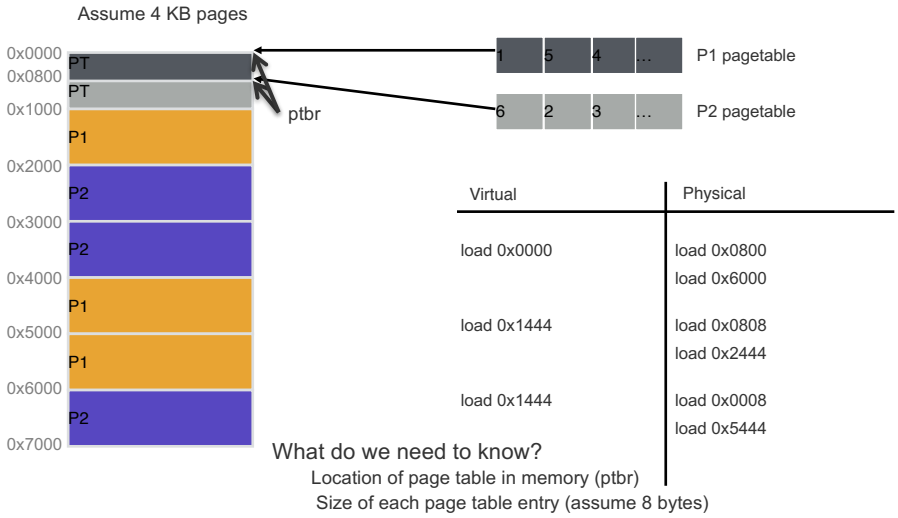
CS 537
Lecture 6
Fast Translation - TLBs

Michael Swift

Faster with TLBS

- **Questions answered in this lecture:**
  - Review paging...
  - How can page translations be made faster?
  - What is the basic idea of a TLB (Translation Lookaside Buffer)?
  - What types of workloads perform well with TLBs?
  - How do TLBs interact with context-switches?

Review: Paging



Review: Paging PROS and CONS

Advantages

- No external fragmentation
  - don't need to find contiguous RAM
- All free pages are equivalent
  - Easy to manage, allocate, and free pages

Disadvantages

- Page tables are too big
  - Must have one entry for every page of address space
- Accessing page tables is too slow [today's focus]
  - Doubles number of memory references per instruction

Translation Steps

H/W: for each mem reference:

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

Which steps are expensive?

Which expensive step will we avoid in today's lecture?

3) Don't always have to read PTE from memory!

Example: Array Iterator

```
int sum = 0;
for (i=0; i<N; i++){
      sum += a[i];
}
```

Assume 'a' starts at 0x3000
Ignore instruction fetches

What virtual addresses?	What physical addresses?
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
	load 0x700C
...	

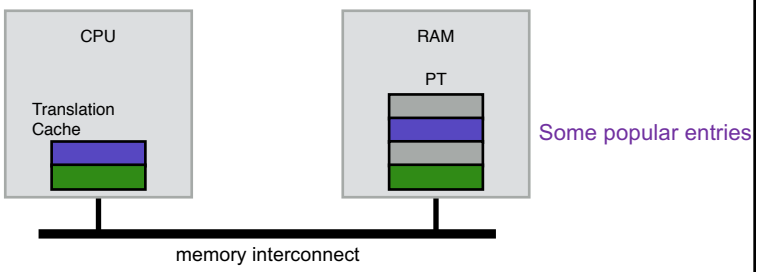
Aside: What can you infer?
- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 -> PPN 7

Observation:
Repeatedly access same PTE because program repeatedly accesses same virtual page

Pages of Page tables

- Page tables are stored in memory
  - Memory is made up of pages
- Page tables are made of of pages (more on Thursday!!!)
- Each page of a page table has some number of translations
  - Sizeof(page)/sizeof(PTE) == # of translations in a page
  - Example: 4kb pages, 32 bit addresses
    - Sizeof(pte) = 20 bits + metadata == 32 bits / 4 byte
    - 1024 PTEs in a page
- Example: 64kb pages, 64 bit addresses
  - Sizeof(PTE) = 48 bits + metadata = 64 bits / 8 bytes
  - 8192 PTEs/page

Strategy: Cache Page Translations



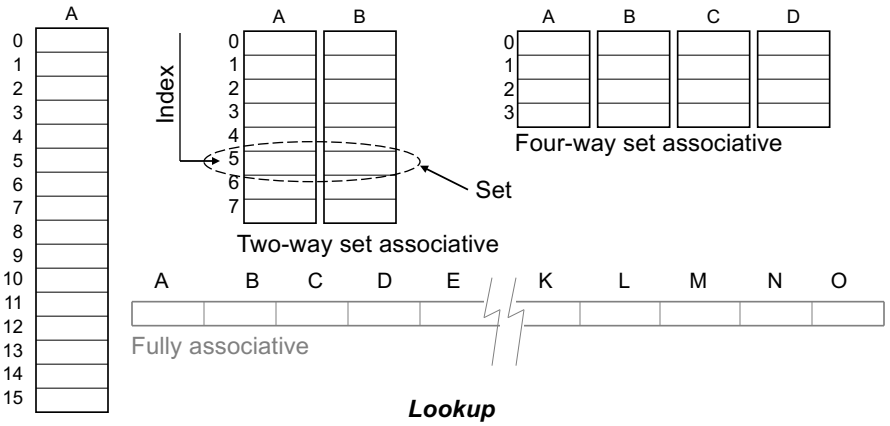
TLB: **T**ranslation **L**ookaside **B**uffer
(yes, a poor name!)

TLB Organization

TLB Entry

Tag (virtual page number)	Physical page number (page table entry)

Various ways to organize a 16-entry TLB (artificially small)



Lookup

- Calculate set (tag % num_sets)
- Search for tag within resulting set

TLB Associativity Trade-offs

Higher associativity

+ Better utilization, fewer collisions
– Slower
– More power

Lower associativity

+ Fast
+ Simple, less hardware
– Greater chance of collisions

TLBs usually set-associative

- x86: 4-way
- ARM (cell phones: 2-way

HW and OS Roles

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

H/W: CPU must know where pagetables are

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

OS: CPU traps into OS upon TLB miss

- "Software-managed TLB"
- OS interprets pagetables as it chooses
- Modifying TLB entries is privileged
  - otherwise what could process do?

Need same protection bits in TLB as pagetable
- rwx

Array Iterator (w/ TLB)

```
int sum = 0;
for (i = 0; i < 2048; i++){
     sum += a[i];
}
```

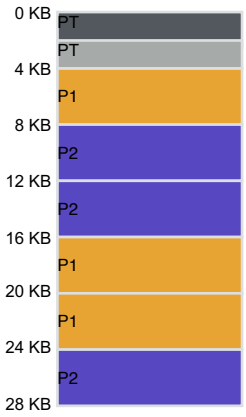
Assume following virtual address stream:
load 0x1000

load 0x1004 What will TLB behavior look like?

load 0x1008

load 0x100C
…

TLB Accesses: SEQUENTIAL Example



PTBR

P1 pagetable

CPU's TLB

1	1	5
1	2	4

Virt	Phys
load 0x1000	load 0x0004
load 0x1004	load 0x5000 (TLB hit)
load 0x1008	load 0x5004 (TLB hit)
load 0x100c	load 0x5008 (TLB hit)
…	load 0x500C
load 0x2000	…
load 0x2004	load 0x0008
	load 0x4000 (TLB hit)
	load 0x4004

Performance of TLB?

```
int sum = 0;
for (i=0; i<2048; i++) {
      sum += a[i];
}
```

Calculate miss rate of TLB for data:
TLB misses / # TLB lookups

TLB lookups?
= number of accesses to a = 2048

TLB misses?
= number of unique pages accessed
= 2048 / (elements of 'a' per 4K page)
= 2K / (4K / sizeof(int)) = 2K / 1K
= 2

Miss rate?
2/2048 = 0.1%

Hit rate? (1 – miss rate)
99.9%

Would hit rate get better or worse with smaller pages?
Worse

TLB PERFORMANCE with Workloads

Sequential array accesses almost always hit in TLB

- Very fast!

What access pattern will be slow?

- Highly random, with no repeat accesses

Workload Access Patterns

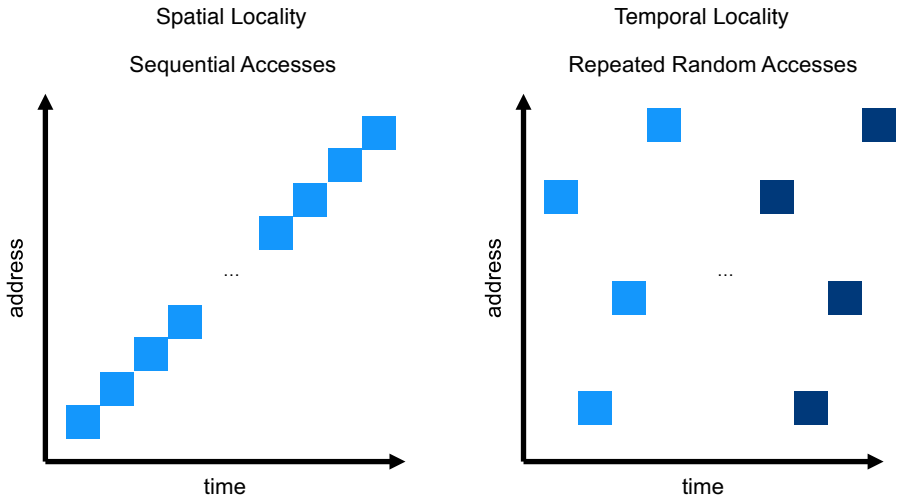
Workload A

```
int sum = 0;
for (i=0; i<2048; i++) {
      sum += a[i];
}
```

Workload B

```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
```

Workload ACCESS PATTERNS



Workload Locality

Spatial Locality: future access will be to nearby addresses
Temporal Locality: future access will be repeats to the same data
What TLB characteristics are best for each type?
Spatial:

- Access same page repeatedly; need same vpn->ppn translation
- Same TLB entry re-used

Temporal:

- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?

TLB Performance

Context switches are expensive

Even with ASID, other processes "pollute" TLB

- Discard process A's TLB entries for process B's entries

Architectures can have multiple TLBs

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for "super pages"

Multi-level TLB

- Problem:
  - First level TLB is part of processor pipeline, timing is critical
  - CPU looks up VPN in TLB before accessing cache
    - Bigger, slower TLB → slower memory access
    - Already 4 cycles in x8t6
- Solution: 2-level TLB
  - Intel: level 1: 64-entry, 4-way set associative
  - Level 2:1536 entries, 12-way set associative
    - Tends to hold entries from multiple processes
  - Uses ASIDs so not flush large L2 TLB

TLB PERFORMANCE

How can system improve TLB performance (hit rate) given fixed number of TLB entries?

Increase page size

Fewer unique translations needed to access same amount of memory)

Large Pages

- TLB reach
  - How much memory can be translated without a TLB miss?
  - # of entries * size of page
    - 64 entries * 4kb pages = 256 kb
    - 1025 entries* 4kb pages = 4 mb
  - Modern machines: 16 GB+ -> 4 million entries
- Solution: larger pages
  - x86: 2mb, 1 GB pages

Multi-size TLBs

- Fully associative:
  - Each entry has a size field
  - Use “don’t care” fields in matching
    - 4kb: 0x12345---
    - 2mb:0x12***---
- Set associative
  - Separate TLB for each sizes
  - X86: 2mb, 1gb pages
    - 32 entries for 2mb -> 64MB without a TLB moss
      - 1536 L2 entries -> 3GB without a page walk
    - 12 entries for 1gb -> 12GB without a miss

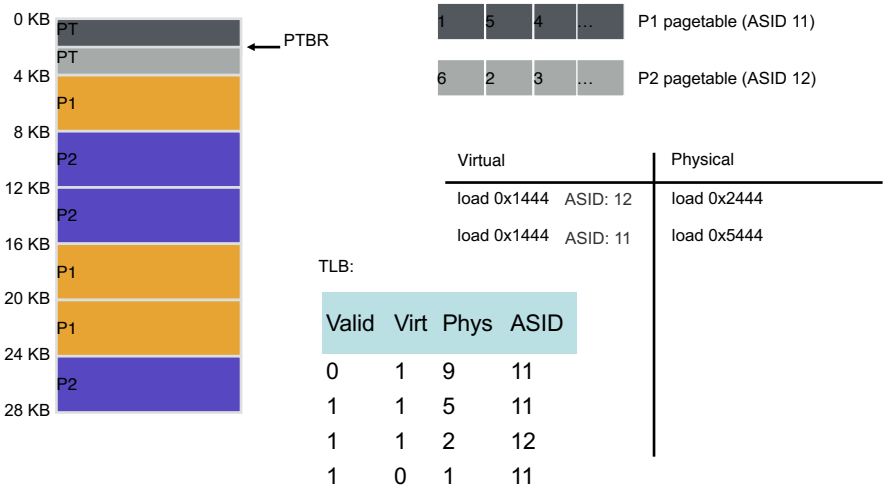
Context Switches

What happens if a process uses cached TLB entries from another process?

Solutions?

1. Flush TLB on each switch
   - Costly; lose all recently cached translations
2. Track which entries are for which process
   - Address Space Identifier
   - Tag each TLB entry with an 8-bit ASID
     - how many ASIDs do we get?
     - why not use PIDs?

TLB Example with ASID



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
  - Increase **TLB reach** by increasing page size
- 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