Cache Memory and Performance

Many of the following slides are taken with permission from

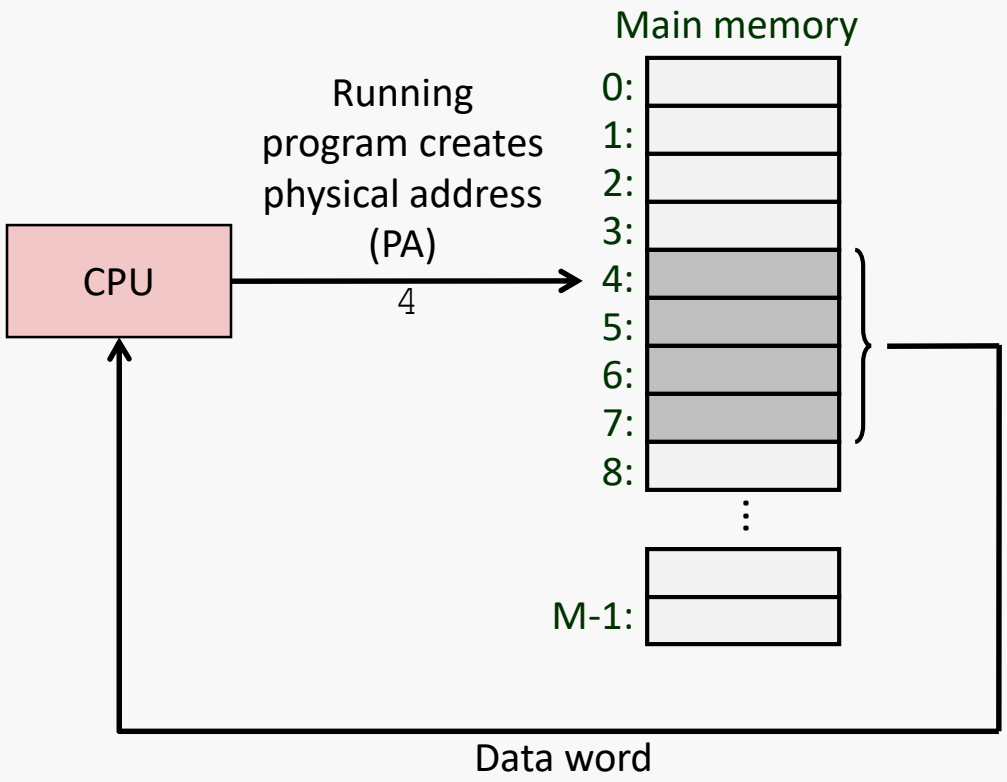
Complete Powerpoint Lecture Notes for
Computer Systems: A Programmer's Perspective (CS:APP)

Randal E. Bryant *and* *David R. O'Hallaron*

http://csapp.cs.cmu.edu/public/lectures.html

The book is used explicitly in CS 2505 and CS 3214 and as a reference in CS 2506.

Physical Memory Addressing



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

Shortcomings

Early systems used physical addressing

- each program kept its entire memory space in DRAM
- limited the number of programs that could be "active" at once
- limited absolute size of program's memory space to size of DRAM
- provided no natural support for address protection

Critical observations: during any interval of time that a program is being executed

- the program will (most likely) access only a small part of its instructions
- the program will (most likely) access only a small part of its data

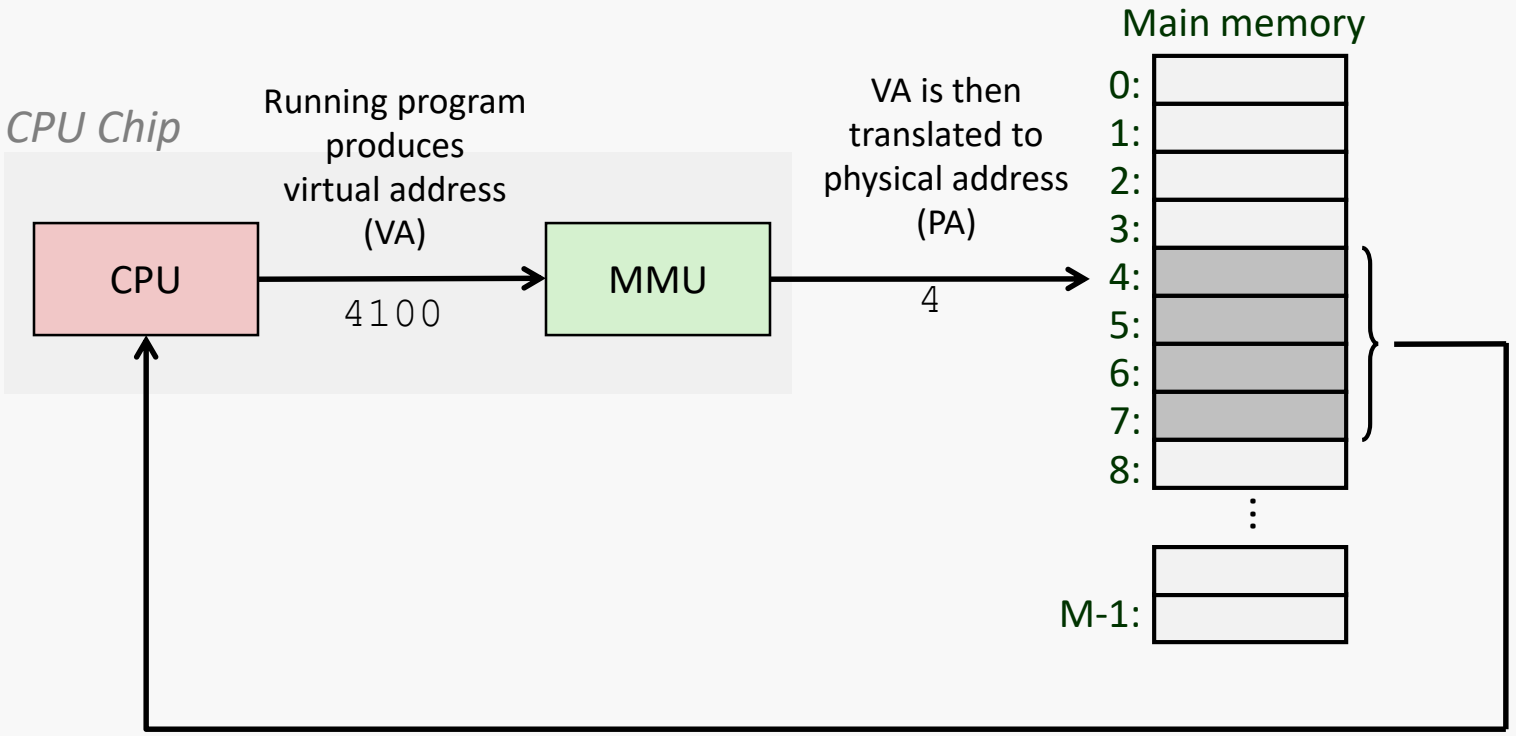
Virtual Memory

Use main memory as a “cache” for secondary (disk) storage
- Managed jointly by CPU hardware and the operating system (OS)

Programs share main memory (DRAM)
- Each gets a private virtual address space holding its code and data
- DRAM holds its frequently-used code and data
- Protected from other programs

CPU and OS translate virtual addresses to physical addresses
- VM “block” is called a page
- VM translation “miss” is called a page fault

Virtual Memory



Used in all modern servers, laptops, and smart phones

One of the great ideas in computer science

Virtual Memory

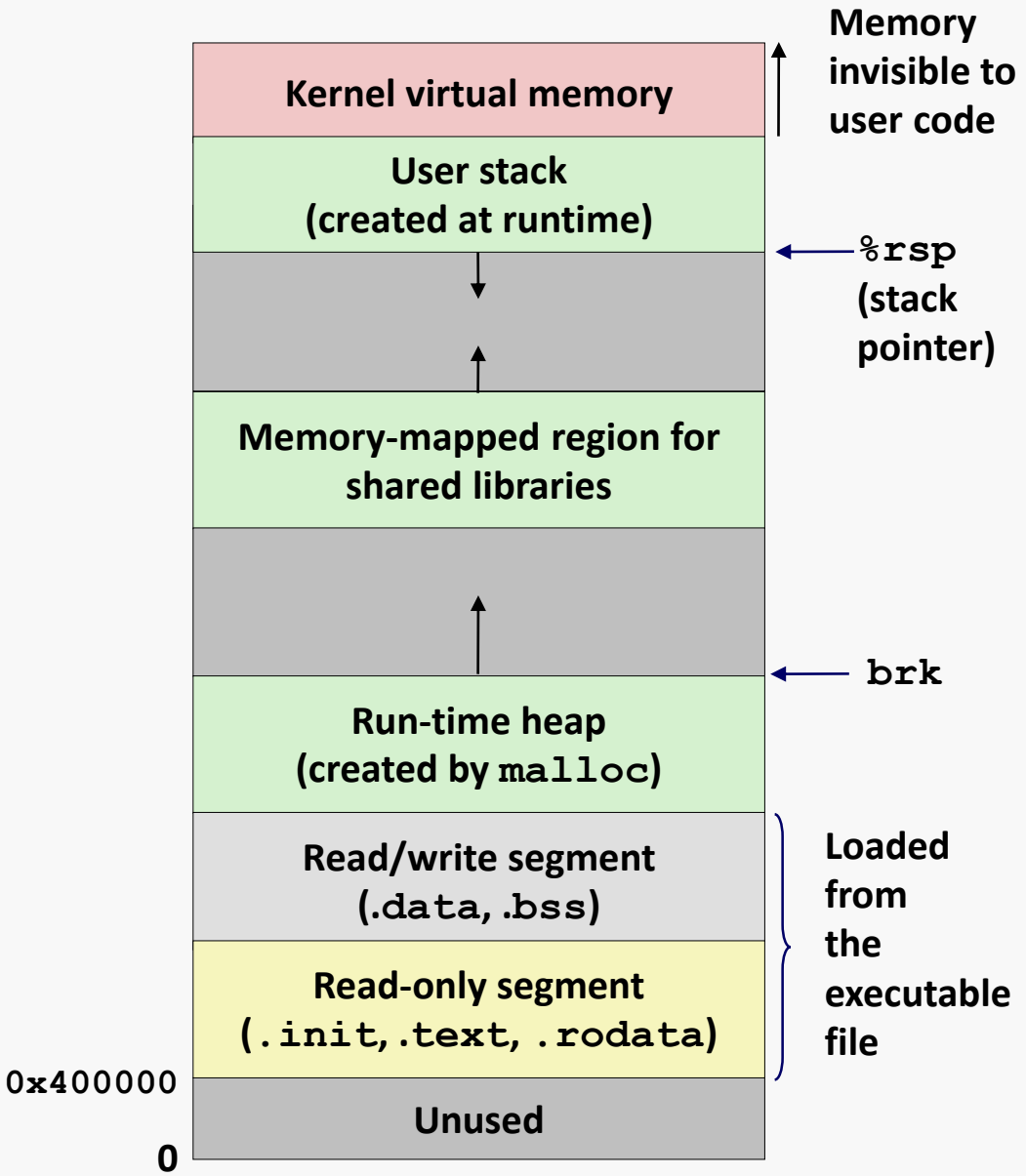
Aside: when you use gdb, you are seeing virtual addresses, not physical addresses:

```
198         csvEntry* newEntry = createCSVEntry(pCSVData);

(gdb) n
203         int32_t foundIdx = findCSVEntry(pList, newEntry);

(gdb) p newEntry
$6 = (csvEntry *) 0x605380

(gdb) p *newEntry
$7 = {CRN = 0x605450 "12958",
      ID = 0x605490 "000000000",
      Name = 0x6054b0 "Hokie, James Robert",
      PID = 0x6054d0 "joebobhokie",
      PPID = 0x6054f0 "",
      totalScore = 88,
      nScores = 4,
      Scores = 0x6053c0}
```

Process Virtual Memory Image

OS maintains:

- structure of each process's address space,
- which addresses are valid,
- what do they refer to,
- even those that aren't in main memory currently



Paging to/from Disk

Idea: hold only those data in physical memory that are actually accessed by a process

Maintain map for each process

{ virtual addresses } $\rightarrow$ { physical addresses } $\cup$ { disk addresses }

OS manages mapping, decides which virtual addresses map to physical (if allocated) and which to disk

Disk addresses include:

- Executable .text, initialized data
- Swap space (typically lazily allocated)
- Memory-mapped (mmap'd) files (see example)

Demand paging: bring data in from disk lazily, on first access

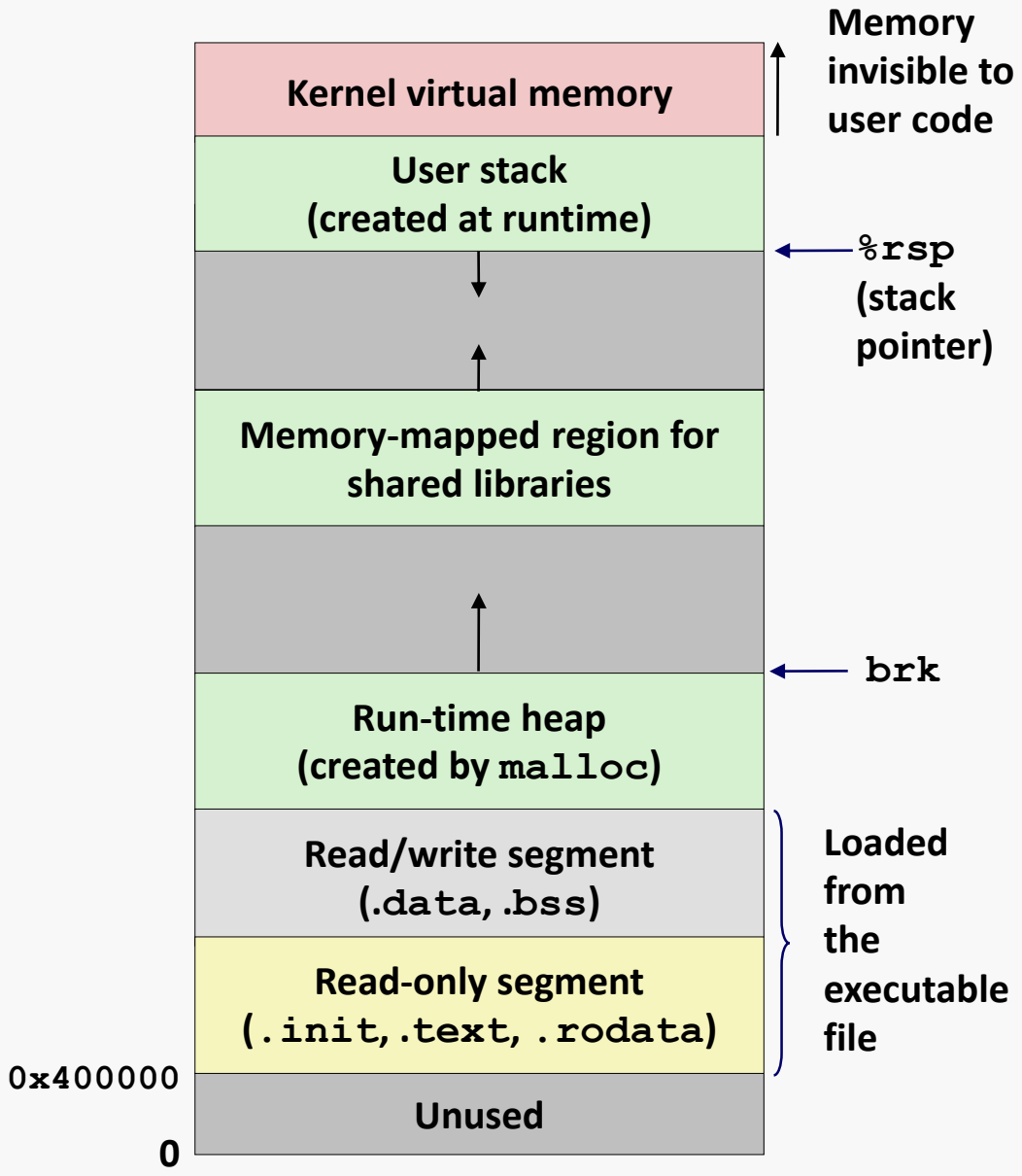
- Unbeknownst to application

Process Virtual Memory Image

"Virtual" space exists on secondary storage

Virtual space is divided into fixed-size "pages"

Virtual pages are copied into DRAM as needed

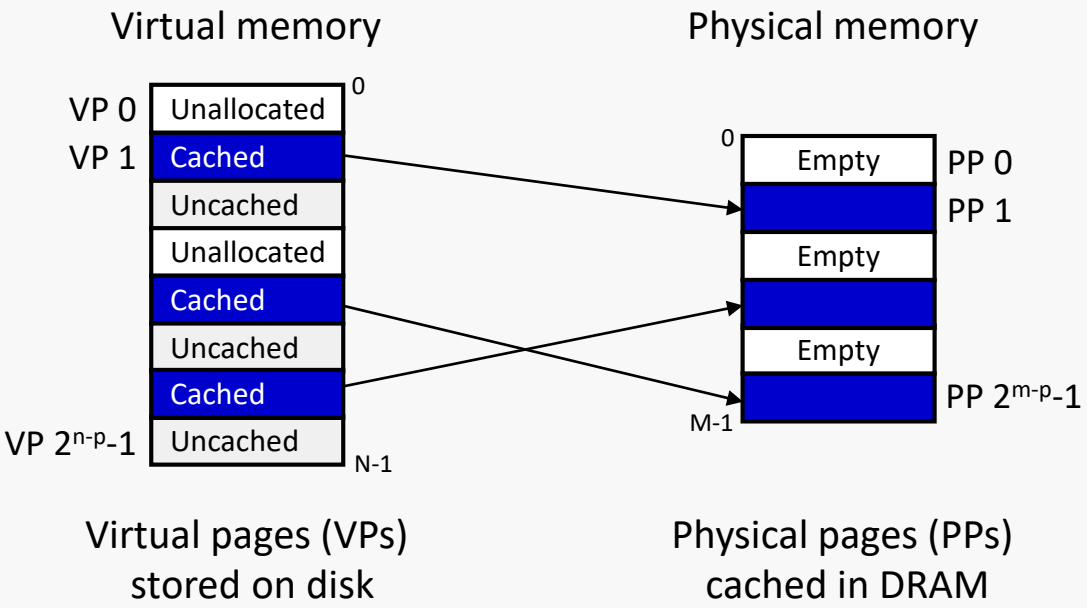


VM as a Tool for Caching

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 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
- Disk is about ***10,000x*** slower than DRAM

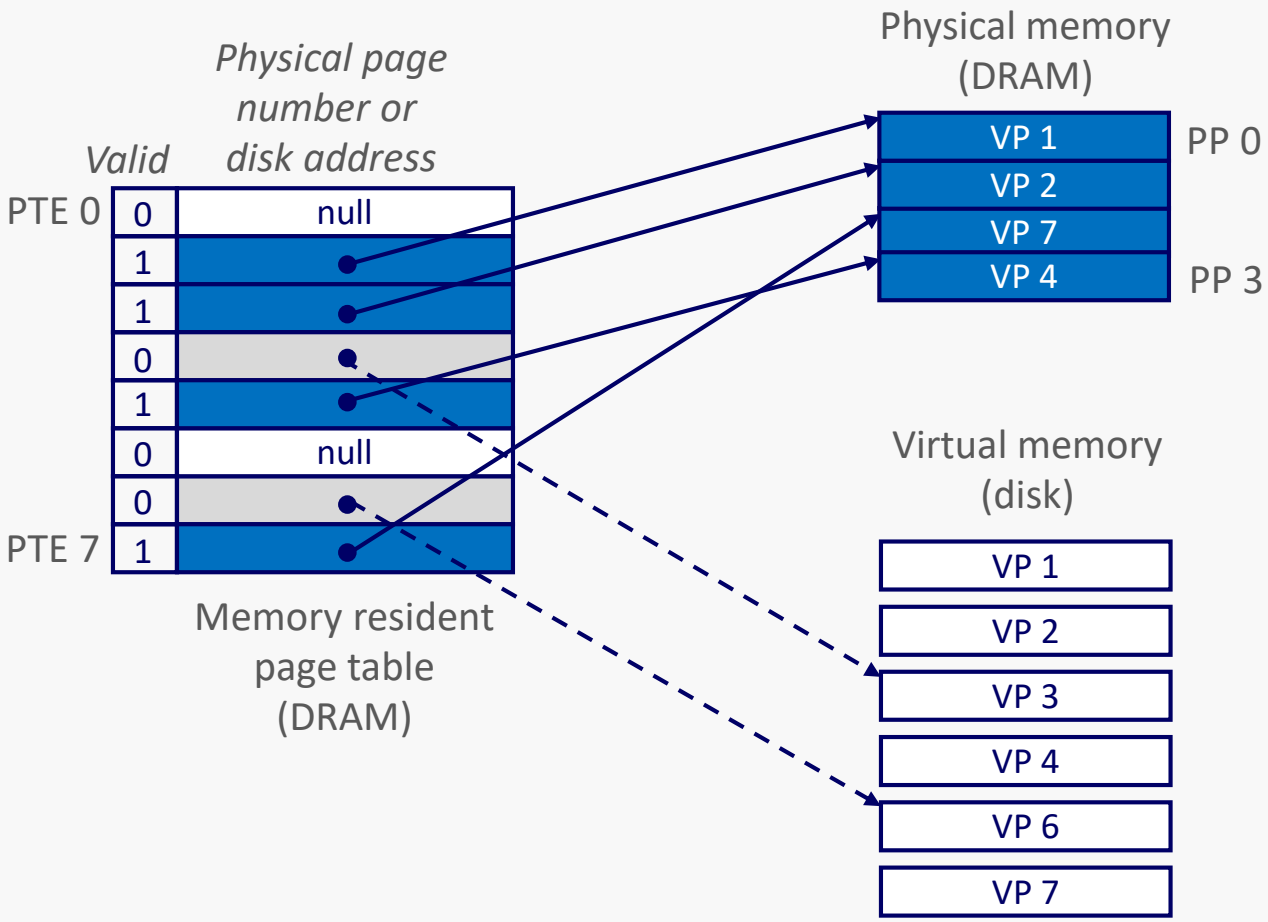
Consequences

- 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

Page Table Enables Address Mapping

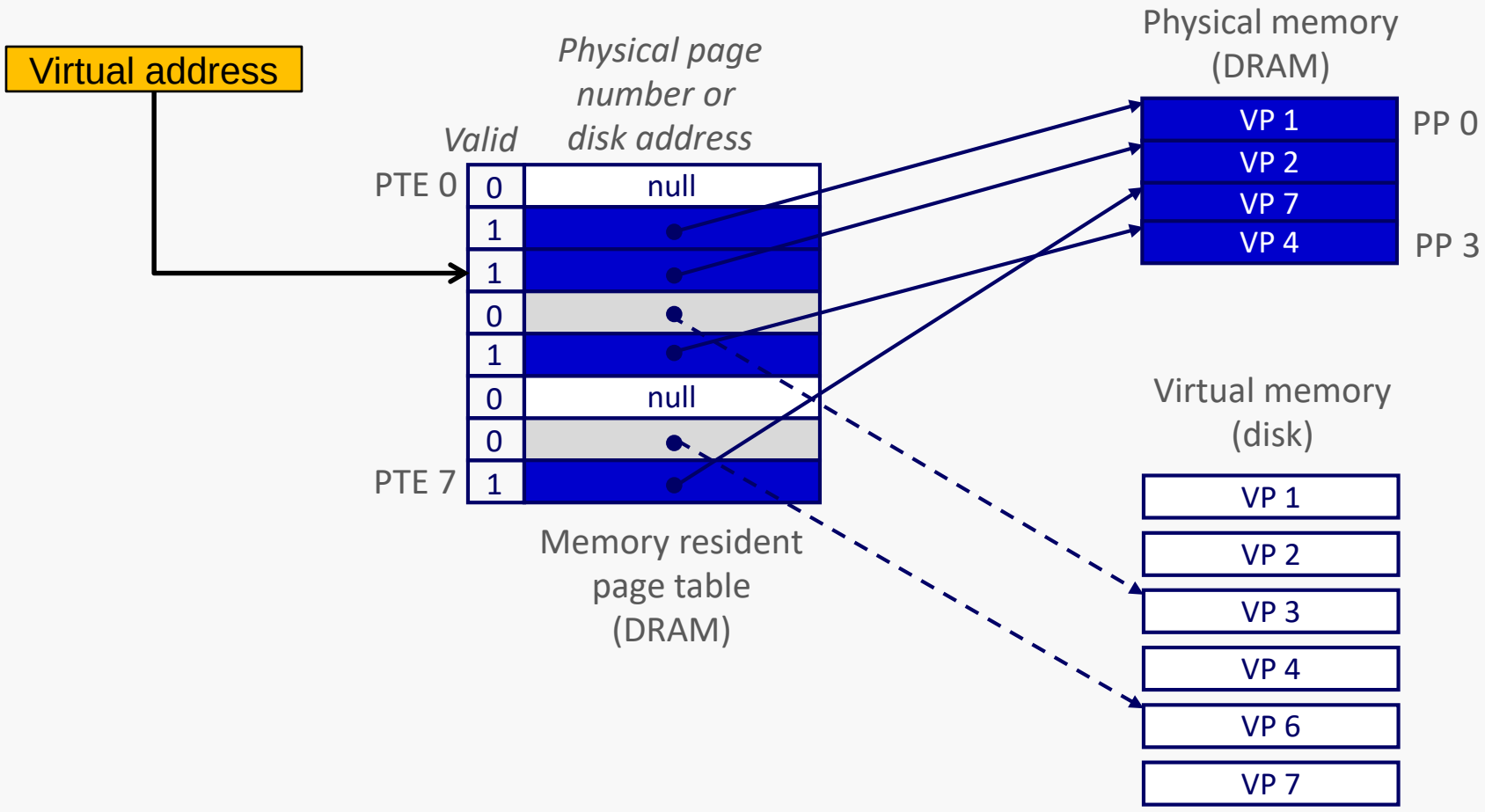
Page table: an array of page table entries (PTEs) that maps virtual pages to physical pages.

Per-process kernel data structure in DRAM



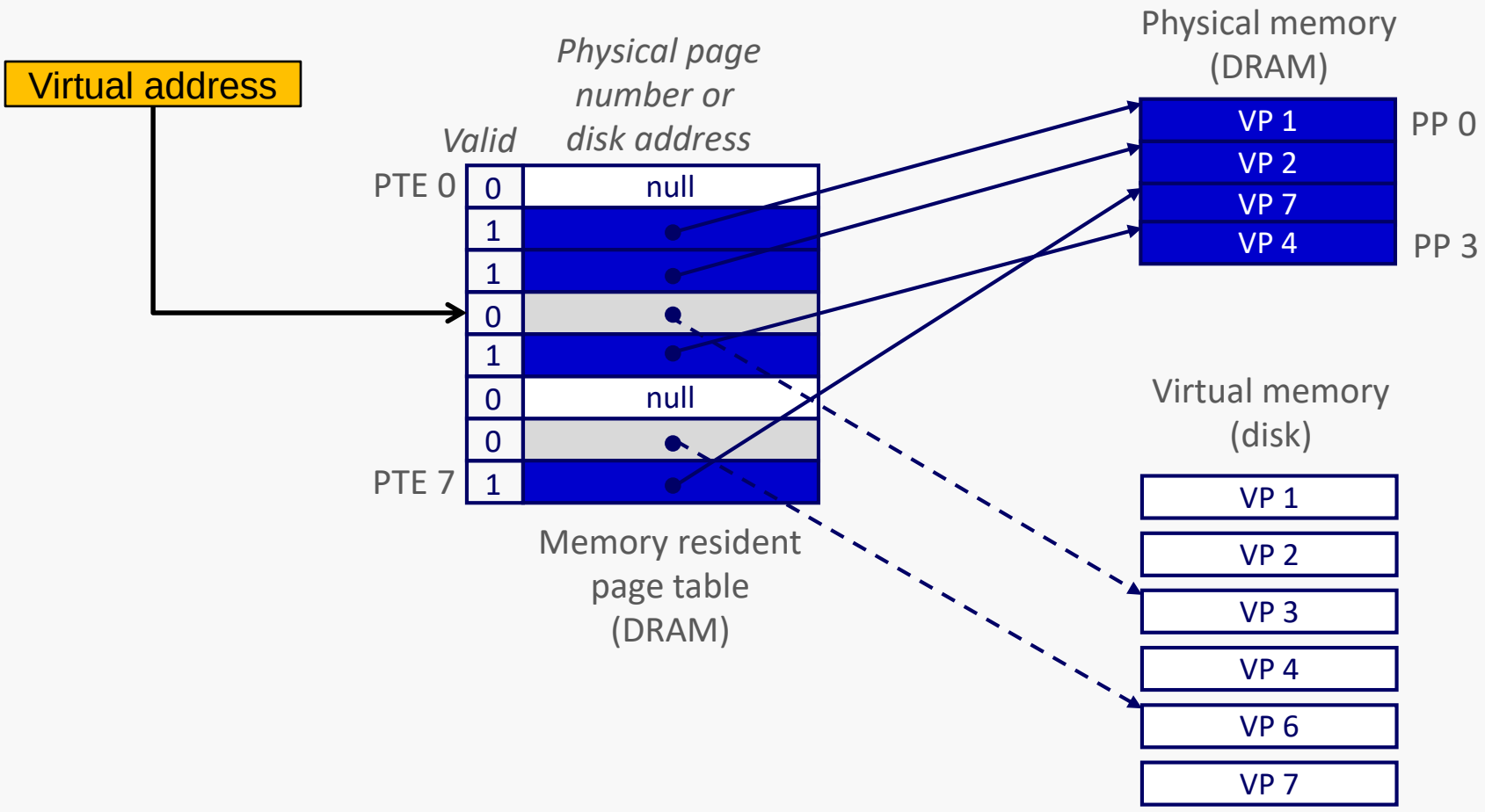
Page Hit

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



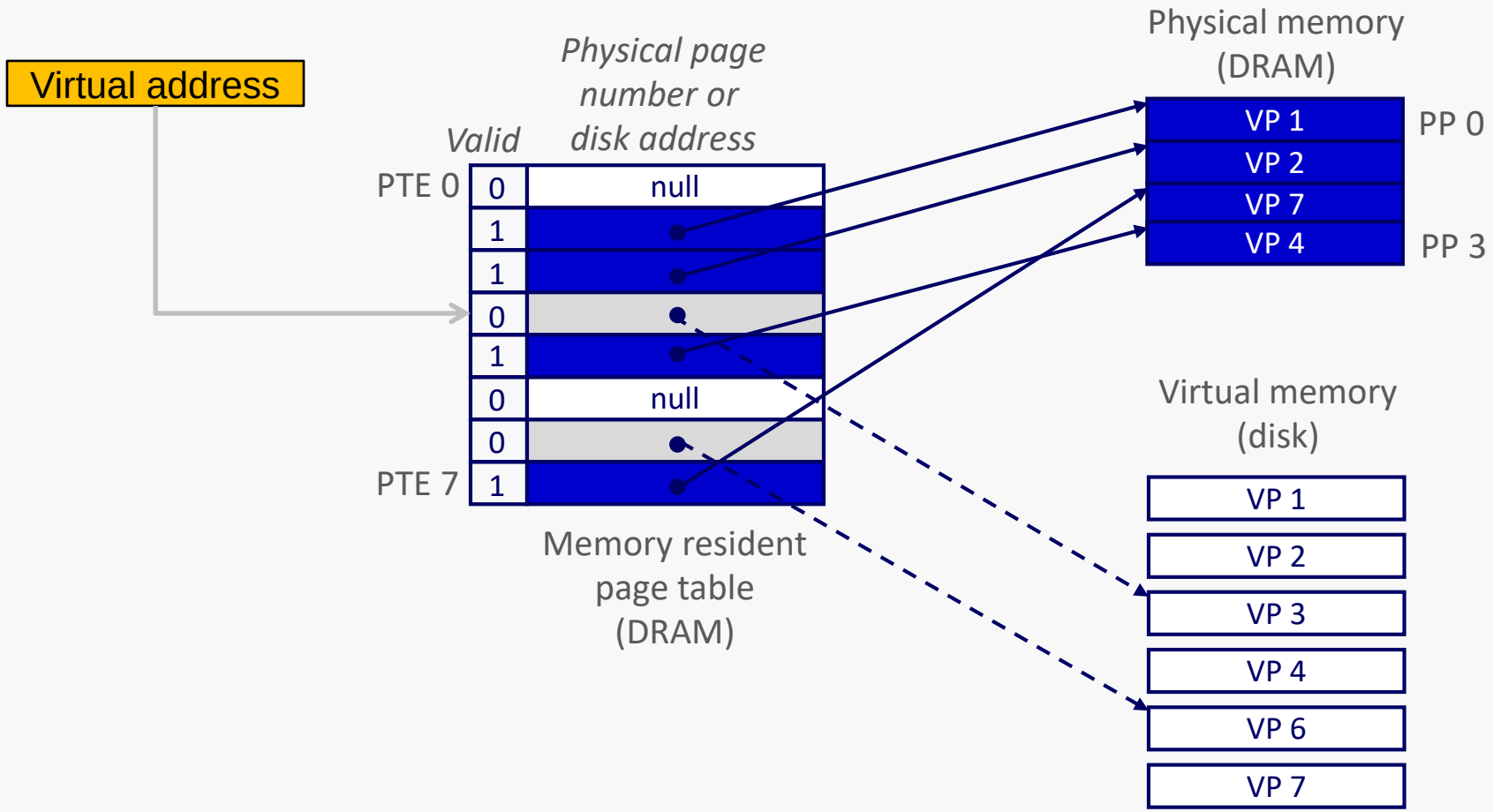
Page Fault

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



Handling Page Fault

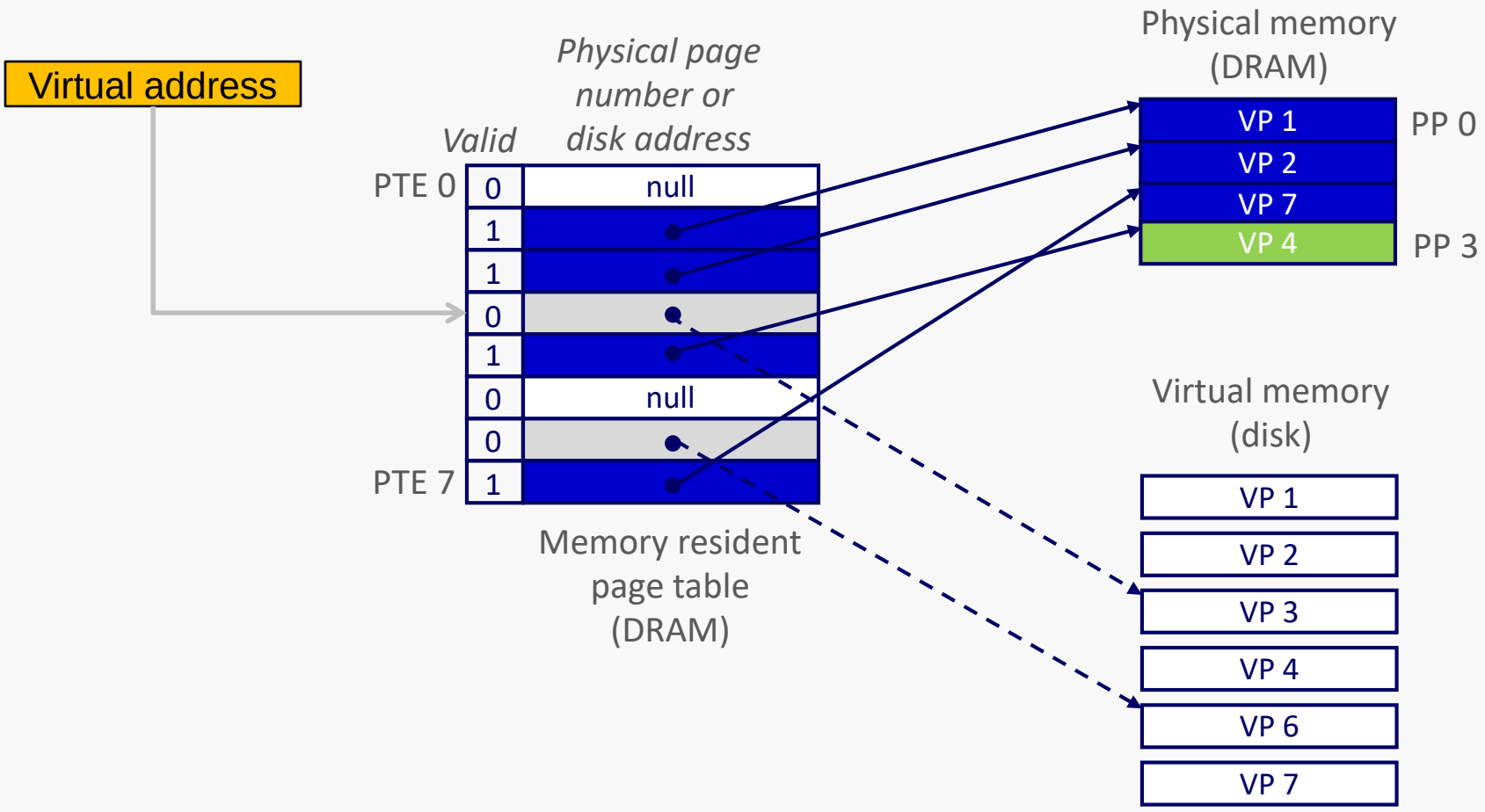
Page miss causes page fault (an exception)



Handling Page Fault

Page miss causes page fault (an exception)

Page fault handler selects a victim to be evicted (here VP 4)



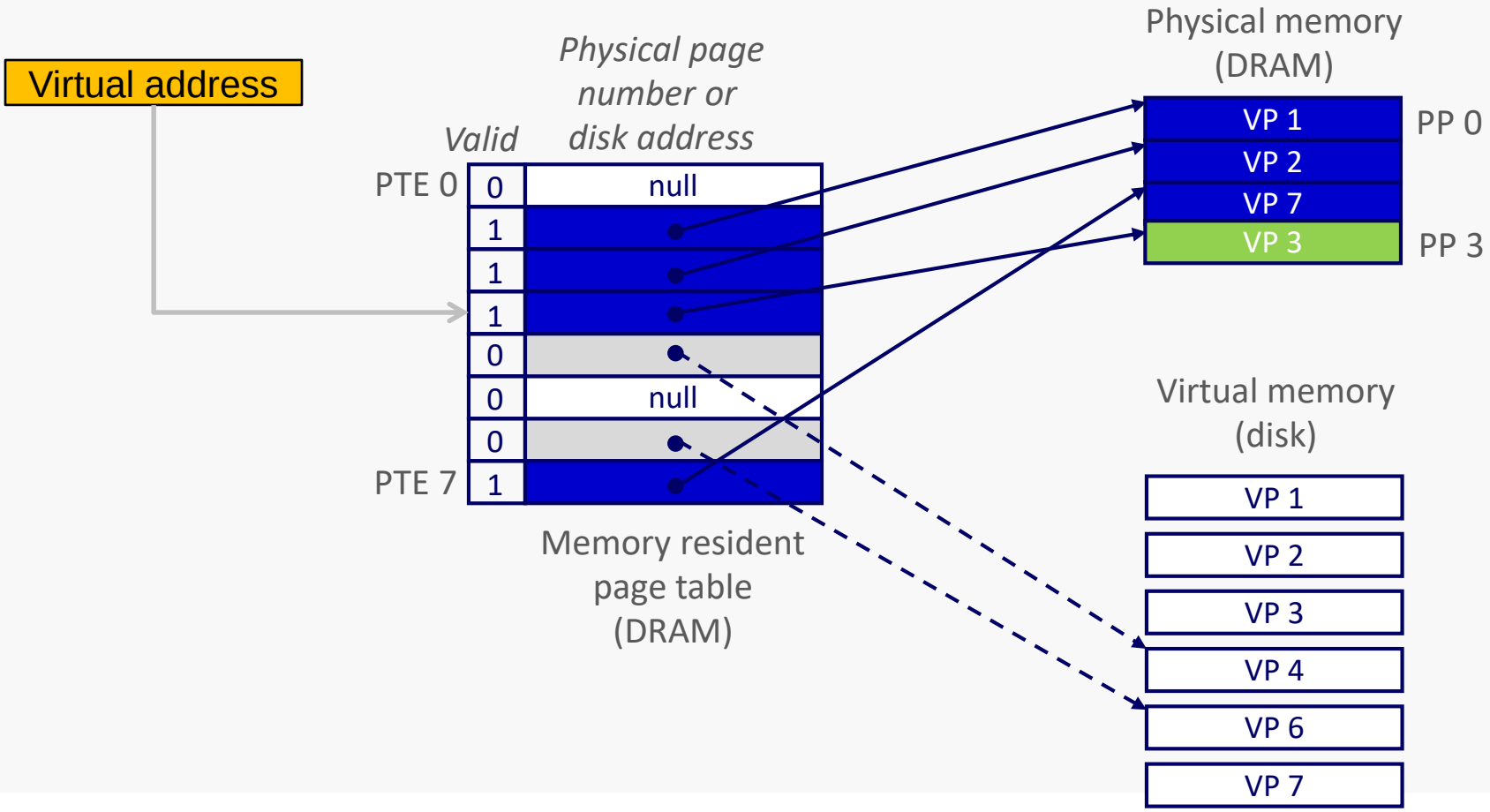
Handling Page Fault

Page miss causes page fault (an exception)

Page fault handler selects a victim to be evicted

Missed VM page (here VP 3) is copied from disk to PM (here PP 3)

Page table is updated

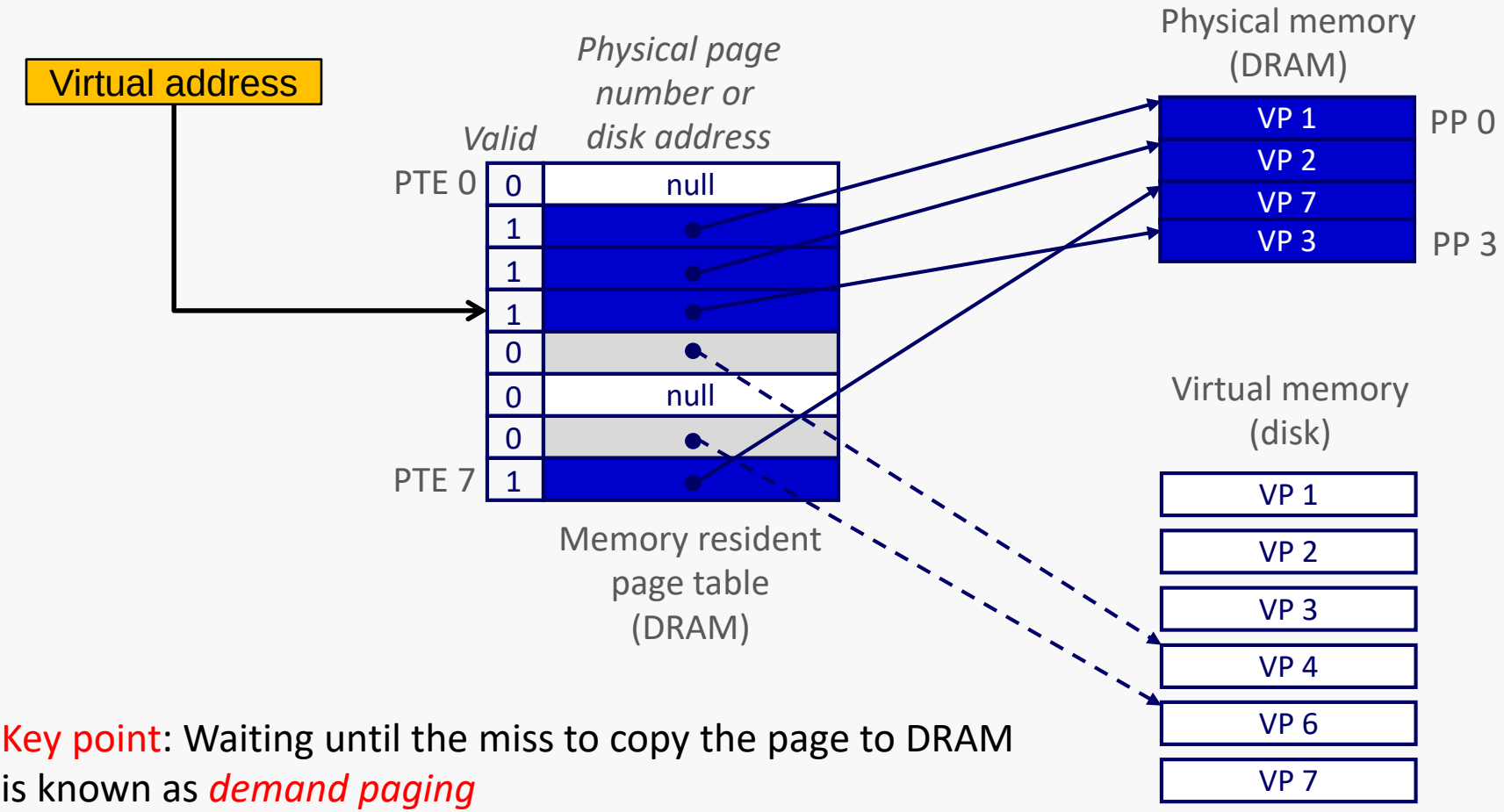


Handling Page Fault

Page miss causes page fault (an exception)

. . .

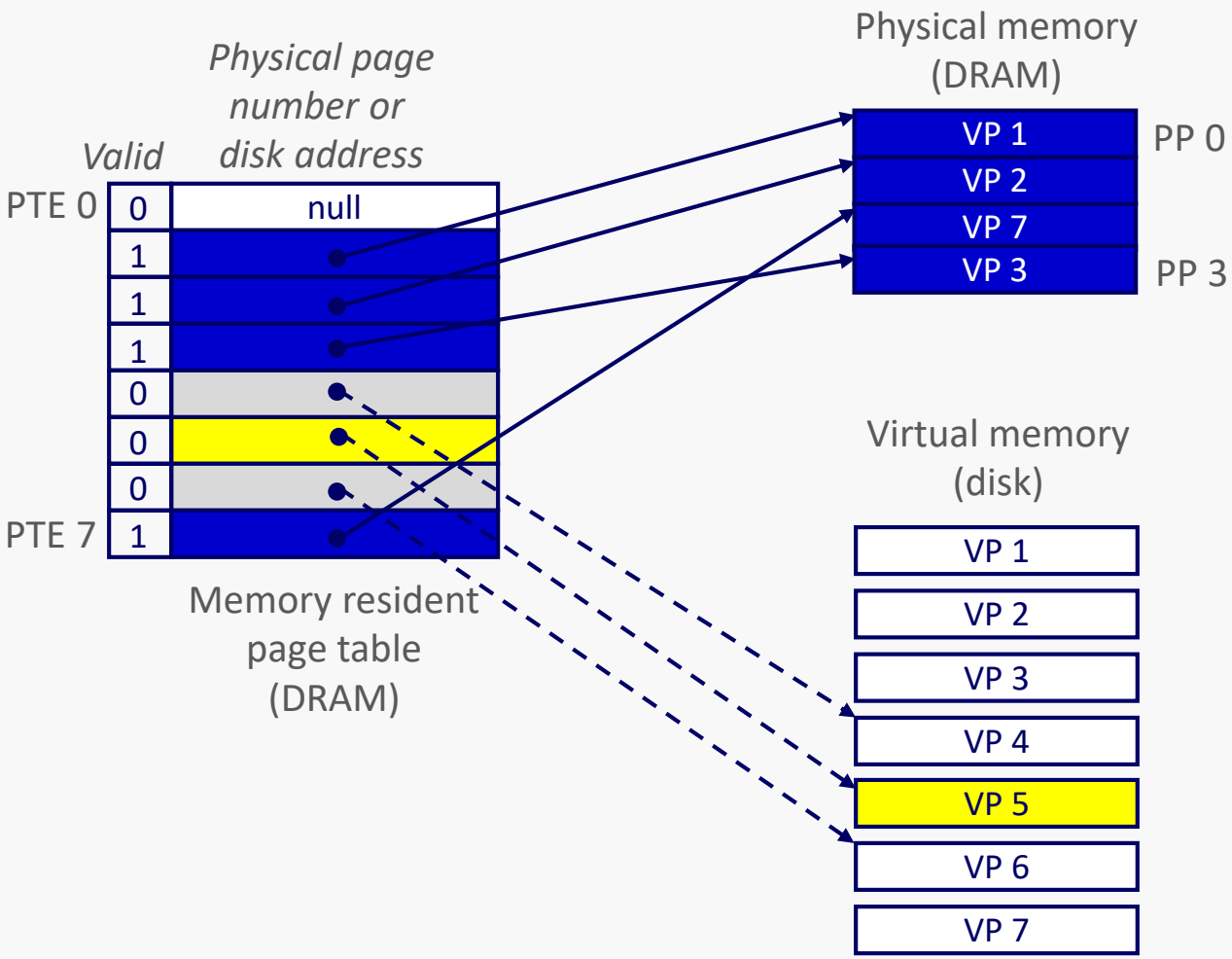
Offending instruction is restarted: page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

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 Address Translation

Virtual Address Space

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

Physical Address Space

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

Address Translation

- $\mathbf{MAP: V \rightarrow P \cup \{\varnothing\}}$
- For virtual address ***a***:
  - ***MAP(a) = a'*** if data at virtual address ***a*** is at physical address ***a'*** in ***P***
  - ***MAP(a) =*** $\varnothing$ 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)

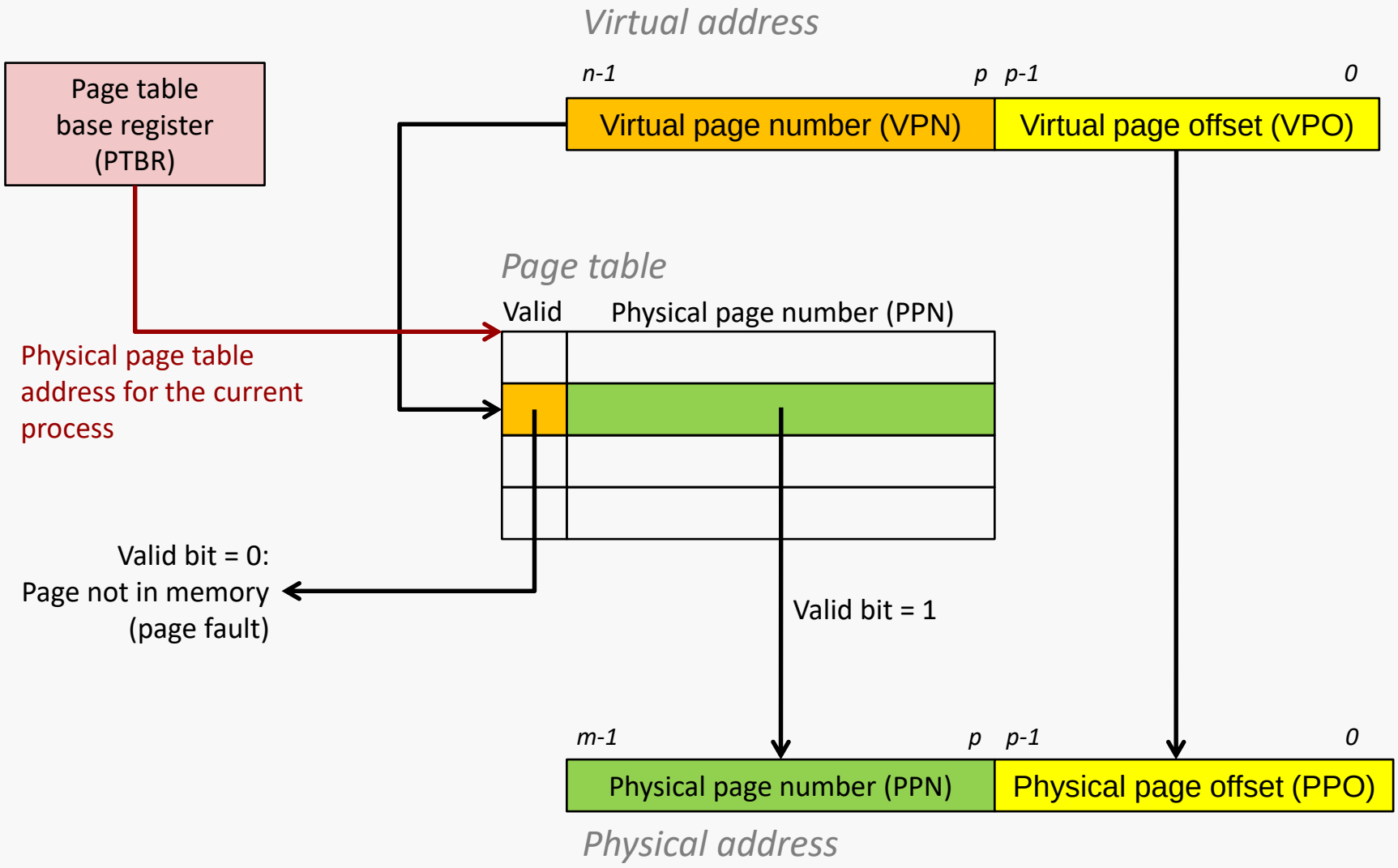
Components of the virtual address (VA)

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

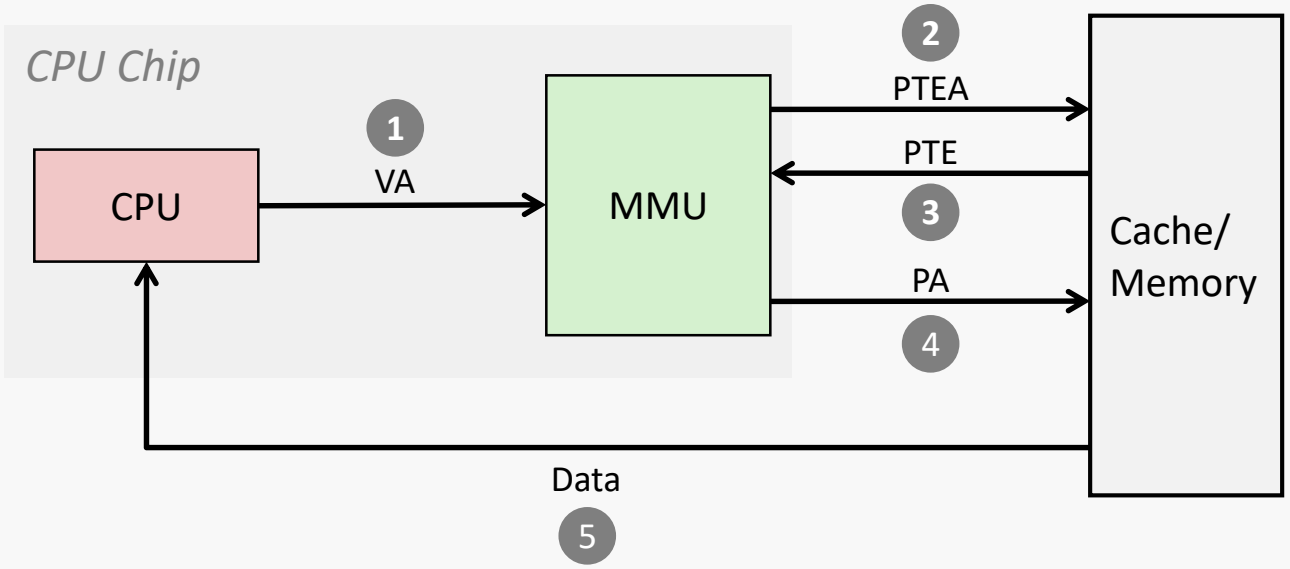
Components of the physical address (PA)

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

Address Translation With a Page Table



Address Translation: Page Hit



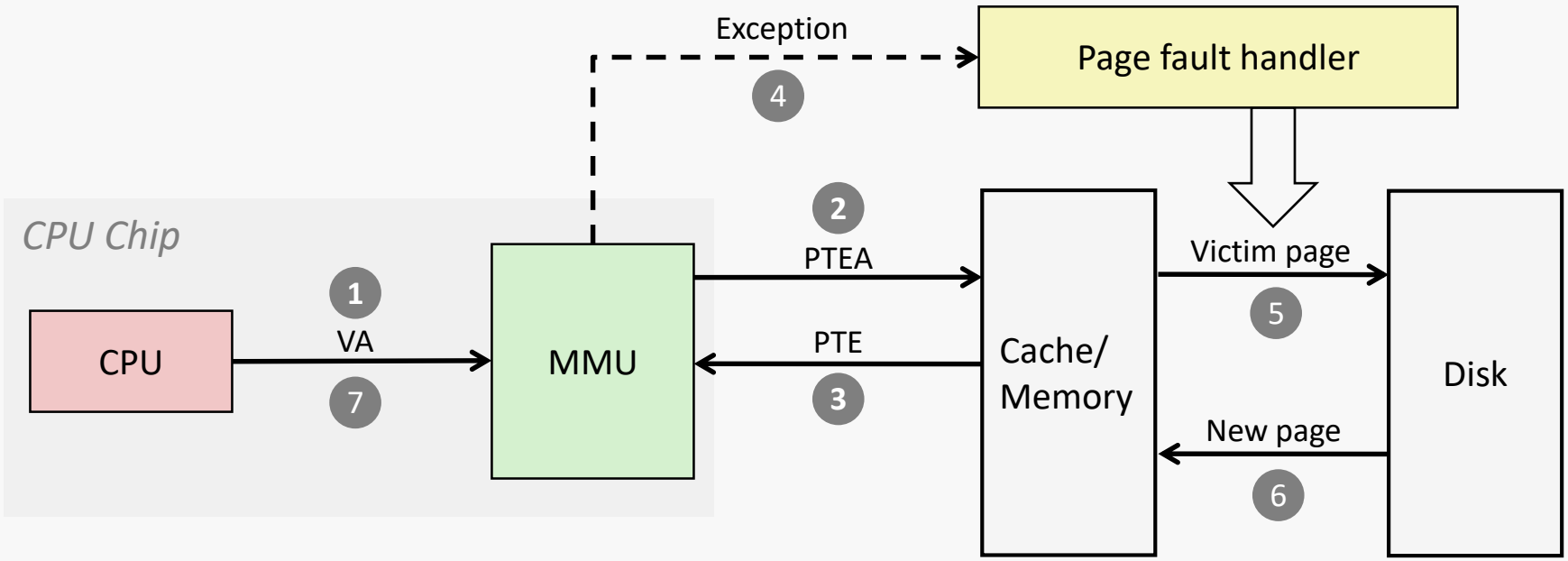
1) Processor sends virtual address to MMU

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

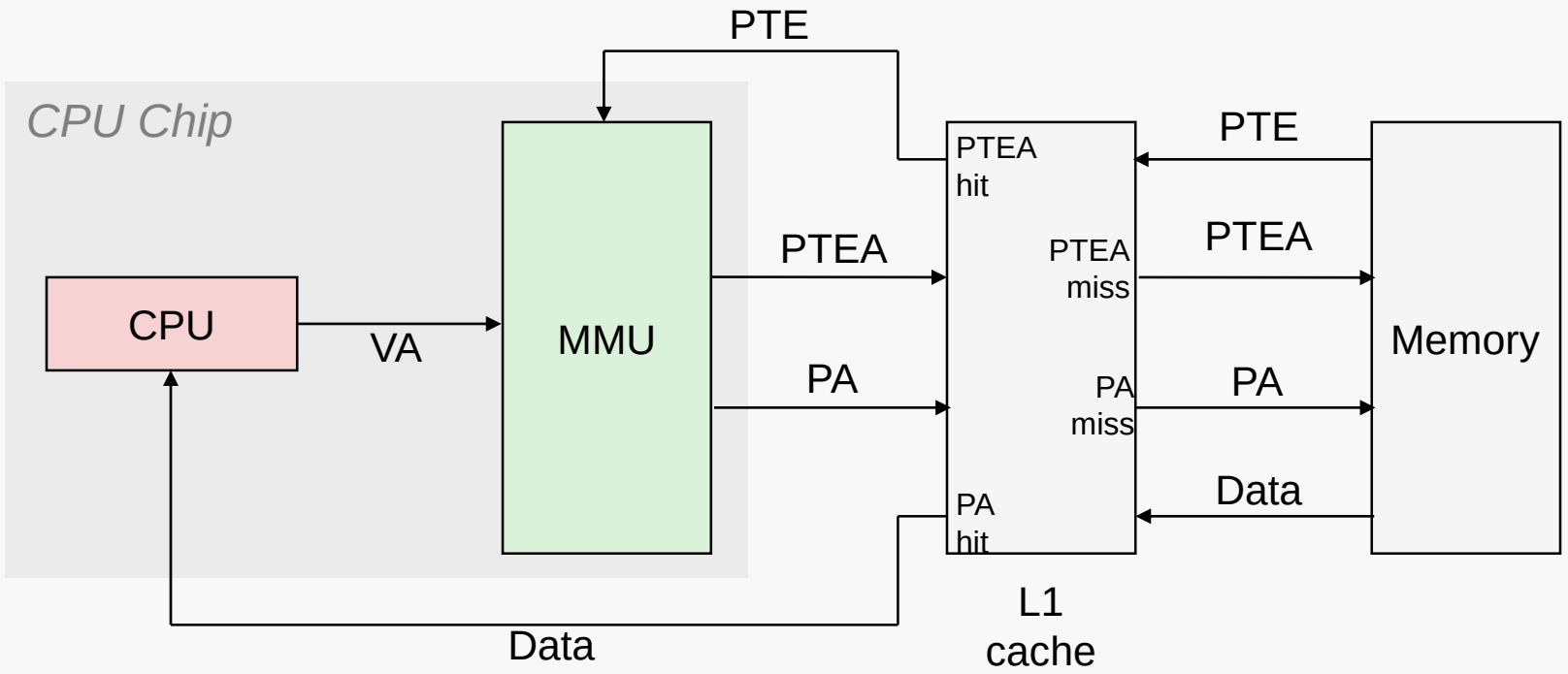
4) Valid bit is zero, so MMU triggers page fault exception

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

If page table entries (PTEs) are cached in L1 like any other memory word

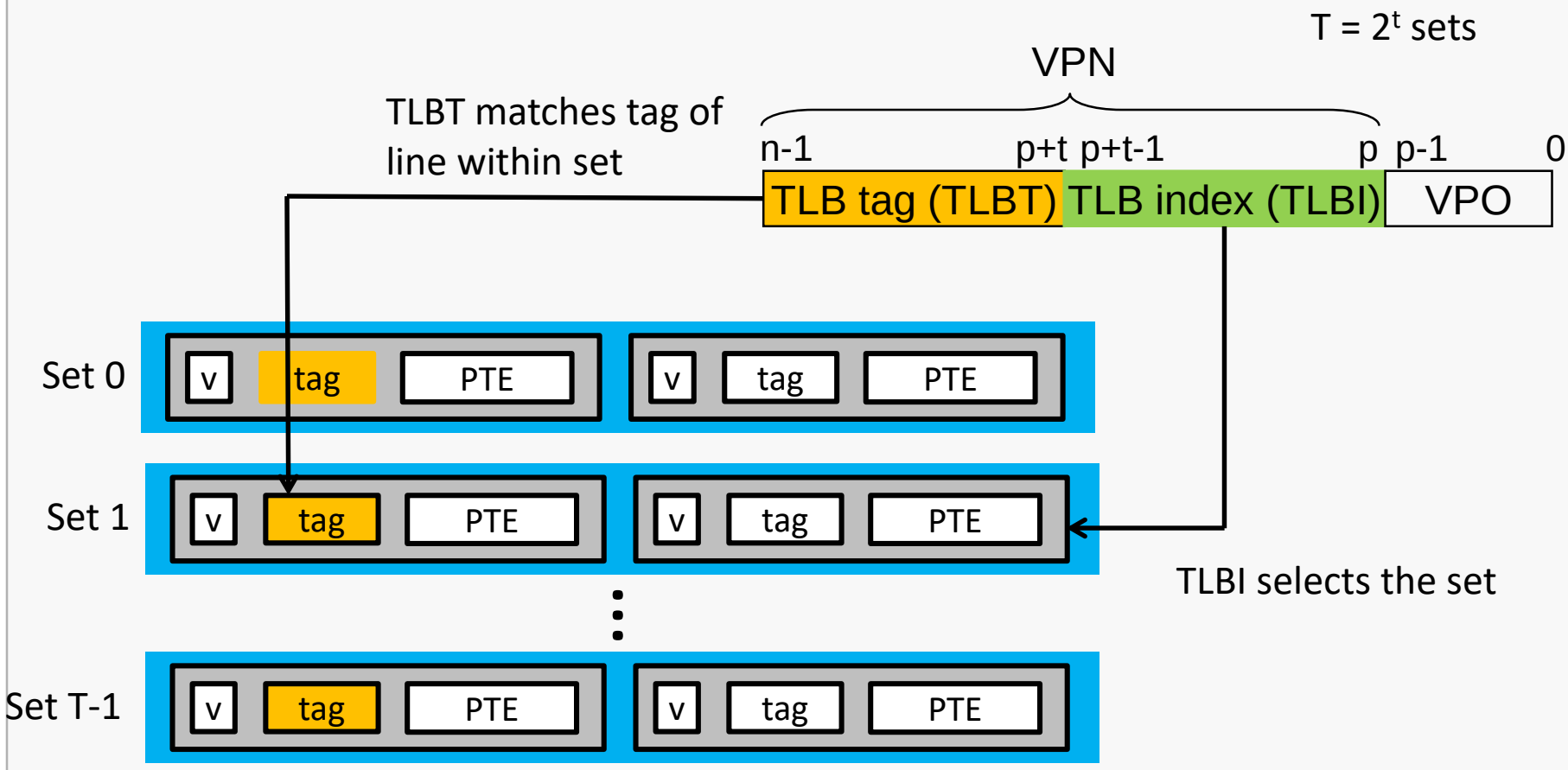
- PTEs may be evicted by other data references
- PTE hit still requires a small L1 delay

Solution: *Translation Lookaside Buffer* (TLB)

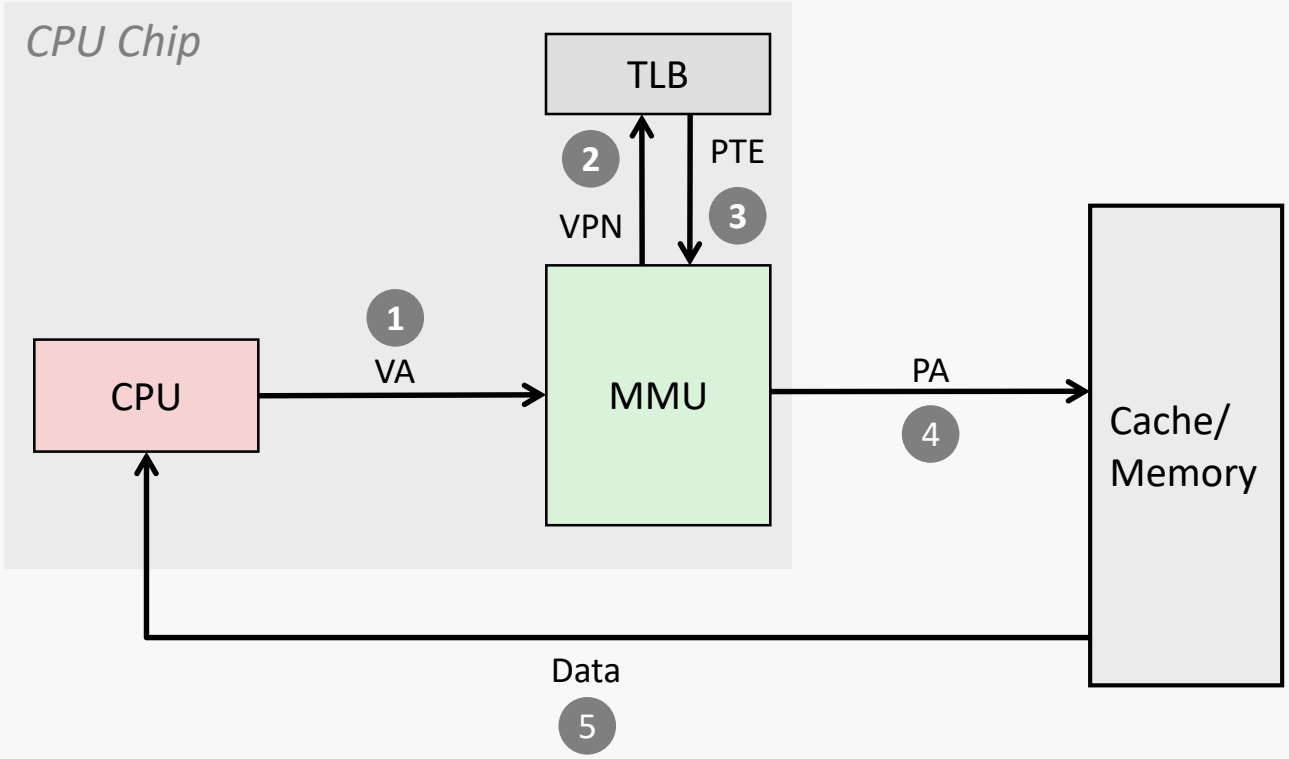
- Small set-associative hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

Accessing the TLB

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

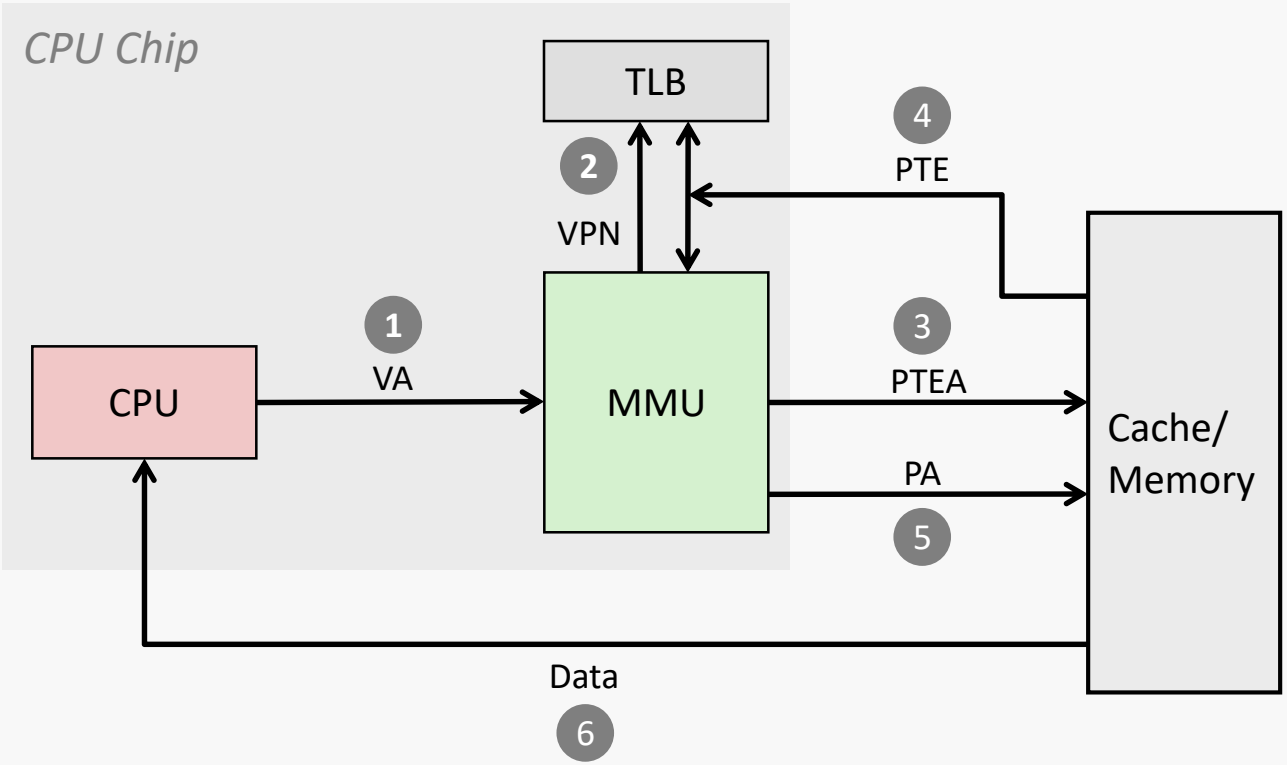


TLB Hit



A TLB hit eliminates a cache/memory access

TLB Miss

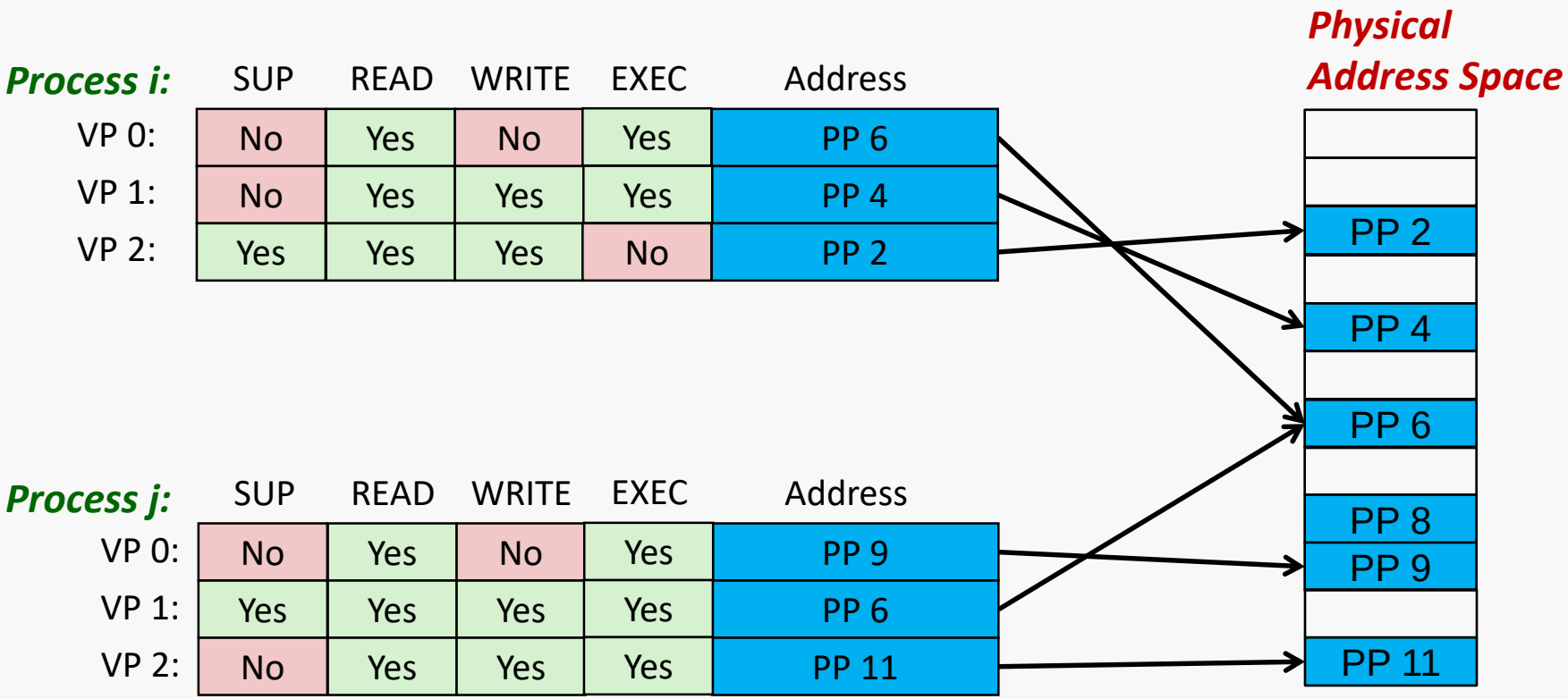


A TLB miss incurs an additional cache/memory access (to get the PTE)

Fortunately, TLB misses are rare. Why?

VM as a Tool for Memory Protection

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



Examples of 2-Level TLB Organization

	Intel Nehalem	AMD Opteron X4
Virtual addr	48 bits	48 bits
Physical addr	44 bits	48 bits
Page size	4KB, 2/4MB	4KB, 2/4MB
L1 TLB (per core)	L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement	L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement
L2 TLB (per core)	Single L2 TLB: 512 entries 4-way, LRU replacement	L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU
TLB misses	Handled in hardware	Handled in hardware

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