Chapter 9: Virtual Memory

Topics

- Motivations for virtual memory VM (not to be confused with virtual machines VM)
- Address translation
- Accelerating translation with TLBs

Announcements

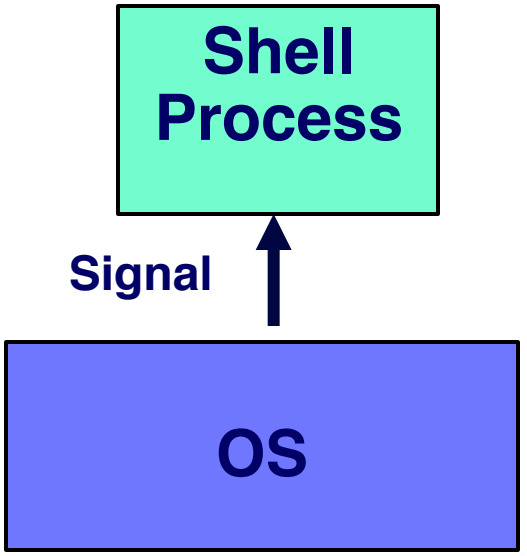
- Shell lab is due Monday Dec 8 by 8 am
  - Interview grading time slots for next week available later this week
- Last Recitation Exercise #5 due Friday Dec 12 by 5 pm
  - Upload to moodle or hand into TA at TA office hours
- Reading:
  - Read Chapter 9, except 9.6 and 9.7 (no case study and no memory mapping, can also skip multi-level page tables)
- Need a volunteer for FCQs on Thursday
- Final exam is Thursday Dec 18, 4:30-7 pm, more next week

Recap

- **How does software execute on modern computers?**
  - It interacts with OS and hardware via HW/SW interrupts, faults, …
- **A Process is an actively executing program**
  - Use `fork()` to create a child process in Unix/Linux
  - Use `exit()` to finish executing a process
  - Parent calls `wait()/waitpid()` to free child process' resources
- **A Shell process**
  - Forked child can run in foreground or background (type "&" at end of command line)
  - Use signals to properly reap child processes and respond to CTL-C/ CTL-Z

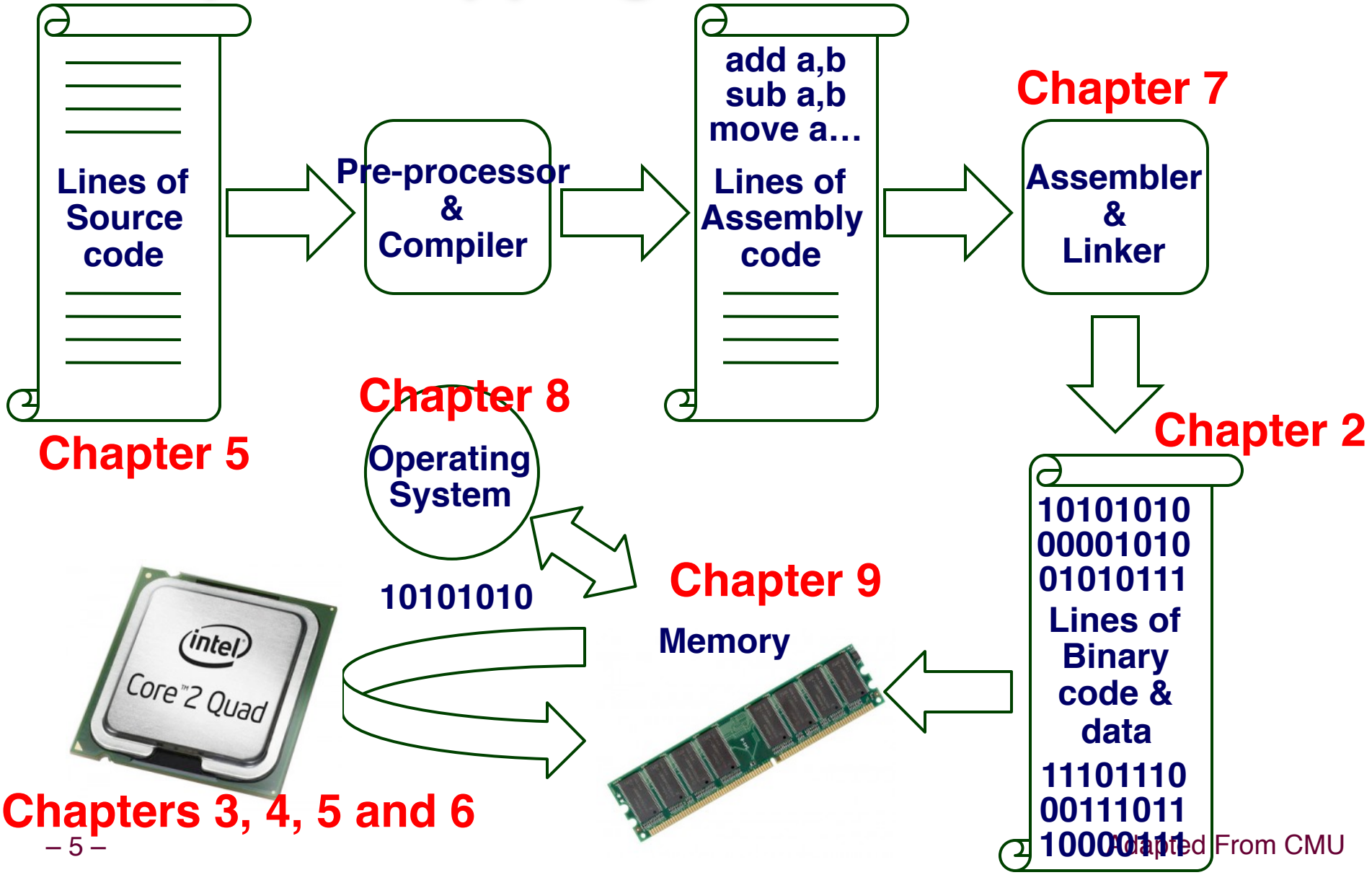
```
int main() {
    char cmdline[MAXLINE];
    ...
    while (1) {
        ...
        /* get string */
        Fgets(cmdline, MAXLINE, stdin);
        ...
        /* execute string */
        if ((pid = Fork()) == 0) {
            /* child runs user job */
            execve(args from cmdline);
        }
        ...
    }
}
```

Recap



- Your shell will have to handle signals
  - CTL-C sends a SIGINT
  - CTL-Z sends a SIGTSTP
  - Child process exiting sends a SIGCHLD
- Register your signal handler with `signal()` or `sigaction()`
- Signals don't queue
  - pending bit indicates at least one signal of that type has occurred – so add a while loop to handler
- Race conditions may occur
  - If signal handler modifies global state while normal control flow also modifies global state – see Ch 8 slides' 2 examples
  - Use masking of signals (or synchronization) to mitigate race conditions

Chapter Mapping

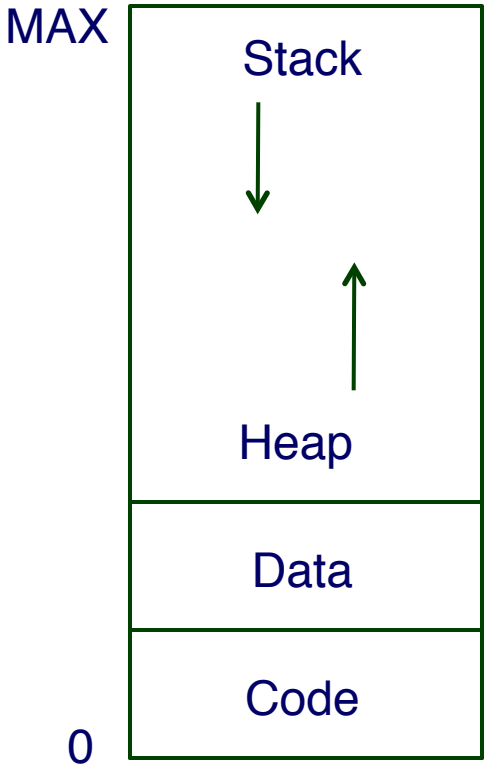


Motivations for Virtual Memory

Provides the Abstraction that a Process is executing in its own Address Space of memory from address 0 to address MAX

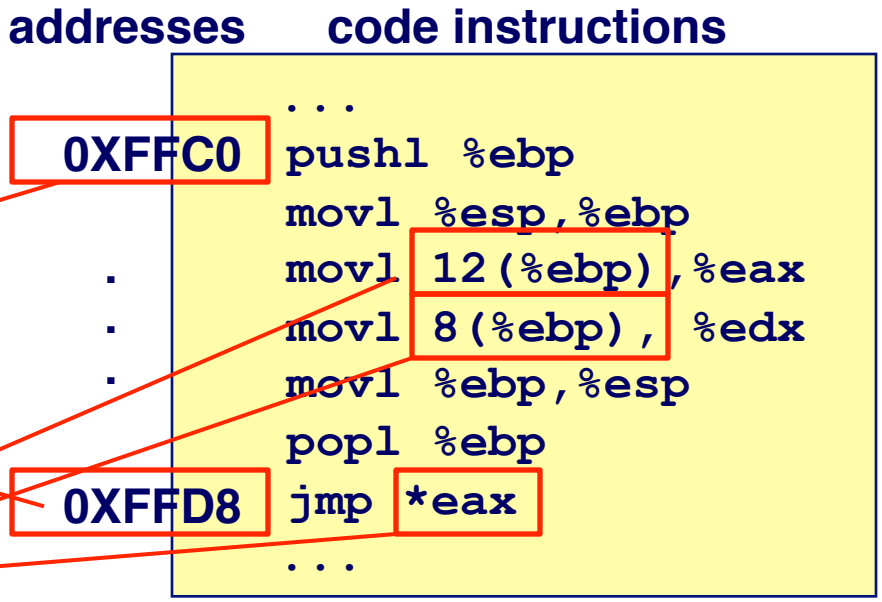
For a Single Process:

- **Compiler and linker can generate code targeting a [0..MAX] memory paradigm**
- **Can be placed anywhere in memory, and virtual addresses are *translated* to physical addresses at run time as each instruction executes**
- **frees a process and the compiler from having to worry where in physical memory the process is or will be located**



Virtual Addresses

- **Assembly code is compiled assuming virtual addresses**
  - address of each instruction is a virtual address, not a physical address
  - data referenced in memory is referred to by its virtual address, not its actual physical address
  - executing program is completely unaware of where it is actually located in physical memory



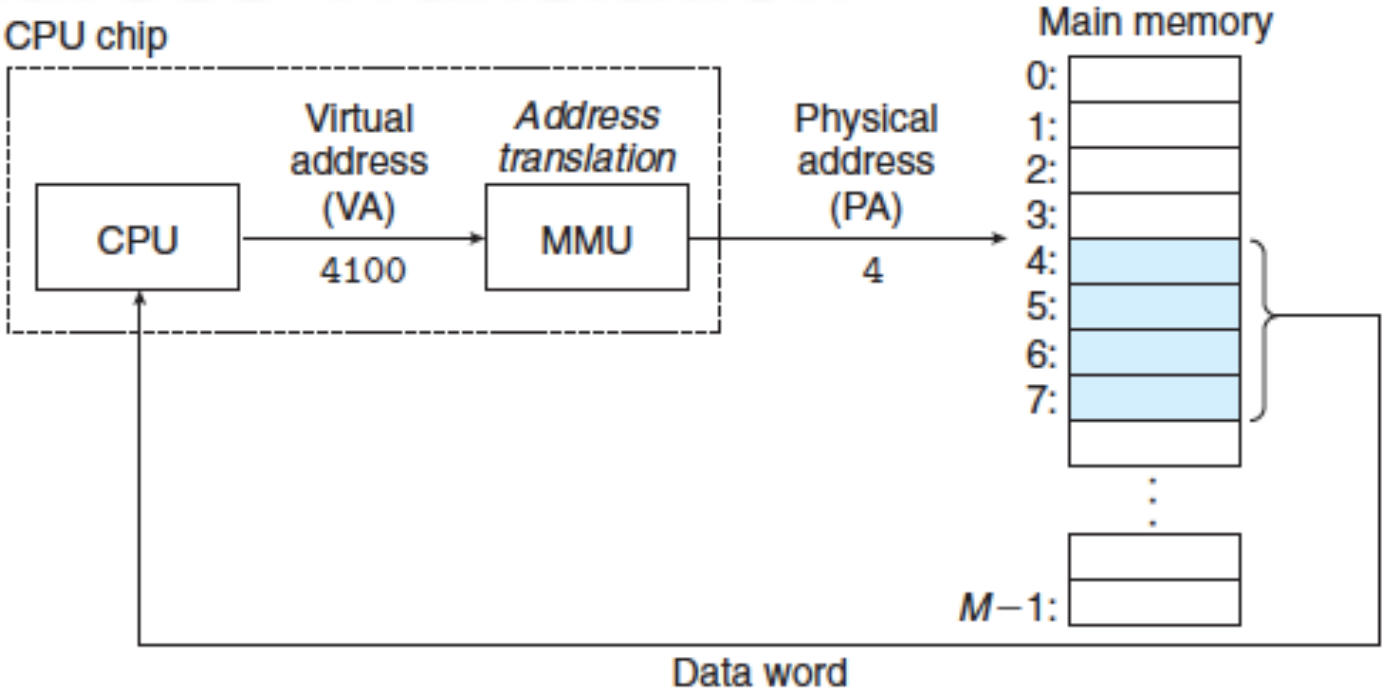
MMU Translates Virtual Addresses to Physical Addresses

As the program executes, a Memory Management Unit (MMU) in the CPU does the following:

- When fetching the next instruction, the MMU translates the virtual address of the instruction into a physical address where the instruction is actually located and fetches it
- When an instruction references a data location in memory, the MMU translates the data's virtual address into its actual physical address, and fetches the data

addresses	code instructions
	`...`
0XFFC0	`pushl %ebp`
	`movl %esp,%ebp`
.	`movl 12(%ebp),%eax`
.	`movl 8(%ebp), %edx`
.	`movl %ebp,%esp`
	`popl %ebp`
0XFFD8	`jmp *eax`
	`...`

MMU: Virtual -> Physical Address Translation



The MMU is a unit in the CPU that operates invisibly, underneath the program, translating its virtual addresses to physical addresses

- **The OS keeps track for each process of how virtual addresses are mapped to physical addresses and informs MMU**

Fragmentation Problem

Virtual addresses allow programs to be placed anywhere in main memory

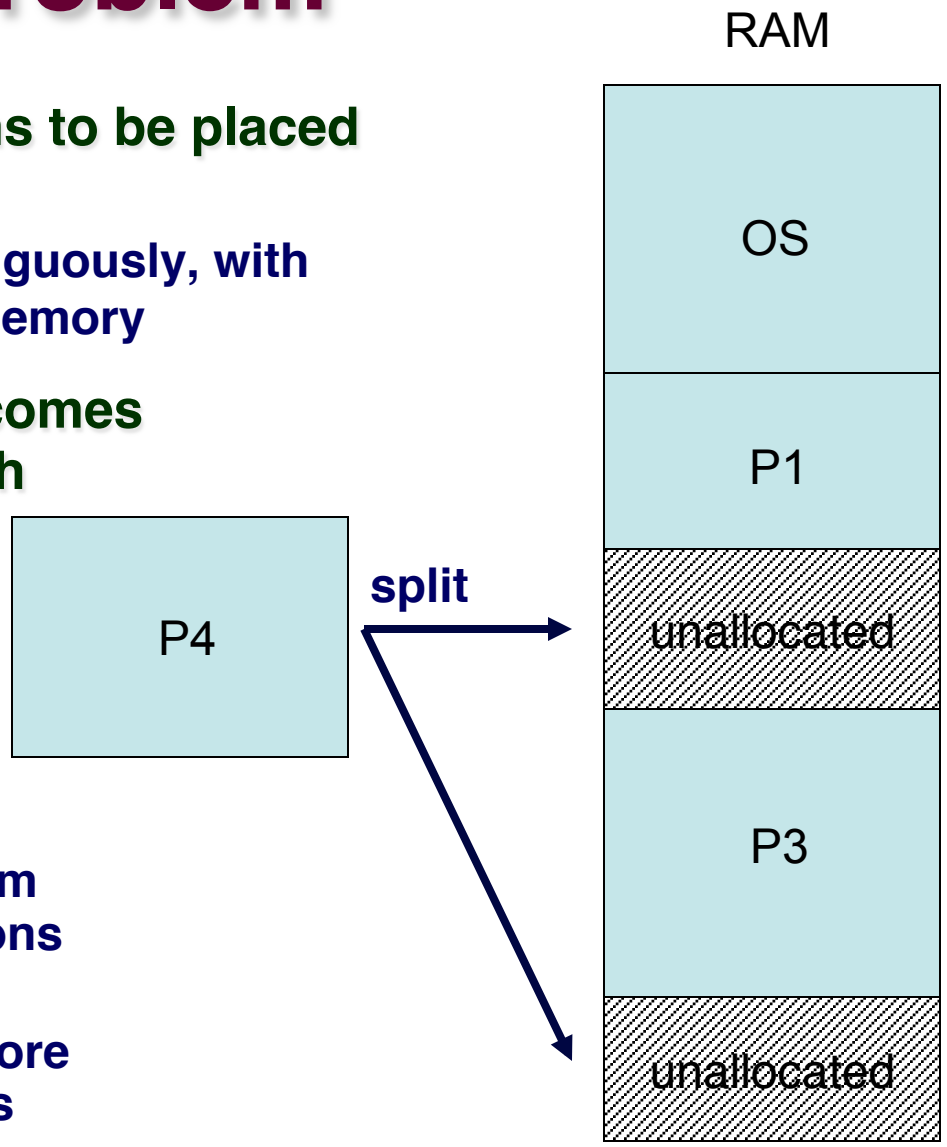
- **P1, P2, and P3 are placed contiguously, with some remaining unallocated memory**

But over time, main memory becomes fragmented as processes finish

- **P2 finishes, and its memory is deallocated**
- **P4 can't fit into any of the unallocated regions**

Better approach:

- **Split P4 into pieces, and fit them each into the unallocated regions of memory**
- **Advantage: memory is used more efficiently, and more processes can run**



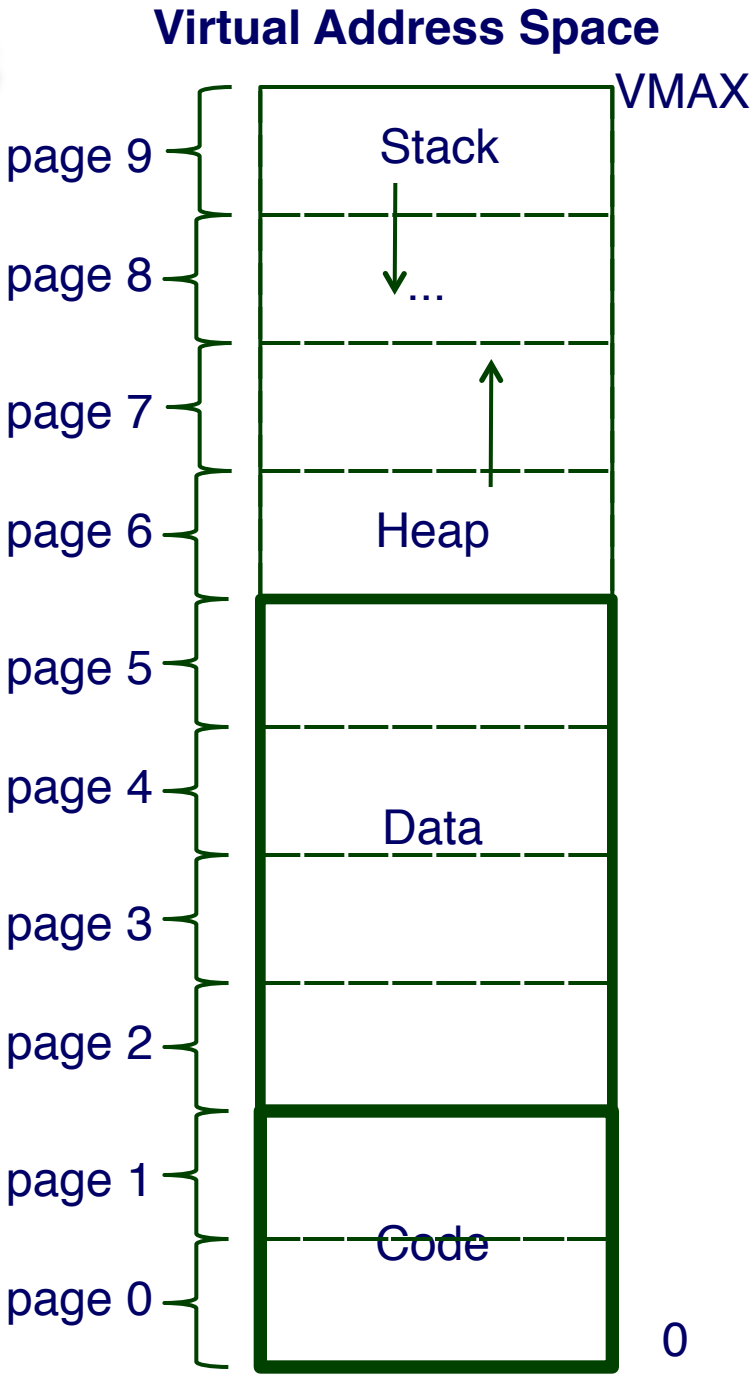
Dividing the Address Space into Pages

Divide the Address Space into Fixed-size Pages

- Can put pages anywhere in physical memory – don't need contiguous space in physical memory to layout entire virtual address space of each process
- Solves fragmentation problem
- Typical page size is 4 KB – depends on OS
- virtual page #

= virtual address/sizeof(page)

= the most significant bits of the virtual address!



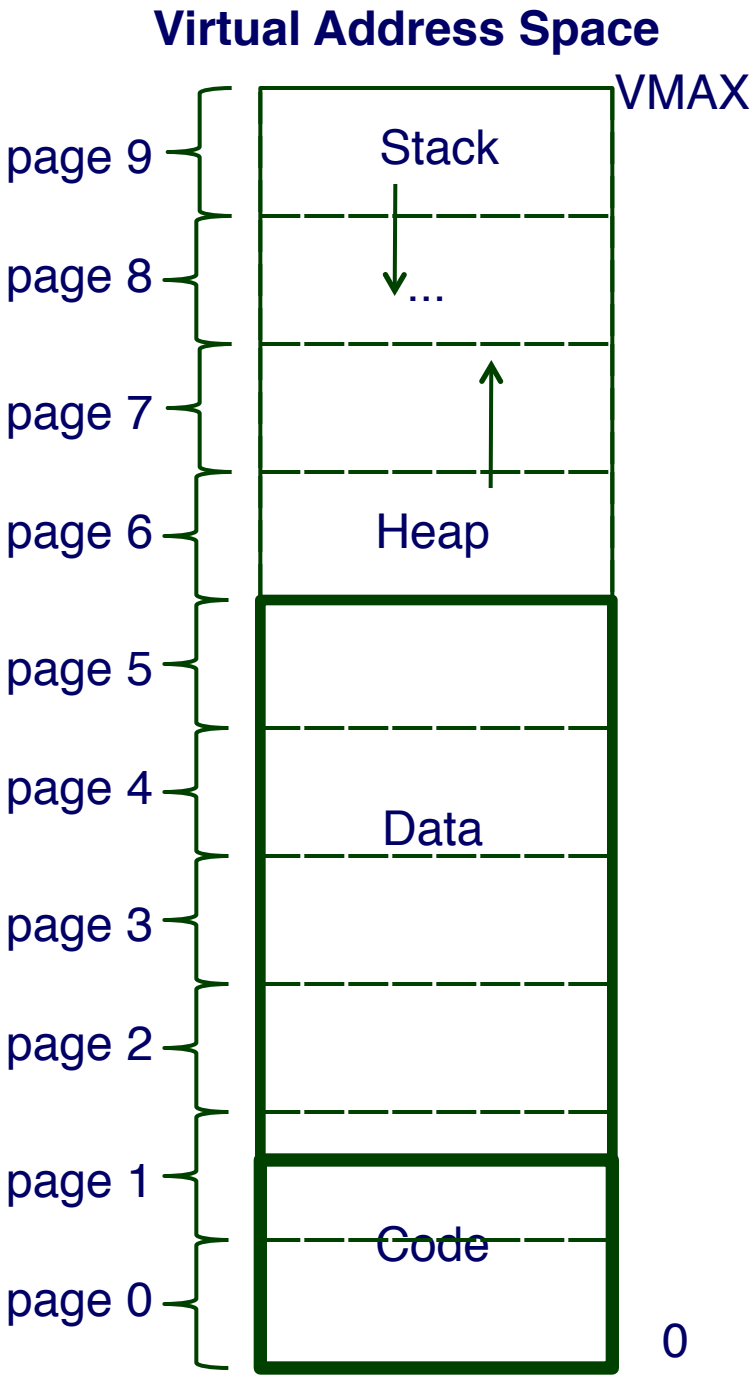
Page Tables

Divide Main Memory into Fixed-size pages as well

Must keep track of which virtual pages map to which physical pages

Introduce a data structure per process called a Page Table that:

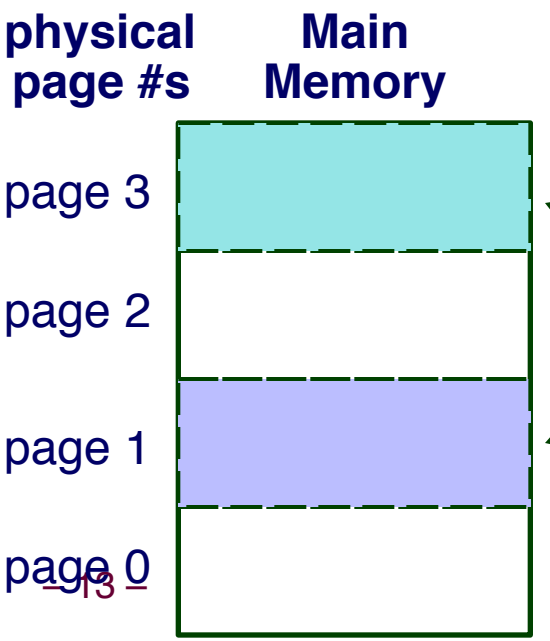
- records virtual page -> physical page mapping
- The page table differs for different processes

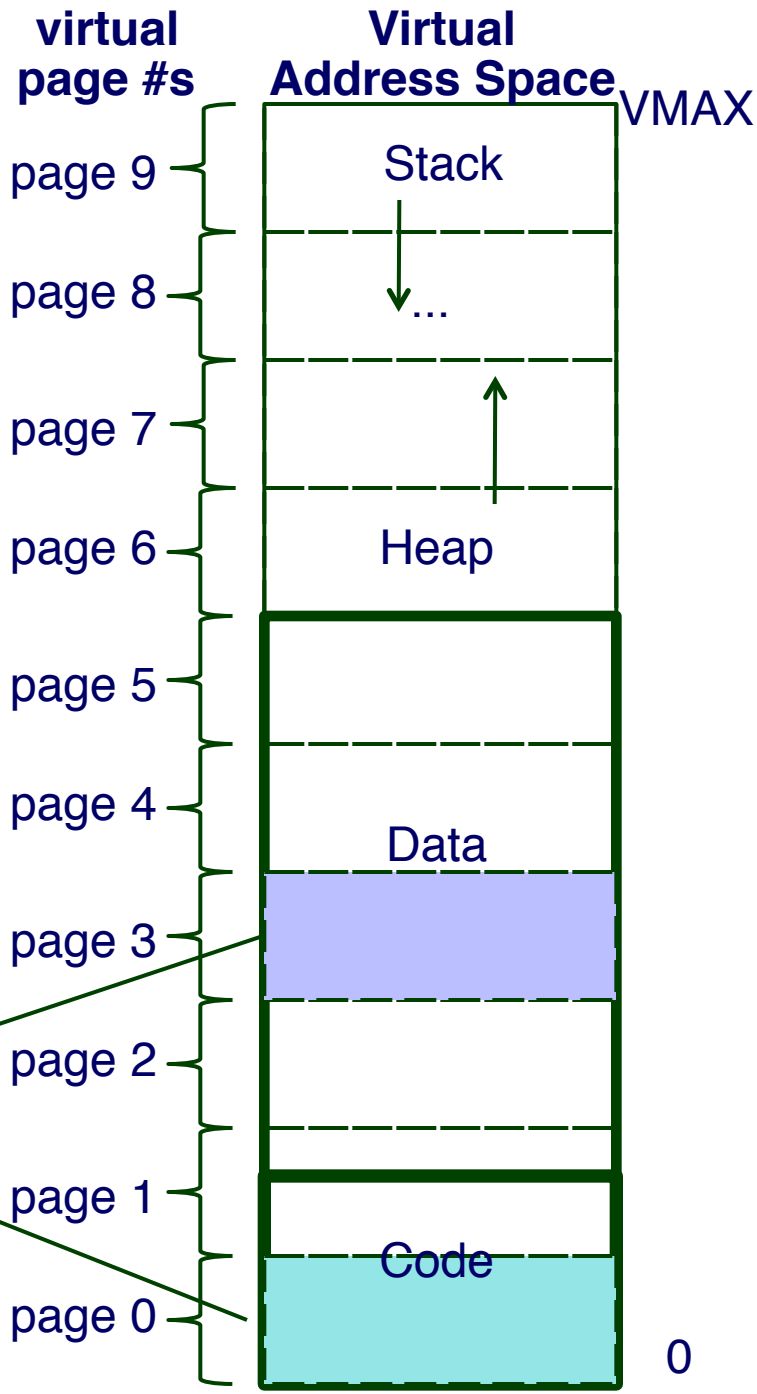


Page Tables

Virtual Page #	Physical Page #	va-lid ?
0	3	1
1		0
2		0
3	1	1
4		0
5		0
6		0
7		0
8		0
9		0

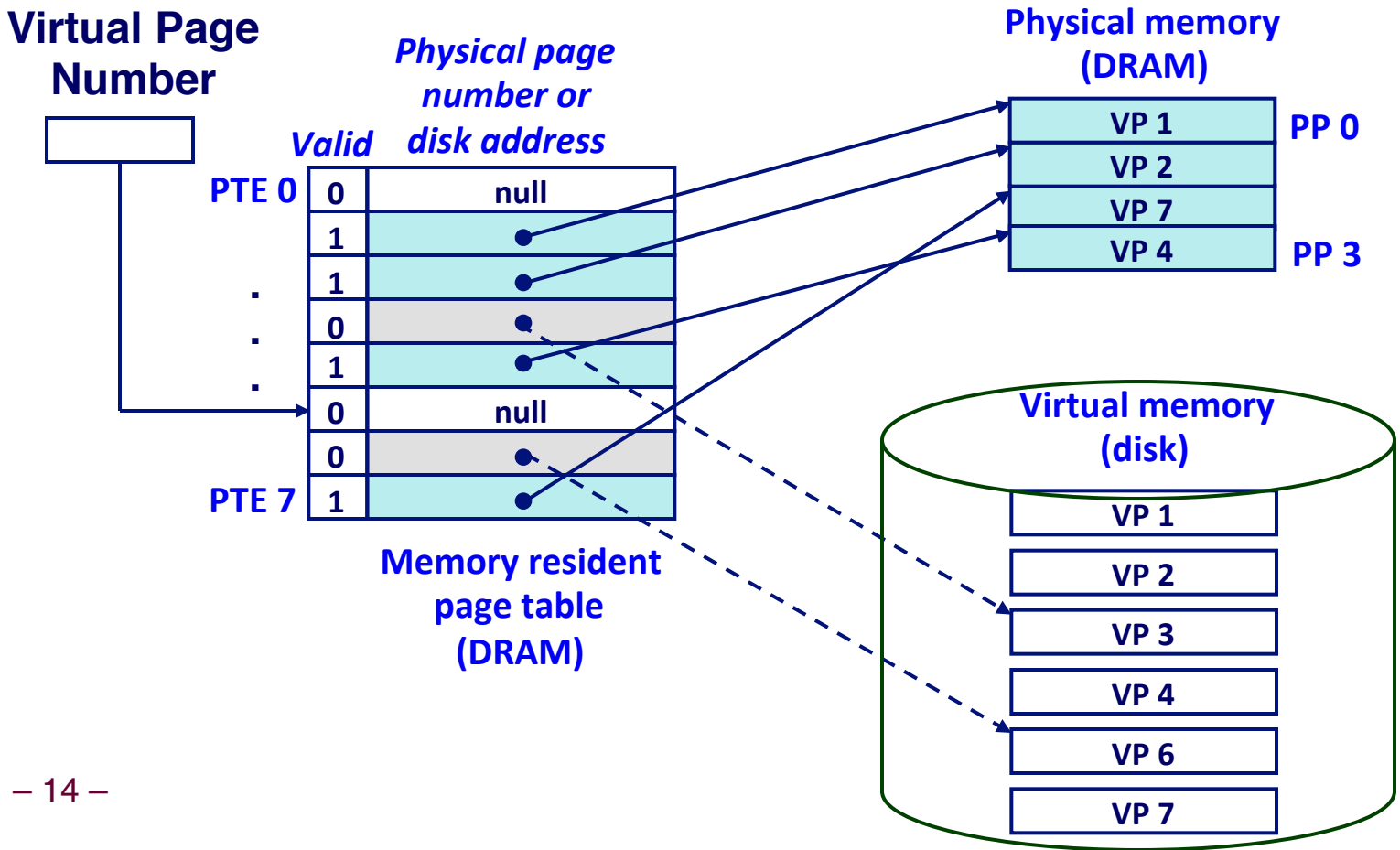
valid bit indicates if page is in memory (more on this later)





Address Translation: Page Tables

- A *page table* is an array of page table entries (PTEs) that maps virtual pages to physical pages. Here: 8 VPs
  - Per-process kernel data structure in DRAM

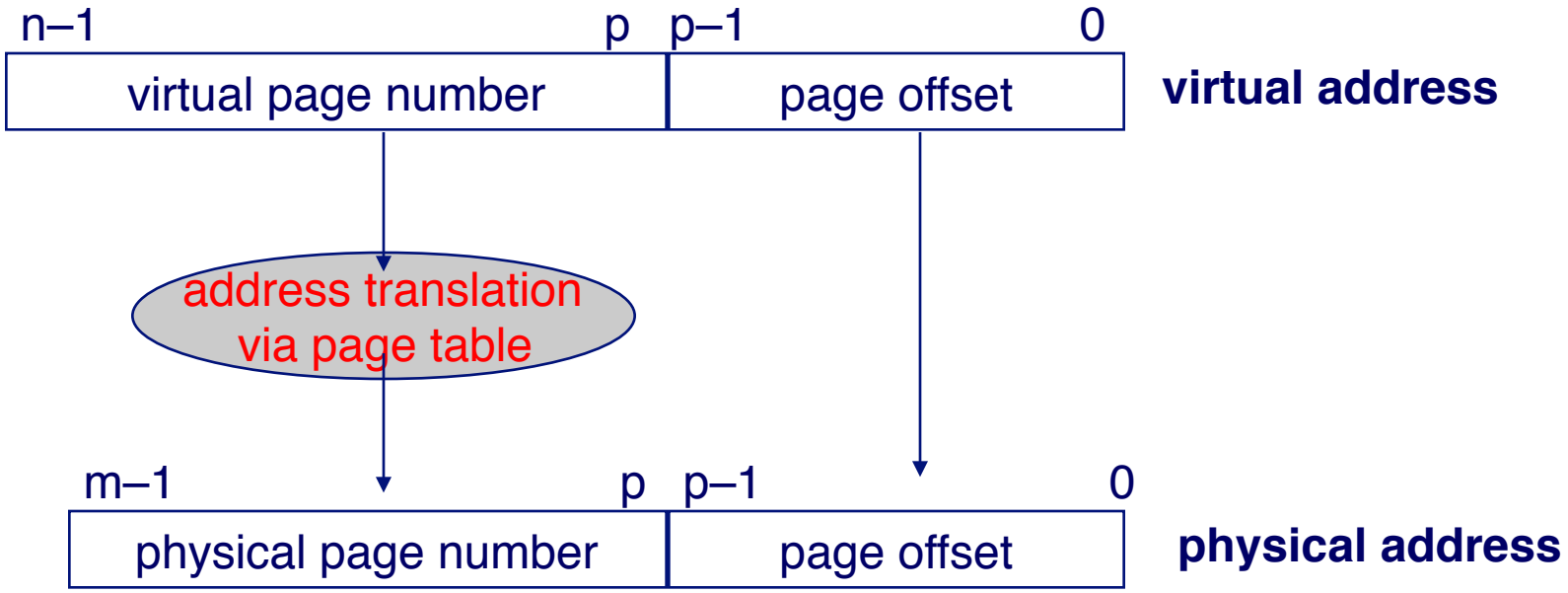


VM Address Translation

Parameters

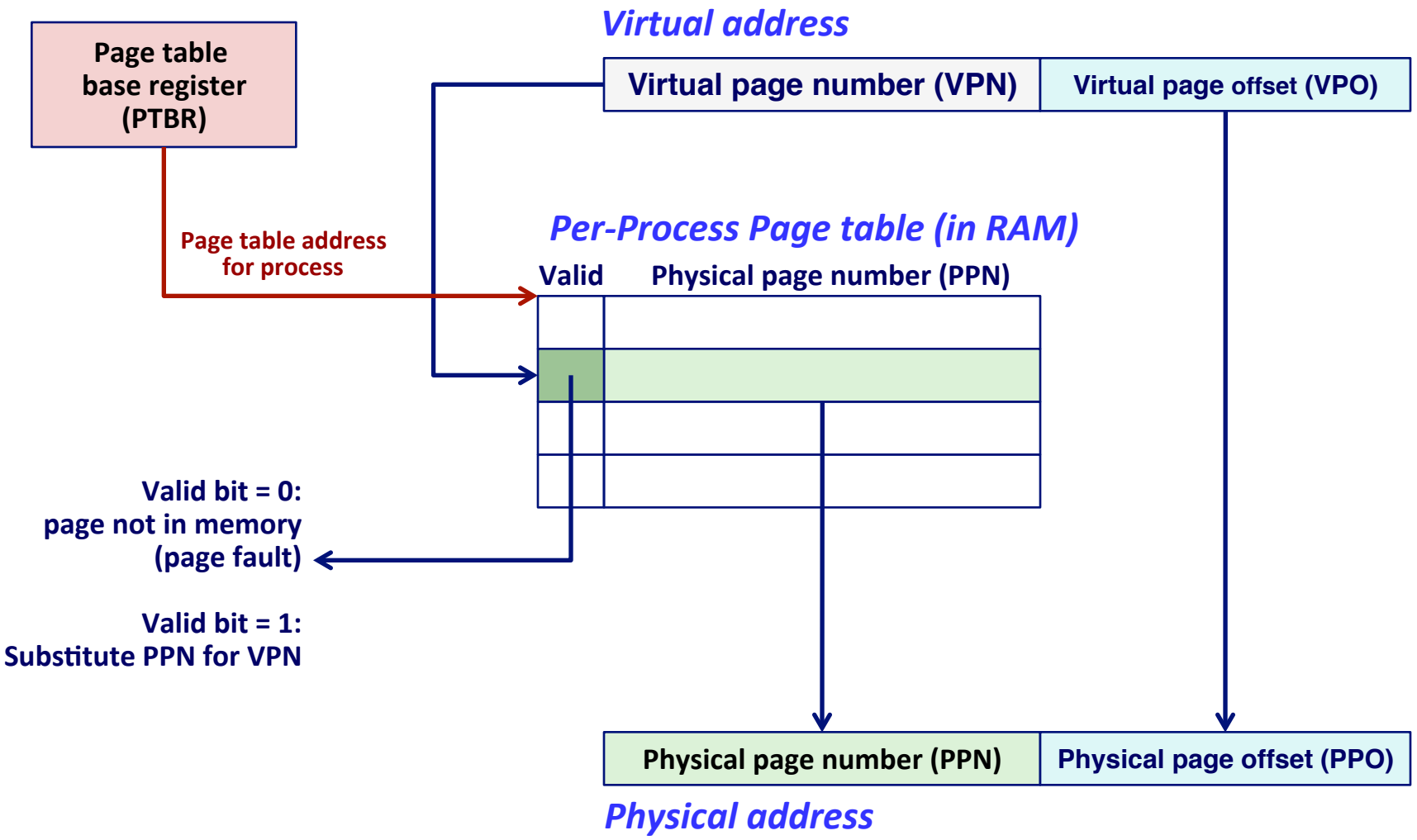
- P = 2^p = page size (bytes).
- N = 2^n = Virtual address limit
- M = 2^m = Physical address limit

Example:
page size = 4 KB = 2^{12},
for 32-bit addresses
then page # = 32-12
= 20 most significant bits



Page offset bits don't change as a result of translation

Address Translation With a Page Table

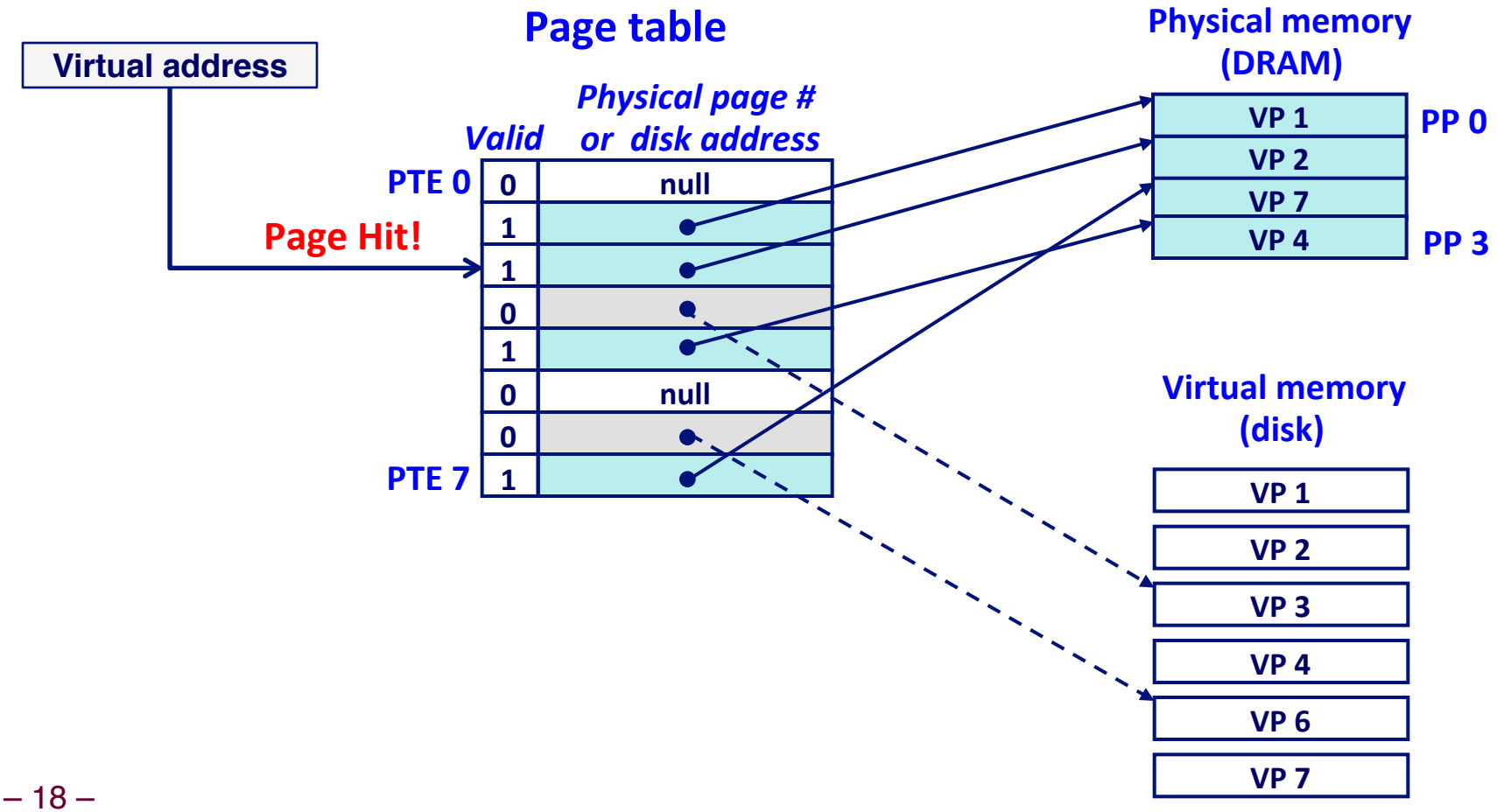


Keep only a subset of pages in memory = Demand Paging

- Not all pages of address space (code, data, etc.) have to be kept in memory
  - Exploit locality, e.g. loops, to keep only the pages that were recently accessed (the "*working set*") in memory
  - A small working set is best
  - Thrashing (performance meltdown) occurs when working set is too large for allocated memory
  - Main memory acts as a cache for disk
- Demand paging = load pages on demand as needed
  - Advantage: can now fit more processes into memory, since only a subset of pages are kept in memory at any one time
- Valid bit in page table keeps track of whether a page is in memory or on disk

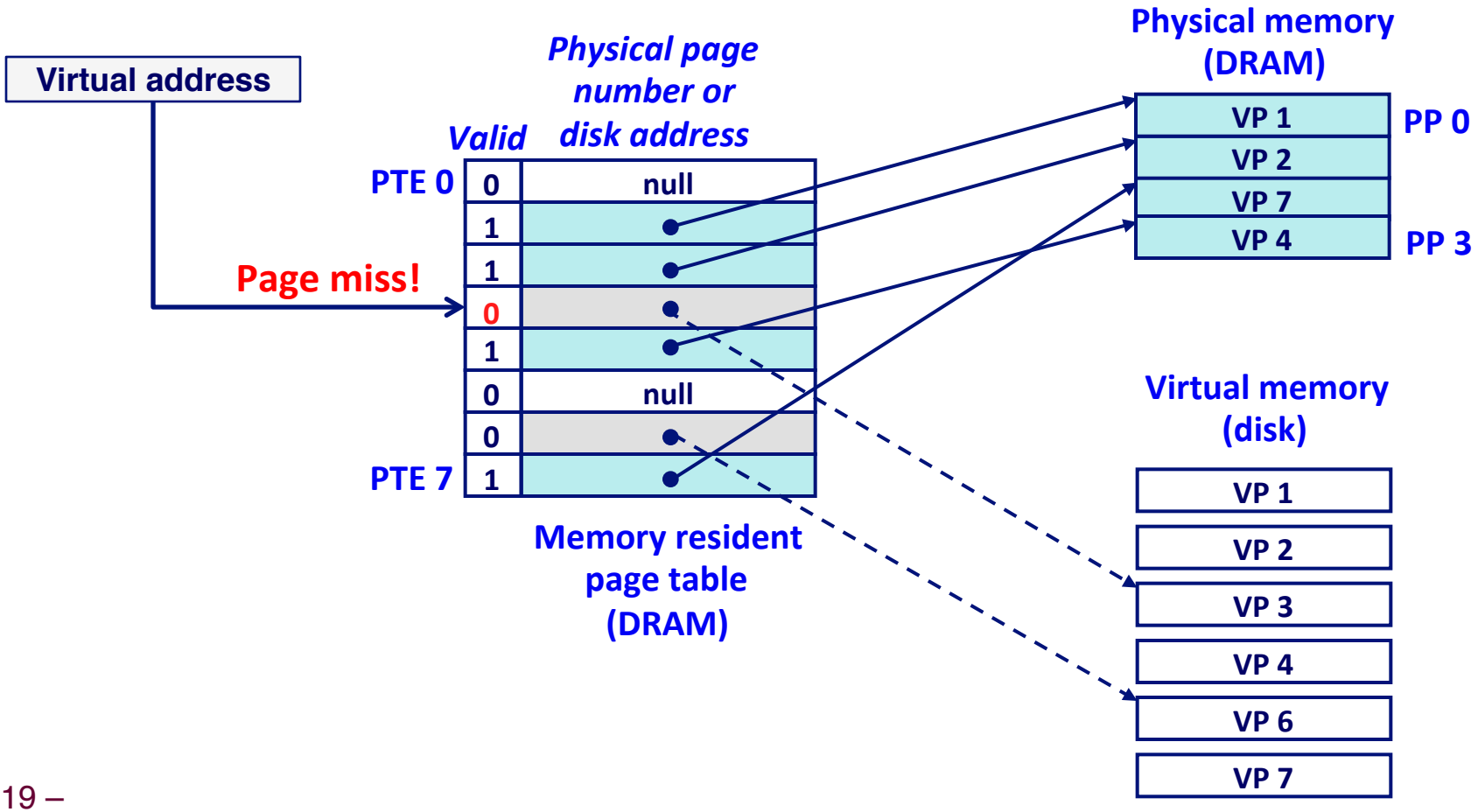
Page Hit

- ***Page hit:*** **reference to a virtual memory page that is in physical memory**



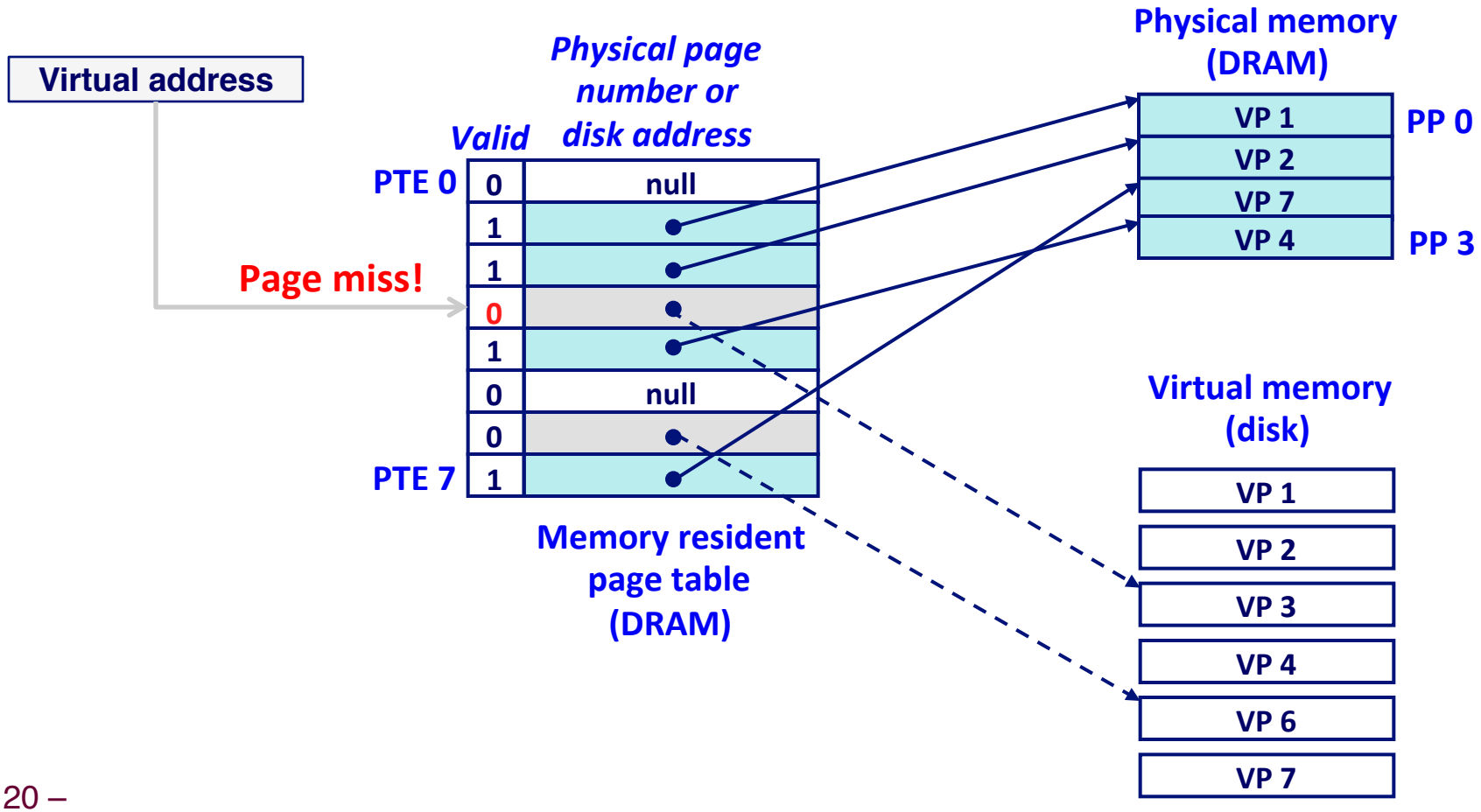
Page Miss

- ***Page miss:*** **reference to VM word that is not in physical memory**



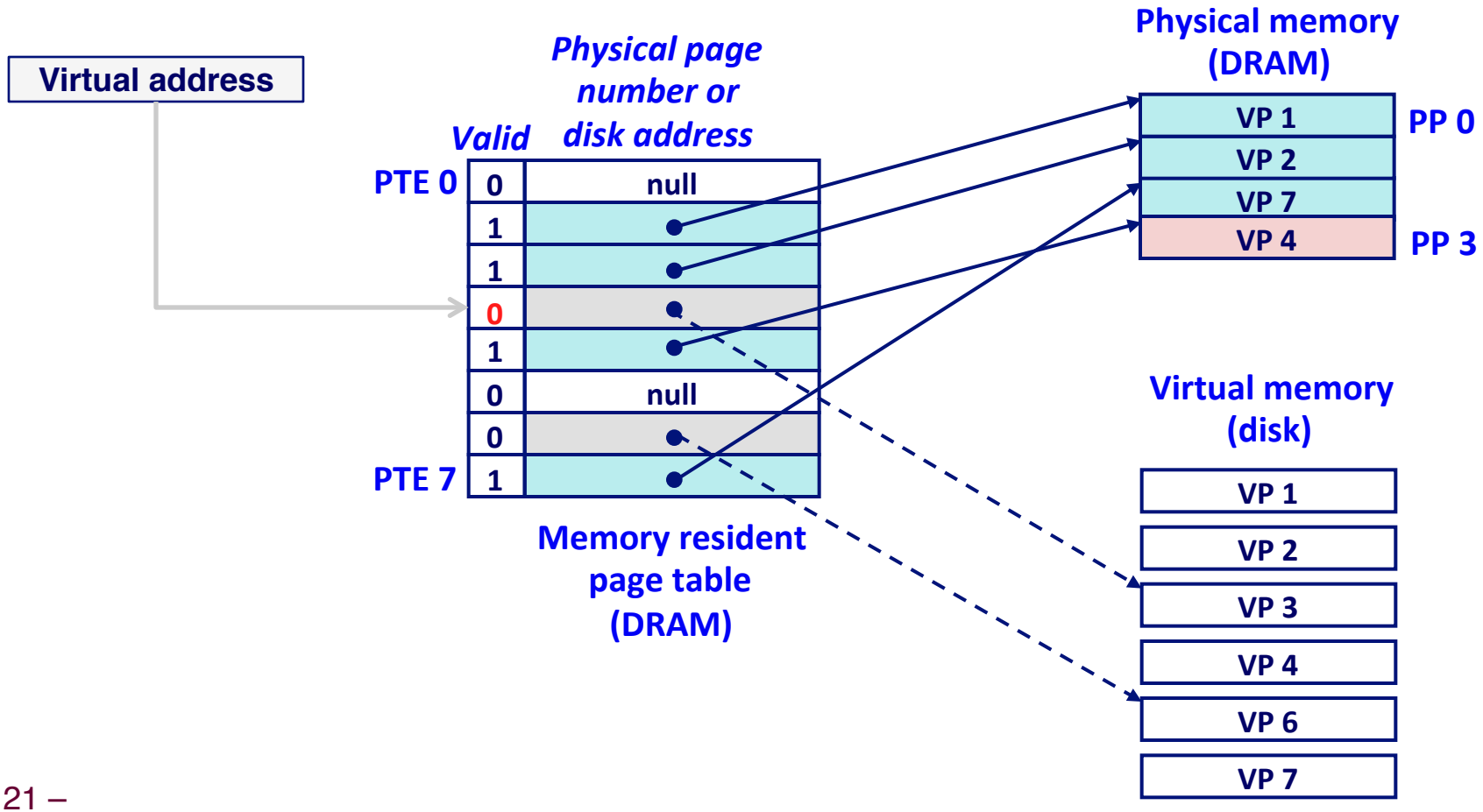
Handling Page Fault

- Page miss causes page fault (an exception)



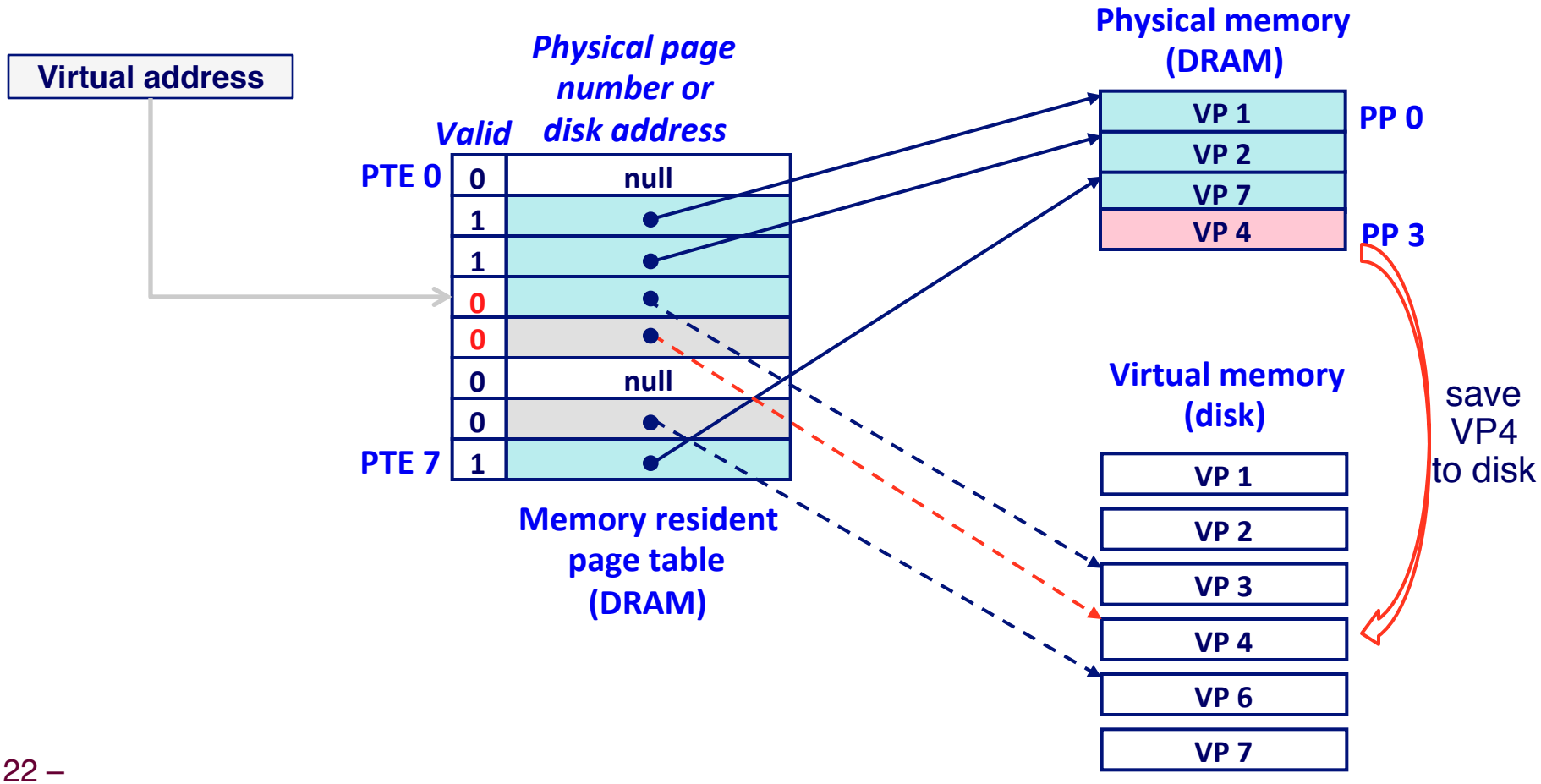
Handling Page Fault

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



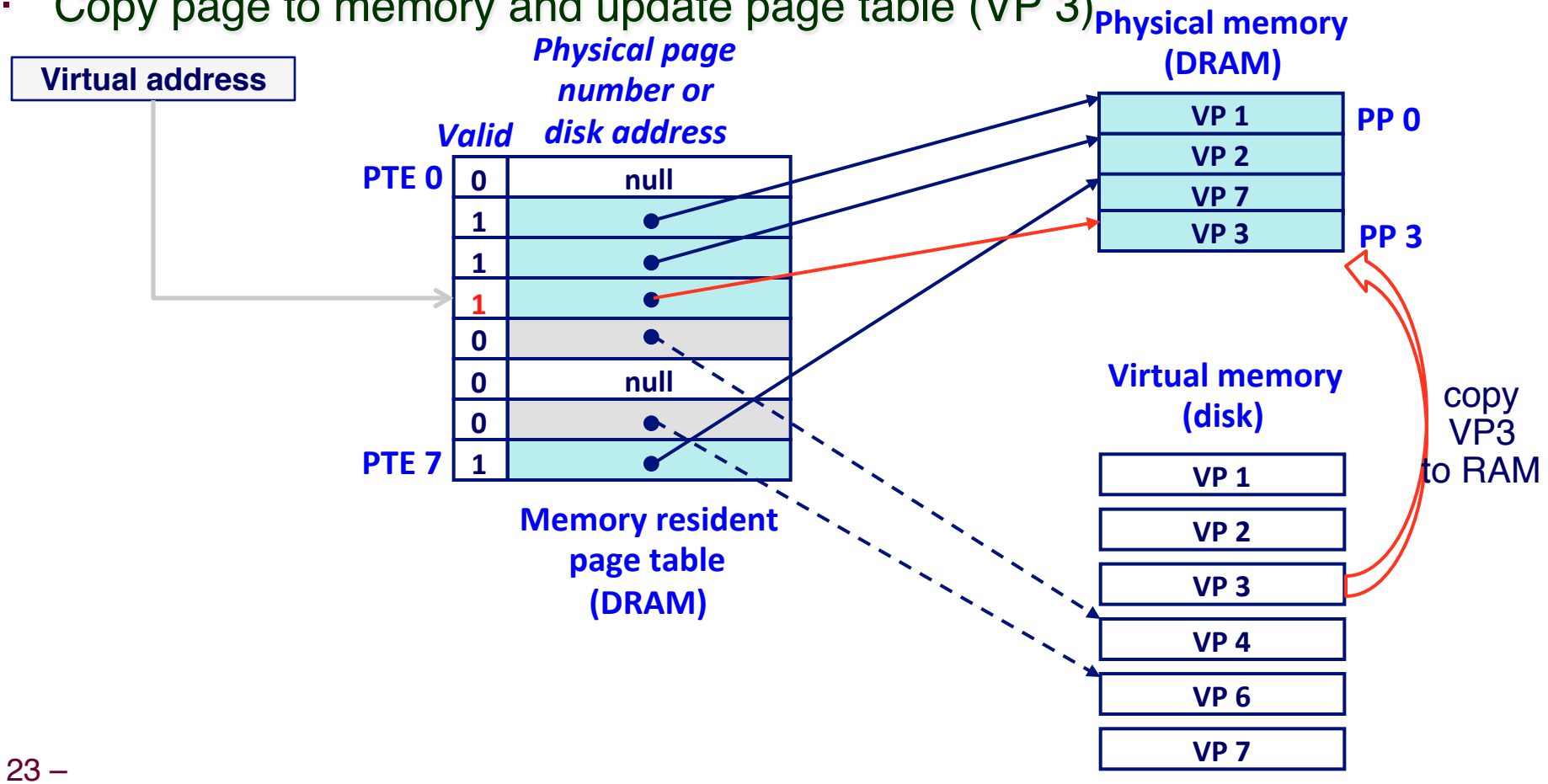
Handling Page Fault

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



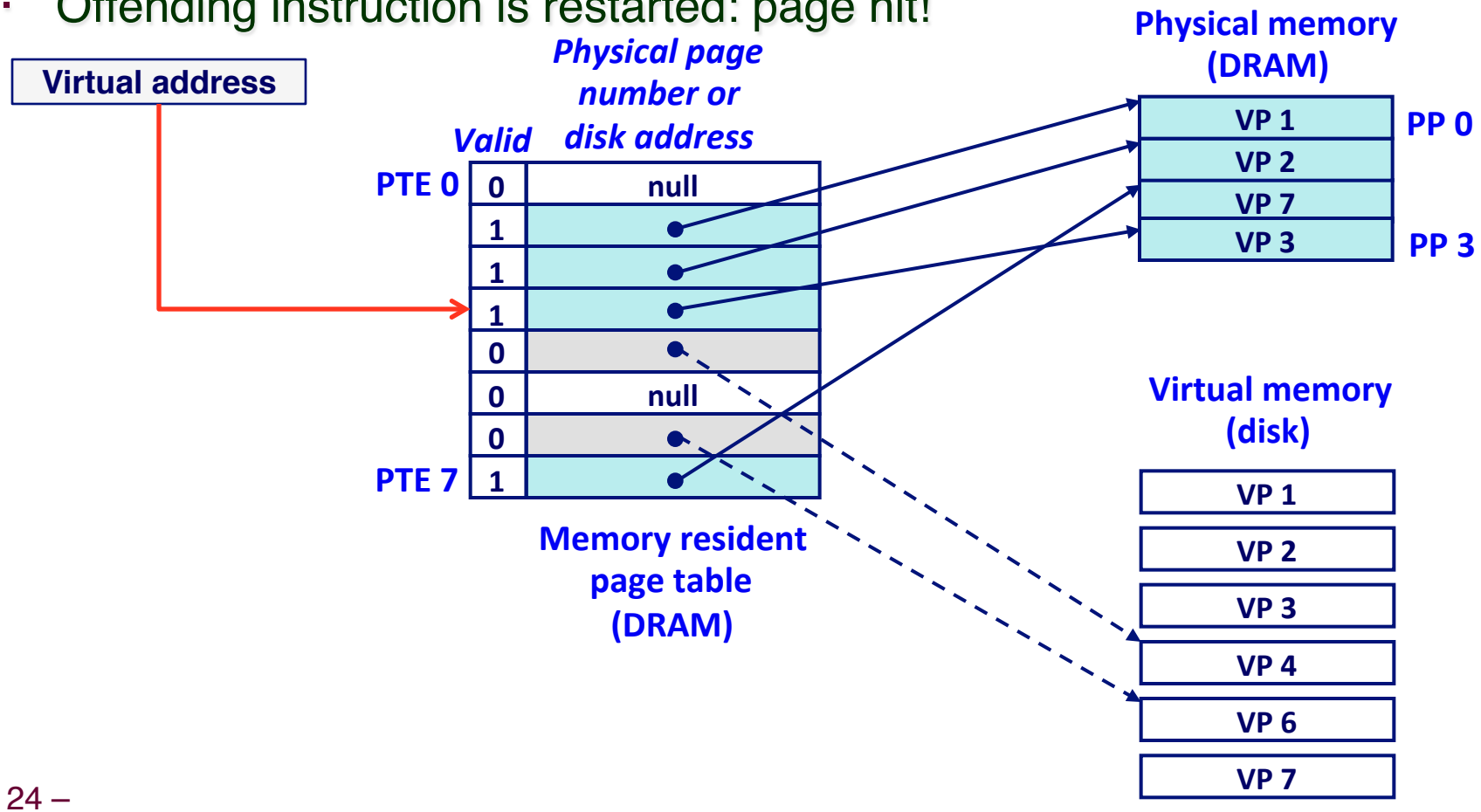
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- Copy page to memory and update page table (VP 3)



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!



Advantages of Page-based Virtual Memory (so far)

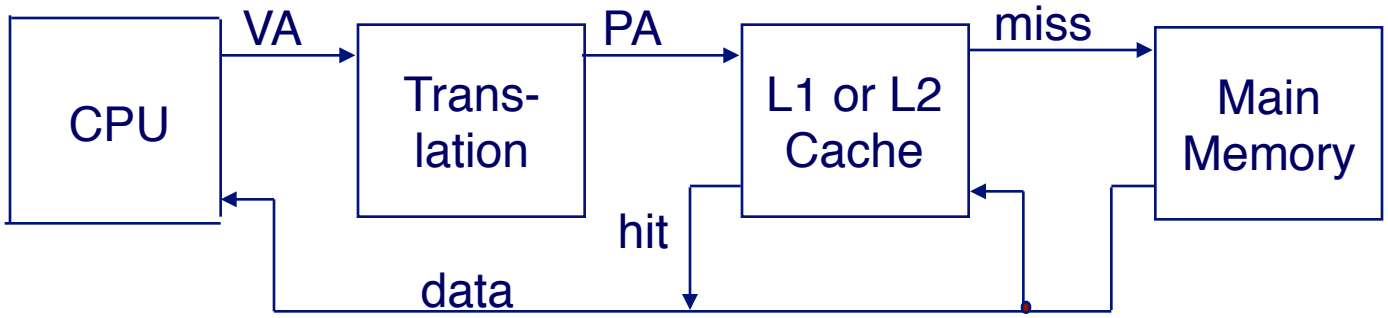
1. Simplifies memory management by compiler, program, and OS – all view the program as executing in an abstract [0..VMAX] address space
2. Can place processes anywhere in physical memory
3. Can fragment process into fixed-size pages that fit into unallocated memory fragments and thus use memory more efficiently
   - Processes no longer have to be contiguous.
   - Page table keeps track of where pages are placed physically in memory.
4. Demand paging keeps only a subset of recently accessed pages in memory – uses memory even more efficiently by allowing even more processes to fit into main memory

More Advantages of Page-based Virtual Memory

5. **Size of virtual address space VMAX is no longer limited to the amount of physical memory allocated to the process, but can equal or exceed the physical memory!**
   - i.e. VMAX can be >= PMAX
   - Heap, stack, code and data segments can all be much larger now and not be constrained by physical memory – the set of addressable memory addresses is much larger

6. **Page tables help protect address spaces**
   - A process X cannot easily write into the pages of another process Y's address space
     - OS typically assigns non-overlapping physical pages to fill in the entries of each process's page table
     - any virtual address X used will be translated by X's page table to a physical page of memory that is typically not shared with any other process like Y

Integrating Virtual Memory and Cache



Most Caches "Physically Addressed"

- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at same time
- Allows multiple processes to share pages
- Cache doesn't need to be concerned with protection issues
  - Access rights checked as part of address translation

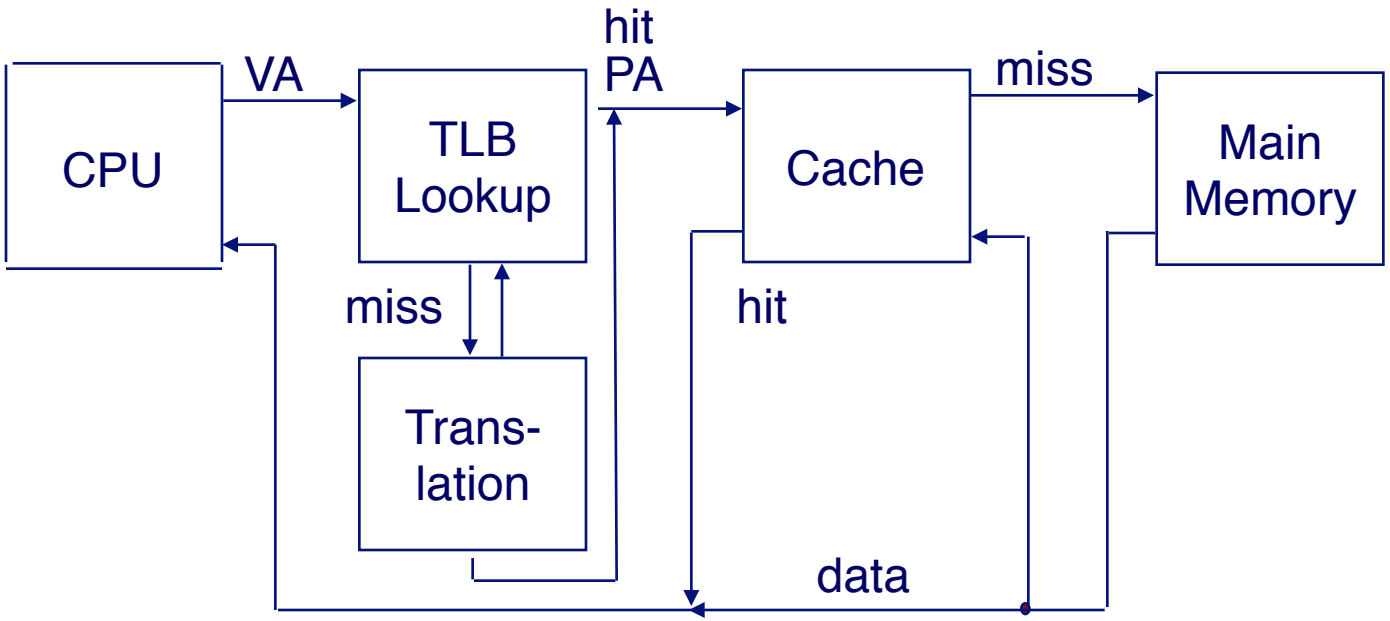
Perform Address Translation Before Cache Lookup

- But this could involve a memory access itself (of the PTE) – Big Slowdown!
- Therefore, cache the page table entries too!

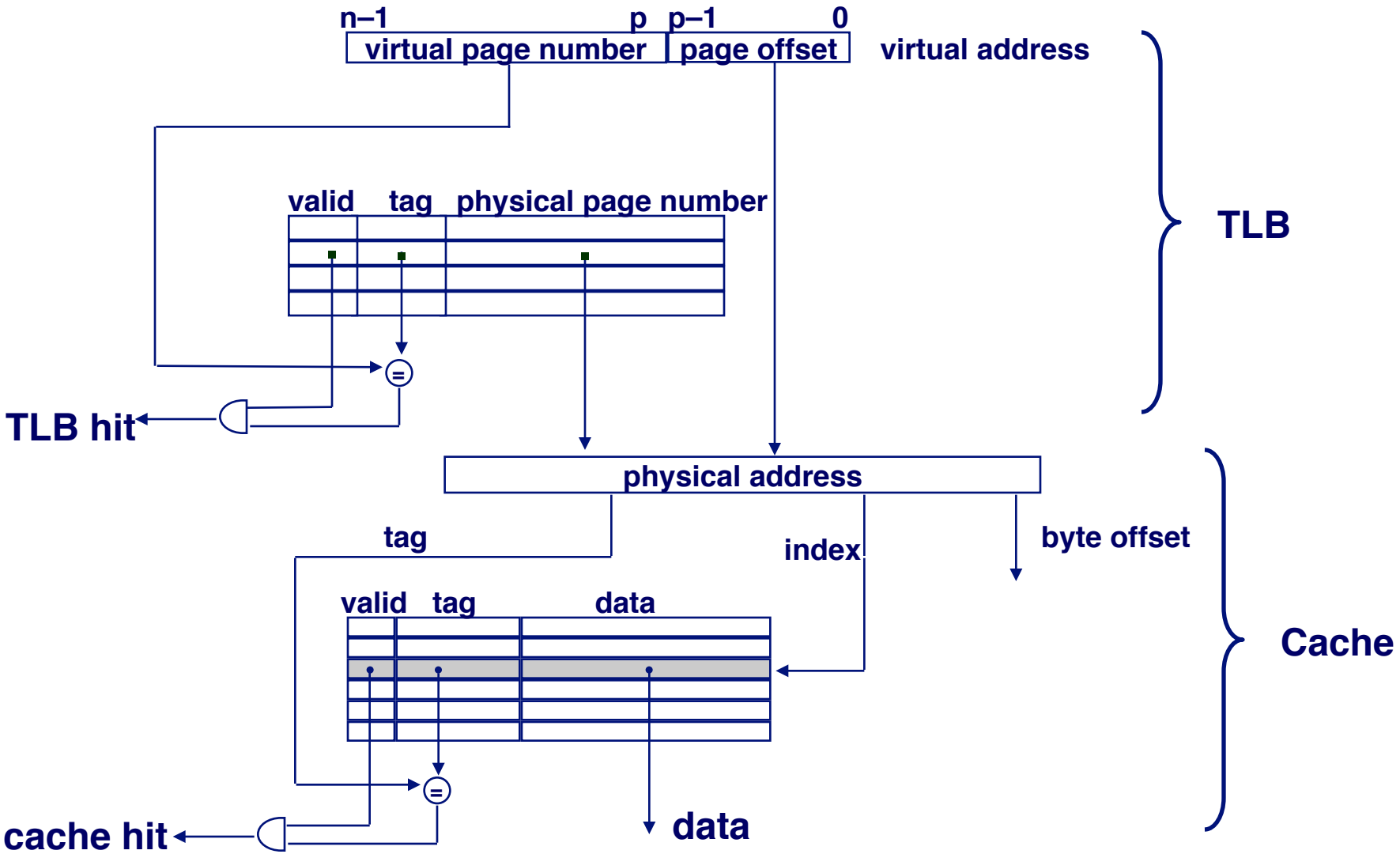
Speeding up Translation with a TLB

"Translation Lookaside Buffer" (TLB)

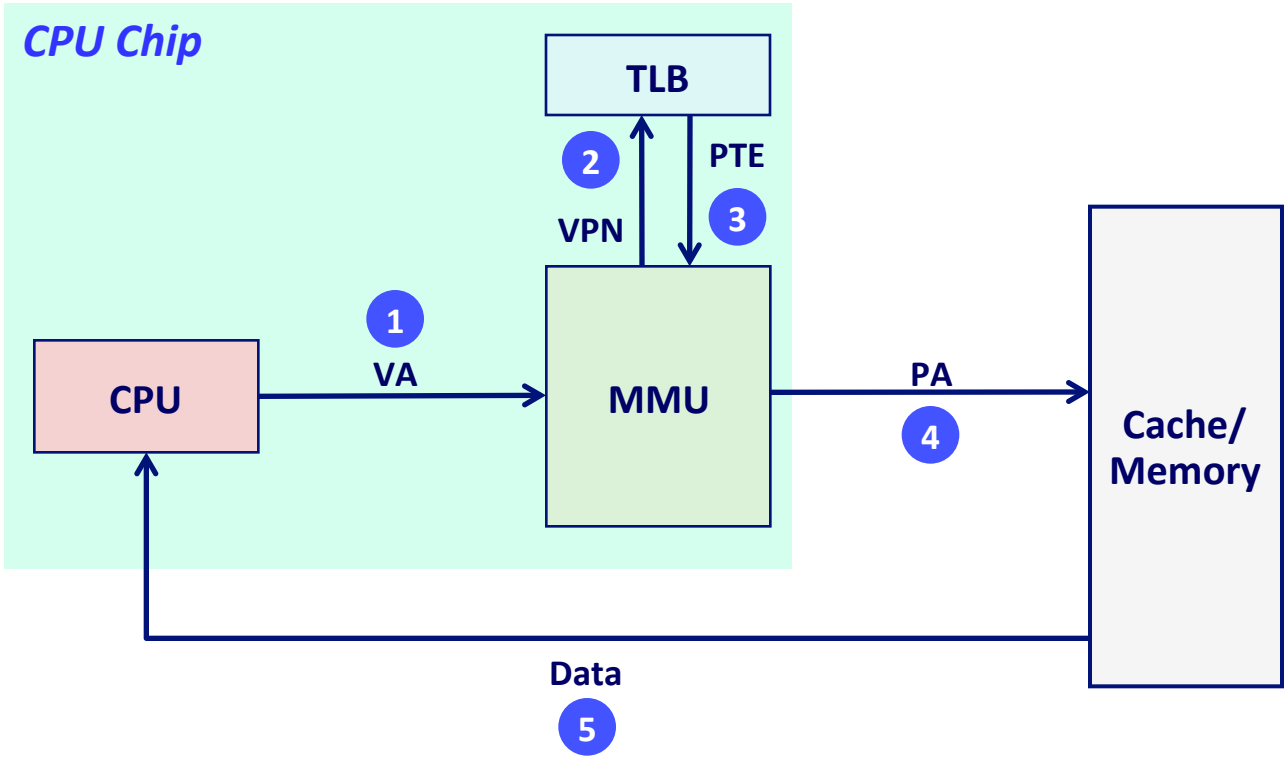
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages



Address Translation with a TLB

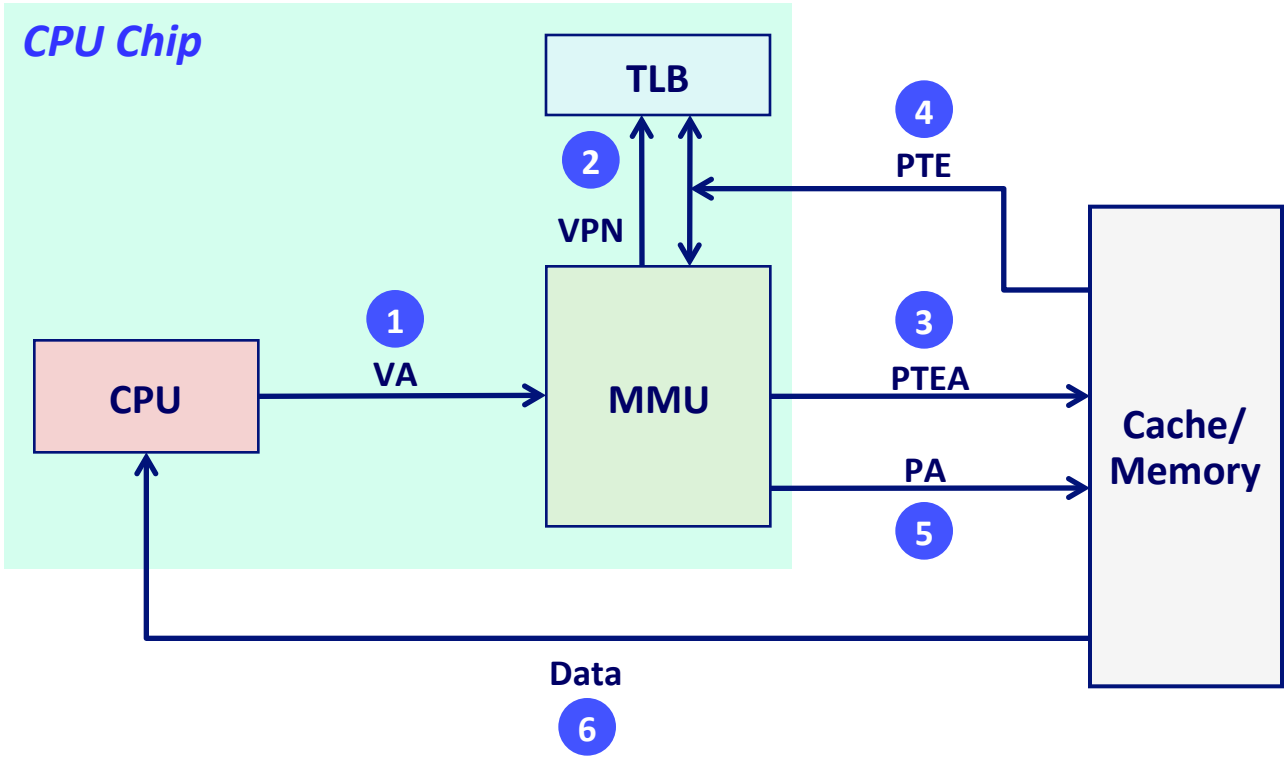


TLB Hit



A TLB hit eliminates a memory access

TLB Miss

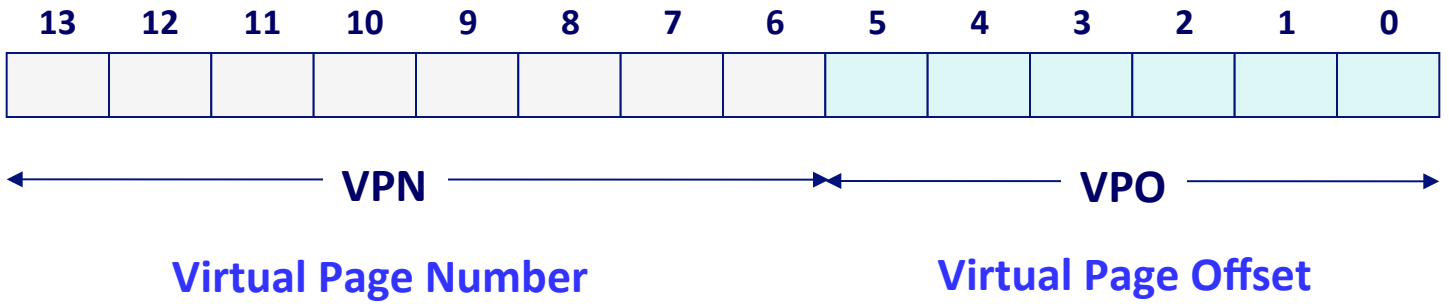


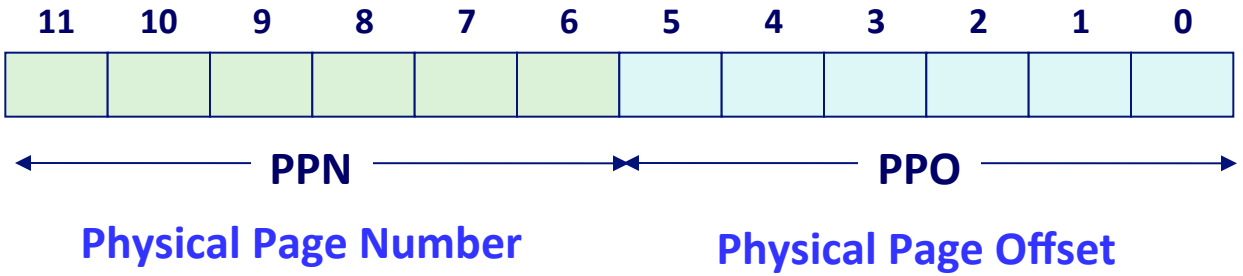
A TLB miss incurs an add'l memory access (the PTE)

Fortunately, TLB misses are rare

Simple Memory System Example

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





Simple Memory System Page Table

Only show first 16 entries (out of 256)

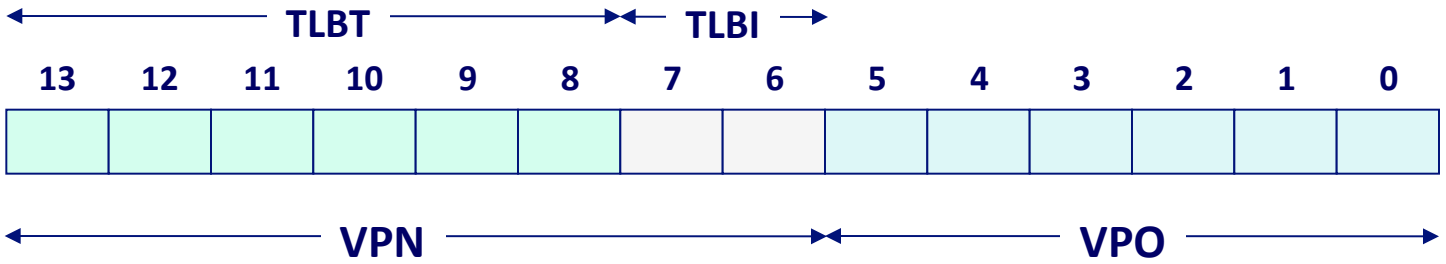
VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

For a later example, let's assume VPN page 0x2E is invalid, i.e. valid bit =0, so it is not in memory

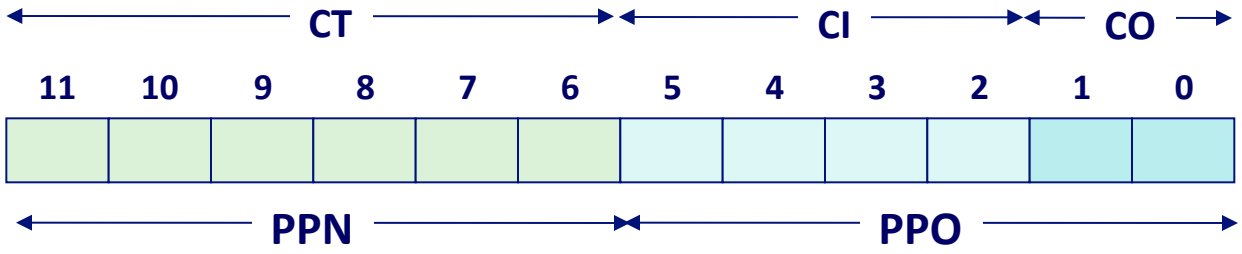
Simple Memory System TLB

- 16 entries
- Virtually addressed
- 4-way associative



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

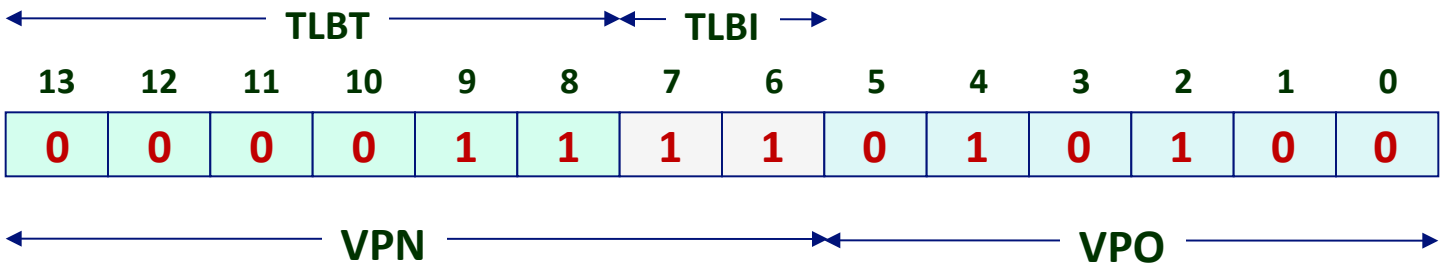


- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

Idx	Tag	Valid	B0	B1	B2	B3
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
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	–	–	–	–

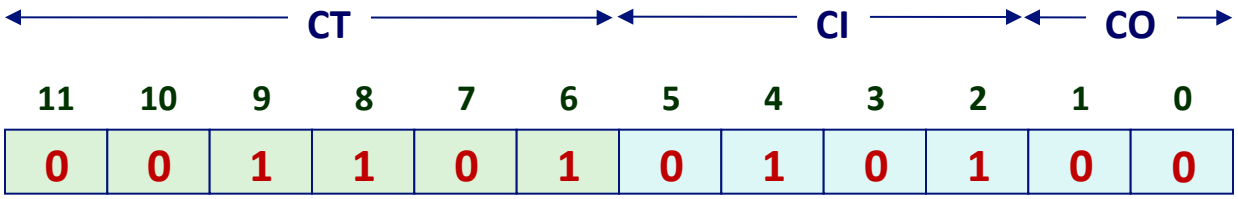
Address Translation Example #1

Virtual Address: 0x03D4



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

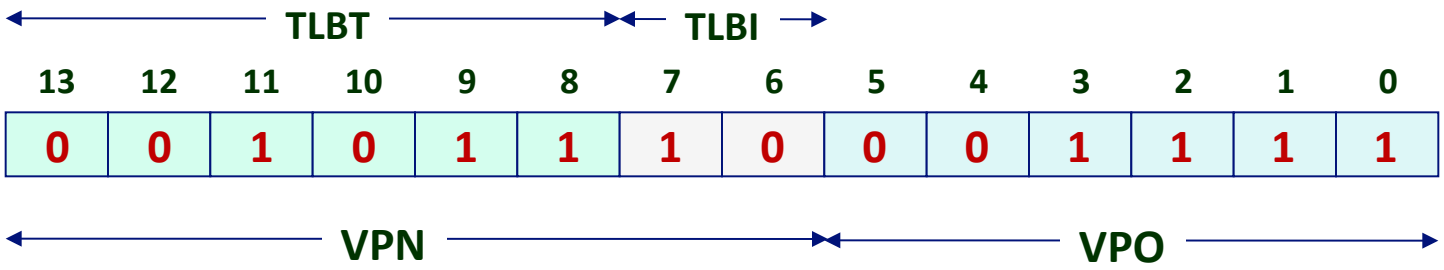
Physical Address



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

Address Translation Example #2

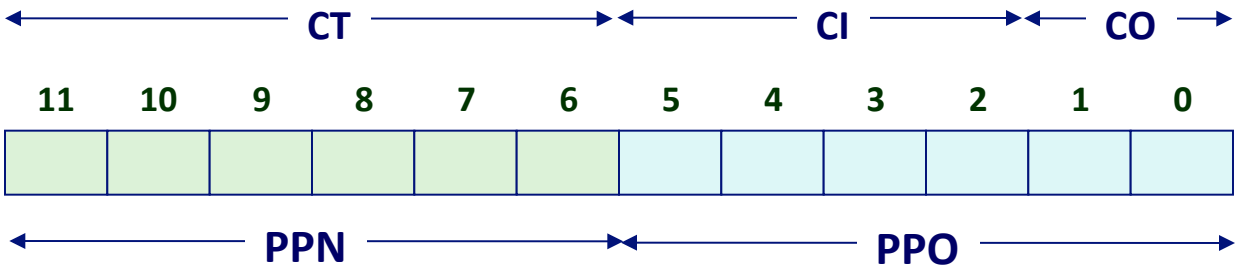
Virtual Address: `0x0B8F`



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

here, the page is neither in TLB nor memory, so have to go out to disk – MMU waits, i.e. valid bits of both TLB and page table =0

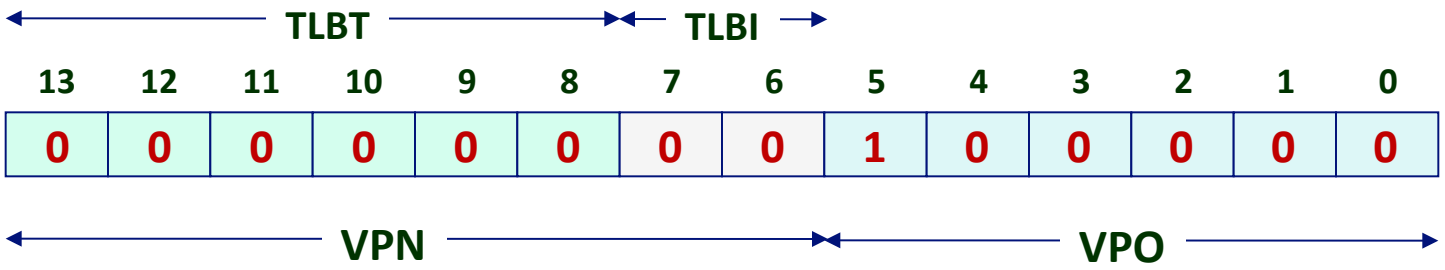
Physical Address



CO ___ CI___ CT ____ Hit? __ Byte: ____

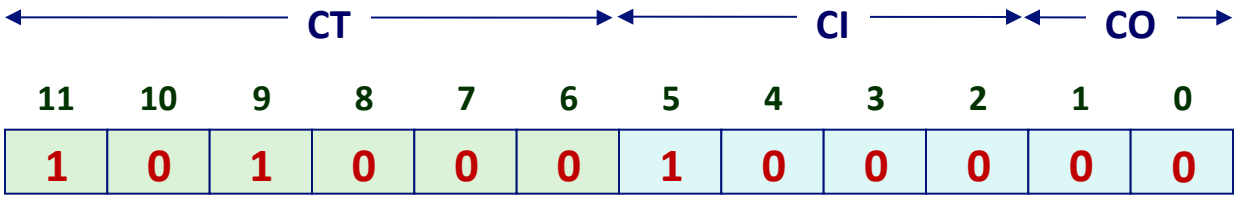
Address Translation Example #3

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

Cache Hit Combinations

Is Virtual->Physical page mapping entry cached in Page Table?

- **Hit on TLB**
- **Miss on TLB, Hit on Main Memory/RAM**
- **Miss on TLB, Miss on Main Memory/RAM = Page Fault**

Is data/instruction associated with a Physical address cached in L1/L2 Cache?

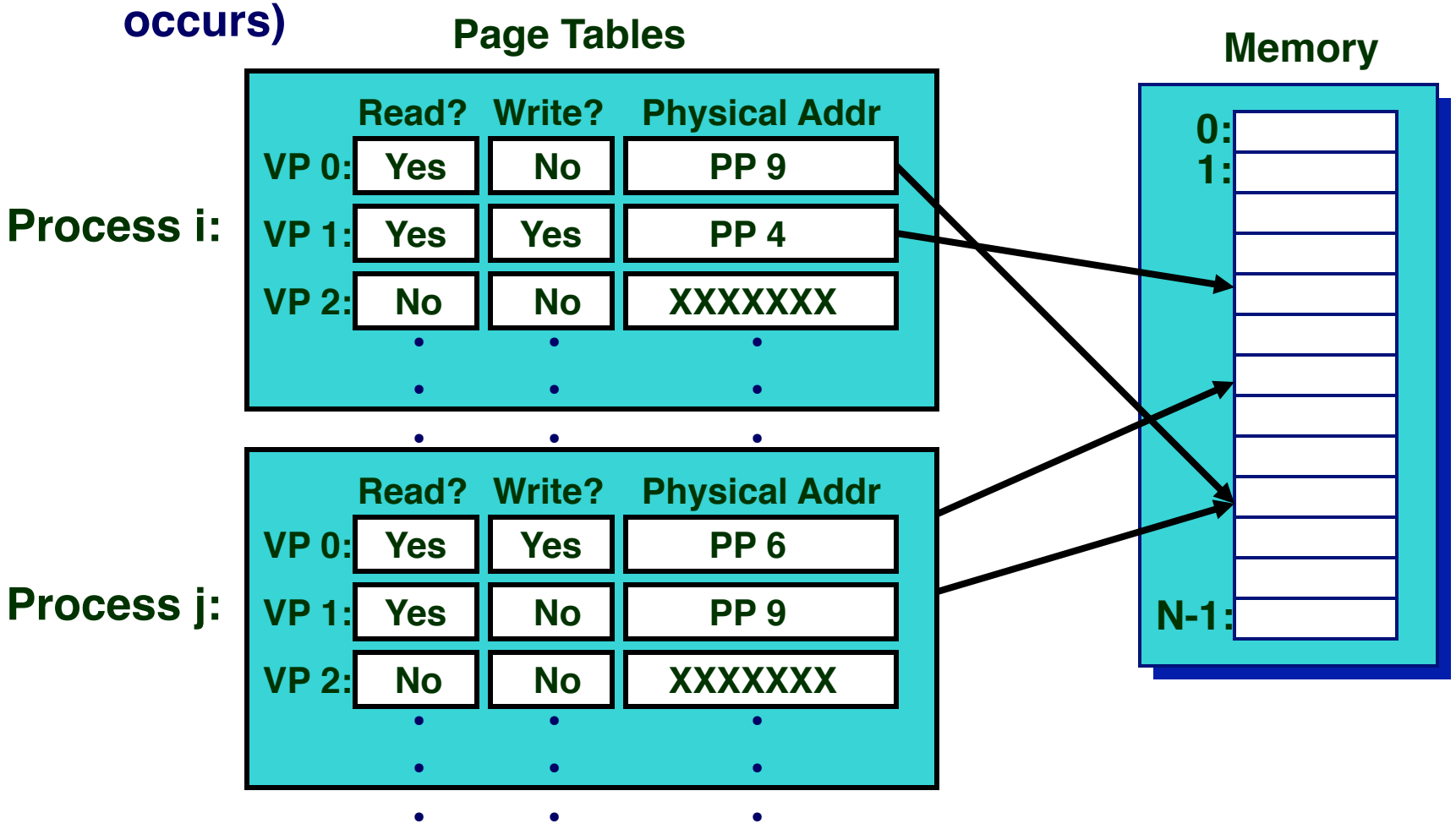
- **Hit on L1**
- **Miss on L1, Hit on L2**
- **Miss on L1 and L2, go to Main Memory/RAM**

Supplementary Slides

Memory Protection

Page table entry contains access rights information

- hardware enforces this protection (trap into OS if violation occurs)



Motivations for Virtual Memory

For *multiple processes*, virtual memory brings benefits:

Use Physical DRAM as a Cache for the Disk

- Only "active" code and data is actually cached in memory for each process
  - Allocate more memory to process as needed.
- Physical memory can support many more processes

Simplify Memory Management

- Each process has its own virtual address space
- OS keeps track of the virtual->physical memory mapping and informs MMU whenever a new process is context-switched in to execute on the CPU

Provide Protection

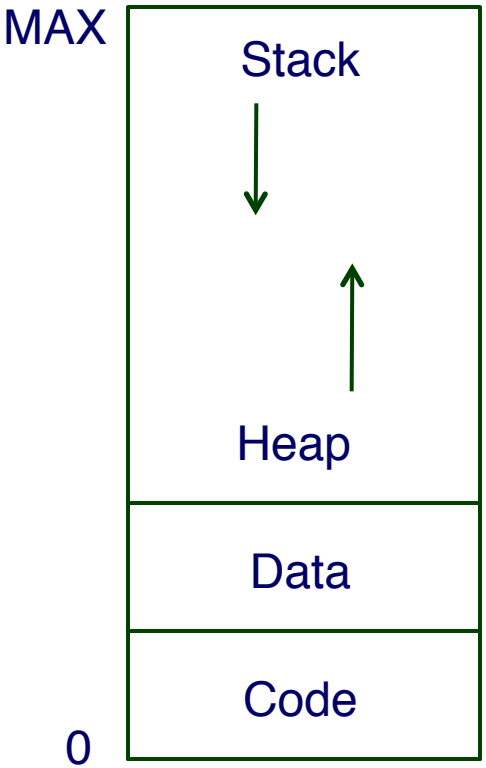
- One process can't interfere with another, because each one operates in its own address space & is mapped to different physical memory – can't overwrite same physical memory
  - different sections of address spaces have different permissions.

Motivations for Virtual Memory (2)

Provides the Abstraction that a Process is executing in its own Address Space of memory from address 0 to address MAX

For a Single Process:

- **Set of virtual addresses = Address Space of a process, which can *exceed* physical memory size**
  - **Only "active" code and data is actually in memory**
  - **frees a process from having to worry about physical limitations of main memory**
    - **for example, could allocate very large arrays and data structures, only a portion of which are kept in memory**
  - **Still limited by maximum size of virtual memory: for a 32-bit system, maximum addressable virtual memory is 2^32 = 4GB**



Motivation #1: DRAM a "Cache" for Disk

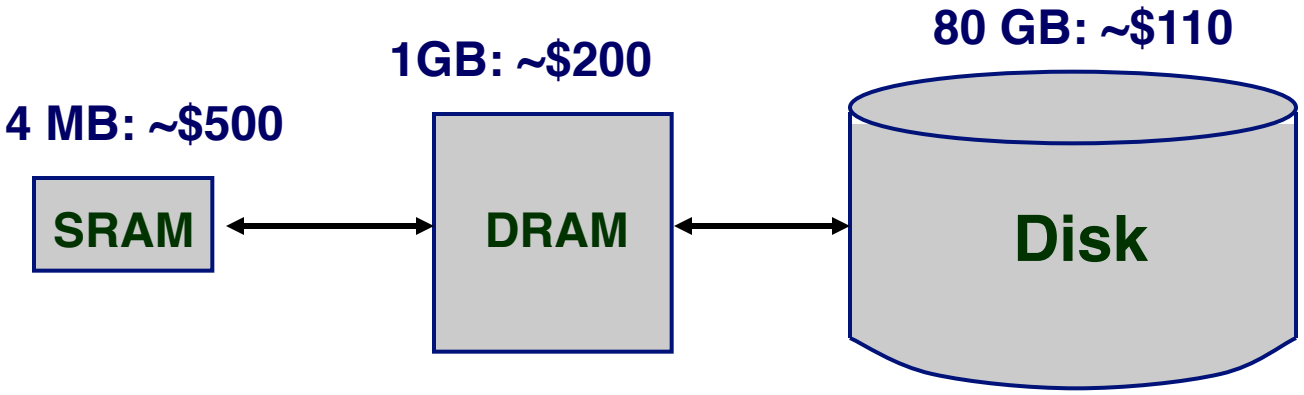
Full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~300X cheaper than DRAM storage

- 80 GB of DRAM: ~ $33,000
- 80 GB of disk: ~ $110

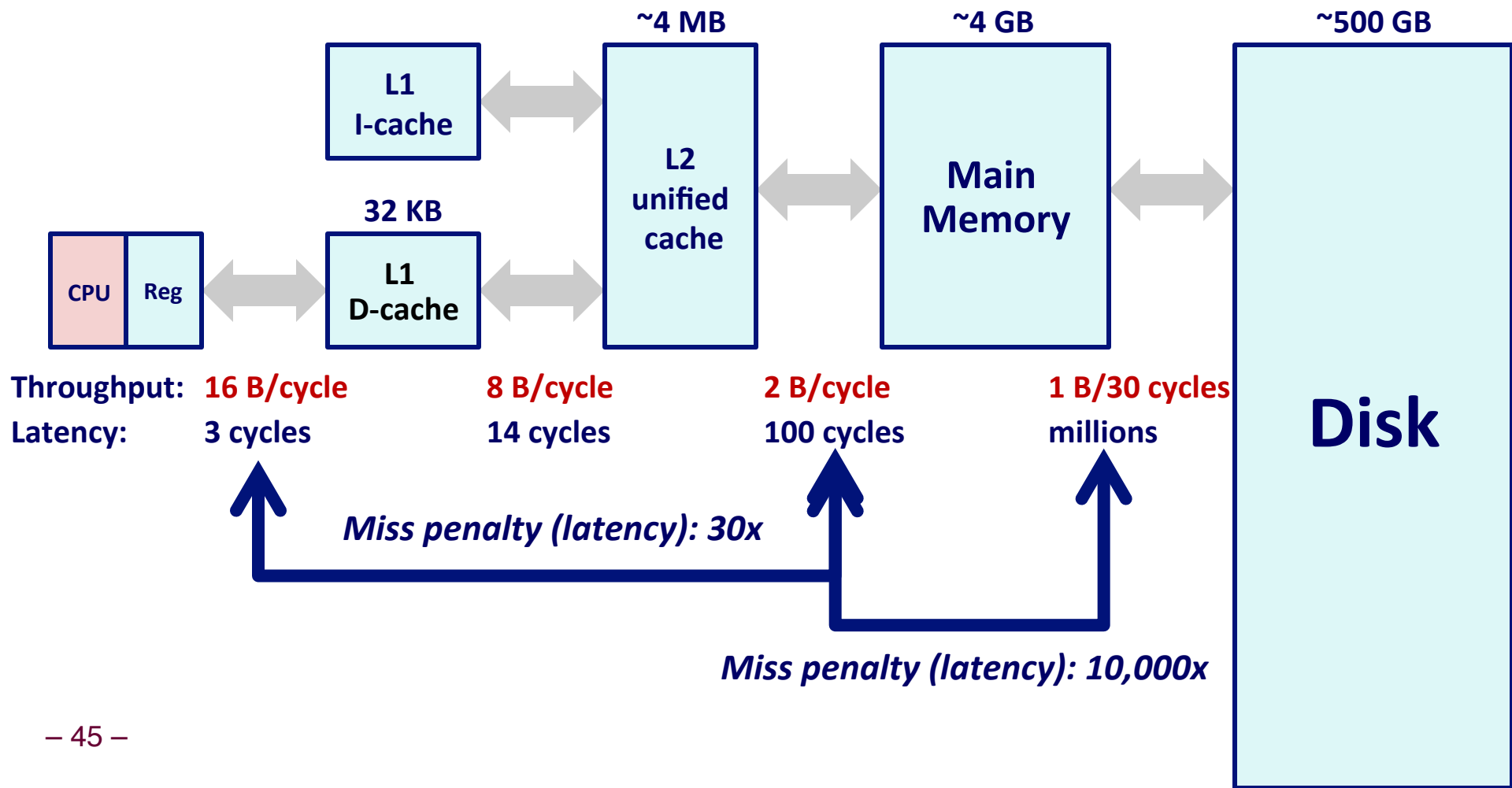
To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Memory Hierarchy: Core 2 Duo

Not drawn to scale

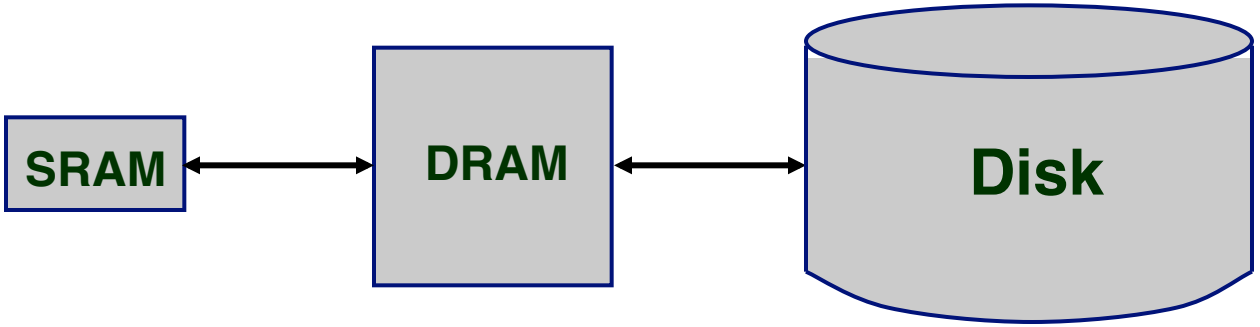
L1/L2 cache: 64 B blocks



DRAM vs. SRAM as a "Cache"

DRAM vs. disk is more extreme than SRAM vs. DRAM

- Access latencies:
  - DRAM ~10X slower than SRAM
  - Disk ~100,000X slower than DRAM
- Importance of exploiting spatial locality:
  - First byte is ~100,000X slower than successive bytes on disk
    - » vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- Bottom line:
  - Design decisions made for DRAM caches driven by enormous cost of misses



Impact of Properties on Design

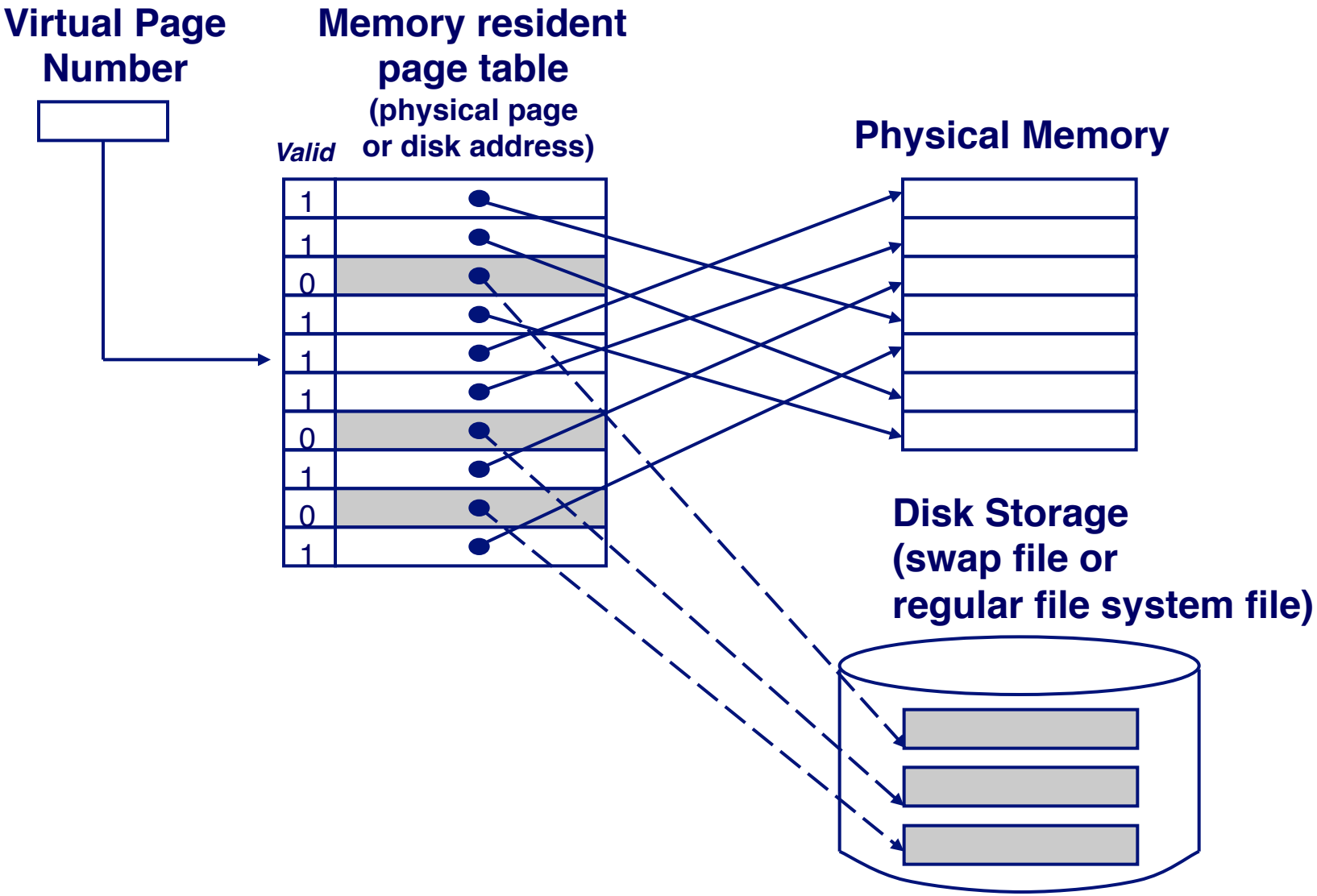
If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- **Line size?**
  - **Large, since disk better at transferring large blocks**
- **Associativity?**
  - **High, to mimimize miss rate**
- **Write through or write back?**
  - **Write back, since can't afford to perform small writes to disk**

What would the impact of these choices be on:

- **miss rate**
  - **Extremely low. << 1%**
- **hit time**
  - **Must match cache/DRAM performance**
- **miss latency**
  - **Very high. ~20ms**
- **tag storage overhead**
  - **Low, relative to block size**

Page Tables


Virtual Page Number
Memory resident page table
(physical page or disk address)
Valid
1
1
0
1
1
1
0
1
0
1
Physical Memory
Disk Storage
(swap file or
regular file system file)

VM Address Translation

Virtual Address Space

- V = {0, 1, …, N–1}

Physical Address Space

- P = {0, 1, …, M–1}
- M < N

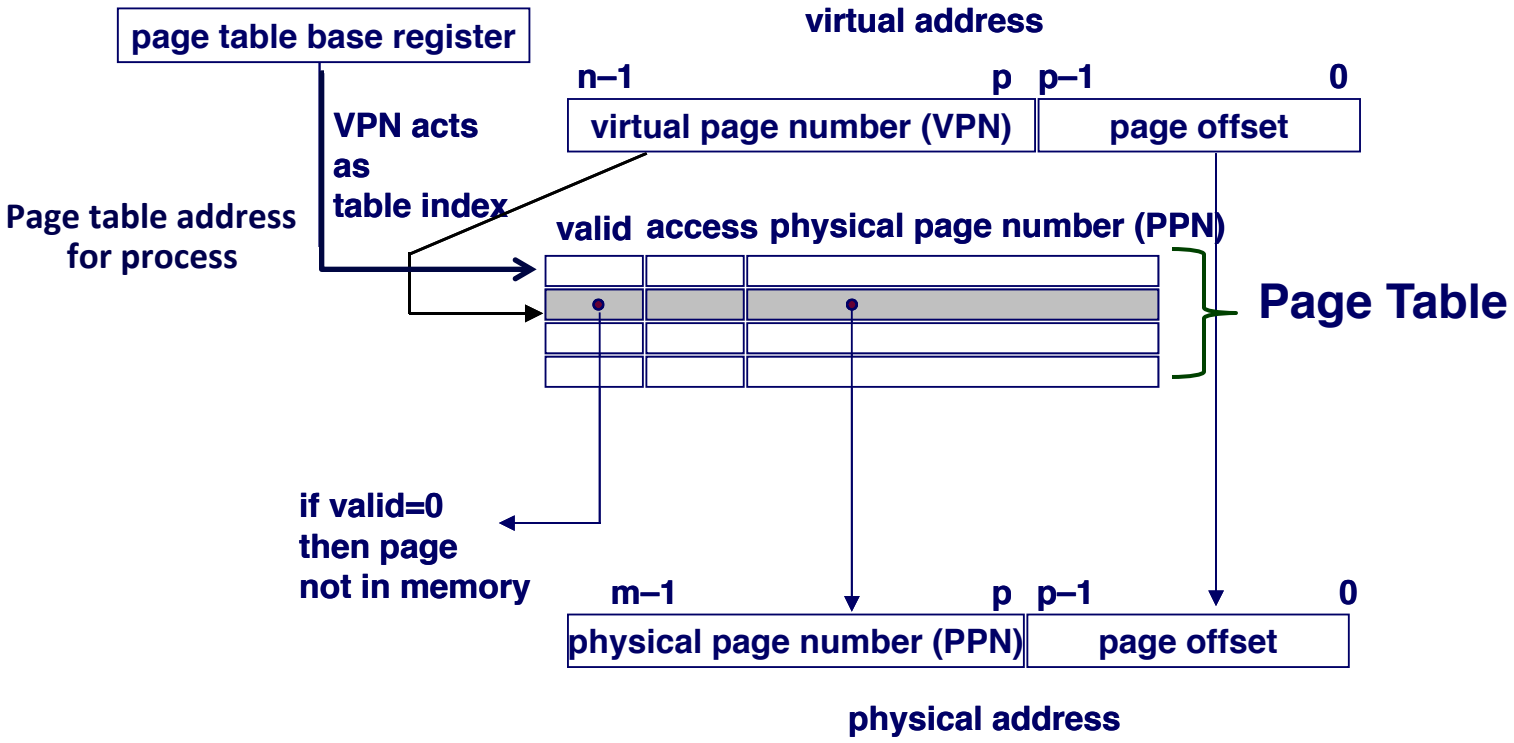
Address Translation

- MAP: V → P U {∅}
- For virtual address a:
  - MAP(a) = a' if data at virtual address a at physical address a' in P
  - MAP(a) = ∅ if data at virtual address a not in physical memory
    - » Either invalid or stored on disk

Page Table Operation

Translation

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)



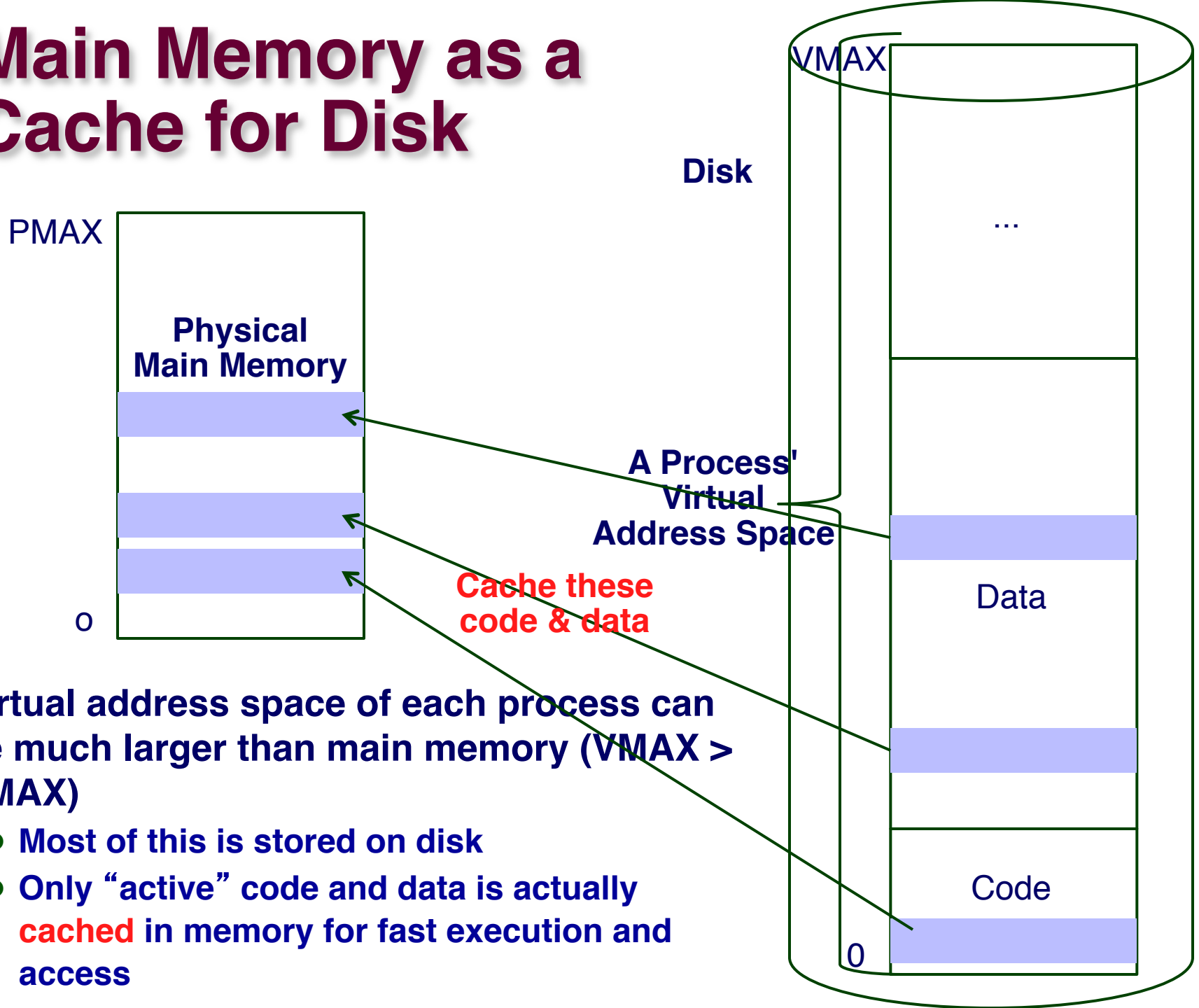
Demand Paging uses Main Memory as a Cache for Disk

- The most recently accessed pages are kept in memory for fast access, compared to the disk, where the rest of the pages are stored
- Similar to caches, which keep the most recently accessed instructions and data in L1, L2, and L3 caches for fast access, compared to memory, where the rest of the instructions & data are stored

Why does Caching Pages work? Locality

- Virtual memory works because of locality
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < amount of physical main memory allocated to a process)
  - Good performance for one process after compulsory misses
- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

Main Memory as a Cache for Disk

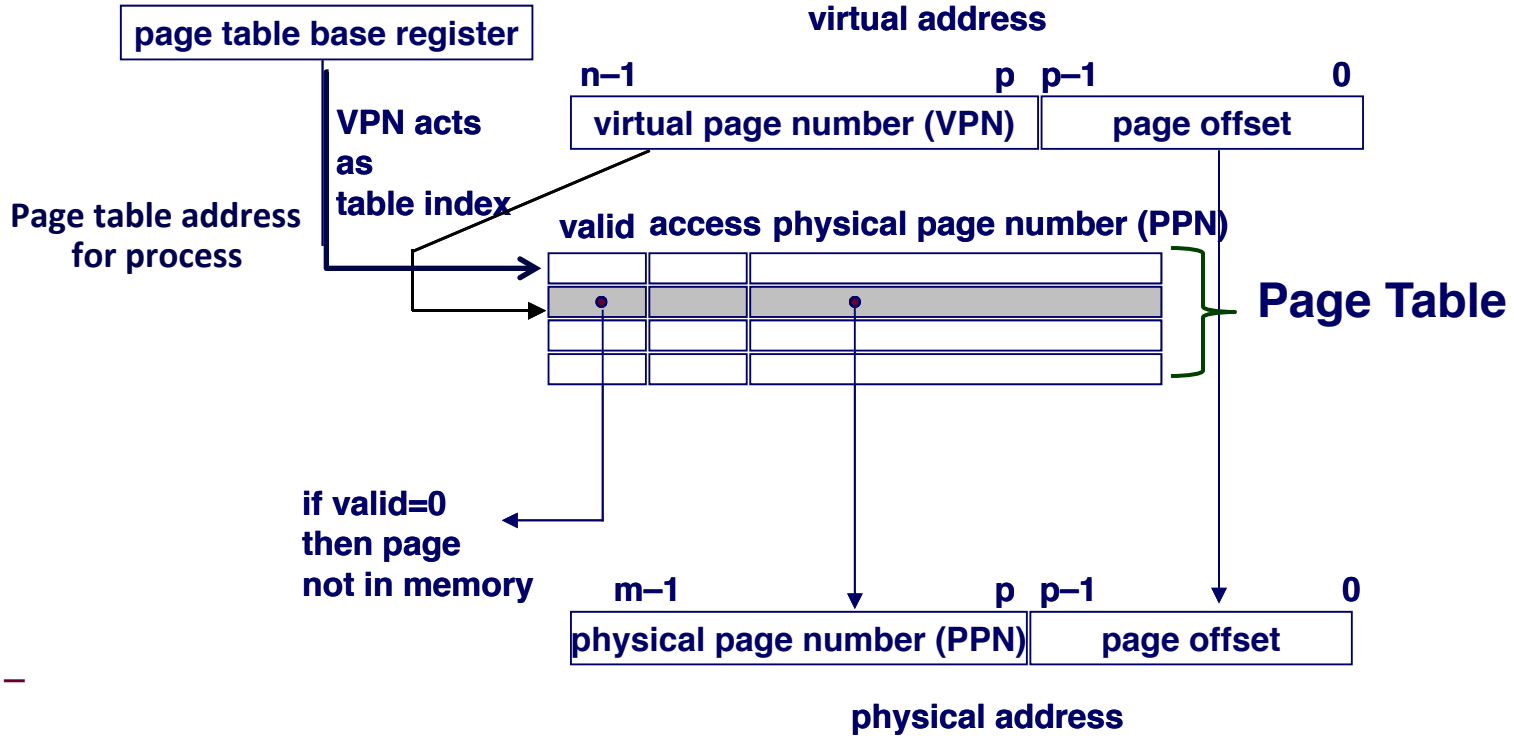


- Virtual address space of each process can be much larger than main memory (VMAX > PMAX)
  - Most of this is stored on disk
  - Only "active" code and data is actually cached in memory for fast execution and access

Page Table Operation with Demand Paging

Computing Physical Address

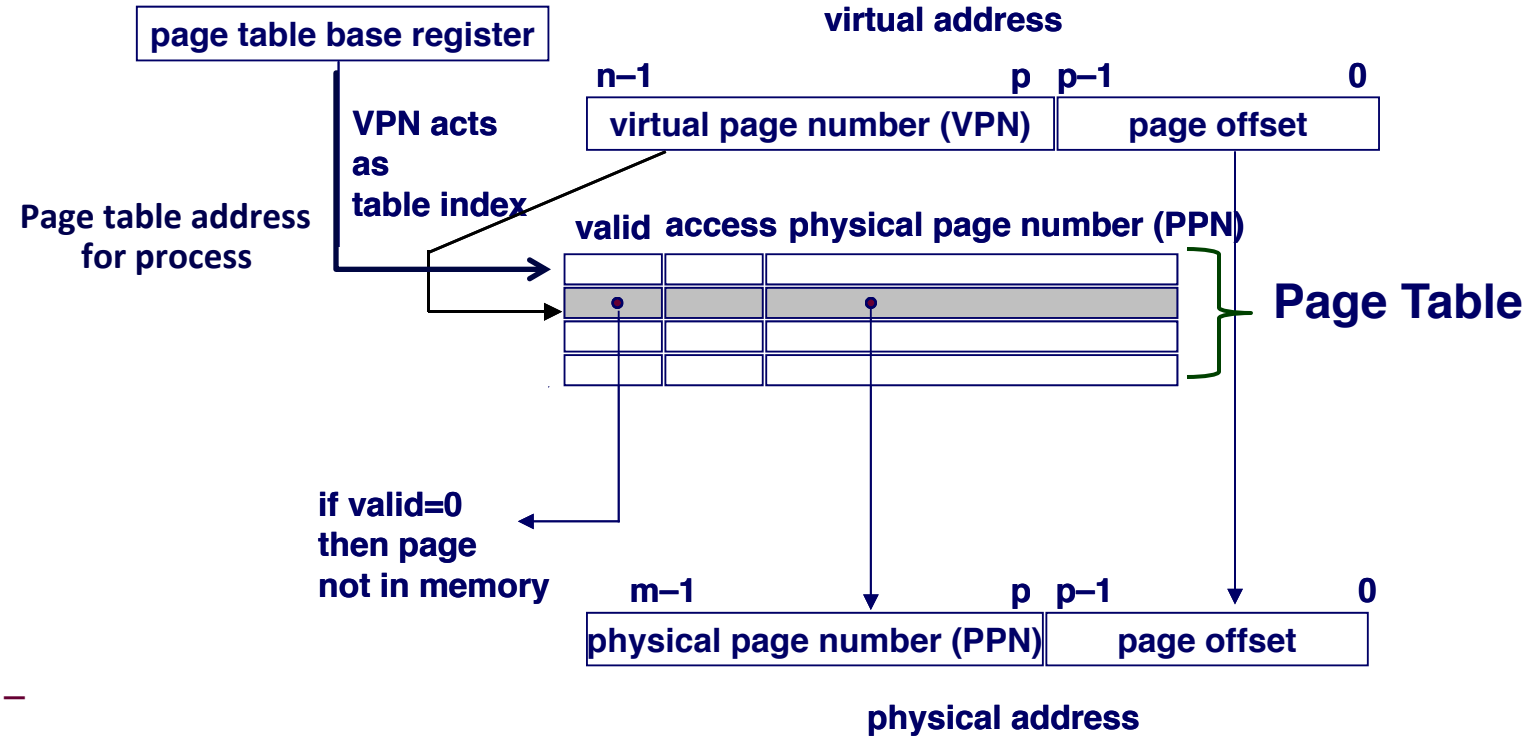
- if (valid bit = 1) then the page is in memory.
  - » Use physical page number (PPN) to construct address
- if (valid bit = 0) then the page is on disk
  - » Page fault – must load page from disk into a free page in memory and update page table



Page Table Operation

Checking Protection

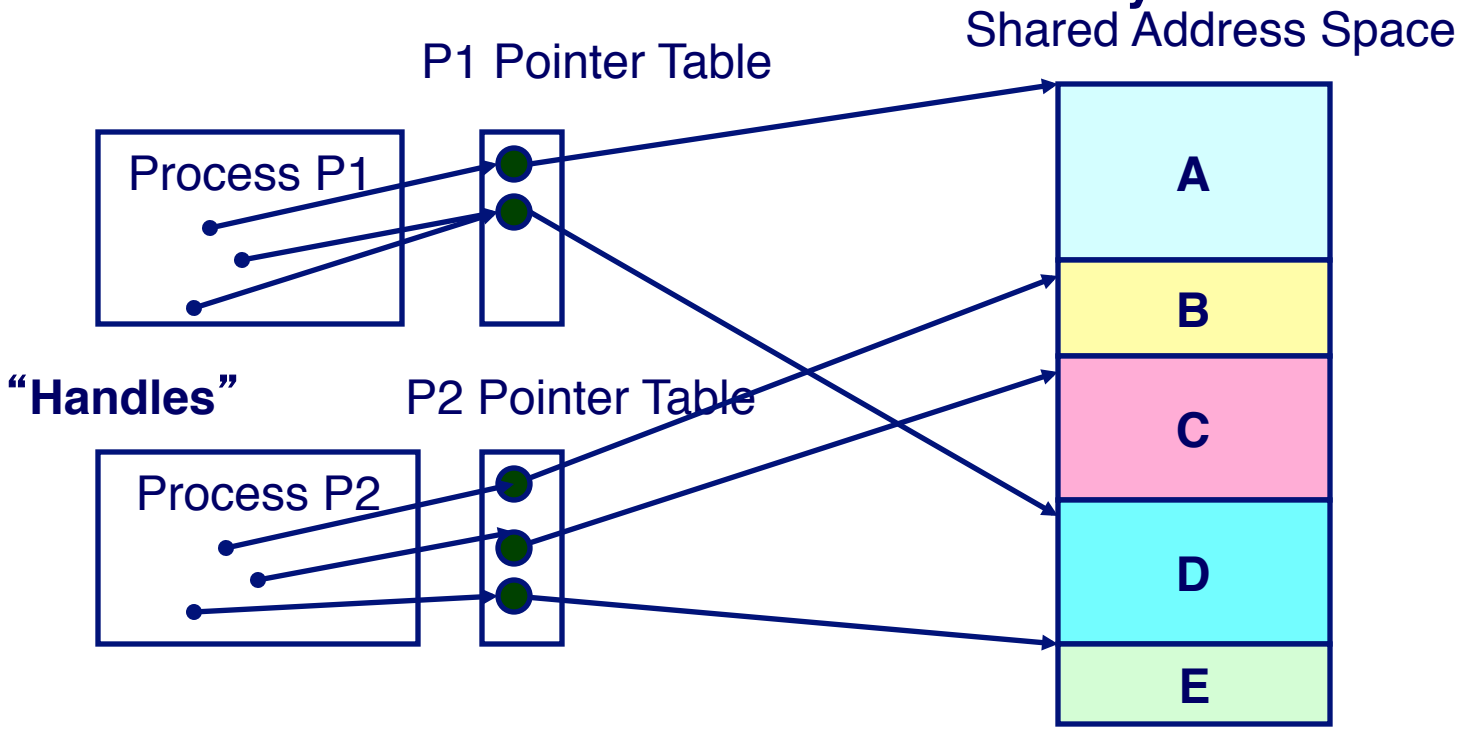
- Access rights field indicate allowable access
  - e.g., read-only, read-write, execute-only
  - typically support multiple protection modes (e.g., kernel vs. user)
- Protection violation fault if user doesn't have necessary permission



Contrast: Macintosh Memory Model

MAC OS 1–9

- **Does not use traditional virtual memory**



All program objects accessed through "handles"

- **Indirect reference through pointer table**
- **Objects stored in shared global address space**

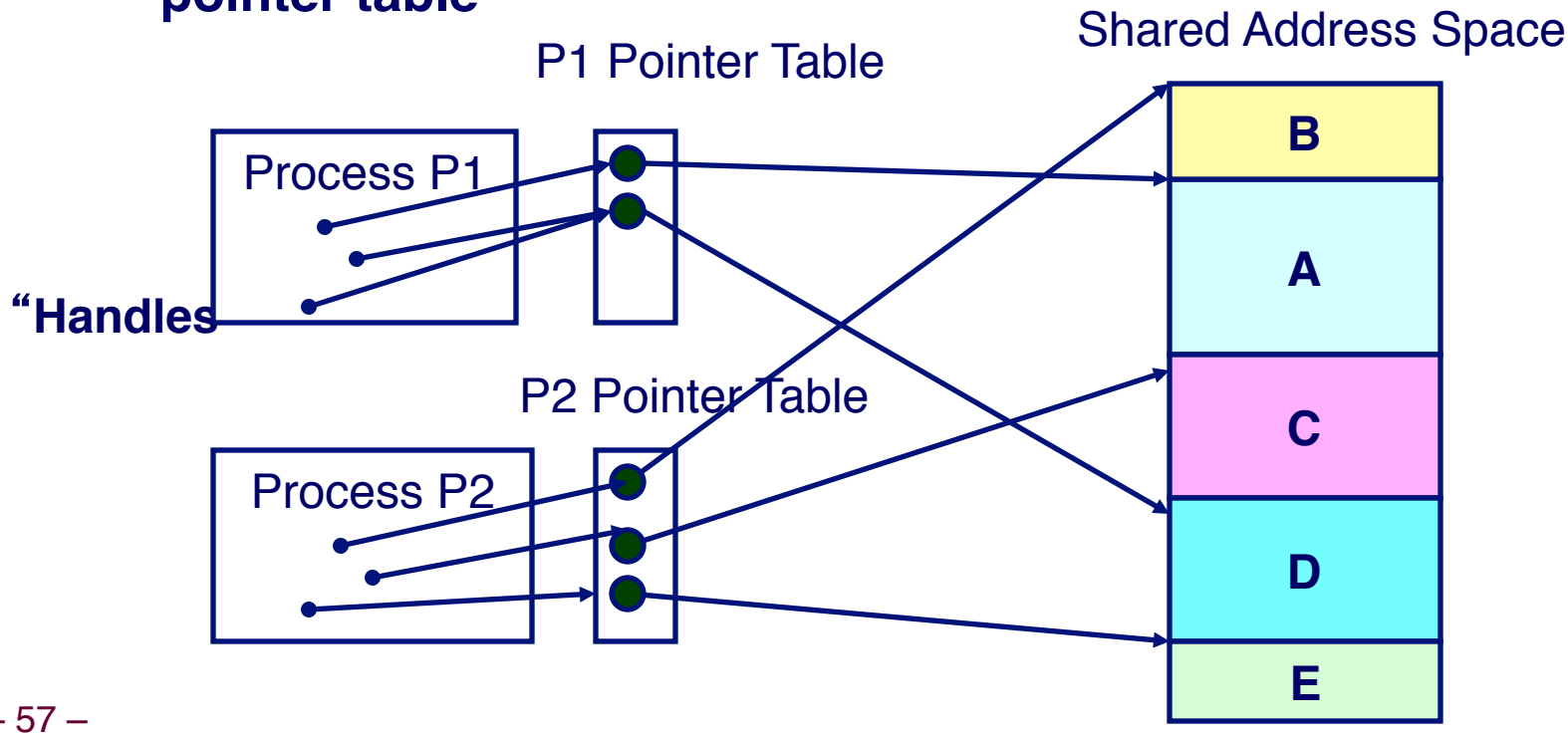
Macintosh Memory Management

Allocation / Deallocation

- Similar to free-list management of malloc/free

Compaction

- Can move any object and just update the (unique) pointer in pointer table



Mac vs. VM-Based Memory Mgmt

Allocating, deallocating, and moving memory:

- can be accomplished by both techniques

Block sizes:

- Mac: variable-sized
  - may be very small or very large
- VM: fixed-size
  - size is equal to *one page* (4KB on x86 Linux systems)

Allocating contiguous chunks of memory:

- Mac: contiguous allocation is *required*
- VM: can map contiguous range of virtual addresses to disjoint ranges of physical addresses

Protection

- Mac: "wild write" by one process can corrupt another's data

Modern O/S

"Modern" Operating System

- Virtual memory with protection
- *Preemptive multitasking*

Windows/NT: Based on VMS from Digital Equipment

- Primary designer (Cutler) from Digital Equipment
- "Heavy" processes, event queues & messages

Mac OS/X: Based on FreeBSD & MACH OS

- FreeBSD derived from "Berkeley Standard Distribution"
  - Contracted by DARPA to build a more robust UNIX
- MACH OS Developed at CMU in late 1980's
  - Key idea was "micro-kernel" - limited functionality implemented by kernel