MEMORY: SMALLER PAGETABLES

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Project 3 available: Shell in Linux (still solo)
  - Discussion sections (fork() and exec())
  - Test scripts available after/during weekend
- Midterm 1: Thursday, Oct 10th from 7:30-9:30pm
  - Fill out Exam Conflict form in Canvas by TODAY
  - Two sample exams posted – with answers!
  - Next discussion sections on practice exam
- Canvas Homeworks
  - "Due" each Tuesday and Thursday

AGENDA / LEARNING OUTCOMES

Memory virtualization

- How we reduce the size of page tables?
- What can we do to handle large address spaces?

RECAP

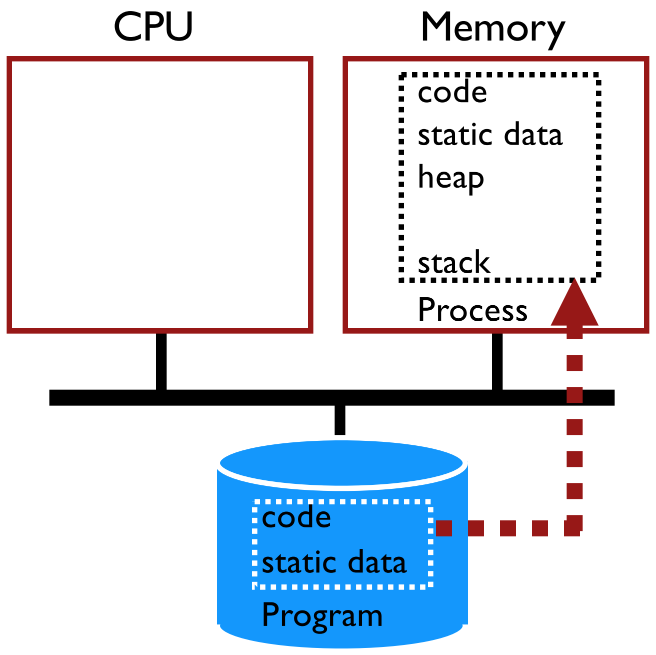
WHEN ARE PAGE TABLES CREATED?

OS creates new page table when creates process

- OS chooses where process code, heap, and stack are placed in RAM
- OS sets up page tables to contain initial mappings

OS modifies page tables when it allocates more process address space

Picks physical locations in RAM



PAGING TRANSLATION STEPS

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory

DISADVANTAGES OF PAGING

Additional memory reference to page table → Very inefficient

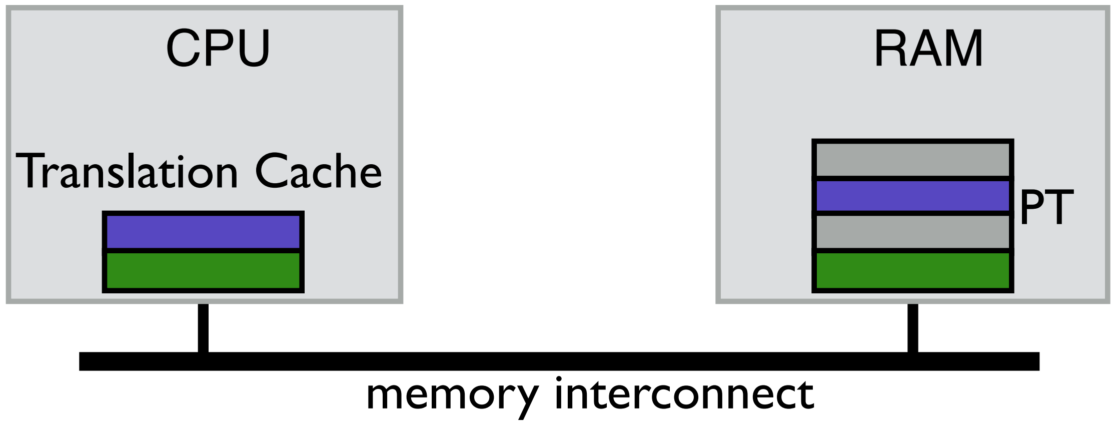
- Page table must be stored in memory
- MMU stores only base address of page table

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 ?

STRATEGY: CACHE PAGE TRANSLATIONS



PAGING TRANSLATION STEPS

For each mem reference:

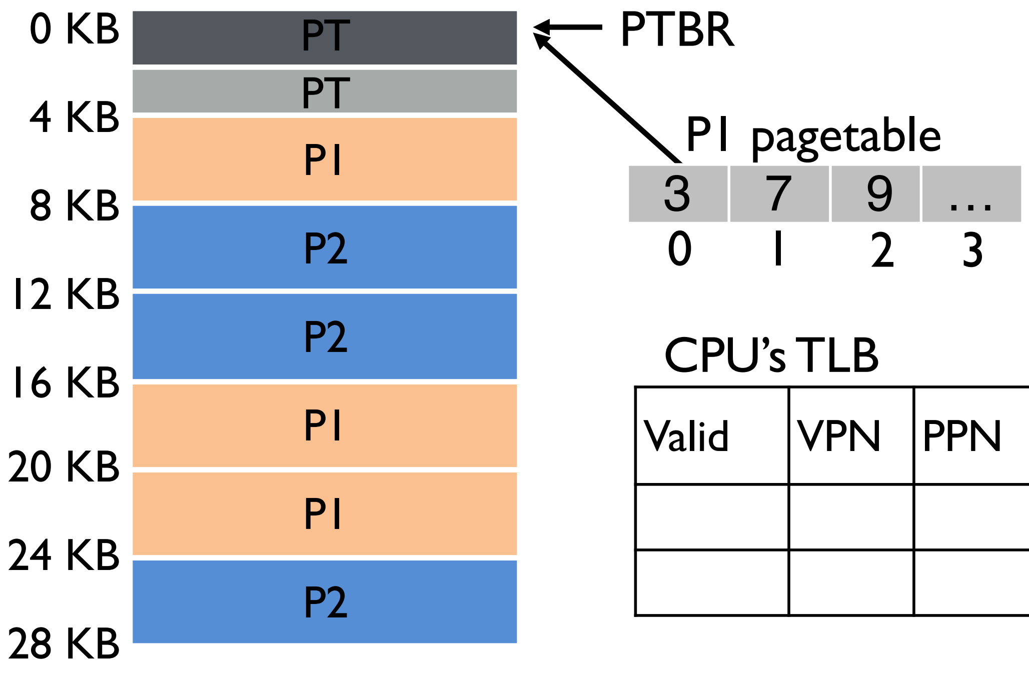
1. extract **VPN** (virt page num) from **VA** (virt addr)
2. check TLB for **VPN**

   if miss:

   3. calculate addr of **PTE** (page table entry)
   4. read **PTE** from memory, replace some entry in TLB

5. extract **PFN** from TLB (page frame num)
6. build **PA** (phys addr)
7. read contents of **PA** from memory

TLB ACCESSES: SEQUENTIAL EXAMPLE



CPU's TLB

Valid	VPN	PPN

Virt	Phys
load 0x0000	
load 0x0004	
…	
…	
load 0x2000	

HW AND OS ROLES

Who Handles TLB Hit?

Who Handles TLB Miss? HW or OS

H/W

H/W must know where pagetables are stored in memory

- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW "walks" known pagetable structure and fills TLB

HW AND OS ROLES

Who Handles TLB MISS? **H/W** or **OS**?

OS:

CPU traps into OS upon TLB miss
"Software-managed TLB"

OS interprets pagetables as it chooses; any data structure possible
Modifying TLB entries is privileged instruction

TLB SUMMARY

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

Cache recent page translations → TLB

- Hardware performs TLB lookup on every memory access

TLB performance depends strongly on workload

- Sequential workloads perform well
- Workloads with temporal locality can perform well (if enough TLB entries)

In different systems, hardware or OS handles TLB misses

TLBs increase cost of context switches

- Flush TLB on every context switch
- Add ASID to every TLB entry

DISADVANTAGES OF PAGING

Additional memory reference to page table → Very inefficient

- Page table must be stored in memory
- 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 ?

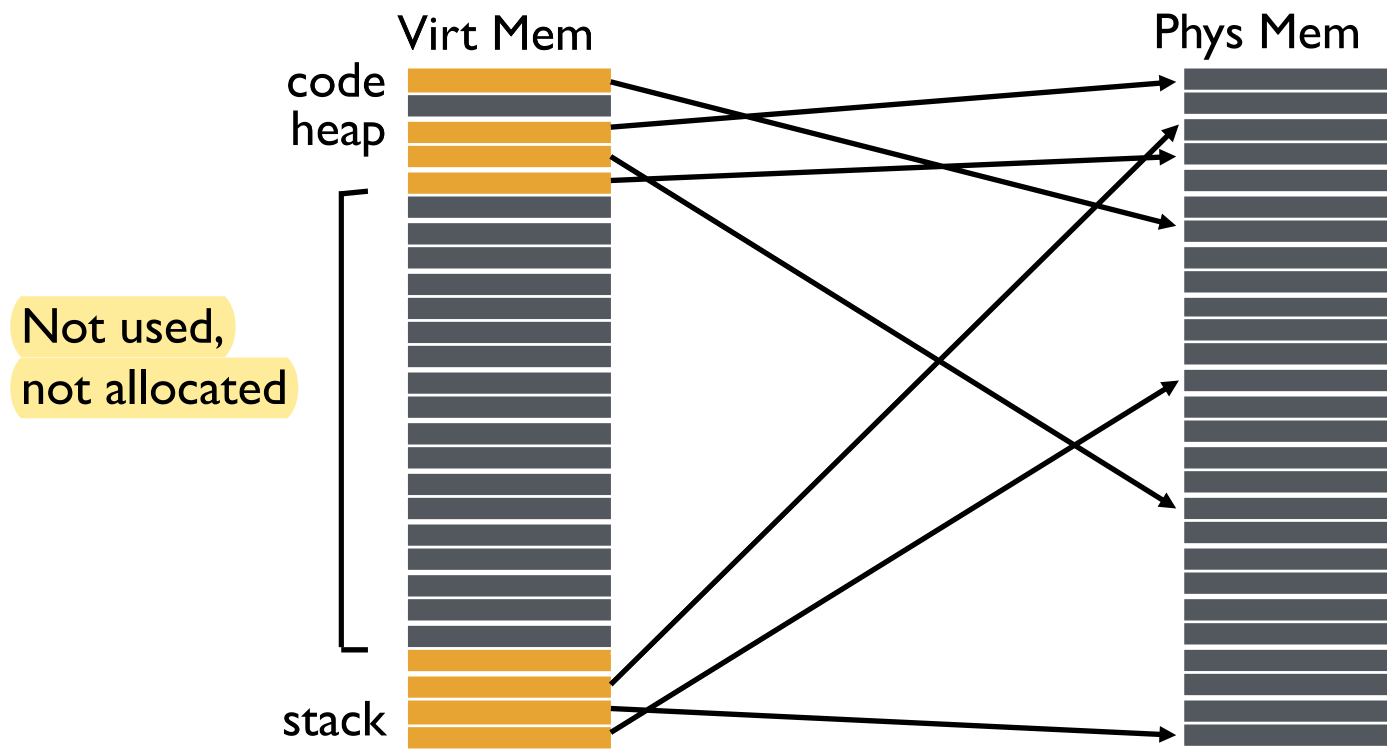
SMALLER PAGE TABLES

QUIZ: HOW BIG ARE PAGE TABLES?

1. PTE's are **2 bytes**, and **32** possible virtual page numbers
2. PTE's are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**
3. PTE's are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**
4. PTE's are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

How big is each page table?

WHY ARE PAGE TABLES SO LARGE?



MANY INVALID PT ENTRIES

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

how to avoid storing these?

Problem:

Linear page tables must still allocate PTE for each page in address space (even for unallocated pages)

AVOID SIMPLE LINEAR PAGE TABLES?

Use more complex page tables, instead of just big array

With software-managed TLB any data structure is possible

Hardware looks for vpn in TLB on every memory access

- If TLB does not contain vpn, TLB miss
  - Trap into OS and let OS find vpn->ppn translation
  - OS notifies TLB of vpn->ppn for future accesses

OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
3. Inverted Pagetables

VALID PTES ARE CONTIGUOUS

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 PTEs?

Note "hole" in addr space: valids vs. invalids are clustered

How did OS avoid allocating holes in address space?

Segmentation

COMBINE PAGING AND SEGMENTATION

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages

Logical address divided into three portions

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

Implementation

- Each segment has a page table
- Each segment track base (physical address) and bounds of the **page table**

CHAT: PAGING AND SEGMENTATION

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read:	
0x202016 read:	
0x104c84 read:	
0x010424 write:	
0x210014 write:	
0x203568 read:	

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

Assume bounds:
# PTE entries

ADVANTAGES OF PAGING AND SEGMENTATION

Advantages of Segments

- Supports sparse address spaces.
  - Decreases size of page tables.
  - If segment not used, not need for page table

Advantages of Pages

- No external fragmentation
- Segments can grow without any reshuffling
- Can run process when some pages are swapped to disk (next lecture)

Advantages of Both

- Increases flexibility of sharing: Share either single page or entire segment

SHARING: PAGING AND SEGMENTATION

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

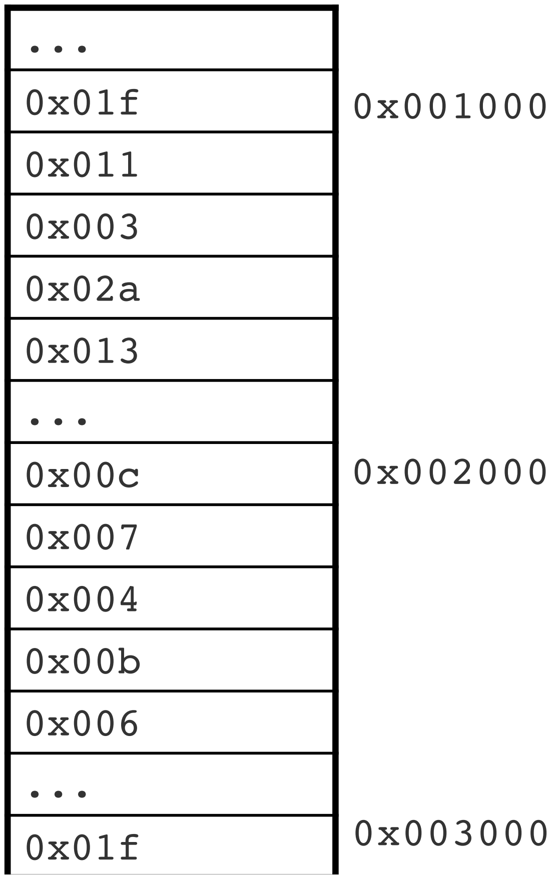
P1:

seg	base	bounds	R W
8	0x002000	0xff	1 0
9	0x000000	0x00	0 0
a	0x001000	0x0f	1 1

P2:

seg	base	bounds	R W
8	0x000000	0x00	0 0
9	0x002000	0xff	1 1
a	0x003000	0x0f	1 1

```
P1: 0x802070 read:
P2: 0x902070 read:
P2: 0xa00100 read:
```



DISADVANTAGES OF PAGING AND SEGMENTATION

Potentially large page tables (for each segment)

- Must allocate each page table contiguously
- More problematic with more address bits
- Page table size?

  Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

  Each page table is:

  = Number of entries * size of each entry

  = Number of pages * 4 bytes

  = 2^{18} * 4 bytes = 2^{20} bytes = 1 MB!!!

OTHER APPROACHES

1. ~~Segmented Pagetables~~
2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
3. Inverted Pagetables

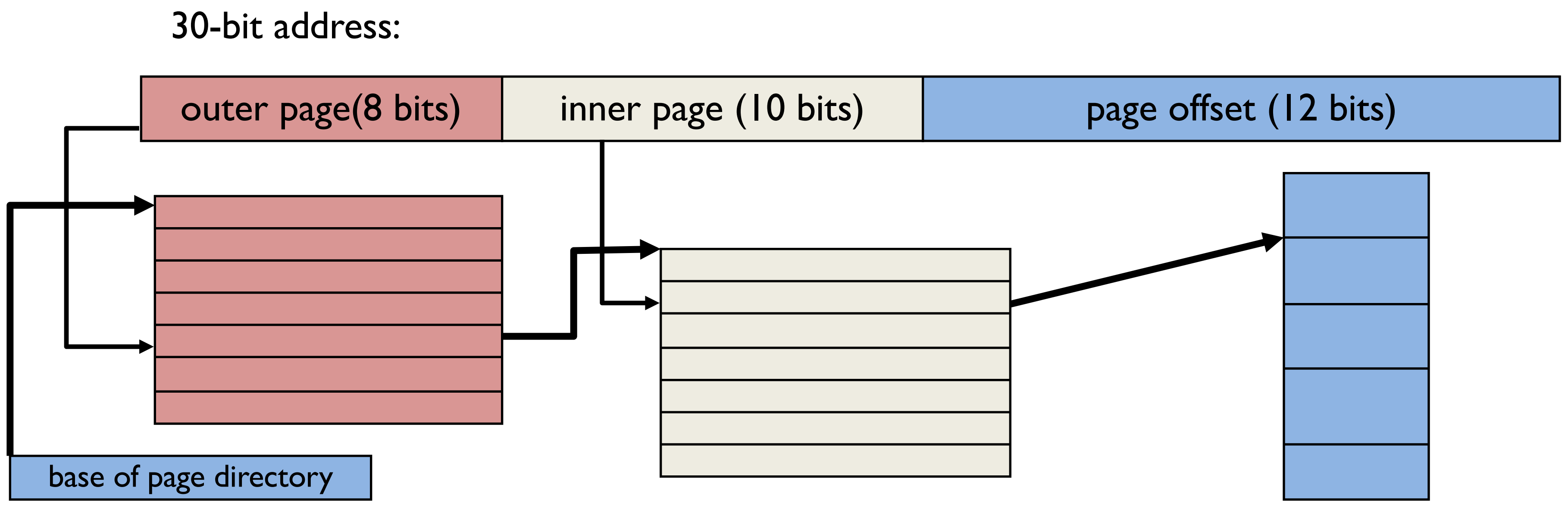
MULTILEVEL PAGE TABLES

Goal: Allow each page tables to be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level "page directory"
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

MULTILEVEL PAGE TABLES



MULTILEVEL EXAMPLE

page directory

PPN	valid
0x3	1
-	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

page of PT (@PPN:0x92)

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

translate 0x01ABC

0x23ABC

20-bit address:

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

NEIGHBOR CHAT: MULTILEVEL

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

page of PT (@PPN:0x92)

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

translate 0x04000

translate 0xFEED0

20-bit address:

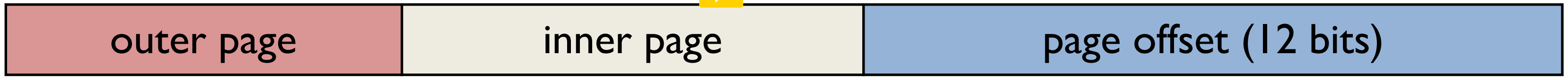
outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

CANVAS HOMEWORK

`paging-multilevel-translate.py`

ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:



How should logical address be structured? How many bits for each paging level?

Goal?

– Each page table fits within a page

– PTE size * number PTE = page size

  Assume PTE size = 4 bytes

  Page size = 2^12 bytes = 4KB

– How many page table entries can we fit on page?

– 4KB / 4bytes = 1K (1024) entries

→ # bits for selecting inner page =

→ 10

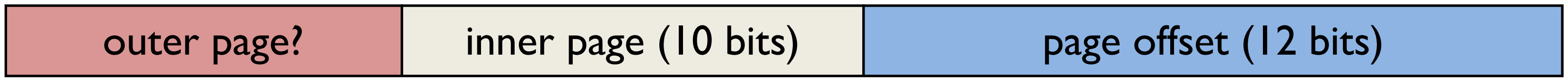
Remaining bits for outer page:

30 -10 -12 = 8 bits

PROBLEM WITH 2 LEVELS?

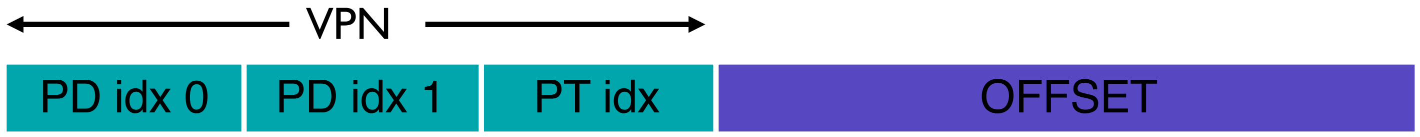
Problem: page directories (outer level) may not fit in a page

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)

4KB / 4 bytes → 1K entries per level

1 level:	
2 levels:	
3 levels:	

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: movl 0x1111, %edi

Total:8

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

Total:1

(c) 0x0519: movl %edi, 0xFF10

Total:5

OTHER APPROACHES

1. ~~Segmented Pagetables~~
2. ~~Multi-level Pagetables~~
   - ~~Page the page tables~~
   - ~~Page the pagetables of page tables…~~
3. Inverted Pagetables

INVERTED PAGE TABLE

Observation:

- Only need entries for virtual pages w/ valid physical mappings
- Have entries based on physical pages (→ inverted!)

Naïve approach:

- Search through data structure <ppn, vpn+asid> to find match
- Too much time to search entire structure

Faster:

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

TLB still manages most cases

- Managing inverted page table requires software-controlled TLB

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

Next Topic:
What if desired address spaces do not fit in physical memory?