

Operating Systems

Paging

Concept of Paging

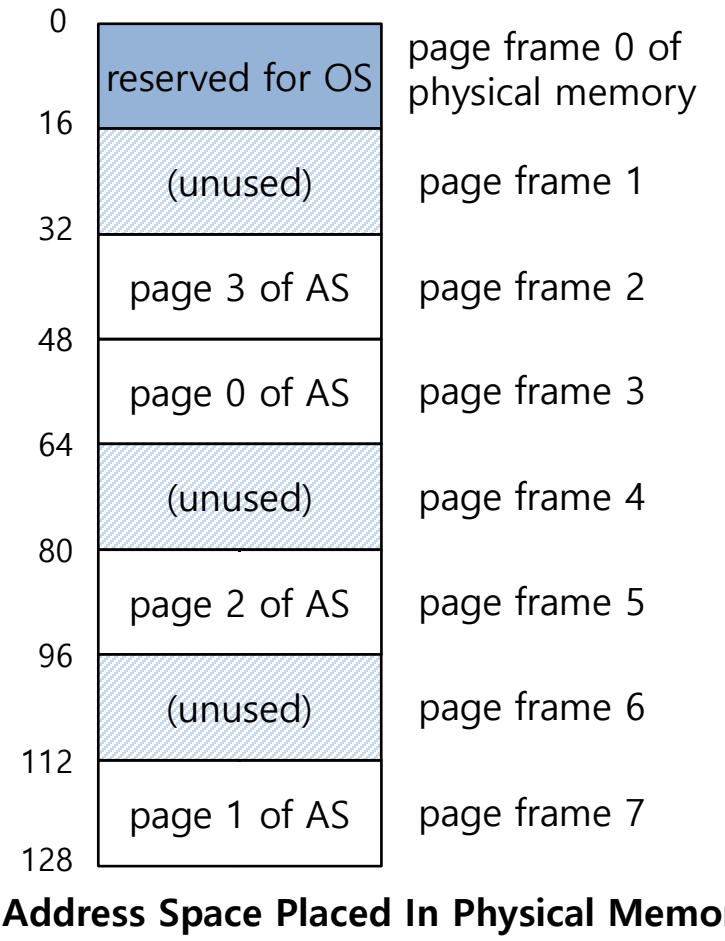
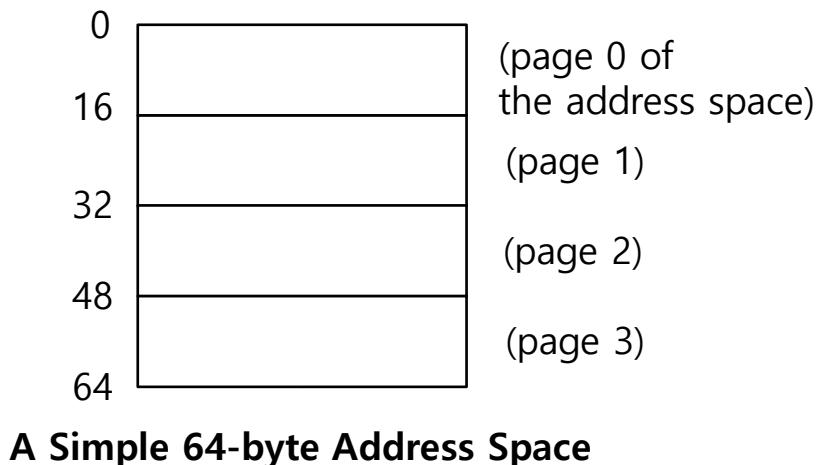
- ▣ Paging **splits up** address space into **fixed-sized** unit called a **page**.
 - ◆ Segmentation: variable size of logical segments(code, stack, heap, etc.)
- ▣ With paging, **physical memory** is also **split** into some number of pages called a **page frame**.
- ▣ **Page table** per process is needed **to translate** the virtual address to physical address.

Advantages Of Paging

- ▣ **Flexibility:** Supporting the abstraction of address space effectively
 - ◆ Don't need assumption 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

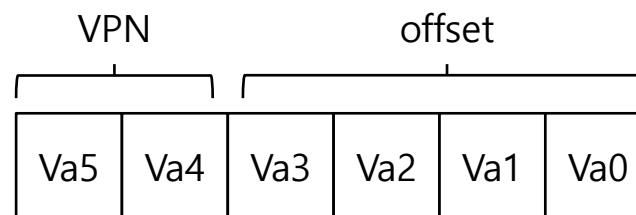
- 128-byte physical memory with 16 bytes page frames
- 64-byte address space with 16 bytes pages



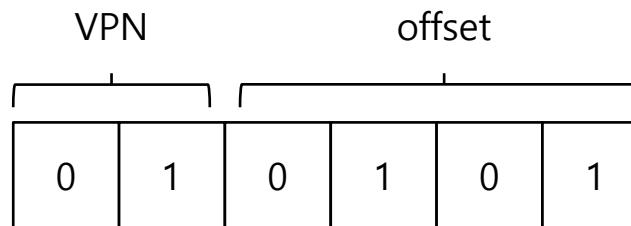
Address Translation

- Two components in the virtual address

- VPN: virtual page number
- Offset: offset within the page

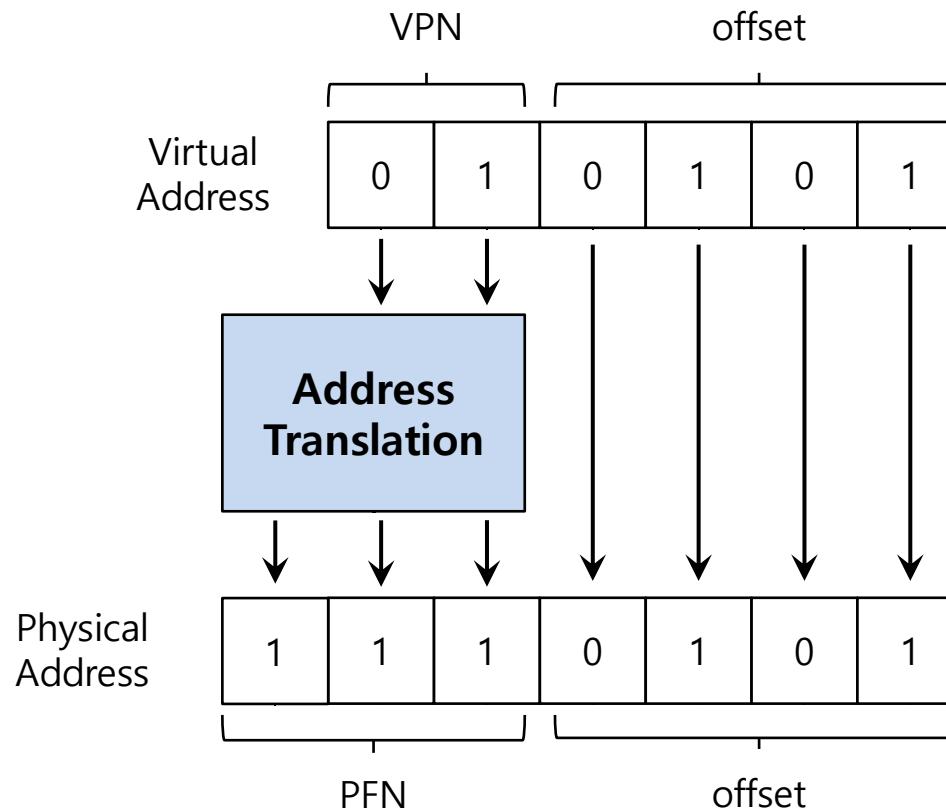


- Example: virtual address 21 in 64-byte address space



Example: Address Translation

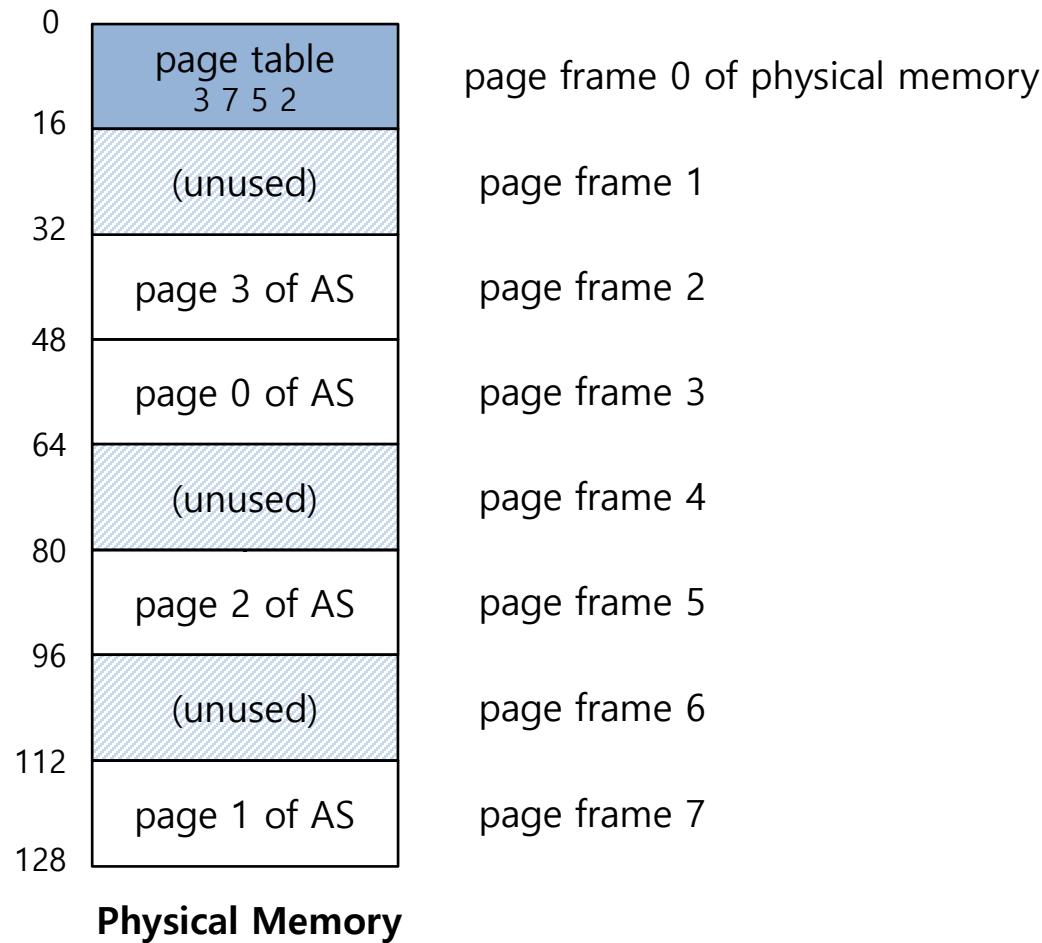
- The virtual address 21 in 64-byte address space



Where Are Page Tables Stored?

- ▣ Page tables can get awfully large
 - ◆ 32-bit address space with 4-KB pages, 20 bits for VPN
 - $4MB = 2^{20} \text{ entries} * 4 \text{ Bytes per page table entry}$
- ▣ Page tables for each process are 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.

Common Flags Of Page Table Entry

- ▣ **Valid Bit:** Indicating whether the particular translation is valid.
- ▣ **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)
- ▣ **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

Paging: Too Slow

- ❑ To find a location of the desired PTE, the **starting location** of the page table is **needed**.
- ❑ For every memory reference, paging requires the OS to perform one **extra memory reference**.

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)
```

A Memory Trace

- Example: A Simple Memory Access

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

- Compile and execute

```
prompt> gcc -o array array.c -Wall -o
prompt>./array
```

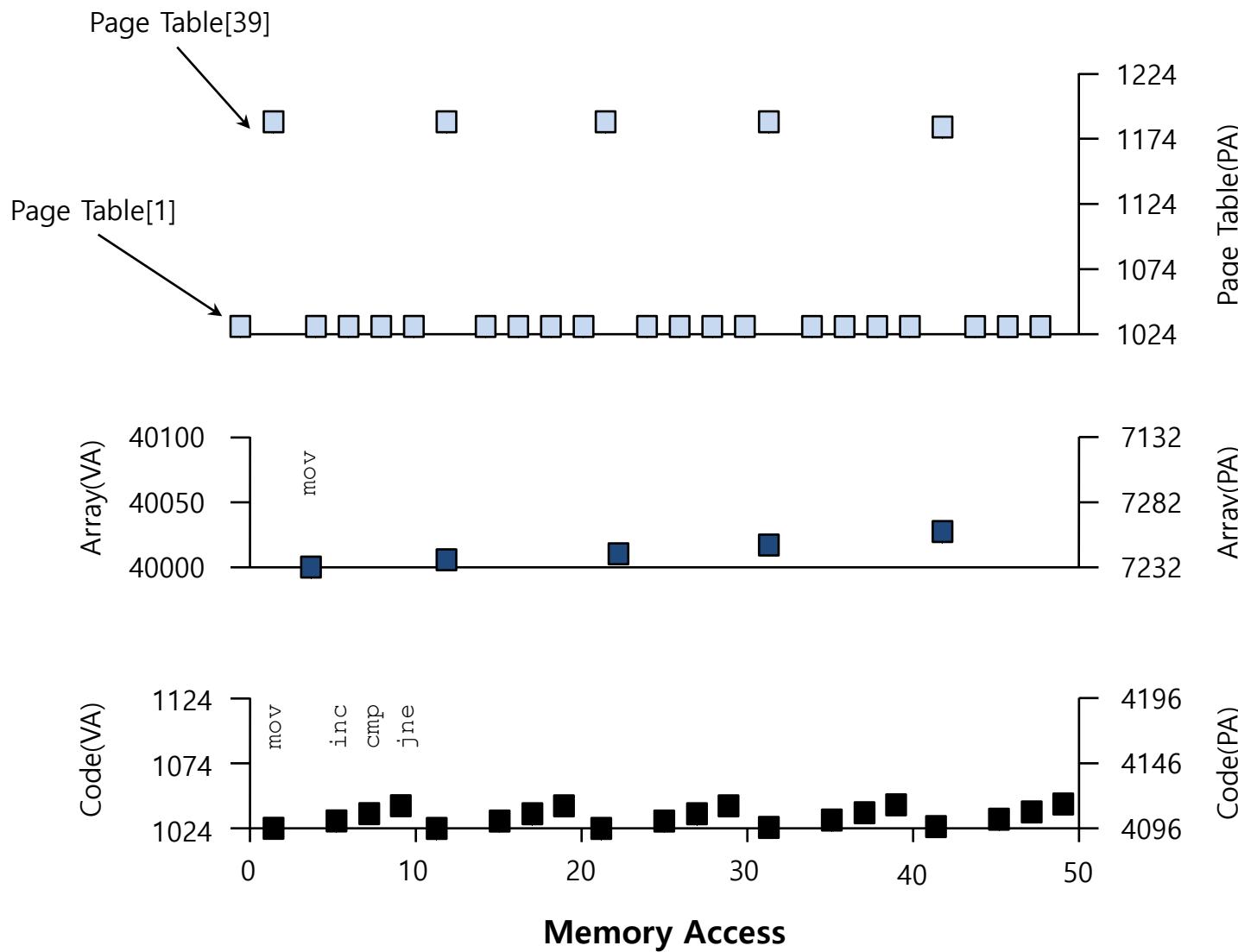
- Resulting Assembly code

```
0x1024 movl $0x0, (%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

A Virtual(And Physical) Memory Trace

- We assume a virtual address space of size 64 KB (unrealistically small). We also assume a page size of 1 KB
- Code resides at 1024
- There is the array itself; Its size is 4000 bytes (1000 integers), and it lives at virtual addresses 40000 through 44000
- Let's assume
- VPN 1 → PFN 4
- VPN 39 → PFN 7
- VPN 40 → PFN 8
- VPN 41 → PFN 9
- VPN 42 → PFN 10
- Page table is located at physical address 1 KB (1024).

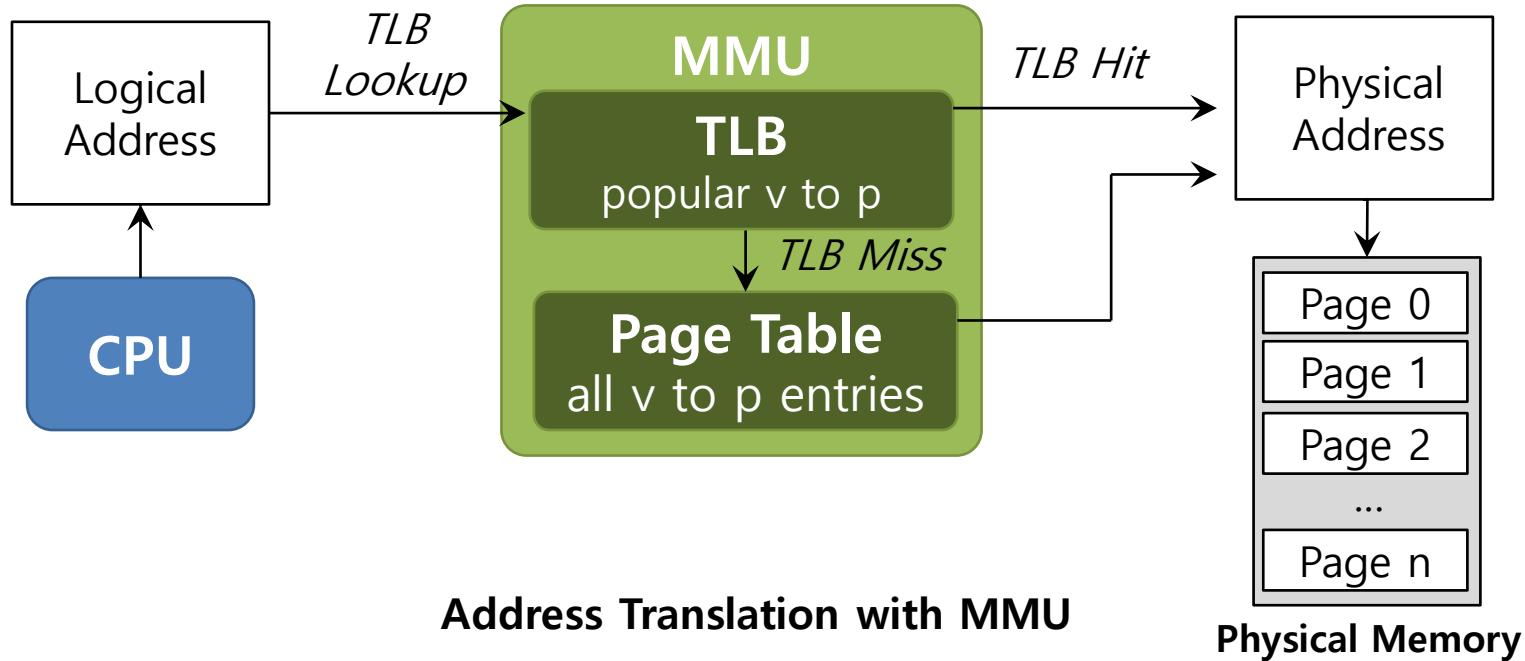
A Virtual(And Physical) Memory Trace



Translation Lookaside Buffers

Translation Lookaside Buffers (TLB)

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



TLB Basic Algorithms

```
1: VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2: (Success, TlbEntry) = TLB_Lookup(VPN)
3: if (Success == True) { // 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 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.

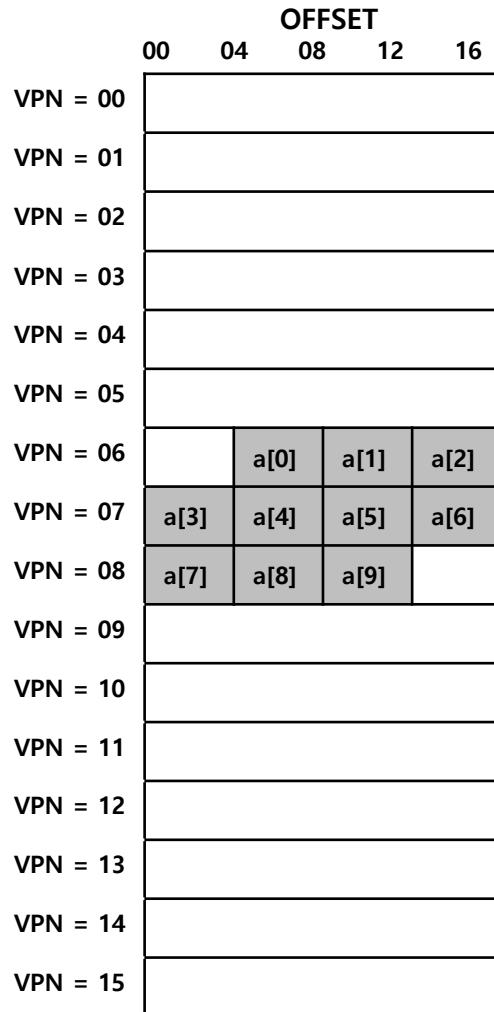
TLB Basic Algorithms (Cont.)

```
11:     }else{ //TLB Miss  
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))  
13:         PTE = AccessMemory(PTEAddr)  
14:         (...)  
15:  
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:     }
```

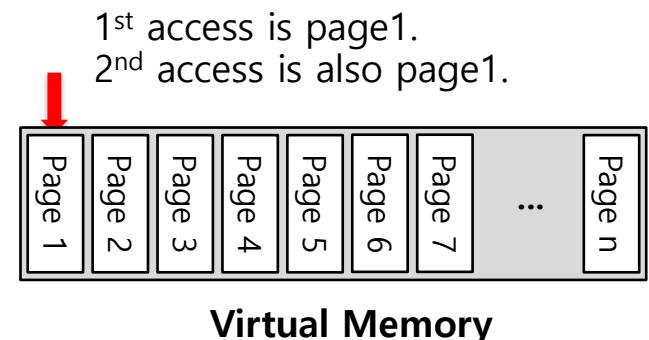
The TLB improves performance
due to spatial locality

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

Locality

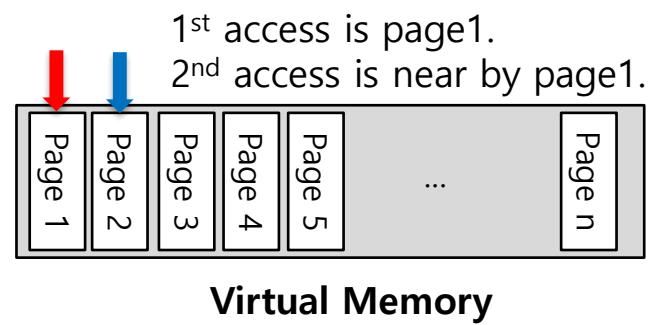
Temporal Locality

- 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 .



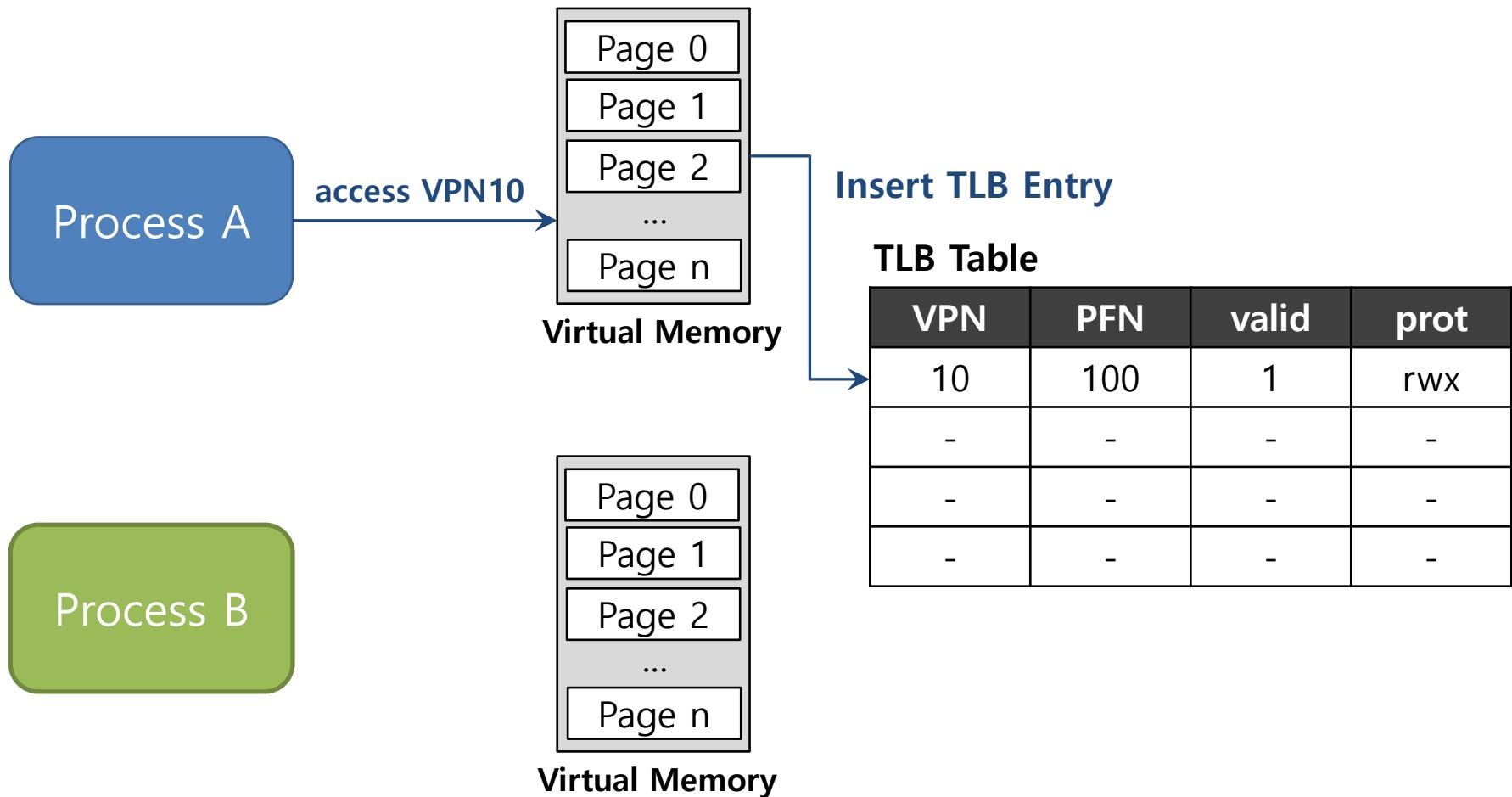
TLB entry

- 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

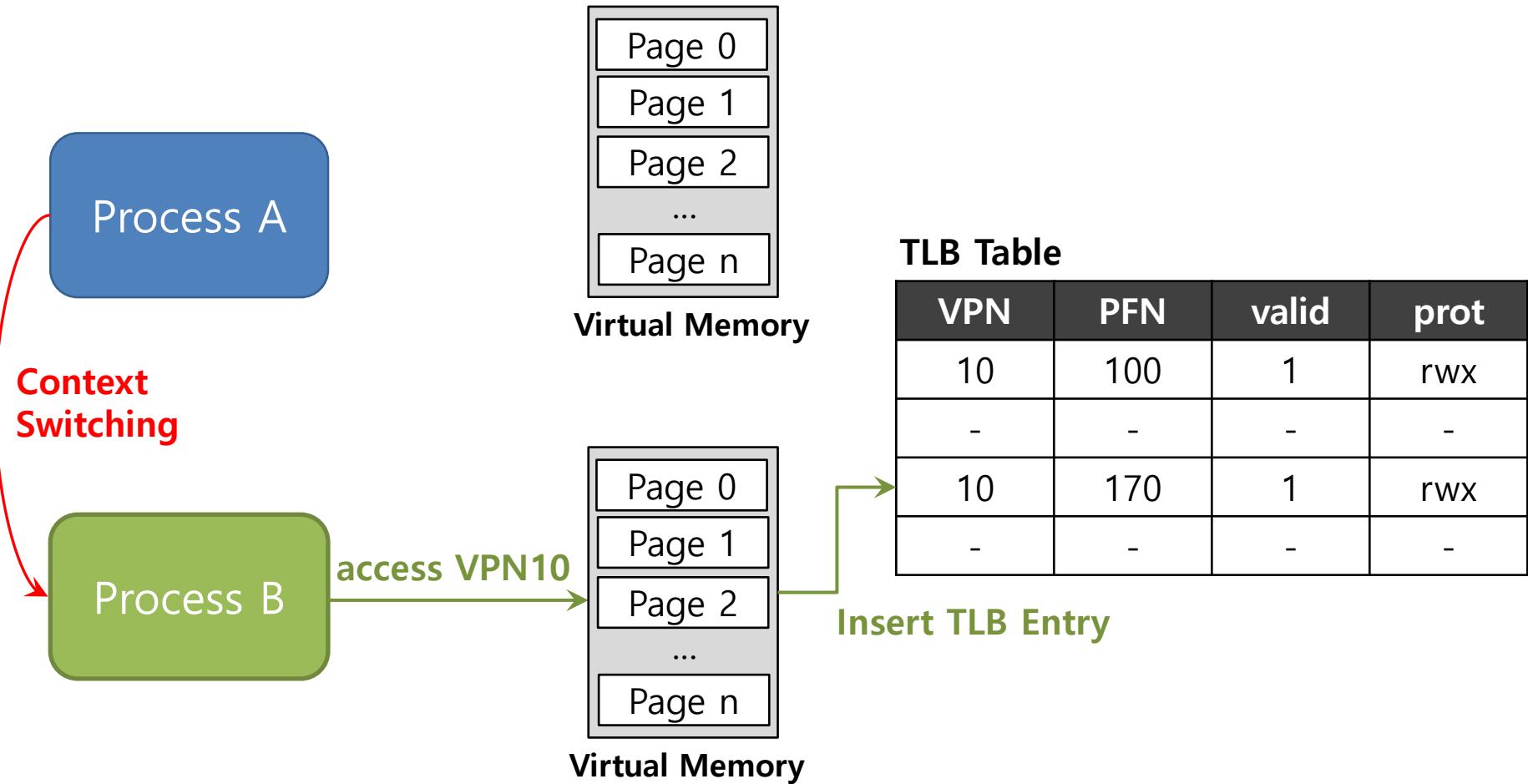


Typical TLB entry look like this

TLB Issue: Context Switching

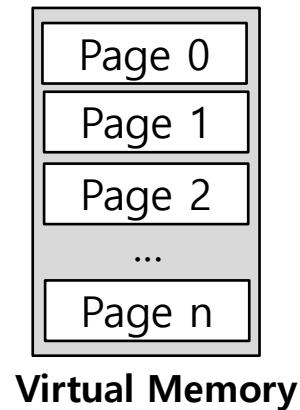


TLB Issue: Context Switching

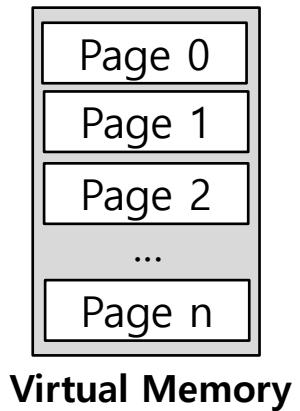


TLB Issue: Context Switching

Process A



Process B



TLB Table

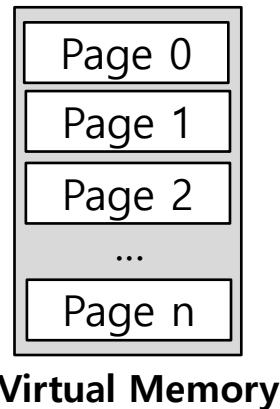
VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

To Solve Problem

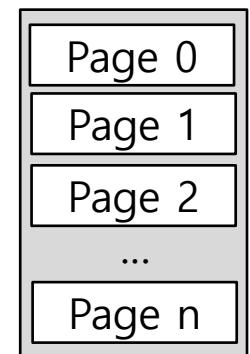
- Provide an address space identifier(ASID) field in the TLB.

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.

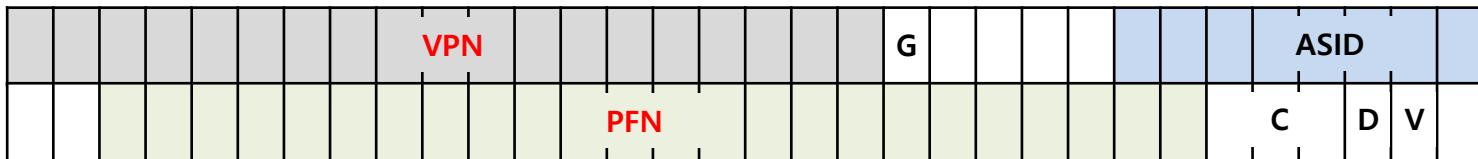
VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

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

A Real TLB Entry

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

0 1 2 3 4 5 6 7 8 9 10 11 ... 19 ... 31



Flag	Content
9999-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support 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.