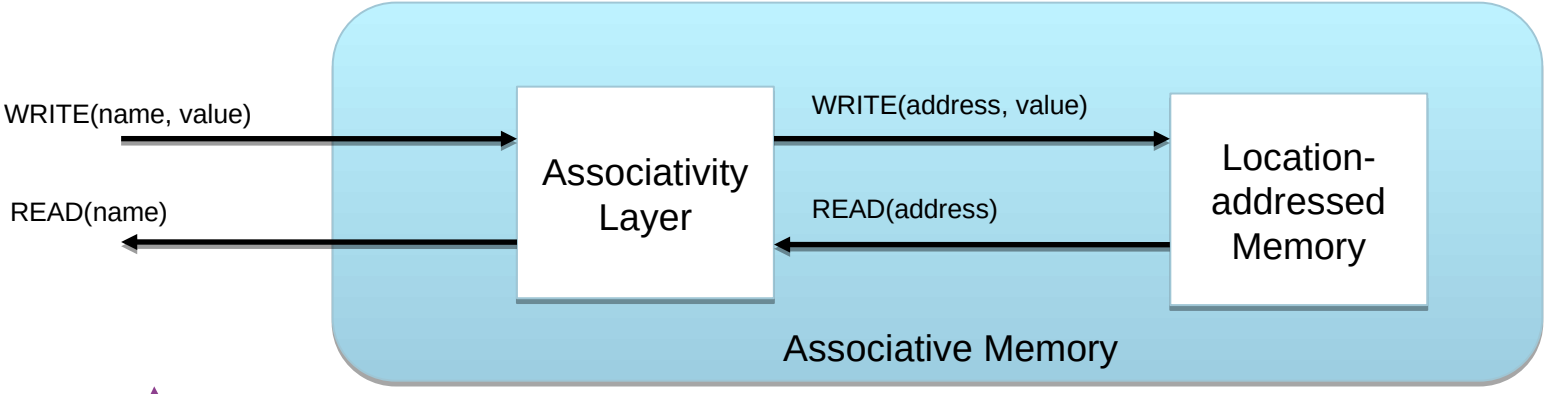

Operating Systems and C
Fall 2022
10. Memory

Memory Abstraction

recall fundamental abstractions

- interpreter during lecture 4
- **memory** **today!**
- communication at end of course

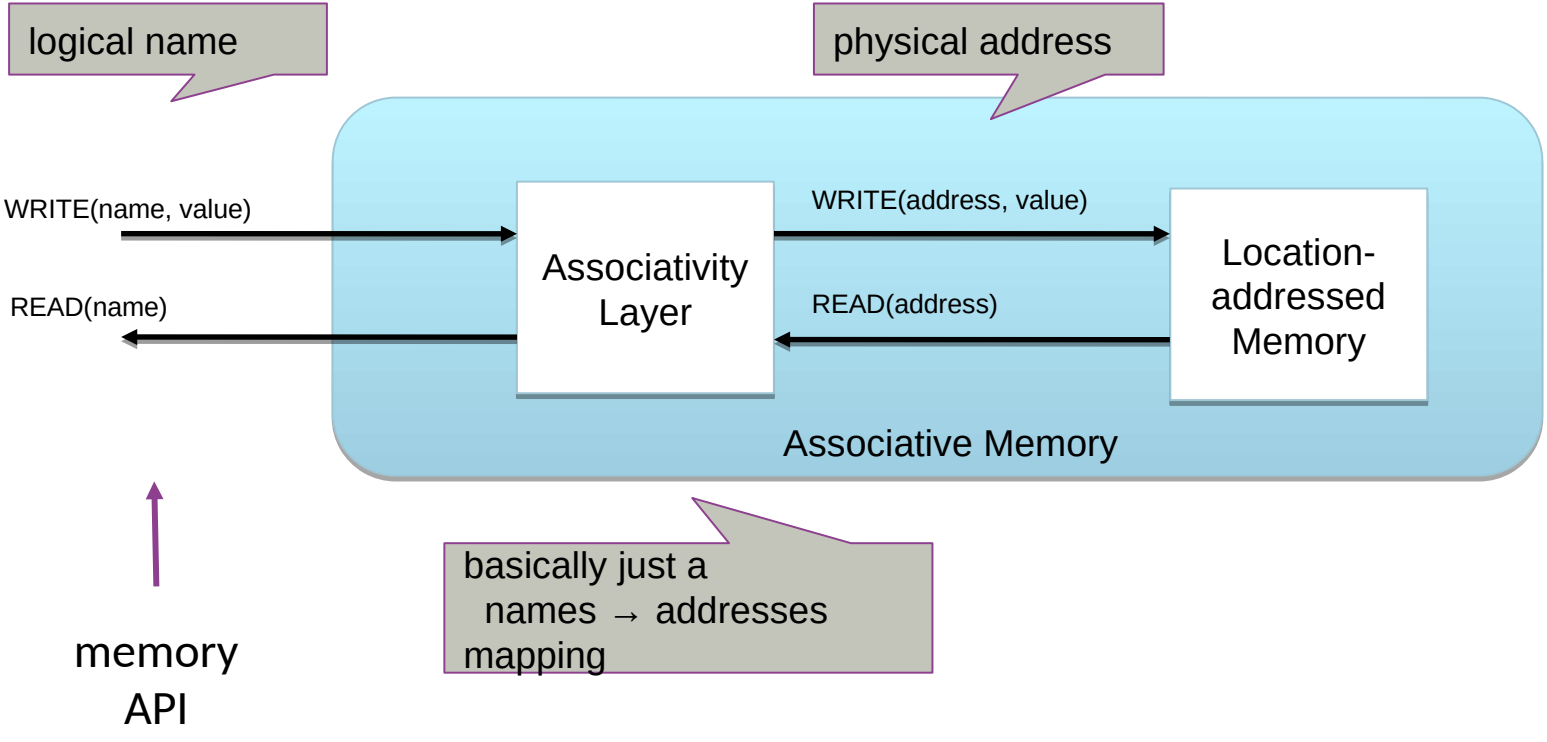


memory API

Source: Saltzer and Kaashoek

Memory Abstraction

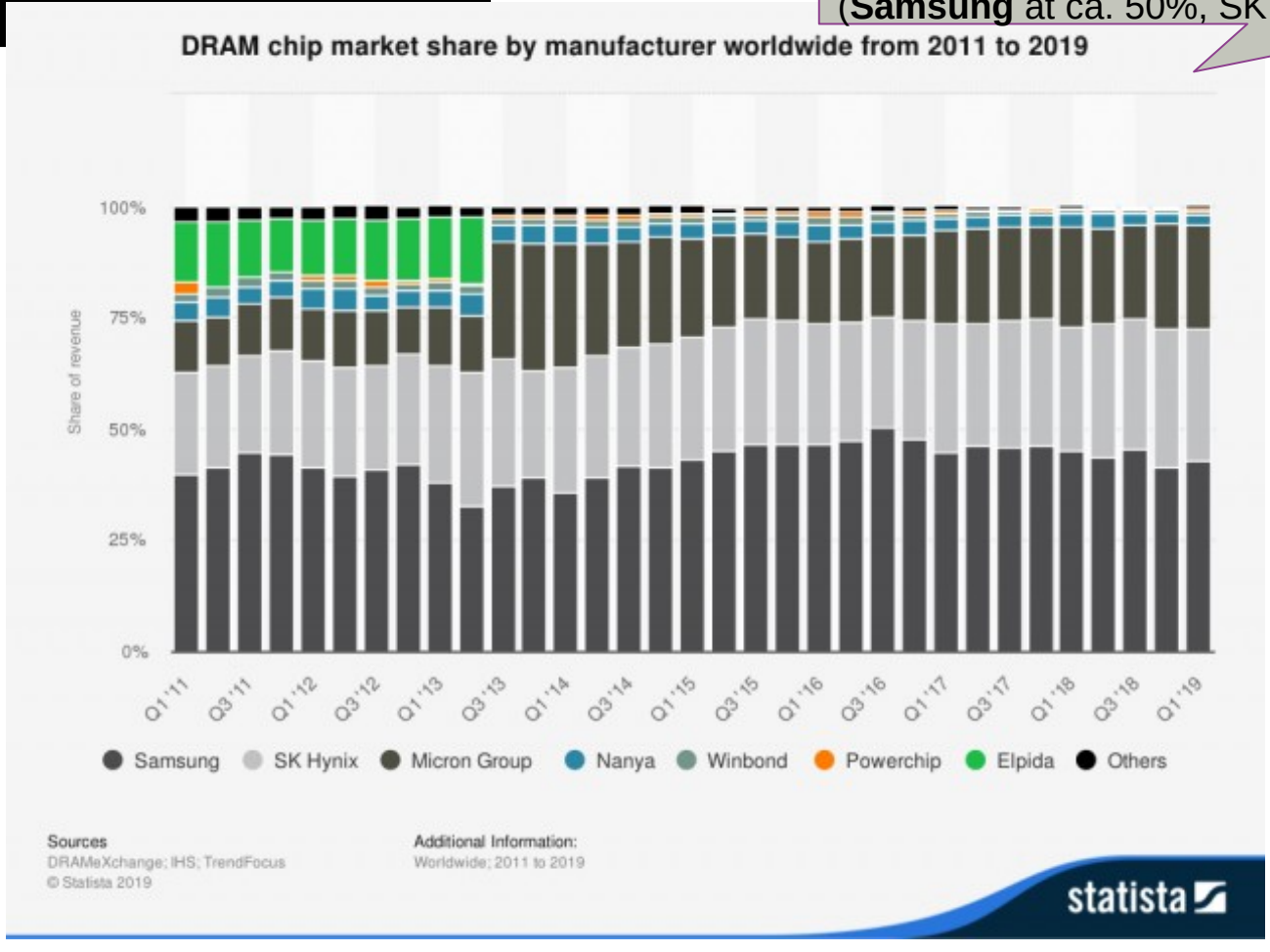
recall fundamental abstractions
- interpreter — during lecture 4
- **memory** — **today!**
- communication — at end of course



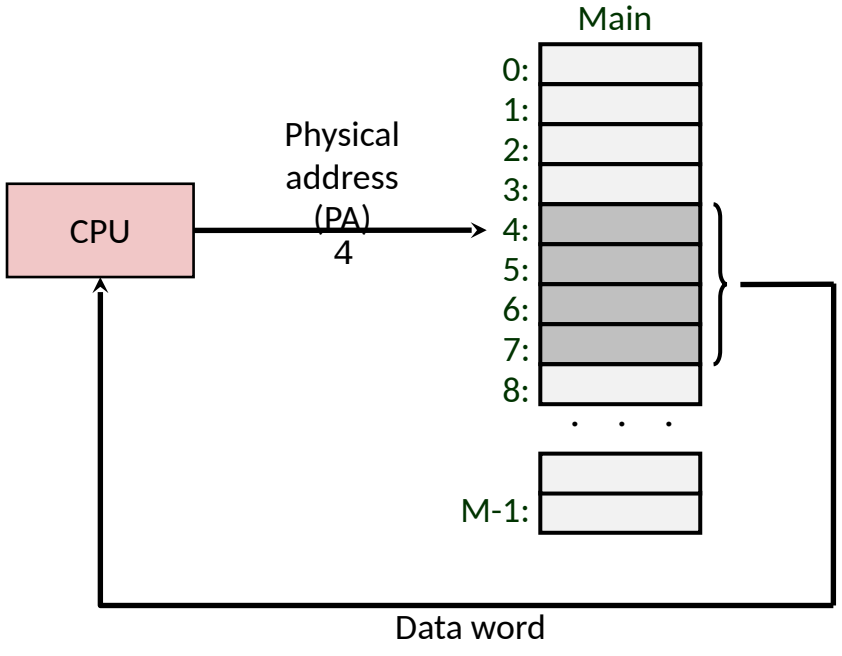
Source: Saltzer and Kaashoek

DRAM Market Shares

geopolitics (10s):
three main manufacturers of memory.
(**Samsung** at ca. 50%, SK Hynix, Micron)



A System Using **Physical** Addressing



address in a CPU request is an actual, physical address.

(so, CPU only accesses physical memory. might be on-chip, might be external)

pro: cheaper… (less chips)
con: caching?
protection? (each proc full access)
manage? (control where/how stored) (performance & security)

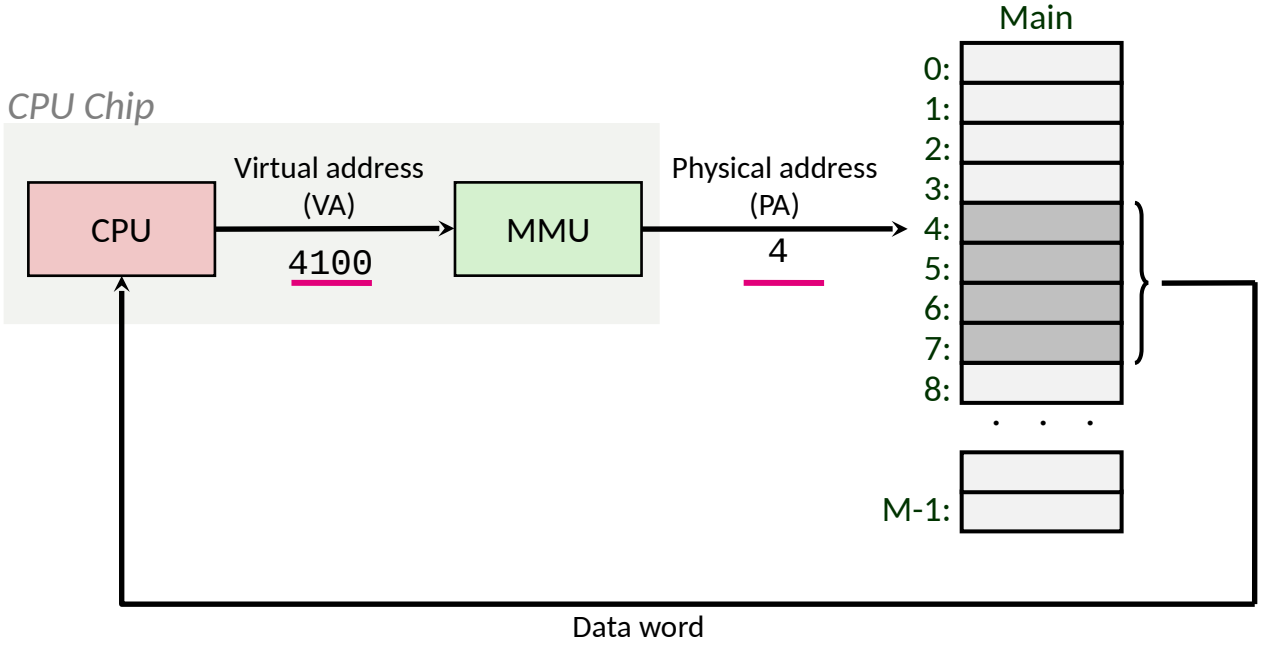
a **level of indirection** is useful (gives us options).

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

A System Using **Virtual** Addressing

level of indirection: maps virtual addresses to physical addresses.

MMU: Memory Management Unit



Used in all modern servers, desktops, and laptops

One of the great ideas in computer science

MMU can do some key clever things. this lecture walks through that.

Address Space

10s

An address space is an ordered set of contiguous addresses (non-negative integers)

Physical address space ⇒ associated to RAM

Virtual address space ⇒ associated to each **process**

from *point of view* of **process**, it has access to whole memory, and that memory belongs to it.
in actuality, it has access to parts of memory, some shared w/ other processes.

Virtual Memory

what does virtual memory give us? (i.e. why VM?)

- Uses main memory efficiently **performance**
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space (from 0 and up)
- Isolates address spaces **security**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information

VM is a Tool for Caching

(i.e. for organizing how physical memory is used)

Virtual memory is an array of N contiguous bytes stored **on disk**.

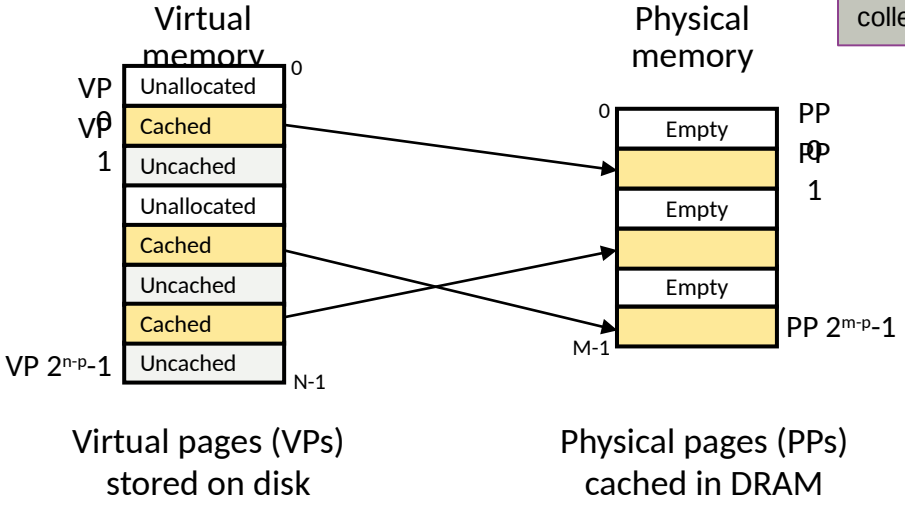
The contents of the array on disk are **cached** in *physical memory* (*DRAM cache*)

These cache blocks are called *pages* (size (quanta) P = 2^p bytes

Unit of transfer between disk & virtual memory

4k, 8k, 16k
VM thus = a collection of pages

Q: how much VM can we have?



VM is a Tool for Caching

(i.e. for organizing how physical memory is used)

Virtual memory is an array of N contiguous bytes stored **on disk**.

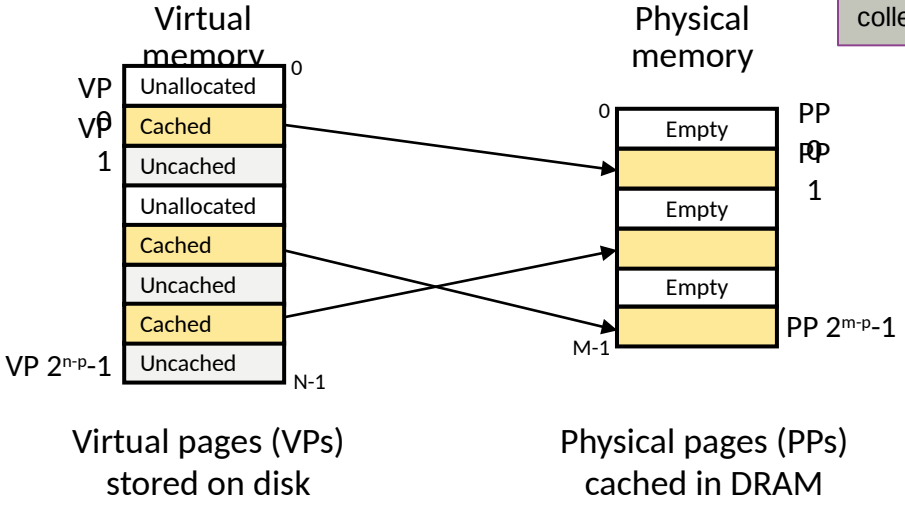
The contents of the array on disk are **cached** in *physical memory* (*DRAM cache*)

These cache blocks are called *pages* (size (quanta) P = 2^p bytes

Unit of transfer between disk & virtual memory

4k, 8k, 16k
VM thus = a collection of pages

Q: how much VM can we have?
A: CPU issues address requests. VM size bounded by CPU word size



VM is a Tool for Caching

(i.e. for organizing how physical memory is used)

Virtual memory is an array of N contiguous bytes stored **on disk**.

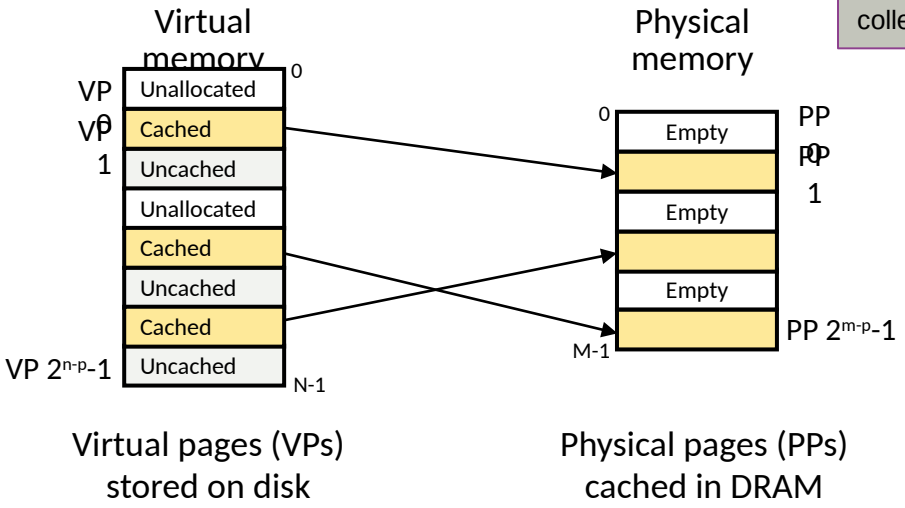
The contents of the array on disk are **cached** in *physical memory* (*DRAM cache*)

These cache blocks are called *pages* (size (quanta) P = 2^p bytes

Unit of transfer between disk & virtual memory

4k, 8k, 16k
VM thus = a collection of pages

Q: how much VM can we have?
A: CPU issues address requests. VM size bounded by CPU word size



when you run out of memory, you don't run out of addresses, but actual phys mem

level of indirection creates opportunities!

history; 2 GB limit

physical address extension (PAE): MMU "hack" enabling more than 2^32 physical mem despite CPU being 32-bit

today, address space size:
virtual mem ⩾ physical mem
VM creates illusion (to proc) of more memory, using caching.

DRAM Cache Organization

DRAM for phys. mem, SRAM for registers

DRAM cache organization driven by the enormous miss penalty

DRAM is about ***10x*** slower than SRAM

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

<u>Consequences:</u>

Large page (block) size: typically 4-8 KB, sometimes 4 MB

Fully associative

- Any VP can be placed in any PP
- Need a “large” mapping function – different from CPU caches

Highly sophisticated, expensive replacement algorithms

- Too complicated and open-ended to be implemented in hardware

Write-back rather than write-through

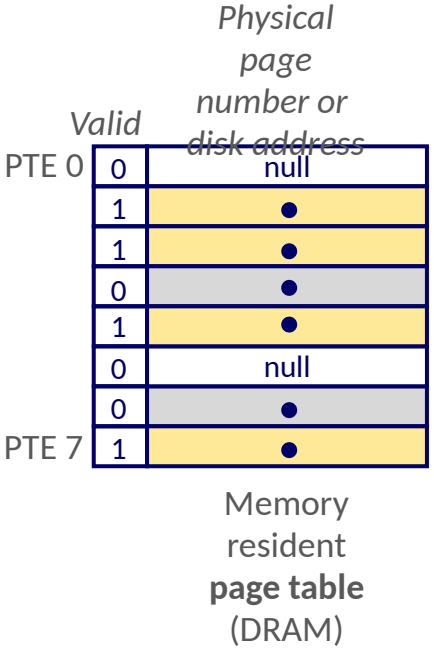
Cannot use the (simple) mechanism we saw in performance-track (CPU cache)

solution: page table (next slide)

Page Tables

serves as our
virt-phys address mapping

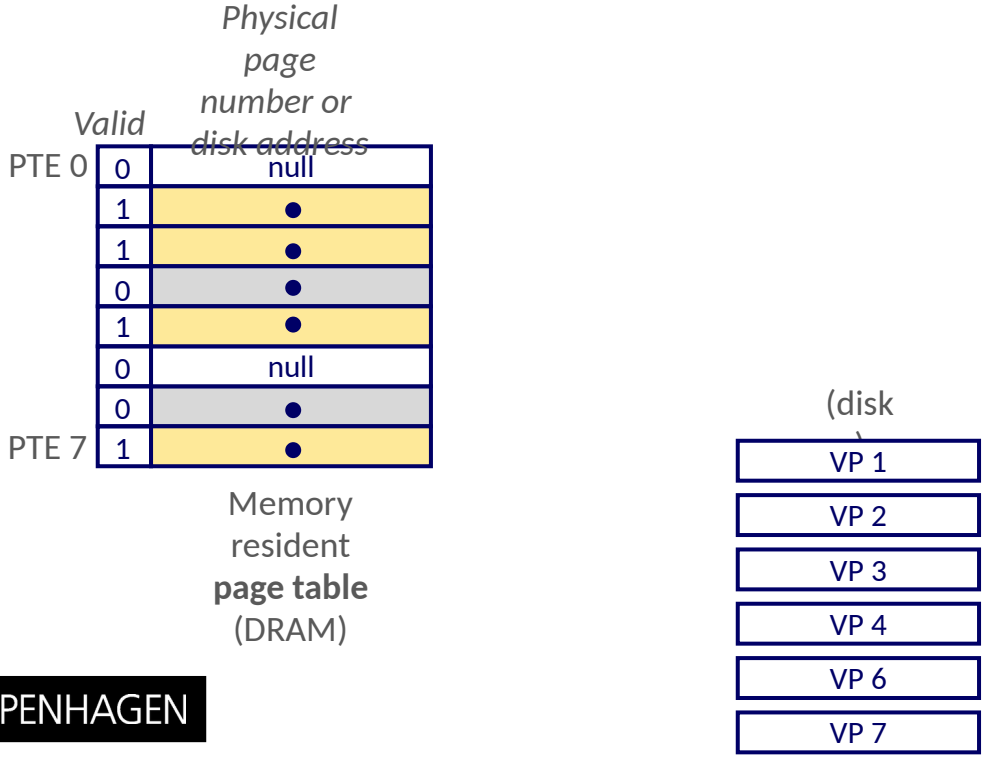
A *page table* is an array of page table entries (PTEs) that maps **virtual** pages to **physical** pages. Per-process kernel data structure in DRAM



Page Tables

serves as our virt-phys address mapping

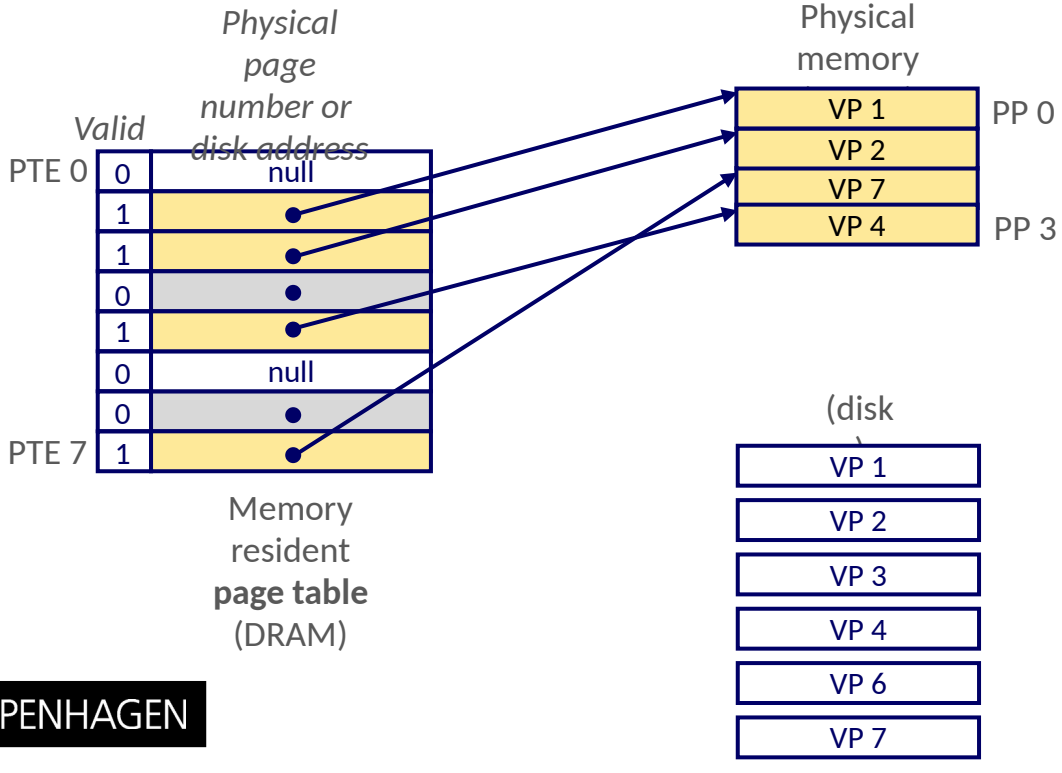
A *page table* is an array of page table entries (PTEs) that maps **virtual** pages to **physical** pages. Per-process kernel data structure in DRAM



Page Tables

serves as our virt-phys address mapping

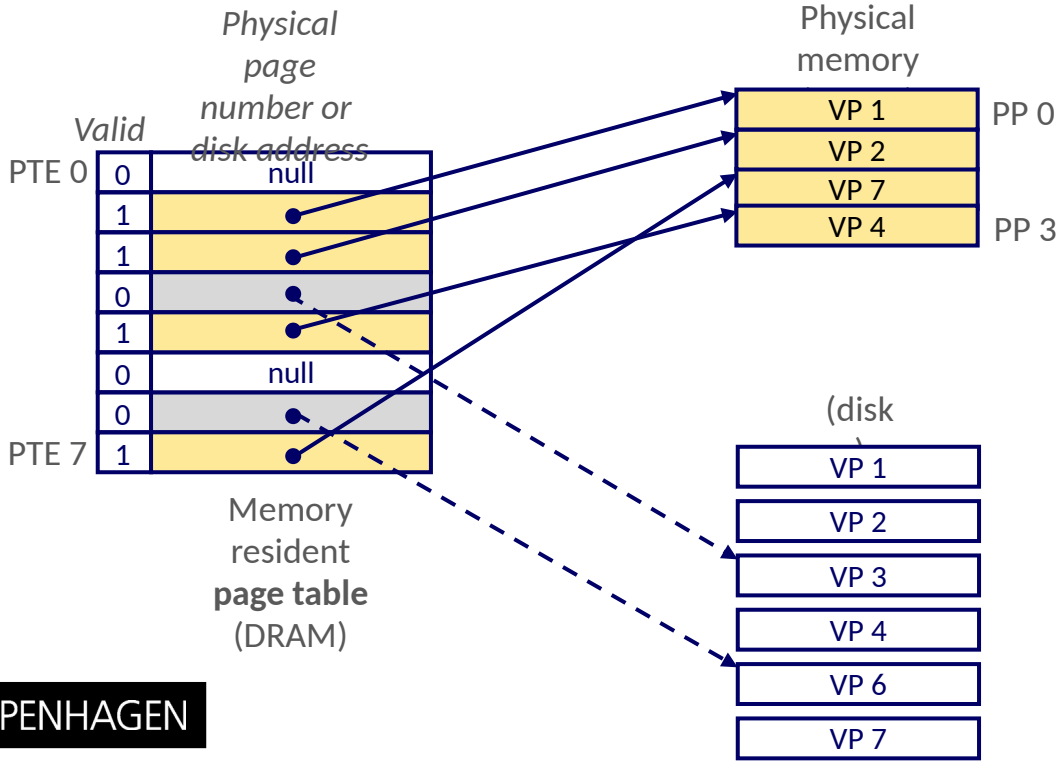
A *page table* is an array of page table entries (PTEs) that maps **virtual** pages to **physical** pages. Per-process kernel data structure in DRAM



Page Tables

serves as our virt-phys address mapping

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



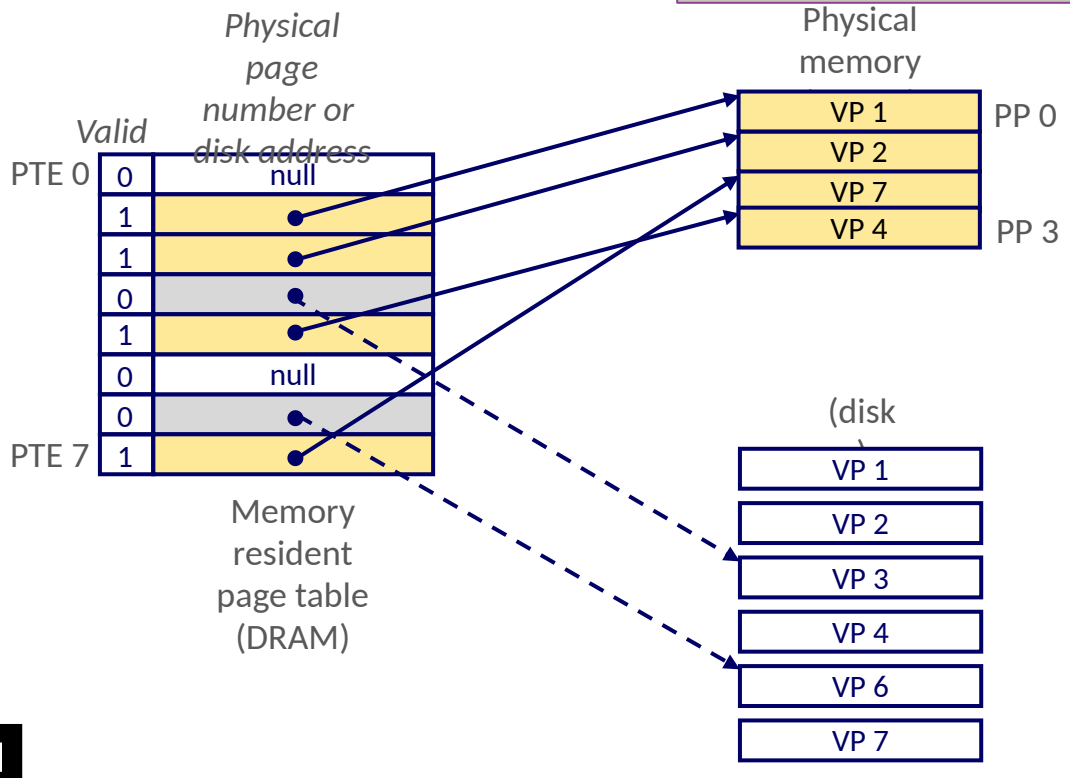
VM creates illusion (to proc) that data is always in DRAM. but in practice, there isn't space for all VM there. hence DRAM is cache.

Page Hit

MMU gets address (virtual), looks up page table…

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

return addr of page in DRAM

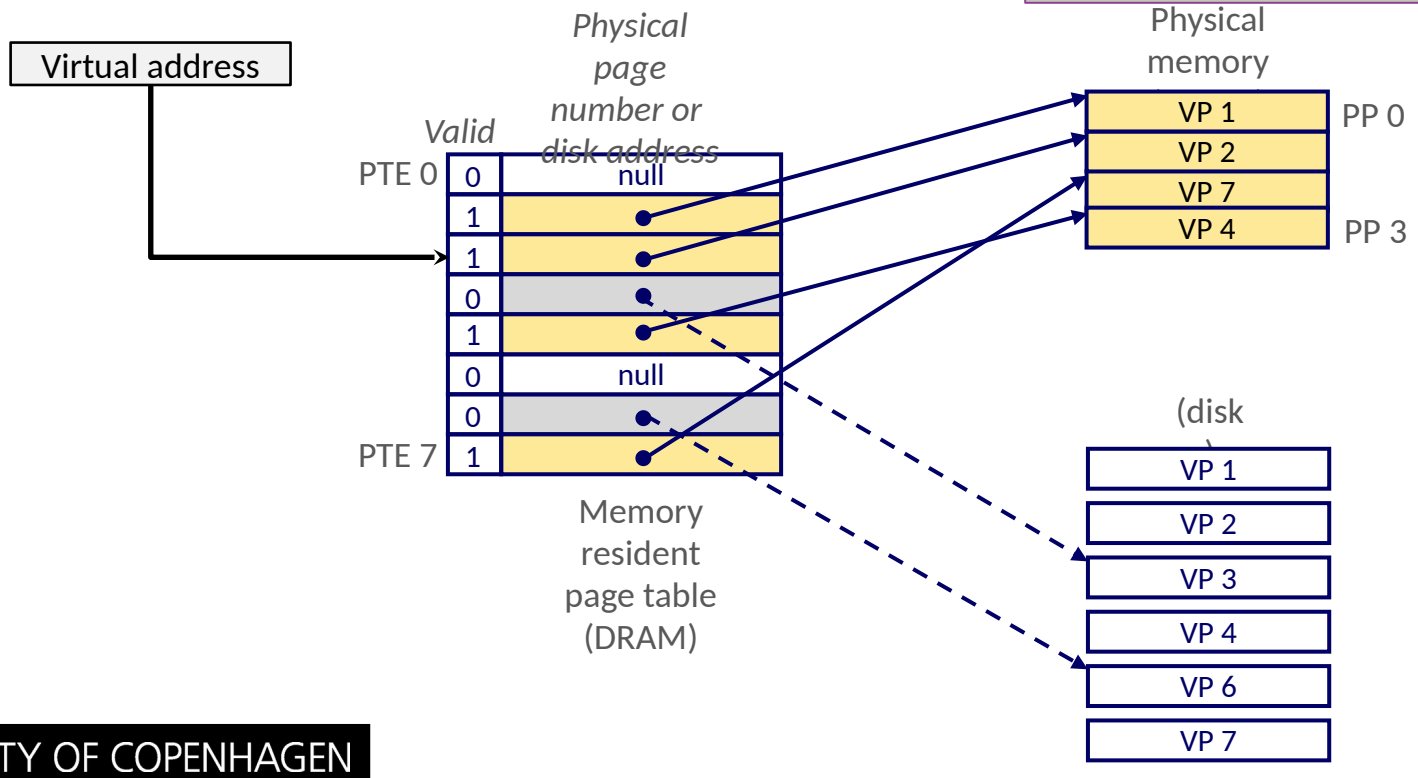


Page Hit

MMU gets address (virtual), looks up page table…

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

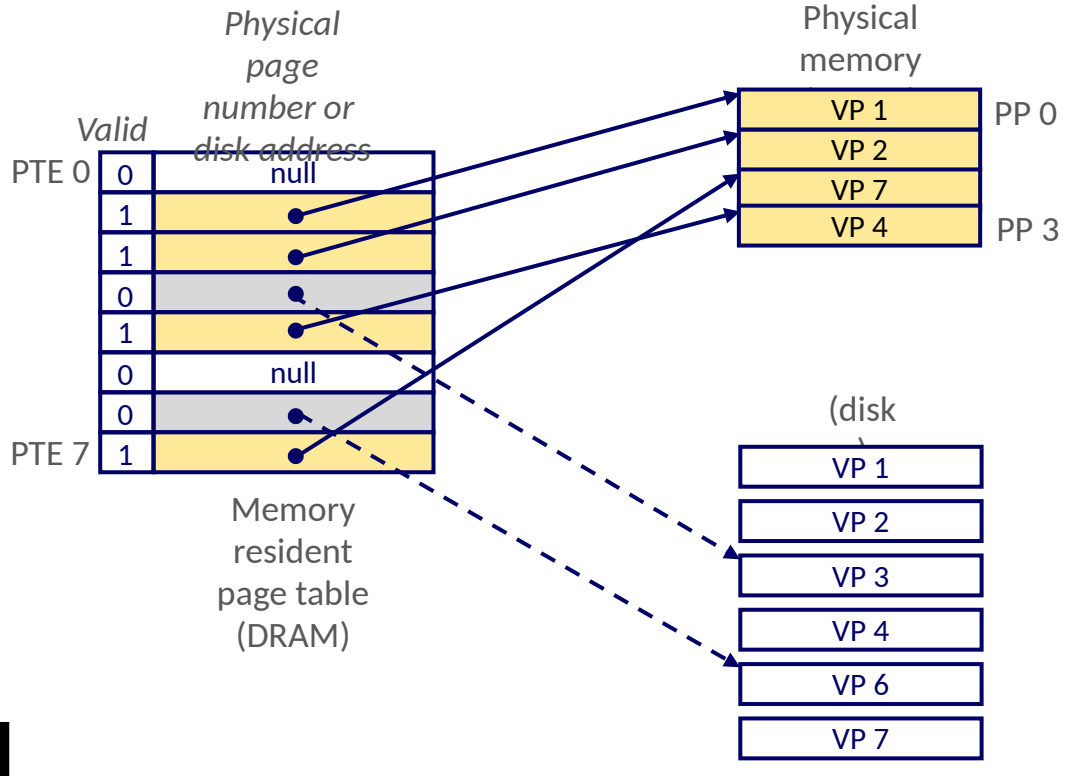
return addr of page in DRAM



Page Fault

what happens on a miss?
(bring the page to the cache)
implemented w/ exceptions.

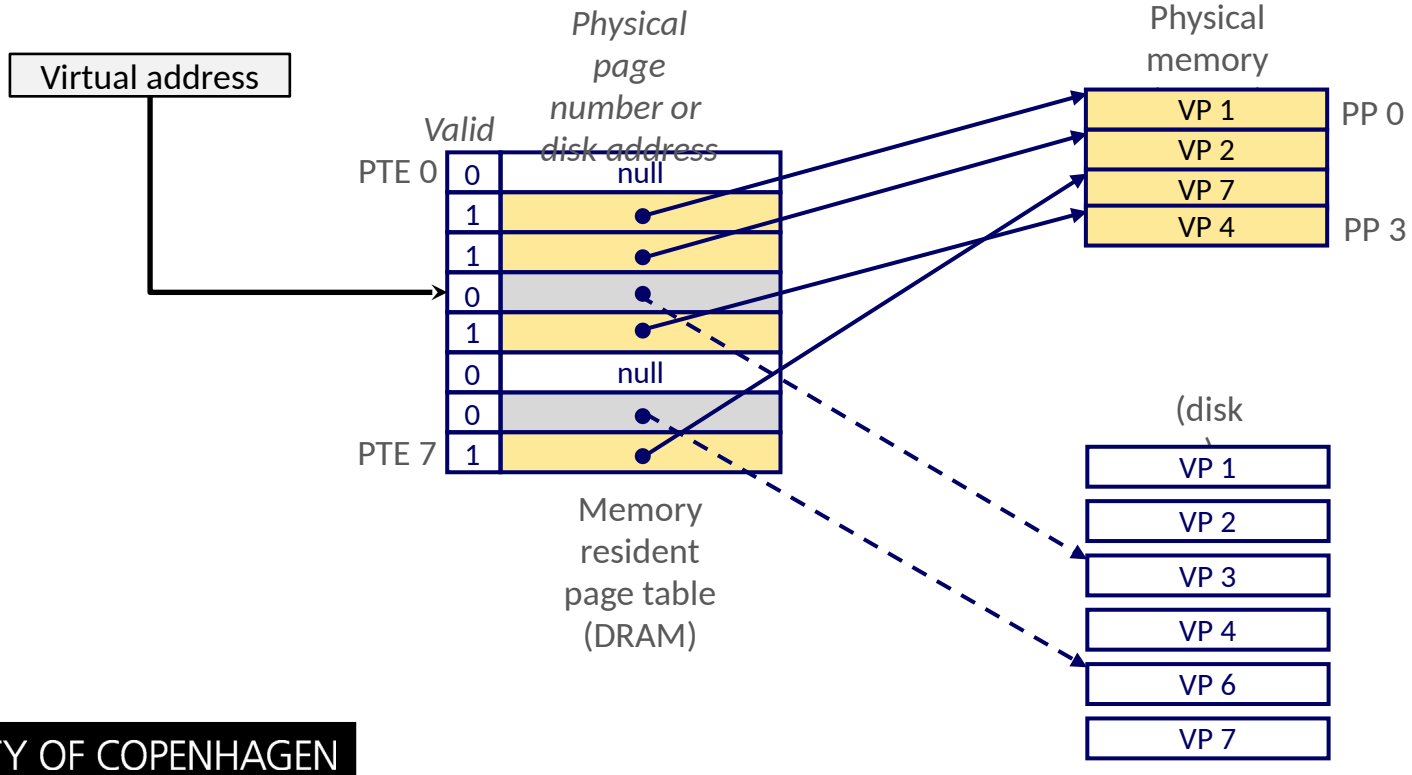
Page fault: reference to VM word that is not in physical memory (DRAM cache miss)



Page Fault

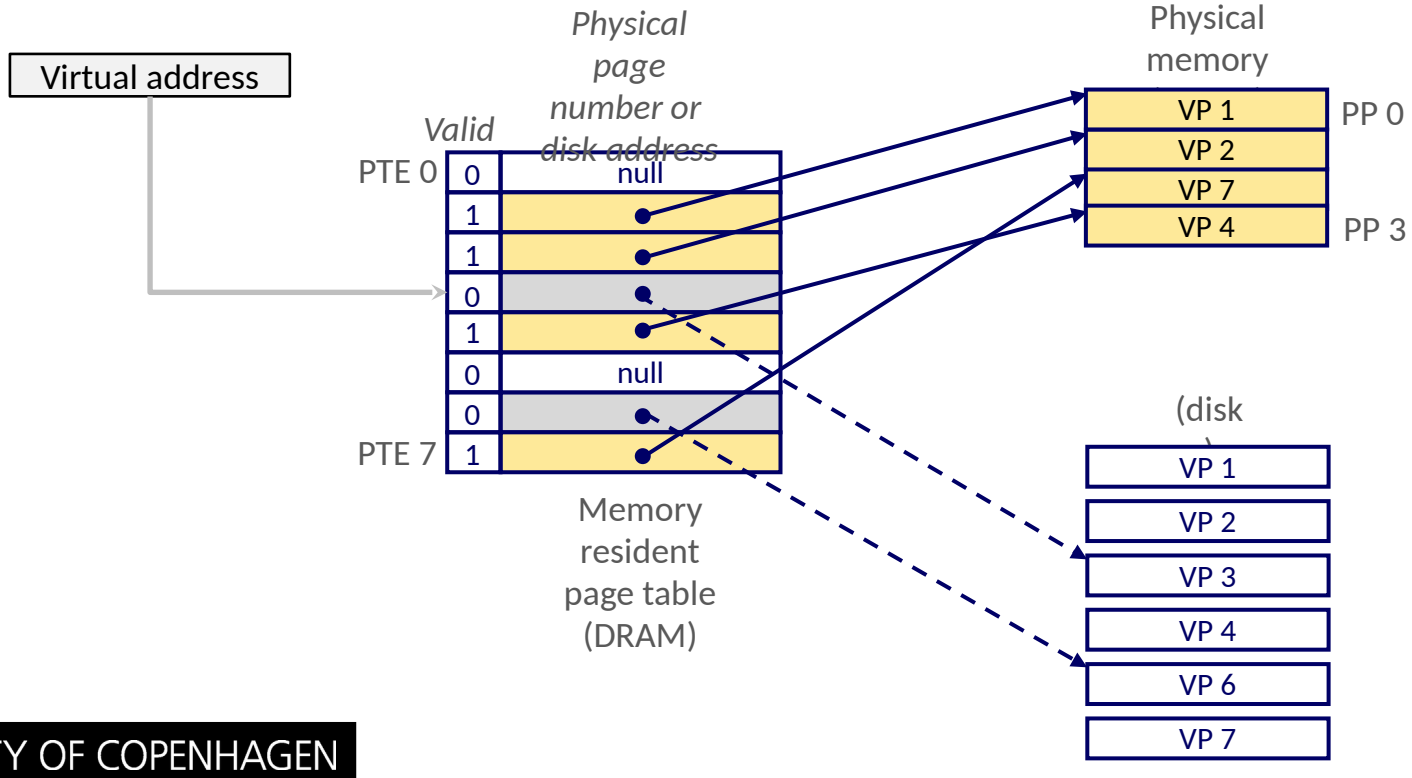
what happens on a miss?
(bring the page to the cache)
implemented w/ exceptions.

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



Handling Page Fault

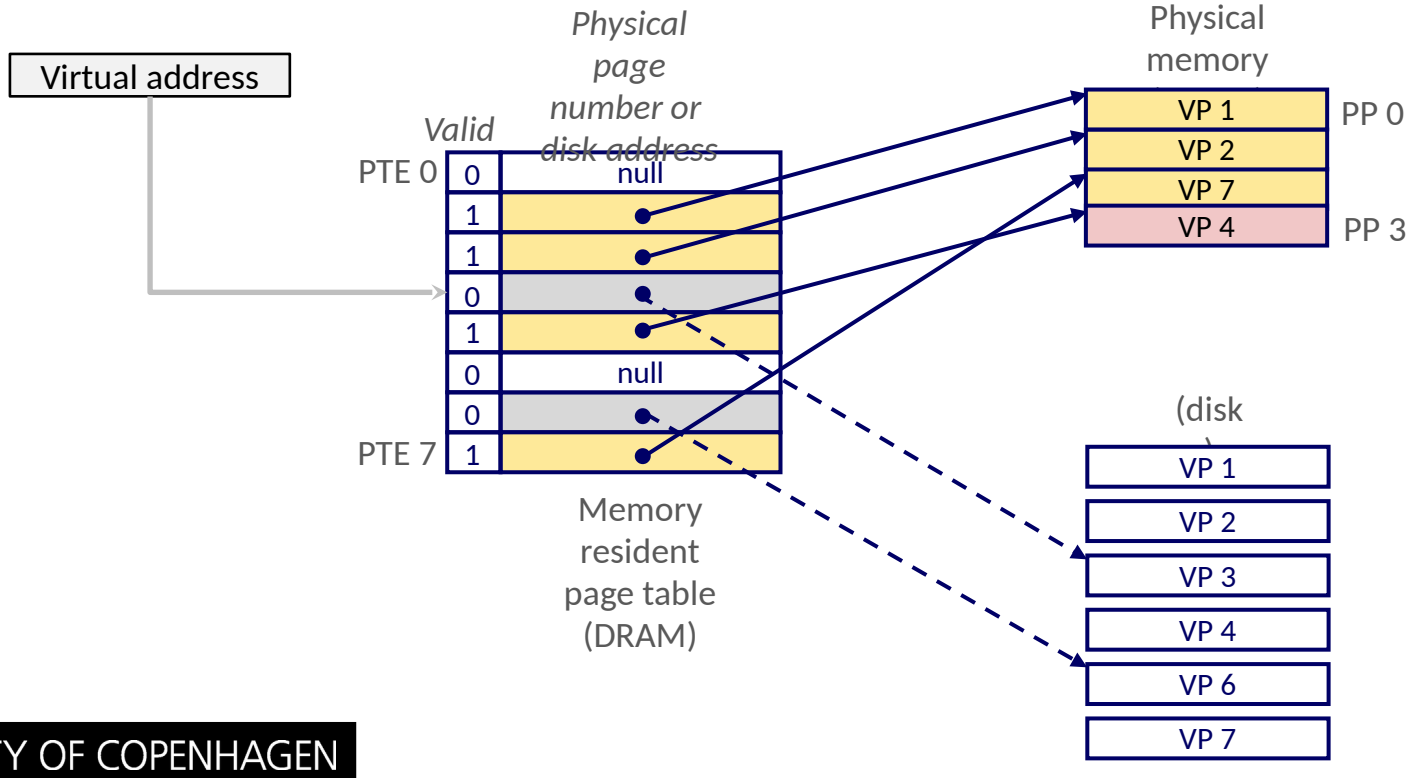
Page miss causes page fault (an exception)



Handling Page Fault

(flip forth)

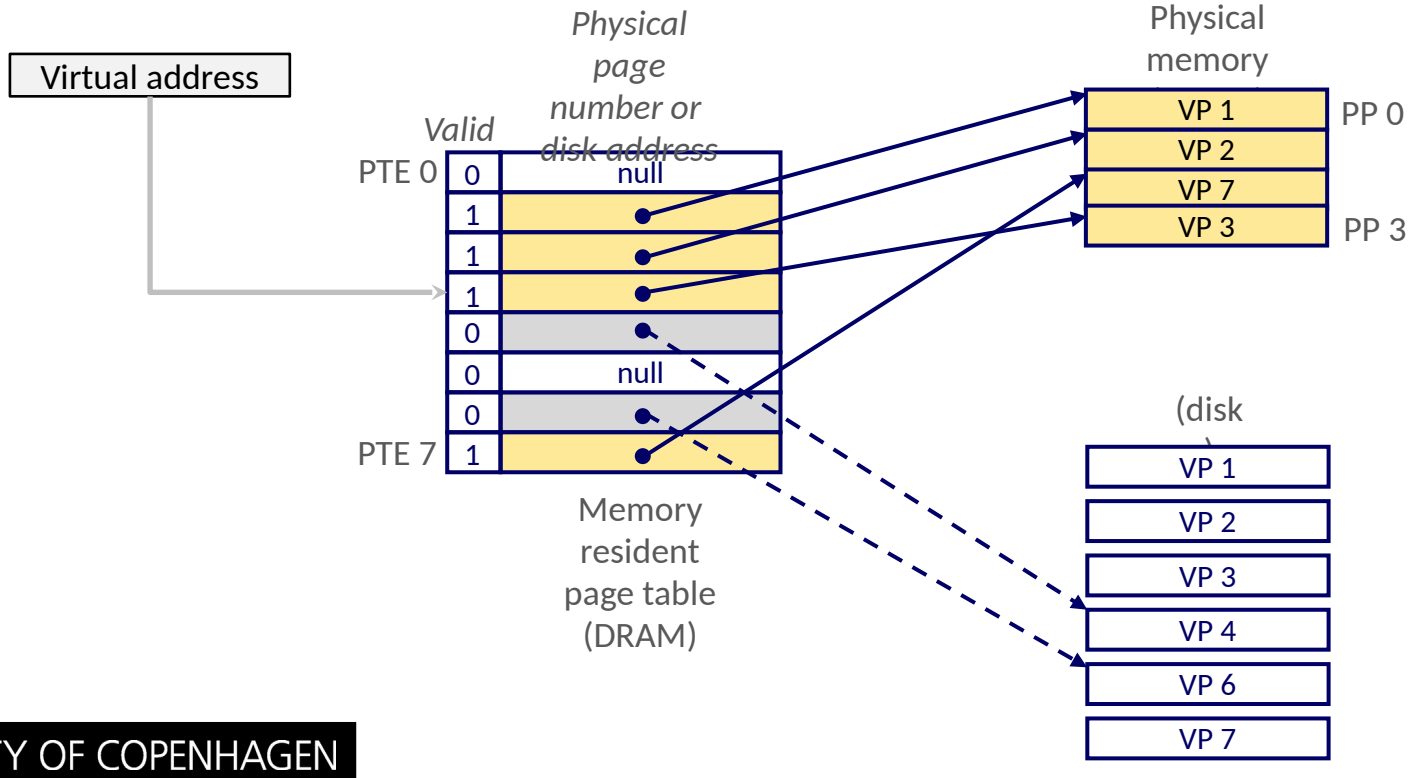
Page miss causes page fault (an exception)
Page fault handler selects a victim to be evicted (here VP 4)



Handling Page Fault

(flip back)

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



Handling Page Fault

this mechanism is making sure to bring data that process needs, to DRAM.

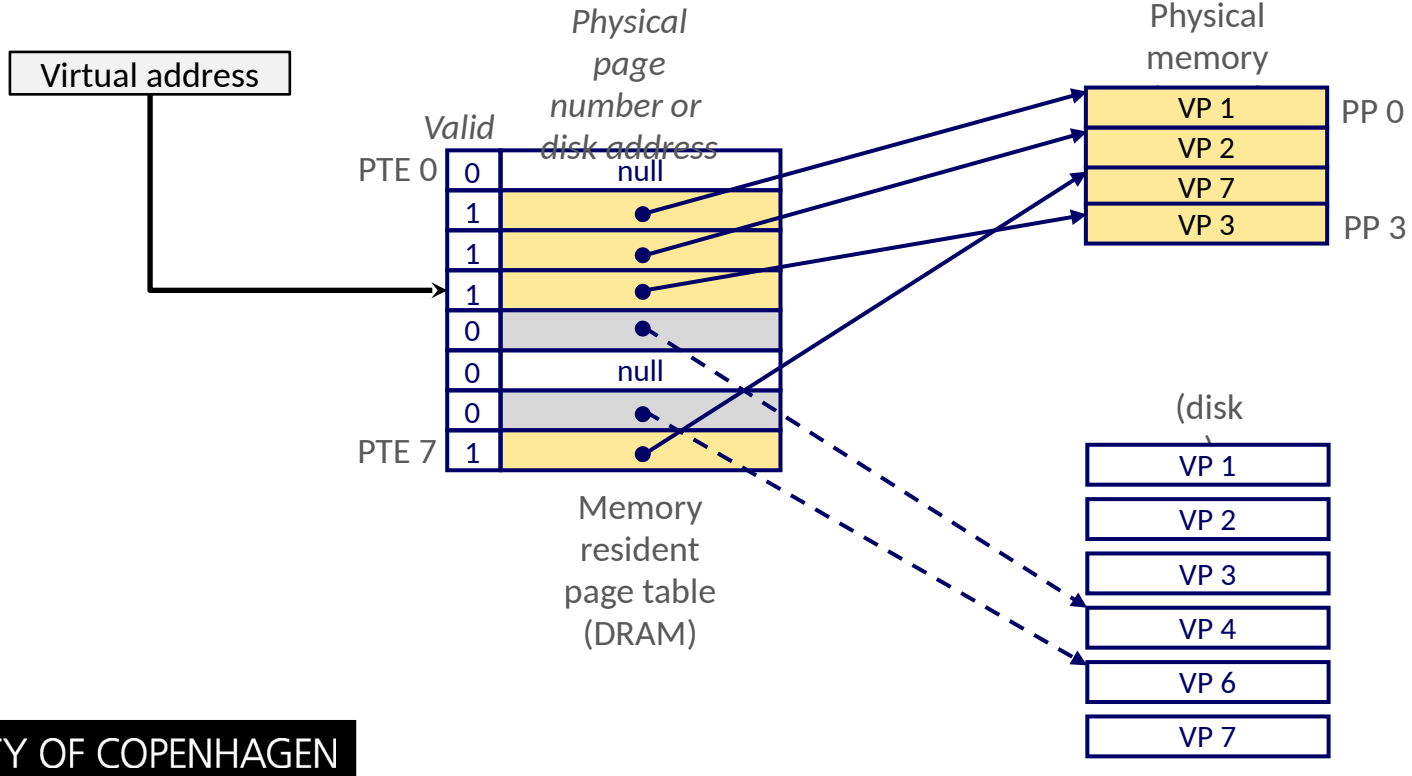
page fault is generated by hardware, but handled by the OS

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!

(how: exception ⇒ control given to OS)



Locality to the Rescue Again!

Virtual memory **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 (to/from disk; “swapping”; slooow)

you can literally hear this in mechanical hard drives; a “grinding” sound

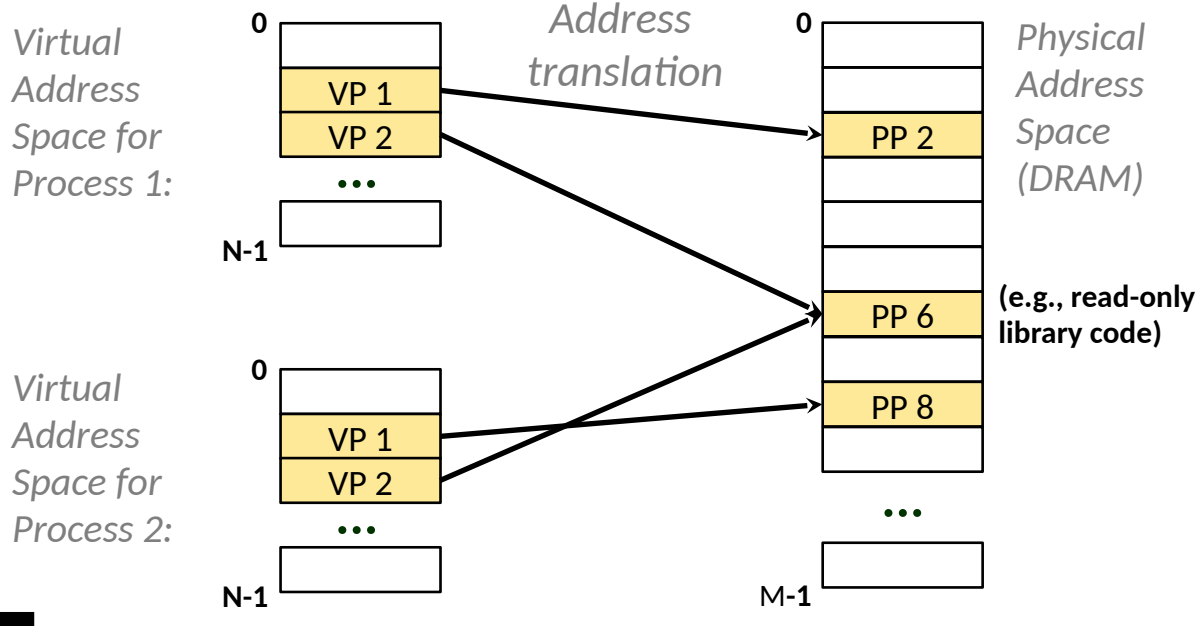
performance-part done (caching, page table & fault). next part: security

VM as a Tool for Memory Management

protection

Key idea: each process has its own virtual address space

- It can view memory as a simple linear array
- Mapping function scatters addresses through physical memory
- Well chosen mappings simplify memory allocation and management



might have same structure (e.g. multiple instances of same program),
but stored differently in physical memory

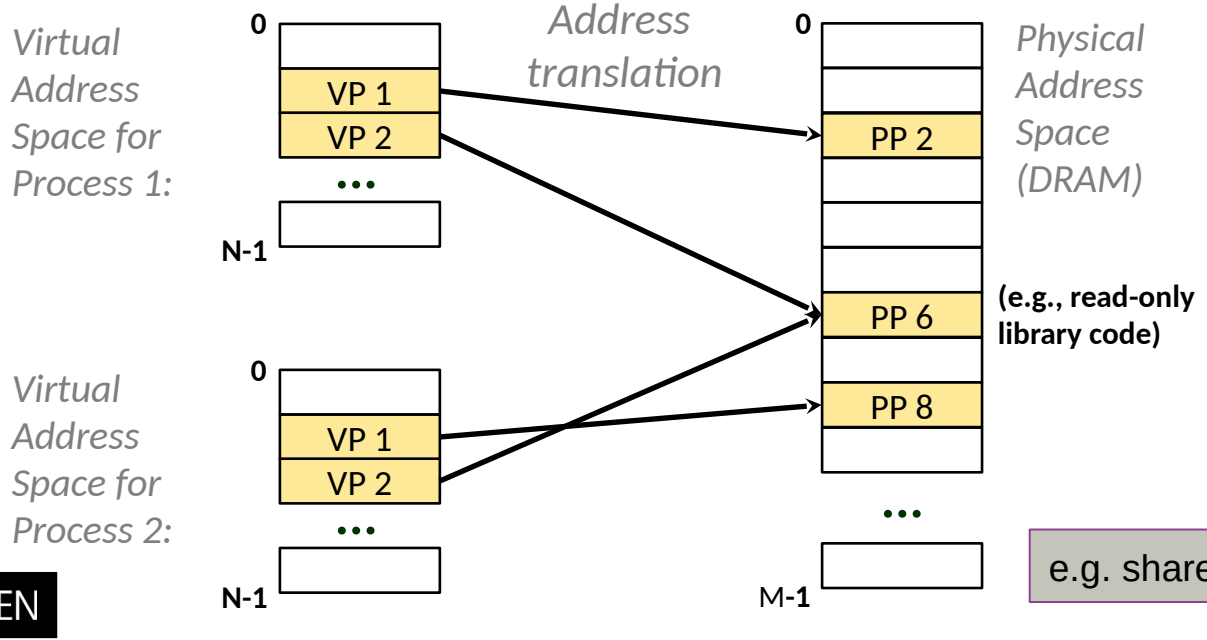
VM as a Tool for Memory Management

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)



locality:
you want, as much as possible, to have contiguous VM pages mapped to contiguous PM pages. (but not too much; inefficient cache use)

e.g. shared library

Shared Memory in Linux

10s

ow-to-use-shared-memory-with-linux-in-c

Shared Memory Segment

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("hello.txt", 'R')) == -1) /*Here the file must exist */
{
        perror("ftok");
        exit(1);
    }

    /*  create the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }
```

Memory Mapped Segment/File

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
  // Our memory buffer will be readable and writable:
  int protection = PROT_READ | PROT_WRITE;

  // The buffer will be shared (meaning other processes can access it), but
  // anonymous (meaning third-party processes cannot obtain an address for it),
  // so only this process and its children will be able to use it:
  int visibility = MAP_SHARED | MAP_ANONYMOUS;

  // The remaining parameters to `mmap()` are not important for this use case,
  // but the manpage for `mmap` explains their purpose.
  return mmap(NULL, size, protection, visibility, -1, 0);
}
```

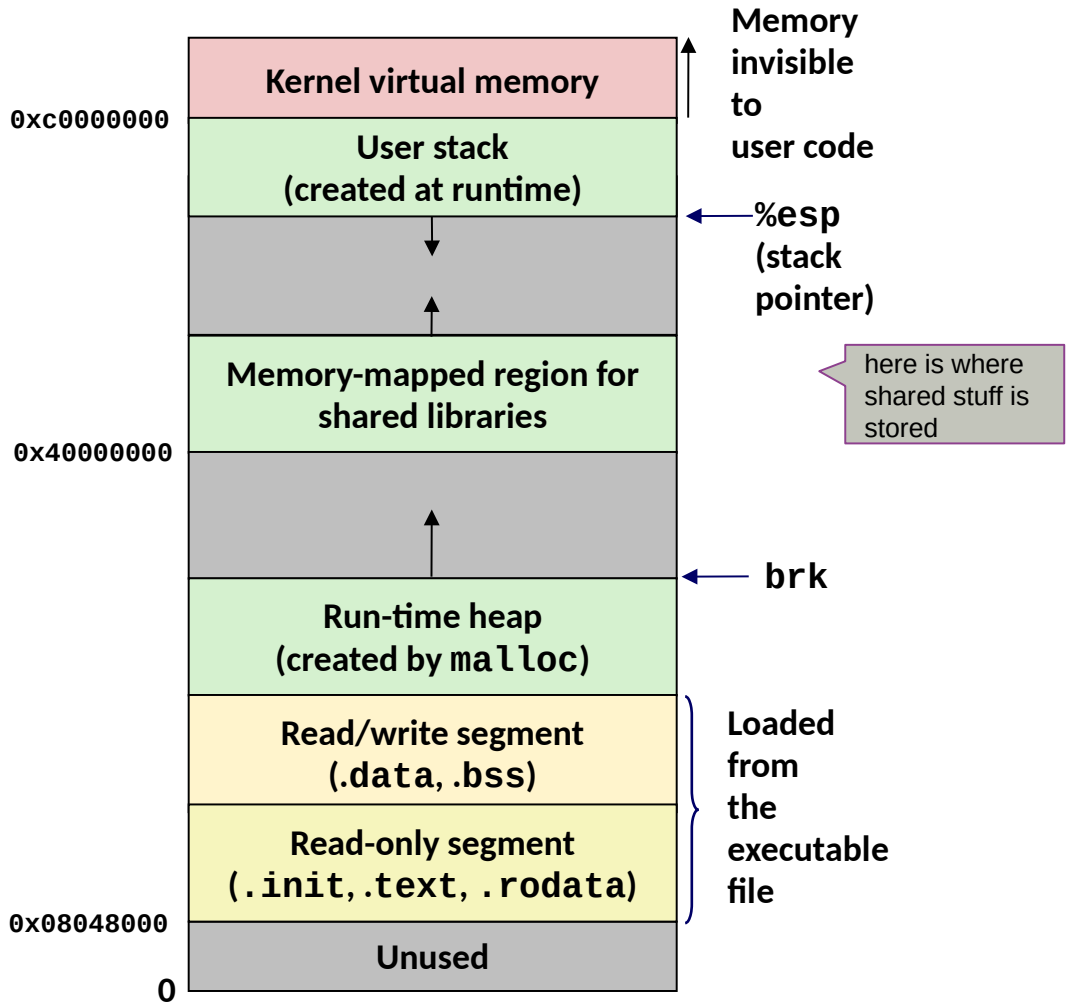
Linux default: no sharing between processes. if you want sharing (shared memory), then you have to work for it.
there are primitives in the C std lib for this.

old UNIX way of sharing between processes.

fork; child shares mem w/ parent

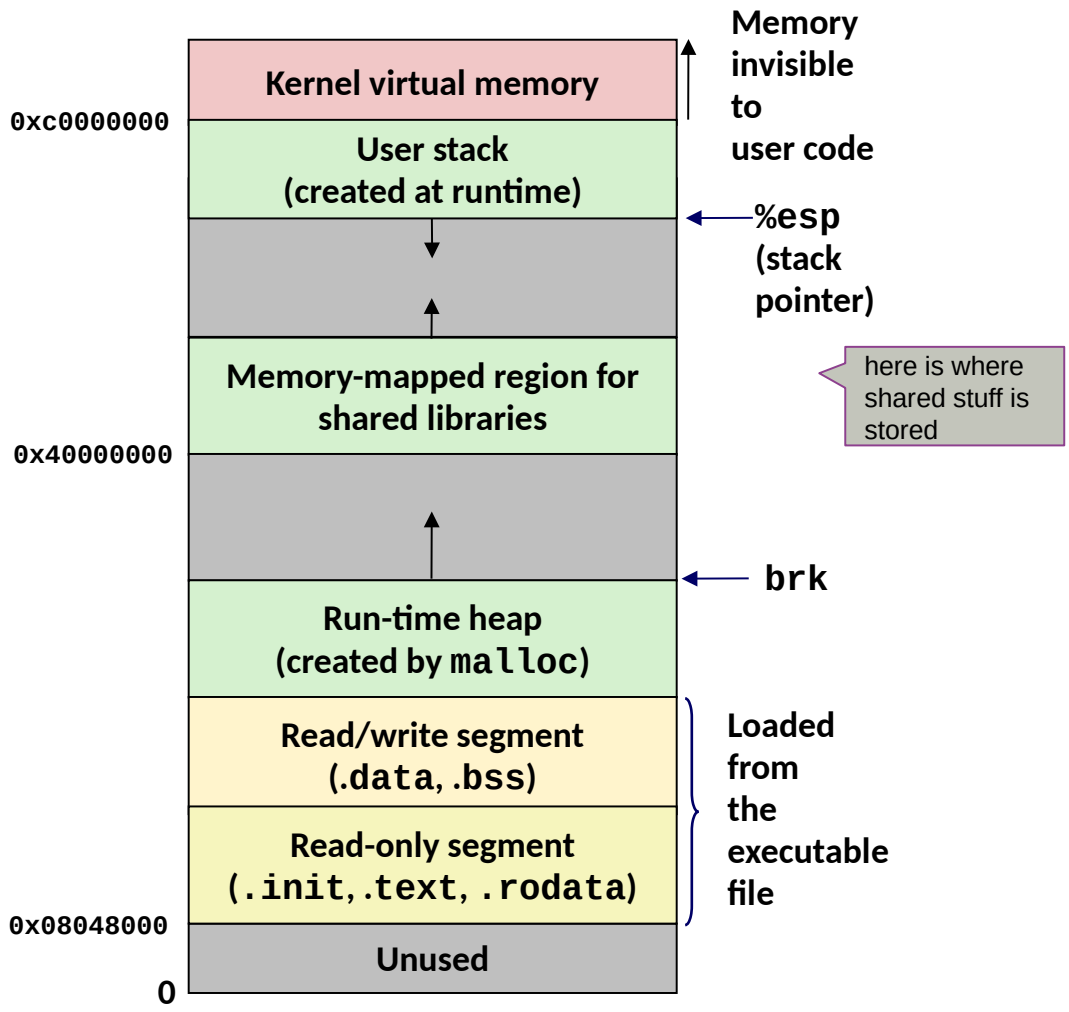
like VP2/PP6 before

Simplifying Linking and Loading



Simplifying Linking and Loading

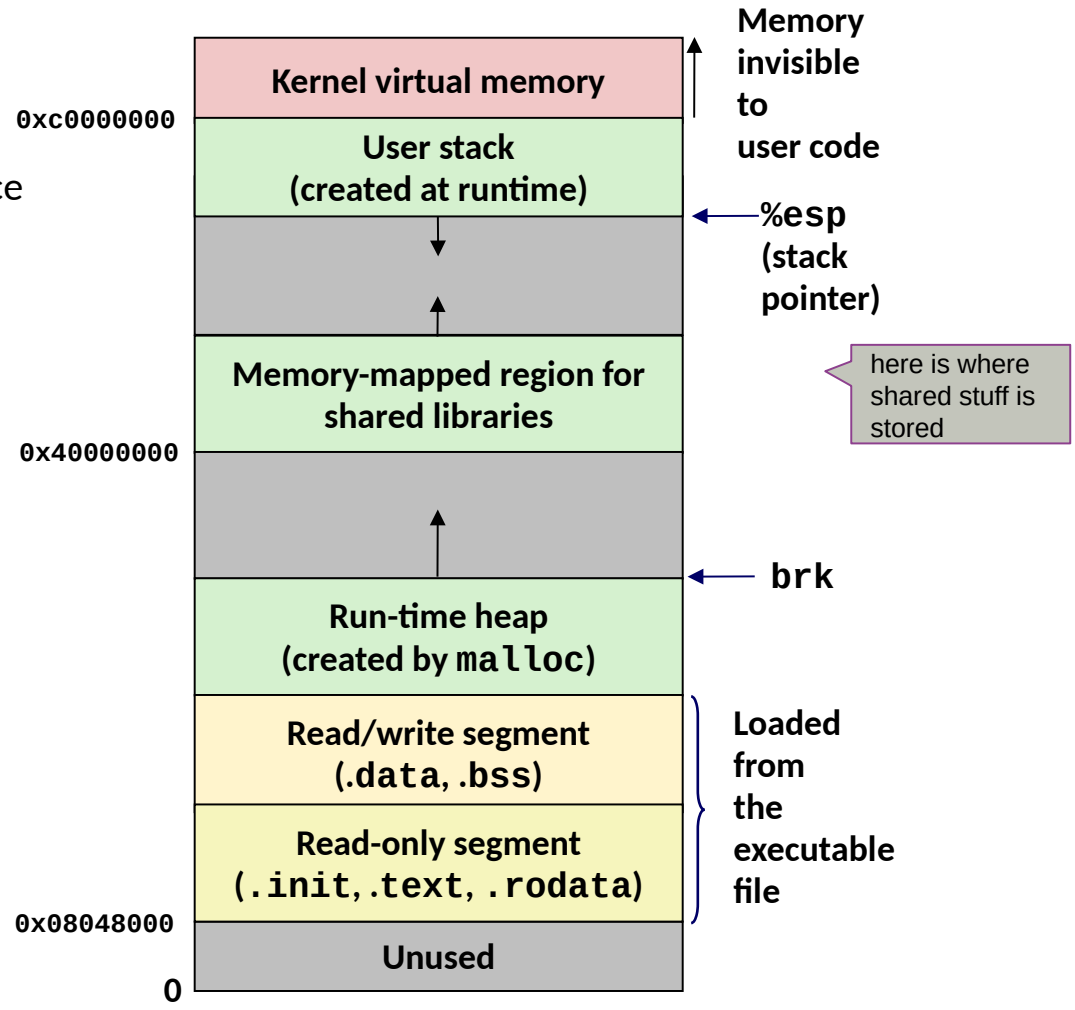
Linking



Simplifying Linking and Loading

Linking

Each program has similar virtual address space

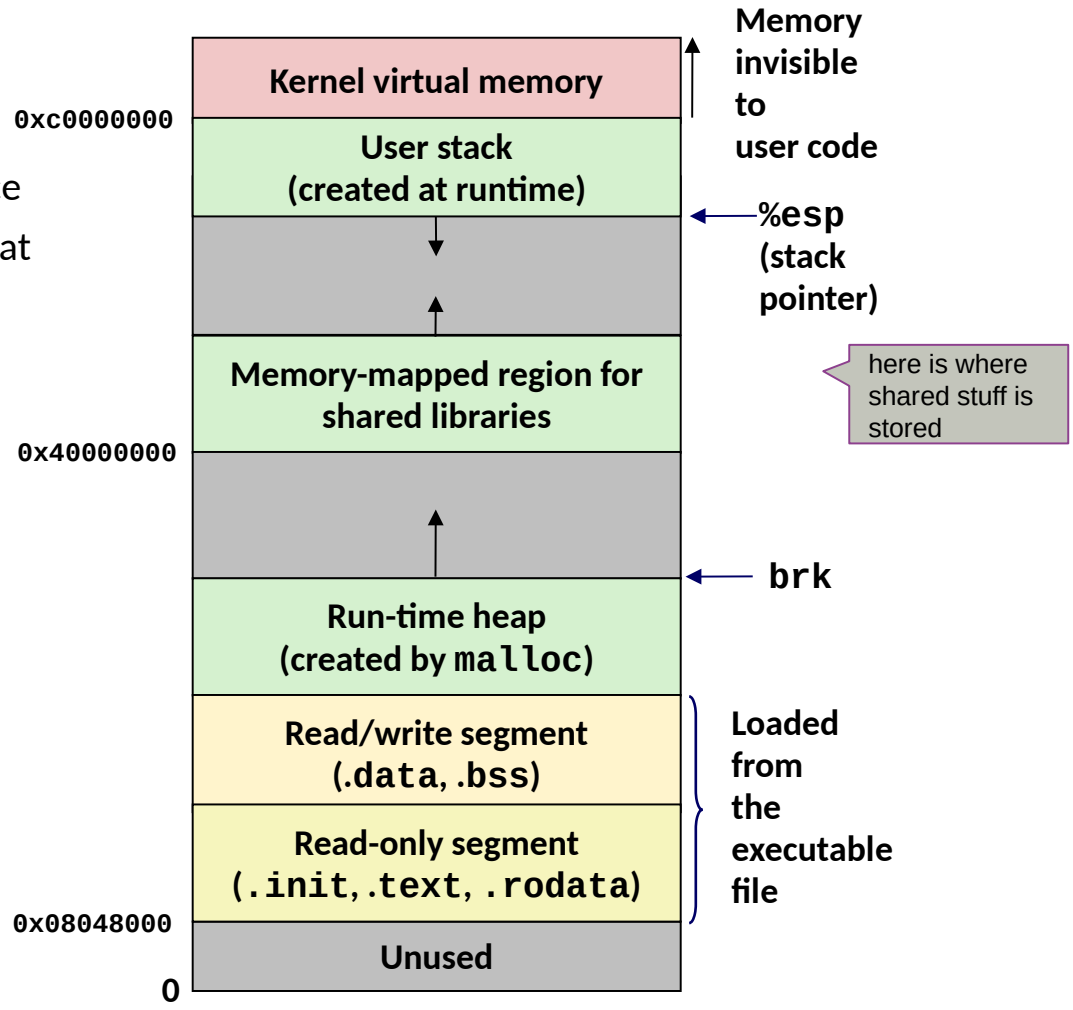


Simplifying Linking and Loading

Linking

Each program has similar virtual address space

Code, stack, and shared libraries always start at the same address

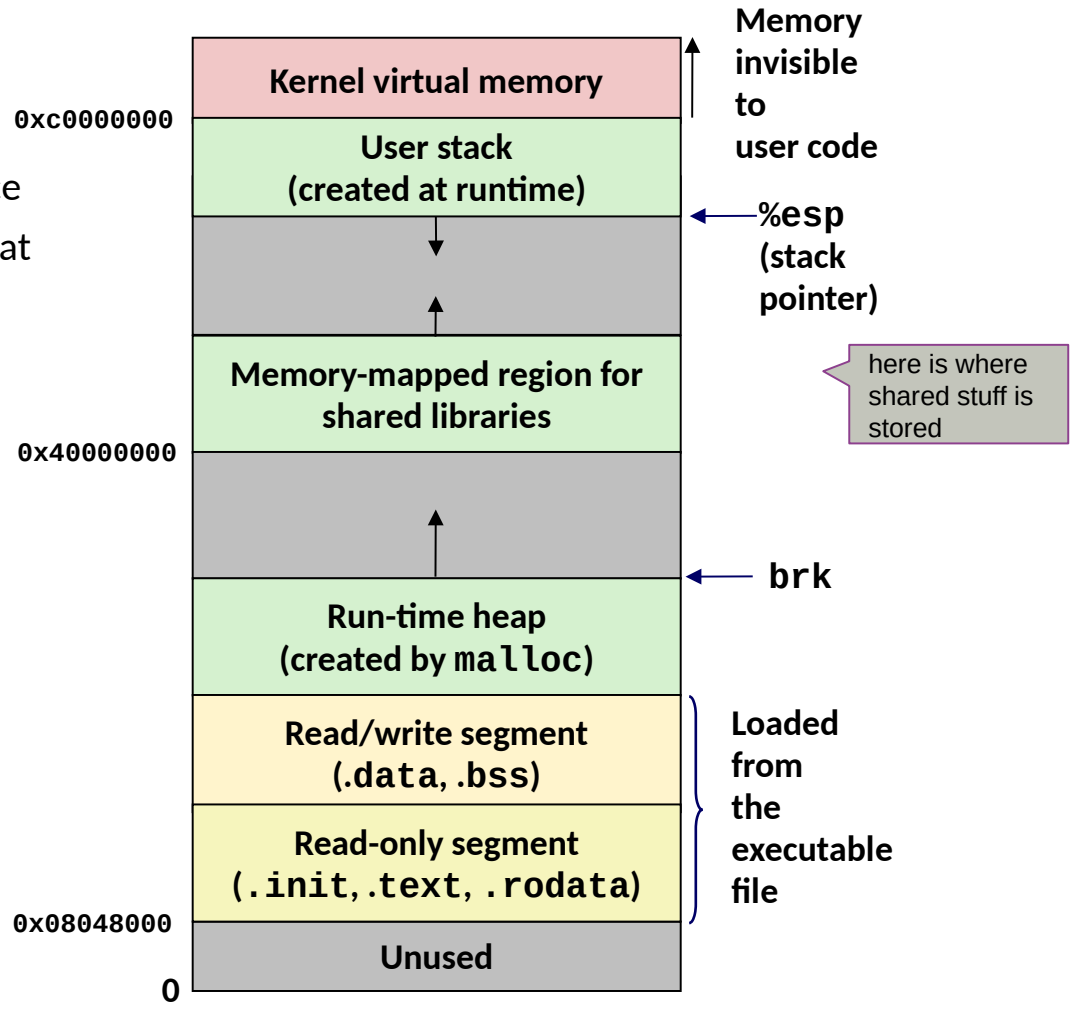


Simplifying Linking and Loading

Linking

Each program has similar virtual address space

Code, stack, and shared libraries always start at the same address



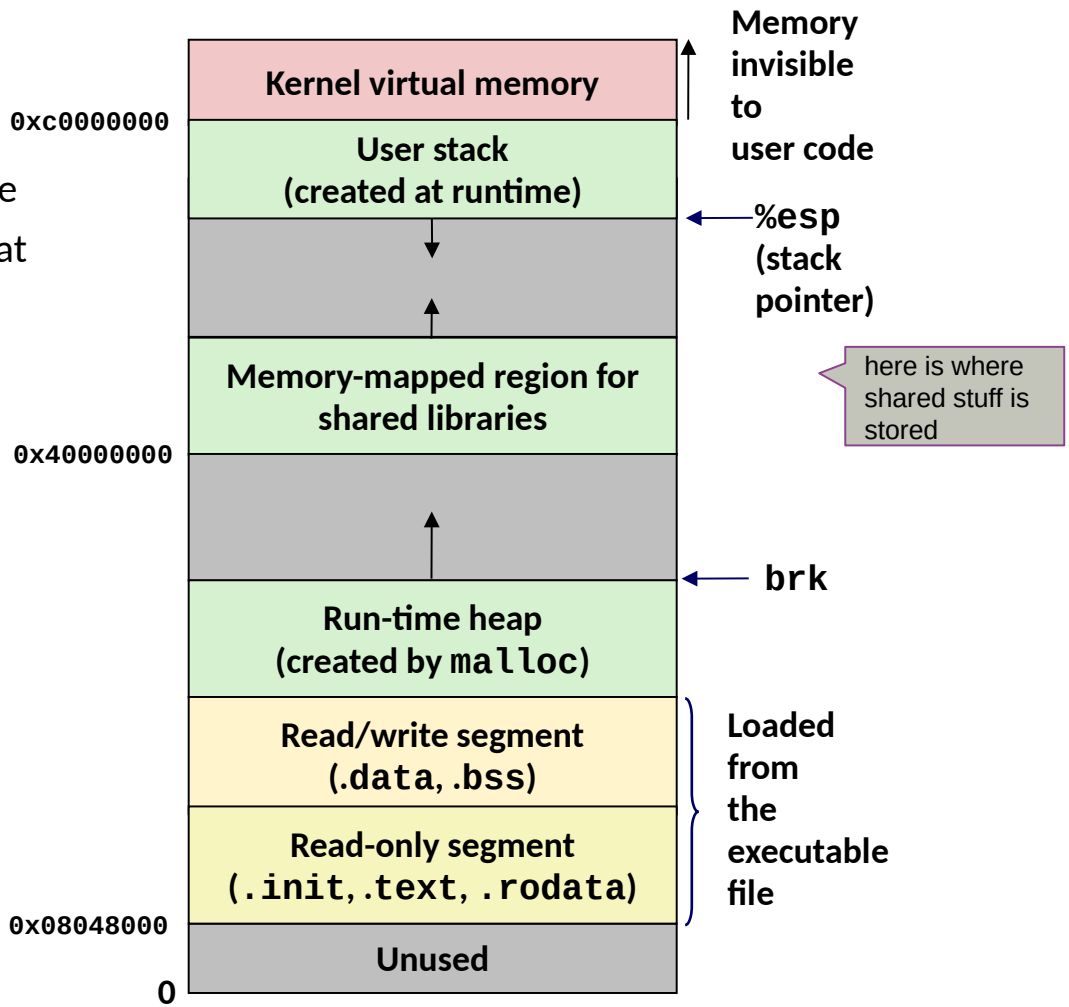
Simplifying Linking and Loading

Linking

Each program has similar virtual address space

Code, stack, and shared libraries always start at the same address

Loading



Simplifying Linking and Loading

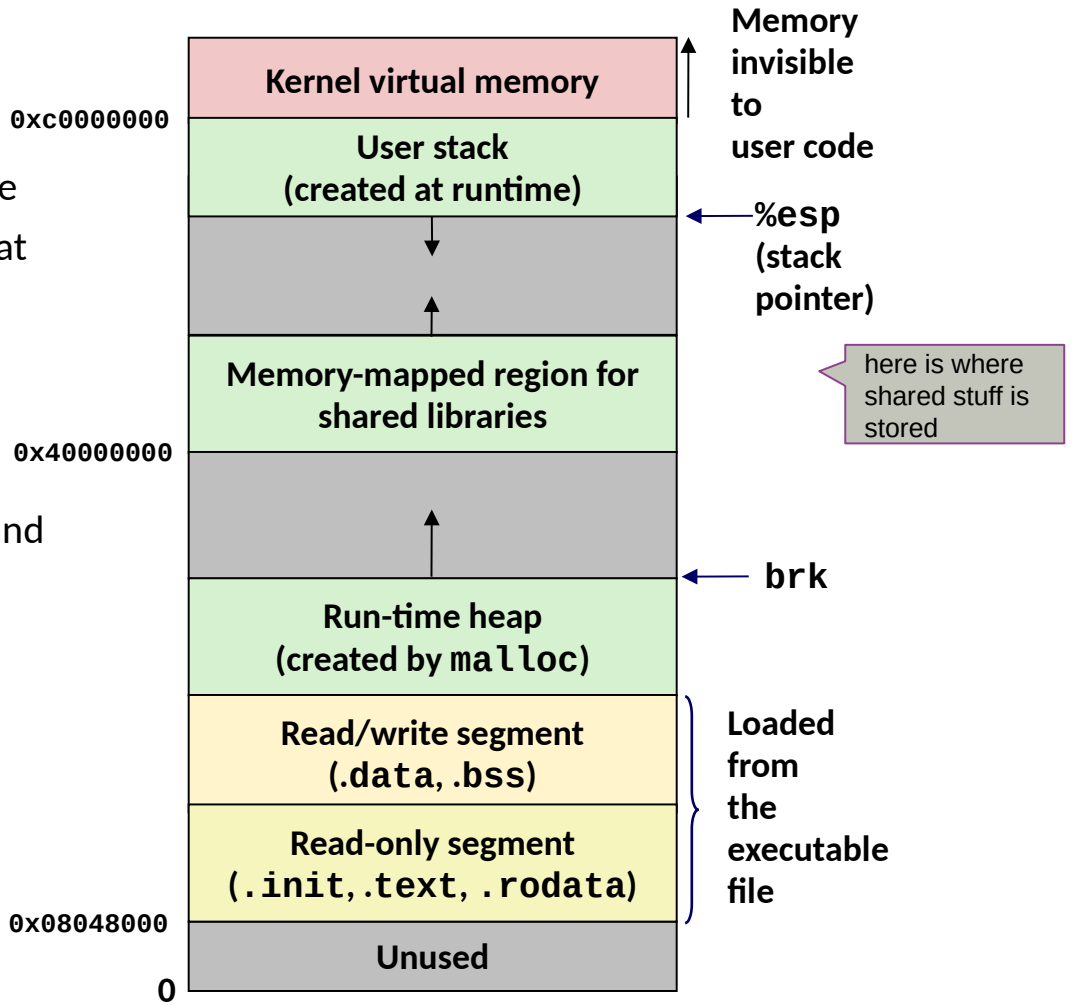
Linking

Each program has similar virtual address space

Code, stack, and shared libraries always start at the same address

Loading

`execve()` allocates virtual pages for .text and .data sections
= creates PTEs marked as invalid



Simplifying Linking and Loading

Linking

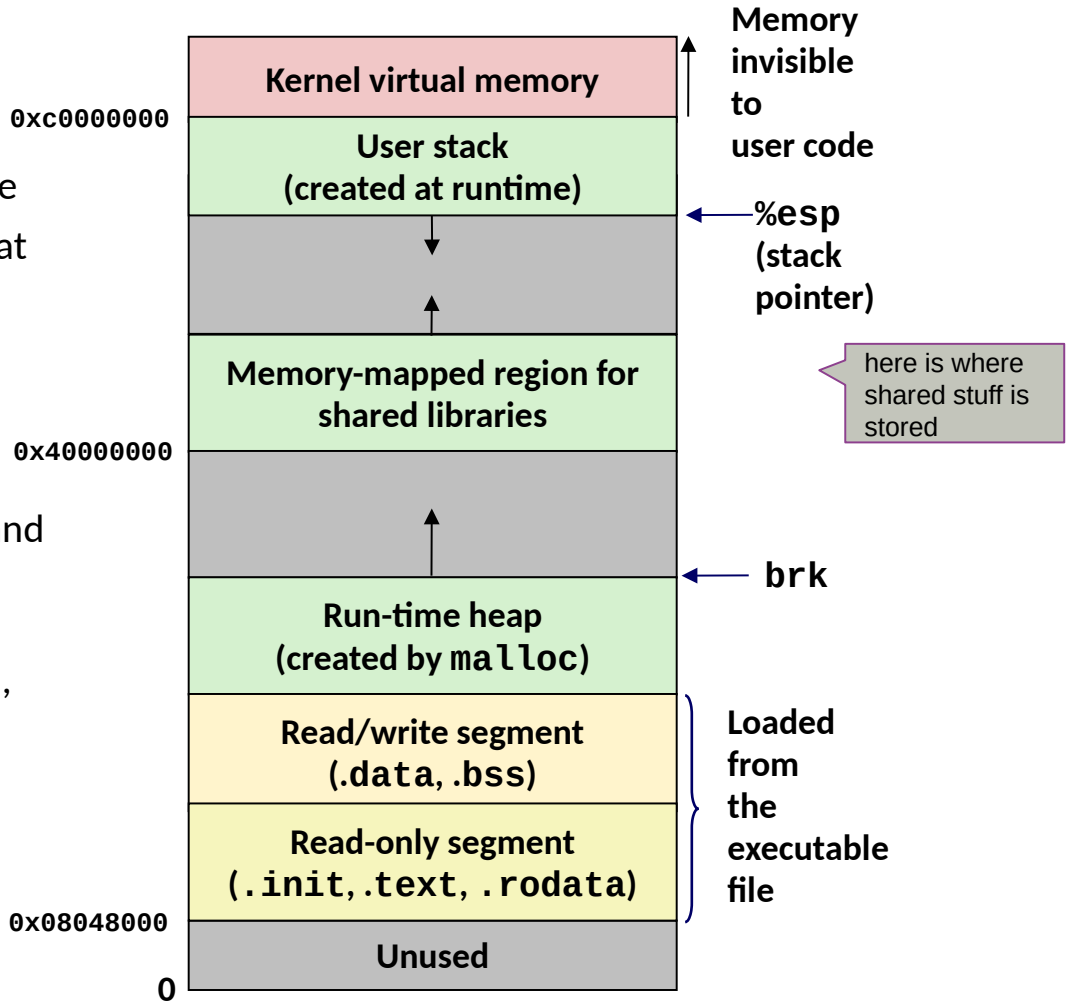
Each program has similar virtual address space

Code, stack, and shared libraries always start at the same address

Loading

`execve()` allocates virtual pages for .text and .data sections
= creates PTEs marked as invalid

The **`.text`** and **`.data`** sections are copied, page by page, on demand by the virtual memory system



Simplifying Linking and Loading

Linking

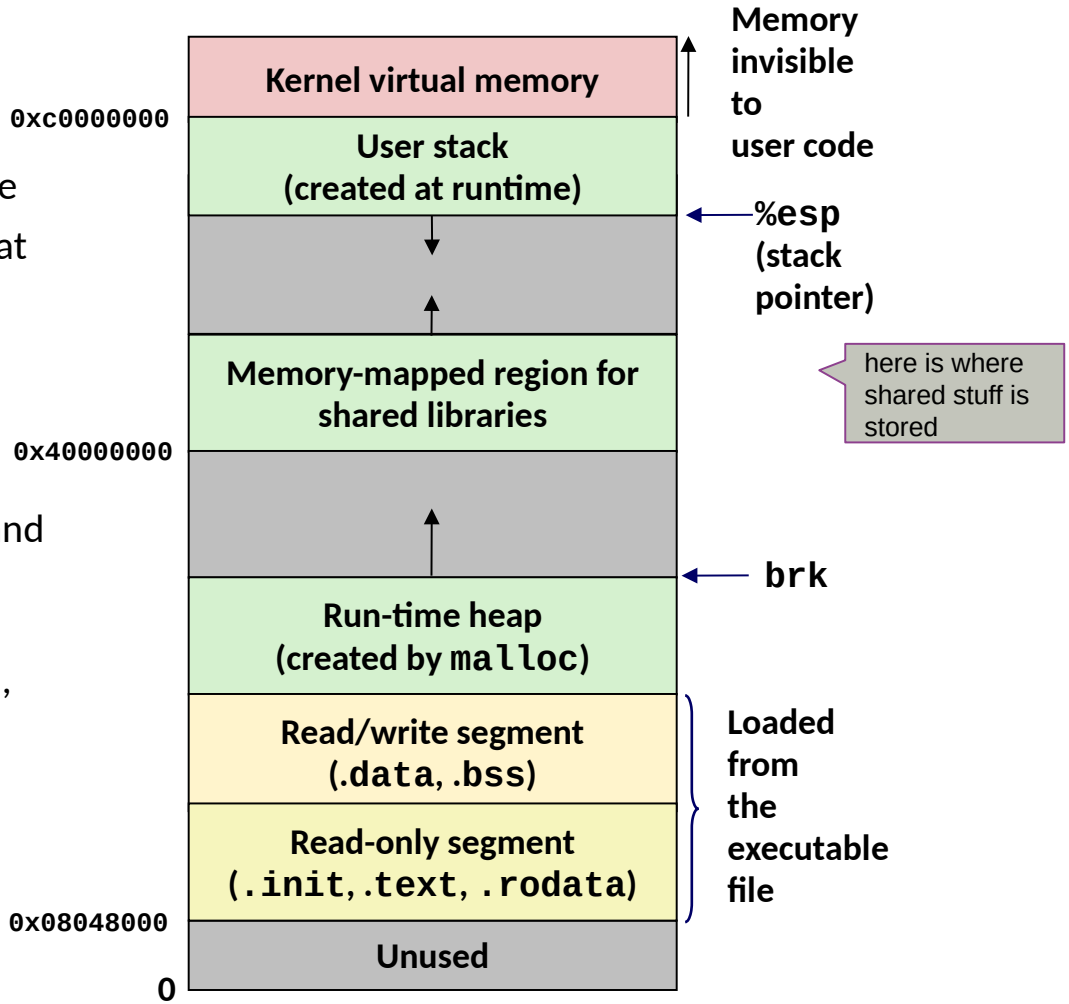
Each program has similar virtual address space

Code, stack, and shared libraries always start at the same address

Loading

`execve()` allocates virtual pages for .text and .data sections
= creates PTEs marked as invalid

The **`.text`** and **`.data`** sections are copied, page by page, on demand by the virtual memory system



VM as a Tool for Memory Protection

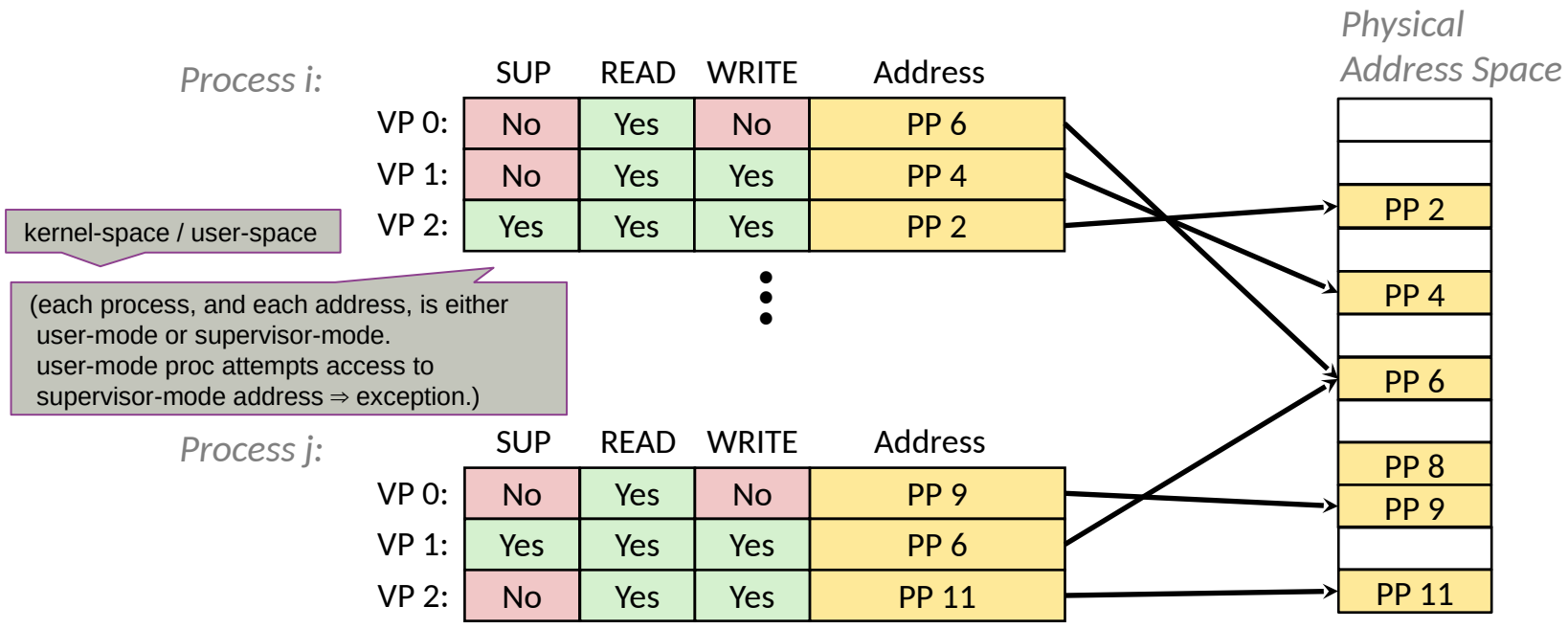
page table entries

Extend PTEs with permission bits

(virt into phys)

Page fault handler checks these bits before remapping

If violated, send process SIGSEGV (segmentation fault)



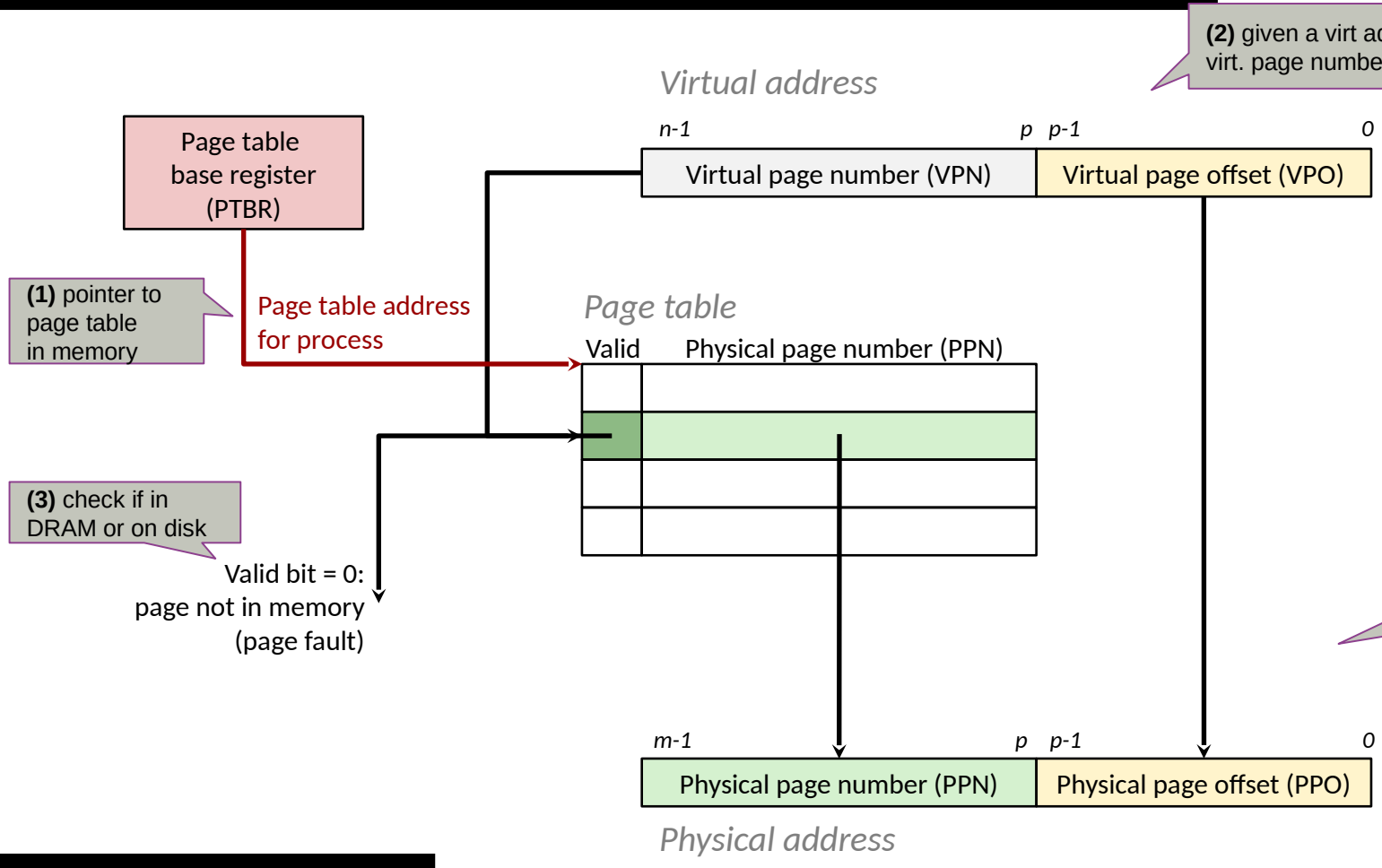
kernel-space / user-space

(each process, and each address, is either user-mode or supervisor-mode. user-mode proc attempts access to supervisor-mode address ⇒ exception.)

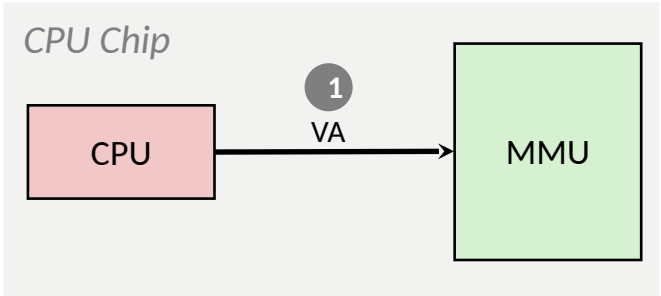
now MMU can check whether proc has access to region in mem!

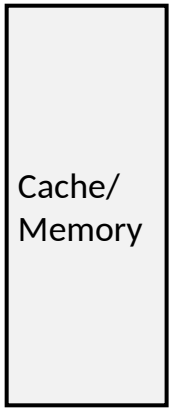
Address Translation With a Page Table

example now where we go through the whole translation.



Address Translation: Page Hit





1) Processor sends virtual address to MMU

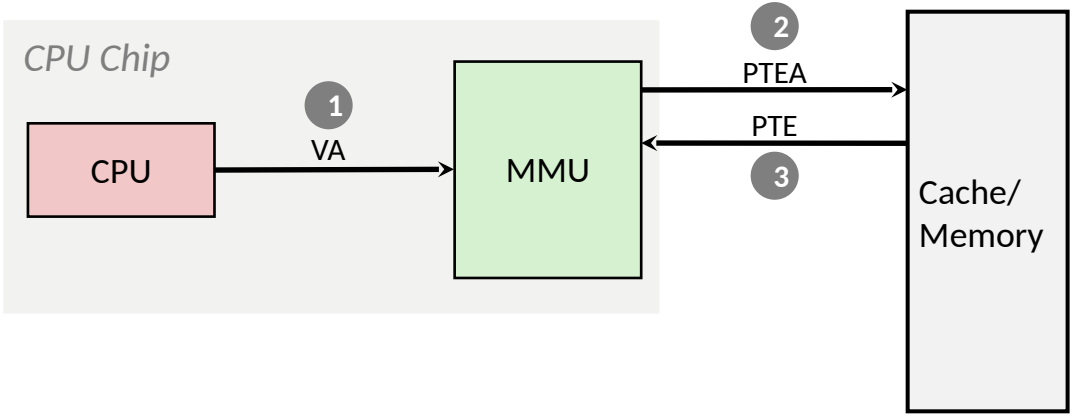
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

very flexible mechanism!
but, if we are not careful, then we have 2 accesses to DRAM for each VA.
(how to optimize that? hold that thought)

Address Translation: Page Hit



1) Processor sends virtual address to MMU

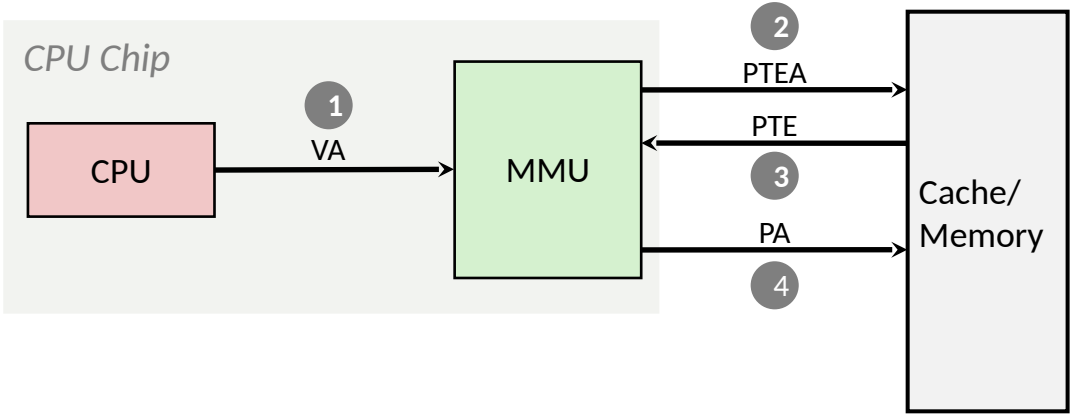
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

very flexible mechanism!
but, if we are not careful, then we have 2 accesses to DRAM for each VA.
(how to optimize that? hold that thought)

Address Translation: Page Hit



1) Processor sends virtual address to MMU

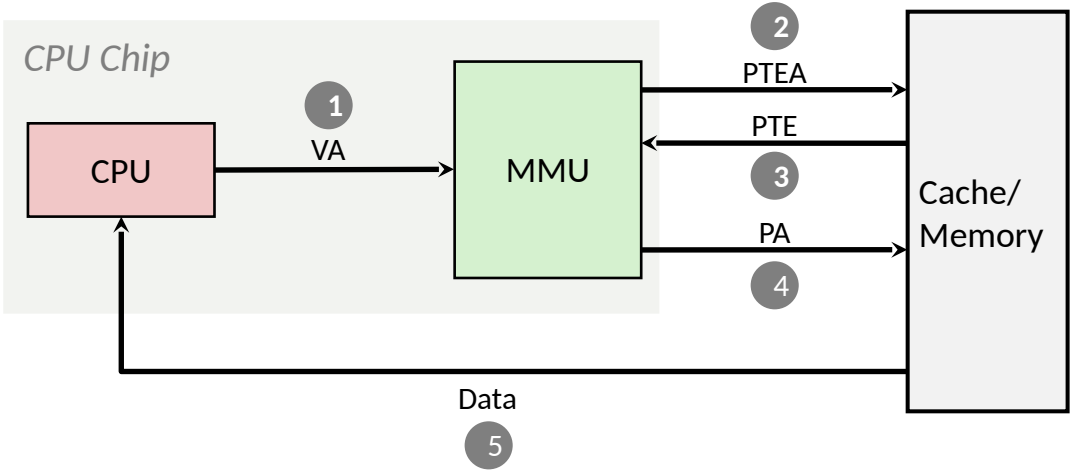
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

very flexible mechanism!
but, if we are not careful, then we have 2 accesses to DRAM for each VA.
(how to optimize that? hold that thought)

Address Translation: Page Hit



1) Processor sends virtual address to MMU

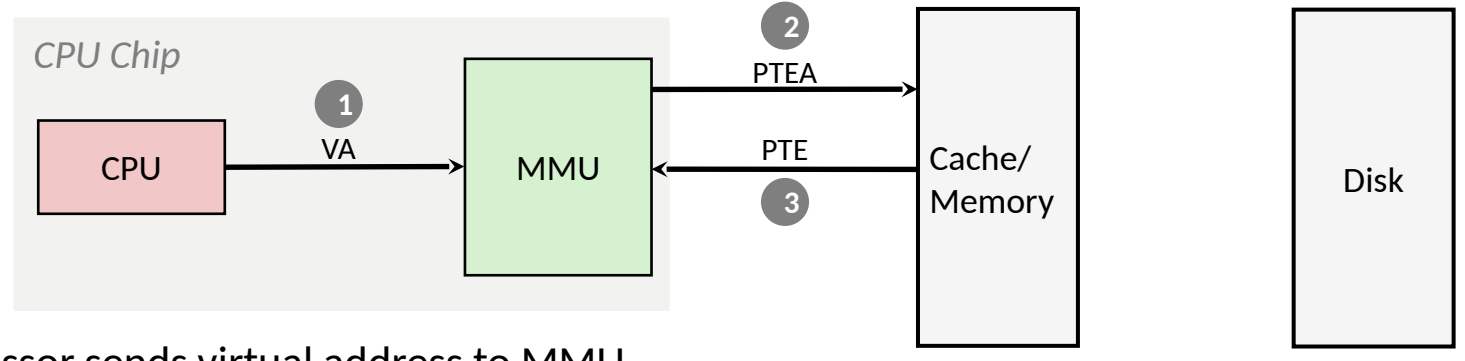
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

very flexible mechanism!
but, if we are not careful, then we have 2 accesses to DRAM for each VA.
(how to optimize that? hold that thought)

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 (page not in DRAM)

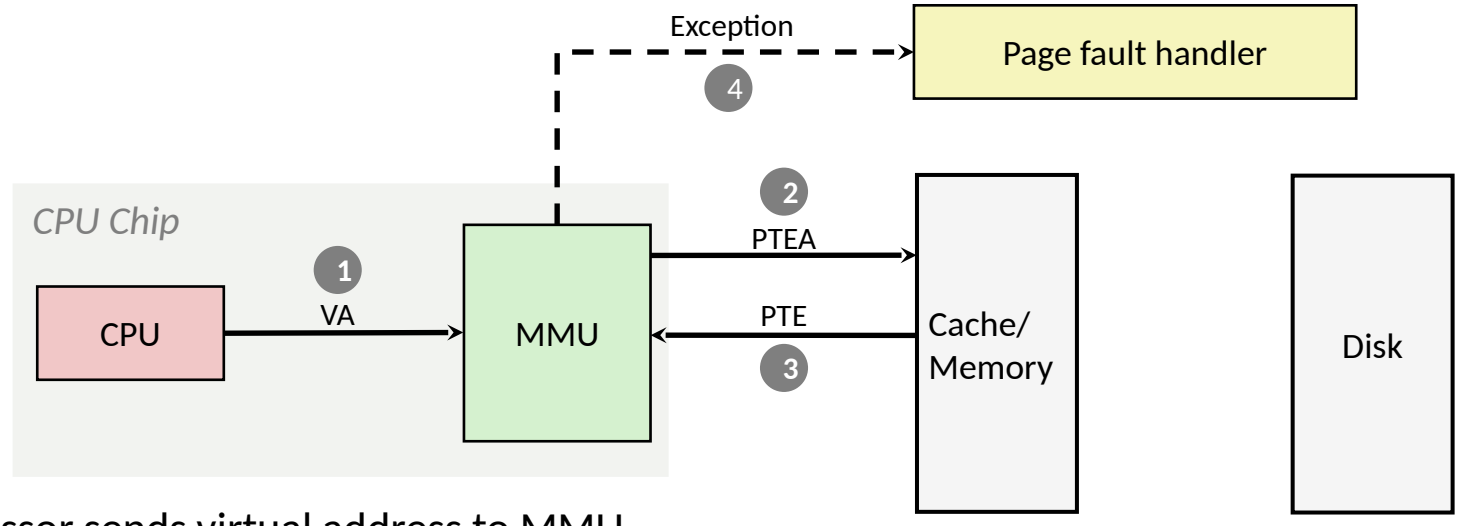
5) Handler identifies victim (and, if dirty, pages it out to disk)

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

7) Handler returns to original process, restarting faulting instruction

costly operation

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 (page not in DRAM)

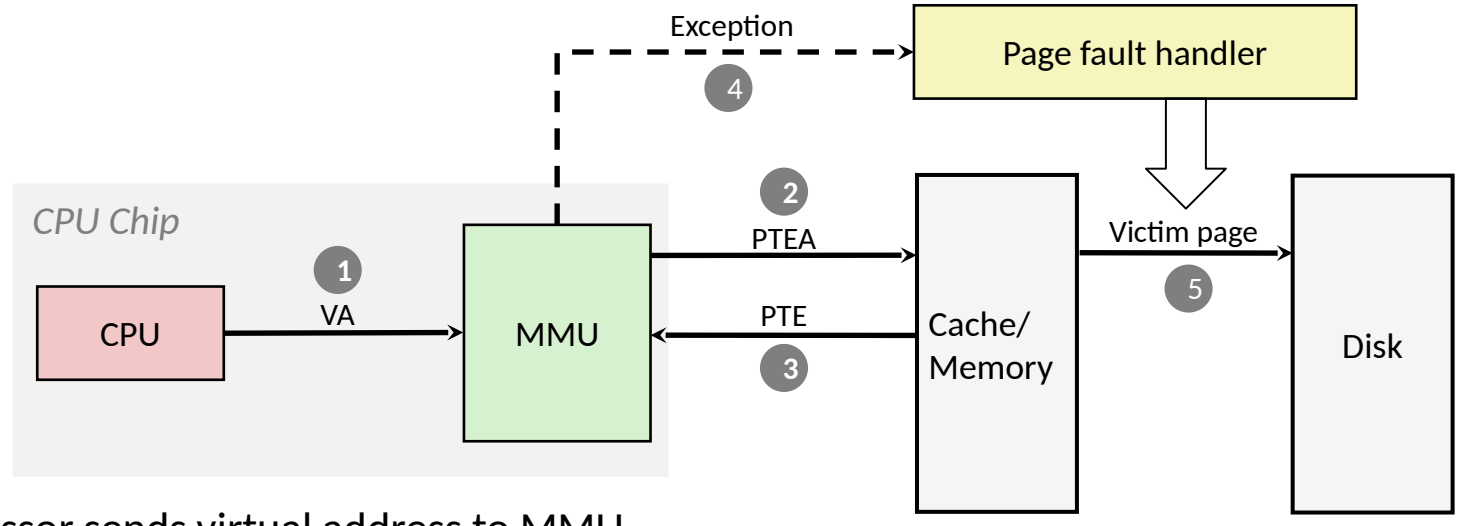
5) Handler identifies victim (and, if dirty, pages it out to disk)

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

7) Handler returns to original process, restarting faulting instruction

costly operation

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 (page not in DRAM)

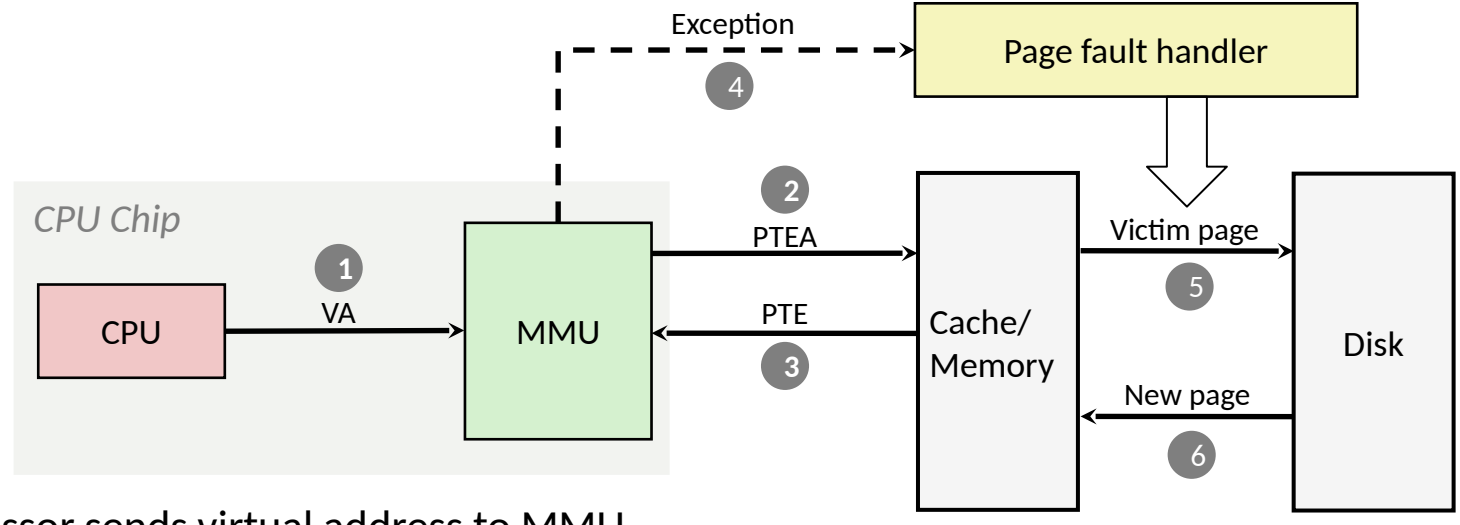
5) Handler identifies victim (and, if dirty, pages it out to disk)

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

7) Handler returns to original process, restarting faulting instruction

costly operation

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 (page not in DRAM)

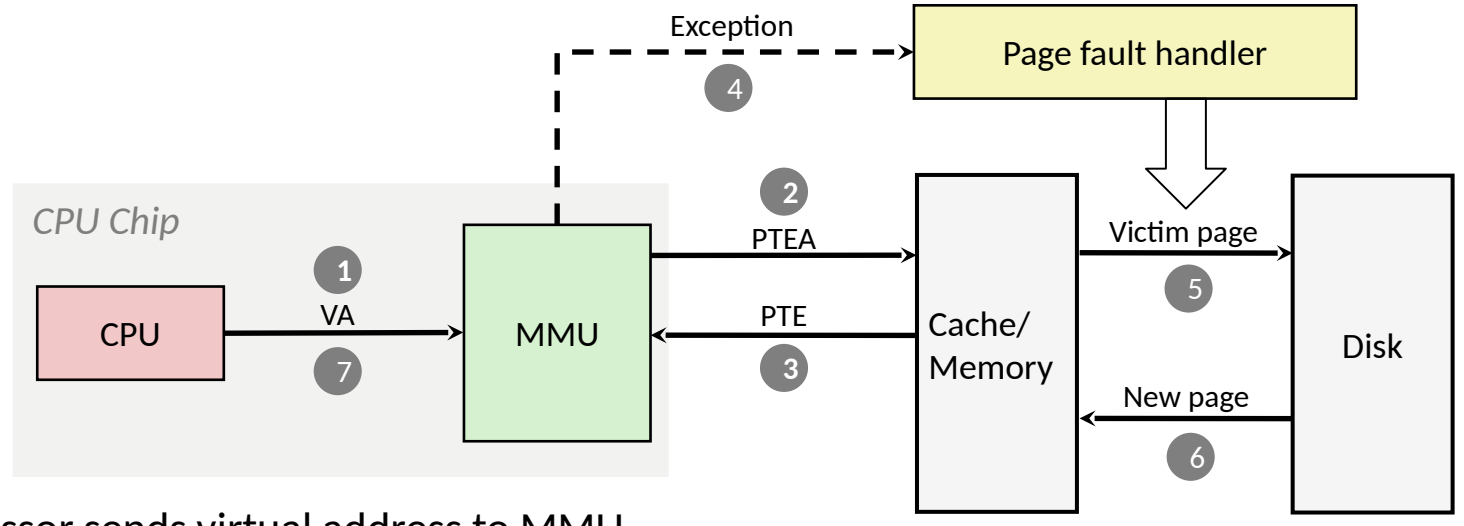
5) Handler identifies victim (and, if dirty, pages it out to disk)

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

7) Handler returns to original process, restarting faulting instruction

costly operation

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 (page not in DRAM)

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

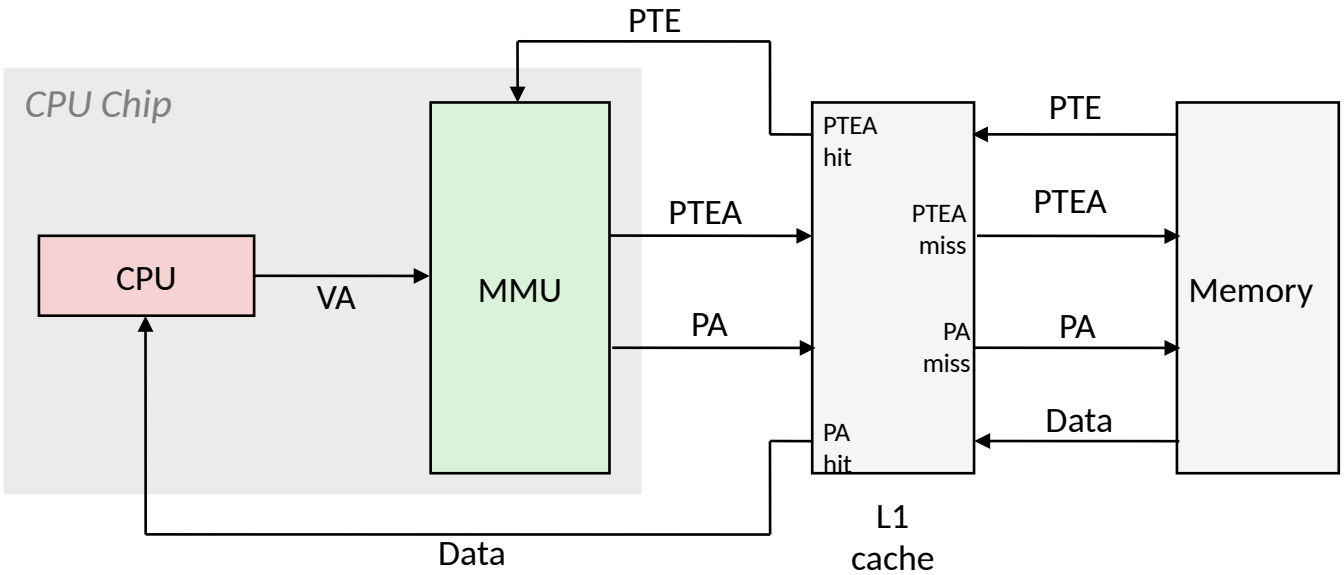
6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

costly operation

Integrating VM and Cache

reality is different
between CPU and DRAM,
we have caches.



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

the further from CPU,
the more CPU must wait
(nanoseconds to 100s of microseconds)

Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay.
- Solution:

we don't want to pollute the L1 cache w/ PTEs

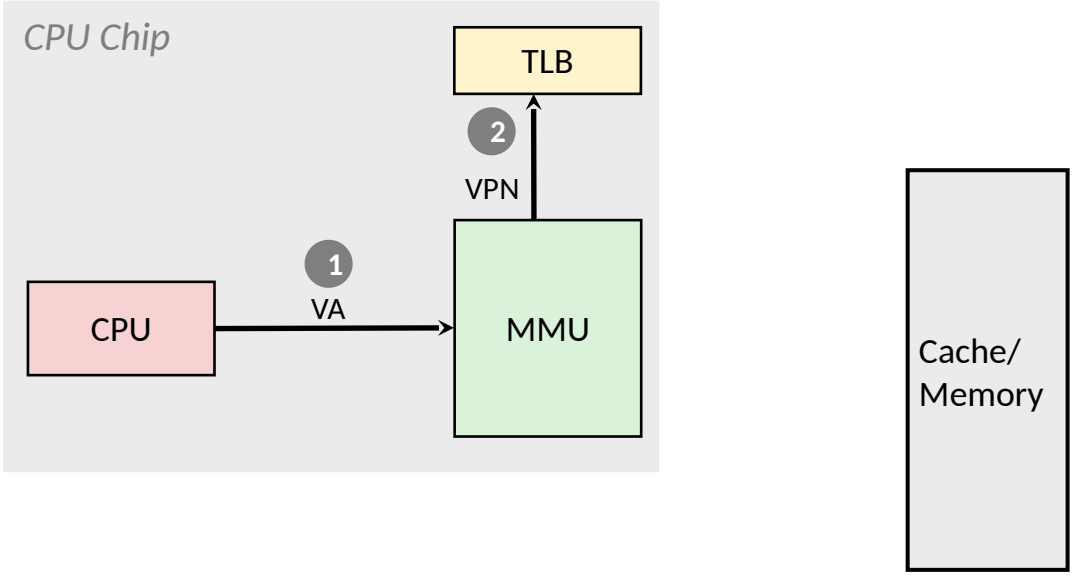
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay.

we don't want to pollute the L1 cache w/ PTEs

- Solution: *Translation Lookaside Buffer* (TLB)
  - Small hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages

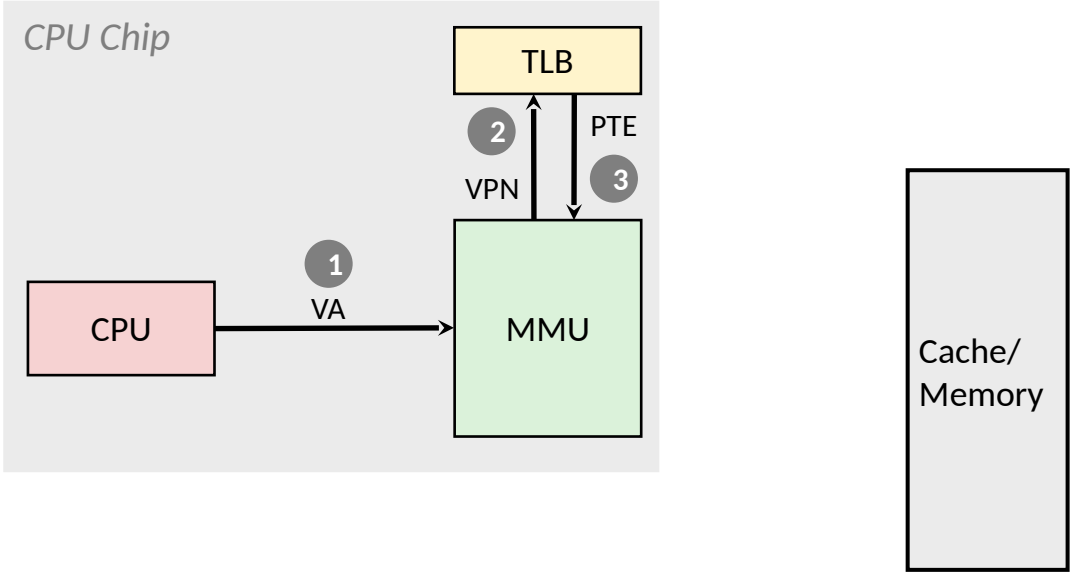
TLB Hit



A TLB hit eliminates a memory access

nice and fast :-)

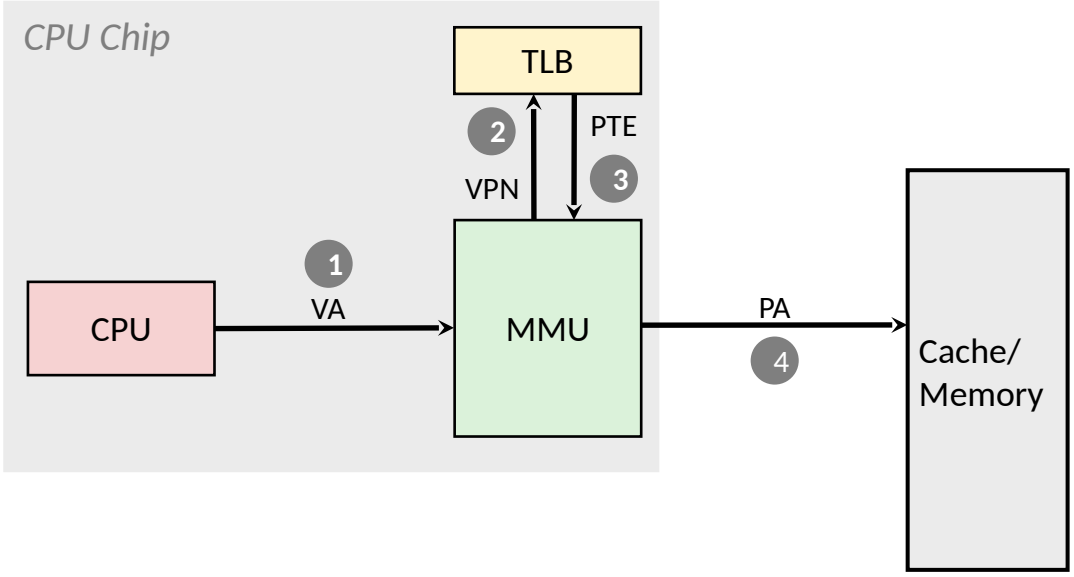
TLB Hit



A TLB hit eliminates a memory access

nice and fast :-)

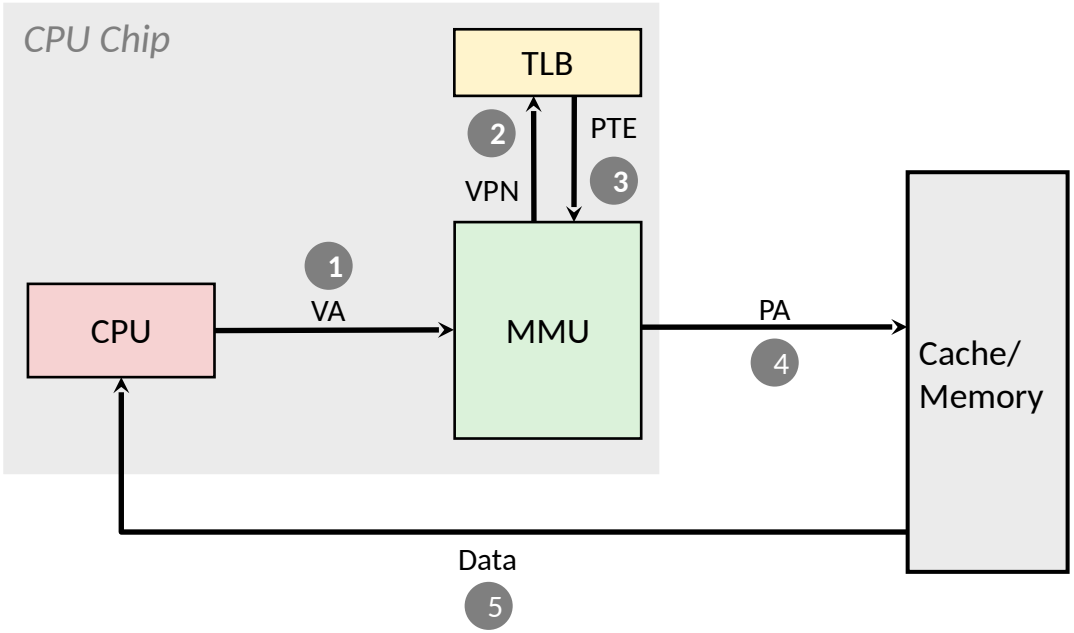
TLB Hit



A TLB hit eliminates a memory access

nice and fast :-)

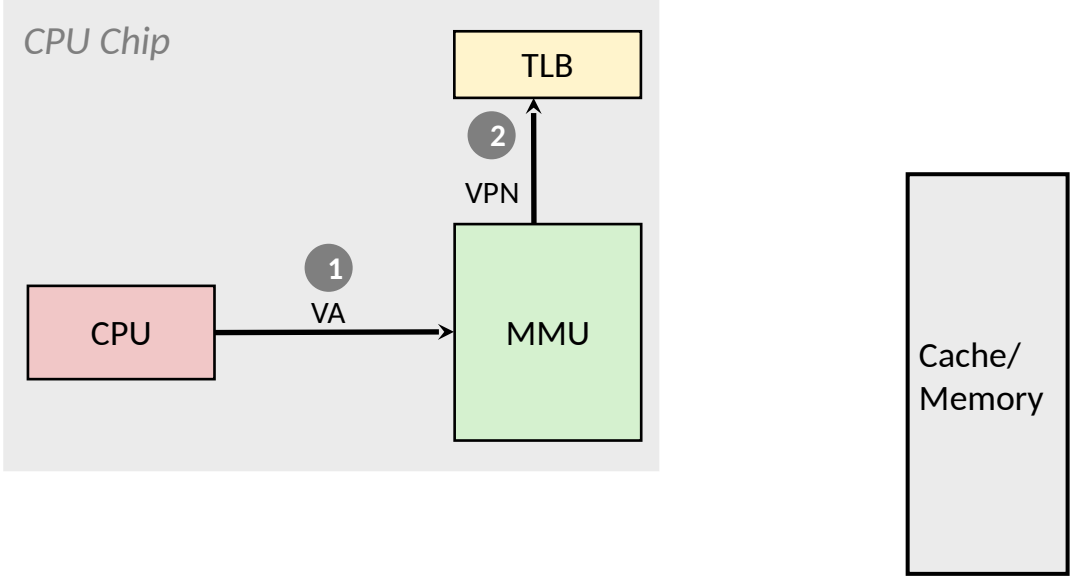
TLB Hit



A TLB hit eliminates a memory access

nice and fast :-)

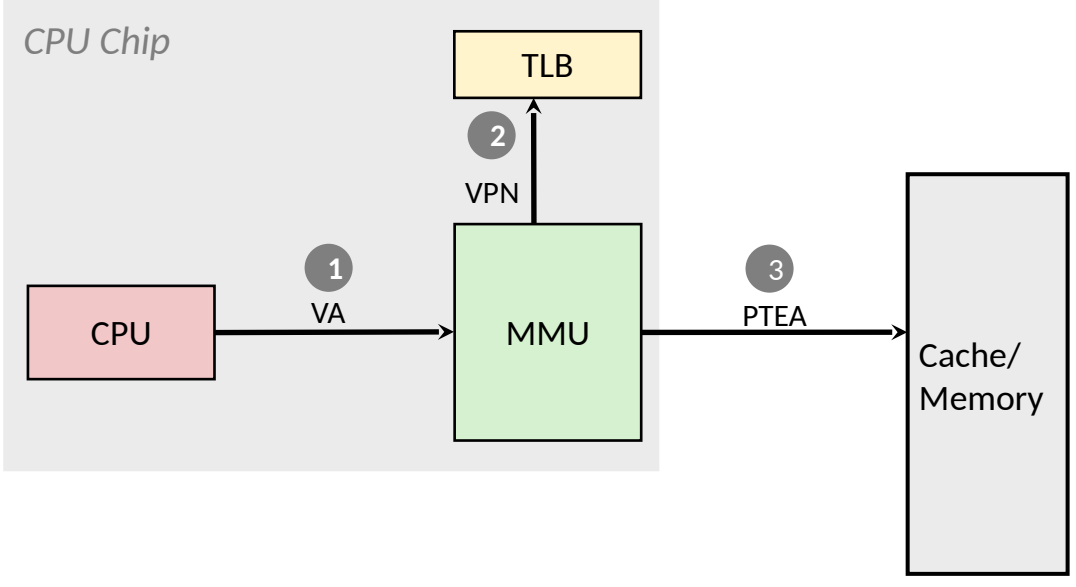
TLB Miss



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

Fortunately, TLB misses are rare. Why?

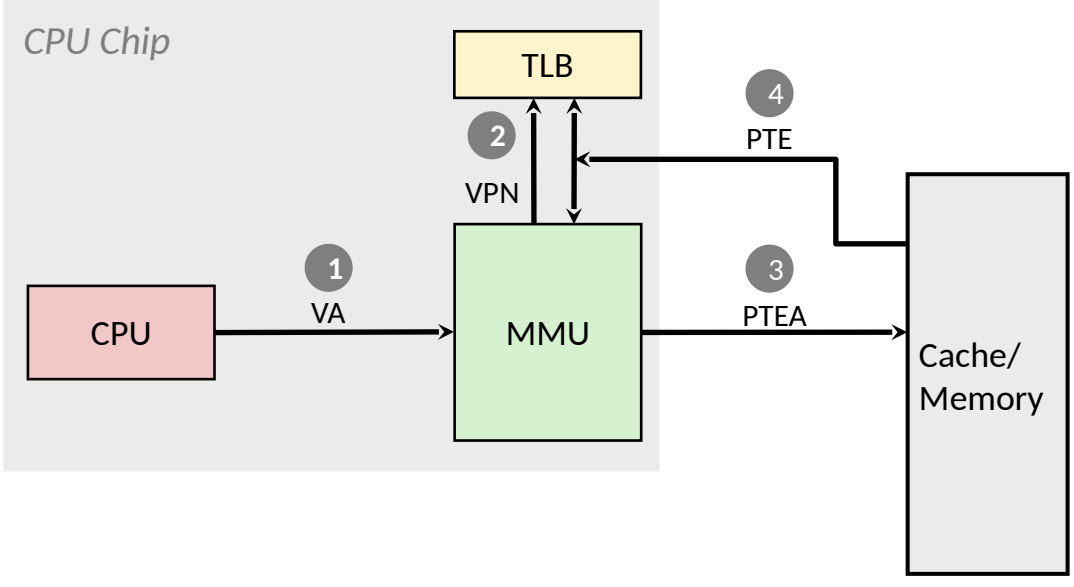
TLB Miss



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

Fortunately, TLB misses are rare. Why?

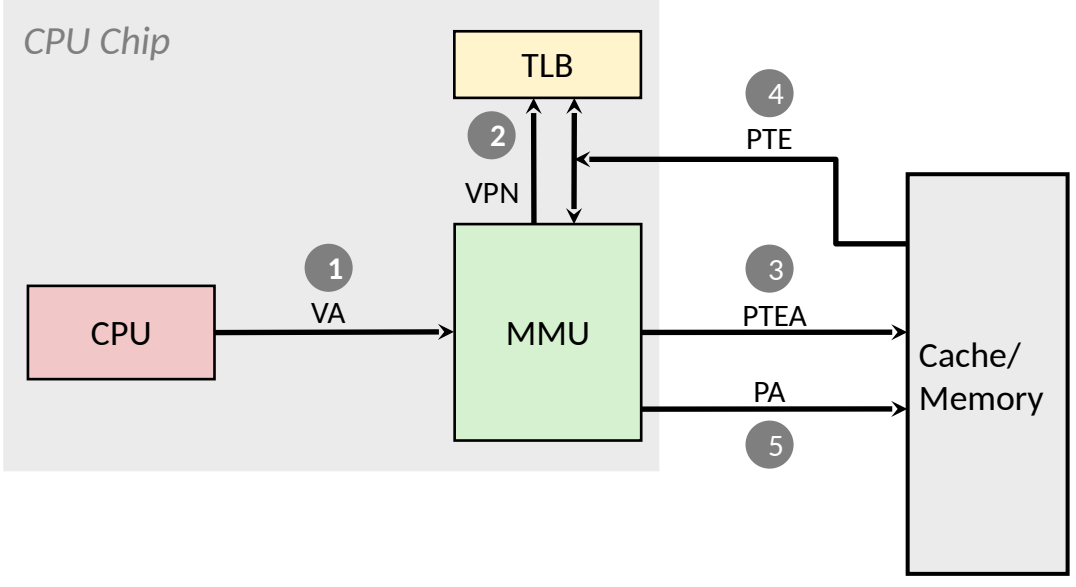
TLB Miss



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

Fortunately, TLB misses are rare. Why?

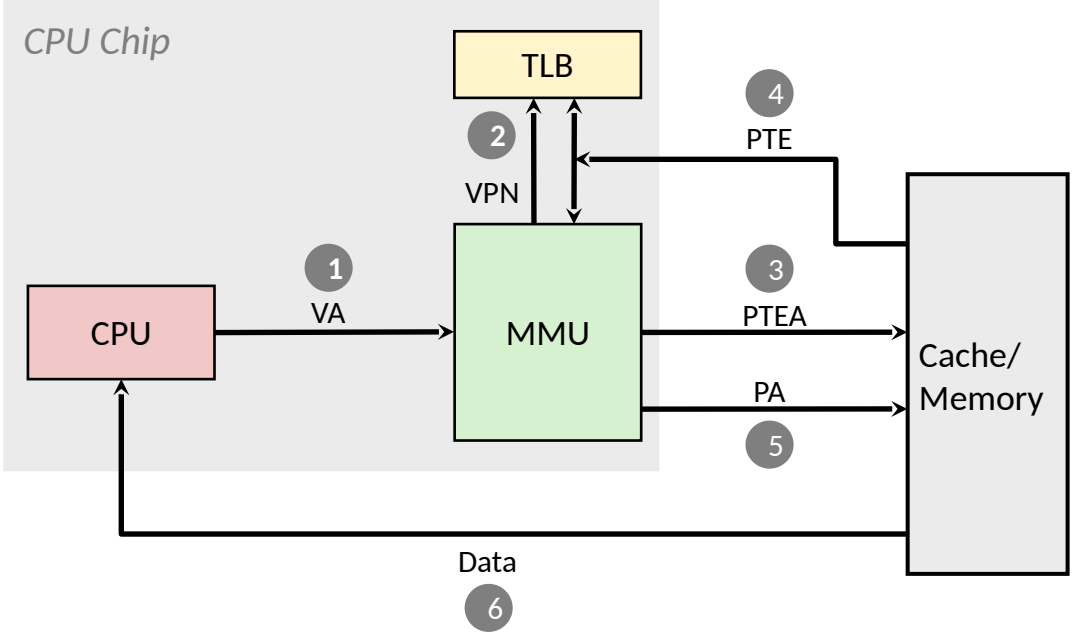
TLB Miss



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

Fortunately, TLB misses are rare. Why?

TLB Miss

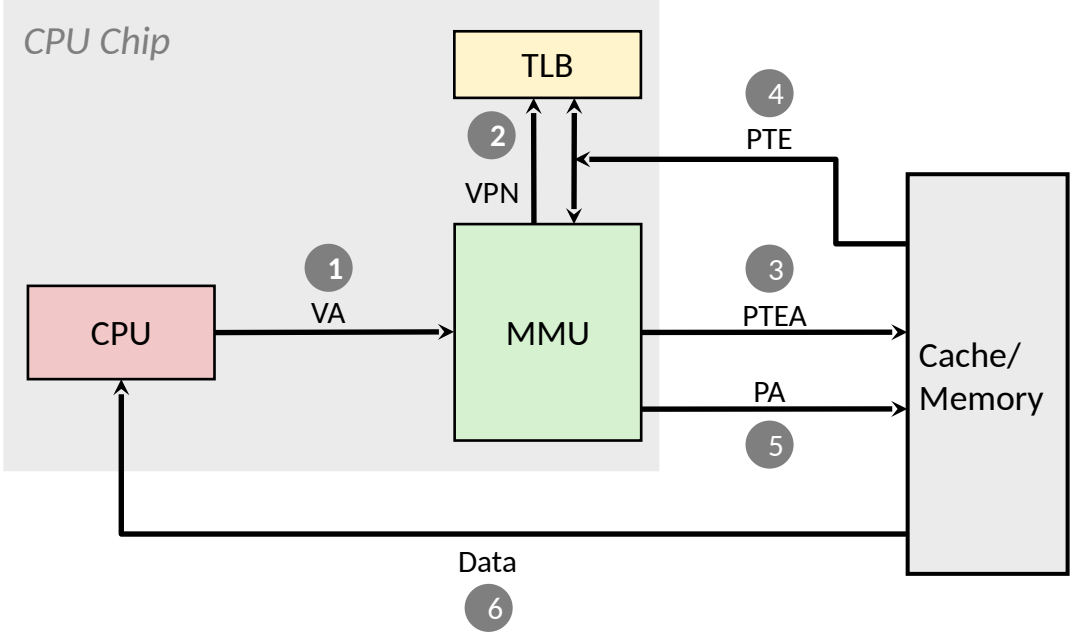


A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

TLB Miss

a way to quantify performance of your program is to monitor nr. of TLB misses. add locality to reduce.



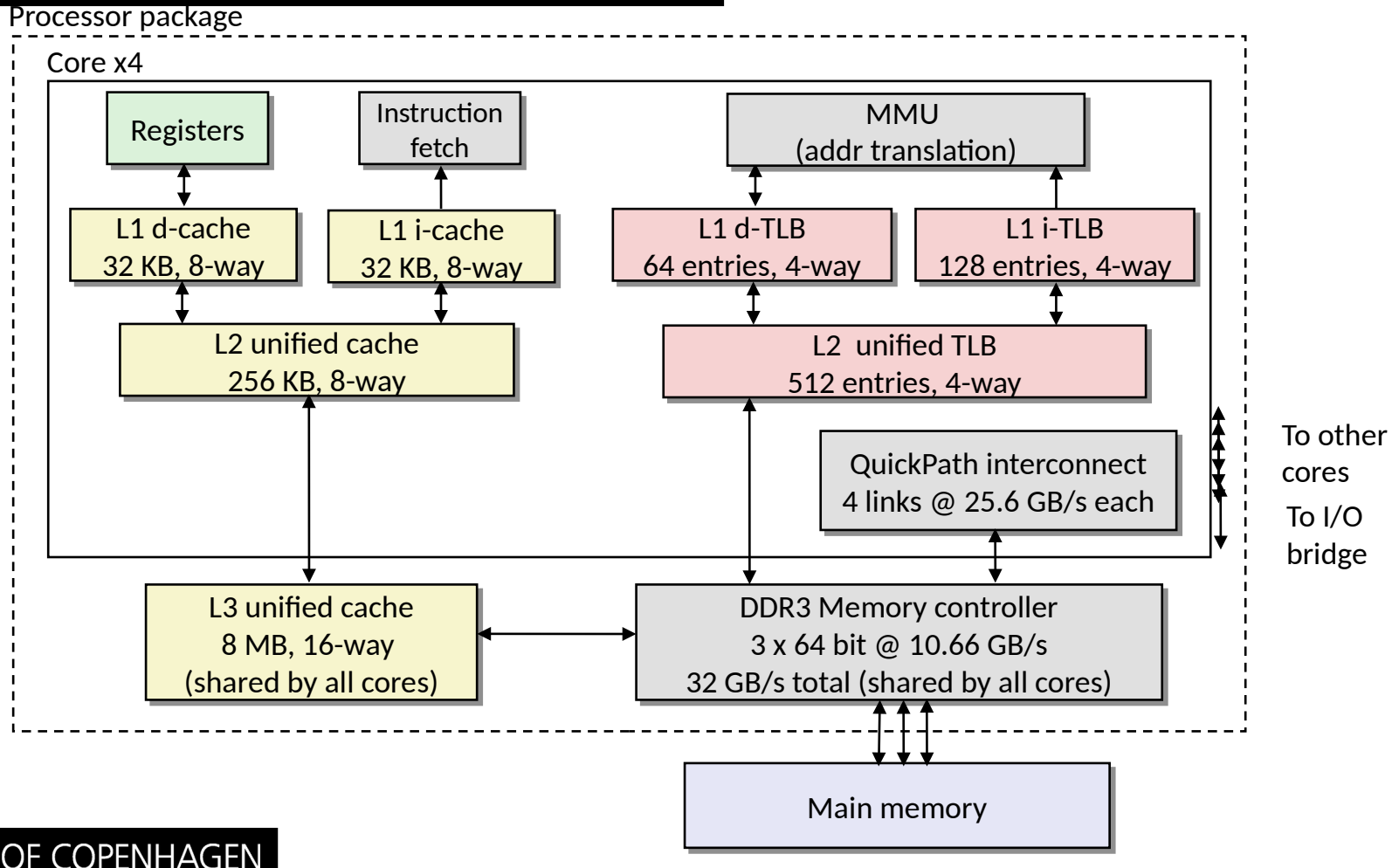
A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?

TLB misses are rare! why: **locality.**

Intel Core i7 Memory System

concrete example (20s)



Take-Aways

You should be able to describe:

- Address Space
- Physical and Virtual Memory
- MMU (its role, how work split between it and OS)
- Pages
- Page Table
- Translation Lookaside Buffer (TLB)

Virtual Memory as a tool for caching, memory management and memory protection.

remaining lectures not about assignments, or programming (you should be done w/ LCTHW). but related to what I can, and will, ask at the exam.

performance and **security**!