

Virtualizing Memory: Faster with TLBs

Questions answered in this lecture:

Review paging...

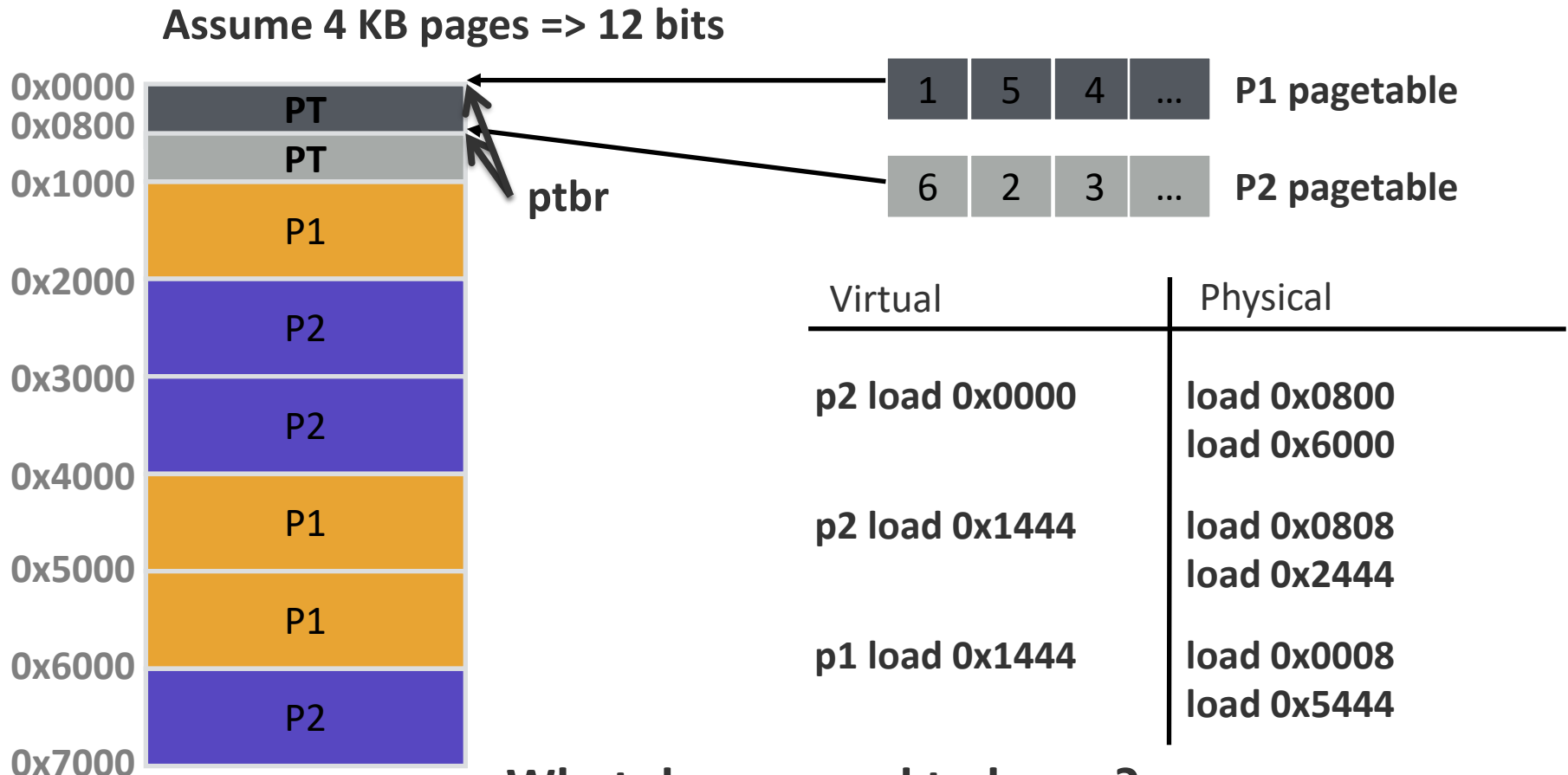
How can page translations be made faster?

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



What do we need to know?

Location of page table in memory (ptbr)

Size of each page table entry (assume 8 bytes)

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

	What virtual addresses?	What physical addresses?
int sum = 0;	load 0x3000	load 0x100C
for (i=0; i<N; i++){		load 0x7000
sum += a[i];	load 0x3004	load 0x100C
}		load 0x7004
Assume 'a' starts at 0x3000	load 0x3008	load 0x100C
Ignore instruction fetches	load 0x300C	load 0x7008
		load 0x100C
	...	load 0x700C

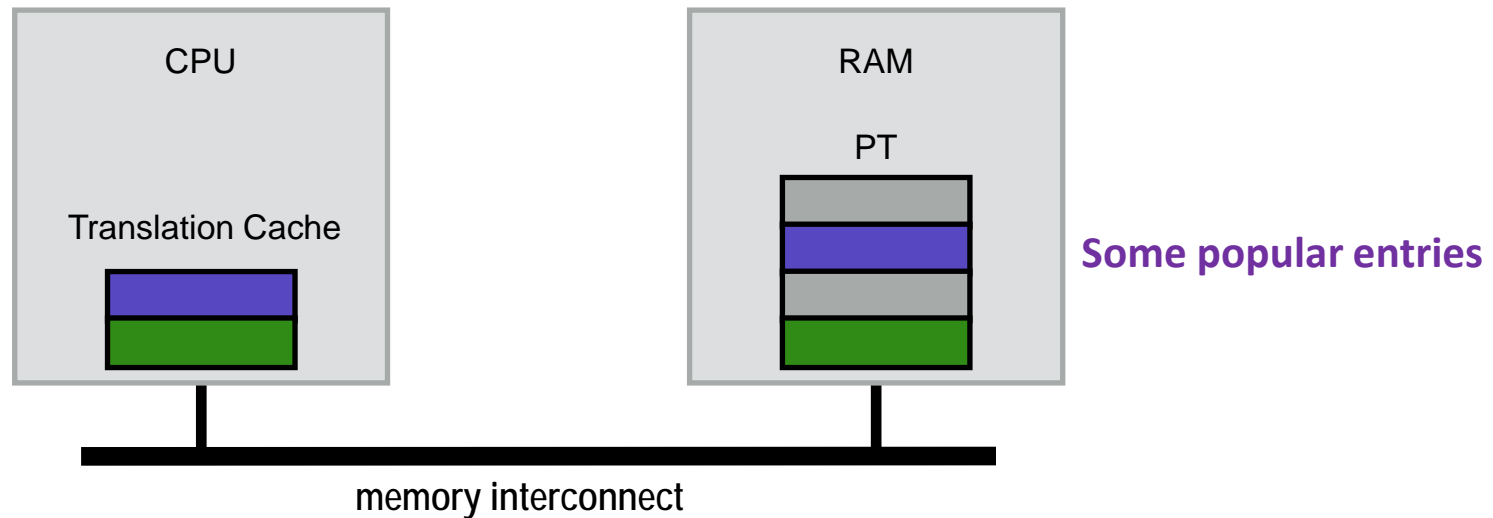
Aside: What can you infer?

- PTE 4 bytes each
- VPN 3 -> PPN 7

Observation:

Repeatedly access same PTE because program repeatedly accesses same virtual page

Strategy: Cache Page Translations



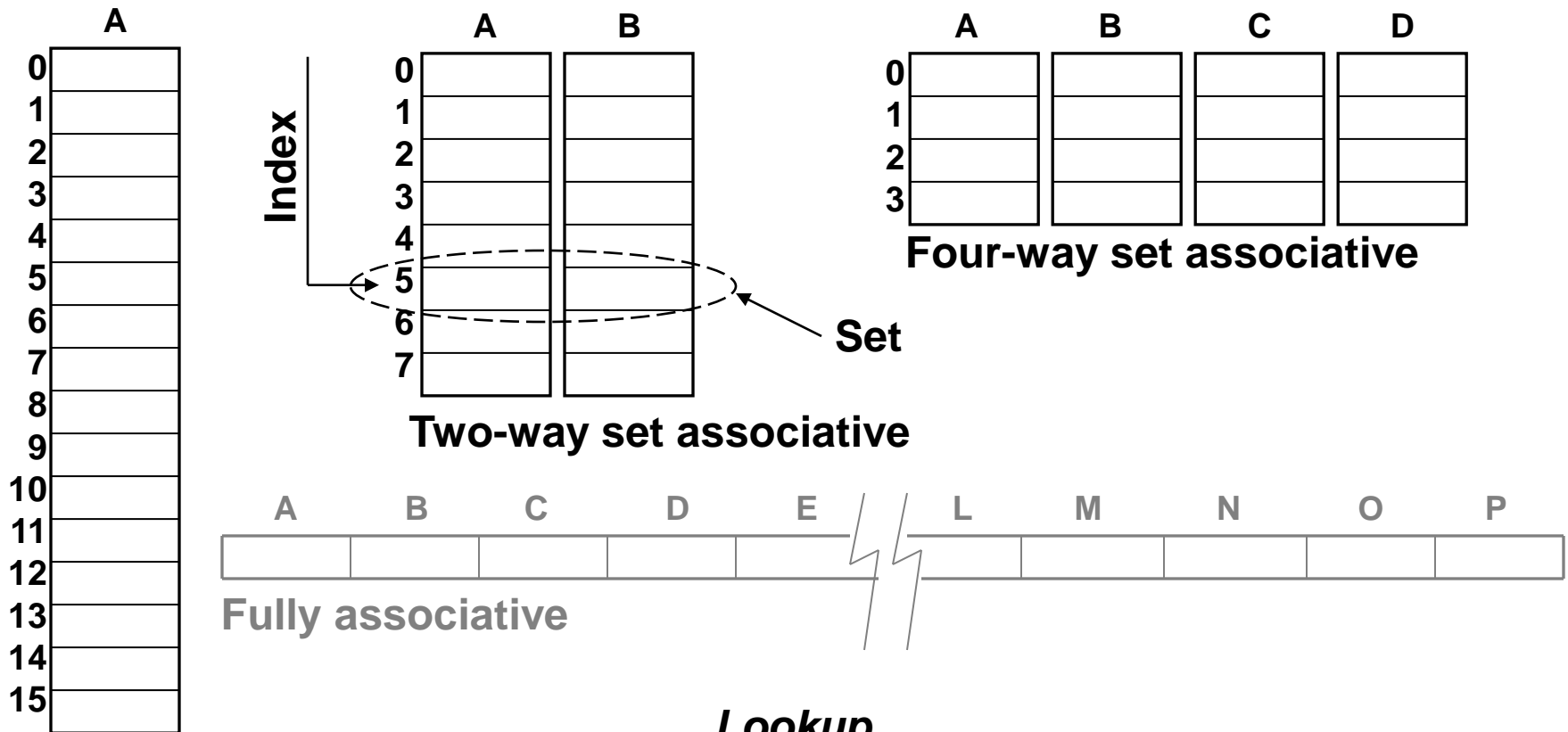
TLB: Translation Lookaside Buffer

TLB Organization

TLB Entry

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

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



Direct mapped

Lookup

- Calculate set ($\text{tag} \% \text{num_sets}$)
- Search for tag within resulting set

TLB Associativity Trade-offs

■ Higher associativity

- + Better utilization, fewer collisions
- Slower
- More hardware

■ Lower associativity

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

■ TLBs usually fully associative

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

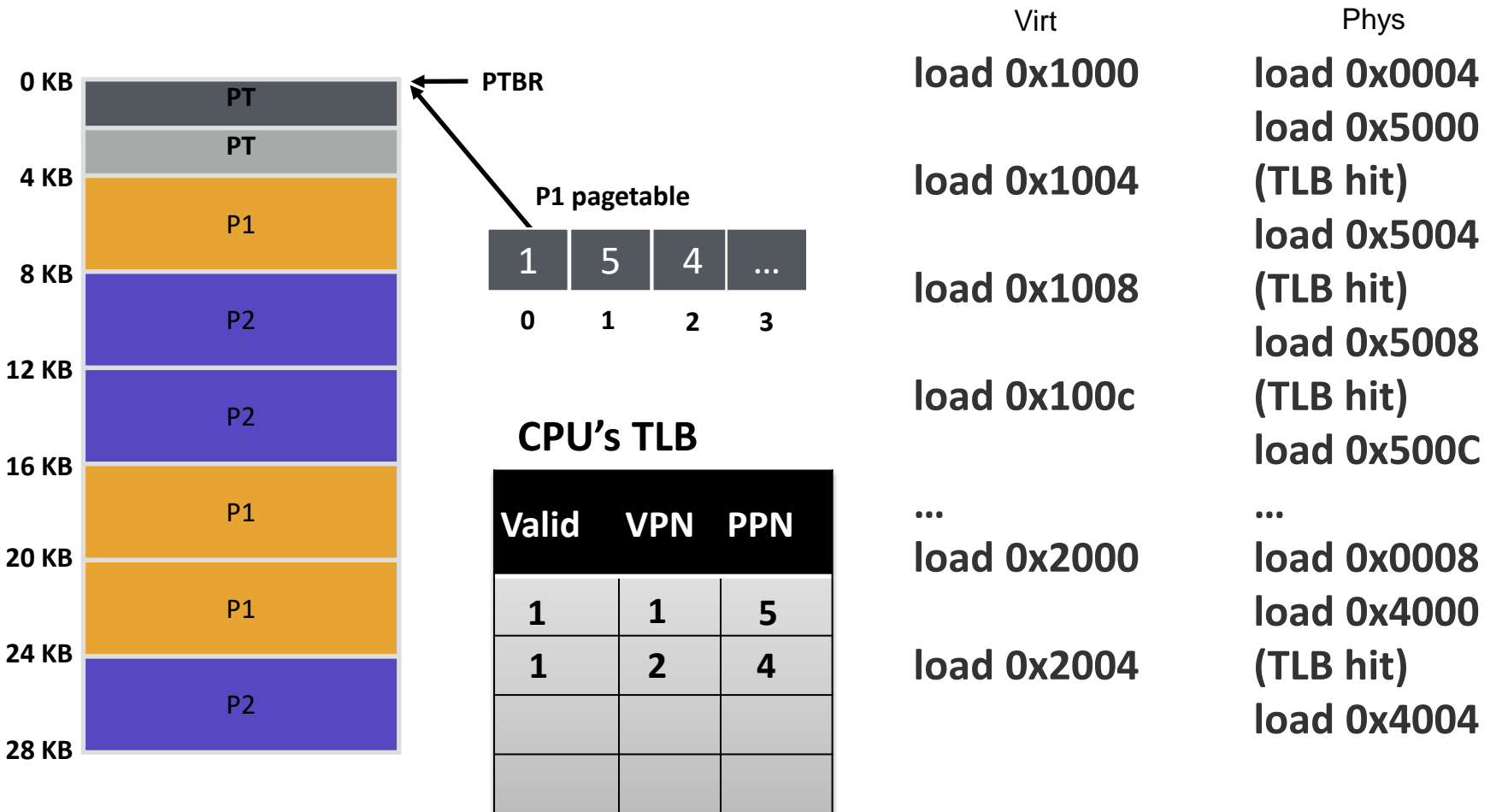
load 0x1008

load 0x100C

...

What will TLB behavior look like?

TLB Accesses: Sequential Example



Can page table exist in cache?

Performance of TLB?

- 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

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

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
 - Huge page in modern processors: 2MB, 1GB
- **TLB Reach:**
 - Number of TLB entries * Page Size

TLB Performance with Workloads

- **Sequential / Repeated** 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];
}
```

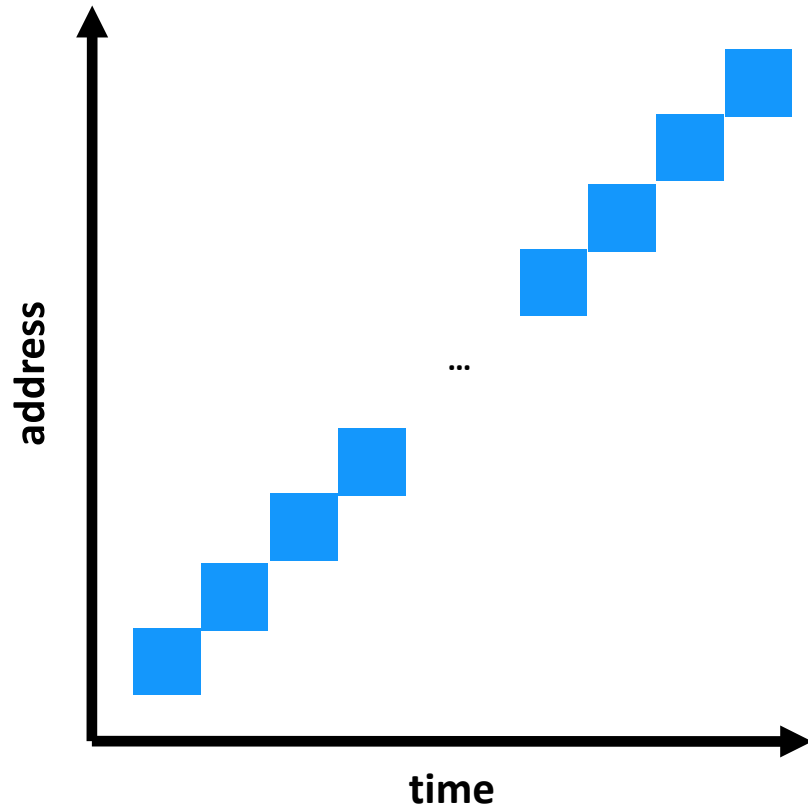
Workload B

```
int sum = 0;
srand(1234);
for (i=0; i< 2048; i++) {
    sum += a[rand() % N];
}
```

Workload Access Patterns

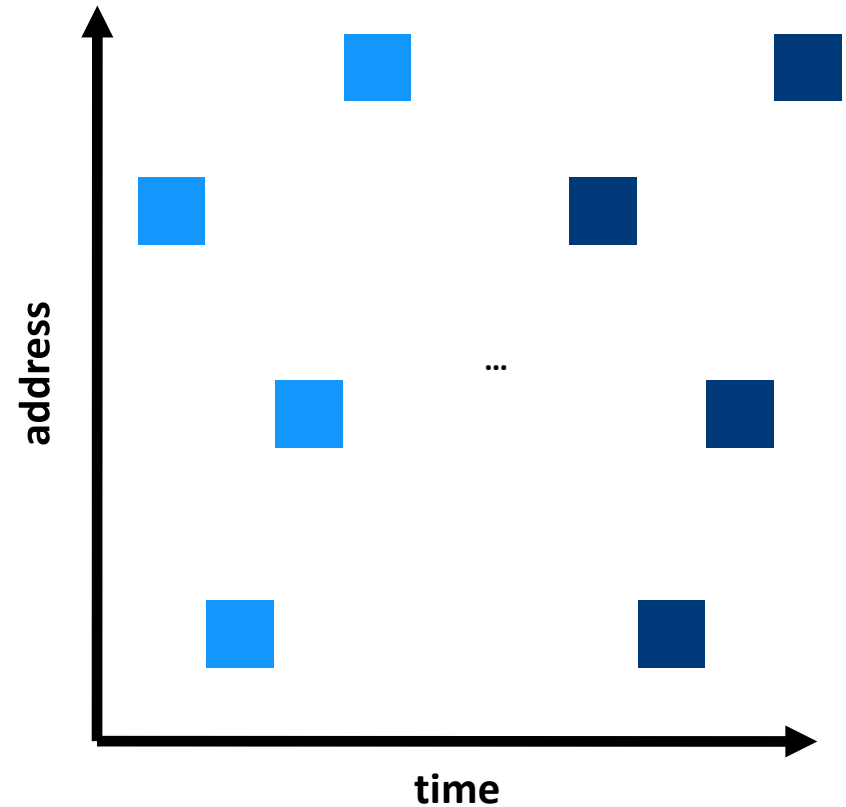
Spatial Locality

Sequential Accesses



Temporal Locality

Repeated Random Accesses



Workload Locality

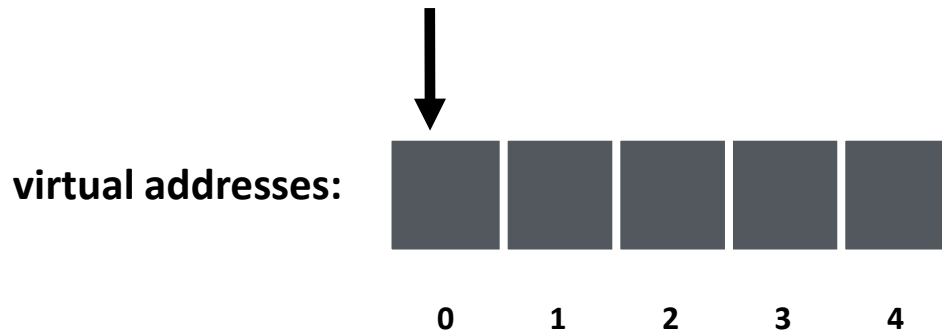
- **Spatial Locality:** future access will be to **nearby addresses**
- **Temporal Locality:** future access will be repeats to the **same data**
- What TLB characteristics are best for each type?
- **Spatial:**
 - Access **same page** repeatedly; need same vpn->ppn translation
 - Same TLB entry re-used
- **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 Replacement Policies

- **LRU**: evict Least-Recently Used TLB slot when needed
 - (More on LRU later in policies)
- **Random**: Evict randomly chosen entry
- Which is better?



LRU Troubles



Valid	Virt	Phys
0	?	?
0	?	?
0	?	?
0	?	?

- Workload repeatedly accesses same offset across **5 pages** (strided access), but only **4 TLB entries**
- What will TLB contents be over time?
- How will TLB perform?

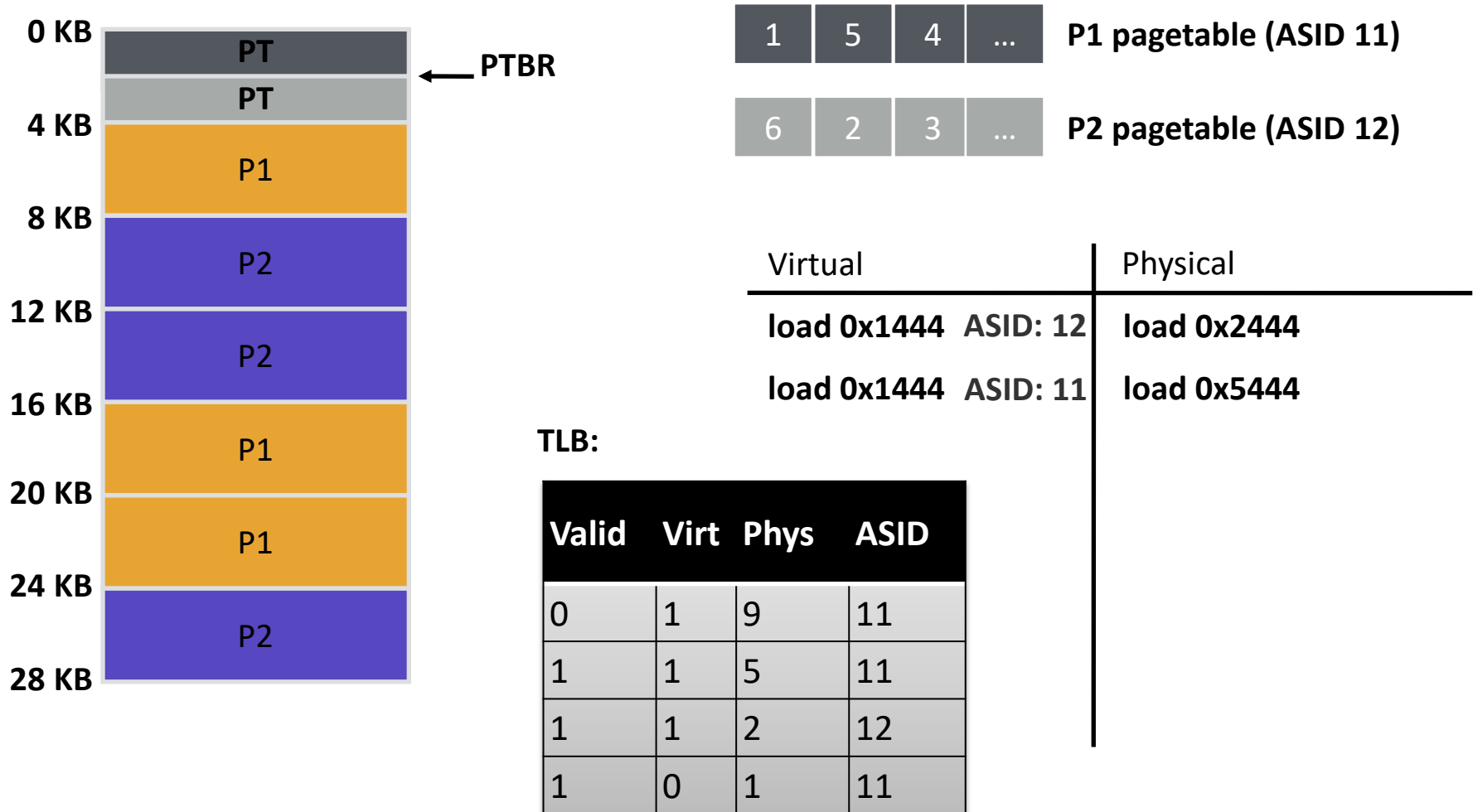
TLB Replacement policies

- **LRU**: evict Least-Recently Used TLB slot when needed
 - (More on LRU later in policies next week)
- **Random**: Evict randomly chosen entry
- Sometimes random is better than a “smart” policy!

Context Switches

- **What happens if a process uses cached TLB entries from another process?**
 - Two processes access their own VPN 0
- **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? (e.g., 8 bits for the ASID versus 32 bits for a PID)

TLB Example with ASID



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”

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

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