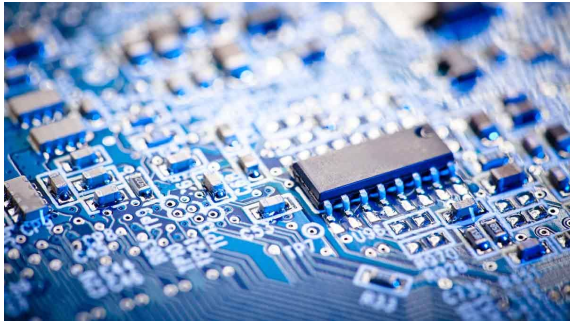
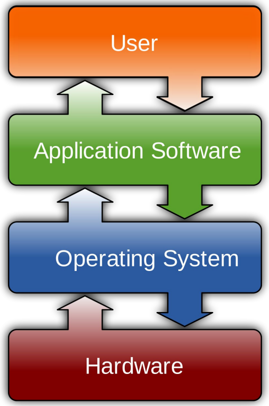
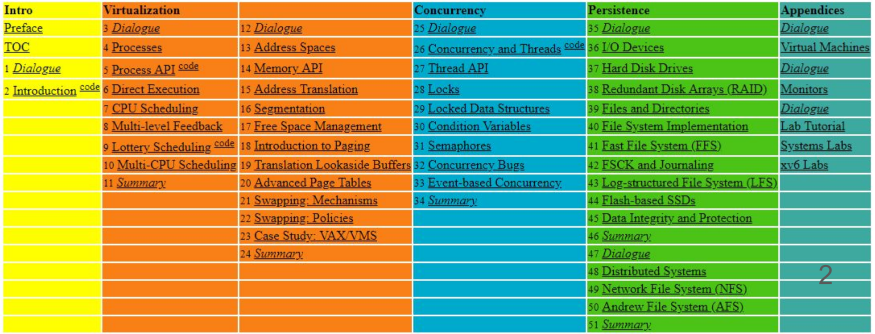


Applications
APP
APP
APP
APP
Monitor
Printer
Hard Drive
Operating System
Mouse
Keyboard

CS 3650
Computer Systems

Ferdinand Vesely - Alden Jackson



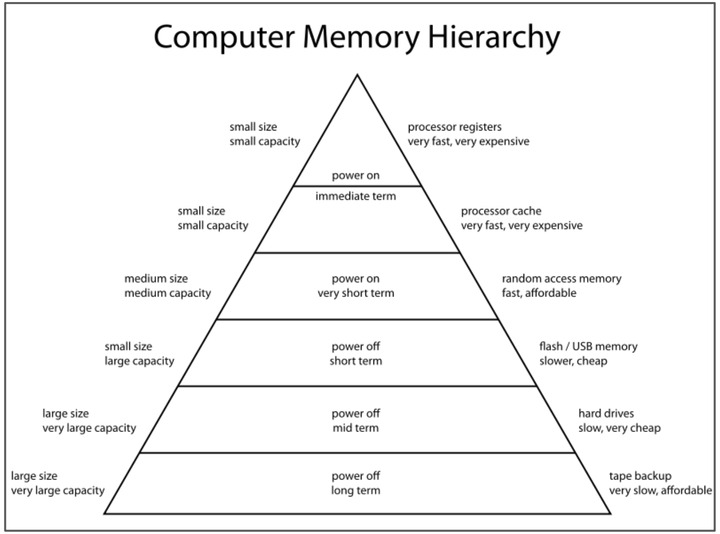


Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	
		22 Swapping: Policies		45 Data Integrity and Protection	
		23 Case Study: VAX/VMS		46 *Summary*	
		24 *Summary*		47 *Dialogue*	
				48 Distributed Systems	
				49 Network File System (NFS)	
				50 Andrew File System (AFS)	
				51 *Summary*	

An Introduction to Caches

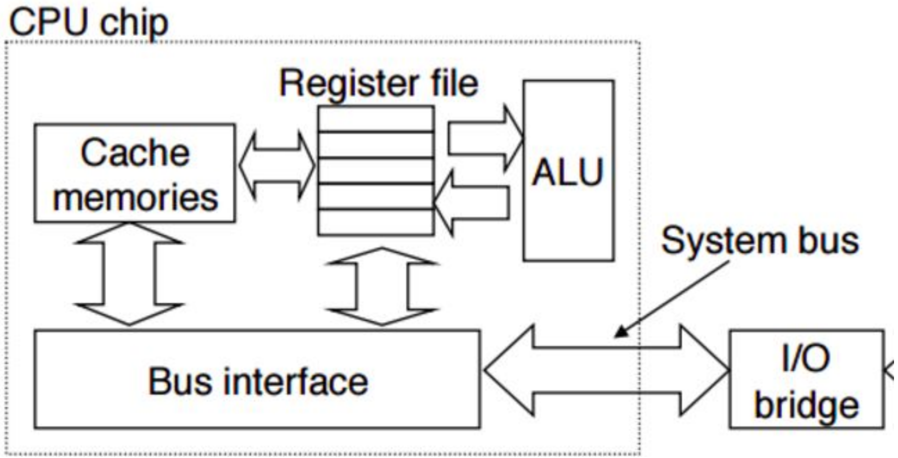
Cache

- Cache: A smaller, faster storage device that acts as 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 have a lot of the cheaper memory that holds a lot of data, and still access data at high speeds using our more limited but fast memory.



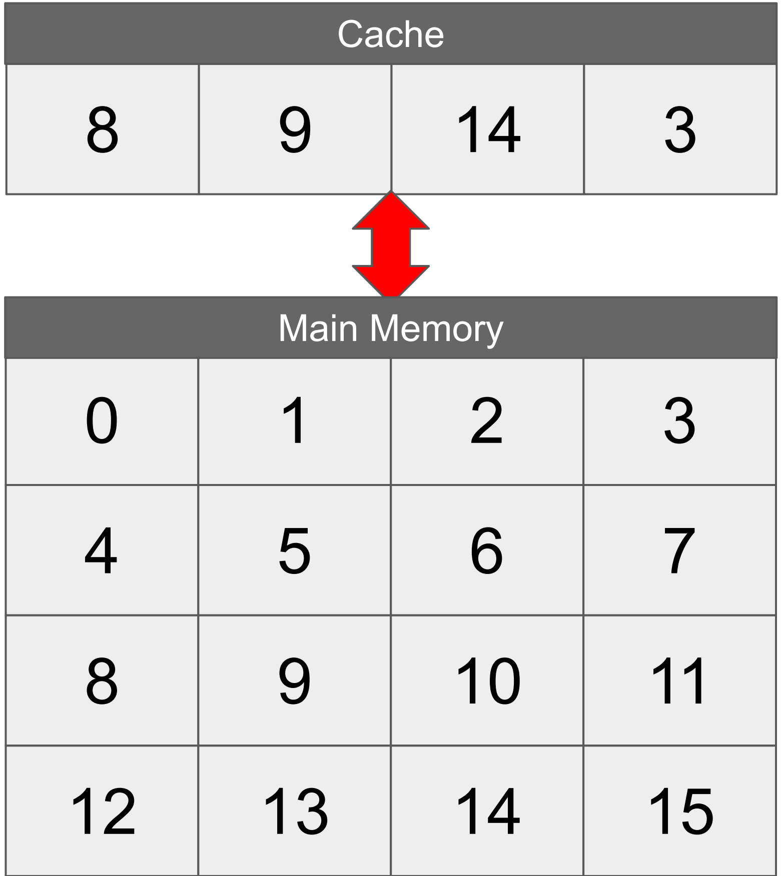
Cache on Hardware

- 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 (Earlier in lecture with the SSD and magnetic disk)



General Cache Concepts

Small Example



Cache			
8	9	14	3

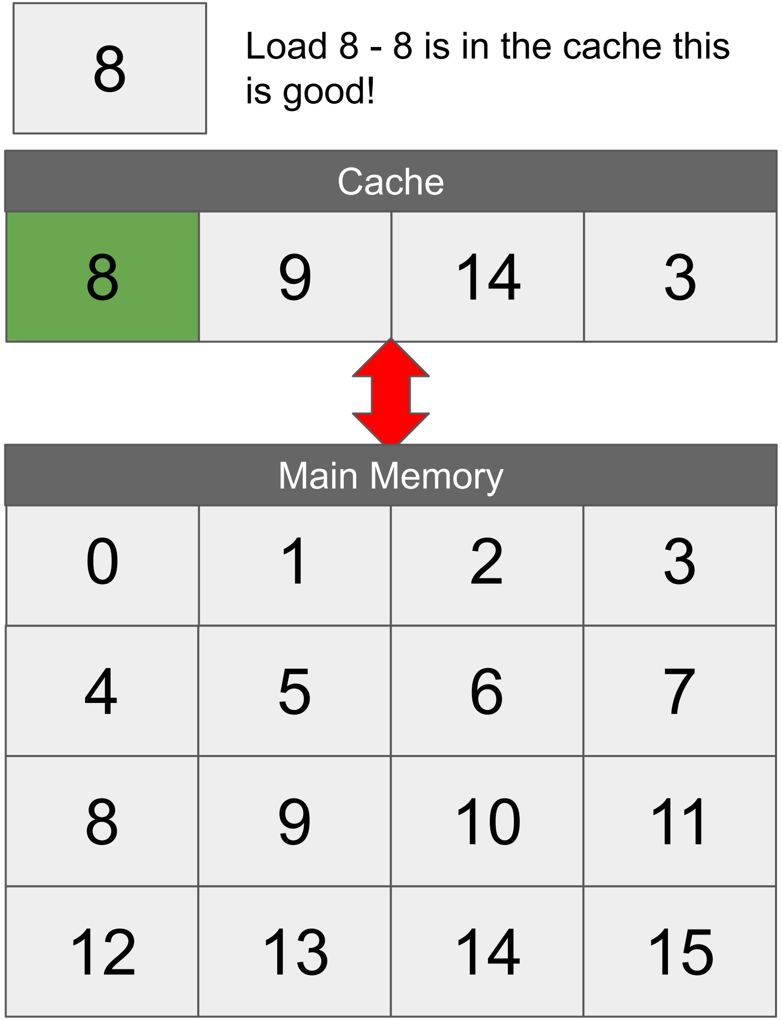
Main Memory			
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

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!

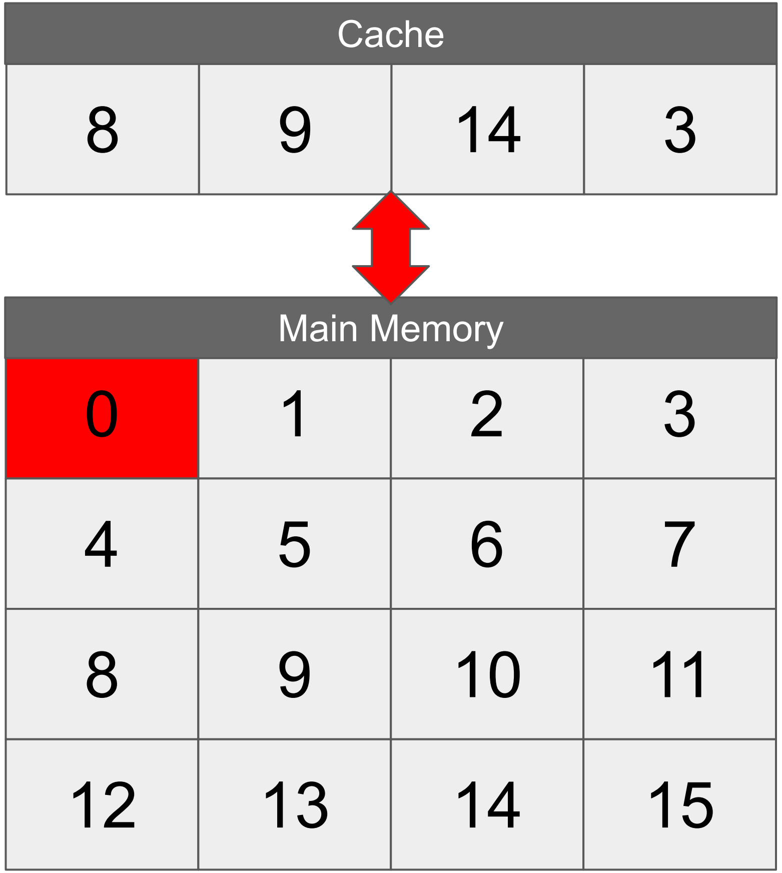
Cache			
🟩 8	9	14	3

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!

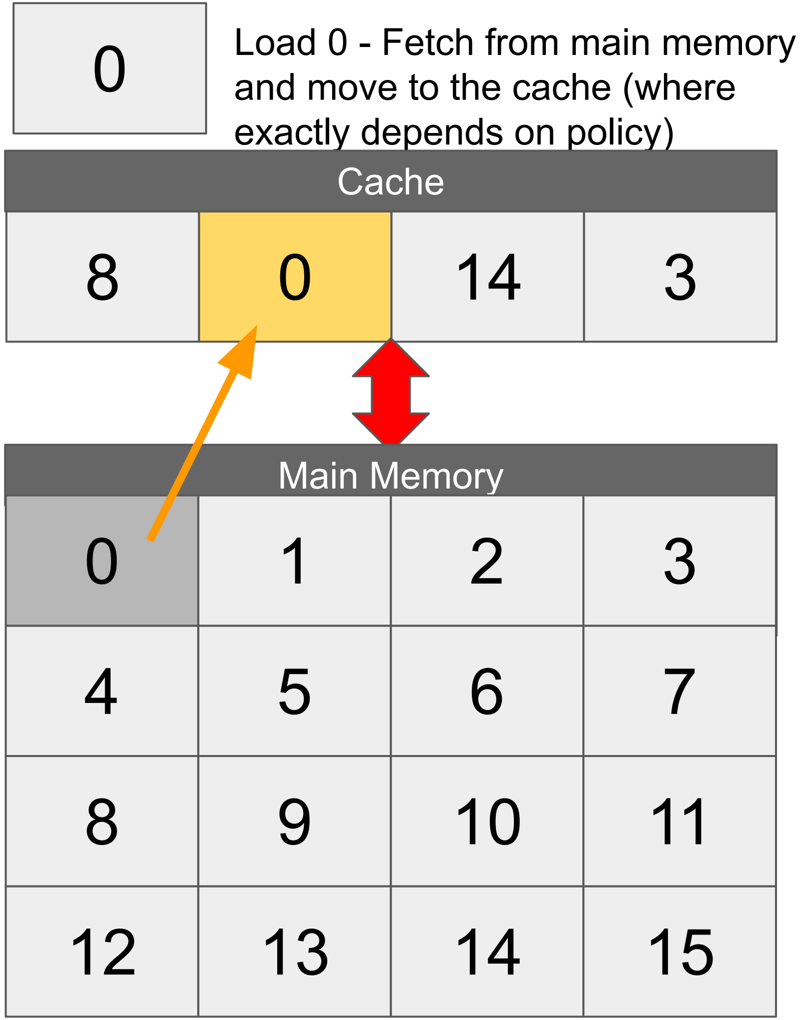


Cache			
8	9	14	3

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

Cache Miss

Load 0 - Fetch from main memory and move to the cache (where exactly depends on policy)



0

Cache

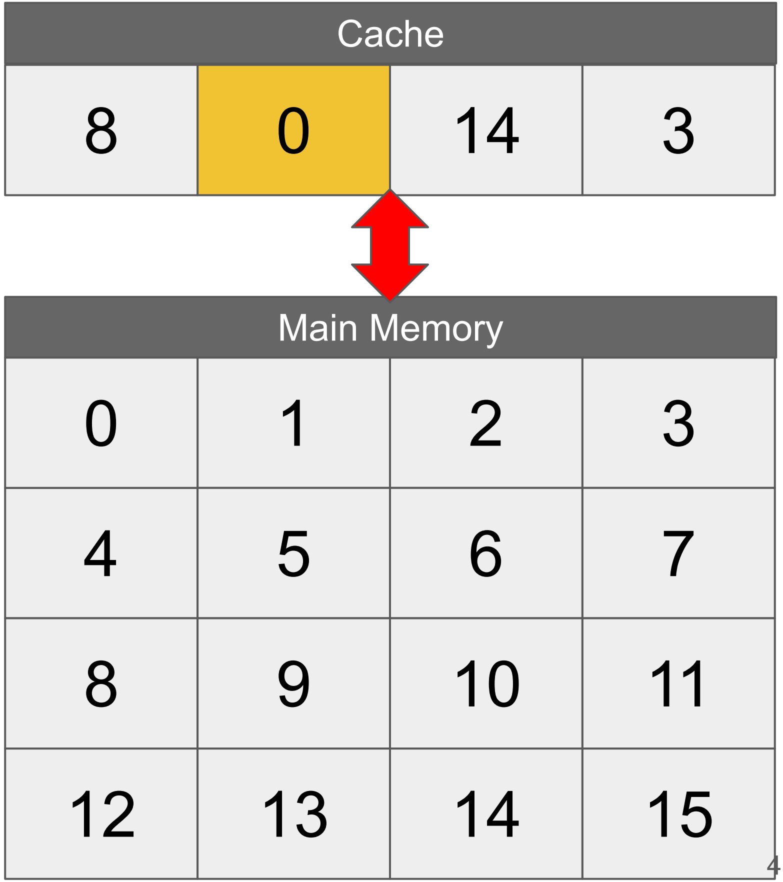
8	0	14	3

Main Memory

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

Note on Fetching

- From our perspective, when we fetch information, it is almost always worthwhile to put the memory into the cache.
- If you are going to pay some latency to retrieve something, might has well have it ready to go in the cache.
- The exact algorithm on how to replace and remove items depends on your policy.



Cache

8	0	14	3

Main Memory

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

Policies

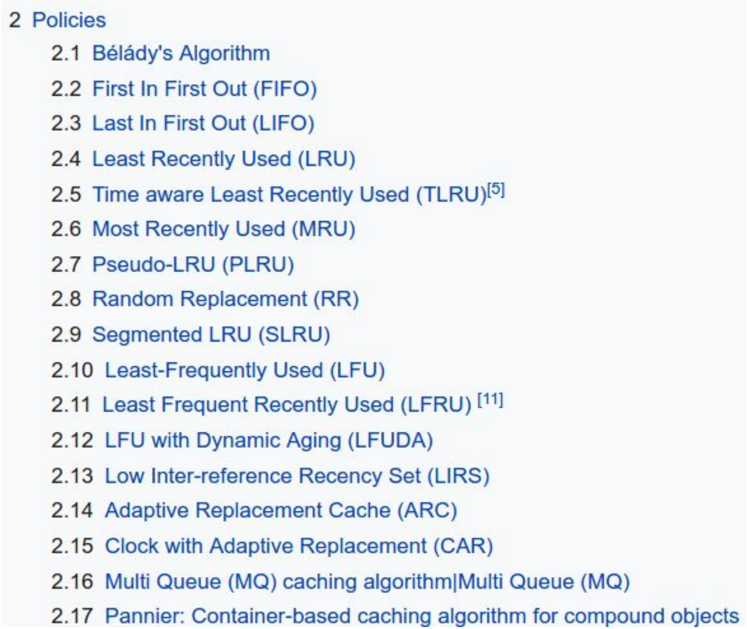
Now how I choose where to put that block is based on:

1. **Placement Policy** - Determine where blocks of memory go in the cache
2. **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

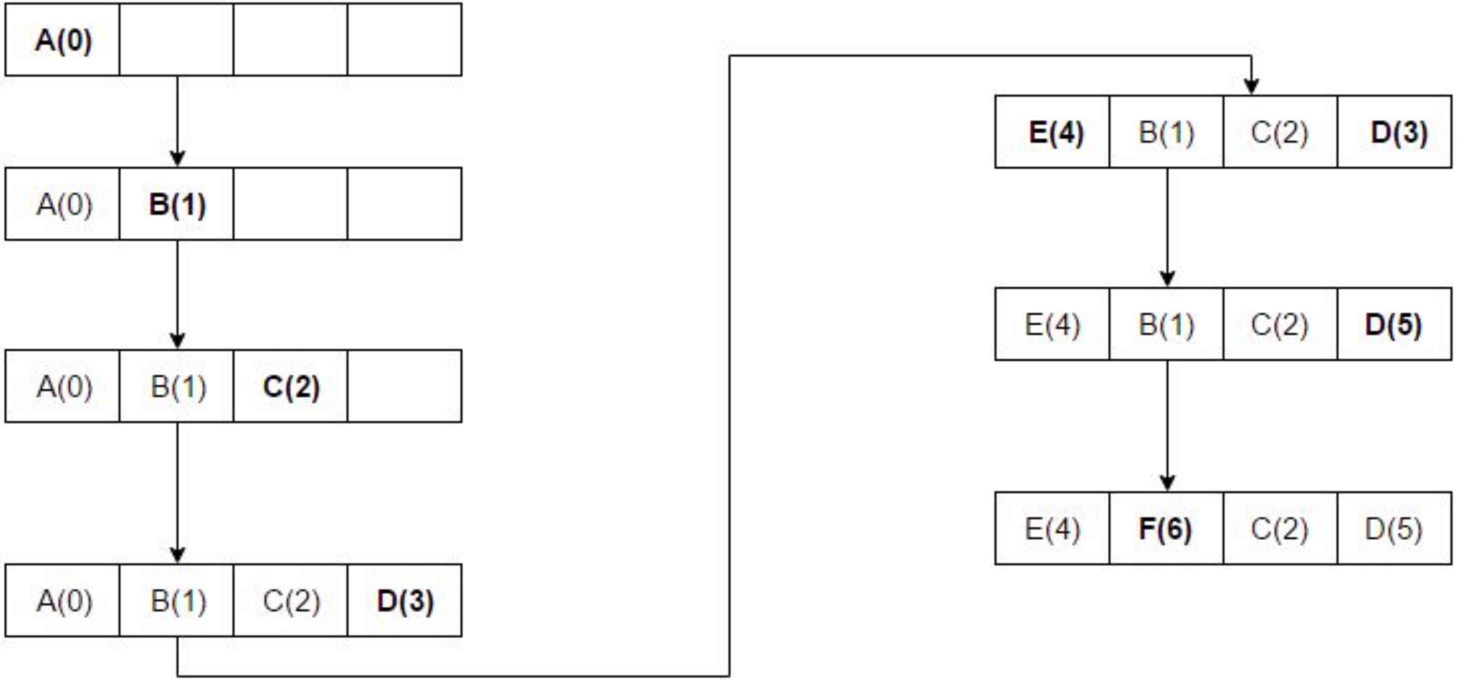
- Random - Just randomly remove something
- Least Recently Used (LRU) - Move out the youngest item.
- Here are some more:
  - https://en.wikipedia.org/wiki/Cache_replacement_policies



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 (Youngest Item)



A(0)			
A(0)	**B(1)**		
A(0)	B(1)	**C(2)**	
A(0)	B(1)	C(2)	**D(3)**
E(4)	B(1)	C(2)	**D(3)**
E(4)	B(1)	C(2)	**D(5)**
E(4)	**F(6)**	C(2)	D(5)

Cache Misses

1. **Cold (Compulsory) Miss** - First time you access a cache (perhaps when you start a program or fresh install of an operating system)
2. **Capacity Miss** - Set of the things you want to keep (your working set) is larger than the cache itself.
3. **Conflict Miss** - Occurs when the level K cache is large enough, but multiple data objects all map to the same level L block.
   - e.g. accessing two arrays that could fit in the cache,but are unaligned and due to organization do not fit.

Caches are everywhere!

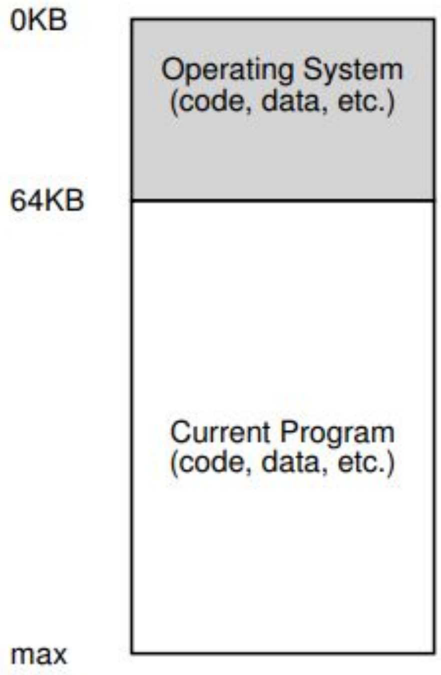
- Registers (Instruction Cache)
- L1 cache
- L2 cache
- Translation Lookaside Buffer (TLB)
- Virtual Memory
- Buffer Cache
- Disk Cache
- Network buffer cache
- Browser cache
- Web Cache, CDNs, ...

Lecture 7 - Virtual Memory

A trip down memory lane to the good old days

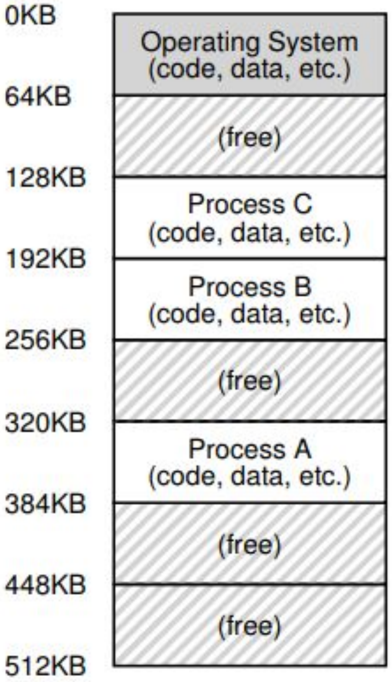
Early Computers

- 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


0KB
Operating System (code, data, etc.)
64KB
Current Program (code, data, etc.)
max

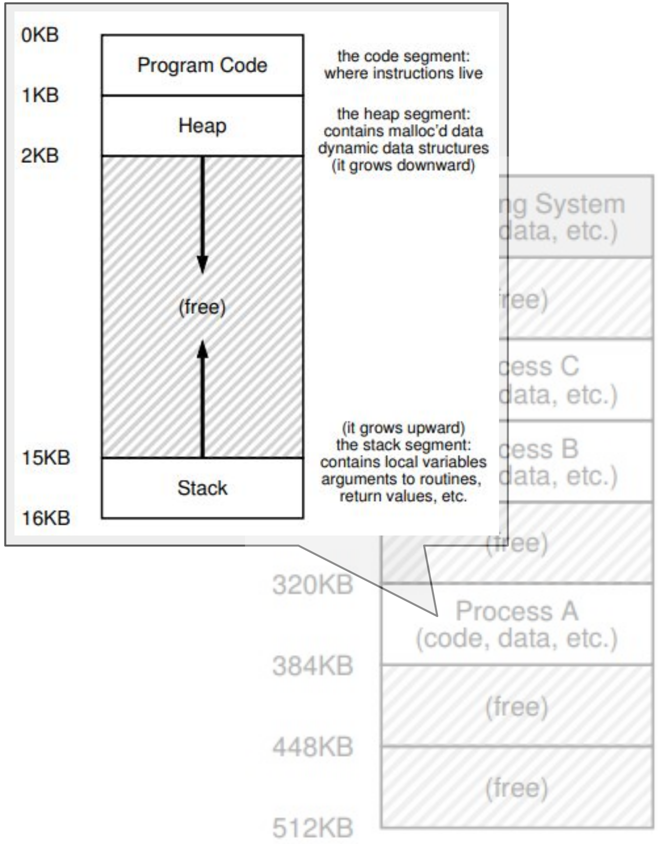
Sharing Memory

- Eventually 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.


0KB
Operating System (code, data, etc.)
64KB
(free)
128KB
Process C (code, data, etc.)
192KB
Process B (code, data, etc.)
256KB
(free)
320KB
Process A (code, data, etc.)
384KB
(free)
448KB
(free)
512KB

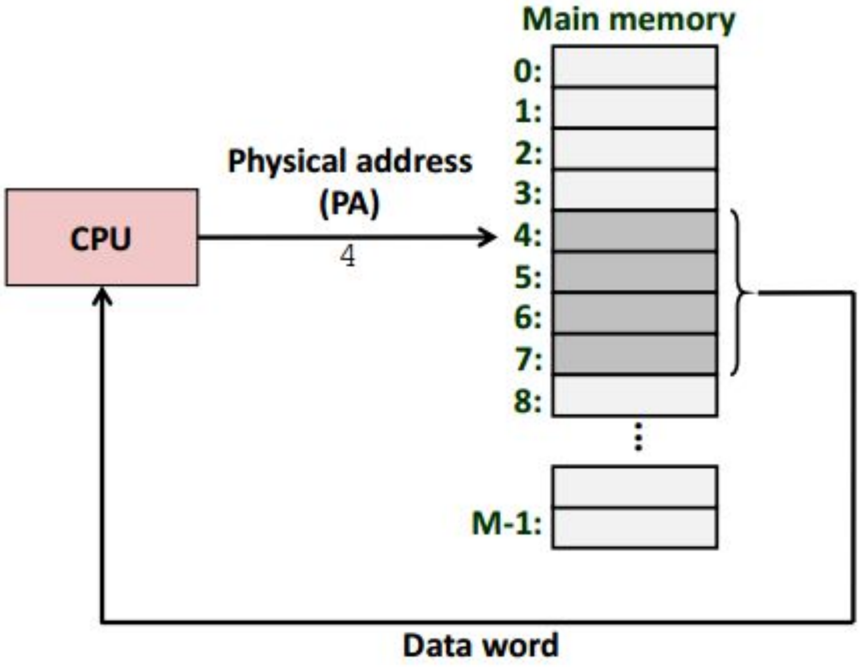
More efficient memory

- But eventually, programmers did not want to have a ‘fixed’ size memory block.
  - Maybe one process needed more or less memory than the other
- Thus processes needed a way to expand and compress based on how much memory was being used.
  - This was also a more efficient way to utilize memory.


0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
ng System data, etc.)
(free)
cess C data, etc.)
cess B data, etc.)
(free)
Process A (code, data, etc.)
(free)
(free)
320KB
384KB
448KB
512KB

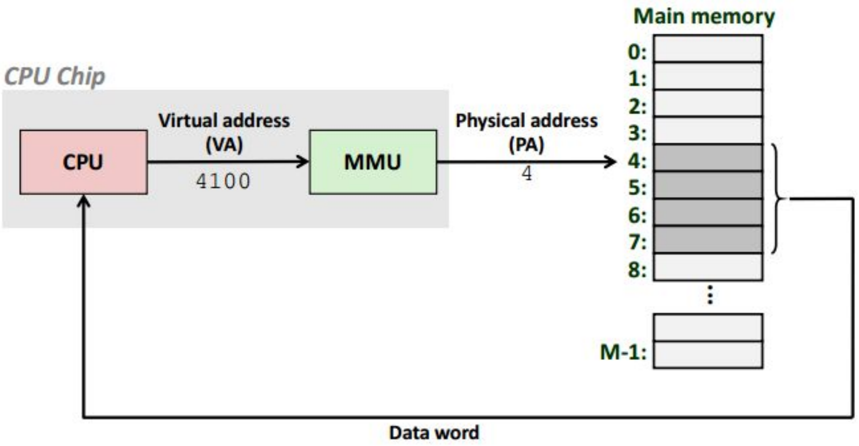
Physical Memory System

- This visualization shows how we have thought of memory
  - Our CPU fetches, decodes, and executes instructions one at a time from memory
  - That memory has *some* address
  - (And this may be a true depiction of what a small embedded processor looks like)
- In this model, what hardware mechanism could help a process to expand its memory?
  - How can the hardware support this? (next slide!)



Introducing the Memory Management Unit (MMU)

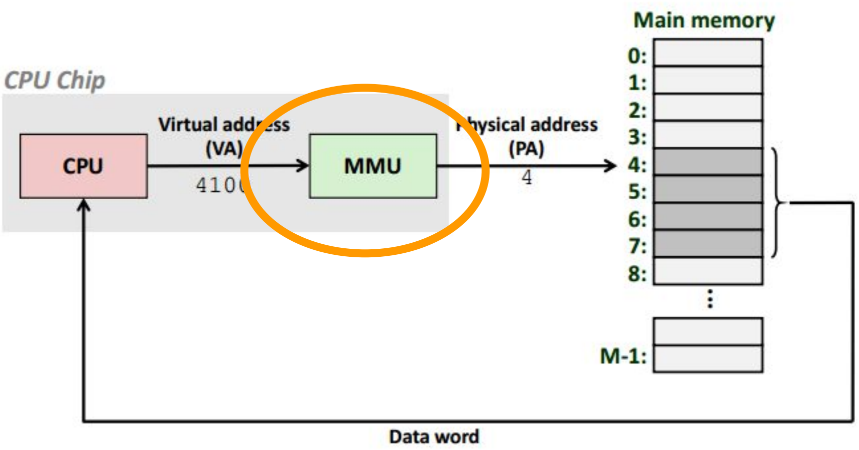
- We still retrieve memory from main memory
- BUT, there is an additional translation step that occurs in the Memory Management Unit (MMU)
- (And as with many things--we have introduced a new layer of abstraction in the hardware to help us)



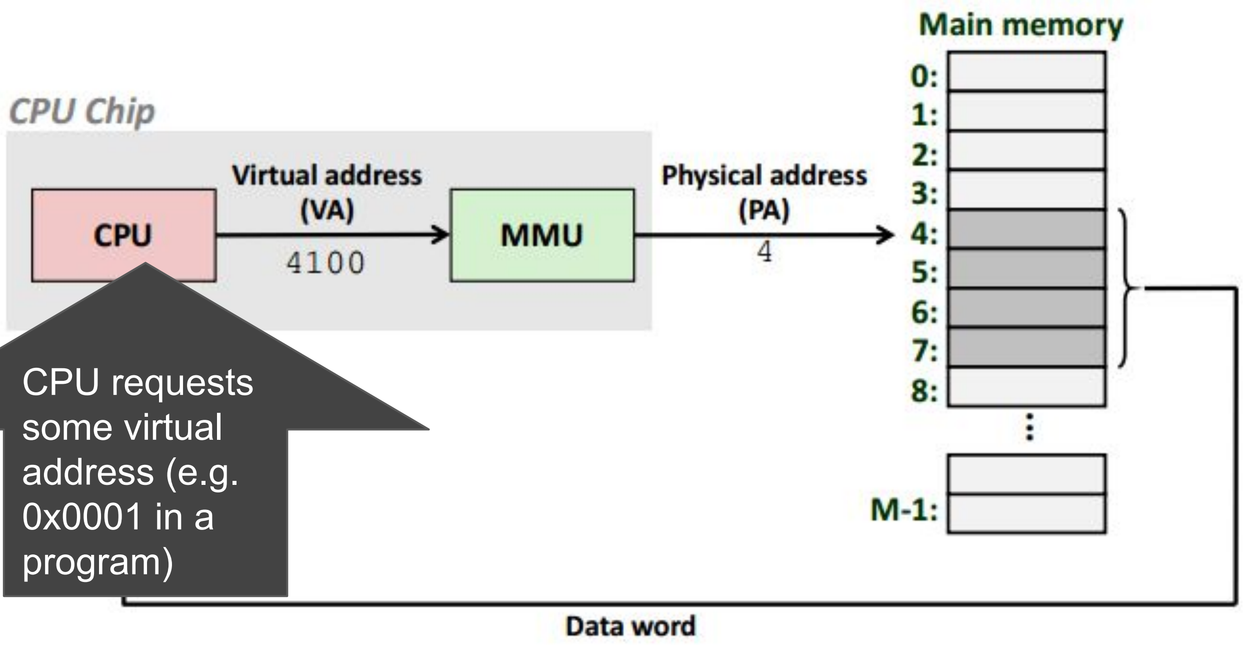
Memory Management Unit (MMU)

MMU's job is to figure out (i.e. translate) the mappings from main memory to what is called a virtual address for a process.

- When the address is determined, the 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)

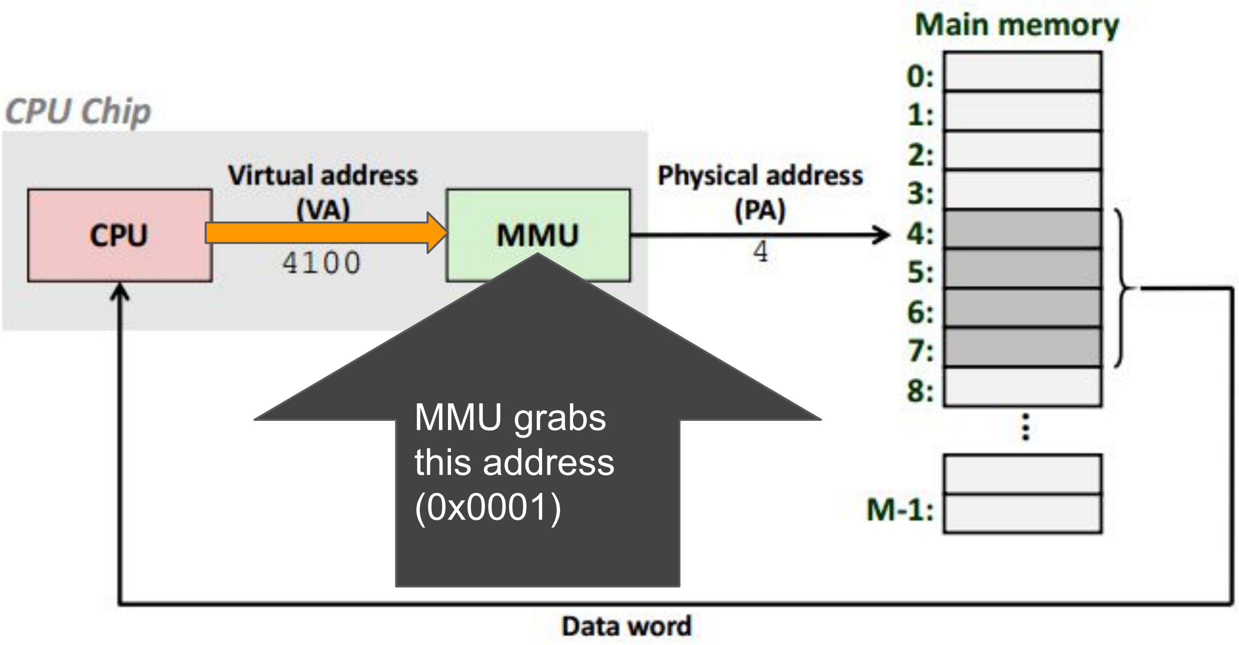


Memory Management Unit (MMU)



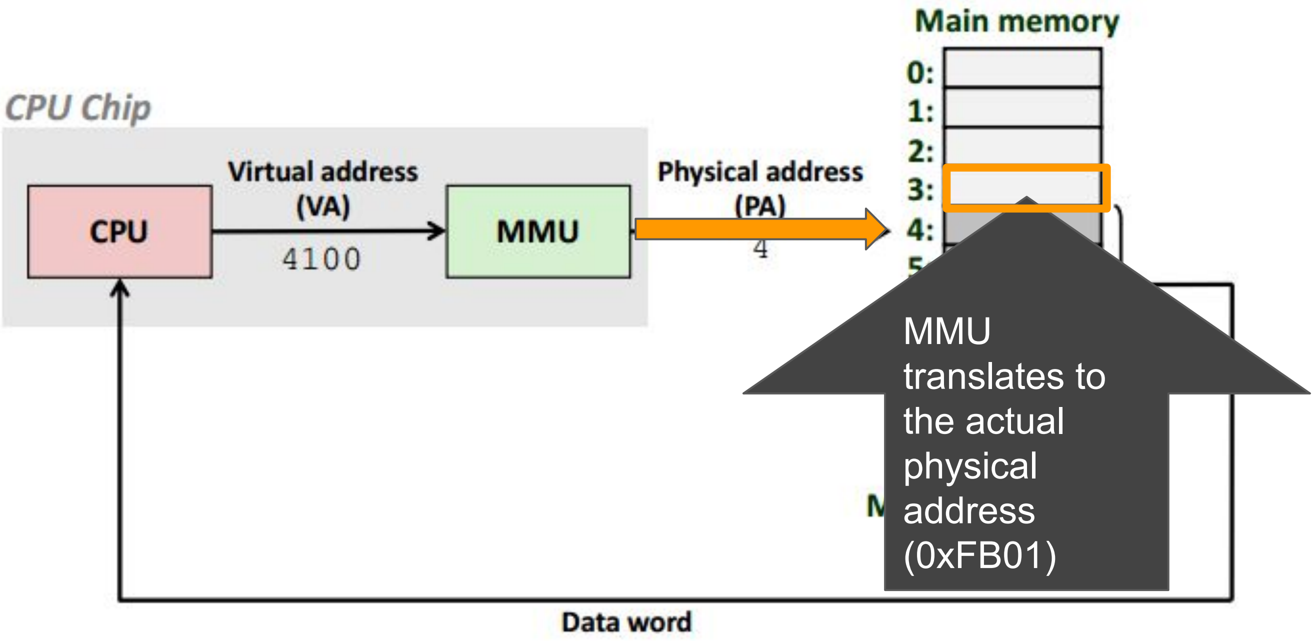
CPU requests some virtual address (e.g. 0x0001 in a program)

Memory Management Unit (MMU)



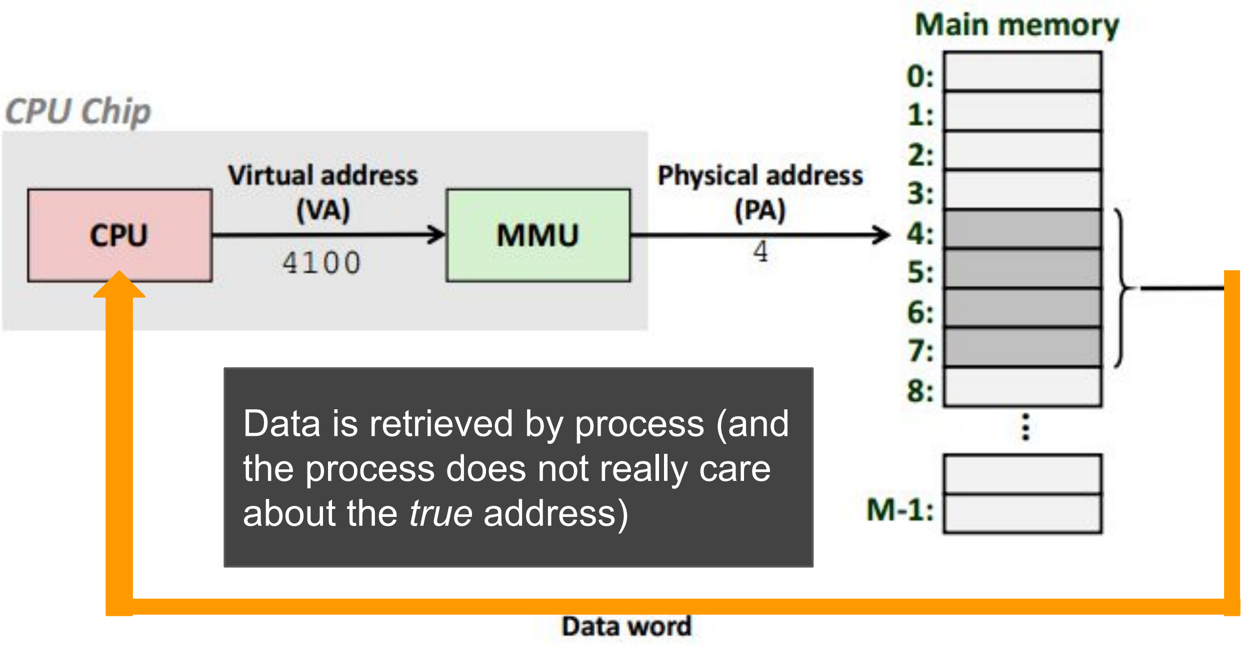
MMU grabs this address (0x0001)

Memory Management Unit (MMU)



MMU translates to the actual physical address (0xFB01)

Memory Management Unit (MMU)



Data is retrieved by process (and the process does not really care about the *true* address)

Fritz-Rudolf Güntsch

- German Physicists
    - (i.e. not a computer scientist)
- Invented the concept of virtual memory in the 1950s
- https://history-computer.com/ModernComputer/Electronic/Atlas.html

Virtual Memory

Three Virtual Memory Advantages

1. **Use Main memory efficiently**

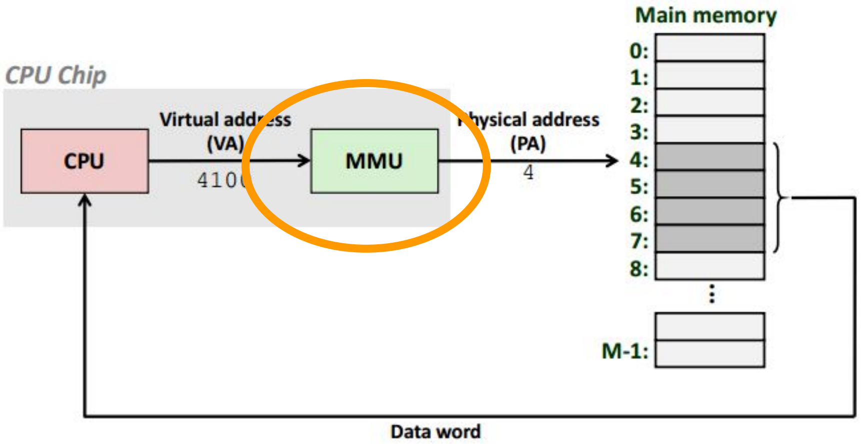
2. **Simplifies memory management (for application developers)**

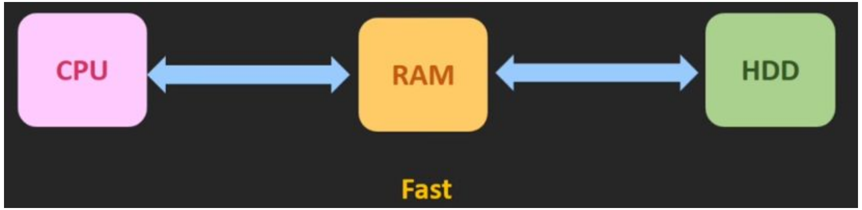
3. **Isolates Address Spaces**

Why Virtual Memory (1/3)

1. **Uses main memory efficiently**

   - Use physical memory as a “cache” for parts of a virtual address space
   - The picture to the left looks a lot like the picture to the right in that there is another layer of abstraction

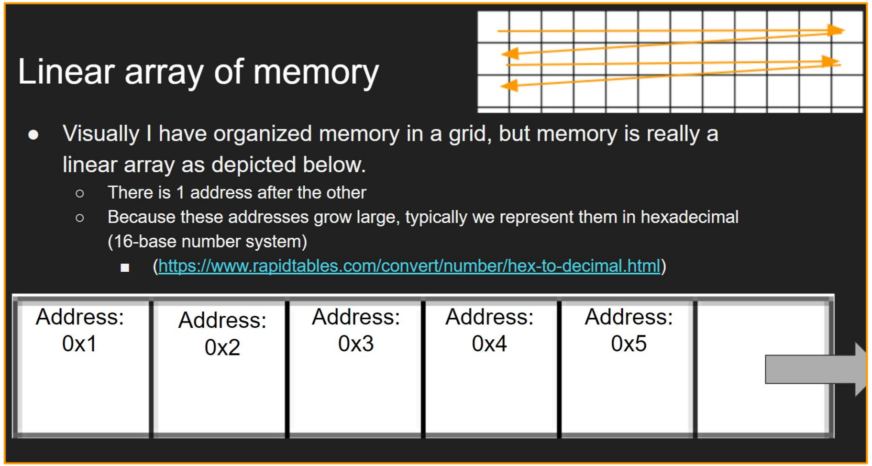




Why Virtual Memory (2/3)

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)



Linear array of memory

- Visually I have organized memory in a grid, but memory is really a linear array as depicted below.
  - There is 1 address after the other
  - Because these addresses grow large, typically we represent them in hexadecimal (16-base number system)
    - (https://www.rapidtables.com/convert/number/hex-to-decimal.html)

Address: 0x1

Address: 0x2

Address: 0x3

Address: 0x4

Address: 0x5

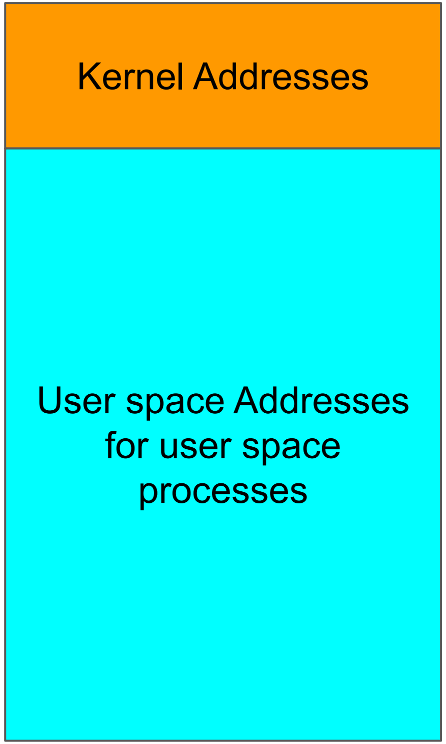
Why Virtual Memory (3/3)

3. Isolates Address Spaces

- One process cannot interfere with another
- User’s program cannot access privileged kernel information and code.
  - That is, imagine we did have access to our whole disk/ram and could return bytes from anywhere!
- 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.



Kernel Addresses

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

Two Address Spaces

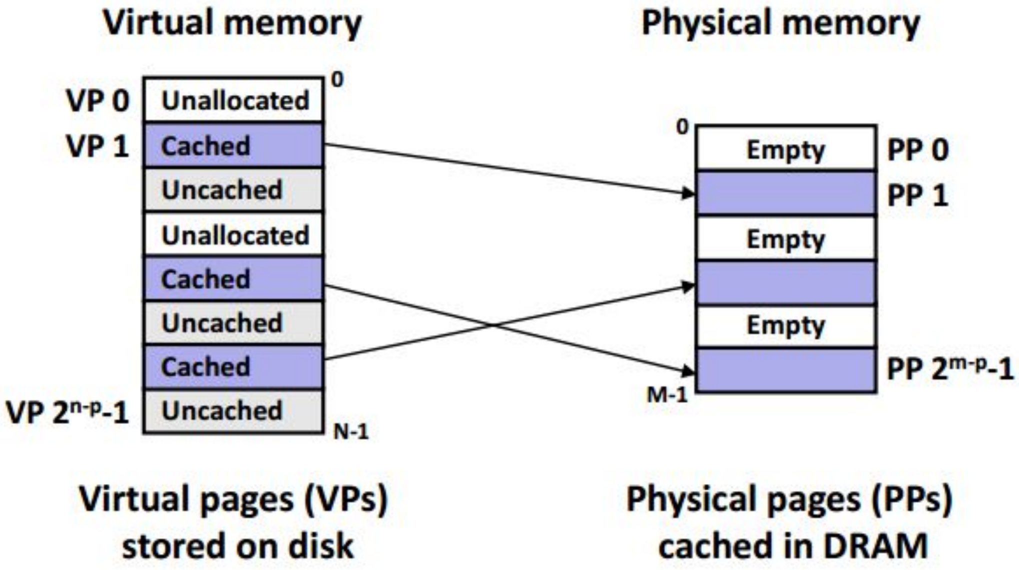
1. Physical Address Space

   Is used by the hardware

2. Virtual Addresses Space
   - Used by the software
   - (Again, this is what we are familiar with)
   - The exact translation (from a physical to a virtual address) happens in hardware for us

Virtual Memory to assist with caching (1/5)

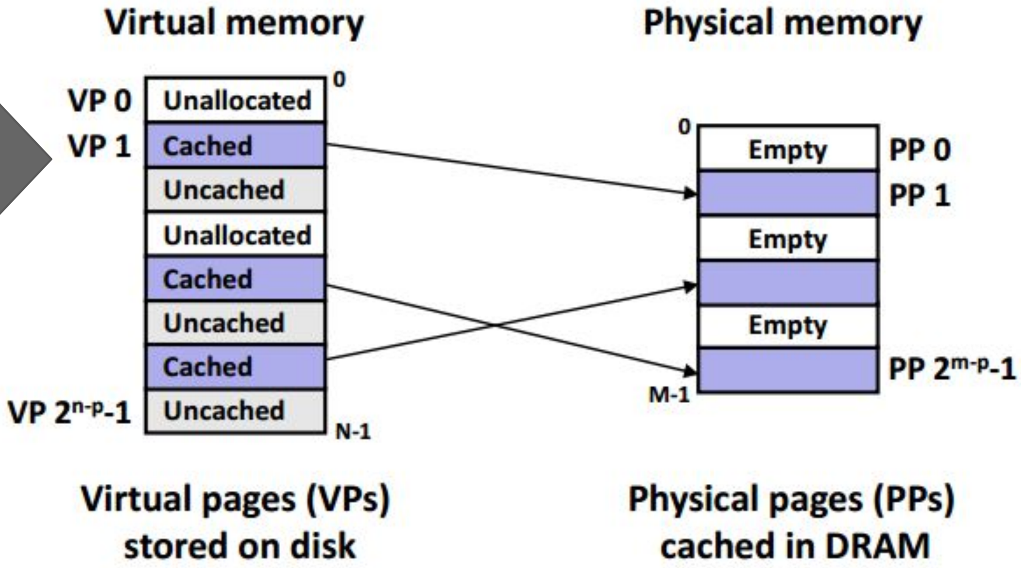
- Conceptually, virtual memory is an array of contiguous bytes stored on disk
- The contents of these arrays are cached in physical memory



Virtual Memory to assist with caching (2/5)

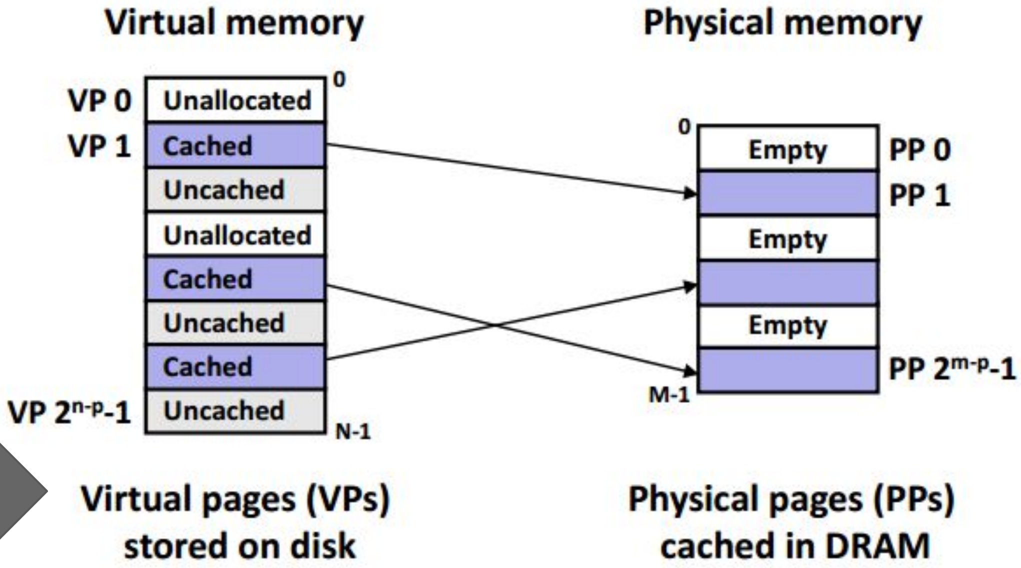
- Conceptually, virtual memory is an array of contiguous bytes stored on disk
- The contents of these arrays are cached in physical memory

I am taking these large ‘blocks’(pages) of memory



Virtual Memory to assist with caching (3/5)

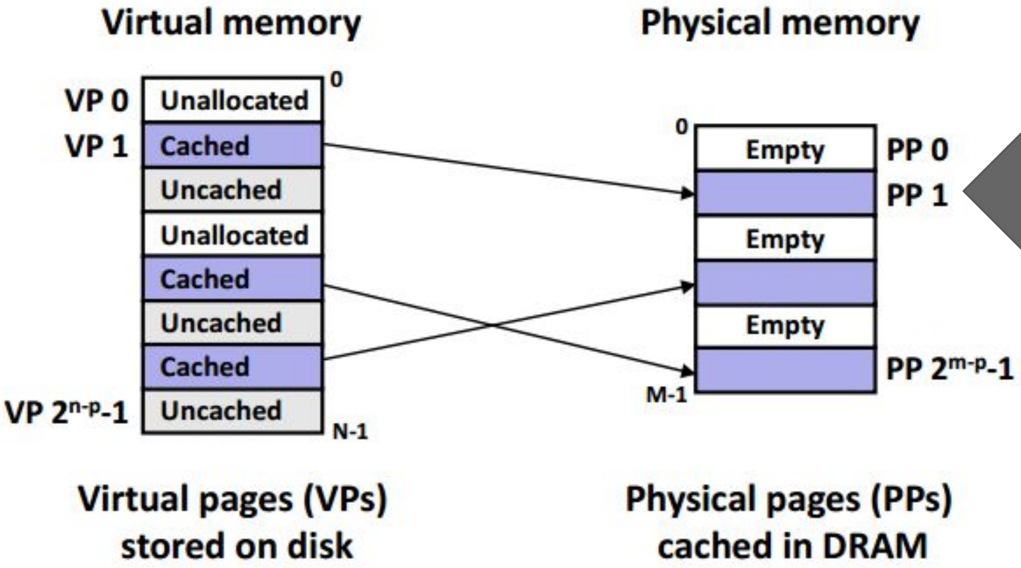
- Conceptually, virtual memory is an array of contiguous bytes stored on disk
- The contents of these arrays are cached in physical memory



They are stored on our slow disk

Virtual Memory to assist with caching (4/5)

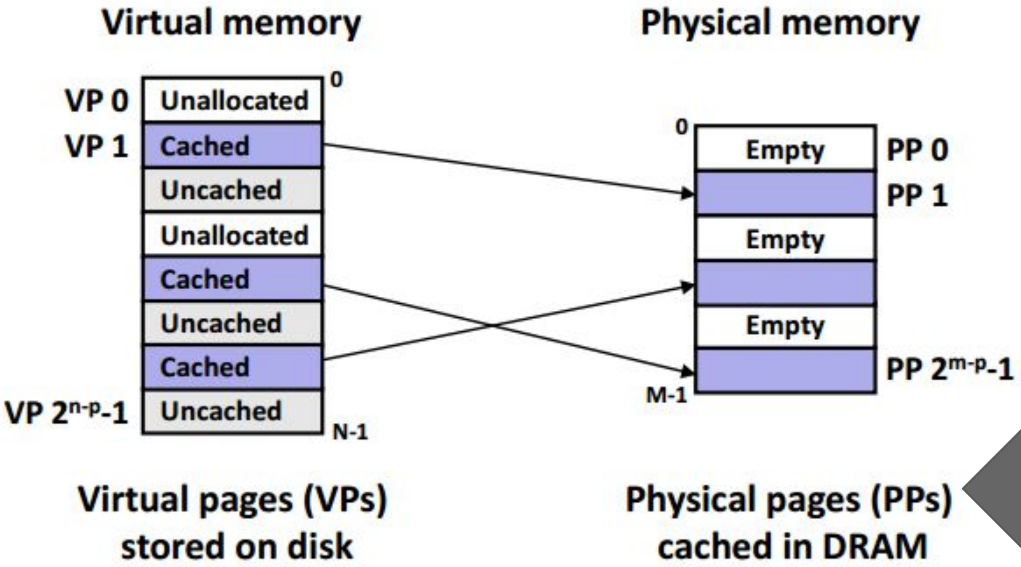
- Conceptually, virtual memory is an array of contiguous bytes stored on disk
- The contents of these arrays are cached in physical memory



Now I have put this large block ('page') of memory into faster memory (DRAM)

Virtual Memory to assist with caching (5/5)

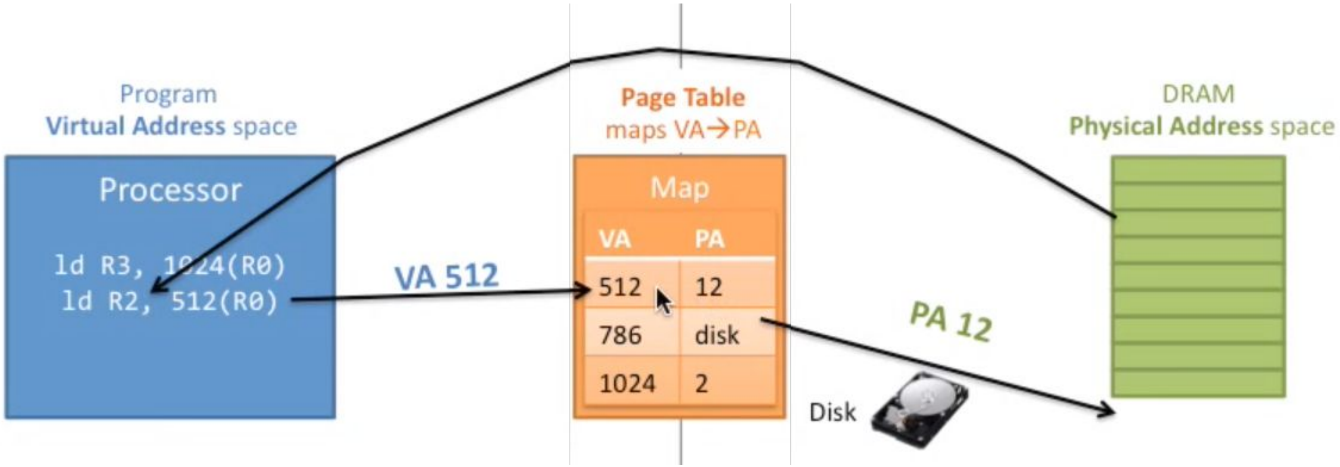
- Conceptually, virtual memory is an array of contiguous bytes stored on disk
- The contents of these arrays are cached in physical memory



Our DRAM is faster than disk

Introducing the Page Table!

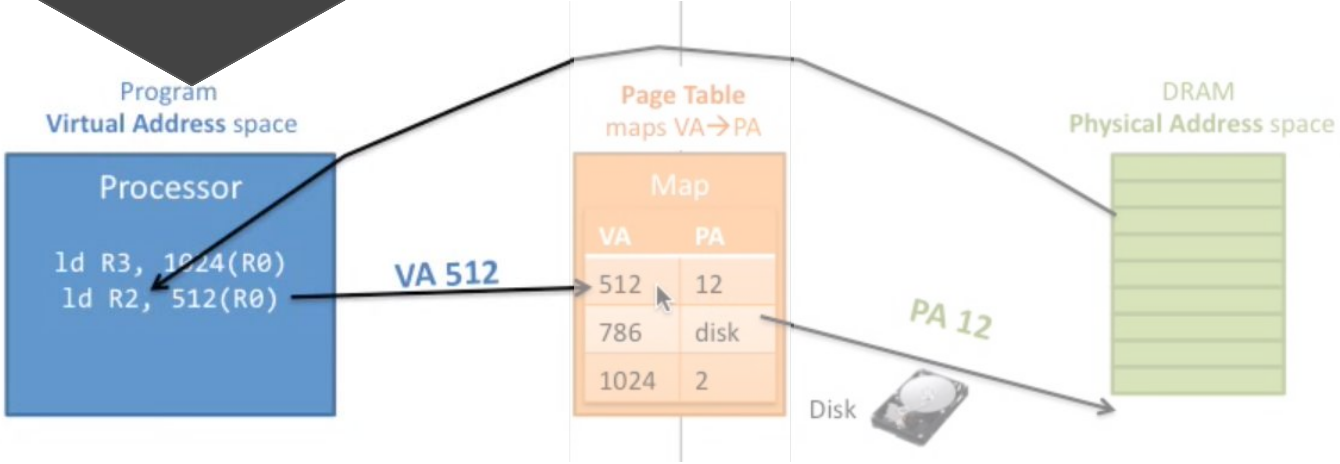
- A page table keeps track of the mapping between virtual and physical addresses. [figure source]



Introducing the Page Table!

- A page table keeps track of the mapping between virtual and physical addres urce]

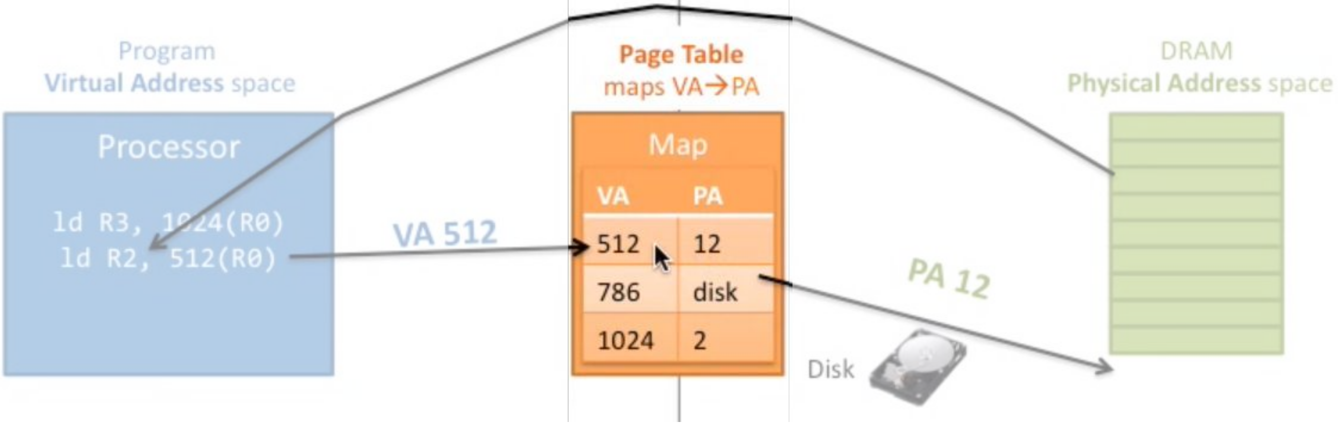
Our process requests some address (which is actually a virtual address)



Introducing the Page Table!

A page table keeps track of the etween virtual and physical addresses. [figure source]

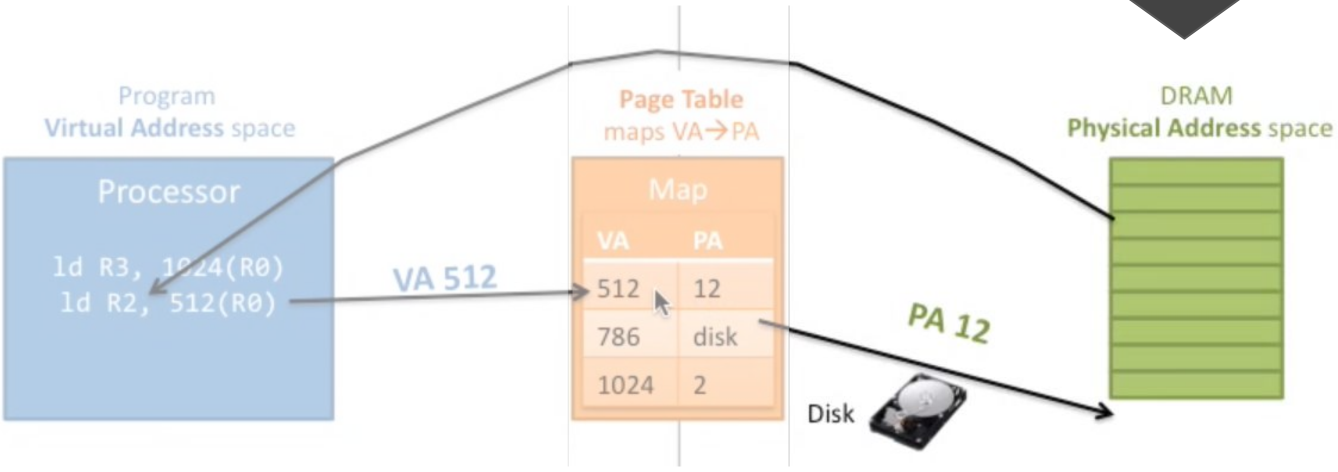
The Page Table maps us to the real physical address in DRAM



Introducing the Page Table!

- A page table keeps track of the mapping between virtu ical addresses. [figure source]

And we retrieve the actual data we need from DRAM.



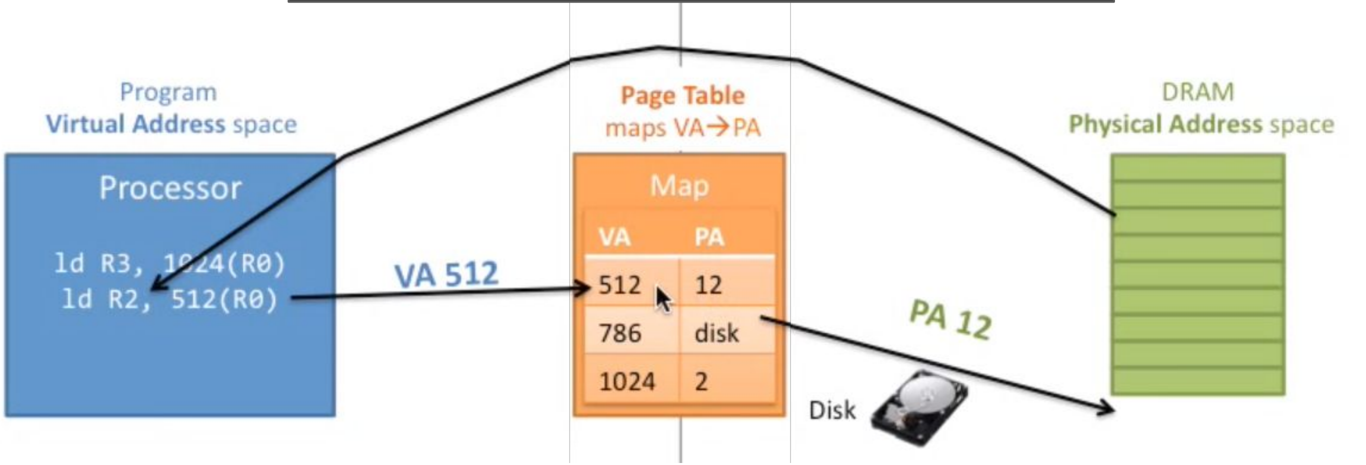
VA	PA
512	12
786	disk
1024	2

Introducing the Page Table!

- A page table keep and physical addresses. [figure

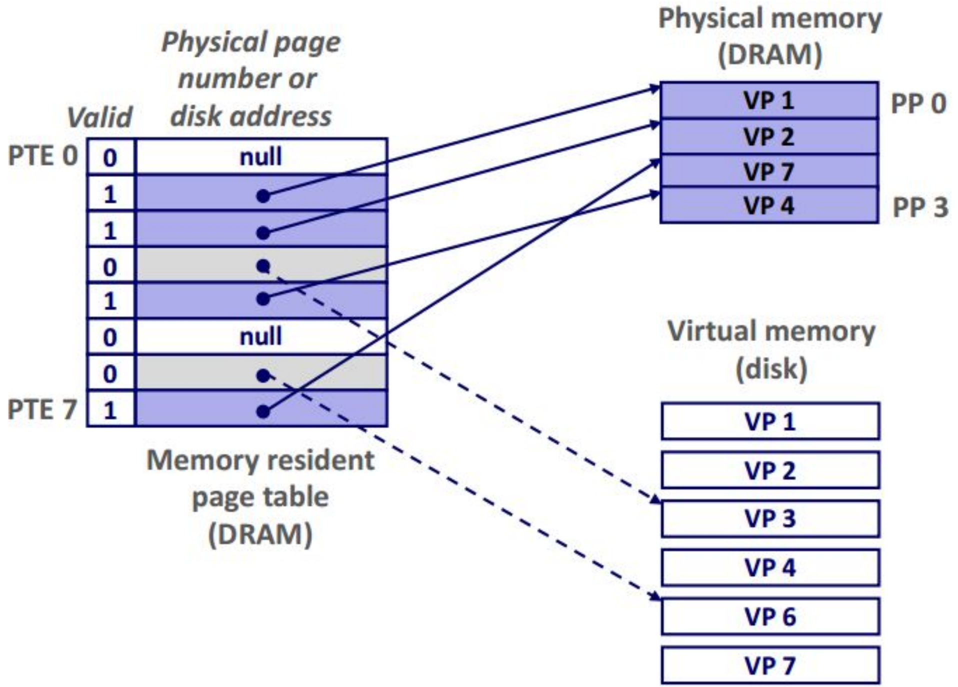
Now remember, we are actually looking up ‘pages’.

(Otherwise we would have lots of 1 byte entries--which would make our page table huge!)



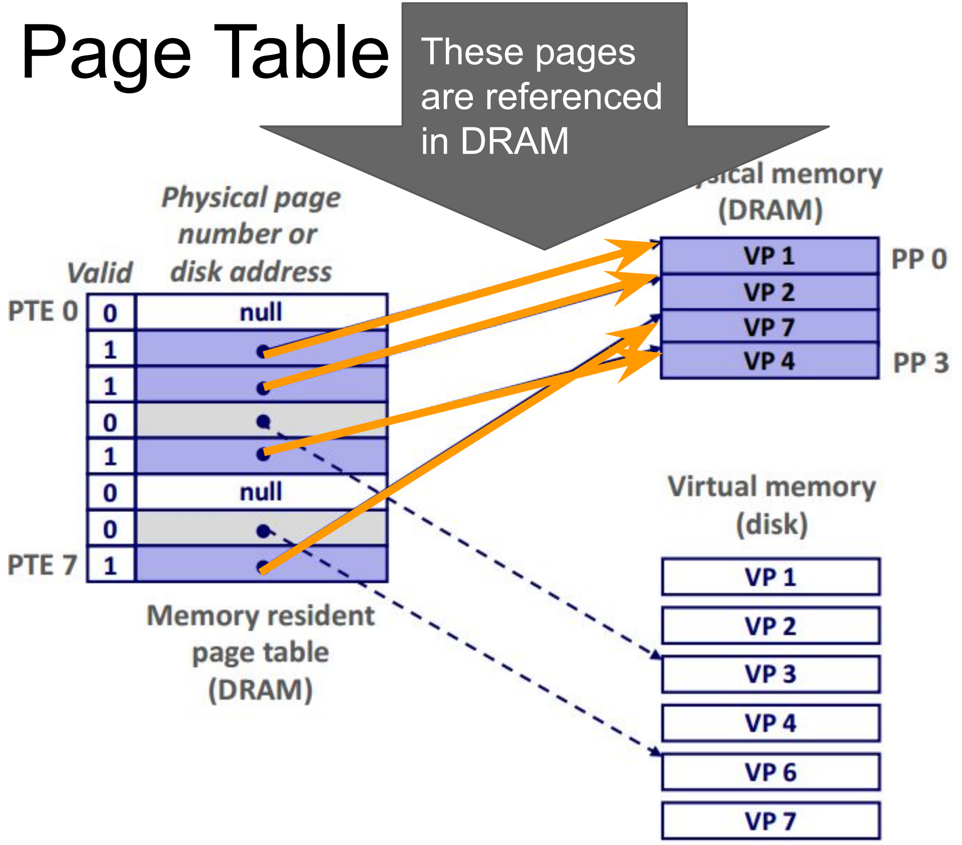
(Again) Enabling Data Structure: Page Table

- We divide memory up into pages
  - (Typically 4096 bytes for 1 page)
- A page table then stores the mappings from a virtual page to its physical page address



Enabling Data Structure: Page Table

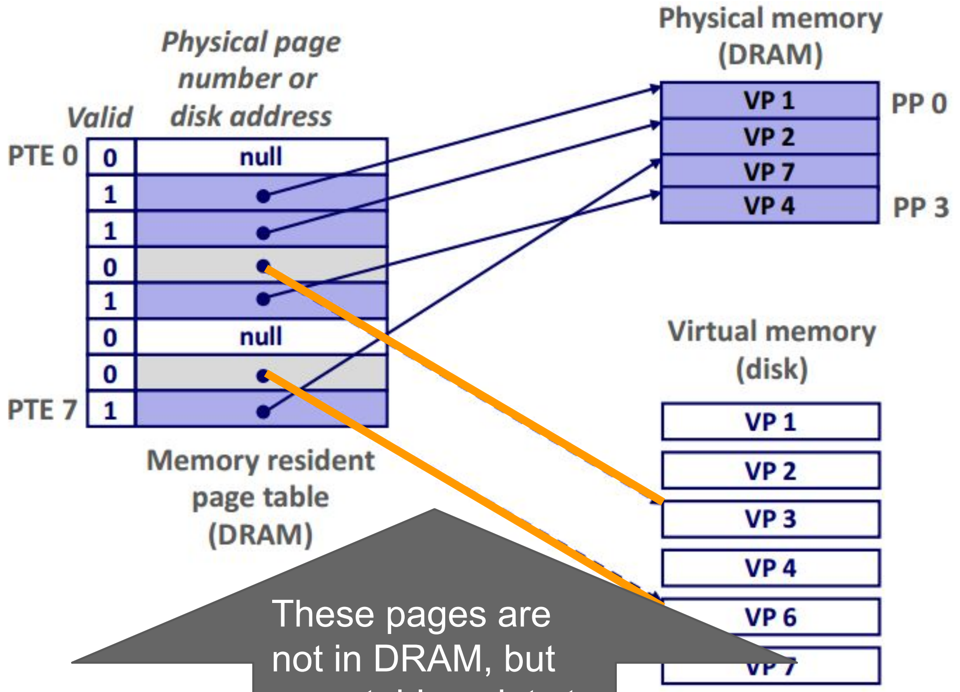
- We divide memory up into pages
  - (Typically 4096 bytes for 1 page)
- A page table then stores the mappings from a virtual page to its physical page address



These pages are referenced in DRAM

Enabling Data Structure: Page Table

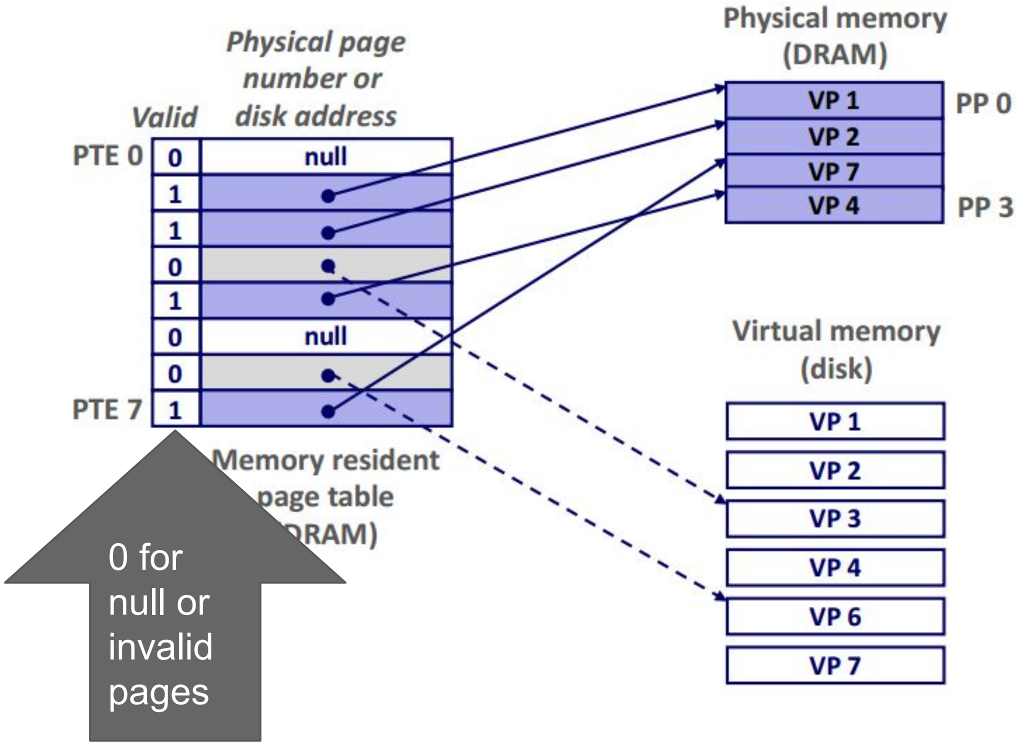
- We divide memory up into pages
  - (Typically 4096 bytes for 1 page)
- A page table then stores the mappings from a virtual page to its physical page address



These pages are not in DRAM, but page table points to where on disk virtual memory is

Enabling Data Structure: Page Table

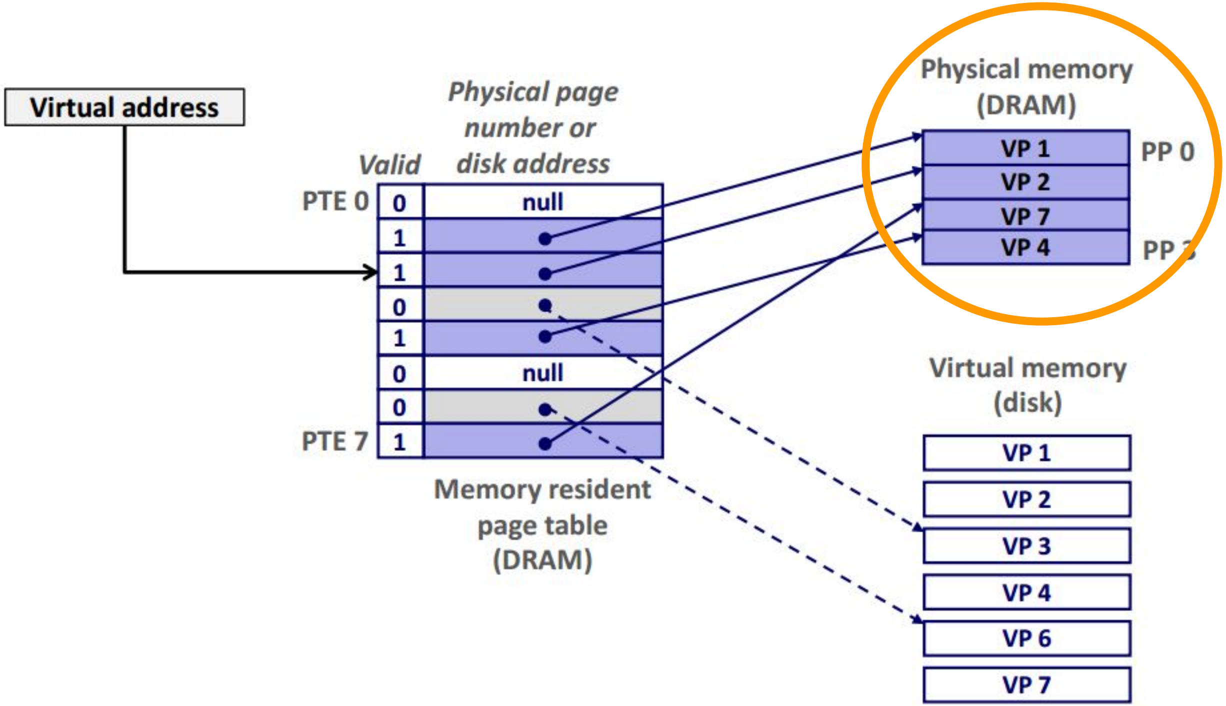
- We divide memory up into pages
  - (Typically 4096 bytes for 1 page)
- A page table then stores the mappings from a virtual page to its physical page address



0 for null or invalid pages

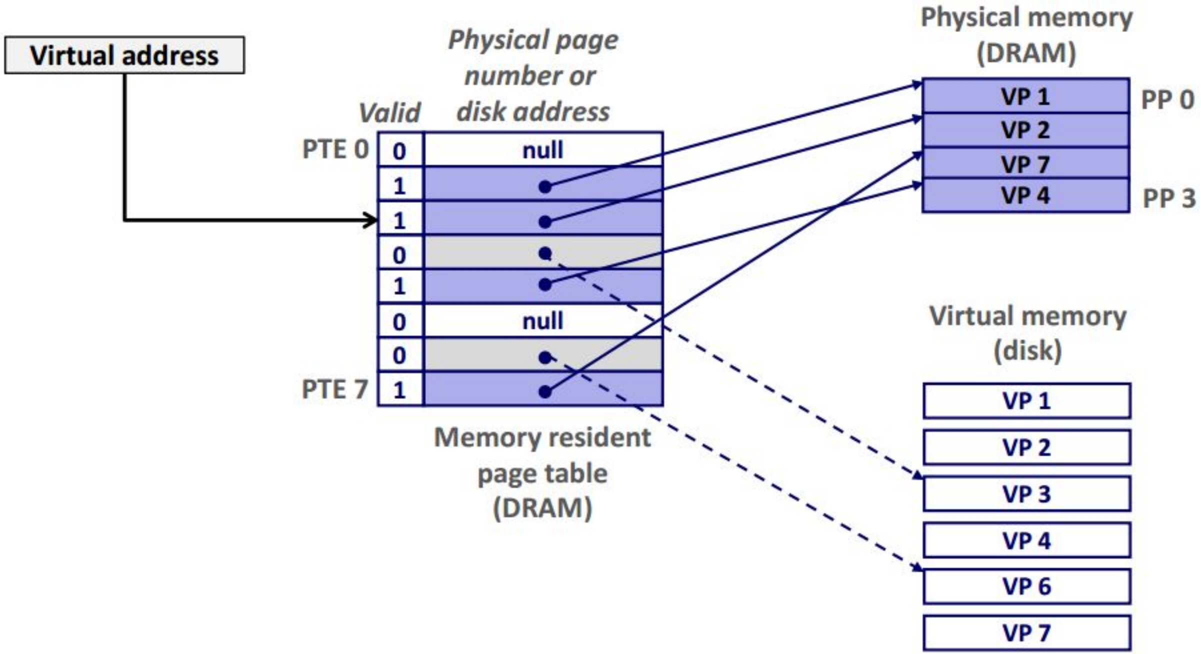
Page Hit

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



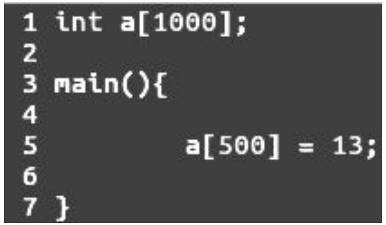
Page miss causes a Page Fault

- 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

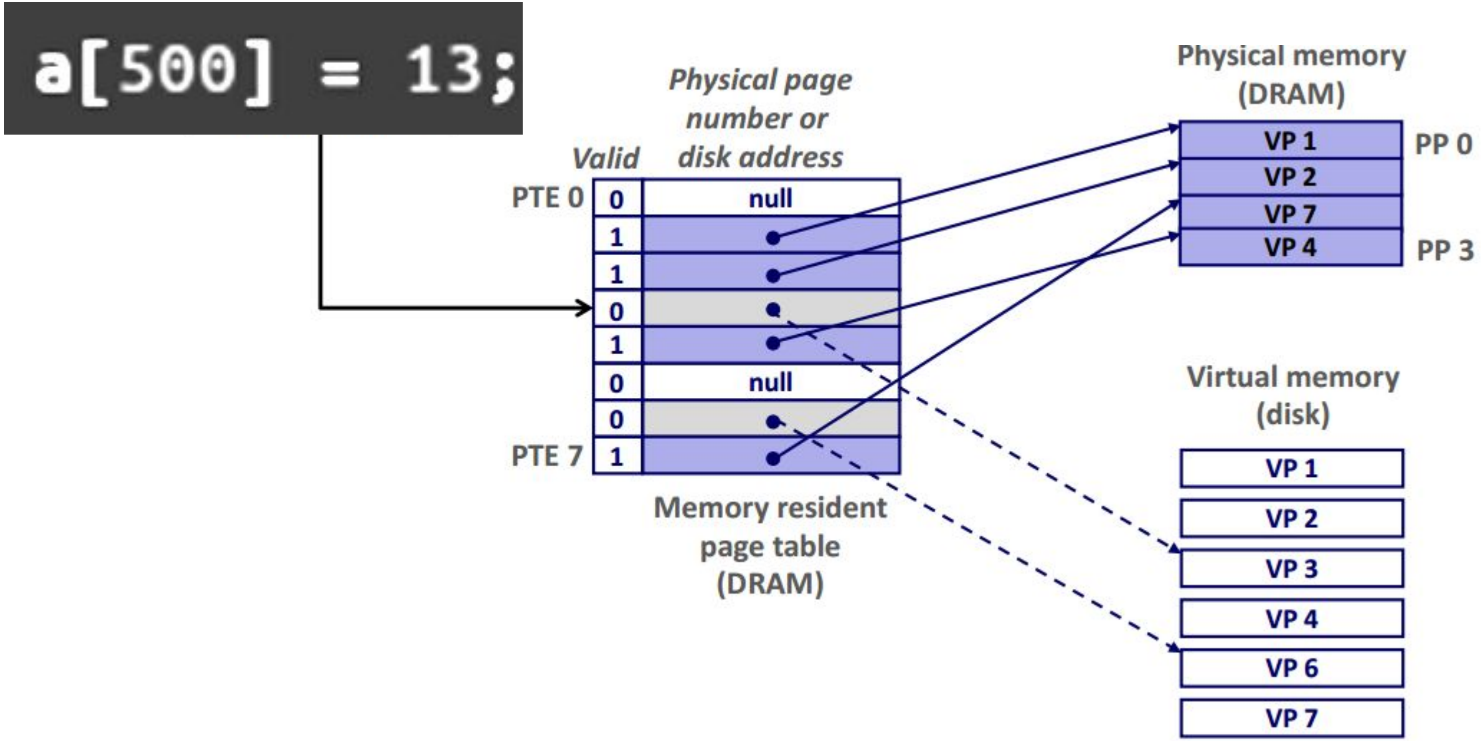
- User attempts to write to memory location



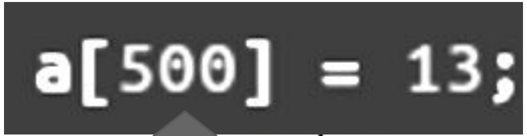
```
int a[1000];

main(){

        a[500] = 13;

}
```

- OS *may* (let’s assume it does) recognize this particular address is not valid.
    - Valid 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

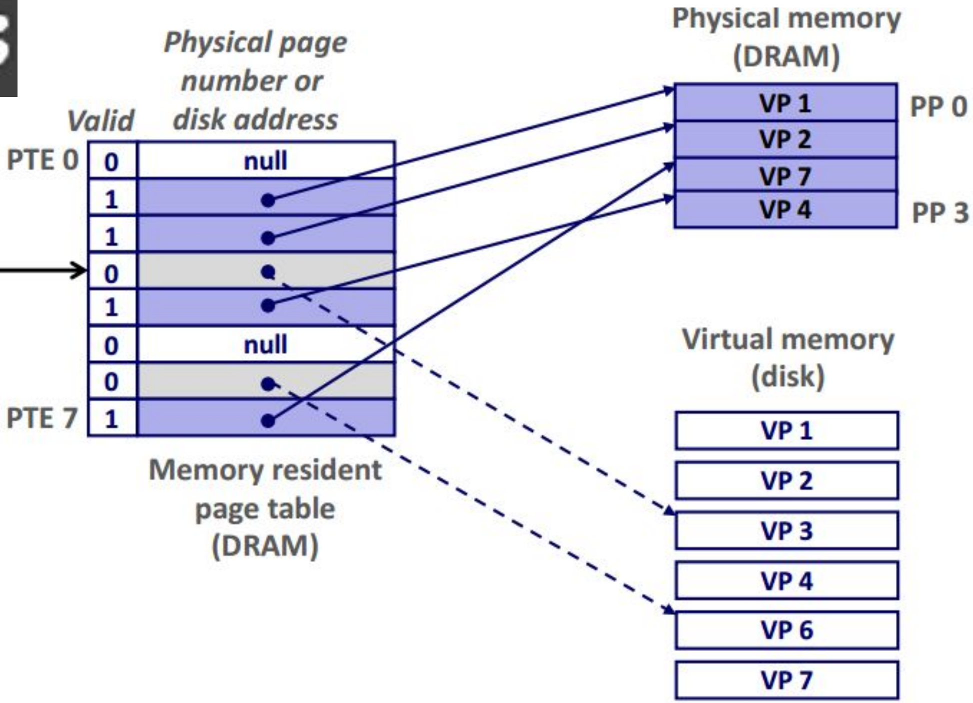


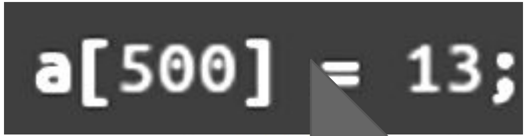
```
a[500] = 13;
```



```
a[500] = 13;
```

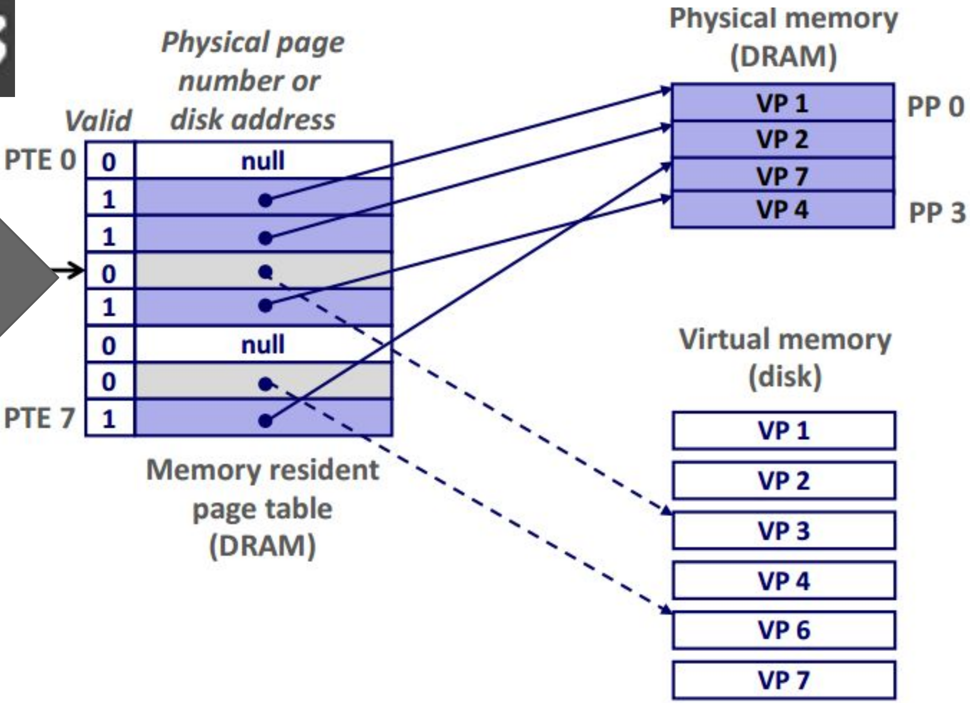
We try to access/write some data



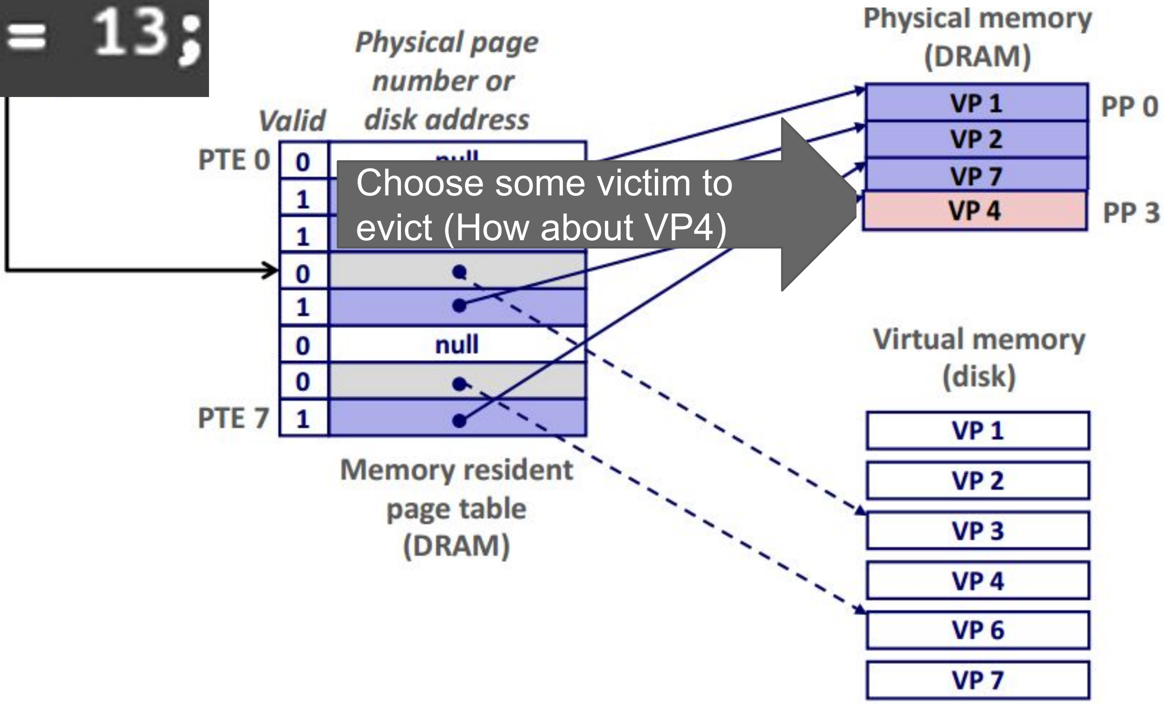


```
a[500] = 13;
```

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

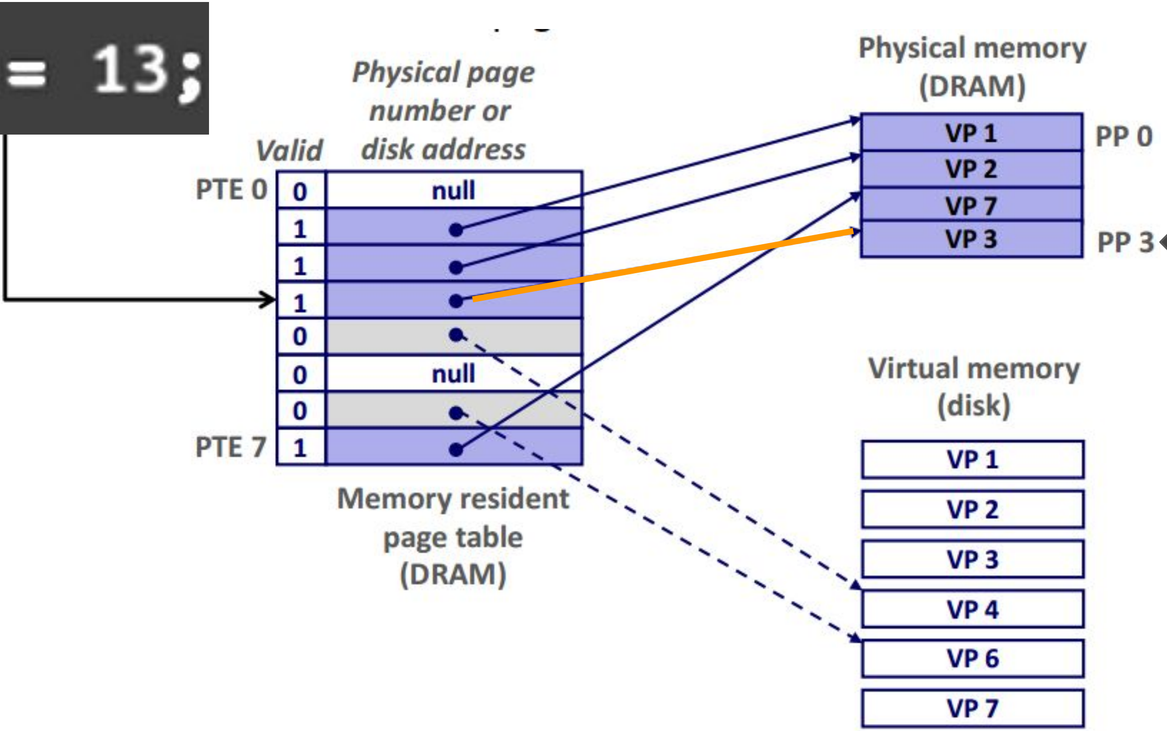


```
a[500] = 13;
```

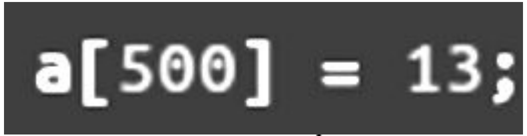


Choose some victim to evict (How about VP4)

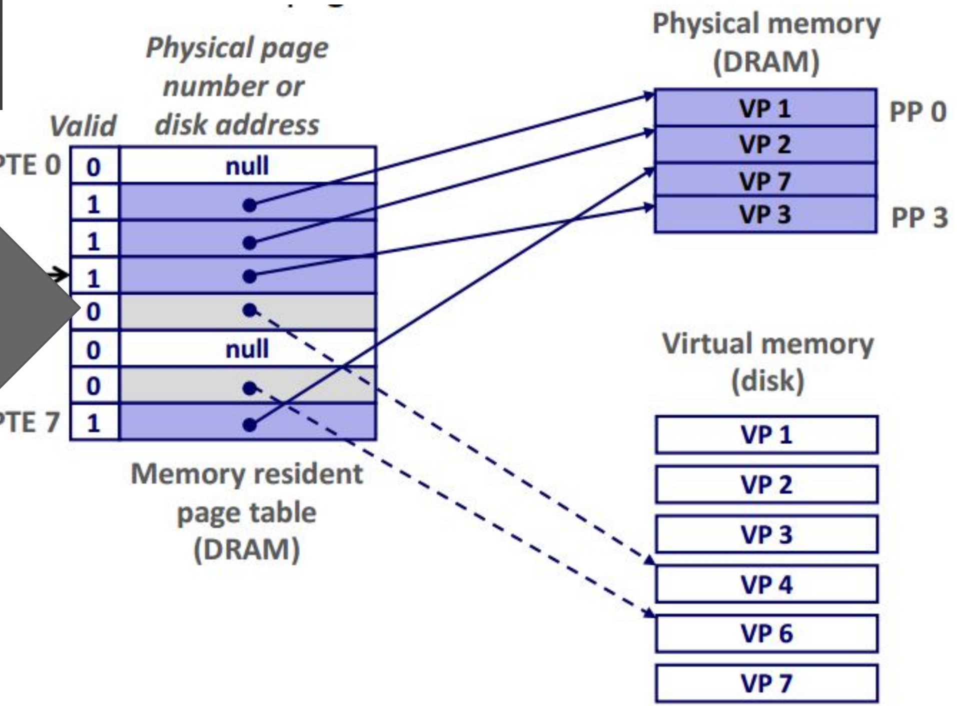
```
a[500] = 13;
```



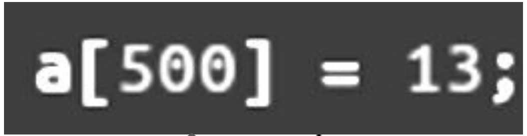
Update to VP3



```
a[500] = 13;
```

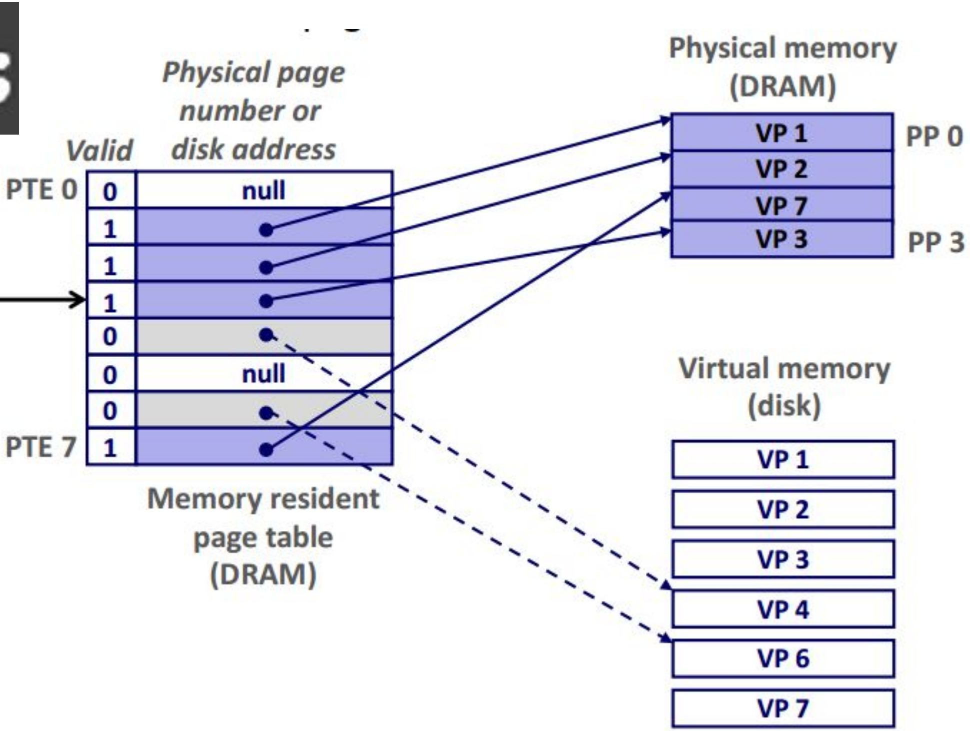


VP4 as a result is evicted



```
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?
    - (Can measure with “`perf record -e page-faults -ag`” if you are on Unix)
        - (Use ‘`perf list`’ to see more events you can record)
    - (Unfortunately our machines do not let us access the performance counters with record).
        - However, can still use
        - Run `` `perf stat ./myProgram` ``
            - Observe the different counts of the page-faults and context-switches shown!

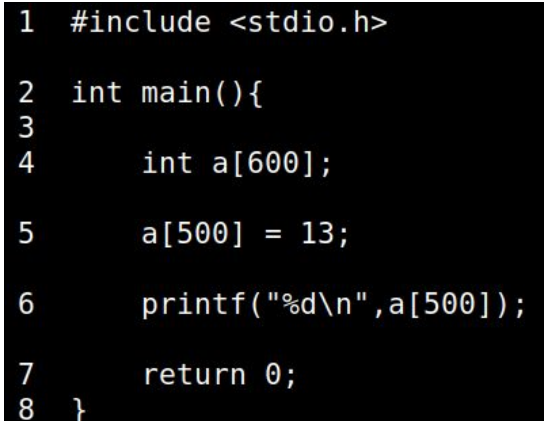
perf example (try at home)

shutterstock.com · 196836959

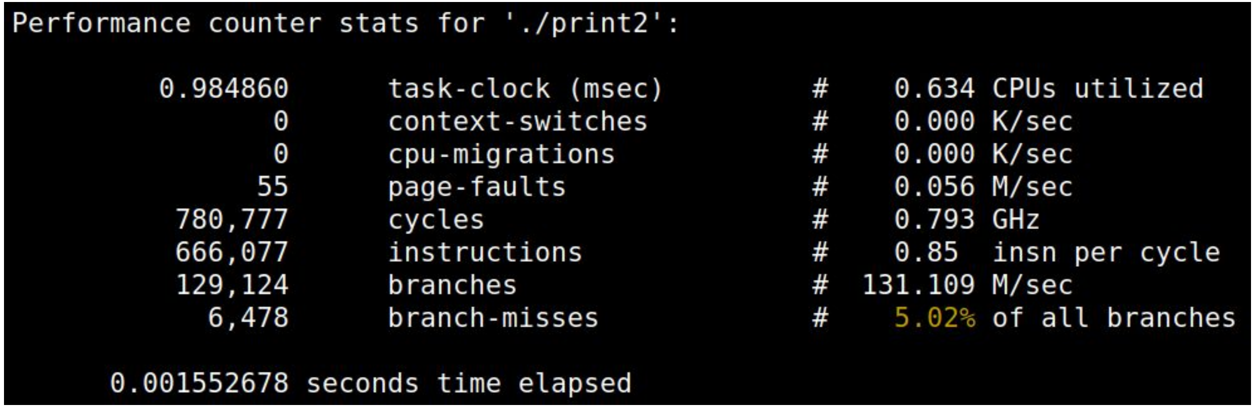
NOTE: Do not run this example on Khoury machines--do it on your VM!

You cannot run ‘**sudo**’ commands on systems. Is everyone paying attention? :)

- `sudo apt install` **`linux-tools-common linux-tools-generic linux-tools-n.n.n-nn-generic`**
- `sudo perf stat ./print2`



```c
#include <stdio.h>

int main(){

    int a[600];

    a[500] = 13;

    printf("%d\n",a[500]);

    return 0;
}
```



```
Performance counter stats for './print2':

        0.984860      task-clock (msec)         #    0.634 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
              55      page-faults               #    0.056 M/sec
         780,777      cycles                    #    0.793 GHz
         666,077      instructions              #    0.85  insn per cycle
         129,124      branches                  #  131.109 M/sec
           6,478      branch-misses             #    5.02% of all branches

     0.001552678 seconds time elapsed
```

Answer and New Question

- When your program executes, do you get a lot of page faults?
    - (Can measure with “`perf record -e page-faults -ag`” if you are on Unix)
    - Typically yes!
        - But this is okay in a sense that 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?

Answer and New Question

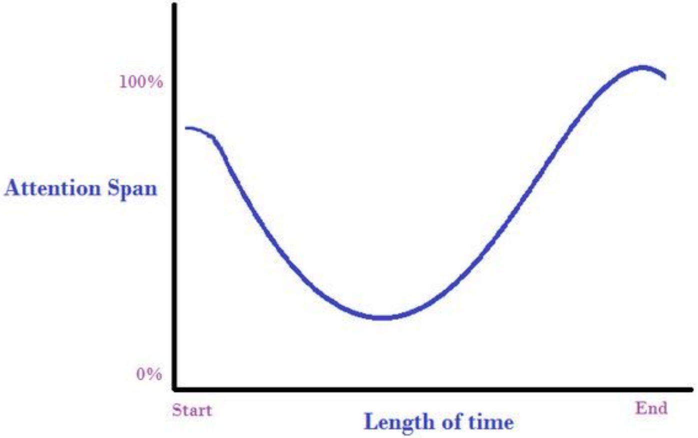
- When your program executes, do you get a lot of page faults?
    - (Can measure with “`perf record -e page-faults -ag`” if you are on Unix)
    - Typically yes!
        - But this is okay in a sense that a lot of the nitty gritty is handled for us.
        - Generally we do not try to predict the access 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

Short 5 minute break

- 1 hour 40 minutes is a long time.
- I will try to never lecture for more than half of that time without some sort of ‘break’ or transition to an in-class activity/lab.
- Use this time to stretch, check your phones, eat/drink something, etc.

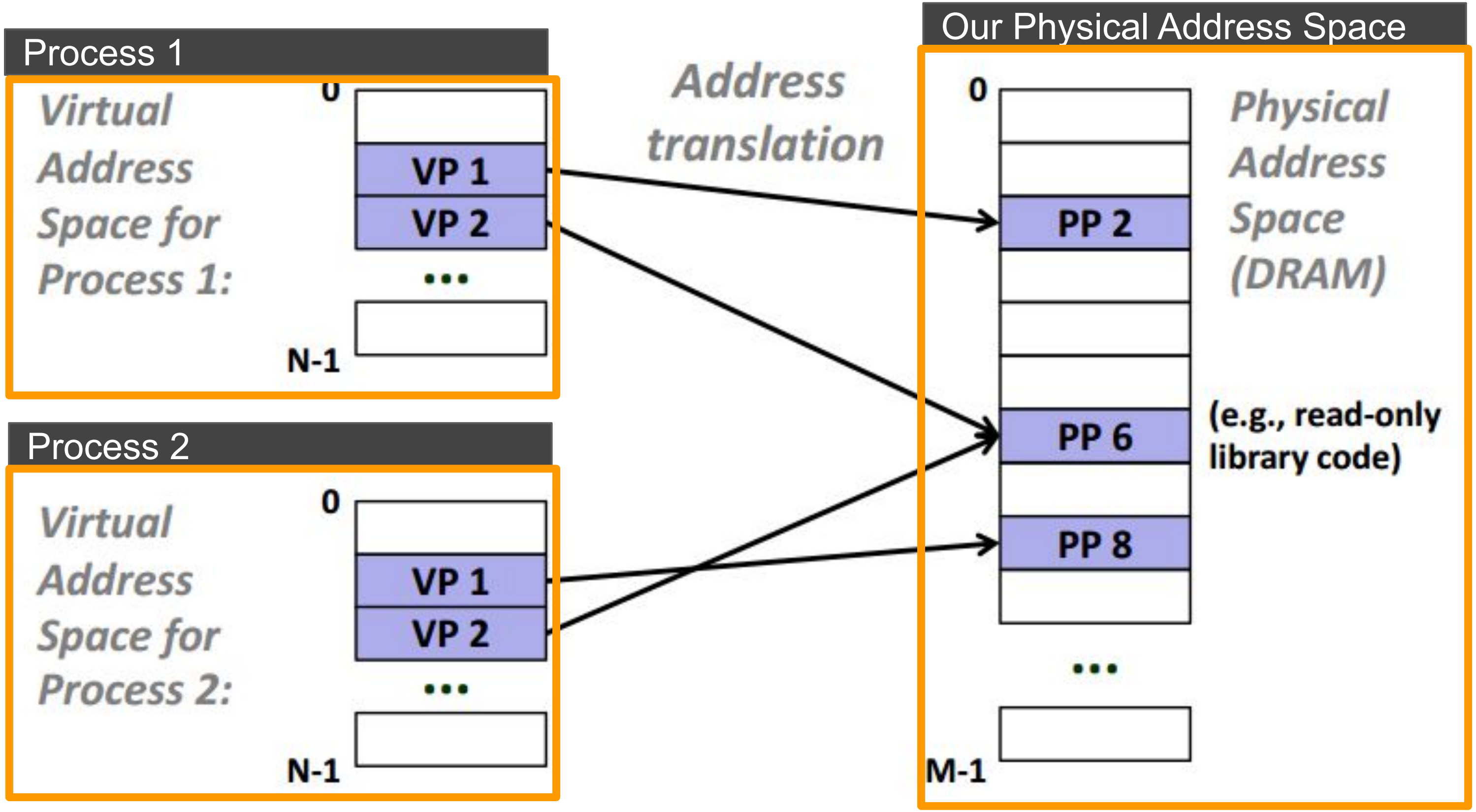


#2 Simplifies memory management (for application developers)

Virtual Memory for Memory Management

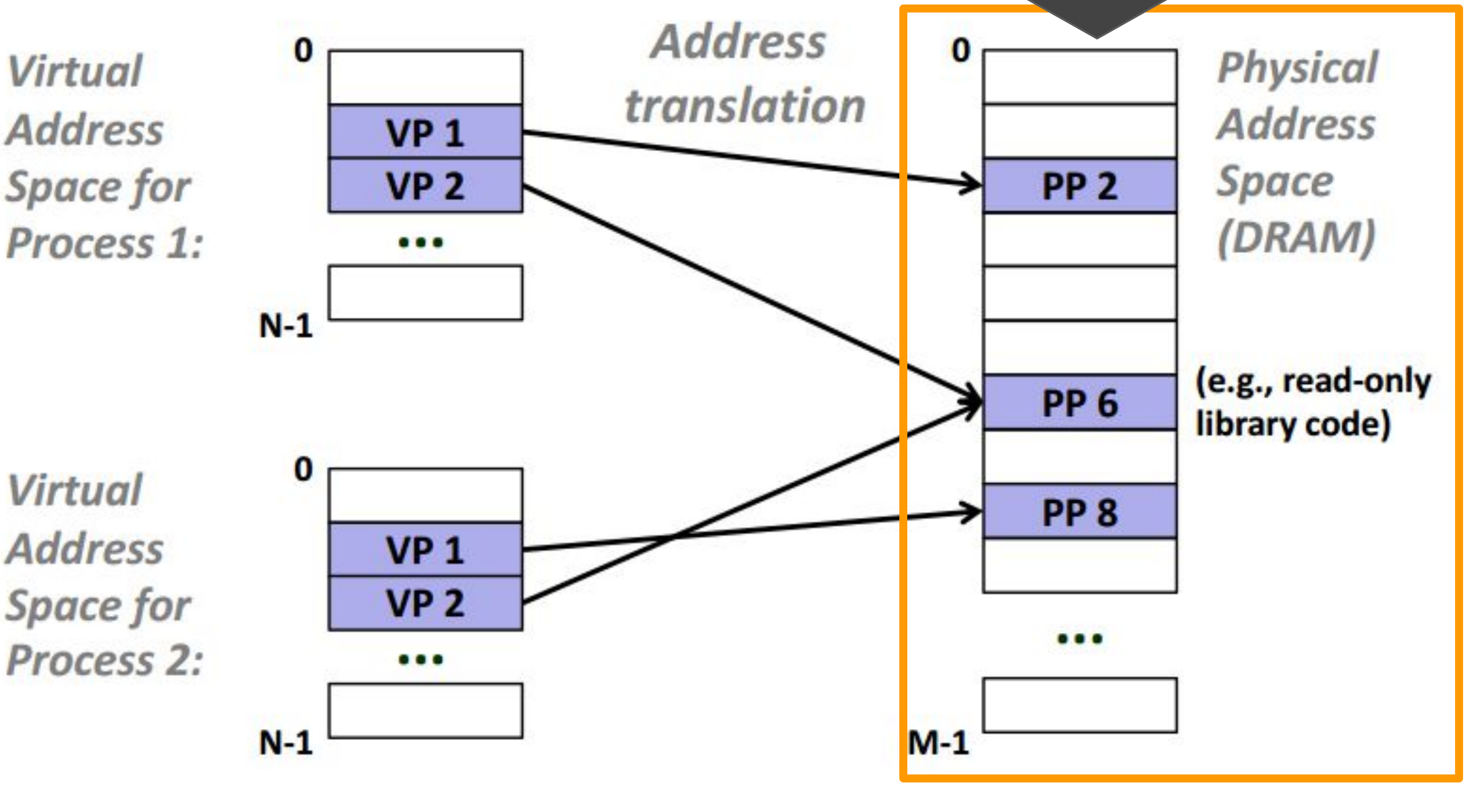
- Each process has its own virtual address space
  - This means we can view (within a process), memory as a linear array.
  - In reality, we known we have many pages scattered around.
    - (This could cause locality issues...so the OS needs to choose good mappings)

Example of page mappings


Process 1
Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Process 2
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1
Address translation
Our Physical Address Space
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)

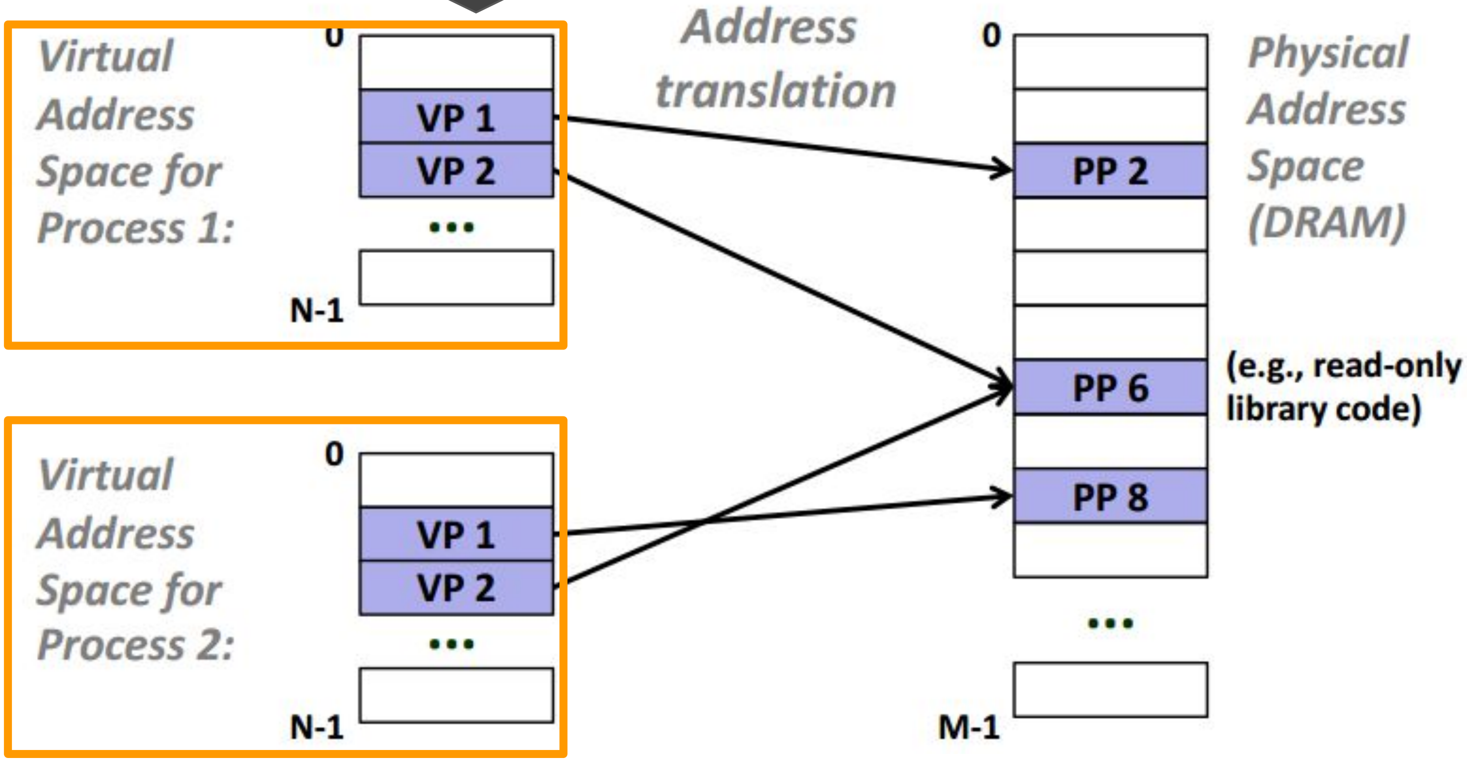
Example of page mappings

All of my physical memory is here

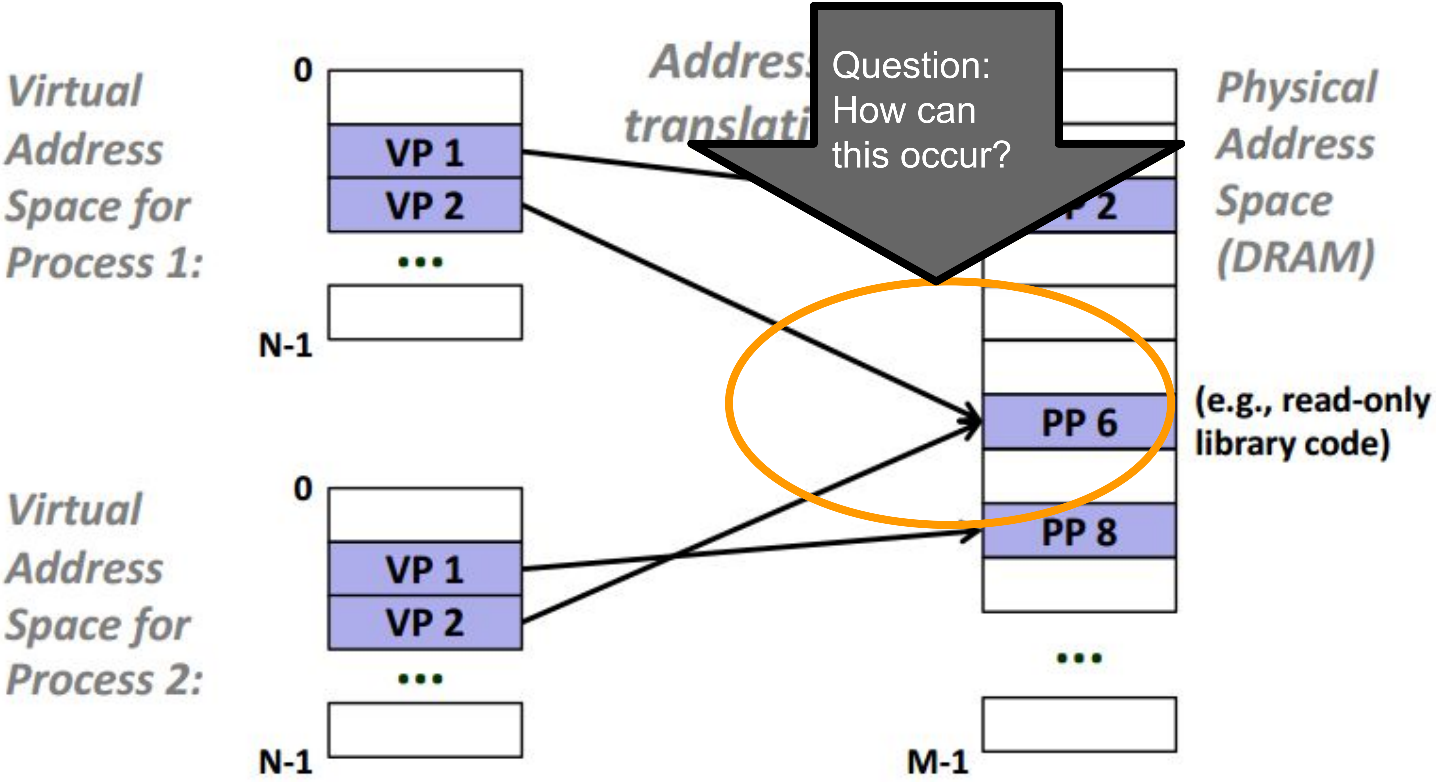

Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Example of gs

And our process sees its memory stored linearly here


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

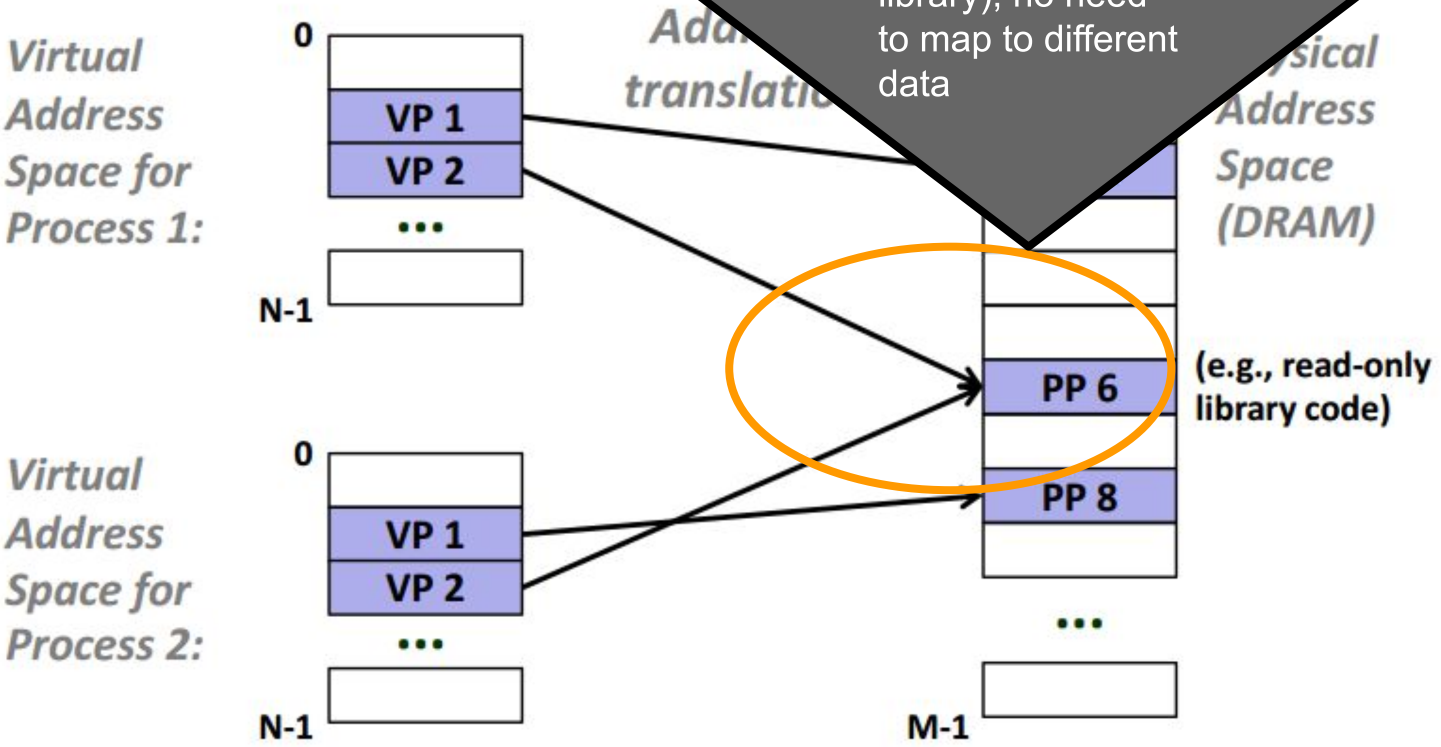
Example of page mappings



Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translati
Question: How can this occur?
P 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Example of page mappings

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

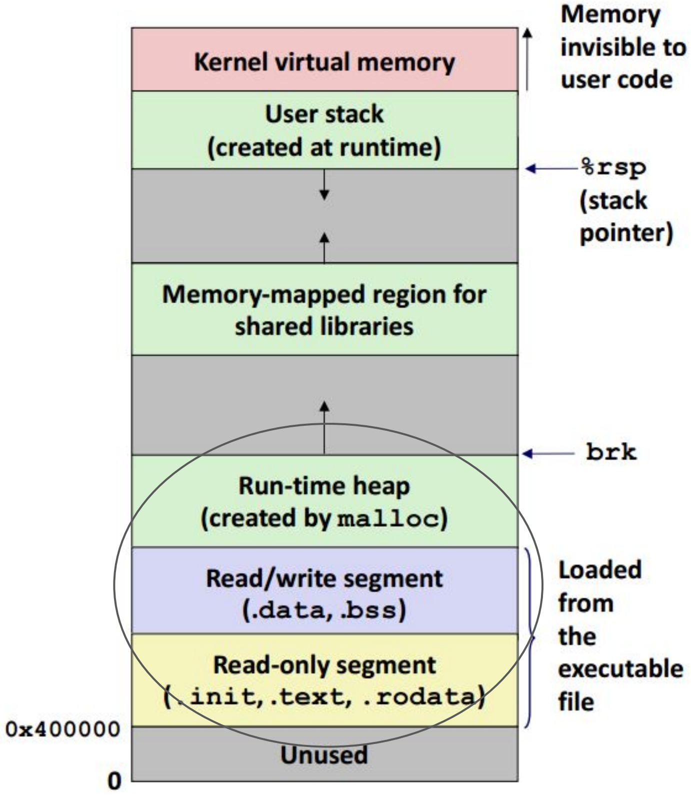


(e.g., read-only library code)

Virtual Memory supports Linking and Loading

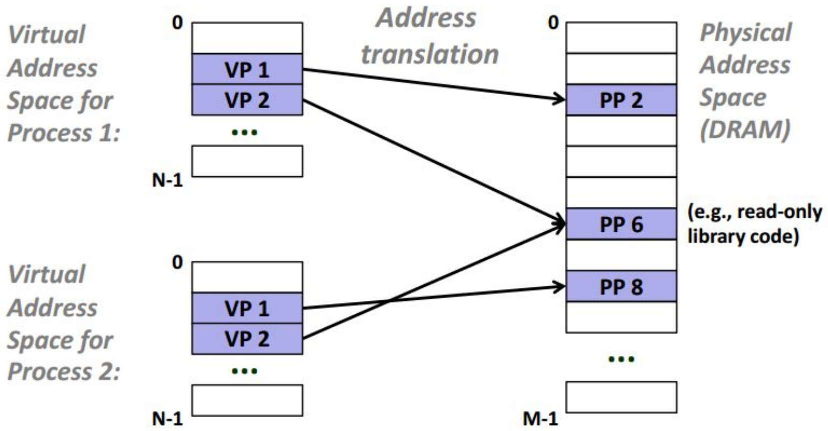
To our program, the virtual address space is roughly the same

- code, data, and heap sections start at same address


Kernel virtual memory
Memory invisible to user code
User stack (created at runtime)
%rsp (stack pointer)
Memory-mapped region for shared libraries
brk
Run-time heap (created by malloc)
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Loaded from the executable file
0x400000
Unused
0

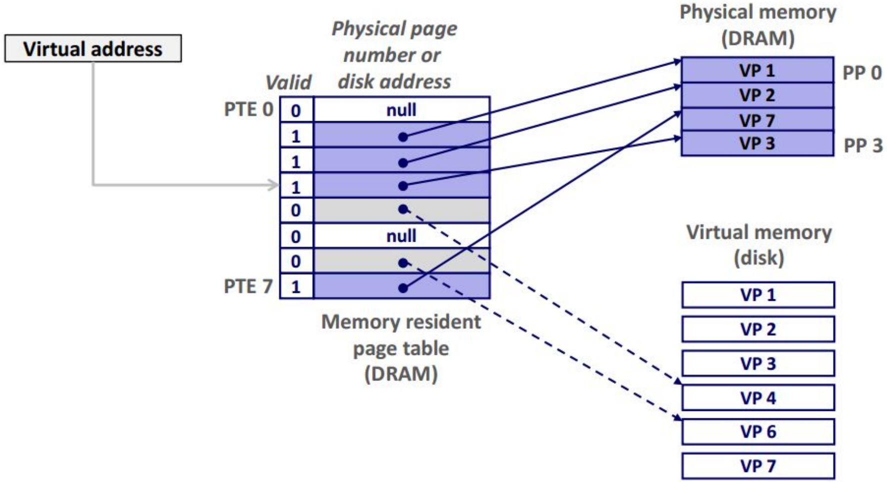
Virtual Memory as Memory Manager Summary

- 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!)


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
(e.g., read-only library code)
PP 8
...
M-1
Physical Address Space (DRAM)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

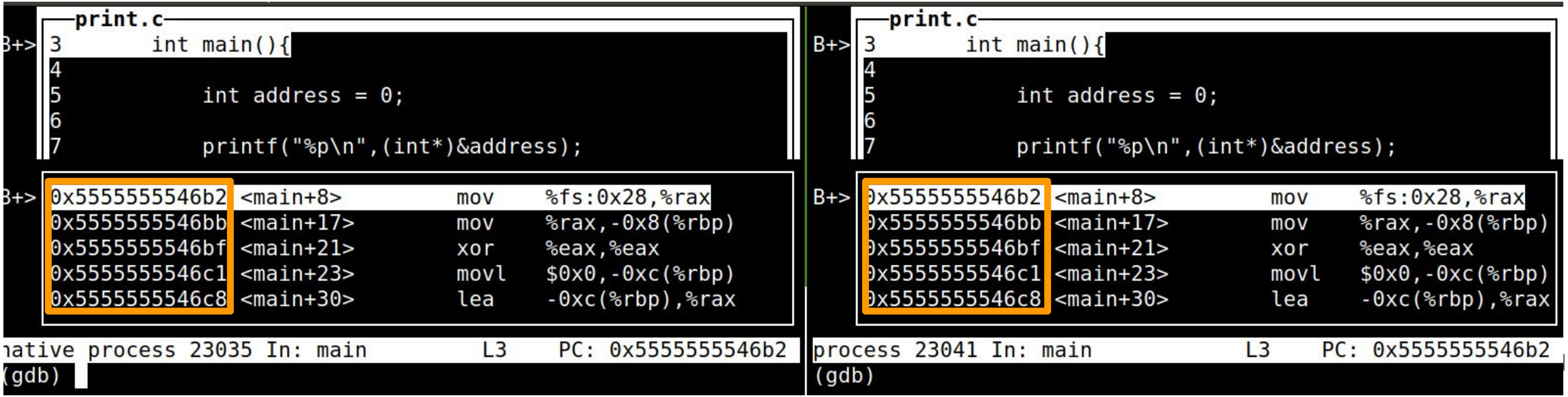
Two different tables

- So let’s keep straight that a page table is what maps physical addresses to virtual pages.
  - i.e “Where is this range of bytes stored”
    - “Oh, in page 1,5, etc.”
    - (Occasionally you will see PTE in literature which means page table entry)
- Then there is a separate mapping of pages for our programs
  - (i.e. a process keeps track of its pages based on how many it needs)



Little real world experiment (1/2)

- Here’s me running two instances of the same program
- From gdb’s perspective, it’s the same ‘address’ range for all of the .text section of our binary.
  - So perhaps we have mapped to the same page which seems efficient.



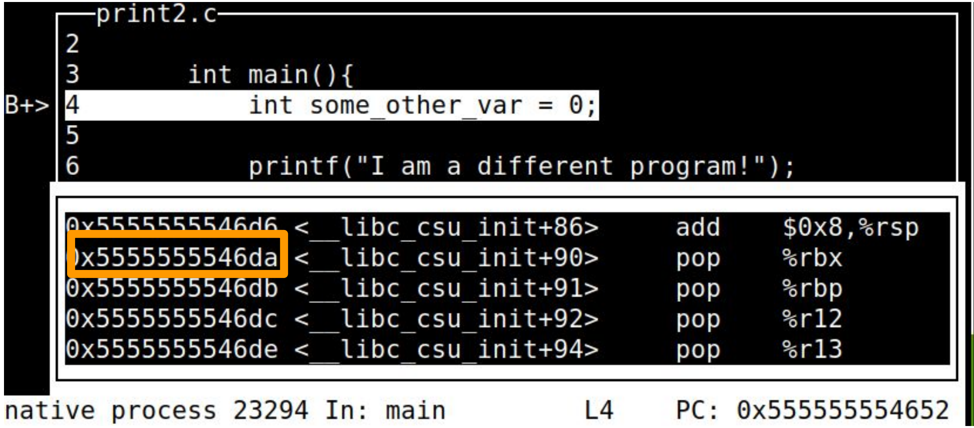
```
   ┌─print.c──────────────────────────────────────
B+>│3       int main(){
   │4
   │5           int address = 0;
   │6
   │7           printf("%p\n",(int*)&address);

B+>│0x5555555546b2 <main+8>      mov    %fs:0x28,%rax
   │0x5555555546bb <main+17>     mov    %rax,-0x8(%rbp)
   │0x5555555546bf <main+21>     xor    %eax,%eax
   │0x5555555546c1 <main+23>     movl   $0x0,-0xc(%rbp)
   │0x5555555546c8 <main+30>     lea    -0xc(%rbp),%rax

native process 23035 In: main          L3    PC: 0x5555555546b2
(gdb) 
```

```
   ┌─print.c──────────────────────────────────────
B+>│3       int main(){
   │4
   │5           int address = 0;
   │6
   │7           printf("%p\n",(int*)&address);

B+>│0x5555555546b2 <main+8>      mov    %fs:0x28,%rax
   │0x5555555546bb <main+17>     mov    %rax,-0x8(%rbp)
   │0x5555555546bf <main+21>     xor    %eax,%eax
   │0x5555555546c1 <main+23>     movl   $0x0,-0xc(%rbp)
   │0x5555555546c8 <main+30>     lea    -0xc(%rbp),%rax

process 23041 In: main          L3    PC: 0x5555555546b2
(gdb)
```

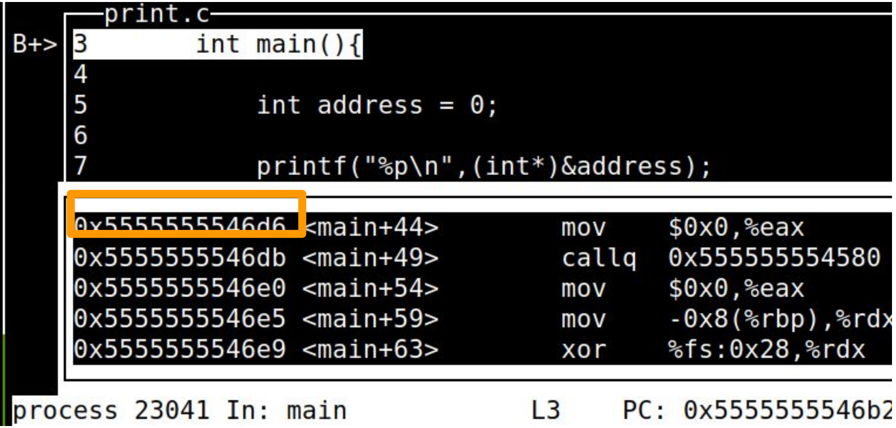
Little real world experiment (2/2)

On a different experiment, running two different programs--the addresses are slightly unaligned

- But if I look closely, the ranges (0x55555555_ _ _ _) do overlap and even repeat in some places!
    - This shows off the a linear range of addresses virtual memory provides--so our programs have the illusion of all starting from 0



```
     print2.c
     2
     3          int main(){
B+>  4                  int some_other_var = 0;
     5
     6                  printf("I am a different program!");

     0x5555555546d6 <__libc_csu_init+86>     add    $0x8,%rsp
     0x5555555546da <__libc_csu_init+90>     pop    %rbx
     0x5555555546db <__libc_csu_init+91>     pop    %rbp
     0x5555555546dc <__libc_csu_init+92>     pop    %r12
     0x5555555546de <__libc_csu_init+94>     pop    %r13

native process 23294 In: main          L4    PC: 0x555555554652
```

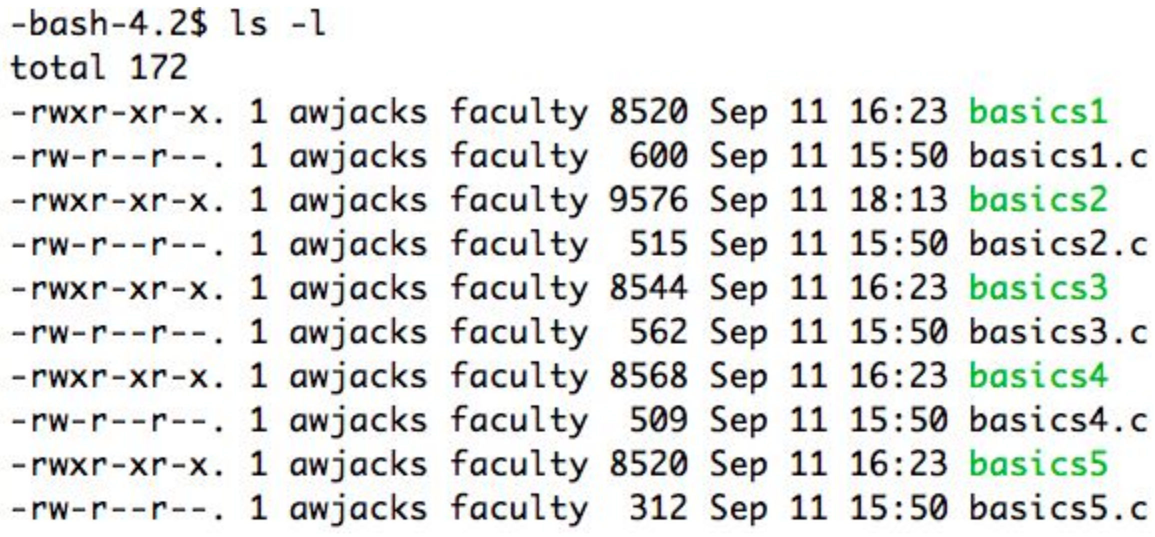


```
     print.c
B+>  3          int main(){
     4
     5                  int address = 0;
     6
     7                  printf("%p\n",(int*)&address);

     0x5555555546d6 <main+44>          mov    $0x0,%eax
     0x5555555546db <main+49>          callq  0x555555554580
     0x5555555546e0 <main+54>          mov    $0x0,%eax
     0x5555555546e5 <main+59>          mov    -0x8(%rbp),%rdx
     0x5555555546e9 <main+63>          xor    %fs:0x28,%rdx

process 23041 In: main              L3    PC: 0x5555555546b2
```

#3 Isolates Address Spaces

Virtual Memory protection

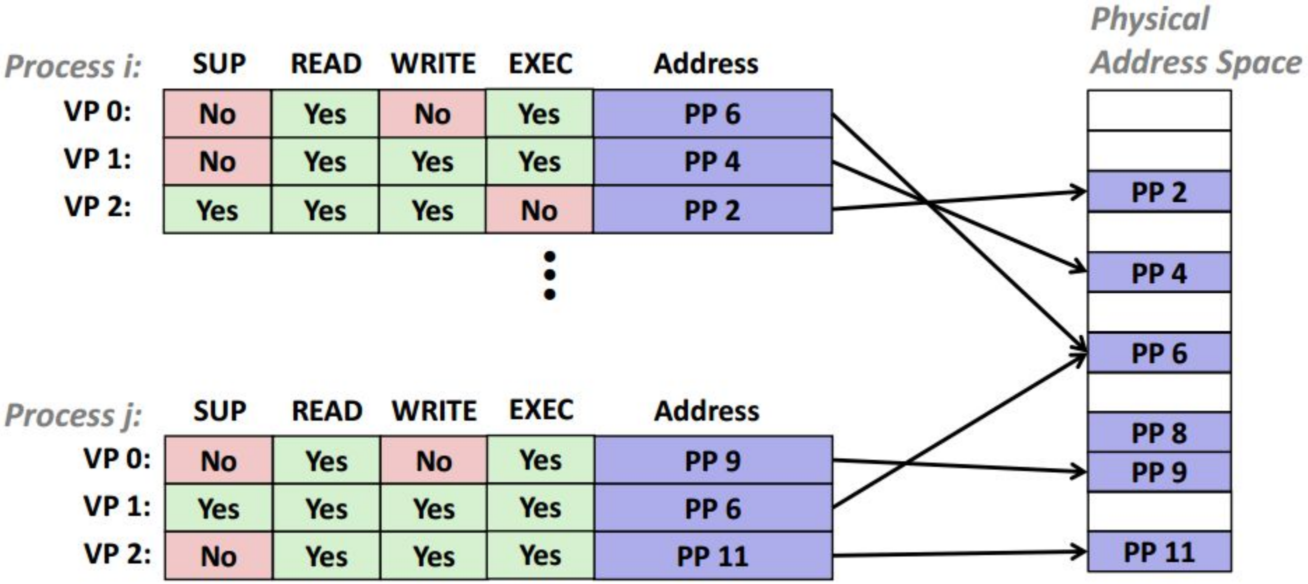
- Certain files have read/write/execute permissions set.
  - This ensures one process cannot just overwrite another, or access data it should not.
- You can view them as follows:



```
-bash-4.2$ ls -l
total 172
-rwxr-xr-x. 1 awjacks faculty 8520 Sep 11 16:23 basics1
-rw-r--r--. 1 awjacks faculty  600 Sep 11 15:50 basics1.c
-rwxr-xr-x. 1 awjacks faculty 9576 Sep 11 18:13 basics2
-rw-r--r--. 1 awjacks faculty  515 Sep 11 15:50 basics2.c
-rwxr-xr-x. 1 awjacks faculty 8544 Sep 11 16:23 basics3
-rw-r--r--. 1 awjacks faculty  562 Sep 11 15:50 basics3.c
-rwxr-xr-x. 1 awjacks faculty 8568 Sep 11 16:23 basics4
-rw-r--r--. 1 awjacks faculty  509 Sep 11 15:50 basics4.c
-rwxr-xr-x. 1 awjacks faculty 8520 Sep 11 16:23 basics5
-rw-r--r--. 1 awjacks faculty  312 Sep 11 15:50 basics5.c
```

Virtual Memory protection

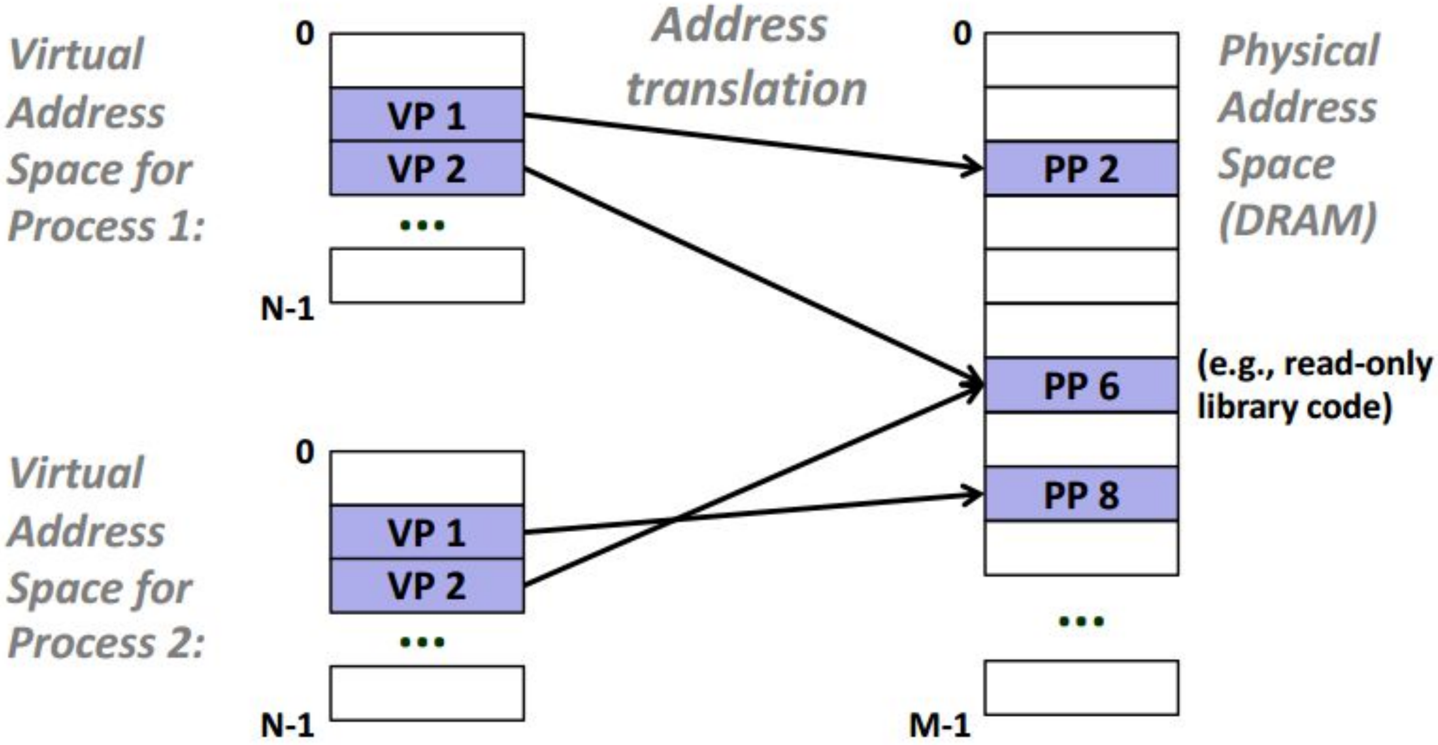
- Depending on the access, the MMU (Memory Management Unit) determines which pages can be executed.



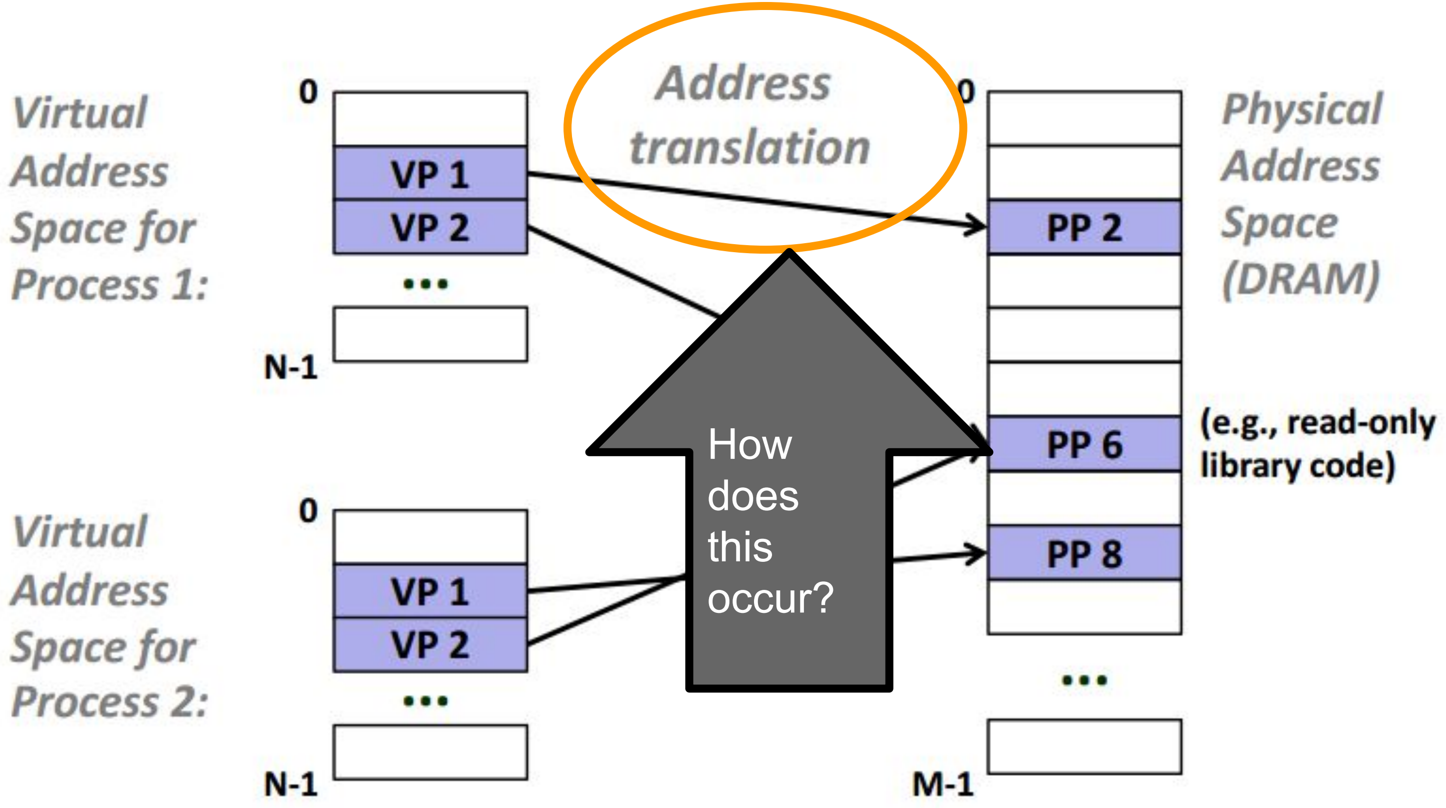
Process i:	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 6
VP 1:	No	Yes	Yes	Yes	PP 4
VP 2:	Yes	Yes	Yes	No	PP 2

Process j:	SUP	READ	WRITE	EXEC	Address
VP 0:	No	Yes	No	Yes	PP 9
VP 1:	Yes	Yes	Yes	Yes	PP 6
VP 2:	No	Yes	Yes	Yes	PP 11

Revisiting our picture - One missing component


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
Physical Address Space (DRAM)
PP 2
PP 6
(e.g., read-only library code)
PP 8
...
M-1
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Revisiting our picture - One missing component


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address translation
0
PP 2
PP 6
PP 8
...
M-1
Physical Address Space (DRAM)
(e.g., read-only library code)
How does this occur?
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

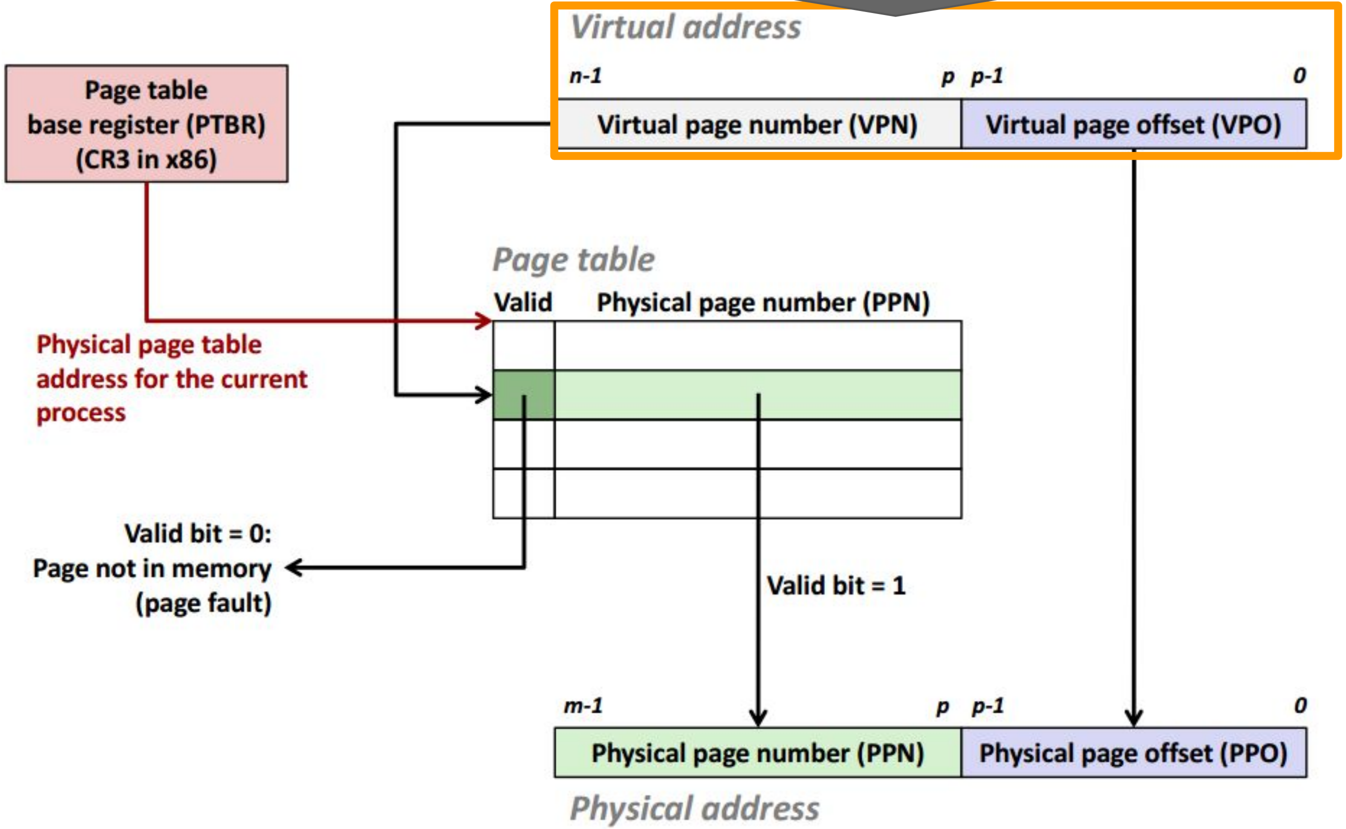
Address Translation Example

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

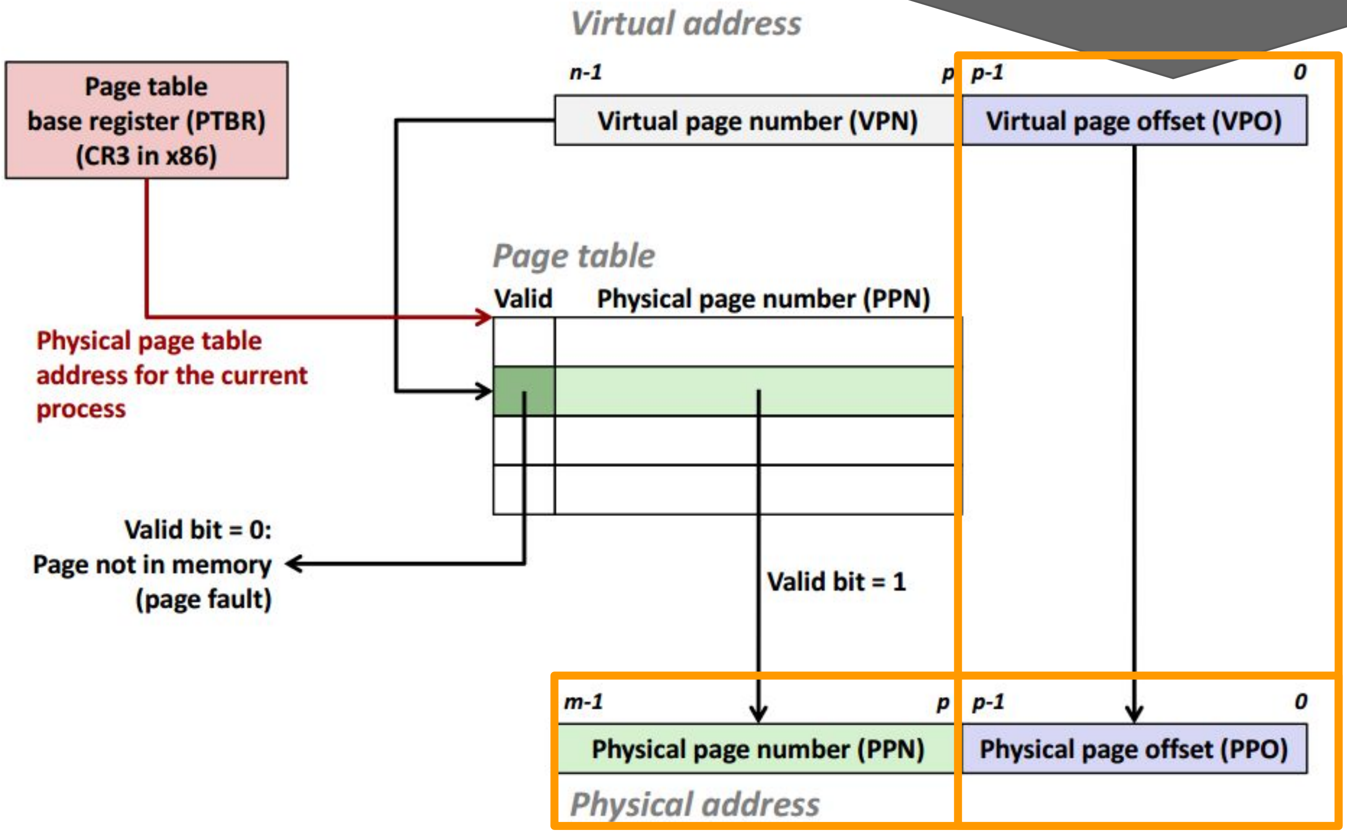
Address Translation with

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



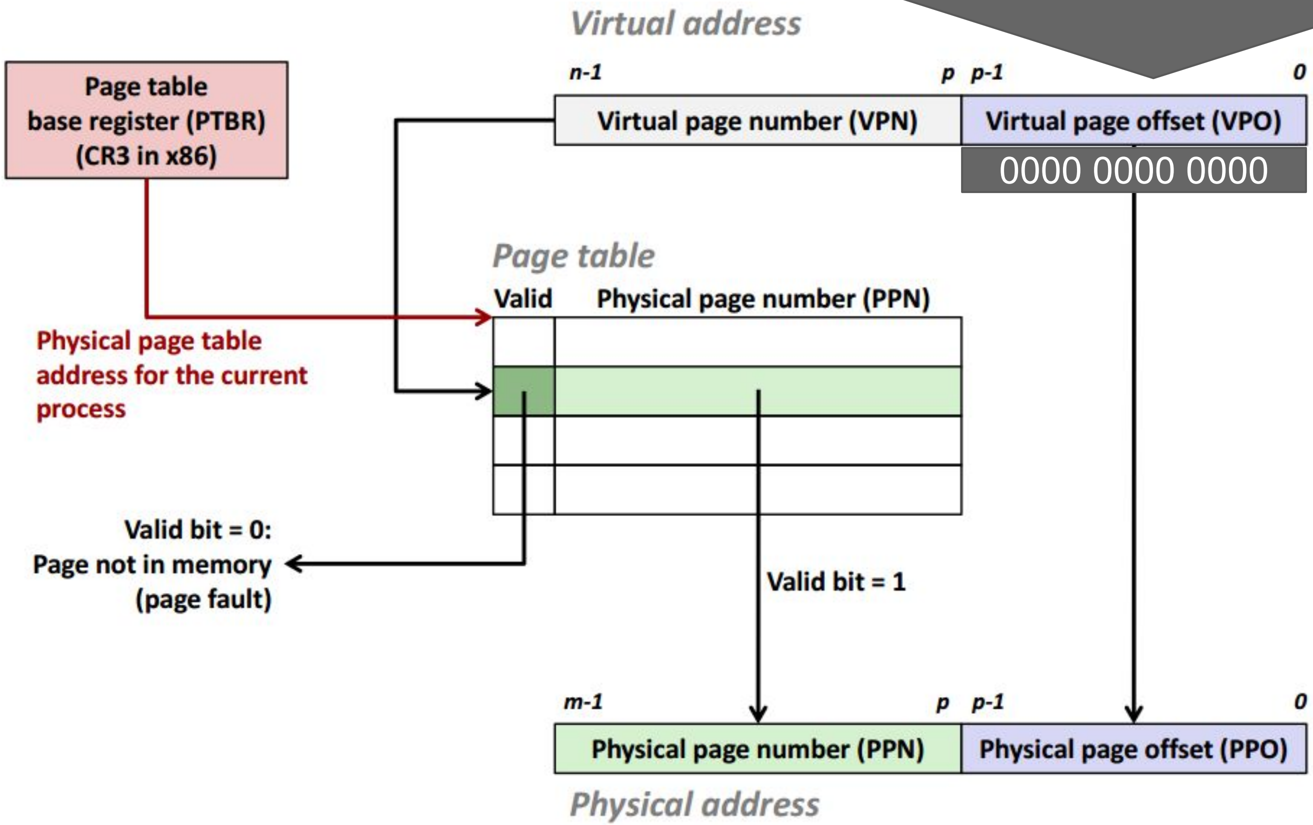
Address Translation with Page Tab

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


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR) (CR3 in x86)
Page table
Valid
Physical page number (PPN)
Physical page table address for the current process
Valid bit = 0: Page not in memory (page fault)
Valid bit = 1
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

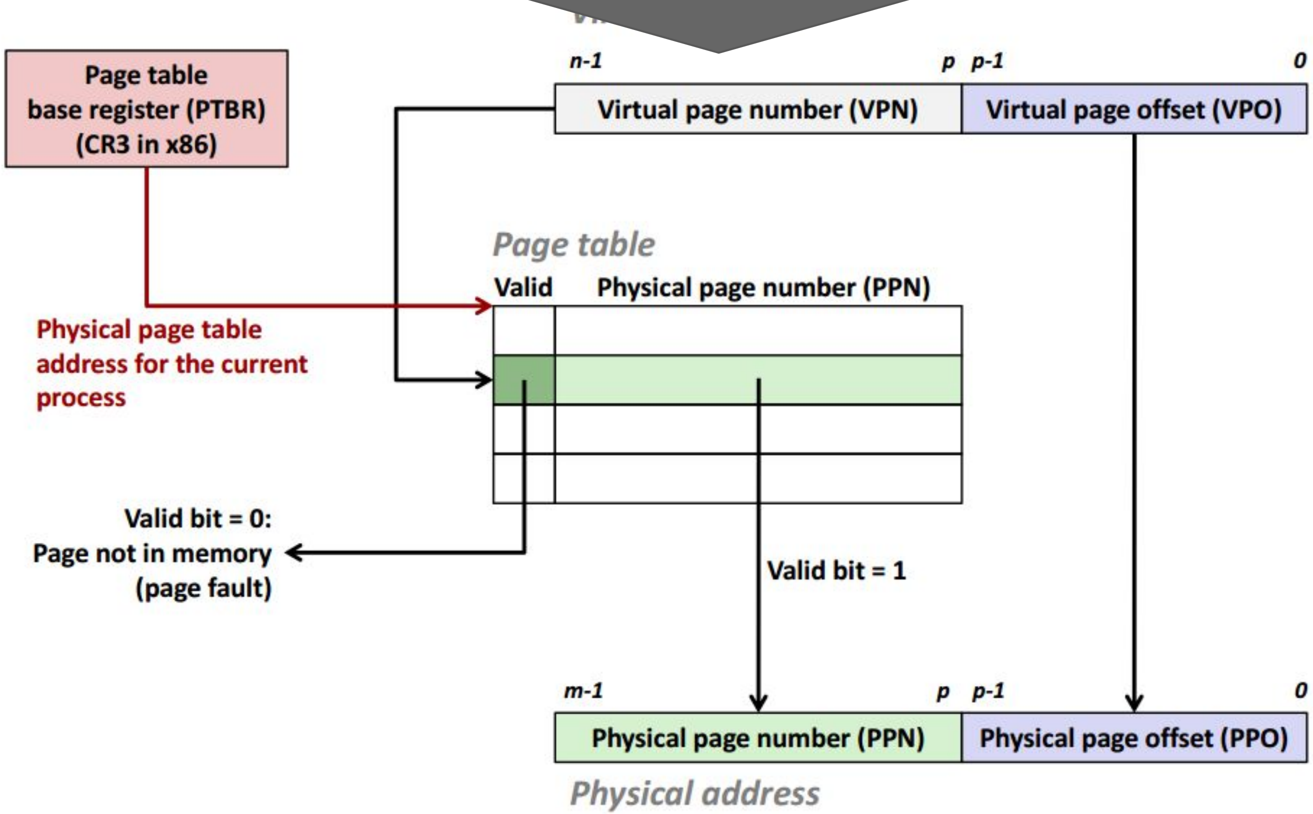
Address Translation with Page Ta

4096 byte page, means 12 bits are used (to tell us where in the page we are)


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
0000 0000 0000
Page table base register (PTBR) (CR3 in x86)
Physical page table address for the current process
Page table
Valid
Physical page number (PPN)
Valid bit = 0: Page not in memory (page fault)
Valid bit = 1
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

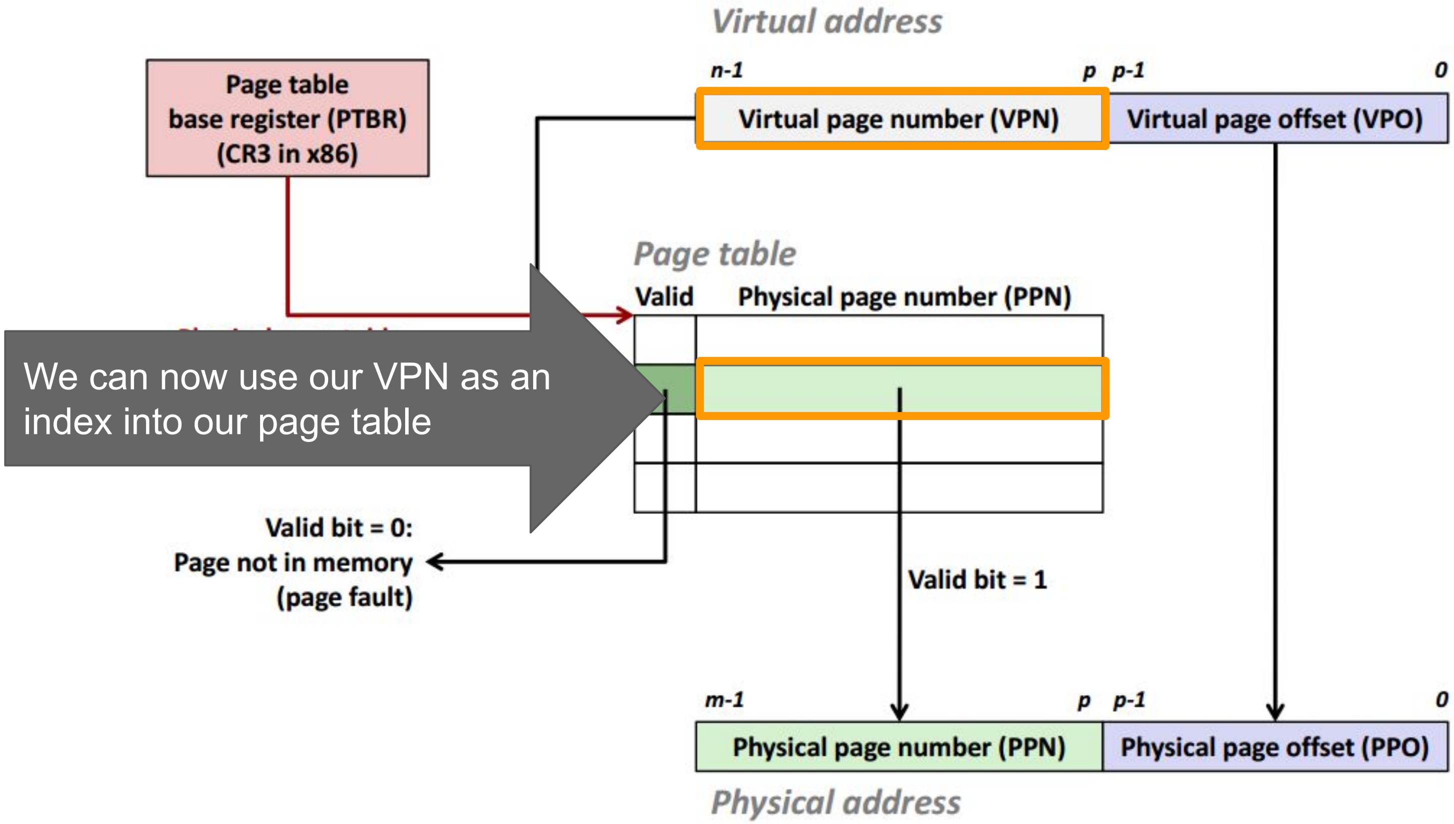
Address Translation

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



n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR) (CR3 in x86)
Physical page table address for the current process
Page table
Valid
Physical page number (PPN)
Valid bit = 0: Page not in memory (page fault)
Valid bit = 1
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

Address Translation with Page Table



Virtual address

n-1

p

p-1

0

Virtual page number (VPN)

Virtual page offset (VPO)

Page table base register (PTBR) (CR3 in x86)

Page table

Valid

Physical page number (PPN)

We can now use our VPN as an index into our page table

Valid bit = 0: Page not in memory (page fault)

Valid bit = 1

m-1

p

p-1

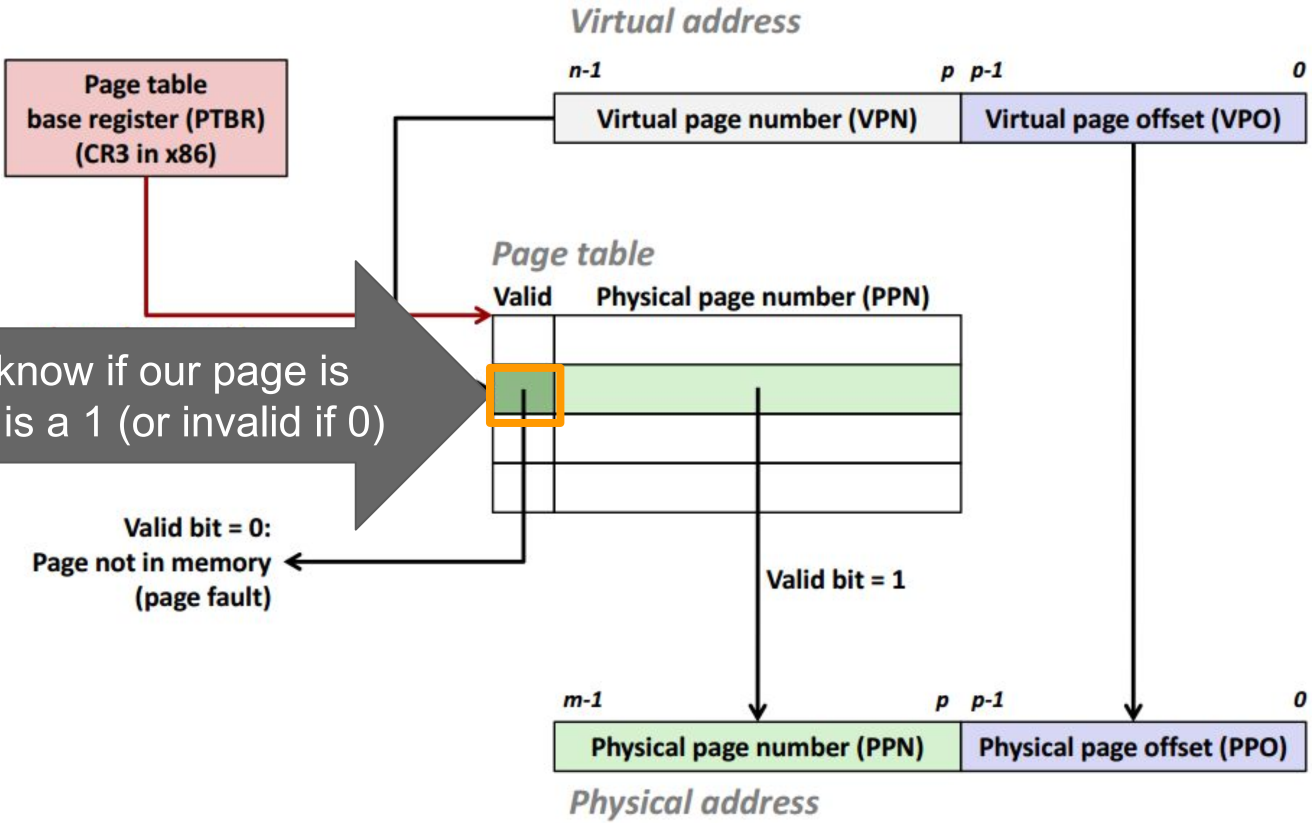
0

Physical page number (PPN)

Physical page offset (PPO)

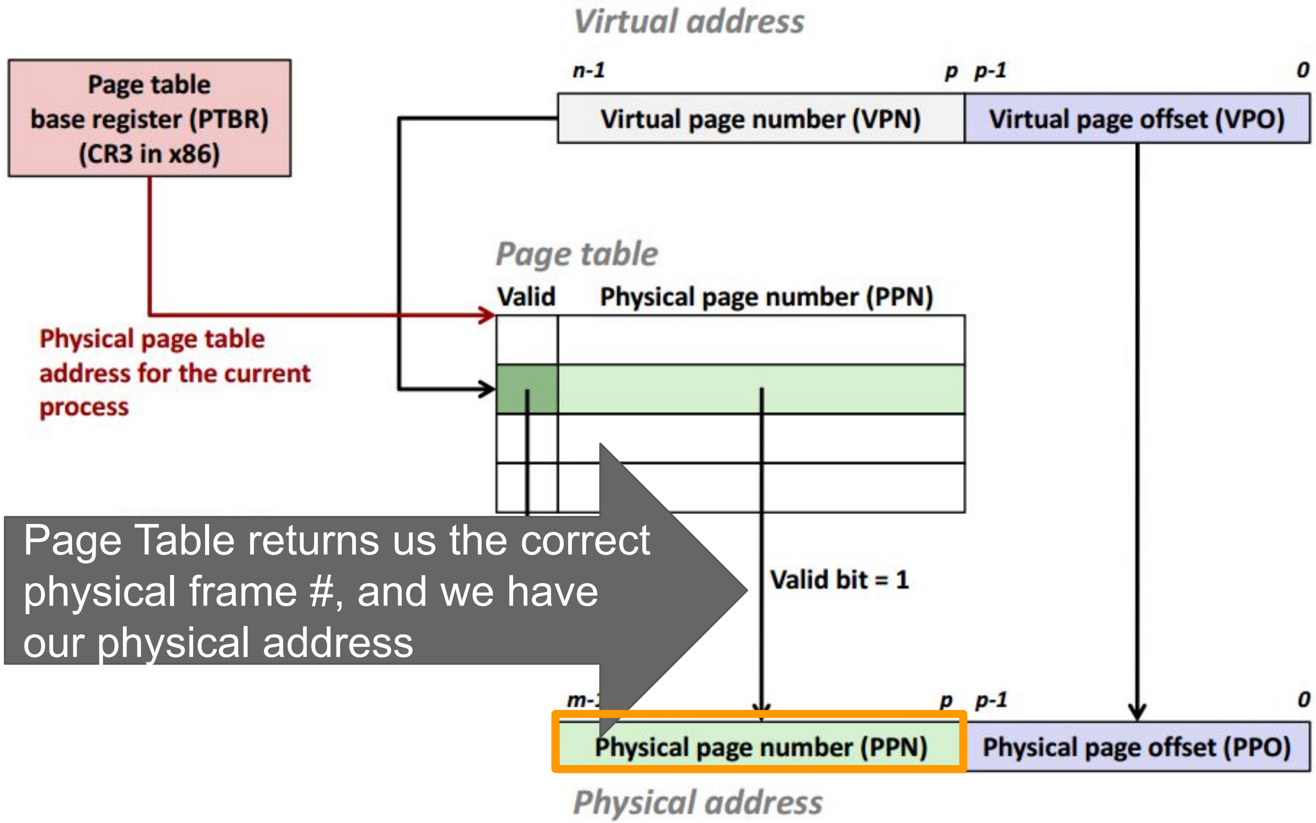
Physical address

Address Translation with Page Table


Virtual address
n-1
p
p-1
0
Virtual page number (VPN)
Virtual page offset (VPO)
Page table base register (PTBR) (CR3 in x86)
Page table
Valid
Physical page number (PPN)
Valid bit = 0: Page not in memory (page fault)
Valid bit = 1
m-1
p
p-1
0
Physical page number (PPN)
Physical page offset (PPO)
Physical address

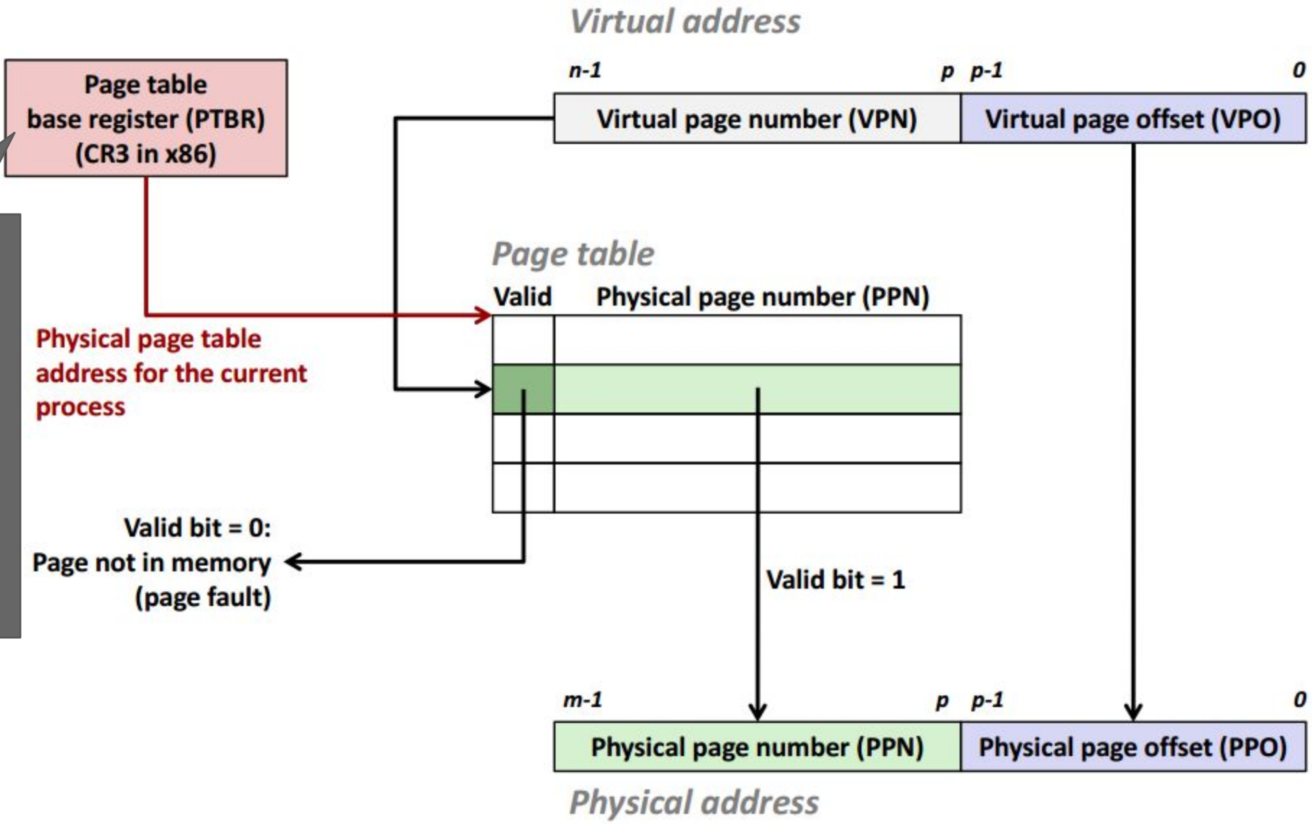
Finally we know if our page is valid if this is a 1 (or invalid if 0)

Address Translation with Page Table



Page Table returns us the correct physical frame #, and we have our physical address

Address Translation with Page Table

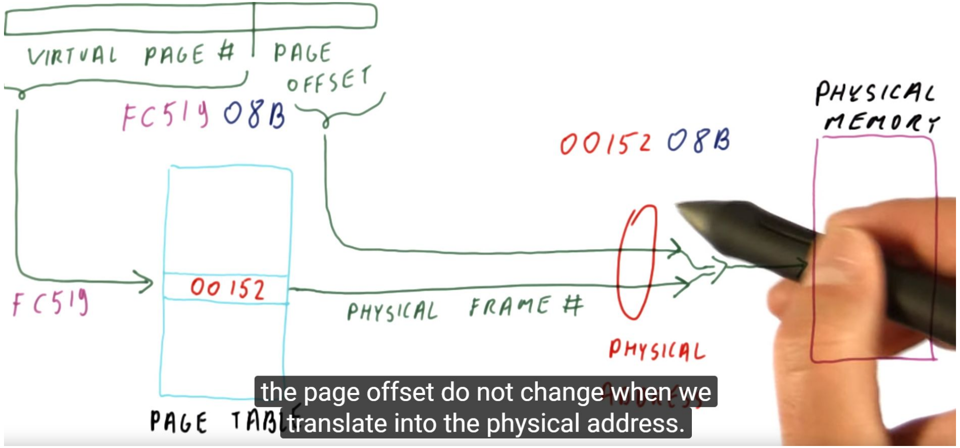


Note: A special register stores a pointer to the actual page table.

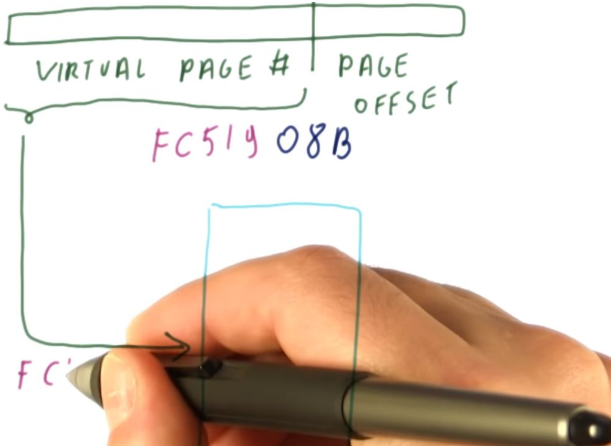
Watch at home...

Recap!

- https://www.youtube.com/watch?v=l7HoguhFVQ4



the page offset do not change when we translate into the physical address.



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

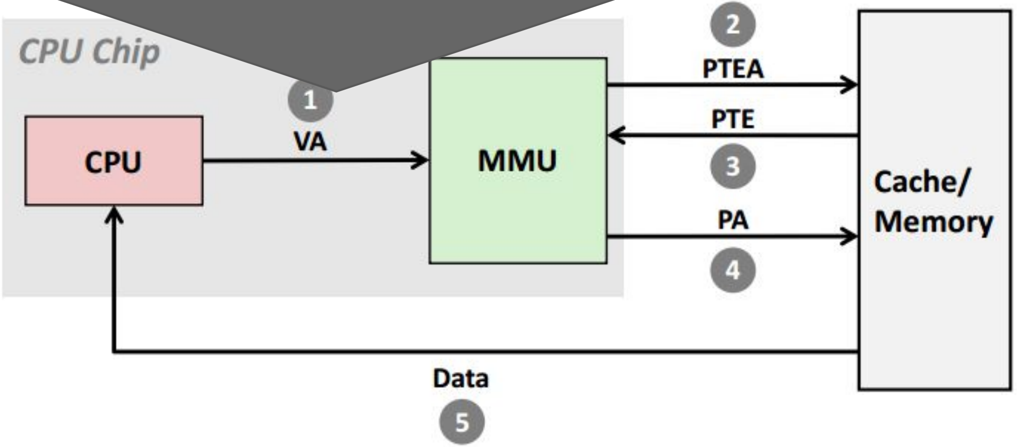
1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor

1. CPU attempts some MOV instr



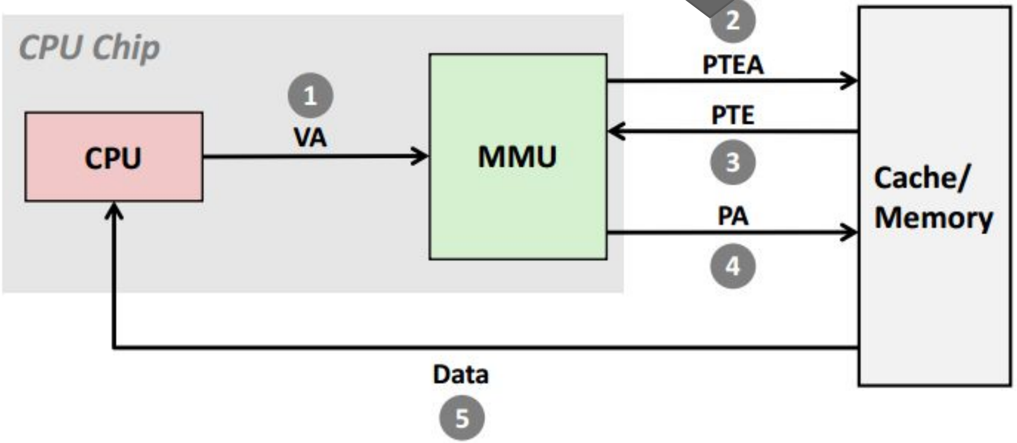
Address Translation: Page Hit

1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor



2.) Get page table entry address

Address Translation: Page Hit

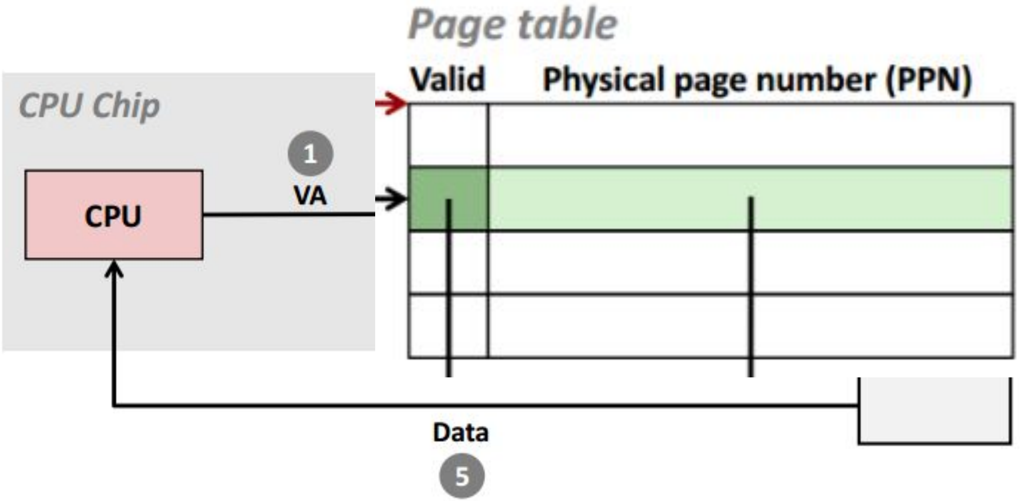
2.) Get page table entry address

1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor



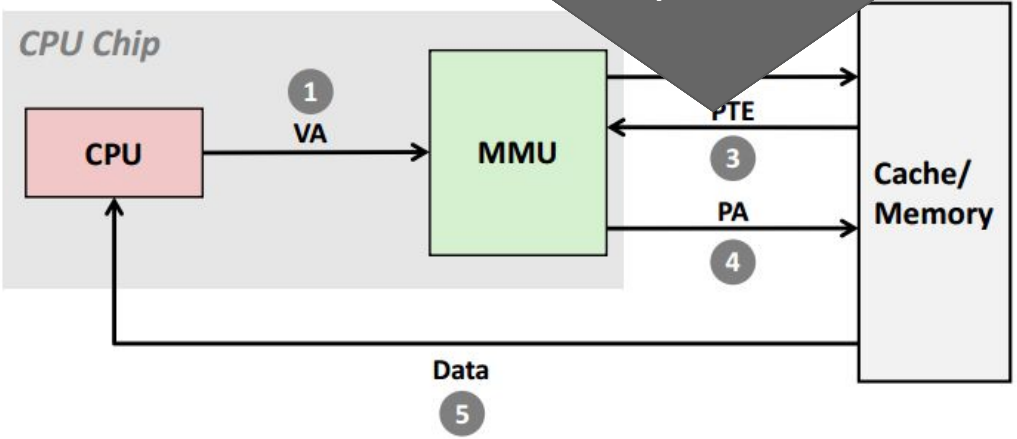
Address Translation: Page Hit

1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor



3.) Now read the memory from the Page table entry

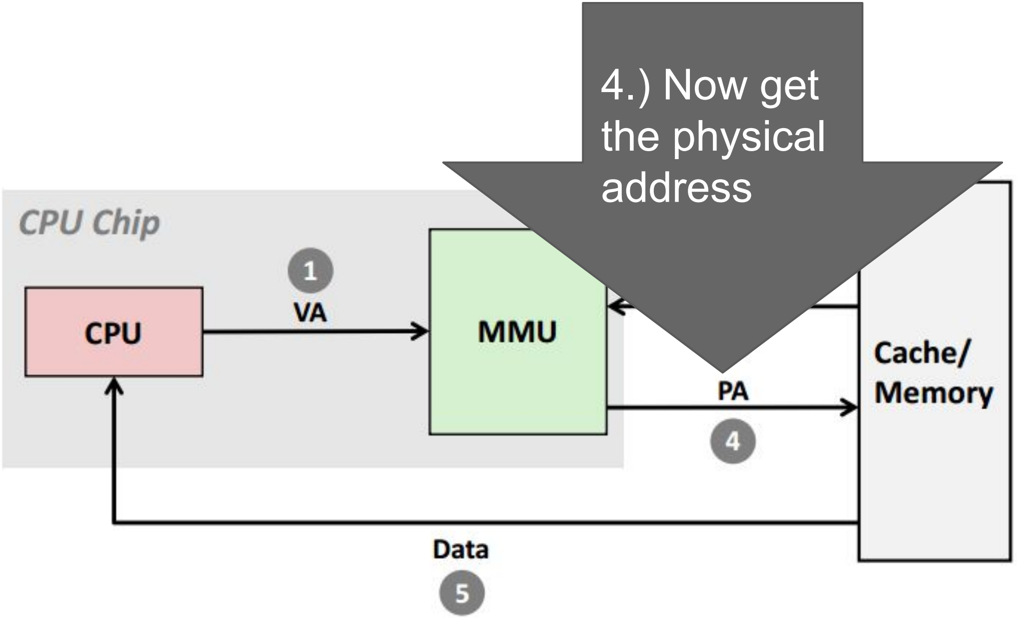
Address Translation: Page Hit

1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor



4.) Now get the physical address

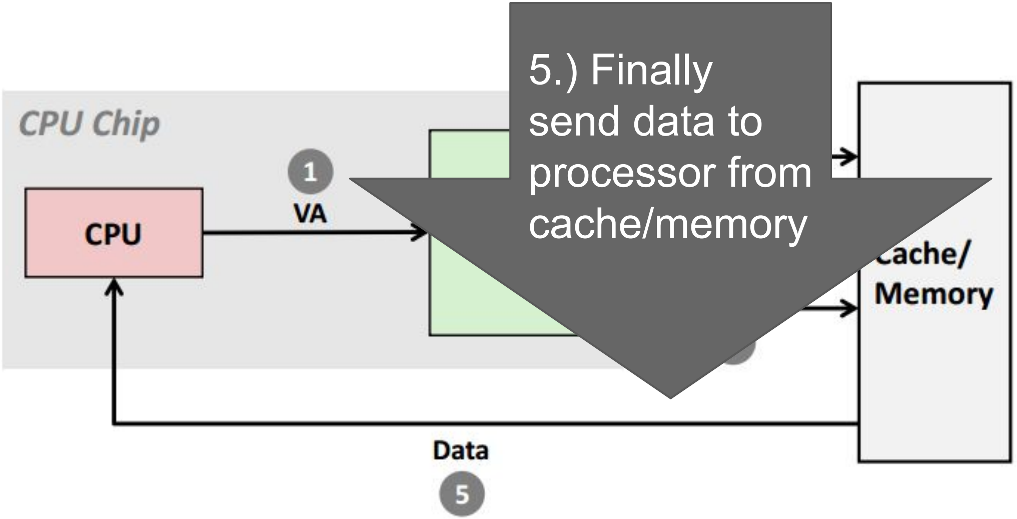
Address Translation: Page Hit

1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor



5.) Finally send data to processor from cache/memory

Address Translation: Page Hit

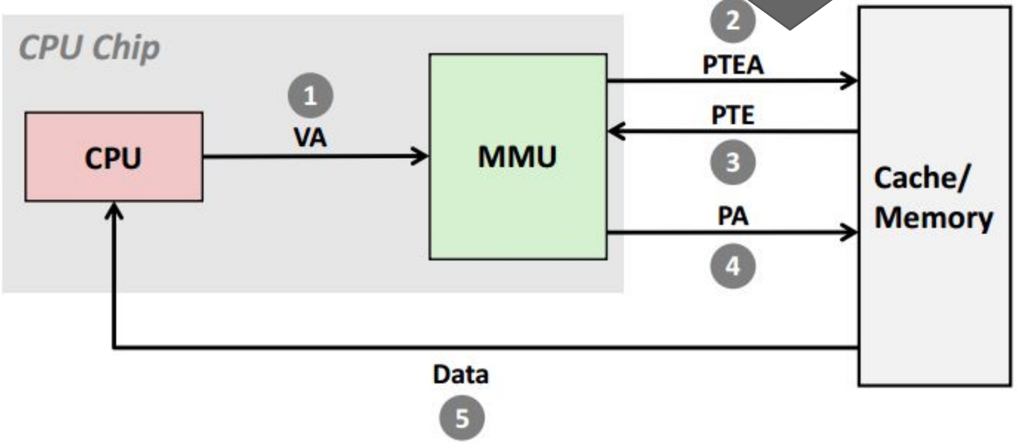
1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor

How many memory accesses are here? (i.e arrows into memory)



Address Translation: Page Hit

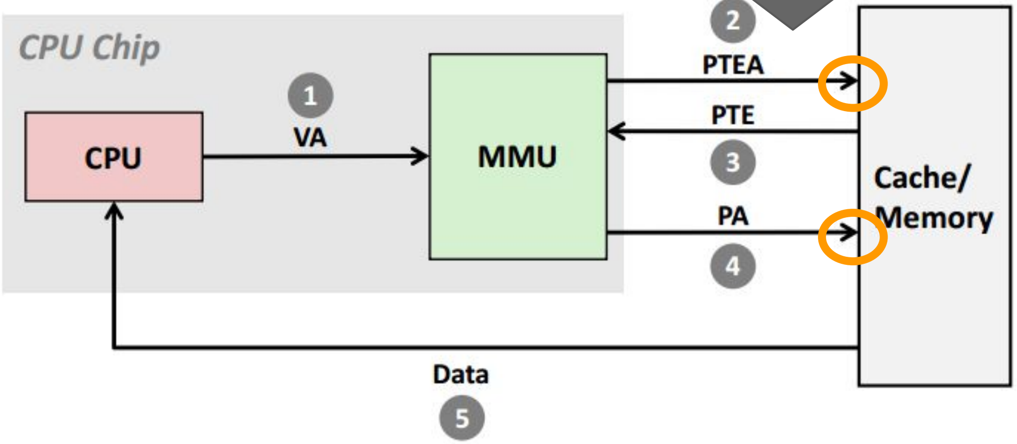
1.) Processor sends virtual address to MMU

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

4.) MMU Sends physical address to cache/memory

5.) Cache/memory sends data word to processor

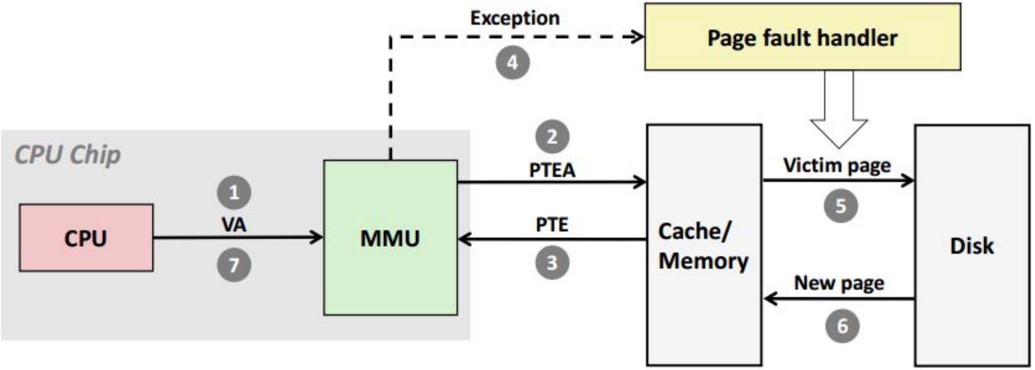
2! So two memory accesses --yikes, expensive!



Address Translation: Page Fault

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

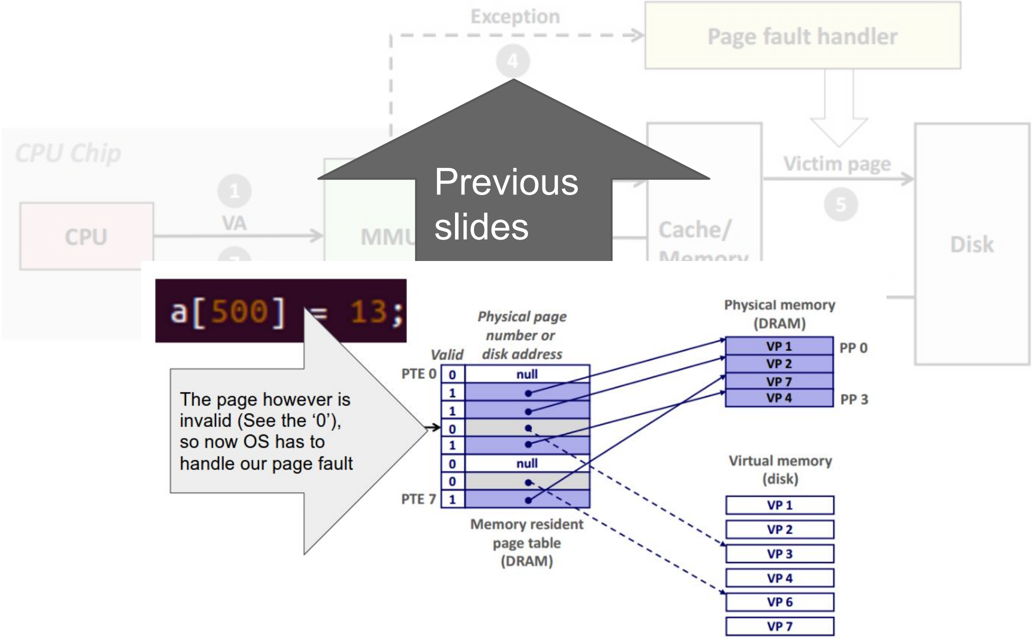
4.) Valid bit is zero; page fault exception!
5.) Handler identifies victim (pages it out to disk)
6.) Handler pages in new page and updates Page table entry in memory
7.) Handler returns to original process, restarting from our 'faulty' instruction



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



Previous slides

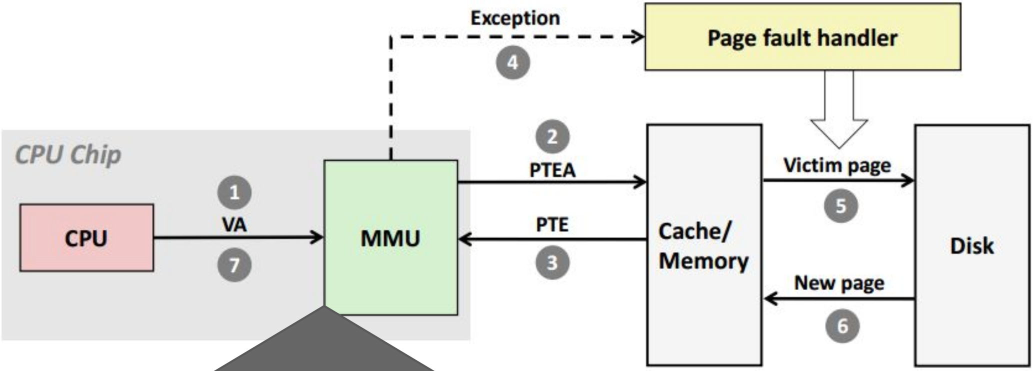
```
a[500] = 13;
```

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

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)

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)

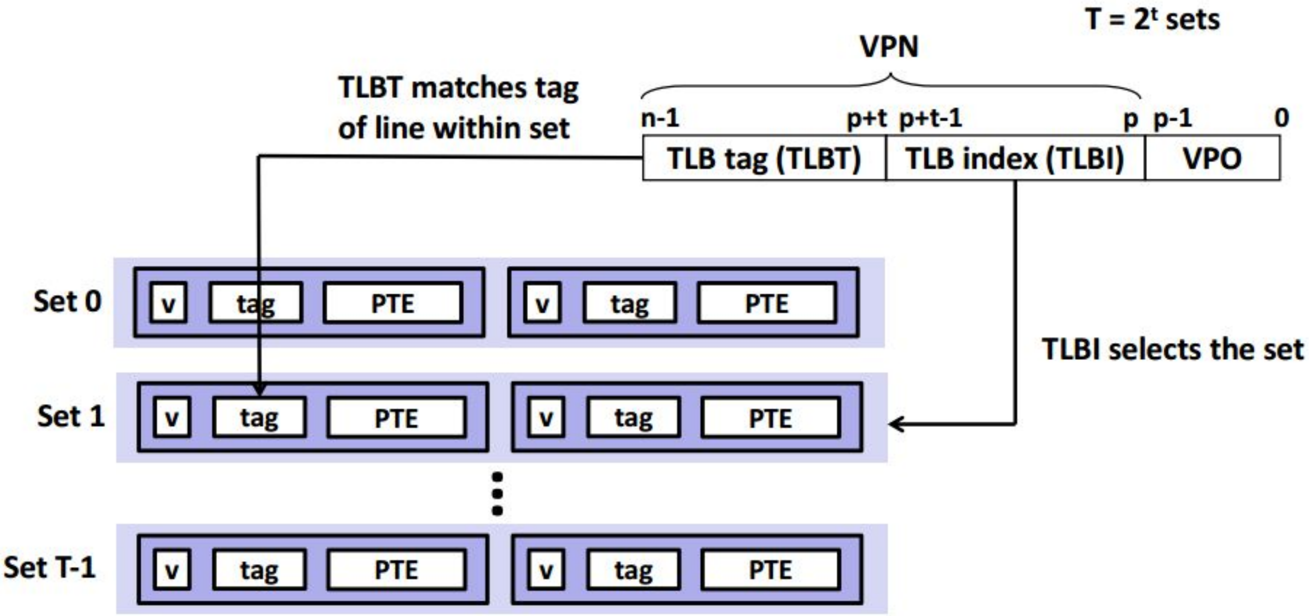
Address Translation - Notation

- 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

Two new items

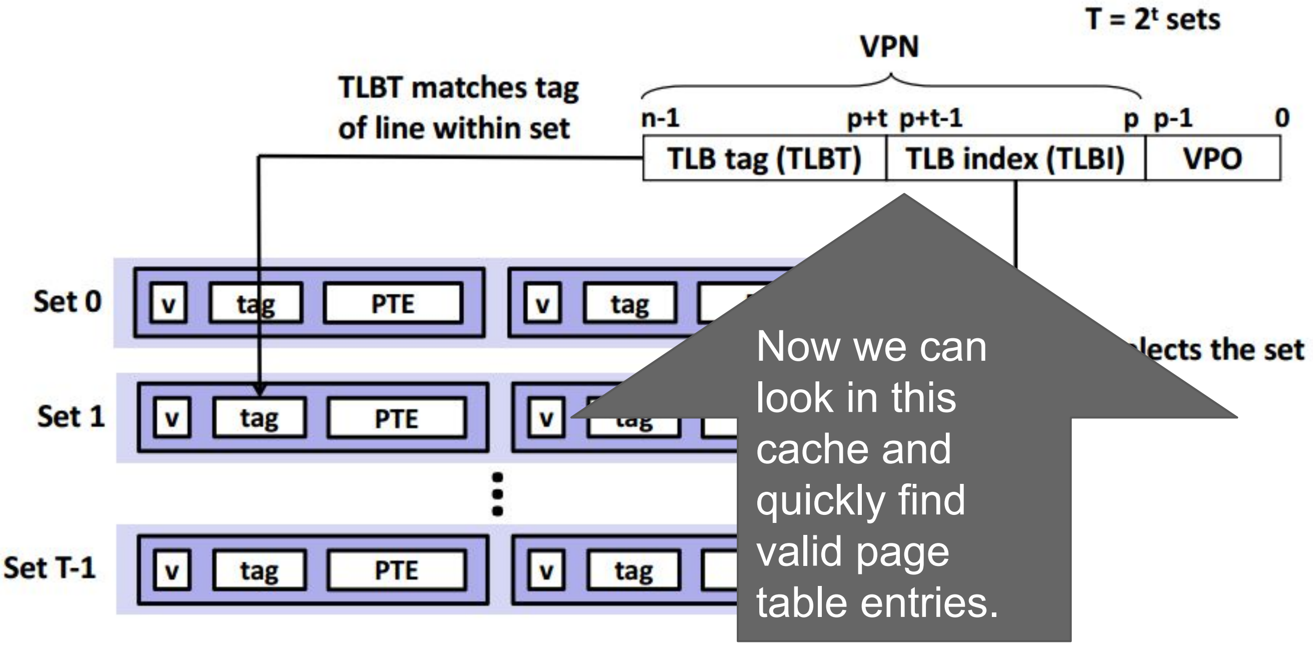
Accessing the Translation Lookaside Buffer (TLB)

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



Accessing the Translation Lookaside Buffer (TLB)

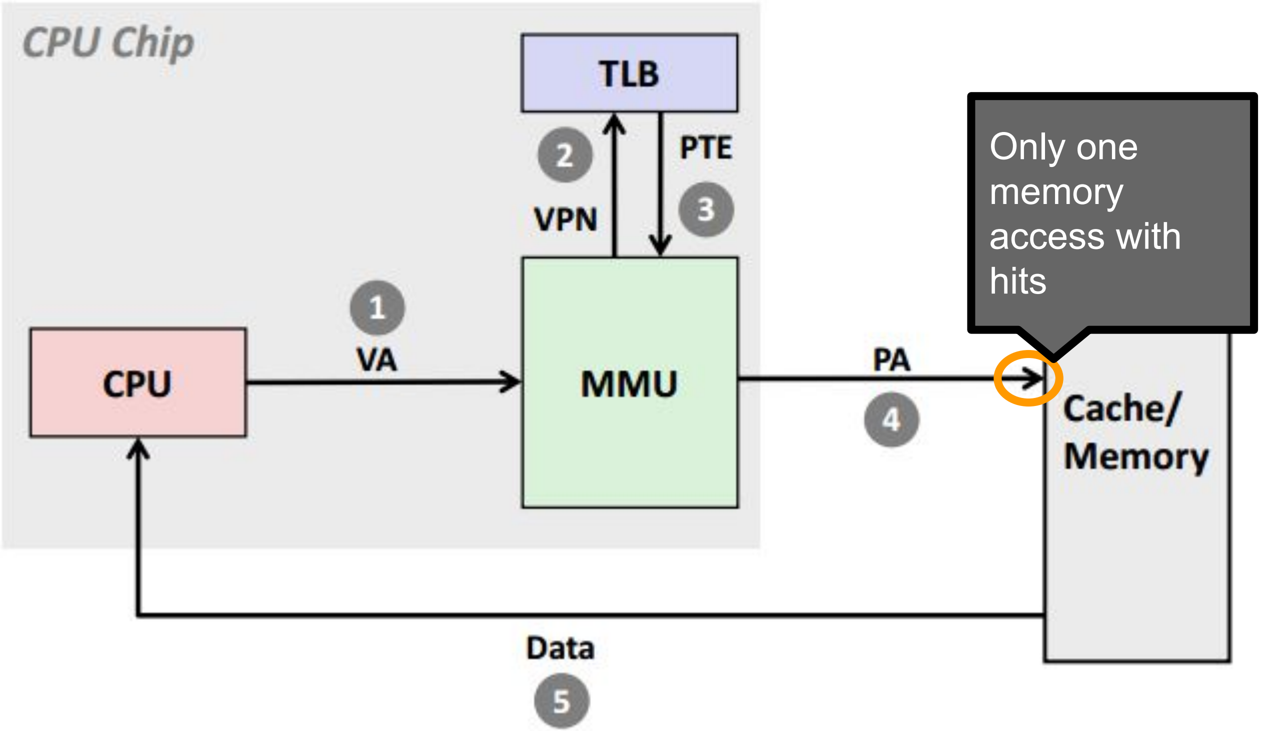
- This looks quite familiar to our set-associative cache!



Now we can look in this cache and quickly find valid page table entries.

Translation Lookaside Buffer (TLB) Hit

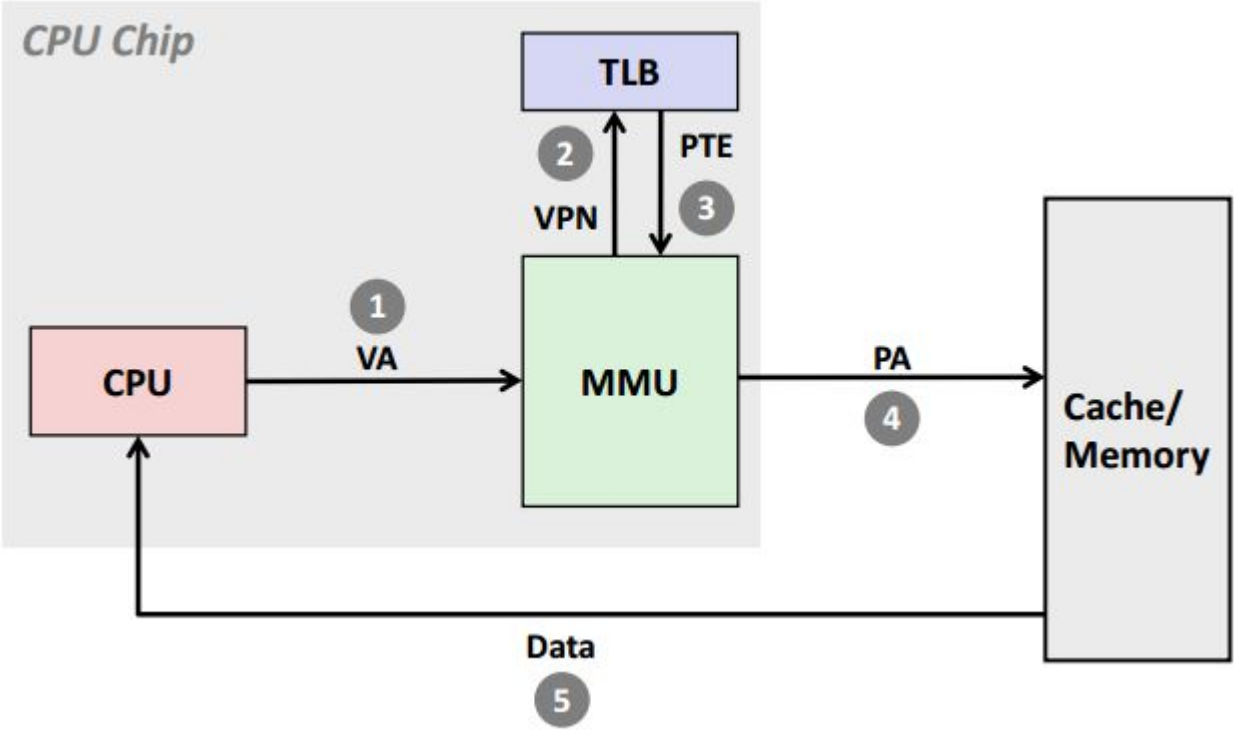
- On a hit, we reduce by 1 memory access
- In practice, misses are rare
  - We pay an extra memory access if so
  - Why?



Only one memory access with hits

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
  - TLB miss can generally be handled in hardware (Doesn't slow software)



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!)
- (Virtual memory and the concept of virtualization is also useful for things like containers and tools like Docker)
  - https://docs.docker.com/get-started/