CS 423
Operating System Design: TLBs and More Page Tables
Feb 17

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

MP0 grades released (on Canvas)

MP1 due 2/25 - Utilize TA office hours if you need help

AGENDA / LEARNING OUTCOMES

Wrap up discussion on TLBs

Smaller page tables: tackle memory overheads of page tables

RECAP

Disadvantages of Paging

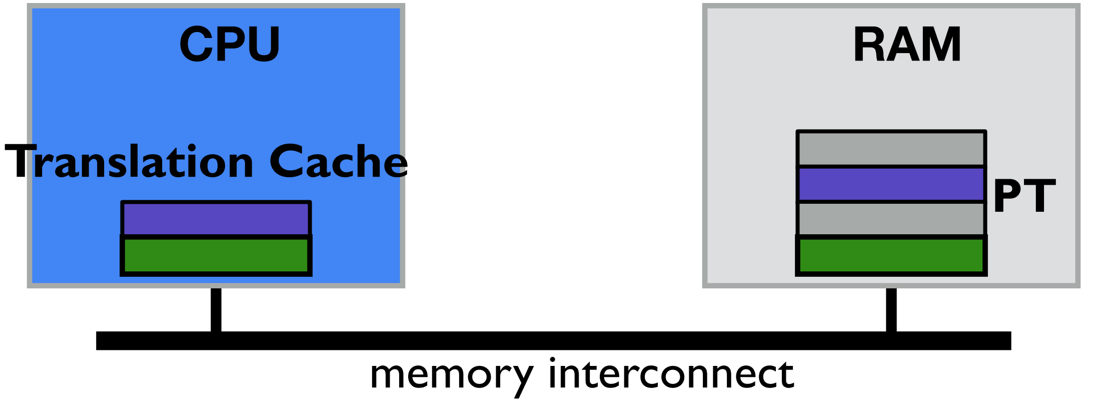
Additional memory reference to page table entry

- Extra memory access needed for each memory access!
- Inefficient, even if page table is stored in memory
- MMU stores only base address of page table
  - Solution: TLBs

Space needed for page tables is too large

- Simple page table: requires PTE for each virtual page number
- PTE needed even if that page is never allocated
- Page tables must be contiguously allocated
  - Solution: paging the page tables!

TLB: CACHE PAGE TRANSLATIONS



TLB: TRANSLATION LOOKASIDE BUFFER

PAGE TRANSLATION WITH TLB

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

HW AND OS ROLES

Who handles TLB hit?

Who handles TLB miss?

TLB MISS - HW AND OS ROLES

If HW, then HW must know where the PT is in memory

- CR3 in x86; page table structure agreed upon between OS and HW
- Hardware "walks" page table structure, fills in TLB

If OS ("software managed TLB")

- HW traps into OS (kernel mode) upon TLB miss
- OS walks page table structure (designed by OS writers), fills in TLB
- Returns to user-mode

Retry the same instruction in the user-process => TLB hit

How to replace TLB entries?

Standard policy problem in any caching solution

LRU: Evict least-recently-used TLB entry
    Needs per-entry bits to track that

Another simple option: Random

TLB Summary

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

Cache recent page translations in TLB

- MMU performs TLB lookup on every memory access

TLB performance depends strongly on workload

- Sequential workloads perform well
- Workloads with temporal locality can perform well

TLB increases cost of context switches

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

TLB miss handling: hardware or OS

Trade-off: speed (can stay in kernel mode!) vs flexibility

END RECAP

Today’s class: space needed for PT

Space for entire page tables can be substantial

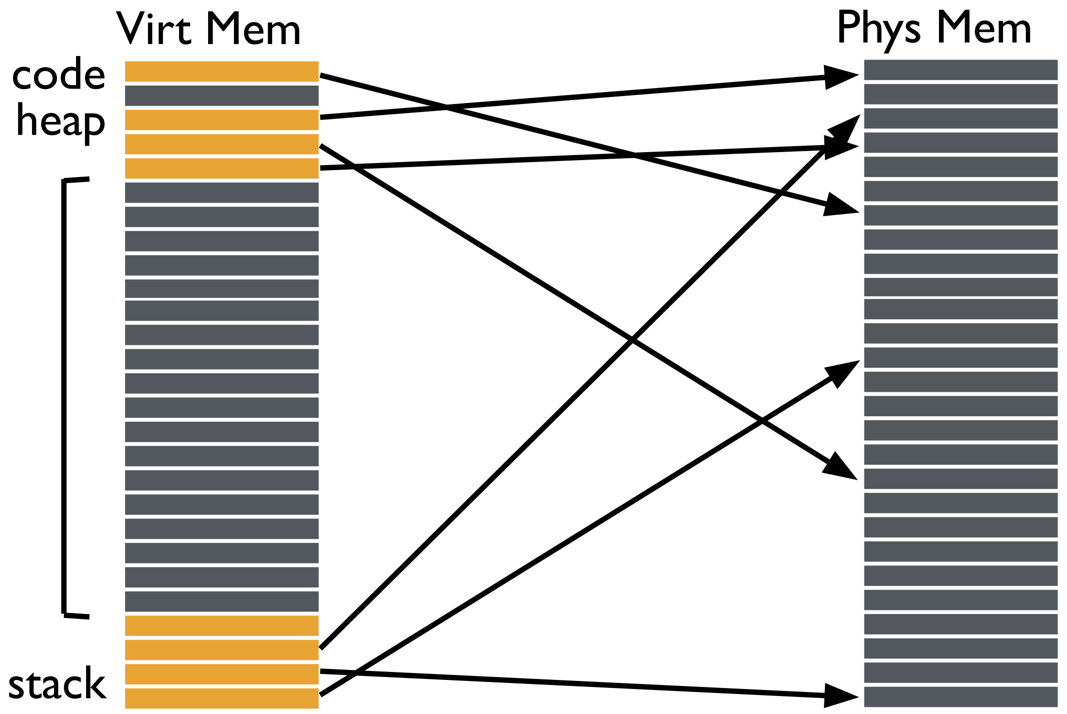
- Simple page table: Require PTE for all pages in address space
- Better ways to save space…

Why are Page Tables so Large?

Unallocated

Unused

But PT entries exist



MANY INVALID PTES

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?

Problem: linear PT must still allocate PTE for each page (even unallocated ones)

DON’T NEED A FLAT PAGE TABLE

Note: page table is just another data structure

Use more complex page tables, instead of just big array

Any data structure is possible if software (OS) handles TLB miss

- MMU looks up vpn in TLB on every memory access
- On TLB miss
  - Trap into OS and let OS find vpn -> ppn translation (PTE)
  - OS inserts PTE into TLB

VARIOUS APPROACHES

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

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?

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

How to avoid allocating holes in the page table?

Combination: Segmented Paging

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)

- Each segment has a page table
- Track base physical address and bounds of the **page table** per segment
  - This is different from original segmentation
  - Bounds => # PTEs in segment

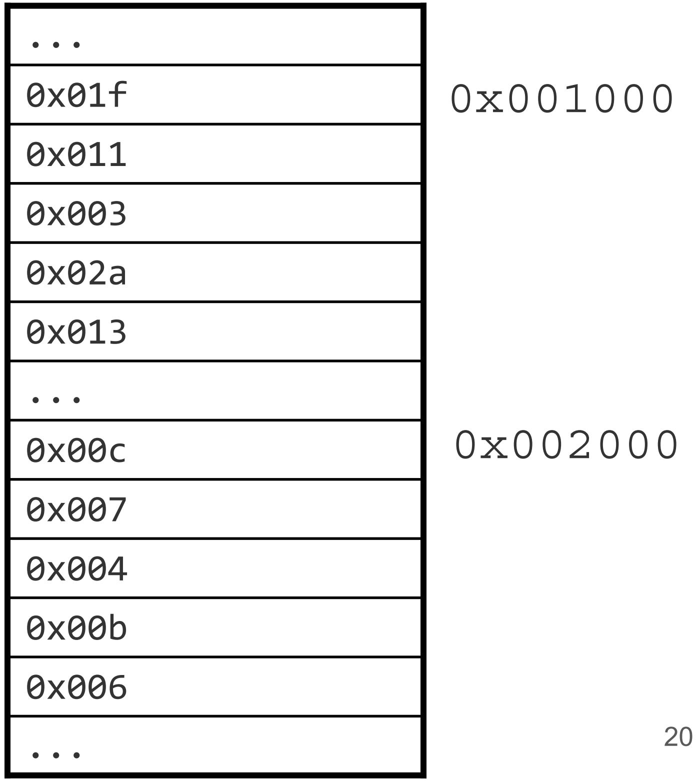
Segmented Paging

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

Bounds = #PTEs

```
0x002070 read: 0x004070
0x202016 read: 0x003016
0x104c84 read: Error (bounds or rw perm)
0x010424 write: Error (read-only perm)
0x210014 write: Error (bounds 0x10 > 0x0f)
0x203568 read: 0x02568
```



Advantages & Disadvantages

Advantages:
- Reduces external fragmentation (unlike Segmentation)
- Segments can grow without the need to reshuffle

Disadvantages:
- Must allocate page table for each segment *contiguously*
- Page table size?
  - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table can grow to:

= 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 paging
2. Multi-level page tables
   - Page the page tables
   - Page the page tables of page tables…
3. Inverted page tables

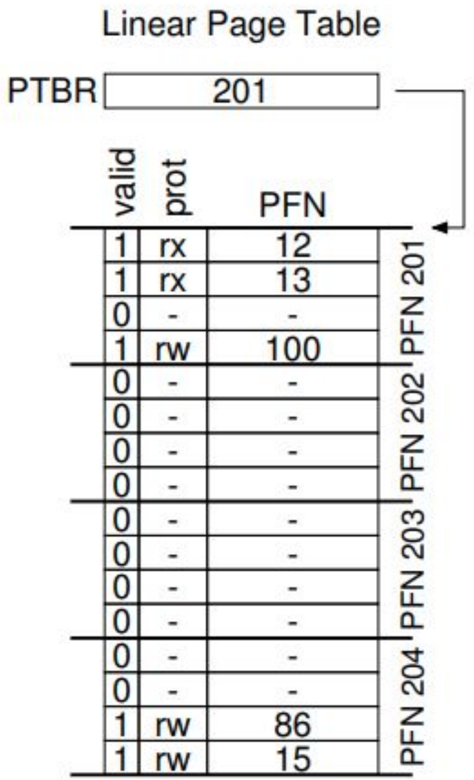
Multi-level Page Tables

Goal: Allow page table to be allocated non-contiguously

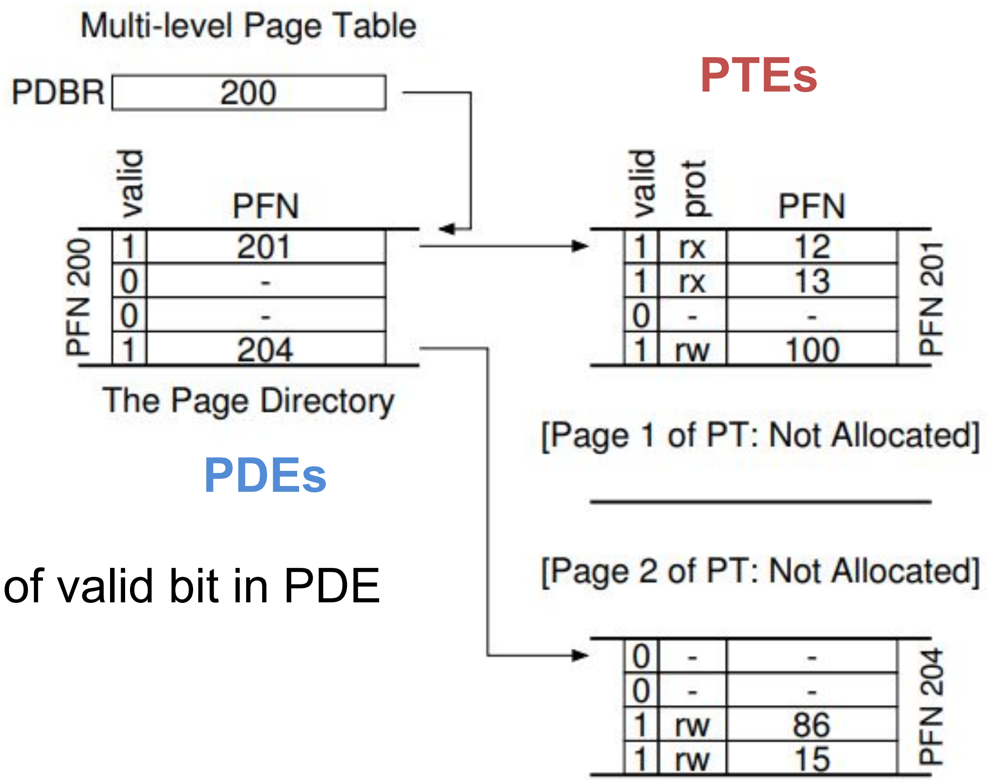
Idea: Page the page tables!

- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Implemented in h/w in 32-bit x86 32-bit
  - 10-10-12 bit split of virtual address

Multilevel Page Table – Key Idea



valid	prot	PFN
1	rx	12
1	rx	13
0	-	-
1	rw	100
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
0	-	-
1	rw	86
1	rw	15



The Page Directory

valid	PFN
1	201
0	-
0	-
1	204

valid	prot	PFN
1	rx	12
1	rx	13
0	-	-
1	rw	100

valid	prot	PFN
0	-	-
0	-	-
1	rw	86
1	rw	15

Meaning of valid bit in PDE and PTE

Multilevel Page Tables

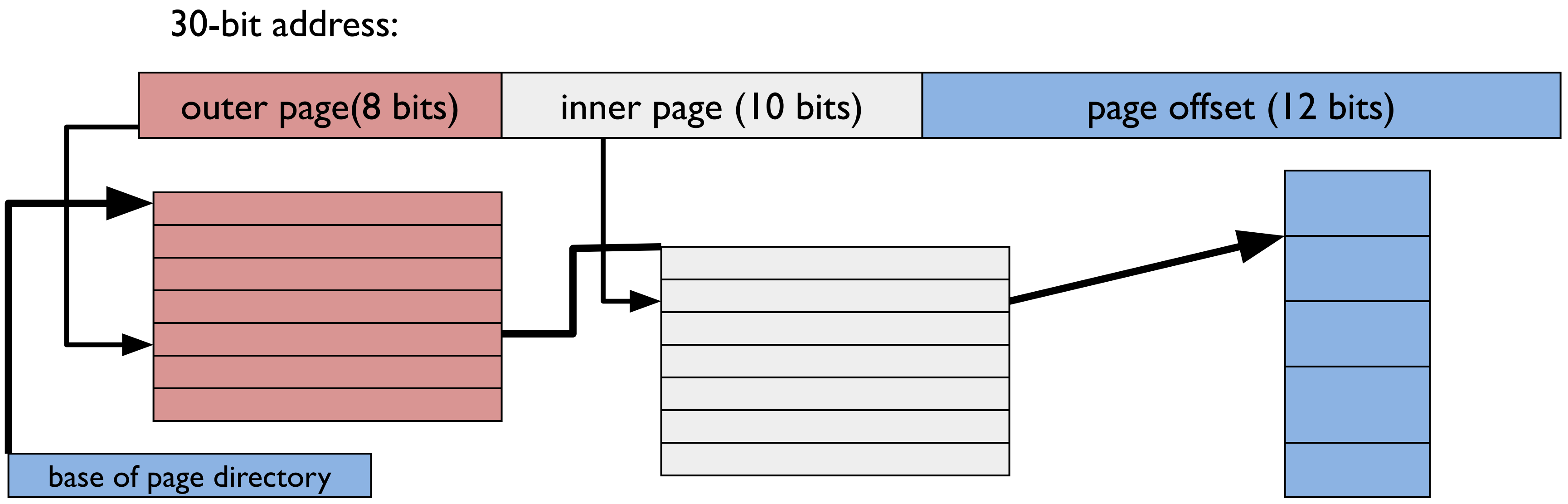
Chat for 1 minute with your neighbors…

Can pages of the PT be dynamically relocated to a different physical location?

Can the PD be dynamically relocated to a different physical location?

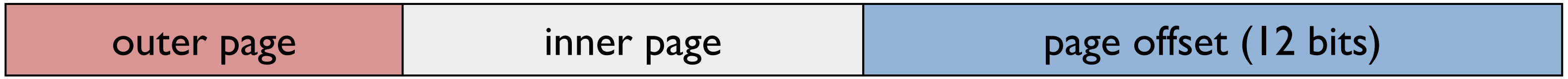
How many memory accesses on a TLB miss with this 2-level PT?

Multilevel Page Tables



Address format for multilevel Paging

30-bit address:



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

Page size dictates #bits for offset

- 4KB page => 12 bit offset

Goal: each inner page table fits within a page

- PTE size * number PTEs = page size
- Assume PTE size = 4 bytes
- Each inner page can have 1024 PTEs
- => 10 bits for inner page

Remaining bits for outer page:

- => 30 – 12 – 10 = 8 bits

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

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

Physical address?

Look PD[0]

Look PT[1] → arrive at

page 0x23

Concat ABC to arrive at

0x23ABC

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

20-bit address:

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

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

VA: 0x04000

PD[0]->PT[4] = 0x80

Concat(0x80, 000) =

0x80000

VA: 0xFEED0

PD[f]->PT[e] = 0x55

Concat(0x55, ED0) =

0x55ED0

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

20-bit address:

PROBLEM WITH 2 LEVELS?

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

Consider 30 bit address with 512-byte pages

=> 9 bits for offset; leaves 21 bits for VPN

Remember our goal: each inner page’s PTEs should fit within a page

So how many PTEs per page? With 512-byte pages and 4-byte PTE, we can have 128 entries → this means 7 bits for inner page, leaving 14 bits for outer page (or directory) → 2^{14} PDEs

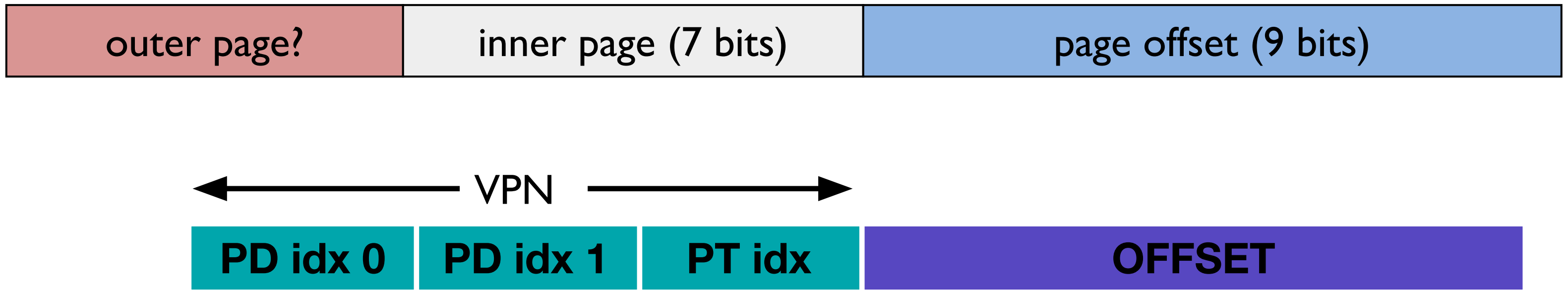
Assume 4-byte PDE, then PD itself will span _128_ pages?

PD cannot be contained in one page now!

PROBLEM WITH 2 LEVELS?

Solution: page the page directory!

Add another level of page directory that points to PD pages



Can keep going recursively! Let page = 4KB (offset is 12 bits); 1K PTEs/page

2 level tree: 1K * 1K * 4K = 2^32 = 4GiB (10 + 10 + 12 bits)

3 level tree: 1K * 1K * 1K * 4K = 2^42 = 4 TiB (10 + 10 + 10 + 12 = 42 bits)

#Memory Accesses Exercise

Assume 3-level page table

Assume 256-byte pages

Assume 16-bit addresses

Assume ASID of current process is 211

How many physical accesses for each instruction?

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

(Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

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

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

FULL SYSTEM WITH TLBS

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)

1. 0xaa: (TLB miss -> 3 for addr trans) + 1 instr fetch
   0x11: (TLB miss -> 3 for addr trans) + 1 movl

2. 0xbb: (TLB hit -> 0 for addr trans) + 1 instr fetch from 0x9113

3. 0x05: (TLB miss -> 3 for addr trans) + 1 instr fetch
   0xff: (TLB hit -> 0 for addr trans) + 1 movl into 0x2310

On TLB miss: lookups with more levels more expensive

Inverted Page Table

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

Better:

- Maintain a hash-map of vpn+asid to ppn
- Smaller number of entries to search for exact match

Used in IBM PowerPC

- Typically requires TLB miss handling by software (OS)

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

- Eg. 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 STEPS

Next class: Swapping!