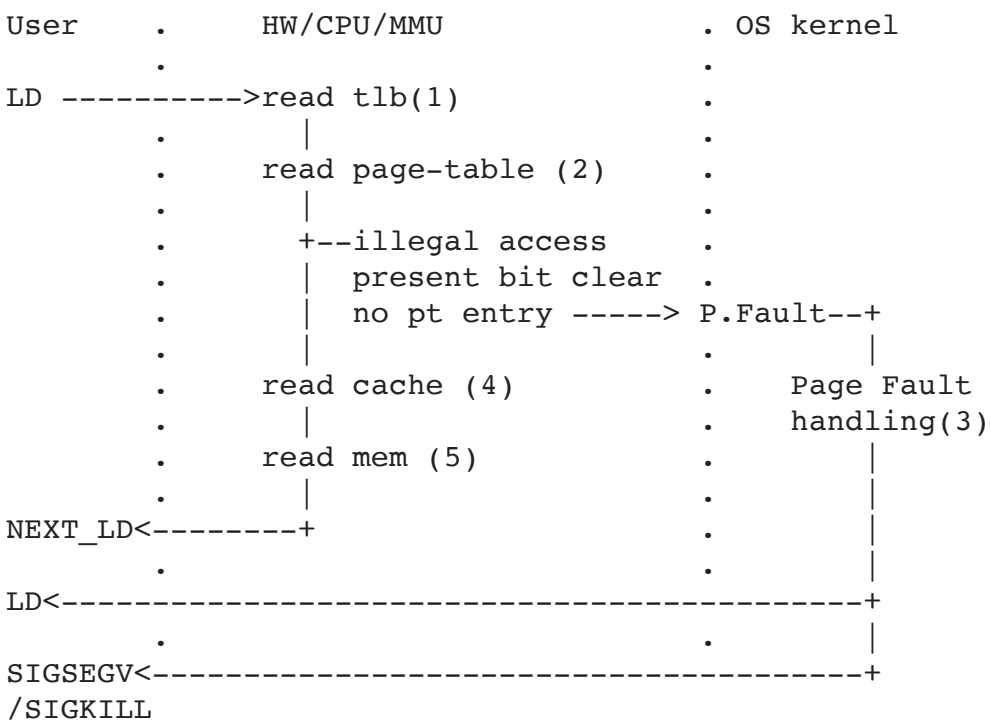
PAGE FAULTS/MEMORY PROTECTION (3.4)

YUZHE TANG

OVERVIEW

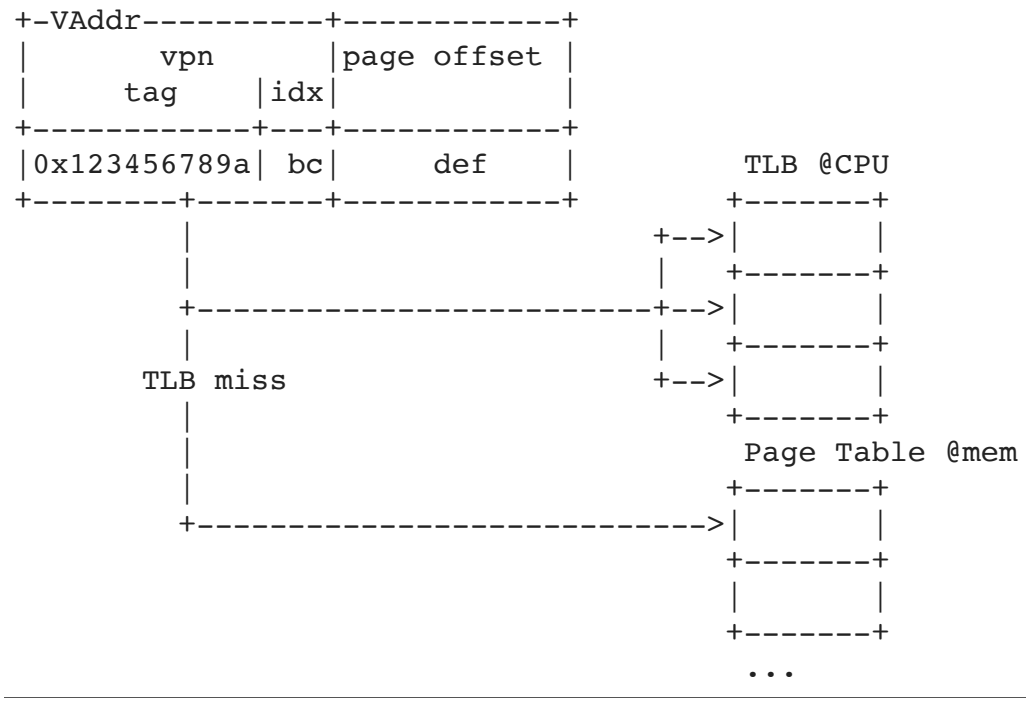
HOW DOES `LD` REALLY WORK?



- (assuming a physically indexed cache)

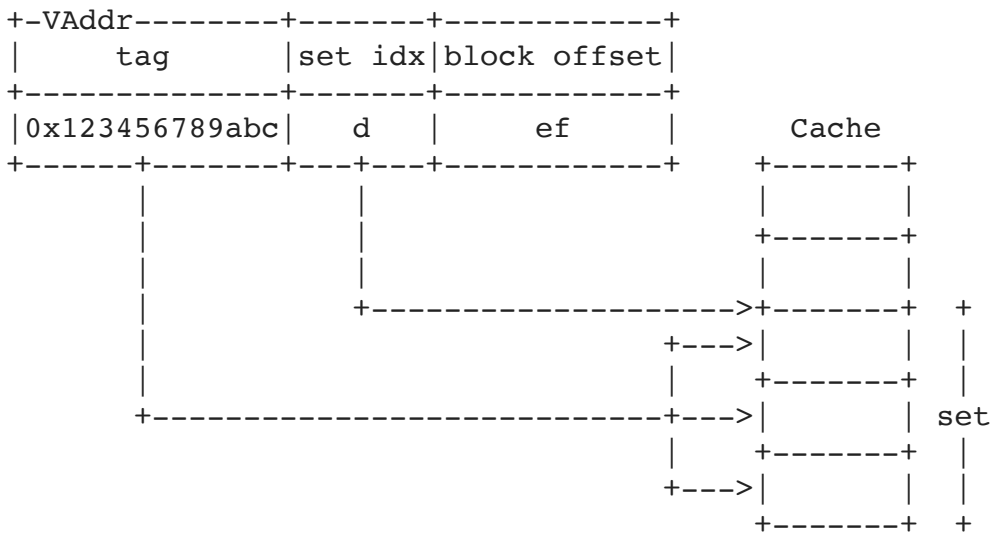
- `LD` requires collaboration between hardware and OS kernel.
  - HW: x86-64
  - OS: Linux
- Briefly on (1), (2), (4) (previously covered) and (5)
- Mainly on (3)

READ TLB(1)/PT(2)



- page size: 4096 words (or 4KB with 1B-sized word)
  - page offset: 12 bits
- memory location of page table is in `PDBR` (in `CR3`)
- page table can be multi-level
  - x86 page table has 3 levels; x86-64 has 4 levels
  - linux can adapt both: `PGD`,(`PUD`),`PMD`,`PTE`

READ CACHE(4)/MEM(5)

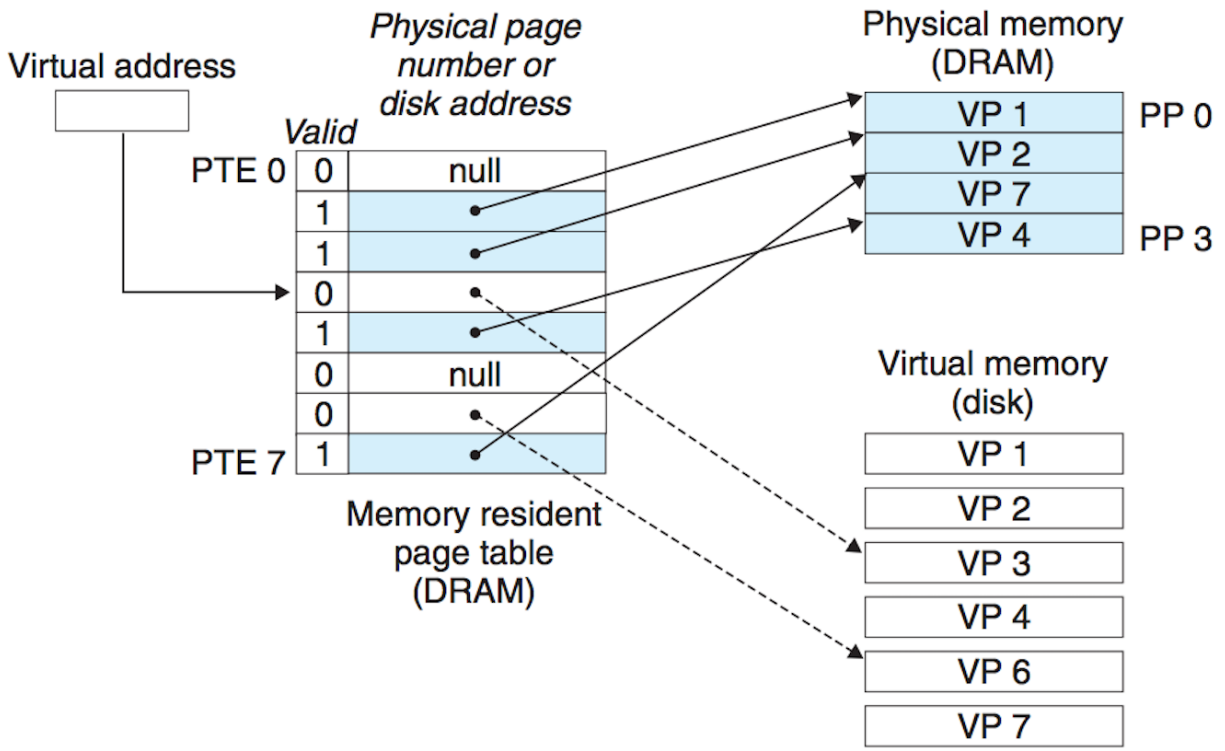


- 256 (2^8) words per memory block
  - 16 sets in cache
- if cache miss, then read memory (5):
  - it loads data from memory to cache, and to register
  - it also loads address translation to TLB

(3) HANDLING PAGE FAULT

PAGE FAULT

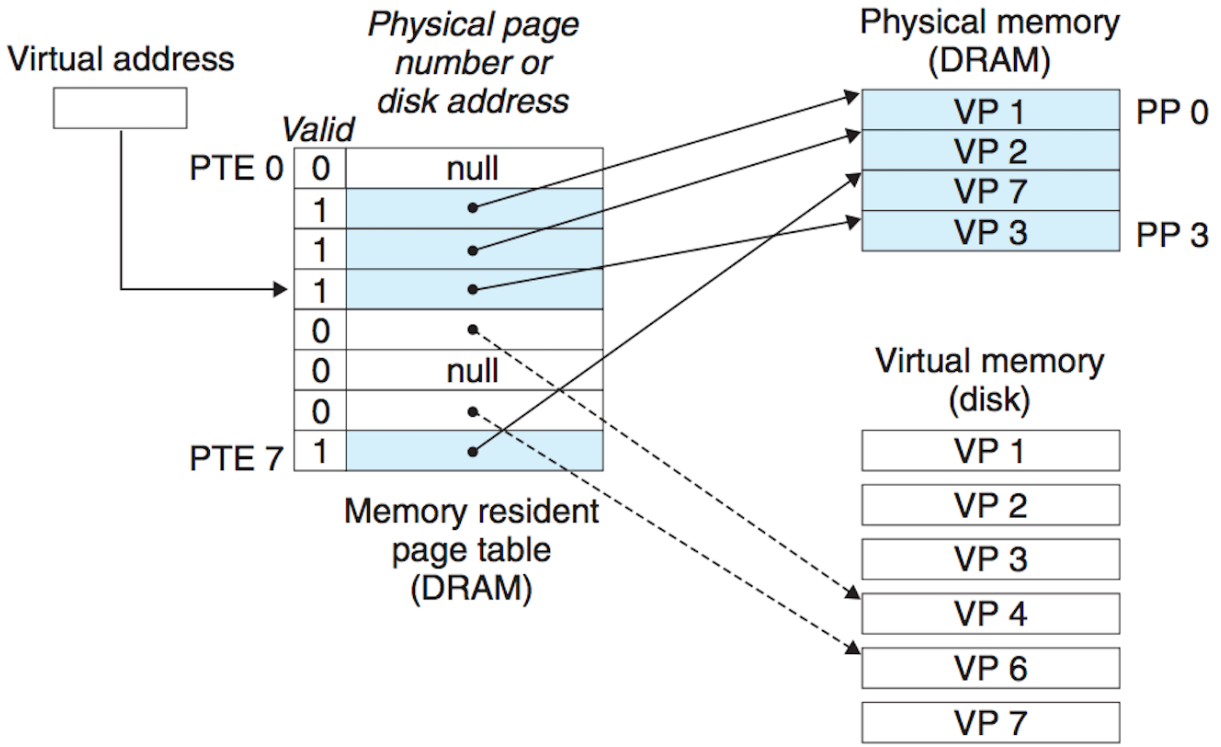
Figure 9.6
VM page fault (before). The reference to a word in VP 3 is a miss and triggers a page fault.



Page fault: before

PAGE FAULT

Figure 9.7
VM page fault (after). The page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk. After the page fault handler restarts the faulting instruction, it will read the word from memory normally, without generating an exception.



Page fault: after

TRIGGERING PAGE FAULT

- Page table entry's `present` bit clear:
  - page frame is on disk
- No page table entry for the referenced virt. addr:
  - page accessed for the first time (i.e. demand paging)
- Illegal access: page-based permission
  - e.g.: `ST` to a read-only page
  - by CPU checking PTE

OS HANDLING WORKFLOW

1. Is it a legal virt. address?
   - e.g.: are you accessing kernel space?
   - Linux: `find_vma() != null` (in `do_page_fault()`)
2. Is the access legal? (1:Yes)
   - e.g.: can you write to `.text`? Segment fault!
   - segment/area based permission
   - checking a kernel data structure
   - Linux: `mm_struct->mmap->vm_area_struct.flag`

3. Who is accessing? (1:No)
    - e.g.: are you accessing kernel space? and you're user code? Segment fault!
    - e.g.: can kernel access unallocated memory? General protection fault!
4. If legal access, do the swapping (1:Yes,2:Yes)
    - choose victim page, swap-out if dirty
    - swap-in (`present` bit clear) or allocate (not present) page frame

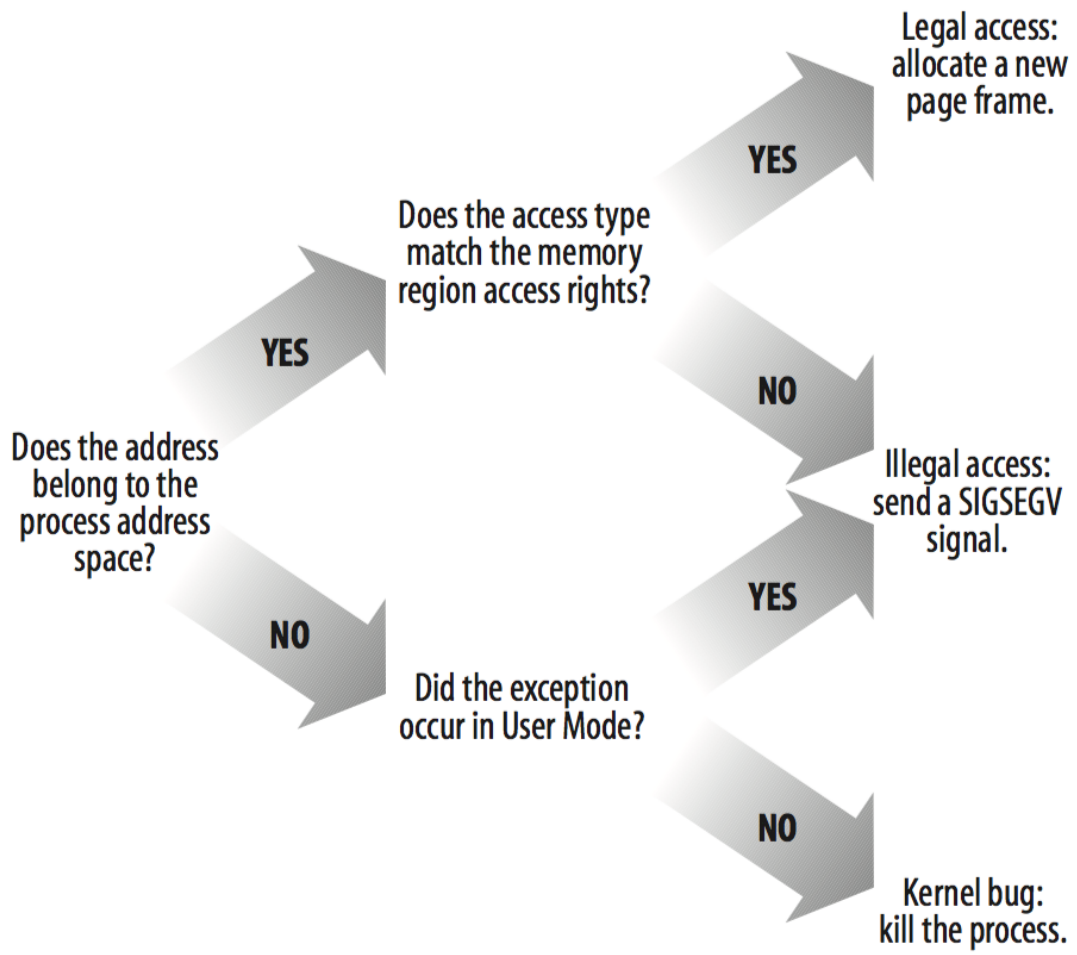


Figure 9-4. Overall scheme for the Page Fault handler

Page fault: logic

MEMORY PROTECTION

PAGE-BASED PERMISSIONS

- A page can be:
  - execution only: code
  - read only: the value of `const pi=3.1415;`
  - read/writes: most data in a program
- Permission bits for each page, in page-table entry
  - `R/W` for read/write; if clear (=0), read-only.
  - `U/S` for user/supervisor; if clear, accessible only by the kernel.

PERMISSION CHECK

- Page-based permission check is done by hardware during translation
  - whenever tlb is accessed
  - whenever page table is accessed
- Raise exception if wrong kind of access

PAGE VERSUS SEGMENT PERMISSIONS

	Page-based permission	**Segment-based permission**
enforced by	hardware	OS
granularity	per page	per segment (multiple pages)
semantics	low-level(e.g. `rwx`)	app-level (e.g. grow downwards?)

USER AND SYSTEM ROLES

Permissions are against programs of diff. roles

- System(kernel) role: Operating systems.
  - OS should do many things apps can’t
  - can read/write any memory (otherwise, how to load a program?)
  - can access physical memory directly w.o. translation (otherwise how to read page-table?)
  - Use instructions/regs apps can't use (e.g. `IRET`, `CR3`)
- User role: User apps.
  - Can't change page table pointer or write to TLB

SYSTEM AND USER MODE

- Processor tracks which mode it's currently in
- Mechanism to transit btwn two modes
  - User code traps into system mode by interrupt instructions (e.g. `INT`, `SYSCALL`)
- Saves current mode, changes to system mode
  - Return from interrupt (RETI in x86) sets mode back to what it was

SYSTEM MODE

- Allows access to physical memory
- Skip address translation and permission checks
  - Allows execution of special instructions
- User mode: “unknown instruction” exception for these
- Most of these provide access to privileged registers
  - Page table pointer (used by TLB misses)
  - Interrupt vector table (used to handle interrupts/exceptions)
  - ... (the list goes on and on)

EXTRA PROTECTION

- How do we prevent a user application from modifying its own page table
  - Remember: page table is stored in phys. Memory
- How do we prevent a user application from directly accessing I/O devices
  - Supposed to use system calls (e.g. read, write, seek)
  - But I/O device (control and data registers) are memory-mapped

Should we use permission bits?

- We don’t map those into address space!
- As far as user process is concerned,
  - page table does not even exist!
  - IO device does not exist in (virt.) memory!
- Can’t address it = can’t access it