



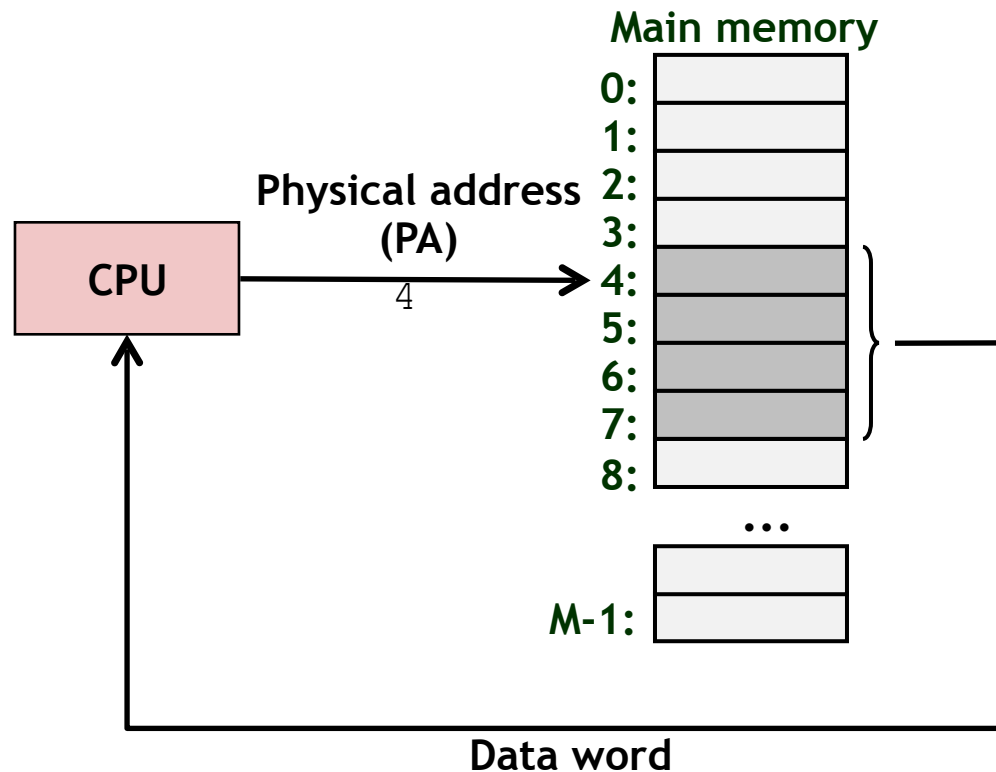
+

Virtual Memory

# + A System Using Physical Addressing

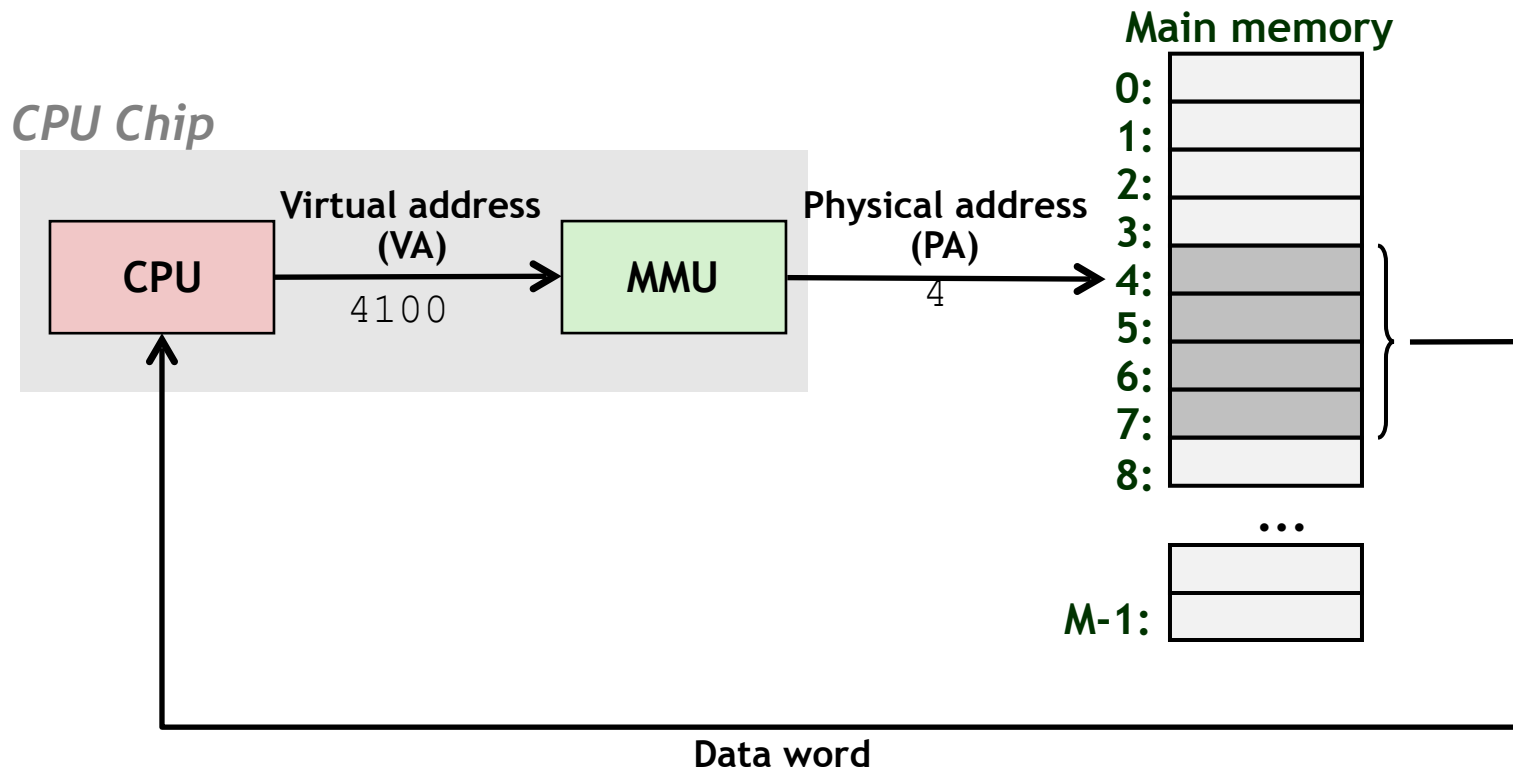


- Used in “simple” systems like embedded microcontrollers in devices like elevators and digital picture frames

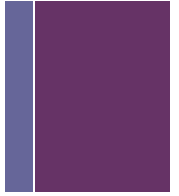


# + A System Using Virtual Addressing

- Used in all modern servers, laptops, and smart phones



# + Address Spaces



- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

$\{ 0, 1, 2, 3 \dots \}$

- **Physical address space:** Set of  $M = 2^m$  physical, linear addresses

$\{ 0, 1, 2, 3, \dots, M-1 \}$

- **Virtual address space:** Set of  $N = 2^n$  virtual, linear addresses

$\{ 0, 1, 2, 3, \dots, N-1 \}$

# + Why Virtual Memory (VM)?



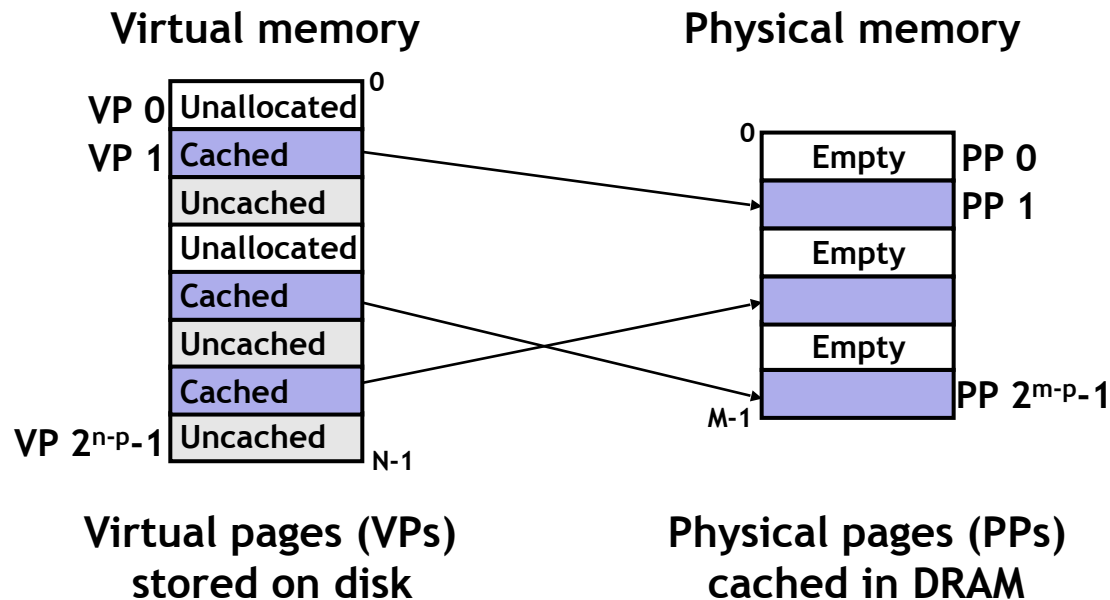
- **Uses main memory efficiently**
  - Use DRAM as a cache for parts of a virtual address space
- **Simplifies memory management**
  - Each process gets the same uniform linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code



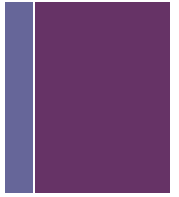
# + VM as a Tool for Caching



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



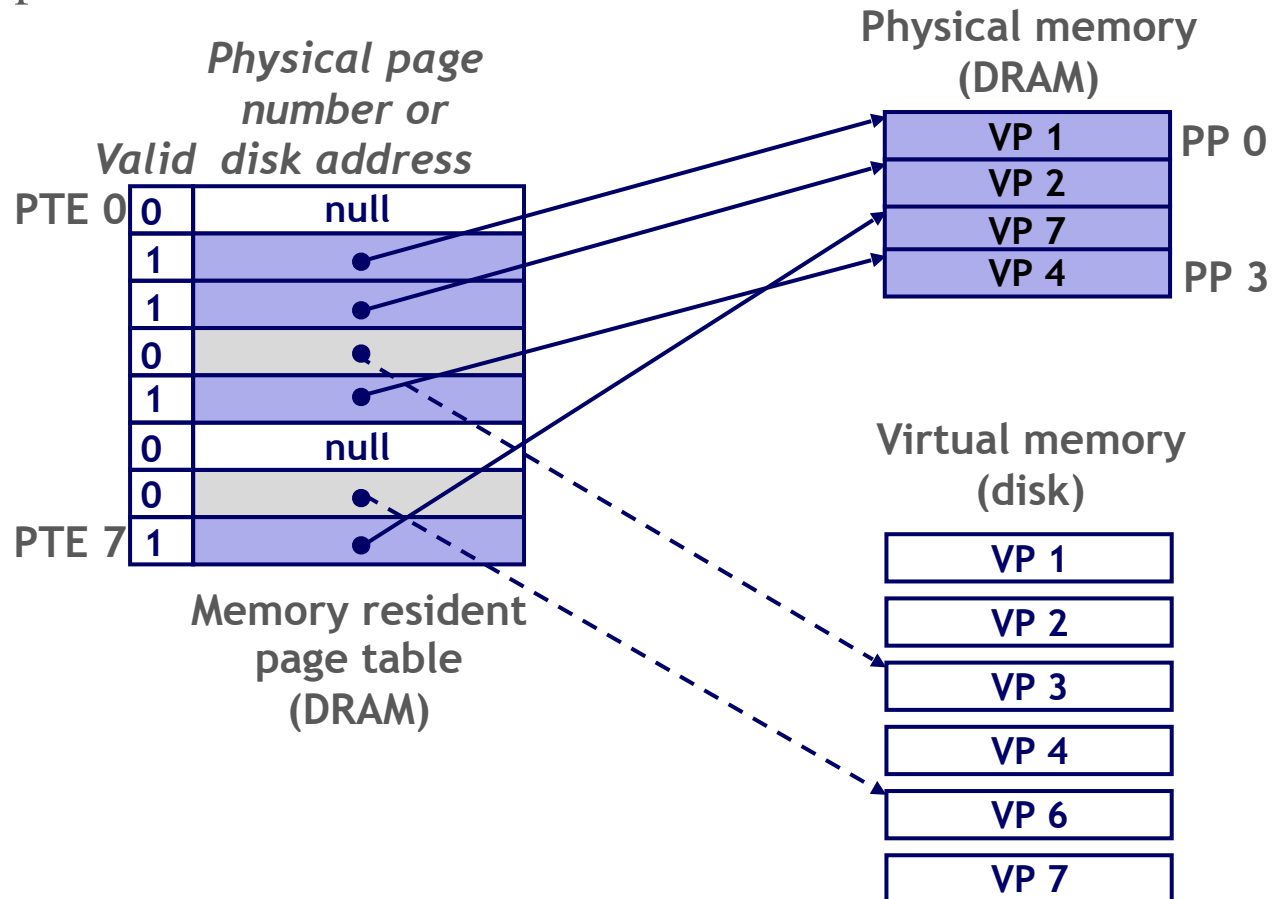
# + DRAM Cache Organization



- **DRAM cache organization driven by the enormous miss penalty**
  - DRAM is about **10x** slower than SRAM
  - 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 complex mapping function
  - Highly sophisticated, expensive replacement algorithms
    - Still faster than going to disk, so worth it.
  - *Write-back* rather than *write-through*

# + Enabling Data Structure: Page Table

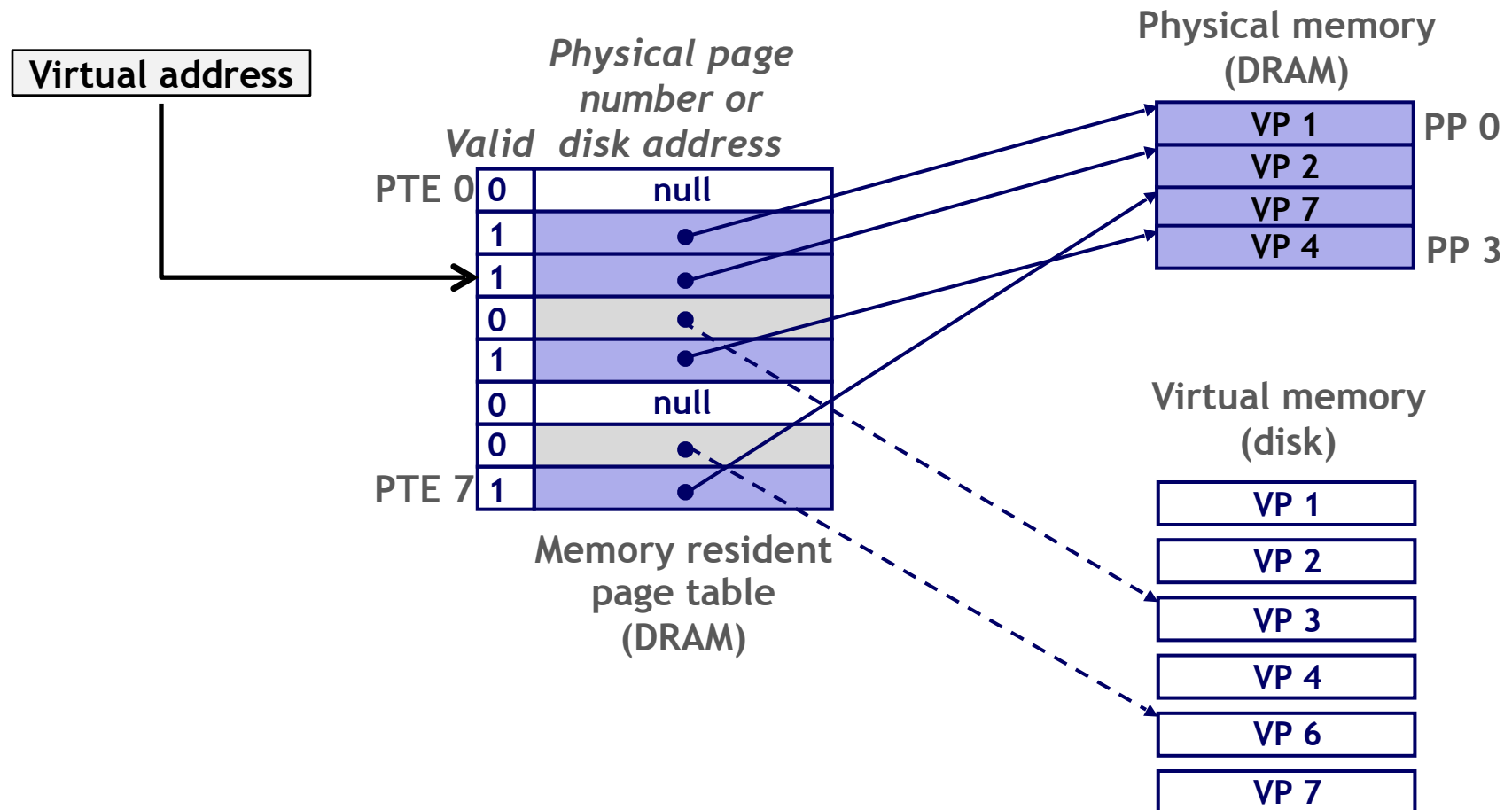
- *Page table*: an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM





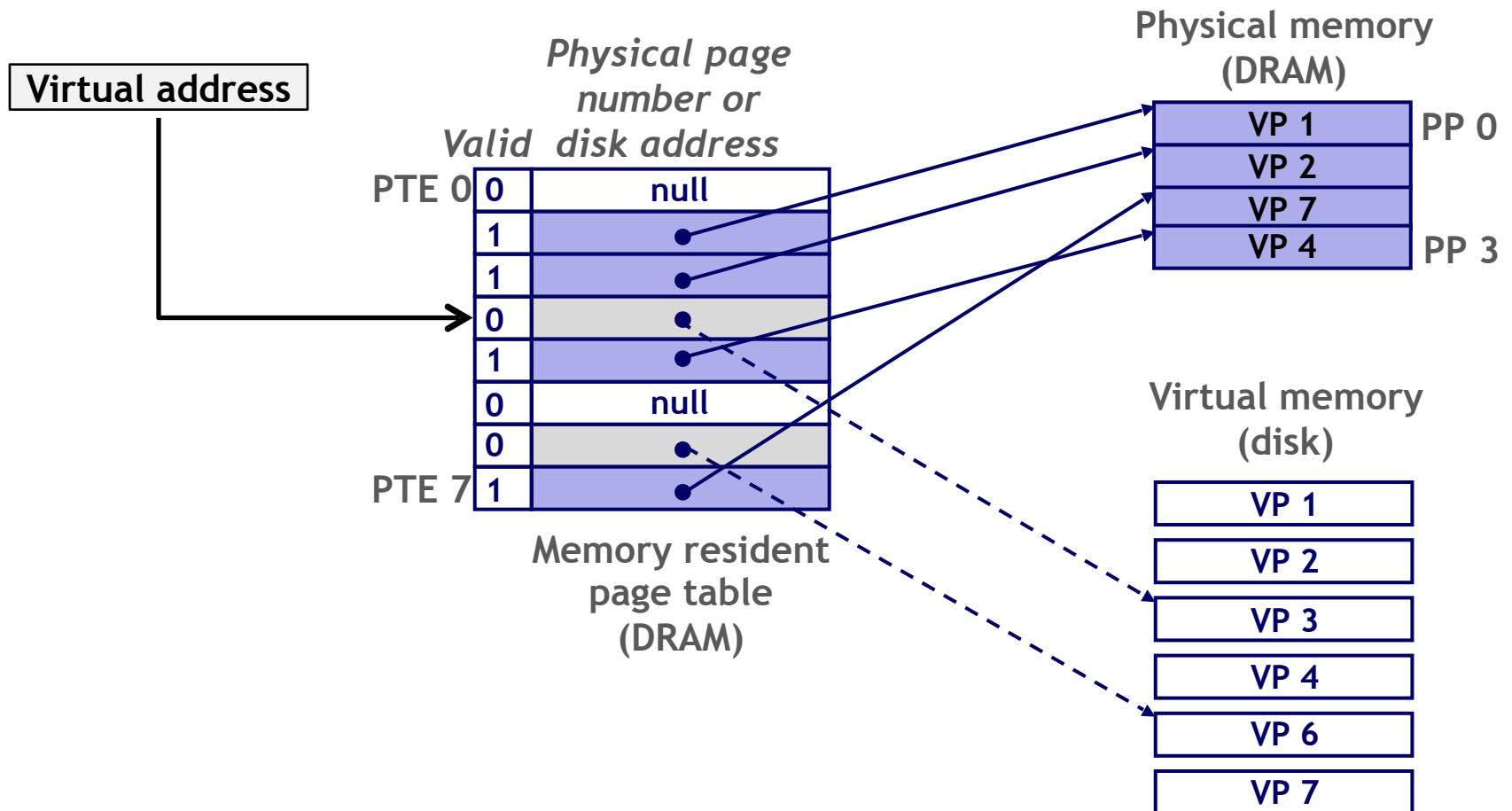
# + Page Hit

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



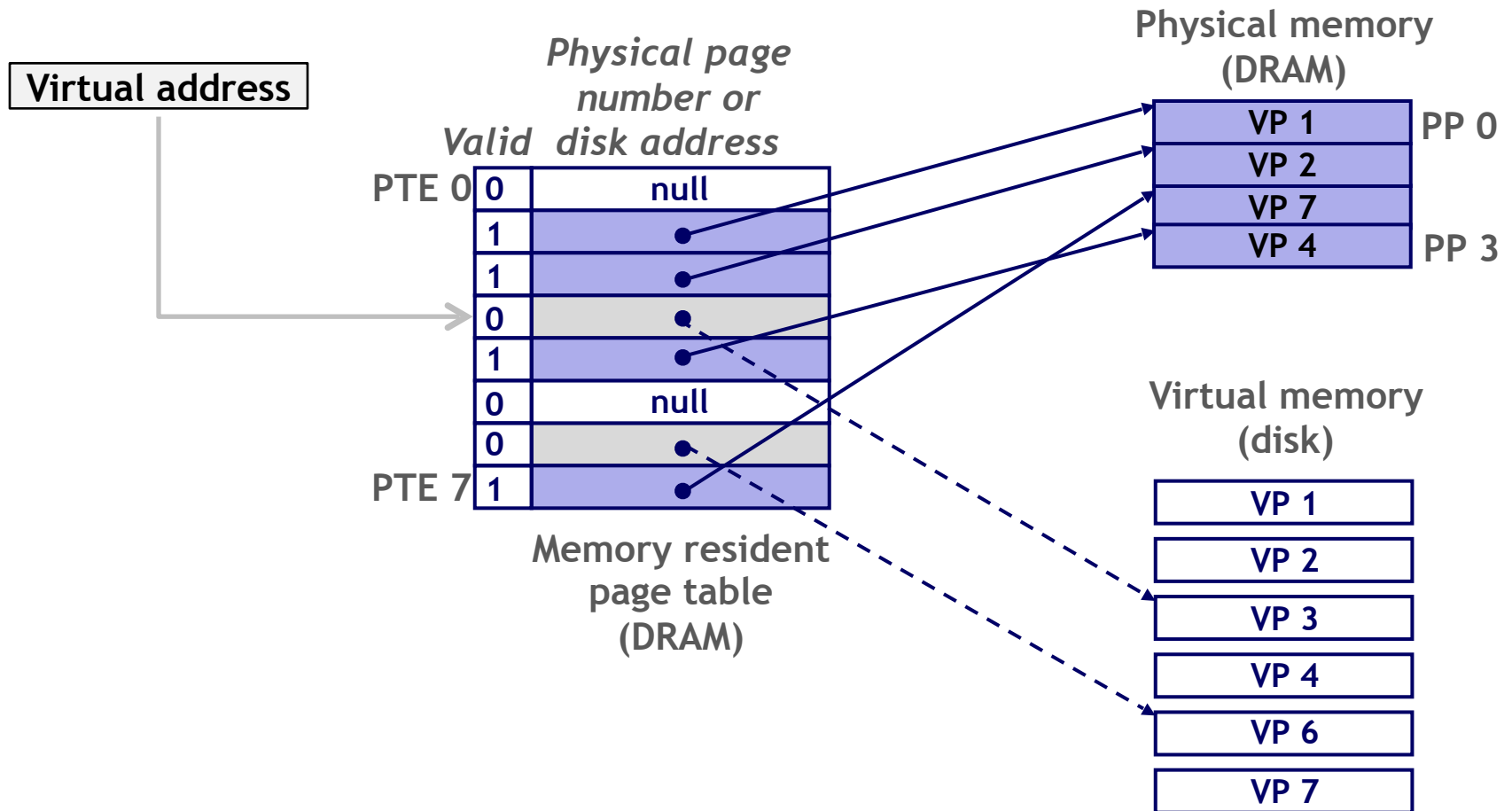
# + Page Faults

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



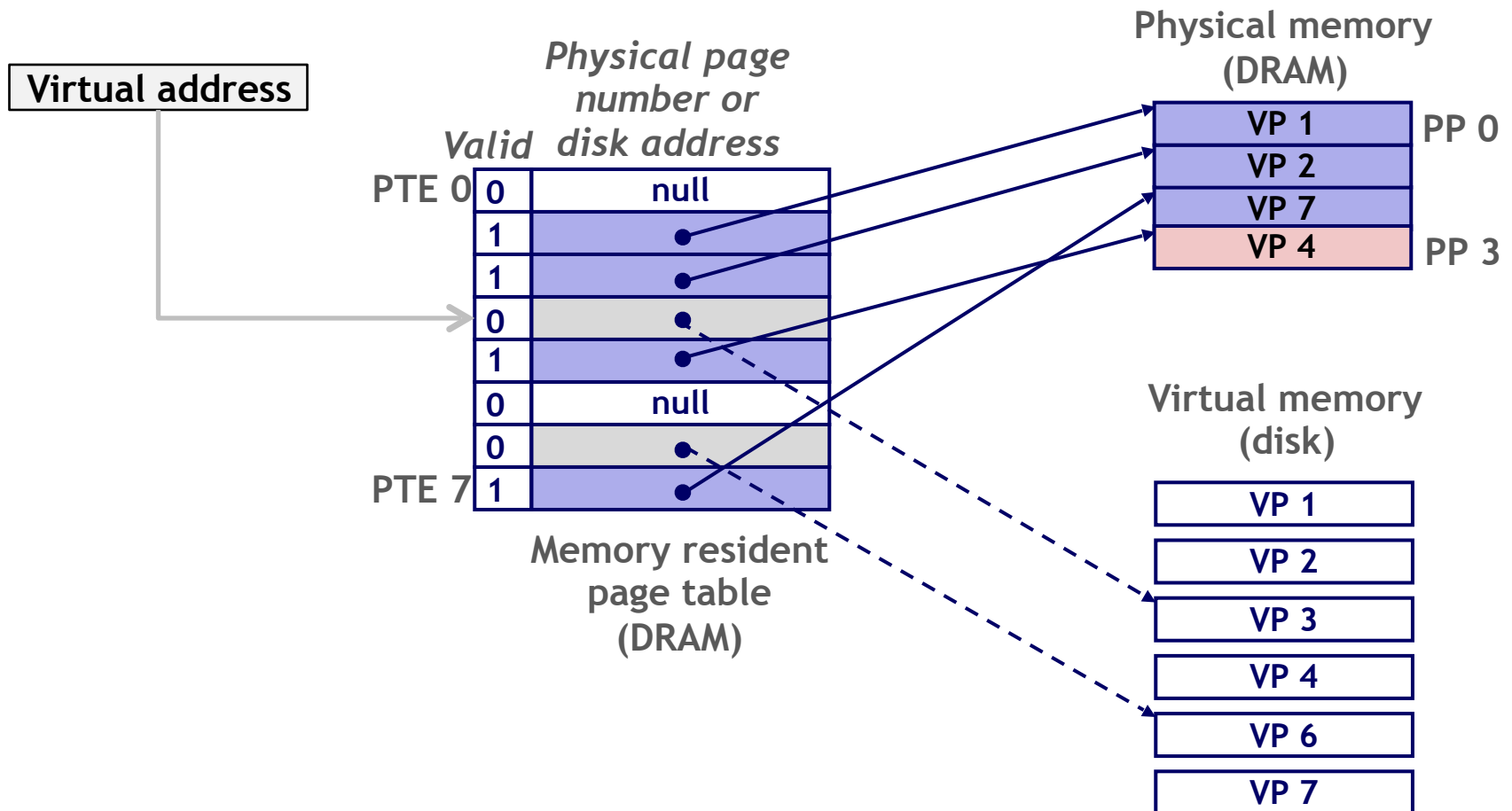
# + Page Faults *con't*

- Page miss causes page fault (an exception)



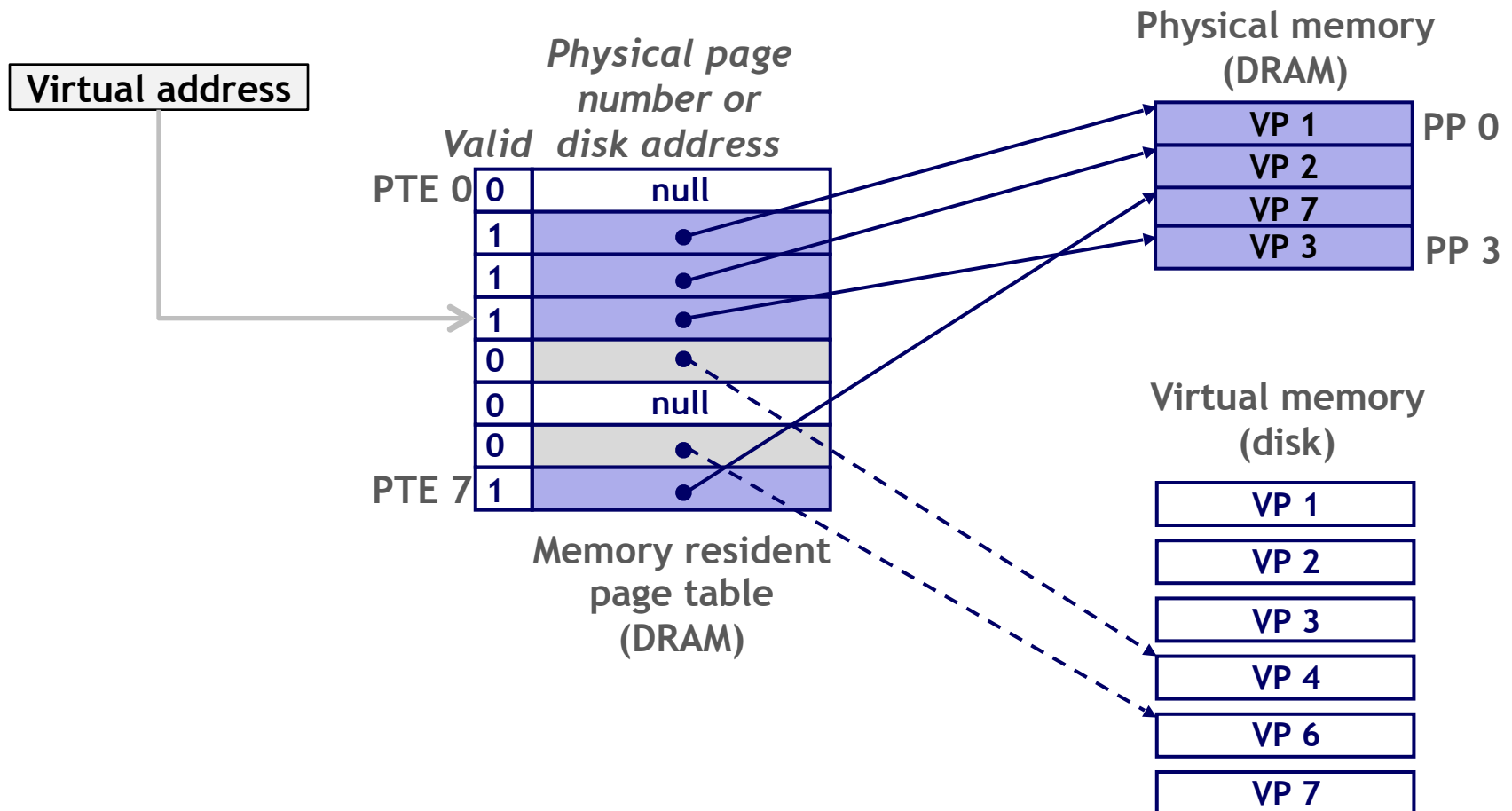
# + Page Faults *con't*

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



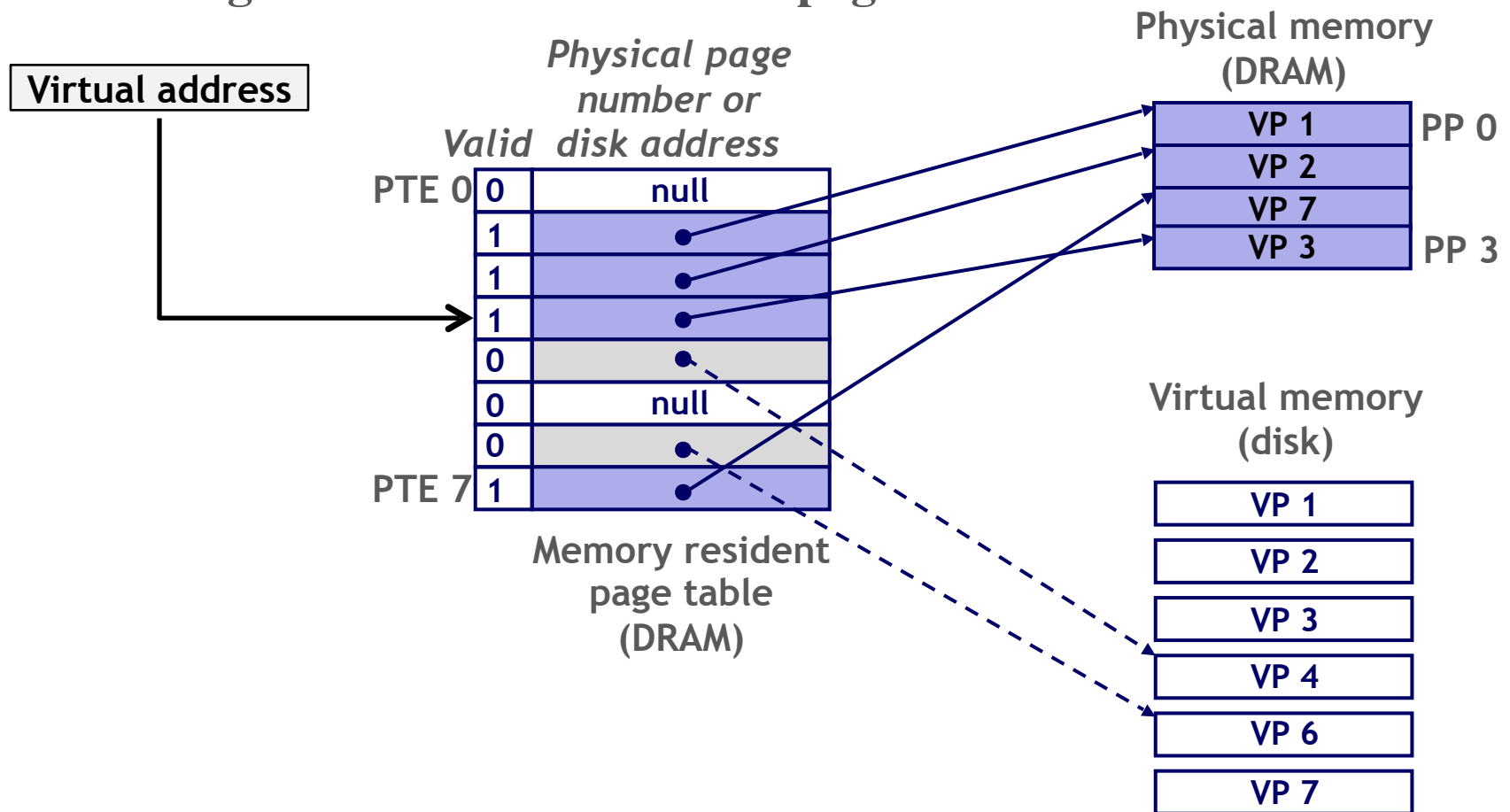
# + Page Faults *con't*

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



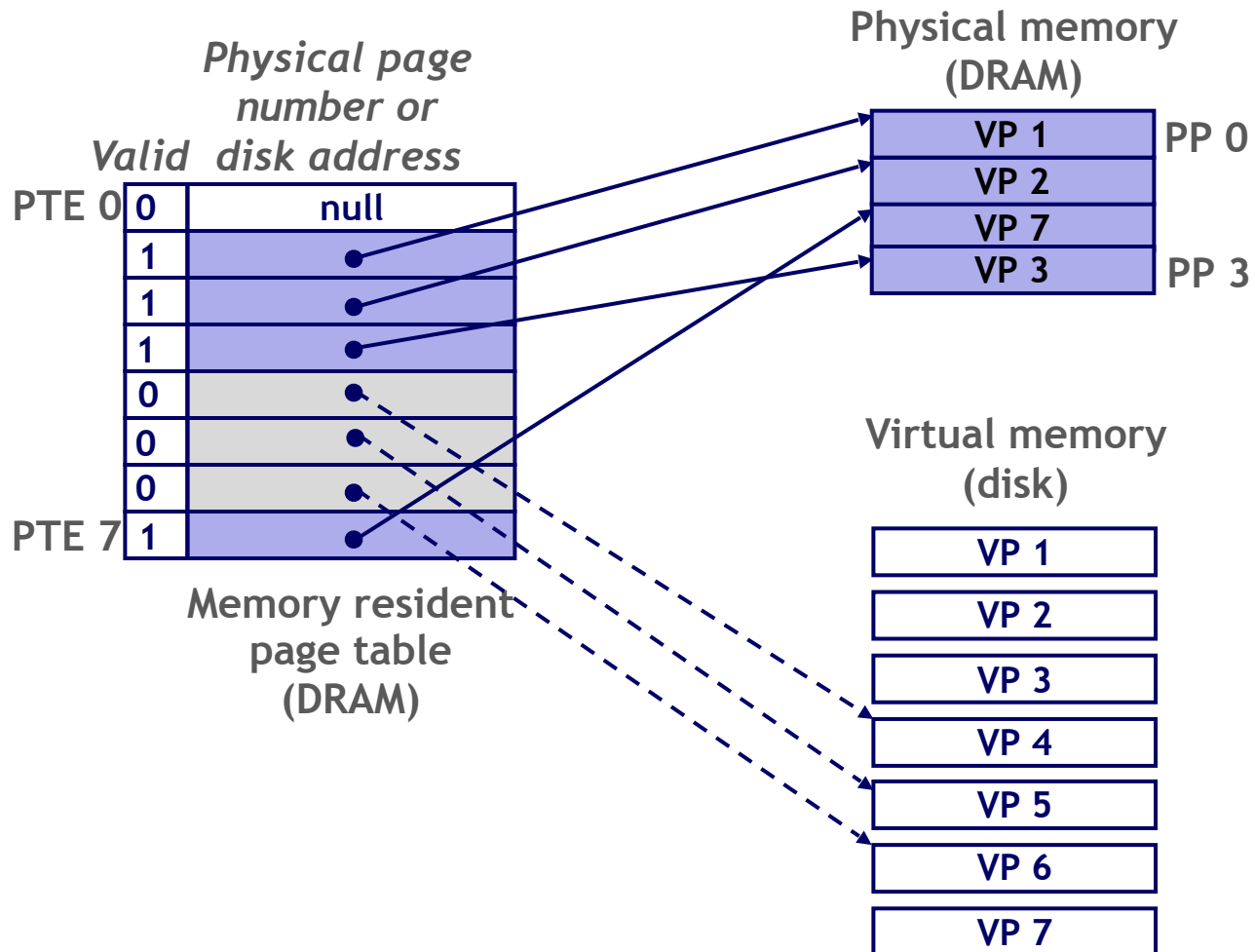
# + Page Faults *con't*

- 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!



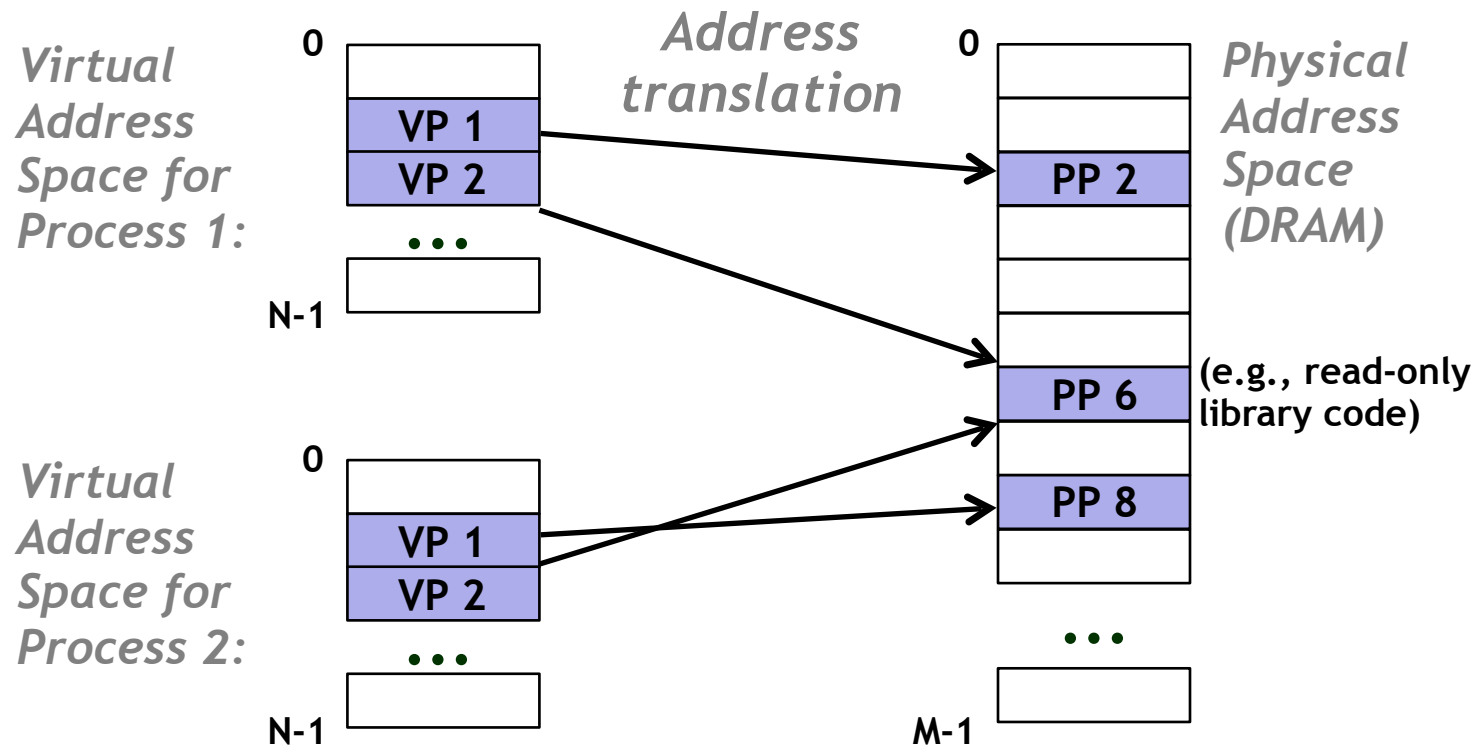
- Virtual memory seems inefficient, but it works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the *working set*
  - Programs with better temporal locality will have smaller working sets
- If (working set size < main memory size)
  - Good performance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



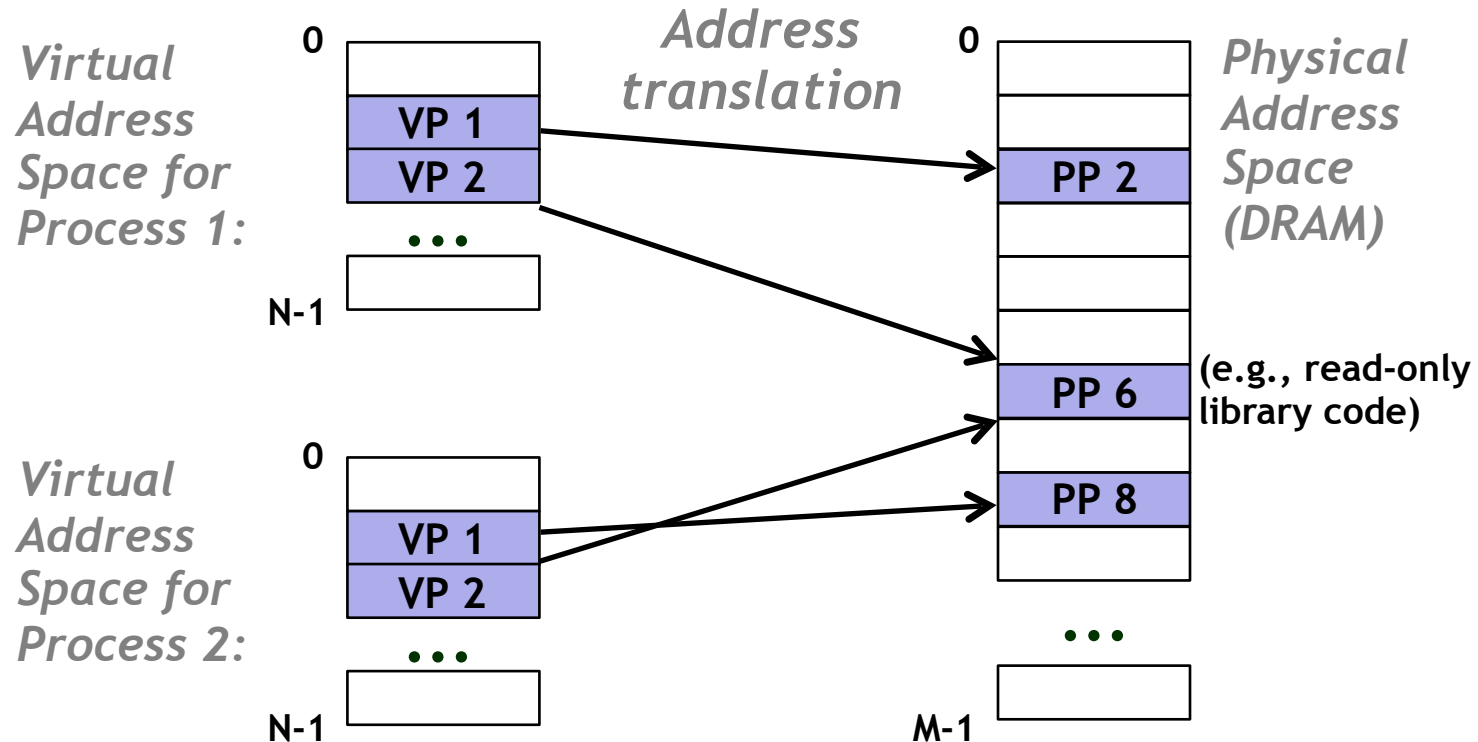
# + VM as a Tool for Memory Management

- **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**

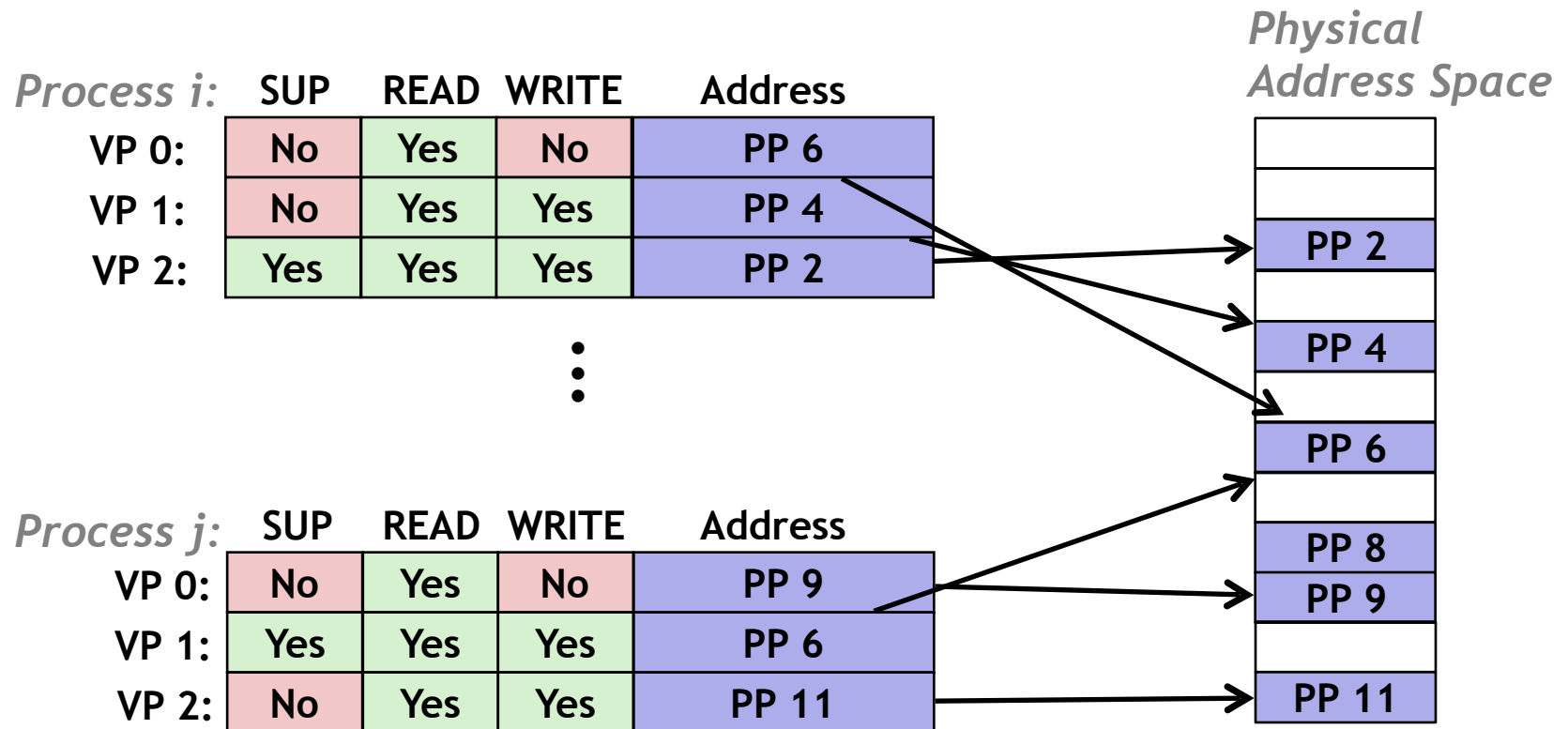
- Map virtual pages to the same physical page (here: PP 6)



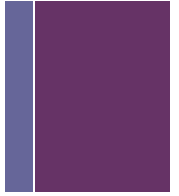
# + VM as a Tool for Memory Protection



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



# + Summary

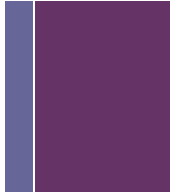


- **Programmer's view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes
- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient because of locality
  - Simplifies memory management and programming
  - Enables memory protection



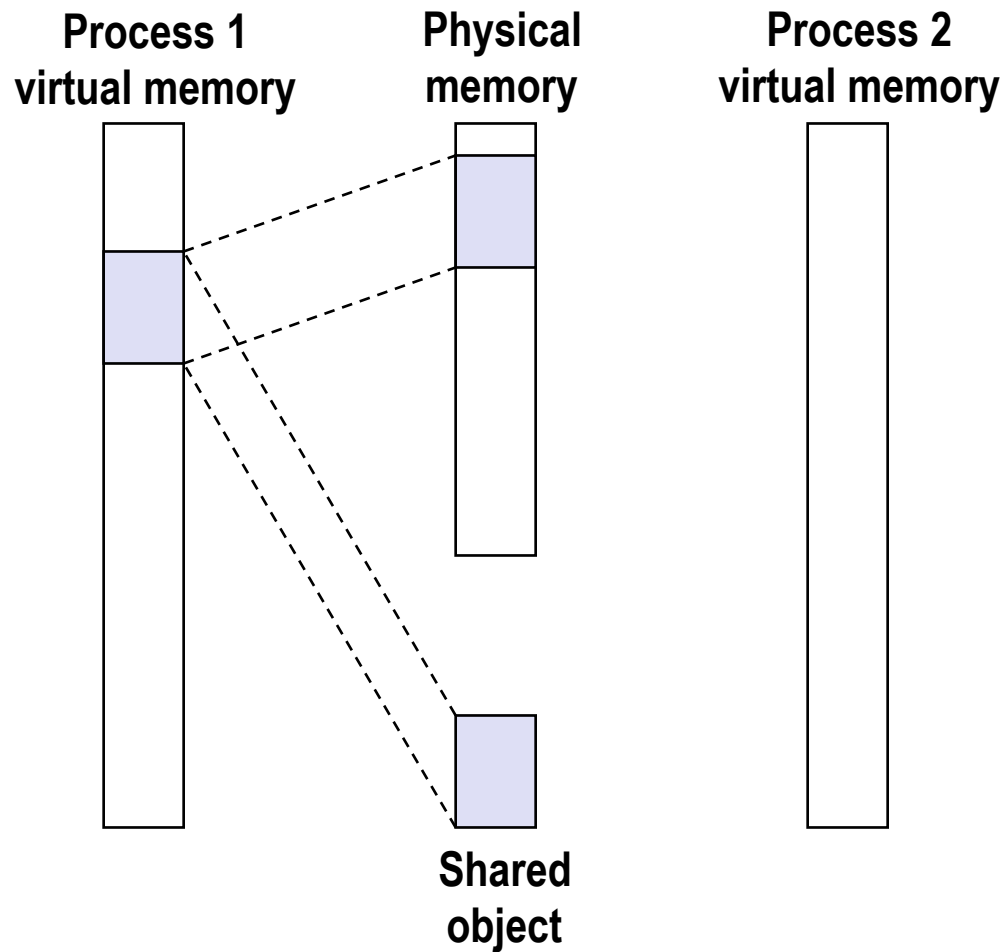
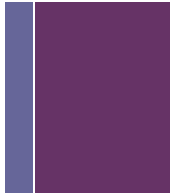
# Memory Mapping

# + Memory Mapping



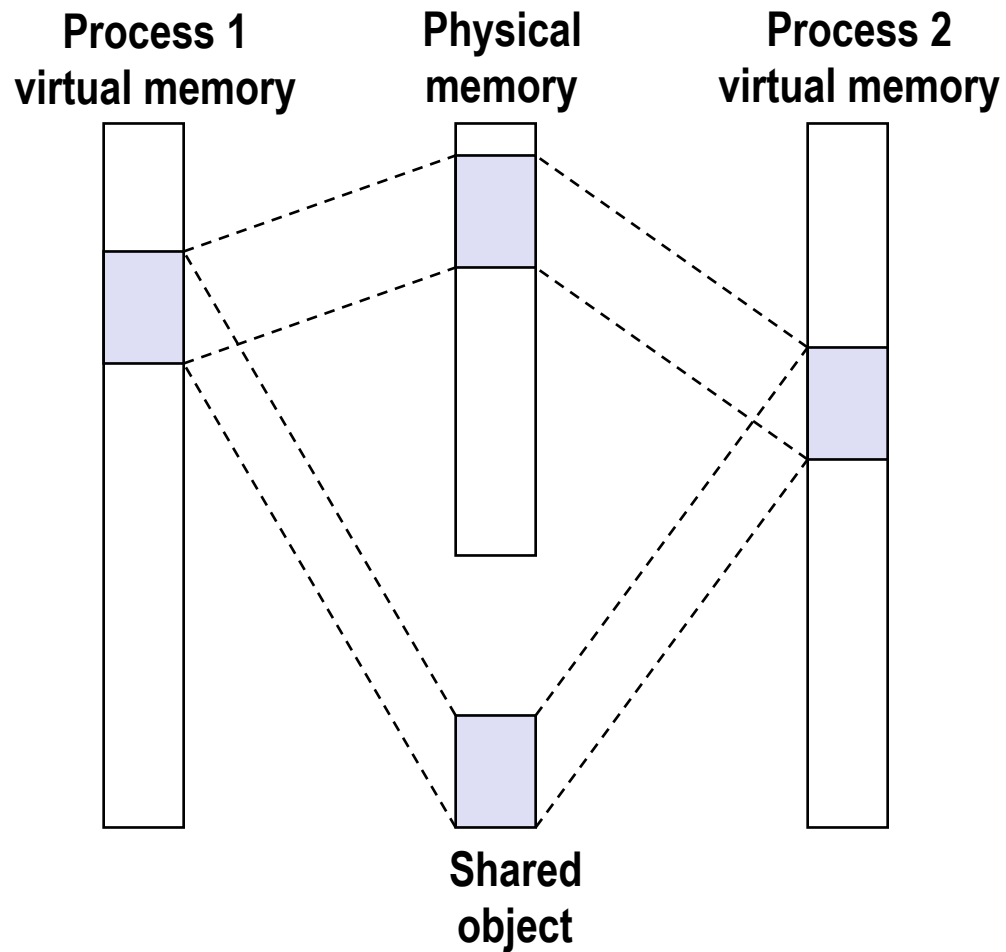
- **VM pages initialized by associating them with disk objects.**
  - Process is known as *memory mapping*
- **Sets of pages can be backed by (i.e., get its initial values from) :**
  - Regular file on disk, like an executable
  - Anonymous file (*see book*)
- **These sets of pages are not put into physical memory until referenced by the CPU.**
- **Moreover, memory mapping is the way the virtual address space is built.**

# + Shared Objects



- **Process 1** maps the shared object.

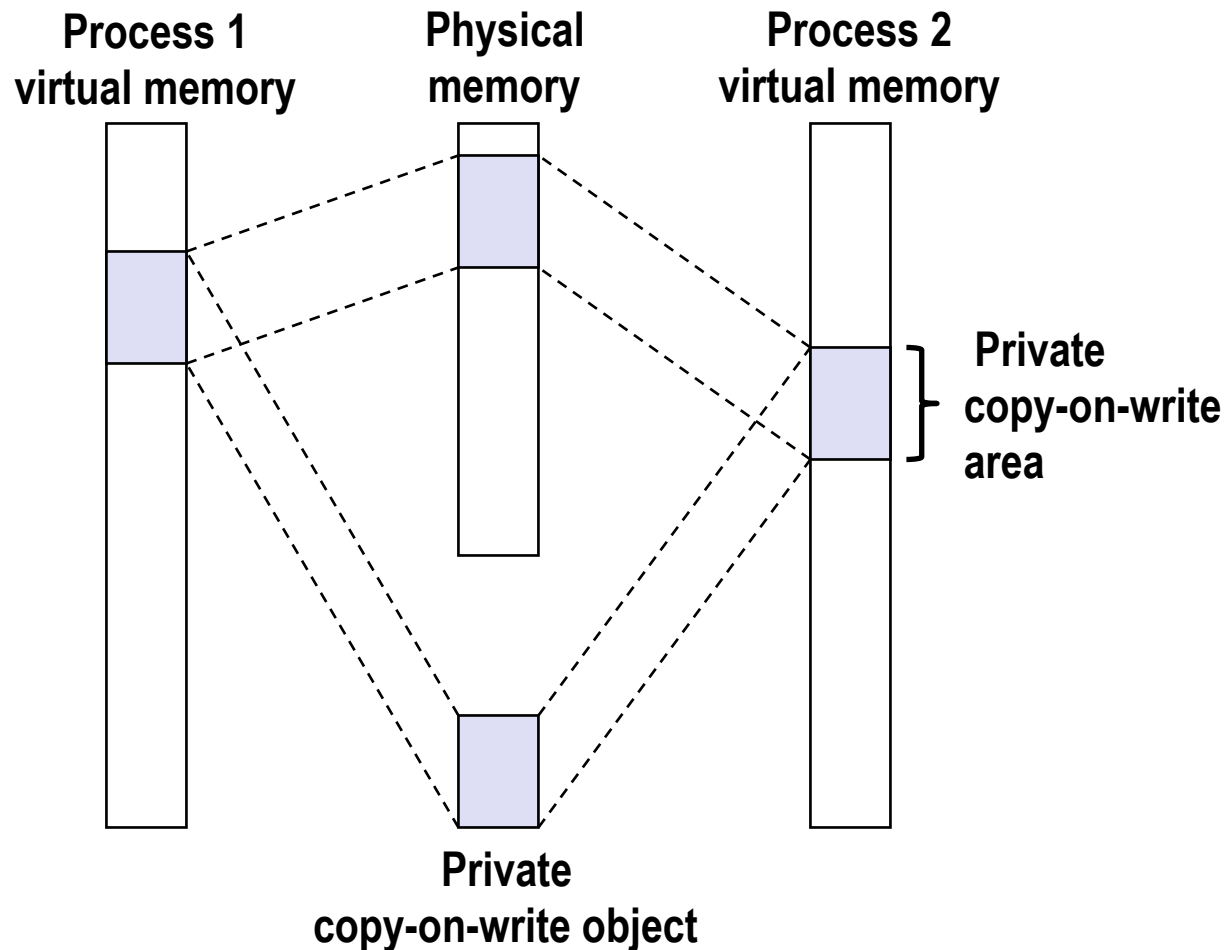
# + Shared Objects



- **Process 2 maps the shared object.**
- **Notice how the virtual addresses can be different**

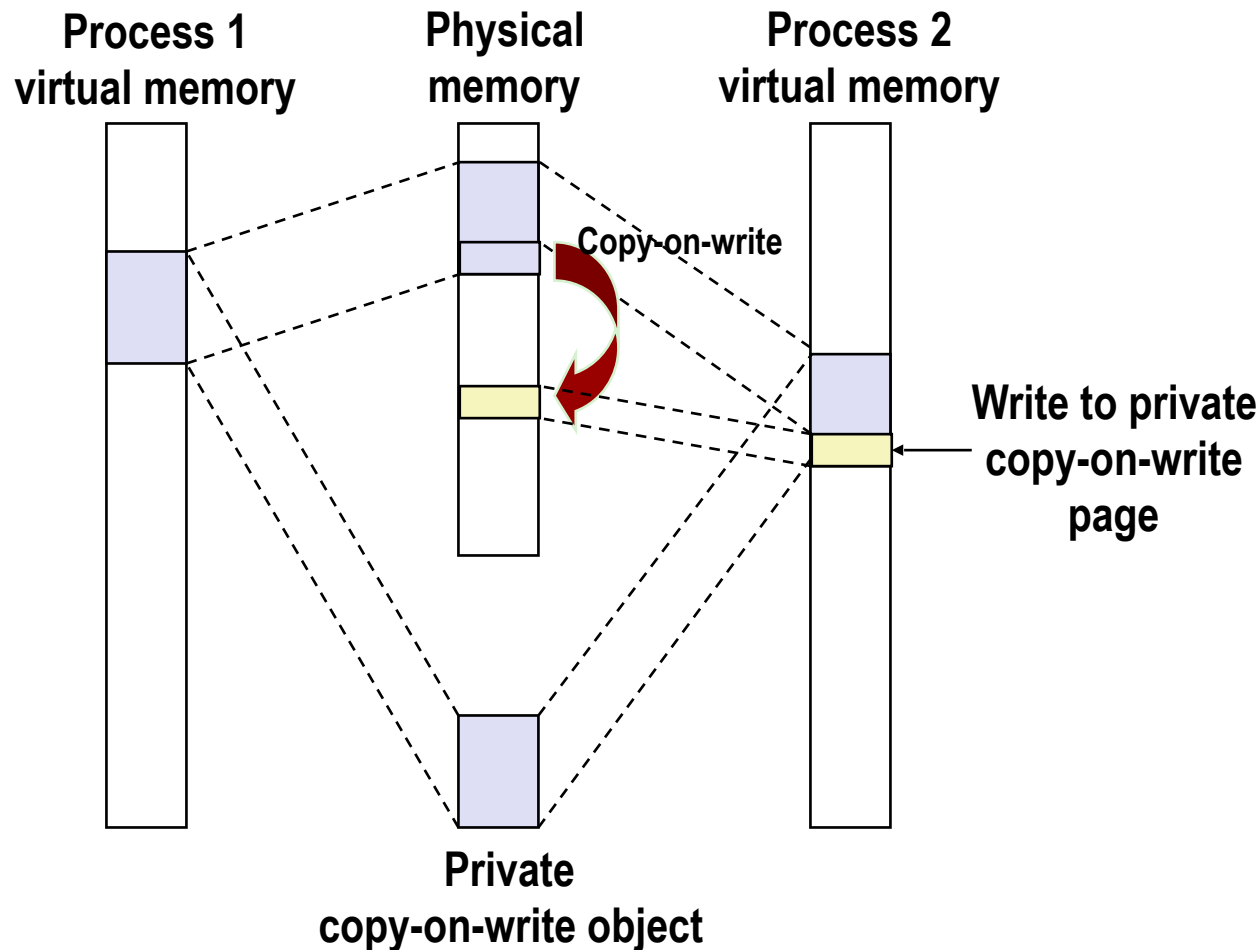


# + Private Copy-on-write (COW) Objects



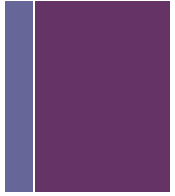
- Two processes mapping a *private copy-on-write (COW) object*.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

# + Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Updates page table for process
- Instruction restarts upon handler return.

# + The `fork` Function Revisited



- VM and memory mapping explain how `fork` provides private address space for each process.
- To create virtual address for new new process
  - Create exact copy of virtual memory address space
  - Flag virtual memory context in both processes as COW
- On return, each process has exact copy of virtual memory
- Subsequent writes create *new pages* using COW mechanism.
- If it were not for this, `fork ()` would be very, very expensive!