU Fetches Page Table Entry from page table in memory

4) Valid bit is zero; page fault exception!

5) Handler identifies victim (pages it out to disk)

6) Handler pages in new page and updates Page table entry in memory

7) Handler returns to original process, restarting from our 'faulty' instruction



Address Translation: Page Fault

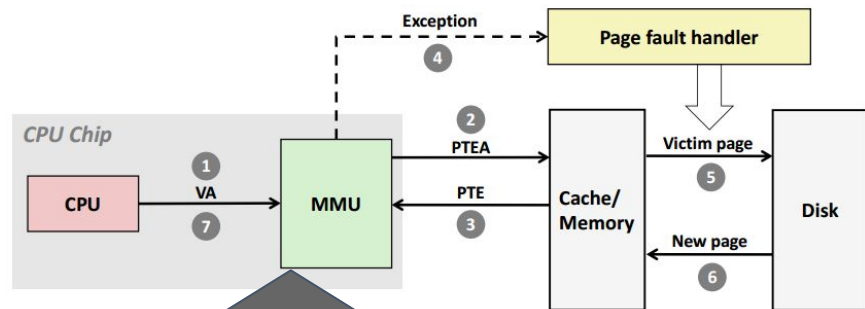
1) Processor sends virtual address to MMU
2-3) MMU Fetches Page Table Entry from page table in memory

4) Valid bit is zero; page fault exception!

5) Handler identifies victim (pages it out to disk)

6) Handler pages in new page and updates Page table entry in memory

7) Handler returns to original process, restarting from our 'faulty' instruction



7) At this point, we will get a hit, and resume (See previous "Page Hit" slides)

Address Translation: Page Fault

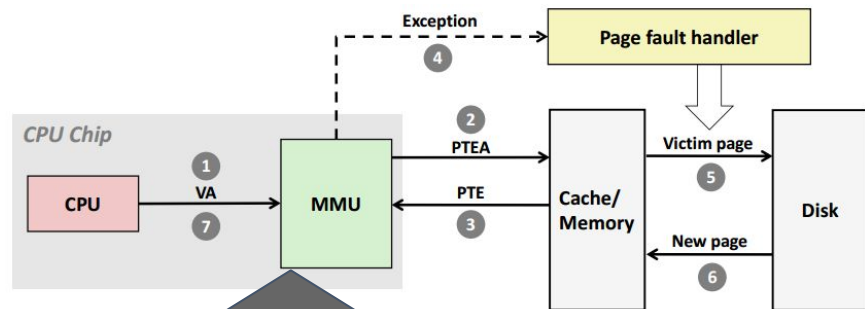
1) Processor sends virtual address to MMU
2-3) MMU Fetches Page Table Entry from page table in memory

4) Valid bit is zero; page fault exception!

5) Handler identifies victim (pages it out to disk)

6) Handler pages in new page and updates Page table entry in memory

7) Handler returns to original process, restarting from our 'faulty' instruction



7) At this point, we will get a hit, and resume (See previous "Page Hit" slides)

How can we speed up the process?

Let's speed up memory accesses

- Translation Lookaside Buffer (TLB)
 - It is called a buffer, but really it is a cache.
 - It's **a set-associative hardware cache in the Memory Management Unit (MMU).**
 - Contains complete page table entries for (some small amount) of pages.
- More simply defined:
 - The TLB - stores recent translations of virtual memory to physical addresses in a table
 - (The Translation Lookaside Buffer is part of the MMU system)

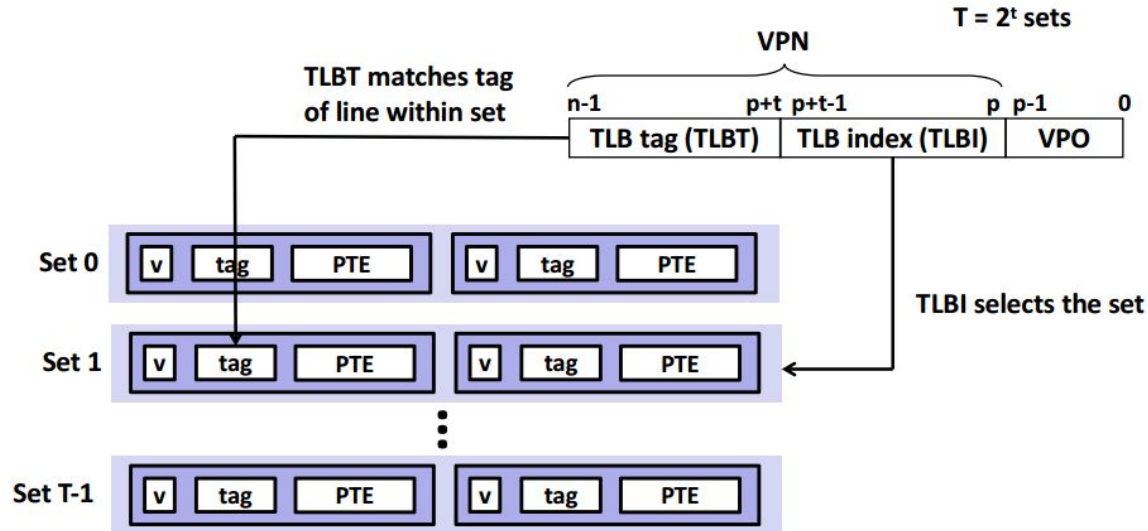
Address Translation - Notation

- Basic Parameters
 - $N=2^n$: Number of addresses in virtual address space
 - $M=2^m$: Number of addresses in physical address space
 - $P=2^p$: Page size (bytes)
- Components of virtual address (VA)
 - **TLBI: TLB index**
 - **TLBT: TLB tag**
 - VPO: Virtual page offset
 - VPN: Virtual page number
- Components of physical address (PA)
 - PPO: Physical page offset (same as VPO)
 - PPN: Physical page number



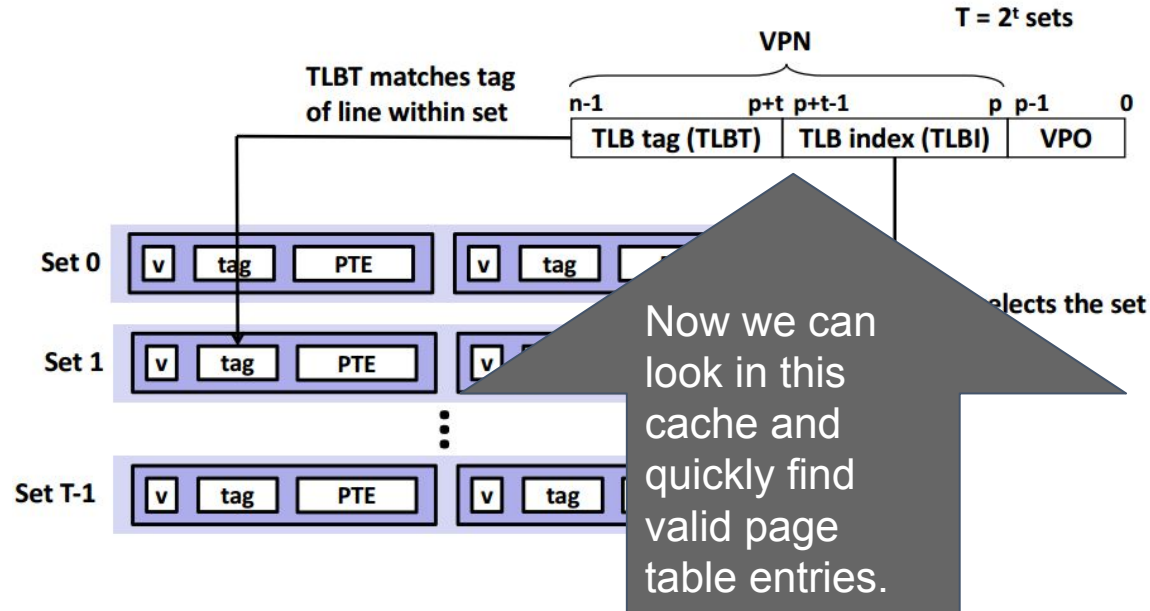
Accessing the Translation Lookaside Buffer (TLB)

- This looks quite familiar to our set-associative cache!



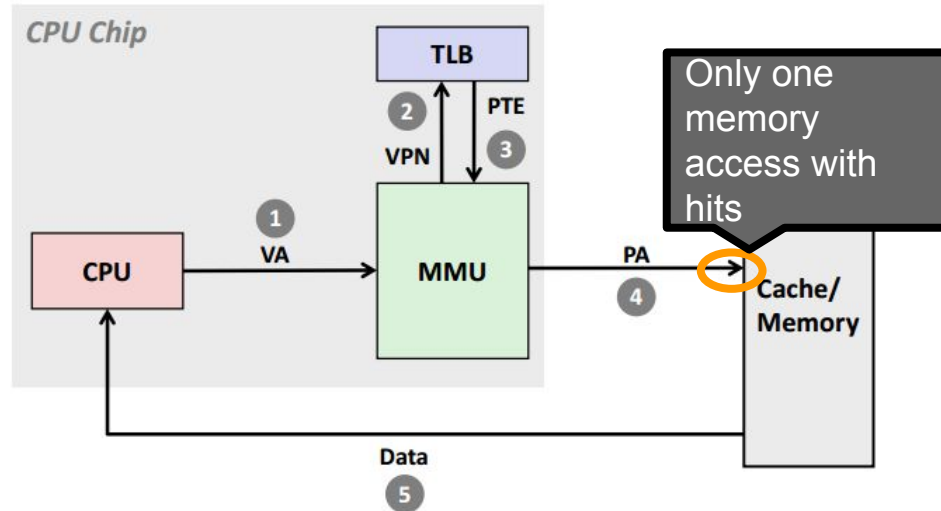
Accessing the Translation Lookaside Buffer (TLB)

- This looks quite familiar to our set-associative cache!



Translation Lookaside Buffer (TLB) Hit

- On a hit, we reduce by 1 memory access
- In practice, misses are rare
 - We pay an extra memory access if so



Summary of Virtual Memory

- Programmers
 - We see a process as owning a private linear address space
[easy to program]
 - Our address space cannot be corrupted by other processes
[isolation]
- System view of virtual memory
 - We use memory efficiently by caching our virtual memory pages
 - Locality saves the day!
 - Memory management and protection is significantly simplified
 - Different configurations could exist, such that we have multiple levels of paging.
 - (As always, there are trade-offs!)