Soongsil Univ.
http://design.ssu.ac.kr

Digital System Design Lab.

Ch. 10 Virtual Memory

Chanho Lee
Soongsil University

1

Objectives

- Virtual memory and benefits.
- Page loading on demand
- Page-replacement algorithms.
- Working set of a process and program locality.
- Virtual memory in Linux, Windows 10

2

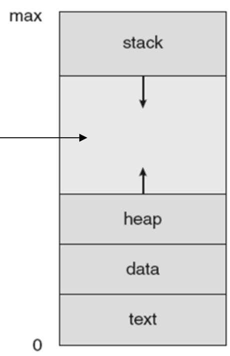
10.1 Background

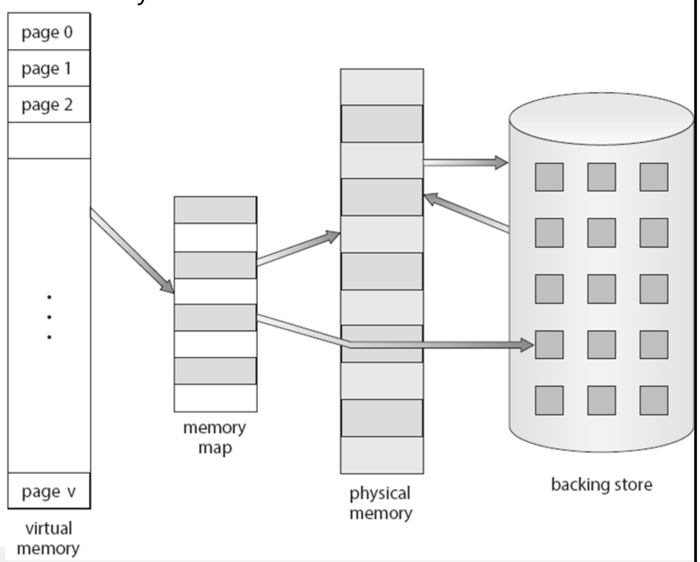
- The instructions being executed must be in physical memory
  - ►limits the size of a program to the size of physical me
  - ►the entire program is not needed in many cases
    - code to handle unusual error conditions: rare error occurrence → almost never executed
    - Arrays, lists, and tables: often allocated more for the worst case.
    - Certain options and features of a program may be used rarely.
    - Frequently used functions are not used at the same time
  - ►Partial loading of a program to memory
    - Programming an extremely large virtual address space → simplifying the programming task
    - Using less physical memory → increase the degree of multiprogramming, CPU utilization and throughput
    - Less I/O for loading or swapping → programs runs faster

10.1 Background

- Virtual memory
  - ►Separation of logical memory from physical memory.
  - ►Virtual address space: logical view of a process in contiguous memory
  - ►A process in physical page frames: not contiguous

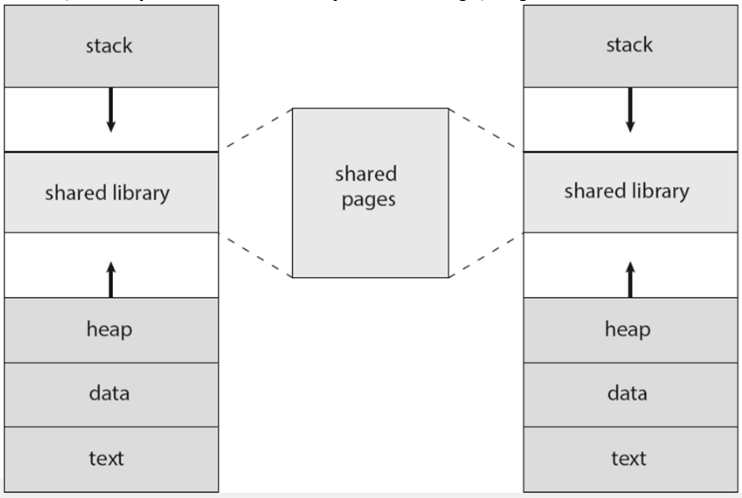
Actual physical pages are required only if the heap or stack grows





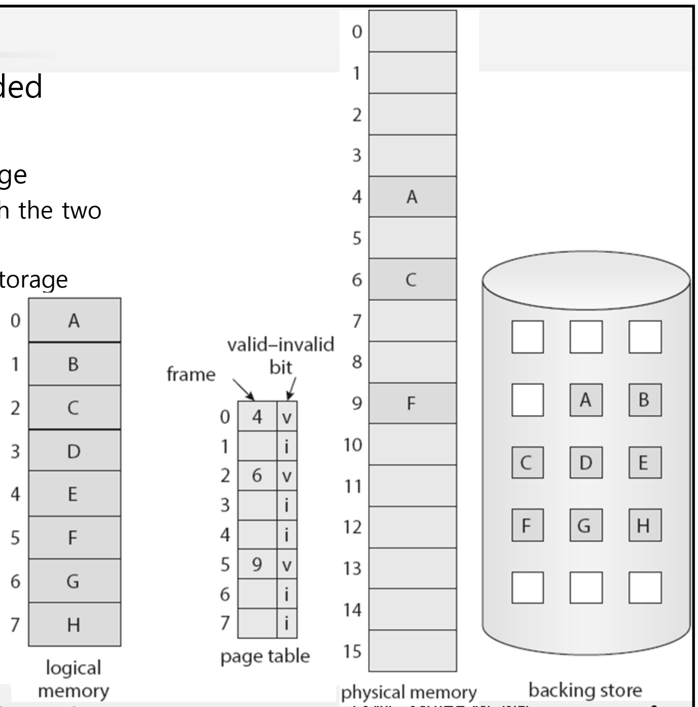
10.1 Background

- Sparse address spaces: virtual address spaces that include holes
  - ▶ beneficial because
    - the holes can be filled as the stack or heap segments grow or
    - if we wish to dynamically link libraries (or possibly other shared objects) during program execution
- Page sharing
  - ▶ files and memory to be shared
    - system libraries
    - sharing memory (IPC)
  - ▶ Part of virtual address space, but in the shared physical memory



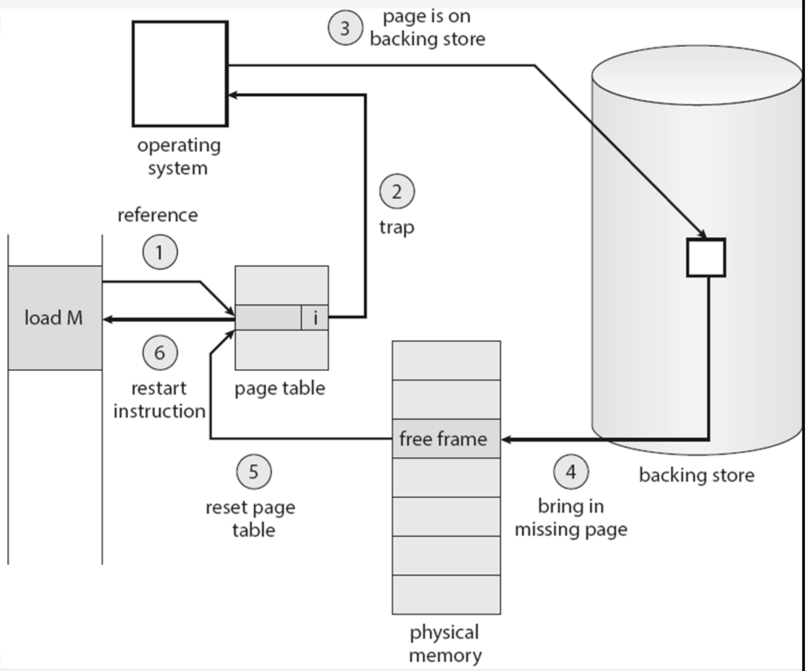
10.2 Demand Paging

- Pages are loaded only when demanded
- Basic Concepts
  - ▶ Pages in memory and in secondary storage
    - Need some hardware support to distinguish the two
    - Valid: both legal and in memory
    - Invalid: either is not valid or in secondary storage
      - – no effect until accessing the page
      - – Access results in a page fault



10.2 Demand Paging

- ► ② Handling page fault
  - check valid or invalid memory access using an internal table
    - – Invalid: termination
    - – Valid: page-in ③



10.2 Demand Paging

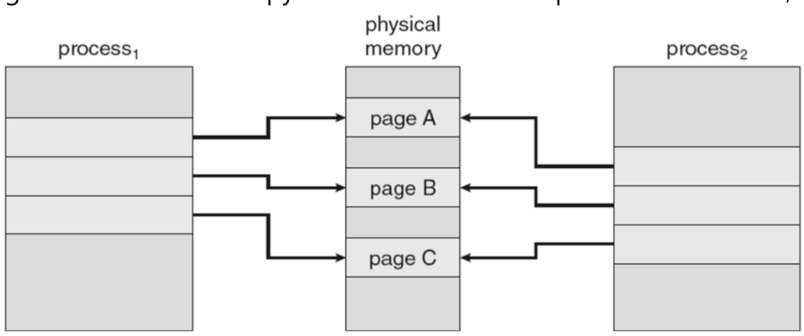
- Pure demand paging
  - ► Process start with no pages in memory → page faults on the first instruction
    - one or more page faults for instruction and data
    - locality of reference
- Hardware to support demand paging
  - ► Page table and secondary memory
  - ► Ability to restart any instruction after a page fault
- Free-Frame List
  - ► a pool of free frames   head → 7 → 97 → 15 → 126 ··· → 75
  - ► zero-fill-on-demand: "zeroed-out" before being allocated
- Performance of Demand Paging
  - ► effective access time = (1 − p) × ma + p × page fault time
    - ma: memory-access time (200ns)
    - p: probability of a page fault ($0 \le p \le 1$)

10.2 Demand Paging

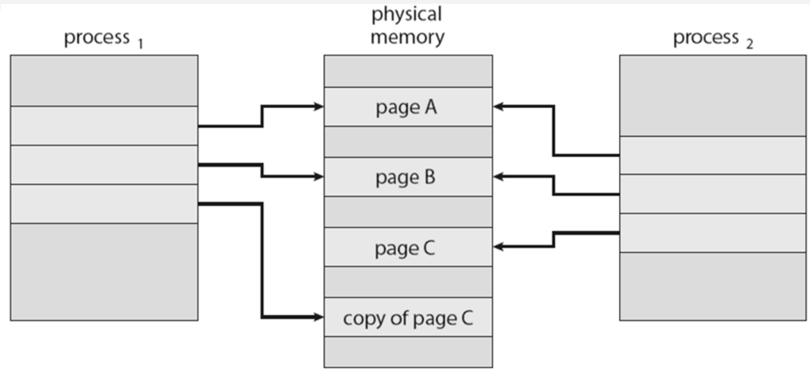
- ►page fault resolving sequence (handler): refer to the textbook for the sequences in detail
  - 1. Service the page-fault interrupt (1 ~ 100us)
  - 2. Read in the page (~8ms for HDD)
  - 3. Restart the process (1 ~ 100us)
- ►effective access time = (1 − p) × (200) + p (8 ms)
  = (1 − p) × 200 + p × 8,000,000
  = 200 + 7,999,800 × p.
  - directly proportional to the page-fault rate
  - If p=0.001, 8.2us → performance degradation by a factor of 40
  - <10% degradation → p < 0.0000025
- ►Swap space (generally faster than the file system)
  - copying an entire file image into the swap space at process startup
  - demand-page from the file system initially but write the pages to swap space as they are replaced
    - Windows, Linux

10.3 Copy-on-Write

- fork() system call: page sharing → bypass demand paging
  - ►rapid process creation and minimizes the number of page allocation
  - ►fork()-exec(): copying of the parent's address space may be unnecessary
- copy-on-write:
  - ►parent and child processes initially to share the same pages.
  - ►shared pages are marked as copy-on-write → if either process writes to it, create a copy



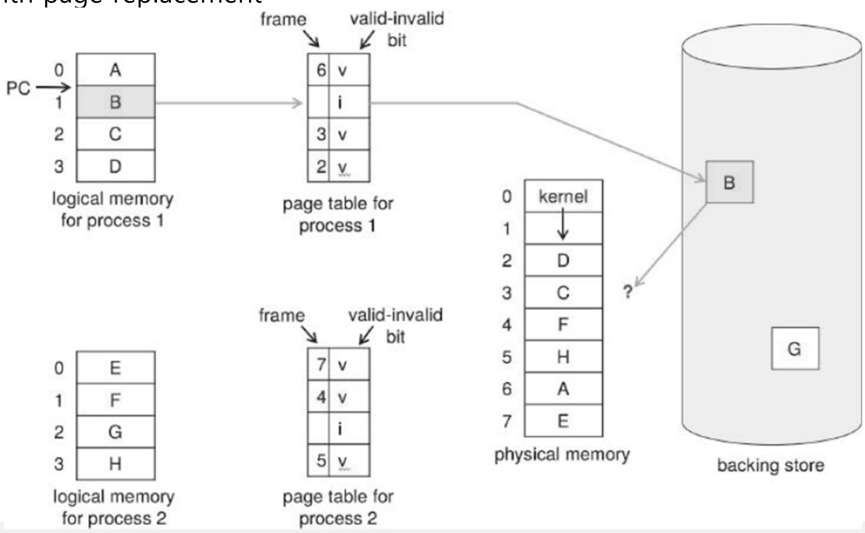
10.3 Copy-on-Write



► only the pages that are modified by either process are copied
► Windows, Linux, and macOS

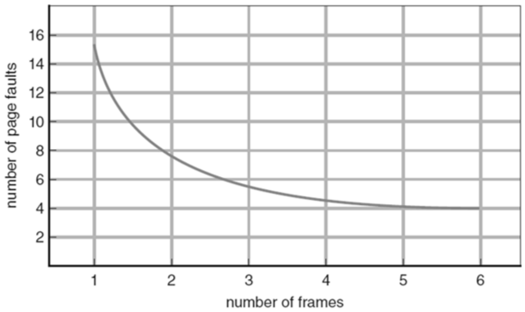
10.4 Page Replacement

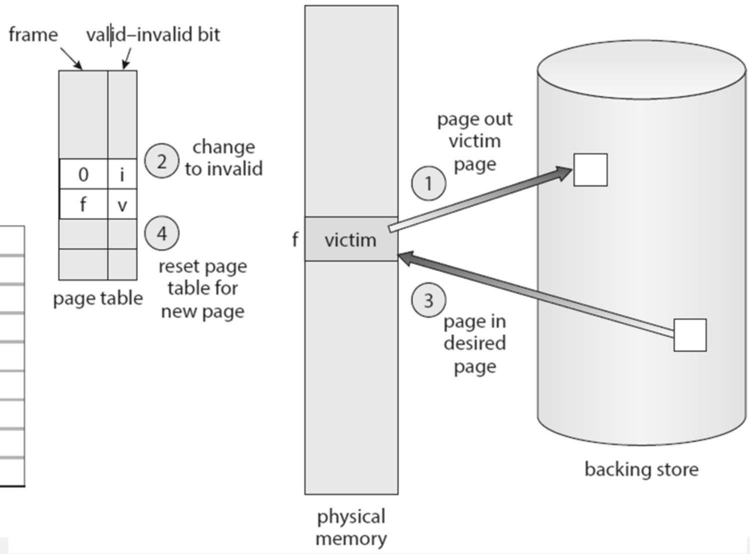
► demand paging → increase degree of multiprogramming → over-allocating memory
► System memory: program pages + I/O buffer → out of free frames
► swapping pages with page replacement



10.4 Page Replacement

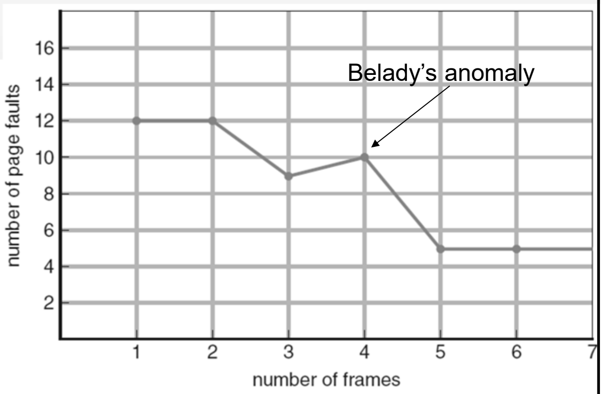
- Basic Page Replacement
  - Replacement: requires two page transfers
  - modify bit (or dirty bit)
    - Not set: no page out
  - frame-allocation algorithm
    - No. of frames to allocate to process
  - page-replacement algorithm
    - select the frames to be replaced





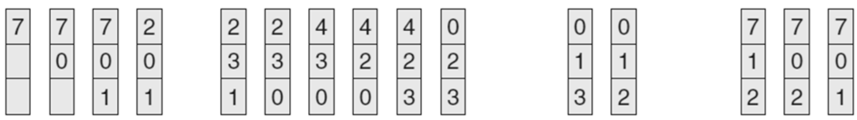
10.4 Page Replacement

- FIFO Page Replacement
  - replace the page at the head of the queue



reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

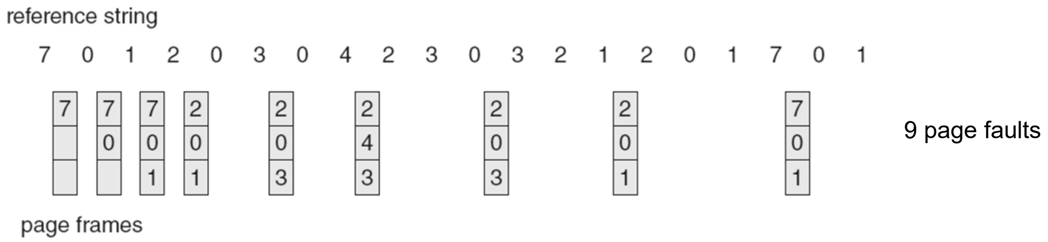


15 page faults

page frames

10.4 Page Replacement

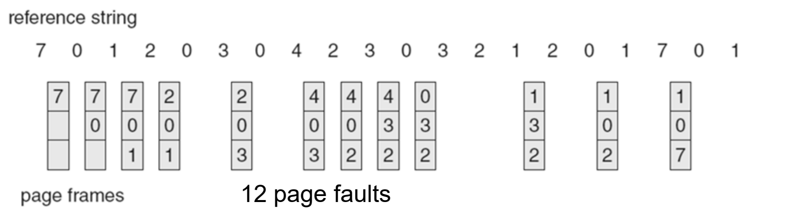
- Optimal Page Replacement
  - ►To overcome Belady's anomaly
  - ►Replace the page not be used for the longest time
  - ►the lowest possible page fault rate for a fixed number of frames

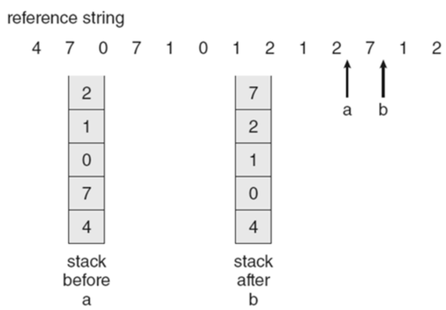


  - ►difficult to implement: it requires future knowledge of the reference string

10.4 Page Replacement

- Least recently used (LRU) Page Replacement
  - ►approximation of the optimal algorithm
  - ►use the recent past as an approximation of the near future
    → the page that has not been used for the longest time





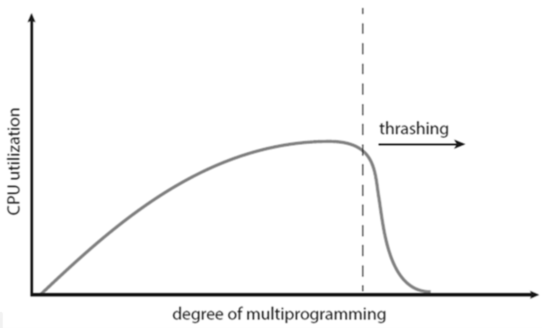
  - ►Implementing may require substantial hardware assistance
    - Counters: store the "time" of the last reference to each page → a page with the smallest is replaced
    - Stack: the most recently used page at the top of the stack → a page at the bottom is replaced

10.5 Allocation of Frames

- Example: simple strategy
  - 128 frames, 35 for kernel, 93 for user, pure demanding page: allocation?
  - 93 page faults, page replacements, termination, 93 free frames
- Minimum Number of Frames
  - No. of frames allocated ↓ → page-fault rate ↑ → performance ↓
  - Allocate frames to hold all the pages that any single instruction can reference
    - defined by the processor architecture
- Allocation Algorithms
  - Equal allocation: m/n frames (split m frames among n processes)
  - Proportional allocation: allocate available memory to each process according to its size.
    - Priority can be combined
- Global versus Local Allocation
  - From the set of all frames
  - From its own set of allocated frames

10.6 Thrashing

- Thrashing
  - High paging activity: spending more time paging than executing
    → severe performance problems
  - Ex. replace a page that will be needed again right away → quickly faults again, and again
- Cause of Thrashing
  - Low CPU utilization → increase the degree of multiprogramming by introducing a new process
  - With a global page-replacement algorithm, new process takes frames → more processes fault
    → waiting for paging device → ready queue empties → low CPU utilization
  - Local replacement algorithm
    (or priority replacement algorithm)
    - limits the effects of thrashing
    - Thrashing process is mainly in the waiting queue
    - The effective access time for paging device
      will increase



10.7 Memory Compression

- Compress several frames into a single frame, enabling the system to reduce memory usage → No swapping
  - ▶Before compression

free-frame list

head → 7 → 2 → 9 → 21 → 27 → 16

modified frame list

head → 15 → 3 → 35 → 26

  - ▶After compression

free-frame list

head → 2 → 9 → 21 → 27 → 16 → 15 → 3 → 35

modified frame list

head → 26

compressed frame list

head → 7

10.9 Other Considerations

- Prepaging
  - ▶to bring some—or all—of the pages that will be needed at one time ↔ pure demand paging
  - ▶to prevent this high level of initial paging
- Page Size
  - ▶Memory is better utilized with smaller pages ↔ increases the size of the page table (per process)
    - Better resolution for locality:
  - ▶Time for read/write pages
    - HDD: seek and latency times are much larger than transfer time (larger page size is desirable)
    - SSD: no seek time and much smaller latency. Different read and write time.
  - ▶TLB Reach (related to hit ratio)
    - The number of entries multiplied by the page size: memory size accessible from the TLB
    - Larger page size is desirable

10.9 Other Considerations

- Program Structure
  - Page size: 128 words
  - Row major: 128 x 128 page faults

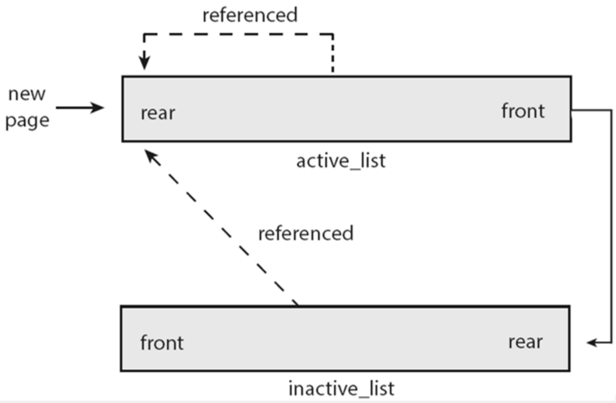
```
int i, j;
int[128][128] data;
for (j = 0; j < 128; j++)
  for (i = 0; i < 128; i++)
    data[i][j] = 0; // zeros one word in each page
```

  - Column major: 128 page faults

```
int i, j;
int[128][128] data;
for (i = 0; j < 128; j++)
  for (j = 0; i < 128; i++)
    data[i][j] = 0; // zeros all the word in a page
```


10.10 Operating-System Examples

- Linux
  - demand paging, allocating pages from a list of free frames
  - global page-replacement policy similar to the LRU-approximation clock algorithm
  - page lists: an active list and an inactive list
    - inactive list: contains pages that have not recently been referenced and are eligible to be reclaimed
    - Each page has an accessed bit: set whenever the page is referenced
  - page-out daemon process `kswapd`
    - periodically checks the amount of free memory
    - Free memory falls below a certain threshold → scanning pages in the inactive list and reclaiming them for the free list



10.10 Operating-System Examples

- Windows
  - ►Windows 10 supports 32- and 64-bit systems running on Intel (IA-32 and x86-64) and ARM
  - ►32-bit systems
    - the default virtual address space: 2 GB,
    - can be extended to 3 GB.
    - 4 GB of physical memory
  - ►64-bit systems:
    - 128-TB virtual address space
    - 24 TB of physical memory (Server: 128 TB)
  - ►shared libraries, demand paging, copy-on-write, paging, and memory compression
  - ►demand paging with clustering
    - Bringing faulting page + several pages immediately preceding and following the faulting page
    - Data page: cluster=3; others: cluster=7

Summary

- Virtual memory
  - ►Definition and benefits
- Demand paging
  - ►A page fault
- Copy-on-write
- page-replacement algorithm
  - ►FIFO, optimal, and LRU. Pure LRU, LRU-approximation algorithms.
  - ►Global page-replacement algorithms vs. local page-replacement algorithms
- Thrashing
- Memory compression
- Other Considerations
  - ►Prepaging, page size, TLB reach, Program structure
- OS examples: Linux, Windows

Exercises, problems and projects

- Exercises
  - ►10.1, 10.5, 10.7

- Problems
  - ►10.16

10.16 A simplified view of thread states is ready, running, and blocked, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O). Assuming a thread is in the running state, answer the following questions, and explain your answers:
a. Will the thread change state if it incurs a page fault? If so, to what state will it change?
b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?
c. Will the thread change state if an address reference is resolved in the TLB? If so, to what state will it change?