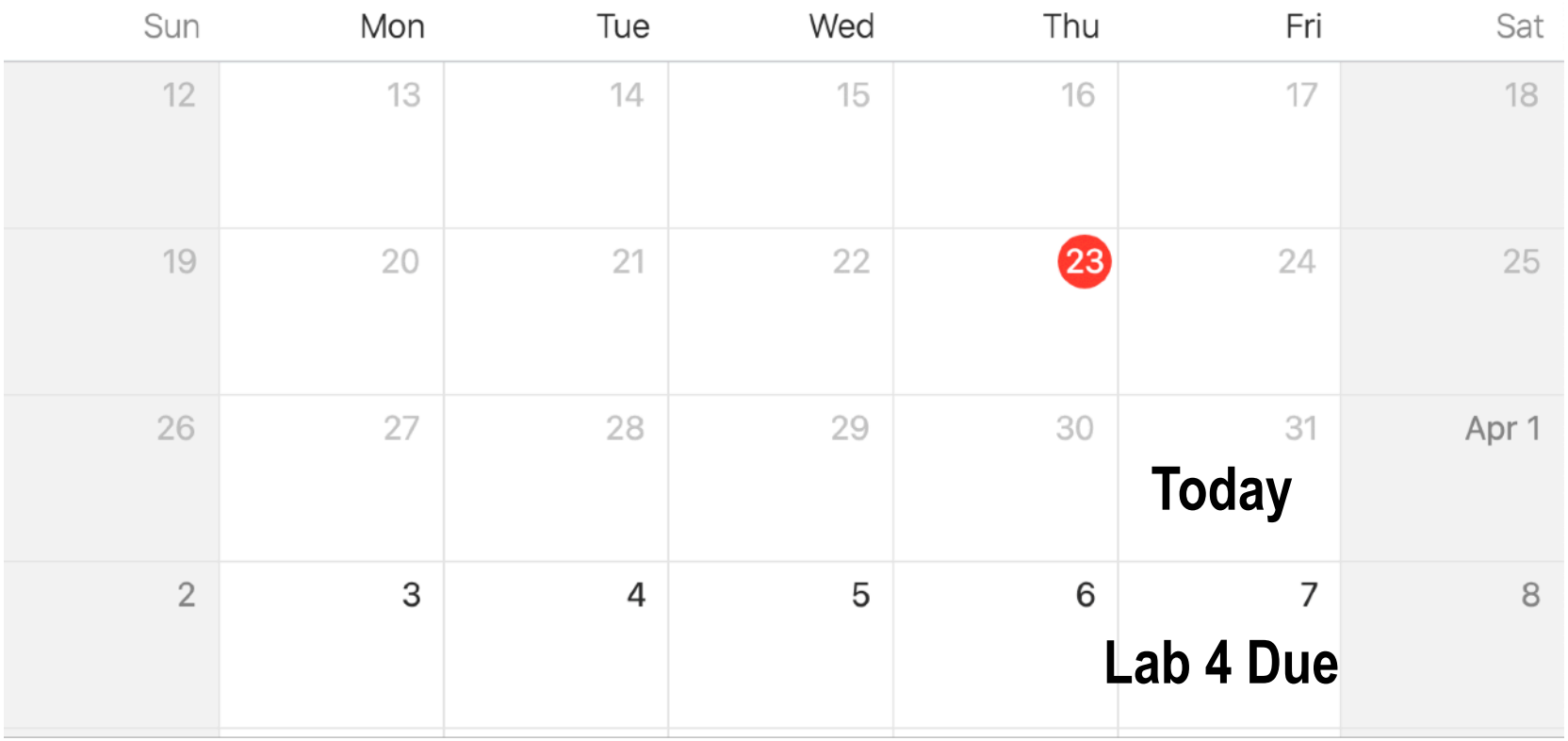
CSC 252: Computer Organization
Spring 2023: Lecture 21

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcements

- Cache problem set: https://www.cs.rochester.edu/courses/252/spring2023/handouts.html Won't be graded.
- Mid-term solution posted: https://www.cs.rochester.edu/courses/252/spring2023/handouts.html



Today

- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

High Level Ideas

- Use physical memory as a “cache” for the hard drive, just like how caches are used w.r.t. physical memory.

High Level Ideas

- Use physical memory as a “cache” for the hard drive, just like how caches are used w.r.t. physical memory.
- Goal: store the data a program will need in the near future in physical memory

High Level Ideas

- Use physical memory as a “cache” for the hard drive, just like how caches are used w.r.t. physical memory.
- Goal: store the data a program will need in the near future in physical memory
  - Moving data back and forth between physical memory and hard drive is hidden from programmers. OS does it.

High Level Ideas

- Use physical memory as a “cache” for the hard drive, just like how caches are used w.r.t. physical memory.
- Goal: store the data a program will need in the near future in physical memory
  - Moving data back and forth between physical memory and hard drive is hidden from programmers. OS does it.
- So need to exploit spatial and temporal locality.

High Level Ideas

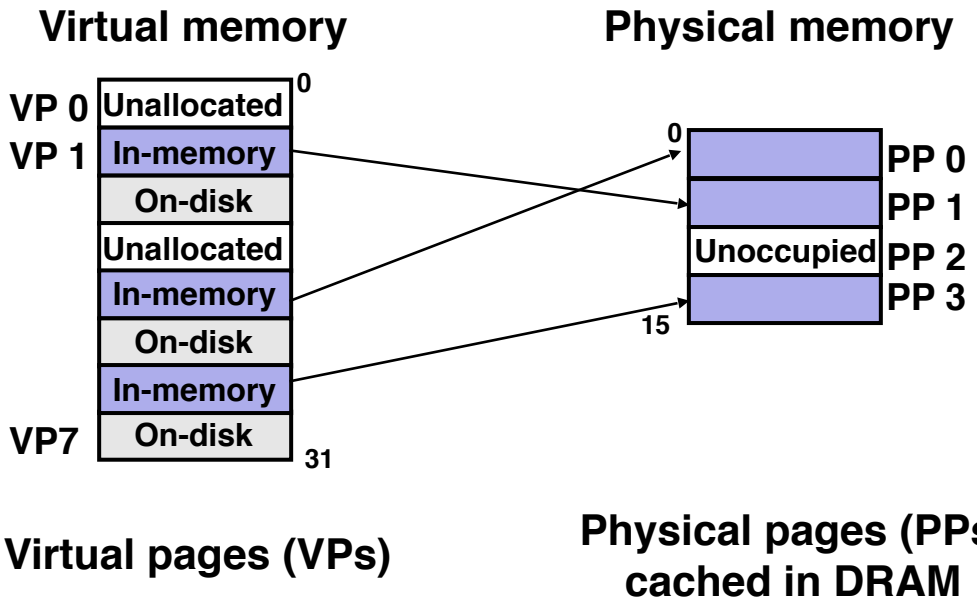
- Use physical memory as a “cache” for the hard drive, just like how caches are used w.r.t. physical memory.
- Goal: store the data a program will need in the near future in physical memory
  - Moving data back and forth between physical memory and hard drive is hidden from programmers. OS does it.
- So need to exploit spatial and temporal locality.
  - Exploiting spatial locality: in what granularity are we moving data between physical memory and hard drive? One byte?

High Level Ideas

- Use physical memory as a “cache” for the hard drive, just like how caches are used w.r.t. physical memory.
- Goal: store the data a program will need in the near future in physical memory
  - Moving data back and forth between physical memory and hard drive is hidden from programmers. OS does it.
- So need to exploit spatial and temporal locality.
  - Exploiting spatial locality: in what granularity are we moving data between physical memory and hard drive? One byte?
  - Exploiting temporal locality: replacement policy (e.g., LRU).

VM Concepts

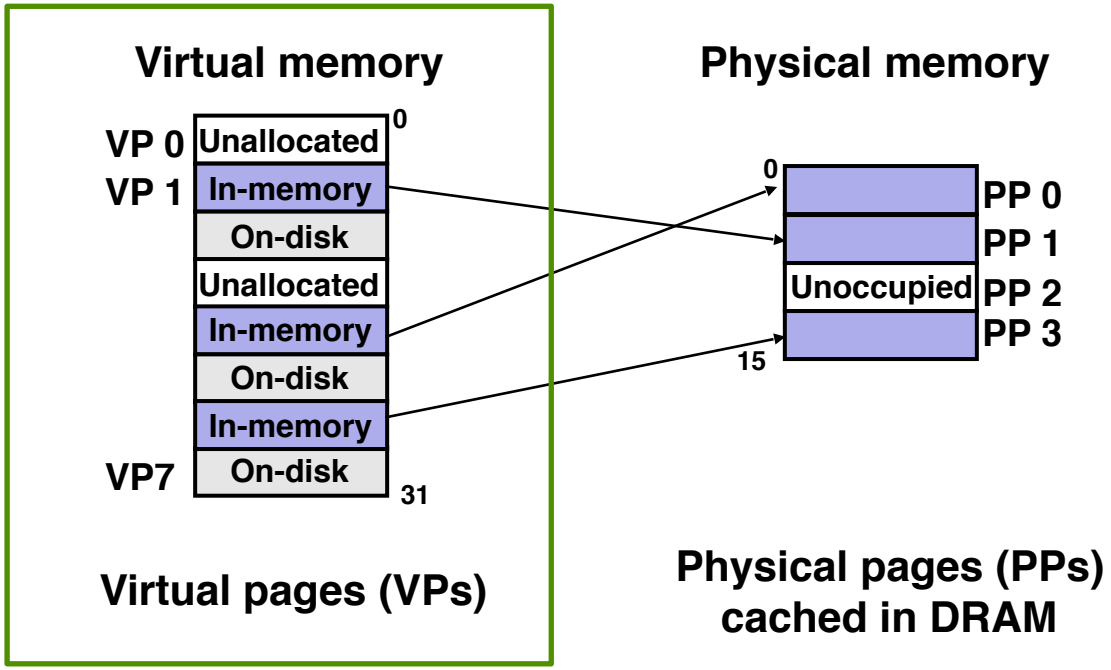
- Divide the *virtual memory* to N contiguous *pages*.
- Each page has a certain amount of continuous bytes, e.g., 4 KB.
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").



VM Concepts

- Divide the *virtual memory* to N contiguous *pages*.
- Each page has a certain amount of continuous bytes, e.g., 4 KB.
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

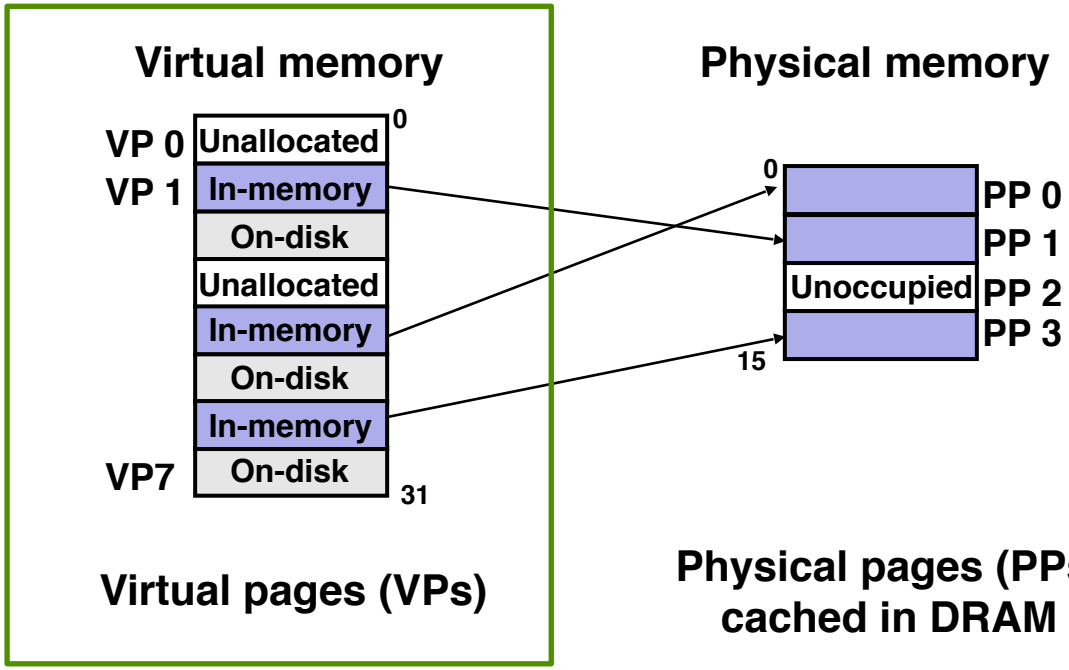
What programmers see



VM Concepts

- Divide the *virtual memory* to N contiguous *pages*.
- Each page has a certain amount of continuous bytes, e.g., 4 KB.
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see

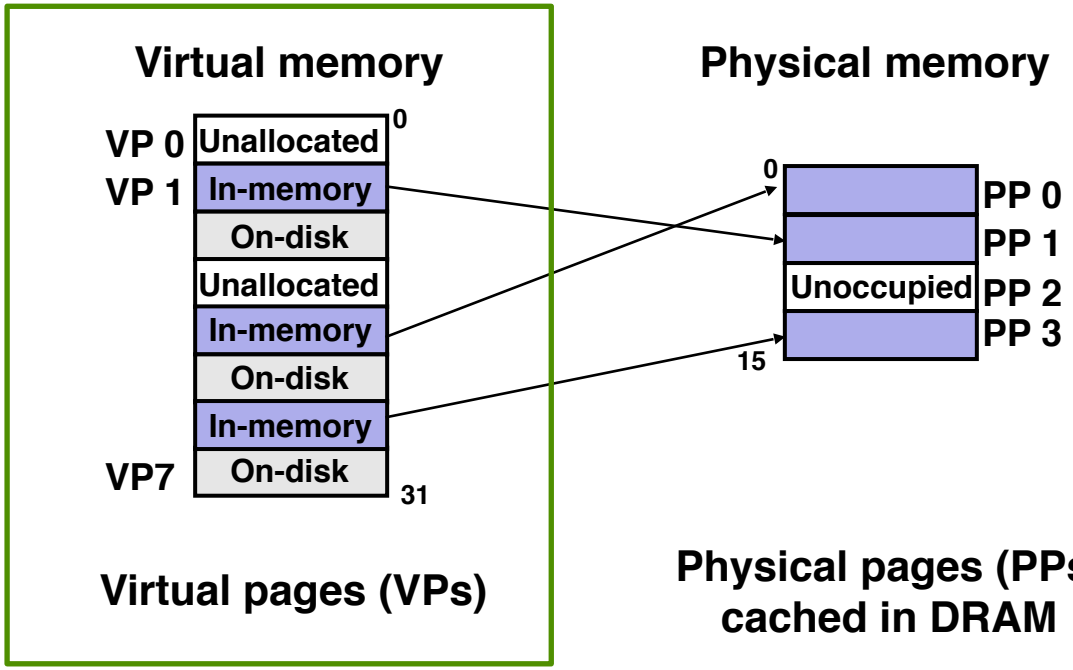


Assuming page size is 4B
Virtual memory size is 32B
Physical memory size is 16B

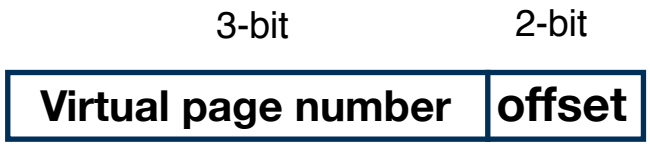
VM Concepts

- Divide the *virtual memory* to N contiguous *pages*.
- Each page has a certain amount of continuous bytes, e.g., 4 KB.
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see



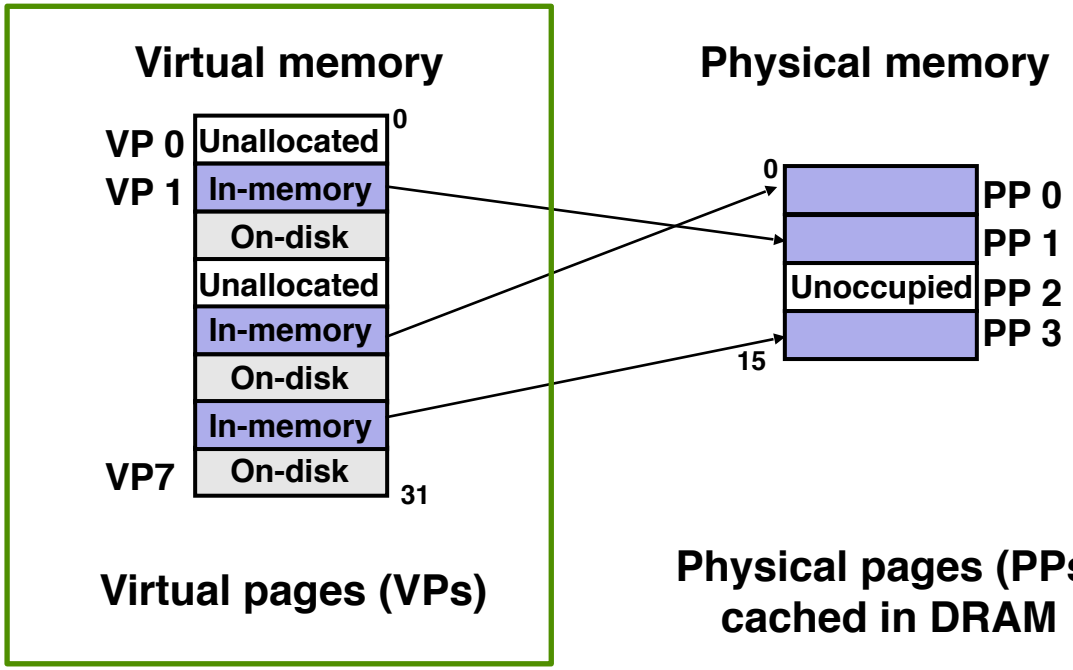
Assuming page size is 4B
Virtual memory size is 32B
Physical memory size is 16B



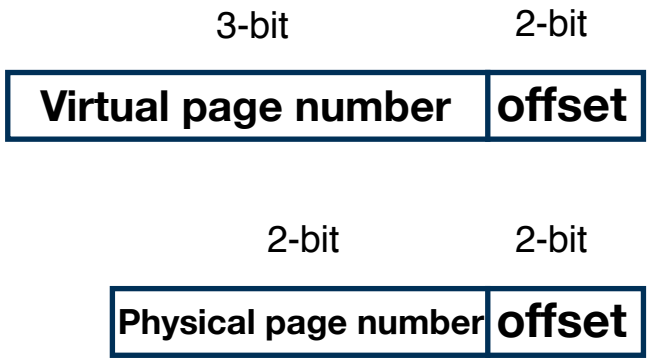
VM Concepts

- Divide the *virtual memory* to N contiguous *pages*.
- Each page has a certain amount of continuous bytes, e.g., 4 KB.
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see



Assuming page size is 4B
Virtual memory size is 32B
Physical memory size is 16B



Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the corresponding virtual page is mapped to the physical memory

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the corresponding virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the corresponding virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?
  - If not mapped, where on the disk is the virtual page?

Enabling Data Structure: Page Table

- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the corresponding virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?
  - If not mapped, where on the disk is the virtual page?
- Do you need a page table for each process?

Enabling Data Structure: Page Table

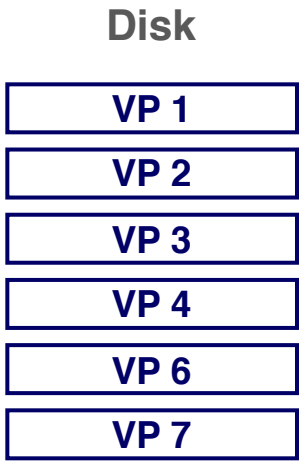
- How do we track which virtual pages are mapped to physical pages, and where they are mapped?
- Use a table to track this. The table is called page table, in which each virtual page has an entry
- Each entry records whether the corresponding virtual page is mapped to the physical memory
  - If mapped, where in the physical memory it is mapped to?
  - If not mapped, where on the disk is the virtual page?
- Do you need a page table for each process?
  - Per-process data structure; managed by the OS kernel

Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.

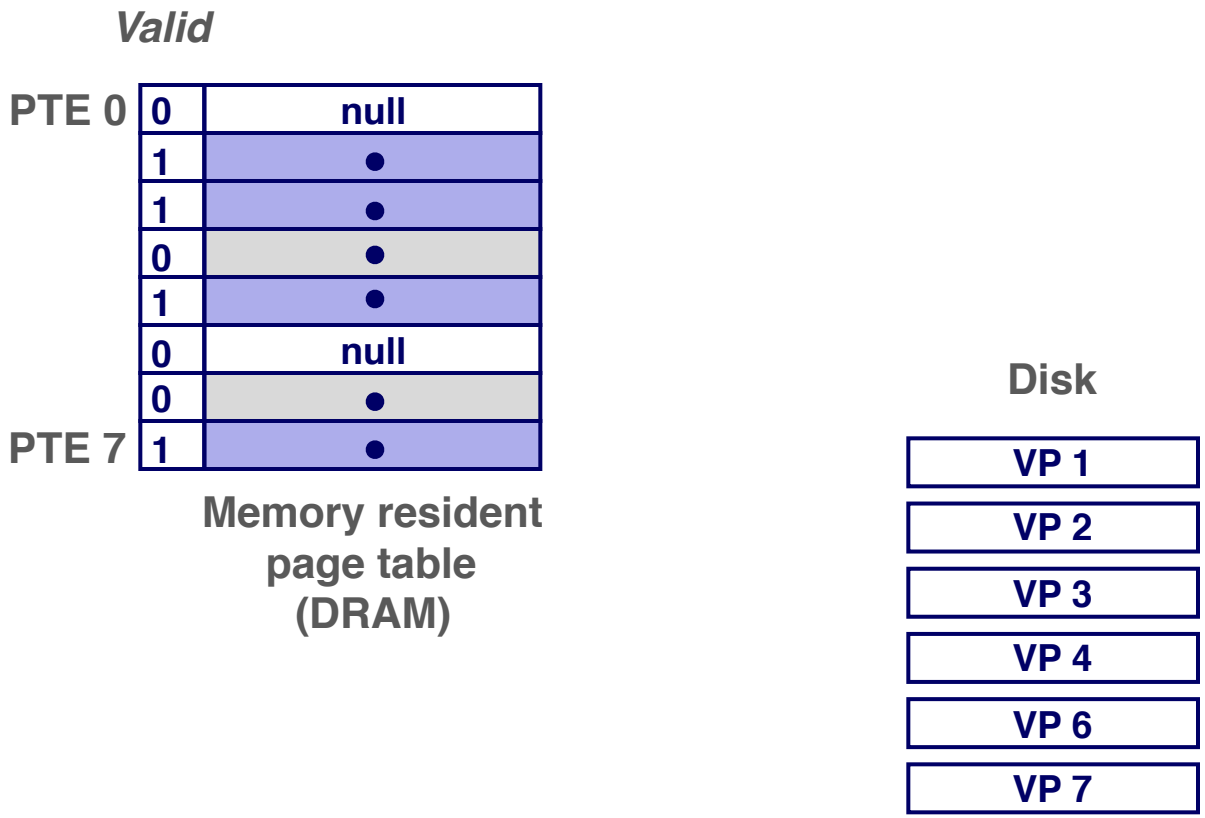
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



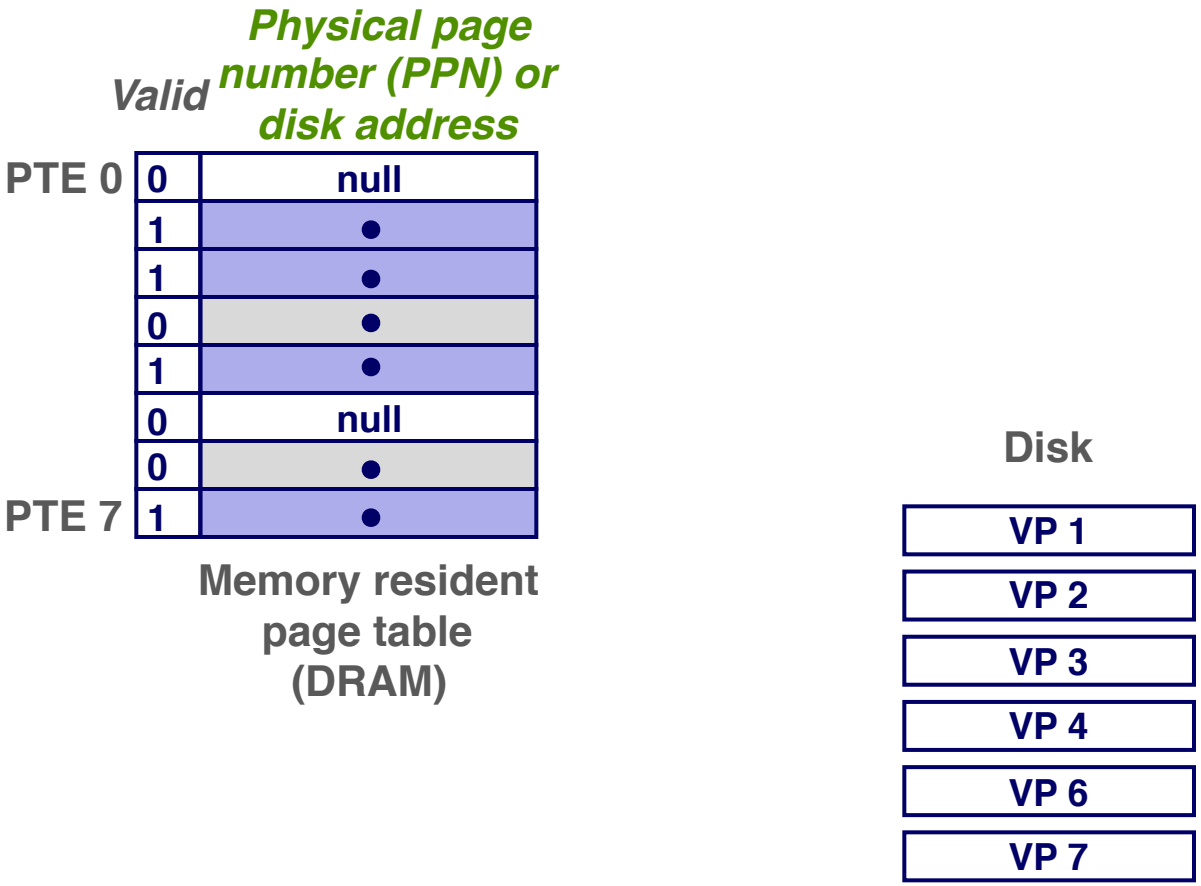
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



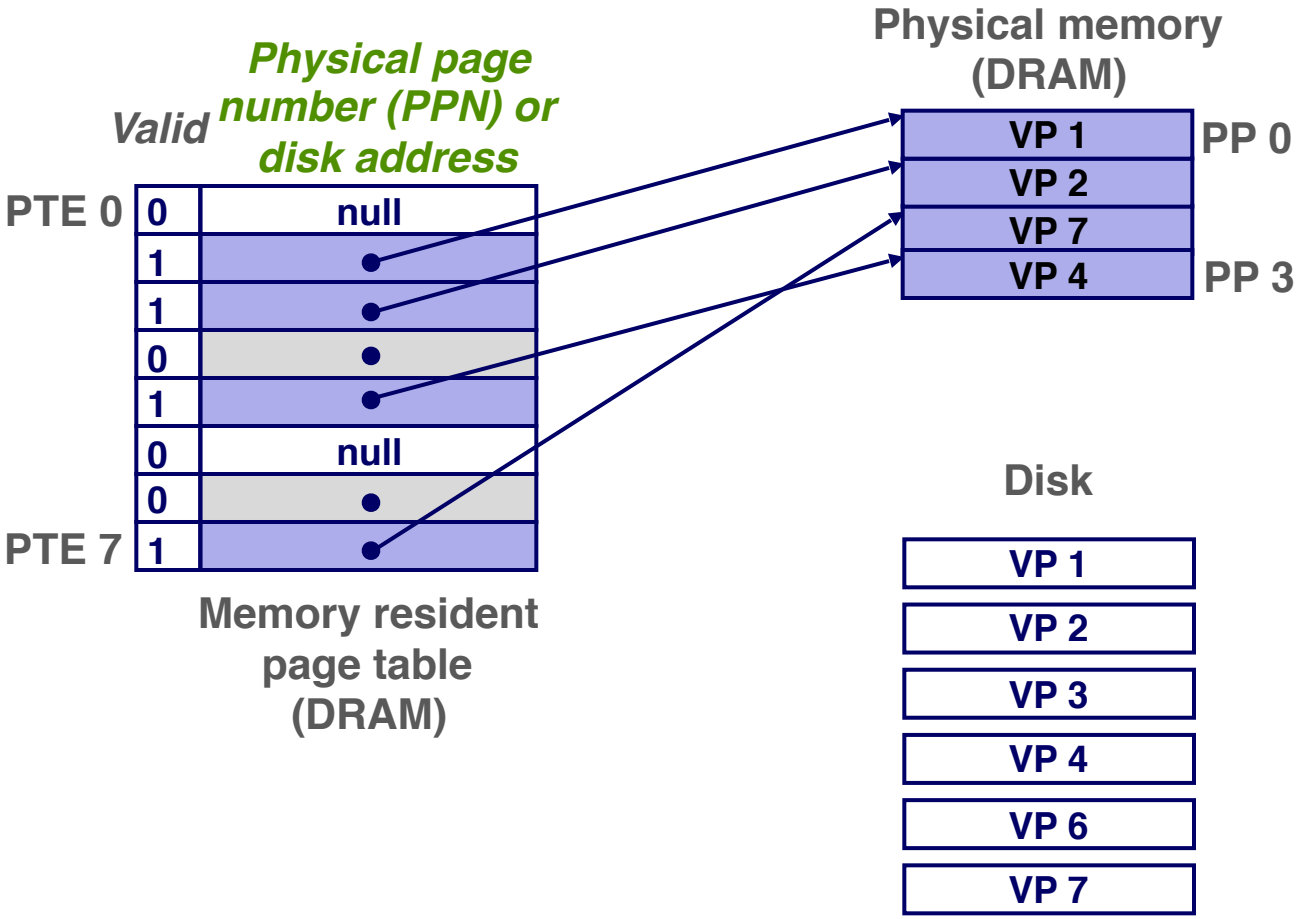
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



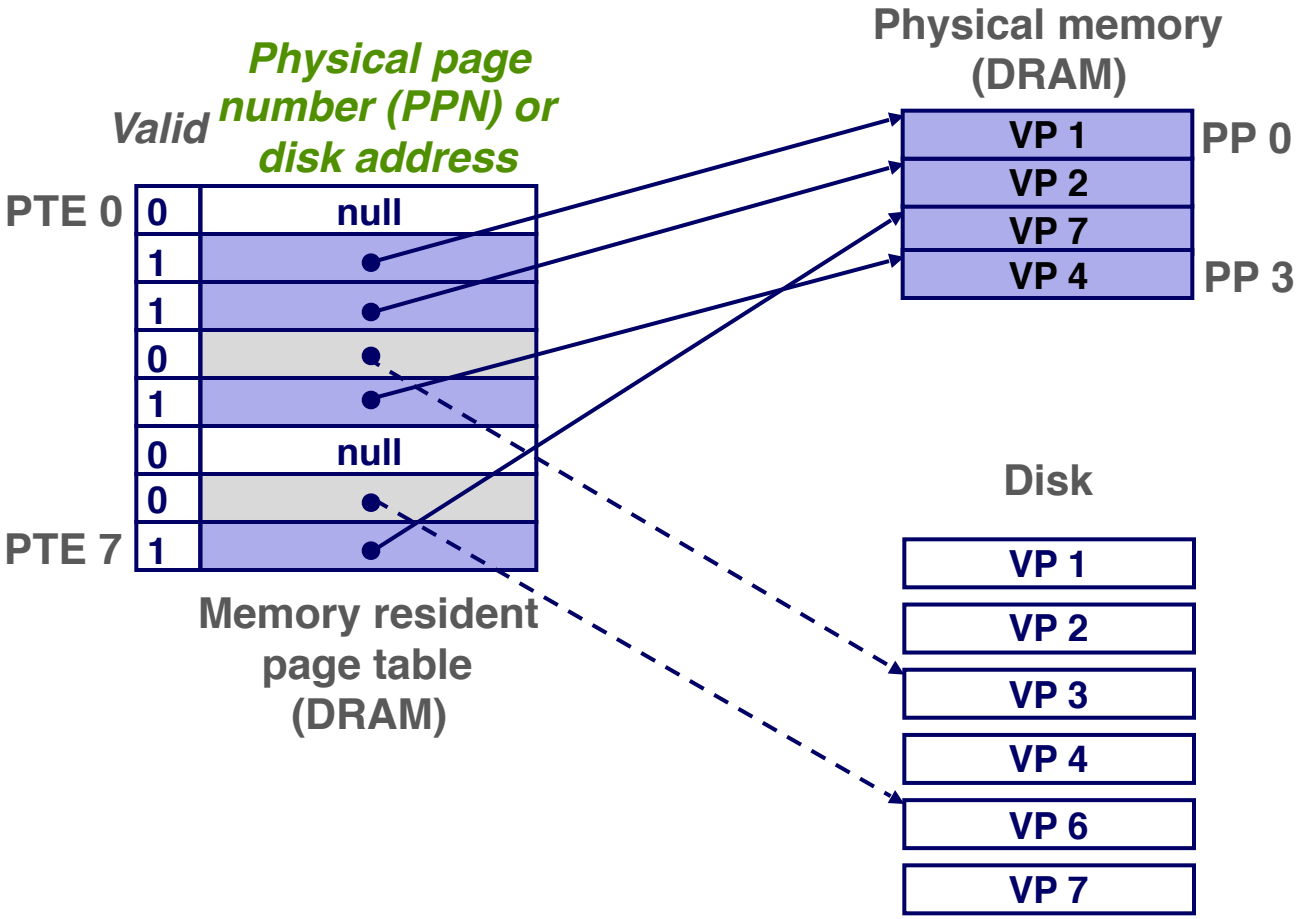
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



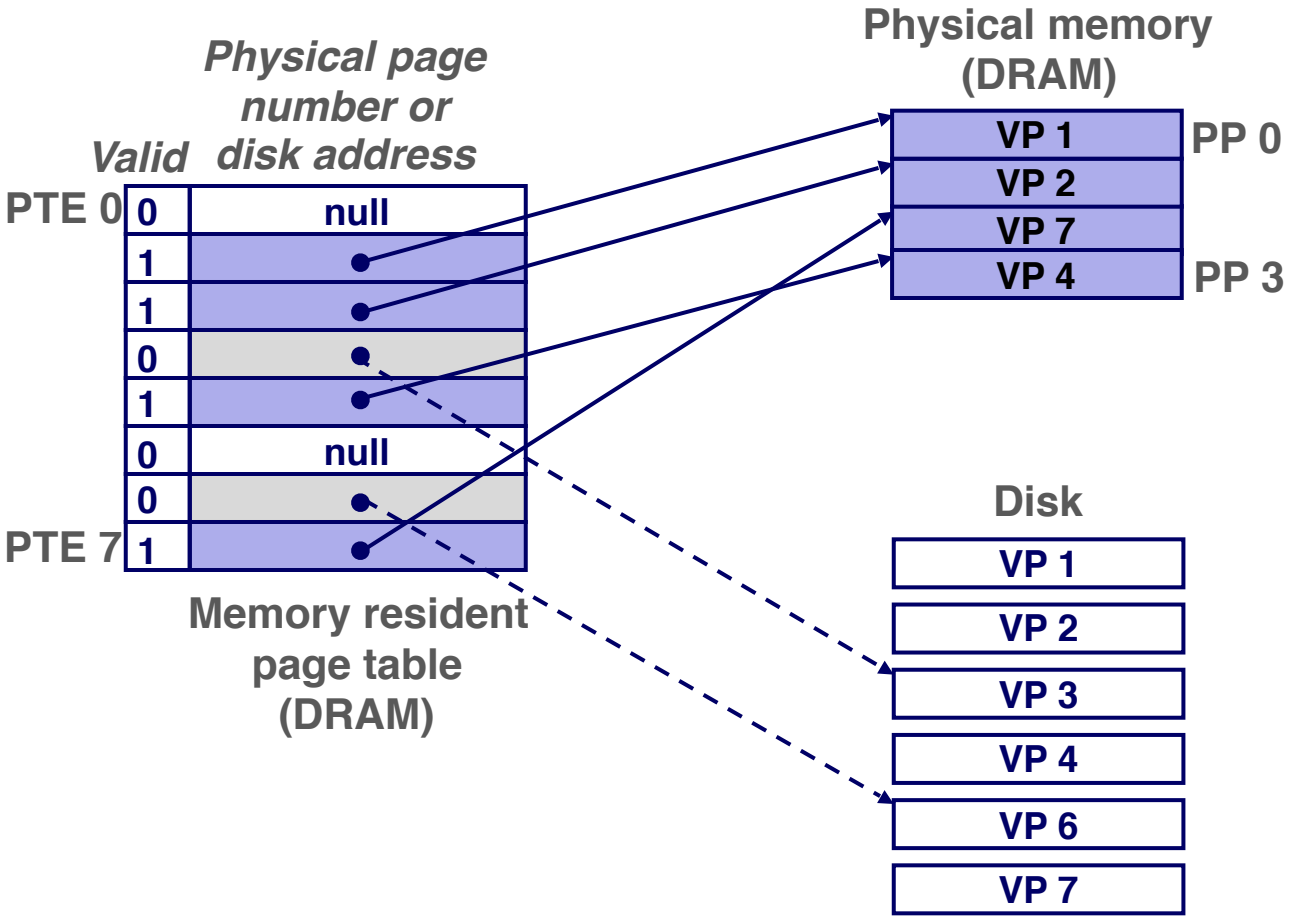
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



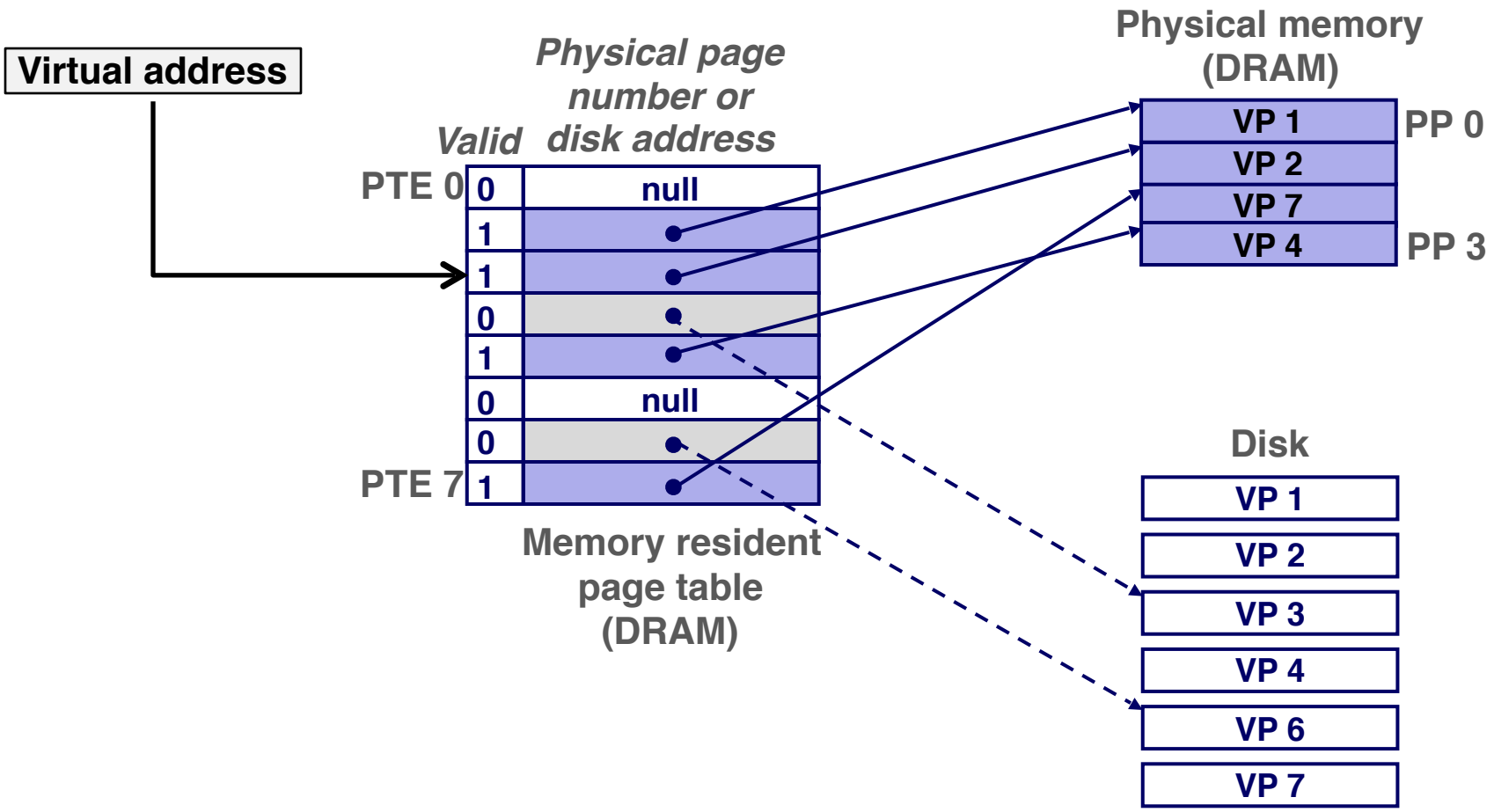
Page Hit

- *Page hit:* reference to VM word that is in physical memory



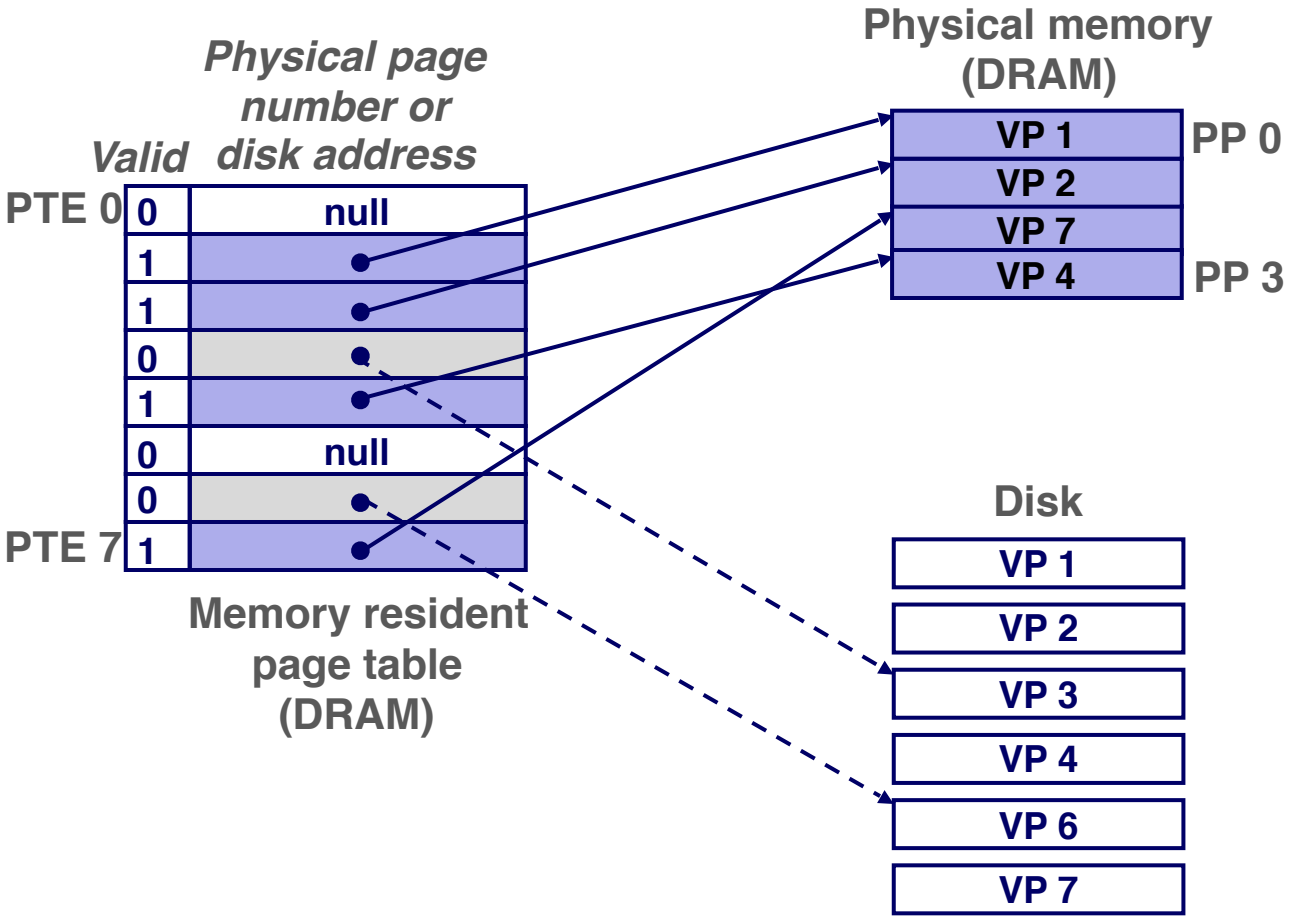
Page Hit

- *Page hit:* reference to VM word that is in physical memory



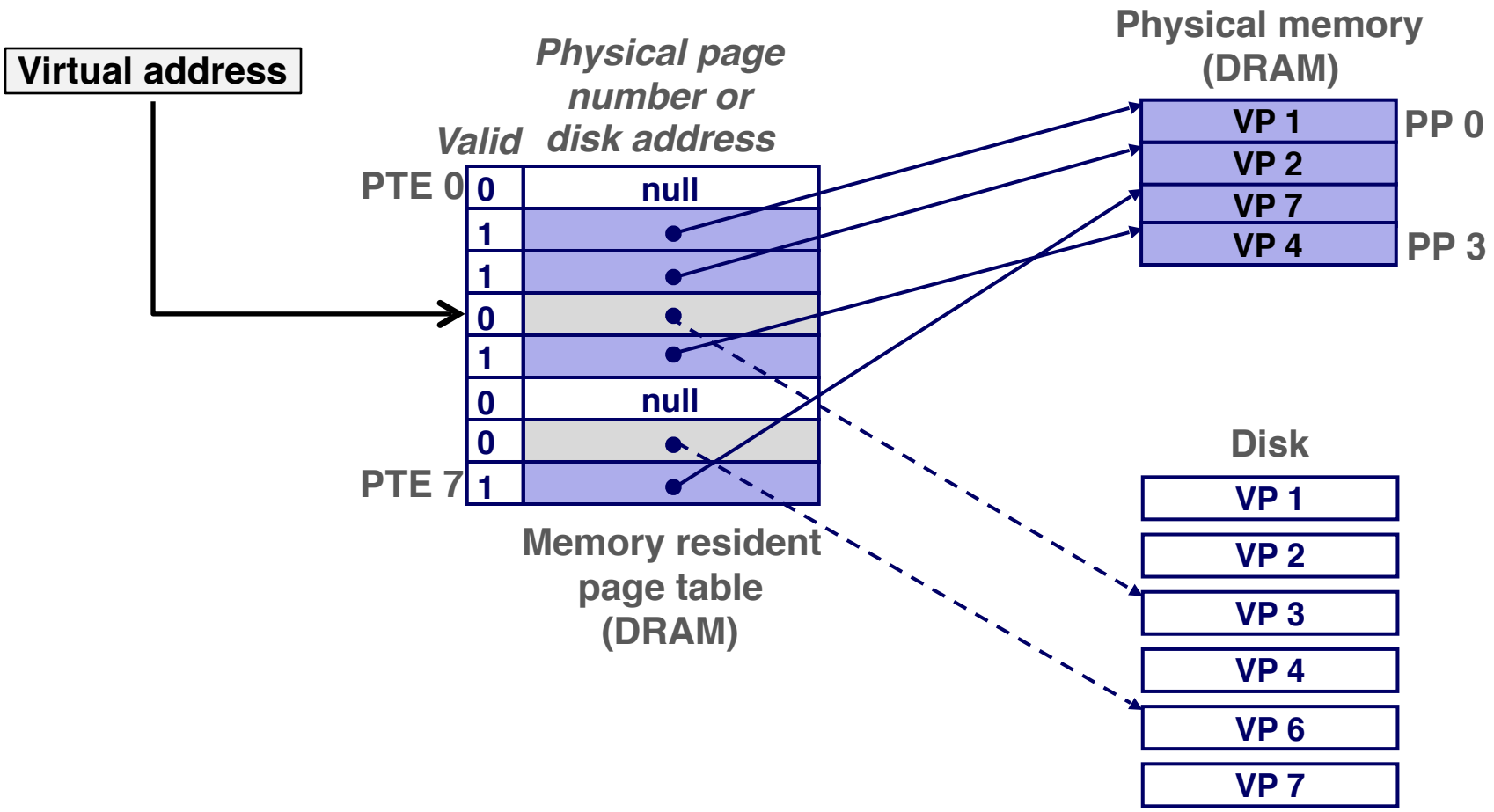
Page Fault

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



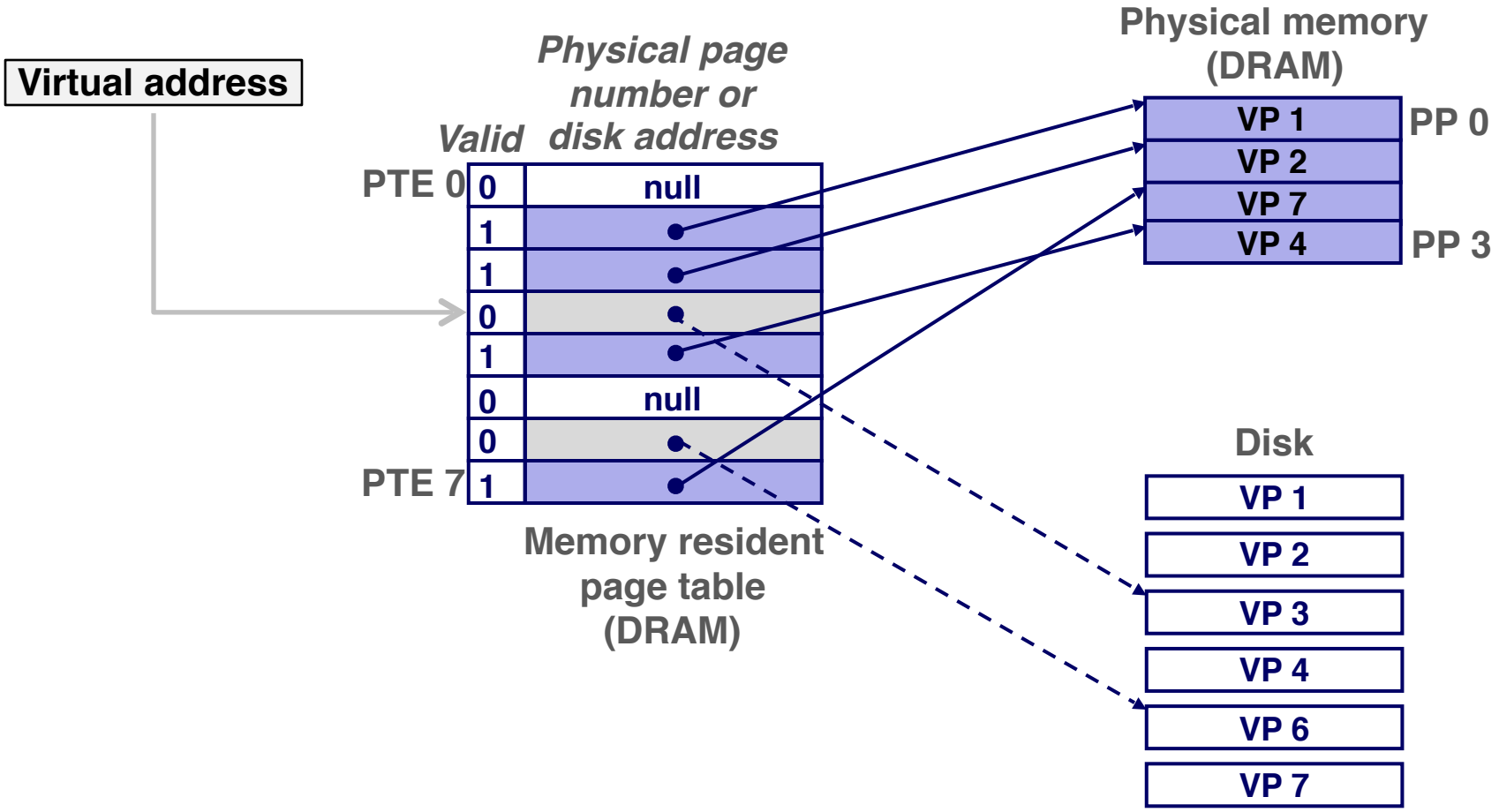
Page Fault

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



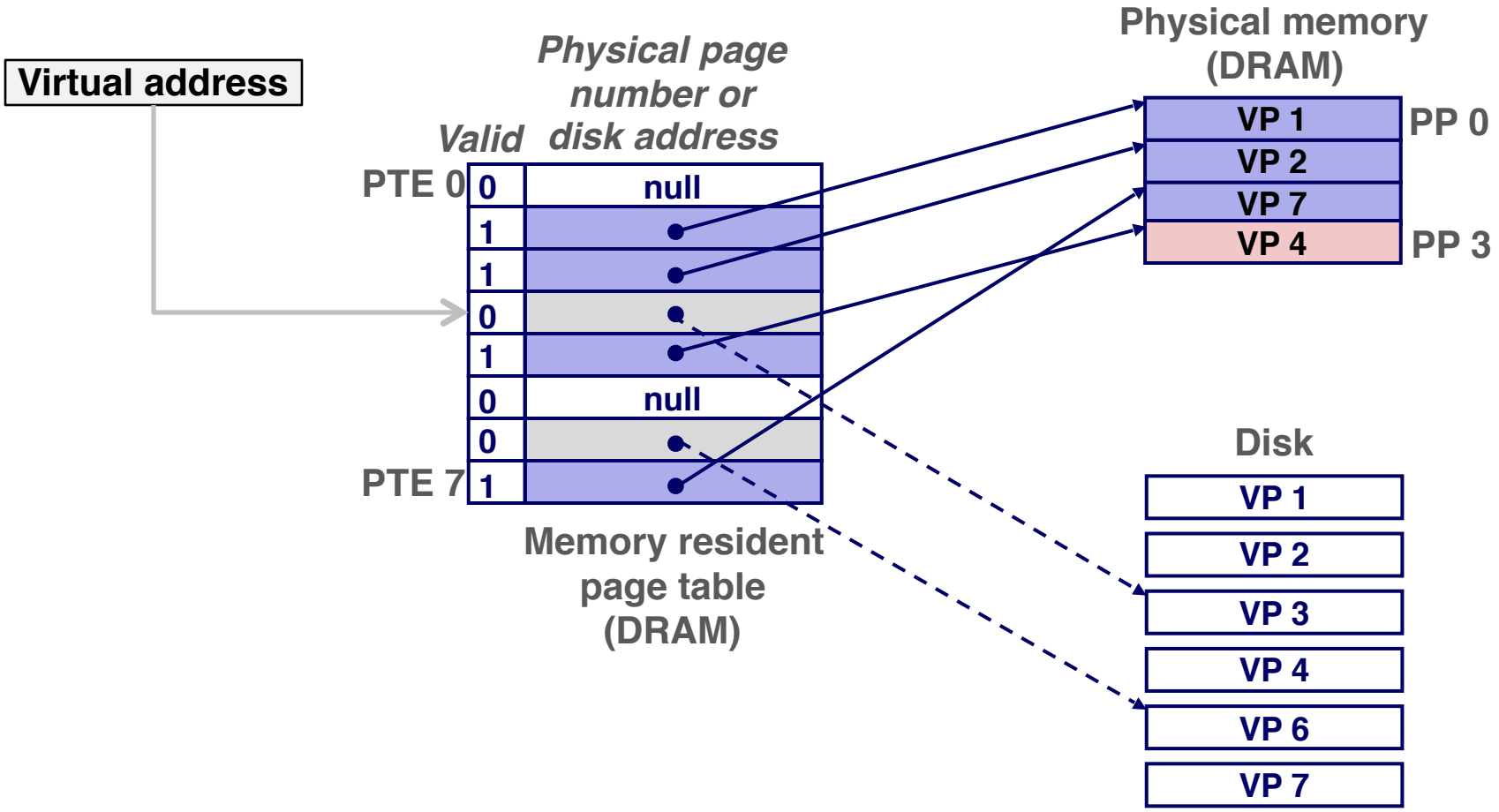
Handling Page Fault

- Page miss causes ***page fault (an exception)***



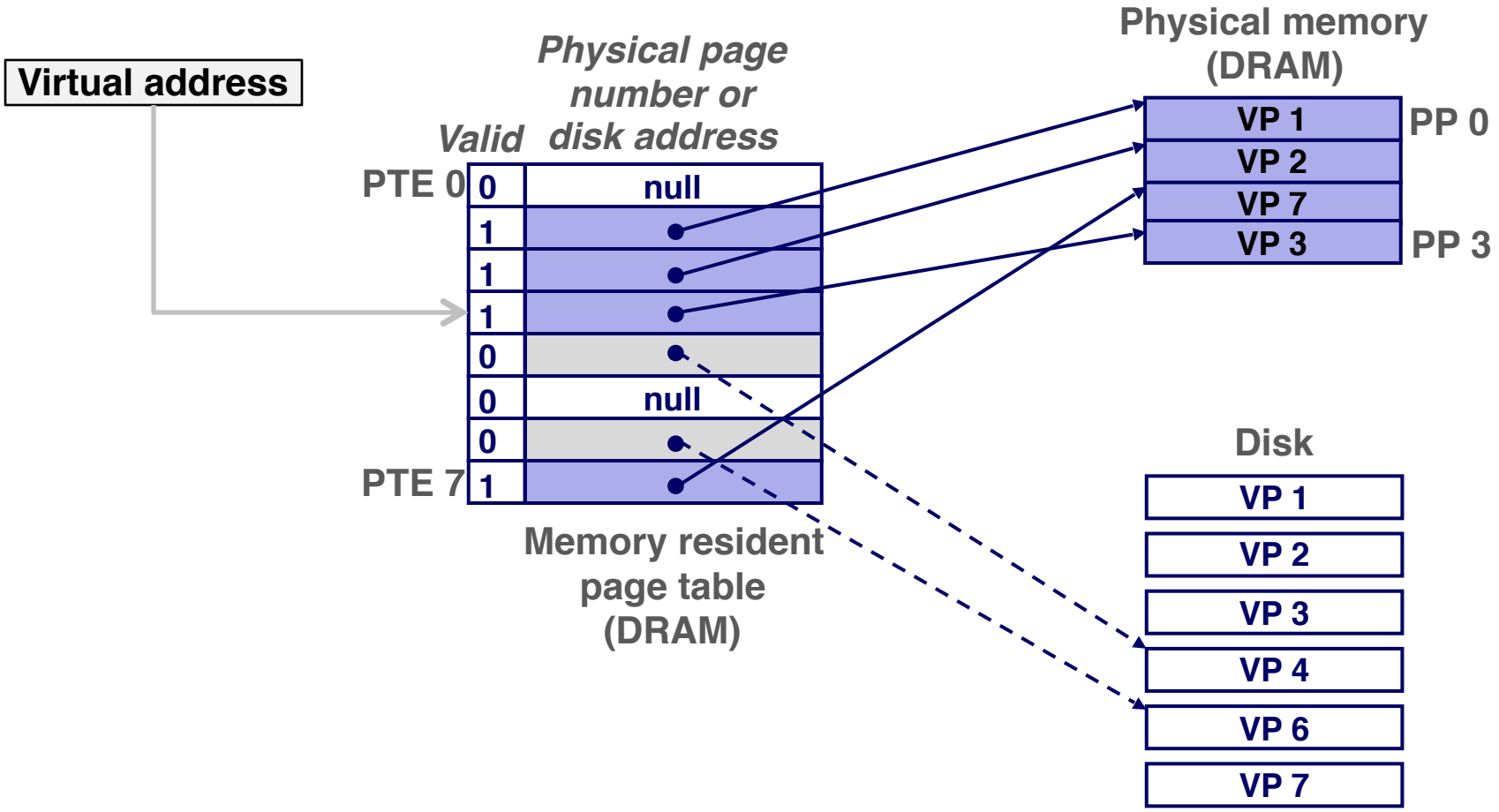
Handling Page Fault

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



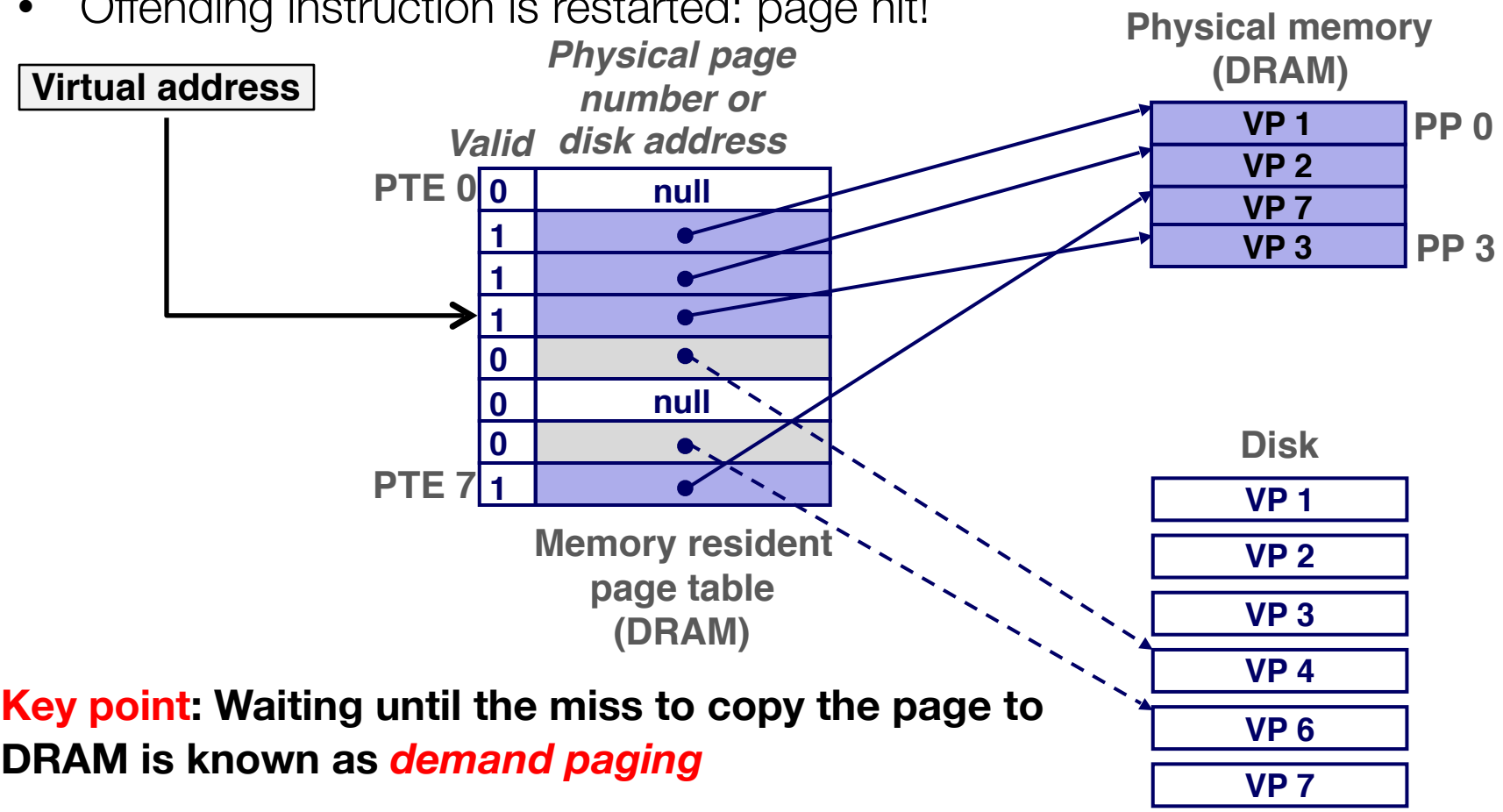
Handling Page Fault

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



Handling Page Fault

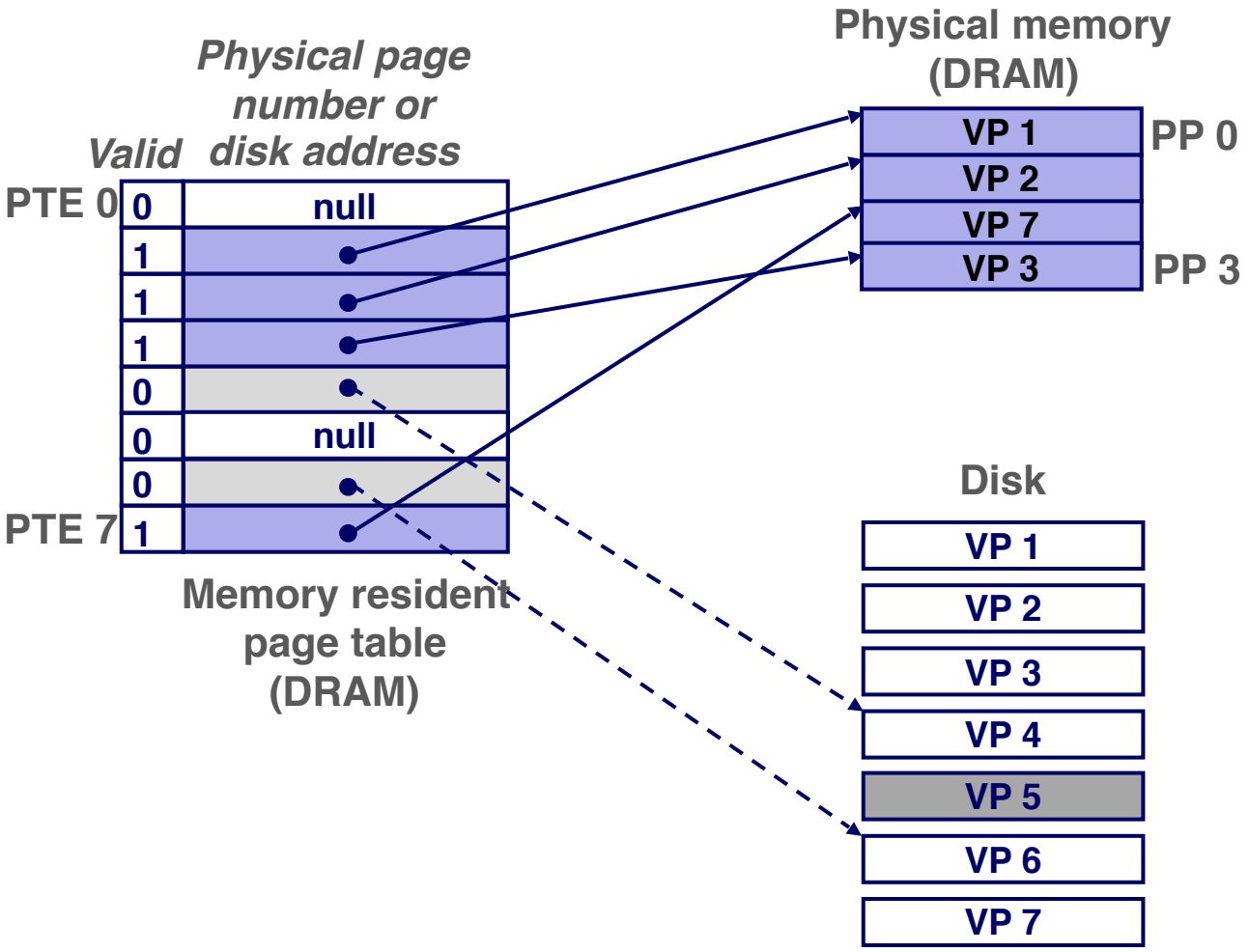
- Page miss causes ***page fault (an exception)***
- Page fault ***handler*** selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



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

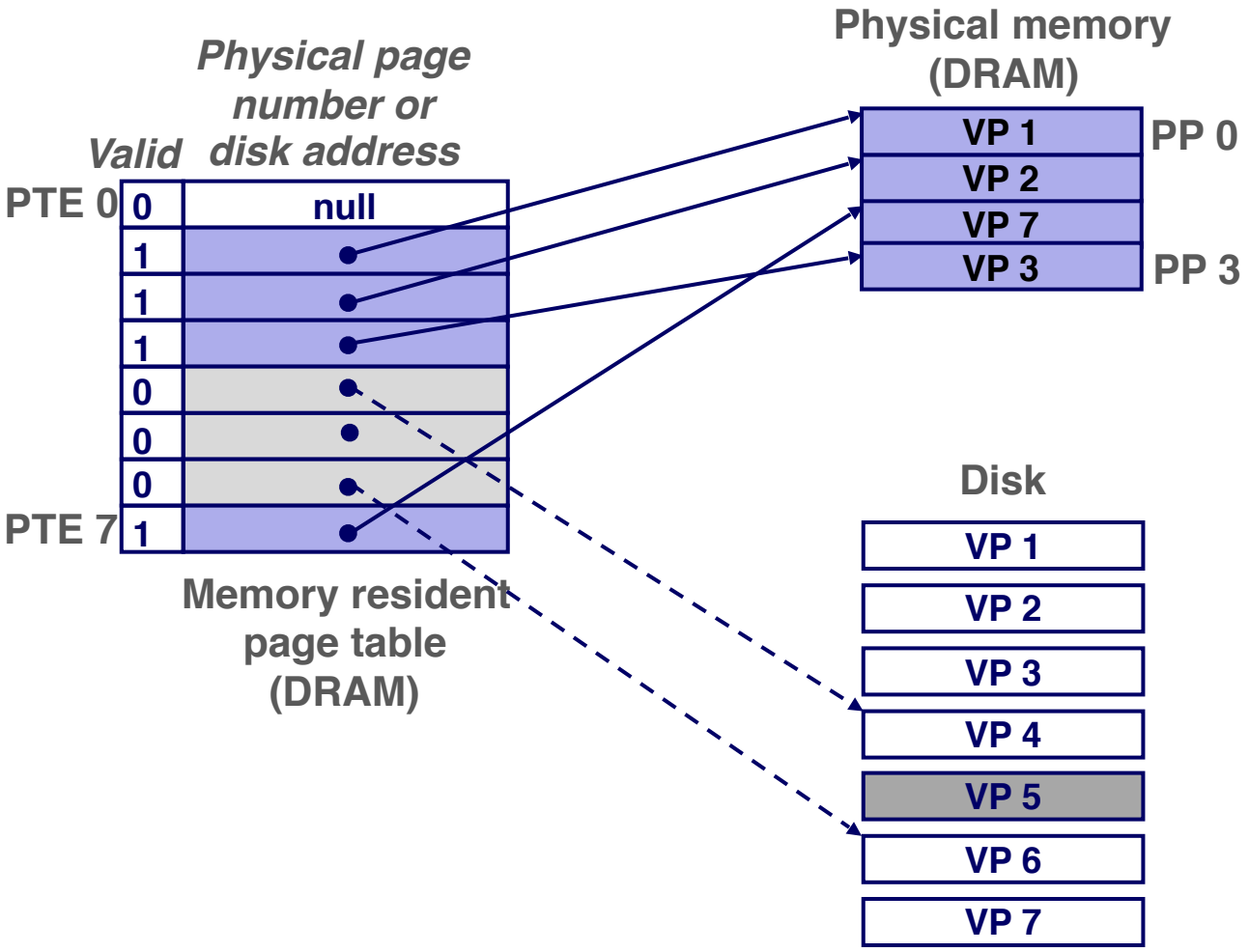
Allocating Pages

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



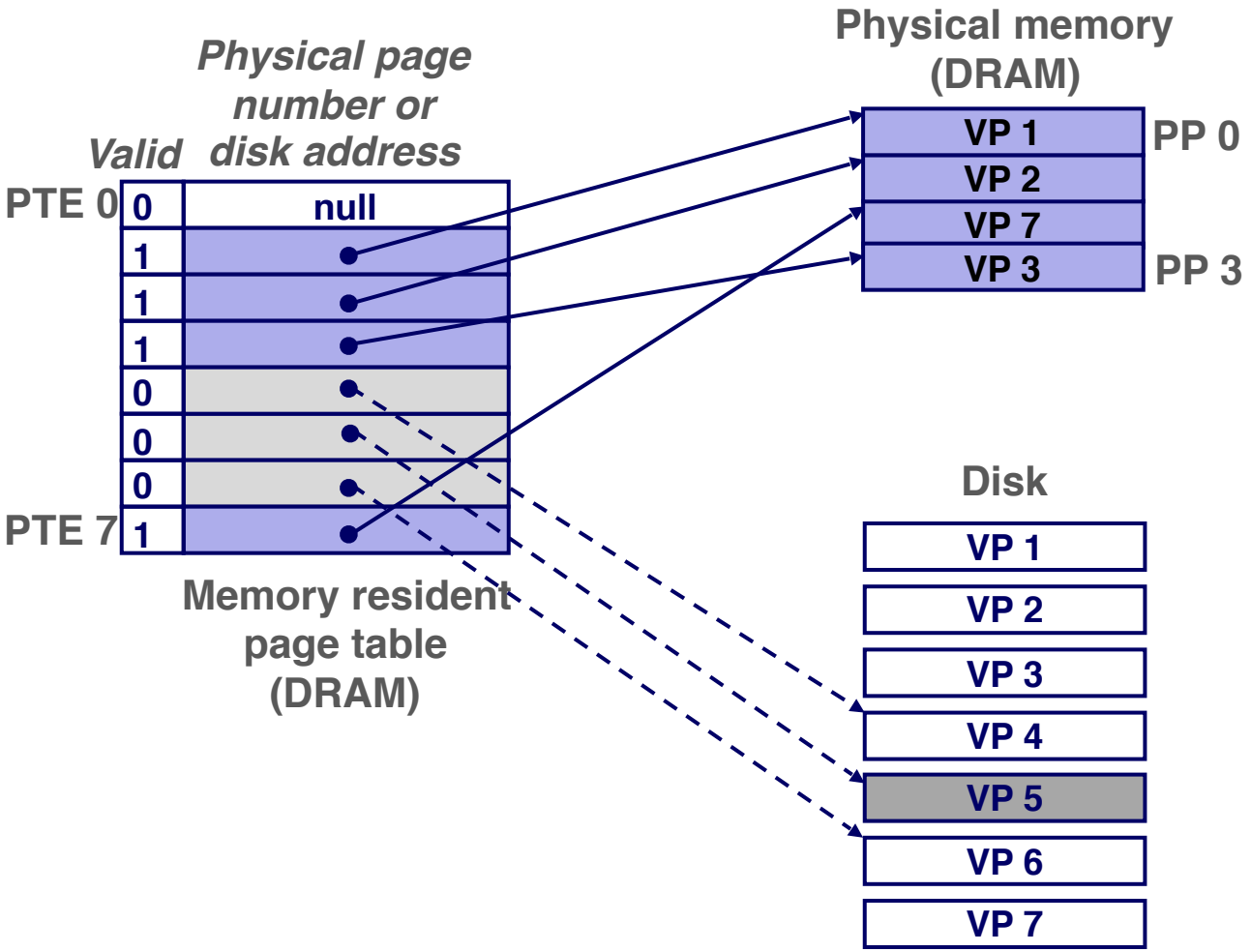
Allocating Pages

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



Allocating Pages

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



Virtual Memory Exploits Locality (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 initial misses

- If (SUM(working set sizes) > main memory size)
  - *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM
- M << N

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM
- M << N
- On a 64-bit machine, virtual memory size = 2^{64}

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM
- M << N
- On a 64-bit machine, virtual memory size = 2^{64}
- Physical memory size is much much smaller:
  - iPhone 8: 2 GB (2^{31})
  - 15-inch Macbook Pro 2017: 16 GB (2^{34})

VM Concepts Summary

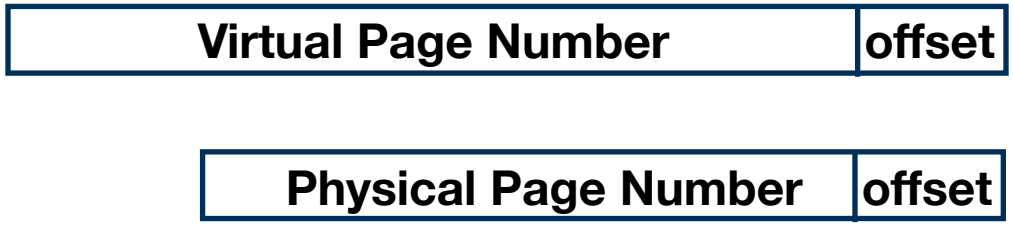
- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM
- M << N
- On a 64-bit machine, virtual memory size = 2^{64}
- Physical memory size is much much smaller:
  - iPhone 8: 2 GB (2^{31})
  - 15-inch Macbook Pro 2017: 16 GB (2^{34})
- Store only the most frequently used pages in the physical memory

VM Concepts Summary

- Conceptually, *virtual memory* is an array of N *pages*
  - Each virtual page is either in physical memory, or on disk, or unallocated
- The physical memory (PM) is an array of M *pages* stored in DRAM.
- Page size is the same for VM and PM
- M << N
- On a 64-bit machine, virtual memory size = 2^{64}
- Physical memory size is much much smaller:
  - iPhone 8: 2 GB (2^{31})
  - 15-inch Macbook Pro 2017: 16 GB (2^{34})
- Store only the most frequently used pages in the physical memory
- If a page is not on the physical memory, have to first swap it from the disk to the DRAM.

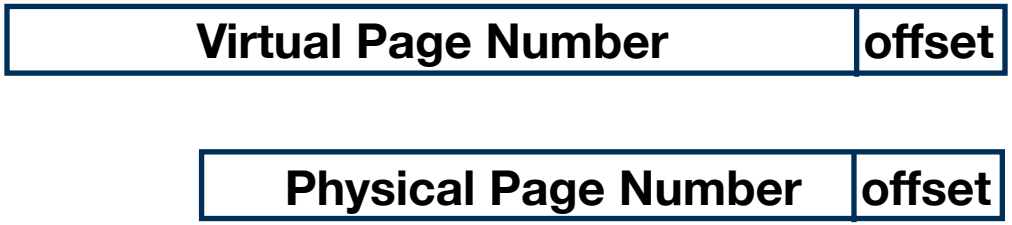
Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.



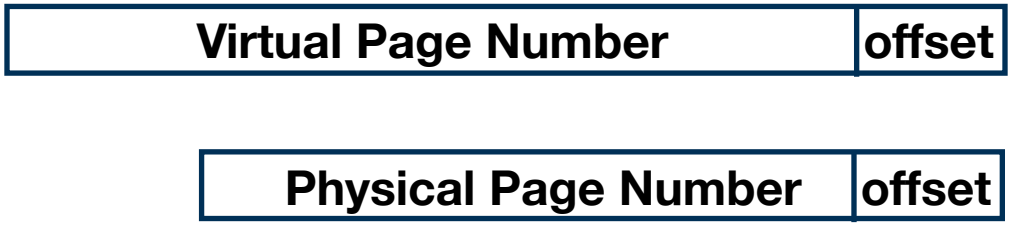
Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
- How many bits for page offset?
  - 12. Same for VM and PM



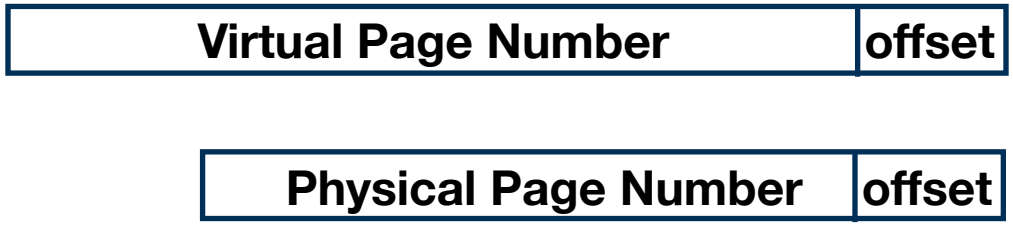
Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
- How many bits for page offset?
  - 12. Same for VM and PM
- How many bits for Virtual Page Number?
  - 52, i.e., 2^{52} virtual pages



Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
- How many bits for page offset?
  - 12. Same for VM and PM
- How many bits for Virtual Page Number?
  - 52, i.e., 2^{52} virtual pages
- How many bits for Physical Page Number?
  - 20, i.e., 2^{20} physical pages



Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes

Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
  - 4GB/4KB = 1M virtual pages

Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
  - 4GB/4KB = 1M virtual pages
  - 1M PTEs in a page table

Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
  - 4GB/4KB = 1M virtual pages
  - 1M PTEs in a page table
  - 8MB total size per page table

Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
  - 4GB/4KB = 1M virtual pages
  - 1M PTEs in a page table
  - 8MB total size per page table
- Do you need a page table for each process?

Calculate the Page Table Size

- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
  - 4GB/4KB = 1M virtual pages
  - 1M PTEs in a page table
  - 8MB total size per page table
- Do you need a page table for each process?
  - Yes

Where Does Page Table Live?

- It needs to be at a specific location where we can find it
  - Some special SRAM?
  - In main memory?
  - On disk?

Where Does Page Table Live?

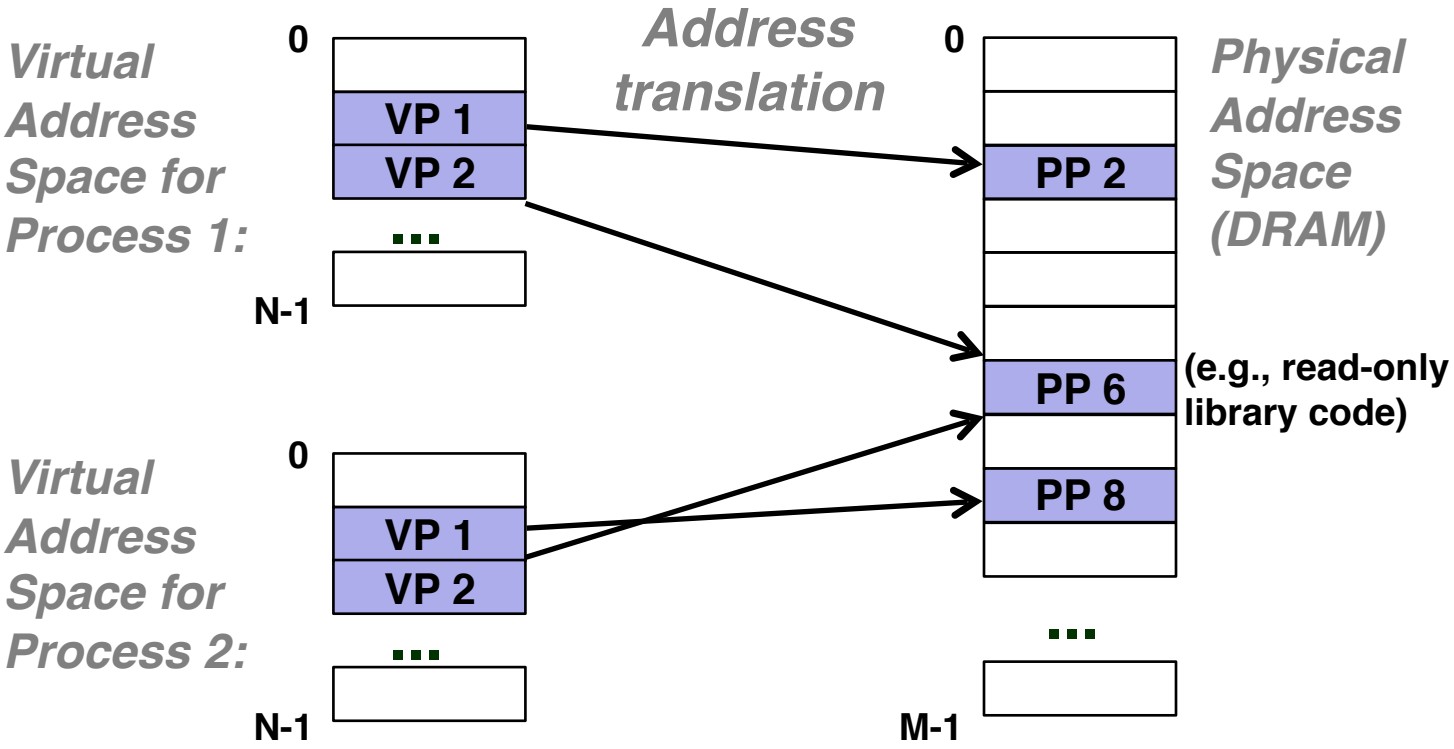
- It needs to be at a specific location where we can find it
  - Some special SRAM?
  - In main memory?
  - On disk?
- ~MBs of a page table *per process*
  - Too big for on-chip SRAM (c.f., a L1 cache is ~32 KB)
  - Too slow to access in disk
  - Put the page table in DRAM, with its start address stored in a special register (Page Table Base Register). More on this later.

Today

- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

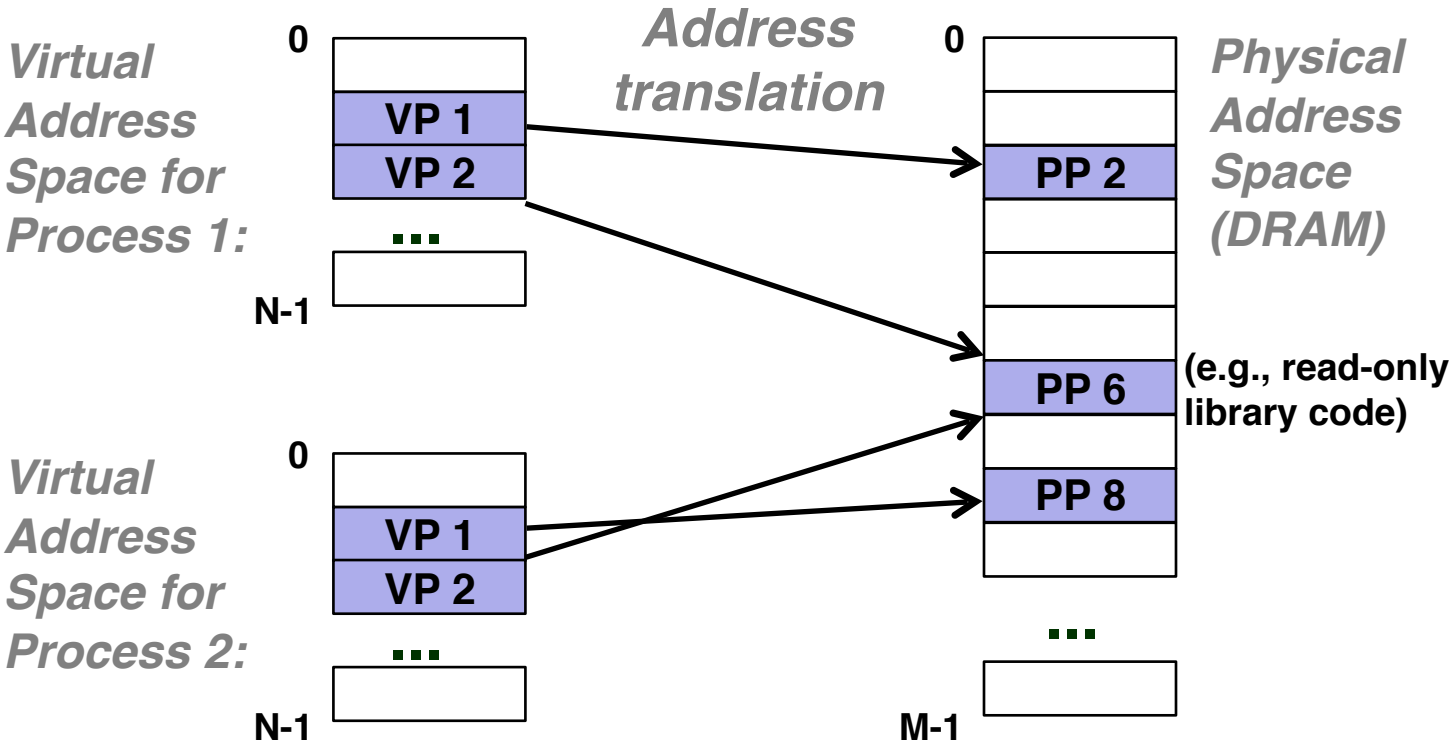
VM as a Tool for Memory Management

- Each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping scatters addresses through physical memory
    - Well-chosen mappings can improve locality



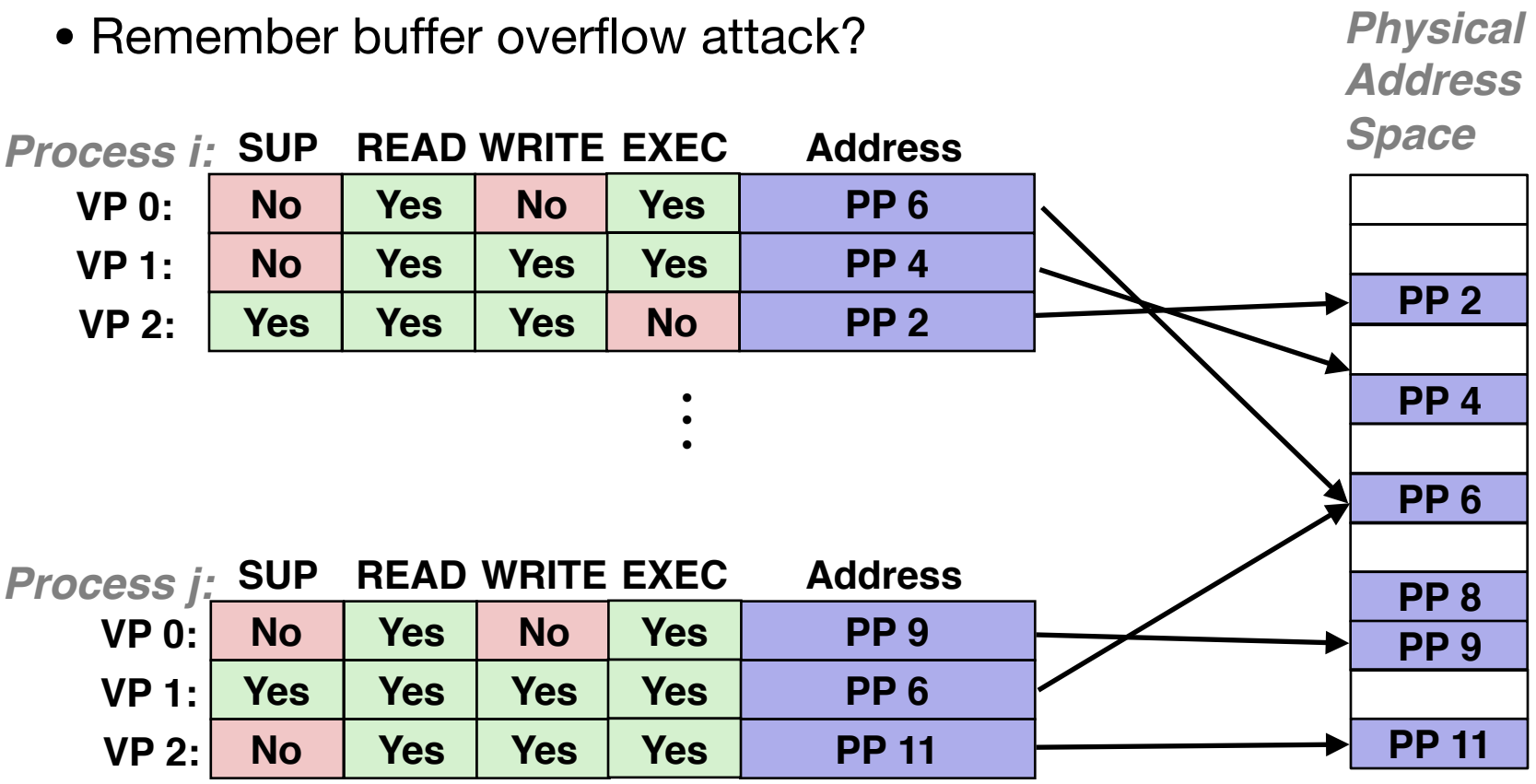
Virtual Memory Enables Sharing

- 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 Provides Further Protection Opportunities

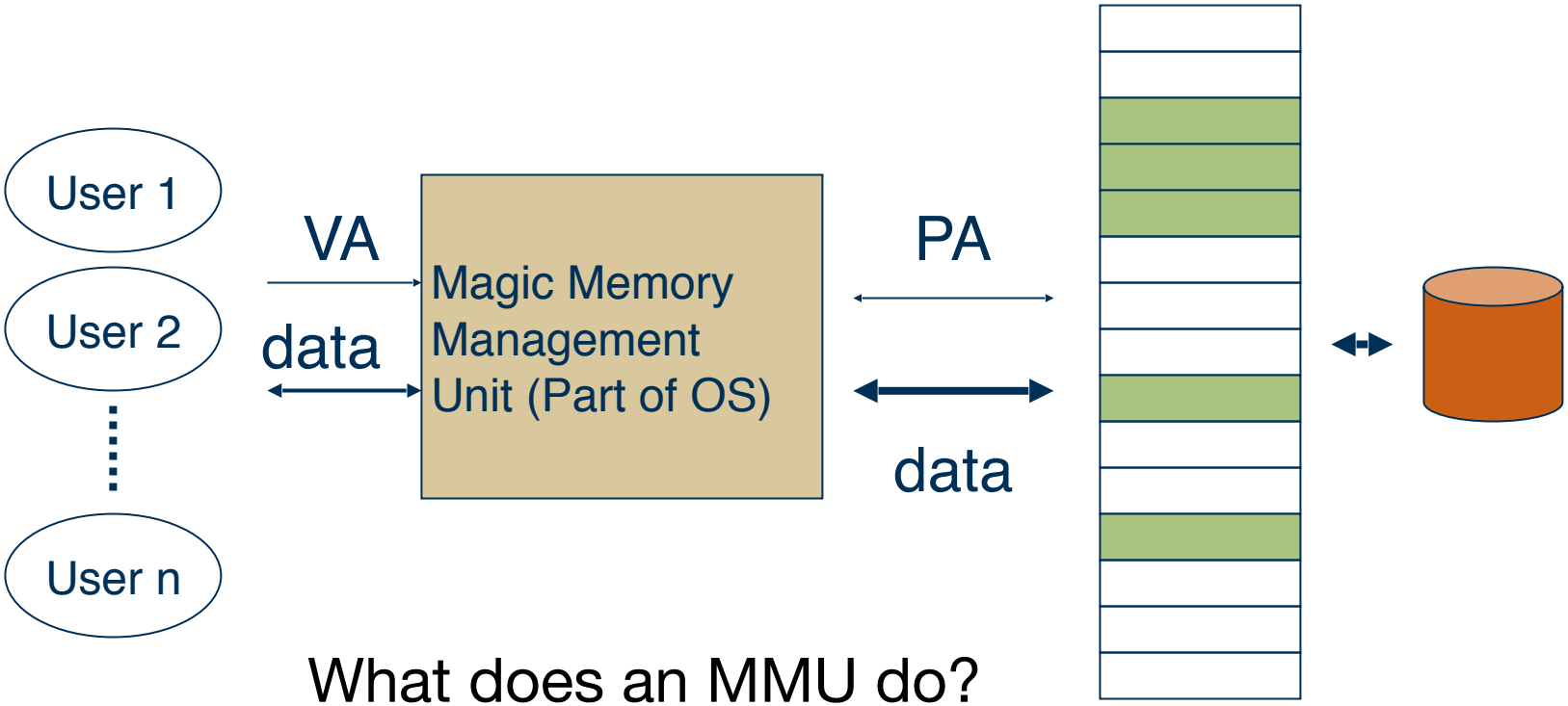
- Extend PTEs with permission bits
- MMU checks these bits on each access (read/write/executable/ accessible only in supervisor mode?)
- Remember buffer overflow attack?



Today

- VM basic concepts and operation
- Other critical benefits of VM
- Address translation

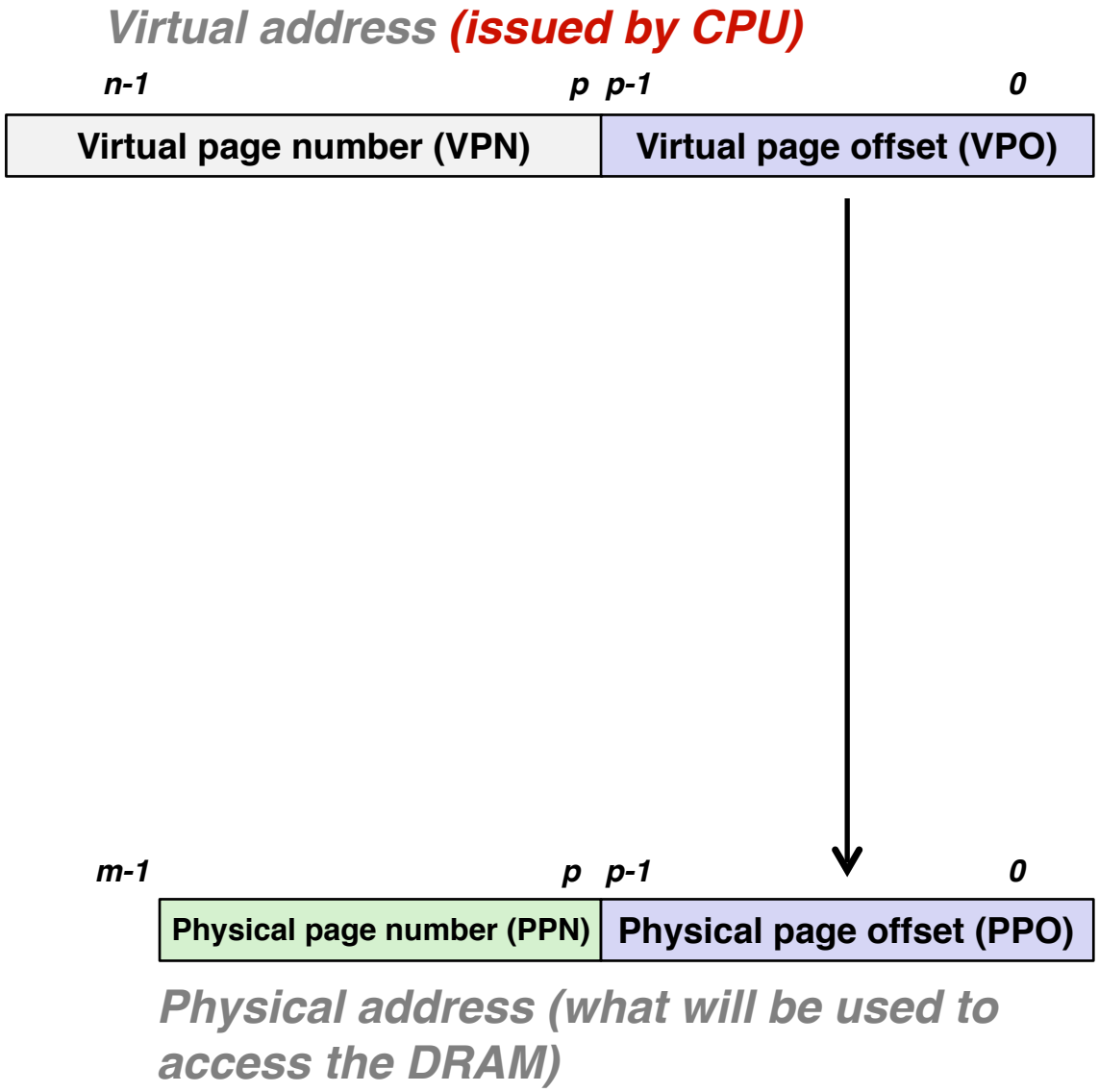
So Far…



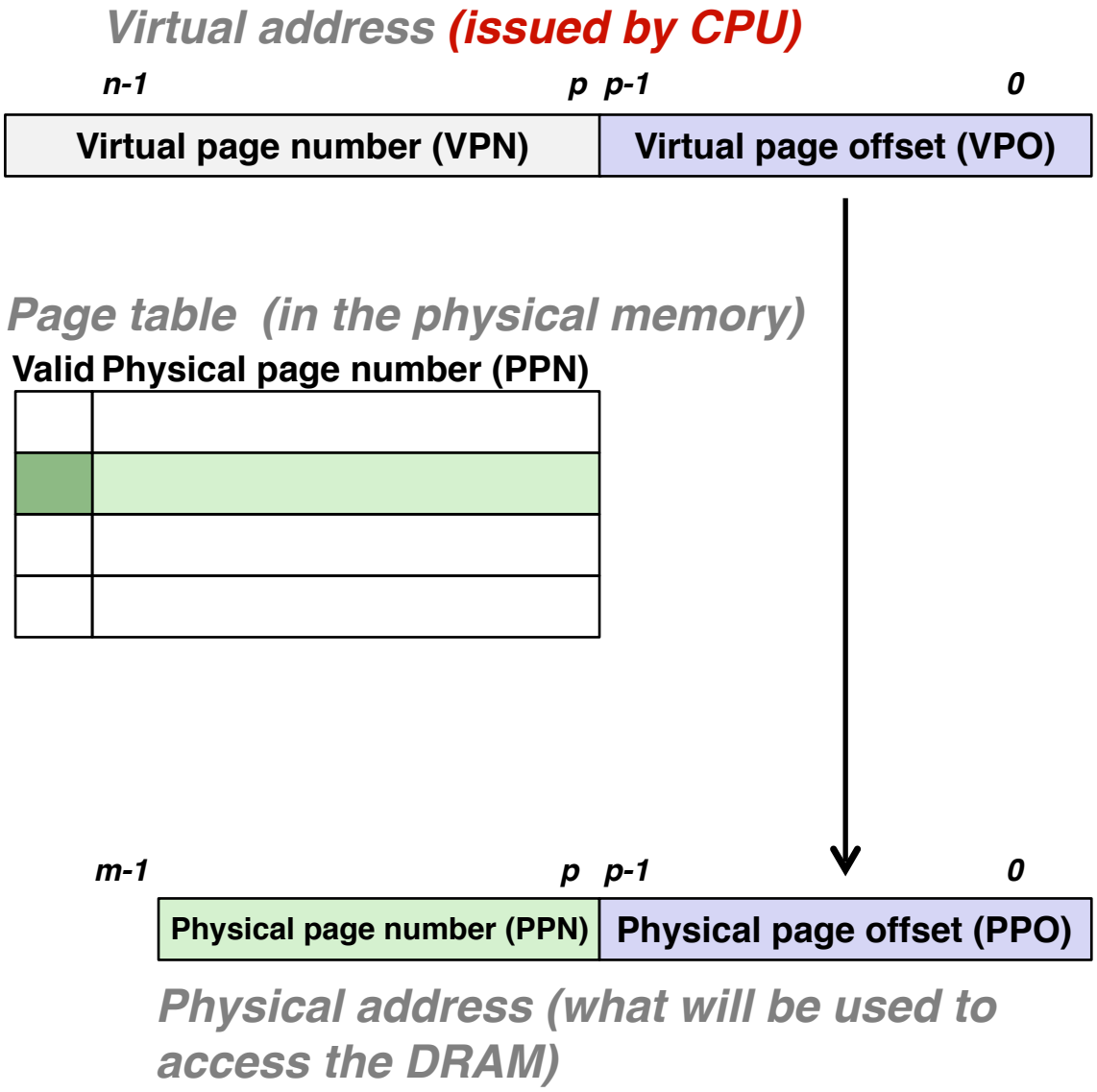
What does an MMU do?

- Translate address from a VA to PA
  - Enforce permissions
  - Fetch from disk

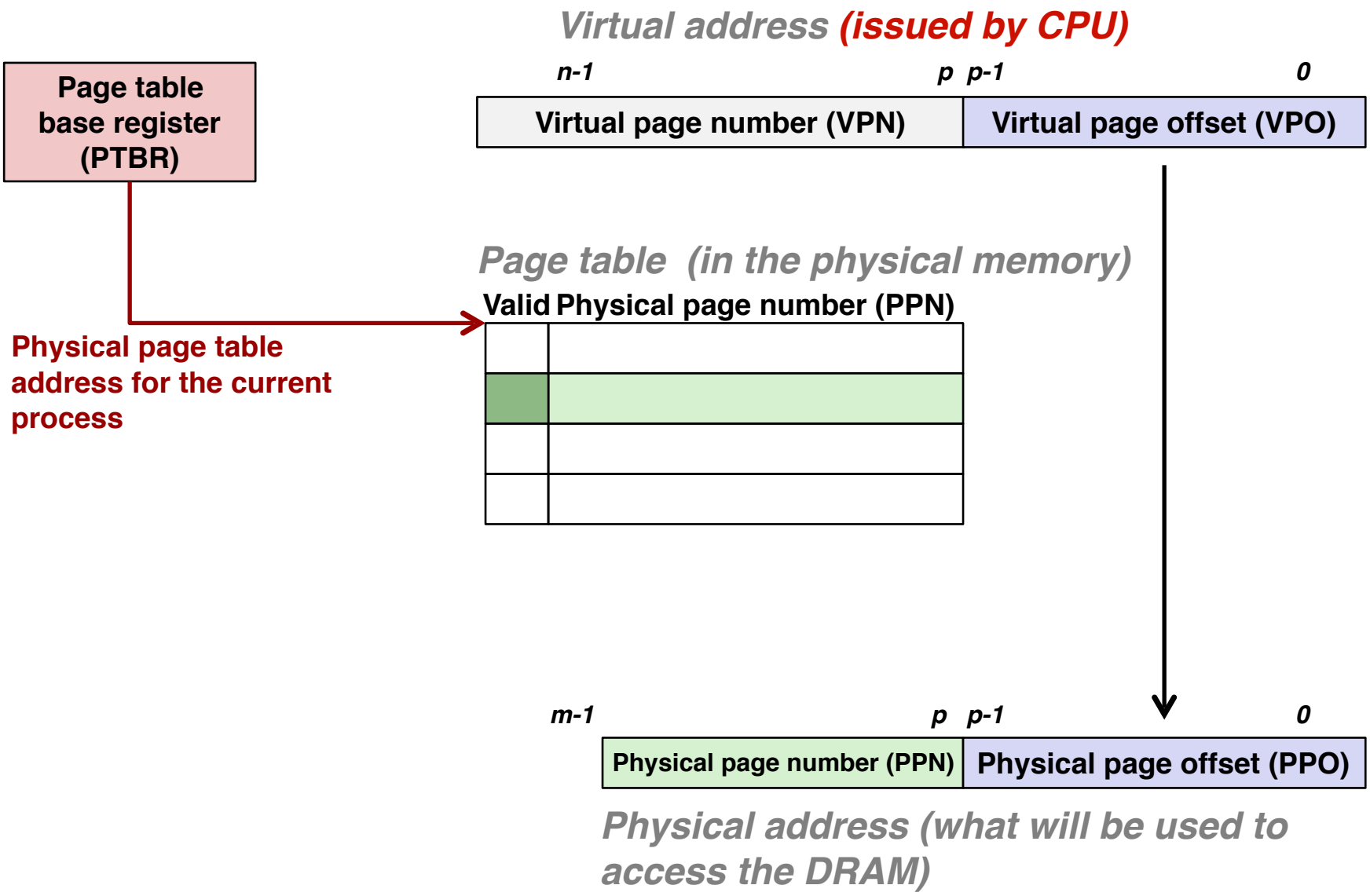
Address Translation With a Page Table



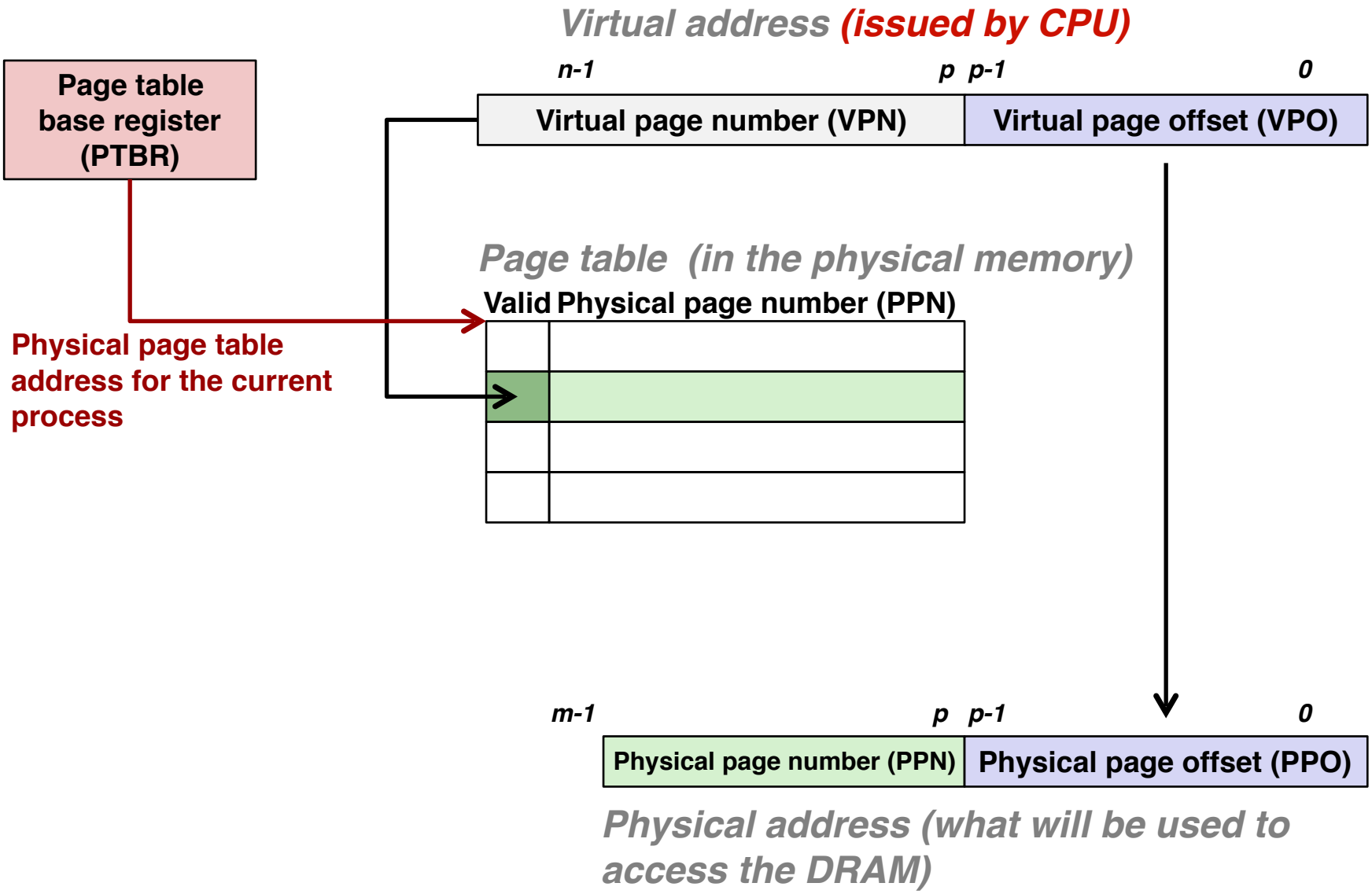
Address Translation With a Page Table



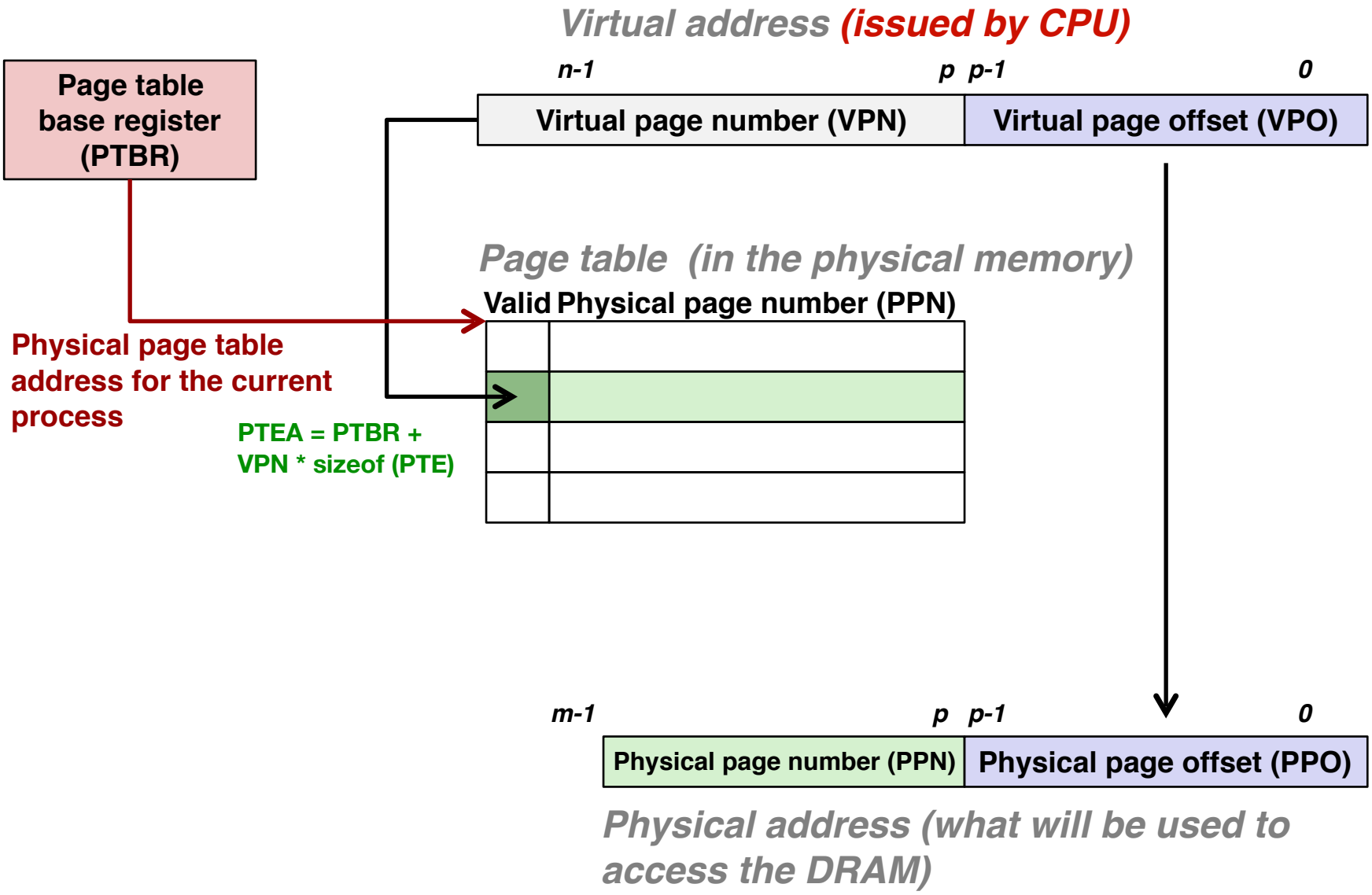
Address Translation With a Page Table



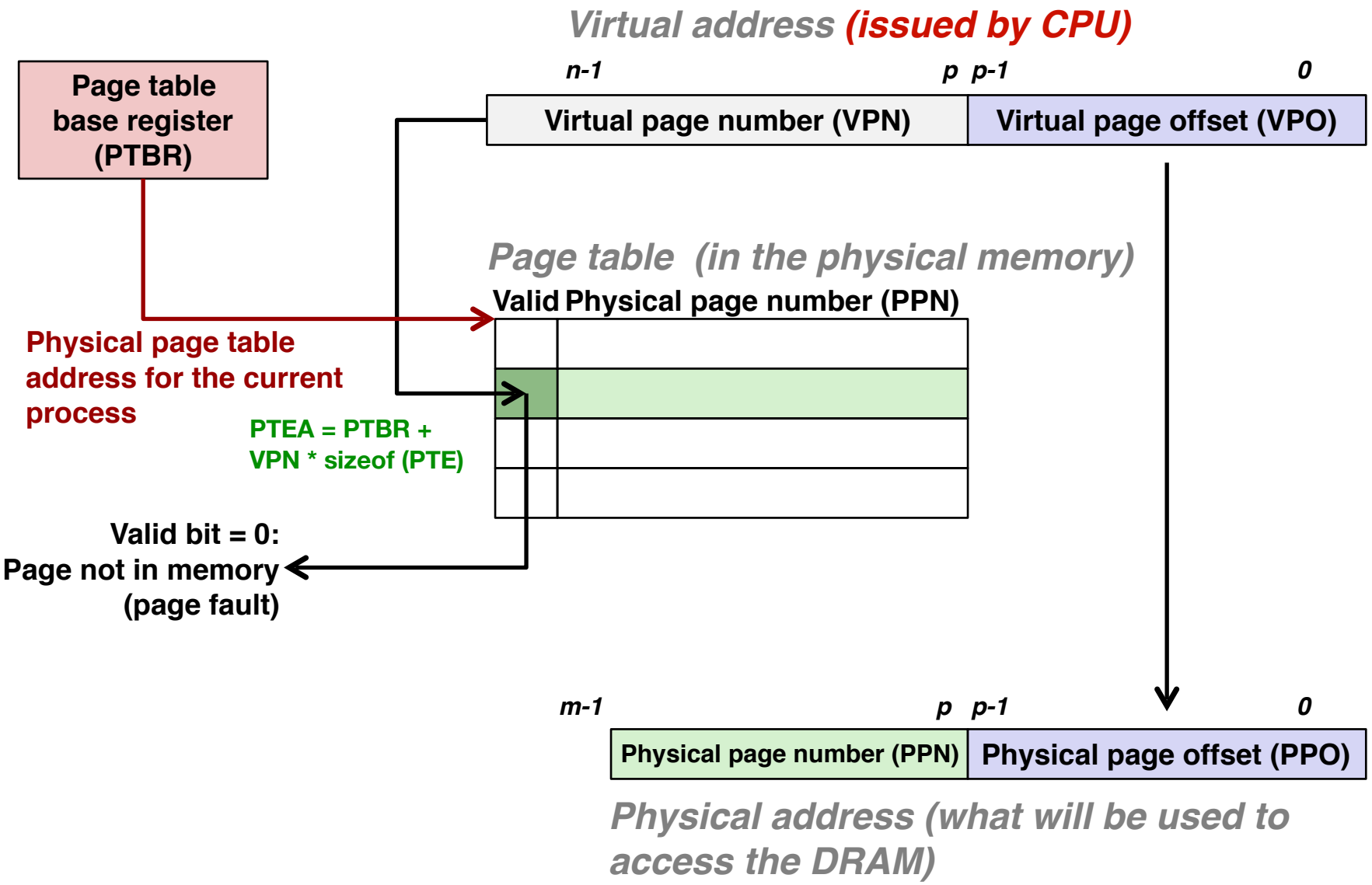
Address Translation With a Page Table



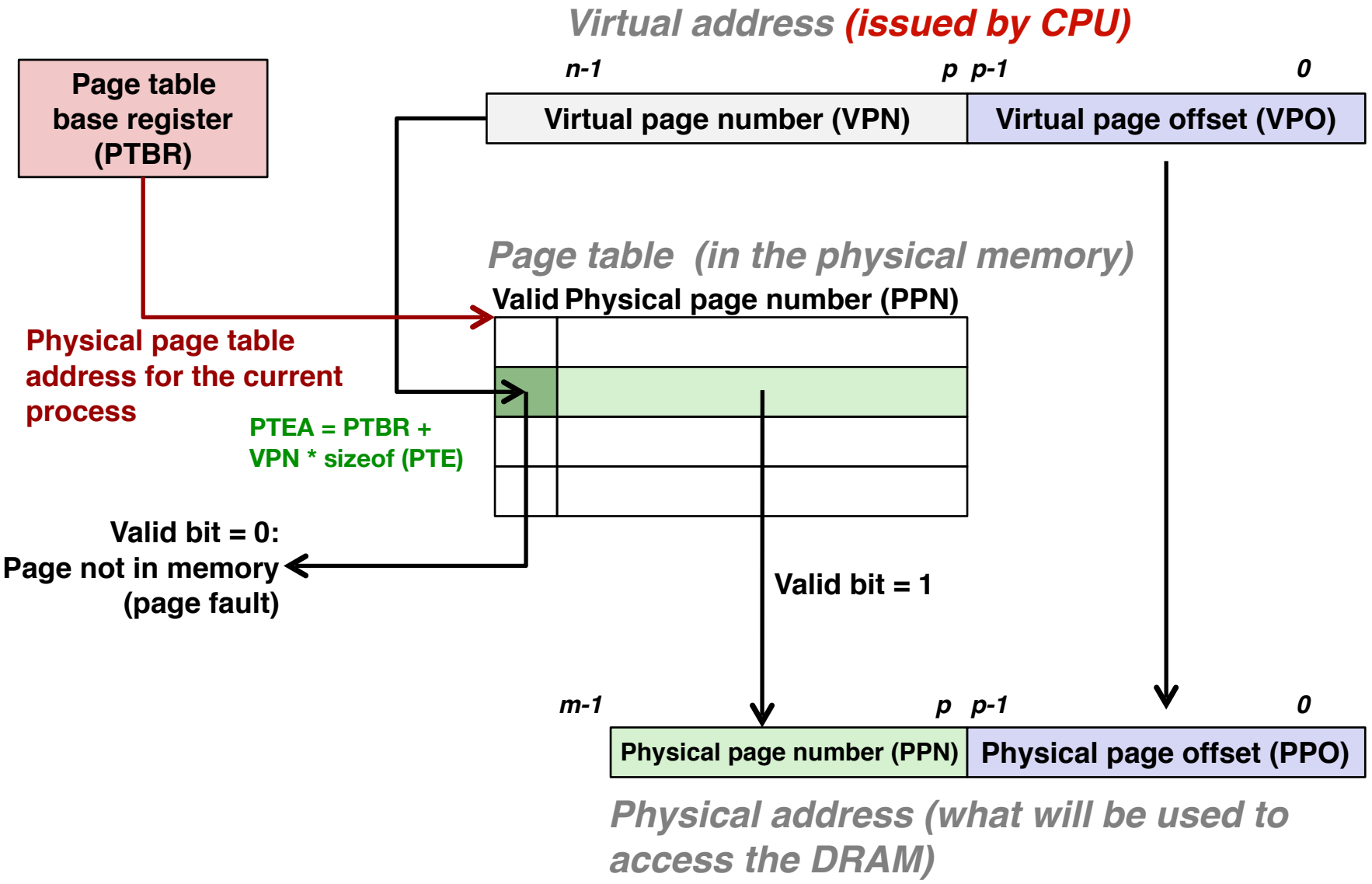
Address Translation With a Page Table



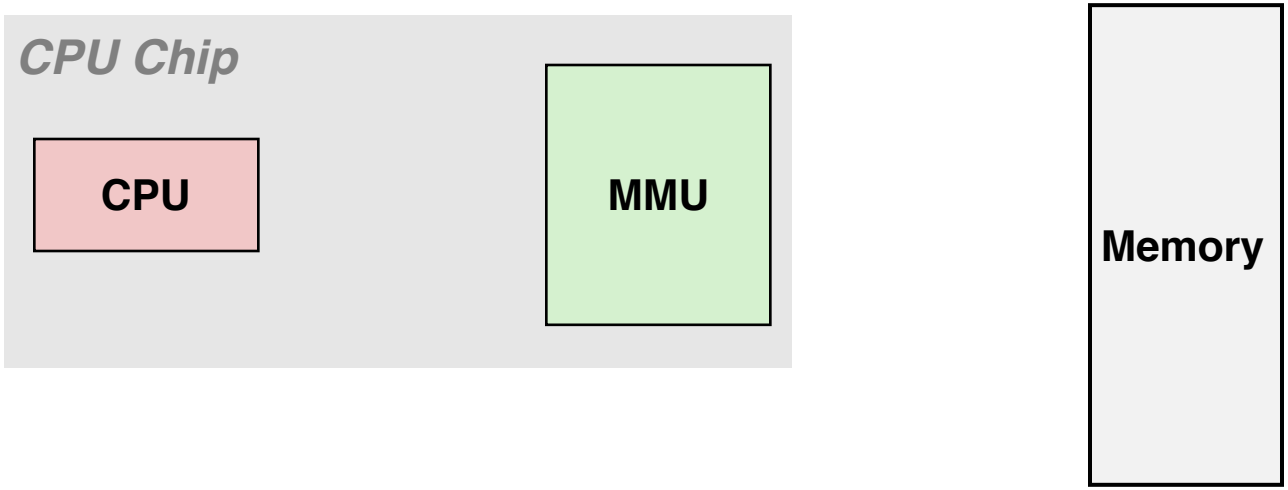
Address Translation With a Page Table



Address Translation With a Page Table

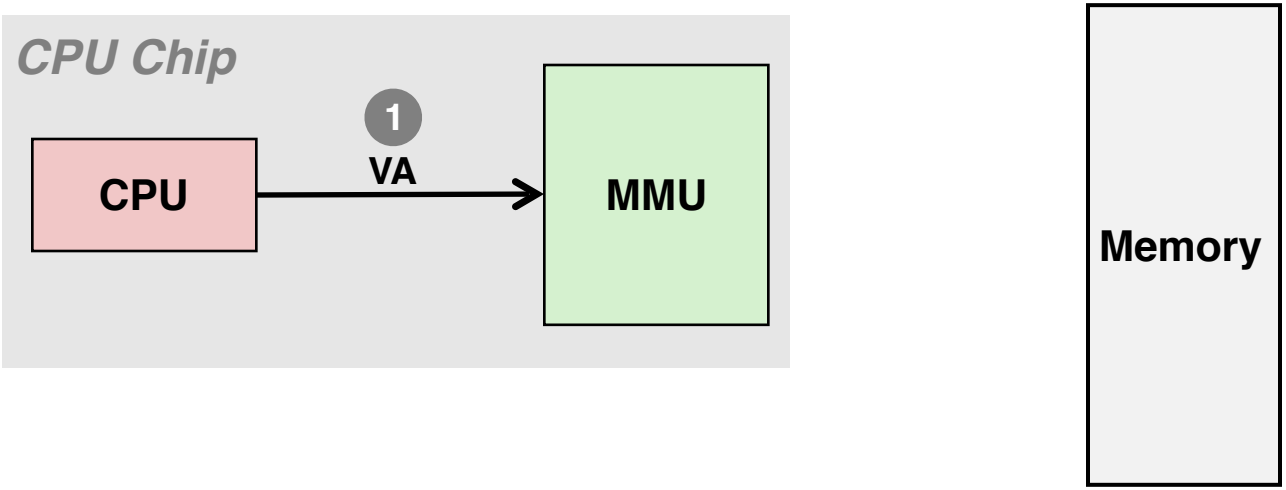


Address Translation: Page Hit



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

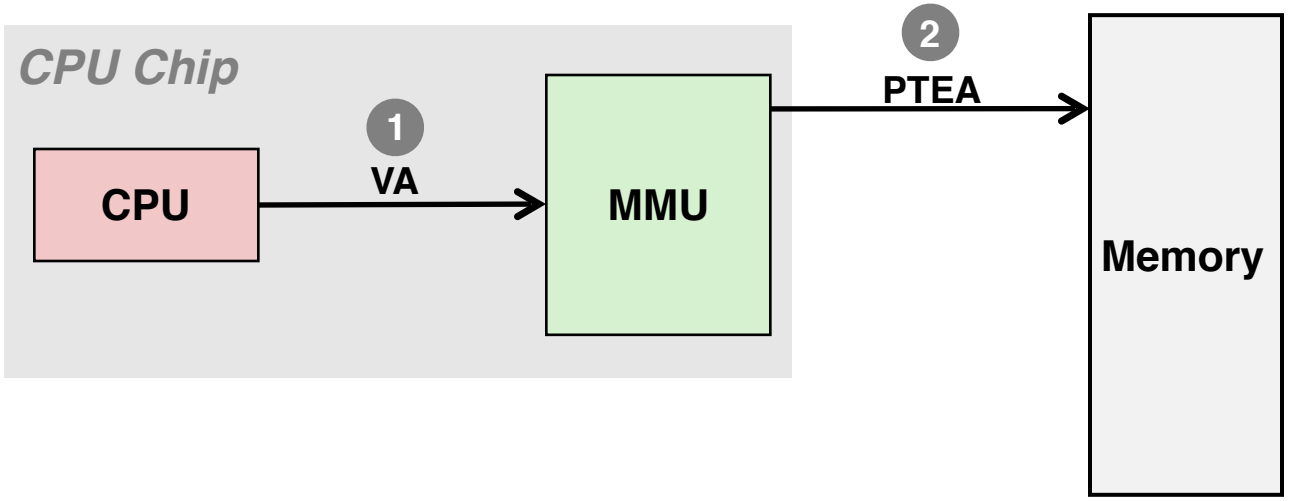
Address Translation: Page Hit



1) Processor sends virtual address to MMU

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

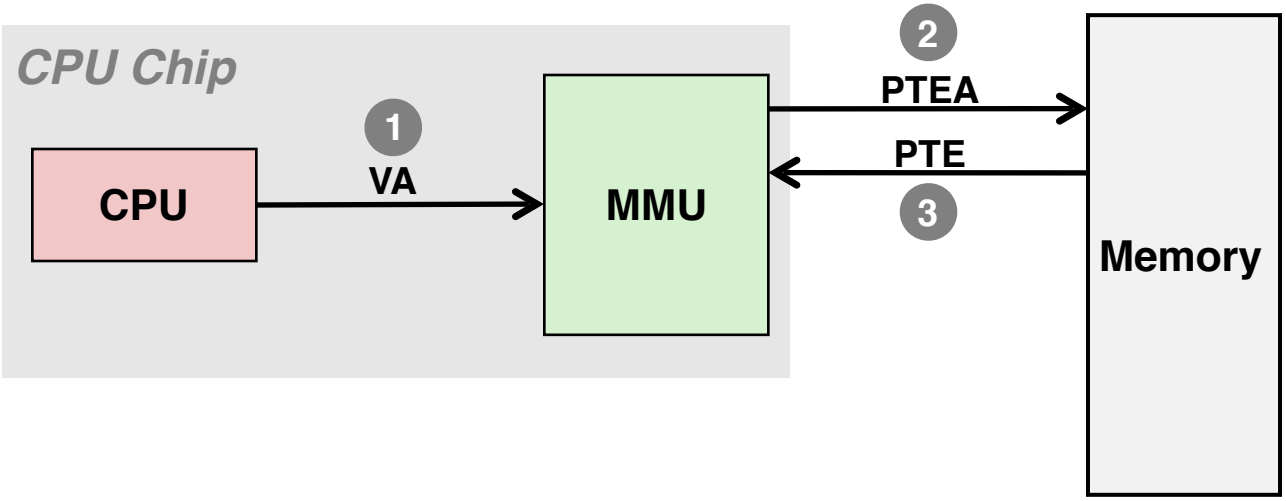
Address Translation: Page Hit



1) Processor sends virtual address to MMU

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

Address Translation: Page Hit

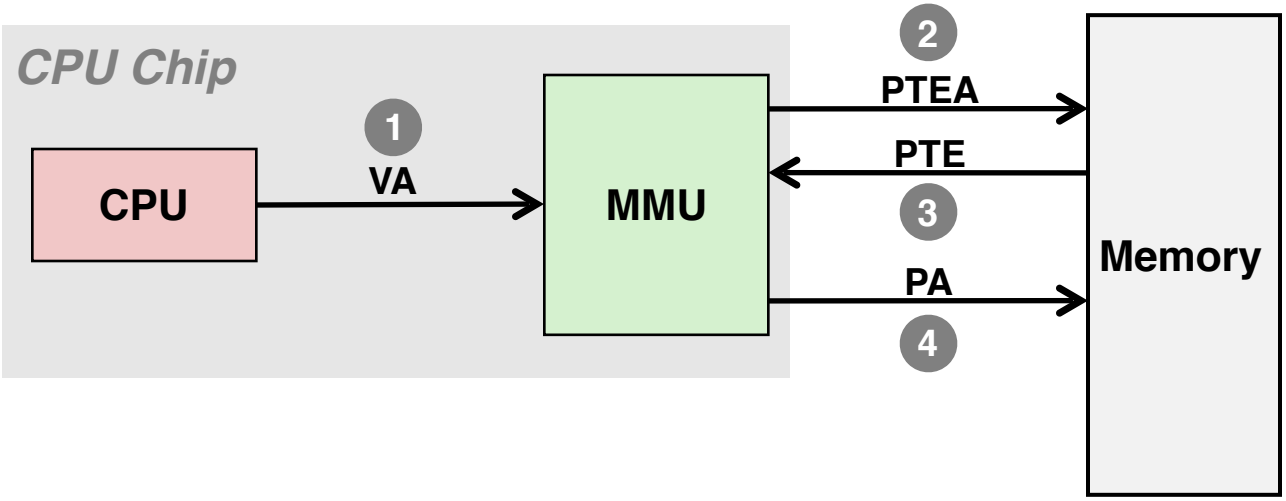


1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

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

Address Translation: Page Hit



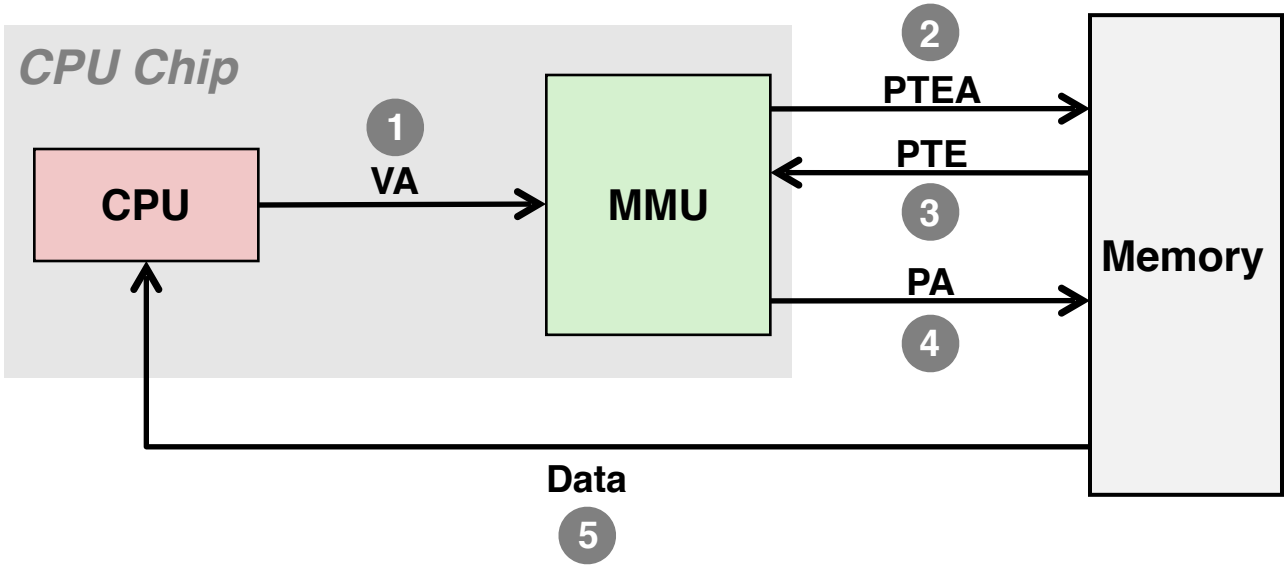
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

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

Address Translation: Page Hit



1) Processor sends virtual address to MMU

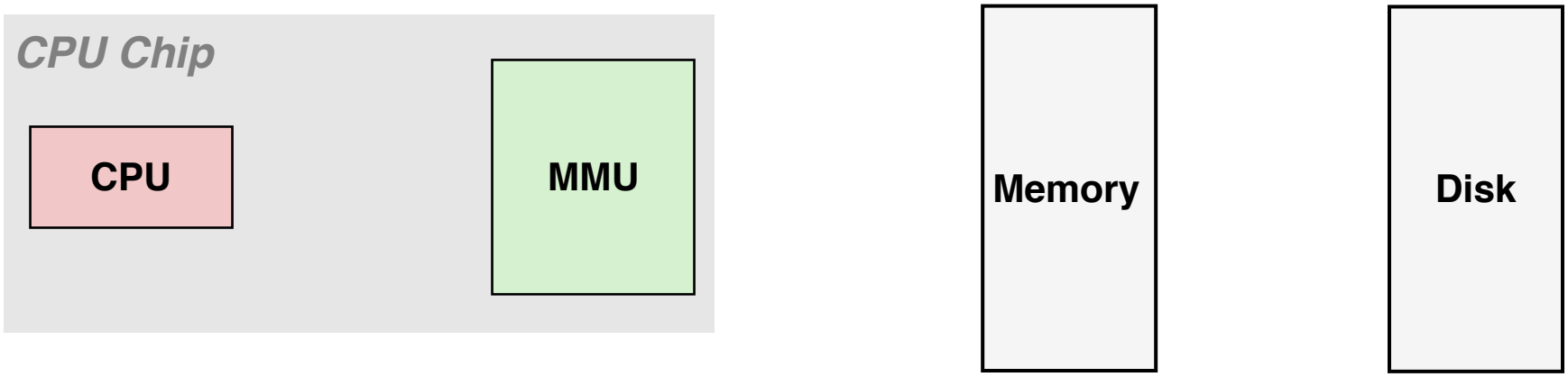
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

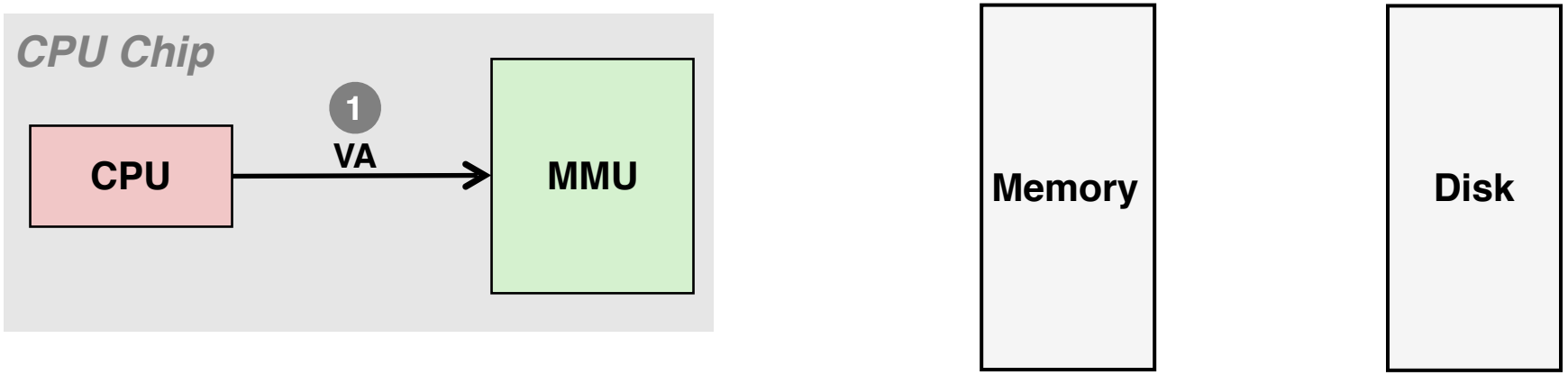
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Address Translation: Page Fault



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

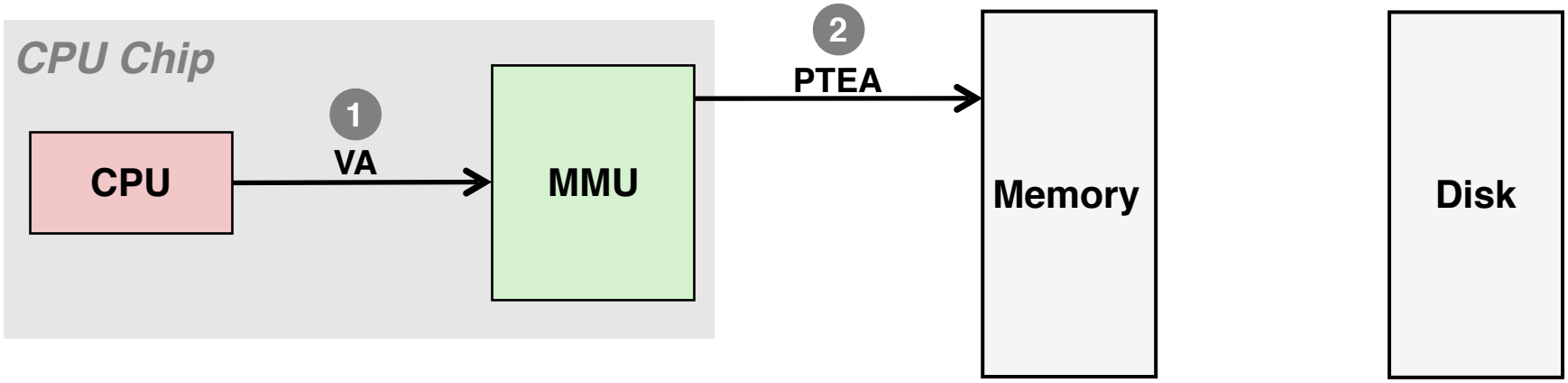
Address Translation: Page Fault



1) Processor sends virtual address to MMU

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

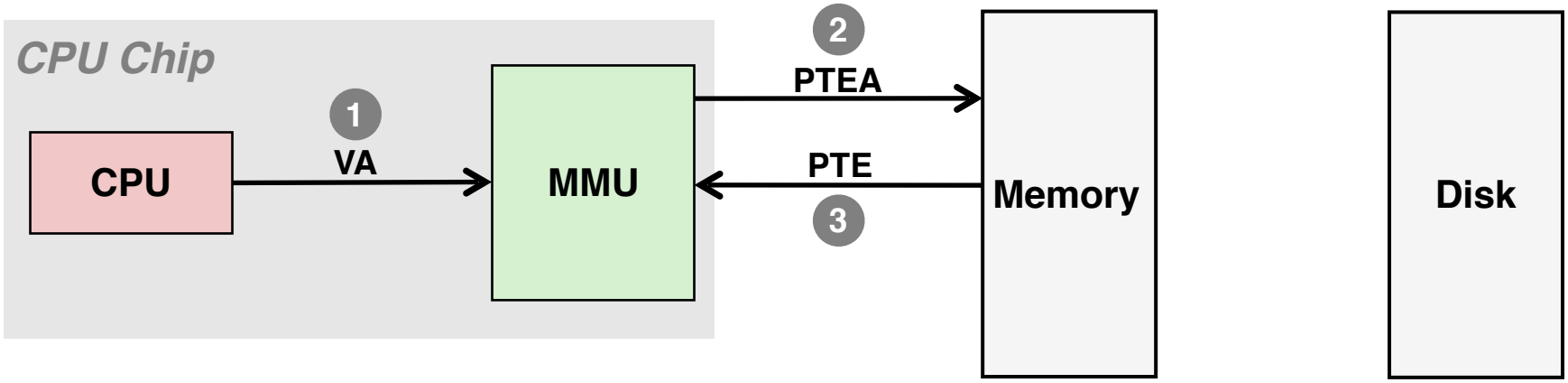
Address Translation: Page Fault



1) Processor sends virtual address to MMU

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

Address Translation: Page Fault

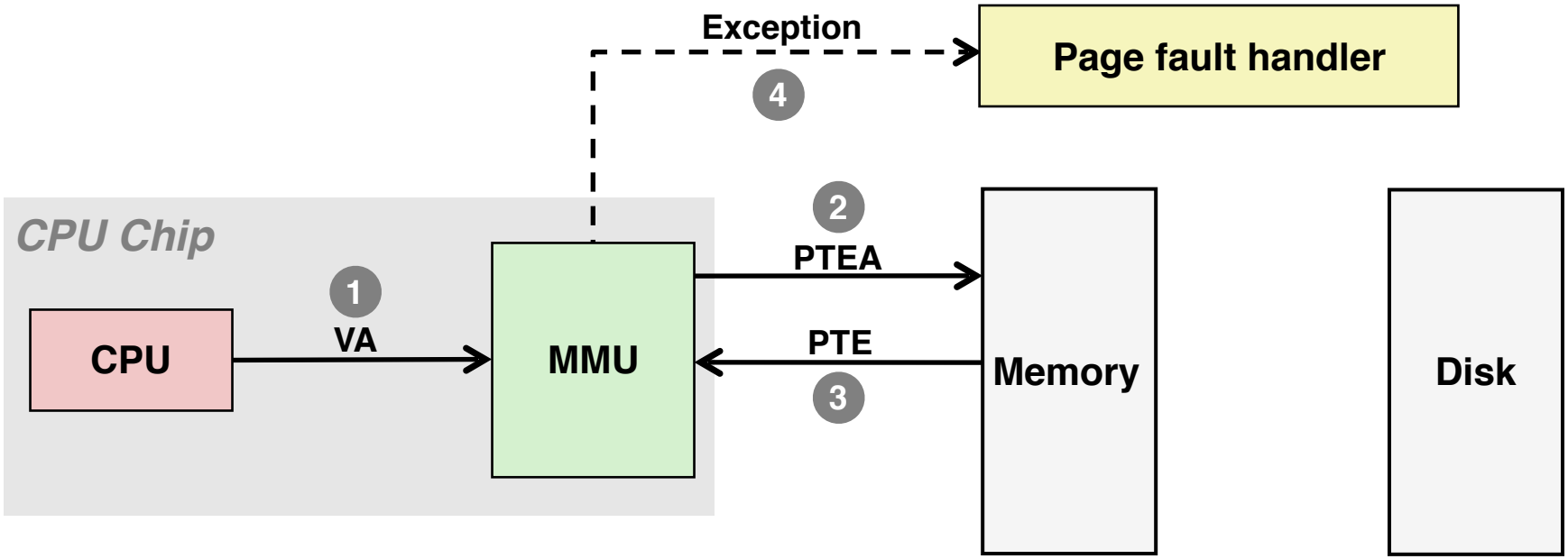


1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

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

Address Translation: Page Fault



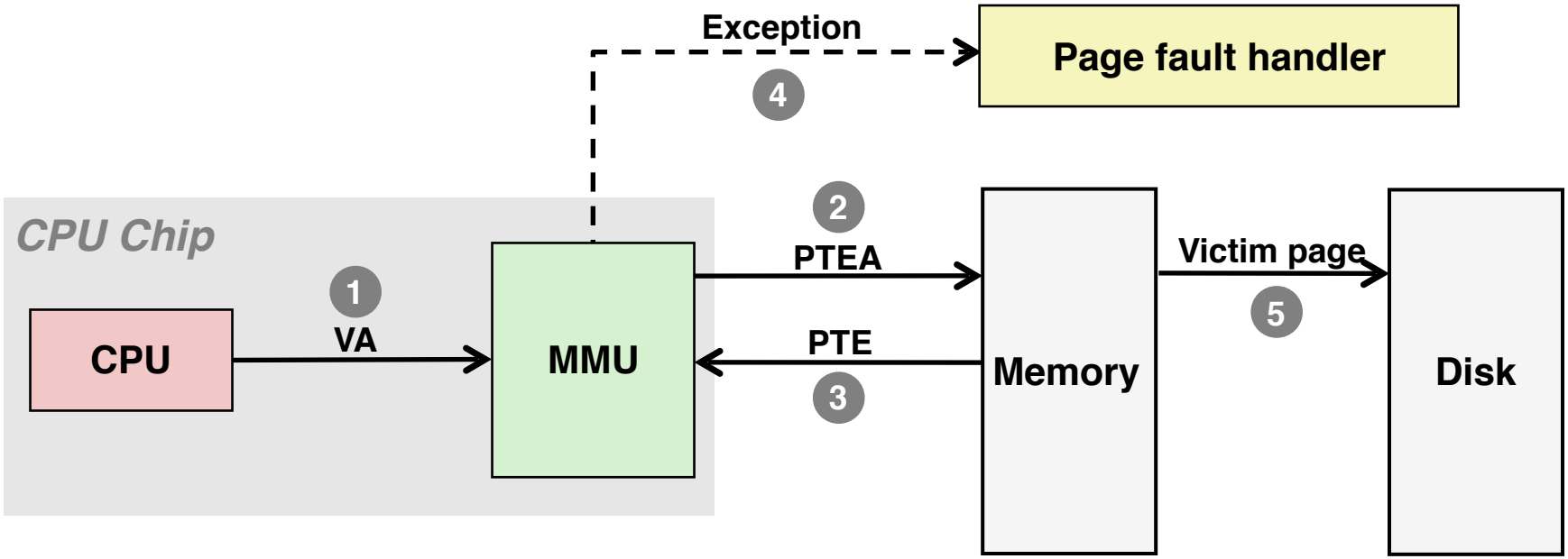
1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

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

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

Address Translation: Page Fault



1) Processor sends virtual address to MMU

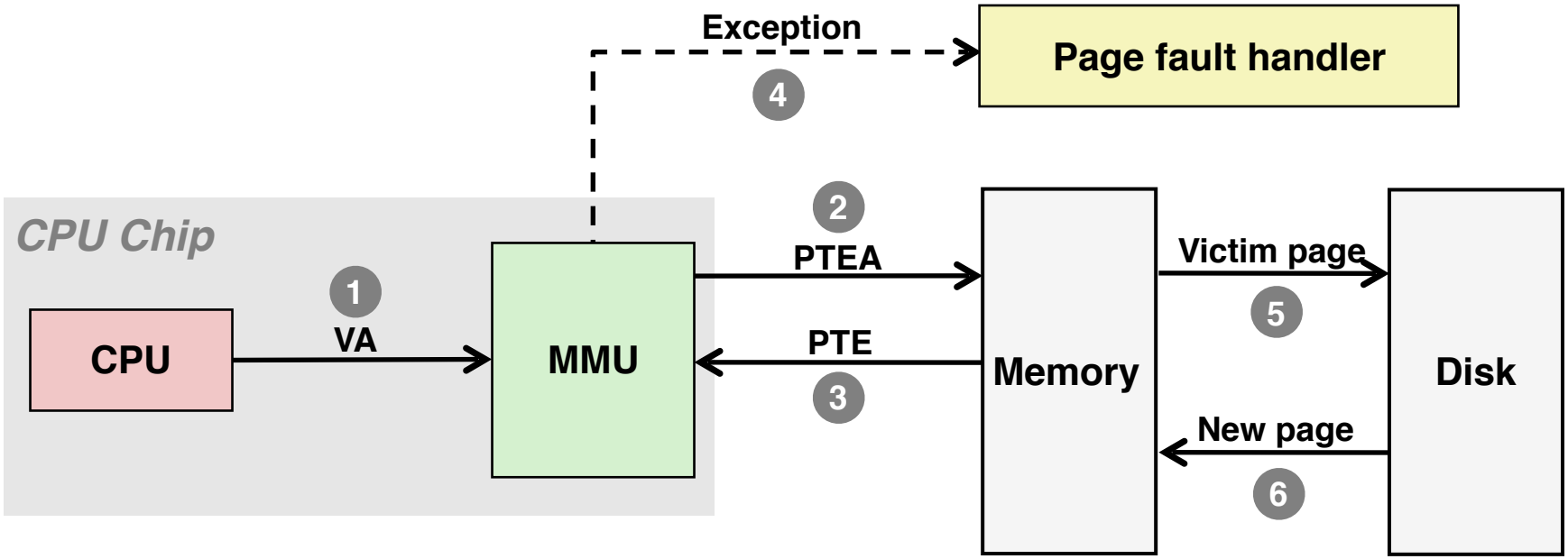
2-3) MMU fetches PTE from page table in memory

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

5) Handler identifies victim (and, if dirty, pages it out to disk)

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

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

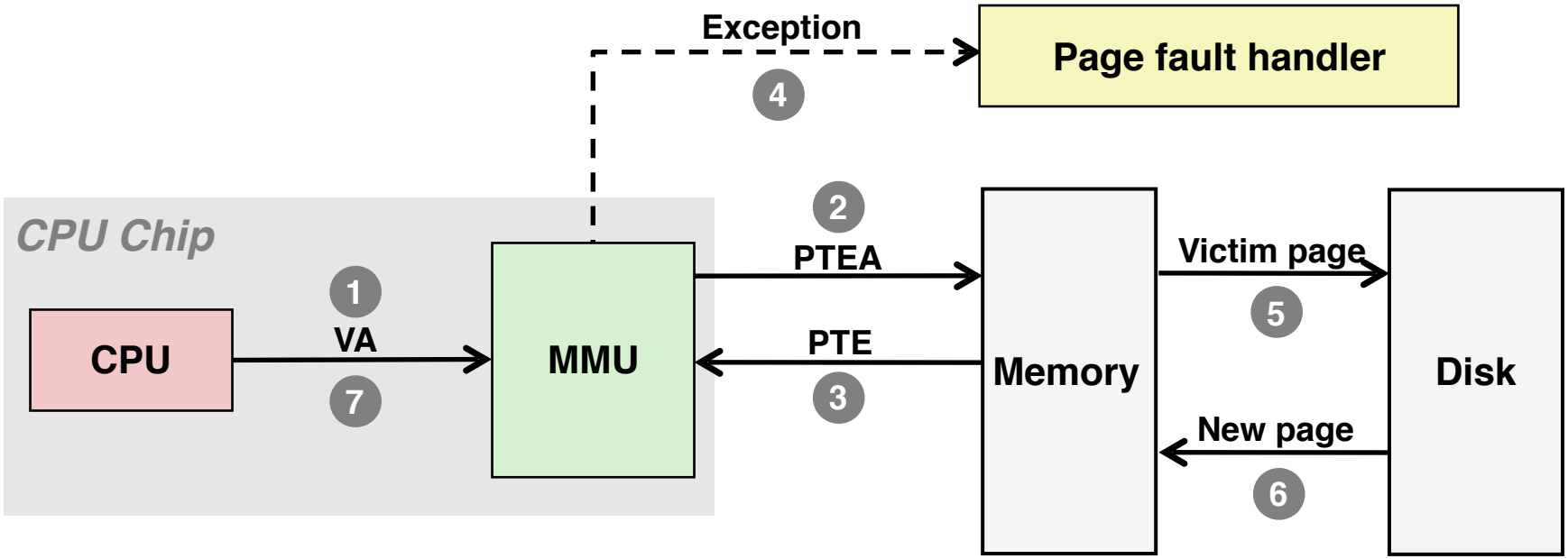
4) Valid bit is zero, so MMU triggers page fault exception

5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

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

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

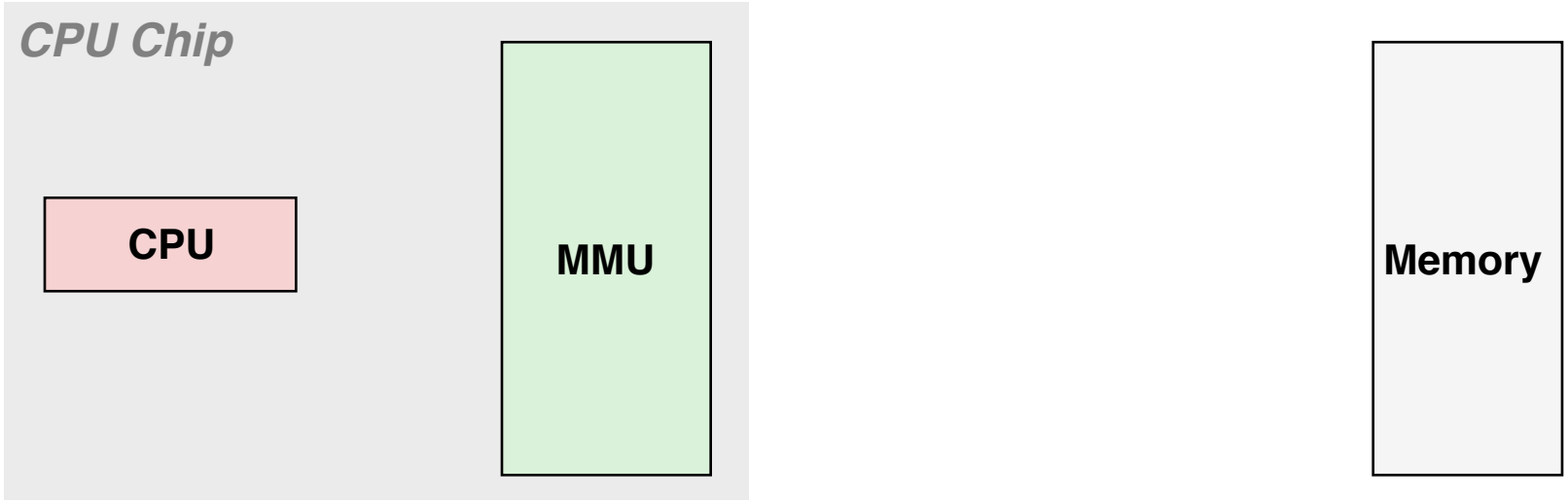
5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

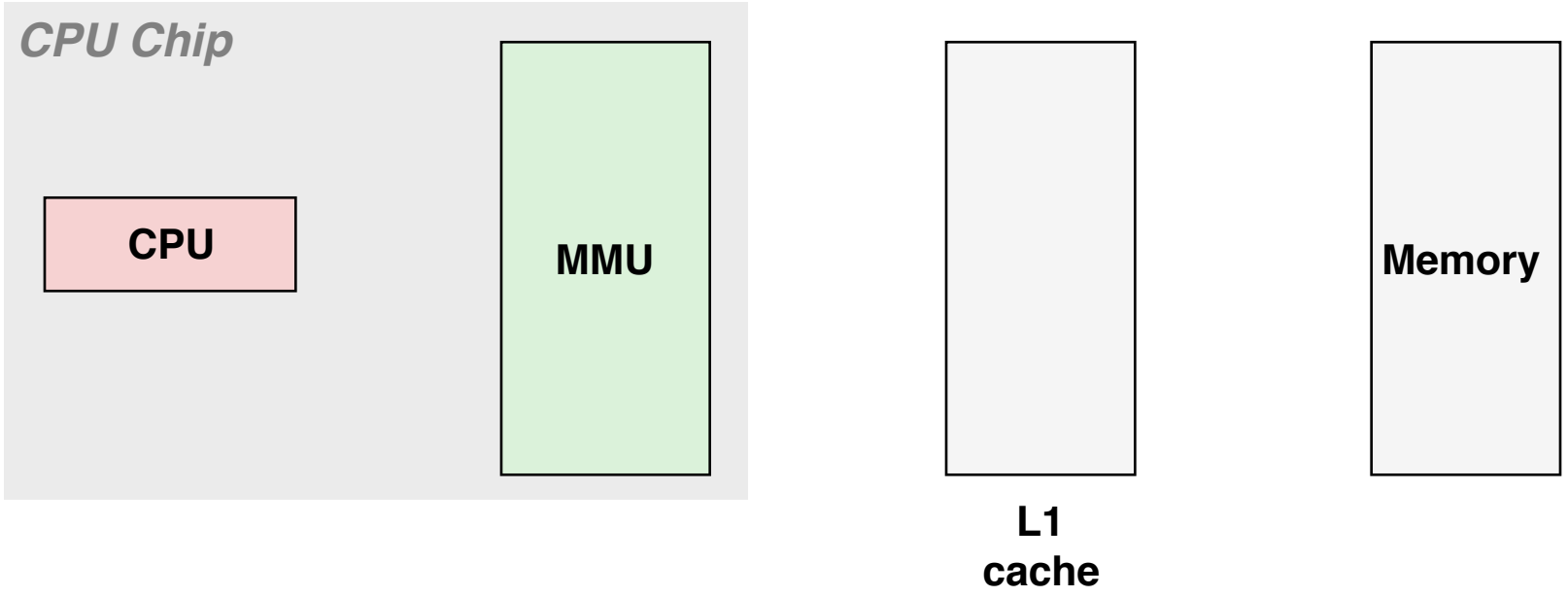
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



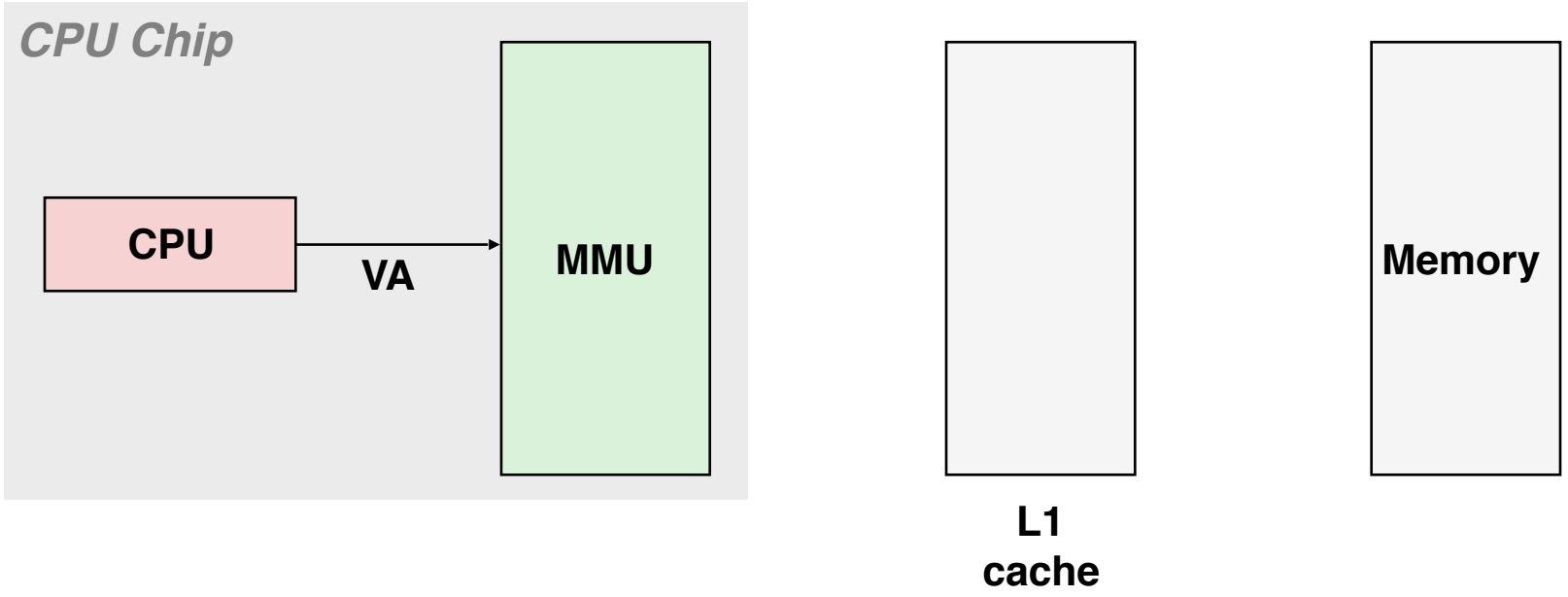
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



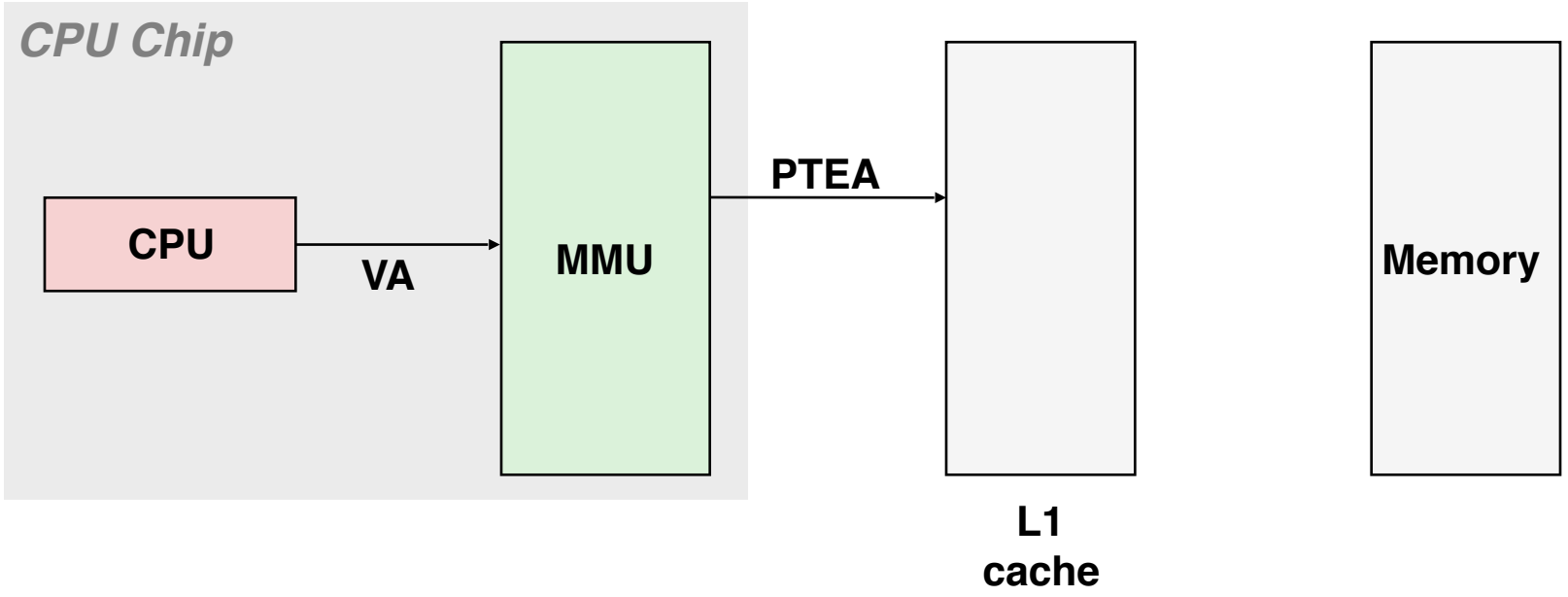
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



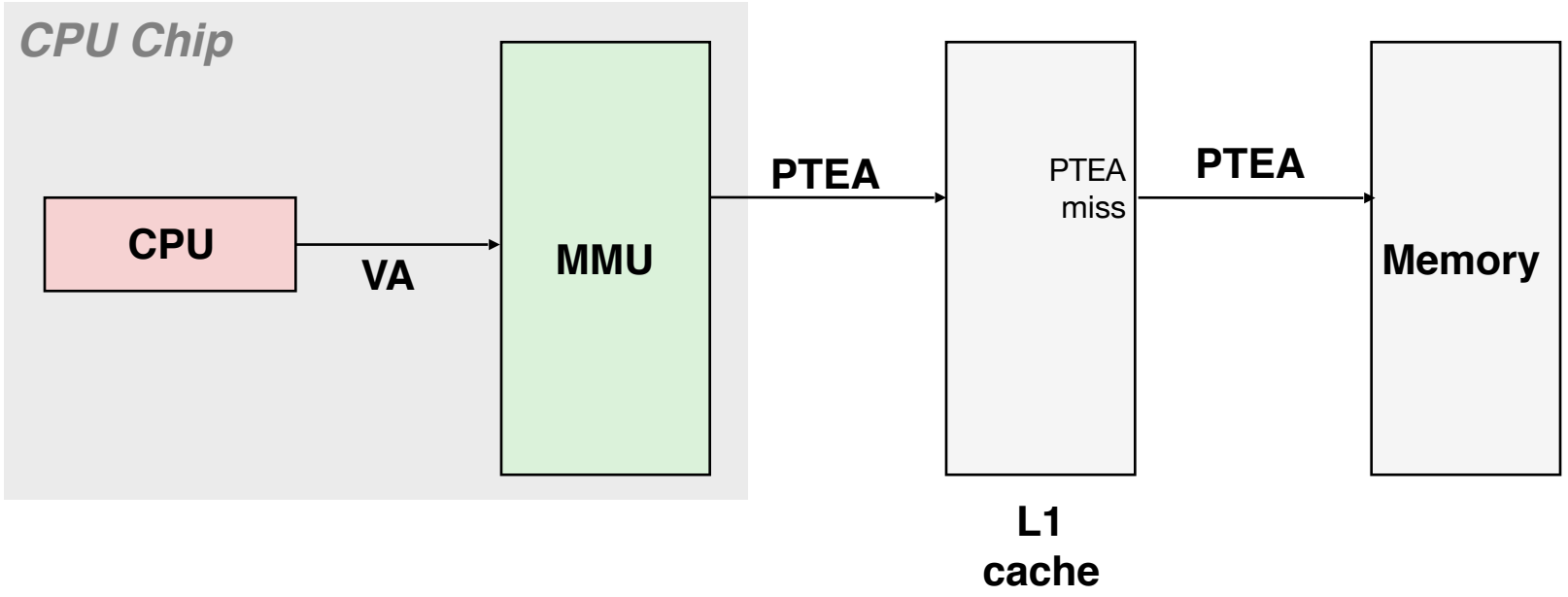
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



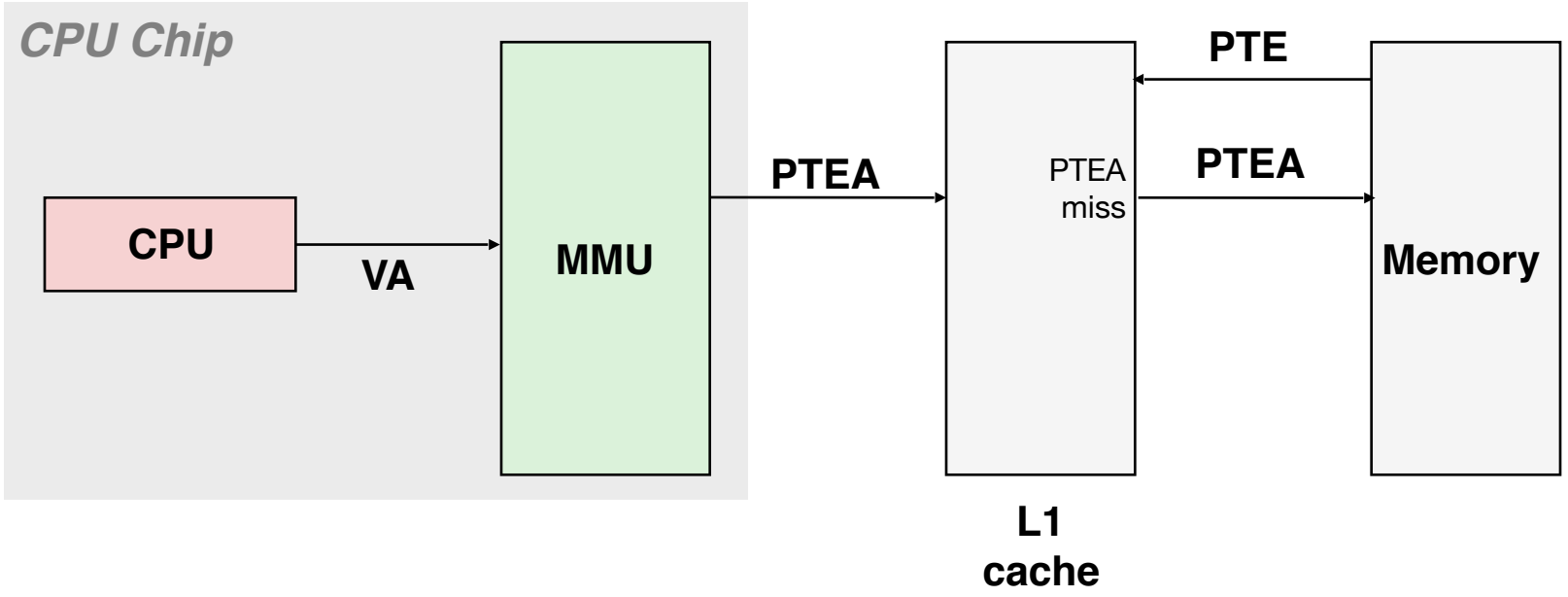
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



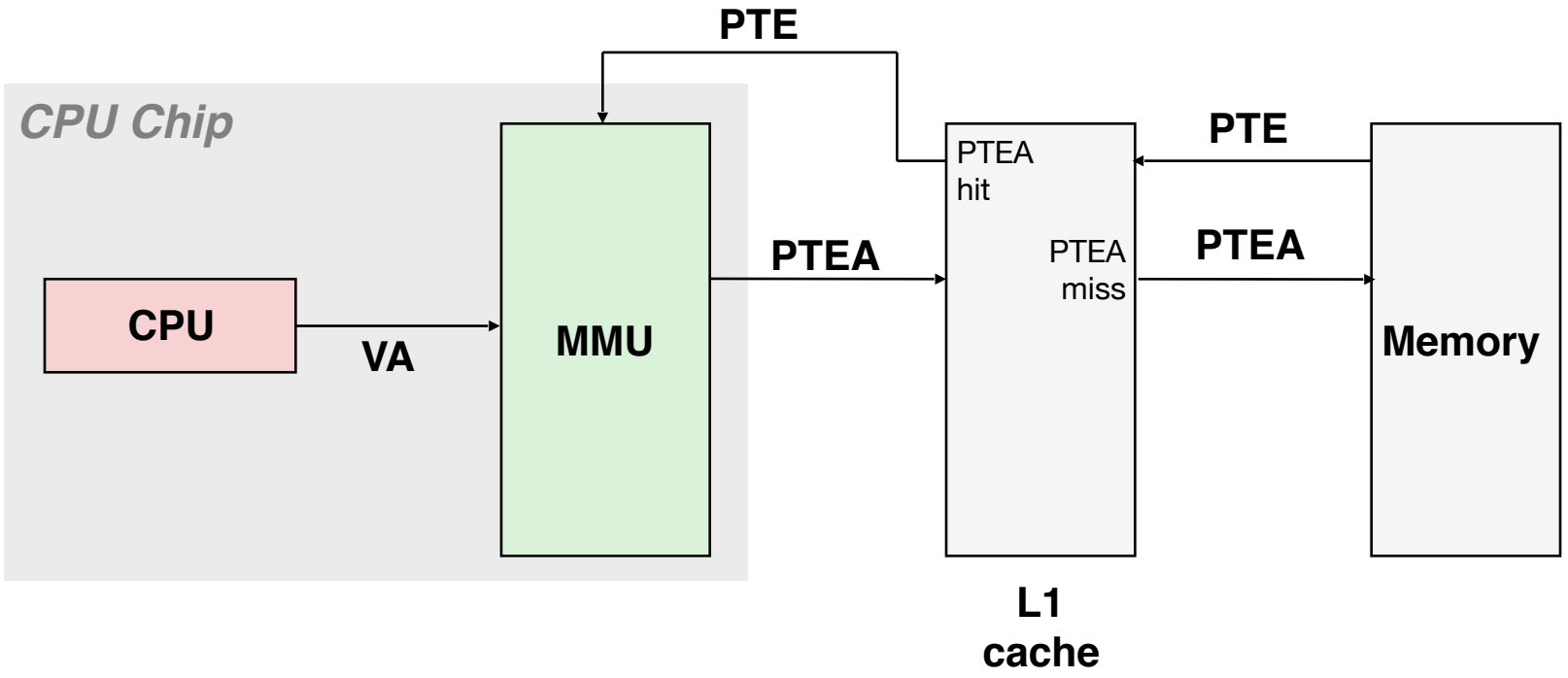
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



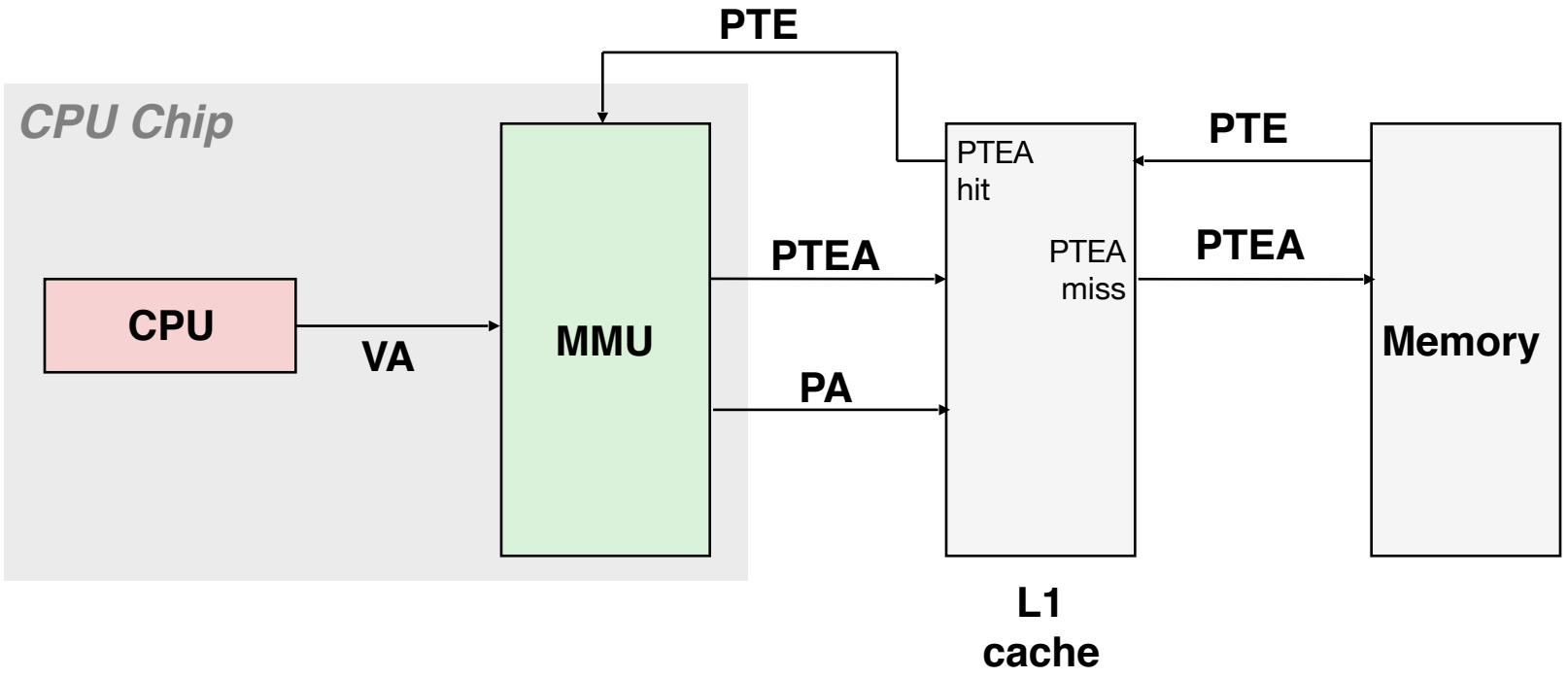
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



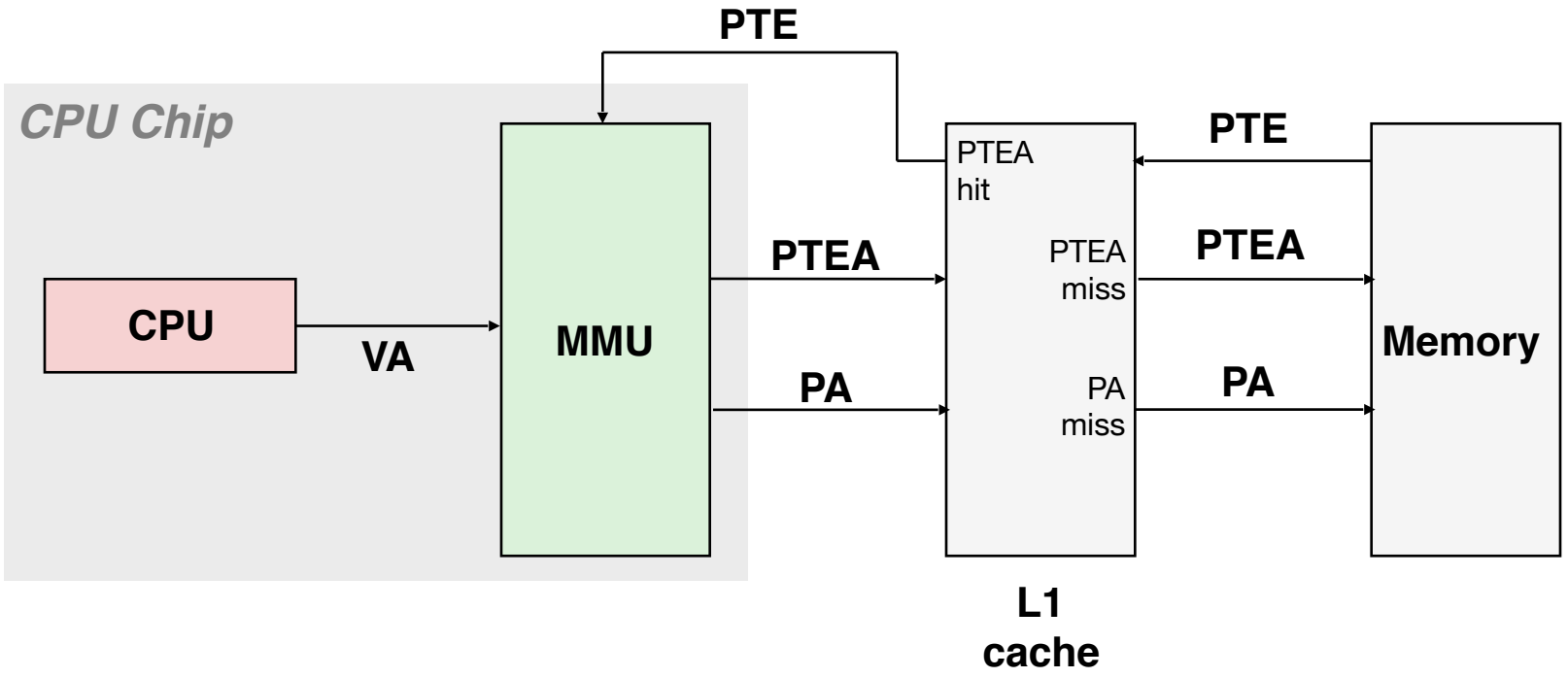
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



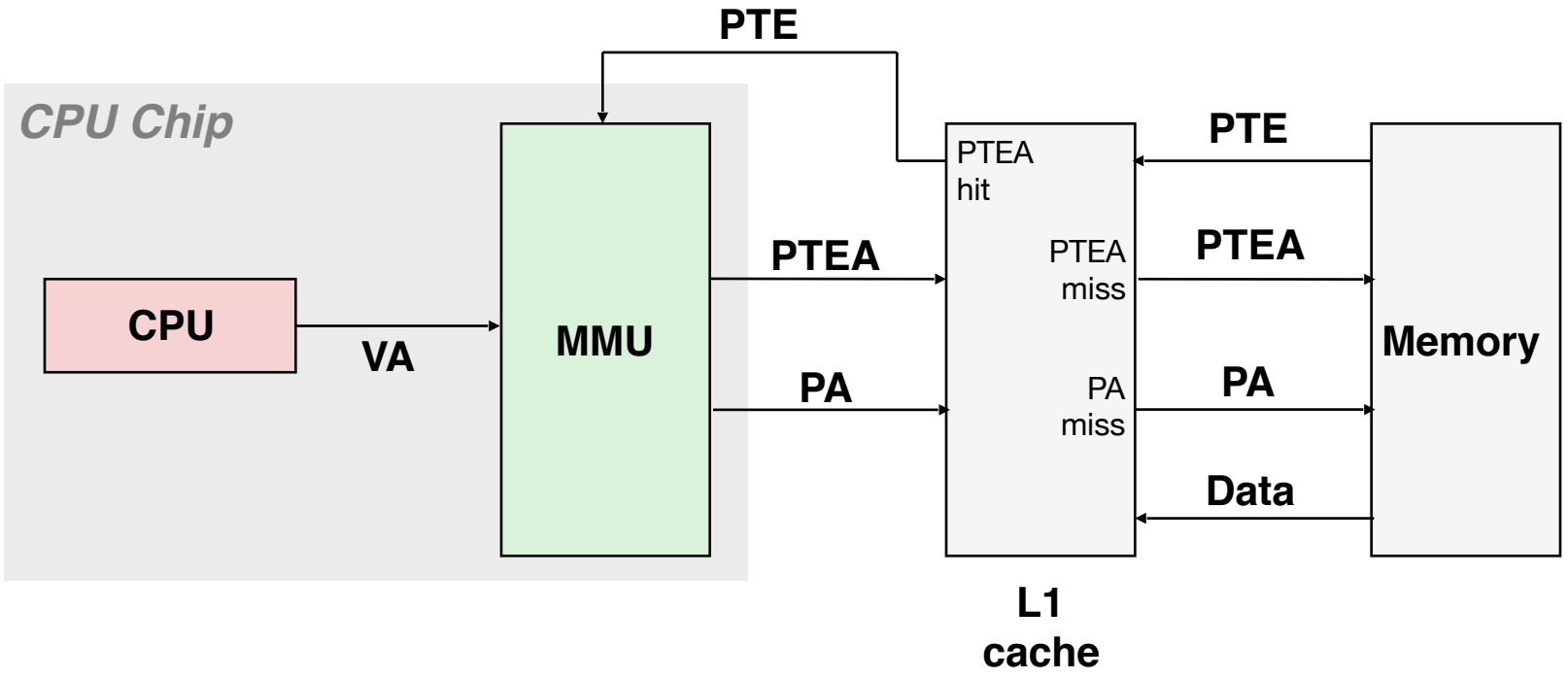
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



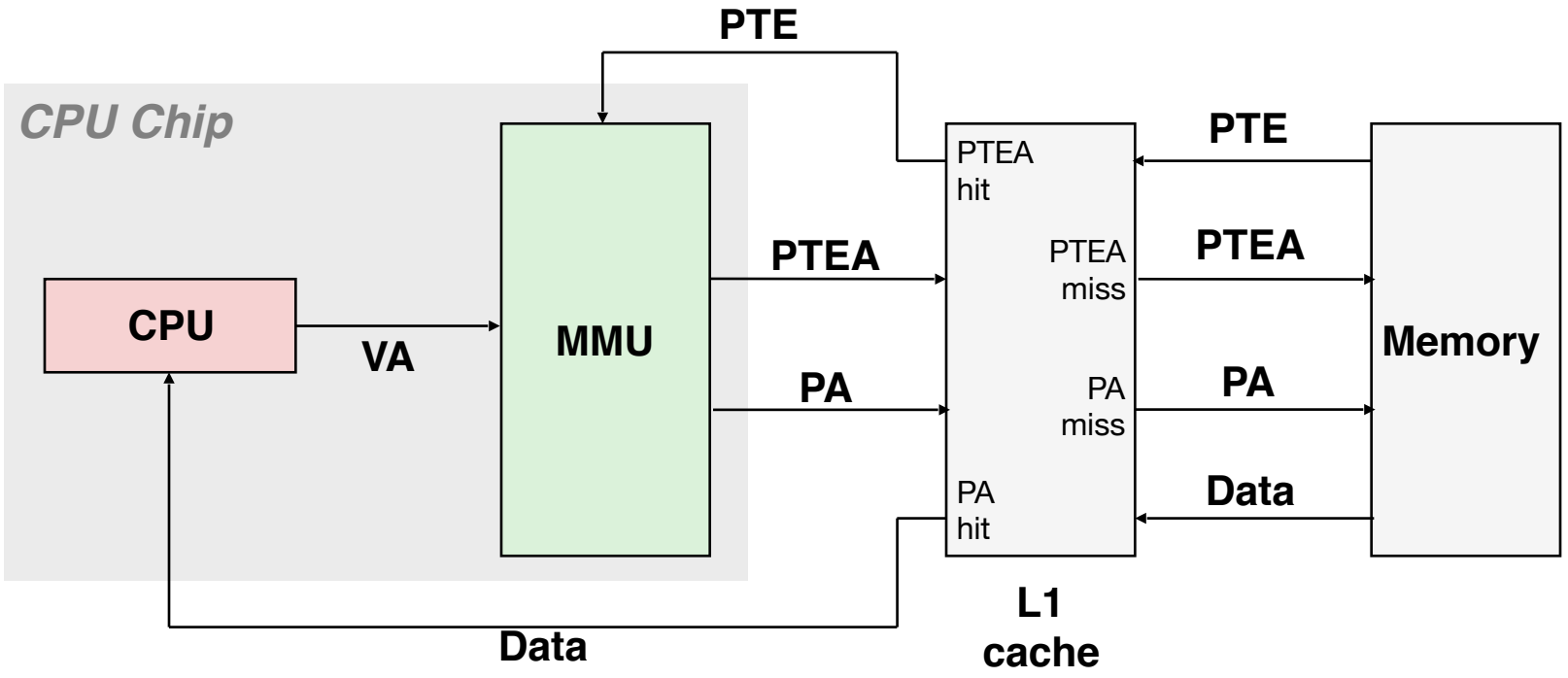
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



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

Integrating VM and Cache



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

Today

- Three Virtual Memory Optimizations
  - TLB
  - Virtually-indexed, physically-tagged cache
  - Page the page table (a.k.a., multi-level page table)
- Case-study: Intel Core i7/Linux example

Speeding up Address Translation

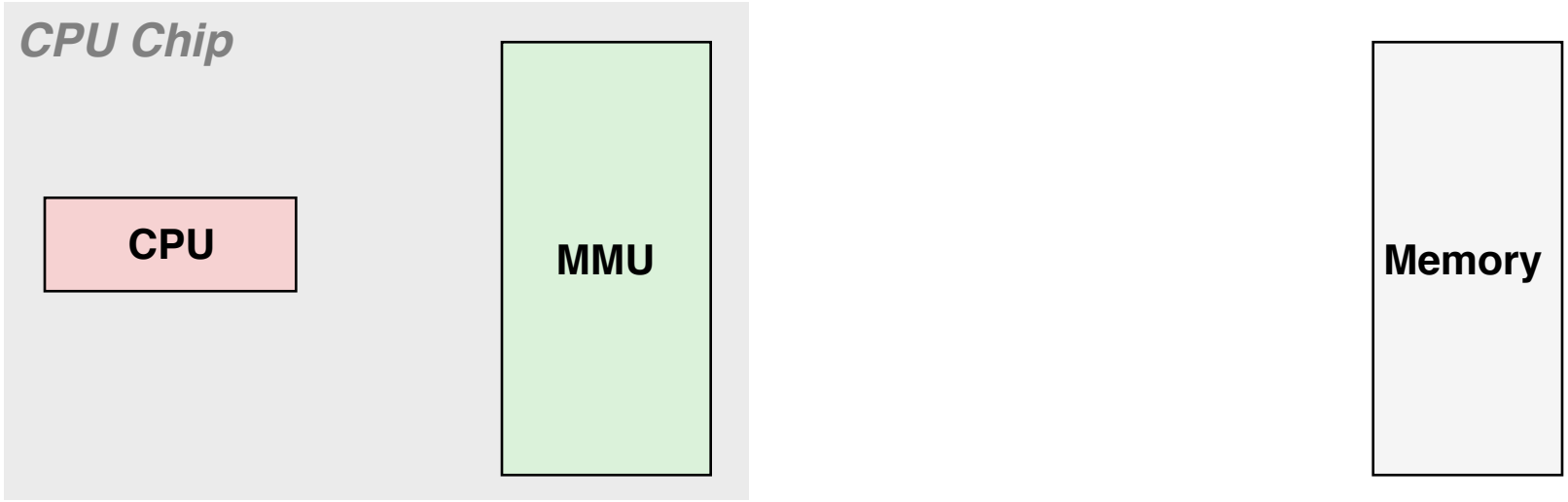
Speeding up Address Translation

- Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  - The PTE access is kind of an overhead
  - Can we speed it up?

Speeding up Address Translation

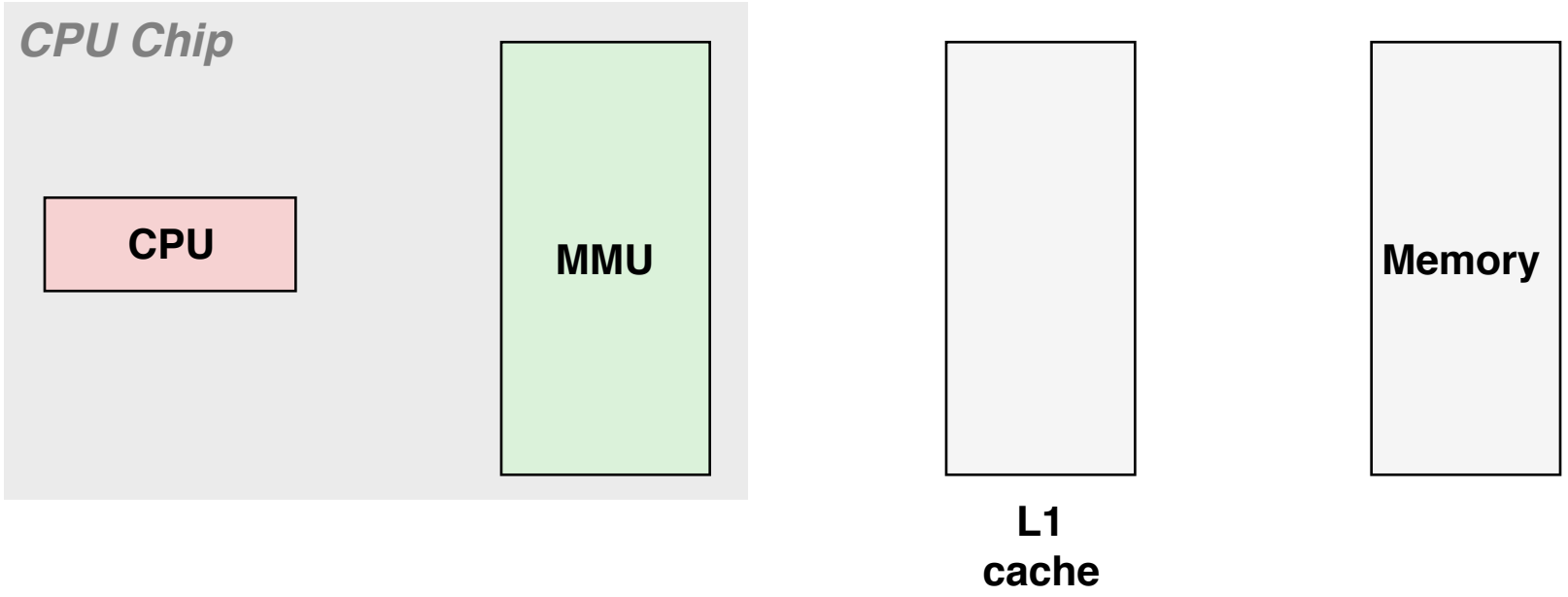
- Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  - The PTE access is kind of an overhead
  - Can we speed it up?
- Page table entries (PTEs) are already cached in L1 data cache like any other memory data. But:
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

Recall: Page Table is Cached



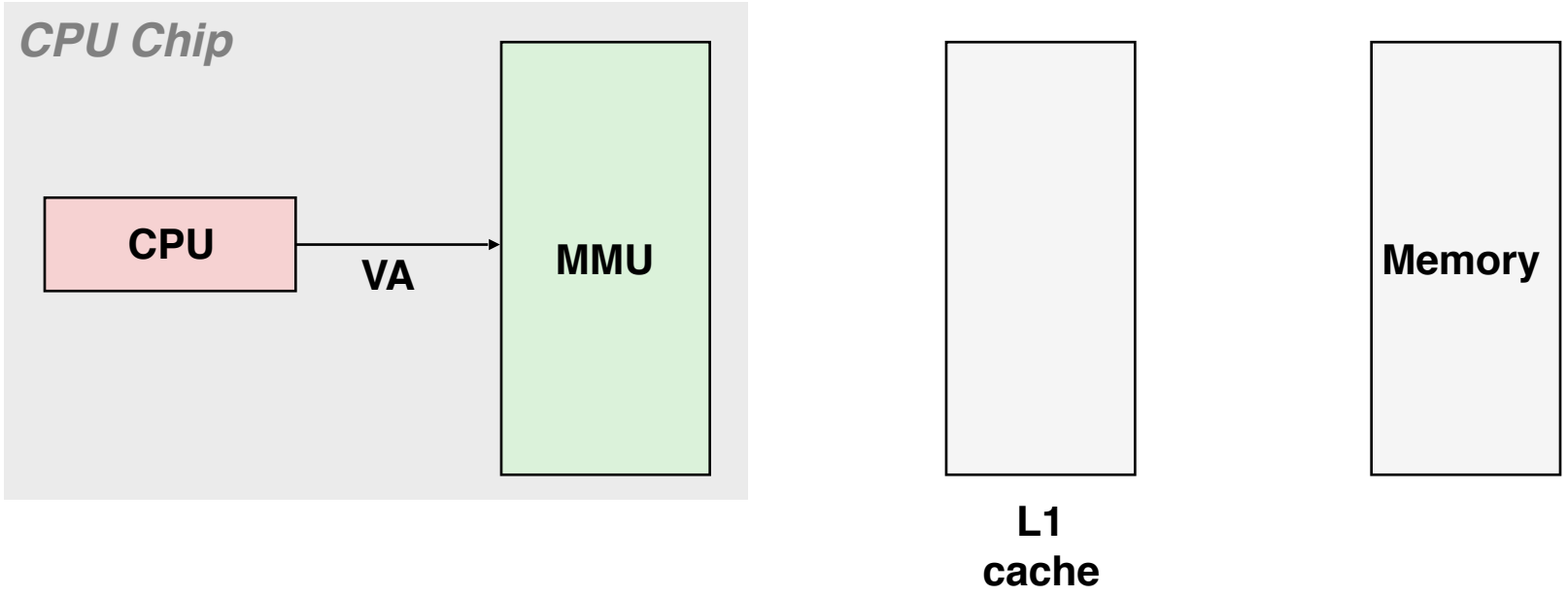
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



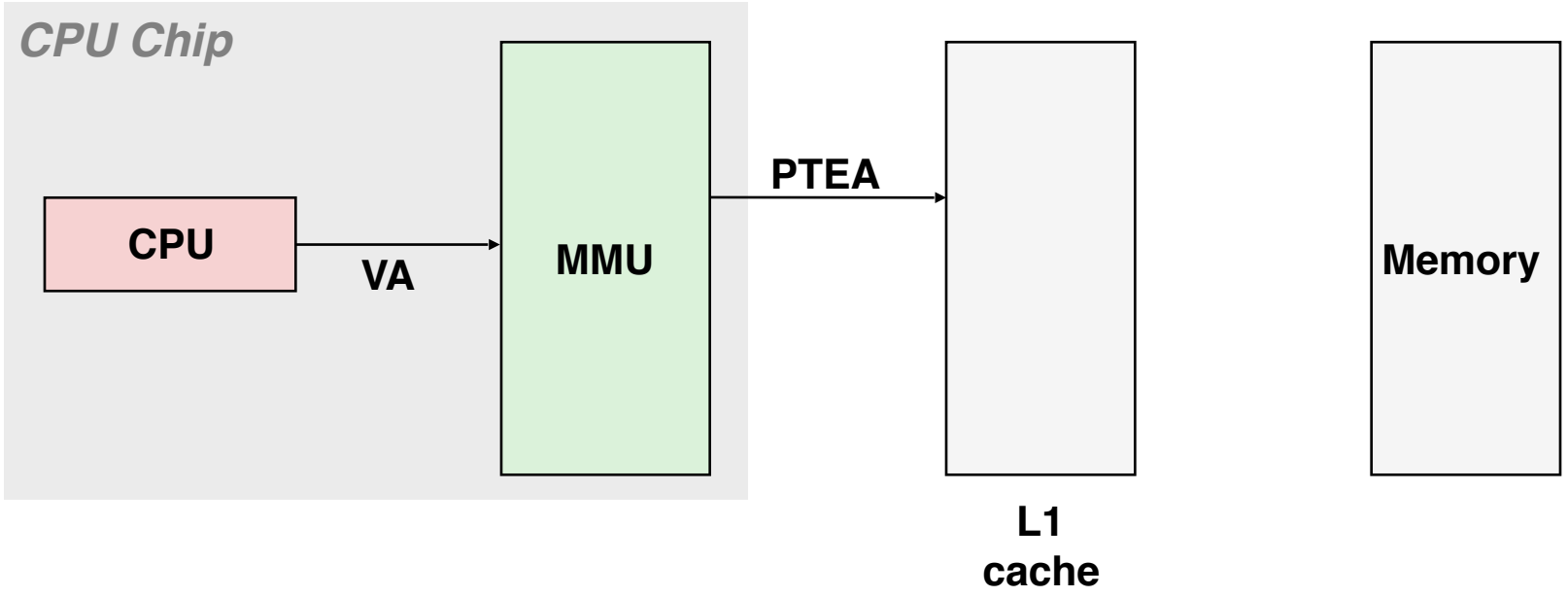
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



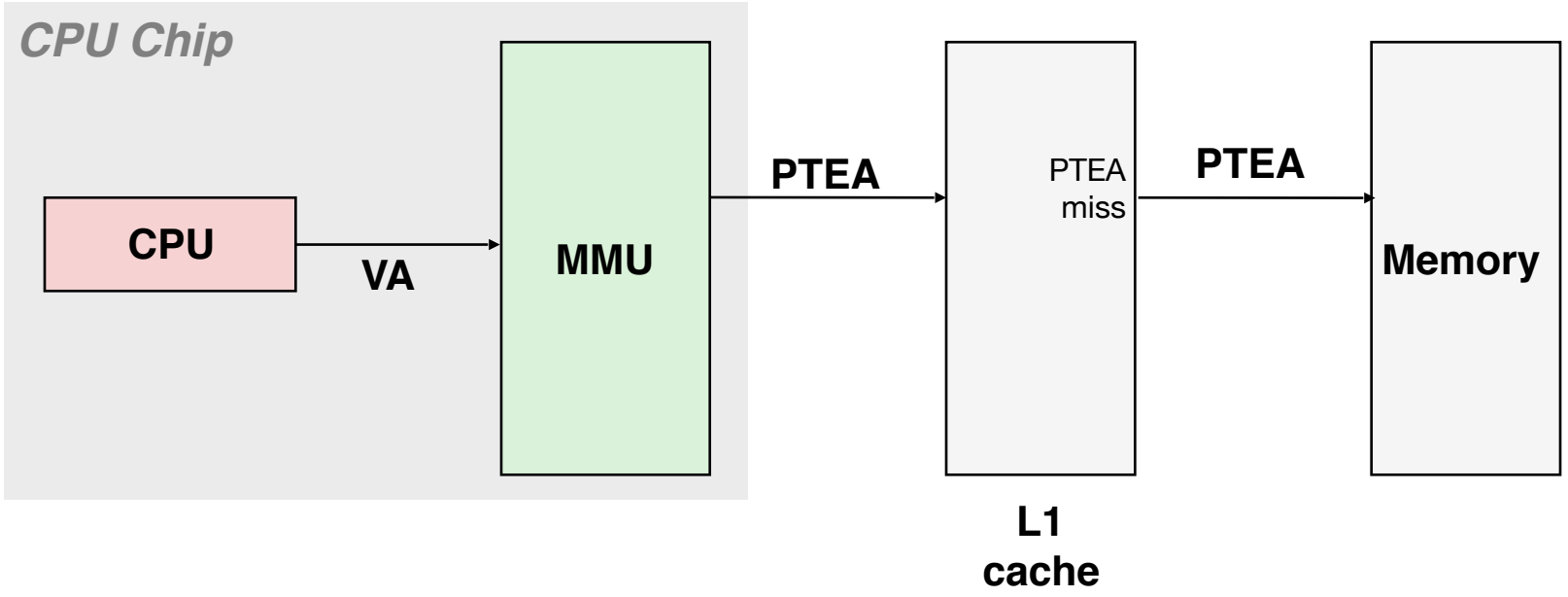
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



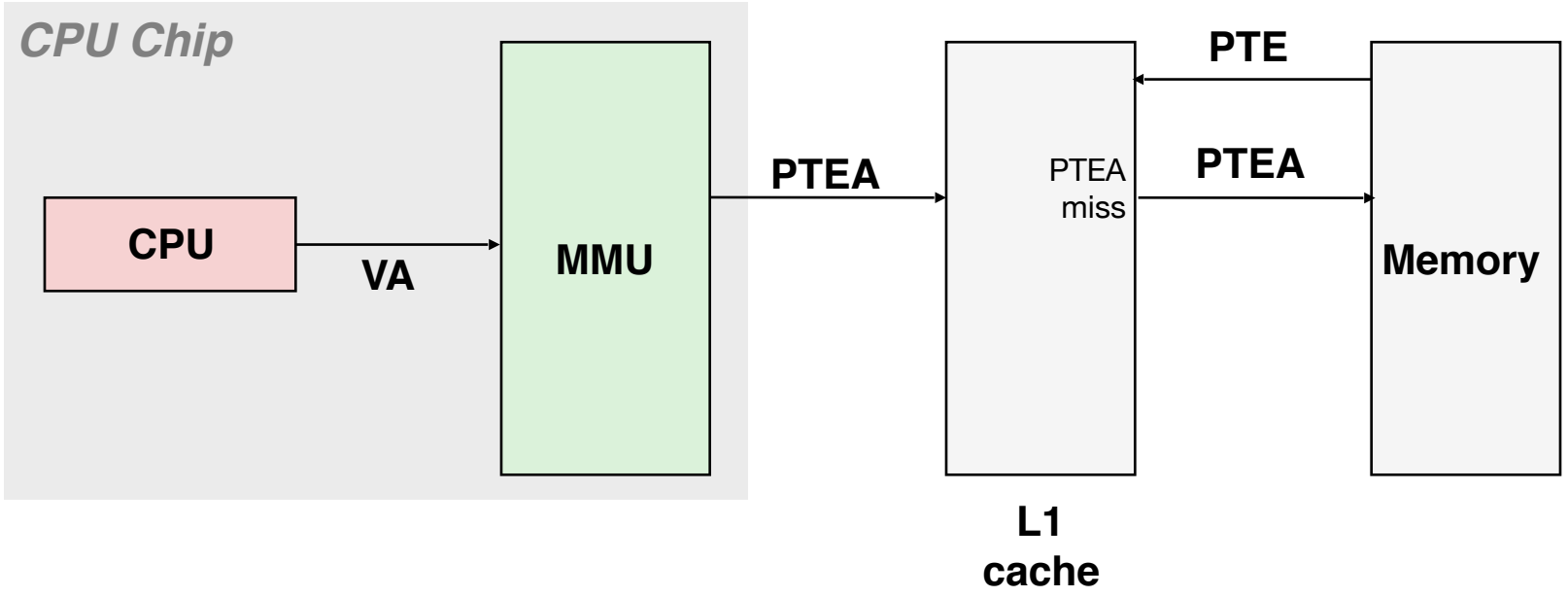
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



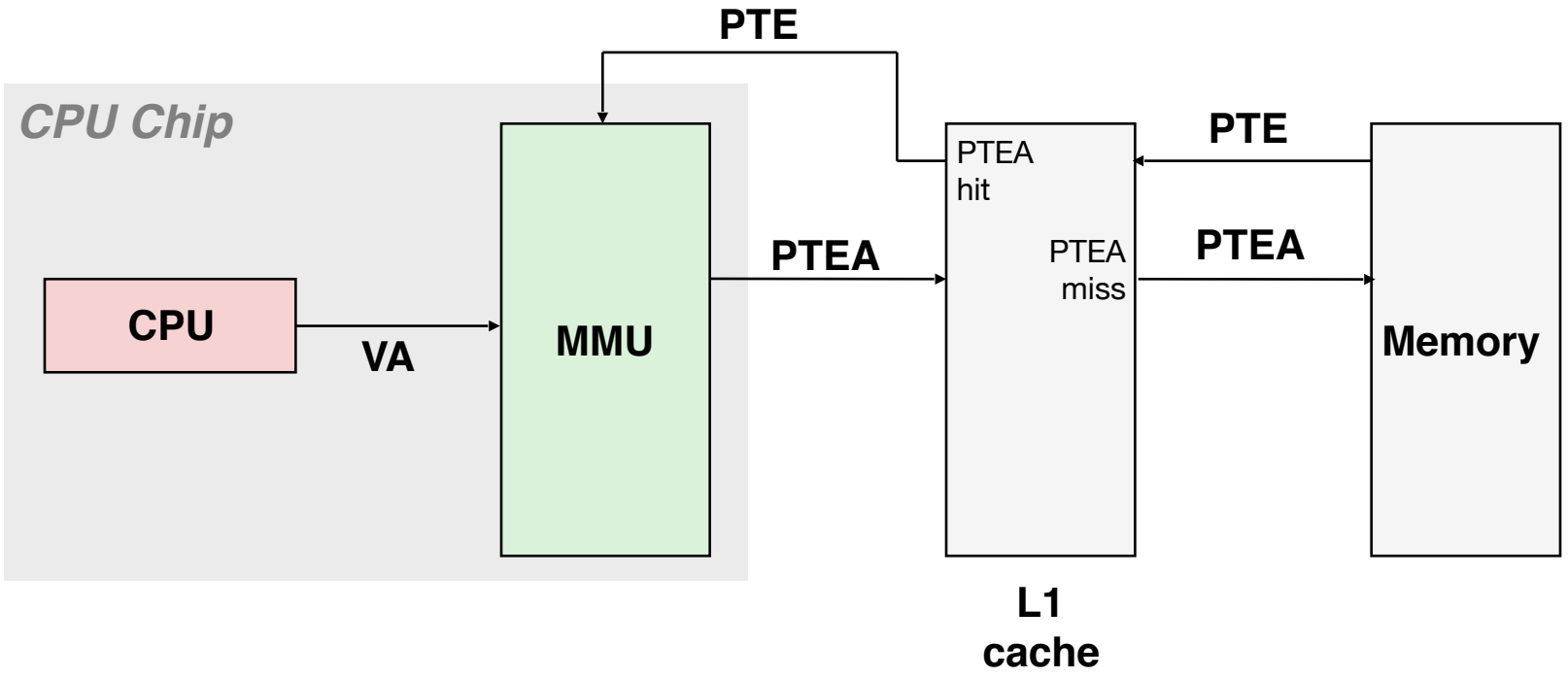
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



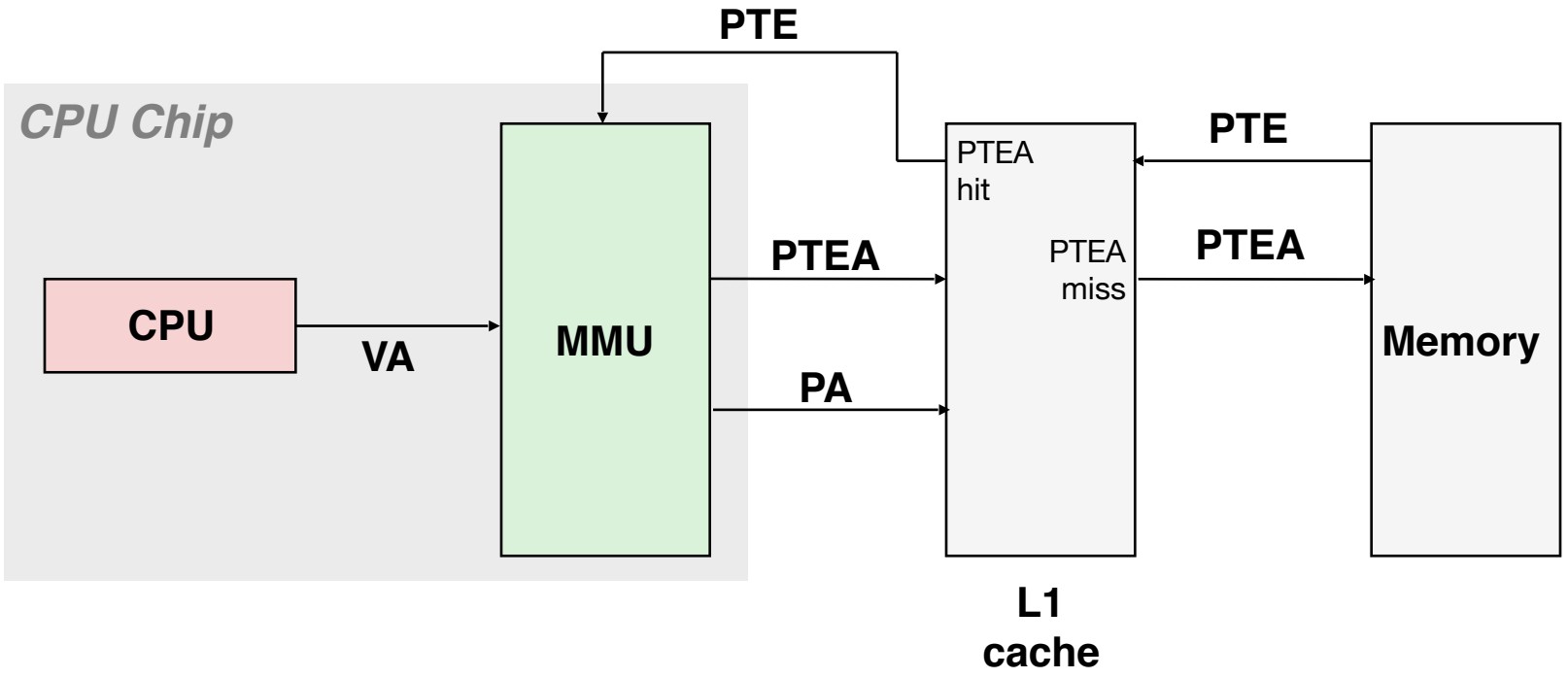
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



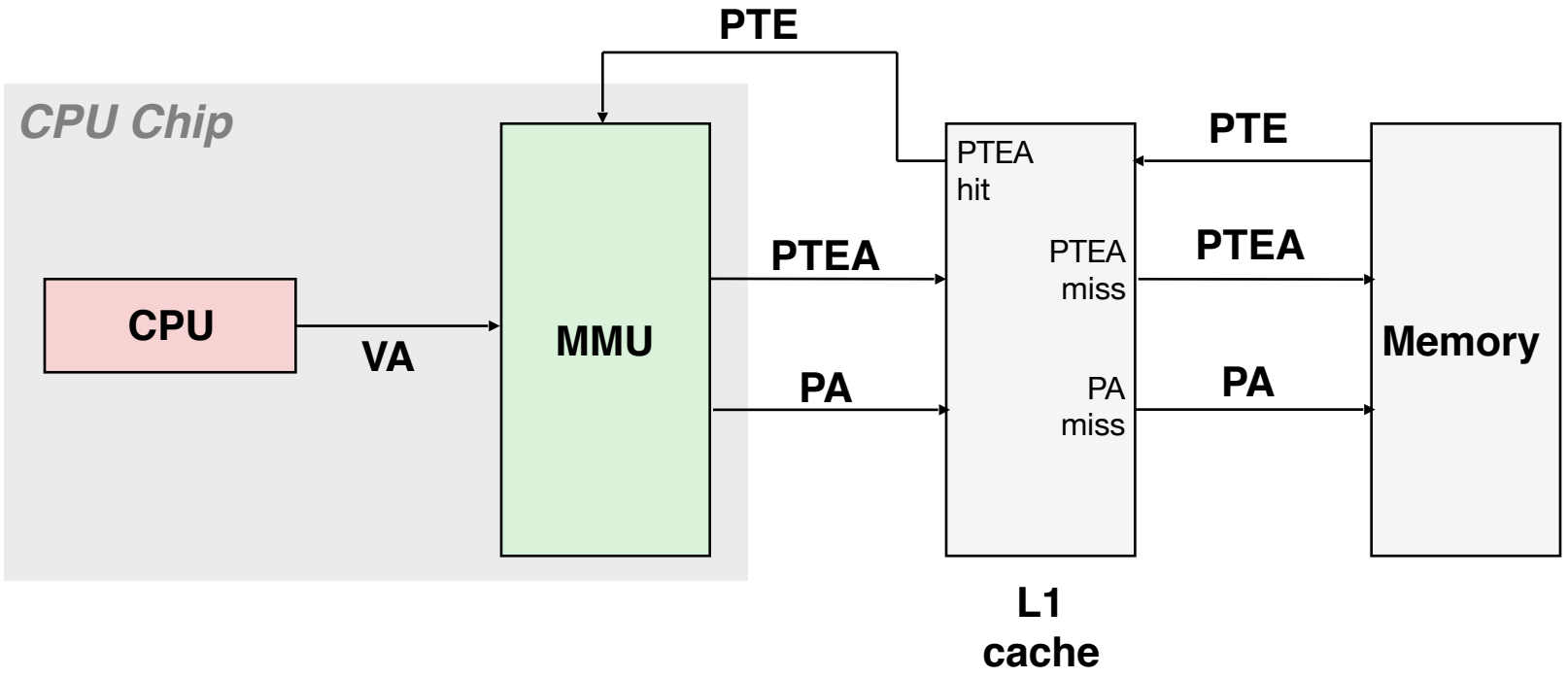
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



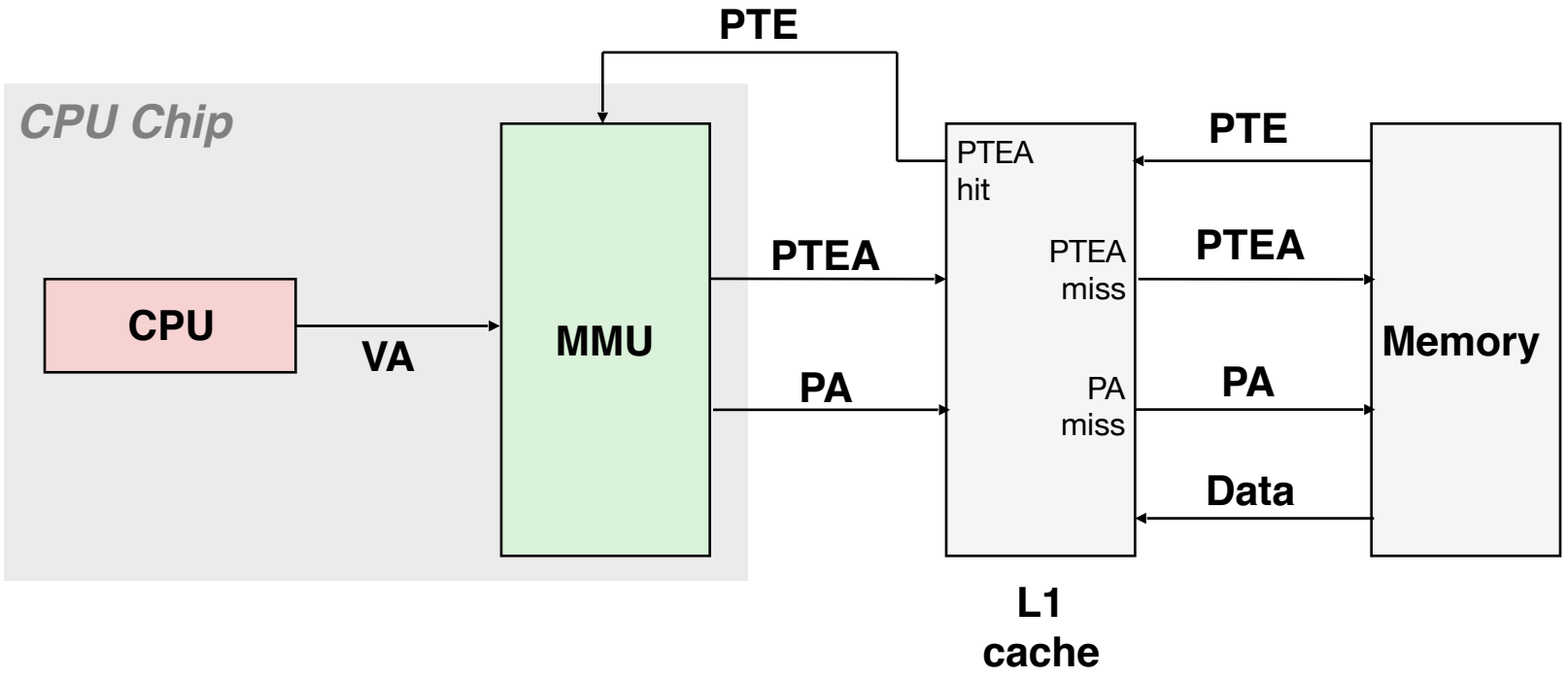
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



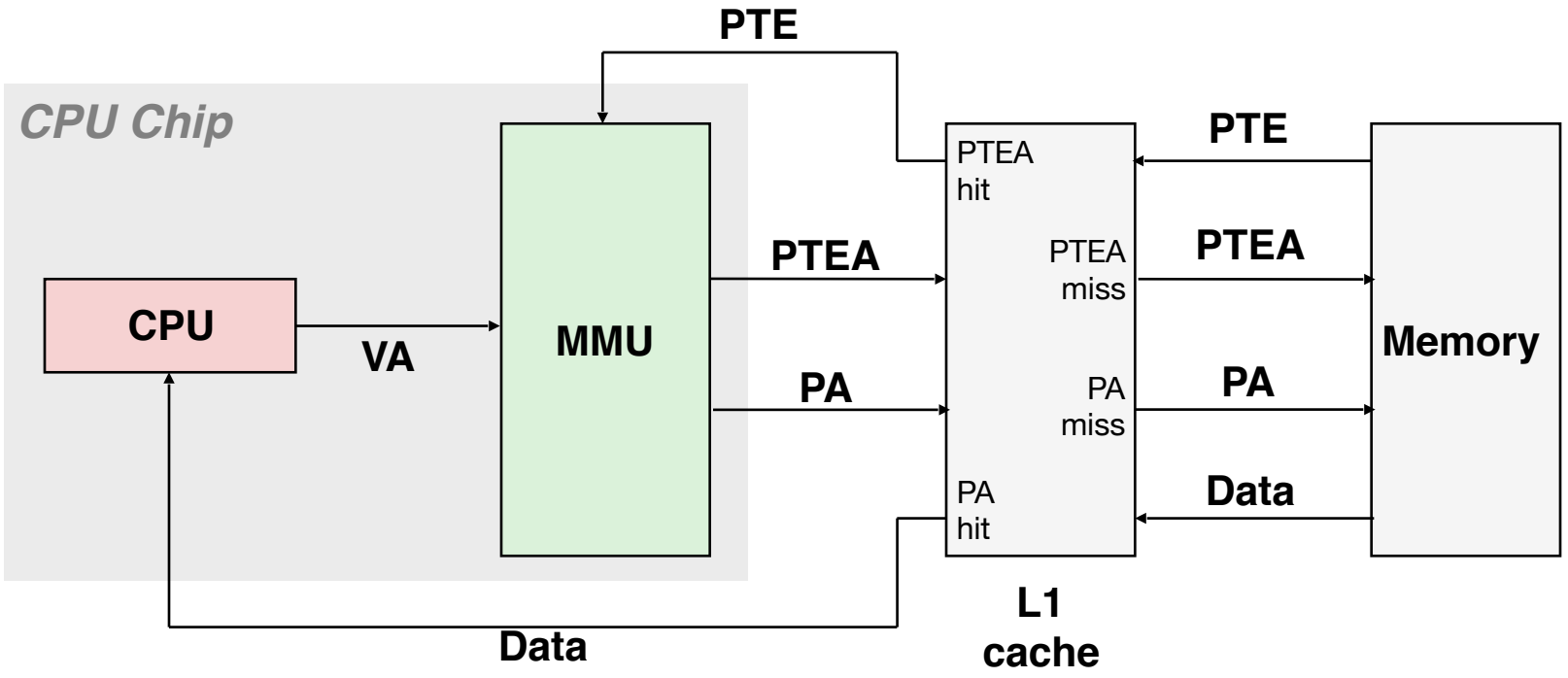
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Recall: Page Table is Cached



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

Recall: Page Table is Cached



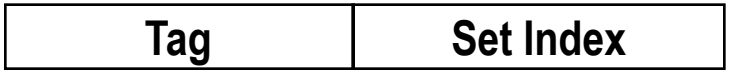
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages

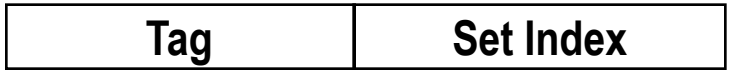
Speeding up Translation with a TLB

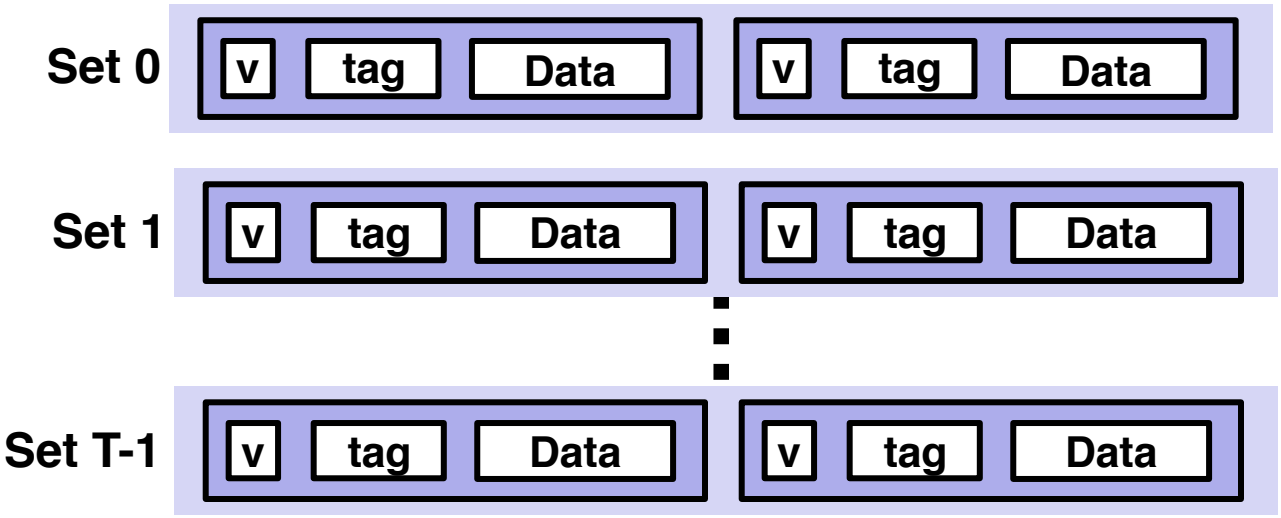
- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages

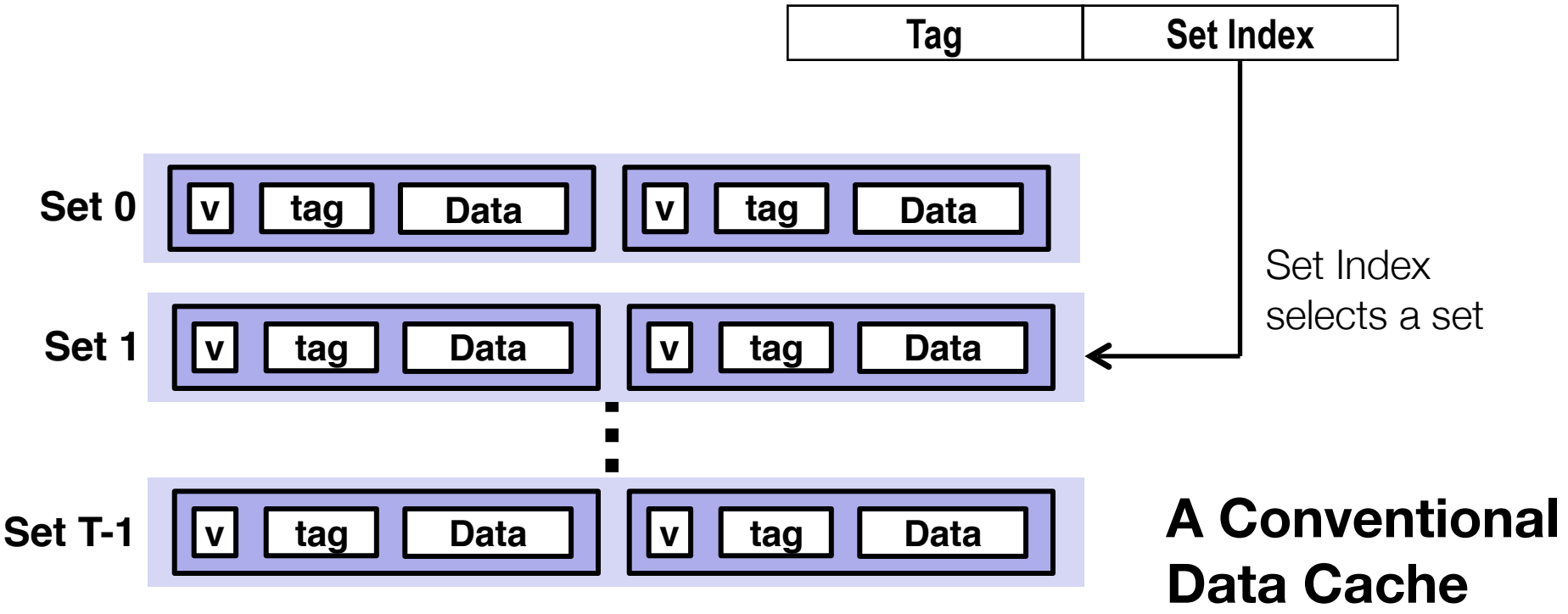




A Conventional Data Cache

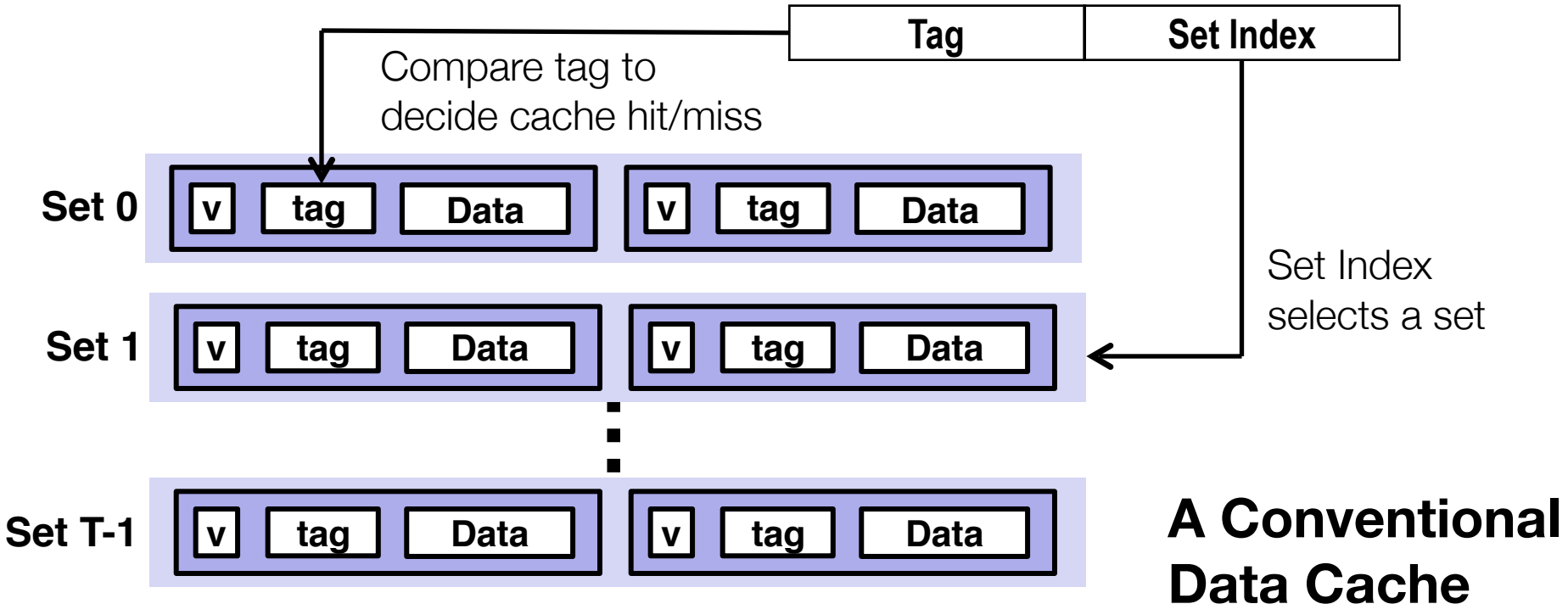
Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



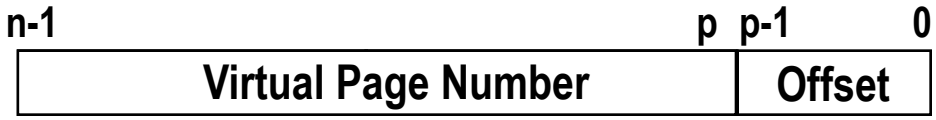
Speeding up Translation with a TLB

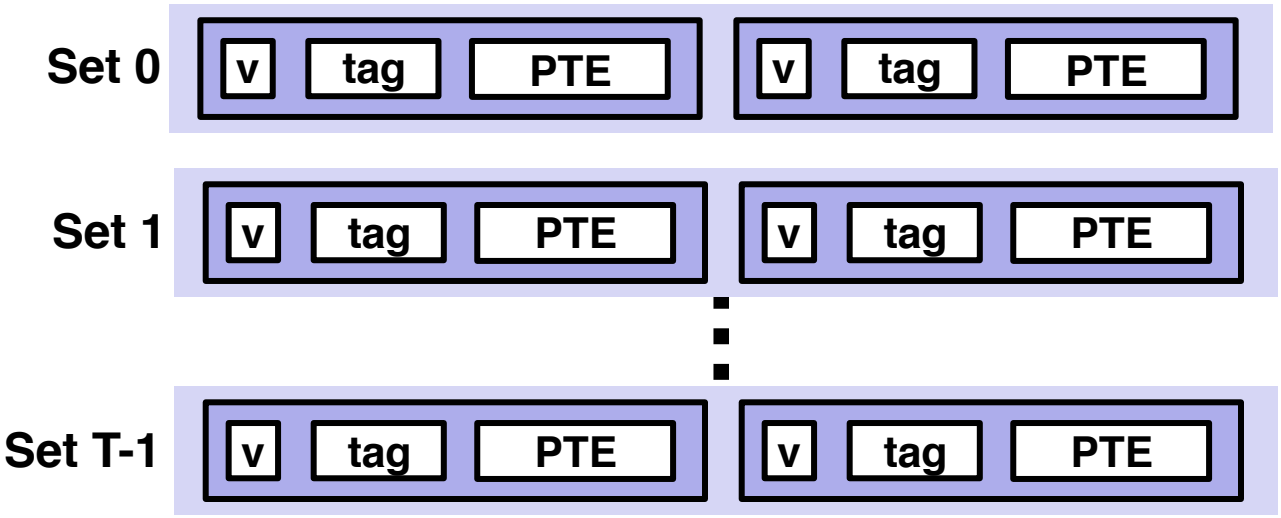
- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

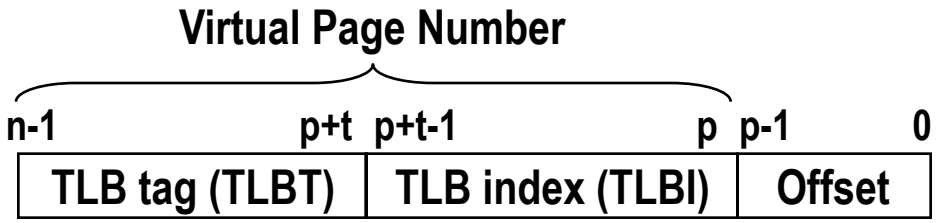


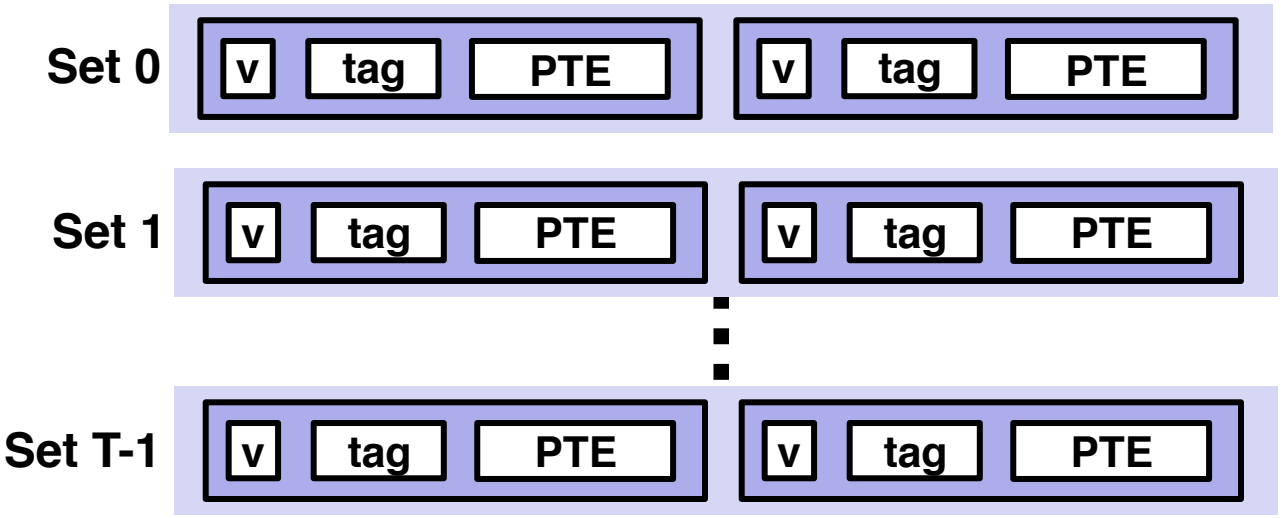


A Page Table Cache

Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

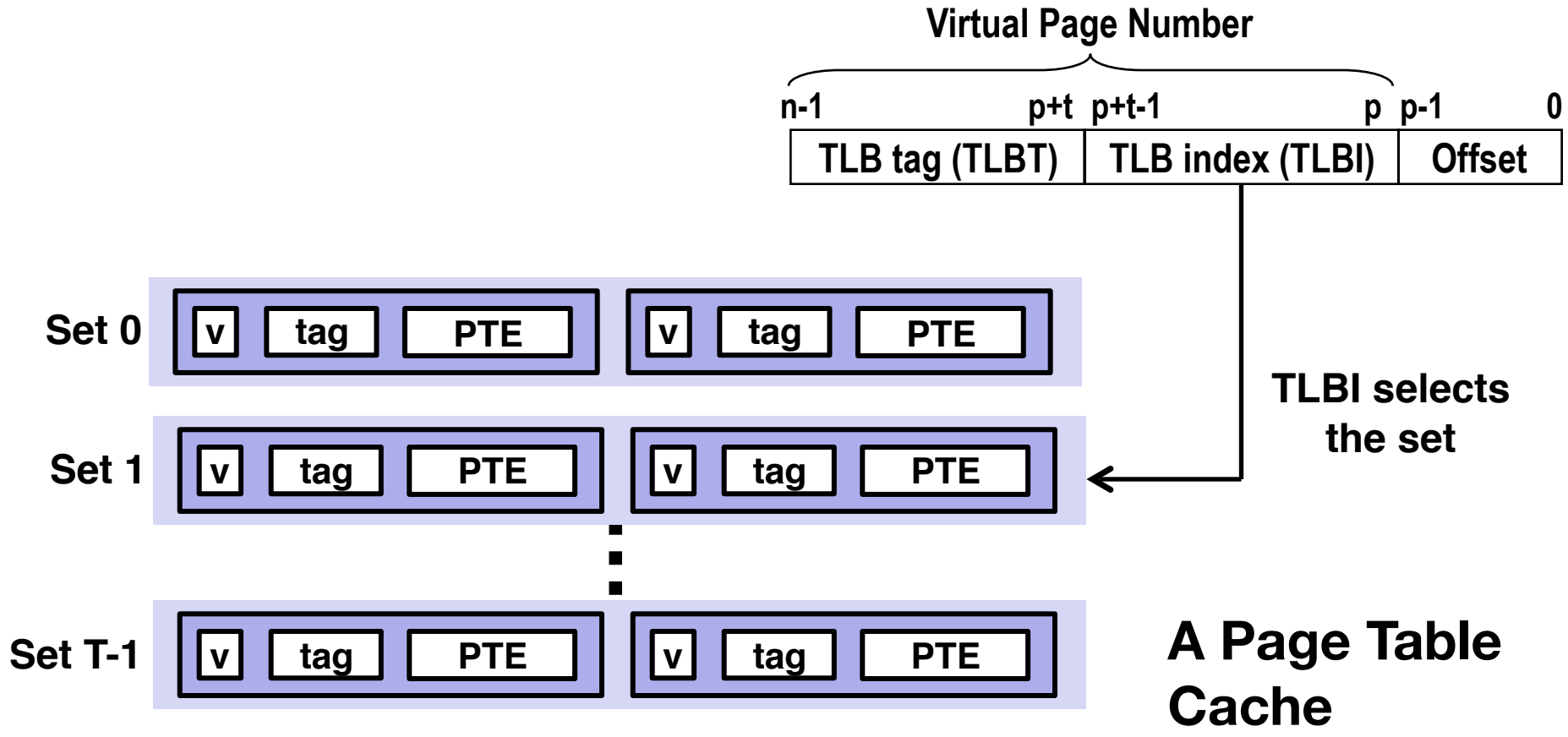




A Page Table Cache

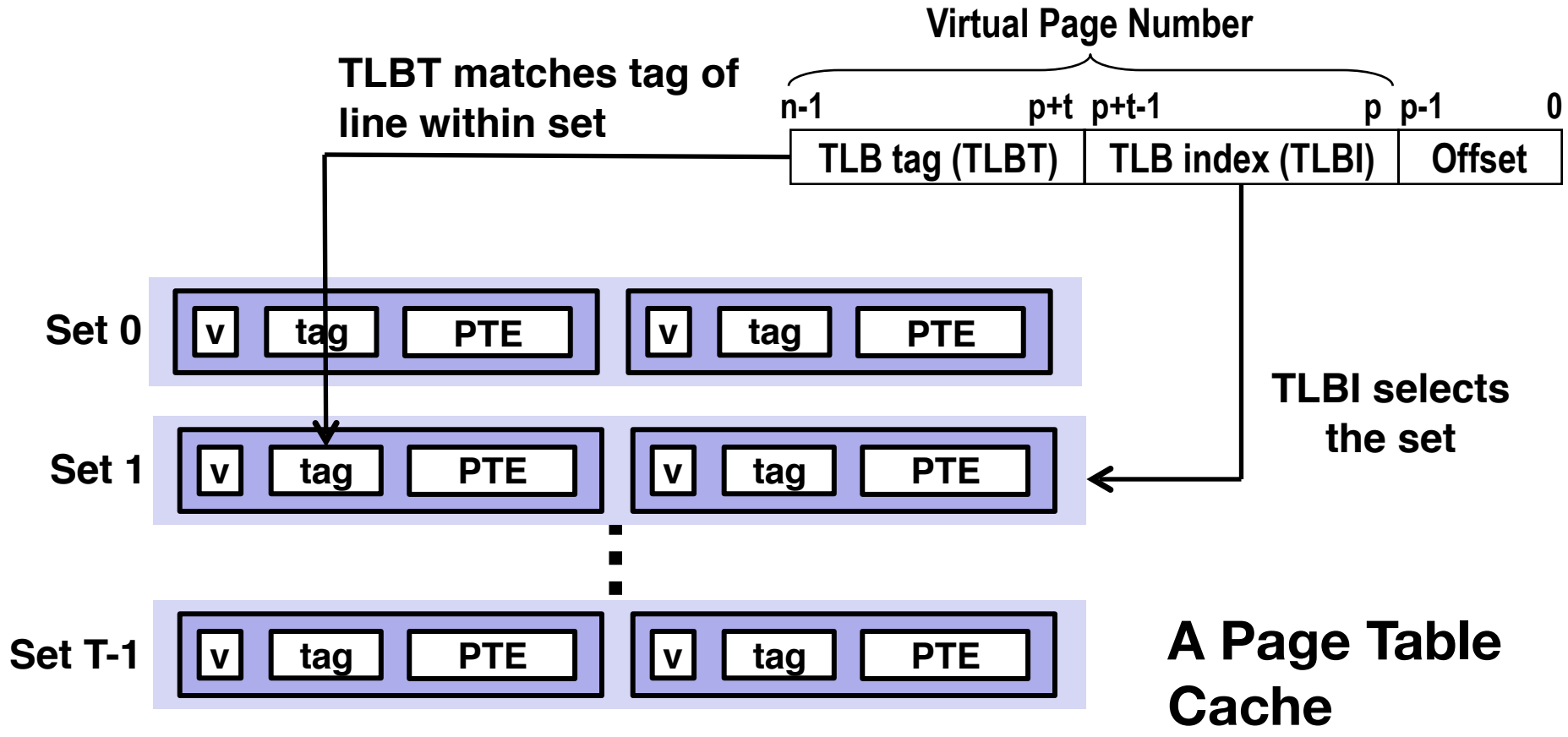
Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

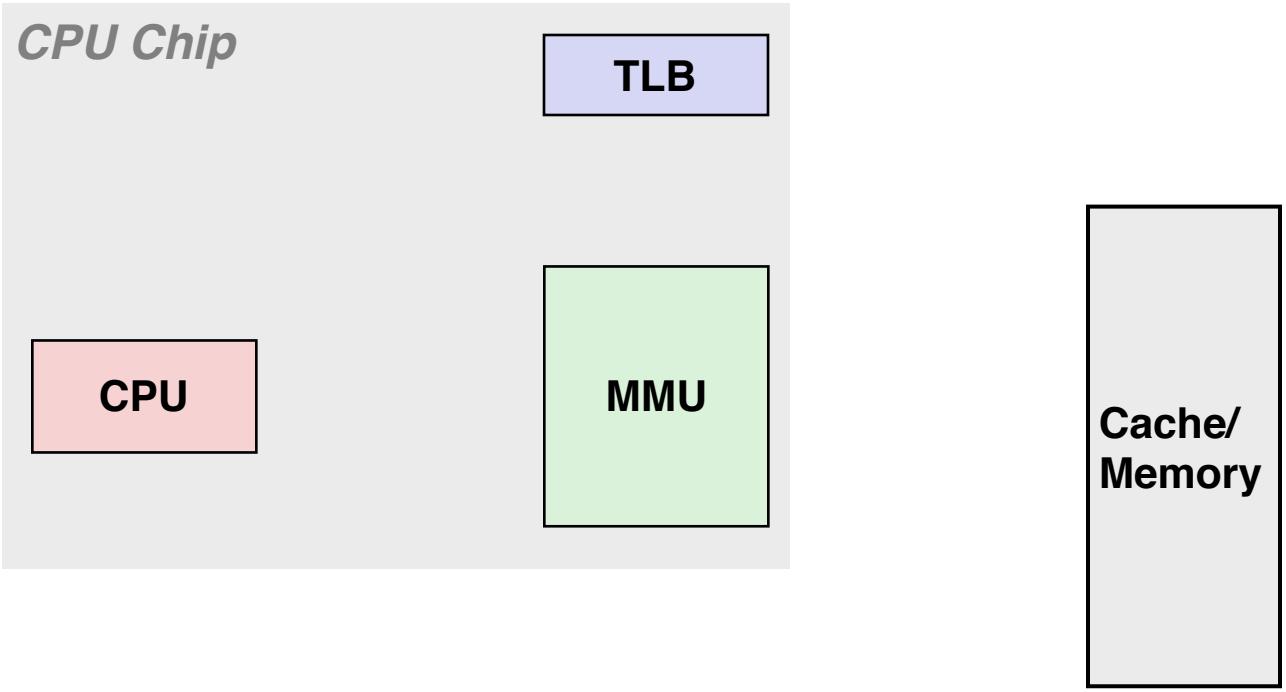


Accessing the TLB

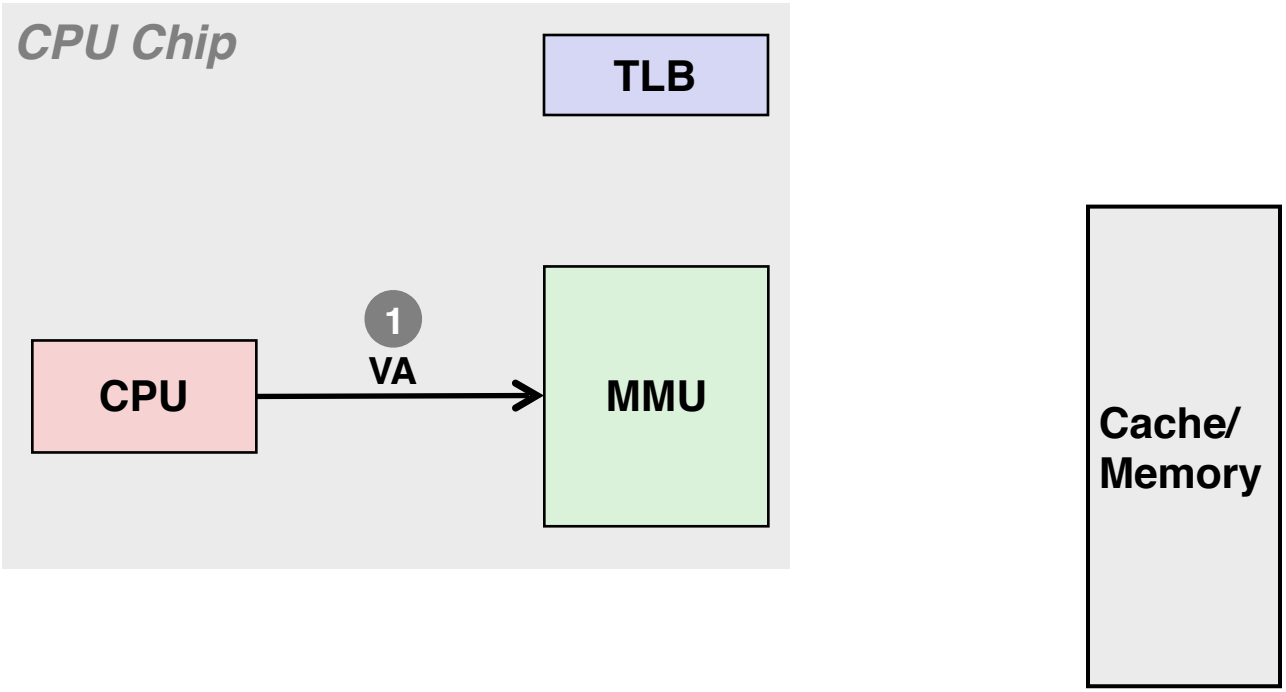
- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:



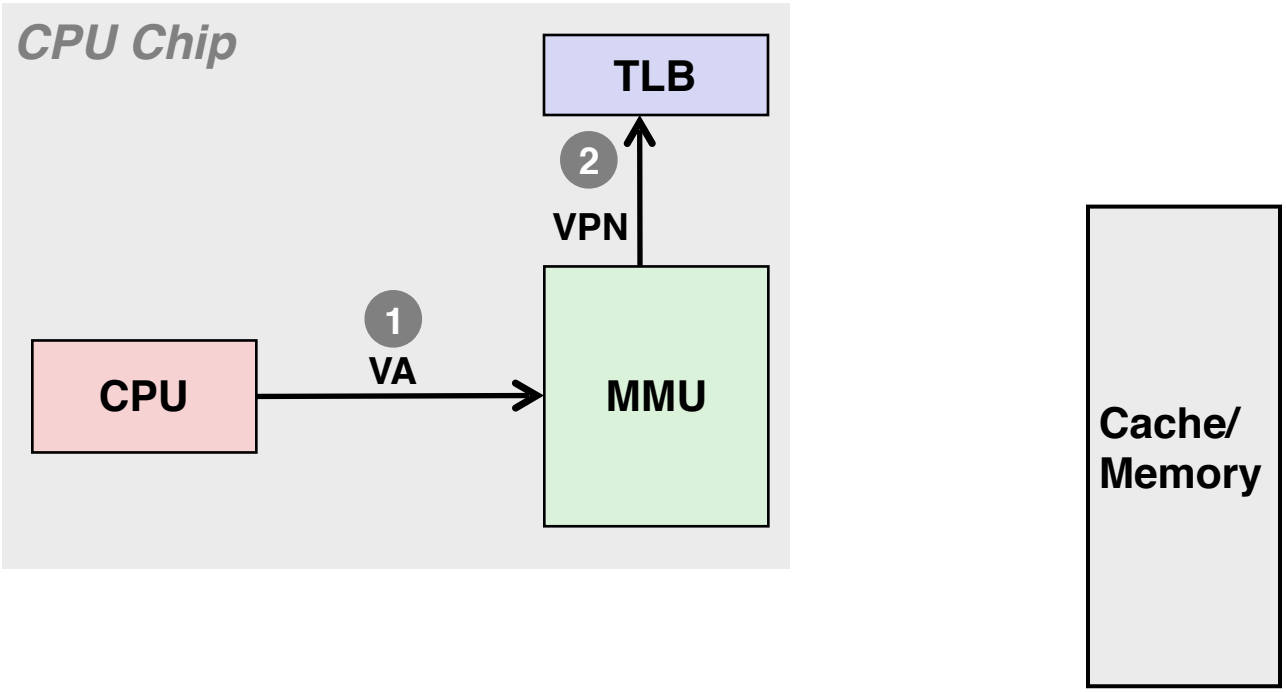
TLB Hit



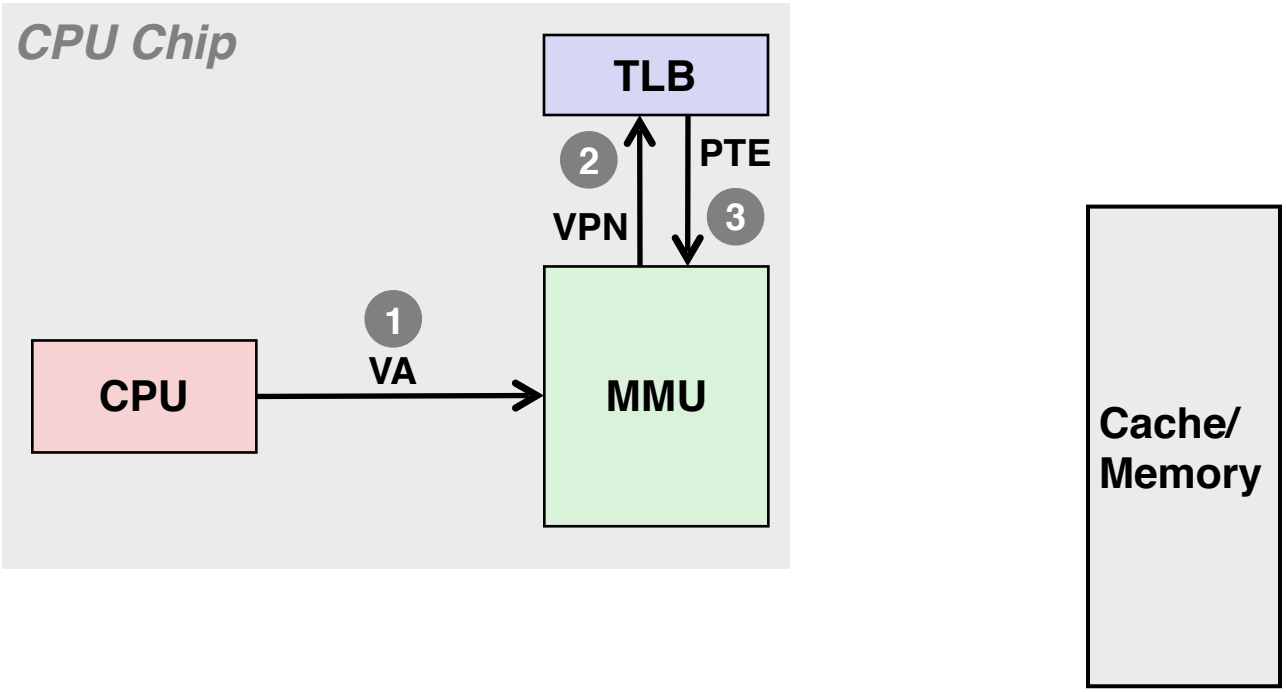
TLB Hit



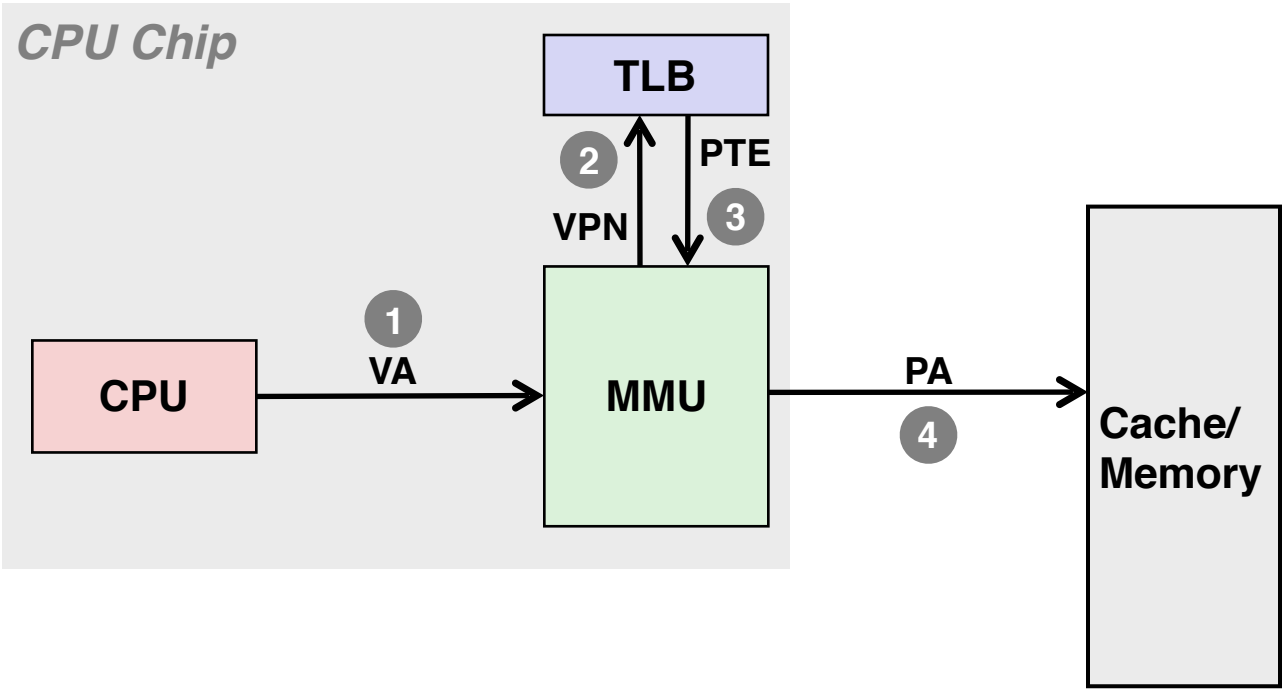
TLB Hit



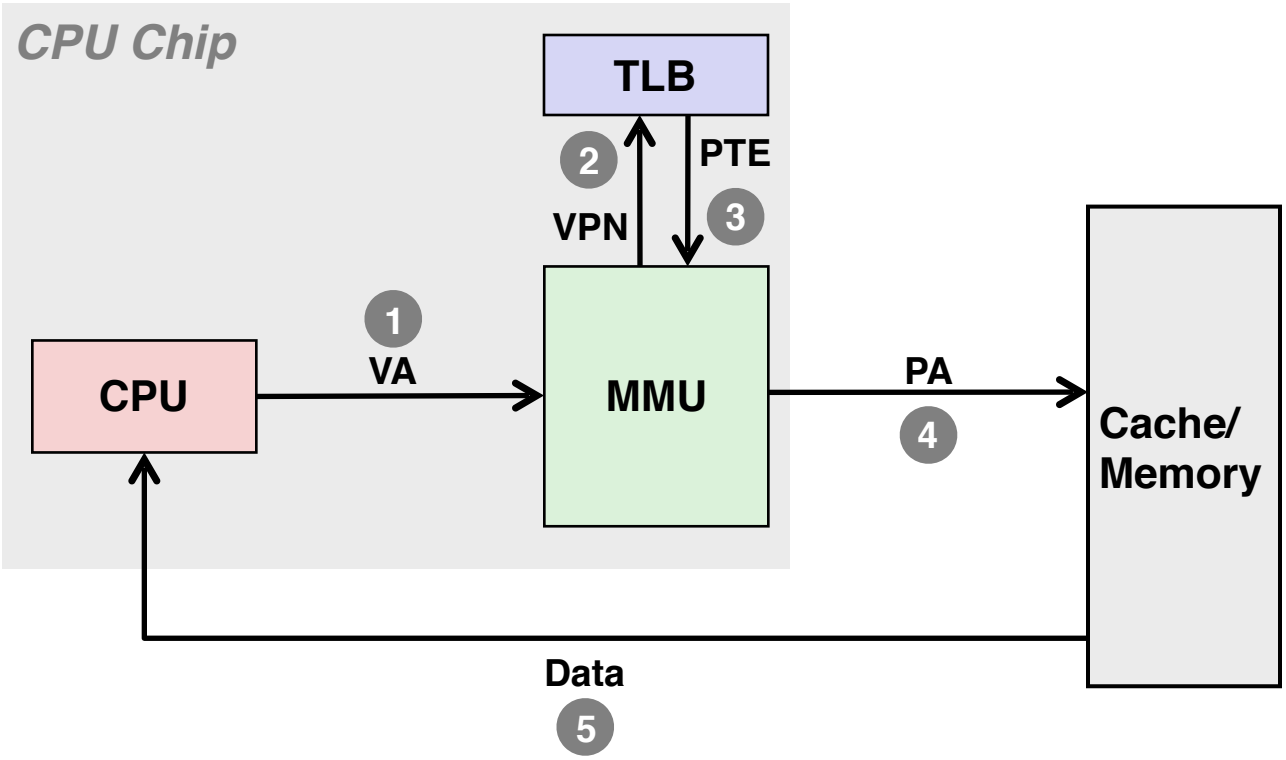
TLB Hit



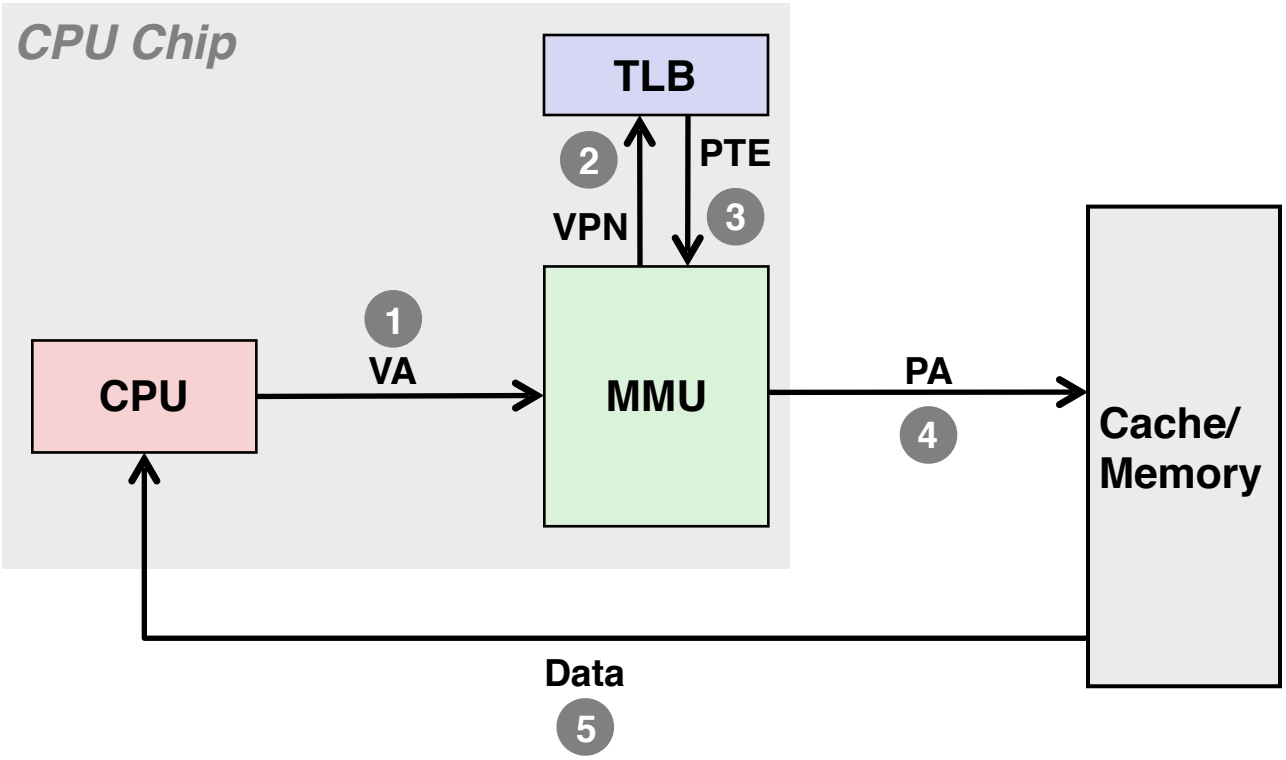
TLB Hit



TLB Hit

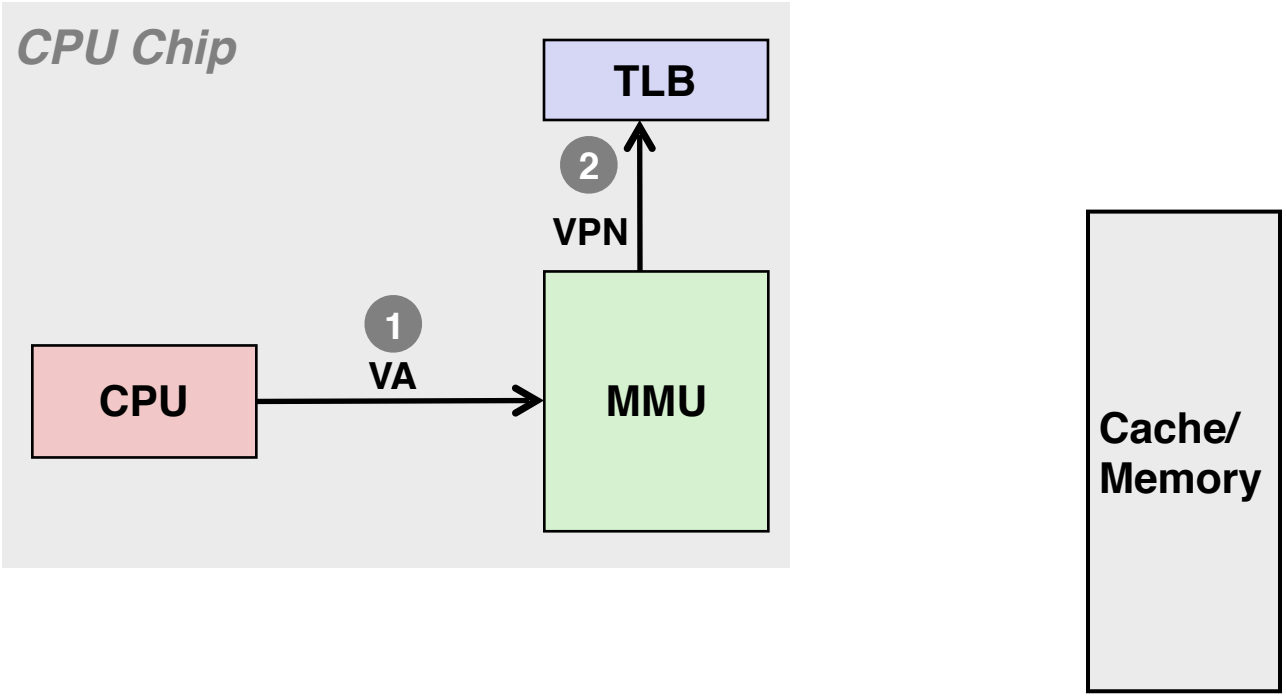


TLB Hit

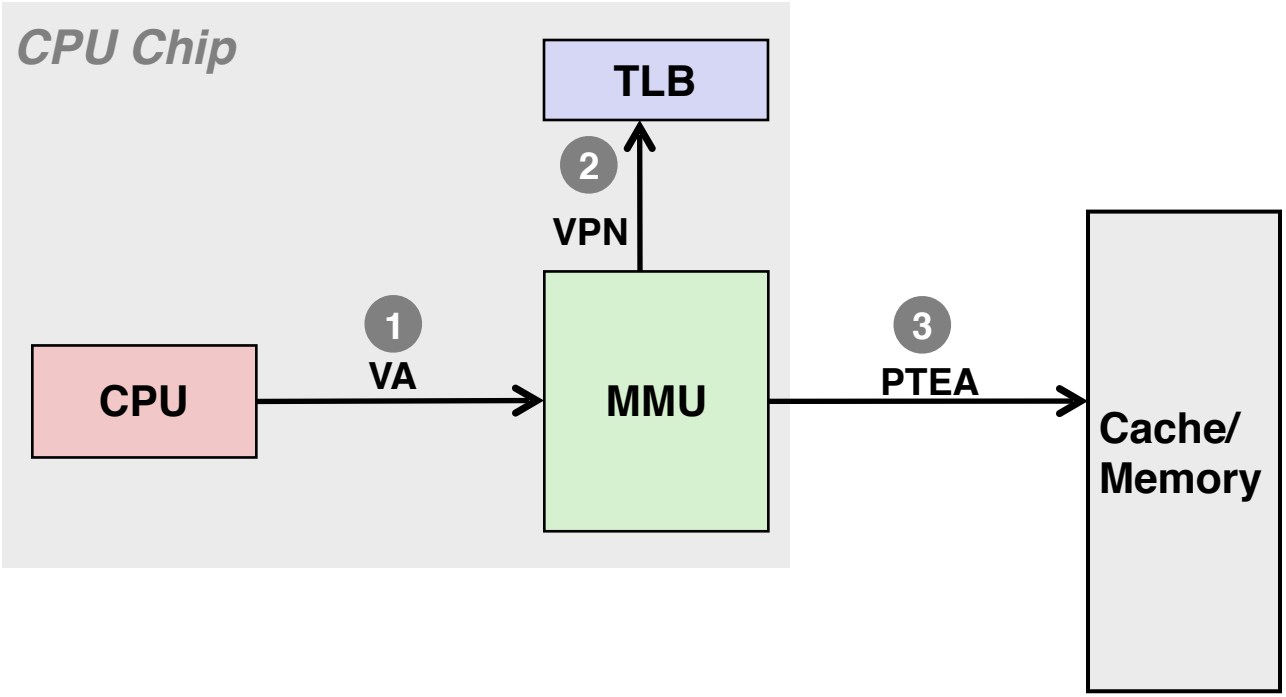


A TLB hit eliminates a memory access

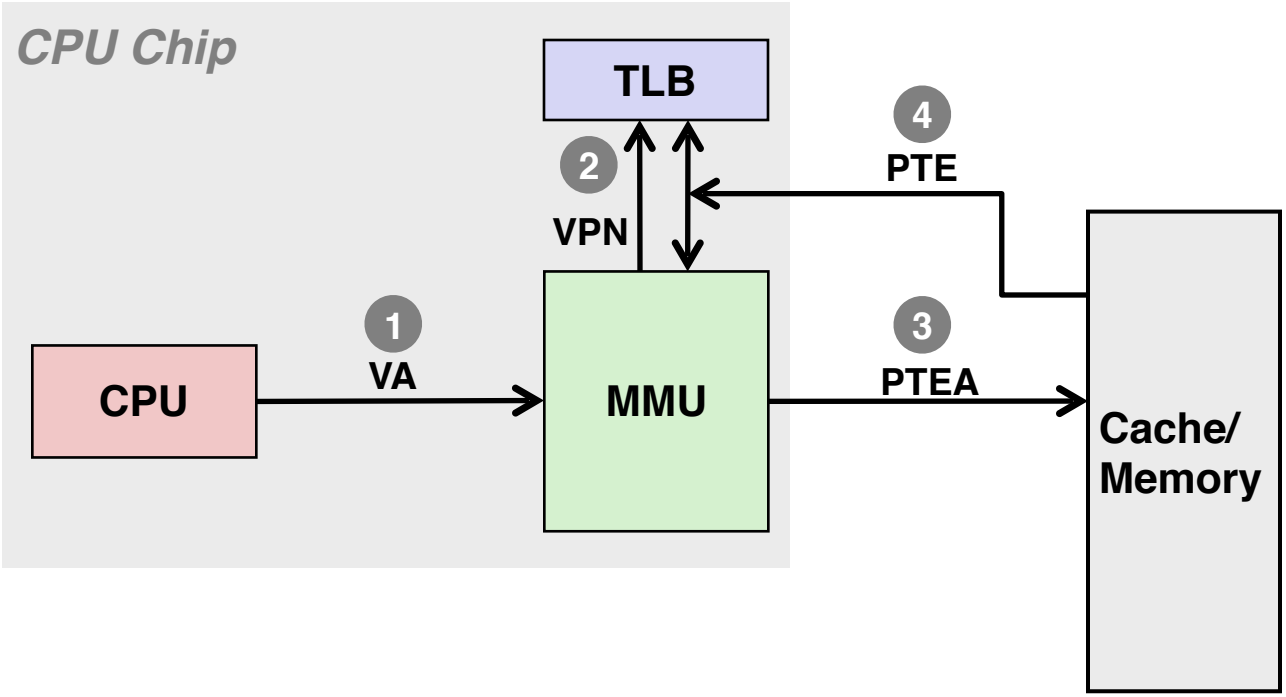
TLB Miss



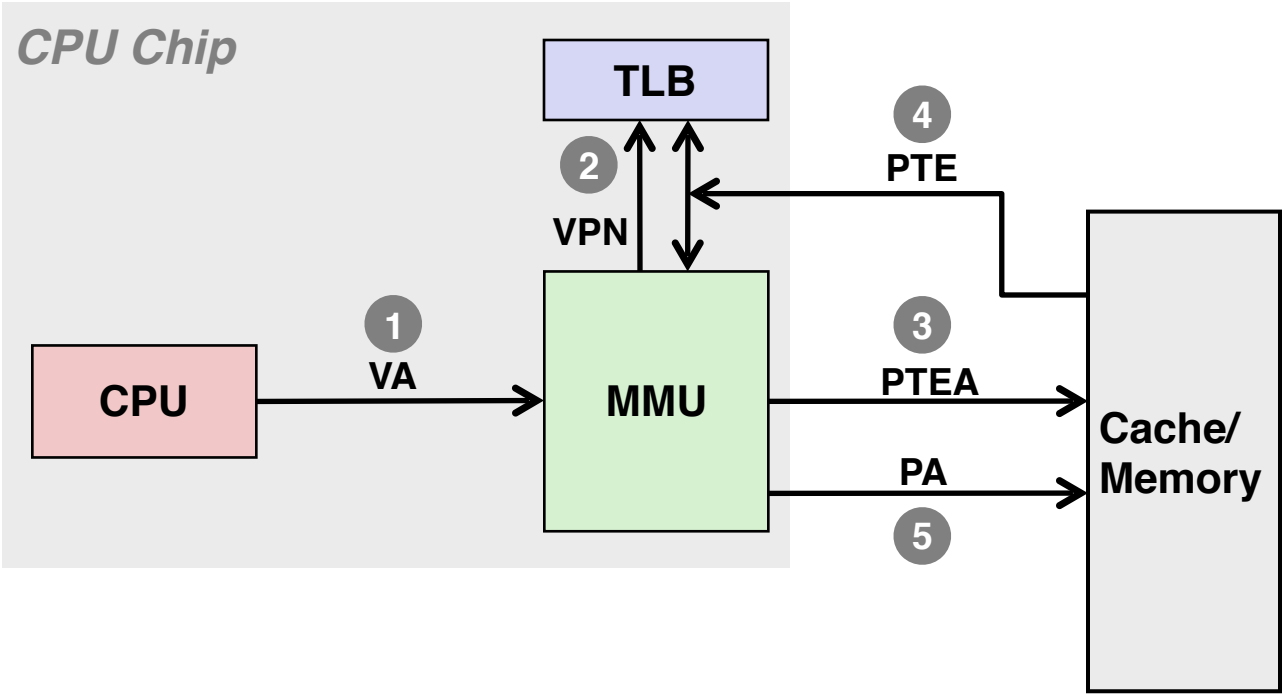
TLB Miss



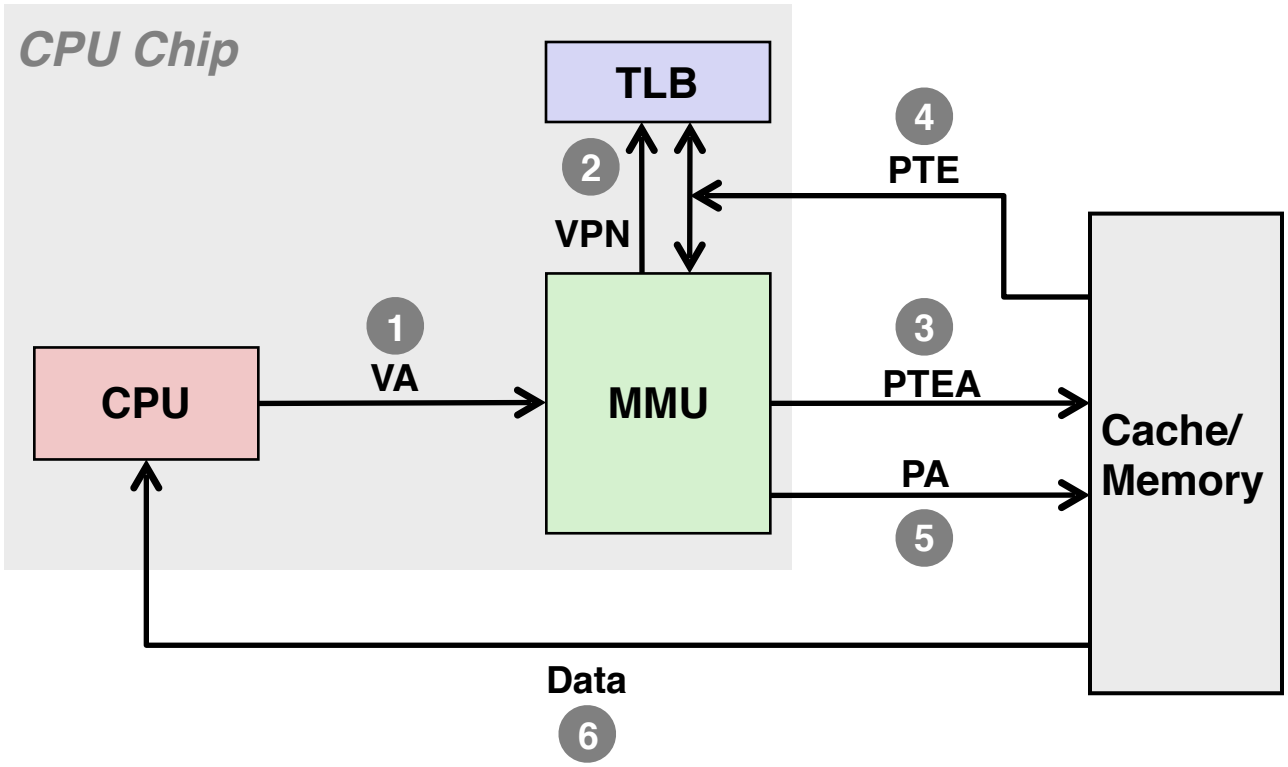
TLB Miss



TLB Miss



TLB Miss


CPU Chip
TLB
4
PTE
2
VPN
1
VA
3
PTEA
CPU
MMU
Cache/
Memory
PA
5
Data
6