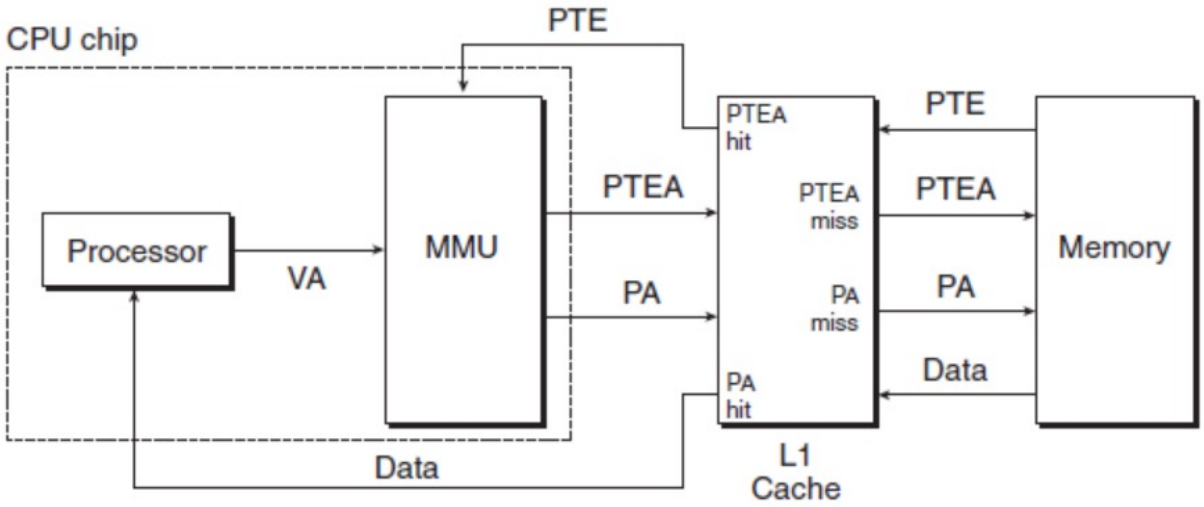
CSE320 System Fundamentals II Virtual Memory 2

YOUNGMIN KWON / TONY MIONE

Integrating Caches and VM



Whether to use Virtual or Physical addresses to access the SRAM cache?

- Most system opt for physical addresses
- Easy for multiple processes to have blocks in the cache
- No need to deal with the memory protection

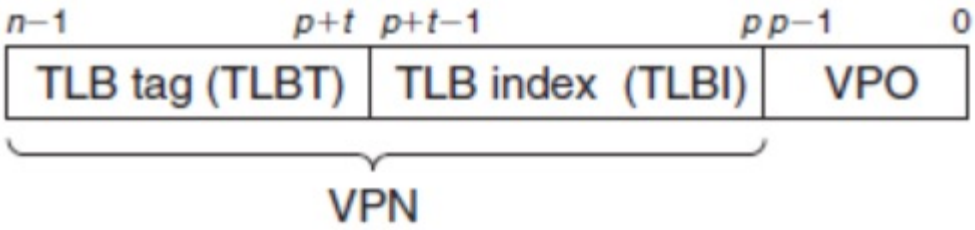
Translation Lookaside Buffer (TLB)

A small cache of PTEs in MMU

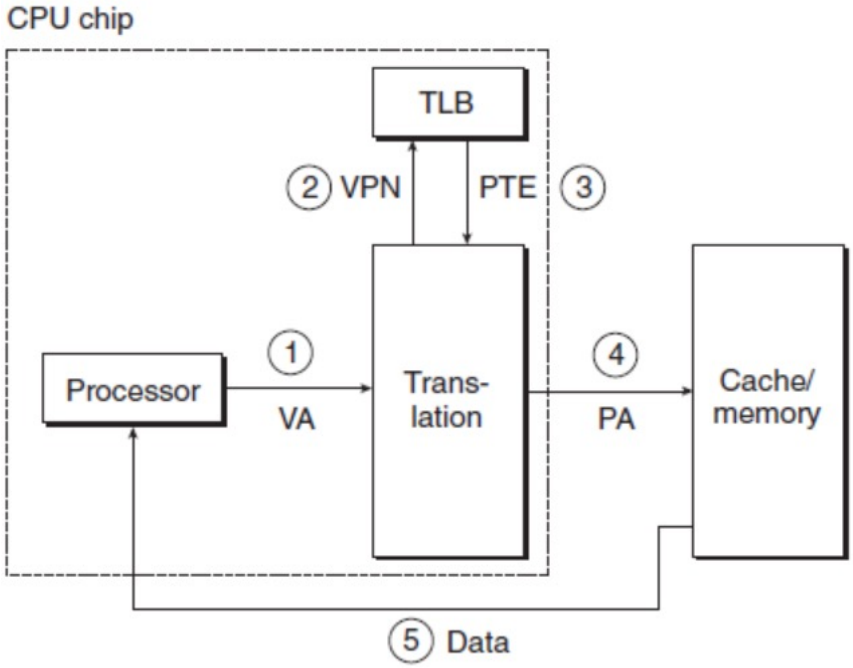
- Each line holds a block consisting of a single PTE

If a TLB has $T=2^t$ sets,

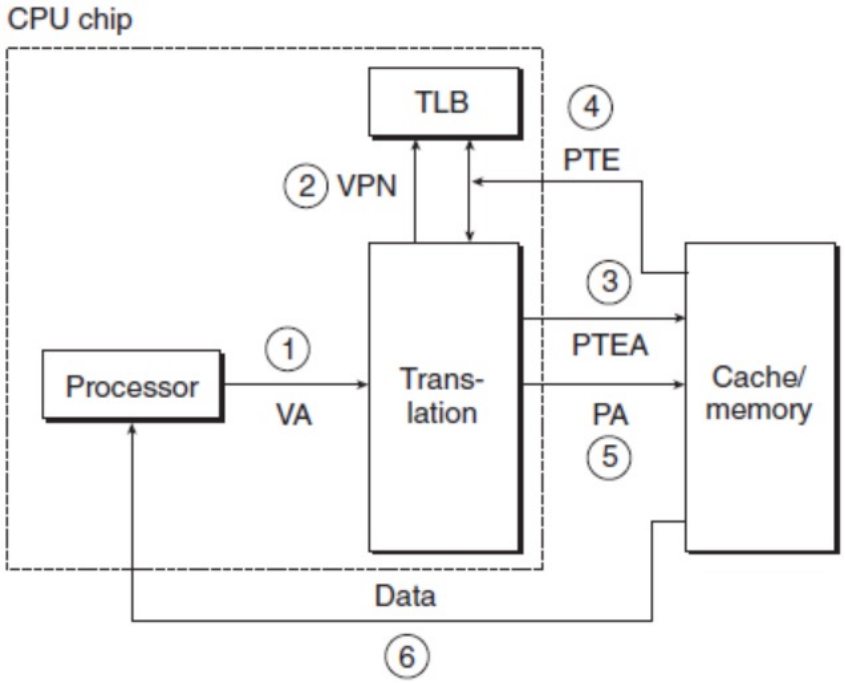
- TLB index (TLBI) consists of the *t* least significant bits of the VPN
- TLB tag (TLBT) consists of the remaining bits in VPN



TLB Hit and Miss Operations



TLB Hit



TLB Miss

Multi-Level Page Table

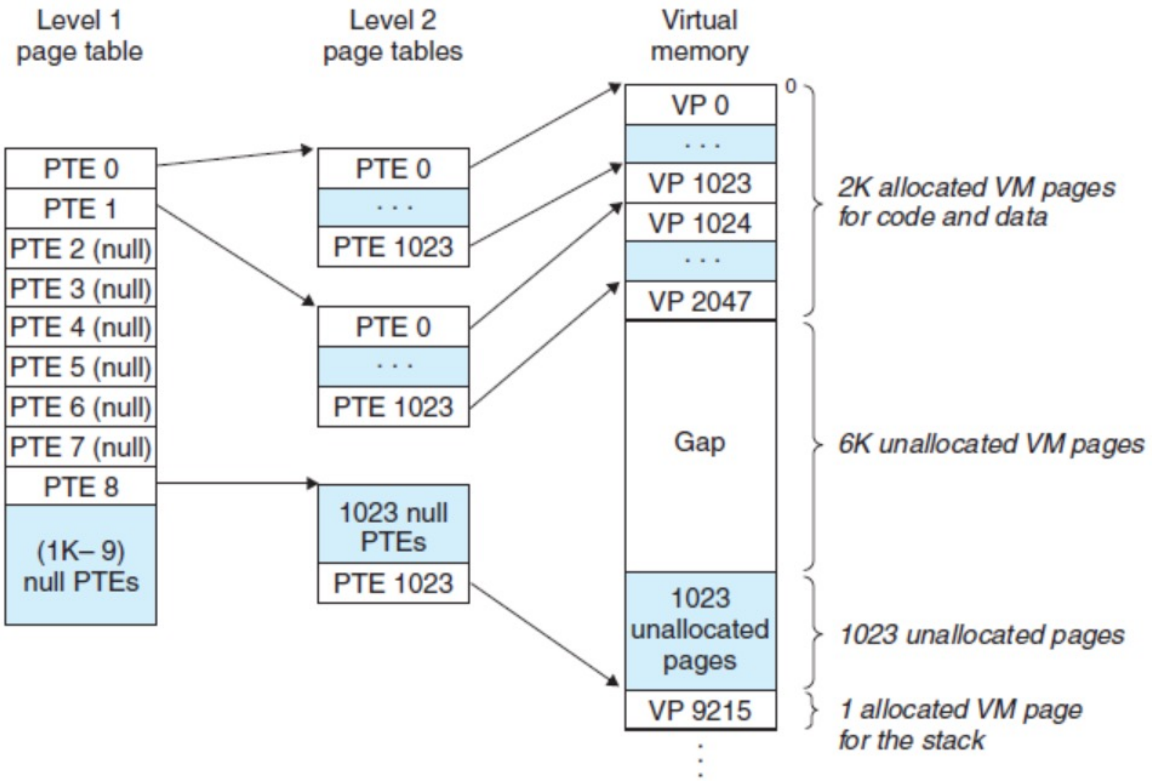
Issue: 32bit address space, 4KB pages, 4MB of PTE

- 4MB page table must reside in memory all the time

Hierarchy of page tables (e.g. 2 level)

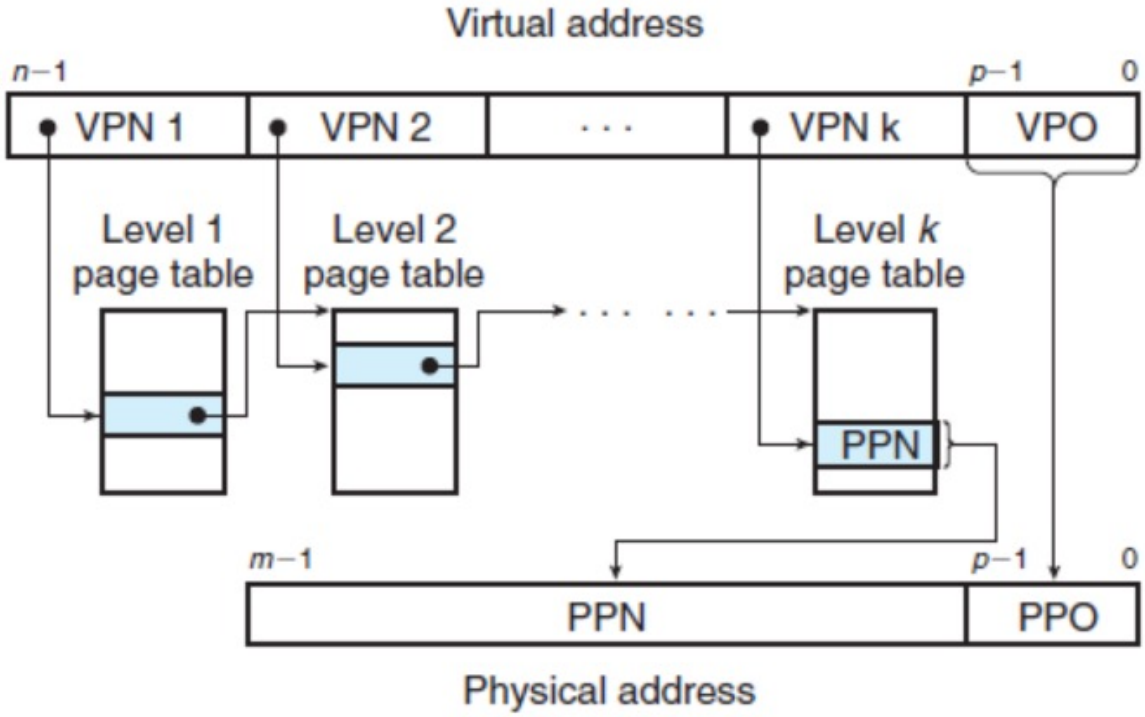
- Level 1 has a page table of 1024 PTEs (4KB)
- Level 2 page tables have 1024 PTEs (4KB) each.
- Each PTE in level 1 is responsible for 4MB chunk of address space
- If every page in chunk *i* is unallocated, PTE *i* in level 1 table is empty
- If at least 1 page in chunk *i* is allocated PTE *i* in level 1 points to the base of level 2 page table

2 Level Page Tables

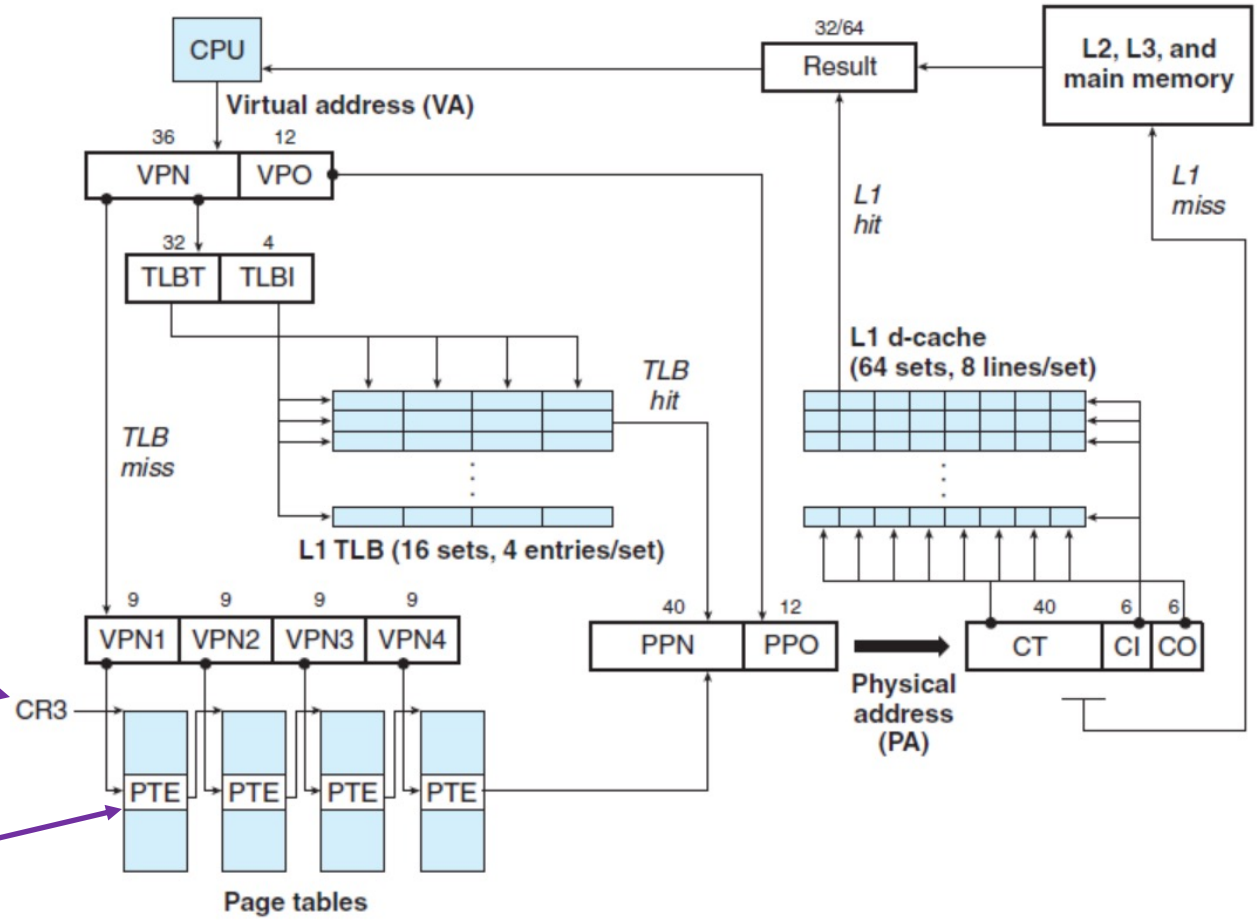


- If PTE in level 1 table is NULL, no need to have a level 2 table in memory
- Only the level 1 table needs to be in memory at all times

k-level Page Tables



Intel Core i7 Memory System



CR3 register: contains the beginning address of the level 1 page table

PTE: R/W bit (read, write), U/S (user, super user), XD (execute disable)

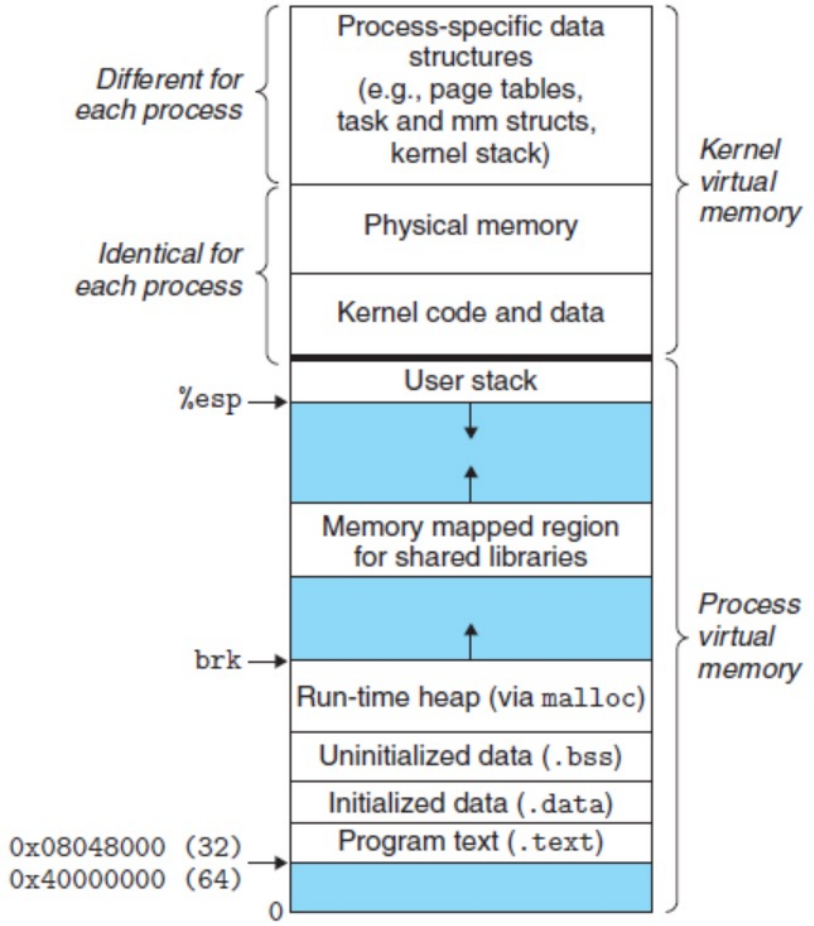
Linux Virtual Memory System

Shared kernel virtual memory

- Kernel's code, global data structure
- Virtual pages mapped directly to physical pages

Private kernel virtual memory

- Page tables, stack, task and mm structs



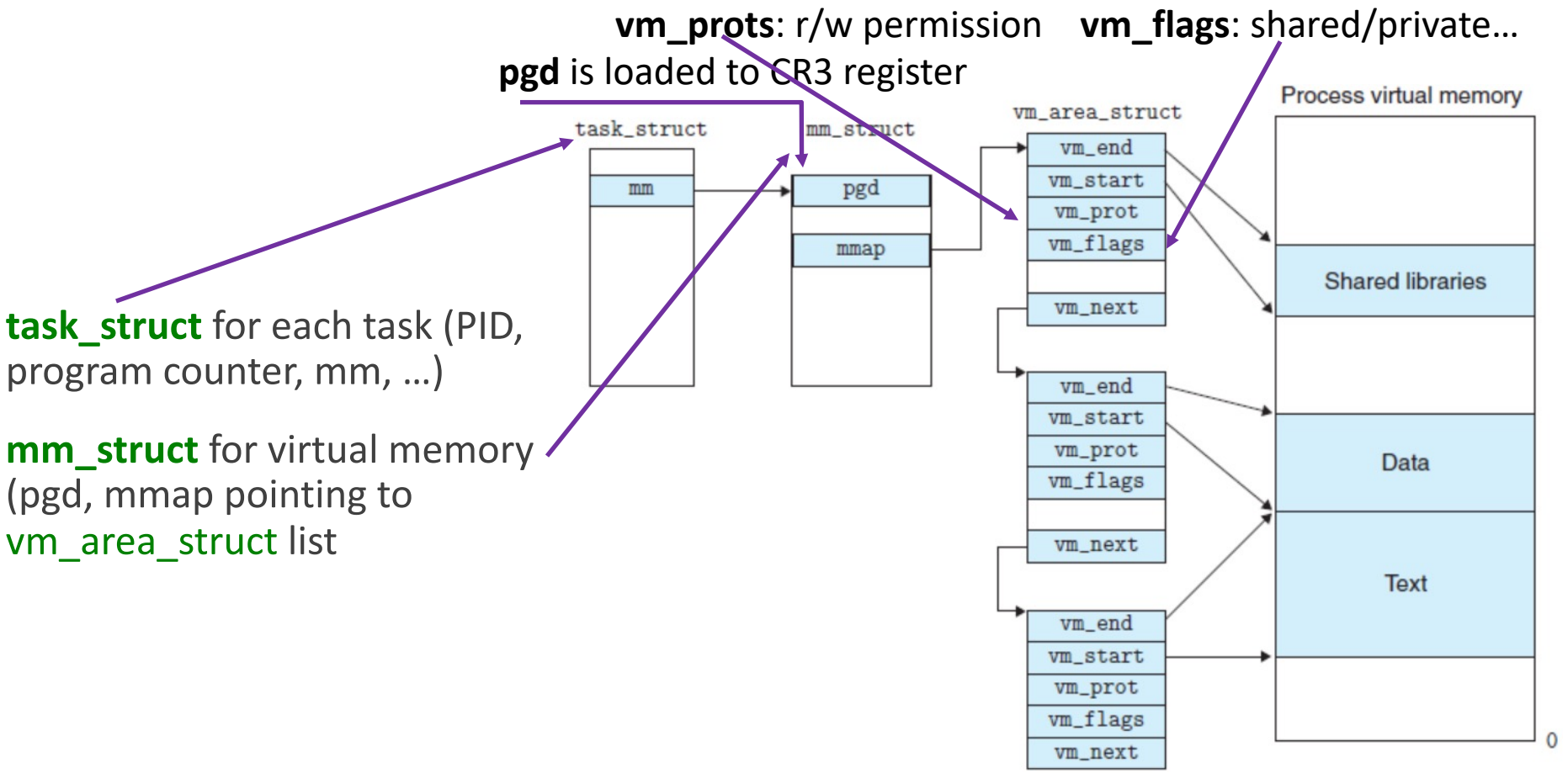
Linux Virtual Memory Areas

Area (Segment)

- A contiguous chunk of existing (allocated) virtual memory whose pages are related
- E.g., code segment, data segment, heap, shared library segment, user stack

- Each existing virtual page is contained in some area
- Any virtual page not contained in an area does not exist and cannot be referenced

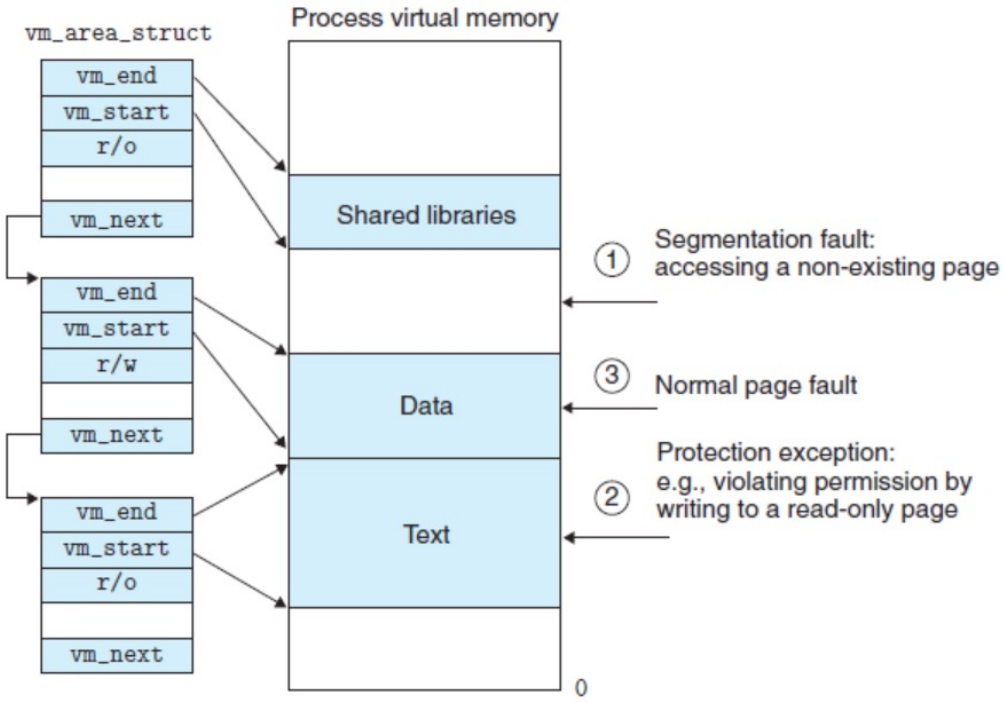
Linux Virtual Memory Area



Linux Page Fault Exception

Suppose that MMU triggers a page fault while translating a virtual address A. The kernel page fault handler does the following:

1. Is virtual address A legal? => segmentation fault
2. Is attempted access legal? => protection exception
3. Otherwise, swap out/in the page and restart the faulting instruction



Memory Mapping

Memory mapping: initialize the contents of virtual memory area by associating it with an object on disk

Regular file in the Linux file system

- File section is divided into page-size pieces
- Demand paging => pages are loaded only when they are used

Anonymous file

- A file, created by the kernel, that contains all binary zeros
- No data are actually transferred between disks and memory

Swap file

- Once a virtual page is initialized, it is swapped back and forth between a special swap file

Shared Objects

Many processes have identical read-only code areas

- Linux shell programs have identical code area
- Standard C library such as printf are common
- Wasteful if each process keeps a duplicate copy

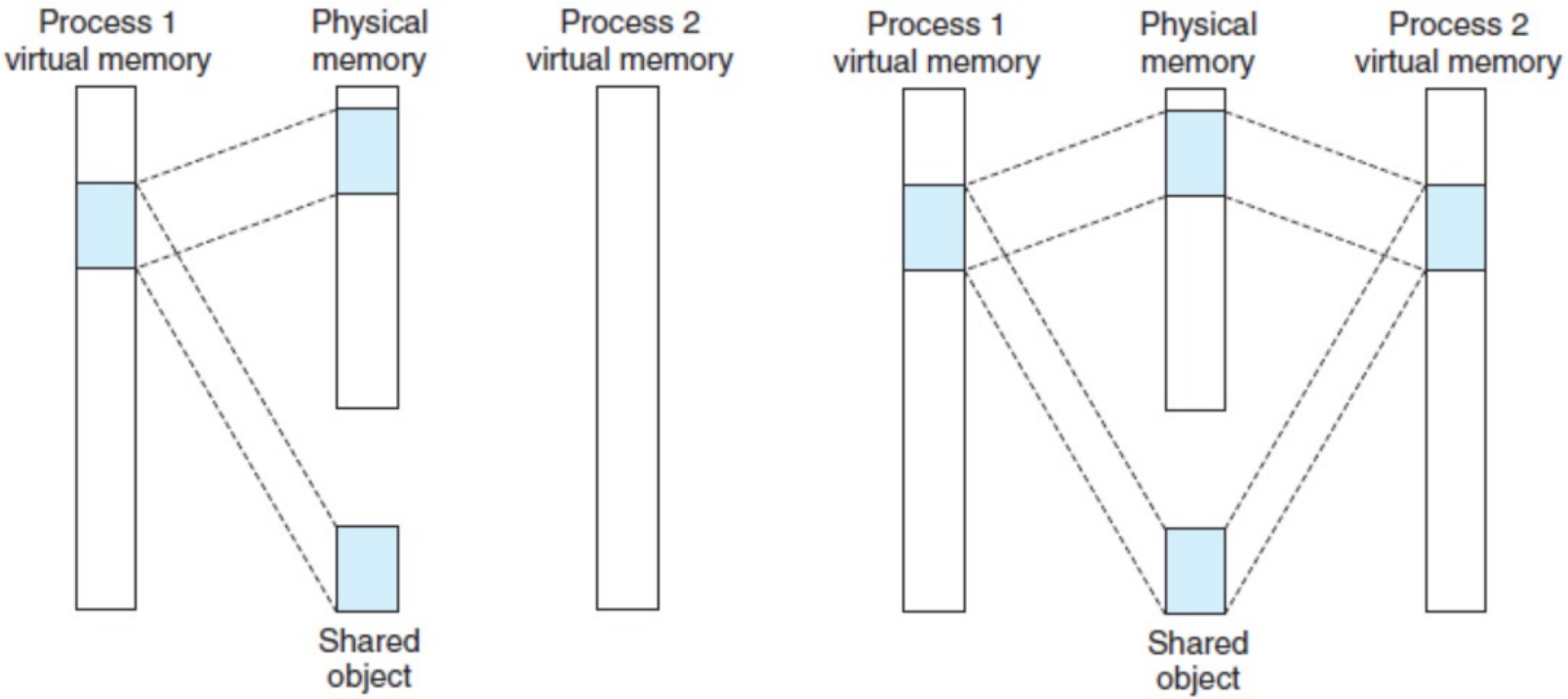
Shared object

- If a process writes to an area mapped to a shared object, the change is visible to other processes that mapped the shared object to their virtual memory
- The shared object on disk is also updated

Private object

- Changes made to an area mapped to a private object are not visible to other processes
- The original object on disk is not updated

Shared Objects



Copy-on-Write

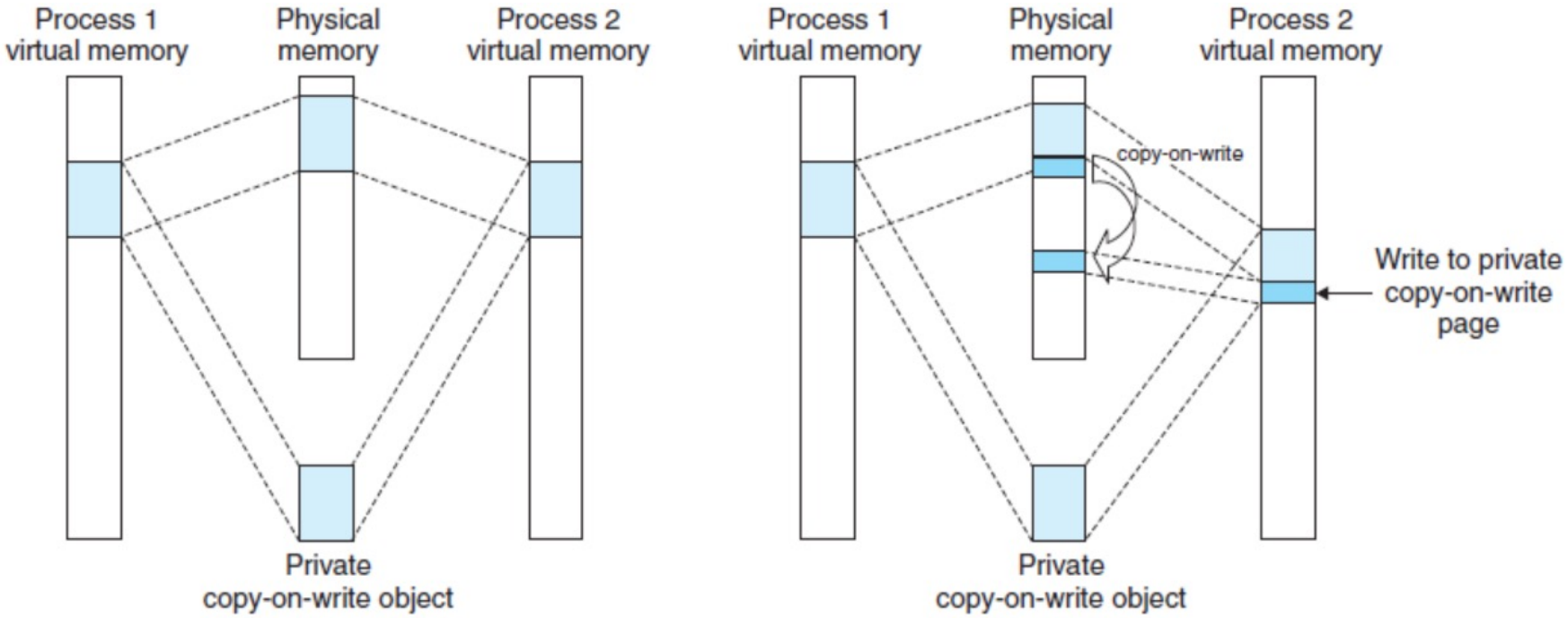
Private objects are mapped into virtual memory like shared objects except that

- Page table entries are flagged as read-only
- Area struct is flagged as private copy-on-write (cow)

When a process tries to write to some private areas

- A protection fault is triggered
- The fault handler checks that the fault is from the private copy-on-write area
- Creates a new copy of the page, updates the page table entry and restores the permissions to the page

Copy-on-Write



Fork function

When fork is invoked

- Kernel creates data structures for the new process
- To create a virtual memory for the new process
  - The current process' mm_struct, area structs and page tables are copied
  - Flag each page in both processes as read-only
  - Flag each area struct in both processes as private copy-on-write
- Both processes have exactly the same virtual memory
- As processes write, new pages are created by the copy-on-write

Execve

Delete existing user areas

Map private areas

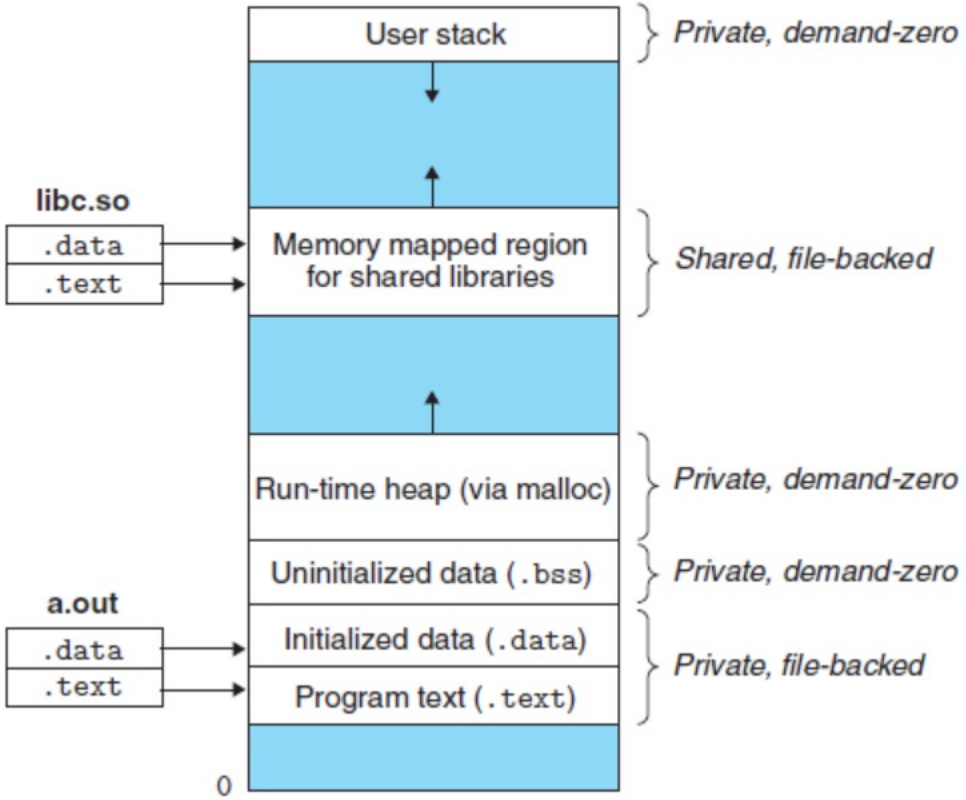
- Create new area structs for code, data, bss, stack
- All areas are flagged as private copy-on-write
- Code and data areas are mapped to .text and .data
- Bss area is demand-zero, mapped to an anonymous file whose size is in the executable file
- Heap and stack are demand-zero, of 0 length

Map shared areas

- Shared objects are dynamically linked into the program and mapped into the shared region

Set the program counter (PC)

How the Loader Maps the Areas



Questions?