

MEMORY MANAGEMENT: PAGING AND THE TLB

Kai Mast

CS 537

Fall 2022

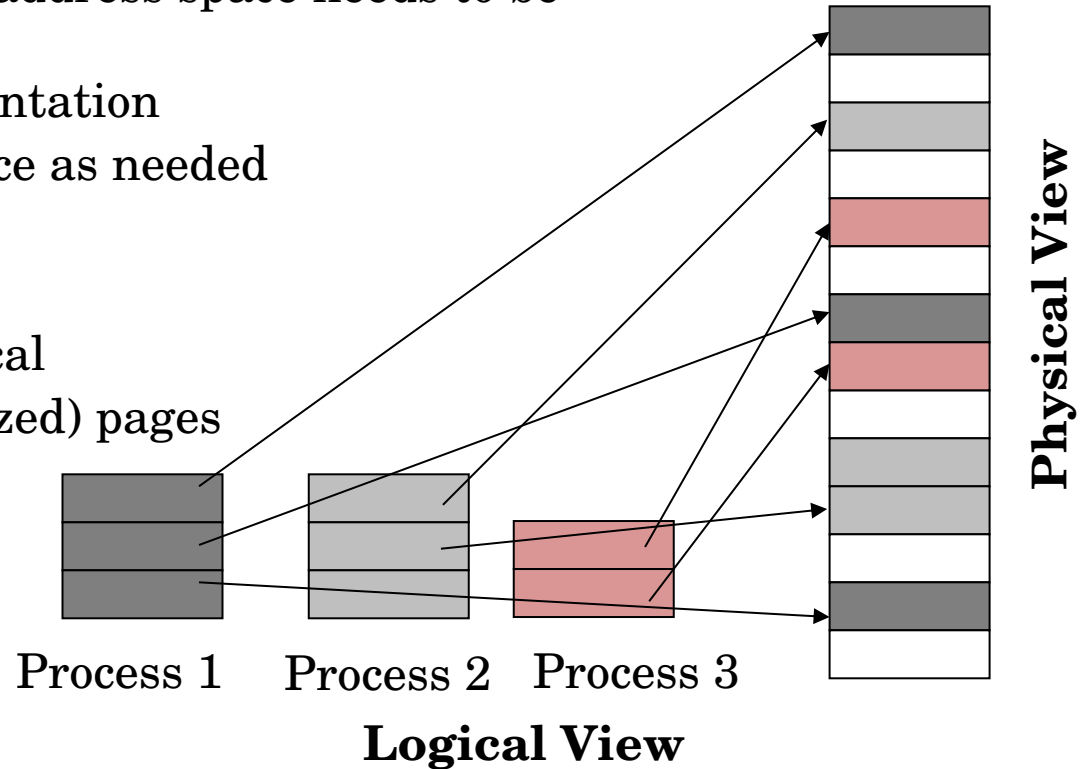
PAGING

Goal: Remove requirement that address space needs to be contiguous

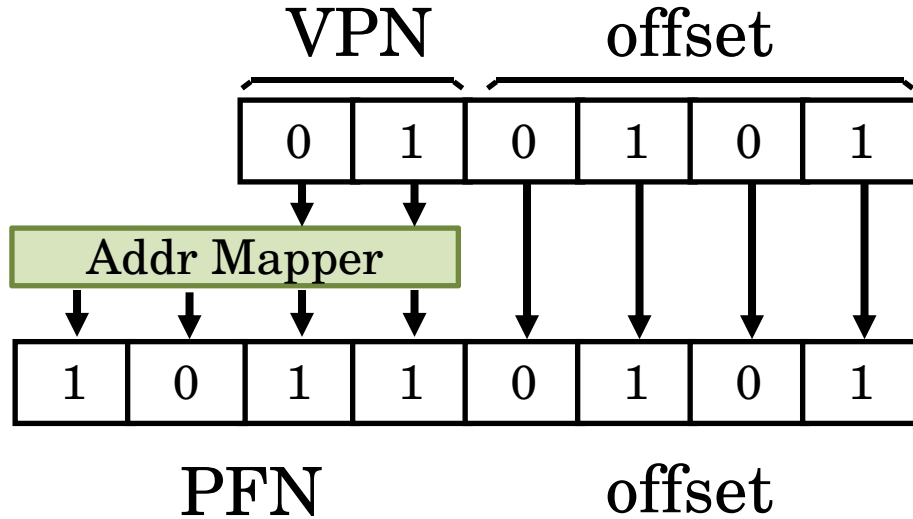
- Eliminates external fragmentation
- Ability to grow address space as needed

Idea:

Divide address spaces and physical memory into fixed-sized (same sized) pages



VIRTUAL TO PHYSICAL PAGE MAPPING



Translation Steps

- Extract virtual page number (VPN) from virtual address
- Translate VPN to physical frame number (PFN)
- Append offset to PFN to get physical address

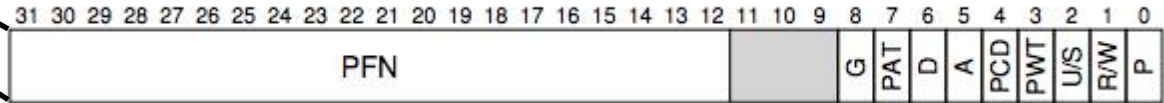
LINEAR PAGETABLES

VPN

0

2^n

- Maintain a list(=array) of VPN to PFN mappings for every process
- Length of array is VirtualAddressSpaceSize/PageSize (or 2^n , where n is number of **top bits**)
- Each list entry contains a PFN and flags



Physical Frame Number

Flags

PAGING SUMMARY

Advantages:

- No external fragmentation
- Fast to allocate and free
- Simple to swap-out portions of memory to disk

Disadvantages (so far):

- Need additional memory access to fetch page table (*this lecture*)
- Page tables can get big (*next lecture*)

CACHING PAGE MAPPINGS USING THE TLB

(Book Chapter 19)

SPATIAL LOCALITY: ARRAY ITERATOR

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

Assume

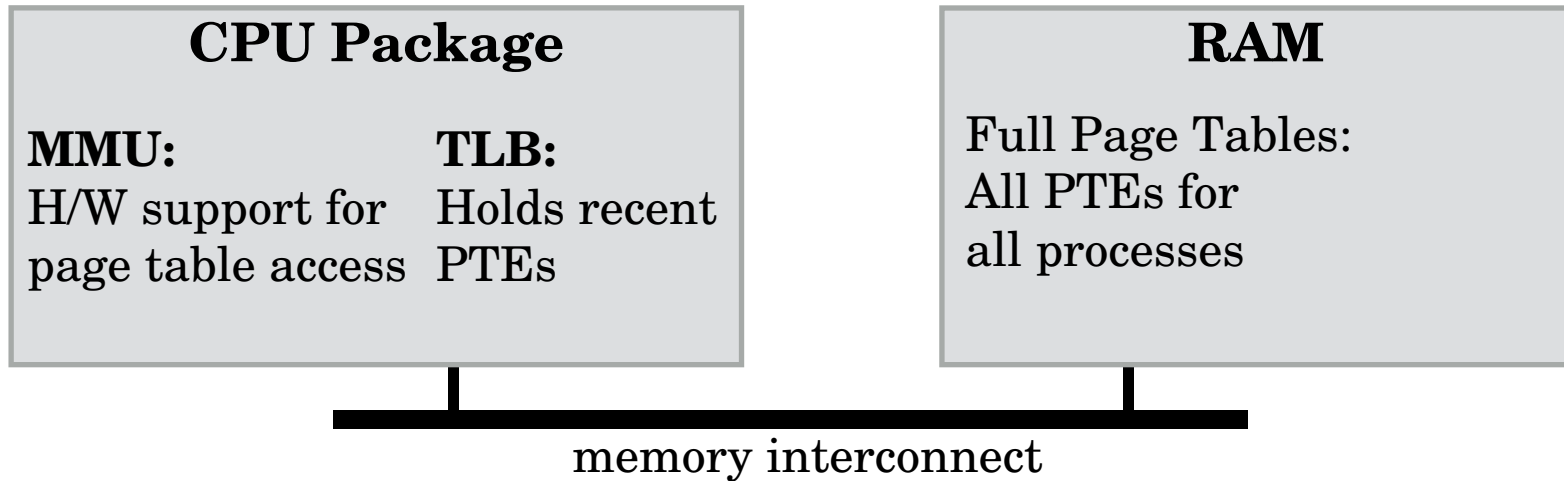
- 'a' starts at virtual address 0x3000
- Each element in 'a' is 4 bytes
- The first 4 bits are the VPN
- Page table entry is located at 0x100C
- Virtual page 3 maps to frame at 0x7000
- Ignore instruction fetches and access to 'i' and 'sum'

Virtual	Physical
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:

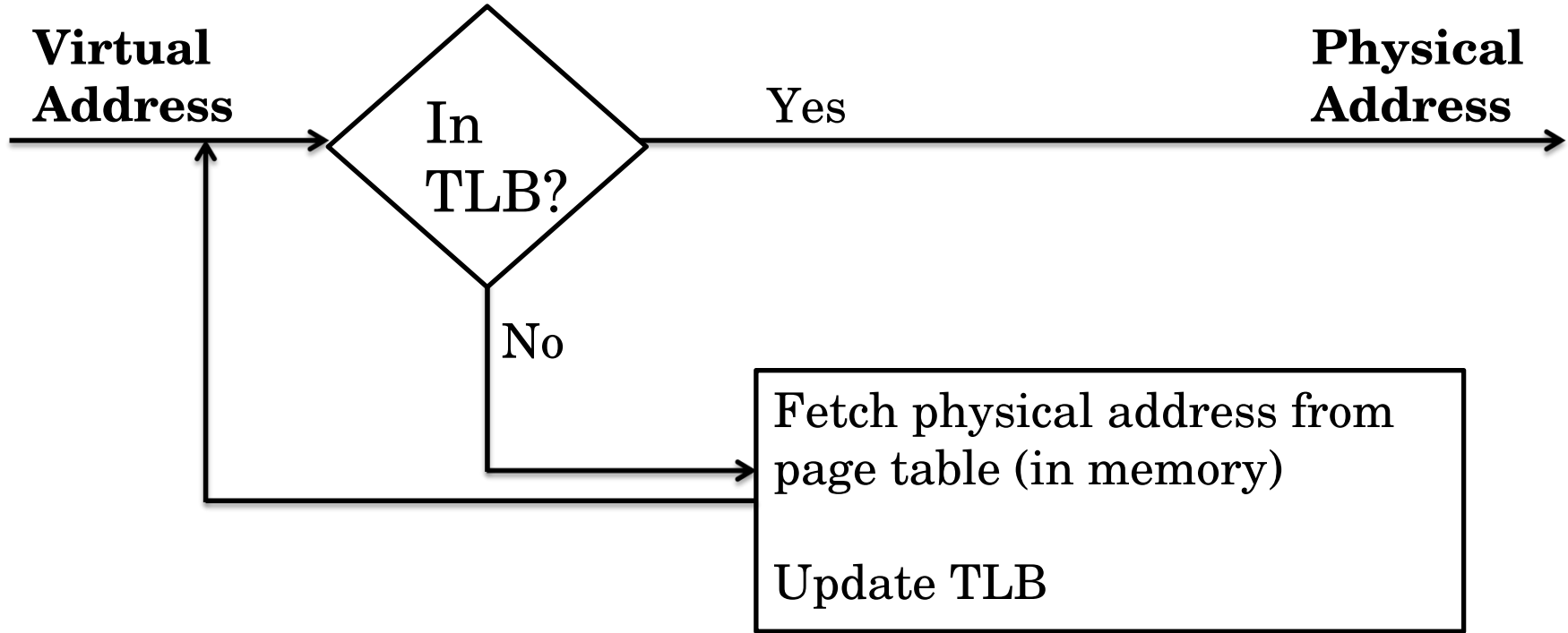
Repeatedly access same PTE
because program repeatedly
accesses same virtual page

STRATEGY: CACHE PAGE TRANSLATIONS



TLB: Translation Lookaside Buffer
(Better name: Page Translation Cache)

TLB BEHAVIOUR

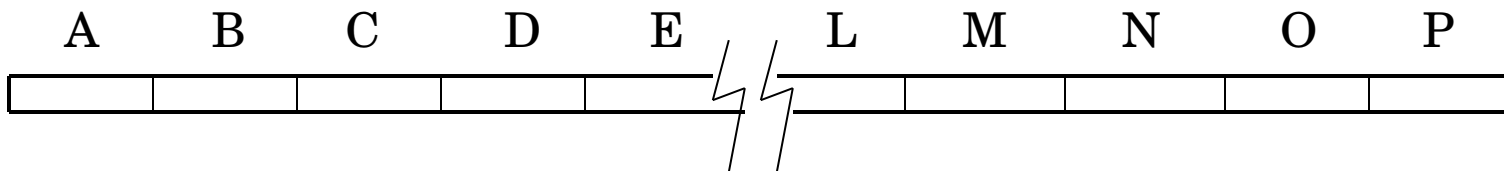


(Some of this may vary depending on the OS and CPU Architecture)

TLB ORGANIZATION

TLB Entry

Tag (virtual page number)	Physical page number (page table entry)	Protection bits (rwx)
------------------------------	--	--------------------------



Fully associative

Any given translation can be anywhere in the TLB
Hardware will search the entire TLB in parallel

TLB ACCESSES: SEQUENTIAL EXAMPLE

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

Assume

- 'a' starts at 0x1000
- Each element in 'a' is 4 bytes
- Ignore instruction fetches and access to 'i' and 'sum'

Now consider the following virtual address stream:

load 0x1000

load 0x1004

load 0x1008

load 0x100C

...

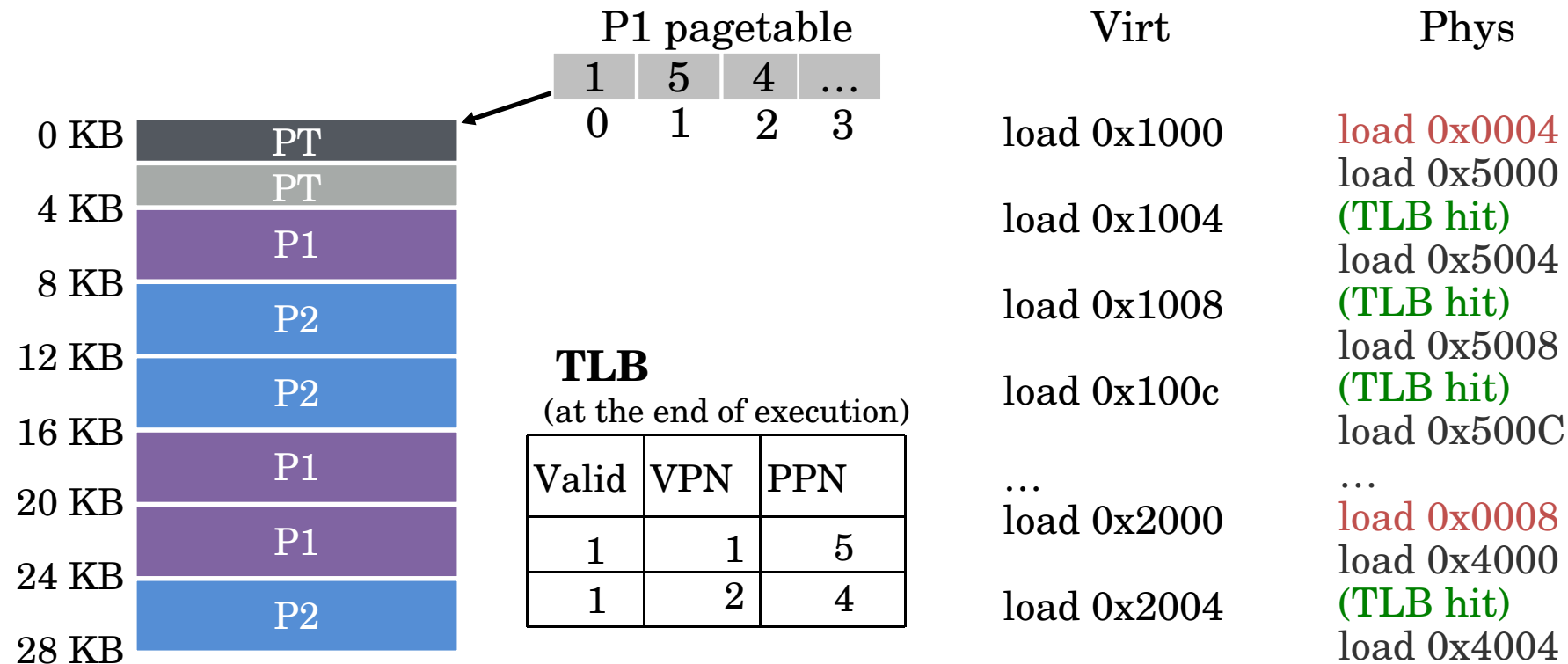
load 0x2000

load 0x2004

...

What will TLB behavior look like?

TLB ACCESSES: SEQUENTIAL EXAMPLE (CONT.)



Remember: 4K = 4096 = 0x1000

PERFORMANCE OF TLB?

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

- Calculate **miss rate** of TLB for data (ignore code + sum)
- Assume 4K pages and sizeof(int)=4

TLB lookups?

= number of accesses to array a[]
= 2048

Miss rate?

(# TLB misses / # TLB lookups)
 $2/2048 \approx 0.1\%$

TLB misses?

= number of unique pages accessed
= $2048 / (\text{elements of } a[] \text{ per } 4\text{K page})$
= $2\text{K} / (4\text{KB} / \text{sizeof(int)}) = 2\text{K} / 1\text{K} = 2 \text{ pages}$

Hit rate?

(1 – miss rate)
 $\approx 99.9\%$

TLB PERFORMANCE (CONT.)

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

Would hit rate get better or worse with smaller pages?

Worse

Would hit rate get better or worse with more iterations
(more elements in the array)?

Stay same!

Miss first access to each page

Always miss 1/1024

TLB PERFORMANCE WITH WORKLOADS

Sequential array accesses almost always hit in TLB

- Do not have to access page tables in memory

What access pattern will **not** hit in TLB?

- Highly random, with no repeated accesses

WORKLOAD ACCESS PATTERNS

Workload A

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

Workload B

```
int sum = 0; // Large N

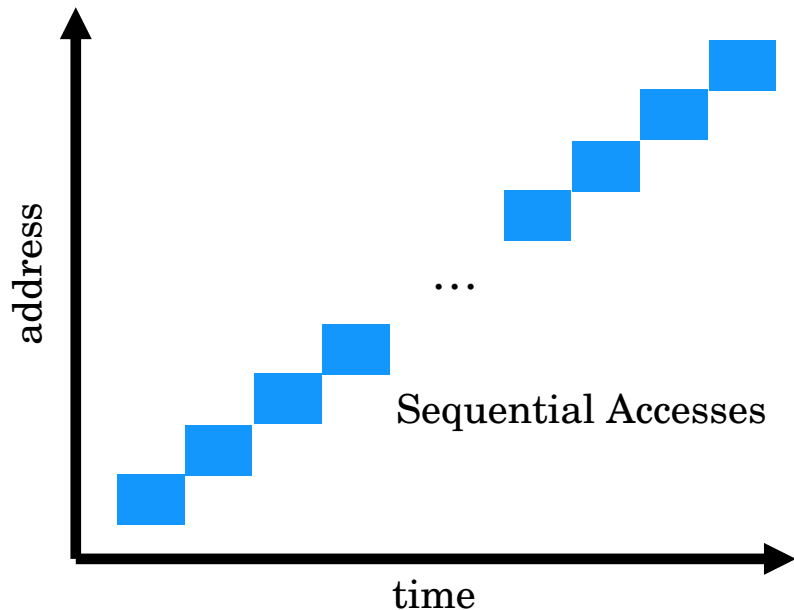
srand(1234);
for (int i=0; i<1024; i++) {
    sum += a[rand() % N];
}

srand(1234);
for (int i=0; i<1024; i++) {
    sum += a[rand() % N];
}
```


WORKLOAD ACCESS PATTERNS

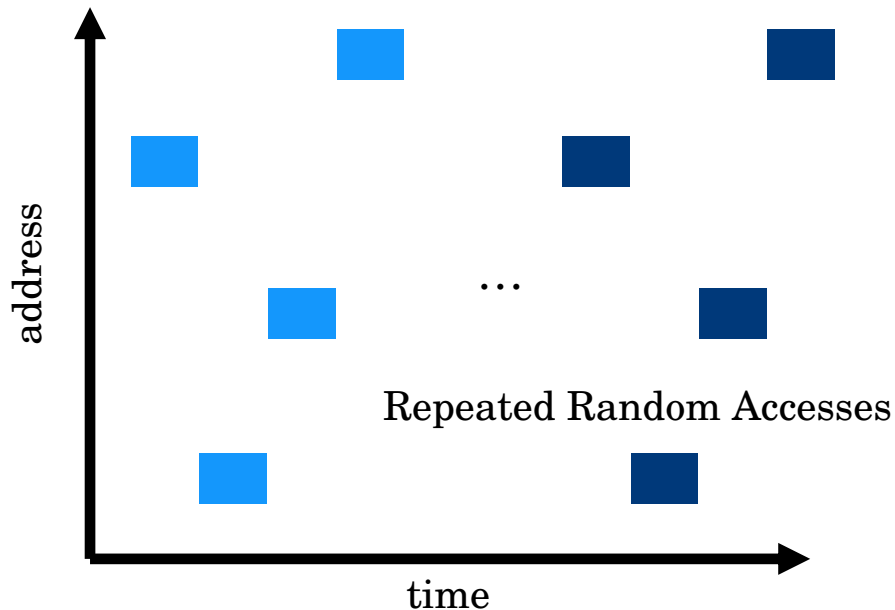
Spatial Locality

Future accesses are to **nearby** addresses



Temporal Locality

Future accesses are to **same** data as past access



WORKLOAD LOCALITY

Spatial Locality: future access will be to **nearby** addresses

Temporal Locality: future access will be repeats to the **same** data as past access

What TLB characteristics are best for each type of locality?

Spatial:

- Repeatedly access nearby addresses on same page; need same vpn to ppn translation
- Same TLB entry re-used

Temporal:

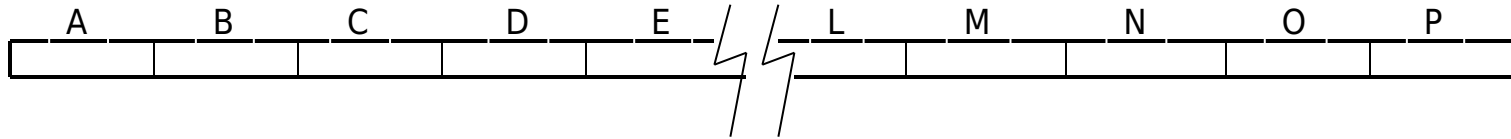
- Access same address near in future
- Same TLB entry re-used sometime in future
- How near in future? How many TLB entries are there?

TLB REPLACEMENT POLICIES

More entries in TLB -> More likely to contain entries accessed in past

LRU: Evict Least-Recently-Used TLB slot when needed
(More on LRU later in policies next week)

Random: Evict randomly-chosen entry



Which is better?

TLB PERFORMANCE FOR CODE?

Code tends to be relatively **sequential**


- Branch for if statements and while loops often in same page
- Good spatial locality

Procedure calls depend on **temporal** locality

Code usually has reasonable TLB performance

Least-Recently-Used (LRU) is a good choice

LRU TROUBLES FOR REPEATED DATA

Initial TLB			TLB after first iteration (5 accesses)			
Valid	VPN	PFN		Valid	VPN	PFN
0	?	?		1	1	13
0	?	?		1	2	64
0	?	?		1	3	22
0	?	?		1	4	62
			virtual addresses:			

Workload repeatedly accesses same offset (0x001) across 5 pages (strided access), but only 4 TLB entries

What will TLB contents be over time? (Assume LRU policy)

How will TLB perform? **Always misses! 100% miss rate**

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 TLB entry from another process?

Access some other processes address space (no protection)

Solutions:

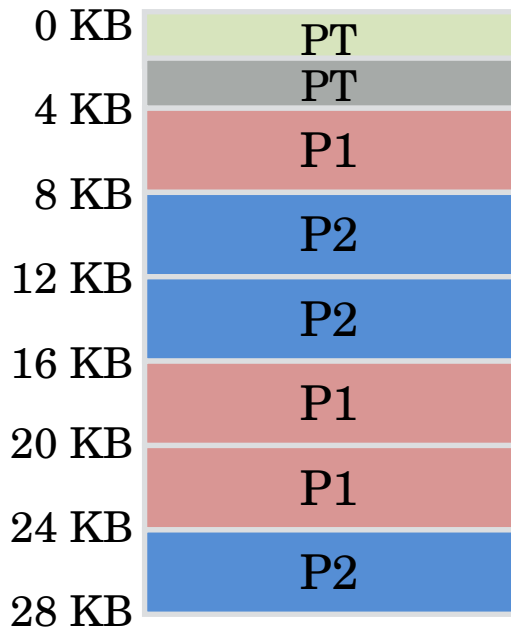
1. Flush TLB on each context switch (privileged instruction)

Poor performance: lose all recently cached translations, increases miss rate

2. Track which TLB entries are for which process

- Address Space Identifier (ASID) – similar to PID (remember in PCB)
- Tag each TLB entry with 8-bit ASID; How many ASIDs do we get?
- Must match ASID for TLB entry to be used

TLB EXAMPLE WITH ASID



1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

TLB:

Valid	Virt	Phys	ASID
0	3	9	11
1	1	5	11
1	1	2	12
1	0	1	11

Virtual	Physical
load 0x1444 (ASID 12)	load 0x2444
load 0x1444 (ASID 11)	load 0x5444

Remember: 4K = 4096 = 0x1000

TLB PERFORMANCE

Context switches are **expensive**

Even with ASID, other processes “pollute” TLB

- Only have a fixed number of TLB entries
- While running process B, will replace process A’s TLB entries for B’s entries
- Process A’s TLB entries may be gone by the time it is scheduled again

Architectures can have **multiple TLBs**

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for “super pages”

SUMMARY OF TLBS

Accessing page tables in RAM for every **memory access is slow**

- TLB caches recent page translations
- Hardware performs TLB lookup on every memory access

TLB performance **depends** strongly **on workload**

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

TLBs increase **cost of context switches**

- Flush TLB on every context switch
- Add ASID to every TLB entry as part of tag

In different systems, hardware or OS can handle TLB misses