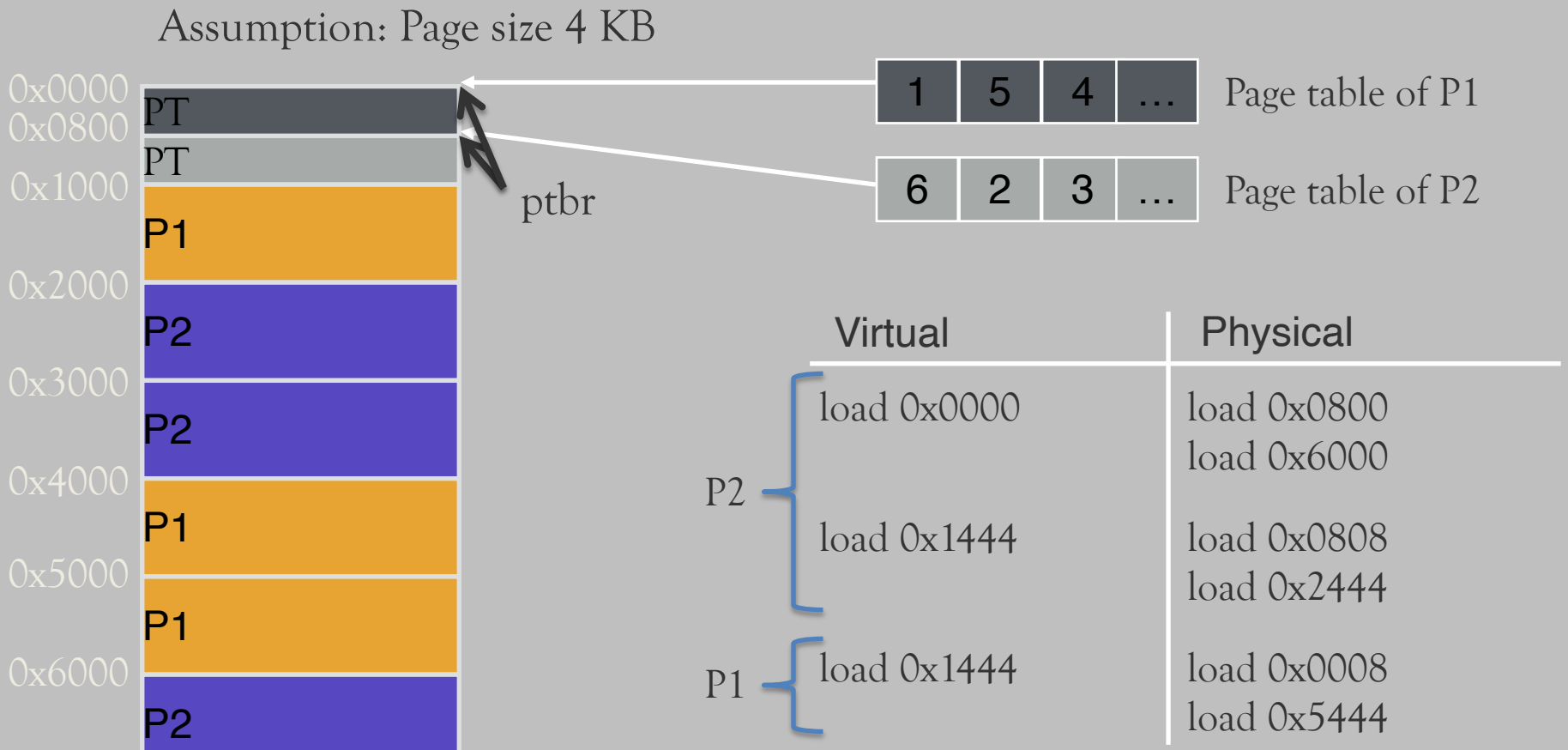
Memory Virtualization:
Faster with TLBs

OSTEP Chapter 19:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf

Jan Reineke
Universität des Saarlandes

Review: Paging



What do we need to know for translation?

- Location of page table in memory (ptbr)
- Size of each page table entry (here: 8 Byte)

Paging: Pros and Cons

Advantages:

- No external fragmentation:
  - free memory does not have to be allocated contiguously
- All free (unallocated) pages are "equal":
  - easy to manage, allocate, and free pages

Disadvantages:

- Page table are too big: (*later*)
  - one entry for every page of address space
- Too slow: (*now*)
  - every "virtual" memory access results in two physical ones

Address translation: Step by step

1. Determine VPN (virtual page number) from VA (virtual address)
2. Calculate address of PTE (page table entry)

expensive 3. Read PTE from memory

4. Extract PFN (page frame number)
5. Build PA (physical address) from PFN and offset

expensive 6. Read contents of PA from memory

Quiz: Which steps are expensive?

Goal of TLBs: avoid step 3

Example: Array iterator

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

Assumption: 'a' begins at 0x3000

Ignore instruction fetches

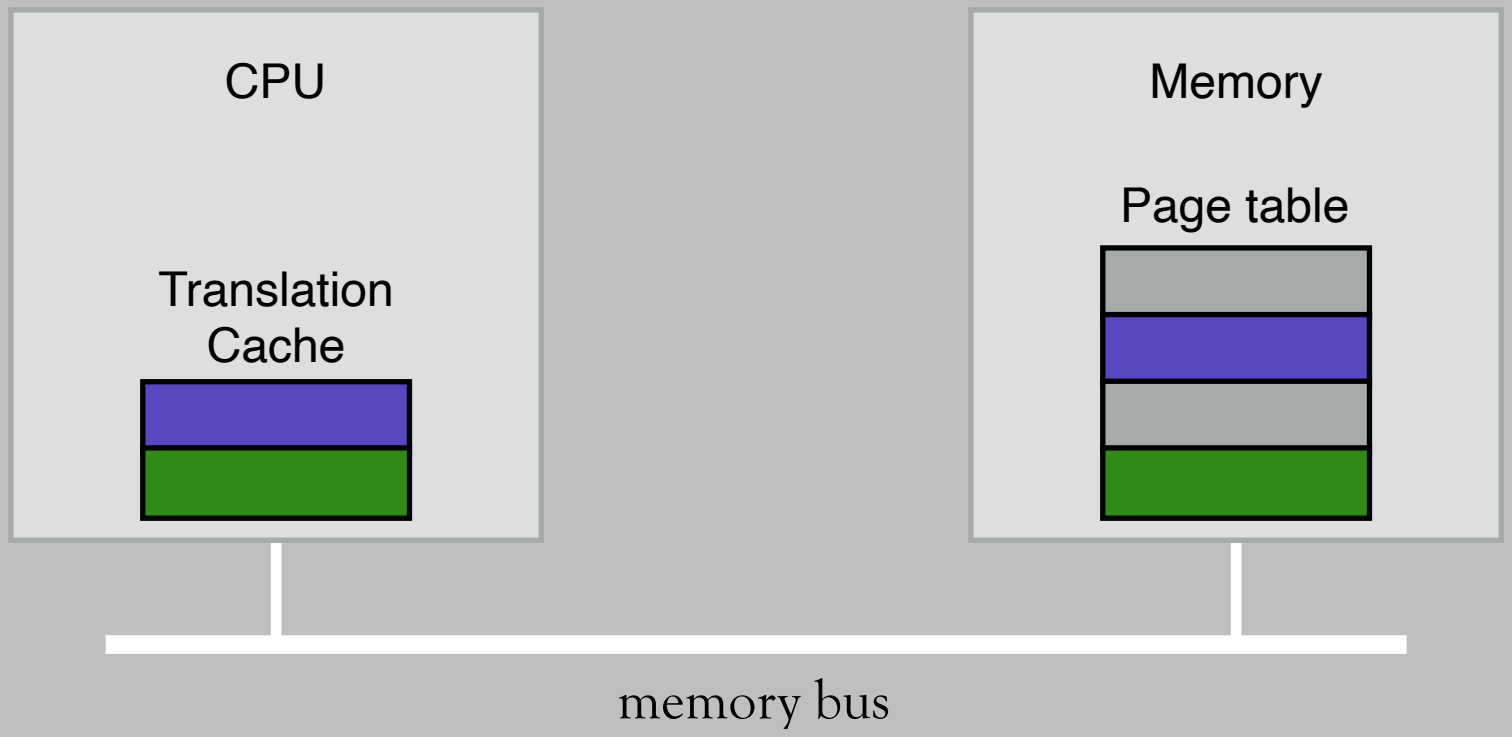
Virtual addresses:	Physical addresses:
load 0x3000	load 0x100C load 0x7000
load 0x3004	load 0x100C load 0x7004
load 0x3008	load 0x100C load 0x7008
load 0x300C	load 0x100C load 0x700C
…	

Observation:
Repeatedly access same PTE of page table

Aside: What can you infer?

- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 → PPN 7

Approach: TLB = Translation "Cache"

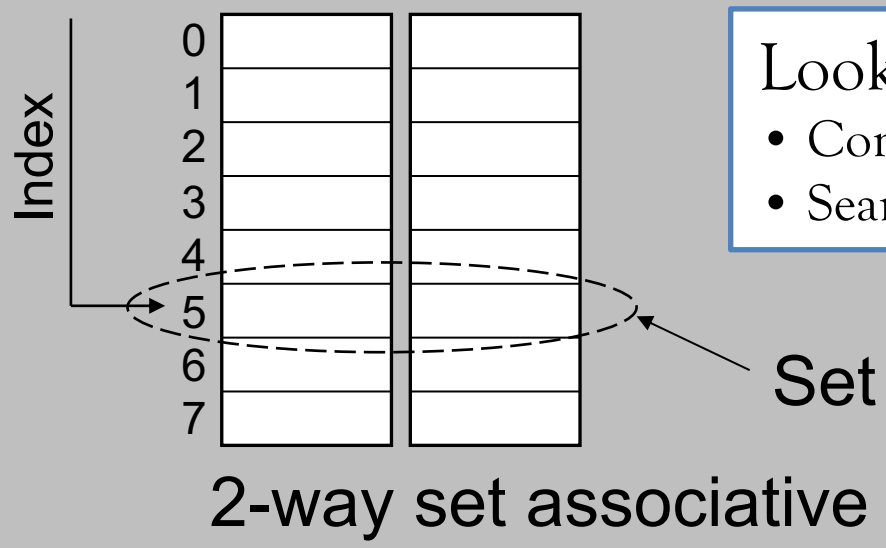


TLB: **T**ranslation **L**ookaside **B**uffer

TLB: Organization

TLB entry:

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



Lookup

- Compute set (tag % number of sets)
- Search for tag within resulting set

Extreme cases:

- Direct mapped: every set contains only one entry
- Fully associative: single set that contains all entries

TLB associativity trade-offs

Higher associativity

+ Fewer collisions
- More hardware

Lower associativity

+ Simple, less hardware
- More collisions

TLBs often small, but fully associative

Array iterator with TLB

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

Assumption: Generates following virtual addresses:

load 0x1000

load 0x1004

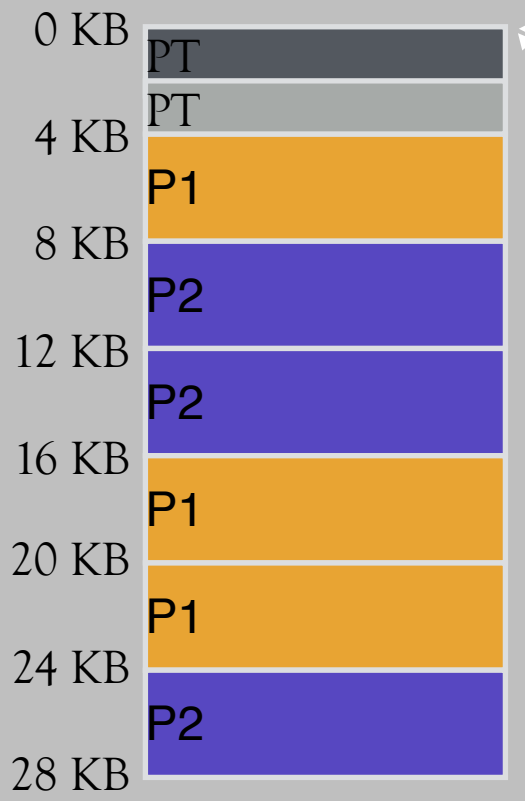
load 0x1008

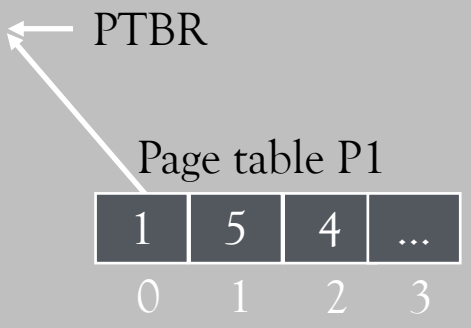
load 0x100C

…

What will TLB behavior look like?

Example: Sequential accesses





TLB state

Valid	VPN	PPN
1	1	5
1	2	4

Virtual	Physical
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	(TLB hit)
	load 0x5004
load 0x1008	(TLB hit)
	load 0x5008
load 0x100c	(TLB hit)
	load 0x500C
...	...
load 0x2000	load 0x0008
	load 0x4000
load 0x2004	(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 accesses:
TLB misses / TLB lookups

TLB lookups = number of accesses to array a = 2048

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

→ Miss rate = 2/2048 = 0.1%
→ Hit rate? (1 – miss rate) = 99.9%

How does the hit rate depend on the page sizes?

TLB performance

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

Increase page size:

→ fewer unique page translations needed to access same amount of memory

→ *but*: potentially more internal fragmentation

TLB performance with different workloads

Sequential array accesses almost always hit in TLB!

What access pattern will be slow?

→ Highly random, with no repeat accesses

Two and a half access patterns

Access pattern A

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

Hit rate ca. 99.9%

Access pattern B

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

Expected hit rate?
TLB size: 1

Depends on N:
Let N = n*1024, then the expected hit rate is exactly 1/n.

Two and a half access patterns

Access pattern A

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

Access pattern B

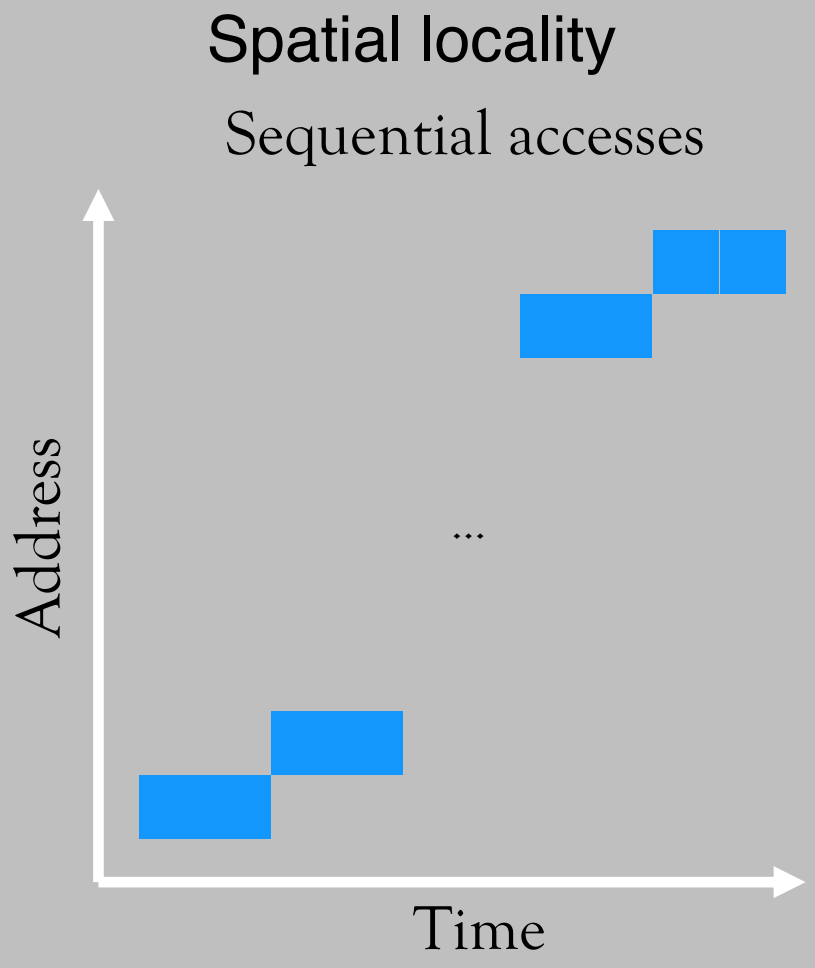
```
int sum = 0;
for (j=0; j<50; j++) {
   srand(1234);
   for (i=0; i<20; i++) {
      sum += a[rand() % N];
   }
}
```

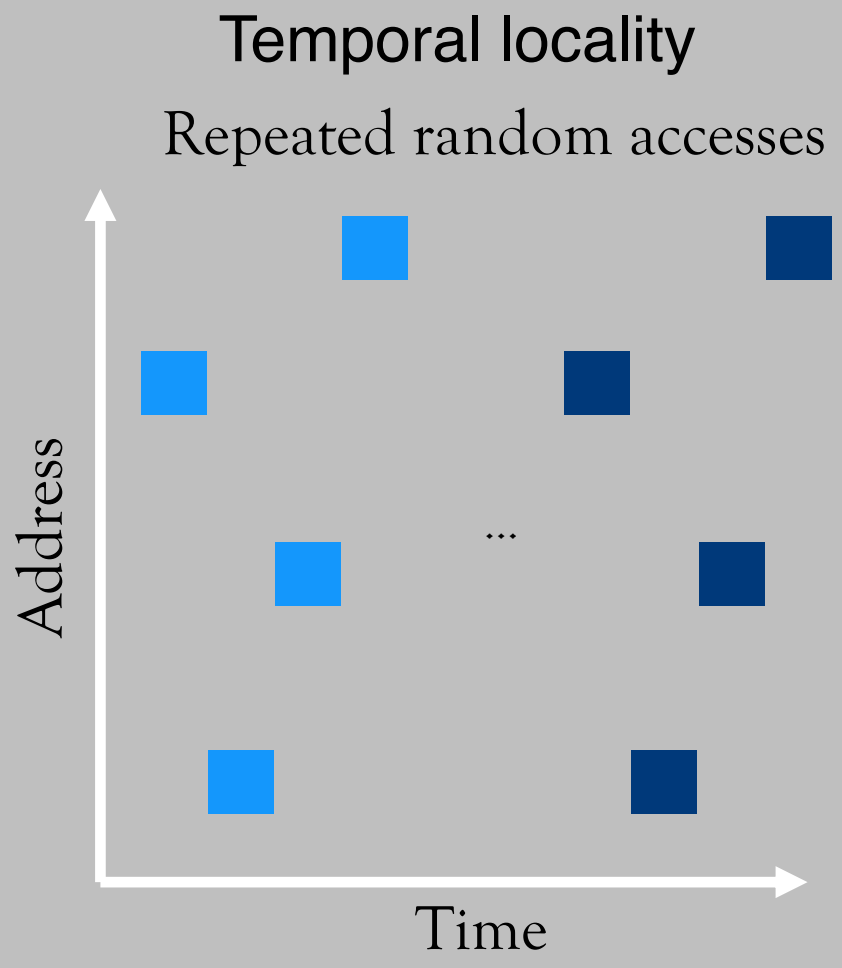
Expected hit rate?

For large N the hit rate depends on the size of the TLB:

- Size of TLB ≥ 20: almost only hits
- Size of TLB < 20: mostly misses

Two access patterns





Locality

Spatial locality:
future accesses will be to nearby addresses

Temporal locality:
future accesses will be repeats to the same address

Which TLB properties determine how well locality is exploited?

The larger the pages, the better spatial locality is exploited.
The more entries, the better temporal locality is exploited.

TLB replacement policies

As with caches:

- LRU: Least-recently-used
  - Evict least-recently-used entry
- FIFO: First-in, First-out
  - Evict entry that has been in the TLB for the longest time
- Farthest-in-the-Future
  - Optimal offline strategy

Context switches

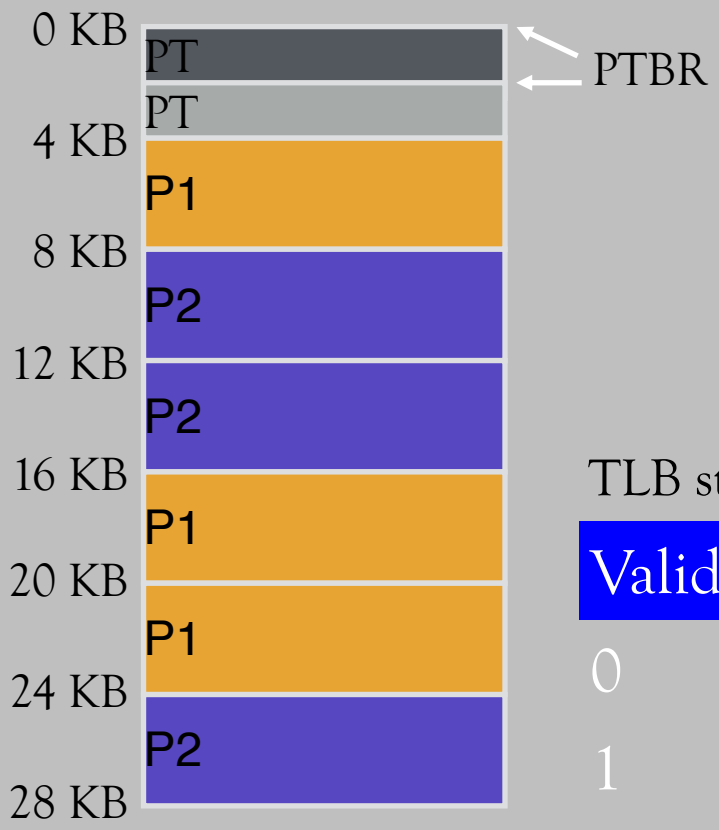
What happens if a process uses cached TLB entries from another process?

Solutions?

Alternatives:

1. Flush TLB on every context switch
2. Track which entries are for which process
   - Address space identifier, ASID
   - Tag each TLB entry with an 8-bit ASID
     - How many ASIDs do we get?
     - Why not PIDs? → typically 32-bit, thus high overhead

Example: TLB with ASIDs



1	5	4	...

Page table of P1 (ASID 11)

6	2	3	...

Page table of P2 (ASID 12)

Virtual	Physical
load 0x1444 ASID: 12	load 0x2444
load 0x1444 ASID: 11	load 0x5444

TLB state:

Valid	Virtual	Physical	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

TLB Performance

Even with ASIDs, context switches are not free

→Process A's TLB entries evict process B's entries, and vice versa

Architectures can have multiple TLBs

- Data TLB + Instruction TLB
- Hierarchy of TLBs
- TLB for regular pages + TLB for "huge pages"

Open questions: HW vs OS

Who handles TLB misses? **HW** or **OS**?

Hardware:

- CPU must know where page tables are
- Page table structure **fixed** and agreed upon between HW and OS
- HW traverses page table and fills TLB

OS: CPU generates exception upon TLB miss

- "Software-managed TLB"
- OS can use **arbitrary** data structures for page table, traverses page table and fills TLB
- Modifying TLB entries is privileged (only in Kernel mode)
  - otherwise, what could process do?

Summary

- Pages are great, but accessing page tables for every memory access is slow
- *Idea*: Cache page translations → TLB
- TLB performance strongly depends on workload
  - Sequential vs random access patterns
  - Larger page sizes increase hit rate,
    but: internal fragmentation
- TLB misses can be handled by either HW or SW