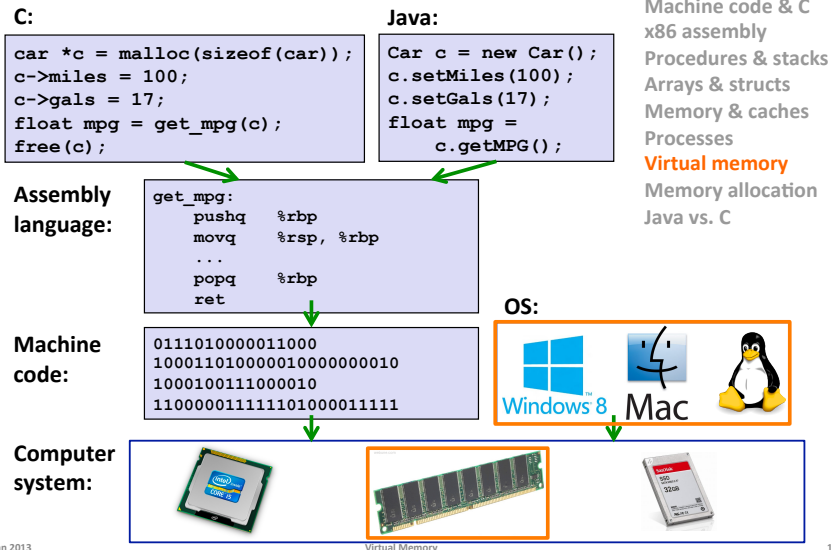


## Roadmap



Autumn 2013

Virtual Memory

1

## Again: Processes

- **Definition: A *process* is an instance of a running program**
  - One of the most important ideas in computer science
  - Not the same as “program” or “processor”
- **Process provides each program with **two key abstractions**:**
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private virtual address space
    - Each process seems to have exclusive use of main memory
- **How are these illusions maintained?**
  - Process executions interleaved (multi-tasking) – done...
  - Address spaces managed by virtual memory system – **now!**

Autumn 2013

Virtual Memory

2

## Virtual Memory (VM)

- Overview and motivation
- VM as tool for caching
- Address translation
- VM as tool for memory management
- VM as tool for memory protection

Autumn 2013

Virtual Memory

3

## Memory as we know it so far... is virtual!

- Programs refer to *virtual* memory addresses
  - `movl (%ecx), %eax`
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - System provides address space private to particular “process”
- **Allocation: Compiler and run-time system**
  - Where different program objects should be stored
  - All allocation within single virtual address space
- **But...**
  - We probably don’t have exactly  $2^w$  bytes of physical memory.
  - We *certainly* don’t have  $2^w$  bytes of physical memory for every process.
  - We have multiple processes that usually should not interfere with each other, but sometimes should share code or data.



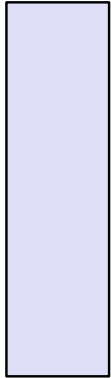
Autumn 2013

Virtual Memory

4

## Problem 1: How Does Everything Fit?

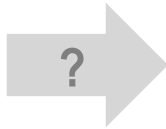
64-bit addresses can address several exabytes  
(18,446,744,073,709,551,616 bytes)



1 virtual address space per process, with many processes...

Physical main memory offers a few gigabytes  
(e.g. 8,589,934,592 bytes)

(Actually, it's smaller than that dot compared to virtual memory.)



## Problem 2: Memory Management

Process 1  
Process 2  
Process 3  
...  
Process n

X

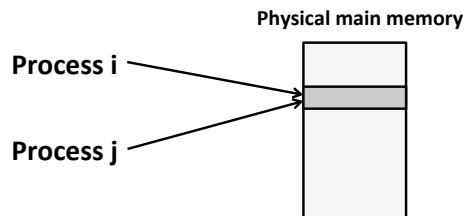
stack  
heap  
.text  
.data  
...

What goes where?

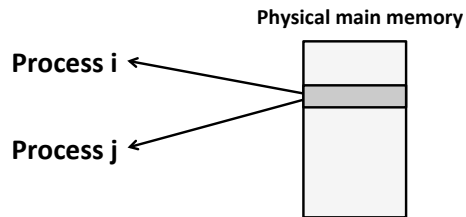
Physical main memory



## Problem 3: How To Protect



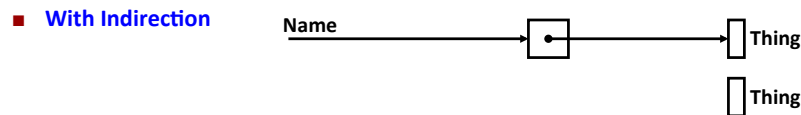
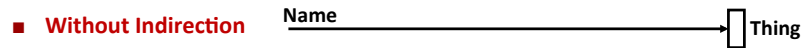
## Problem 4: How To Share?



## How can we solve these problems?

## Indirection

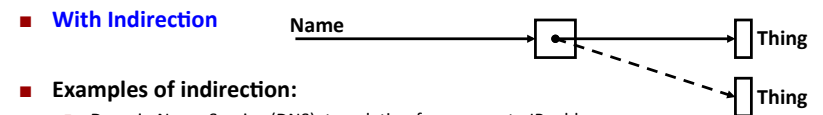
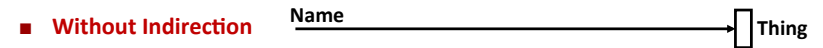
- **“Any problem in computer science can be solved by adding another level of indirection.”** –David Wheeler, inventor of the subroutine (a.k.a. procedure)



What if I want to move Thing?

## Indirection

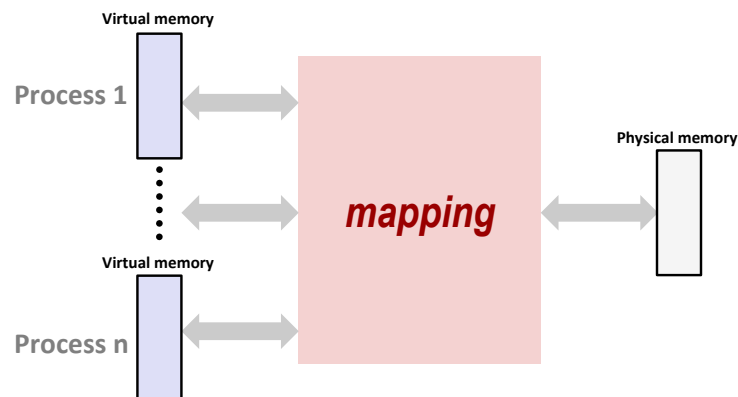
- **Indirection:** the ability to reference something using a name, reference, or container instead of the value itself. A flexible mapping between a name and a thing allows changing the thing without notifying holders of the name.



- **Examples of indirection:**

- Domain Name Service (DNS): translation from name to IP address
- phone system: cell phone number portability
- snail mail: mail forwarding
- 911: routed to local office
- Dynamic Host Configuration Protocol (DHCP): local network address assignment
- call centers: route calls to available operators, etc.

## Indirection in Virtual Memory

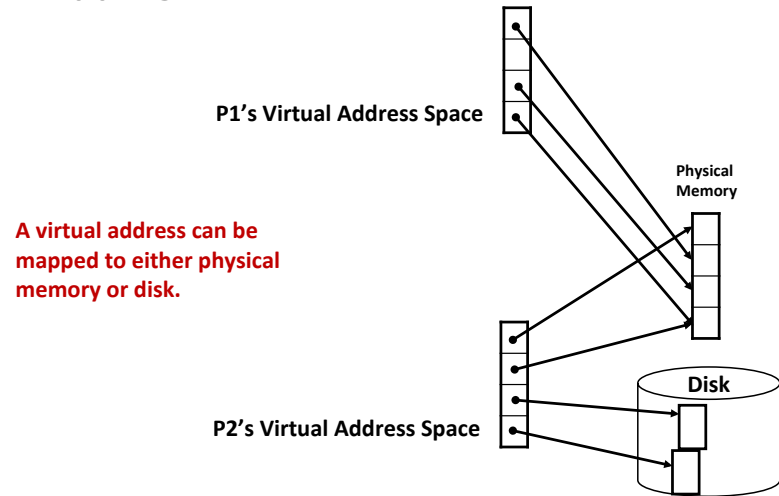


- Each process gets its own private virtual address space
- Solves the previous problems

## Address Spaces

- **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 ( $n \geq m$ )  $\{0, 1, 2, 3, \dots, M-1\}$
- **Every byte in main memory has:**
  - one physical address
  - zero, one, or more virtual addresses

## Mapping

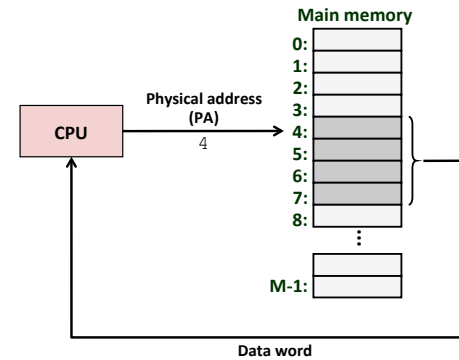


Autumn 2013

Virtual Memory

13

## A System Using Physical Addressing



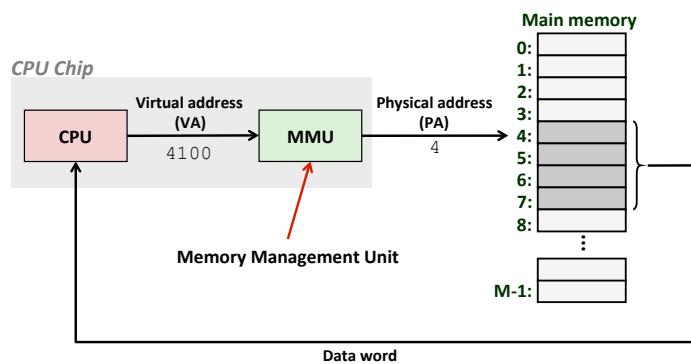
- Used in “simple” systems with (usually) just one process:
  - embedded microcontrollers in devices like cars, elevators, and digital picture frames

Autumn 2013

Virtual Memory

14

## A System Using Virtual Addressing



- Physical addresses are *completely invisible to programs*.
- Used in all modern desktops, laptops, servers, smartphones...
- One of the great ideas in computer science

Autumn 2013

Virtual Memory

15

## Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed
- **Simplifies memory management for programmers**
  - Each process gets the same full, private linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions

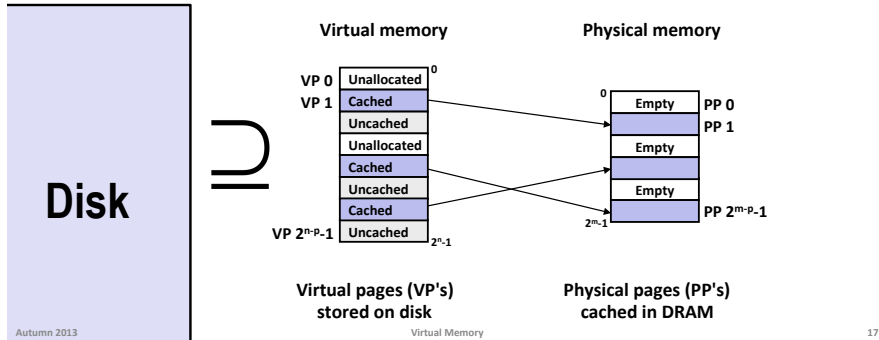
Autumn 2013

Virtual Memory

16

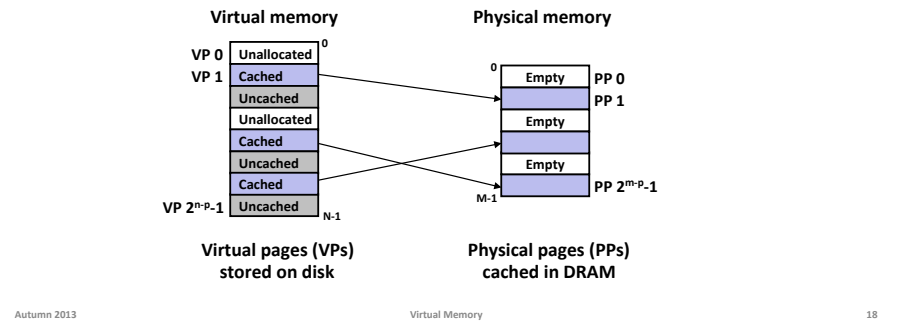
## VM and the Memory Hierarchy

- Think of virtual memory as array of  $N = 2^n$  contiguous bytes.
- **Pages** of virtual memory are usually stored in physical memory, but sometimes spill to disk.
  - Pages are another unit of aligned memory (size is  $P = 2^p$  bytes)
  - Each virtual page can be stored in *any* physical page.



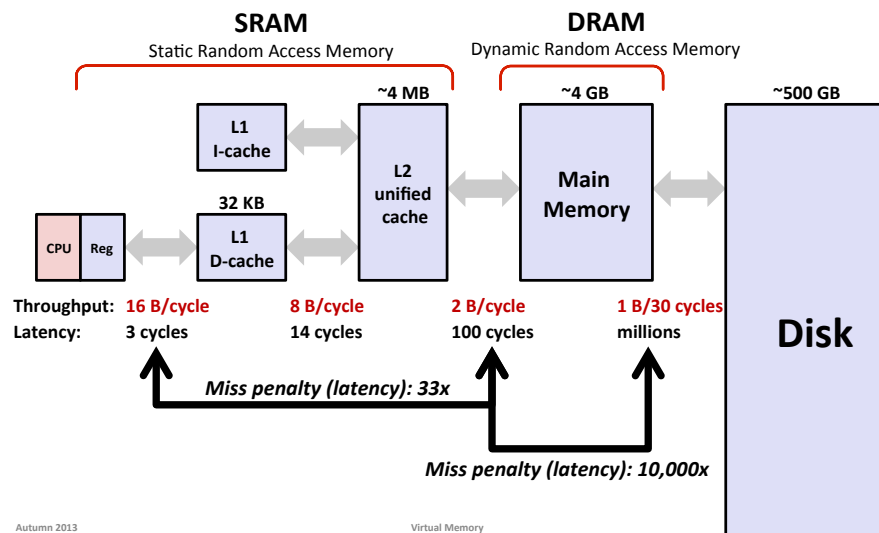
## or: Virtual Memory as DRAM Cache for Disk

- Think of virtual memory as an array of  $N = 2^n$  contiguous bytes stored *on a disk*.
- Then physical main memory is used as a **cache** for the virtual memory array
  - The cache blocks are called **pages** (size is  $P = 2^p$  bytes)



## Memory Hierarchy: Core 2 Duo

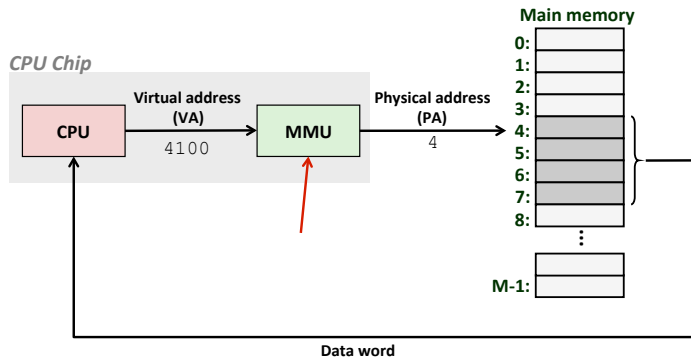
Not drawn to scale



## Virtual Memory Design Consequences

- **Large page size:** typically 4-8 KB, sometimes up to 4 MB
- **Fully associative**
  - Any virtual page can be placed in any physical page
  - Requires a “large” mapping function – different from CPU caches
- **Highly sophisticated, expensive replacement algorithms in OS**
  - Too complicated and open-ended to be implemented in hardware
- **Write-back rather than write-through**

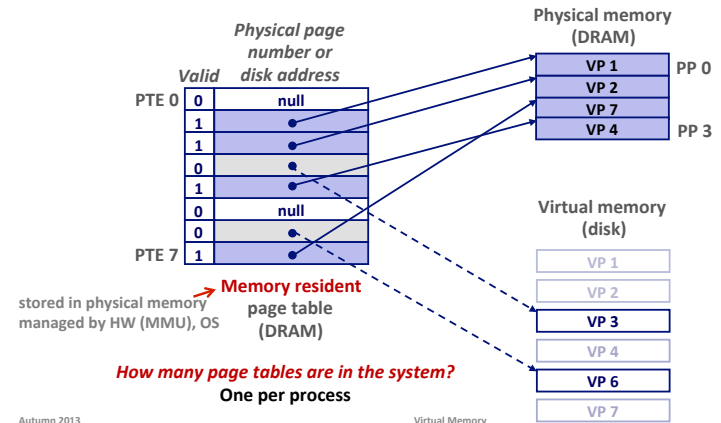
# Address Translation



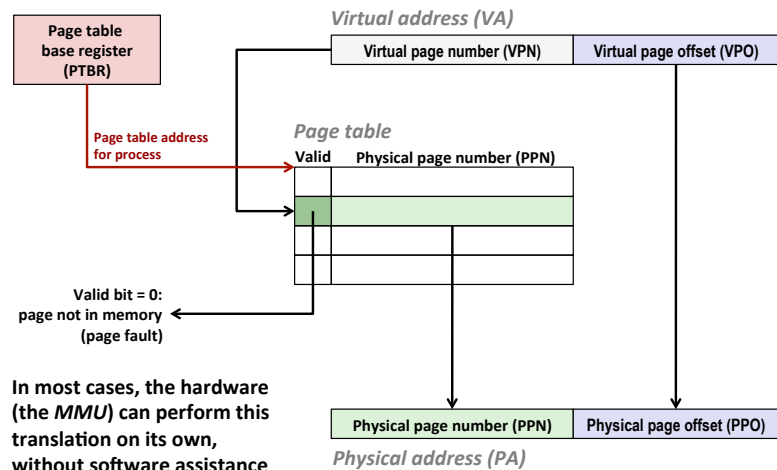
How do we perform the virtual -> physical address translation?

# Address Translation: Page Tables

A **page table** is an array of **page table entries** (PTEs) that maps virtual pages to physical pages.



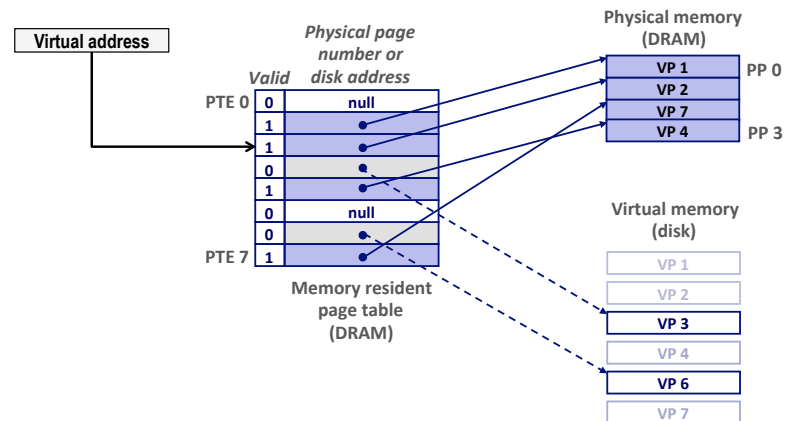
# Address Translation With a Page Table



This feels familiar...

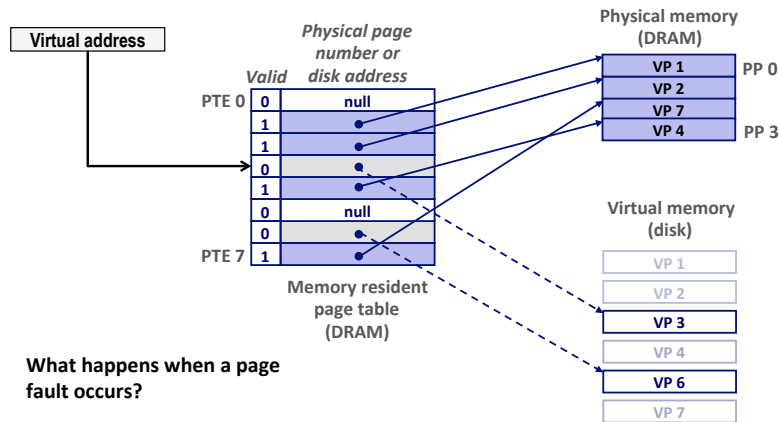
# Page Hit

**Page hit:** reference to VM byte that is in physical memory



# Page Fault

- **Page fault:** reference to VM byte that is **NOT** in physical memory

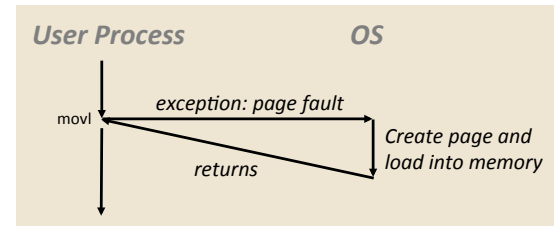


# Fault Example: Page Fault

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

- User writes to memory location
- That portion (page) of user's memory is currently on disk

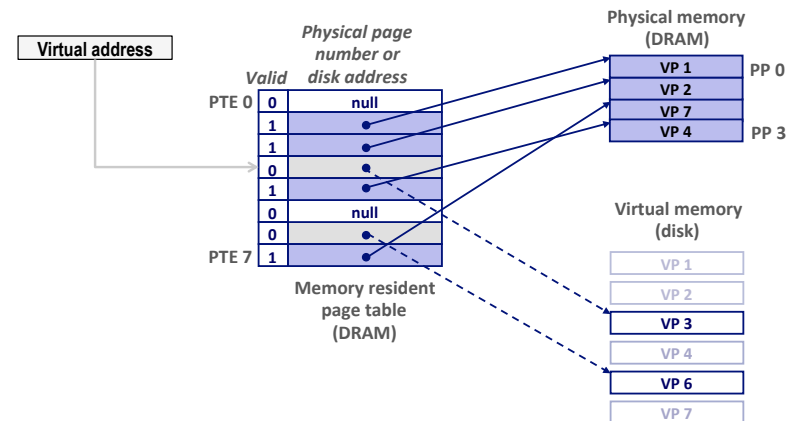
80483b7: c7 05 10 9d 04 08 0d movl \$0xd,0x8049d10



- Page handler must load page into physical memory
- Returns to faulting instruction: `movl` is executed *again*!
- Successful on second try

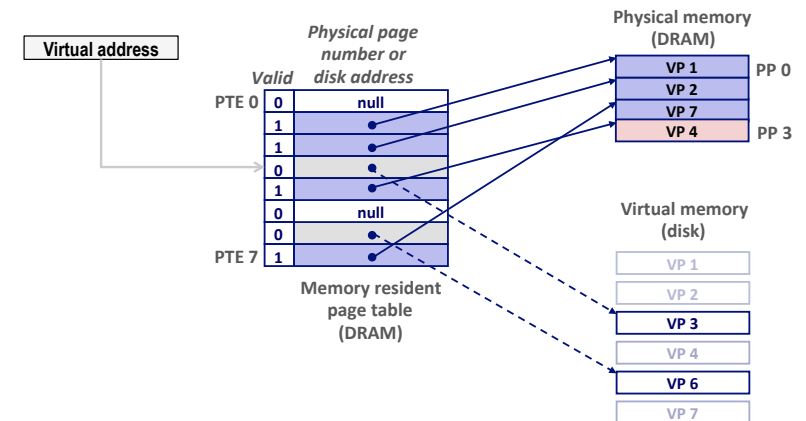
# Handling Page Fault

- Page miss causes page fault (an exception)



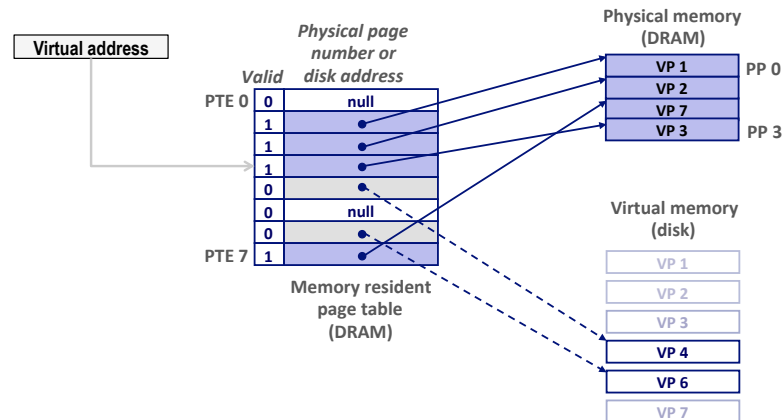
# Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)



## Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)



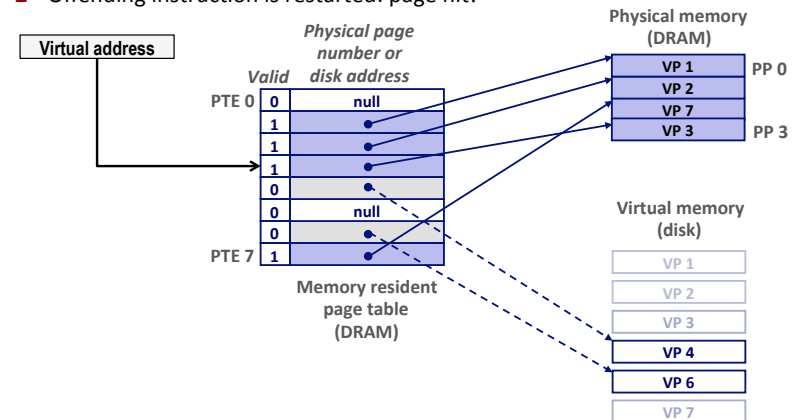
Autumn 2013

Virtual Memory

29

## Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a *victim* to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Autumn 2013

Virtual Memory

30

## Why does it work?

Autumn 2013

Virtual Memory

31

## Why does VM work on RAM/disk? Locality.

- Virtual memory works well for avoiding disk accesses because of locality
  - Same reason that L1 / L2 / L3 caches work
- The set of virtual pages that a program is “actively” accessing at any point in time is called its *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size of one process < main memory size):
  - Good performance for one process after compulsory misses
- But if  $\text{SUM}(\text{working set sizes of all processes}) > \text{main memory size}$ :
  - **Thrashing**: Performance meltdown where pages are swapped (copied) between memory and disk continuously. CPU always waiting or paging.
  - Full quote: “Every problem in computer science can be solved by adding another level of indirection, but that usually will create another problem.”

Autumn 2013

Virtual Memory

32



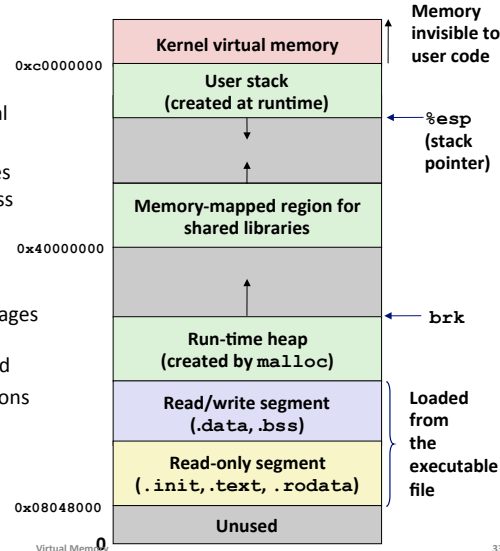
## Simplifying Linking and Loading

### ■ Linking

- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

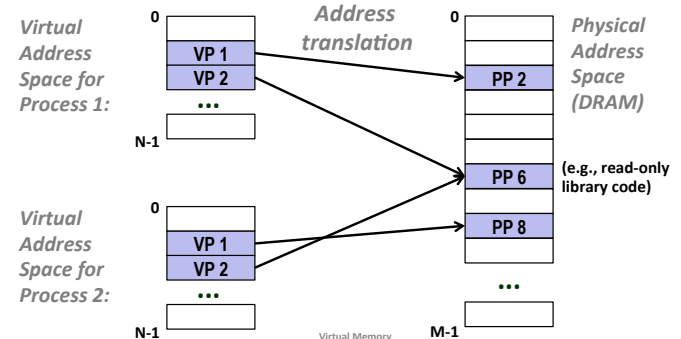
### ■ Loading

- `execve()` allocates virtual pages for `.text` and `.data` sections = creates PTEs marked as invalid
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system



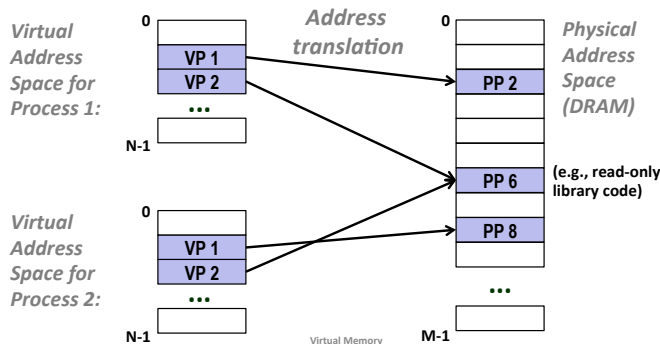
## VM for Managing Multiple Processes

- **Key abstraction:** each process has its own virtual address space
  - It can view memory as a *simple linear array*
- **With virtual memory, this simple linear virtual address space need not be contiguous in physical memory**
  - Process needs to store data in another VP? Just map it to *any* PP!



## VM for Protection and Sharing

- The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes.
  - **Sharing:** just map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection:** process simply can't access physical pages to which none of its virtual pages are mapped (here: Process 2 can't access PP 2).

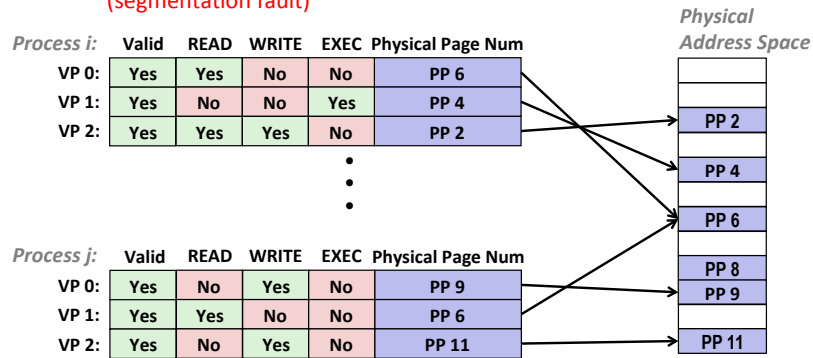


## Memory Protection Within a Single Process

- Can we use virtual memory to control read/write/execute permissions? How?

## Memory Protection Within a Single Process

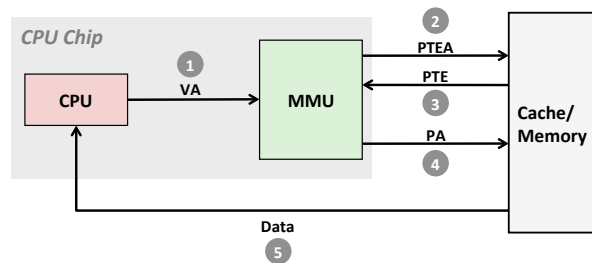
- Extend page table entries with permission bits
- MMU checks these permission bits on every memory access
  - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)



## Terminology

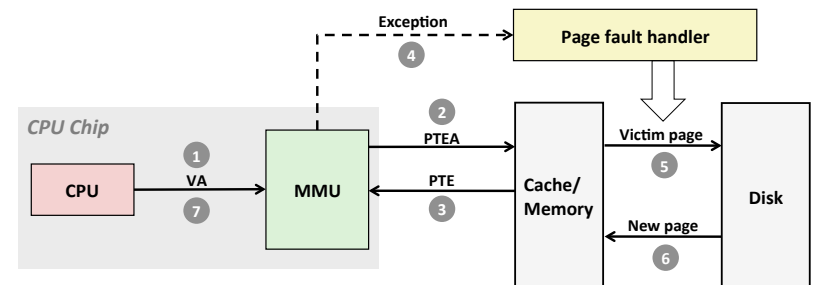
- context switch
  - Switch between processes on the same CPU
- page in
  - Move pages of virtual memory from disk to physical memory
- page out
  - Move pages of virtual memory from physical memory to disk
- thrash
  - Total working set size of processes is larger than physical memory
  - Most time is spent paging in and out instead of doing useful computation

## Address Translation: Page Hit



- 1) Processor sends virtual address to MMU (*memory management unit*)
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

## Address Translation: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in cache/memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

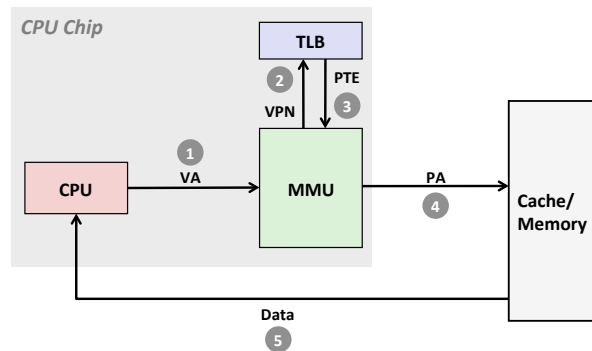
## Hmm... Translation Sounds Slow!

- The MMU accesses memory *twice*: once to first get the PTE for translation, and then again for the actual memory request from the CPU
  - The PTEs *may* be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles
- *What can we do to make this faster?*

## Speeding up Translation with a TLB

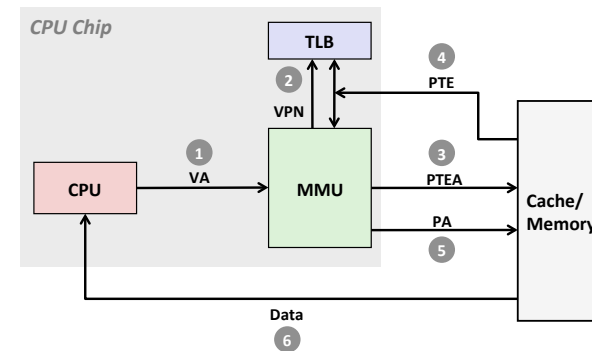
- **Solution: add another cache!**
- **Translation Lookaside Buffer (TLB):**
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete *page table entries* for small number of pages
    - Modern Intel processors: 128 or 256 entries in TLB
  - Much faster than a page table lookup in cache/memory

## TLB Hit



A TLB hit eliminates a memory access

## TLB Miss

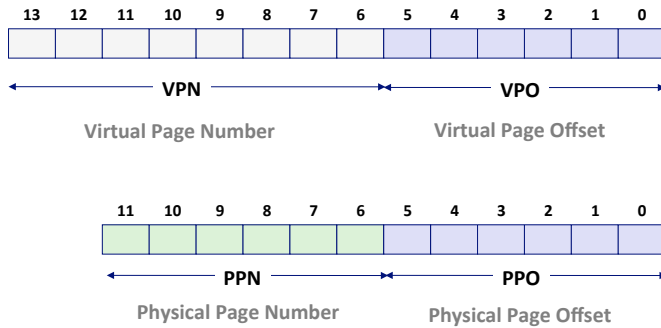


A TLB miss incurs an additional memory access (the PTE)  
 Fortunately, TLB misses are rare. **Does a TLB miss require disk access?**

## Simple Memory System Example (small)

### ■ Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes



## Simple Memory System Page Table

- Only showing first 16 entries (out of 256 = 2<sup>8</sup>)

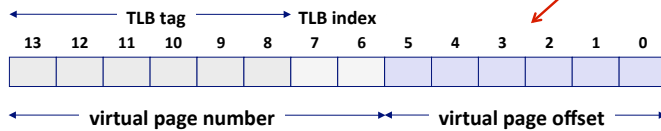
VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	-	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	-	0
04	-	0	0C	-	0
05	16	1	0D	2D	1
06	-	0	0E	11	1
07	-	0	0F	0D	1

- What about a real address space? Read more in the book...

## Simple Memory System TLB

- 16 entries
- 4-way associative

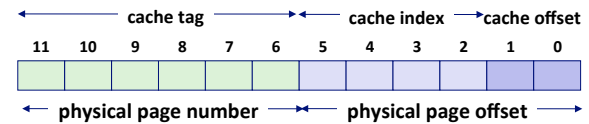
TLB ignores page offset. Why?



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

## Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped



Idx	Tag	Valid	B0	B1	B2	B3	Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11	8	24	1	3A	00	51	89
1	15	0	-	-	-	-	9	2D	0	-	-	-	-
2	1B	1	00	02	04	08	A	2D	1	93	15	DA	3B
3	36	0	-	-	-	-	B	0B	0	-	-	-	-
4	32	1	43	6D	8F	09	C	12	0	-	-	-	-
5	0D	1	36	72	F0	1D	D	16	1	04	96	34	15
6	31	0	-	-	-	-	E	13	1	83	77	1B	D3
7	16	1	11	C2	DF	03	F	14	0	-	-	-	-

# Current state of caches/tables

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	-	0	09	0D	1	00	-	0	07	02	1
1	03	2D	1	02	-	0	04	-	0	0A	-	0
2	02	-	0	08	-	0	06	-	0	03	-	0
3	07	-	0	03	0D	1	0A	34	1	02	-	0

VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	08	13	1
01	-	0	09	17	1
02	33	1	0A	09	1
03	02	1	0B	-	0
04	-	0	0C	-	0
05	16	1	0D	2D	1
06	-	0	0E	11	1
07	-	0	0F	0D	1

Page table

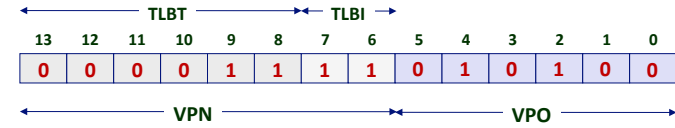
Cache

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	-	-	-	-
2	1B	1	00	02	04	08
3	36	0	-	-	-	-
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	-	-	-	-
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	-	-	-	-
A	2D	1	93	15	DA	3B
B	0B	0	-	-	-	-
C	12	0	-	-	-	-
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	-	-	-	-

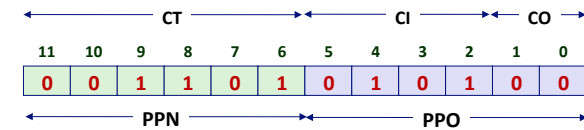
# Address Translation Example #1

Virtual Address: 0x03D4



VPN 0x0F TLBI 3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

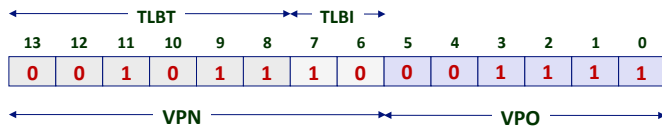
Physical Address



CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36

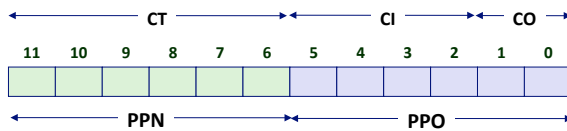
# Address Translation Example #2

Virtual Address: 0x0B8F



VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? N Page Fault? Y PPN: TBD

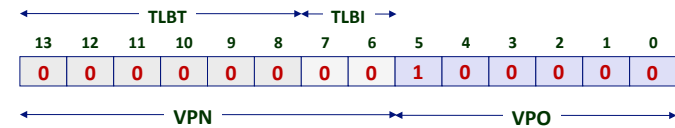
Physical Address



CO \_\_\_ CI \_\_\_ CT \_\_\_ Hit? \_\_\_ Byte: \_\_\_

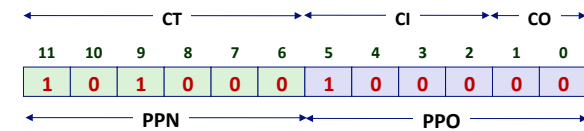
# Address Translation Example #3

Virtual Address: 0x0020



VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

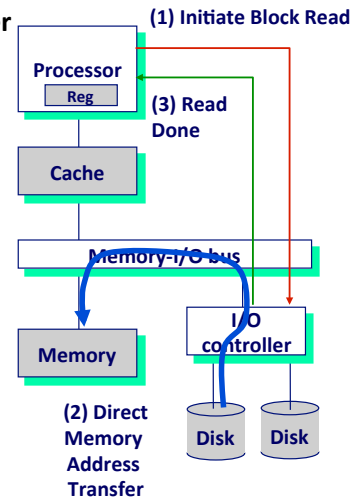
Physical Address



CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem

## Servicing a Page Fault

- **(1) Processor signals disk controller**
  - Read block of length P starting at disk address X
  - Store starting at memory address Y
- **(2) Read occurs**
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- **(3) Controller signals completion**
  - Interrupts processor
  - OS resumes suspended process



## Summary

- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and sharing
  - Simplifies protection by providing a convenient interpositioning point to check permissions

## Memory System Summary

- **L1/L2 Memory Cache**
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware
- **Virtual Memory**
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)

## Memory System Summary

- **L1/L2 Memory Cache**
  - Controlled by hardware
  - Programmer cannot control it
  - Programmer *can* write code in a way that takes advantage of it
- **Virtual Memory**
  - Controlled by OS and hardware
  - Programmer cannot control mapping to physical memory
  - Programmer can control sharing and some protection
    - via OS functions (not in 351)