Chapter 9: Virtual Memory

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Objectives/Outline

Objectives

- Describe the benefits of a virtual memory system
- Explain the concepts of demand paging, page replacement algorithms, and allocation of page frames

Outline

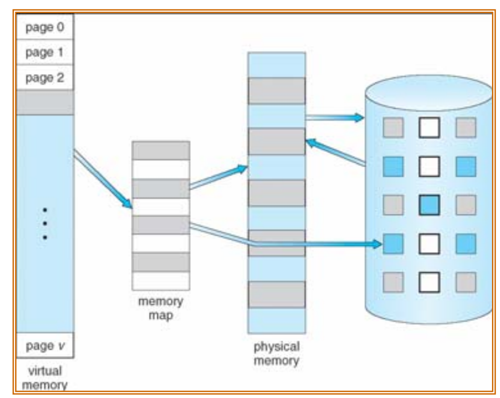
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Demand Segmentation
- Operating System Examples

Background

- All memory management strategies have the same goal to keep many processes in memory simultaneously to allow multiprogramming and hence increase the CPU utilization
- Memory management is required because instructions must be in physical memory to be executed
  - Put the entire process in physical memory (problem: limited memory)
  - Dynamic loading can ease this restriction (requires extra work by the programmer)
- Virtual memory is a technique that allows the execution of processes that are not completely in memory
  - Creates illusion of a "virtual" memory that can be larger than real memory but still nearly as fast

Background (cont.)

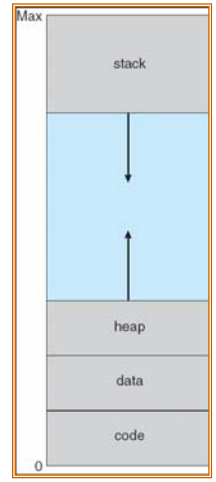
- Virtual memory – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
    - Potentially as big as a disk, but normally constrained by the address size [2^{32}=4GB]



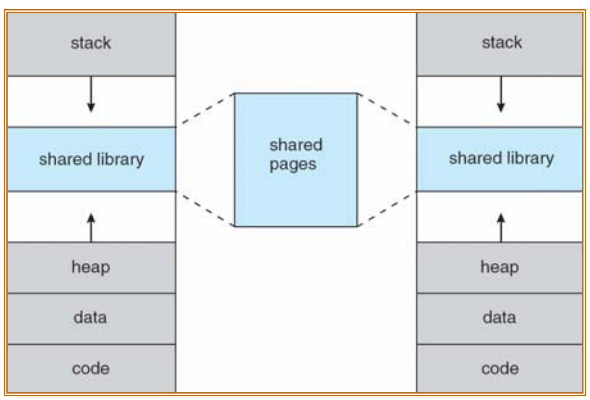
Background (cont.)

- Advantages:
  - Processes can be larger can physical memory in the system
  - Programmers need not worry about the memory storage limitations
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
    Allows processes to share files
- Disadvantages
  - Not easy to implement
  - May substantially decrease the performance if it is used carelessly

Virtual-address Space



Shared Library Using Virtual Memory

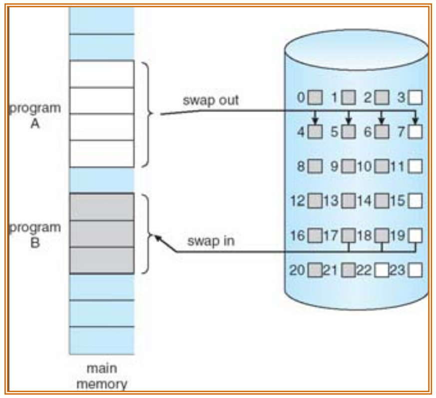


Background (cont.)

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Demand Paging

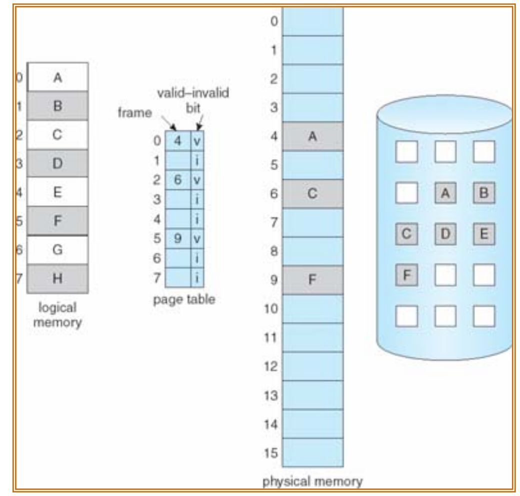
- Bring a page into memory only when it is needed during program execution
  - Less physical memory needed
  - Reduce the swap time and thus has faster response
  - Accommodate more users
- A swapper manipulates the entire process, whereas a pager (a lazy swapper) manipulates just individual pages associated with a process
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory



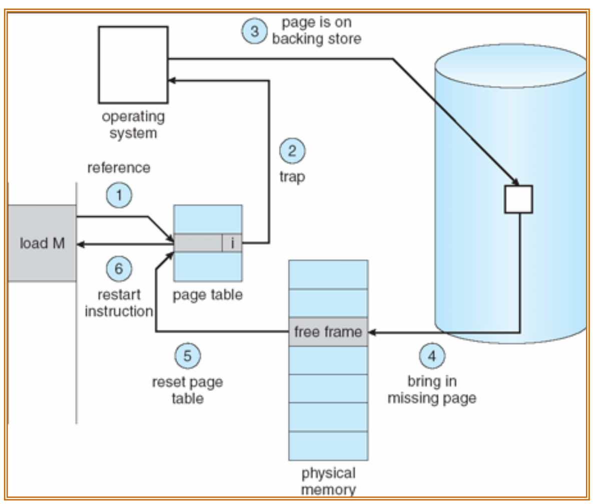
Transfer of a Paged Memory to Contiguous Disk Space

Page Table When Some Pages Are Not in Main Memory

Each page table entry has a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)



Steps in Handling a Page Fault



Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system
- Performance metrics:
  - Page Fault Rate (p),
    - $0 \le p \le 1.0$
    - if $p = 0$ no page faults
    - if $p = 1$, every reference is a fault
  - Effective Access Time (EAT)

    $$EAT = (1 - p) \times \text{memory access} + p \times \text{page fault time}$$

Example

- Memory access time = 100 nanoseconds
- Average page-fault service = 25 microseconds
- Then:

  EAT = (1 – p) x 100 + p x 25,000,000
  = 100 + 24,999,900 p

Activity: What should be the value for *p* if we want the performance degradation to be less than 10%?

Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

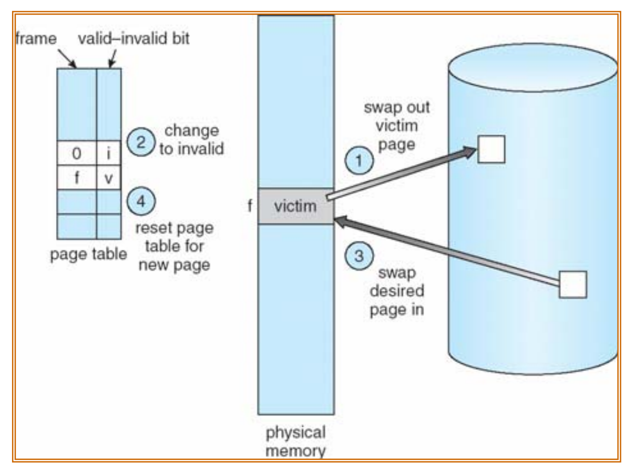
  If either process modifies a shared page, only then is the page copied

- COW allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a *pool* of zeroed-out pages

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use *modify* (*dirty*) *bit* to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

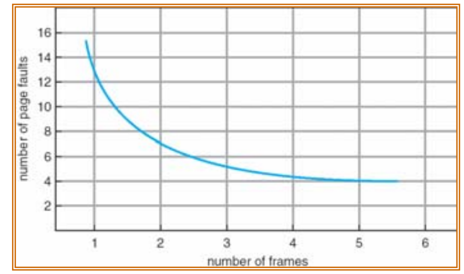


Basic Page Replacement (cont.)

1. Find the location of the desired page on disk
2. Find a free frame:
   1. If there is a free frame, use it
   2. If there is no free frame, use a page replacement algorithm to select a victim frame; write the victim frame to the disk, change the page and frame tables accordingly
3. Read the desired page into the (newly) free frame; Update the page and frame tables
4. Restart the user process

Page Replacement Algorithms

- Many page replacement algorithms exist
- Want *lowest* page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - Reference strings are either generated randomly or using by tracing a given system
- As the number of frames available to the process increases, the number of page faults decreases



Page Replacement Algorithms (cont.)

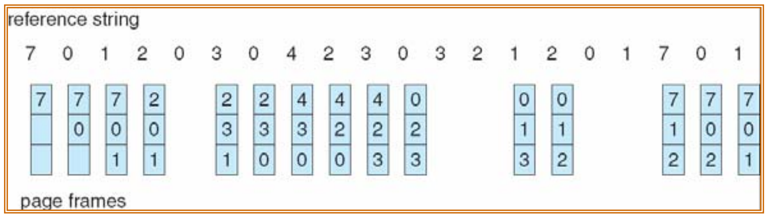
- Replacement algorithms
  - FIFO Page Replacement (the simplest)
  - Optimal Page Replacement (OPT or MIN)
  - Least Recently Used (LRU) Page Replacement
  - LRU Approximation Page Replacement
  - Counting algorithms (not commonly used)
    - Least frequently used (LFU) algorithm
    - Most frequently used (MFU) algorithm

FIFO Algorithm

- Replaces the oldest page in the memory
- Easy to understand and program
- Performance is not always good
- Suffer from Belady's anomaly

FIFO Algorithm (cont.)

- Given reference string
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- Number of frames is 3
- Allocation and replacement



Num of page faults = 15

Another Example

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

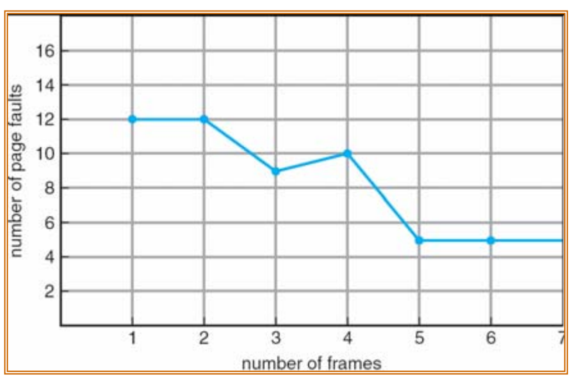
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

Another Example (cont.)

- FIFO Replacement suffers from Belady's Anomaly
  - more frames does not guarantee less page faults!!!

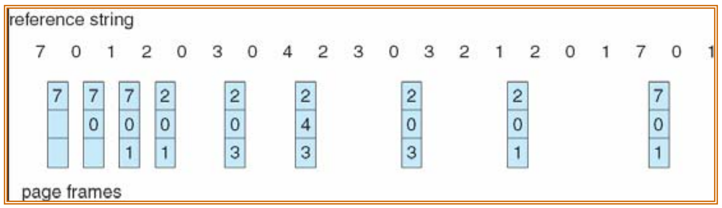


Optimal Algorithm

- Replace the page that will not be used for the longest period of time
- Has the lowest page fault rate
- Never suffer from Belady's anomaly
- Difficult to implement
  - it requires future knowledge of the reference string
- Used for measuring how well other algorithms perform (benchmark)

Optimal Algorithm (cont.)

- Given reference string
  - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- Number of frames is 3
- Allocation and replacement

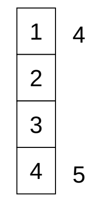


Num of page faults = 9

Another Example

- 4 frames example

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



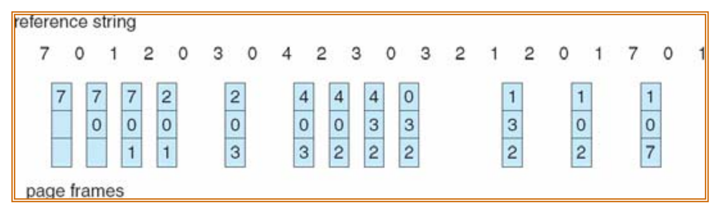
6 page faults

Least Recently Used (LRU) Algorithm

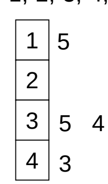
- FIFO uses the past (looks backward)
- OPT uses the future (looks forward)
- LRU uses the recent past as an approximation of the near future
  - Replace the page that has NOT been used for the longest period of time
- Counter implementation
  - Every page entry has a counter; every time the page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Algorithm (cont.)

- Example

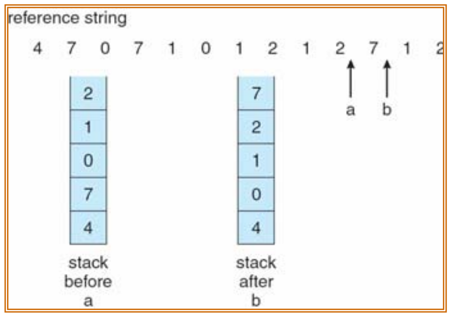


- Another example:
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



LRU Algorithm (Cont.)

- Stack implementation: keep a stack of page numbers in a double link form. If a page is referenced, then move it to the top. At worst, we require 6 pointers to be changed
- No search for replacement and does not suffer from Belady's anomaly
- Requires hardware support

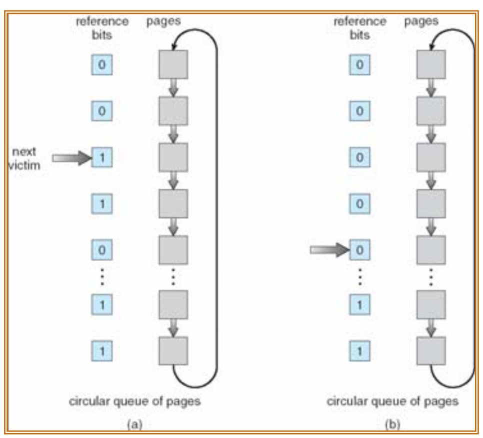


LRU Approximation Algorithms

- Some systems provide no hardware support for LRU
- Many system provide some help in the form of a reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however.
- Record reference bits at regular interval can provide ordering information
- Second-chance algorithm
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Second-Chance (Clock) Algorithm

- A pointer indicates which page needs to be replaced next
- When a frame is needed, the pointer is advanced until it finds a page with reference bit = 0
- As the pointer advances, it clears the reference bits



Enhanced second-chance algorithm

- uses the reference bit and the modify bit as an ordered pair (R, M)
- There are four possible classes for each page
  - (0, 0) neither recently used nor modified – the best page to replace
  - (0, 1) not recently used but modified – not quite as good; as it needs to be written out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and needs to be written out before replacement
- Page replacement use a similar algorithm as the clock algorithm but by considering the page class
  - Replace the first page encountered in the lowest nonempty class

Counting Algorithms

- Keep a counter of the number of references that have been made to each page:
  - Least Frequently Used (LFU) Algorithm:
    - Replaces page with the smallest count
    - An actively used page should have a large reference number
  - Most Frequently Used (MFU) Algorithm
    - Replaces page with the highest count
    - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Not commonly used: expensive implementation + not approximating OPT

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle move instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Major allocation schemes
  - Fixed allocation (equal, proportional)
  - Priority allocation

Fixed Allocation

- Equal allocation: e.g., if 100 frames and 5 processes, give each 20 pages
- Proportional allocation: Allocate according to size of process

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64, \quad s_i = 10, \quad s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

- Both depend on the degree of multiprogramming

Priority Allocation

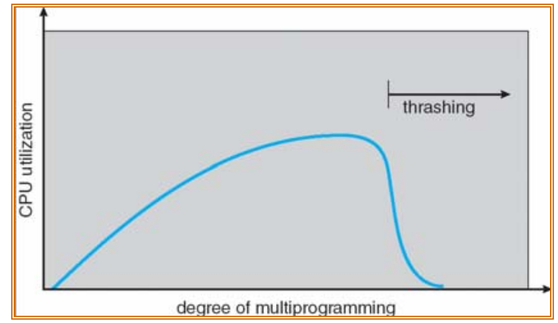
- Use a proportional allocation scheme using priorities rather than size
  - A high priority process is given more frames to speed its execution
- If process P_i generates a page fault
  - Select for replacement one of its frames
  - Select for replacement a frame from a process with lower priority number

Local vs. Global Allocation

- **Local** replacement:
  - Each process selects from only its own set of allocated frames
  - The number of frames allocated to each process does not change
  - Can hinder a process by not making available to it a less used page
- **Global** replacement:
  - Process selects a replacement frame from the set of all frames
  - One process can take a frame from another
  - A process can increase its frames on the expense of other unfortunate processes
  - Thus a process can not control its fault rate; it depends not only on its paging behavior but also on other processes
  - Generally results in greater system throughput and hence it is more common

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

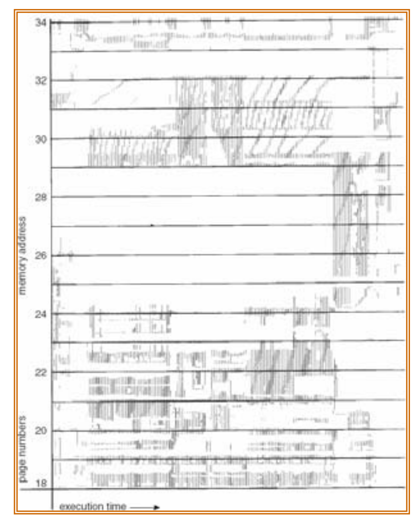


Thrashing (cont.)

- Thrashing is a high paging activity that may occur when the number of frames allocated to a process is below the minimum number of frames required to support its execution
- A process is thrashing if it is busy spending more time paging than executing
- Can we limit the effects of thrashing?
  - Use local replacement algorithm
  - Provide a process as many frames as possible?? How??
    - Working-set strategy which is based the assumption of locality

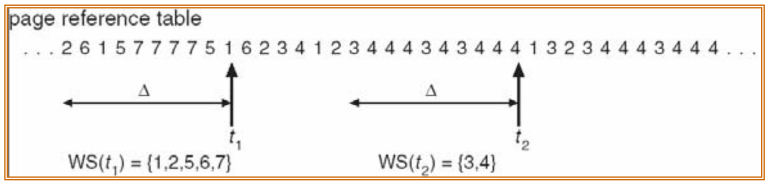
Locality in memory reference pattern

- A locality is a set of pages that are actively used together



Working-Set Model

- The working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set size of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)

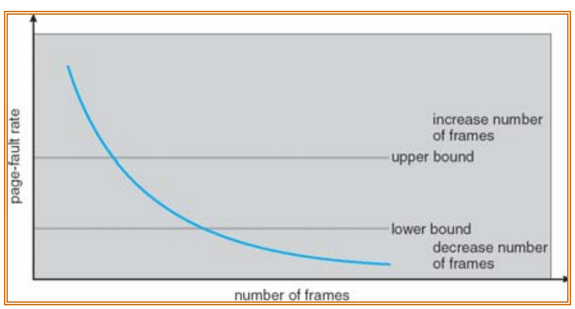


Working-Set Model (cont.)

- The size of Δ and locality:
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - if $D > m \Rightarrow$ Thrashing,
    - where m is the number of total frames available
- Based on , the OS monitor the working set and allocates enough frames for the working set;
  - If no available frames, the OS selects a process to suspend
- The problem is keeping track of the working set

The Page-Fault Frequency (PFF) Strategy

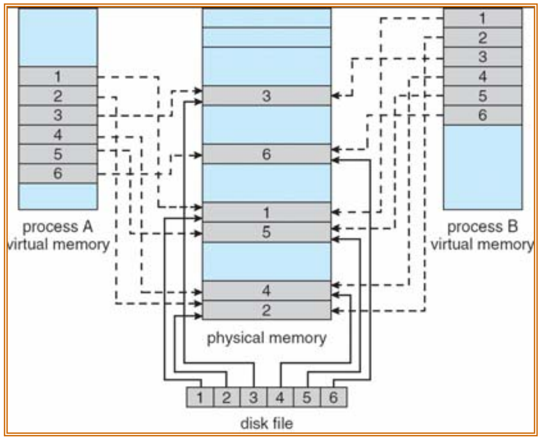
- Define an upper bound U and lower bound L for page fault rates
- Allocate more frames to a process if fault rate is higher than U
- Allocate less frames if fault rate is < L
- The resident set size should be close to the working set size W
- We suspend the process if the PFF > U and no more free frames are available



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read()** and **write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory-Mapped Files



Another Consideration

- A program structure example:
  - Array A[1024, 1024] of integer. Each row is stored in one page
  - Program 1

    for *j* := 1 to 1024 **do**

    for *i* := 1 to 1024 **do**

    A[*i*,*j*] := 0;
  - 1024 x 1024 page faults
  - Program 2

    for *i* := 1 to 1024 **do**

    for *j* := 1 to 1024 **do**

    A[*i*,*j*] := 0;
  - 1024 page faults

A[1,1], A[1,2],…,A[1,1024]
A[2,1], A[2,2],…,A[2,1024]
.
.
.
A[1024,1],A[1024,2],…,A[1024,1024]

Selected Topics of Chapter 9

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005