OSTEP
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?

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?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

What physical addresses?

load 0x100C
load 0x7000
load 0x100C
load 0x7004
load 0x100C
load 0x7008
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

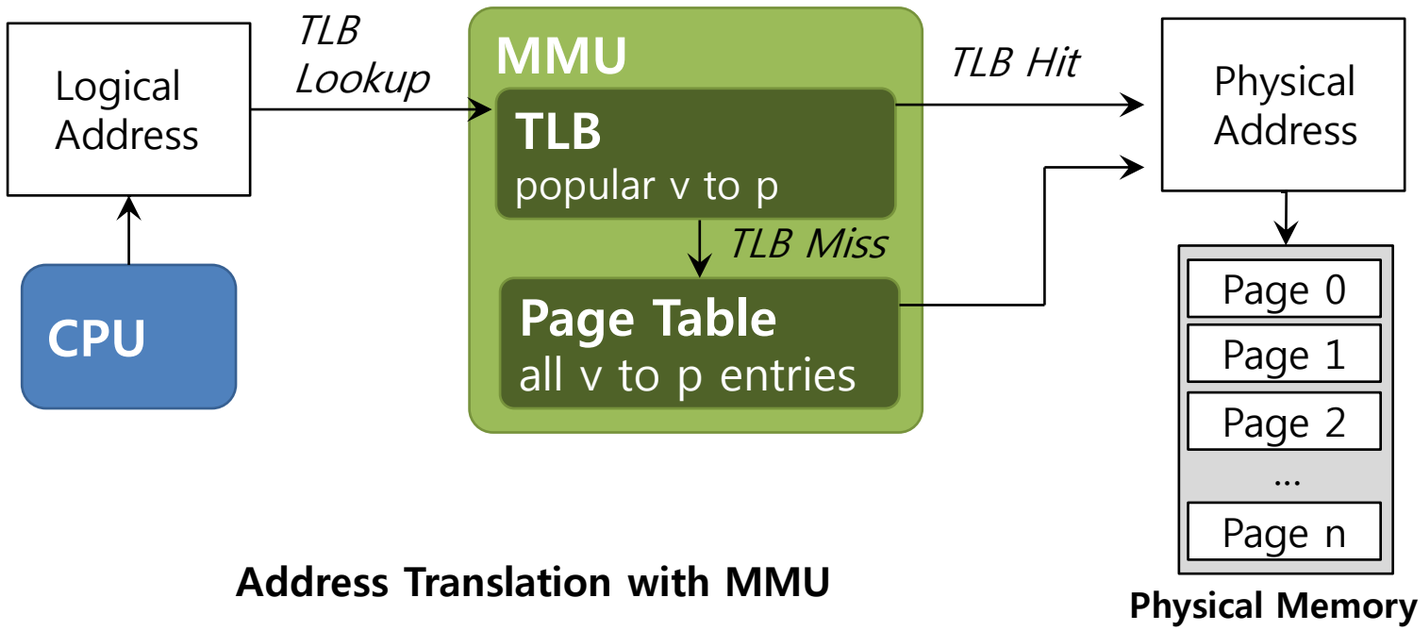
Strategy: Cache Page Translations

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

TLB

Part of the chip's memory-management unit(MMU).

A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

TLB Basic Algorithms

```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == Ture){ // TLB Hit
4:    if(CanAccess(TlbEntry.ProtectBit) == True ){
5:        offset = VirtualAddress & OFFSET_MASK
6:        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:        AccessMemory( PhysAddr )
8:    }else RaiseException(PROTECTION_ERROR)
```

- (1 lines) extract the virtual page number(VPN).
- (2 lines) check if the TLB holds the transalation for this VPN.
- (5-8 lines) extract the page frame number from the relevant TLB entry, and form the desired physical address and access memory.

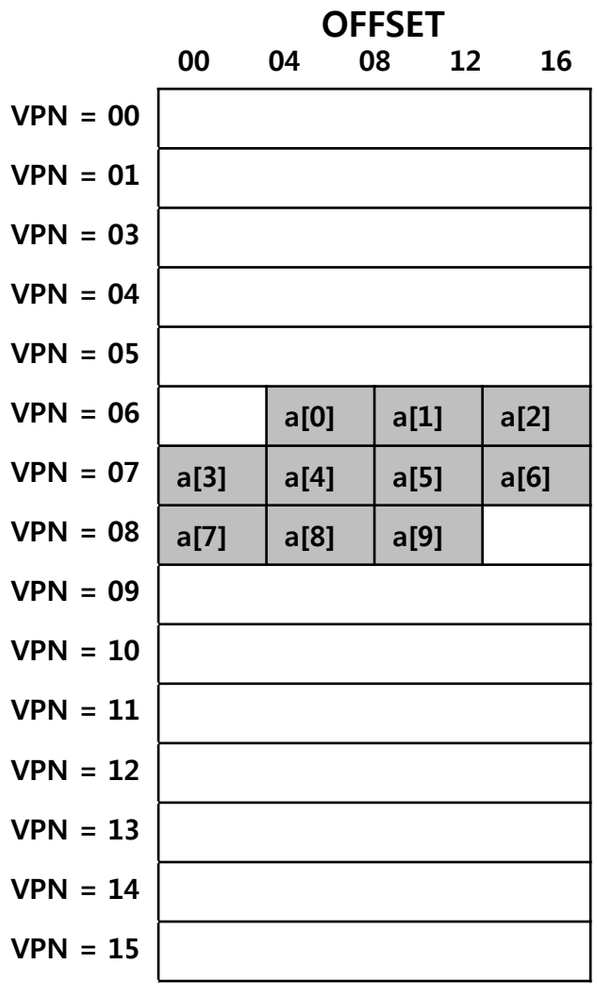
TLB Basic Algorithms (Cont.)

```
11:     }else{ //TLB Miss
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))
13:         PTE = AccessMemory(PTEAddr)
14:         (…)
15:     }else{
16:         TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:         RetryInstruction()
18:     }
19:}
```

- (11-12 lines) The hardware accesses the page table to find the translation.
- (16 lines) updates the TLB with the translation.

Example: Accessing An Array

- How a TLB can improve its performance.




```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

The TLB improves performance due to spatial locality

3 misses and 7 hits.
Thus TLB hit rate is 70%.

Cache Types

Direct-Mapped: only one place to put entries

Four-Way Set Associative: 4 options

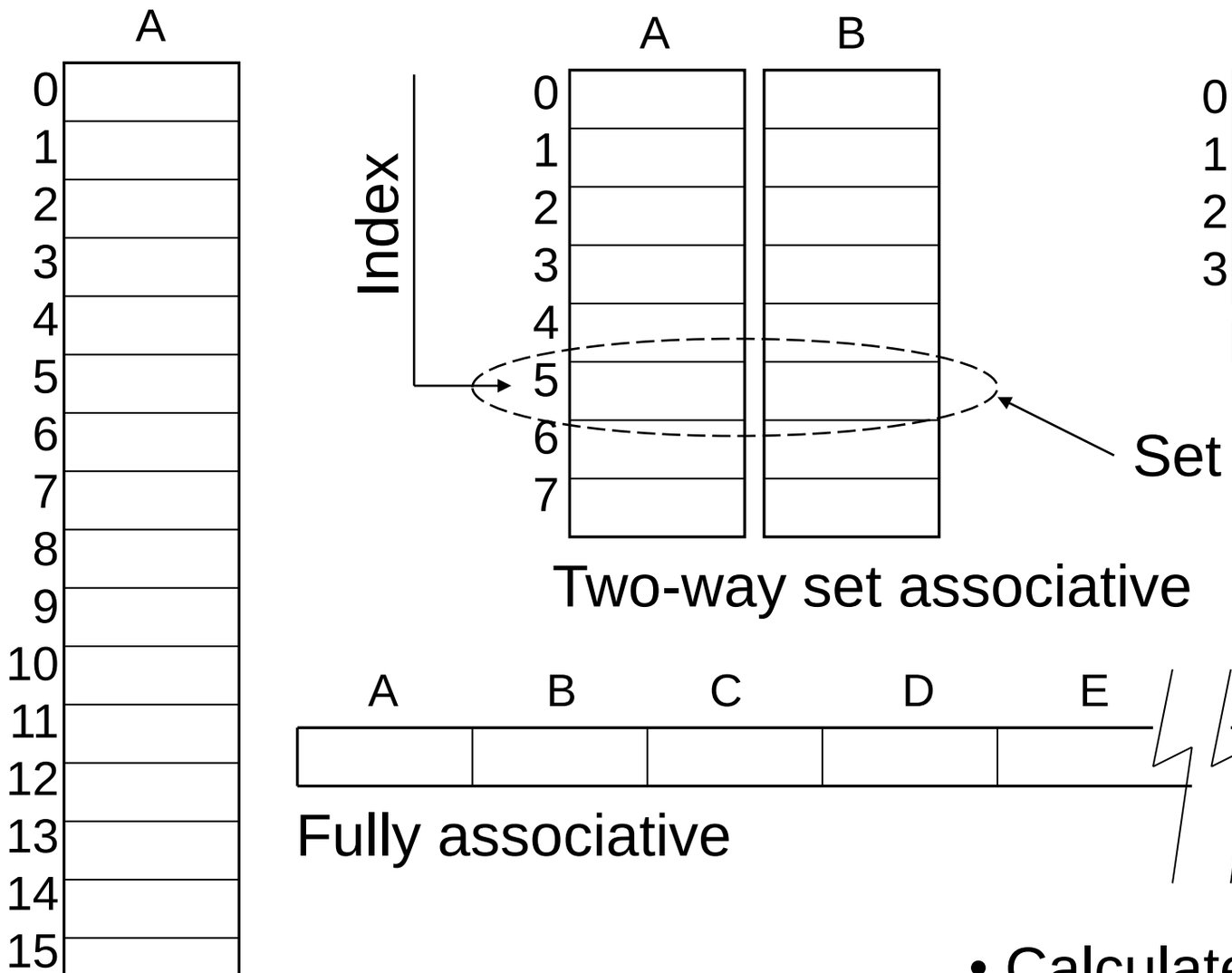
Fully-Associative: entries can go anywhere
- most common for TLBs
- must store whole key/value in cache
- search all in parallel

TLB Organization

TLB Entry

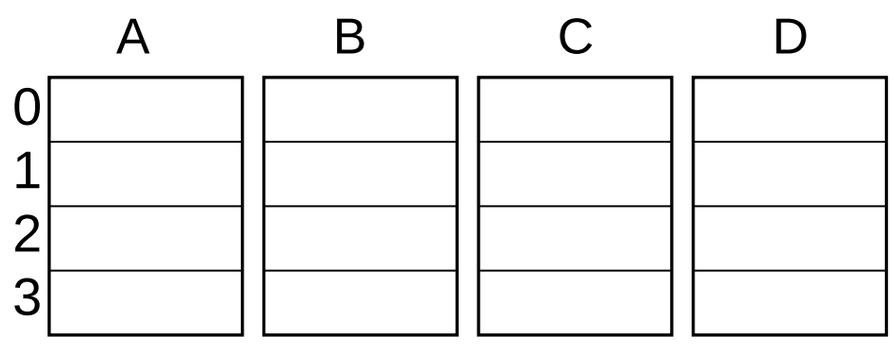
Tag (virtual page number)	Physical page number (page table entry)

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



Direct mapped

Two-way set associative



Four-way set associative

Fully associative

Lookup

- Calculate set (tag % 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

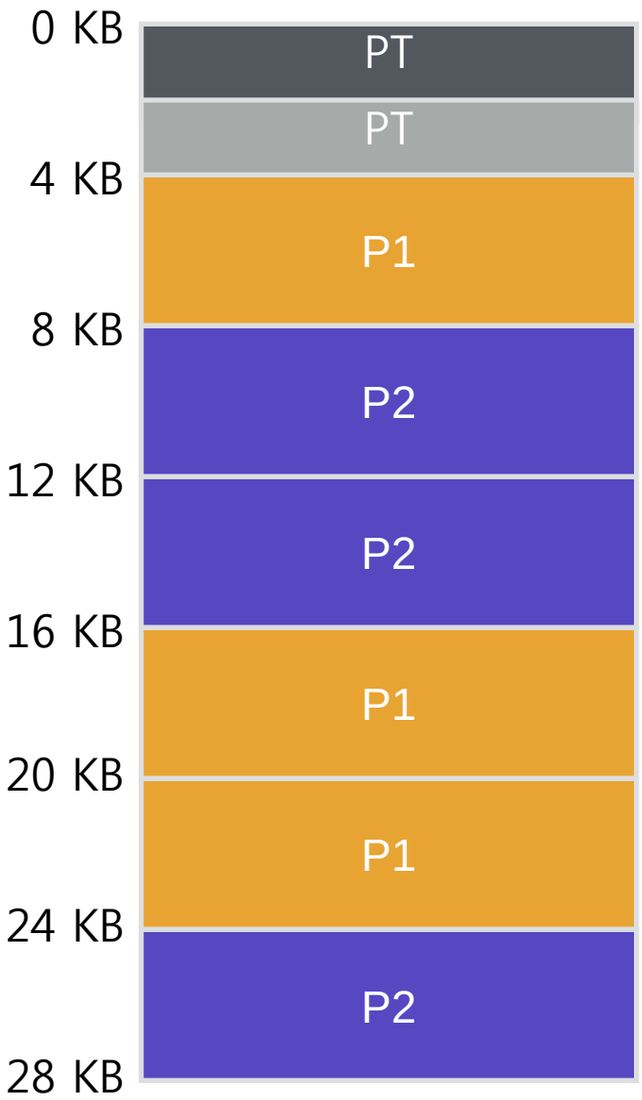
load 0x1008

load 0x100C

…

What will TLB behavior look like?

TLB Accesses: SEQUENTIAL Example



PTBR

P1 pagetable

1	5	4	…
0	1	2	3

CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virt	Phys
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:
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

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

Increase page size

Fewer unique page translations needed to access same amount of memory

TLB Reach:

Number of TLB entries * Page Size

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

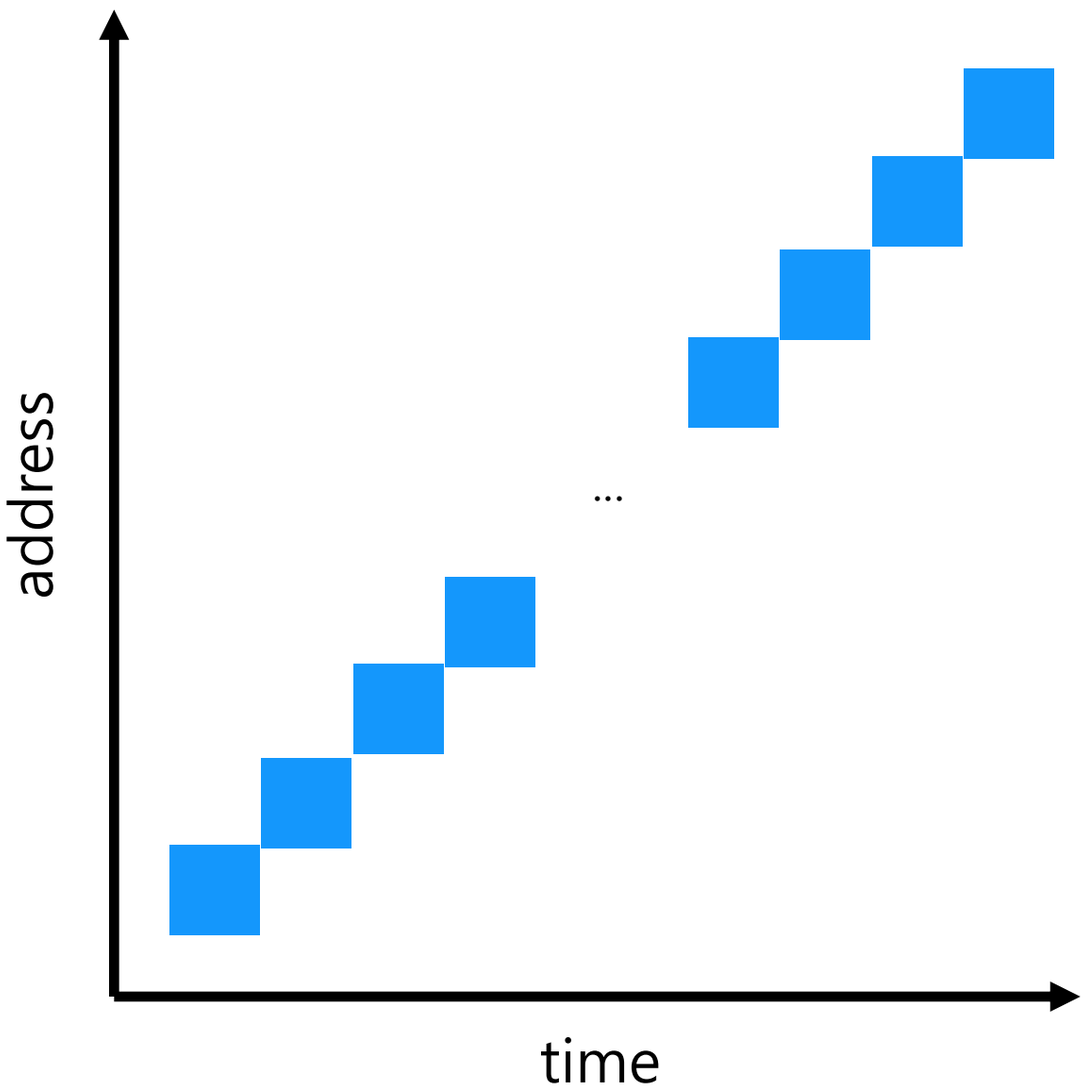
```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
     sum += a[rand() % N];
}
```


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

Workload Access Patterns

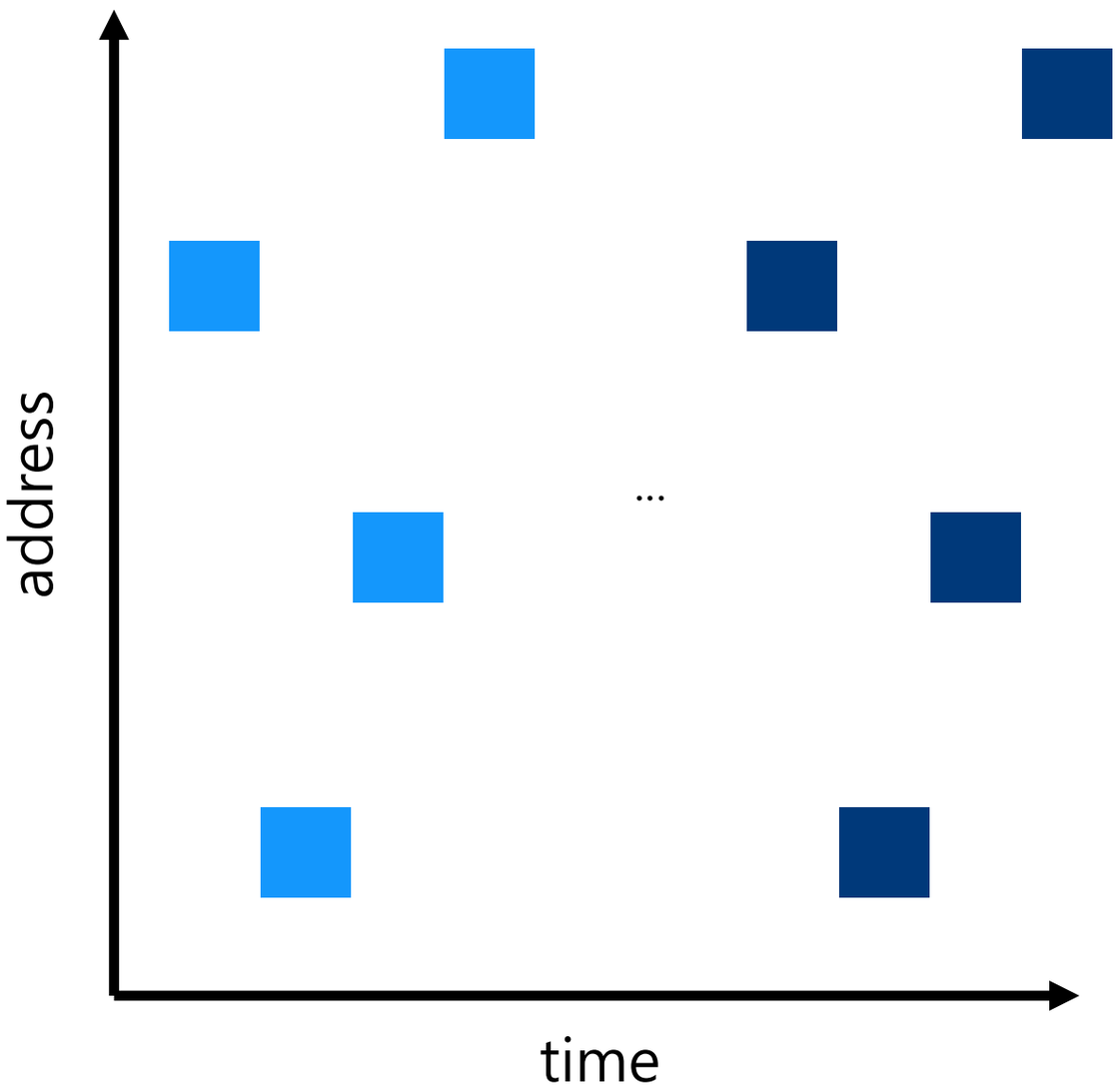
Spatial Locality

Sequential Accesses



Temporal Locality

Repeated Random Accesses



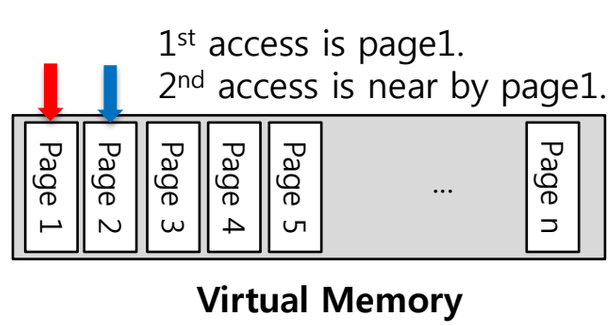
Workload Locality

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?

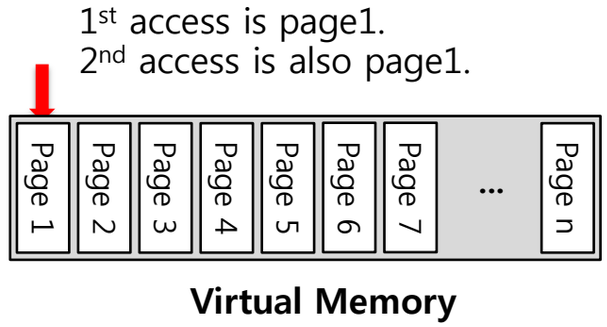
Spatial:

- 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?



Who Handles The TLB Miss?

Hardware handle the TLB miss entirely on CISC.

- The hardware has to know exactly where the page tables are located in memory.
- The hardware would “walk” the page table, find the correct page-table entry and extract the desired translation, update and retry instruction.
- **hardware-managed TLB.**

Who Handles The TLB Miss? (Cont.)

RISC have what is known as a **software-managed TLB.**

- On a TLB miss, the hardware raises exception(trap handler).
  - **Trap handler is code** within the OS that is written with the express purpose of handling TLB miss.

TLB Control Flow algorithm(OS Handled)

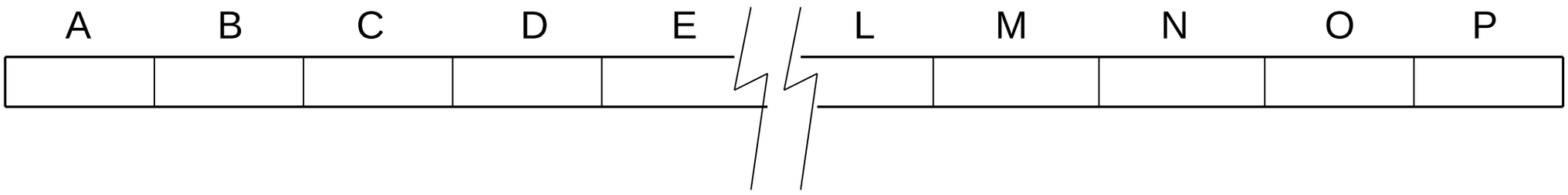
```
1:      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:      (Success, TlbEntry) = TLB_Lookup(VPN)
3:      if (Success == True) // TLB Hit
4:           if (CanAccess(TlbEntry.ProtectBits) == True)
5:                Offset = VirtualAddress & OFFSET_MASK
6:                PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:                Register = AccessMemory(PhysAddr)
8:           else
9:                RaiseException(PROTECTION_FAULT)
10:     else // TLB Miss
11:          RaiseException(TLB_MISS)
```

TLB Replacement policies

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

Random: Evict randomly choosen entry

Which is better?



LRU Troubles

virtual addresses:

0 1 2 3 4

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?

LRU Troubles

virtual addresses:

0 1 2 3 4

Valid	Virt	Phys
0		
0		
0		
0		

LRU Troubles

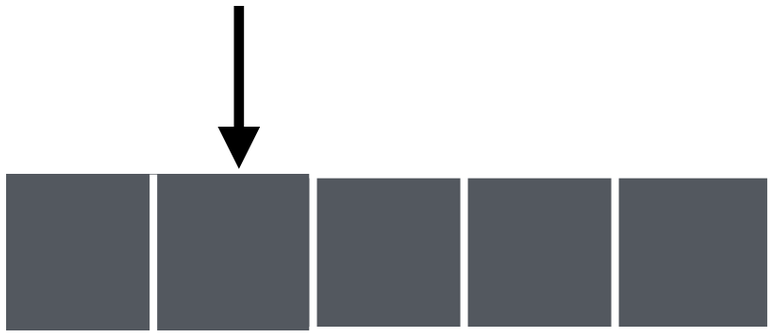
virtual addresses:

Valid	Virt	Phys
1	0	?
0		
0		
0		

miss!

LRU Troubles

virtual addresses:

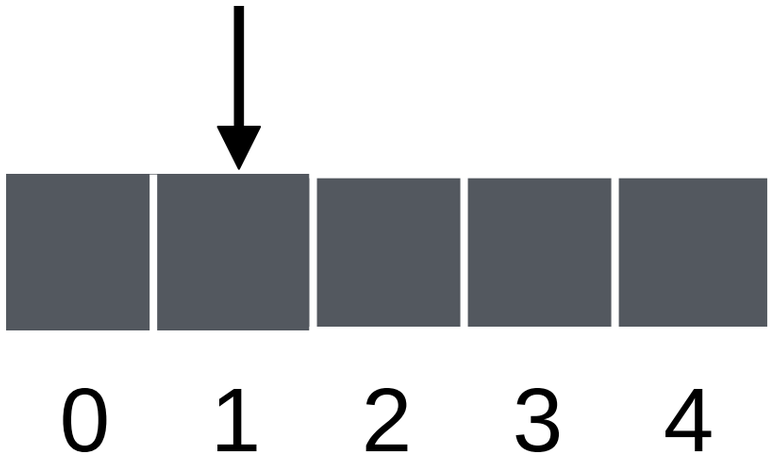


0 1 2 3 4

Valid	Virt	Phys
1	0	?
0		
0		
0		

LRU Troubles

virtual addresses:

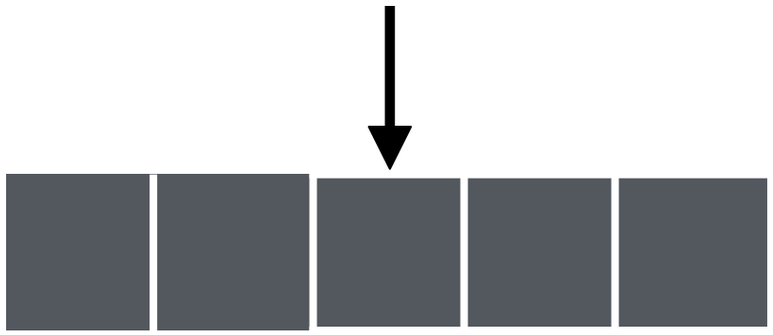


Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

miss!

LRU Troubles

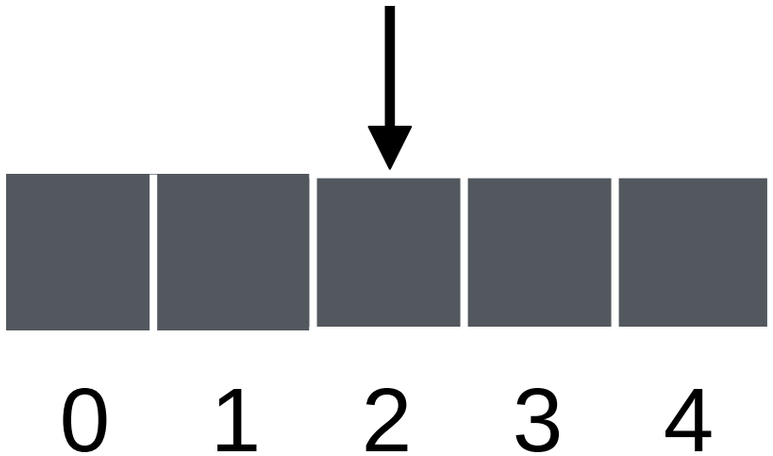
virtual addresses:



Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

LRU Troubles

virtual addresses:

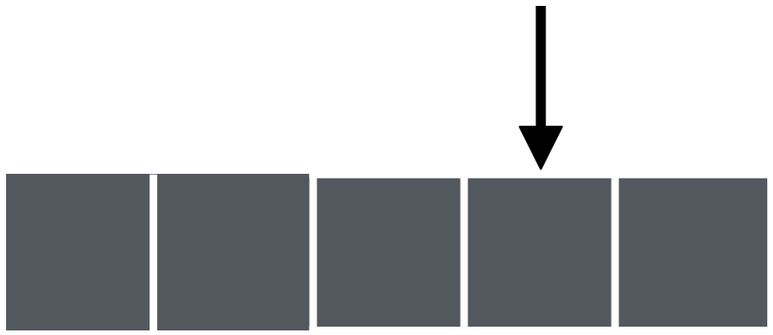


Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

miss!

LRU Troubles

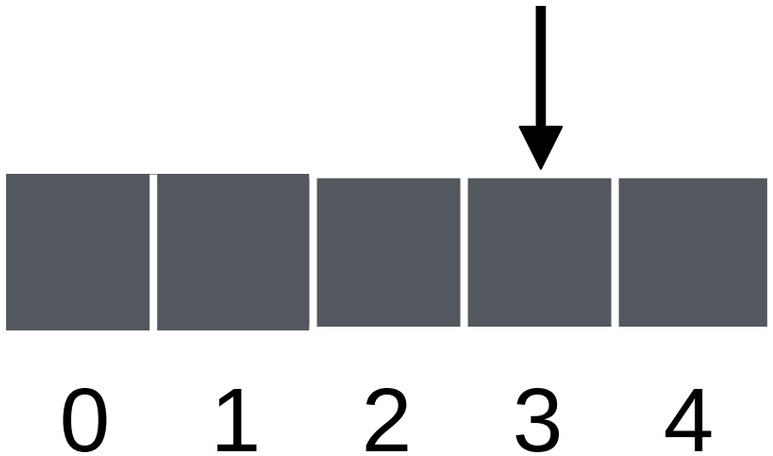
virtual addresses:



Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

LRU Troubles

virtual addresses:



Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0	3	?

miss!

LRU Troubles

virtual addresses:

0 1 2 3 4

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0	3	?

LRU Troubles

virtual addresses:

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
0	3	?

miss!

LRU Troubles

virtual addresses:

0 1 2 3 4

Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
0	3	?

LRU Troubles

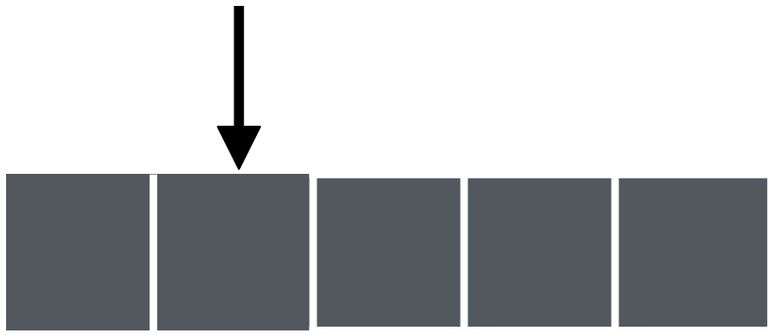
virtual addresses:

Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
0	3	?

miss!

LRU Troubles

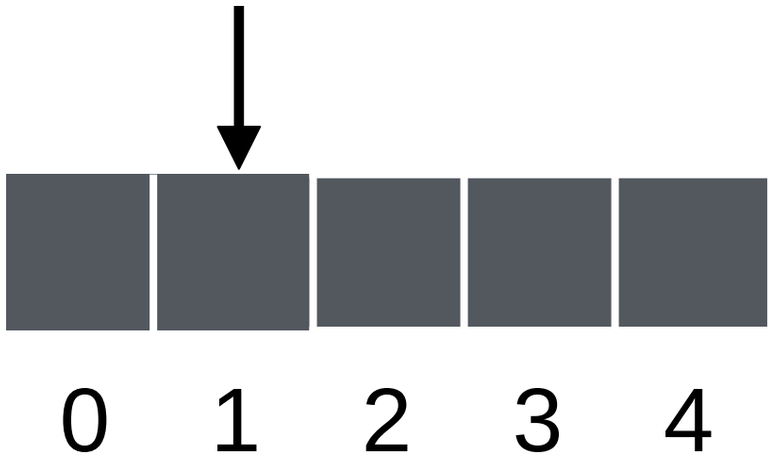
virtual addresses:



Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
0	3	?

LRU Troubles

virtual addresses:

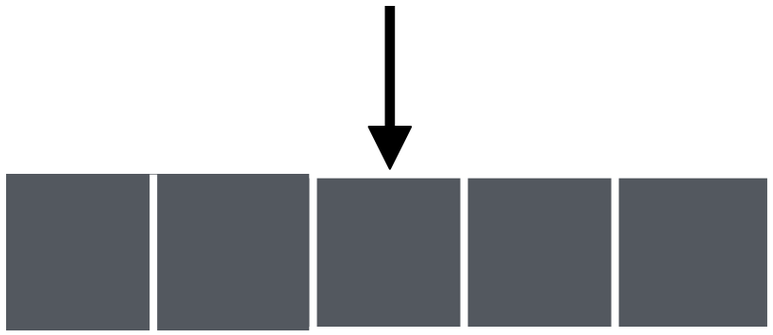


Valid	Virt	Phys
1	4	?
1	0	?
1	1	?
0	3	?

miss!

LRU Troubles

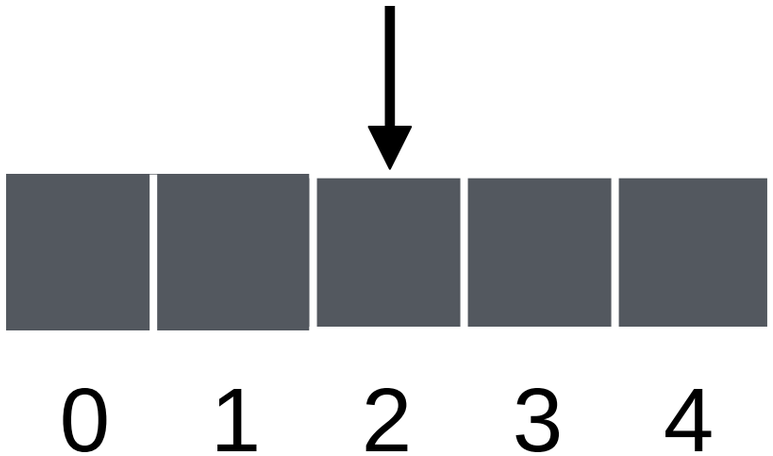
virtual addresses:



Valid	Virt	Phys
1	4	?
1	0	?
1	1	?
0	3	?

LRU Troubles

virtual addresses:



Valid	Virt	Phys
1	4	?
1	0	?
1	1	?
0	2	?

miss!

TLB Replacement policies

LRU: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

Random: Evict randomly choosen entry

Sometimes random is better than a “smart” policy!

TLB Performance

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

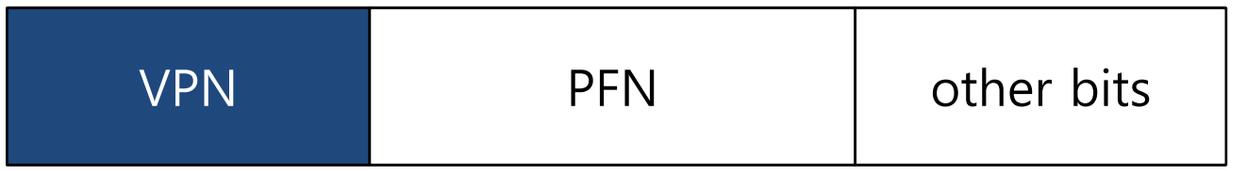
Increase page size

Fewer unique translations needed to access same amount of memory)

TLB entry

TLB is managed by **Full Associative** method.

- A typical TLB might have 32,64, or 128 entries.
- Hardware search the entire TLB in parallel to find the desired translation.
- other bits: valid bits , protection bits, address-space identifier, dirty bit



Typical TLB entry look like this

Context Switches

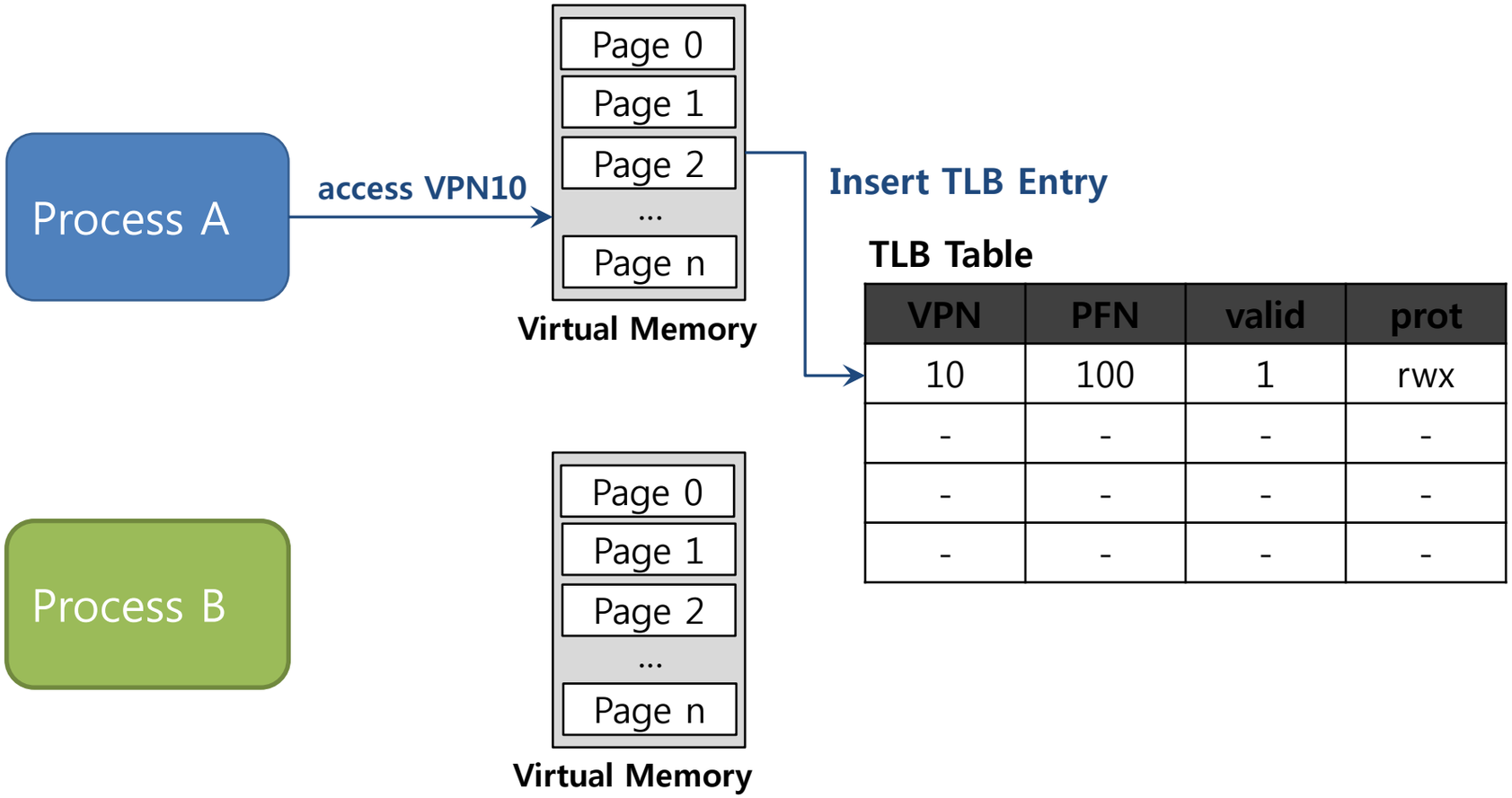
What happens if a process uses cached TLB entries from another process?

Solutions?

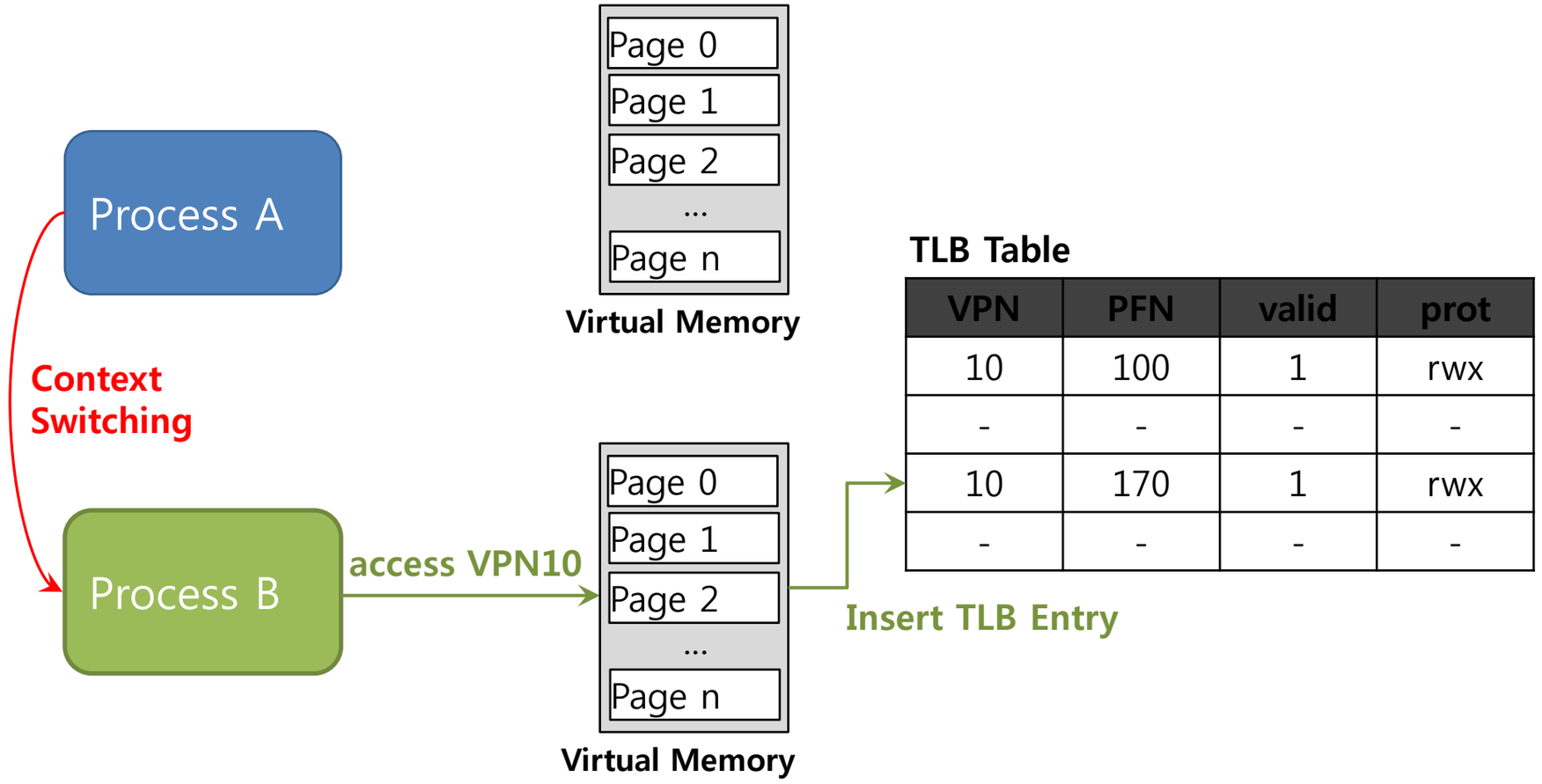
1. Flush TLB on each switch
   - Costly; lose all recently cached translations

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?
     - what if there are more PIDs than ASIDs?

TLB Issue: Context Switching

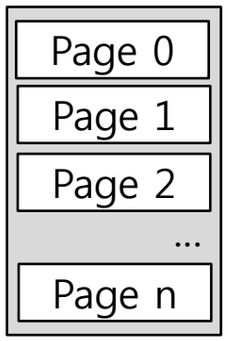


TLB Issue: Context Switching



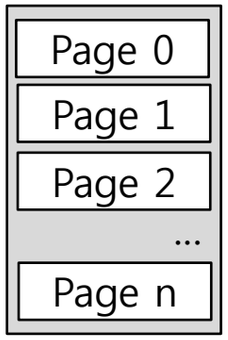
TLB Issue: Context Switching

Process A



Virtual Memory

Process B



Virtual Memory

TLB Table

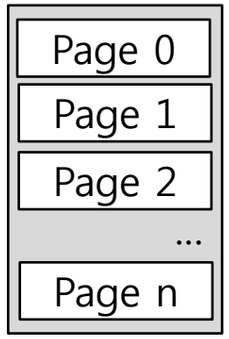
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

To Solve Problem

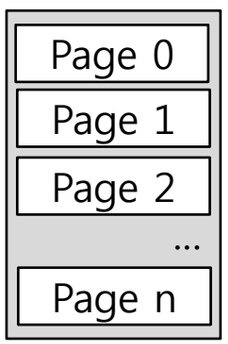
Provide an address space identifier(ASID) field in the TLB.

Process A



Virtual Memory

Process B



Virtual Memory

TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Another Case

Two processes share a page.

- Process 1 is sharing physical page 101 with Process2.
- P1 maps this page into the 10th page of its address space.
- P2 maps this page to the 50th page of its address space.

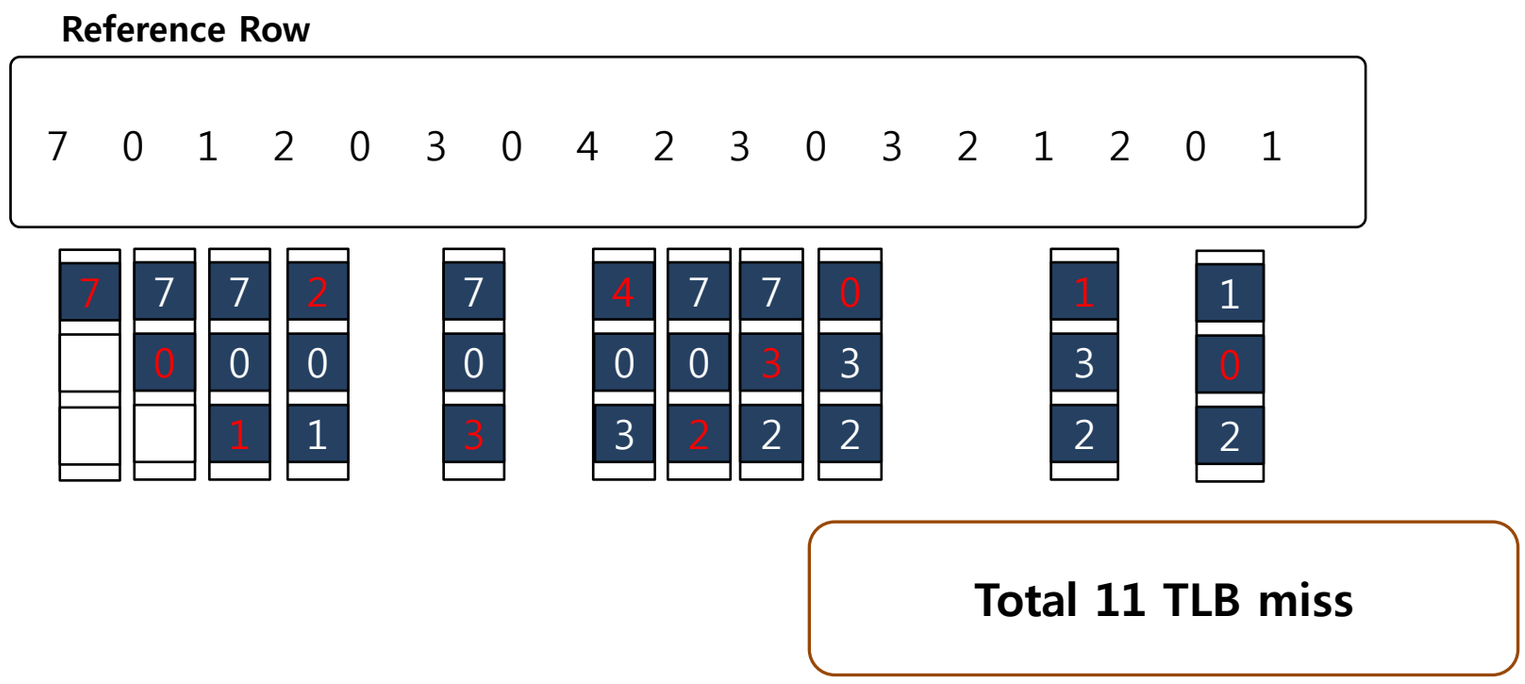
VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Replacement Policy

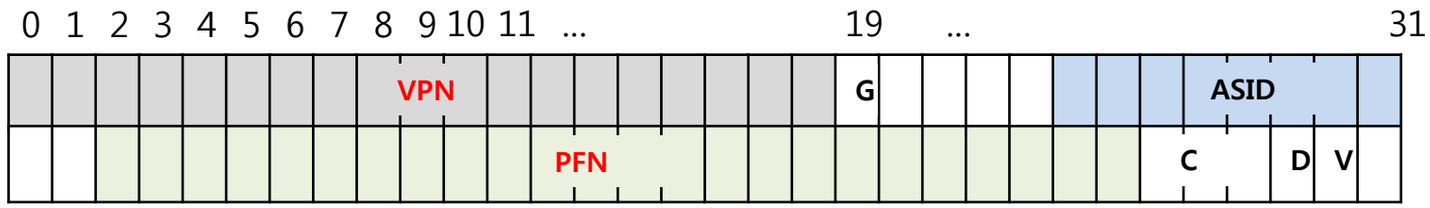
LRU(Least Recently Used)

- Evict an entry that has not recently been used.
- Take advantage of *locality* in the memory-reference stream.



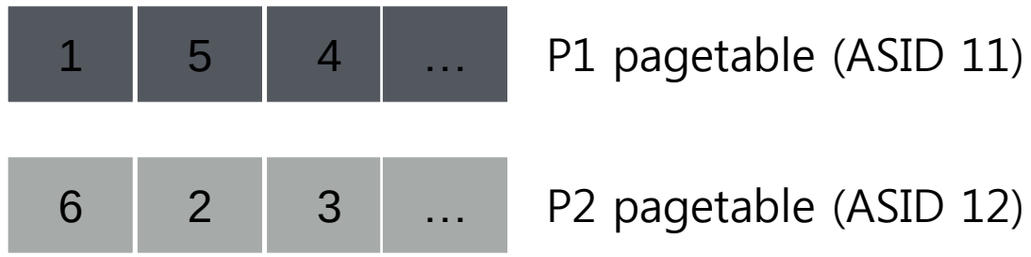
A Real TLB Entry

All 64 bits of this TLB entry(example of MIPS R4000)



Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory($2^{24} * 4KB$ pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

TLB Example with ASID



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

TLB:

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

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"

Who changes the TLB?

H/W or **OS**?

H/W: CPU must know where pagetables are
- CR3 on x86
- pagetable structure not flexible
- “walk” the pagetable

OS: CPU traps into OS upon TLB miss
- how to avoid double traps?
- more modern

Security

Modifying TLB entries is privileged
 - otherwise what could you 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