# Virtual Memory I

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

# Today's Topics

- **What is virtual memory?**
- **Virtual memory implementation**
  - Paging

# Virtual Memory (1)

- **Example**

```
#include <stdio.h>

int n = 0;

int main ()
{
     printf (“&n = 0x%08x\n”, &n);
}

% ./a.out
&n = 0x08049508
% ./a.out
&n = 0x08049508
```

  - What happens if two users simultaneously run this application?

# Virtual Memory (2)

- **Virtual Memory (VM)**
  - Use virtual addresses for memory references.
    - Large and contiguous
  - CPU performs address translation at run time.
    - From a virtual address to the corresponding physical address
  - Physical memory is dynamically allocated or released on demand.
    - Programs execute without requiring their entire address space to be resident in physical memory.
    - Lazy loading
  - Virtual addresses are private to each process.
    - Each process has its own isolated virtual address space.
    - One process cannot name addresses visible to others.

# Virtual Memory (3)

- **Virtual addresses**
  - To make it easier to manage memory of multiple processes, make processes use virtual addresses (logical addresses)
    - Virtual addresses are independent of the actual physical location of data referenced.
    - OS determines location of data in physical memory
    - Instructions executed by the CPU issue virtual addresses.
    - Virtual addresses are translated by hardware into physical addresses (with help from OS).
    - The set of virtual addresses that can be used by a process comprises its virtual address space.
  - Many ways to translate virtual addresses into physical addresses…

# Virtual Memory (4)

- **Advantages**
  - Separates user's logical memory from physical memory.
    - Abstracts main memory into an extremely large, uniform array of storage.
    - Frees programmers from the concerns of memory-storage limitations.
  - Allows the execution of processes that may not be completely in memory.
    - Programs can be larger than physical memory.
    - More programs could be run at the same time.
    - Less I/O would be needed to load or swap each user program into memory.
  - Allows processes to easily share files and address spaces.
  - Provides an efficient mechanism for protection and process creation.

# Virtual Memory (5)

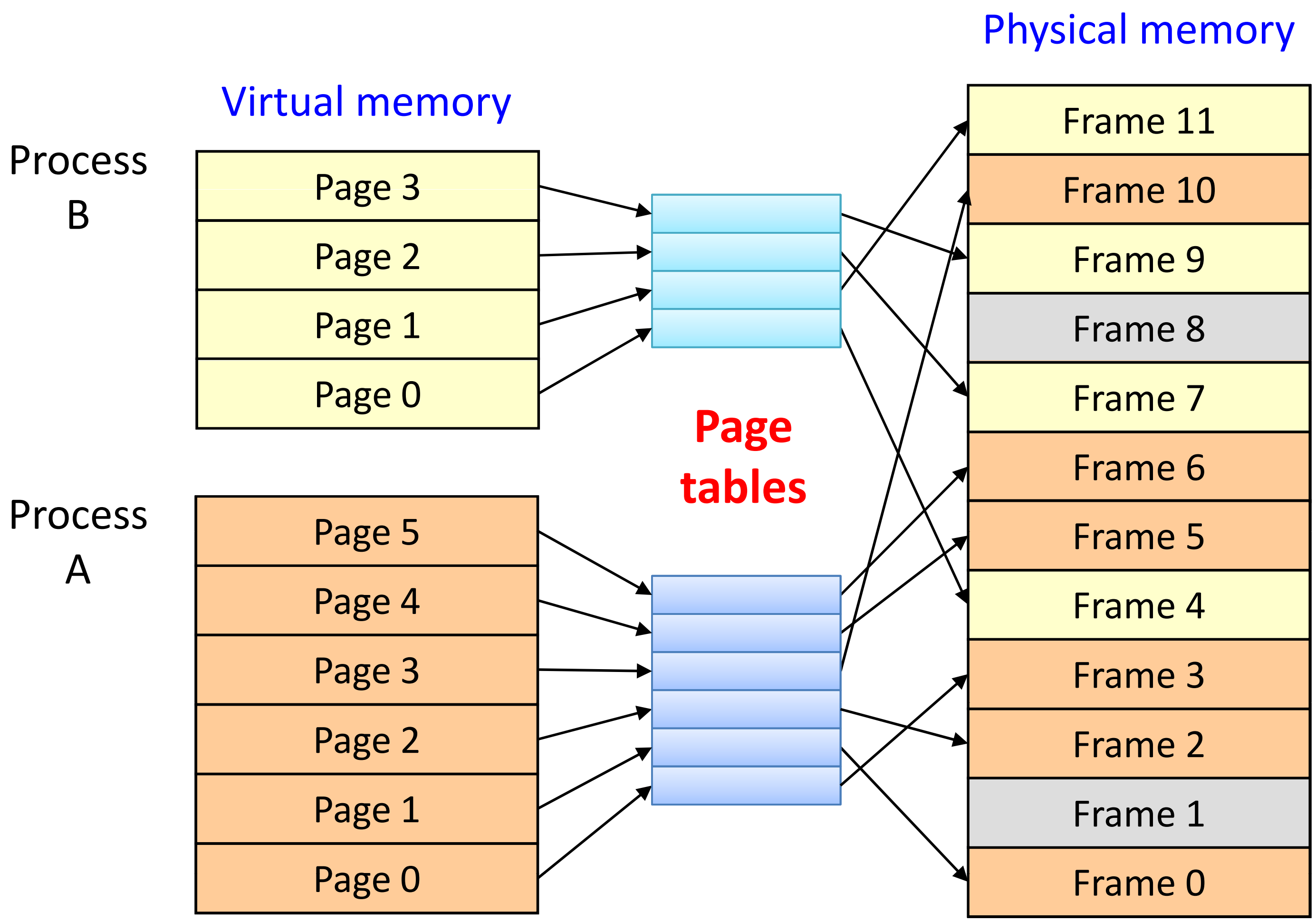
- **Disadvantages**
  - Performance!!!
    - In terms of time and space

- **Implementation**
  - Paging
  - Segmentation

# Paging (1)

- **Paging**
  - Permits the physical address space of a process to be noncontiguous.
  - Divide physical memory into fixed-sized blocks called frames.
  - Divide logical memory into blocks of same size called pages.
    - Page (or frame) size is power of 2 (typically, 512B – 8KB)
  - To run a program of size n pages, need to find n free frames and load program.
  - OS keeps track of all free frames.
  - Set up a page table to translate virtual to physical addresses.

# Paging (2)



Physical memory
Virtual memory
Process B
Page 3
Page 2
Page 1
Page 0
Page tables
Process A
Page 5
Page 4
Page 3
Page 2
Page 1
Page 0
Frame 11
Frame 10
Frame 9
Frame 8
Frame 7
Frame 6
Frame 5
Frame 4
Frame 3
Frame 2
Frame 1
Frame 0

# Paging (3)

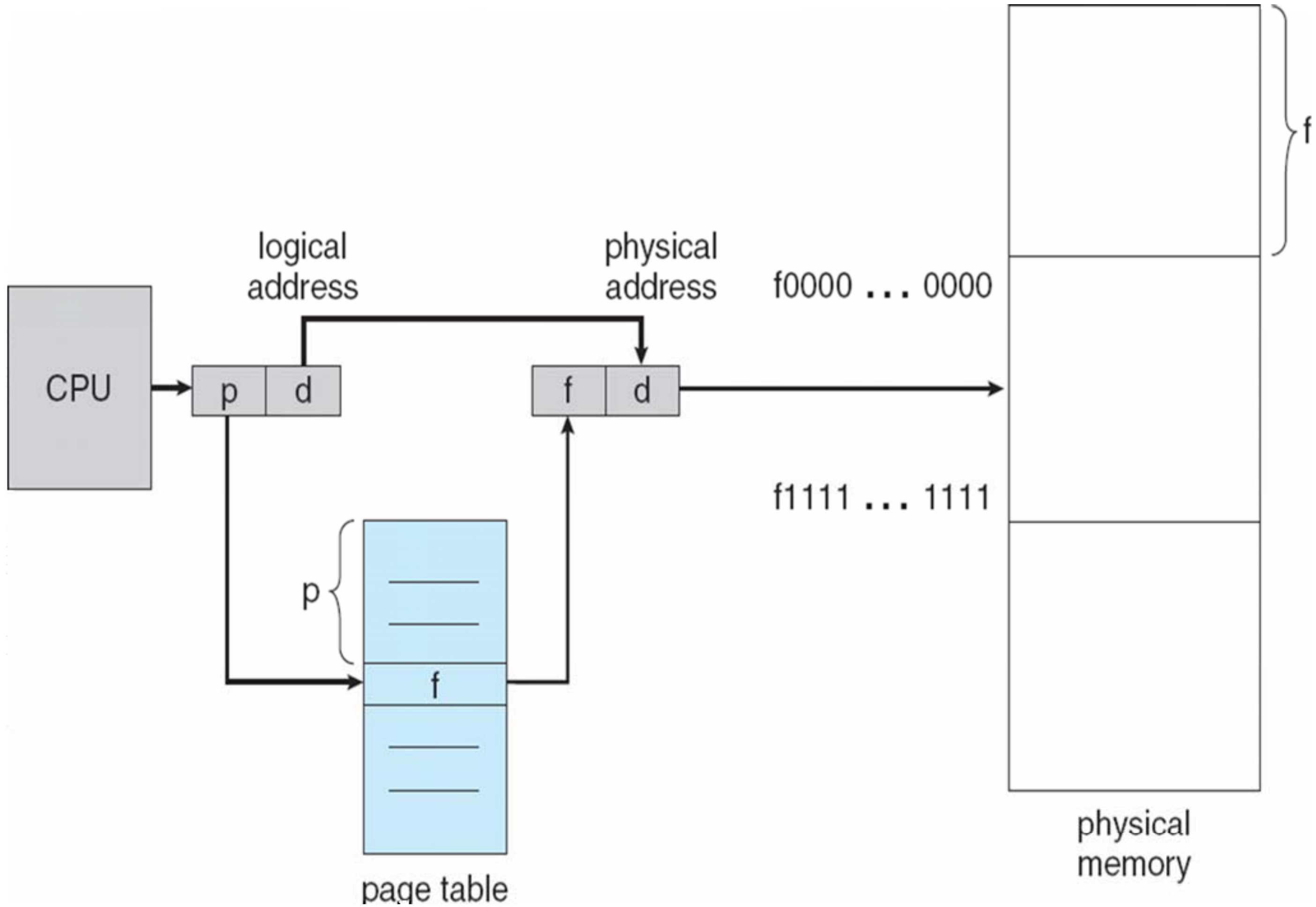
- **User's perspective**
  - Users (and processes) view memory as one contiguous address space from 0 through N.
    - Virtual address space (VAS)
  - In reality, pages are scattered throughout the physical memory.
    - Virtual-to-physical mapping
    - This mapping is invisible to the program.
  - Protection is provided because a program cannot reference memory outside of its VAS.
    - The virtual address 0xdeadcafe maps to different physical addresses for different processes.

# Paging (4)

- **Translating addresses**
  - A virtual address has two parts:
    <virtual page number (VPN)::offset>
  - VPN is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is <PFN::offset>
- **Page tables**
  - Managed by OS
  - Map VPN to PFN
    - VPN is the index into the table that determines PFN
  - One page table entry (PTE) per page in virtual address space, i.e. one PTE per VPN

# Paging (5)

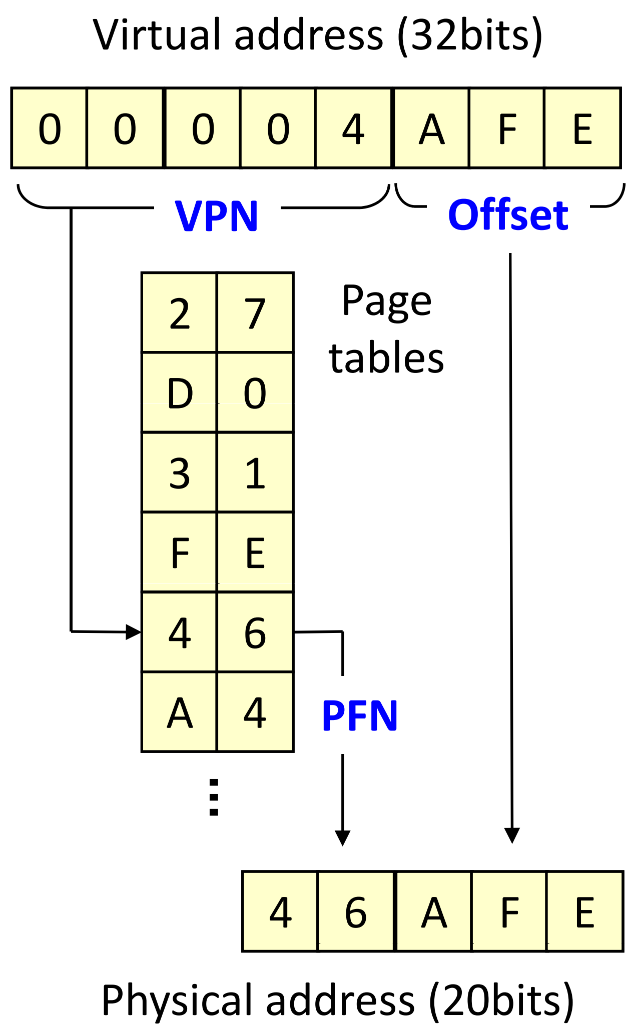
- **Address translation architecture**



# Paging (6)

- **Paging example**
  - Virtual address: 32 bits
  - Physical address: 20 bits
  - Page size: 4KB

  - Offset: 12 bits
  - VPN: 20 bits
  - Page table entries: $2^{20}$



# Paging (7)

- **Protection**
  - Memory protection is implemented by associating protection bit with each frame.
  - Valid / Invalid bit
    - "Valid" indicates that the associated page is in the process' virtual address space, and is thus a legal page.
    - "Invalid" indicates that the page is not in the process' virtual address space.
  - Finer level of protection is possible for valid pages.
    - Provide read-only, read-write, or execute-only protection.

# Paging (8)

- **Page Table Entries (PTEs)**

| 1 | 1 | 1 | 2 | 20 |
|---|---|---|---|---|
| V | R | M | Prot | Page Frame Number (PFN) |

- Valid bit (V) says whether or not the PTE can be used.
  - It is checked each time a virtual address is used.
- Reference bit (R) says whether the page has been accessed.
  - It is set when a read or write to the page occurs.
- Modify bit (M) says whether or not the page is dirty.
  - It is set when a write to the page occurs.
- Protection bits (Prot) control which operations are allowed on the page.
  - Read, Write, Execute, etc.
- Page frame number (PFN) determines physical page.

# Paging (9)

- **Advantages**
  - Easy to allocate physical memory.
    - Physical memory is allocated from free list of frames
    - To allocate a frame, just remove it from its free list
  - No external fragmentation.
  - Easy to "page out" chunks of a program.
    - All chunks are the same size (page size).
    - Use valid bit to detect reference to "paged-out" pages.
    - Pages sizes are usually chosen to be convenient multiple of disk block sizes.
  - Easy to protect pages from illegal accesses.
  - Easy to share pages.

# Paging (10)

## Disadvantages

- Can still have internal fragmentation
  - Process may not use memory in exact multiple of pages.
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution: get a hardware support (TLB)
- Memory required to hold page tables can be large
  - Need one PTE per page in virtual address space
  - 32-bit address space with 4KB pages = $2^{20}$ PTEs
  - 4 bytes/PTE = 4MB per page table
  - OS's typically have separate page tables per process (25 processes = 100MB of page tables)
  - Solution: page the page tables, multi-level page tables, inverted page tables, etc.

# Demand Paging (1)

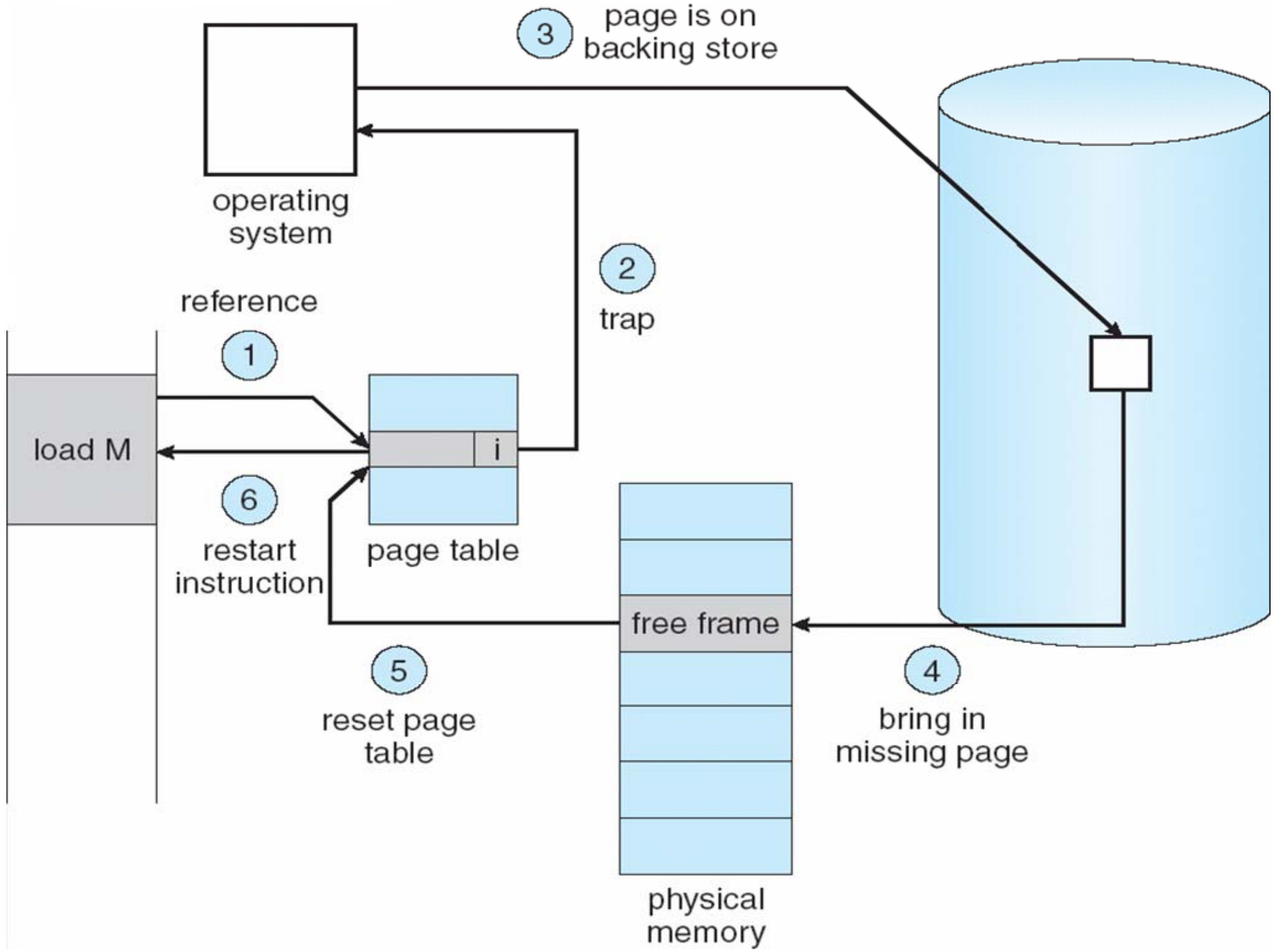
- **Demand paging**
  - Bring a page into memory only when it is needed.
    - Less I/O needed
    - Less memory needed
    - Faster response
    - More users
  - OS uses main memory as a (page) cache of all of the data allocated by processes in the system.
    - Initially, pages are allocated from physical memory frames.
    - When physical memory fills up, allocating a page requires some other page to be evicted from its physical memory frame.
  - Evicted pages go to disk (only need to write if they are dirty)
    - To a swap file
    - Movement of pages between memory/disks is done by the OS
    - Transparent to the application

# Demand Paging (2)

- **Page faults**
  - What happens to a process that references a virtual address in a page that has been evicted?
    - When the page was evicted, the OS sets the PTE as invalid and stores (in PTE) the location of the page in the swap file.
    - When a process accesses the page, the invalid PTE will cause an exception to be thrown.
  - The OS will run the page fault handler in response.
    - Handler uses invalid PTE to locate page in swap file.
    - Handler reads page into a physical frame, updates PTE to point to it and to be valid.
    - Handler restarts the faulted process.
  - Where does the page that's read in go?
    - Have to evict something else (page replacement algorithm)
    - OS typically tries to keep a pool of free pages around so that allocations don't inevitably cause evictions.

# Demand Paging (3)

- **Handling a page fault**



# Demand Paging (4)

- **Why does this work?**
  - Locality
    - Temporal locality: locations referenced recently tend to be referenced again soon.
    - Spatial locality: locations near recently referenced locations are likely to be referenced soon.
  - Locality means paging can be infrequent.
    - Once you've paged something in, it will be used many times.
    - On average, you use things that are paged in.
    - But this depends on many things:
      - Degree of locality in application
      - Page replacement policy
      - Amount of physical memory
      - Application's reference pattern and memory footprint

# Demand Paging (5)

- **Why is this "demand" paging?**
  - When a process first starts up, it has a brand new page table, with all PTE valid bits "false".
    - No pages are yet mapped to physical memory.
  - When the process starts executing:
    - Instructions immediately fault on both code and data pages.
    - Faults stop when all necessary code/data pages are in memory.
    - Only the code/data that is needed (demanded!!) by process needs to be loaded.
    - What is needed changes over time, of course…