Engineering &
Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Memory – Paging

Endadul Hoque

Acknowledgement

- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

Revisit: Segmentation

- For virtual memory, OS can follow:
  - **Segmentation**: Chop an address space into variable-sized logical segments (code, stack, heap, etc.)

- Issue: **Fragmentation** in physical memory makes allocation difficult over time

Concept of Paging

- **Paging** splits up an address space into fixed-sized units, each of which is called a **page**

- With paging, **physical memory** is also **split** into some number of pages (fixed-sized slots) called a **page frame**.

- **Page table** per process is needed **to translate** the virtual address to physical address.

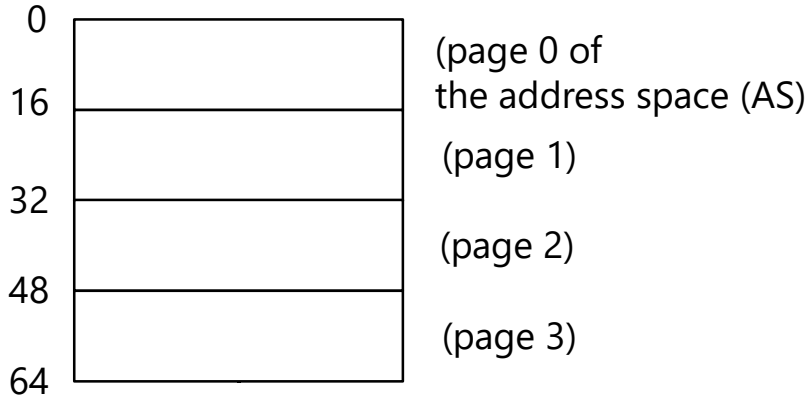
Advantages of Paging

- **Flexibility**: Supporting the abstraction of address space effectively
  - Don't need assumption on how heap and stack grow and are used.

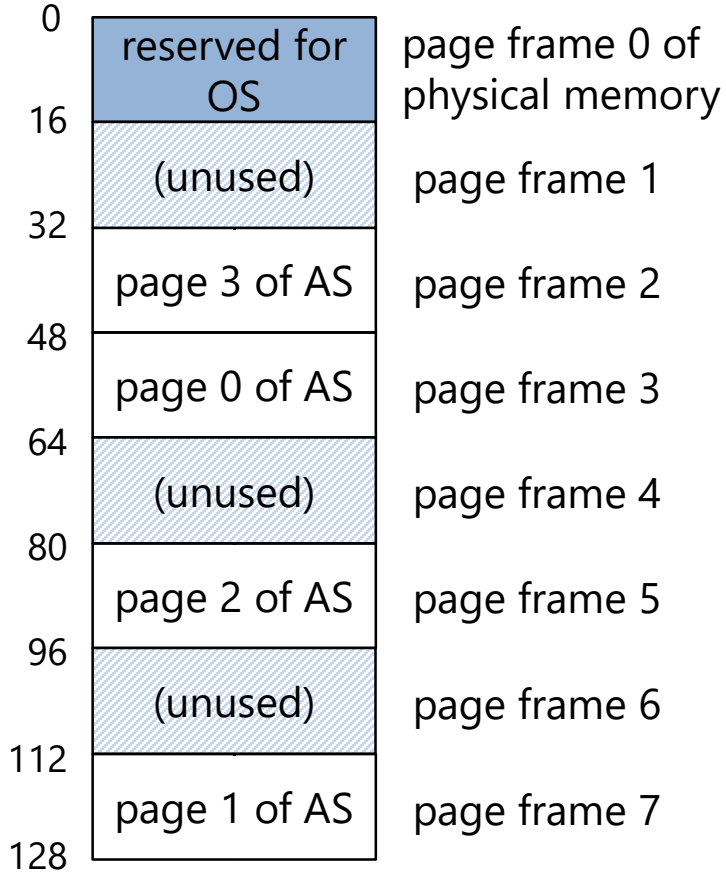
- **Simplicity**: ease of free-space management
  - The page in address space and the page frame are the same size.
  - Easy to allocate and keep a free list

Example: A Simple Paging

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages



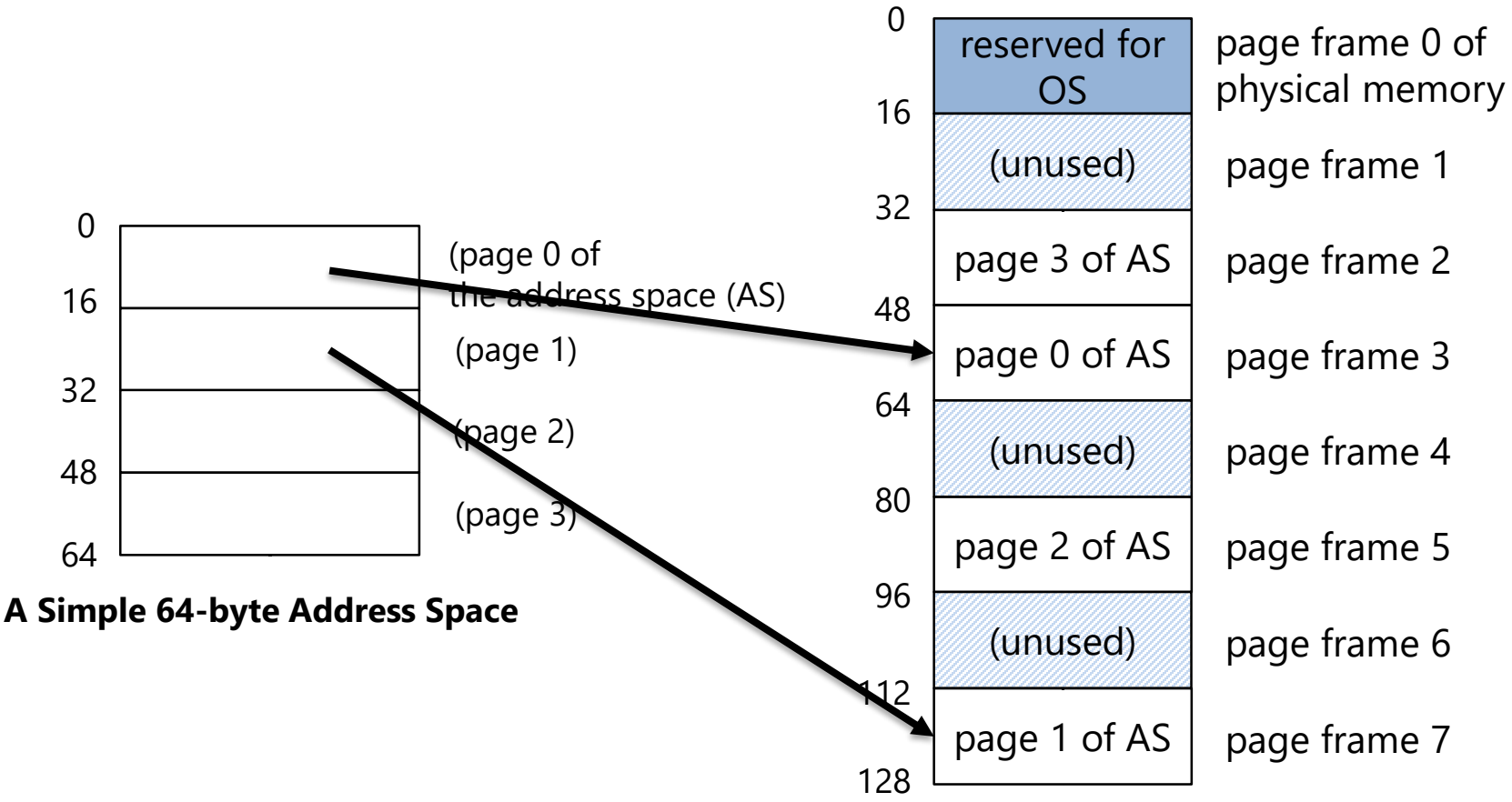
A Simple 64-byte Address Space



64-Byte Address Space Placed In Physical Memory

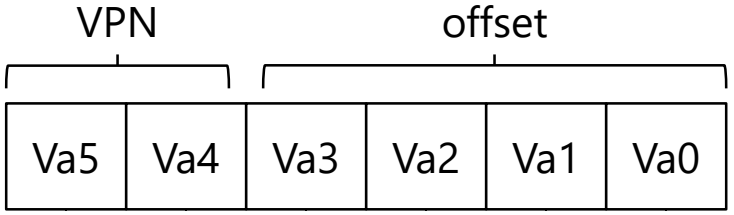
Example: A Simple Paging

- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages

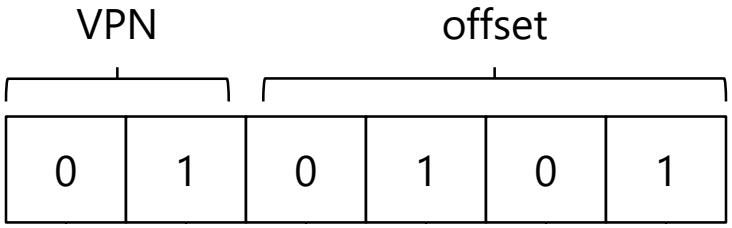


Address Translation

- Two components in the virtual address
  - **VPN**: virtual page number
  - **Offset**: offset within the page

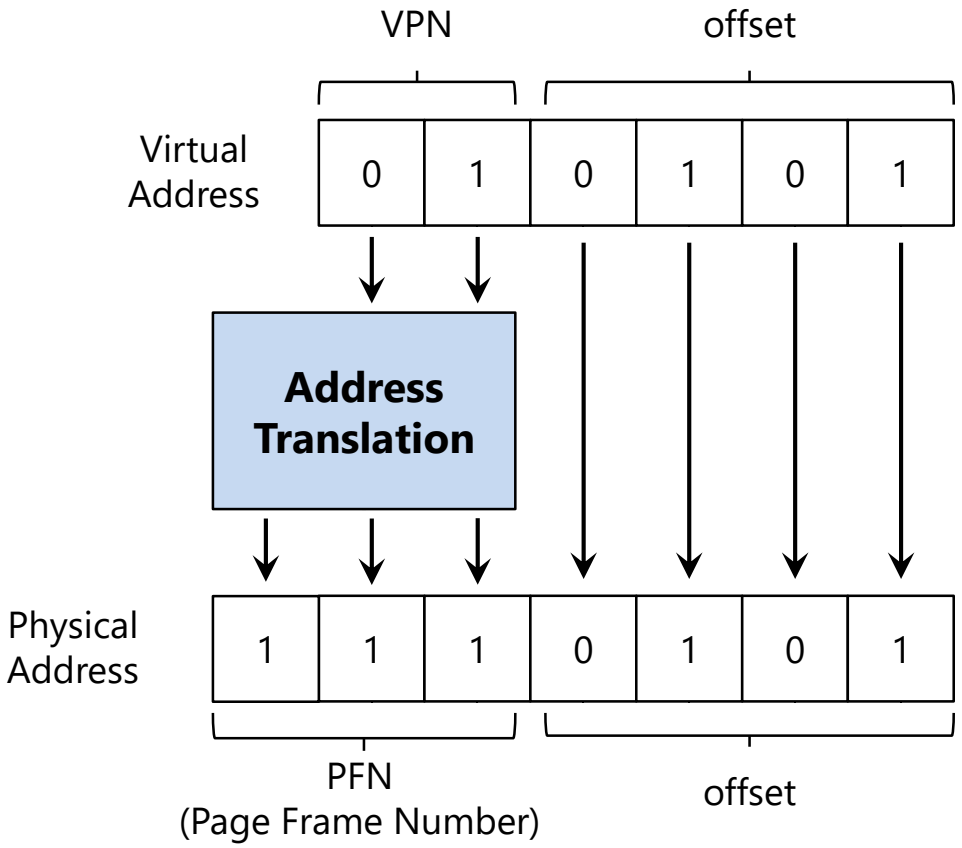


- Example: virtual address 21 in 64-byte address space



Example: Address Translation

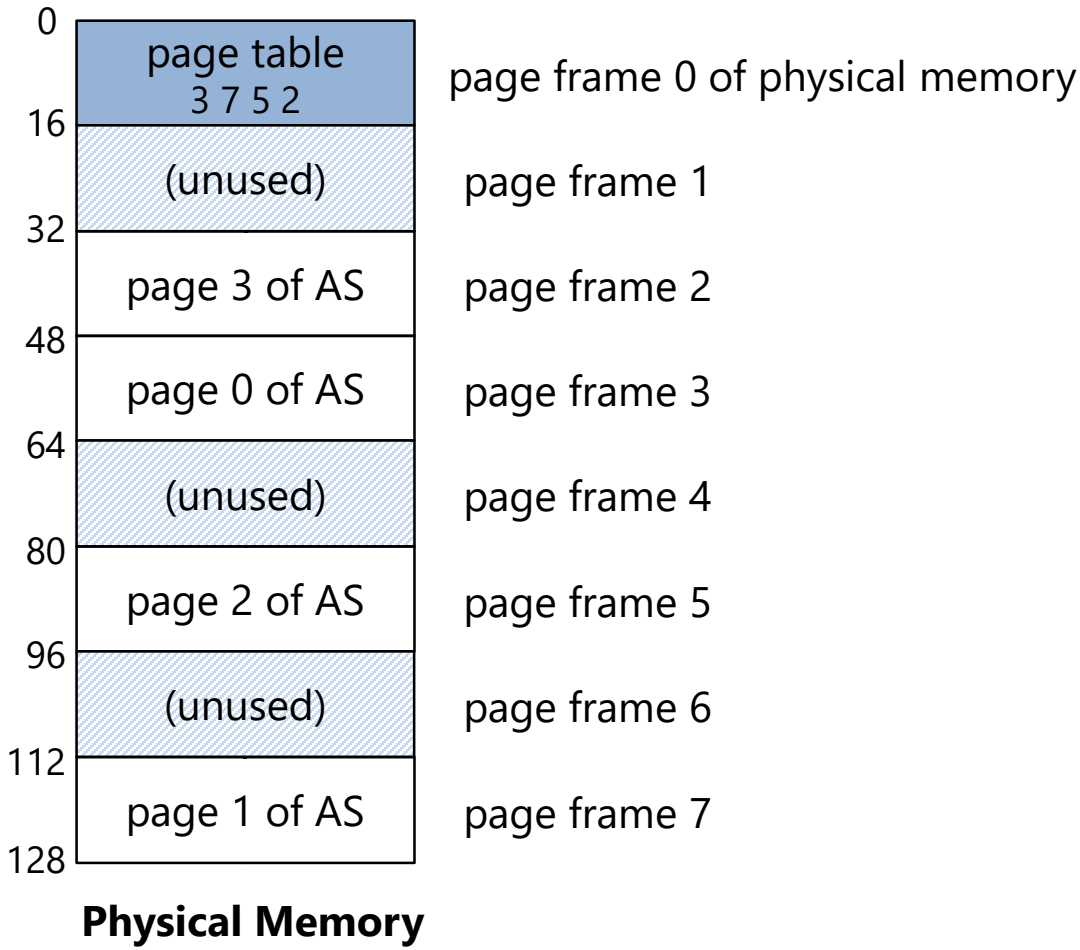
- The virtual address 21 in 64-byte address space



Where Are Page Tables Stored?

- Page tables can get awfully large
  - 32-bit address space with 4-KB pages, 20 bits for VPN
    - $4MB = 2^{20}\ entries * 4\ Bytes\ per\ page\ table\ entry$
- Page table for each processes is stored in memory.

Example: Page Table in Kernel Physical Memory



What Is In The Page Table?

- The page table is just a **data structure** that is used to map the virtual address to physical address.
  - Simplest form: a linear page table, an array

- The OS **indexes** the array by VPN, and looks up the page-table entry (PTE)
  - Purpose: to find the desired physical frame number (PFN)

	PFN	Other bits/Flags
VPN 0	PFN 37	
VPN1		
…	…	…
VPN *n*	PFN 23	

A typical Page Table

Common Flags Of Page Table Entry

- **Valid Bit**: Indicating whether the particular translation is valid.
  - Mark the unused VPNs invalid, no need to allocate physical frames
- **Protection Bit**: Indicating whether the page could be read from, written to, or executed from
- **Present Bit**: Indicating whether this page is in physical memory or on disk (swapped out)
  - *More on this when we'll talk about Swapping*
- **Dirty Bit**: Indicating whether the page has been modified since it was brought into memory
- **Reference Bit (Accessed Bit):** Indicating that a page has been accessed
  - Useful for page replacement policies

Paging: Too Slow

```
movl 21, %eax
```

- Address 21 needs to be mapped to the actual physical address
- But for this translation, OS needs to fetch the desired PTE
  - For that, the **starting location** of the page table is **needed**.
- For every memory reference, paging requires the OS to perform one **extra memory reference**.
- How to know the start location of the page table?
  - Simplest form: **Page-table base register** (PTBR)

Accessing Memory With Paging

```
1         // Extract the VPN from the virtual address
2         VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4         // Form the address of the page-table entry (PTE)
5         PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7         // Fetch the PTE
8         PTE = AccessMemory(PTEAddr)
9
10        // Check if process can access the page
11        if (PTE.Valid == False)
12                  RaiseException(SEGMENTATION_FAULT)
13        else if (CanAccess(PTE.ProtectBits) == False)
14                  RaiseException(PROTECTION_FAULT)
15        else
16                  // Access is OK: form physical address and fetch it
17                  offset = VirtualAddress & OFFSET_MASK
18                  PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19                  Register = AccessMemory(PhysAddr)
```

TLB (Translation Lookaside Buffer)

Revisit: Paging

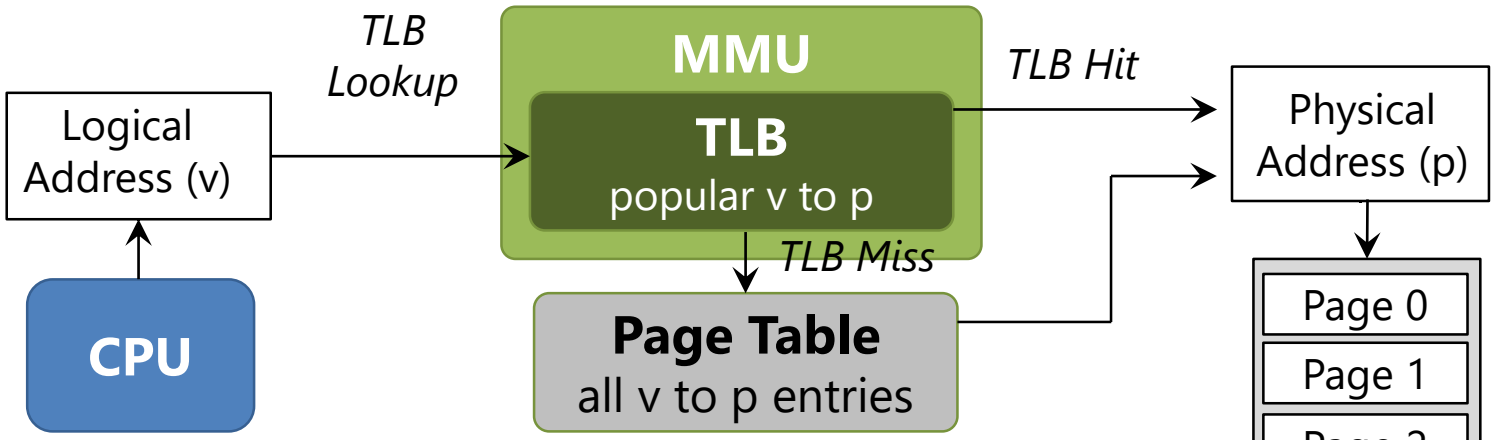
- The core mechanism of Paging solves the issues with Segmentation, but it is **too slow**
- It incurs high performance overhead
  - Large amount of mapping information
  - Logically require an extra memory reference for each instruction fetch or explicit load/store

How to speed up address translation while using Paging?

OS seeks help from the hardware

Translation-lookaside Buffer (TLB)

- Part of the chip's memory-management unit(MMU).
- A hardware cache of **popular** virtual-to-physical address translation.



Address Translation with MMU

Physical Memory

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:        Register = AccessMemory( PhysAddr )
8:    }else RaiseException(PROTECTION_FAULT)
```

- (1 lines) extract the virtual page number (VPN).
- (2 lines) check if the TLB holds the translation 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.

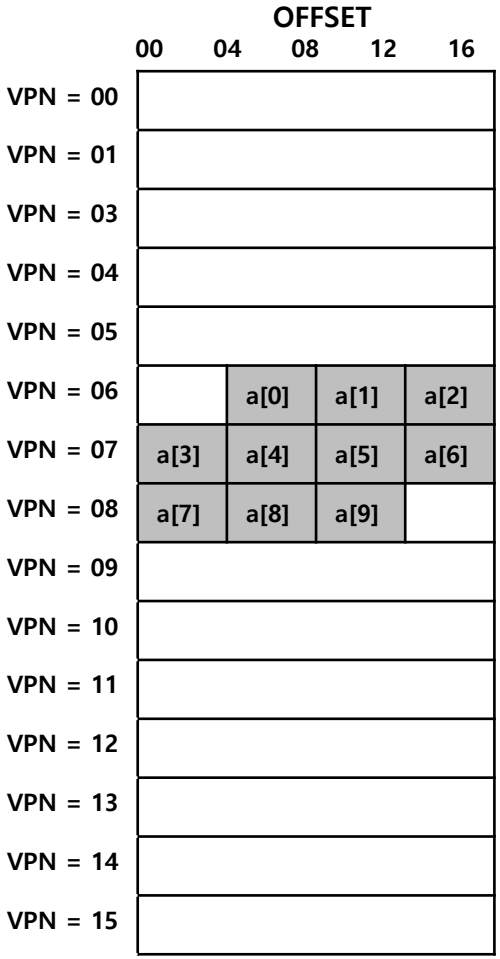
TLB Basic Algorithms

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

- (12-13 lines) The hardware accesses the page table to find the translation.
- (16 line) updates the TLB with the translation.
- (17 line) the hardware retry the instruction (will be a TLB hit)

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:      }
```

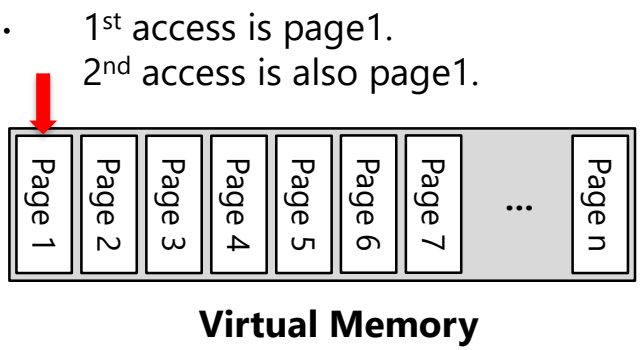
miss, hit, hit, miss, hit, hit, hit, miss, hit, hit

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

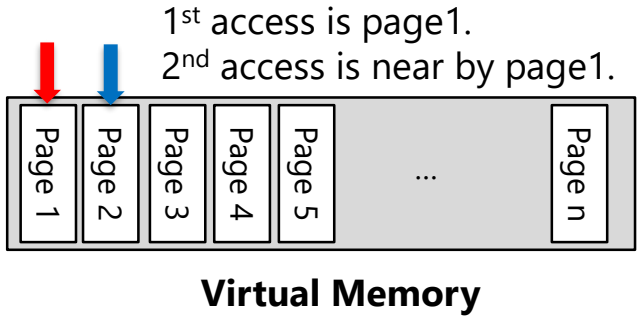
The TLB improves performance due to spatial locality

Locality

- Temporal Locality
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future.



- Spatial Locality
  - If a program accesses memory at address `x`, it will likely soon access memory near `x`.



Who Handles The TLB Miss?

- Hardware handles the TLB miss entirely on CISC (Complex-instruction set computers).
  - The hardware has to know exactly where the page tables are located in memory. (utilize PTBR)
  - 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?

- RISC (Reduced-instruction set computers) 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)

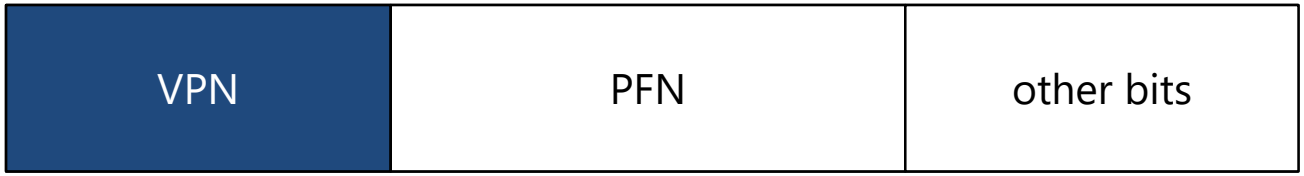
```
1:        VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:        (Success, TlbEntry) = TLB_Lookup(VPN)
3:        if (Success == True) // TLB Hit
4:                if (CanAccess(TlbEntry.ProtectBits) == True)
5:                        Offset = VirtualAddress & OFFSET_MASK
6:                        PhysAddr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

Issues with Software-Managed TLB

- Return-to-trap is different here
  - Why?: Hardware must **resume** execution at the instruction that caused the trap
  - Many solutions: e.g., H/W must **save** the proper PC when trapping into the OS
- May cause infinite chain of TLB misses
  - Why?: TLB miss-handling code within the OS
  - Many solutions: e.g.,
    - **keep** TLB miss-handling code in physical memory (not subject to address translation)
    - **Reserve** some entries in TLB for permanently-valid translation

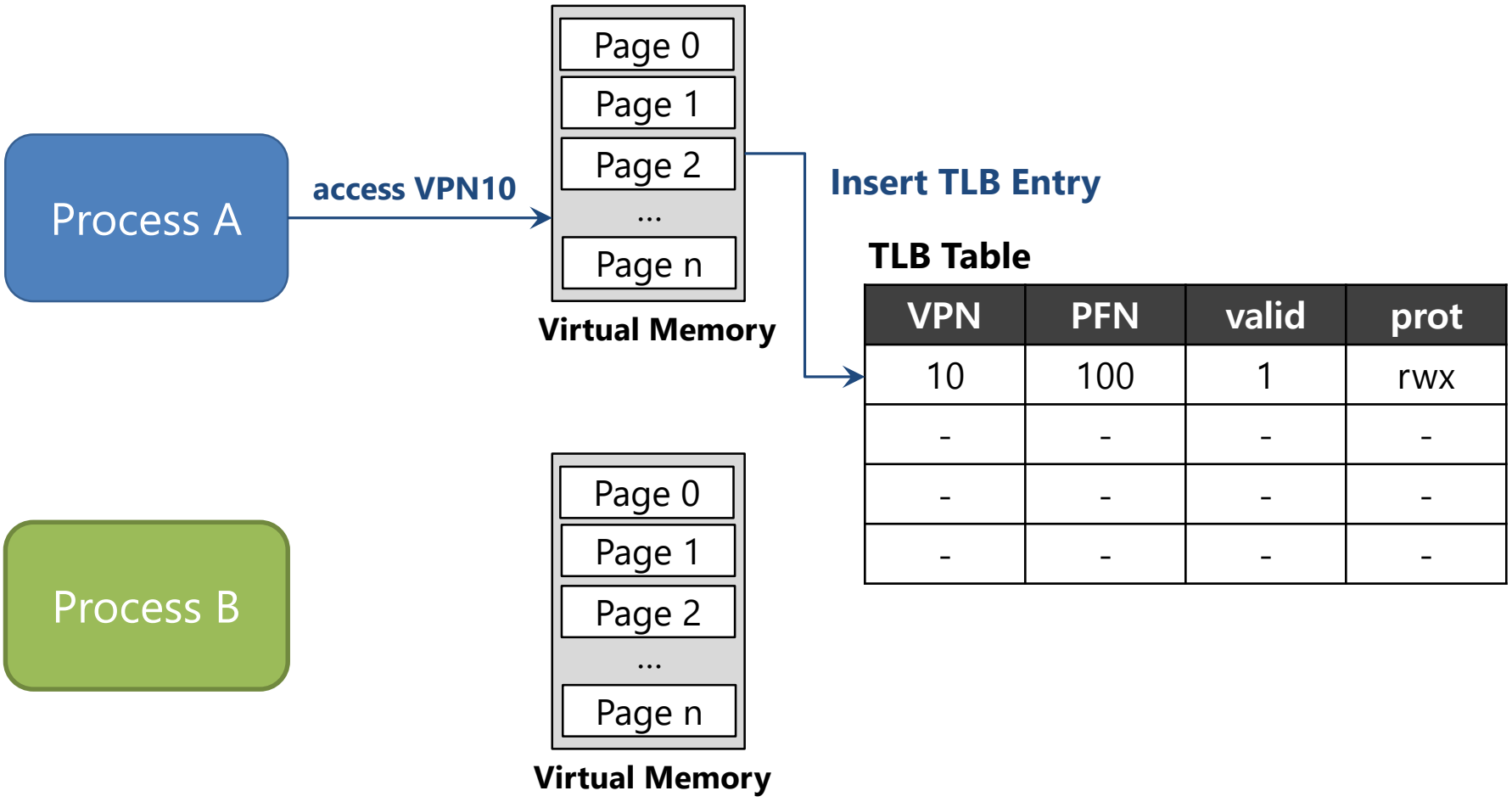
TLB entry

- 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

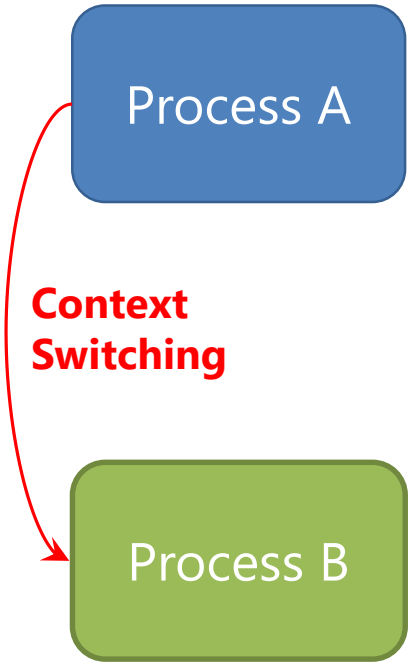


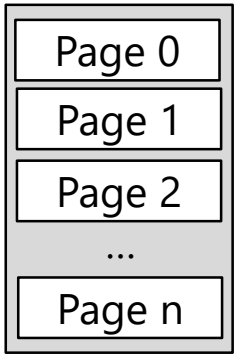
Typical TLB entry look like this

TLB Issue: Context Switching

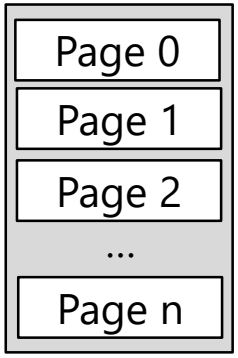


TLB Issue: Context Switching





Virtual Memory

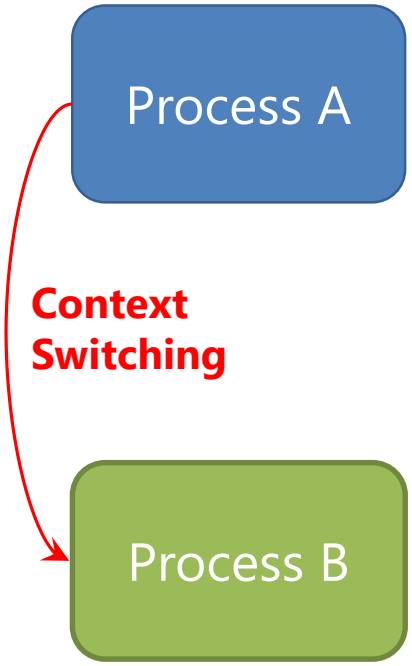


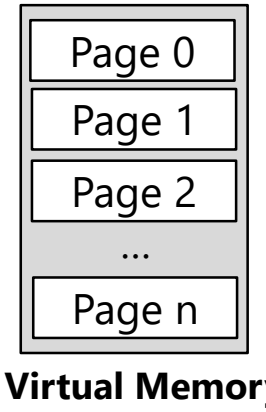
Virtual Memory

TLB Table

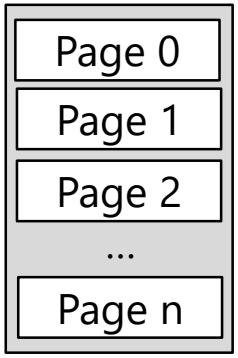
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
-	-	-	-
-	-	-	-

TLB Issue: Context Switching





Virtual Memory



Virtual Memory

Option 1 (flush)

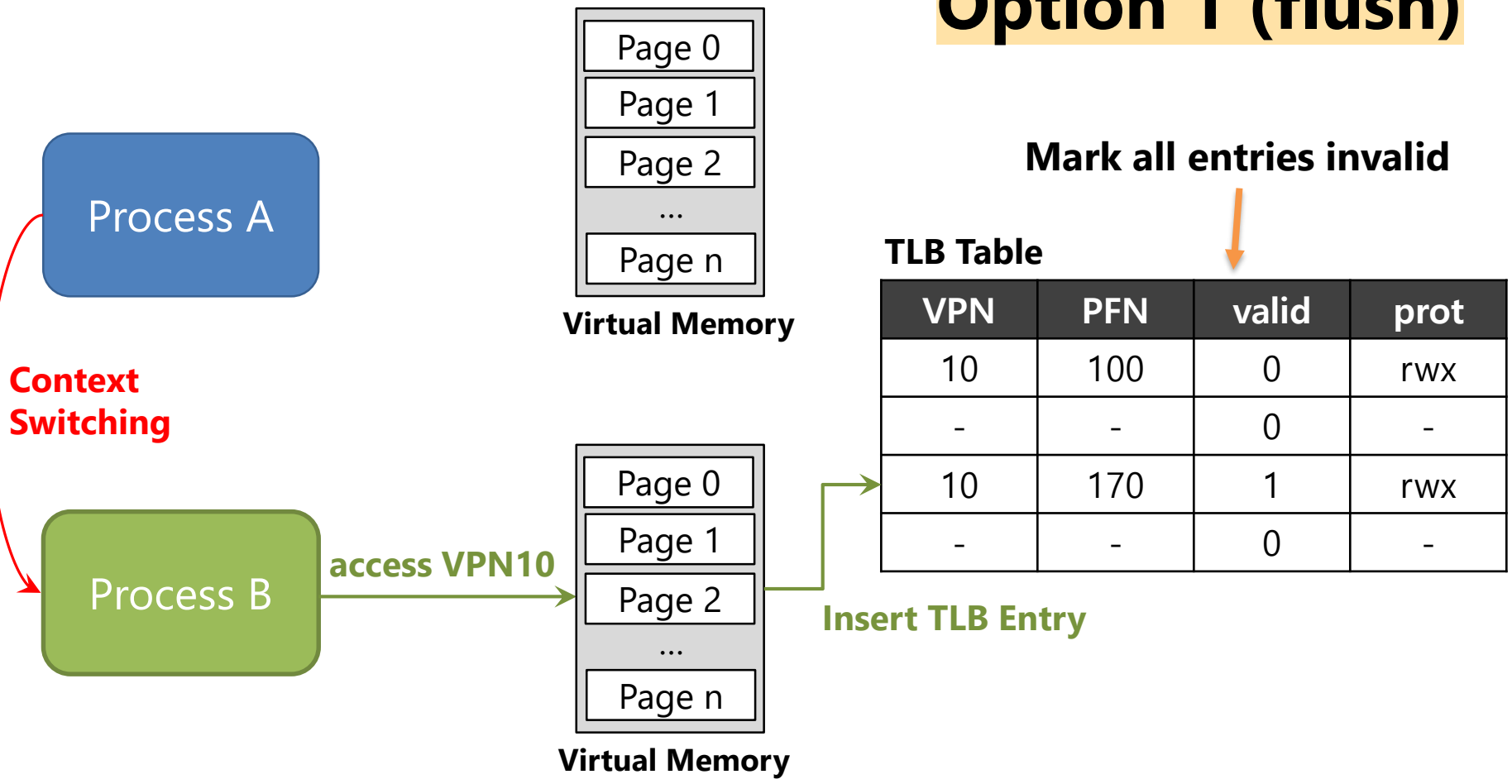
Mark all entries invalid

TLB Table

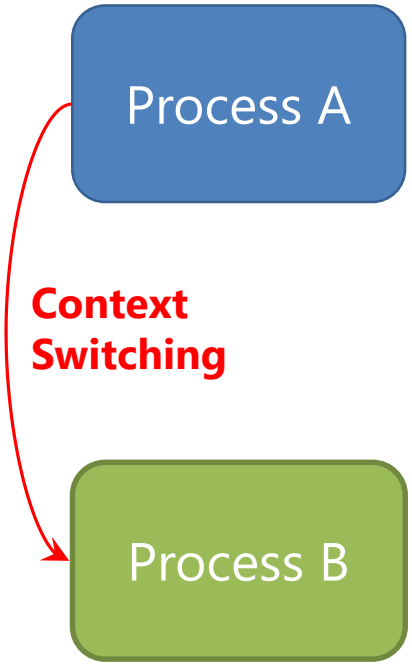
VPN	PFN	valid	prot
10	100	0	rwx
-	-	0	-
-	-	0	-
-	-	0	-

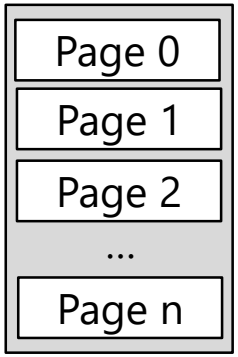
TLB Issue: Context Switching

Option 1 (flush)

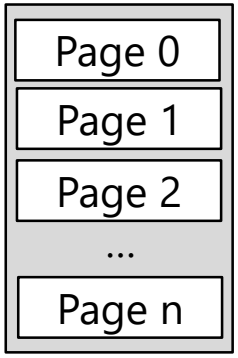


TLB Issue: Context Switching





Virtual Memory



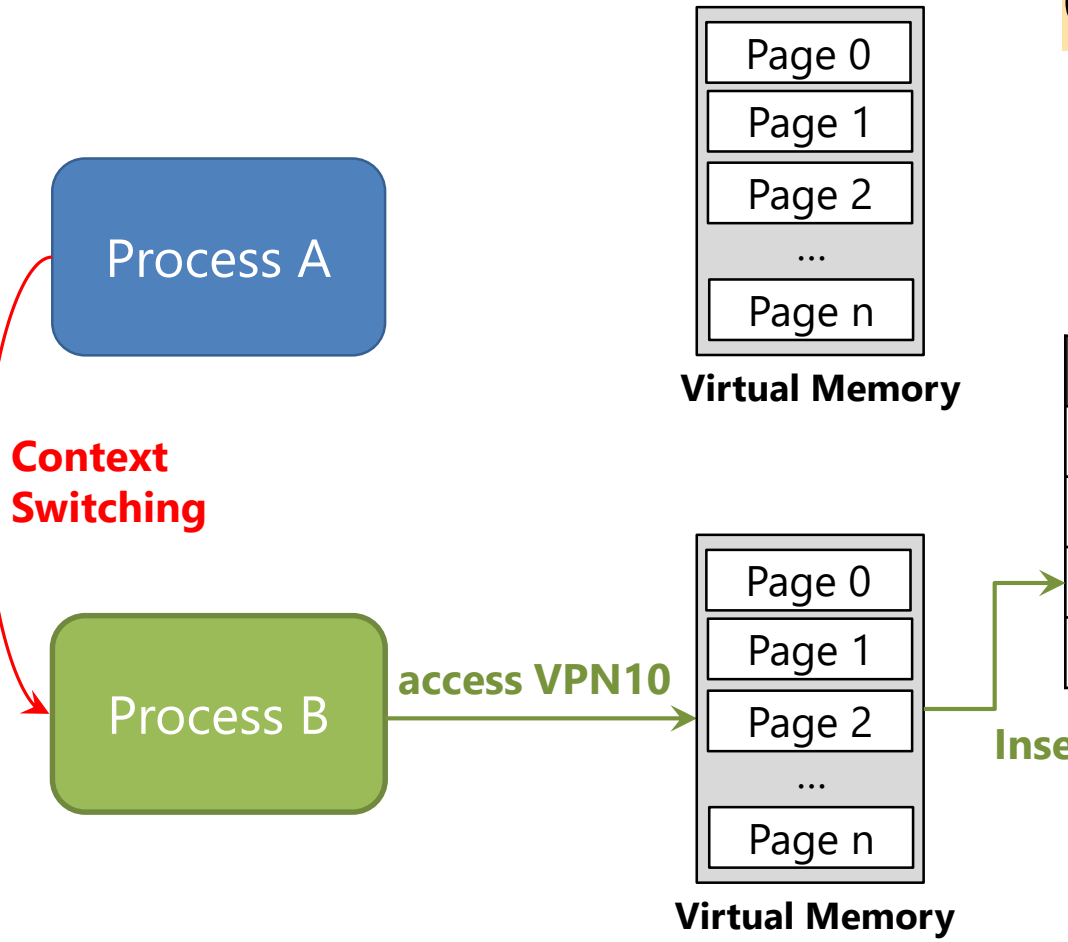
Virtual Memory

TLB Table

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

TLB Issue: Context Switching

Option 2 (sharing)



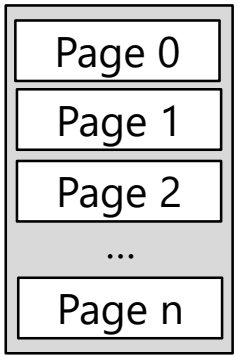
TLB Table

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

TLB Issue: Context Switching

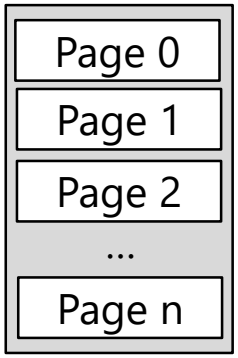
Option 2 (sharing)

Process A



Virtual Memory

Process B



Virtual Memory

TLB Table

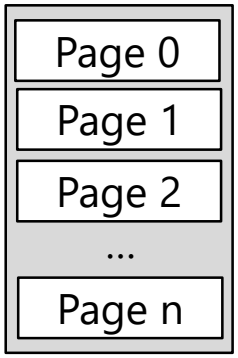
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

TLB Issue: Context Switching – Solve the problem

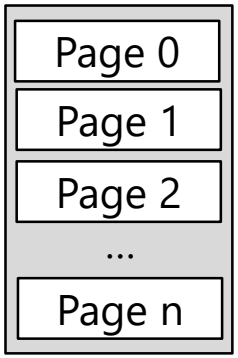
Option 2 (sharing)

Process A



Virtual Memory

Process B



Virtual Memory

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

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
-	-	-	-	-

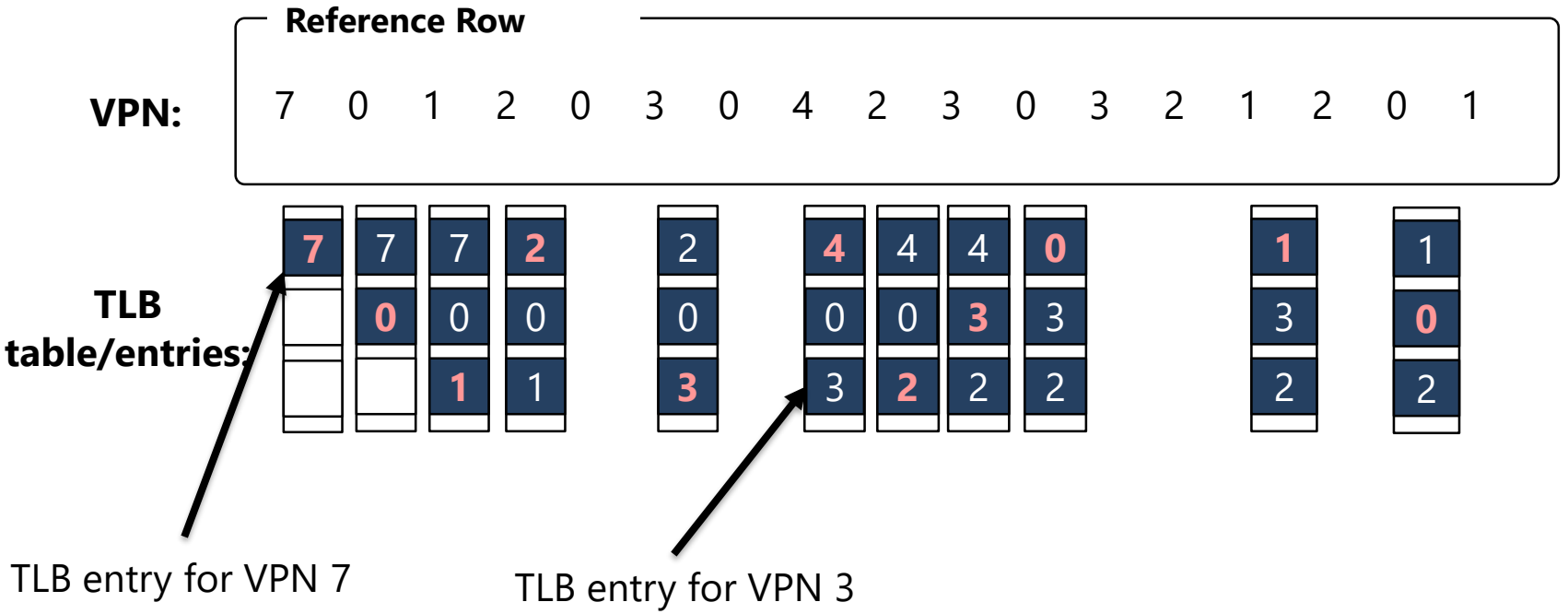
Sharing of pages is useful as it reduces the number of physical pages in use.

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.

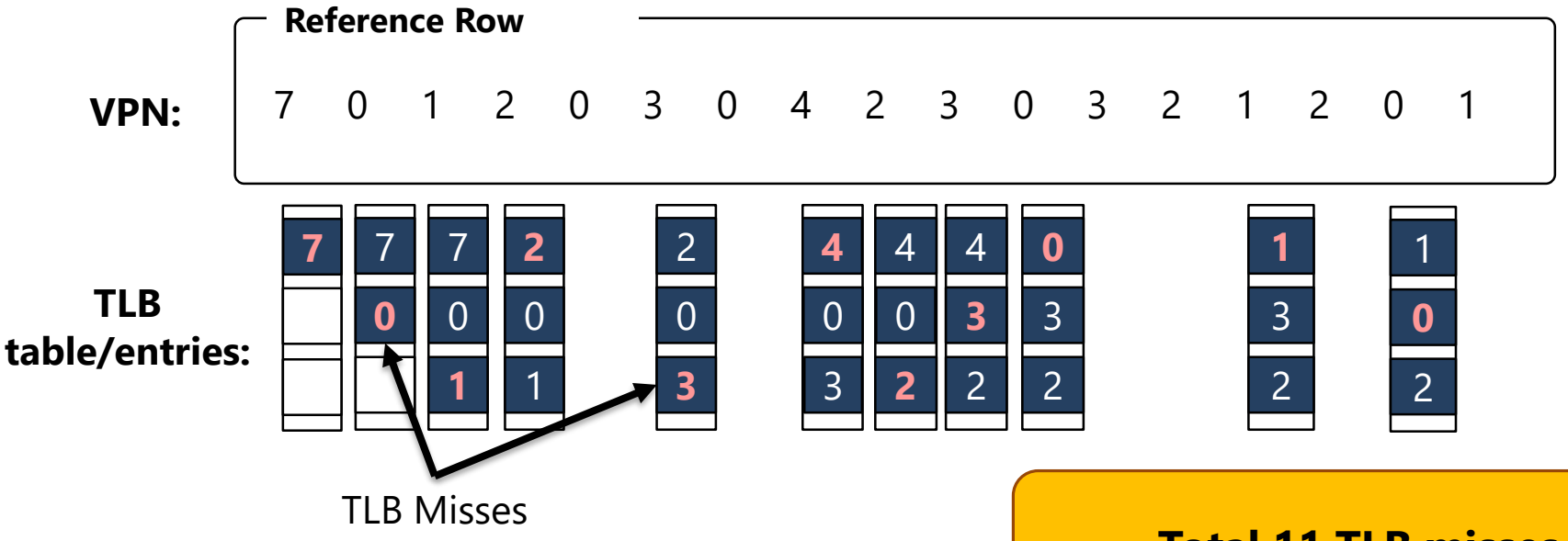
TLB Replacement Policy

- Example: LRU
  - Consider an address space with 8 pages (0-7)
  - Consider a TLB table that can hold only 3 entries



TLB Replacement Policy

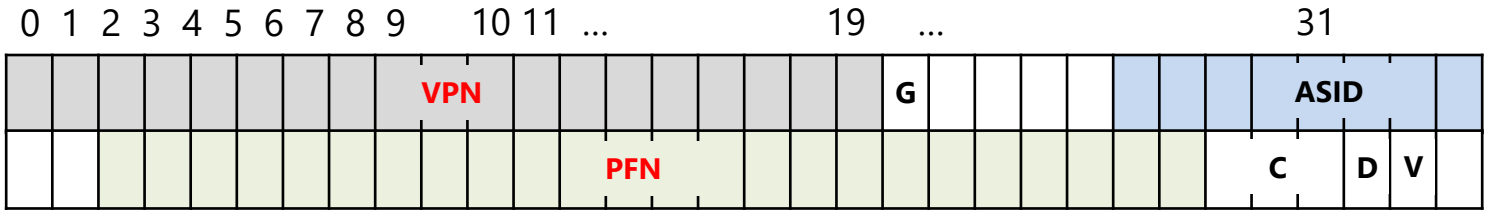
- Example: LRU
  - Consider an address space with 8 pages (0-7)
  - Consider a TLB table that can hold only 3 entries



Total 11 TLB misses

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.

Reading Material

- **Chapter 18-19** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)
  http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf
  http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf

Questions?