Paging: Introduction

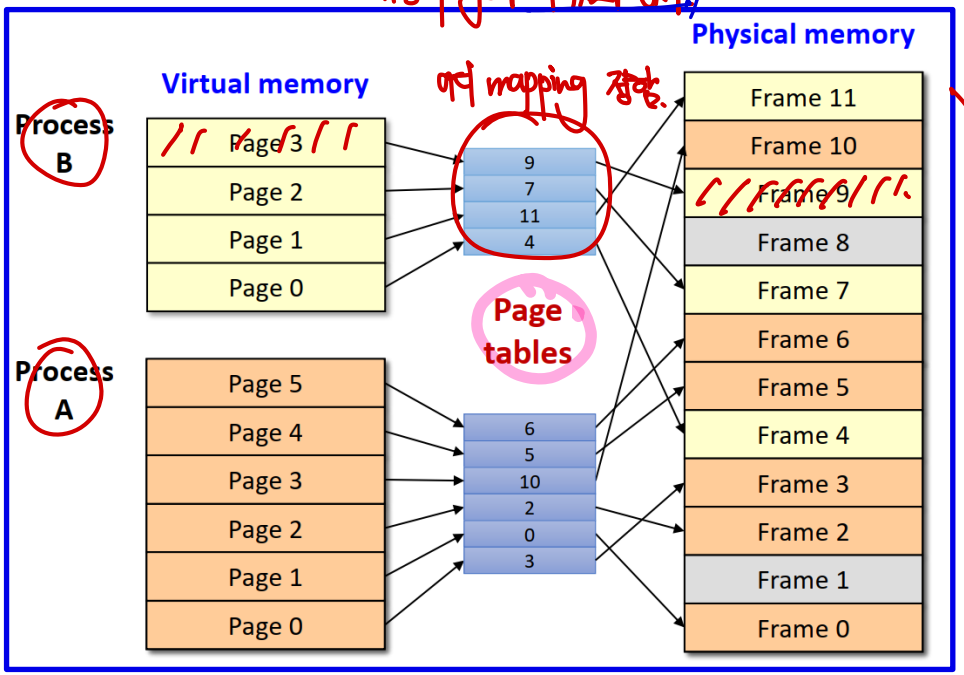
Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Paging: Concept

- **OS takes one of two approaches memory space-management:**
  1) Chopping the space up into variable-sized pieces (Segmentation)
  2) Chopping the space up into fixed-sized pieces (Paging)

- **Segmentation has inherent difficulties**
  - The space itself can become fragmented when dividing a space into different-size chunks
  - Thus, the allocation becomes more challenging over time

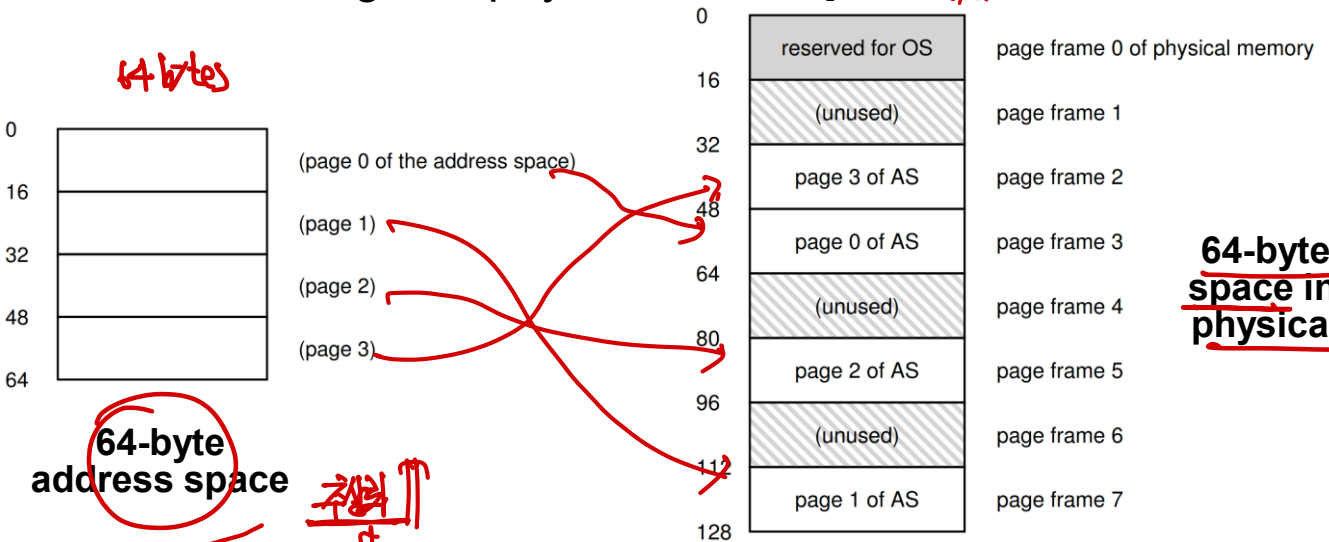


- **Paging divides the space into fixed-sized units**
  - Each of the unit is called a page
  - We view physical memory as an array of fixed-sized slots called page frames
  - Each frame can contain a single virtual-memory page

Courtesy of Prof. Jin-Soo Kim @ SNU

Simple Example

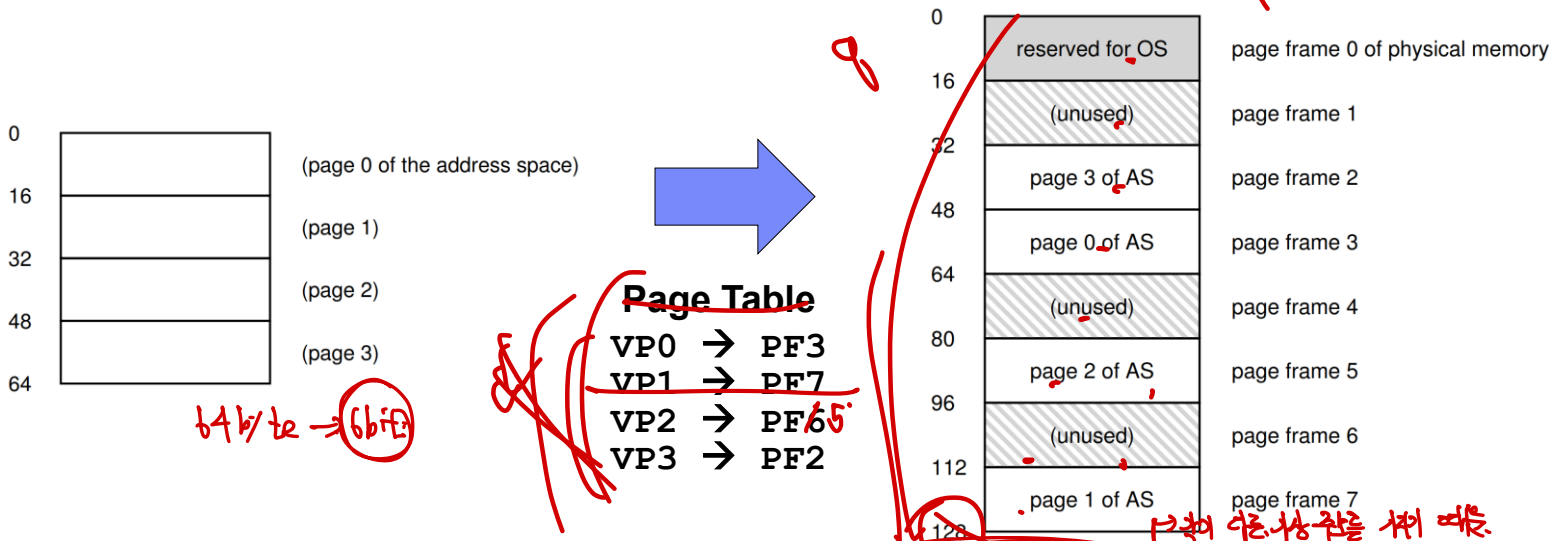
- **Let's consider an example of a tiny address space:**
  - Only 64 bytes total in size, with four 16-byte pages (virtual pages 0, 1, 2, 3)
  - Physical memory consists of eight fixed-sized slots, making 128-byte memory
  - As can be seen, the pages of the virtual address space have been placed at different locations throughout physical memory



- **Paging has a number of advantages over previous approaches**
  - Flexibility: supporting abstraction of address space effectively; we don't need to know the growing directions for heap and stack
  - Simplicity: ease of free-space management; simple free list (same sized units)

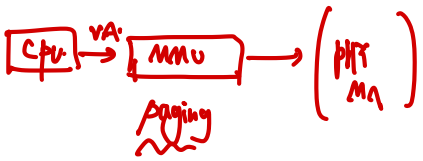
Paging Overview

- **To record where each virtual page is placed in physical memory, OS keeps a per-process data structure known as a page table**
  - The major role of the page table is to store address translations for each of the virtual pages of the address space (mapping from virtual to physical)
  - The page table for the previous example would have the following four entries: (Virtual Page 0 → Physical Frame 3), (VP 1 → PF 7), (VP 2 → PF 5), (VP 3 → PF 2)

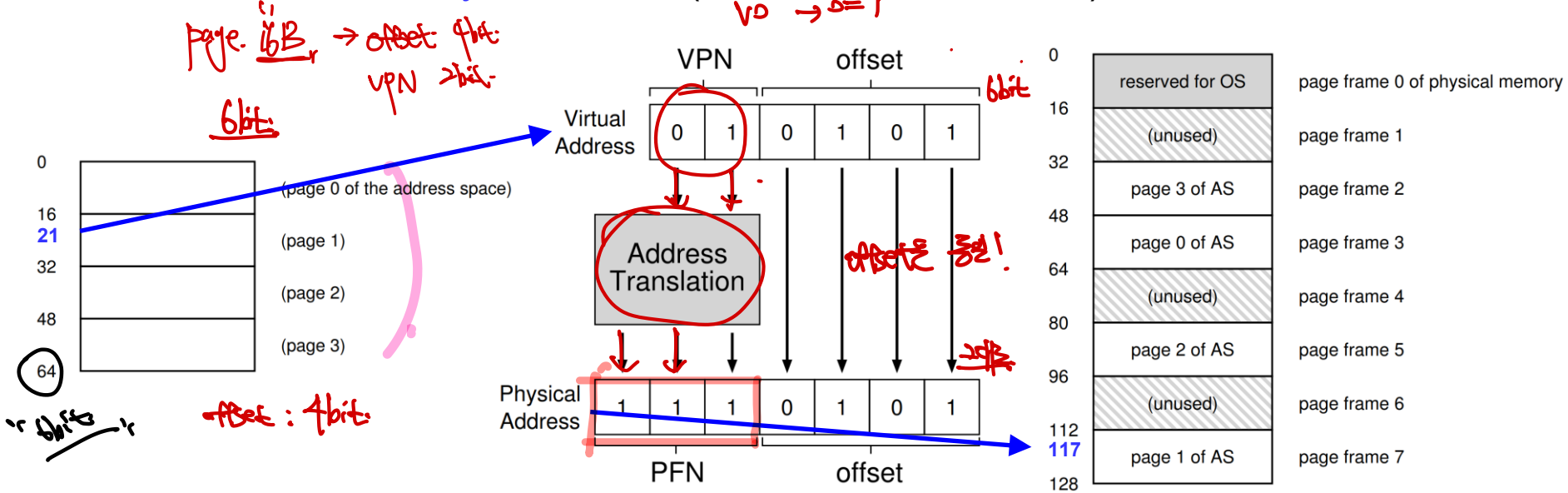


- **Remember that this page table is a per-process data structure**
  - If another process were to run for the example, OS have to manage a different page table for it, as its virtual pages obviously map to different physical pages

Address Translation

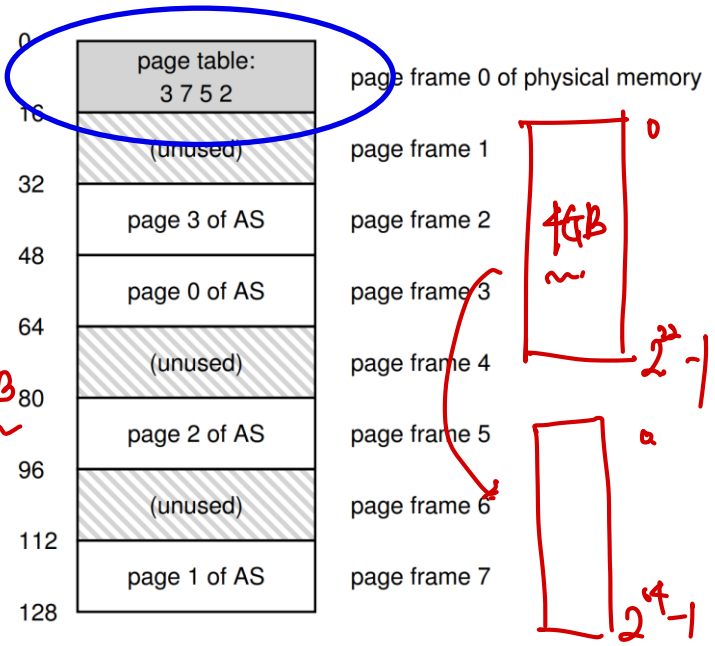


- **To translate virtual address, we first split it into two components: virtual page number (VPN) and offset within the page**
  - In the example, we need 6-bit address (2^6=64); 4-bit offset (2^4=16-byte page) and 2-bit VPN (4 pages: each page size is 16 bytes in a 64-byte address space)
  - To access the virtual address 21 (01_0101), VPN of 01 is translated into physical page frame number (PFN) of 7 (111); replacing VPN with PFN
  - PFN is referred to as physical page number (PPN), Usually, # of VPN ≥ # of PFN
  - Note the offset stays the same (i.e., it is not translated)



Where Are Page Table Stored?

- **Page tables can get terribly large, much bigger than the small segment table or base/bounds pair we discussed earlier**
  - 32-bit address space with 4KB pages → 20-bit VPN and 12-bit offset (2^{12}=4KB)
  - A 20-bit VPN implies that there are 2^{20} translations that the OS would have to manage for each process
  - If each page table entry (PTE) in the page table needs 4 bytes to hold address translation and useful stuffs, total 4MB ($4B \times 2^{20}$) is required for each page table
  - What if 100 processes is running? → 400MB!
  - What if 64-bit address space? → !!
  - Since page tables are so big, the MMU cannot hold them in it but store the tables in memory
  - Usually stored in PCB in kernel space

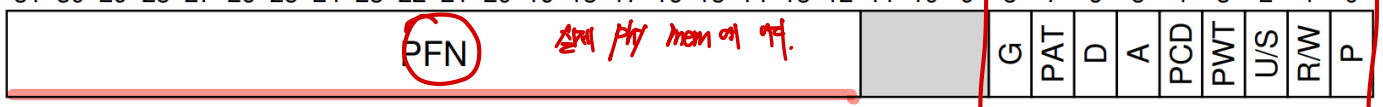


What's Actually in the Page Table?

- **The page table is a data structure that is used to map virtual addresses (i.e. VPN) to physical address (i.e. PFN)**
  - The simplest form is called a linear page table (array)
  - OS indexes the array by the VPN, and looks up the PTE at that index to find the corresponding PFN
- **Each PTE includes several information bits**

Bit	Description
Valid Bit	whether the particular translation is valid
Protection Bit	whether the page could be read from, written to, or executed from
Present Bit	whether this page is in physical memory or on disk
Dirty Bit	whether the page has been modified since it was brought into memory
Reference Bit	whether a page has been accessed (for page replacement)

x86 PTE



PWT, PCD, PAT, and G bits determine how hardware caching works for the pages

Paging: Also Too Slow

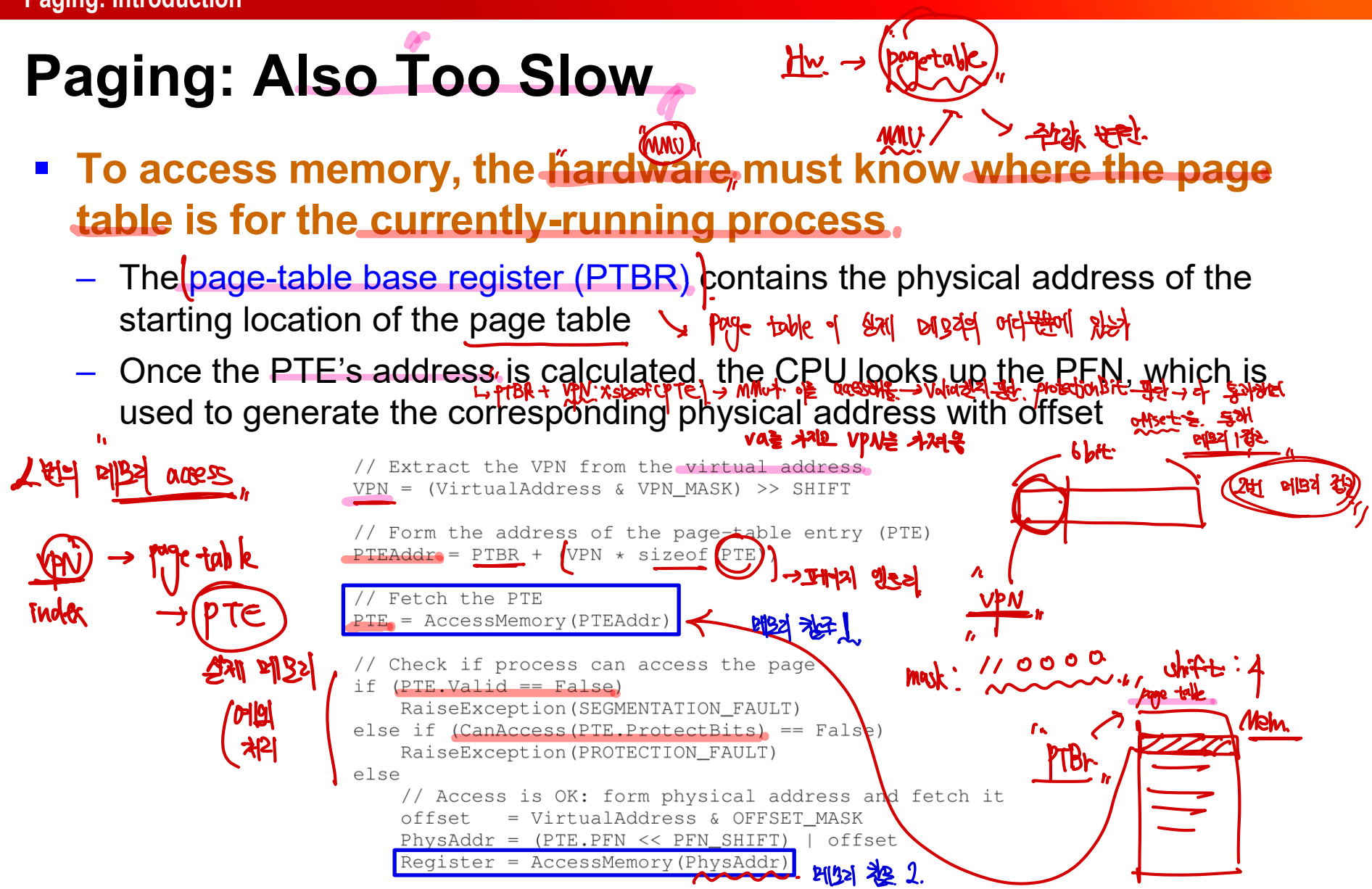
- **To access memory, the hardware must know where the page table is for the currently-running process.**
  - The page-table base register (PTBR) contains the physical address of the starting location of the page table
  - Once the PTE's address is calculated, the CPU looks up the PFN, which is used to generate the corresponding physical address with offset

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```



  - For every memory reference, paging requires us to perform one extra memory reference to first fetch the translation from page table → a lot of extra work!

A Memory Trace

- **To trace memory access, let's see an example with assumptions**

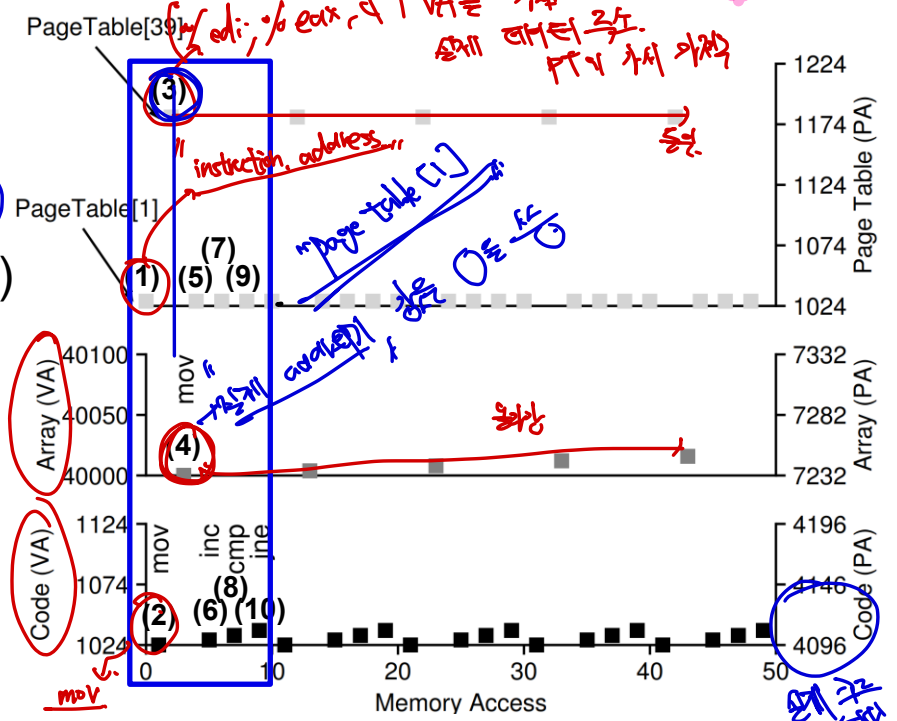
```
int array[1000];
..
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

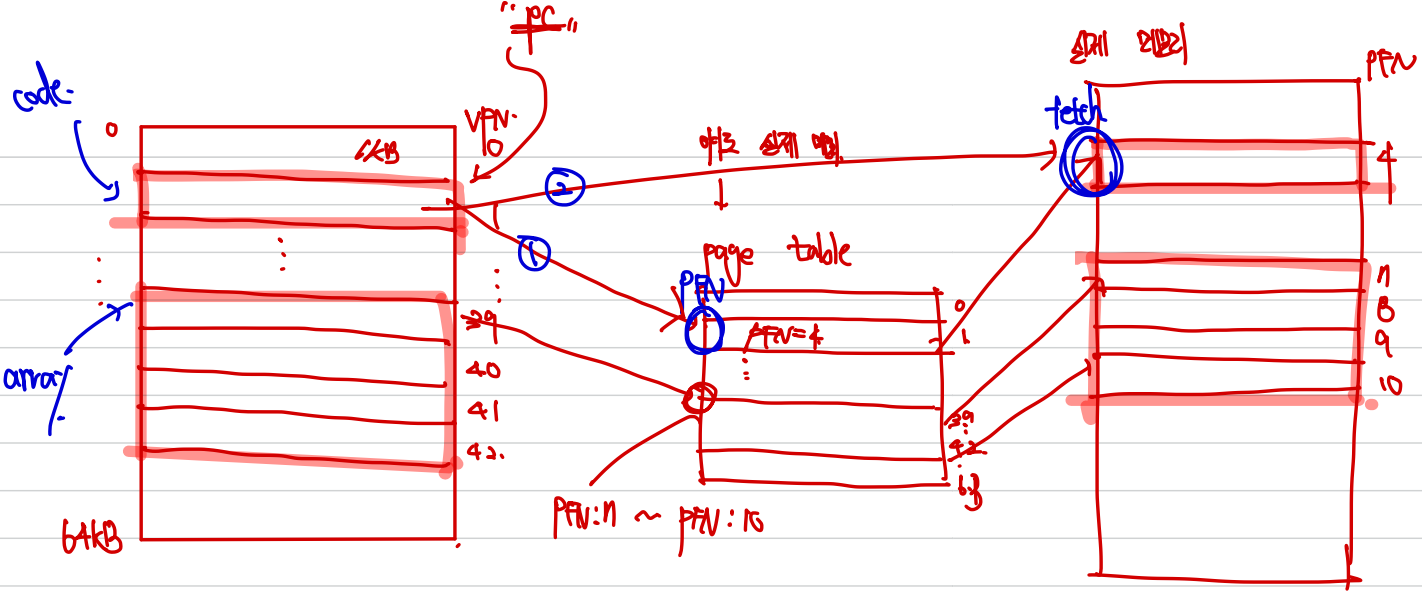
Compile

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

%edi: base address of array
%eax: i

  - We have virtual address space of 64KB and page size of 1KB, and a linear (array-based) page table and its location is physical address of 1K (1024)
  - Code: VPN 1 → PFN 4 (4K=4096)
    (VPN 0: VA 0, VPN 1: VA 1K,
    VPN 2: VA 2K, VPN 3: VA 3K, …)
  - Array: VA 40000 ~ 44000 (VPN 39~42)
    and VPN 39~42 → PFN 7~10 (7K~10K)
  - Access sequence:
    1) PT[1] for instruction address
    2) Instruction Fetch (**movl**)
    3) PT[39] for data address
    4) Data Fetch (**mem[%edi+%eax×4]**)
    5) (PT[1] and instruction fetch)×3
    ➔ Total 10 memory access





가상주소

1024 movl $0x0, (%edi, %eax, 4).

① PT[i] VPN

② Instruction fetch

③ PT[39] VPN 39.

④ mem[%edi + 4x%eax] 에 data fetch.

∴ 인스트럭션을 PT[]을 참조해서 실제로 fetch를 해온다.

그럼. 메모리 가상주소를 실제 주소로 매핑해 온다. 그럼 → 실제 주소에 0 을 데이터 fetch.

4번

+3x2번

= 10번.

Summary

- **Paging divides the space into fixed-sized units**
  - To record where each virtual page is placed in physical memory, OS keeps a per-process data structure known as a page table, which maps VPN to PFN
  - OS indexes the array by the VPN, and looks up the PTE at that index to find the corresponding PFN