Welcome back!

MEMORY: SMALLER PAGE TABLES AND SWAPPING

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

Project 3 is due next week

Code review ~ 10 - 15 min . Post Piazza

Midterm 1: Oct 15th

- Practice exams on Canvas
- Review session in class

AGENDA / LEARNING OUTCOMES

Memory virtualization

- What are the challenges with paging ?
- How we go about addressing them?

Size of Page table

- How we support virtual mem larger than physical mem?
- What are mechanisms and policies for this?

RECAP

PROS/CONS OF PAGING

Pros

No external fragmentation

- Any page can be placed in any frame in physical memory

Fast to allocate and free

- Alloc: No searching for suitable free space
- Free: Doesn't have to coalesce with adjacent free space

Cons

Additional memory reference

- MMU stores only base address of page table

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
- Entry needed even if page not allocated ?

TLB SUMMARY

Pages are great, but accessing page tables for every memory access is slow

Cache recent page translations → TLB → MMU

- MMU performs TLB lookup on every memory access
  - ↳ hit then directly access the physical page

TLB performance depends strongly on workload

TLBs increase cost of context switches

- Flush TLB on every context switch
- Add ASID to every TLB entry → multiple processes share the TLB

In different systems, hardware or OS handles TLB misses

↳ know layout of page tables

WHY ARE PAGE TABLES LARGE?

Linear Page Tables

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

how to avoid storing these?

→ every virtual page

N virtual pages

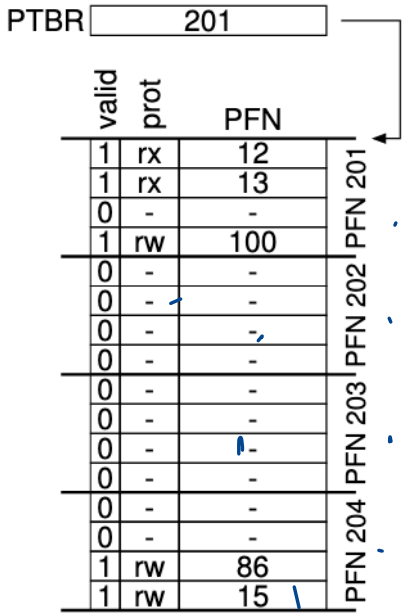
4 bytes

size = 4 * N

MULTILEVEL PAGE TABLES

wasting 2 pages

Linear Page Table



4 pages to store page table

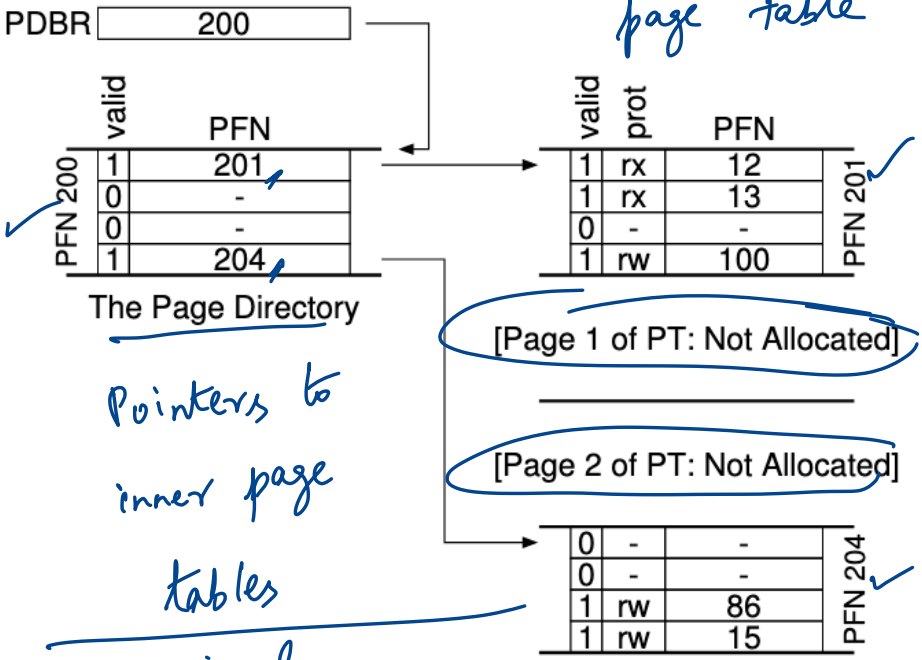
Creates multiple levels of page tables

Only allocate page tables for pages in use

Allow page table to be allocated non-contiguously

Multi-level Page Table

3 pages in this 2-level page table



Pointers to inner page tables

Additional memory reference

MULTILEVEL TRANSLATION EXAMPLE

page directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0x01ABC

offset = ABC

outer = 0

inner = 1

0x23 = PPN

PA: 0x23ABC

20-bit address:

fits in a page

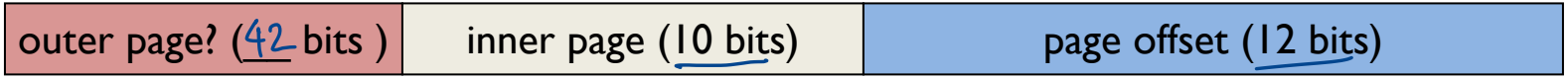
4KB pages

outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

MORE THAN 2 LEVELS?

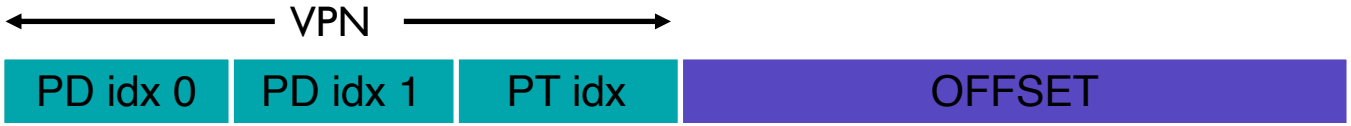
Problem: page directories (outer level) may not fit in a page

64-bit address:



Solution:

- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.



How large is virtual address space with 4 KB pages, 4 byte PTEs, (each page table fits in page)

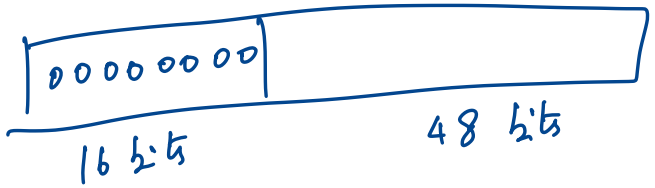
1 level: 1024 entries Page table x 4K page = 4 MB → virtual address space

2 levels: 1024 x 1024 x 4K = 4 GB → virtual addr space

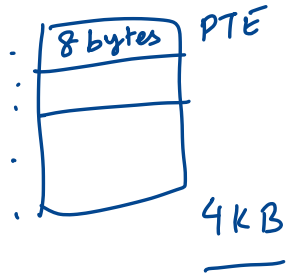
4KB / 4 bytes → 1K entries per level

EXAMPLE: X86-64 → 64 bit mode

Virtual address: 48 bits
– Upper 16 bits are sign extended (all 0, all 1)
Physical address: 48 bits



PTE size = 48 bits + metadata = 8 bytes
PTEs/4kb page = 512 = 9bits



$= \frac{4KB}{8} = 512$

47 - 39	38-30	29-21	21-12	11-0
Page Map Lvl 4 (9 bits)	Page Pointer Dir. (9 bits)	Page directory (9 bits)	Page table (9 bits)	offset (12 bits)

4 levels in our page table → 512 entries in a 4KB page

→ 4KB pages

FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: addl $0x5, %edh

fetch instruction. No entry 0xAA in TLB. TLB miss = 4
3 mem accesses to get translation. 1 mem access

(b) 0xBB13: addl $0x3, %edi

PA = 0x9113

1 mem access for instr. fetch

INVERTED PAGE TABLE

inverted

PPN → VPN + ASID

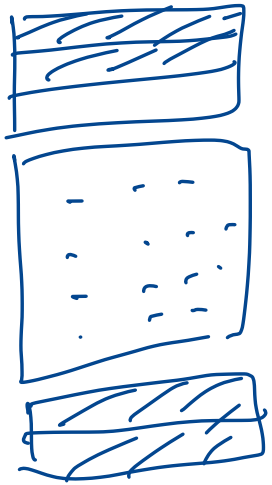
Only store entries for virtual pages w/ valid physical mappings

Naïve approach:

Search through data structure <ppn, vpn+asid> to find match

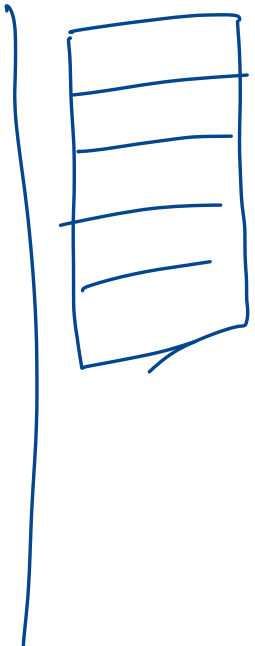
Too much time to search entire table

Linear Page table



4 valid entries

VPN → PPN



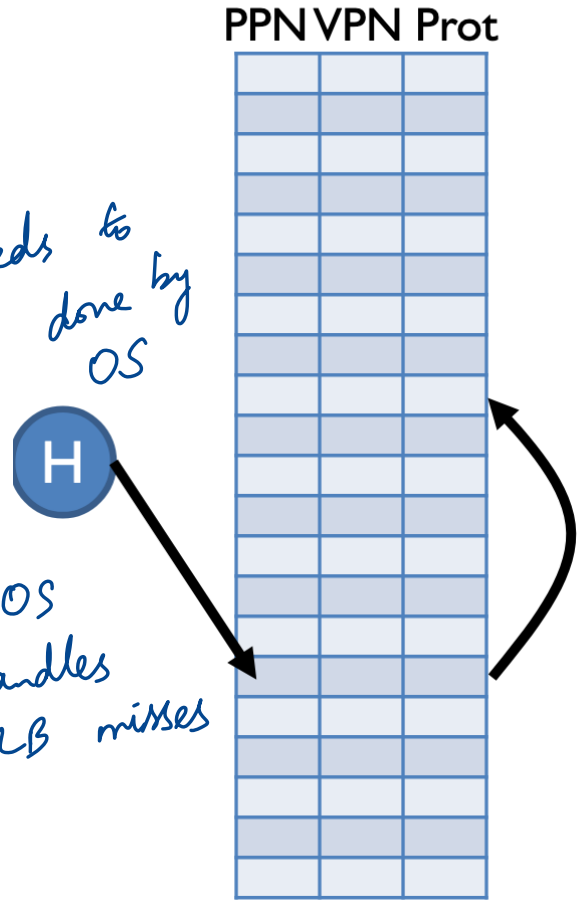
INVERTED PAGE TABLE

Better:

- Find possible matches entries by hashing vpn+asid
- Use chaining to handle collisions
- Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB

Used in IBM POWER and Intel Itanium, but complicated



QUIZ 7

https://tinyurl.com/cs537-fa24-q7

What problem(s) can be solved by using ASIDs ?

→ TLB flushed across context switches

For a hardware-managed TLB miss, which of the following statements are true?

→ HW knows page tables are located

OS plays no role

For a software-managed TLB miss, which of the following statements are true?

→ HW privilege level → kernel mode from user mode

OS computes new entry

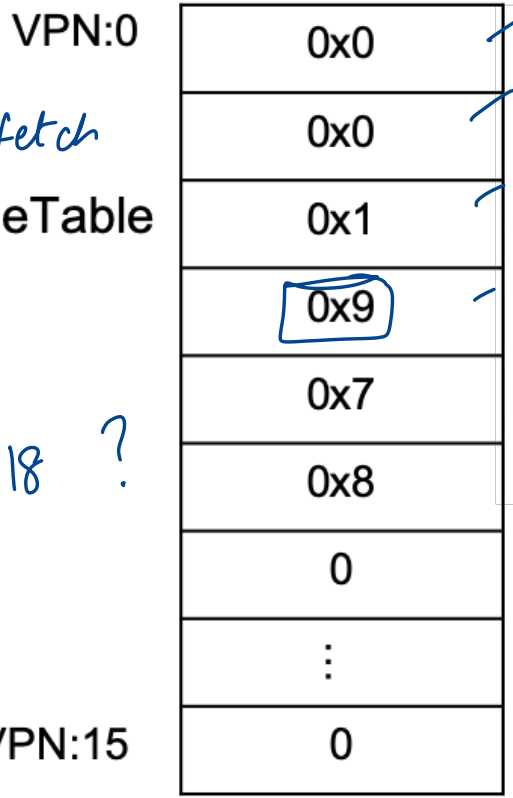
16-bit address space and 4kB page size.
Assume the page table is at 0x2000
Each PTE is of 4 bytes. No TLB

0x200C first mem access
0x9000 second
↳ load ....

Instr 1:
2 mem access instr fetch
2 mem access load PageTable
= 4

Instr 2: 2 + 2 = 4
Inst 3: 2 + 0 = 2
Inst 4 = 2 + 2 = 4
Inst 5 = 2 + 2 = 4

18 ?



Virtual Addresses
0x3000: load 0x5320, %eax
0x3004: load 0x4004, %ebx
0x3008: mul %ecx, %eax, %ebx
0x300C: store %ebx, 0x5324
0x3010: load 0x5328, %ebx

16-bit address space and 4kB page size.
Assume the page table is at 0x2000
Each PTE is of 4 bytes. With 5 entry TLB

TLB
0x3 → 0x9
0x5 → 0x8
0x4 → 0x7

Inst 1 : 2 + 2
= 4

Inst 2 : TLB hit
1 + 2 = 12

Inst 3 : 1

Inst 4 : 1 + 1 = 2

Inst 5 : 1 + 1 = 2

	PageTable
VPN:0	0x0
	0x0
	0x1
	0x9
	0x7
	0x8
	0
	⋮
VPN:15	0

Virtual Addresses
0x3000: load 0x5320, %eax
0x3004: load 0x4004, %ebx
0x3008: mul %ecx, %eax, %ebx
0x300C: store %ebx, 0x5324
0x3010: load 0x5328, %ebx

EFFECT OF PAGE SIZE

1024 pages
32 entries

page →  32 entries

Assume 4KB pages = 1024 integers
32 TLB entries

↑ randomly accessing

```
int sum = 0;
int a[1024*1024];
for (i=0; i<1024*1024; i++) {
    sum += a[rand() %
             (1024*1024)];
}
```

↑ ~1M integers

Miss rate of TLB: # TLB misses / # TLB lookups

TLB lookups? number of accesses to a = 1M lookups

Chance of a TLB miss? 1 - 32/1024
=
TLB hit = 32/1024 = size of TLB / number of pages

Assume 2MB pages
32 TLB entries

↑ ~500K integers

= 2 pages for array a

TLB lookups? number of accesses to a =

Chance of a TLB miss?
=
2 misses for entire loop

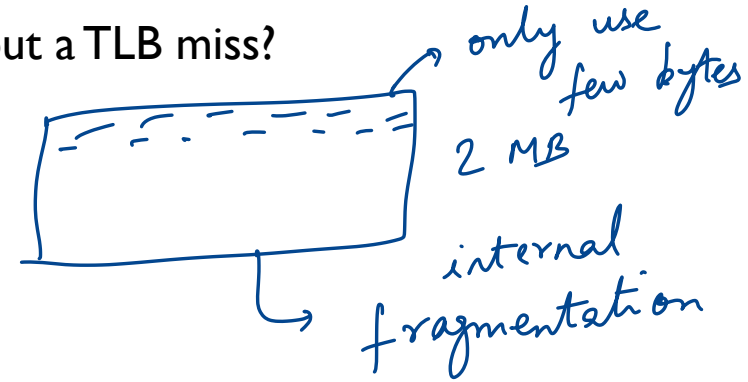
Large Pages ⇒ Fewer TLB Miss

LARGE PAGES (HUGE PAGES)

TLB reach: how much memory can be accessed without a TLB miss?

1000 entries 4KB pages → 4MB

Large pages
1000 entries, 2MB pages → 2GB!



How to use?

- Programmer requested: mmap(MAP_HUGE) returns huge pages
- Transparent Huge Pages (THP, in Linux)
  – OS uses huge pages when available for > 2MB allocations

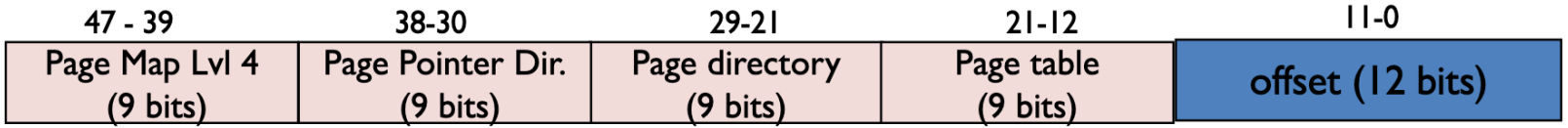
↳ 2MB pages for your allocation

TRANSLATING LARGE PAGES

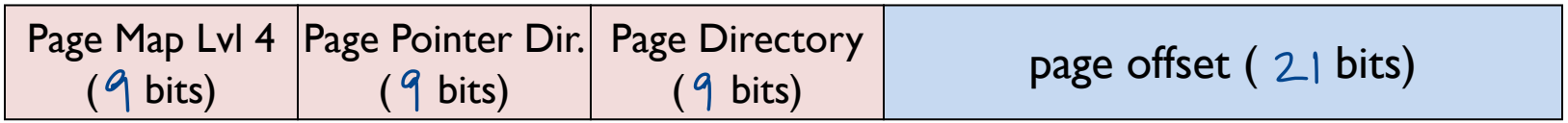
x86-64
= 48 bits
virtual addr

HugePages saves TLB entries. But how does it affect page translation?

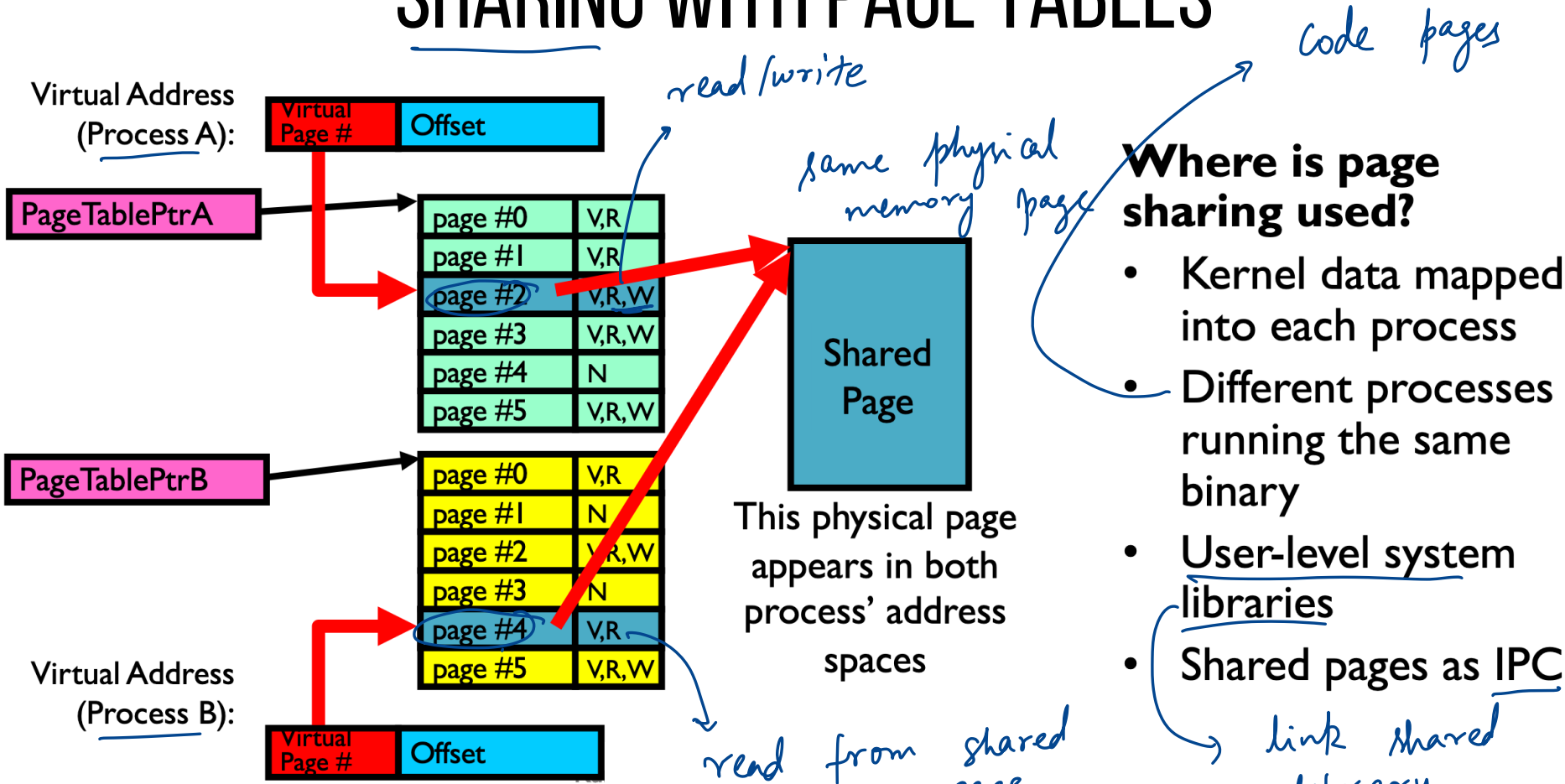
4KB pages: 4 levels → 4 memory accesses



2MB pages: 3 level → 3 mem access for address translation



SHARING WITH PAGE TABLES



SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format

- Multi-level page tables used in x86 architecture
- Each page table fits within a page

Large pages can reduce TLB use and number of accesses for translation

SWAPPING

MOTIVATION

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

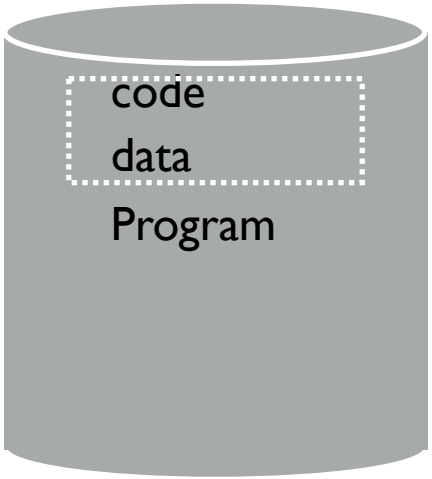
User code should be independent of amount of physical memory

- Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does this work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)



Virtual Memory

LOCALITY OF REFERENCE

Leverage locality of reference within processes

- Spatial: reference memory addresses **near** previously referenced addresses
- Temporal: reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

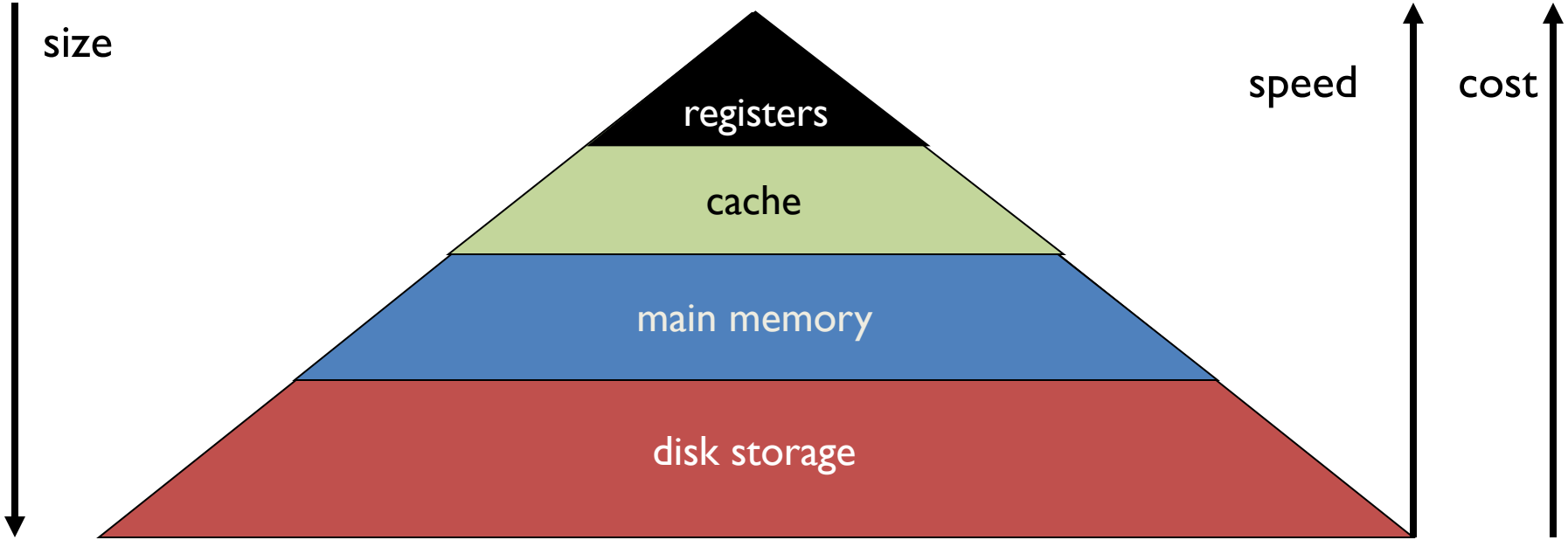
Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

MEMORY HIERARCHY

Leverage memory hierarchy of machine architecture

Each layer acts as "backing store" for layer above



SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to make large disk seem like memory

- Same behavior as if all of address space in main memory

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** to determine which pages live in memory and which on disk

VIRTUAL ADDRESS SPACE MECHANISMS

Each page in virtual address space maps to one of three locations:

- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: present

- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk
  - Causes trap into OS when page is referenced
  - Trap: page fault

Disk

Phys Memory

PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
28	1	rw-	0
4	1	rw-	1

What if access vpn 0xb?

VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

Else ...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory (i.e., present bit is cleared)

Else

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
  - Write victim page out to disk if modified (use dirty bit in PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

SWAPPING POLICIES

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions

- Page selection

  When should a page (or pages) on disk be **brought into** memory?

- Page replacement

  Which resident page (or pages) in memory should be **thrown out** to disk?

NEXT STEPS

Project 3: Due soon!