Virtual Memory

Computer Systems Principles

Objectives

- **Virtual Memory**
  - What is it?
  - How does it work?
- **Virtual Memory**
  - Address Translation

Problem

- **Lots of executing programs**
  - Take up lots of space!
- **Memory is comparatively small**
  - Not enough room!
- **Solution:**
  - Pretend we have lots of memory
  - Use disk to store programs & data
  - Use RAM as a cache!

Overview

- **Virtual Memory**
  - An elegant interaction of:
    - hardware exceptions
    - hardware address translation
    - main memory
    - disk files
    - kernel

Overview

- **Provides:**
  - A process with a **large**, **uniform**, and **private** address space.
- **Three Important Capabilities**
  1. Main/real memory is **cache**.
  2. **Uniform address space**.
  3. **Protects** process address space

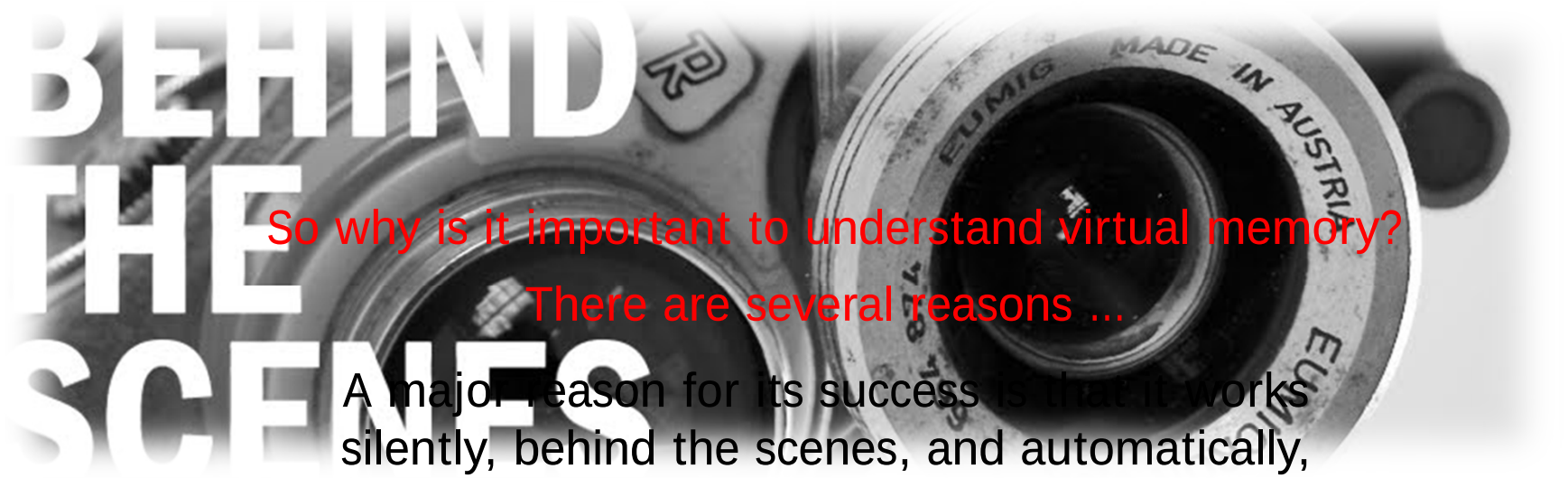
Overview

A major reason for its success is that it works silently, behind the scenes, and automatically, without intervention by the programmer

Overview

So why is it important to understand virtual memory?

A major reason for its success is that it works silently, behind the scenes, and automatically, without intervention by the programmer



So why is it important to understand virtual memory?

There are several reasons ...

A major reason for its success is that it works silently, behind the scenes, and automatically, without intervention by the programmer

Basic Idea

- **Main Memory**
  - Organized as an array of M contiguous byte-sized cells.
  - Each byte has a unique *physical address* (PA)
  - First byte has address 0, next 1, next 2, …
- **Natural Access**
  - Access main memory using PA
  - This is called *physical addressing*
    - Sometimes also called *real addresses, real addressing, real memory*

Physical Addressing

- **CPU executes load instruction at address 4**

Physical Addressing

- **CPU executes load instruction at address 4**
  - Generates a physical address

Physical Addressing

- **CPU executes load instruction at address 4**
  - Generates a physical address
  - Passes it to main memory over the memory bus

Physical Addressing

- **CPU executes load instruction at address 4**
  - Generates a physical address
  - Passes it to main memory over the memory bus
  - Main memory fetches 4-byte word at physical address 4 and returns it to the CPU

Physical Addressing

- **CPU executes load instruction at address 4**
  - Generates a physical address
  - Passes it to main memory over the memory bus
  - Main memory fetches 4-byte word at physical address 4 and returns it to the CPU
  - The CPU stores it in a register for quick access

Physical Addressing

- **CPU executes load instruction at address 4**
  - Generates a physical address
  - Passes it to main memory over the memory bus
  - Main memory fetches 4-byte word at physical address 4 and returns it to the CPU
  - The CPU stores it in a register for quick access
- **Early PCs used physical addressing**
- **Microcontrollers & Cray super computers**
  - Still use physical addressing

Virtual Addressing

- **With virtual addressing**
  - CPU accesses main memory by generating a *virtual address* (VA)
  - The VA is converted into a corresponding PA **before** it is sent to main memory

Virtual Addressing

- **With virtual addressing**
  - CPU accesses main memory by generating a *virtual address* (VA)
  - The VA is converted into a corresponding PA **before** it is sent to main memory
- **Virtual Address -> Physical Address**
  - Is known as *address translation*
  - Requires close cooperation between CPU and OS
  - Dedicated Hardware: *Memory Management Unit (MMU)*

VM Organization

- **VM organized as**
  - Array of N contiguous byte-size cells stored on disk
  - Each byte has a unique address to index array
  - Contents of array are cached in main memory
- **Data on Disk**
  - Represents “memory” of entire program.
  - Parts of program that are being actively used are in physical/real/main memory.

VM Partitioning

- **Virtual Memory Partitioning**
  - Partitioned into fixed-sized "blocks" called **virtual pages**
  - Each virtual page is $P = 2^p$ bytes in size
- **Physical Memory Partitioning**
  - Partitioned into fixed-sized "blocks" called **physical pages**
  - Each physical page is also P bytes in size
  - Physical pages may *reside on disk!*

VM Allocation

- **Allocation**
  - VM pages are allocated and cached
  - At any point in time, the set of virtual pages is partitioned into *three disjoint subsets*
- **Unallocated**
  - Pages that have not yet been allocated. No data associated with them, do not occupy space on disk or main memory
- **Cached (also: Resident, Swapped in)**
  - Allocated pages that are currently cached in physical memory
  - May or may not (yet) have a copy on disk
- **Uncached (also: Non-resident, Swapped out)**
  - Allocated pages that are not cached in physical memory

Page Tables

- **Where is my page?**
  - VM needs a way to find a page if it is cached, find it on disk if a miss, cache it, and handle eviction
  - Provided by OS, software, & *Memory Management Unit* (address translation)

Page Tables

- **Where is my page?**
  - VM needs a way to find a page if it is cached, find it on disk if a miss, cache it, and handle eviction
  - Provided by OS, software, & *Memory Management Unit* (address translation)
- **Page Table**
  - Data structure that maps VA to PA
  - OS managed; OS transfers pages: DISK<->RAM

Page Tables

- **Structure**
  - Array of *page table entries* (PTE)
    Each page in VA space has a PTE (conceptually, anyway)
  - PTE consists of:
    - Valid bit
    - *n*-bit address field
  - Each page in VA space has a PTE at a fixed offset

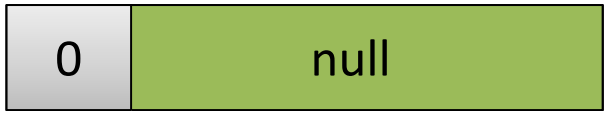
Page Tables

- **Structure**
  - Array of *page table entries* (PTE)
    Each page in VA space has a PTE (conceptually anyway)
  - PTE consists of:
    - Valid bit
    - *n*-bit address field

  - Each page in VA space has a PTE at a fixed offset
- **Valid Bit**
  - Indicates if VP is cached in RAM

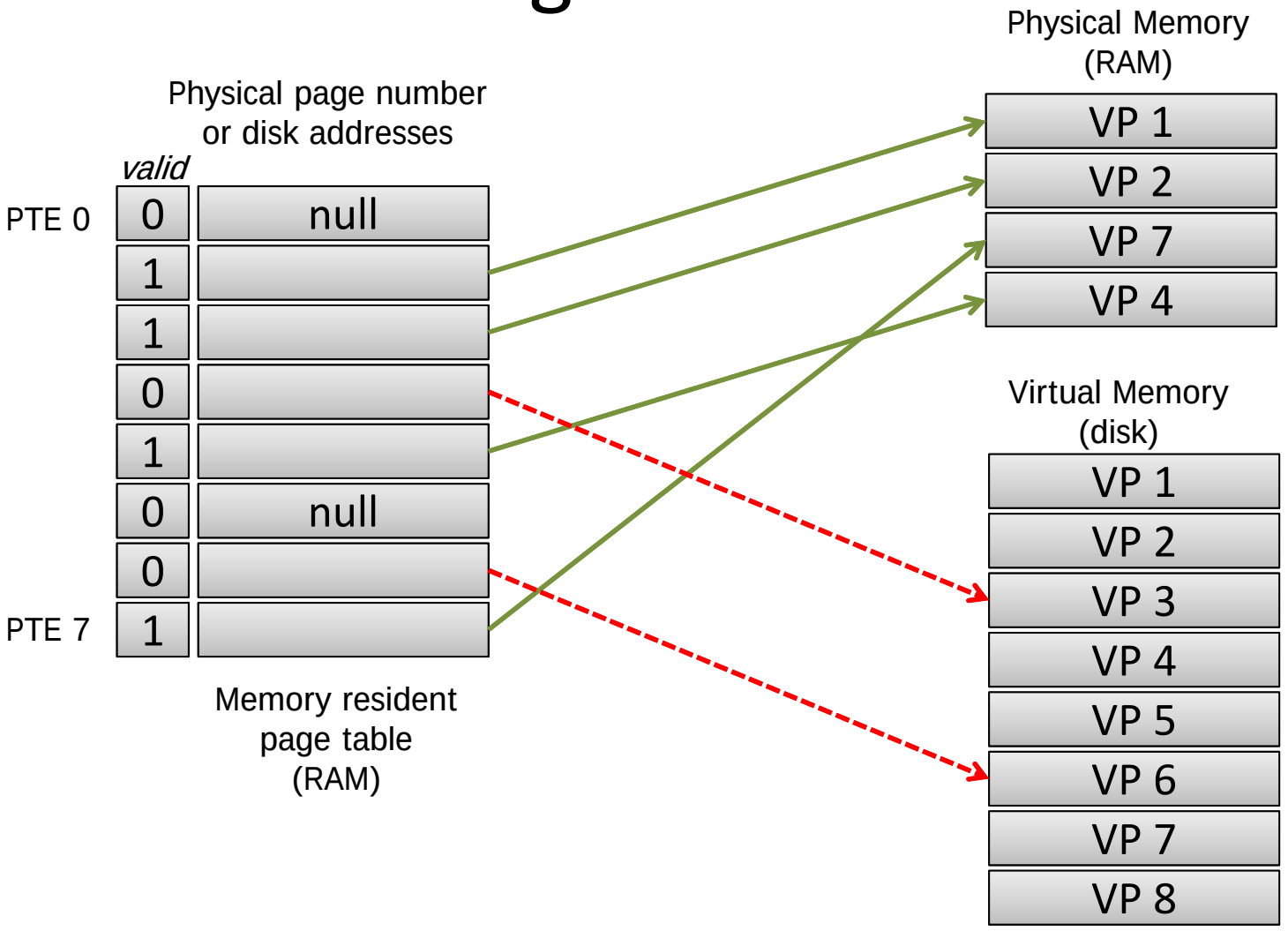
Page Tables

- **Structure**
  - Array of *page table entries* (PTE)
    Each page in VA space has a PTE (conceptually anyway)
  - PTE consists of:
    - Valid bit
    - *n*-bit address field



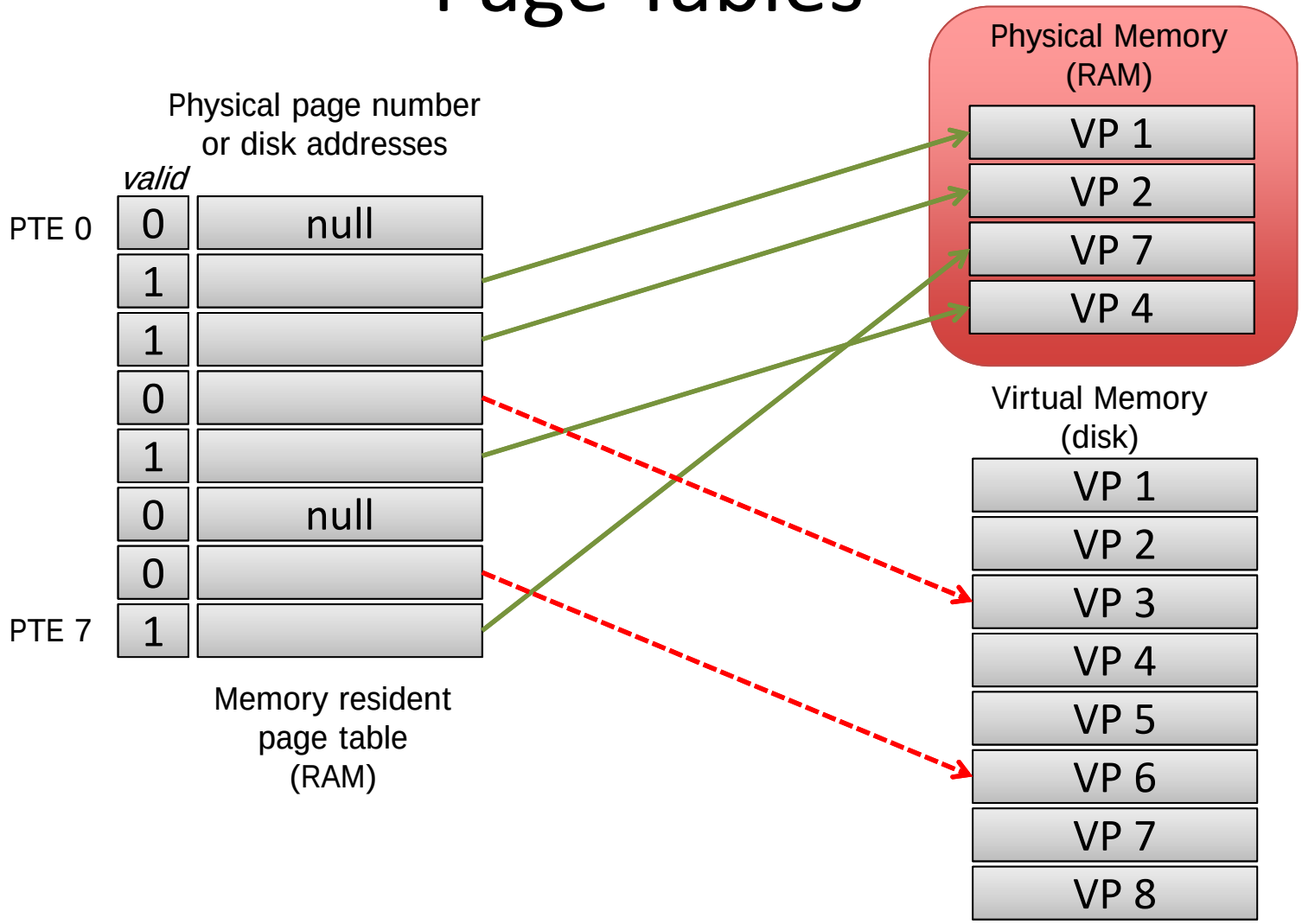
  - Each page in VA space has a PTE at a fixed offset
- **Valid Bit**
  - Indicates if VP is cached in RAM
- **Address Field**
  - Physical page in RAM where VP is stored
  - **Null** if VP has not been allocated yet (or simply ignored)

Page Tables


Physical Memory
(RAM)
VP 1
VP 2
VP 7
VP 4
Physical page number
or disk addresses
valid
PTE 0
0
null
1
1
0
1
0
null
0
PTE 7
1
Virtual Memory
(disk)
VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8
Memory resident
page table
(RAM)

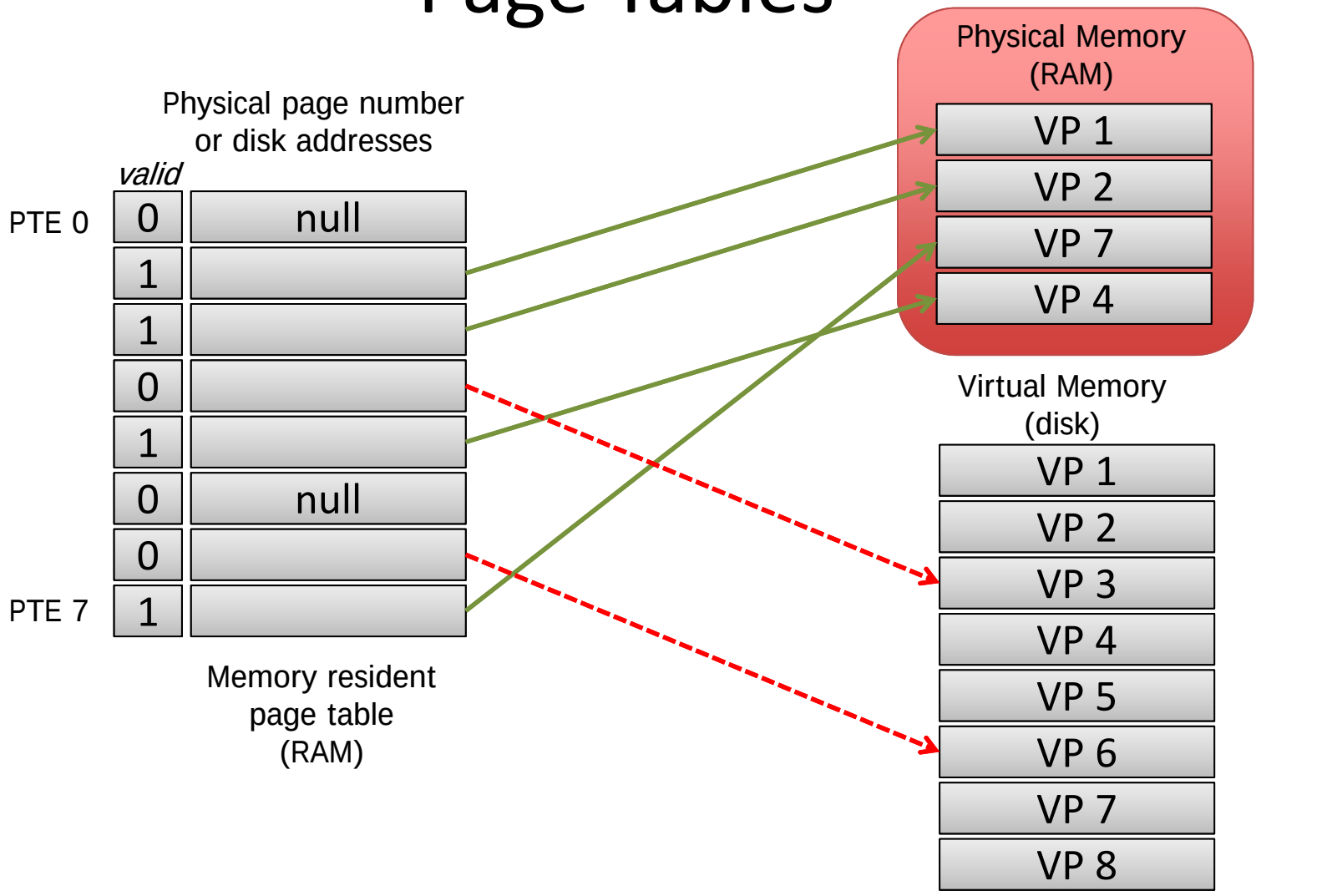
Fully Associative

Page Tables

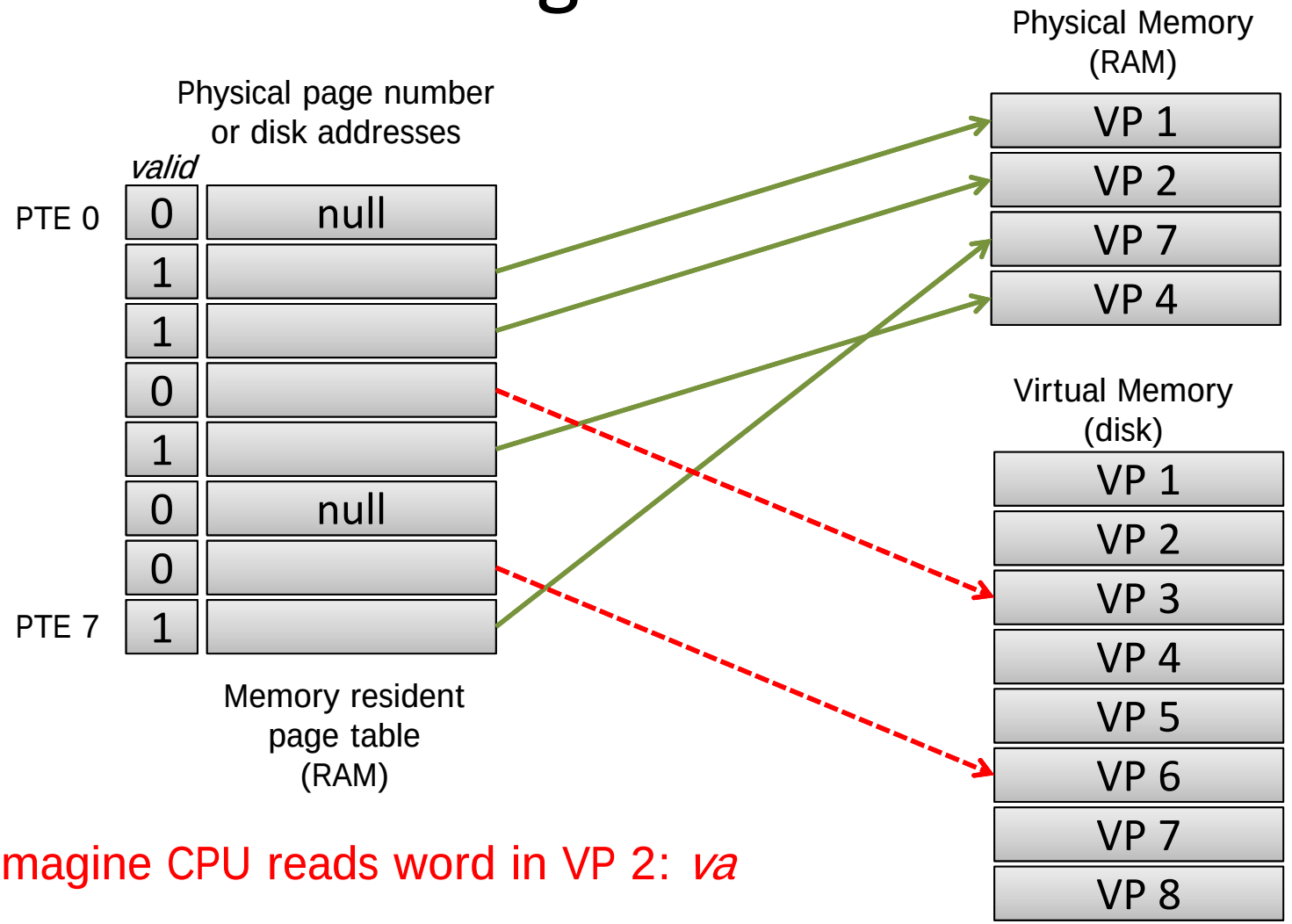


Page Tables

Fully Associative
VP can go anywhere!

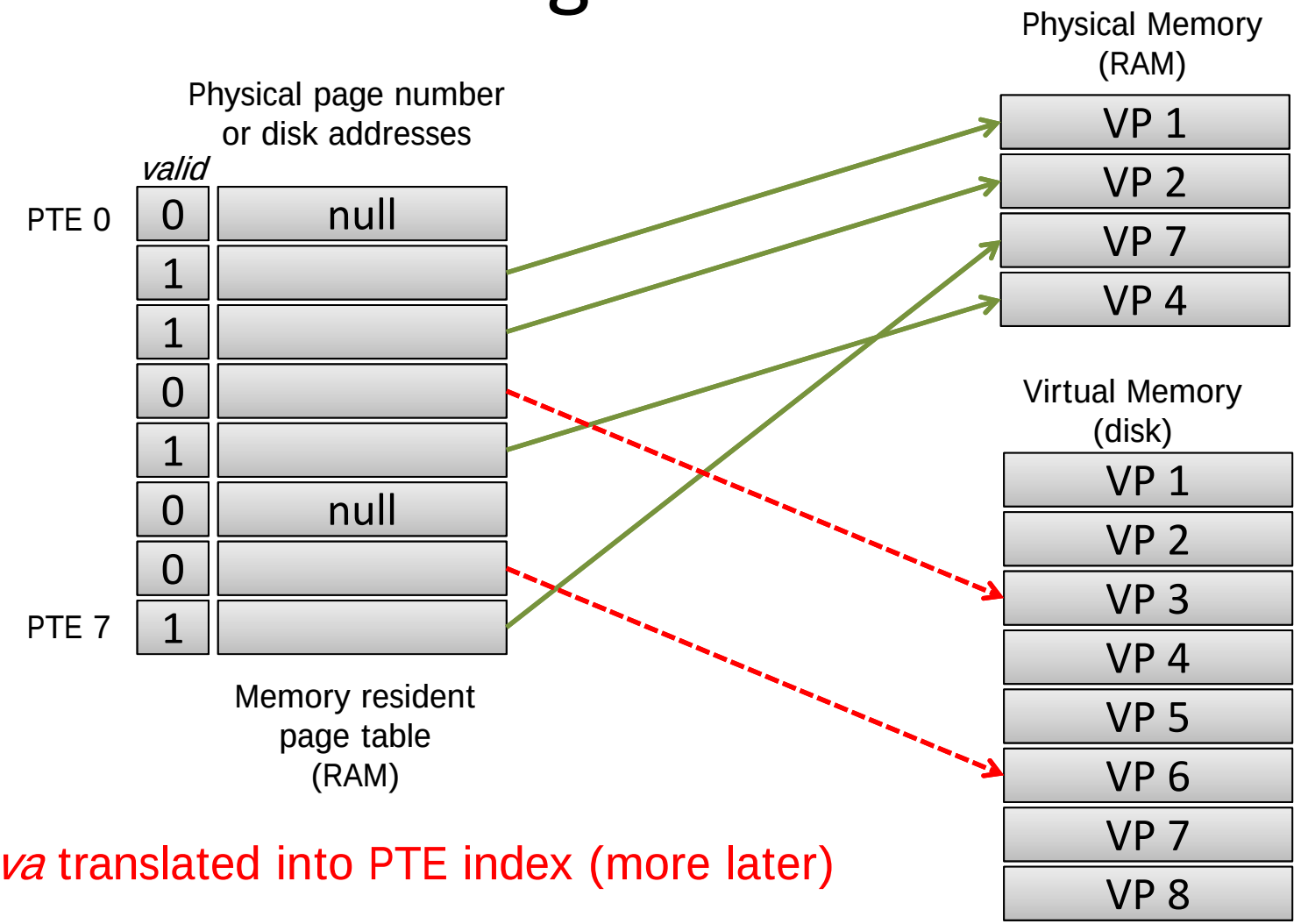


Page Tables



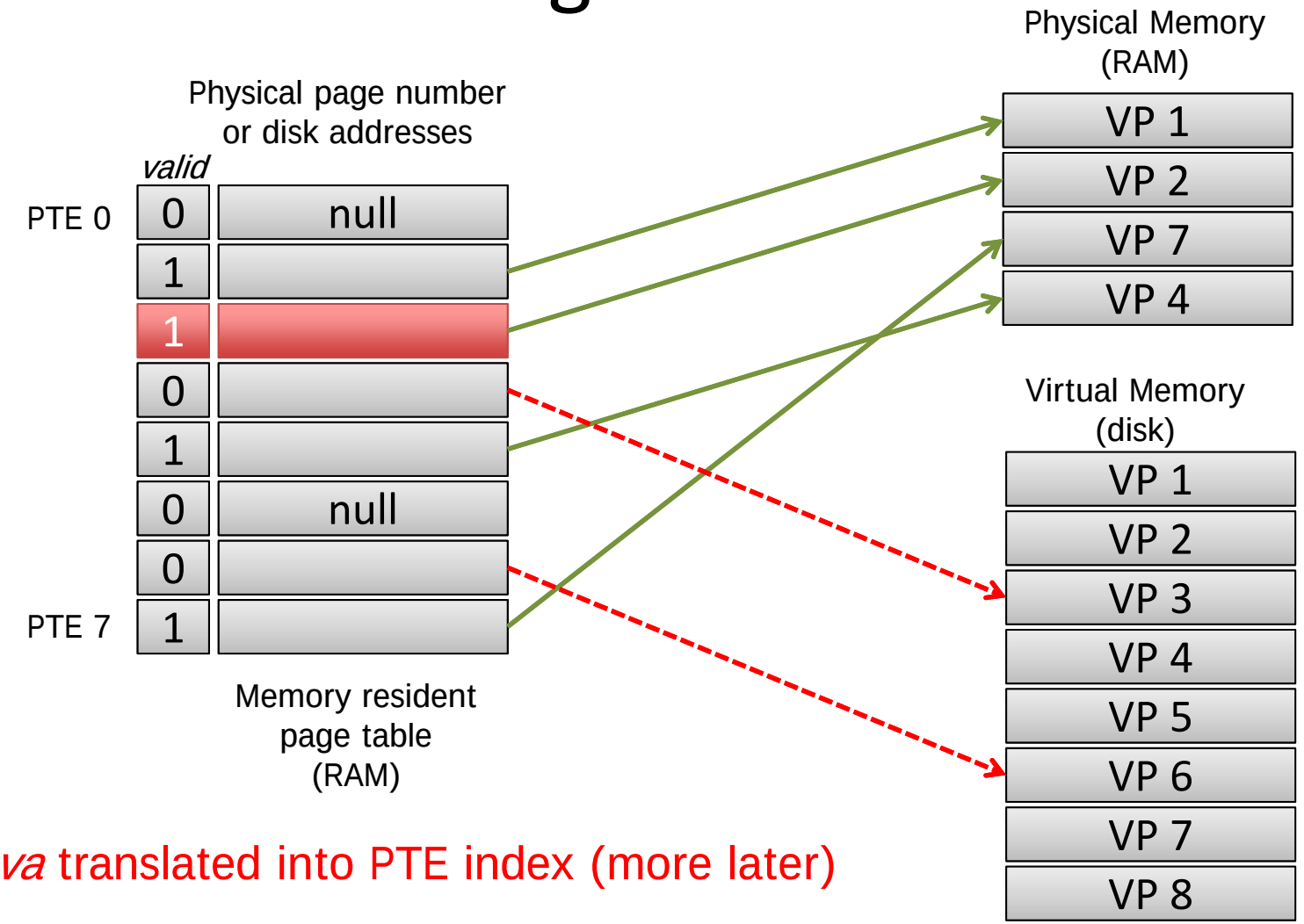
Imagine CPU reads word in VP 2: *va*

Page Tables



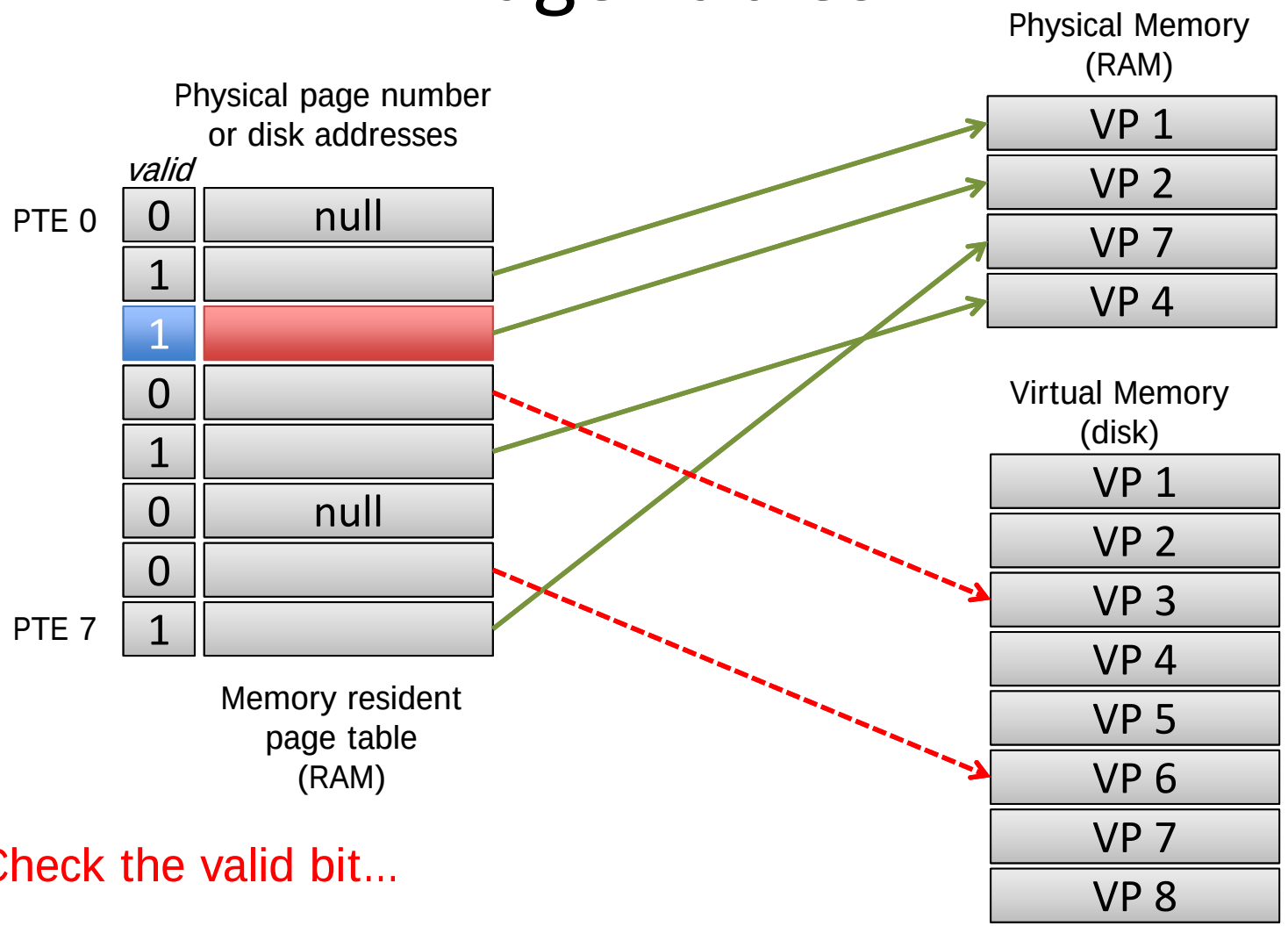
va translated into PTE index (more later)

Page Tables



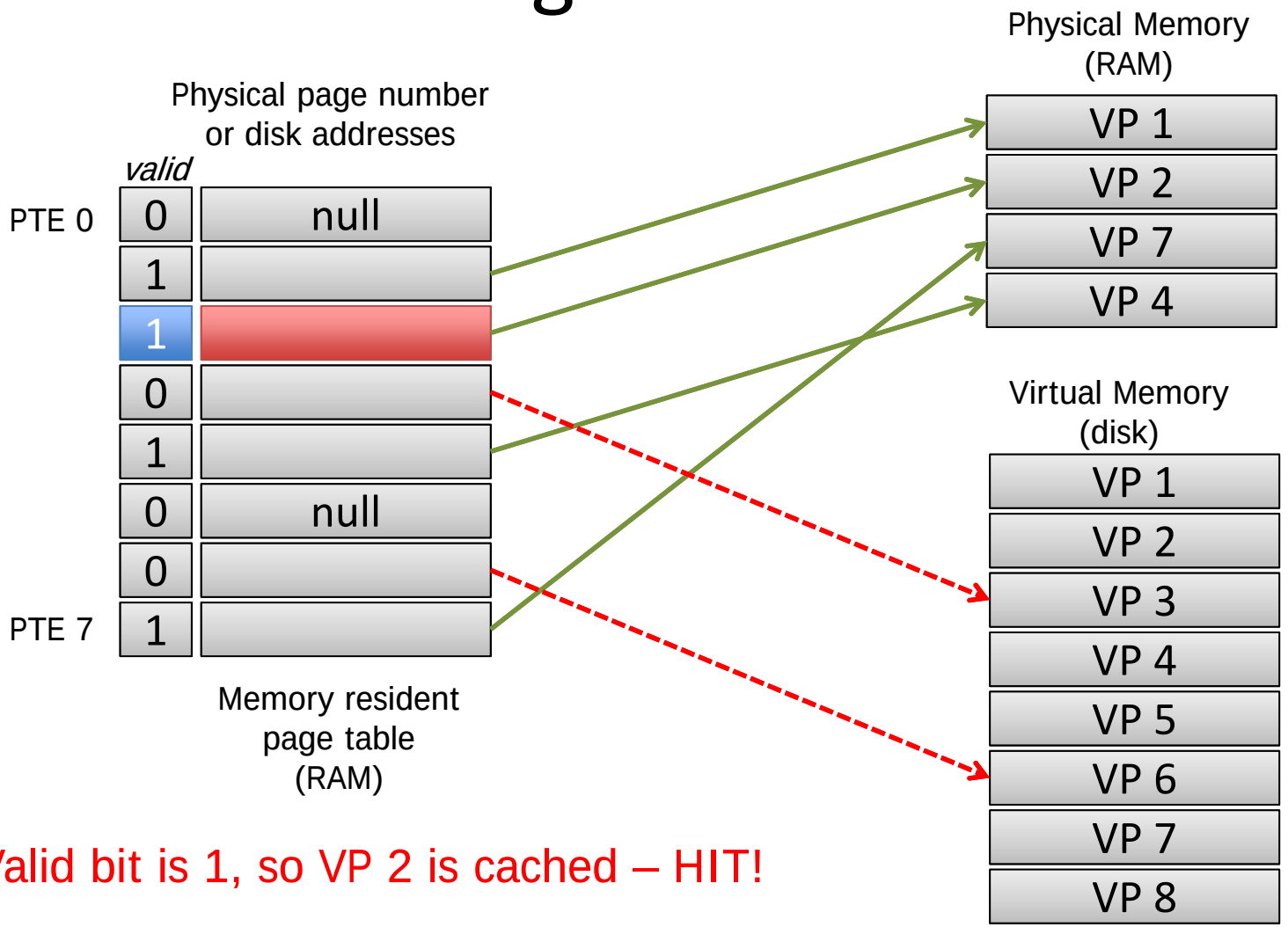
va translated into PTE index (more later)

Page Tables



Check the valid bit...

Page Tables



Valid bit is 1, so VP 2 is cached – HIT!

Page Tables

- **What happens on a miss?**
- **Page Faults**
  - These are complicated
  - Requires a precise interaction between *hardware exceptions* and kernel code
  - The hardware exception is called a: *segmentation violation*

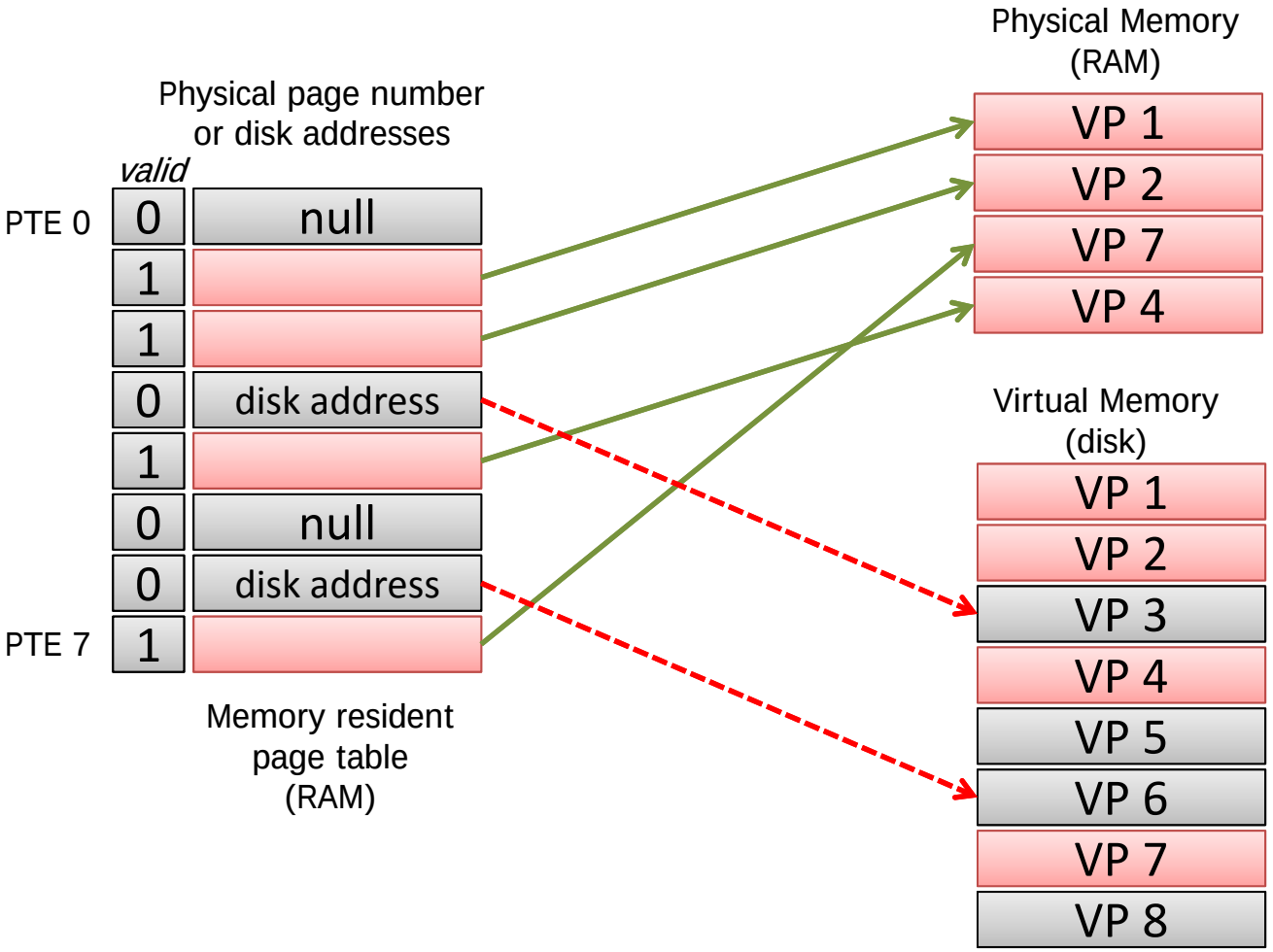
iClicker question

A segmentation violation exception (as distinguished from a signal) should be:

a) Ignorable
b) Blockable
c) Restartable
d) Continue at the next instruction
e) Fatal (kill the process)

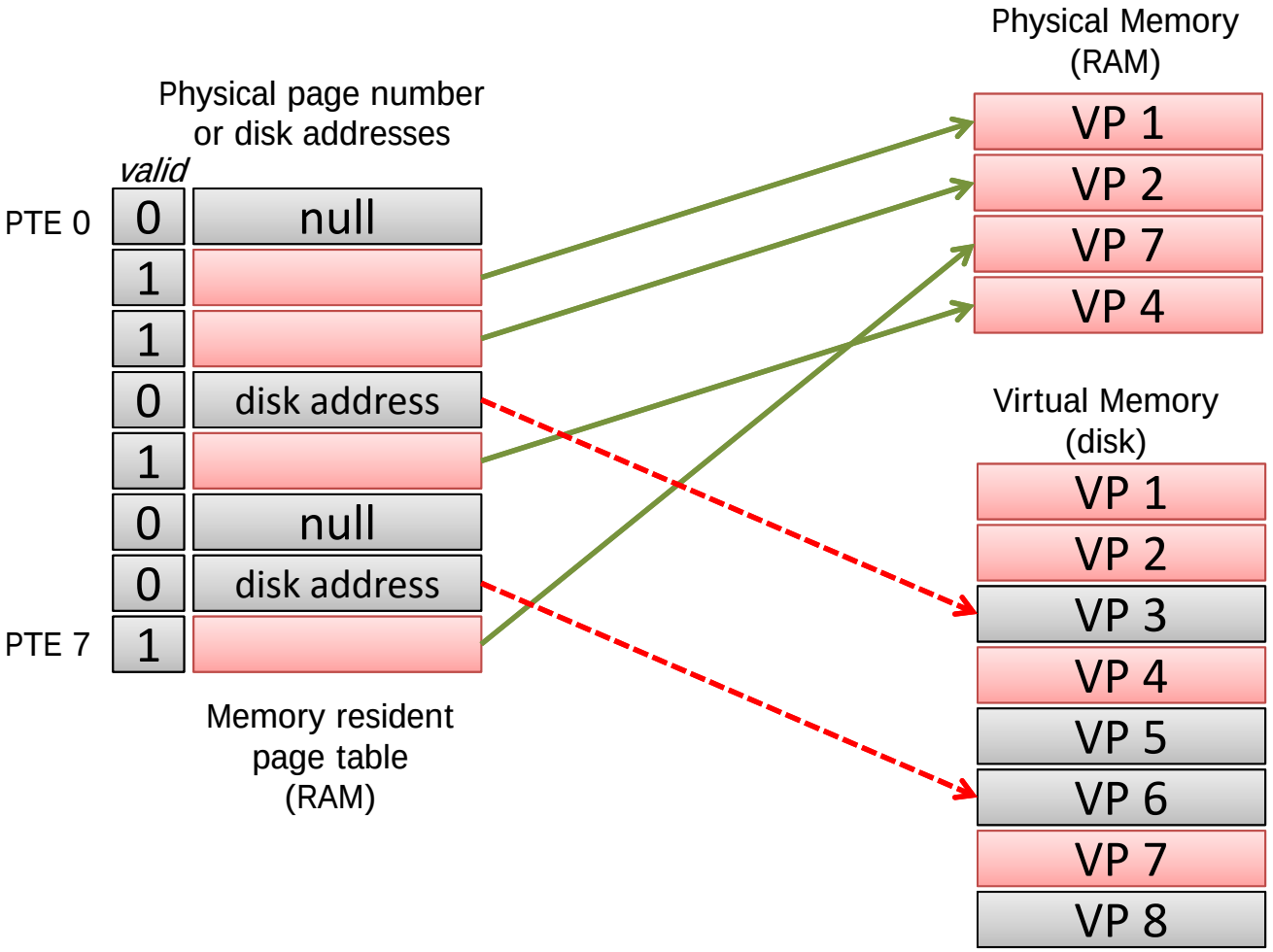
Page Faults

In this example, VP 1, 2, 4, & 7 are cached in RAM



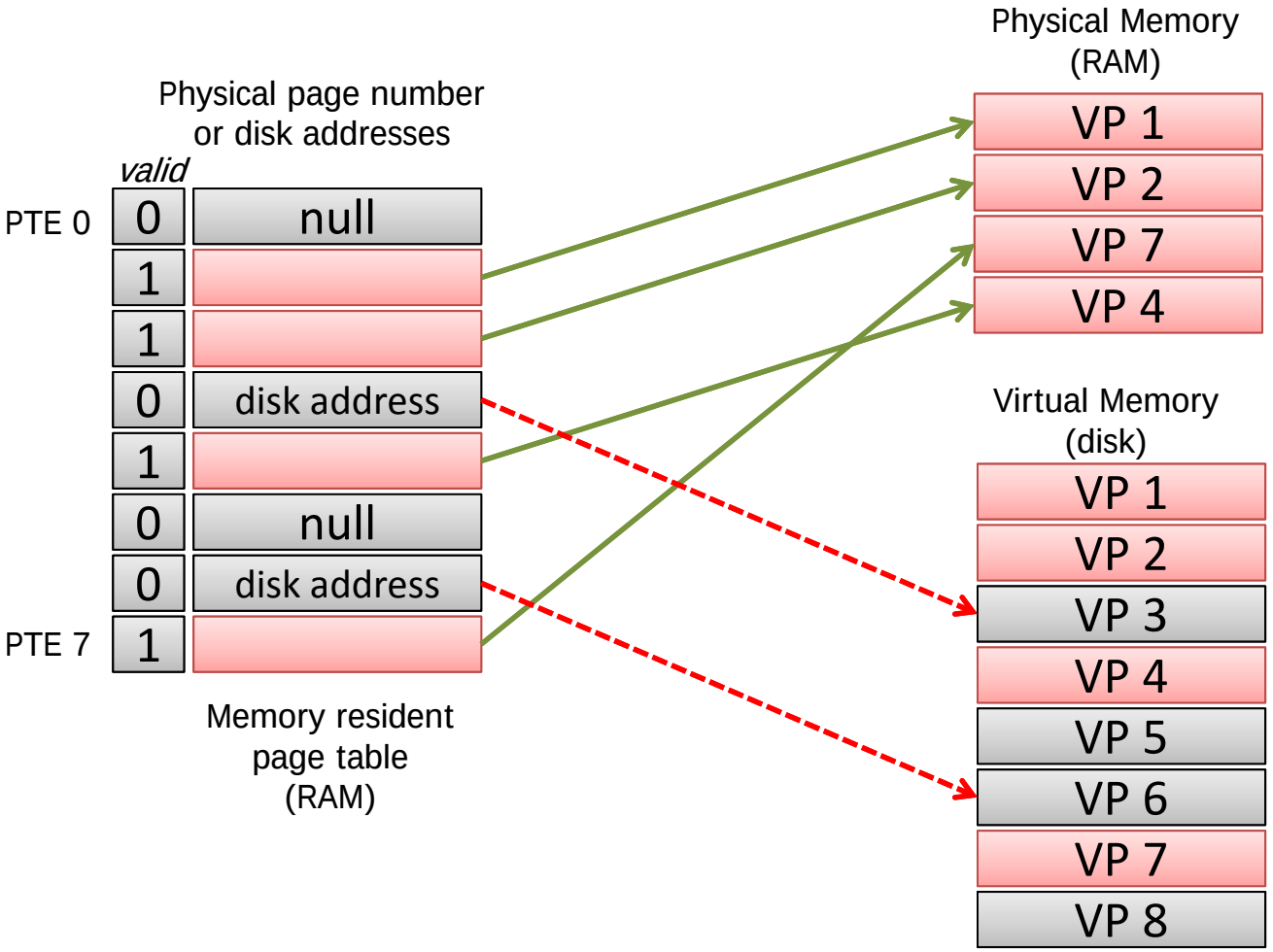
Page Faults

If we receive a virtual address in VP 1, 2, 4, or 7: we have a **hit**!



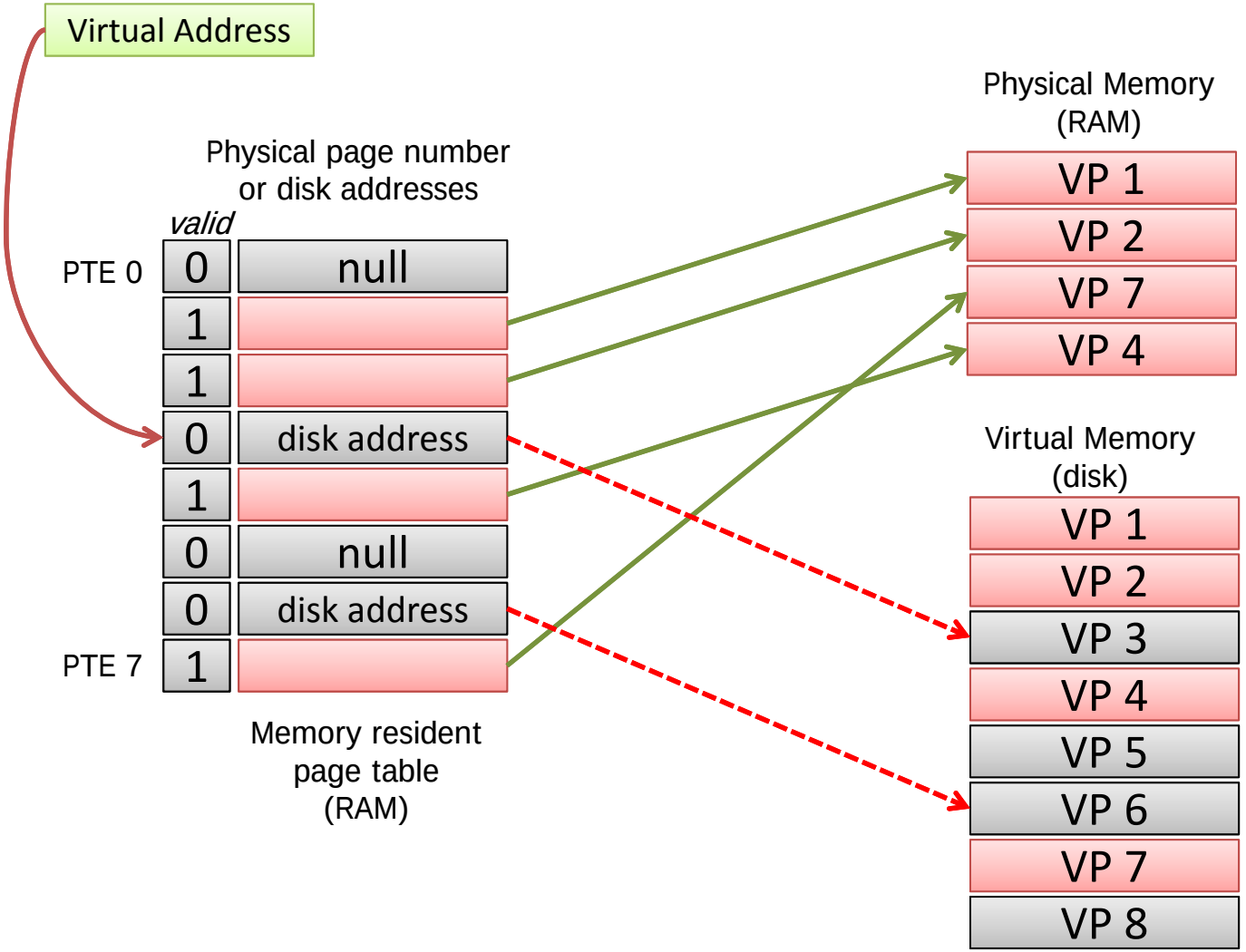
Page Faults

What if we receive a virtual address that maps to PTE 3?



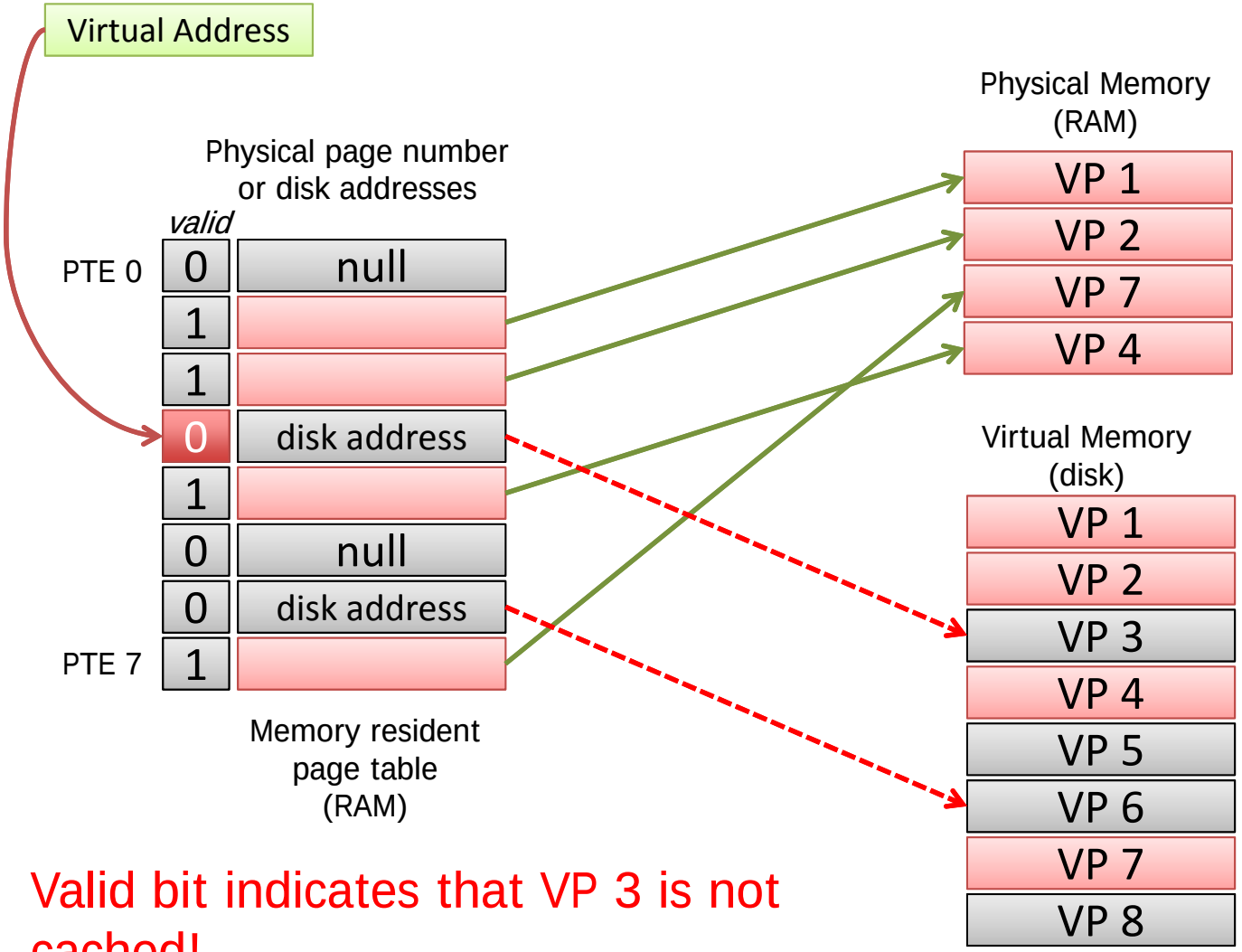
Page Faults

What if we receive a virtual address that maps to PTE 3?



Page Faults

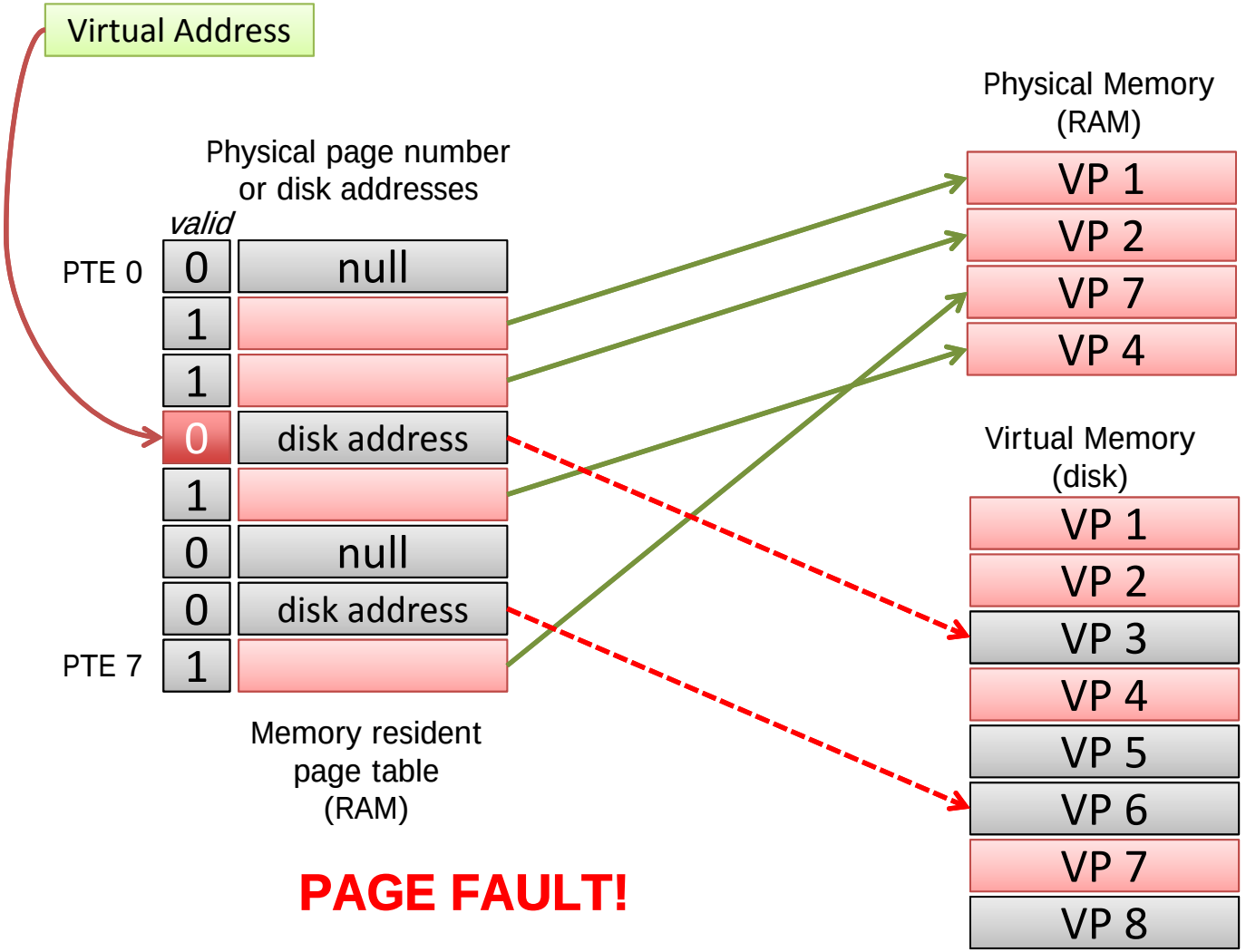
What if we receive a virtual address contained in PTE 3?



Valid bit indicates that VP 3 is not cached!

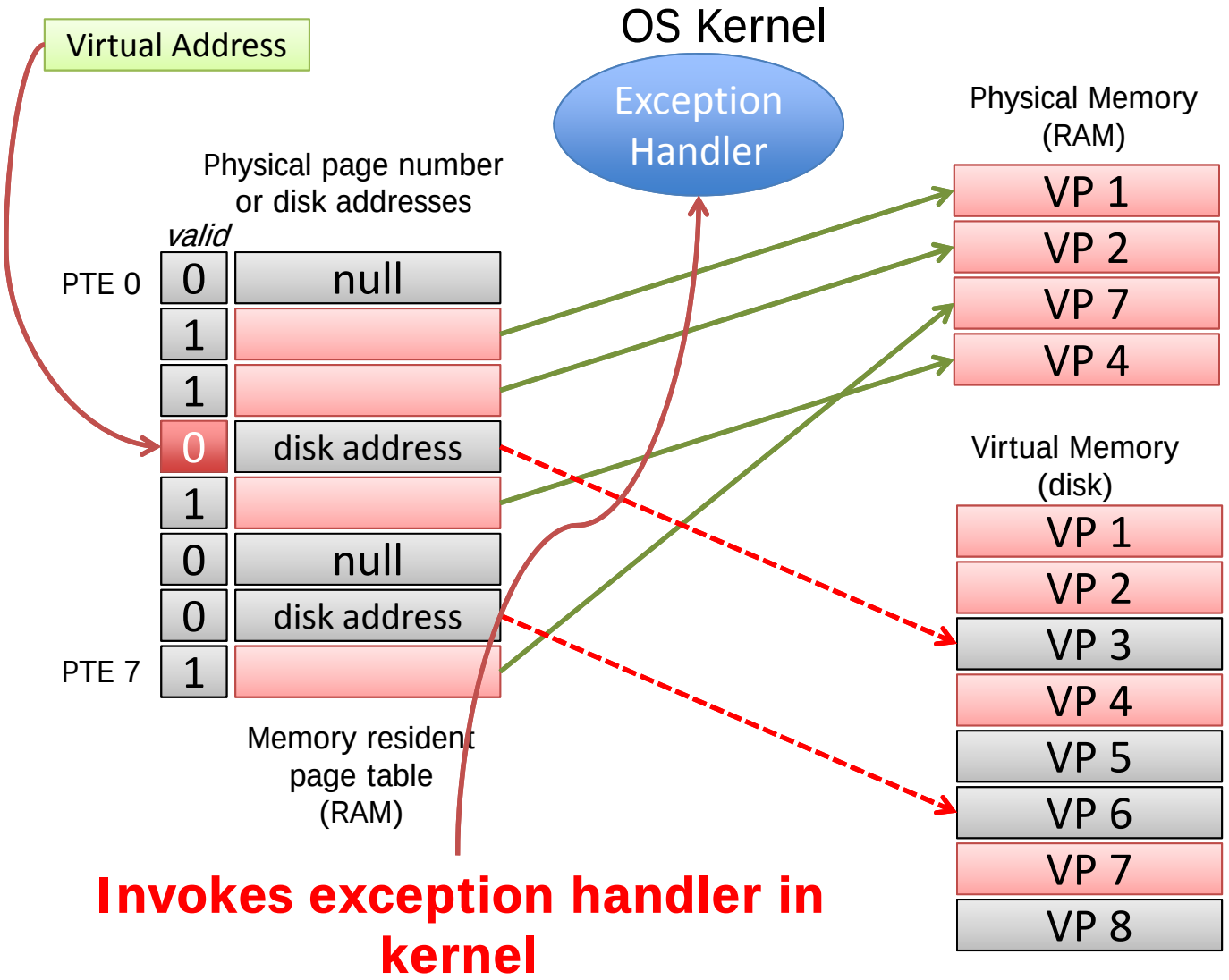
Page Faults

What if we receive a virtual address contained in PTE 3?



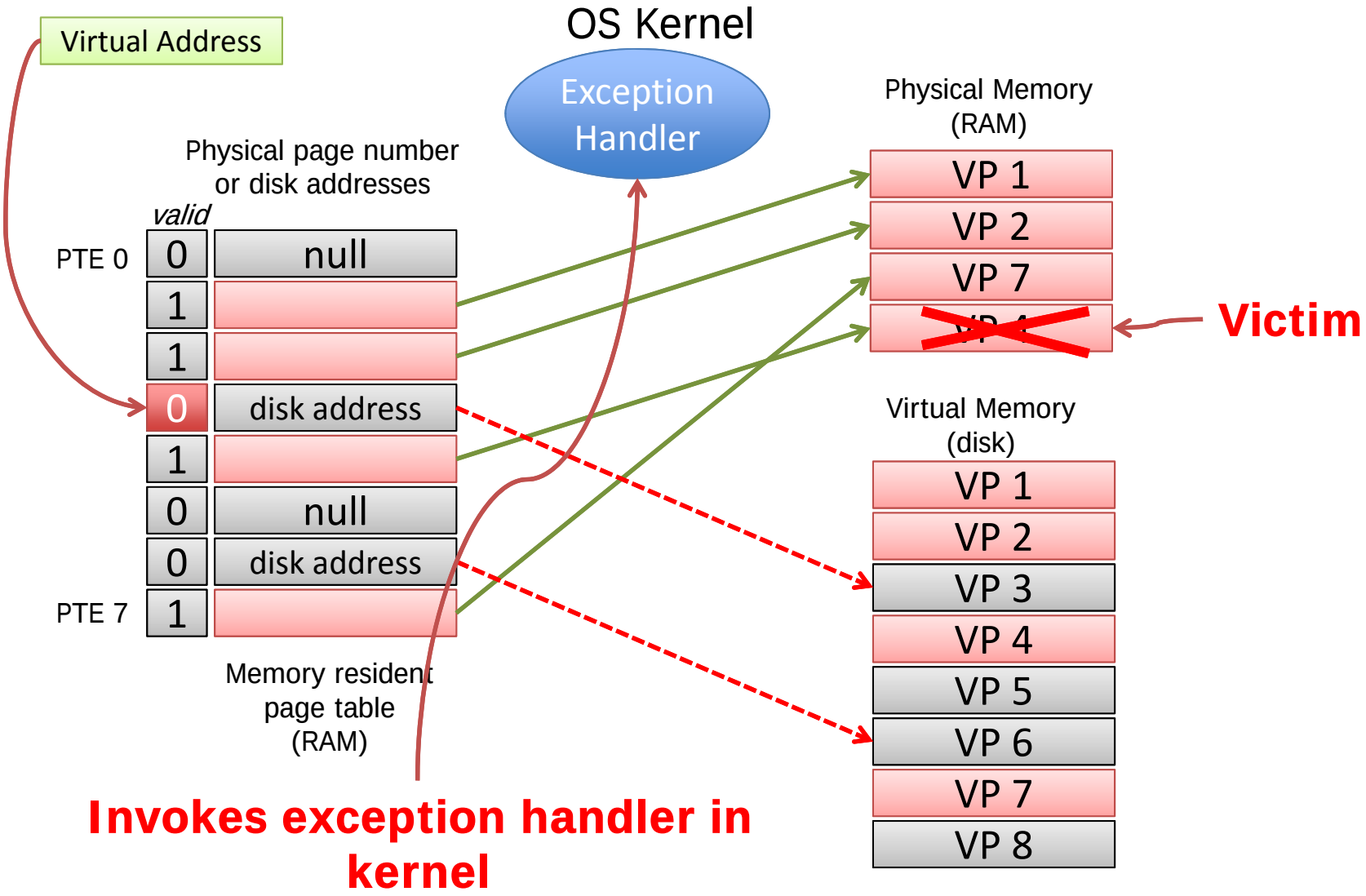
Page Faults

What if we receive a virtual address contained in PTE 3?



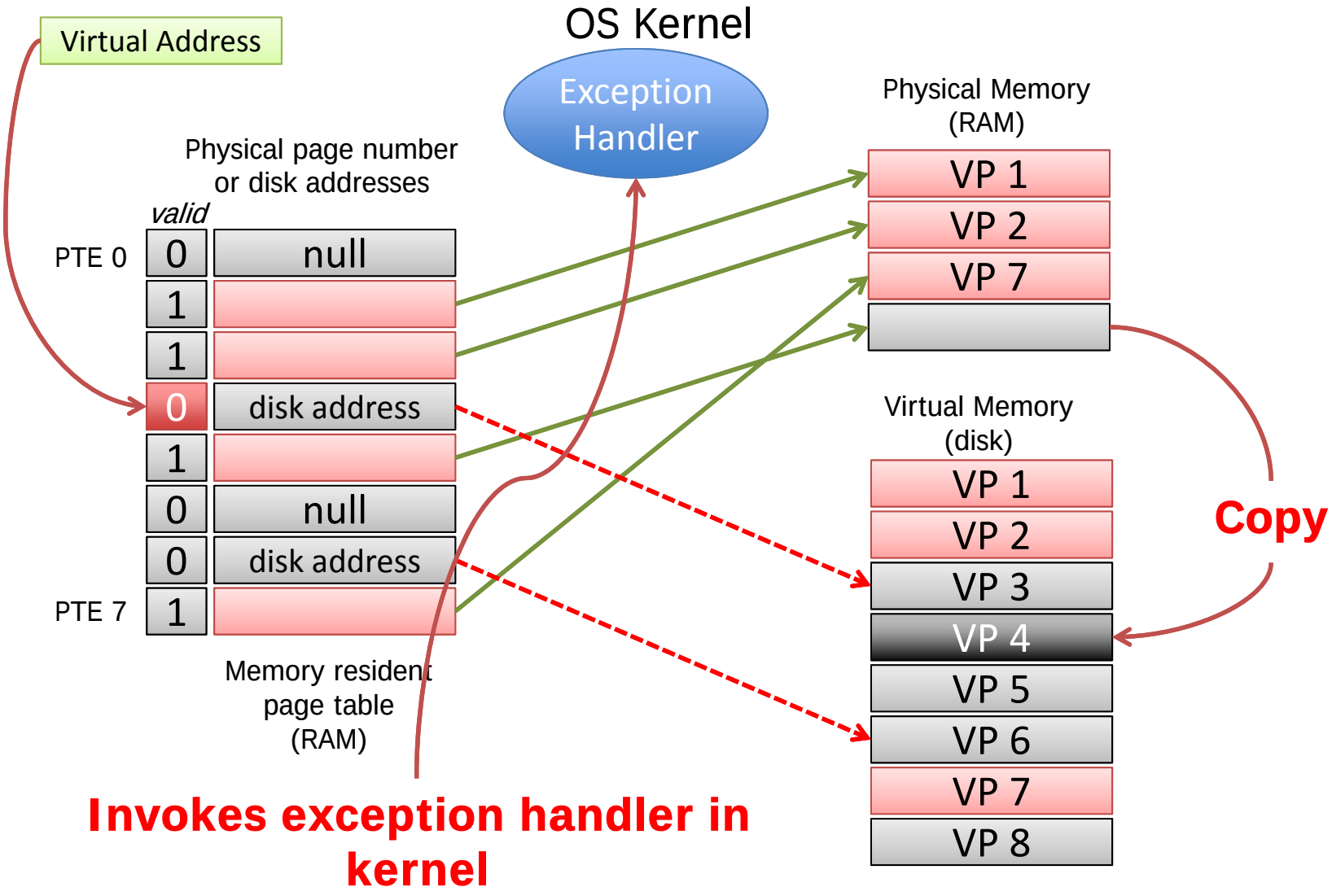
Page Faults

What if we receive a virtual address contained in PTE 3?



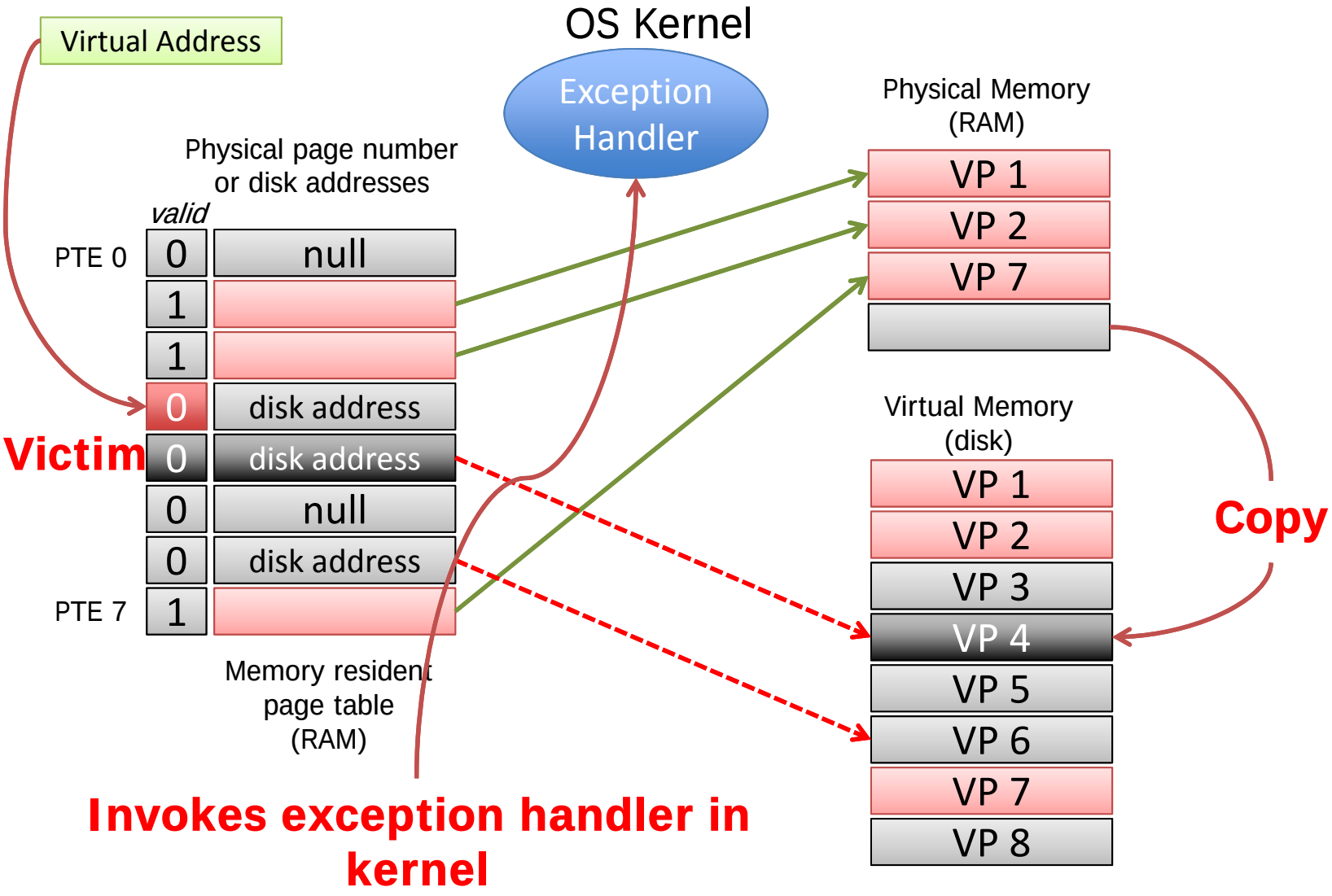
Page Faults

What if we receive a virtual address contained in PTE 3?



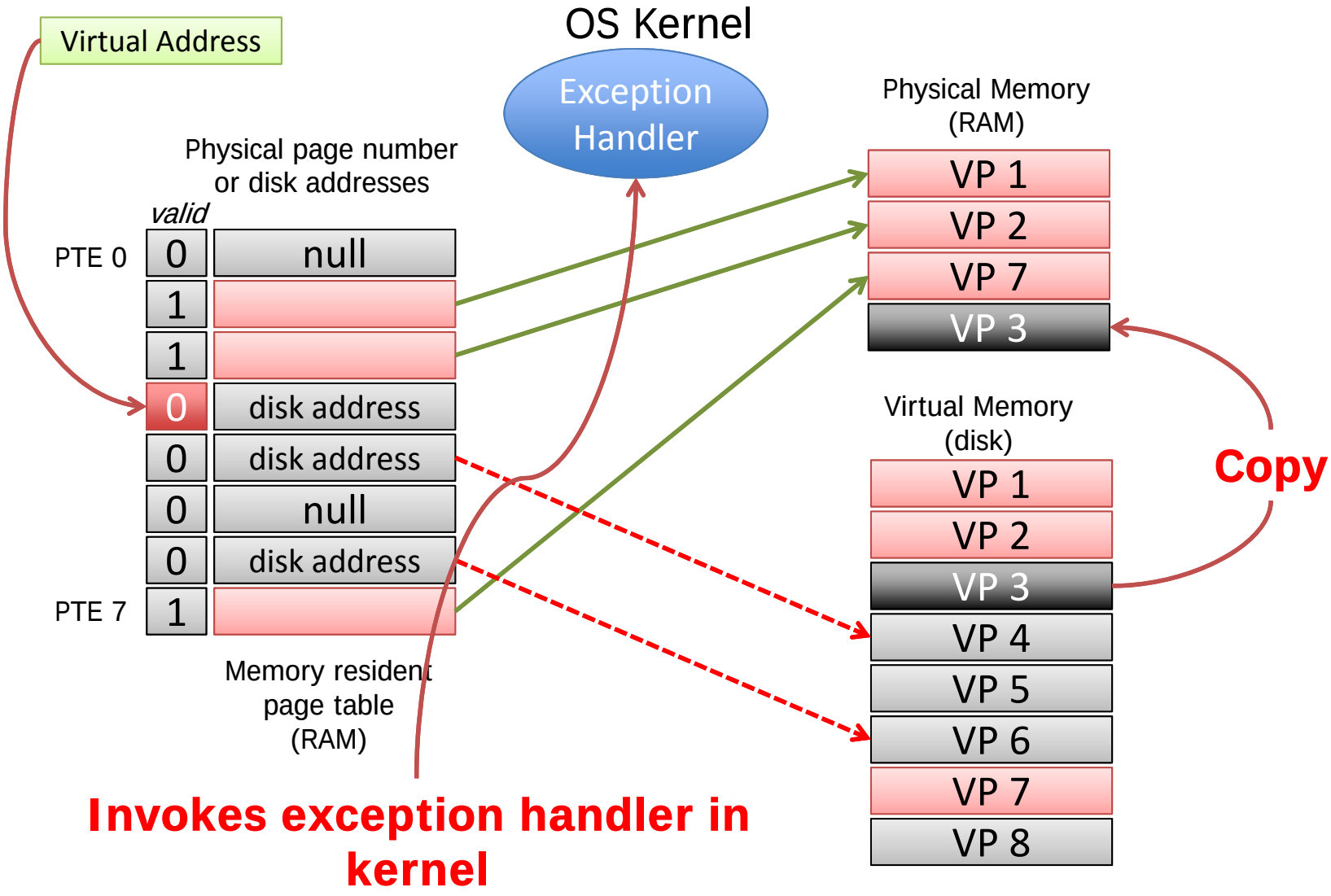
Page Faults

What if we receive a virtual address contained in PTE 3?



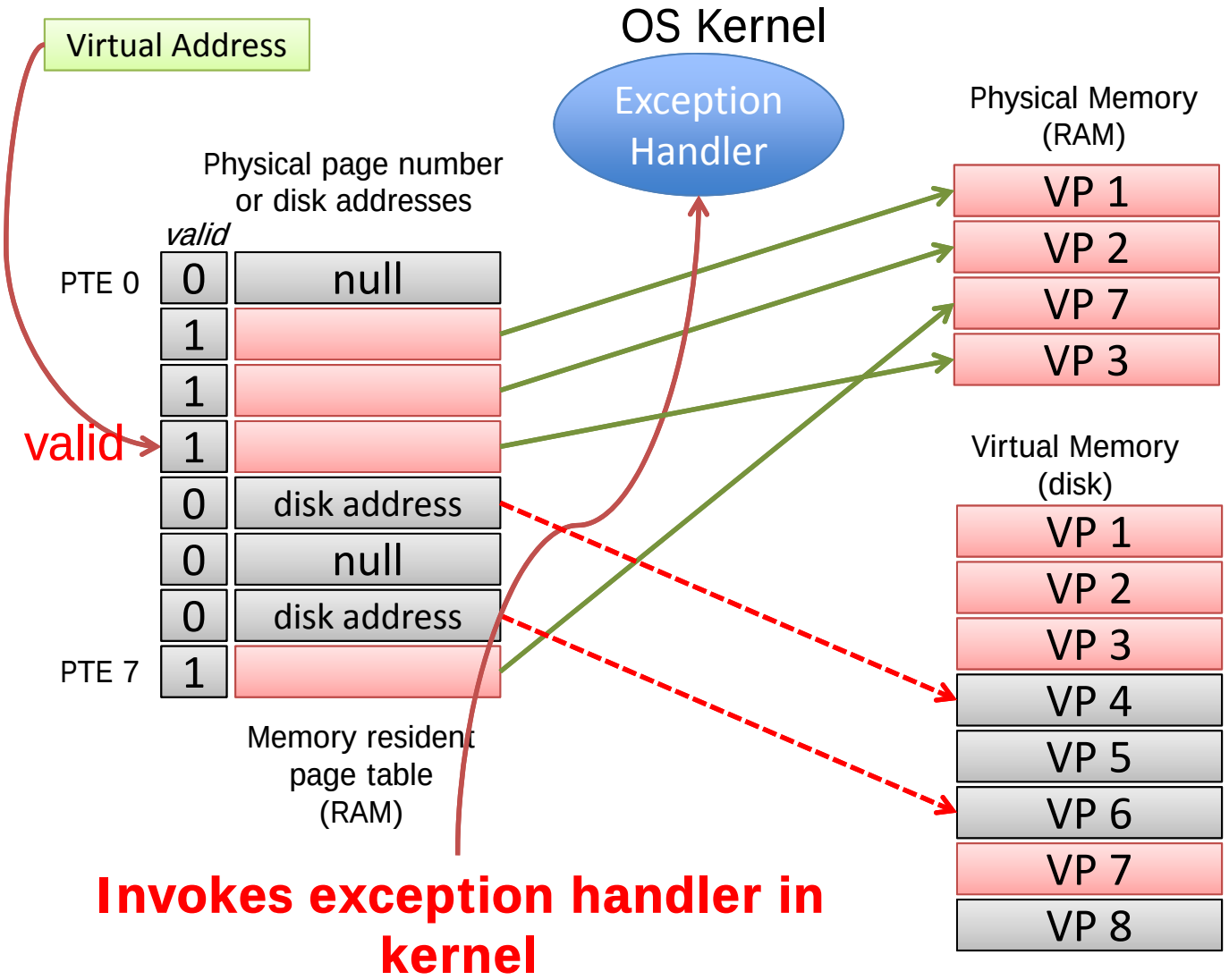
Page Faults

What if we receive a virtual address contained in PTE 3?



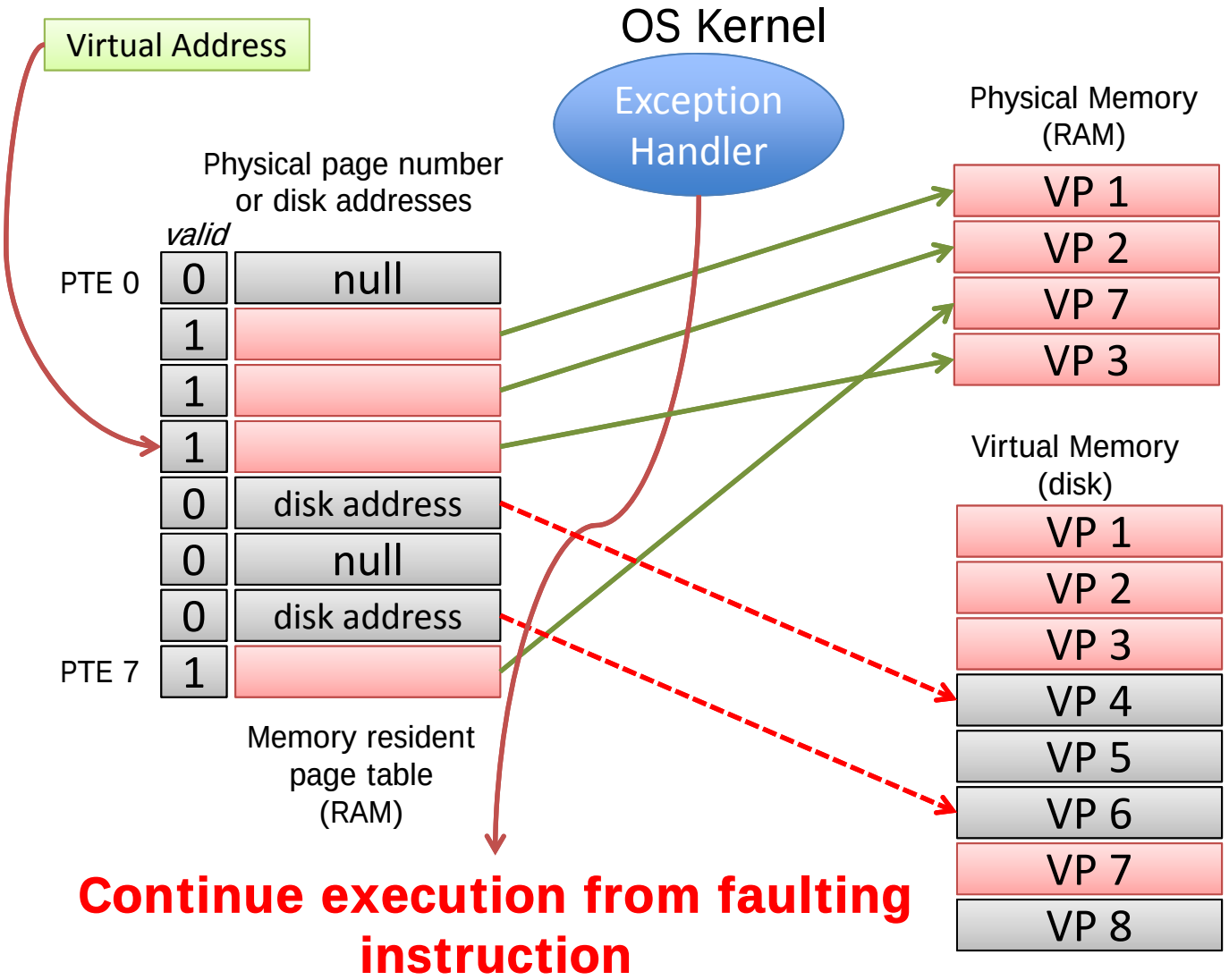
Page Faults

What if we receive a virtual address contained in PTE 3?



Page Faults

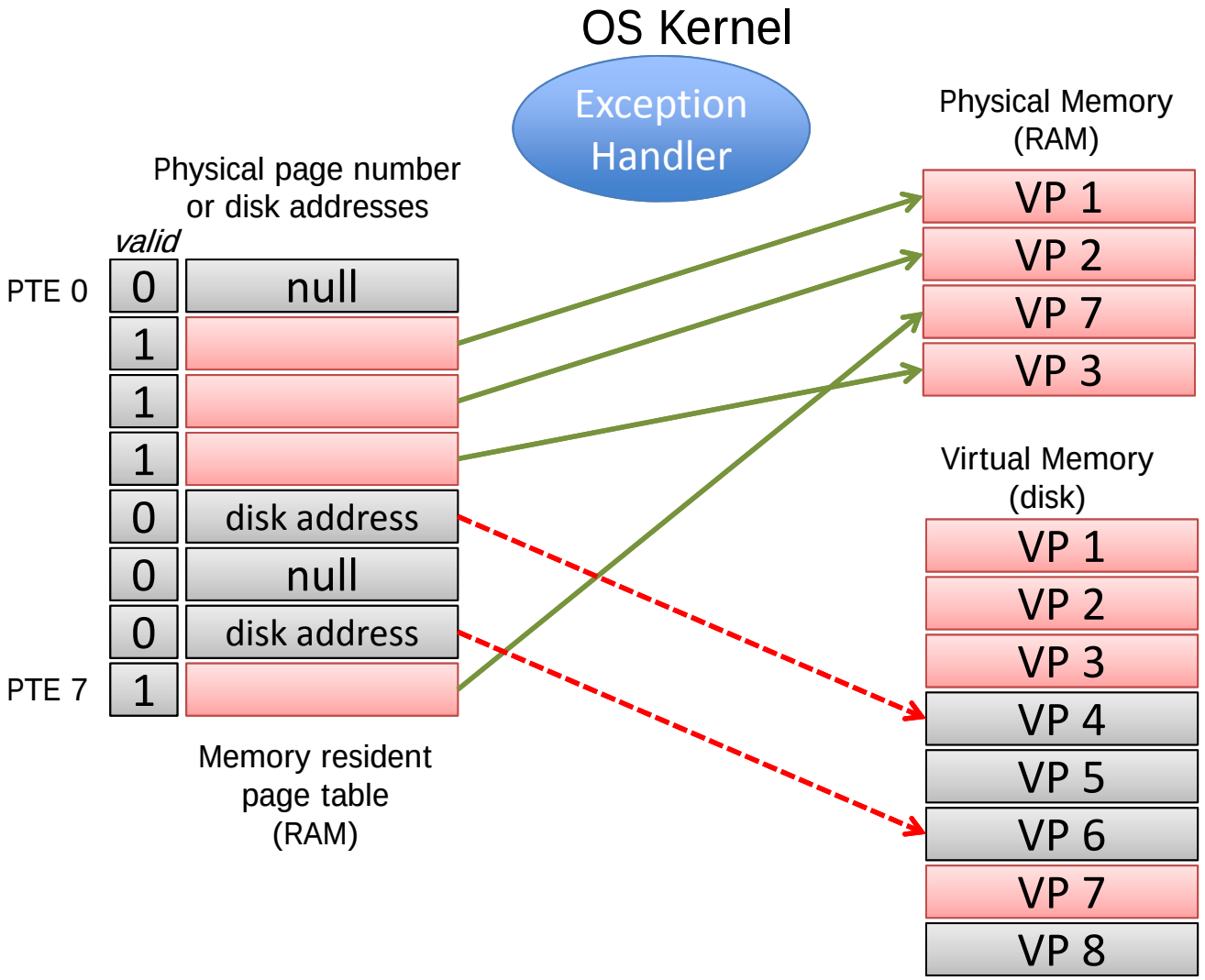
What if we receive a virtual address contained in PTE 3?



Continue execution from faulting instruction

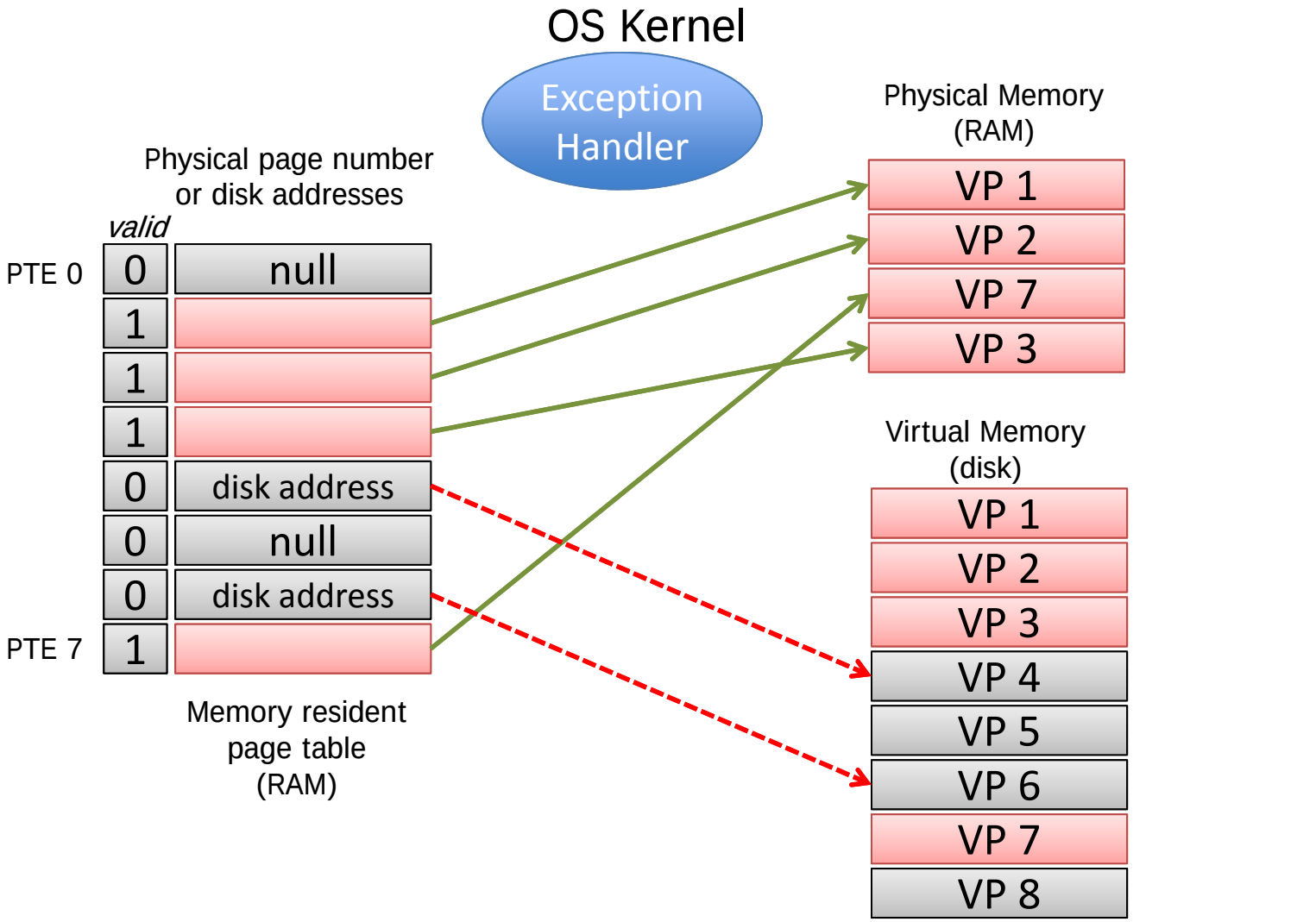
Page Faults

This process is called **paging** or **swapping**.



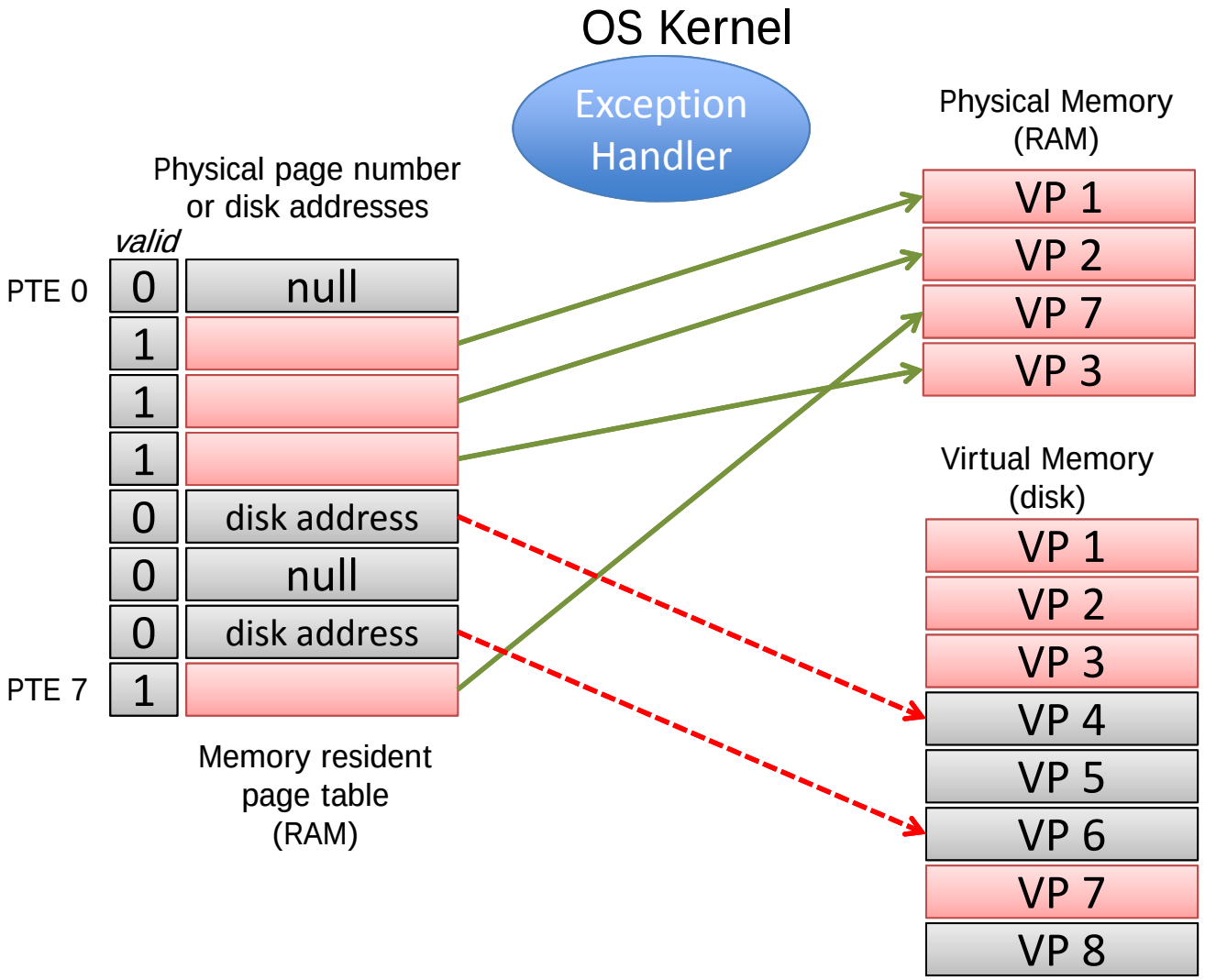
Page Allocation

What if we need to allocate a **new** page: malloc(sizeof(foo));



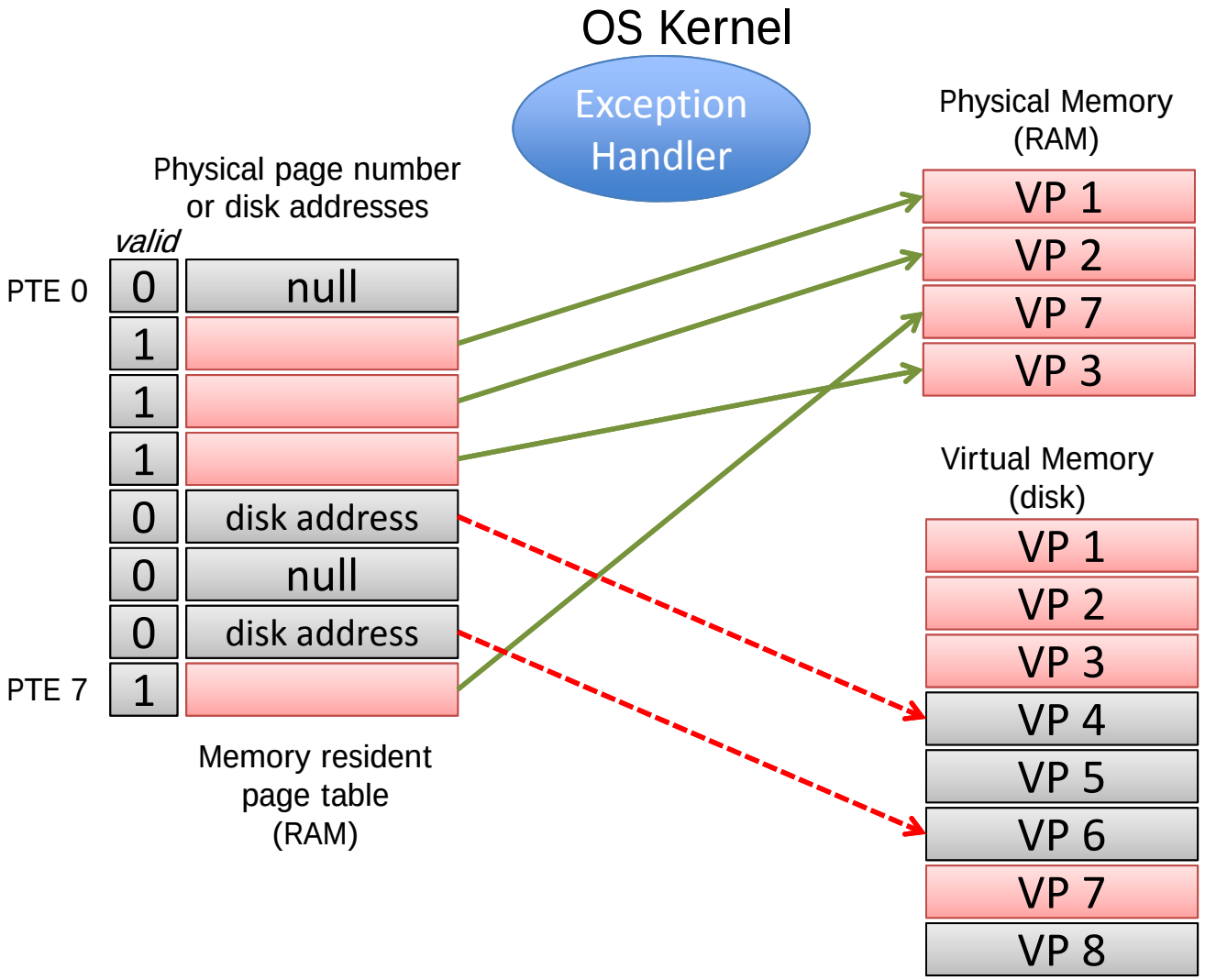
Page Allocation

Imagine we need to allocate more space, because we are full



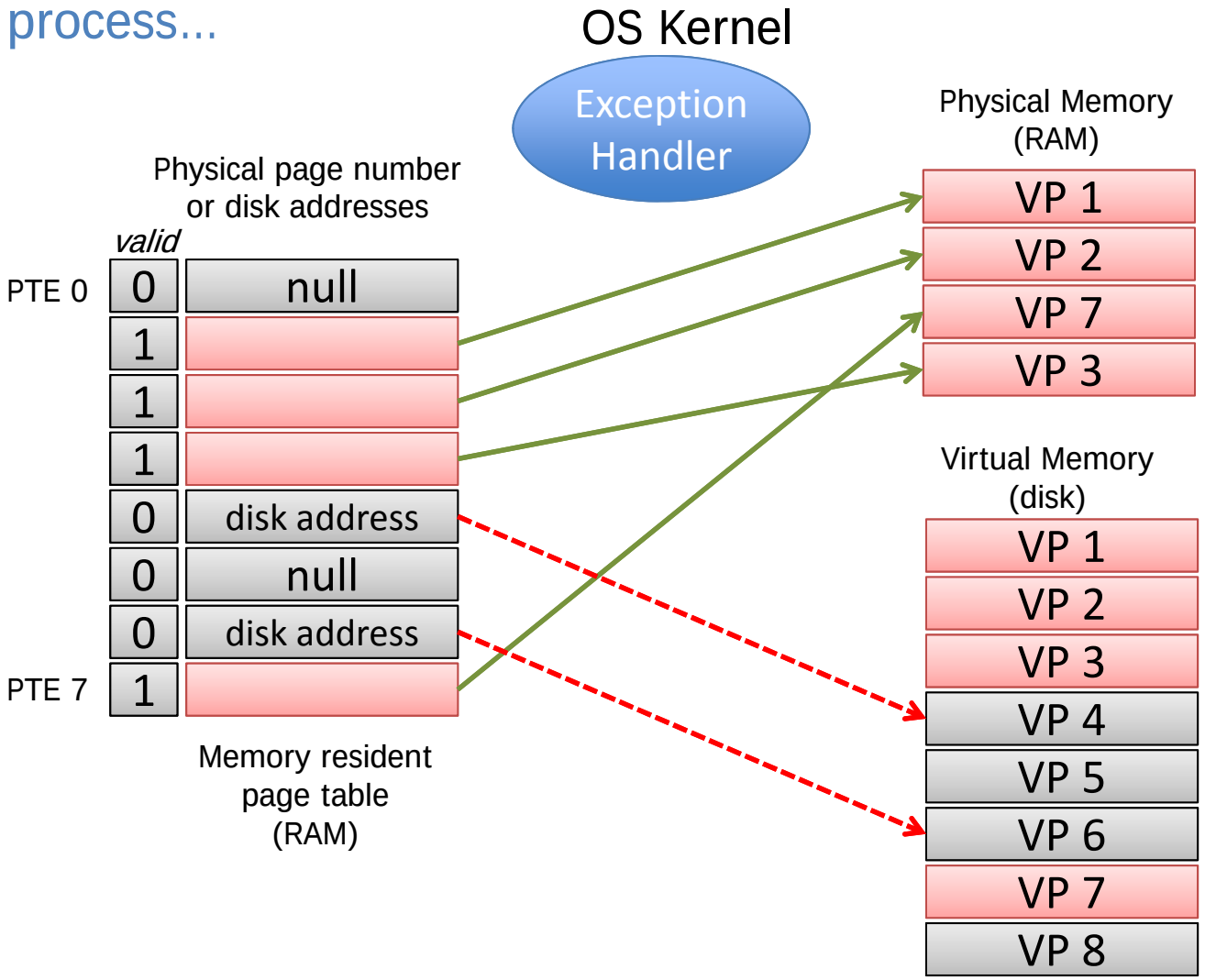
Page Allocation

Need to allocate a new page, say VP 5



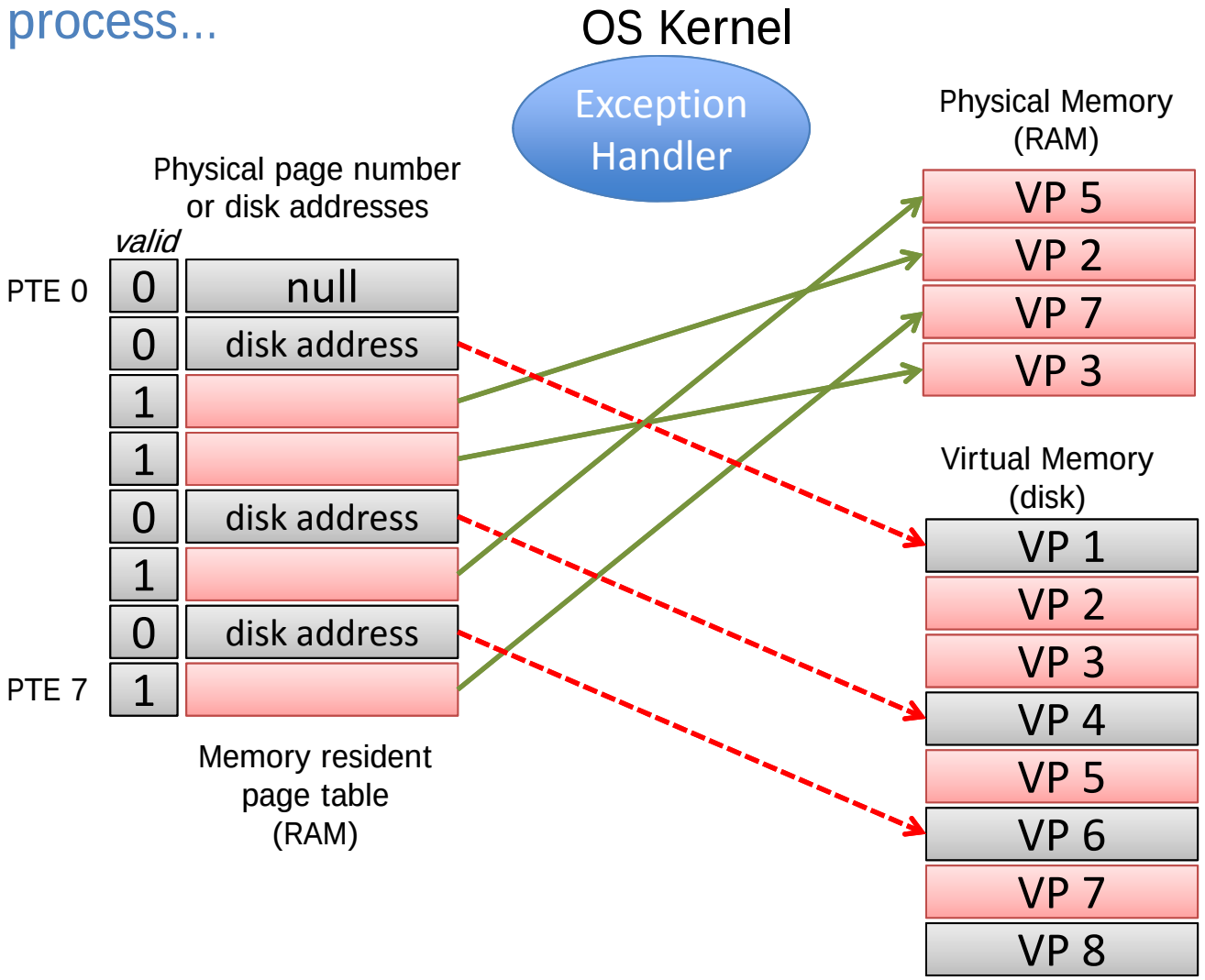
Page Allocation

Need to allocate a new page, say VP 5, and do page fault process...



Page Allocation

Need to allocate a new page, say VP 5, and do page fault process...



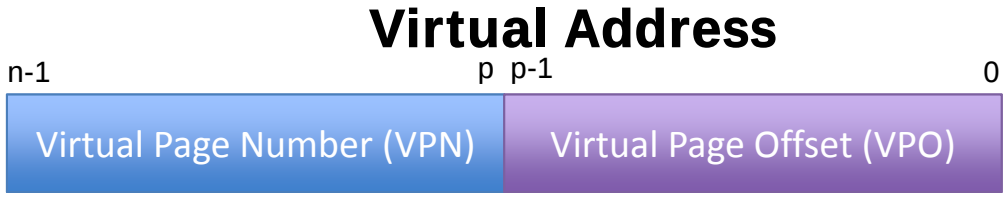
Locality

- **A page fault is very expensive!**
  - So, why do this at all?
- **Locality to the rescue!**
  - Programs tend to work on the same data for long periods of time: small set of **active pages**
  - This is known as the **working set** or **resident set**
  - As long as our programs have good *temporal* locality, VM systems work very well
- **If the working set exceeds the size of physical memory: thrashing**

Address Translation

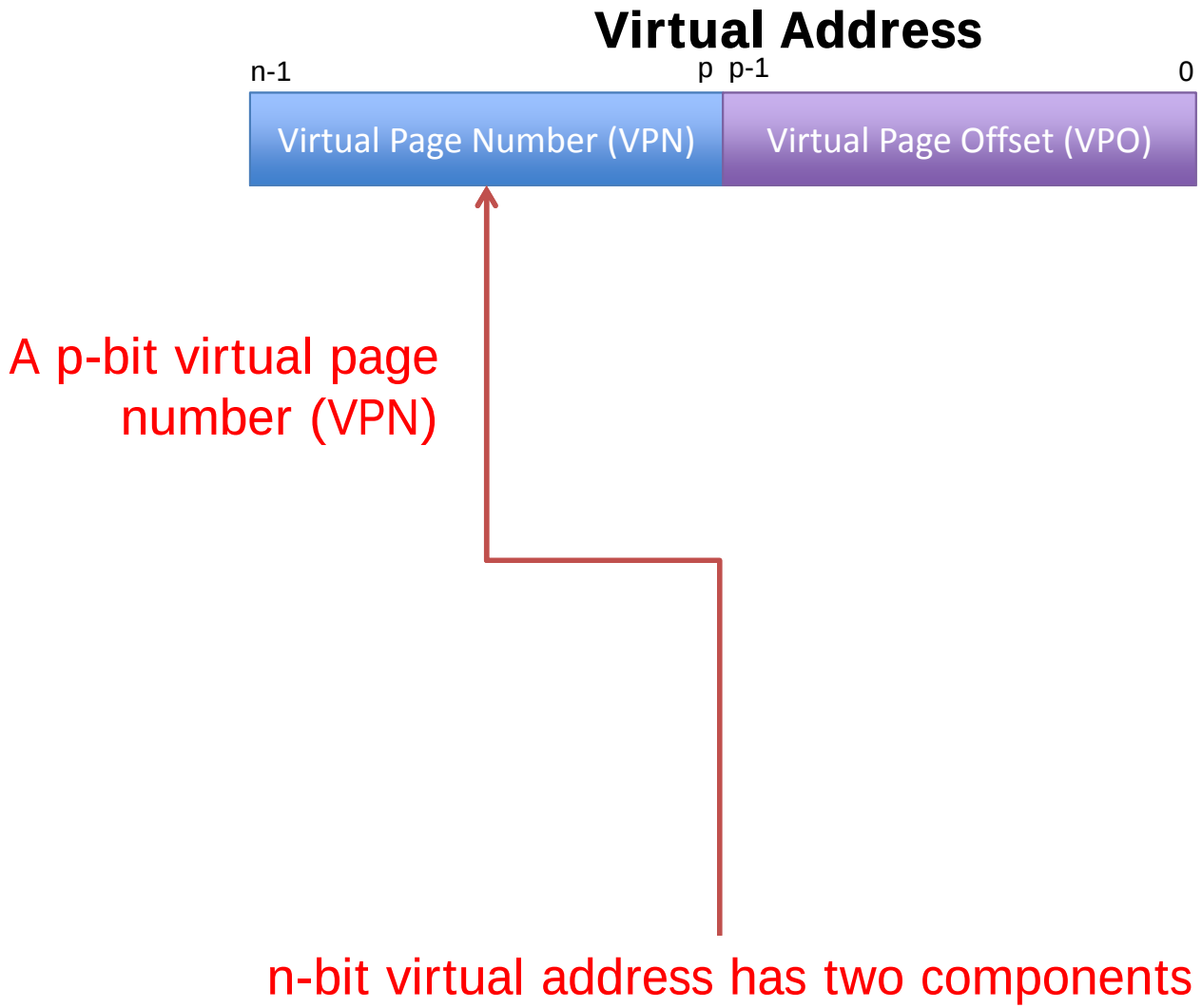
- **We saw that virtual address "magically" index to a PTE.**
  - So, how are virtual addresses translated into physical addresses anyways?

Address Translation

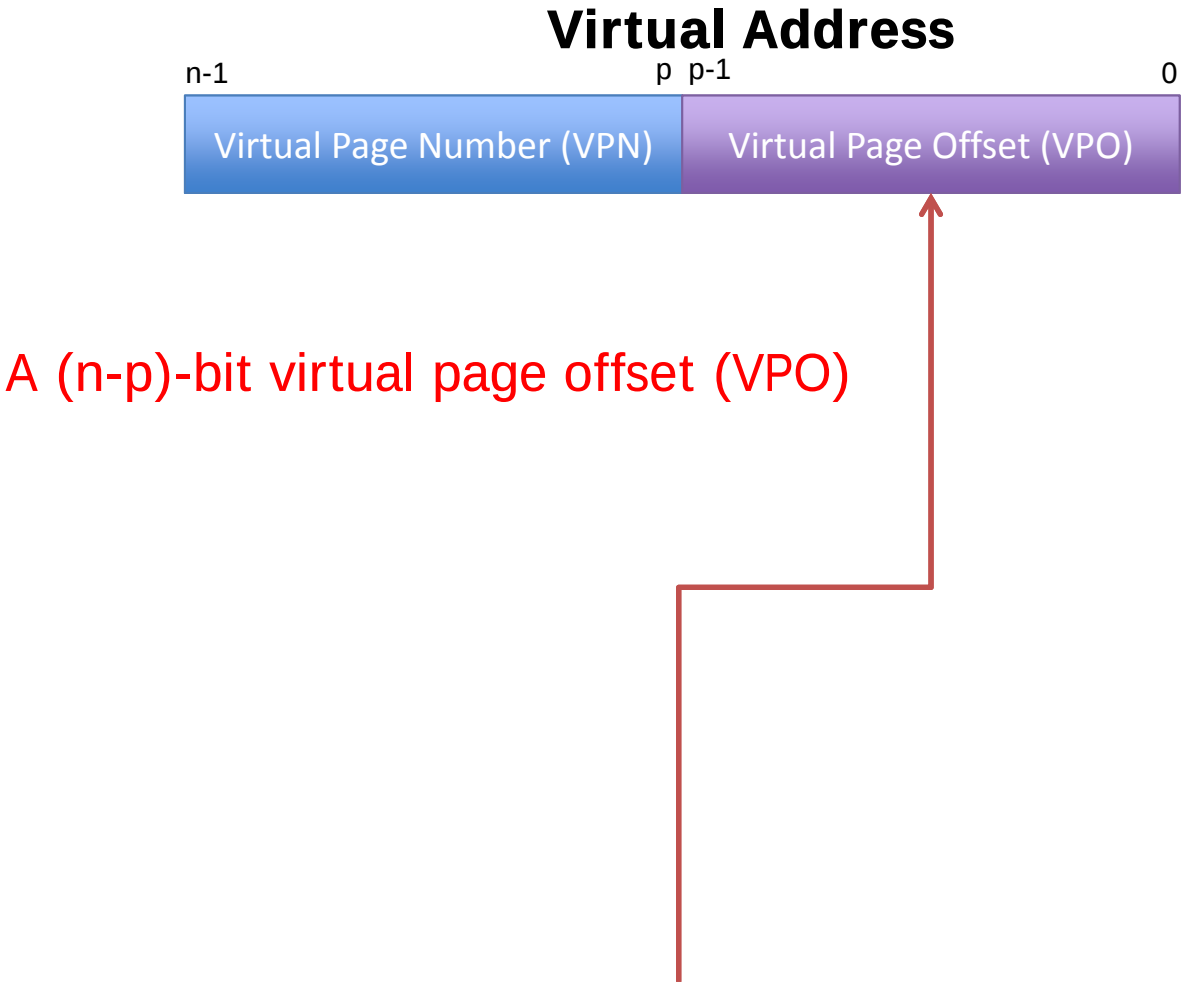


n-bit virtual address has two components

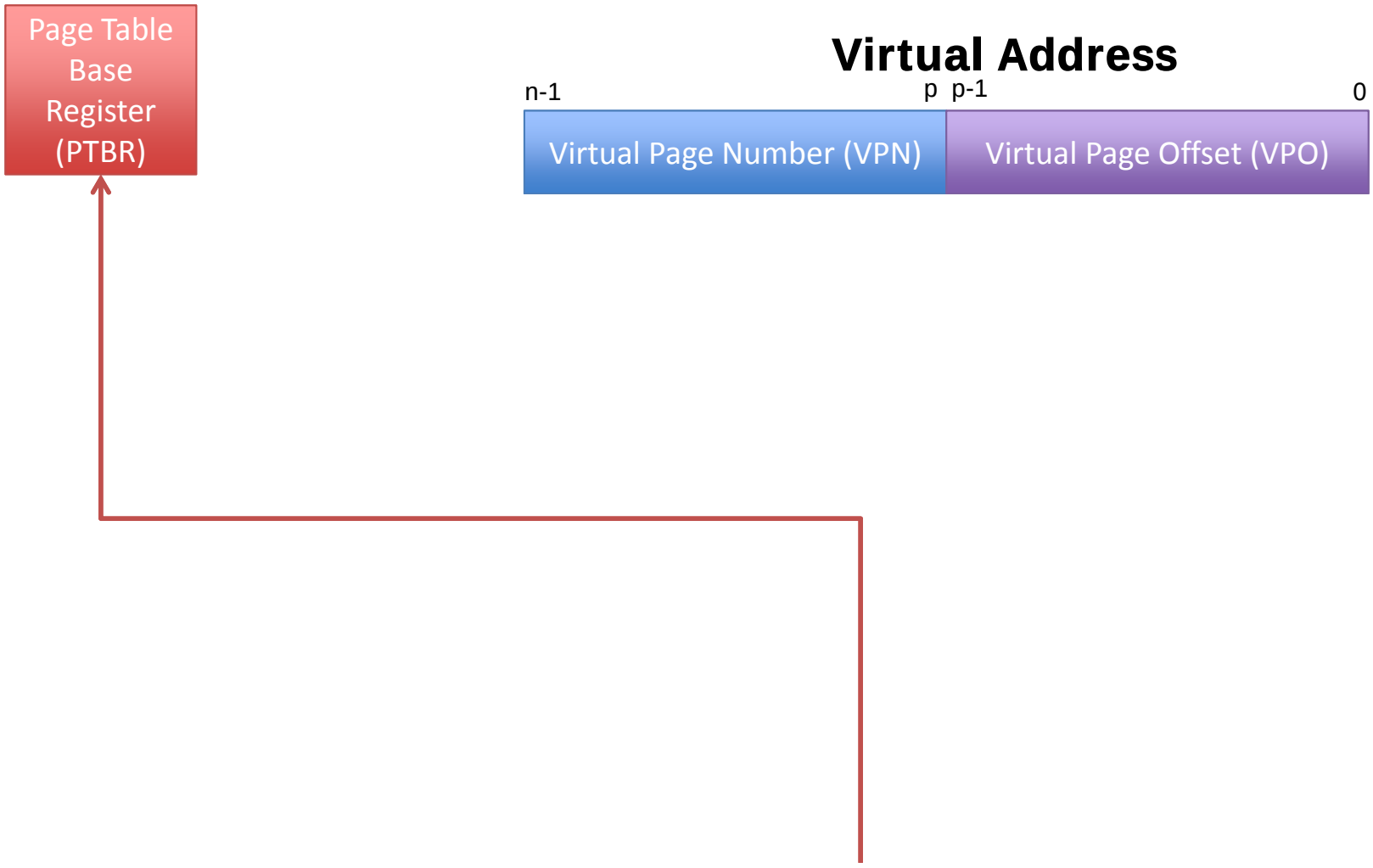
Address Translation



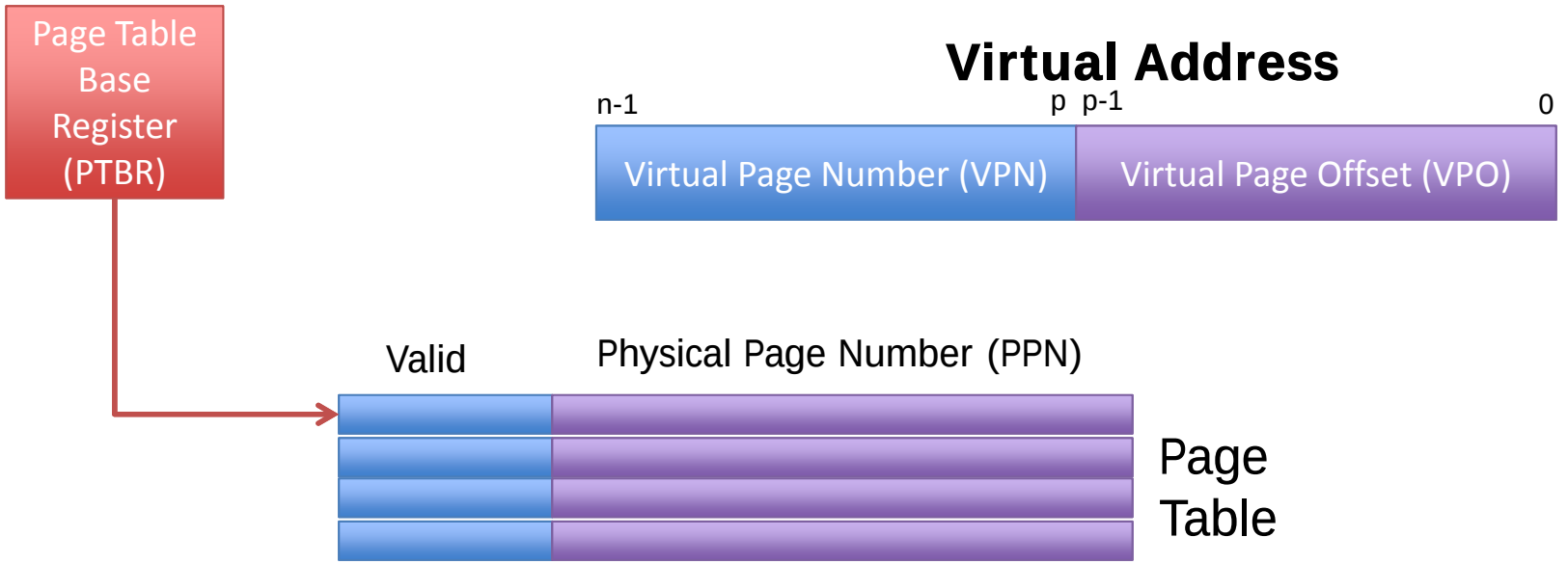
Address Translation



Address Translation

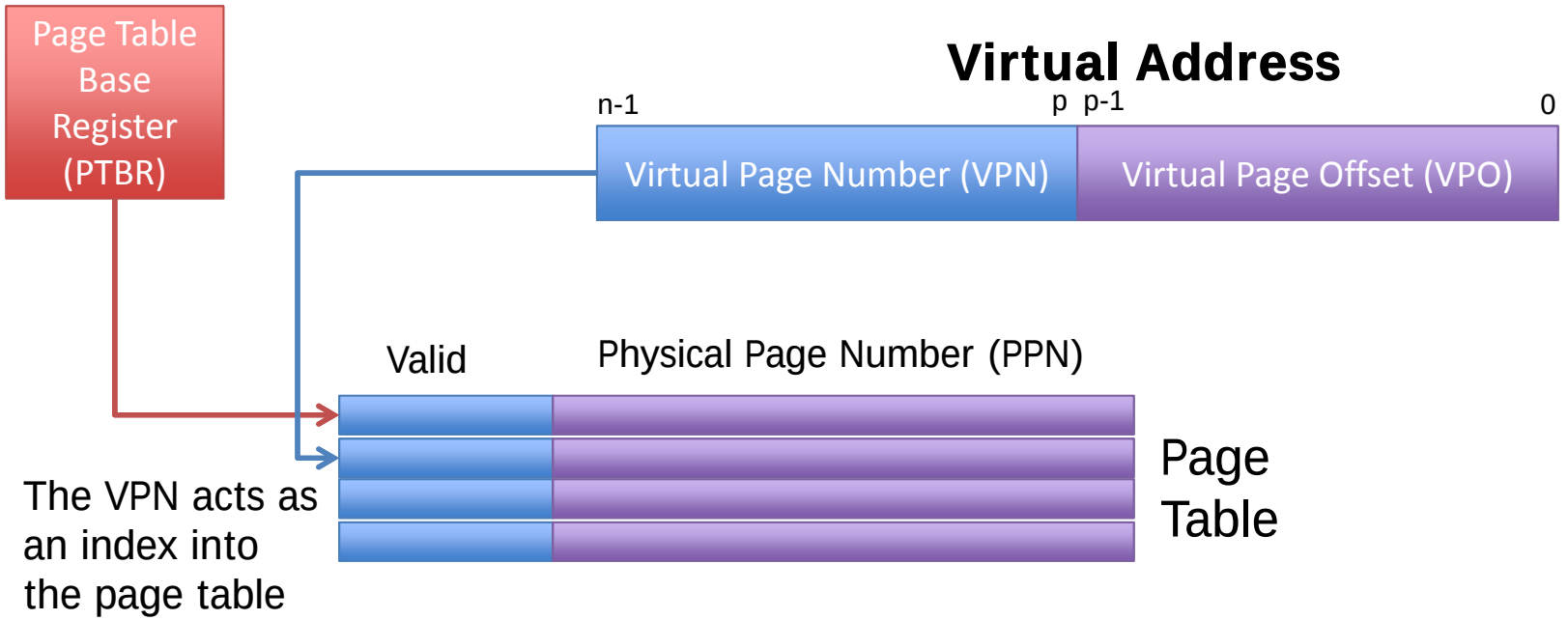


Address Translation



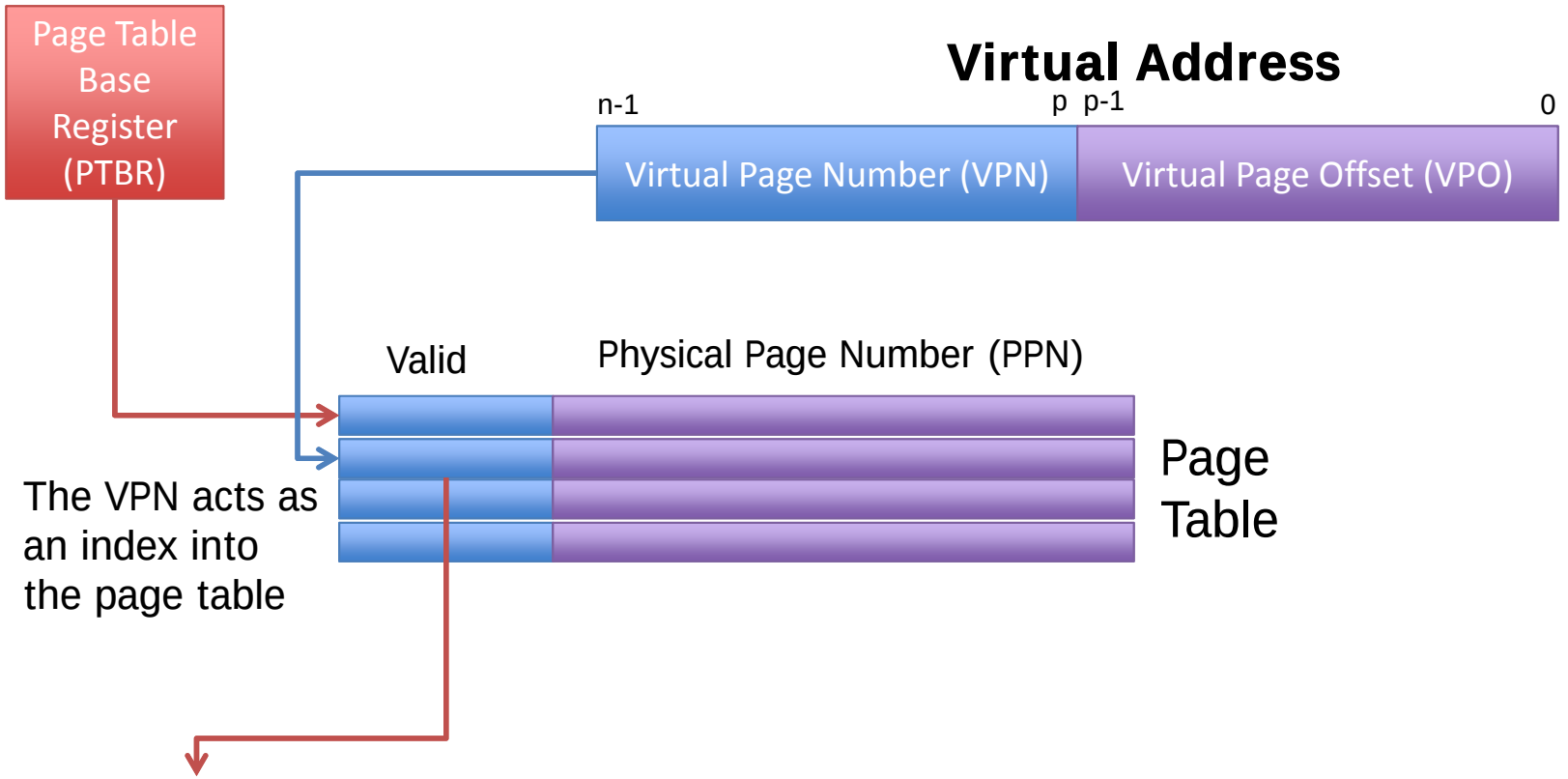
A control register in the CPU points to the current page table

Address Translation



The MMU uses the VPN to select the PTE

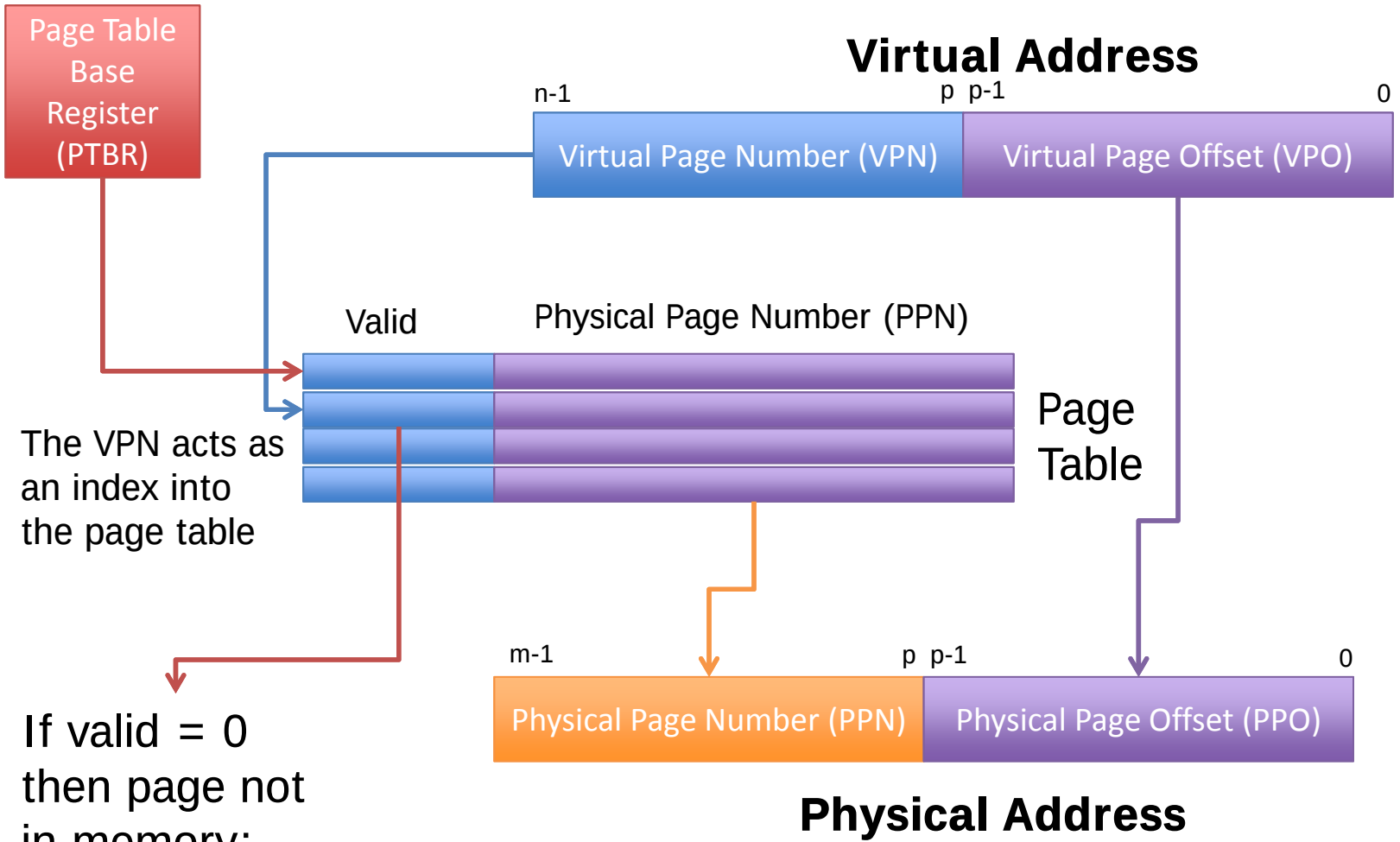
Address Translation



If valid = 0 then page not in memory: page fault

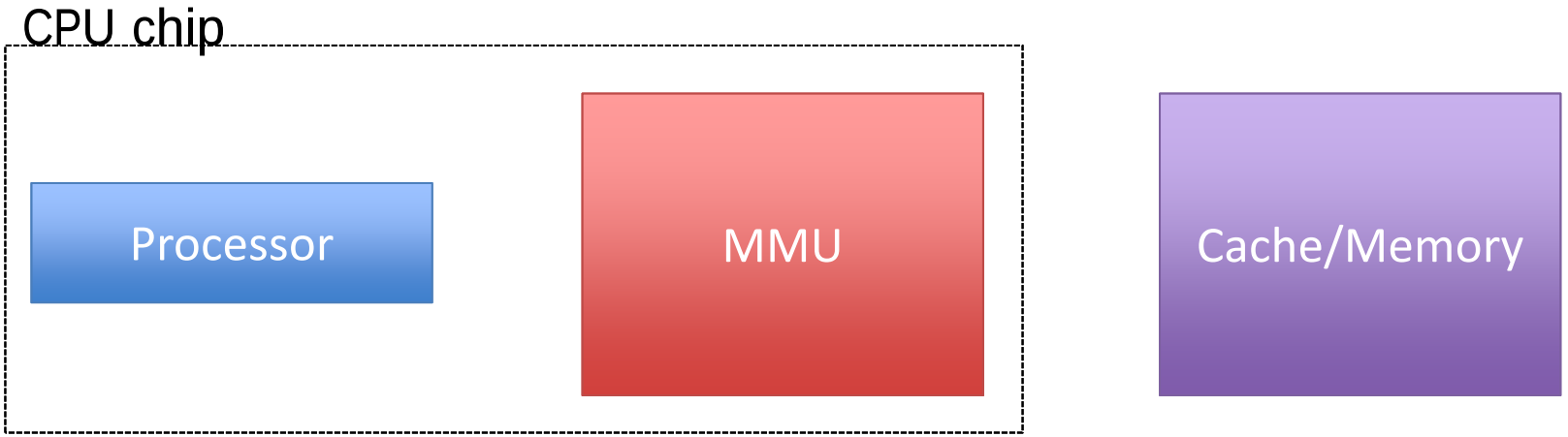
The MMU uses the VPN to select the PTE

Address Translation

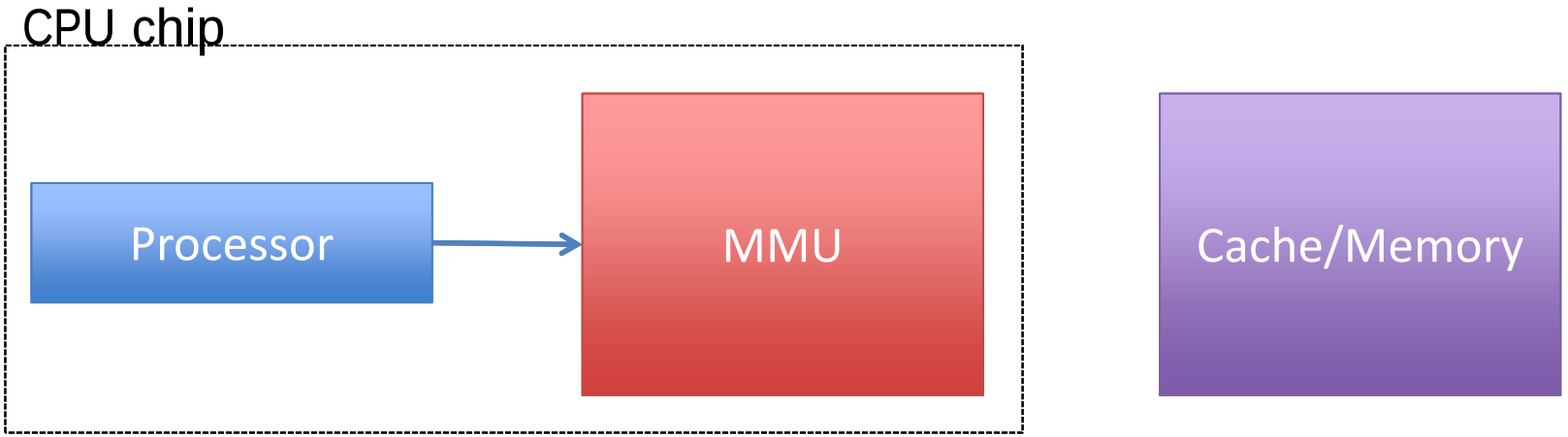


The physical address is created from concatenating PPN with VPO

Page Hit

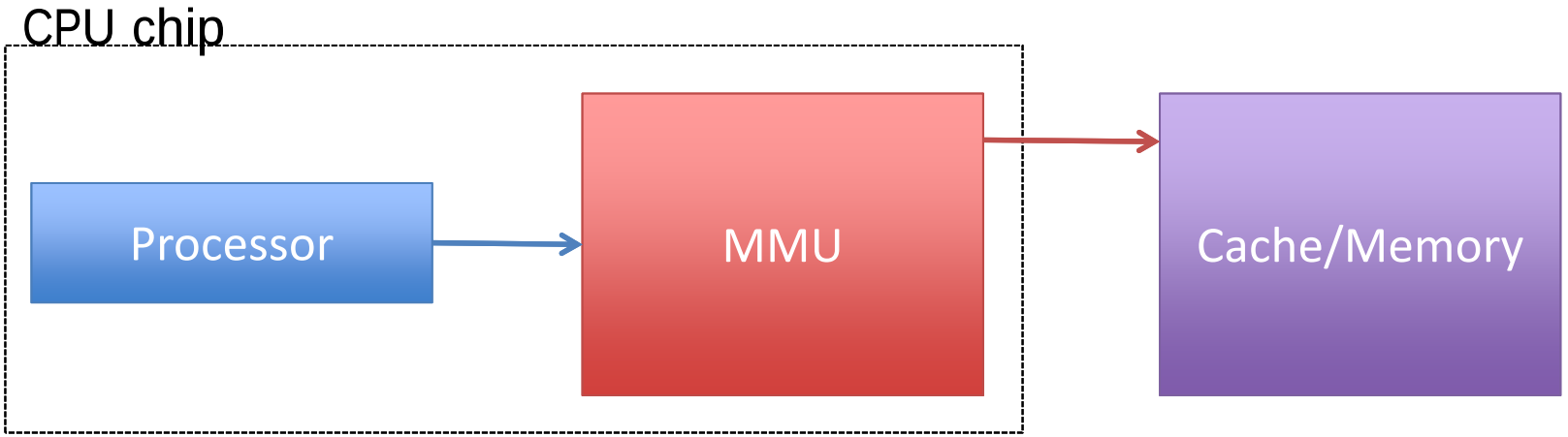


Page Hit



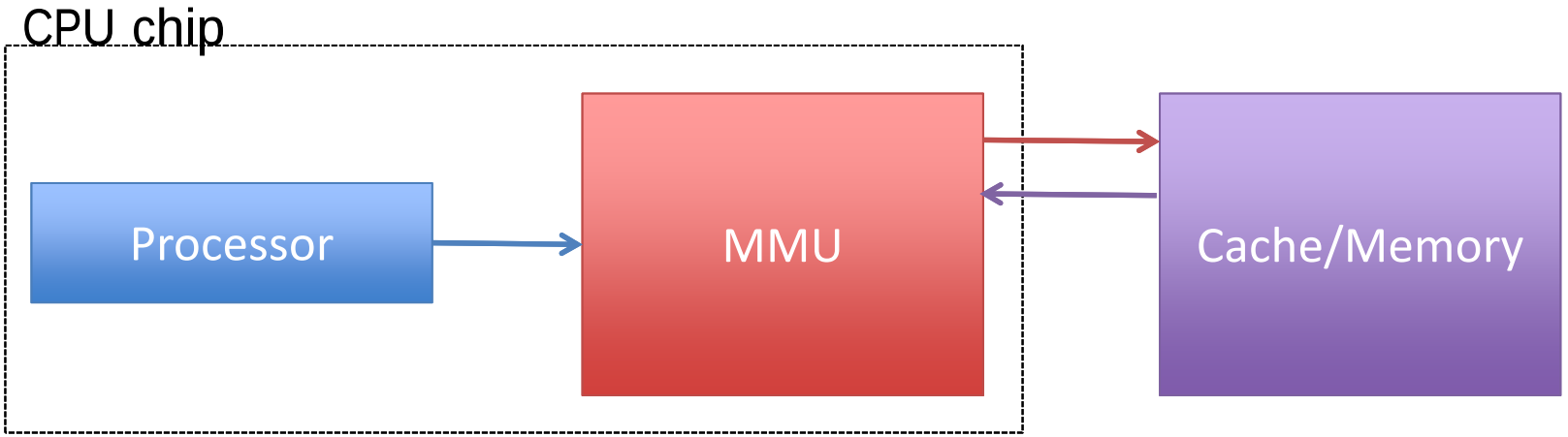
1. **Processor generates VA and sends to MMU**

Page Hit



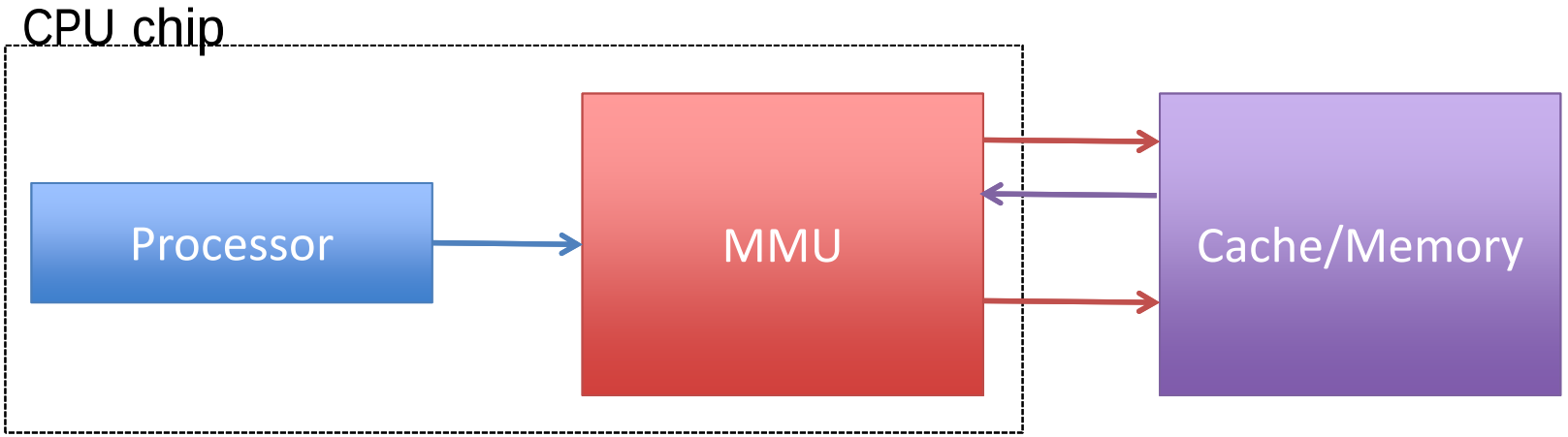
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**

Page Hit



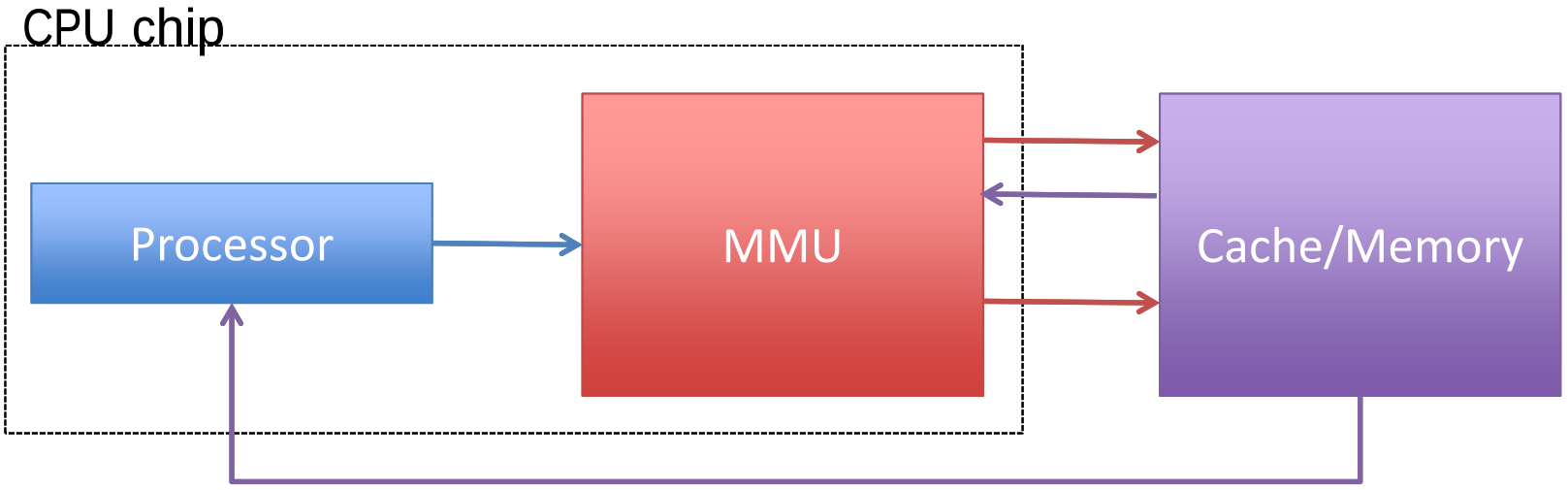
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**

Page Hit



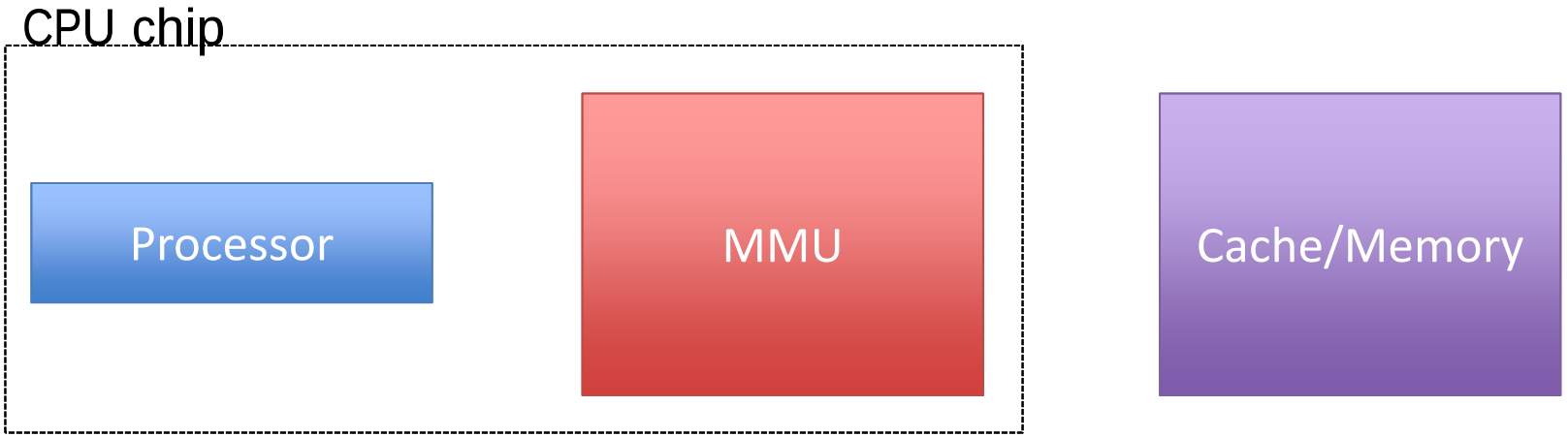
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**
4. **The MMU constructs the PA and sends it to cache/main memory**

Page Hit

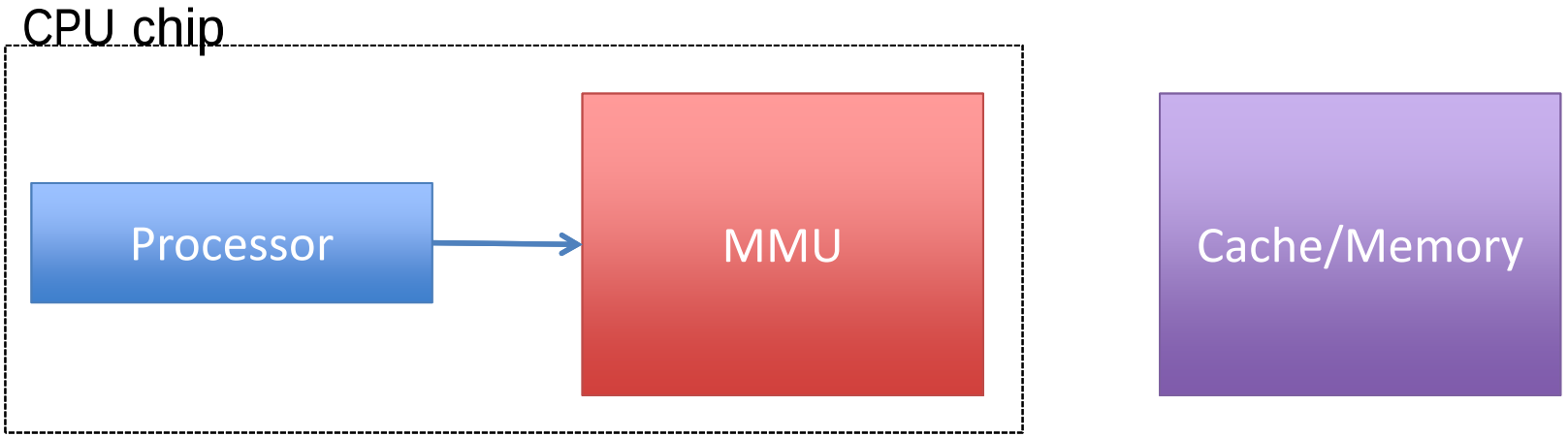


1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**
4. **The MMU constructs the PA and sends it to cache/main memory**
5. **The cache/main memory returns the requested data to the processor**

Page Miss

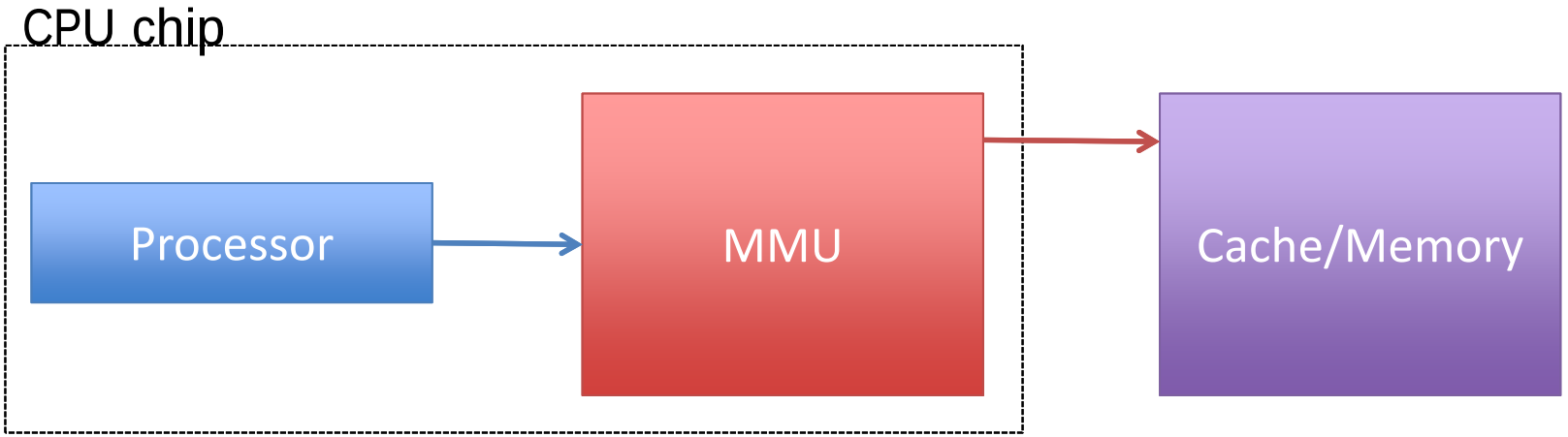


Page Miss



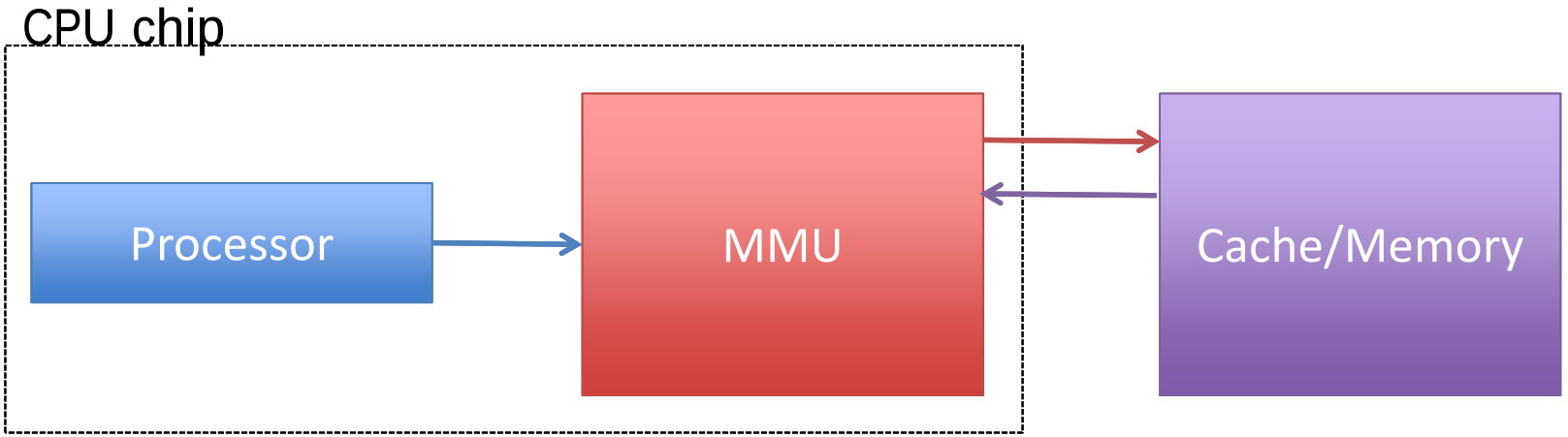
1. **Processor generates VA and sends to MMU**

Page Miss



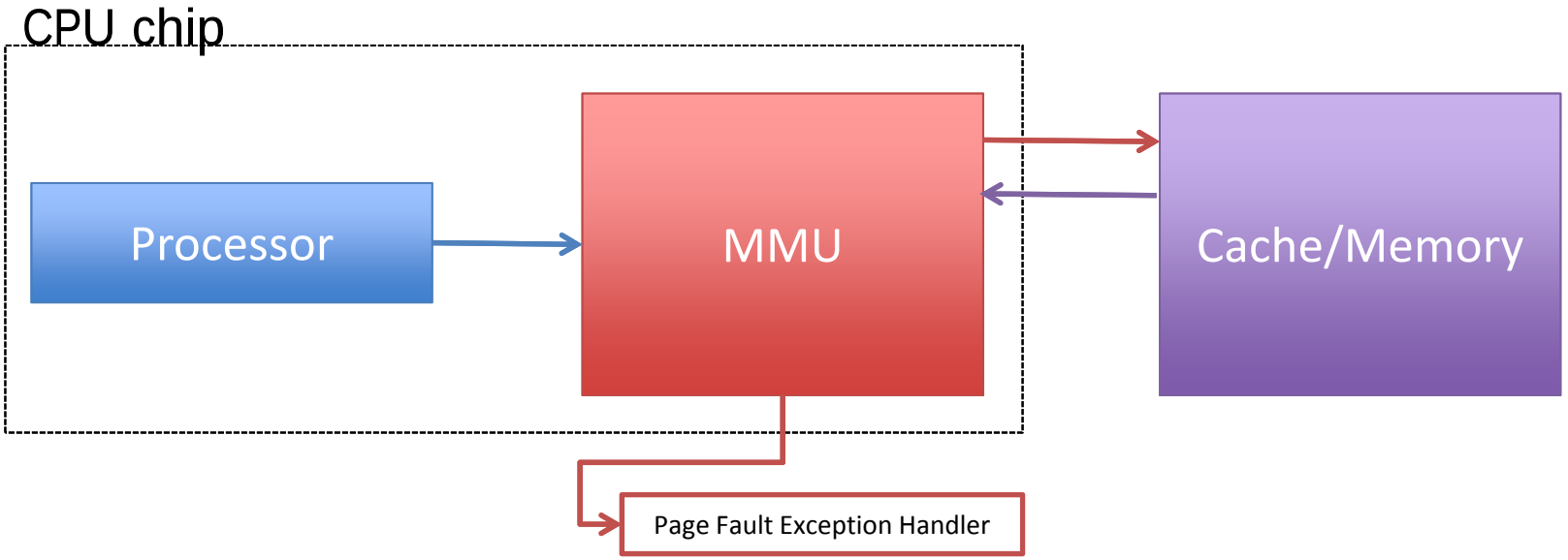
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**

Page Miss



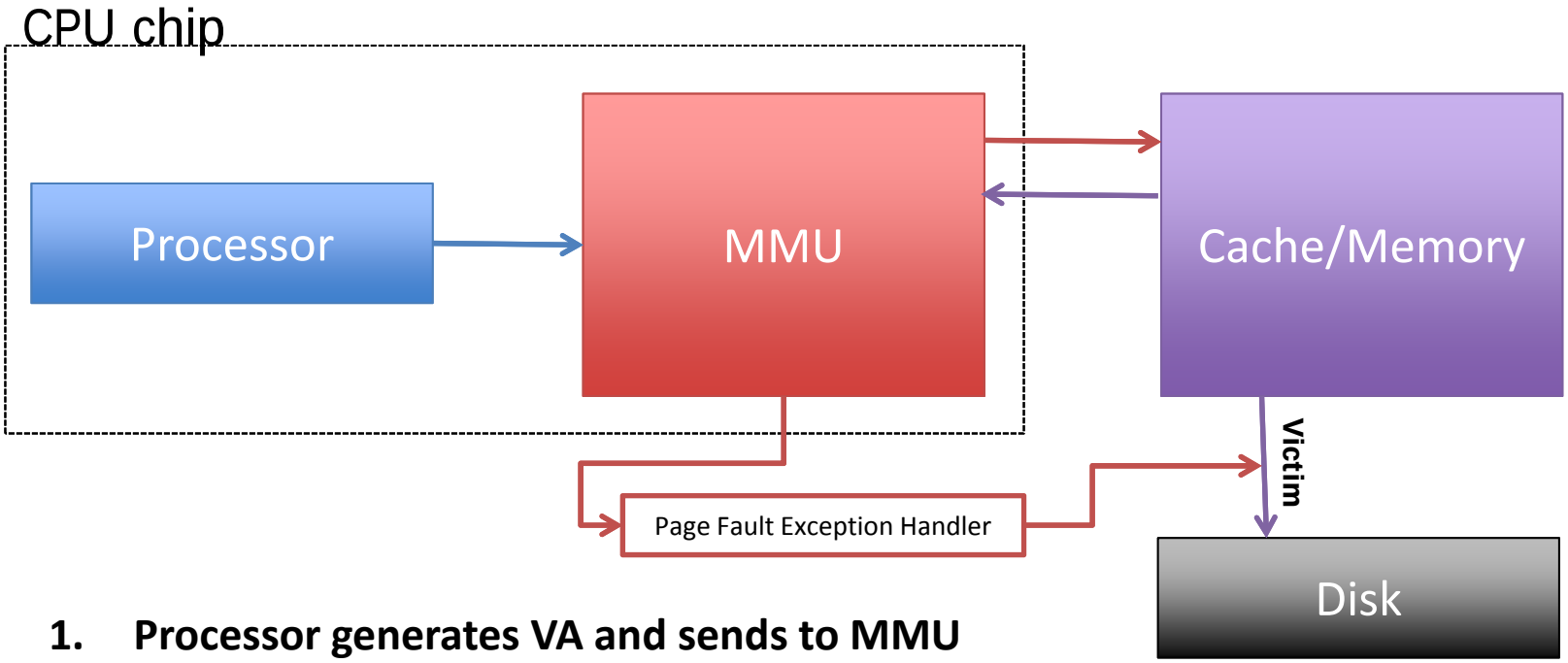
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**

Page Miss



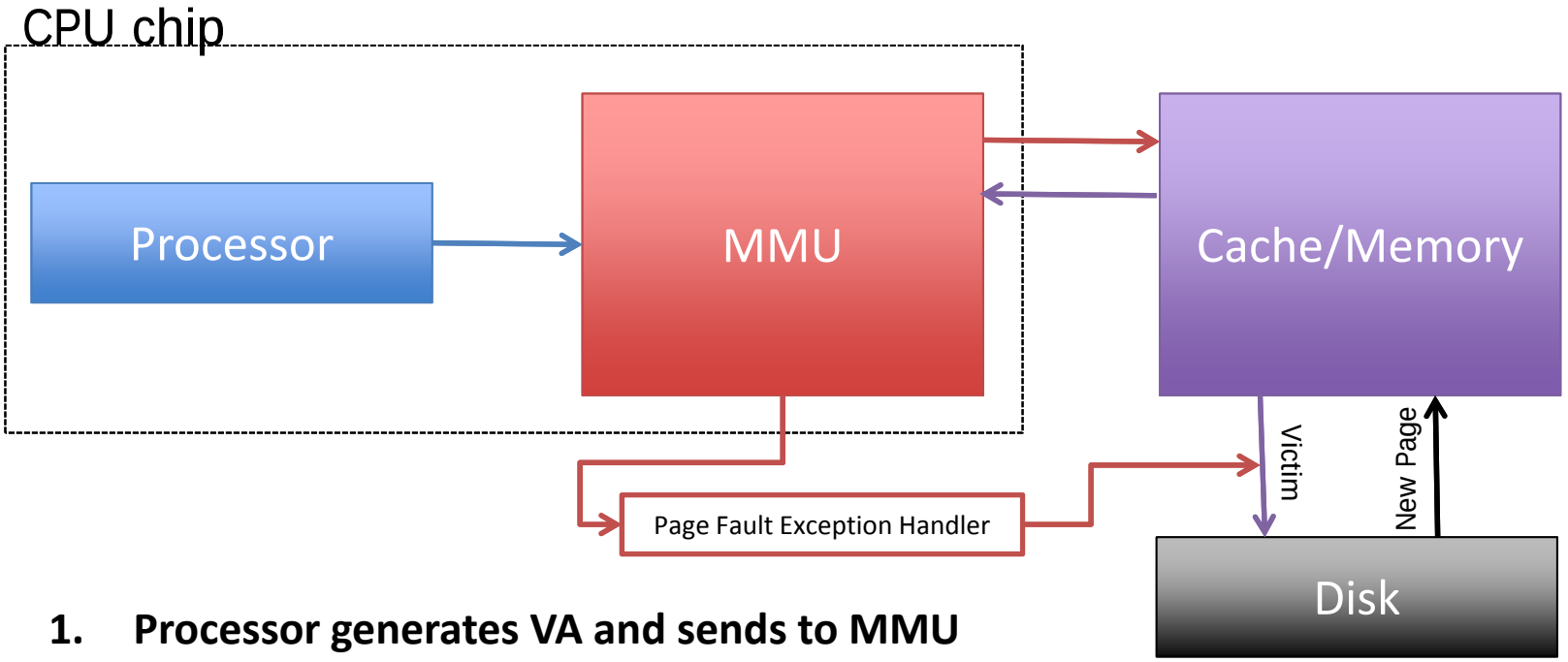
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**
4. **Valid bit is 0, triggers exception, CPU transfers control to fault handler**

Page Miss



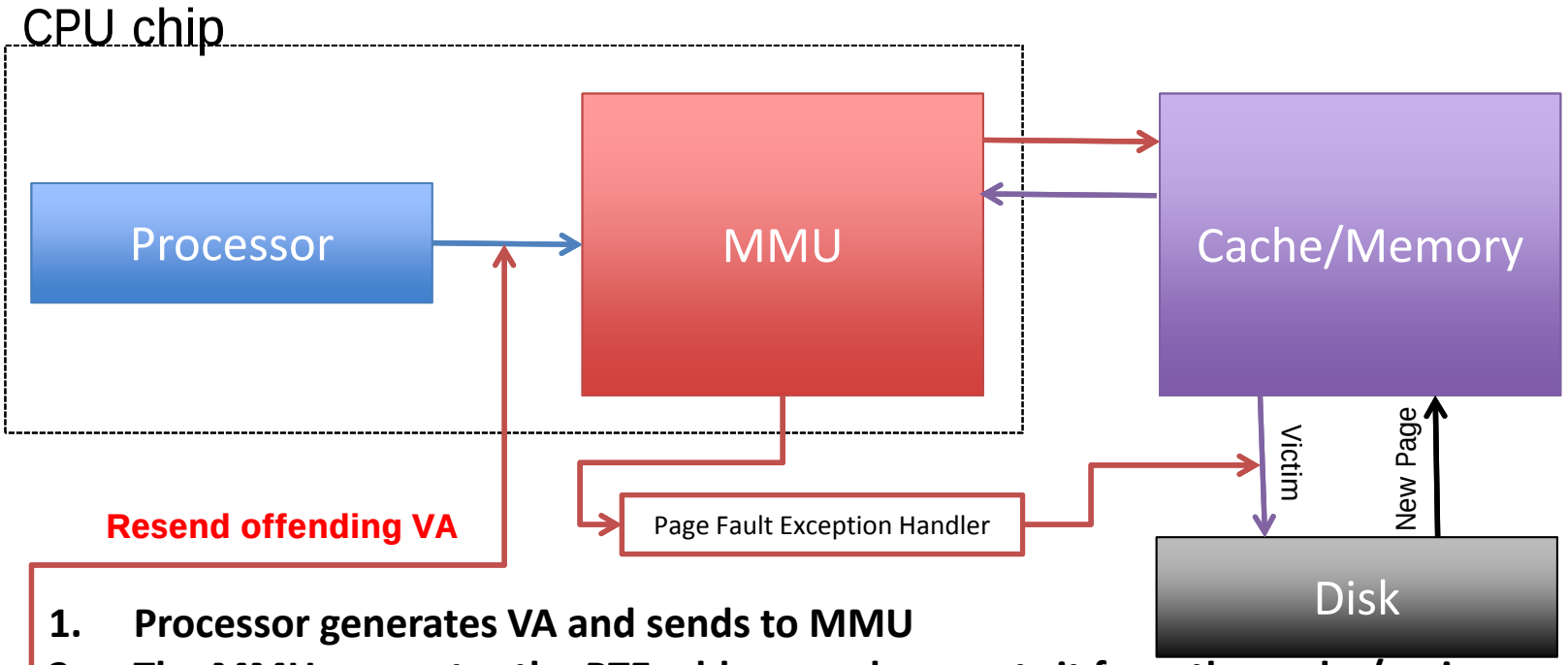
1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**
4. **Valid bit is 0, triggers exception, CPU transfers control to fault handler**
5. **Fault handler identifies victim page and sends to disk (if modified)**

Page Miss



1. **Processor generates VA and sends to MMU**
2. **The MMU generates the PTE address and requests it from the cache/main memory**
3. **The cache/main memory returns the PTE to the MMU**
4. **Valid bit is 0, triggers exception, CPU transfers control to fault handler**
5. **Fault handler identifies victim page and sends to disk (if modified)**
6. **Fault handler pages in new page and updates PTE in memory**

Page Miss



1. Processor generates VA and sends to MMU
2. The MMU generates the PTE address and requests it from the cache/main memory
3. The cache/main memory returns the PTE to the MMU
4. Valid bit is 0, triggers exception, CPU transfers control to fault handler
5. Fault handler identifies victim page and sends to disk (if modified)
6. Fault handler pages in new page and updates PTE in memory
7. Fault handler returns to process and restarts offending instruction

Previous slides were a (small) lie …

- **Most caches use the *virtual address***
  - Cache access can happen during translation
  - Gives address in case of a cache *miss*
- **PTEs also include *access protection* bits**
  - Indicate whether read, write, execute allowed on bytes of that page
- **PTEs also include *dirty* bit**
  - To know if a page needs write back to disk
- **Caches need scrubbing when swapping**

… but these are just refinements

- **Basic idea is to use main memory as a cache for pages, whose "natural home" is disk**
  - Also called the swap file
- **Address translation can be sped up …**
  - Using a *cache* of PTEs!
  - Usually fully associative
  - Called a *translation lookaside buffer* (TLB)
  - Separate ones for instructions and data, similar to split L1 instruction and data caches: ITLB, DTLB

But what about SIGSEGV?

- **A process needs to *ask* for most pages to be allocated**
  - Stack will grow automatically, within bounds
  - Otherwise, **mmap** (memory map) system call
- **Access to an unasked-for page → SIGSEGV**
  - An exception (fault) causes and OS response
  - Only *sometimes* is that response a *signal*
- **Improper access to a protected page?**
  - Also SIGSEGV, reflected as signal to process