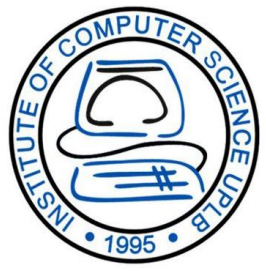
CMSC 125: Operating Systems

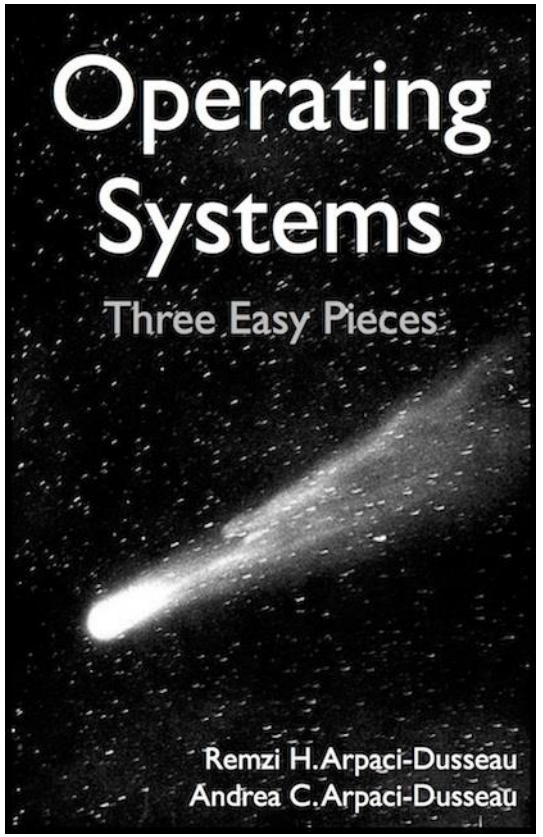
- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org



Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

18. Paging: Introduction

Operating System: Three Easy Pieces

Concept of Paging

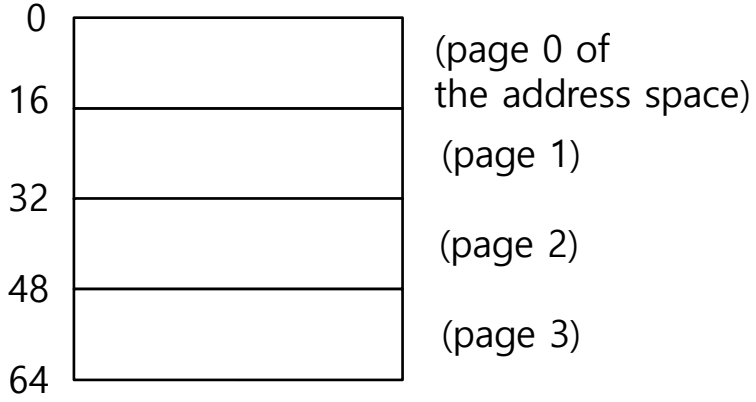
- Split **virtual address space** into **fixed-size** units called a **page**
  - Unlike in segmentation where sizes for each logical segments/sections (code, stack, heap, etc.) are variable
- Split **physical memory** into **fixed-size** units **page frames**
- A **page table** per process is needed **to translate** the **virtual address** to **physical address**

Advantages Of Paging

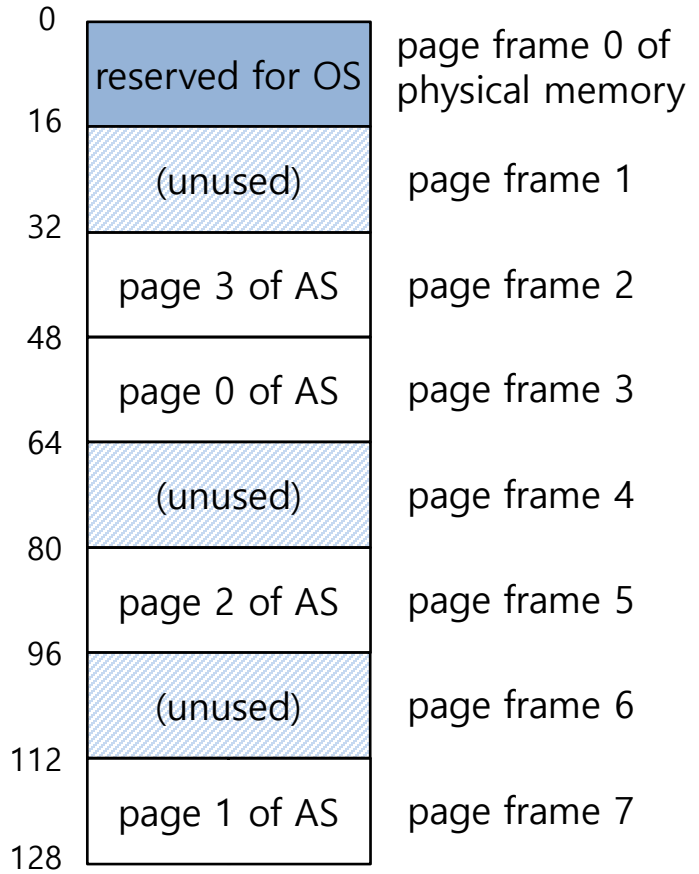
- **Flexibility:** Supporting the abstraction of address space effectively
  - Don’t need assumptions how heap and stack grow and are used
- **Simplicity**: ease of free-space management
  - The page in virtual address space and the page frame in physical memory are the same size
  - Easy to allocate and maintain a free list

Example: A Simple Paging

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



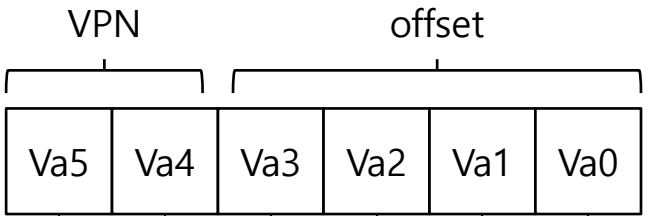
A Simple 64-byte Virtual Address Space(AS)



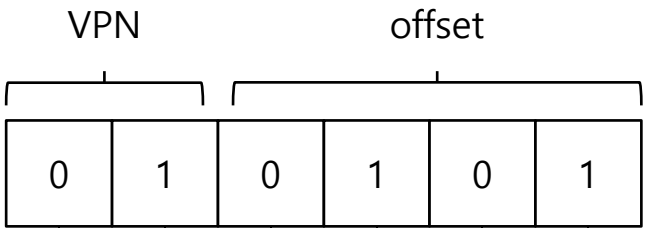
64-Byte Virtual Address Space Placed In Physical Memory

Address Translation

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

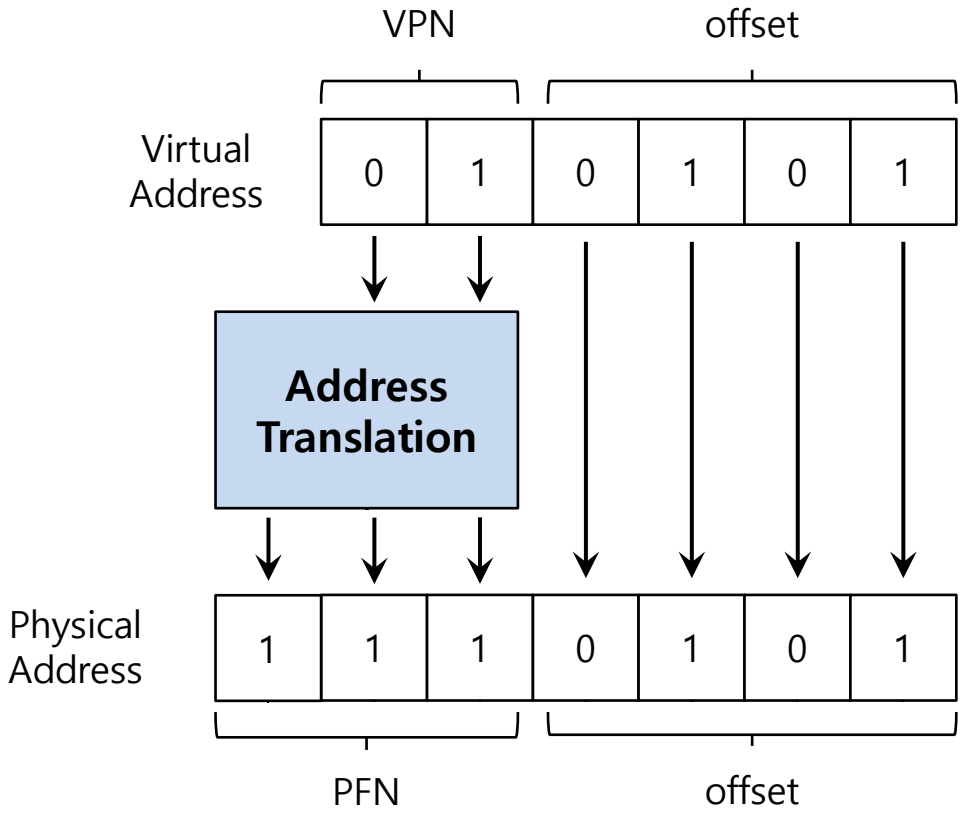


- Example: virtual address 21 in 64-byte address space



Example: Address Translation

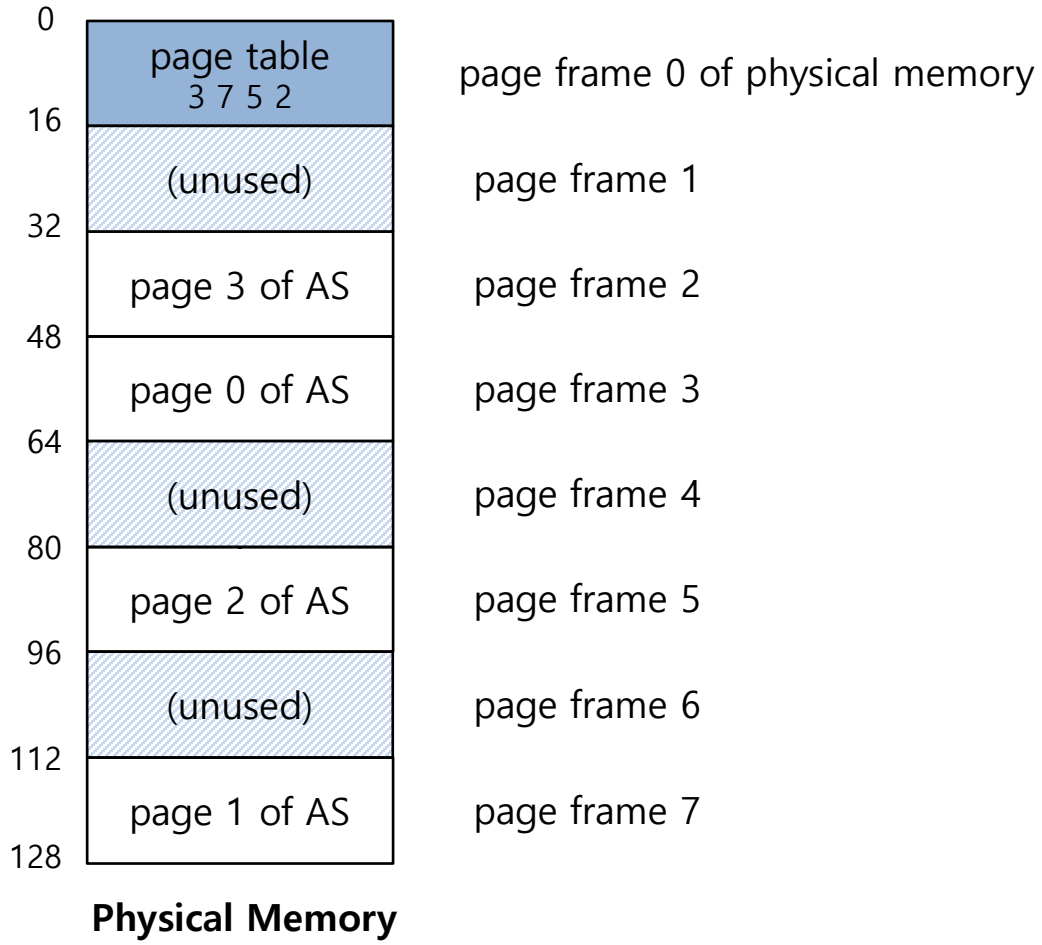
- The virtual address 21 in 64-byte address space



Where Are Page Tables Stored?

- **Pages tables** map the Virtual Page Number (VPN) to Physical/Page Frame Number (PFN)
- 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 tables **for each process** are stored in **memory** (may become very big in size)
  - **Note: One page table per process!**
  - Example: Given a 32-bit address space with 4KB pages
    - 20 bits will be used for VPN and 12 bits for offset
    - Assume four bytes will be used to store an entry in the page table -> 4MB page table per process is needed!

Example: Page Table in Kernel Physical Memory



What Is In The Page Table?

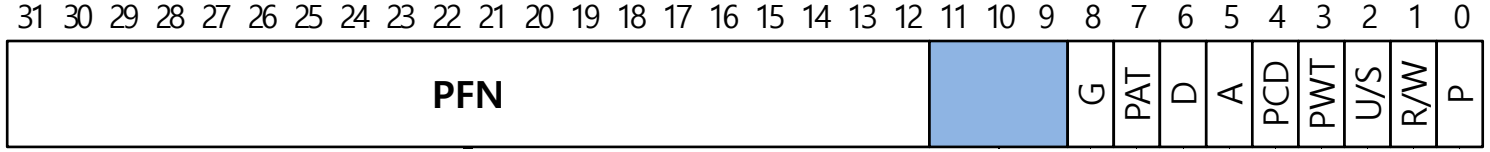
- The page table is just a **data structure** that is used to map the virtual address to physical address
  - Simplest form: a linear page table (an array)

- The OS **indexes** the array(**page table**) by VPN, and looks up the **page table entry (PTE)**

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

Example: x86 Page Table Entry



An x86 Page Table Entry(PTE)

- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

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**
  - Address of the start of page table is stored in a **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)
```

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
```

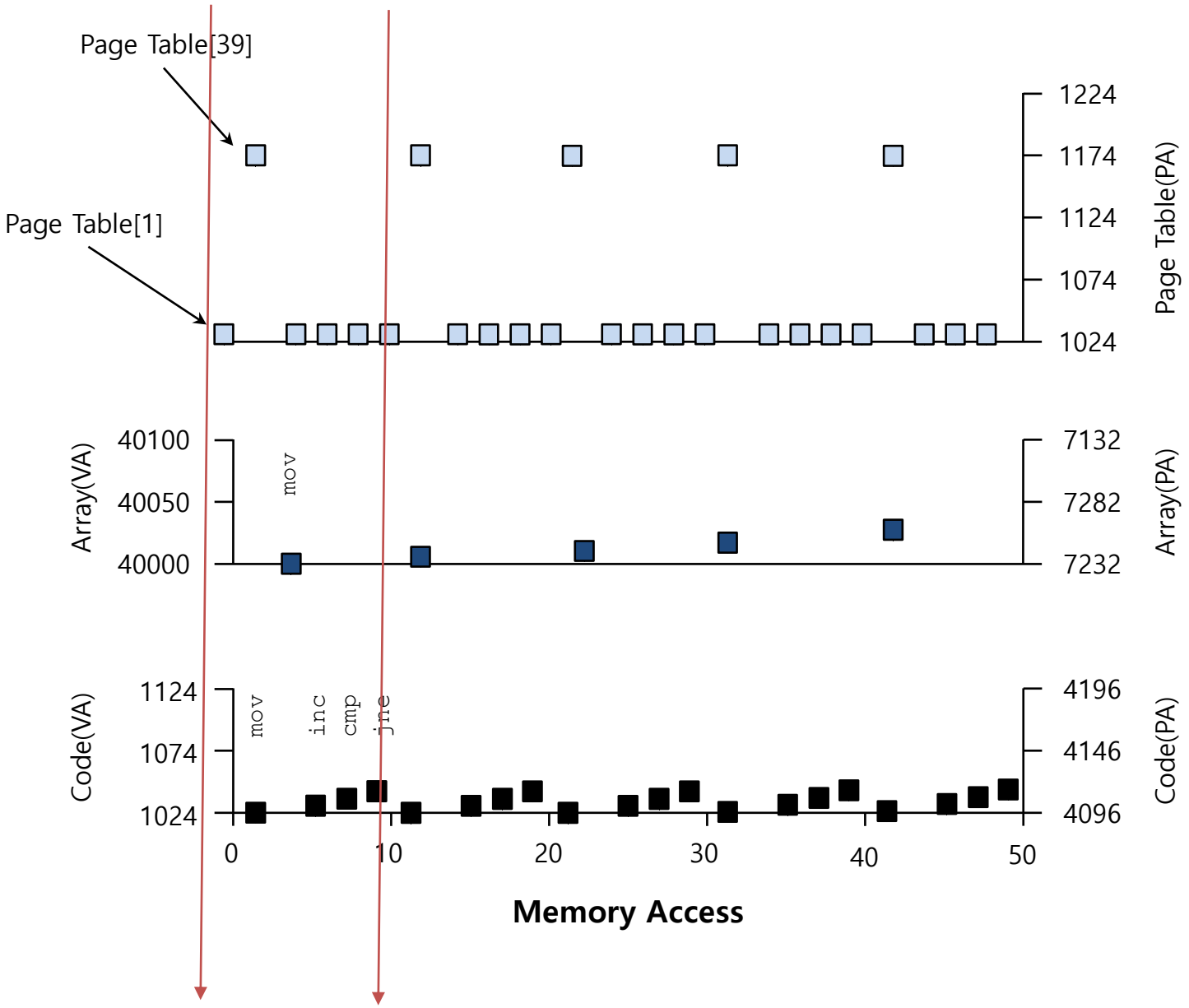
- Resulting Assembly code

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

A Memory Trace: Some Assumptions

- Virtual Address Space Size: 64KB
- Page Size: 1 KB
- Linear Page Table (array-based) located at physical address 1KB(1024)
- Contents of Page Table
  1. Code: Assume VPN 1 -> PFN 4
  2. Array:
     - 4000 bytes (1000*sizeof(int))
     - Assume at virtual address 40000 to 44000
       - VPN 39 -> PFN 7
       - VPN 40 -> PFN 8
       - VPN 41 -> PFN 9
       - VPN 42 -> PFN 10

A Virtual(And Physical) Memory Trace: First 5 iterations



- Each instruction fetch generates two memory references:
  - one to the page table and
  - another to the actual location of the i nstruction
- The mov instruction will also require two memory references:
  - one to the page table and
  - another to the array element itself