

# Introduction to Operating Systems and Processes

**HPPS**

David Marchant

**Based on slides by:**

Randal E. Bryant and David R. O'Hallaron with  
modifications by Troels Henriksen

# A quick note about me

- **PhD student from Scotland**
- **Jeg taler ikke dansk :(**
- **'David' on the Discord**
- **Office: HCØ, Building B, room 772-01-0-S06**

# Why study operating systems?

- **They are where the magic happens**
- **For inspiration**
  - One of the most potent *engineering abstractions* in computing
  - Each program thinks it has an entire machine to itself
  - Controlled communication between programs.
  - Abstracts over hardware differences
- **Practical skills**
  - Performance characteristics of the abstraction
  - What is fundamentally possible?

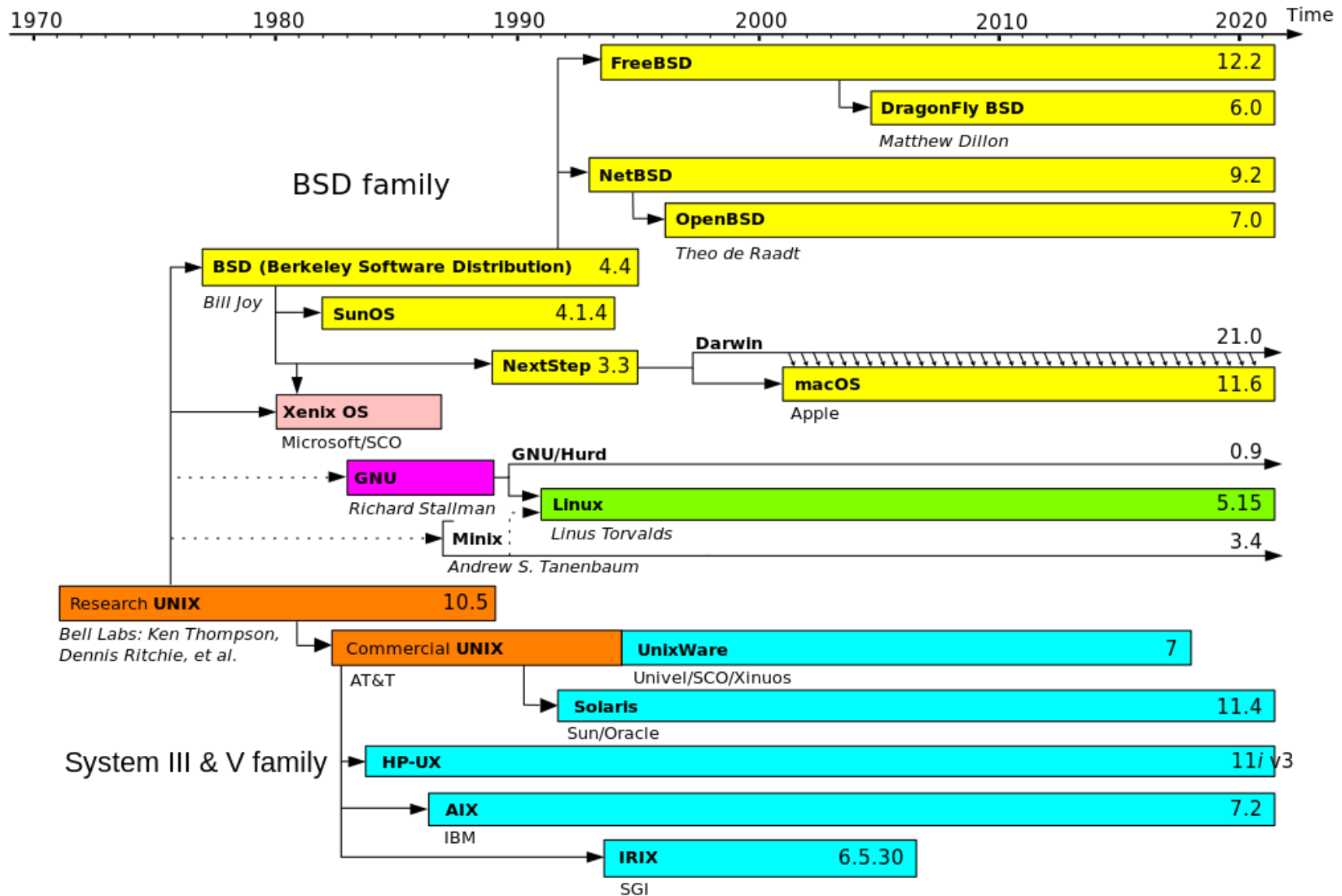
# Unix

## ■ What is Unix?

- Unix is an operating system developed in the 1970s by Ken Thompson and Dennis Ritchie
- Most modern operating systems heavily influenced by Unix (even Windows)
- Many operating systems are *direct descendants*: Linux, iOS, macOS, the \*BSDs, etc

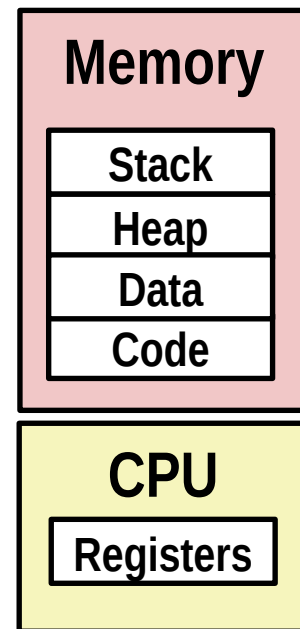
## ■ Why Unix?

- Unix is *simple* and *representative* of modern systems
- We will use Unix designs for all examples

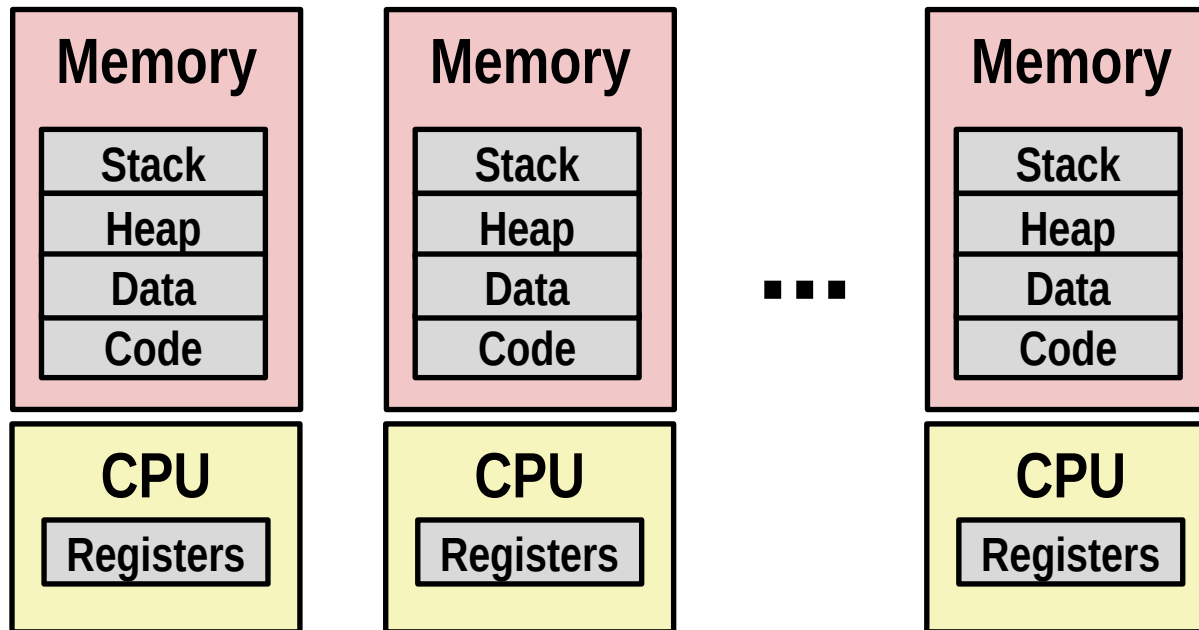


# Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”
- **Process provides each program with two key abstractions:**
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*



# Multiprocessing: The Illusion



- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, ...
  - Background tasks
    - Monitoring network & I/O devices

# Multiprocessing Example

```

Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND    %CPU TIME    #TH    #WQ    #PORT    #MREG    RPRVT    RSHRD    RSIZE    VPRVT    VSIZE
99217-  Microsoft Of 0.0 02:28.34 4      1      202      418      21M      24M      21M      66M      763M
99051   usbmuxd     0.0 00:04.10 3      1      47       66      436K     216K     480K     60M      2422M
99006   iTunesHelper 0.0 00:01.23 2      1      55       78      728K     3124K    1124K    43M      2429M
84286   bash        0.0 00:00.11 1      0      20       24      224K     732K     484K     17M      2378M
84285   xterm       0.0 00:00.83 1      0      32       73      656K     872K     692K     9728K    2382M
55939-  Microsoft Ex 0.3 21:58.97 10     3      360      954      16M      65M      46M      114M     1057M
54751   sleep       0.0 00:00.00 1      0      17       20      92K      212K     360K     9632K    2370M
54739   launchdadd  0.0 00:00.00 2      1      33       50      488K     220K     1736K    48M      2409M
54737   top         6.5 00:02.53 1/1    0      30       29      1416K    216K     2124K    17M      2378M
54719   automountd  0.0 00:00.02 7      1      53       64      860K     216K     2184K    53M      2413M
54701   ocsdpd     0.0 00:00.05 4      1      61       54      1268K    2644K    3132K    50M      2426M
54661   Grab       0.6 00:02.75 6      3      222+     389+     15M+     26M+     40M+     75M+     2556M+
54659   cookied    0.0 00:00.15 2      1      40       61      3316K    224K     4088K    42M      2411M
53818   mdworker   0.0 00:01.67 4      1      52       91      7628K    7412K    16M      48M      2438M
50978   mdworker   0.0 00:11.17 3      0      53       91      2464K    6148K    9976K    44M      2434M
50078   emacs      0.0 00:06.70 1      0      20       35      52K      216K     88K      18M      2392M

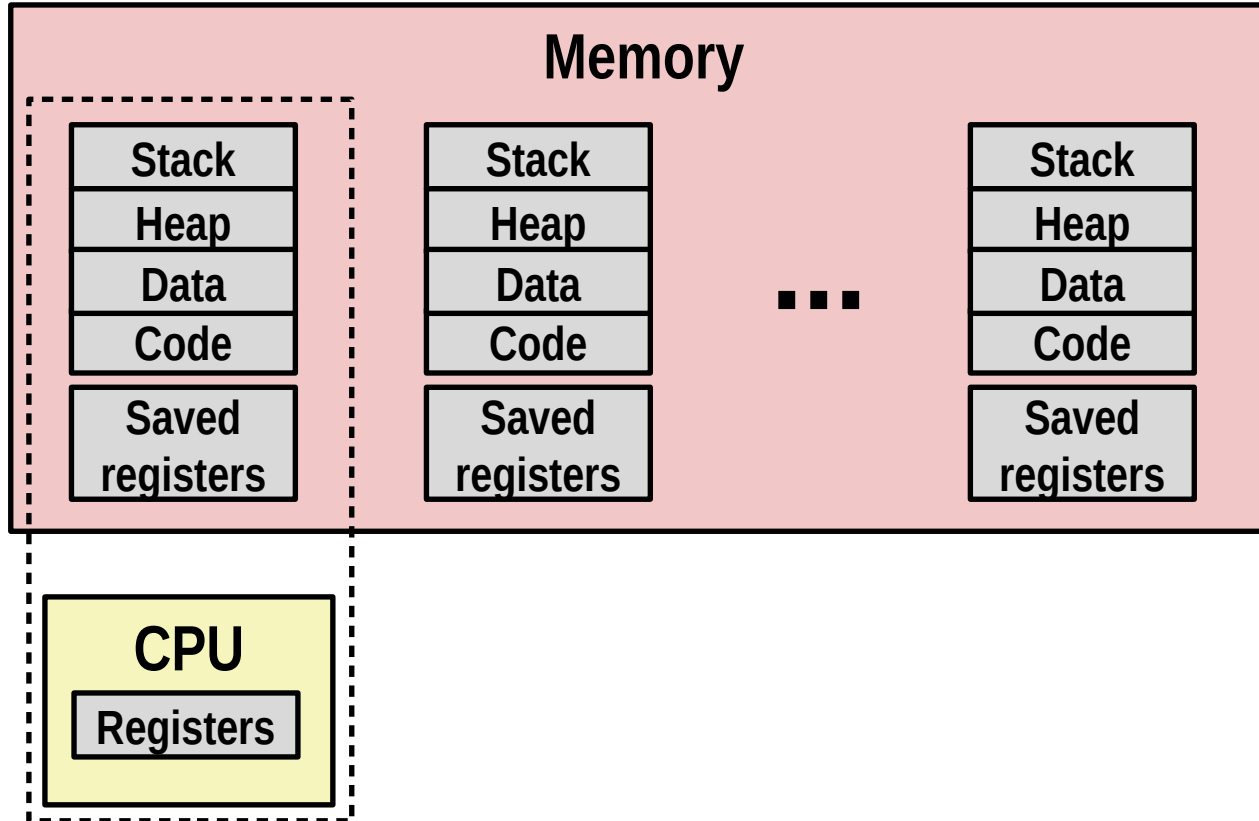
```

## ■ Running program “top” on Mac

- System has 123 processes, 5 of which are active
- Identified by Process ID (PID)

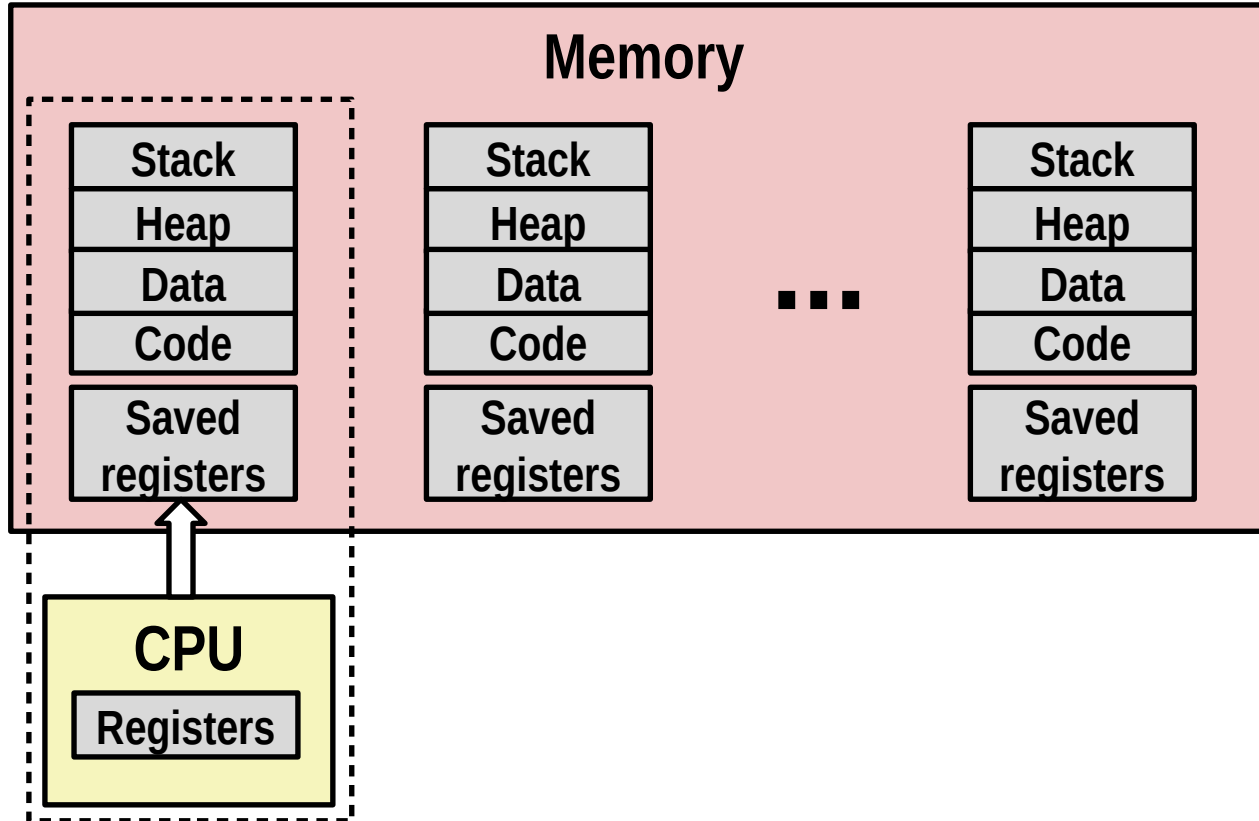


# Multiprocessing: The (Traditional) Reality



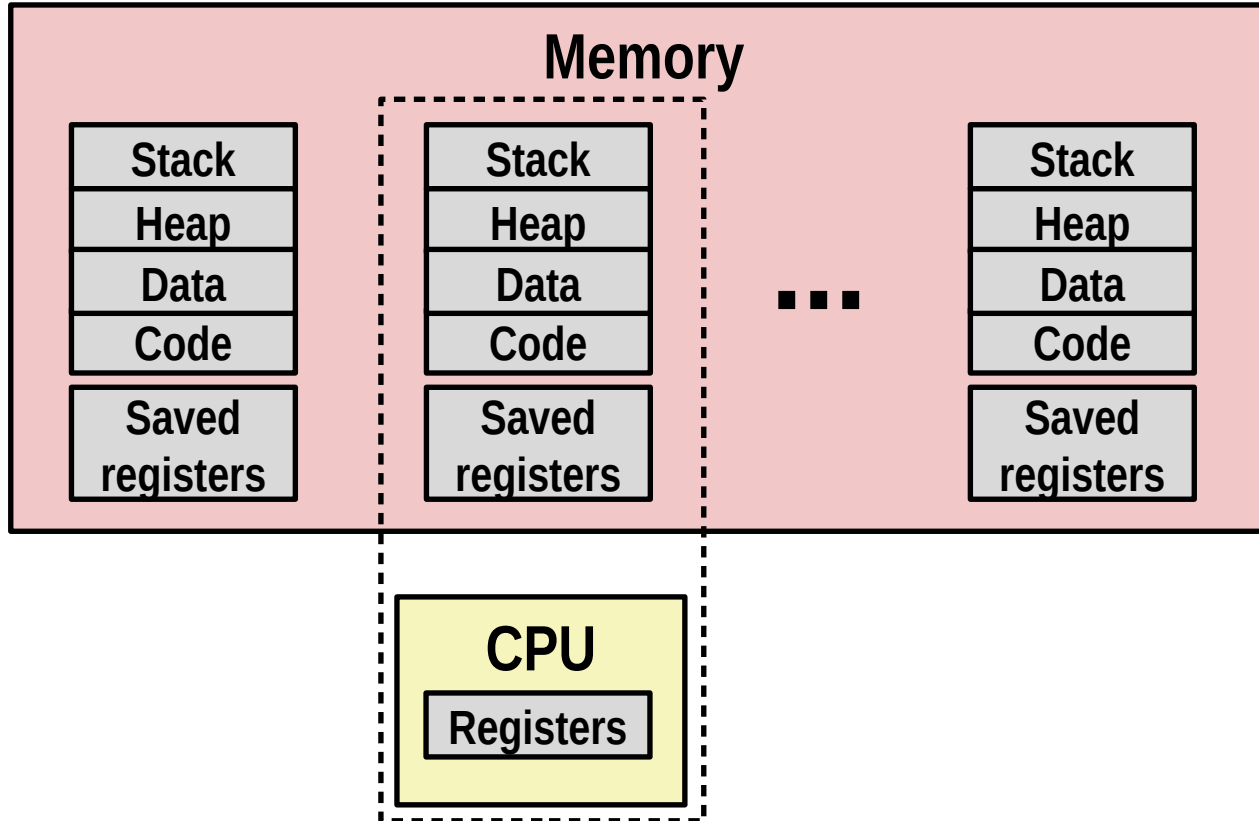
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for non-executing (suspended) processes saved in memory

# Multiprocessing: The (Traditional) Reality



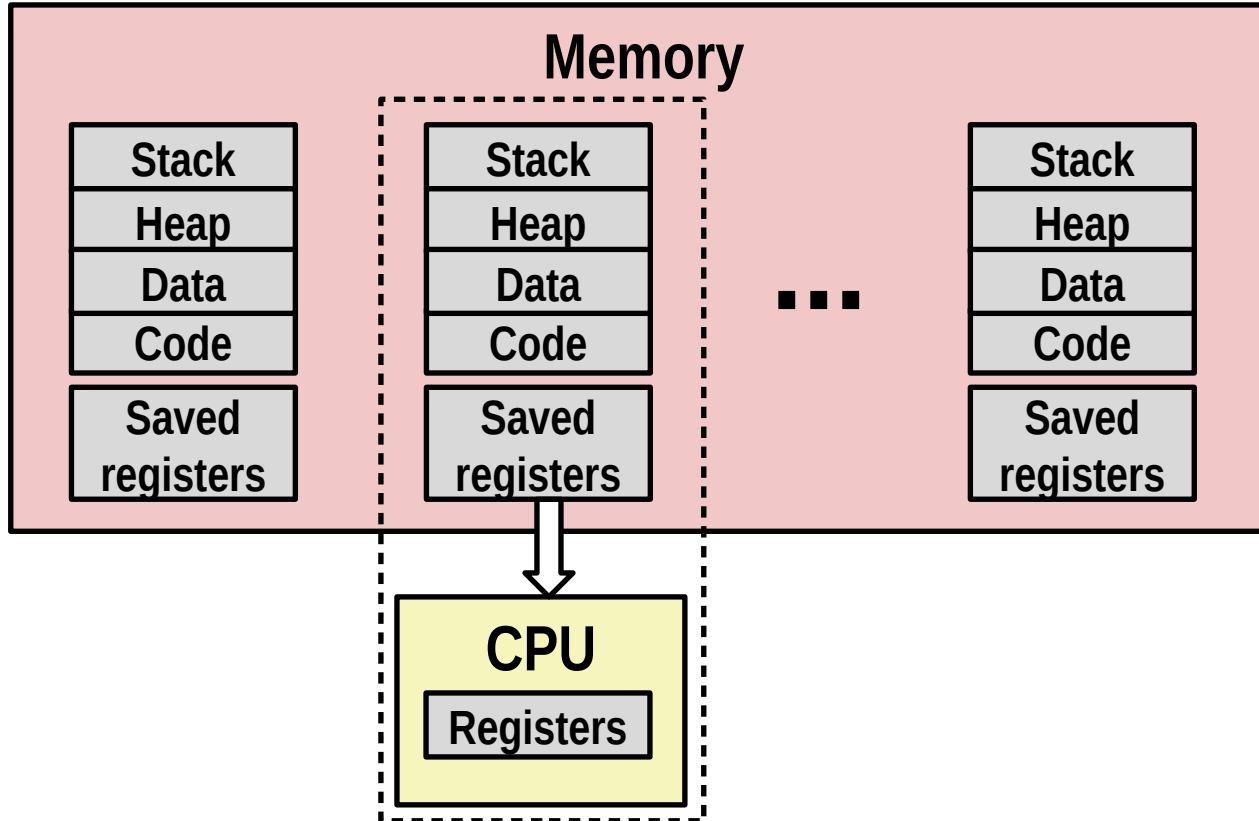
- **Save current registers in memory**

# Multiprocessing: The (Traditional) Reality



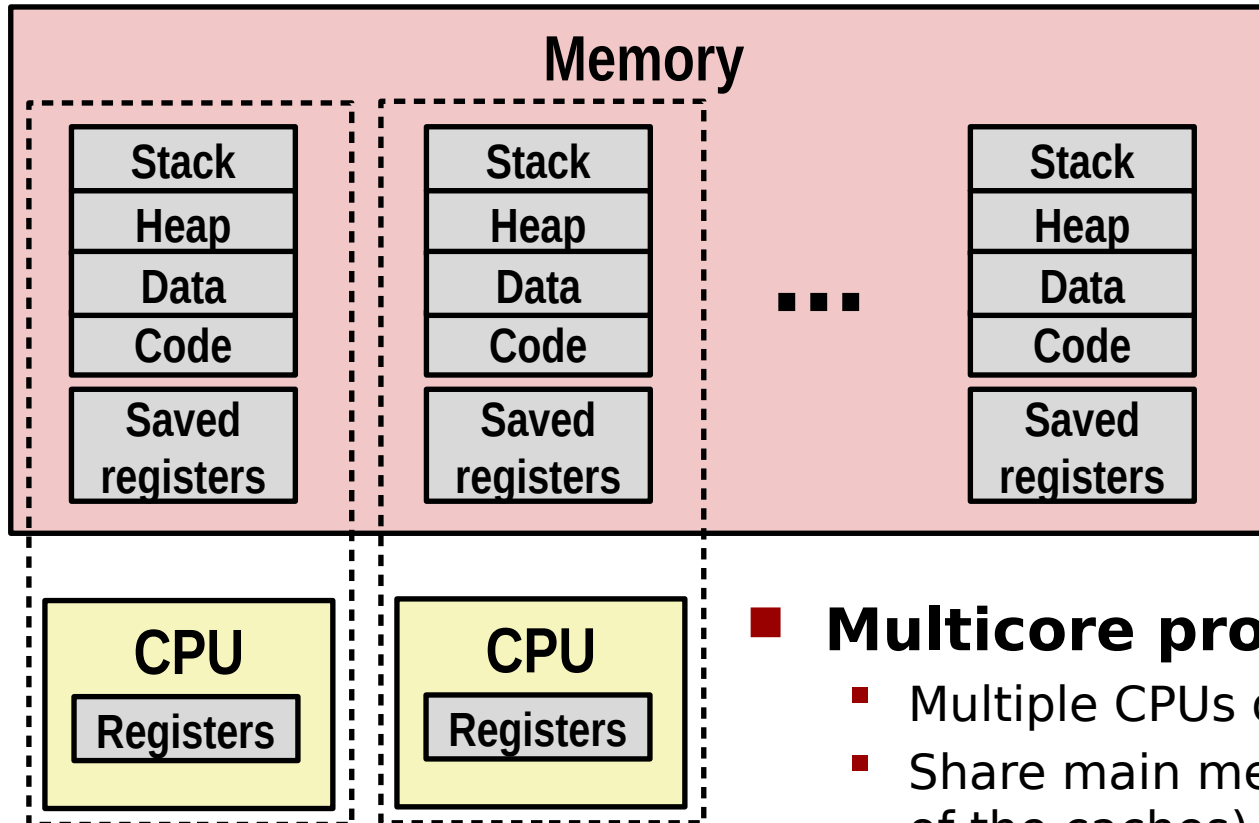
- **Schedule next process for execution**

# Multiprocessing: The (Traditional) Reality



- **Load saved registers and switch address space (context switch)**

# Multiprocessing: The (Modern) Reality

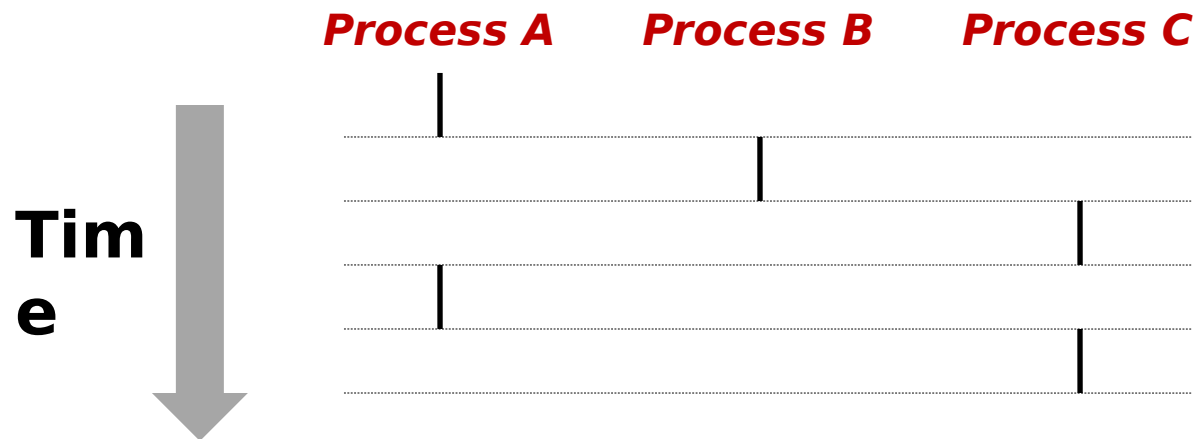


## ■ Multicore processors

- Multiple CPUs on single chip
- Share main memory (and some of the caches)
- Each can execute a separate process
  - Scheduling of processors onto cores done by kernel

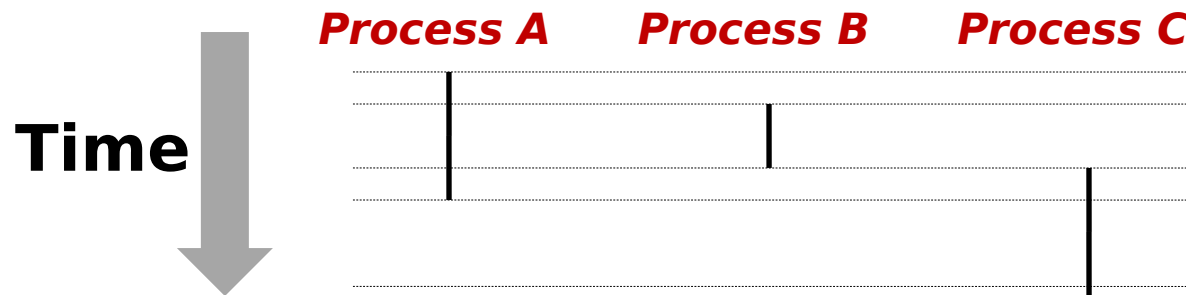
# Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (are *concurrent*) if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



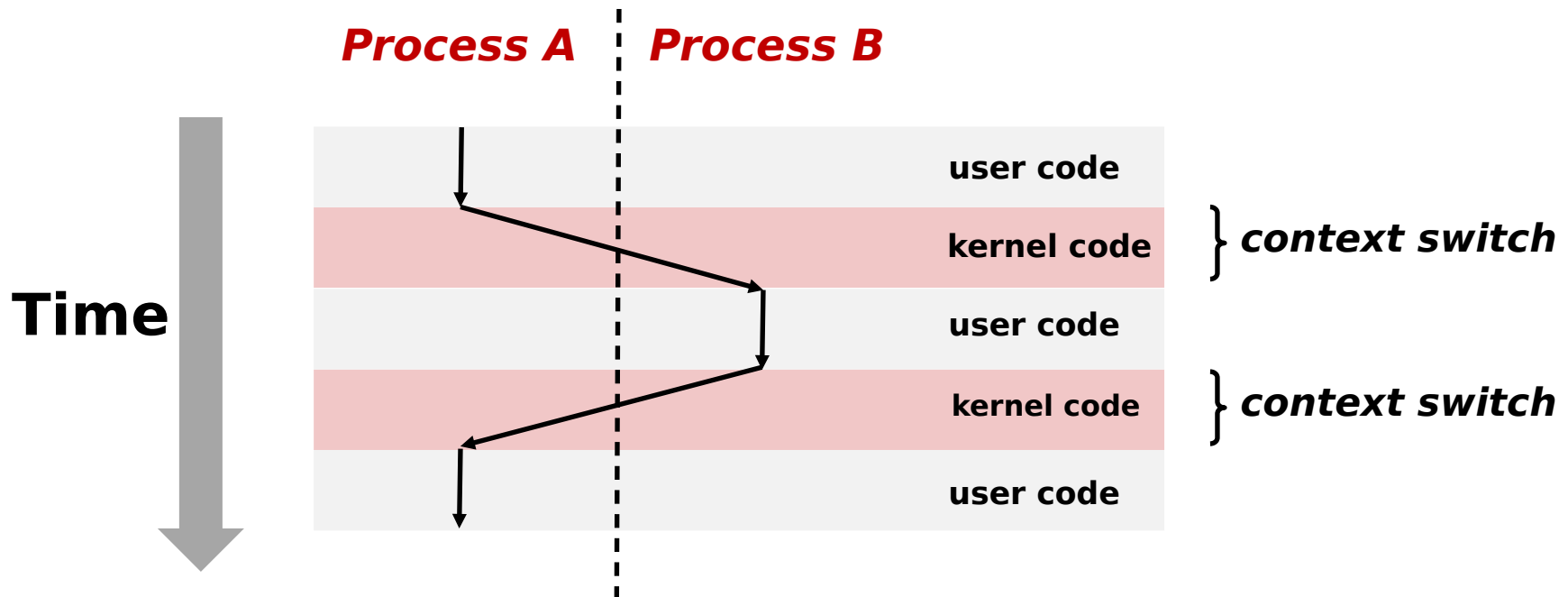
# User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other



# Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the **kernel**
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a **context switch**





# Overheads

- **Context switch is not free**

- It is a fairly quick operation to switch from one process to another (roughly 1.2 microseconds plus cache misses)
- But many processes switching can pile on the overheads

- **Most schedulers have some variant of round-robin scheduling**

- This is where the scheduler will try to ensure each process will get some execution time reasonably regularly
- Avoids livelock (where things can progress, but don't)

- **Excessive numbers of processes cause *thrashing***

- **We will be returning to concurrency in the new year**

# System Calls

- Systems calls ask the operating system to perform a task on behalf of the process.
- Each x86-64 system call has a unique ID number
- Examples:

<i>Number</i>	<i>Name</i>	<i>Description</i>
0	read	Read file
1	write	Write file
2	open	Open file
3	close	Close file
4	stat	Get info about file
57	fork	Create process
59	execve	Execute a program
60	_exit	Terminate process
62	kill	Send signal to process

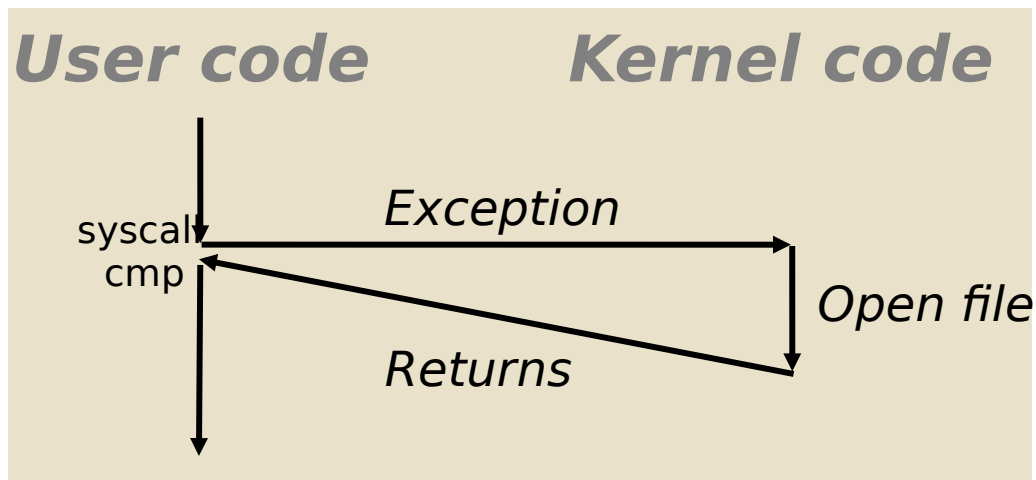
# System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
000000000000e5d70 <__open>:
```

```
...
```

```
e5d79:  b8 02 00 00 00      mov  $0x2,%eax    # open is syscall #2
e5d7e:  0f 05               syscall           # Return value in %rax
e5d80:  48 3d 01 f0 ff ff    cmp  $0xffffffffffffffff001,%rax
...
e5dfa:  c3                  retq
```



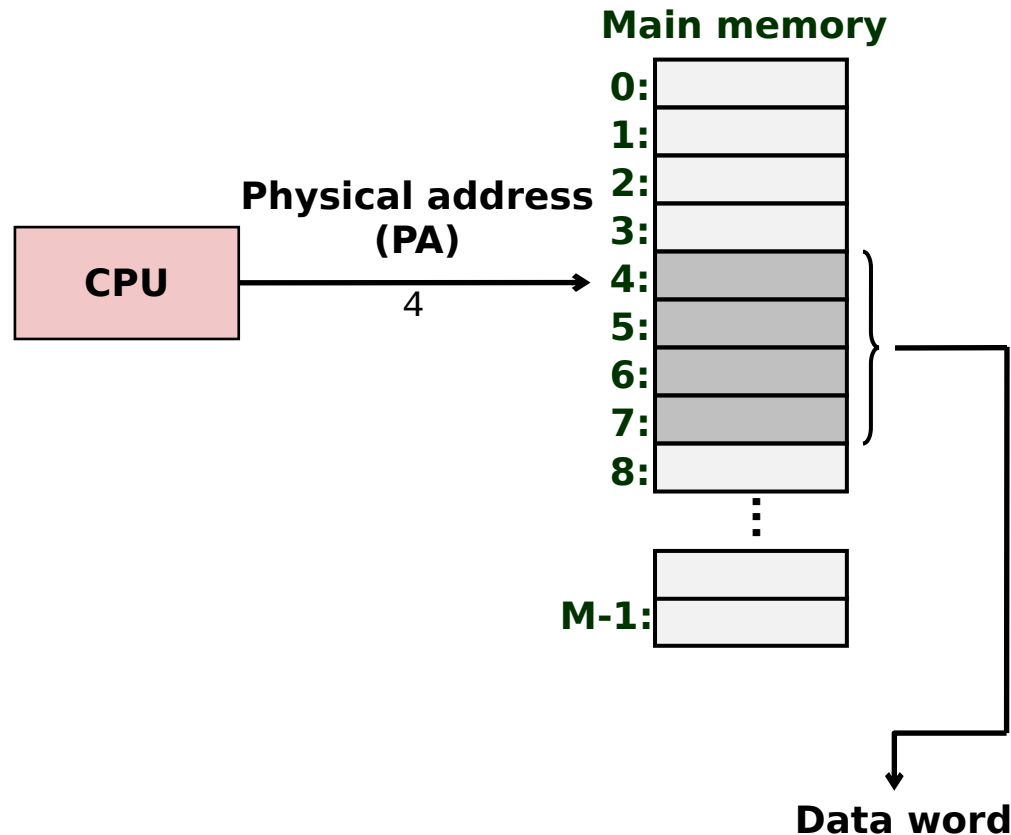
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

# Summary

## ■ Processes

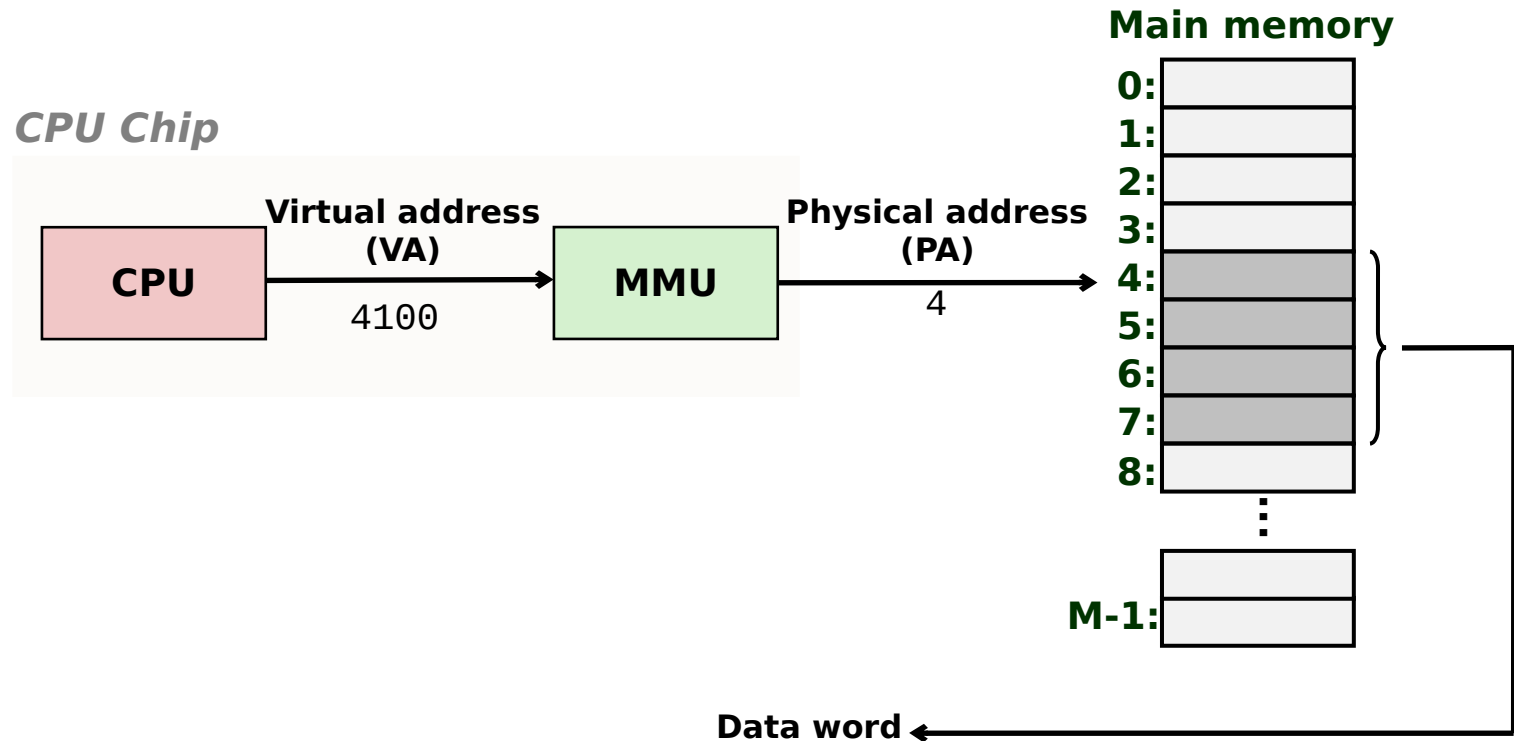
- At any given time, system has multiple active processes
- Only one can execute at a time on a single core, though
- Each process appears to have total control of processor + private memory space

# A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

# A System Using Virtual Addressing



- Used in all modern servers, laptops, and smart phones
- One of the great ideas in computer science

# Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:  
 $\{0, 1, 2, 3 \dots \}$
- **Virtual address space:** Set of  $N = 2^n$  virtual addresses  
 $\{0, 1, 2, 3, \dots, N-1\}$
- **Physical address space:** Set of  $M = 2^m$  physical addresses  
 $\{0, 1, 2, 3, \dots, M-1\}$

# Why Virtual Memory (VM)?

- **Uses main memory efficiently**

- Use DRAM as a cache for parts of a virtual address space

- **Simplifies memory management**

- Each process gets the same uniform linear address space

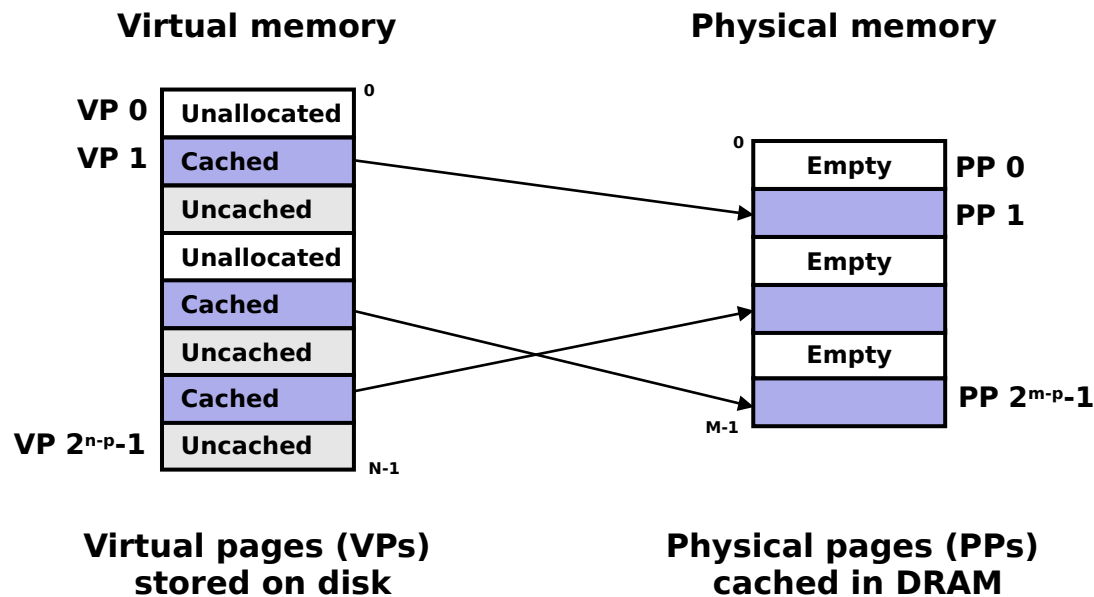
- **Isolates address spaces**

- One process can't interfere with another's memory
- User program cannot access privileged kernel information and code



# VM as a Tool for Caching

- Conceptually, **virtual memory** is an array of  $N$  contiguous bytes stored on disk (*from a caching perspective!*)
- The contents of the array on disk are cached in **physical memory (DRAM cache)**
  - These cache blocks are called *pages* (size is  $P = 2^p$  bytes)

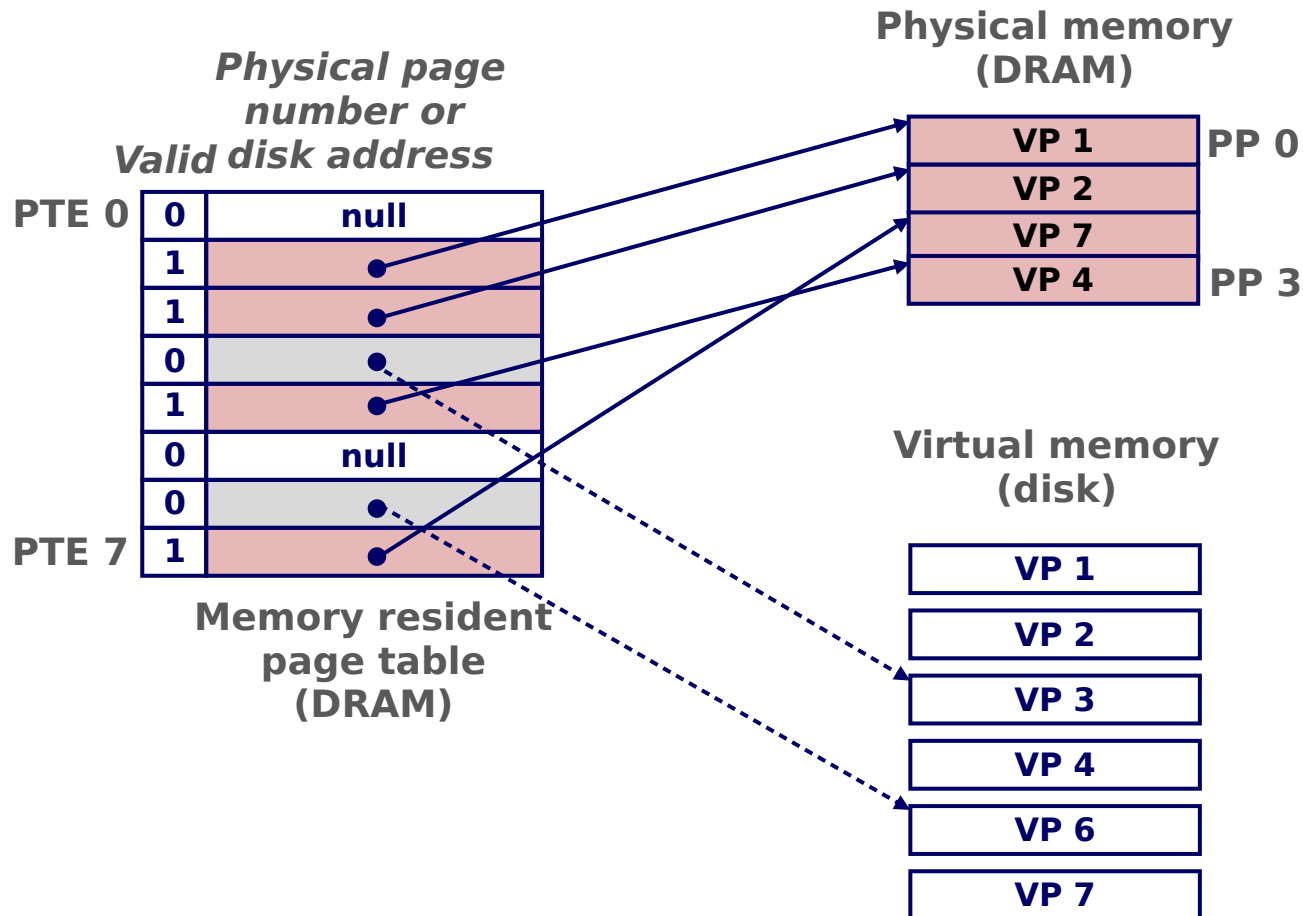


# DRAM Cache Organization

- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about **10x** slower than SRAM (CPU cache)
  - Disk is about **10,000x** slower than DRAM
  
- **Consequences**
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

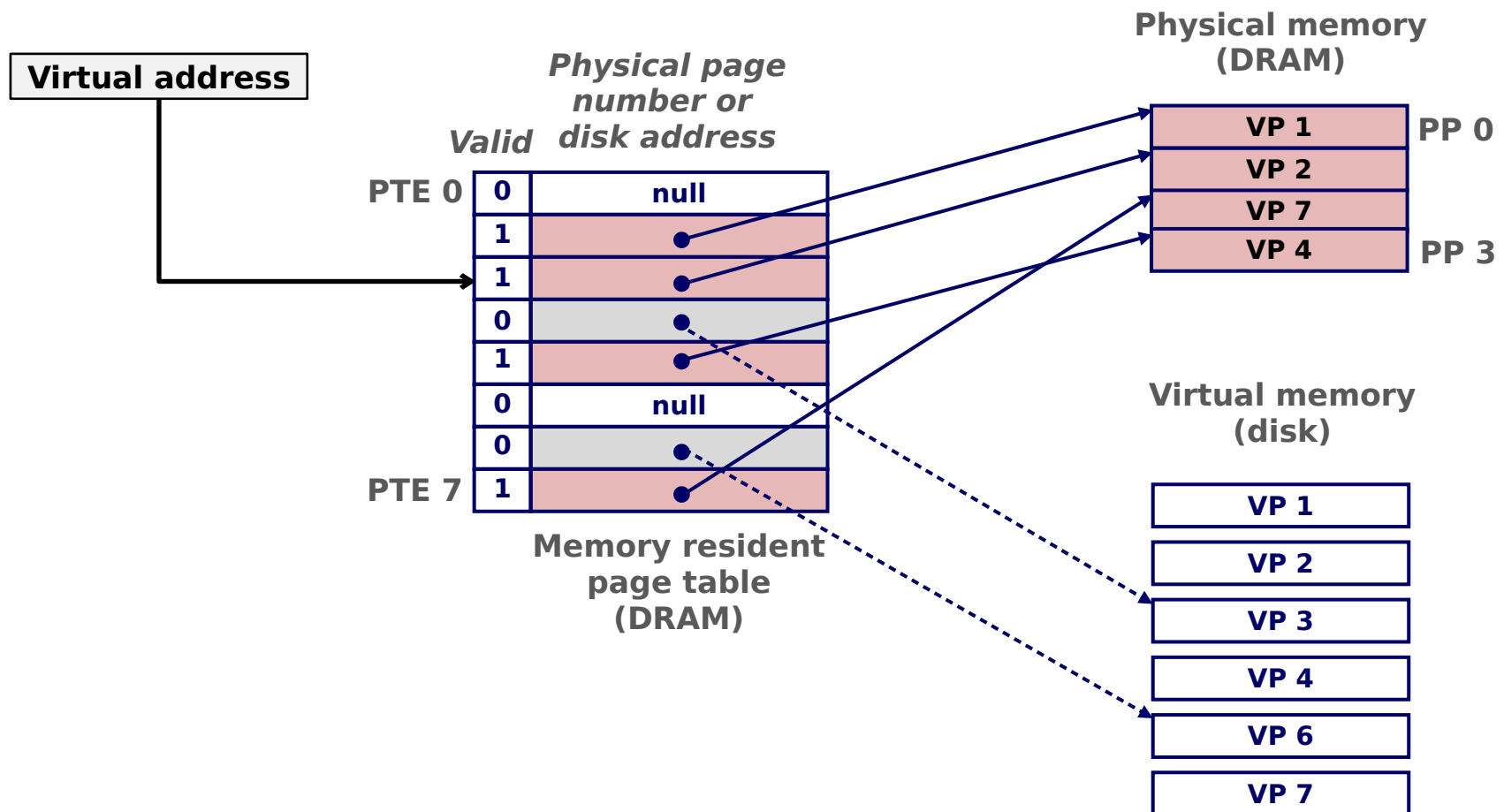
# Enabling Data Structure: Page Table

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



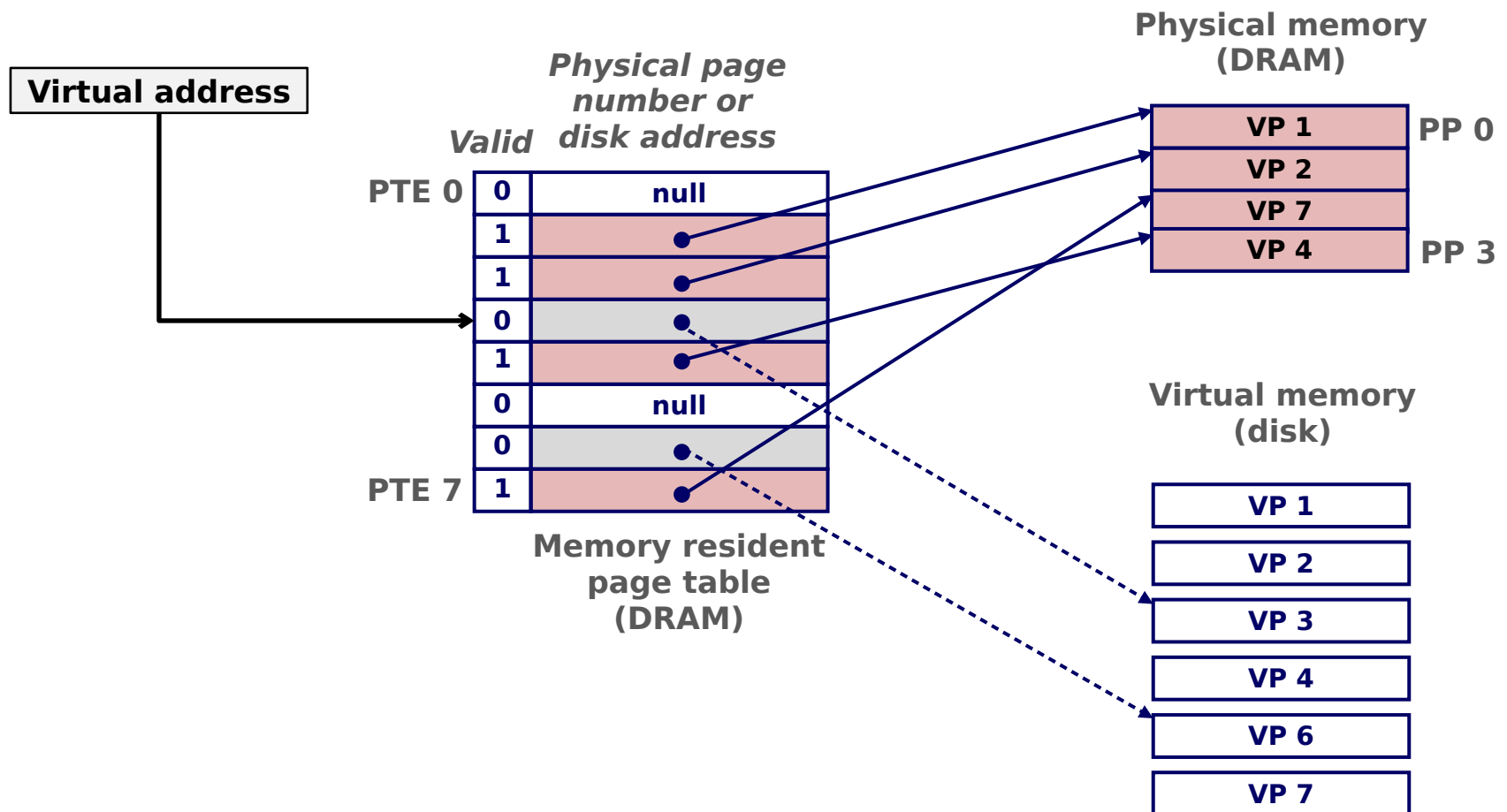
# Page Hit

- **Page hit:** reference to VM word that is in physical memory (DRAM cache hit)



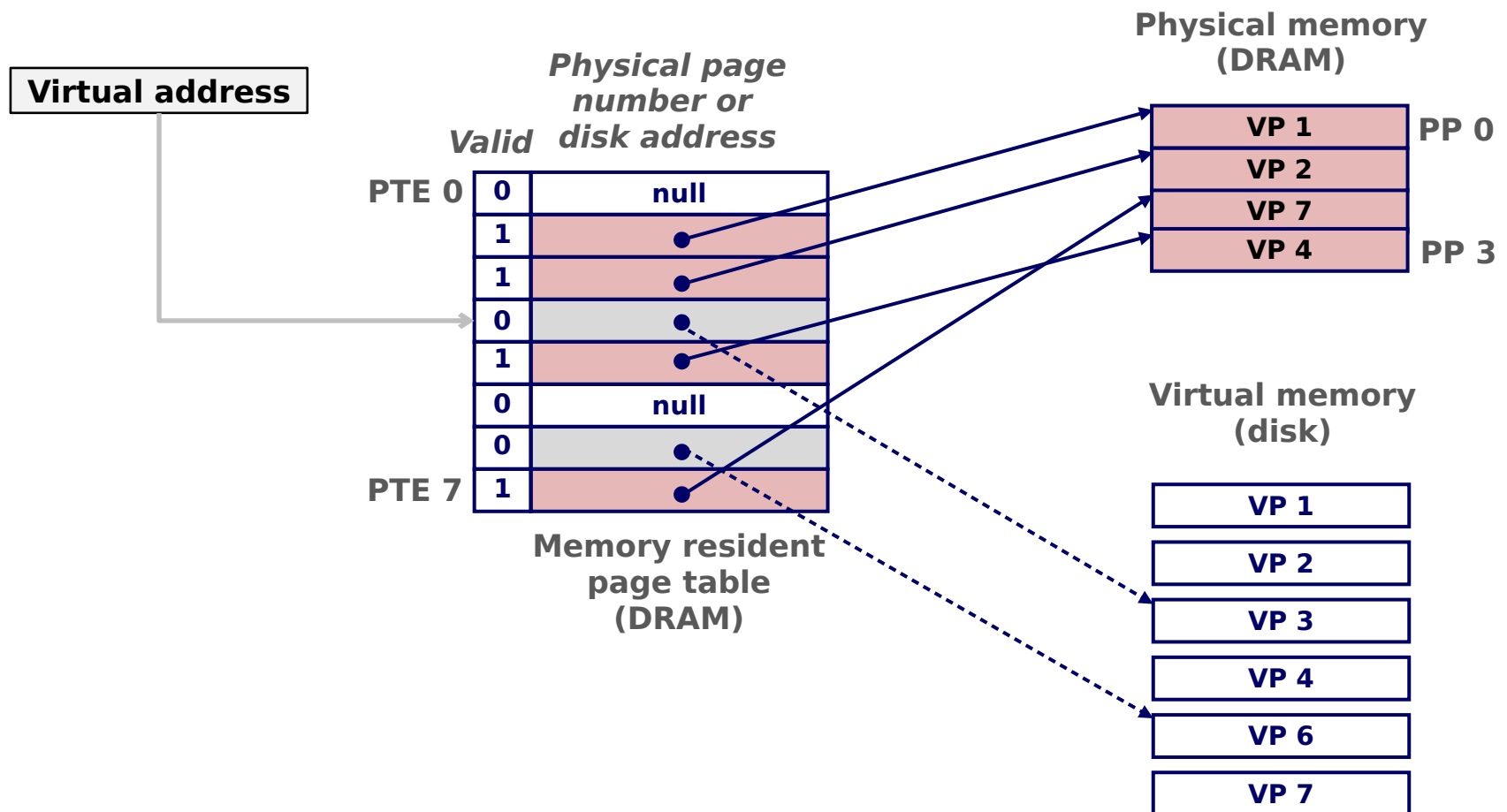
# Page Fault

- **Page fault:** reference to VM word that is not in physical memory (DRAM cache miss)



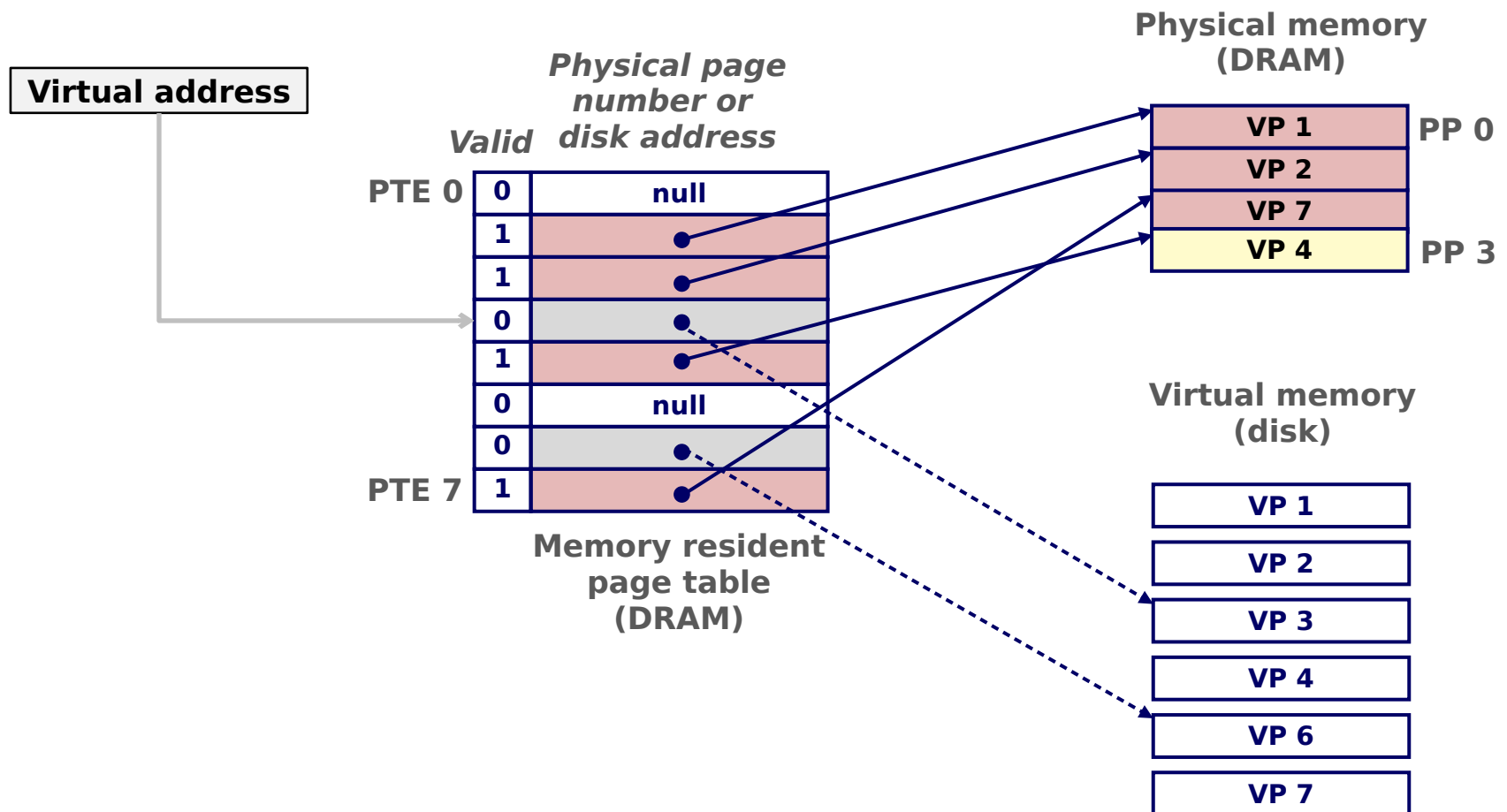
# Handling Page Fault

- Page miss causes page fault (an exception)



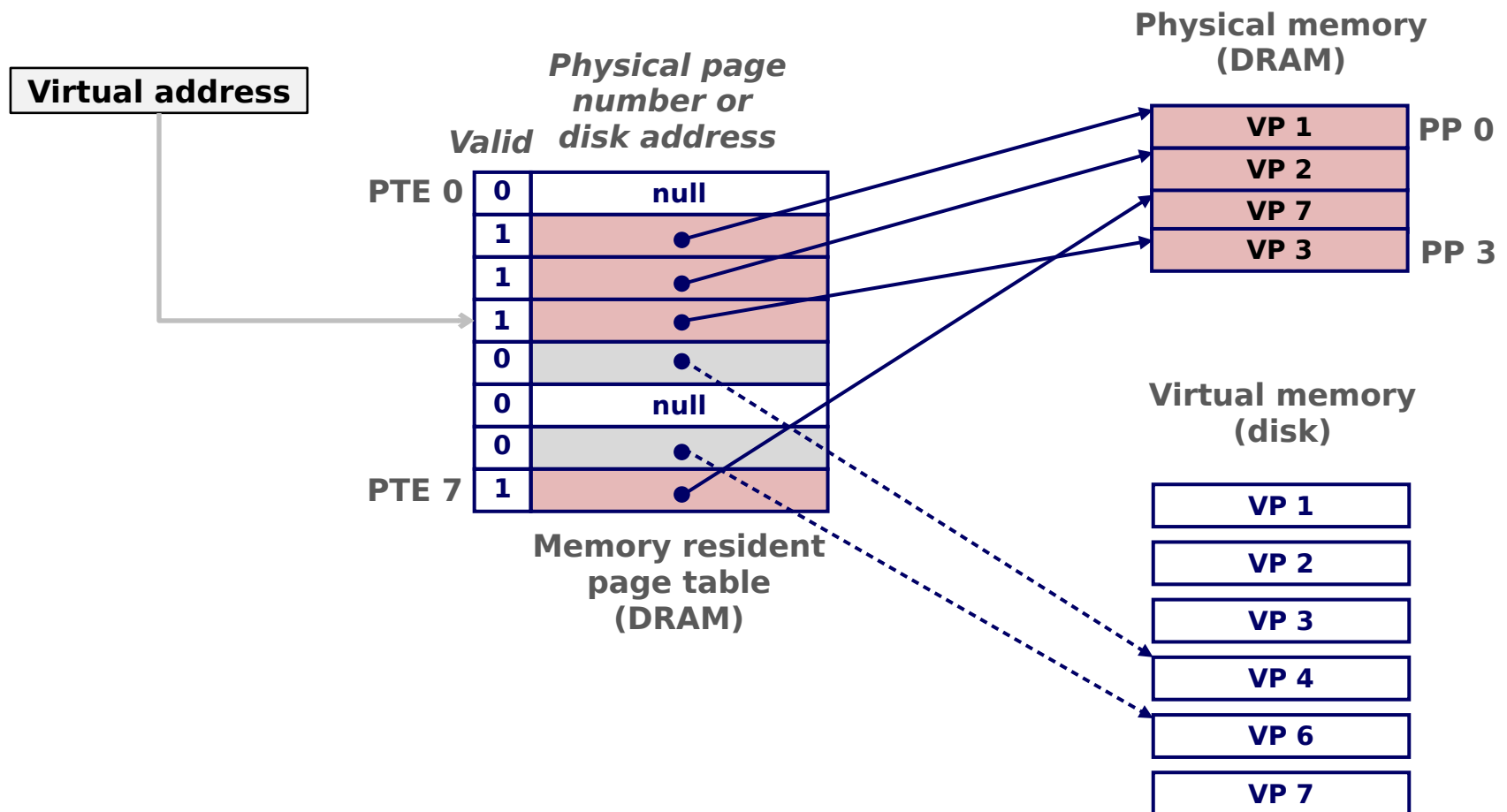
# Handling Page Fault

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



# Handling Page Fault

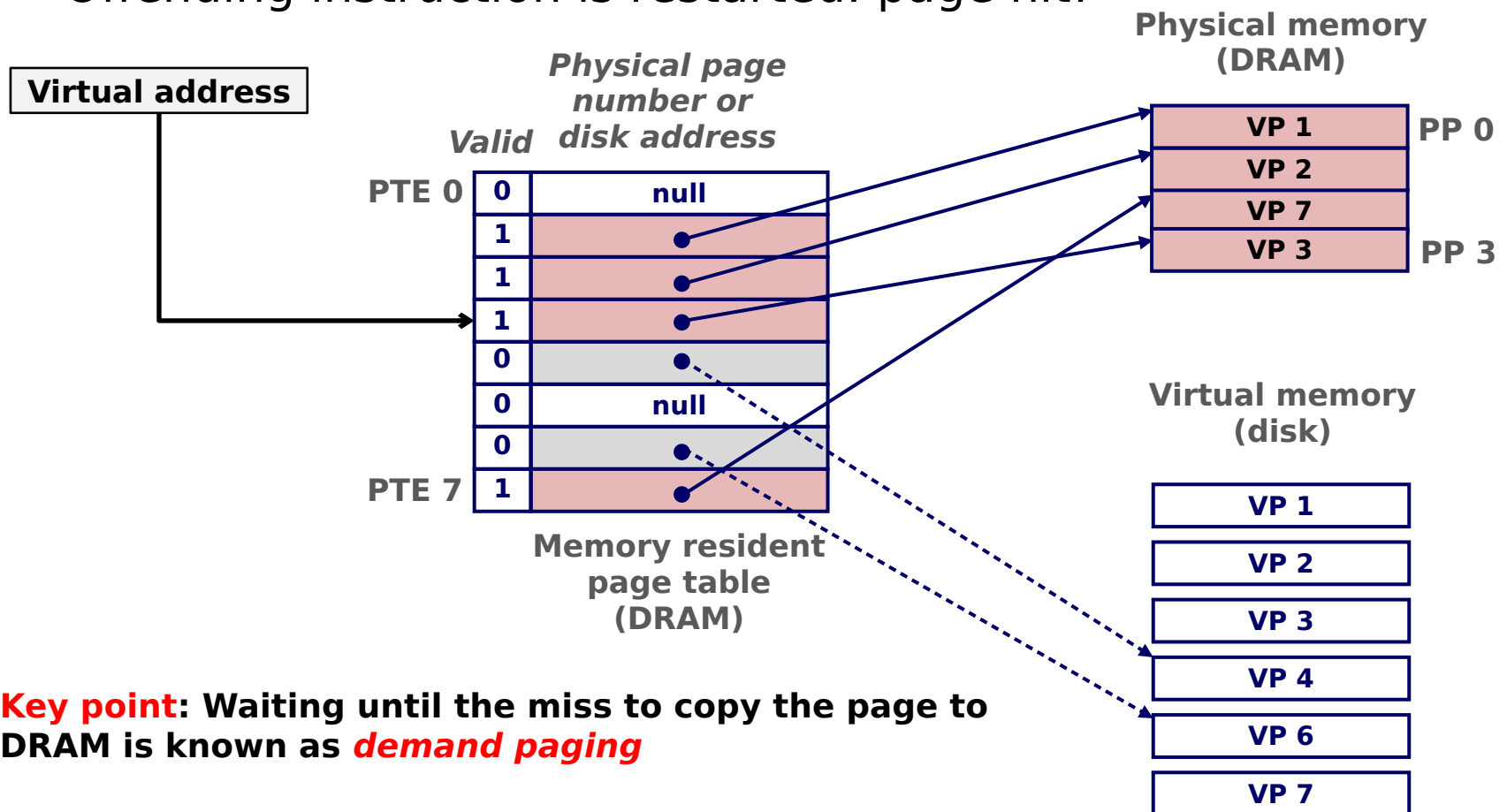
- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)





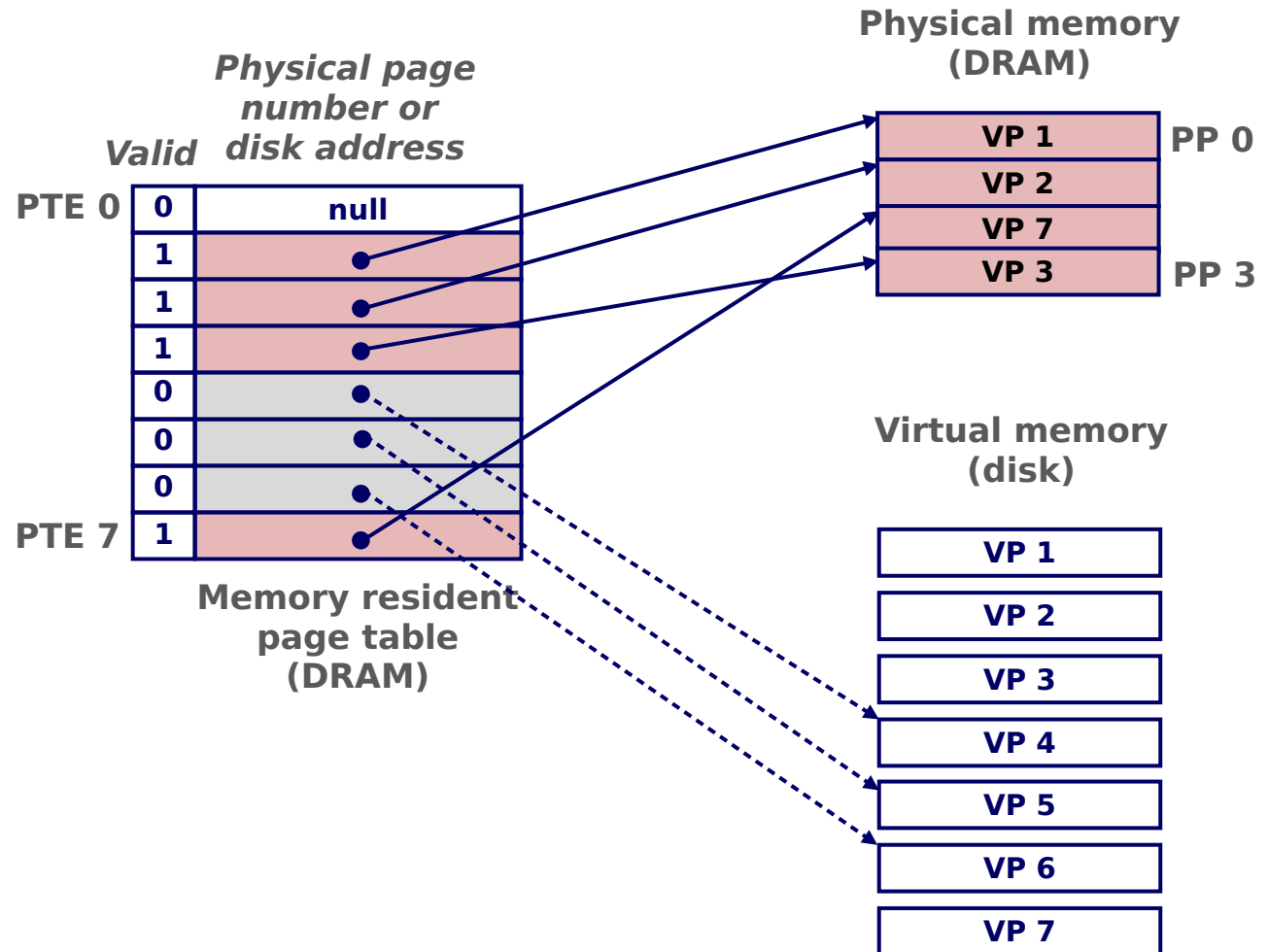
# Handling Page Fault

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



# Allocating Pages

- Allocating a new page (VP 5) of virtual memory.

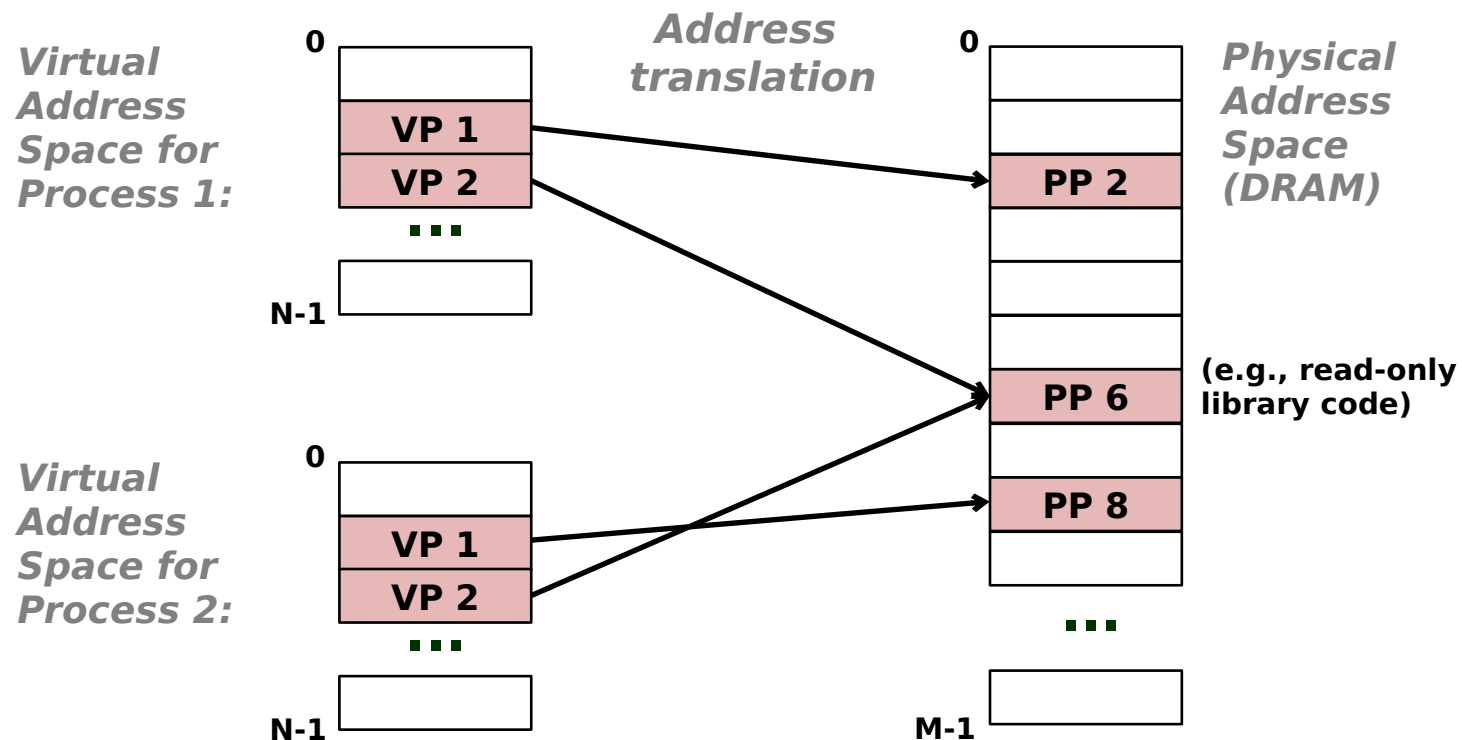


# Locality to the Rescue Again!

- Virtual memory seems terribly inefficient, but it 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 < main memory size)
  - 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

# VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well-chosen mappings can improve locality



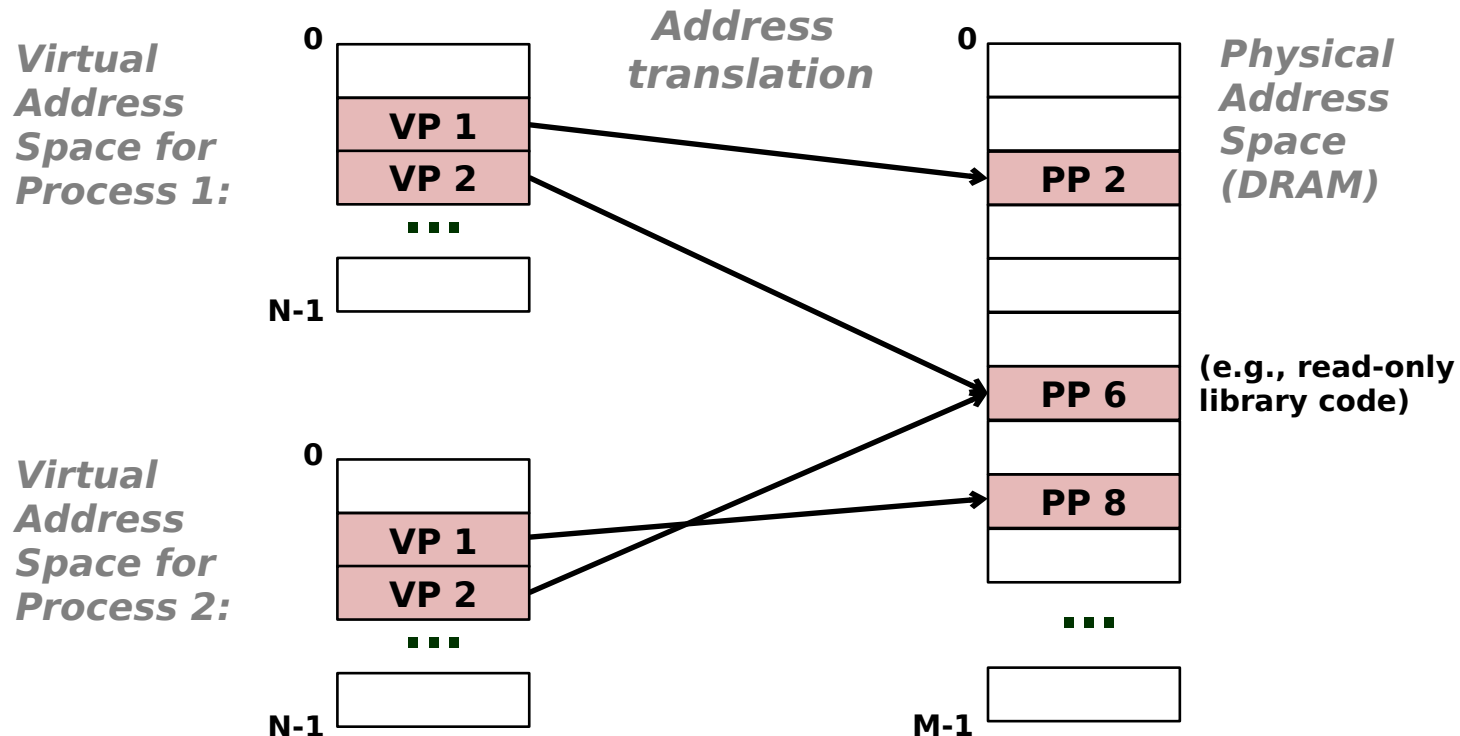
# VM as a Tool for Memory Management

## ■ Simplifying memory allocation

- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times

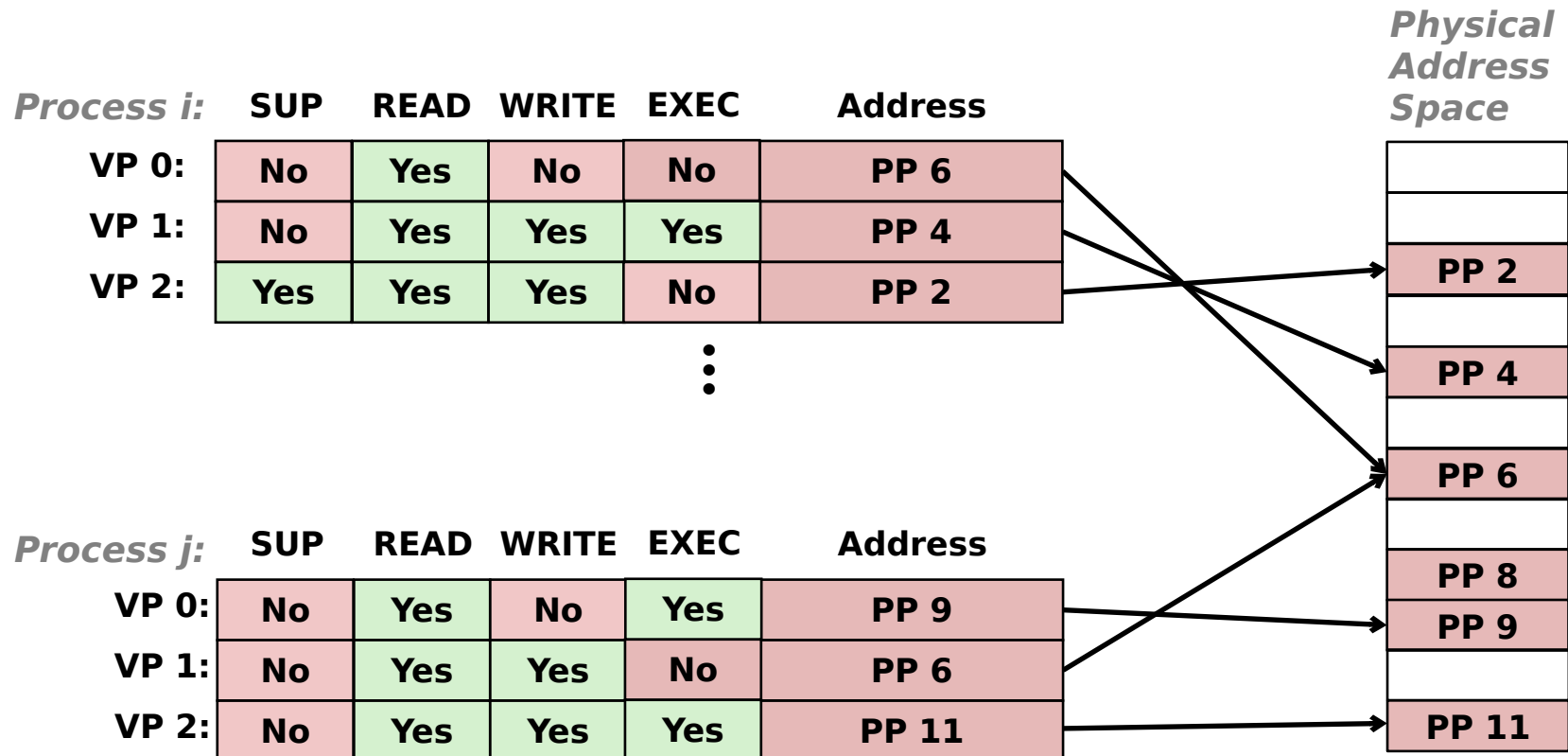
## ■ Sharing code and data among processes

- Map virtual pages to the same physical page (here: PP 6)



# VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access



# VM Address Translation

## ■ Virtual Address Space

- $V = \{0, 1, \dots, N-1\}$

## ■ Physical Address Space

- $P = \{0, 1, \dots, M-1\}$

## ■ Address Translation

***MAP***:  $V \rightarrow P \cup \{\neg\}$

- For virtual address ***a***:
    - ***MAP(a)*** = ***a'*** if data at virtual address ***a*** is at physical address ***a'*** in ***P***
    - ***MAP(a)*** =  $\neg$  if data at virtual address ***a*** is not in physical memory
- } Either invalid or stored on disk

# Summary of Address Translation Symbols

## ■ 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 the virtual address (VA)

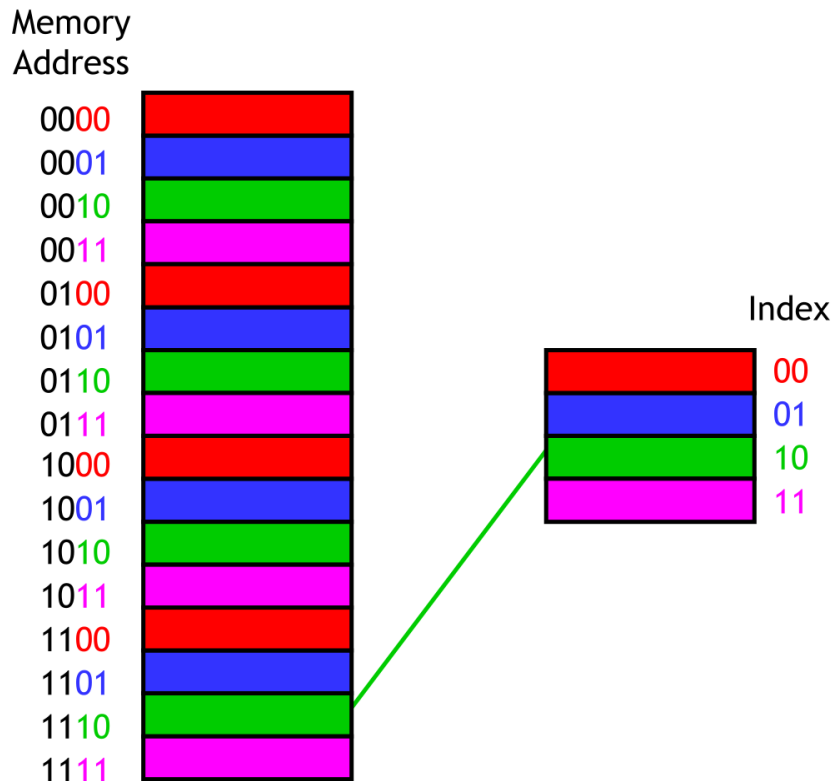
- **TLBI**: TLB index
- **TLBT**: TLB tag
- **VPO**: Virtual page offset
- **VPN**: Virtual page number

## ■ Components of the physical address (PA)

- **PPO**: Physical page offset (same as VPO)
- **PPN**: Physical page number



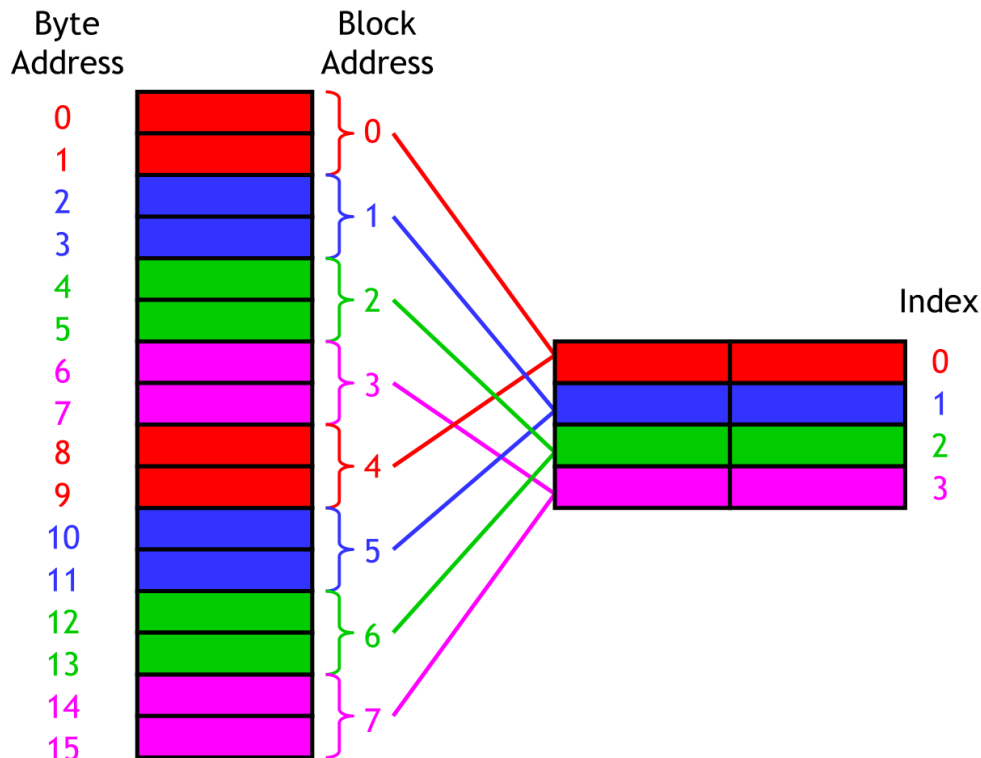
# Memory Mapping



## Direct Memory Mapping

- Each memory address maps to one cache block.
- Least significant bits of the address gives us the tag in the cache.
- Gives rise to conflict misses if we keep using say memory address 0000, 1000, 0000, 1000.
- But its quick and easy to implement (the quick is often the enemy of the good).

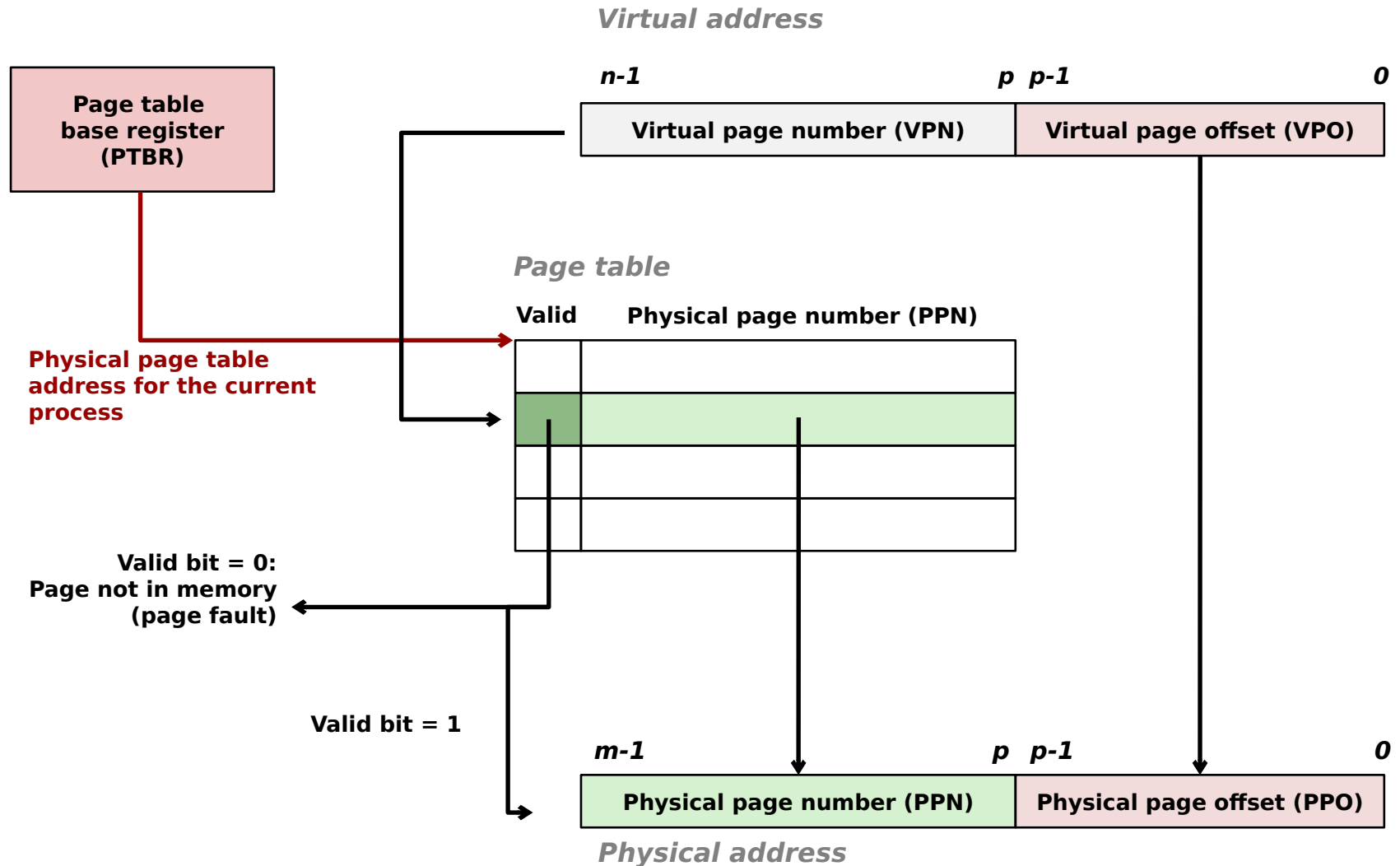
# Memory Mapping



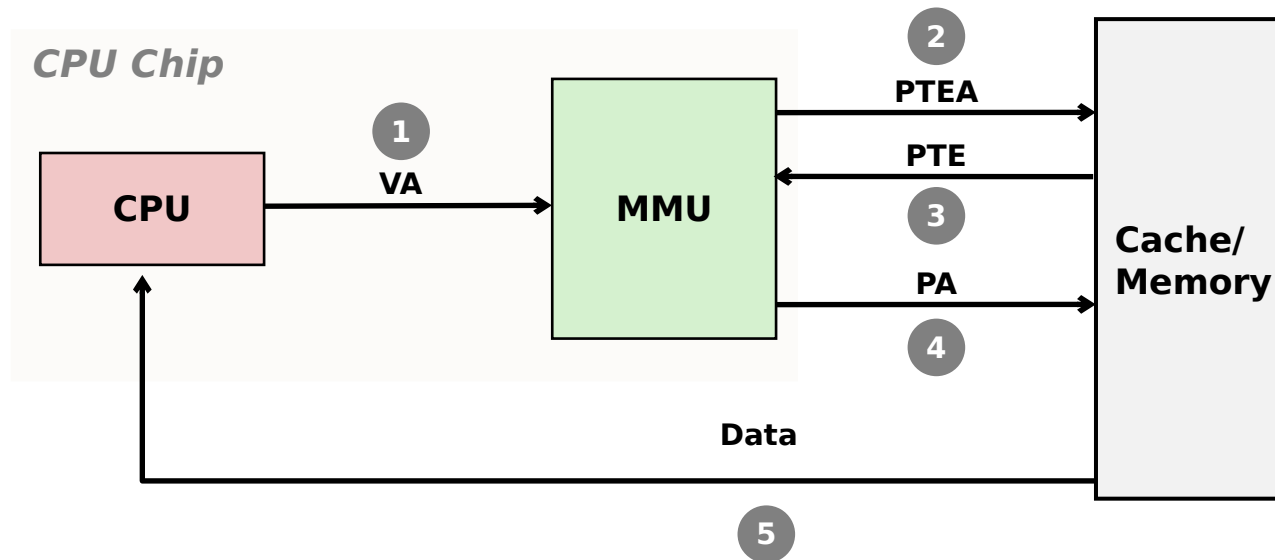
## Set-Associate Mapping

- We always store more than a single byte at a time in practice.
- Typically 4-32 KiBs, though this can differ by hardware.
- If we only stored single bytes, then spatial locality would mean nothing...
- Most examples present as though only a single byte for space/simplicity.
- Note the distinction between the memory address and the block address.

# Address Translation With a Page Table

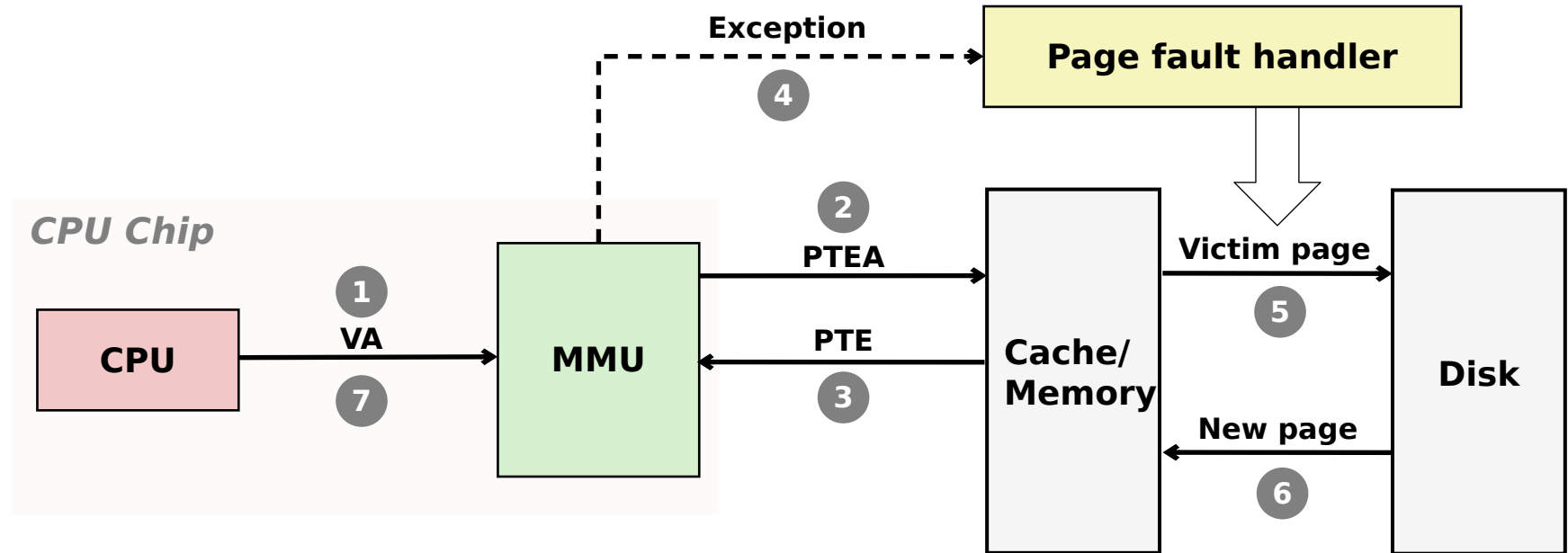


# Address Translation: Page Hit



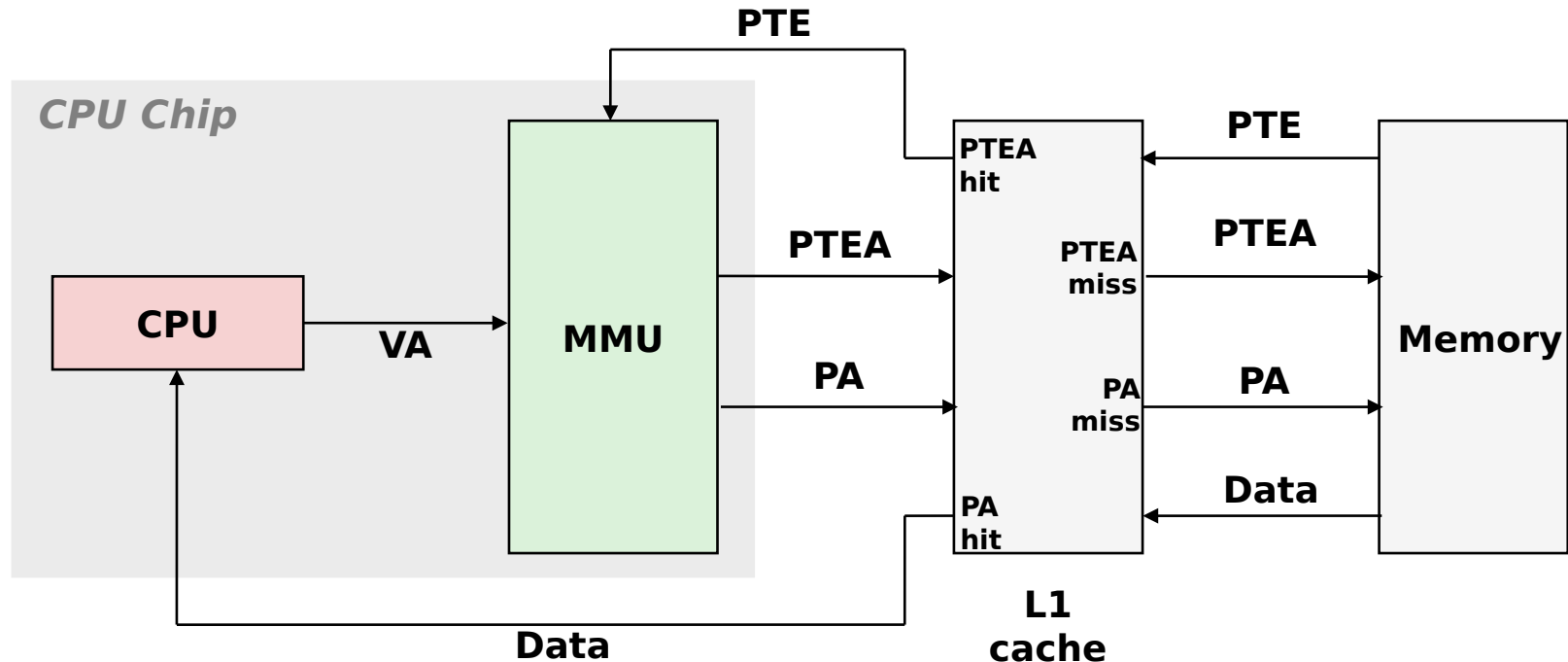
- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

# Address Translation: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in 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

# Integrating VM and Cache



**VA: virtual address**  
**PA: physical address**  
**PTE: page table entry**  
**PTEA = PTE address**

# Speeding up Translation with a TLB

- **Page table entries (PTEs) are cached in L1 like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

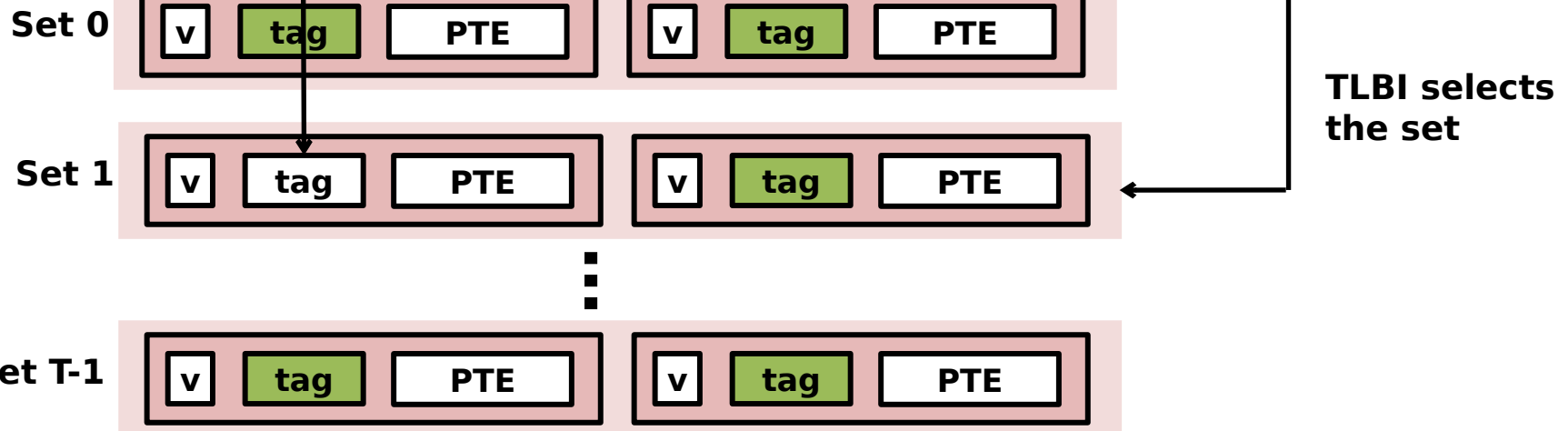
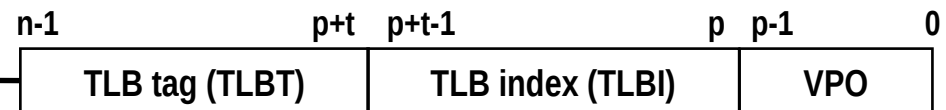
# Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

$T = 2^t$   
sets

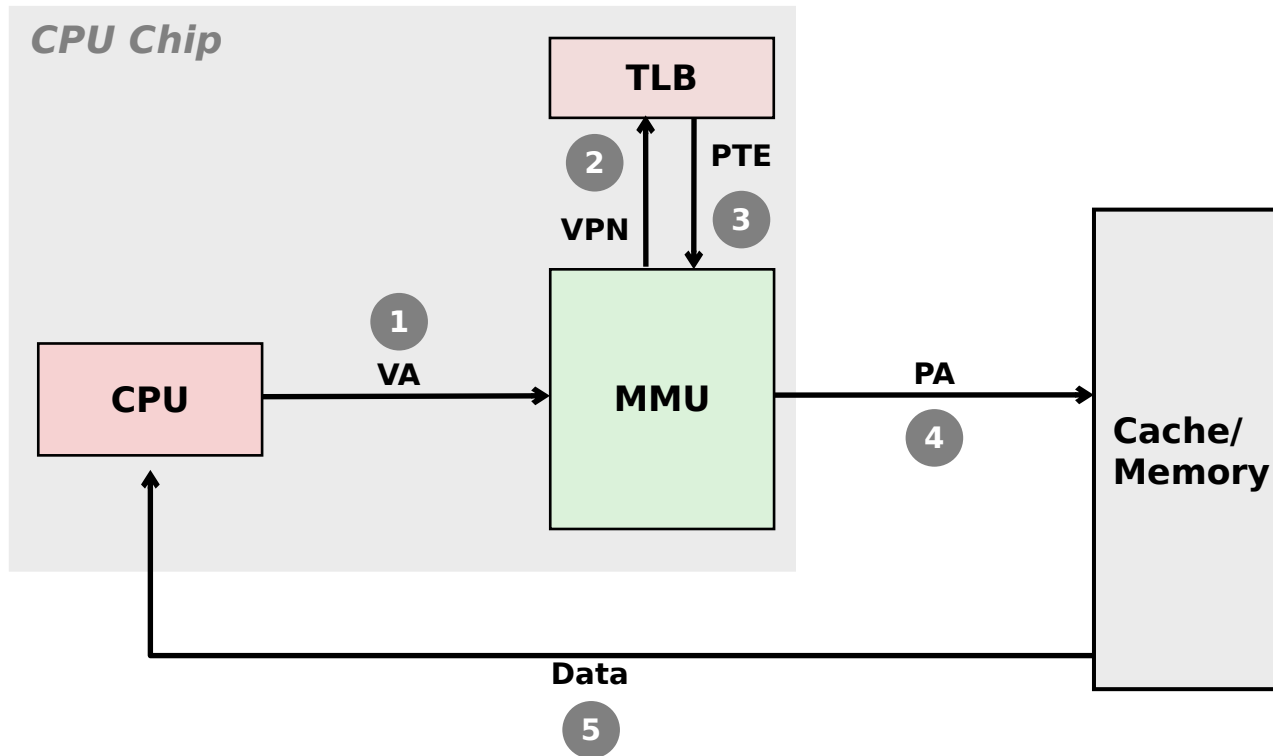
TLBT matches tag  
of line within set

VPN



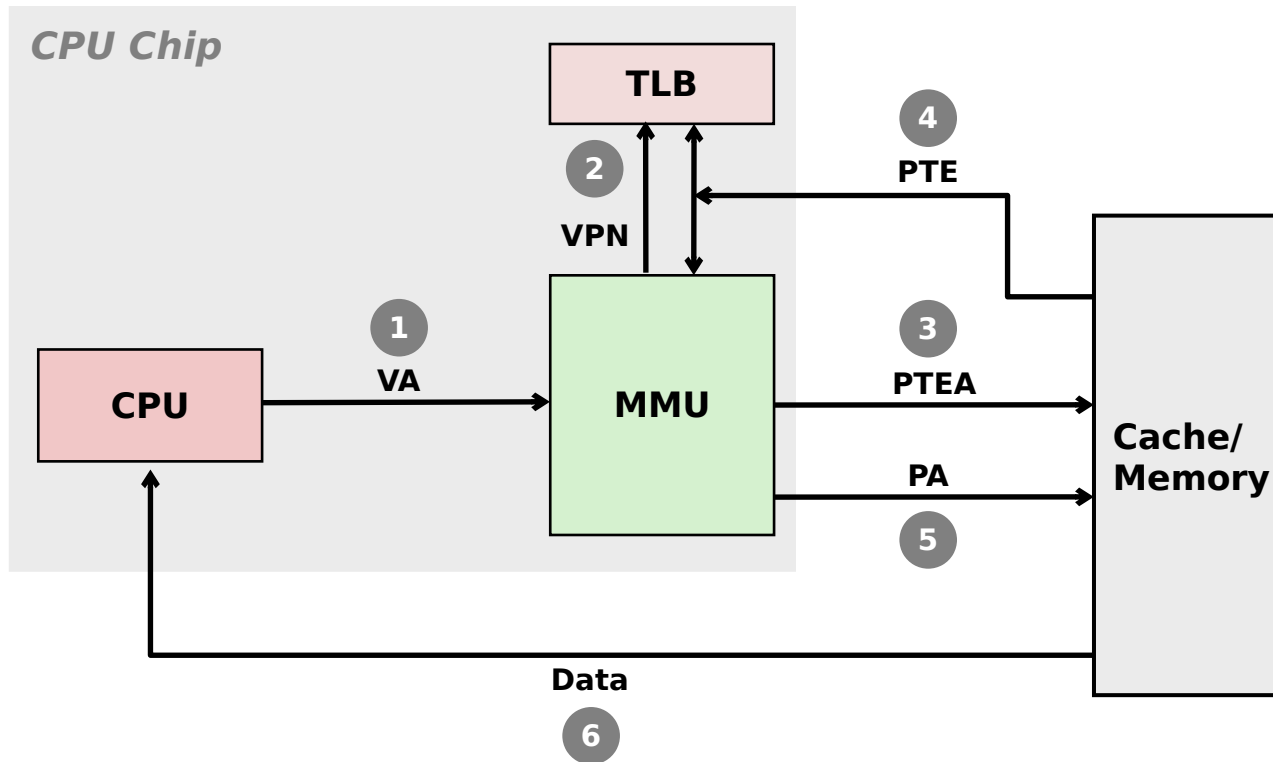


# TLB Hit



**A TLB hit eliminates a memory access**

# TLB Miss



**A TLB miss incurs an additional memory access (the PTE)**

Fortunately, TLB misses are rare. Why?

# Multi-Level Page Tables

## ■ Suppose:

- 4KB ( $2^{12}$ ) page size, 48-bit address space, 8-byte PTE

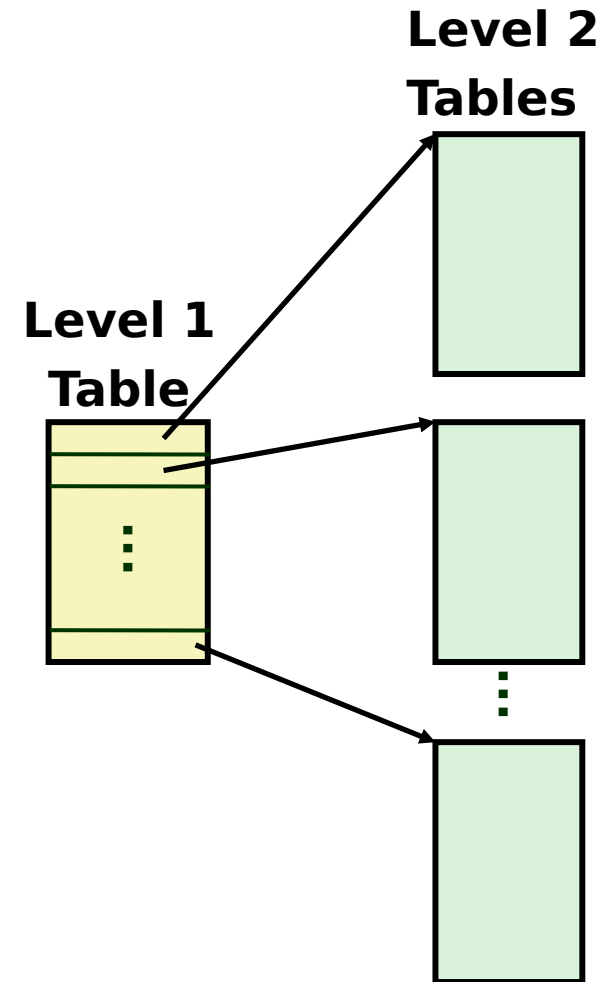
## ■ Problem:

- Would need a 512 GB page table!
  - $2^{48} * 2^{-12} * 2^3 = 2^{39}$  bytes

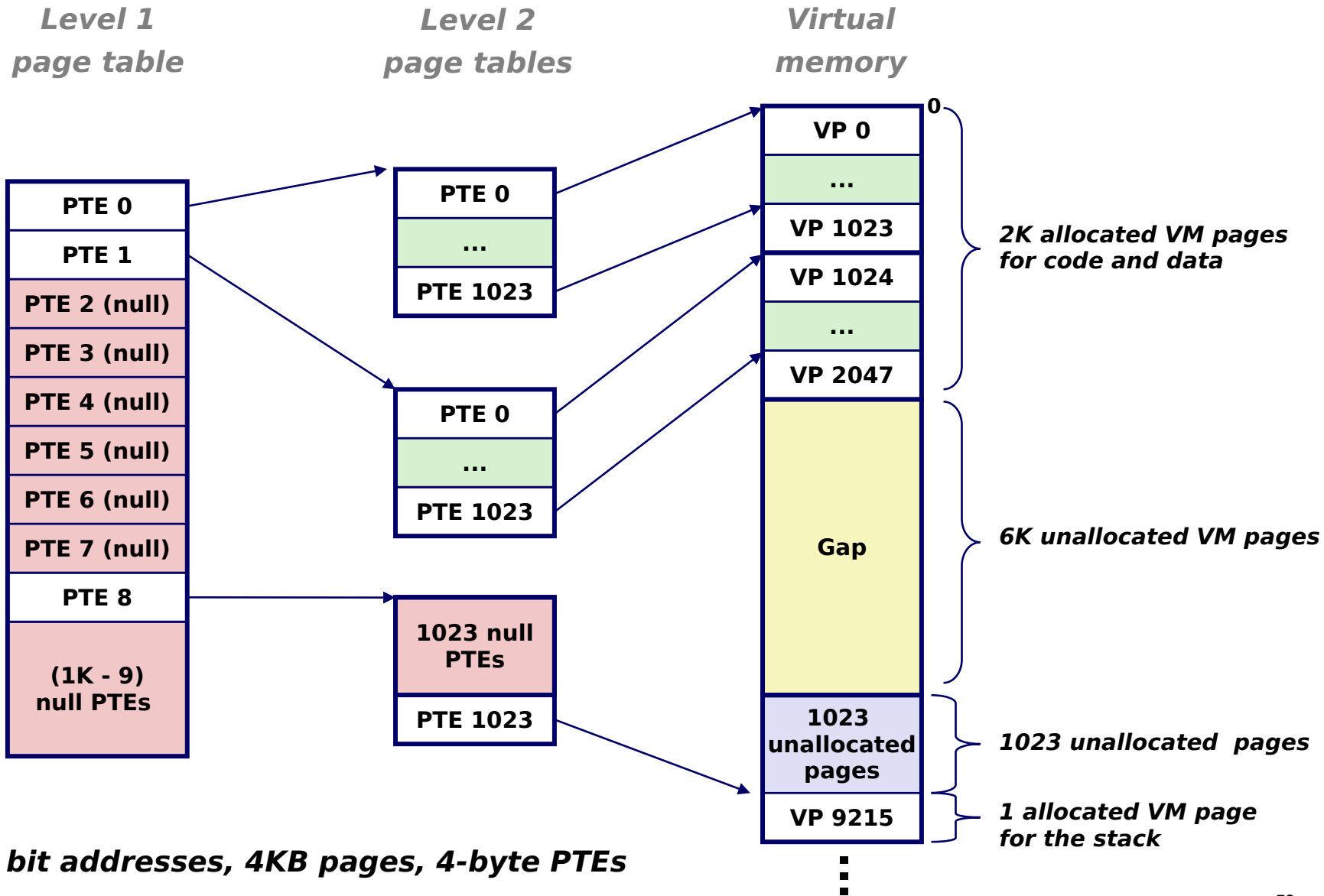
## ■ Common solution: Multi-level page table

## ■ Example: 2-level page table

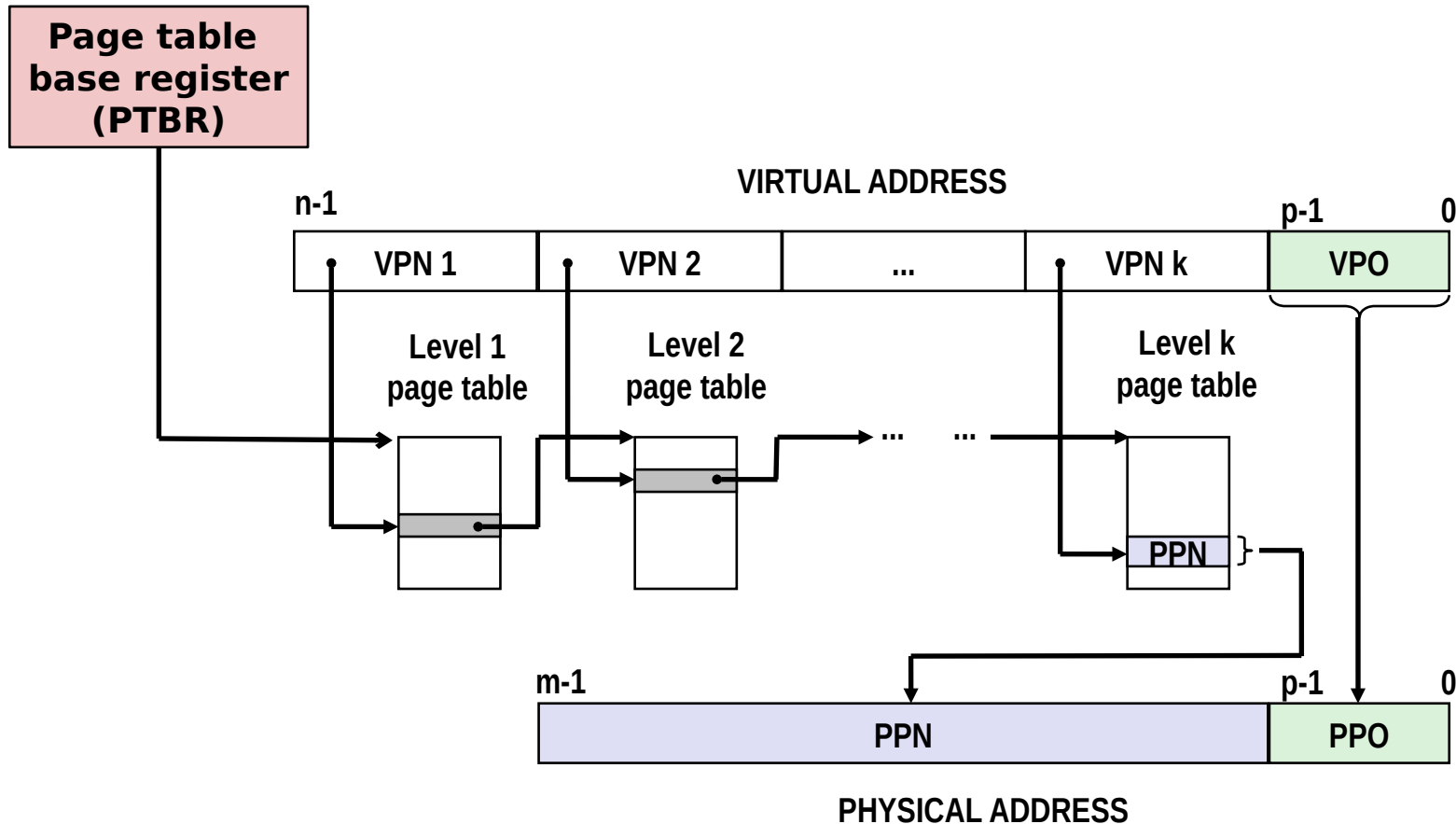
- Level 1 table: each PTE points to a page table (always memory resident)
- Level 2 table: each PTE points to a page (paged in and out like any other data)



# A Two-Level Page Table Hierarchy



# Translating with a k-level Page Table



# 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 programming
- Simplifies protection by providing a convenient interpositioning point to check permissions