# Virtual Memory II

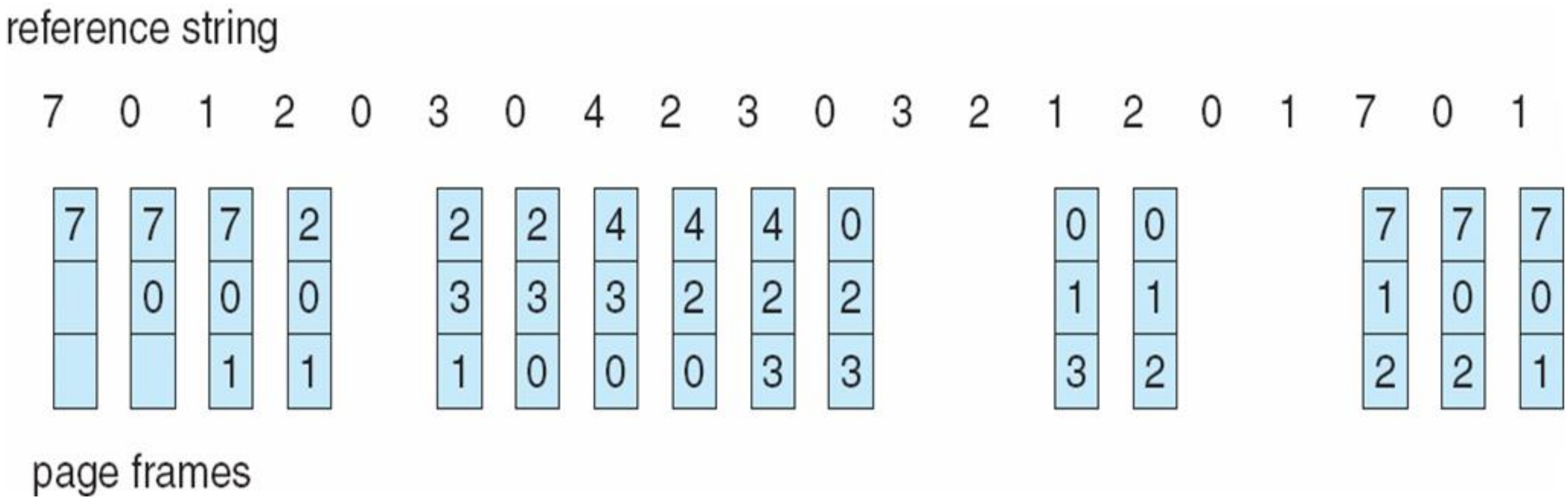
# Page Fault Handling: Basic Framework

1. Find the desired page on disk (back store)
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Bring the desired page into the free frame; update the page tables
4. Restart the process

# Page Replacement

Requirement: an ideal page replacement algorithm should result in a minimum number of page faults

# FIFO Page Replacement



# First-In-First-Out (FIFO) Algorithm

- 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 |
| 3 | 3 | 2 | 4 |

9 page faults

- 4 frames

| | | | |
|---|---|---|---|
| 1 | 1 | 5 | 4 |
| 2 | 2 | 1 | 5 |
| 3 | 3 | 2 | |
| 4 | 4 | 3 | |

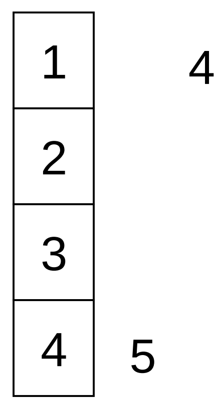
10 page faults

- Belady's Anomaly: more frames $\Rightarrow$ more page faults

# Optimal (Ideal) Algorithm

- Replace pages that will not be used for the longest period of time
- 4 frames example

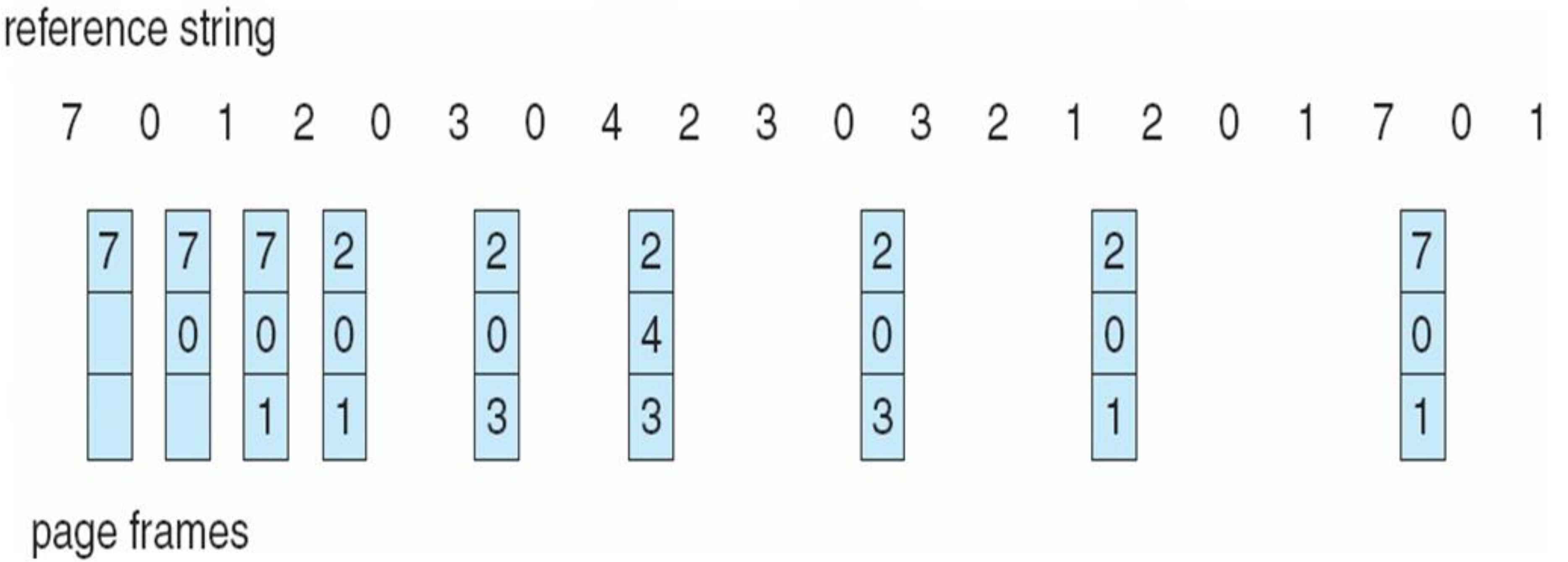
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5




6 page faults

- How do you know this? No.
- Why do we care this algorithm? Used for measuring how well your algorithm performs

# Optimal Page Replacement



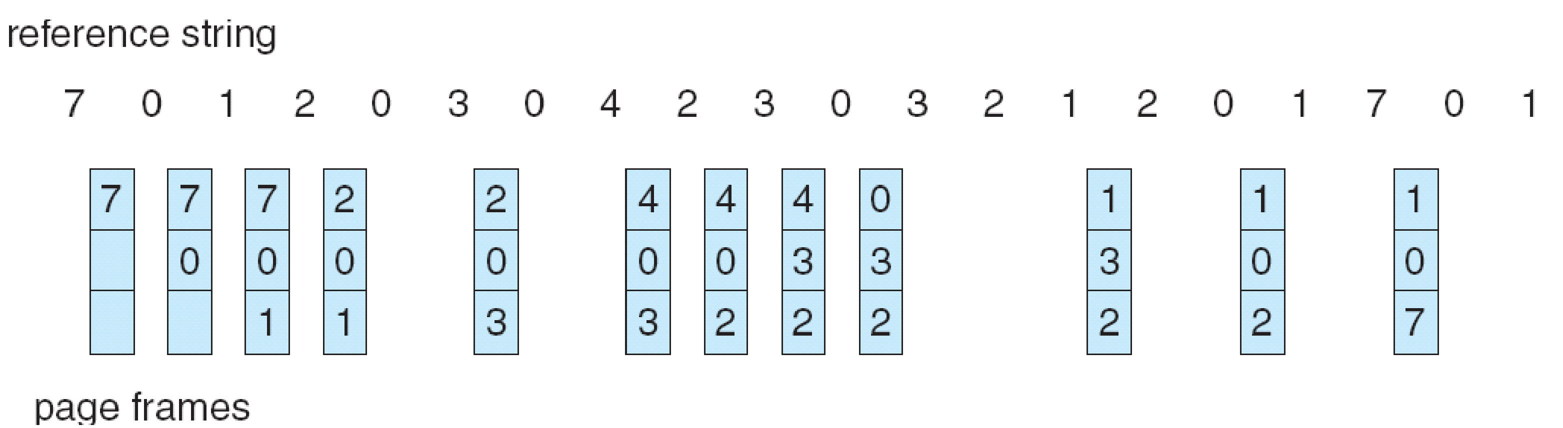
# Least Recently Used (LRU) Algorithm

Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | **5** |
| 2 | 2 | 2 | 2 | 2 |
| 3 | **5** | 5 | **4** | 4 |
| 4 | 4 | **3** | 3 | 3 |

# LRU Page Replacement

- Counter implementation
  - Every frame has a counter; every time the page in a frame is referenced, copy the clock into the counter of the frame
  - When a page needs to be replaced, the page in the frame with the oldest counter value is to be replaced



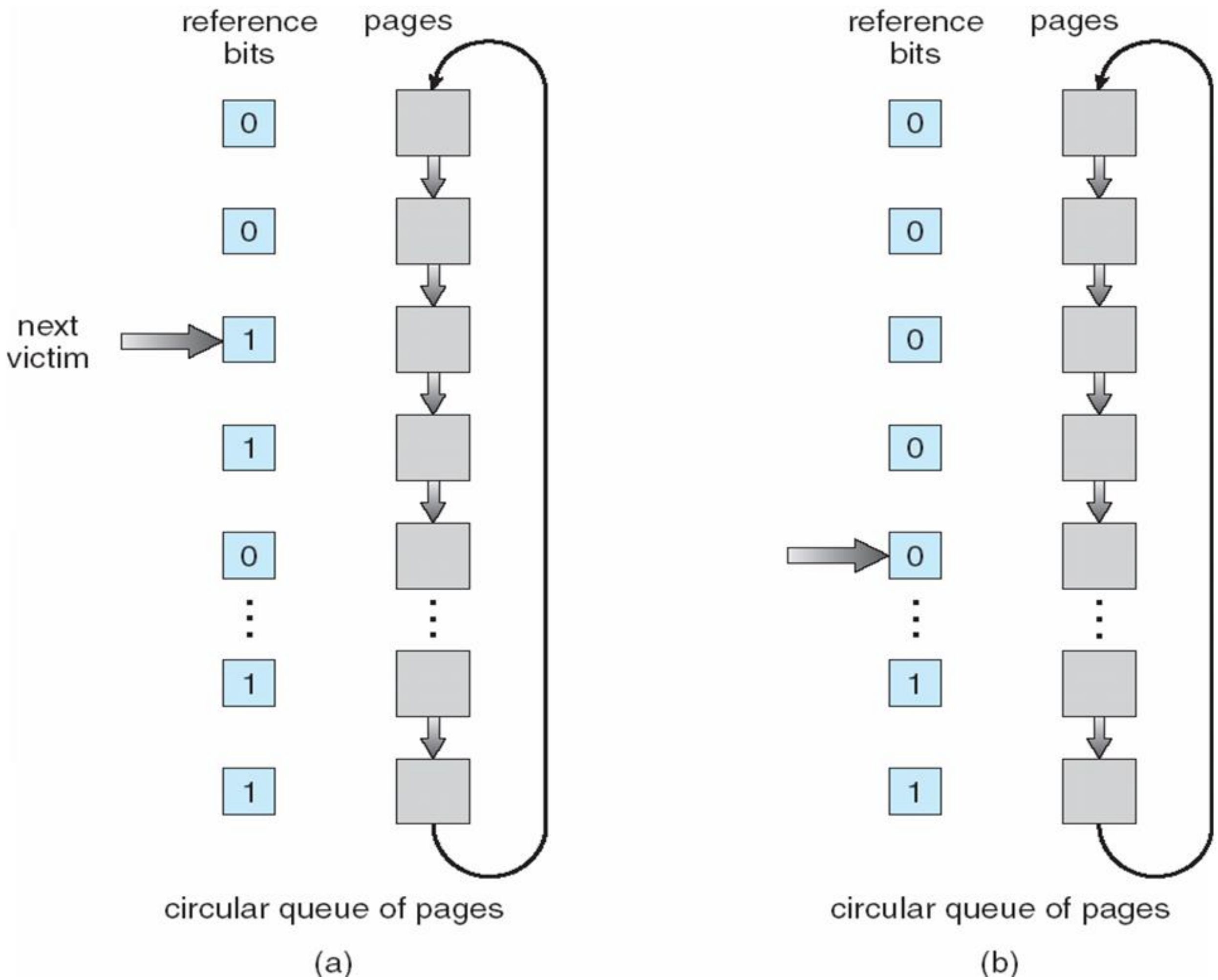
# LRU Algorithm: Stack Implementation

- Keep a stack of page numbers
- Page referenced is in the stack: move it to the top
- Page referenced is not in the stack:
  - If there is free frame → push the page into the stack
  - If there is no free frame → swap out the page on the stack bottom; push the new page into the stack
- Example
  - Reference string 70120304230321201701
  - #of frames=3

# Second-Chance: An Approximate LRU

- A reference bit associated with each page in physical memory
- All pages in physical memory form a circular queue; a pointer pointing to the head element (most recently accessed page)
- When a page is referenced, its reference bit is set to 1.
- When a page should be replaced:
  - Step 1. Move the pointer by one step
  - Step 2. Check the page pointed by the pointer:
    - If the associated bit is 0 → replace the page
    - If the associated bit is 1 → change the bit to 0 and go to step 1.

# Second-Chance Page-Replacement Algorithm

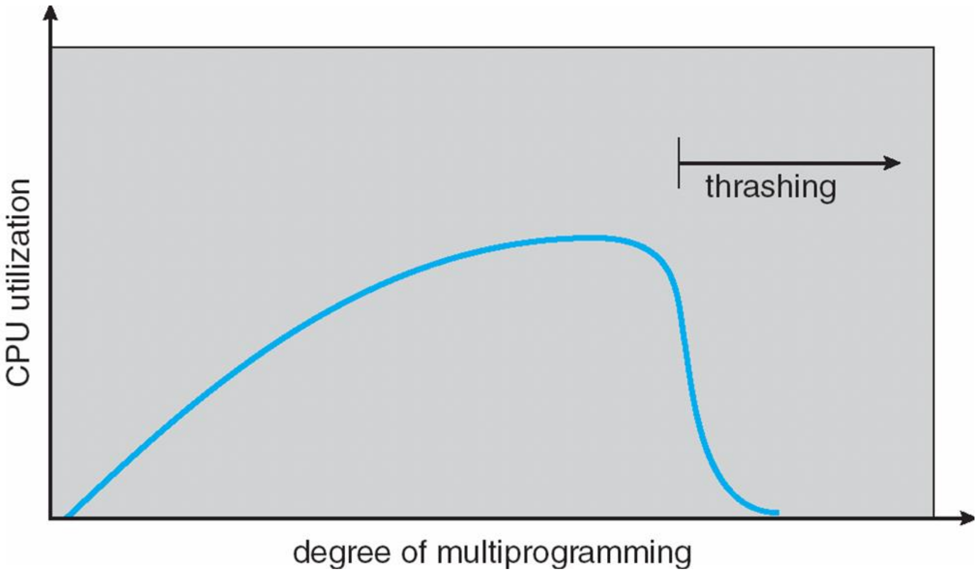


# Minimal Number of Pages Needed

- Each process needs *minimum* number of pages: determined by computer architecture
- Example: IBM 370 needs 6 pages to handle MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*

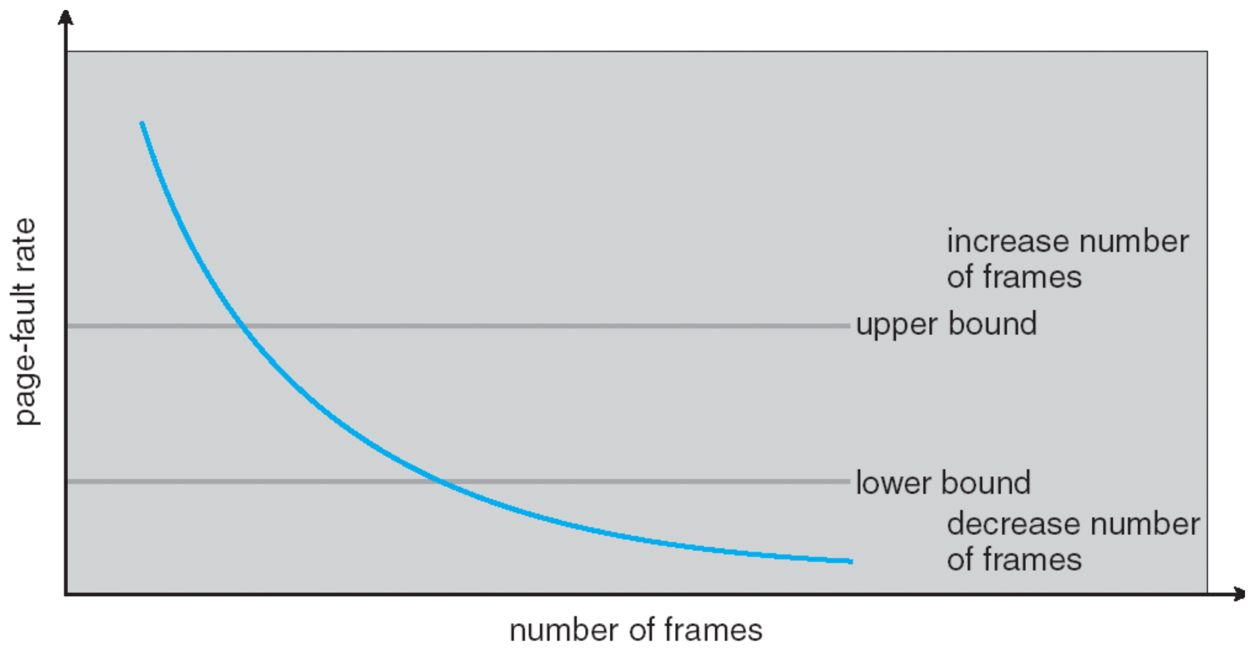
# How many pages are "enough"? Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization: handling page-fault; frequent proc scheduling
  - (because CPU is not fully used) OS thinks that it needs to increase the degree of multiprogramming: another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out



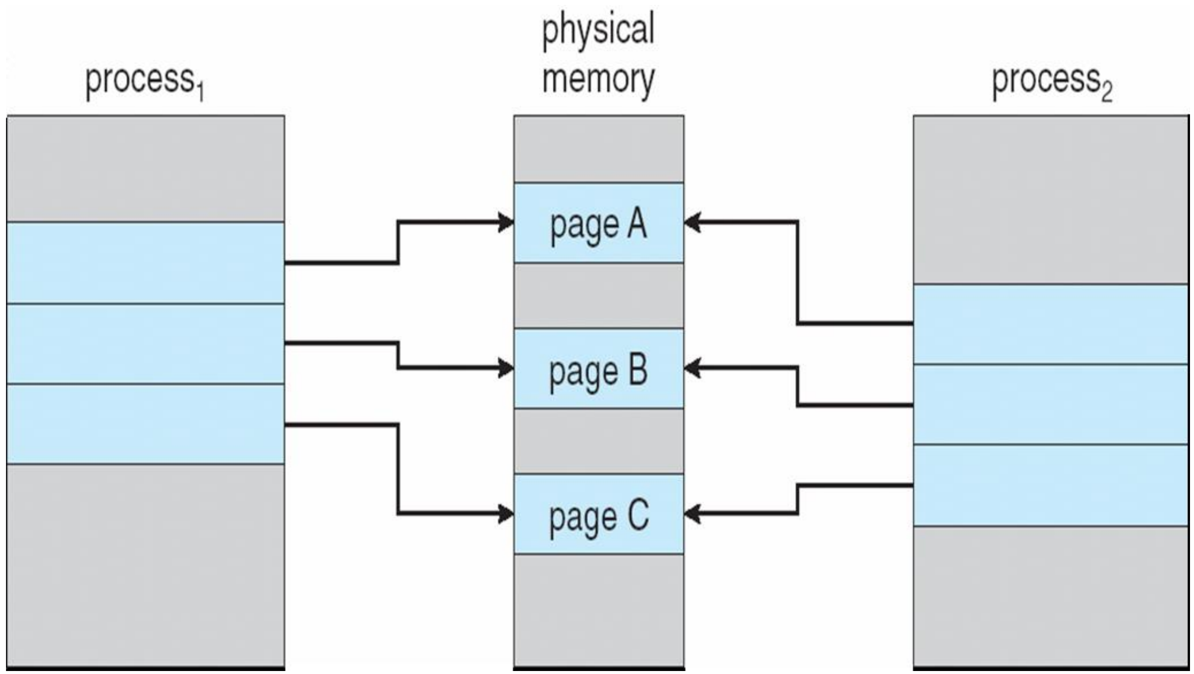
# Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate for a system
- Keep track of the actual page-fault rate for each process
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



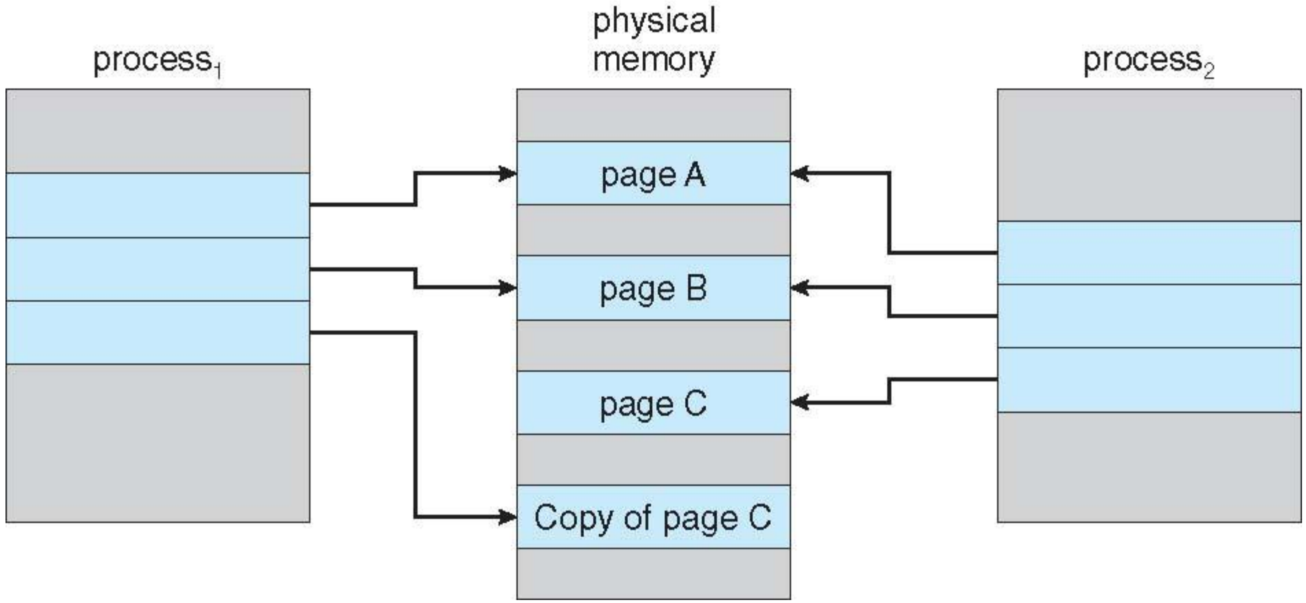
# Copy-on-Write: Speed Up Process Creation

- When a new process is created, it copies the image of its parent.
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory



# Copy-on-Write: Speed Up Process Creation

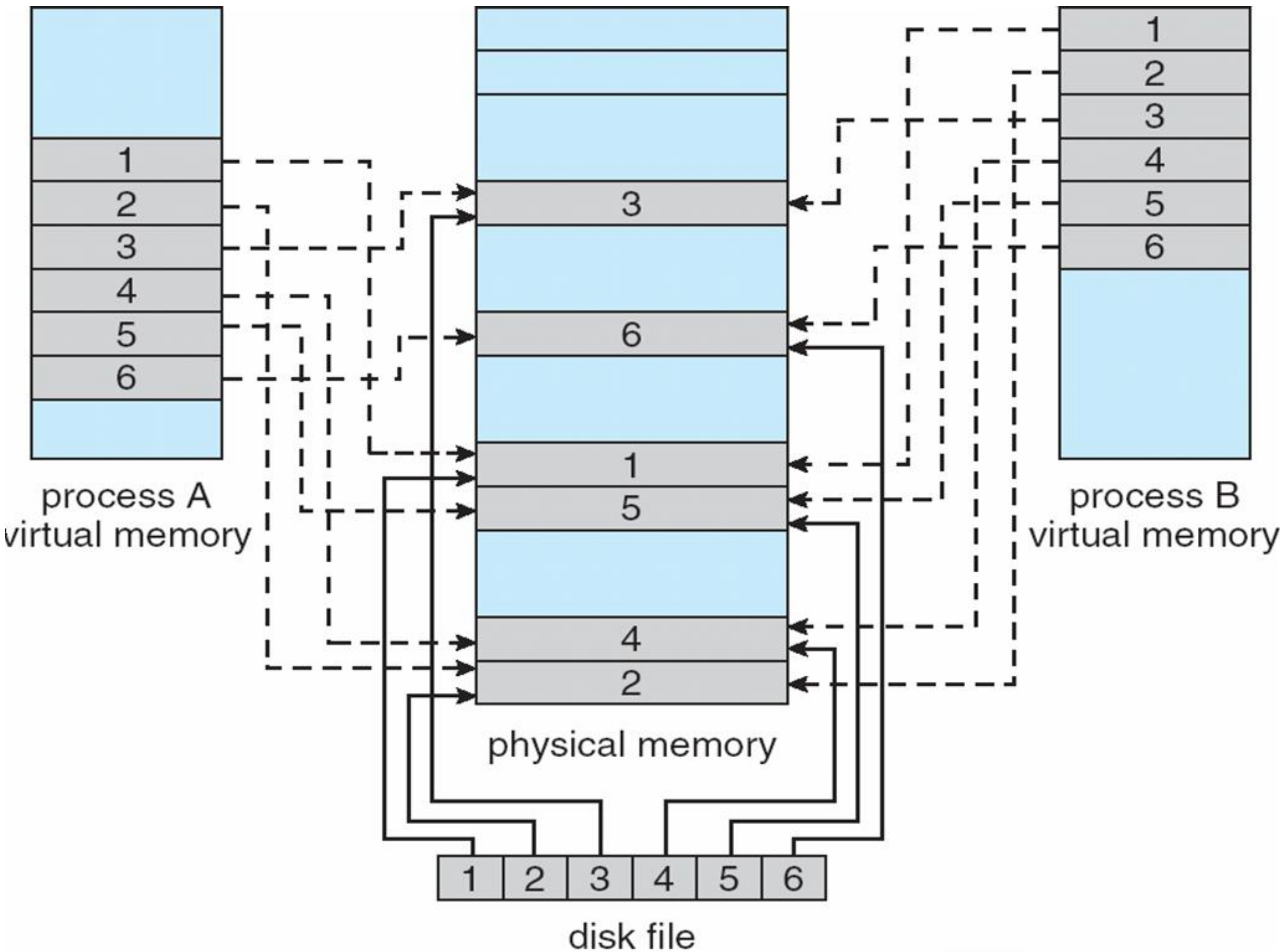
- If either process modifies a shared page, only then is the page copied



# 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()`** **`write()`** system calls.

# Memory Mapped Files



# Review: Synchronization

- Problems:
  - Race condition; Critical Section (Mutual Exclusion); Limited access (counting); Bounded Buffer (producers/consumers); Readers/writers
- Early-day algorithms:
  - Peterson's algorithm; enabling/disabling interrupt
- Hardware mechanisms (atomic instructions):
  - Test-and-set instruction; swap instruction

# Review: Synchronization

- Semaphore:
  - Definitions (wait/signal operations; binary/counting semaphores);
  - Applications (solving bounded buffer problems, readers/writers problems, others like in your assignments)
  - Implementations (how to use test-and-set or swap instructions to implement the wait/signal operations)
- Monitor:
  - Definitions (mutually-exclusive procedures; condition variables with wait/signal operations; queues in a monitor: entry, ready/urgent, condition variable queues)
  - Applications (solving classical synchronization problems)

# Review: Deadlock

- Necessary conditions for deadlocks (4)
- Resource allocation graph; wait-for graph
- Deadlock prevention (how to break necessary conditions for deadlocks)
- Deadlock avoidance (banker's algorithm, resource allocation graph)
- Deadlock detection (like banker's algorithm, wait-for graph)

# Main Memory Management

- Contiguous allocation & drawback (external fragmentation)
- Paging
  - Concepts: logical/physical address, page, frames, page table, related registers
  - Translations between logical address and physical address
  - Hierarchical page tables; inverted page table
- Segmentation
  - Concepts: segment, logical/physical address, segment table (base, limit, access rights), related registers
  - Translation between logical and physical addresses
- Segmentation with paging

# File System Interface

# Files

- Contiguous logical address space
- File Attributes
  - Name: only information kept in human-readable form
  - Identifier: unique tag (number) identifying file within a file system
  - Type: needed for systems that support different types
  - Location: pointer to file location on device
  - Size: current file size
- Types
  - Program: OS-specific
  - Data: numeric, binary, character

# File Structure

- None: sequence of words, bytes
- Simple record structure
  - Lines: Fixed length, or Variable length
- Complex Structures
  - Formatted document
  - Relocate-able load file (executable)
- The first can simulate last two by inserting appropriate control characters
- Who decides the format?
  - Operating system
  - Program

# File Operations

- Create
- Write
- Read
- Reposition within file
- Delete
- Truncate
- Open and Close
  - *Open($F_i$)* – search the directory structure on disk for entry $F_i$, and copy the content of entry to memory
  - *Close ($F_i$)* – move the content of entry $F_i$ in memory to directory structure on disk

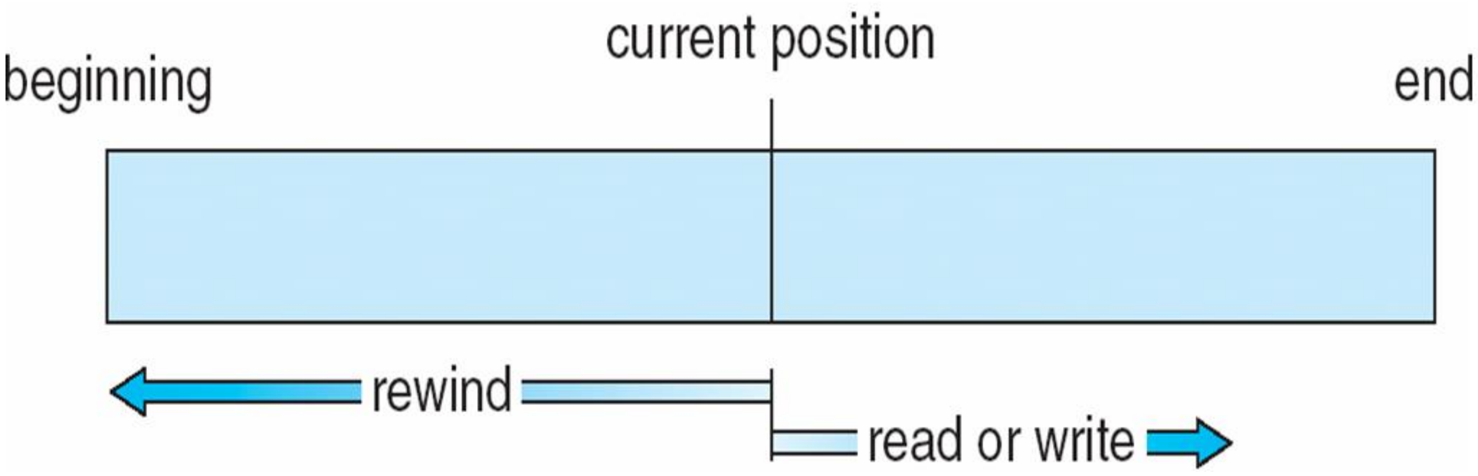
# Open Files

- Some file systems should be open before being accessed; some do not
- Each process maintains an open-file table
  - File pointer: pointer to last read/write location
  - Access rights
- Information maintained by the OS (in a system-level open-file table) :
  - File-open count: counter of number of times a file is opened – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file

# Access Methods

- Sequential Access
  - Current position is maintained

    read next
    write next
    reset



# Access Methods

- Direct Access

read $n$
write $n$
position to $n$

$n$ = relative block number

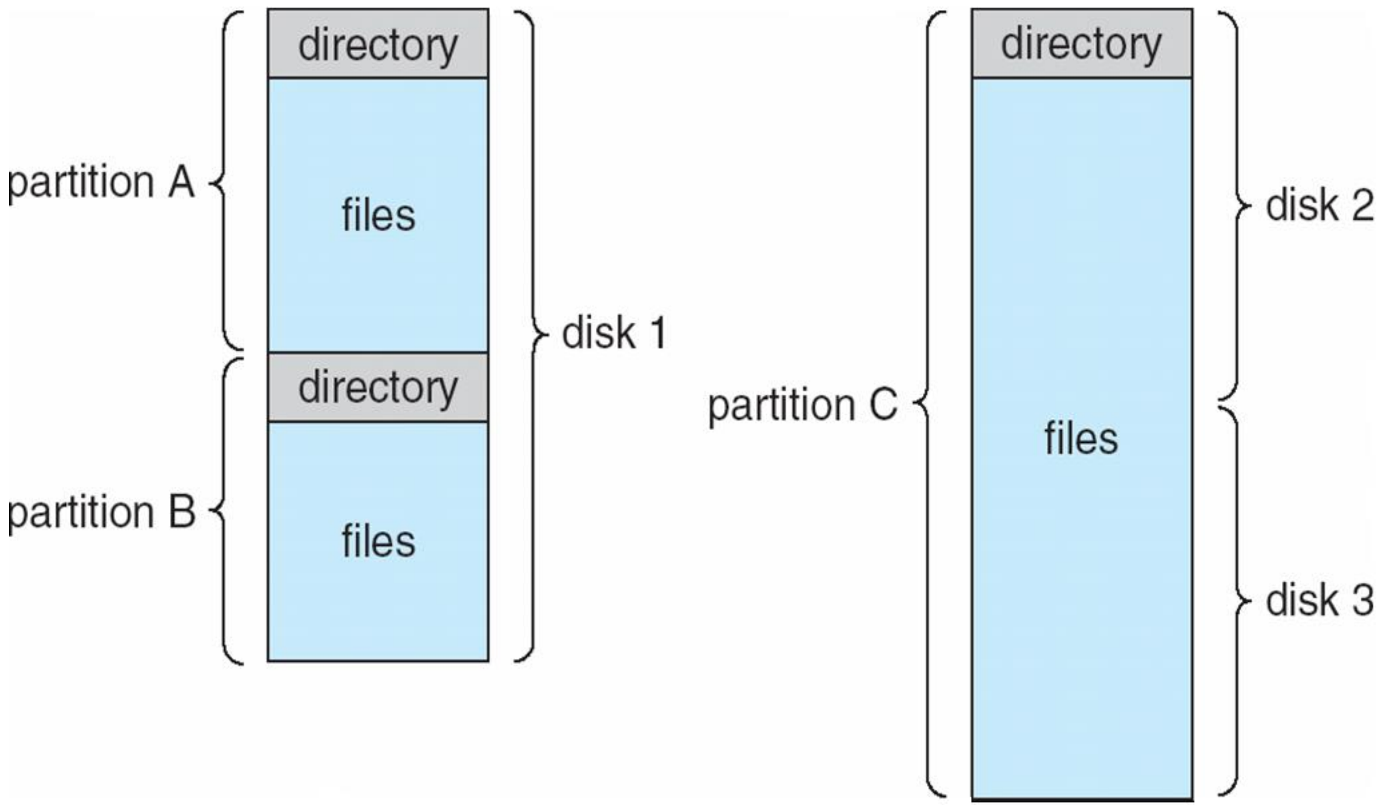
# Simulation of Sequential Access on Direct-access File

| sequential access | implementation for direct access |
|---|---|
| *reset* | *cp* = 0; |
| *read next* | *read cp*;<br>*cp* = *cp* + 1; |
| *write next* | *write cp*;<br>*cp* = *cp* + 1; |

# Disk Structure

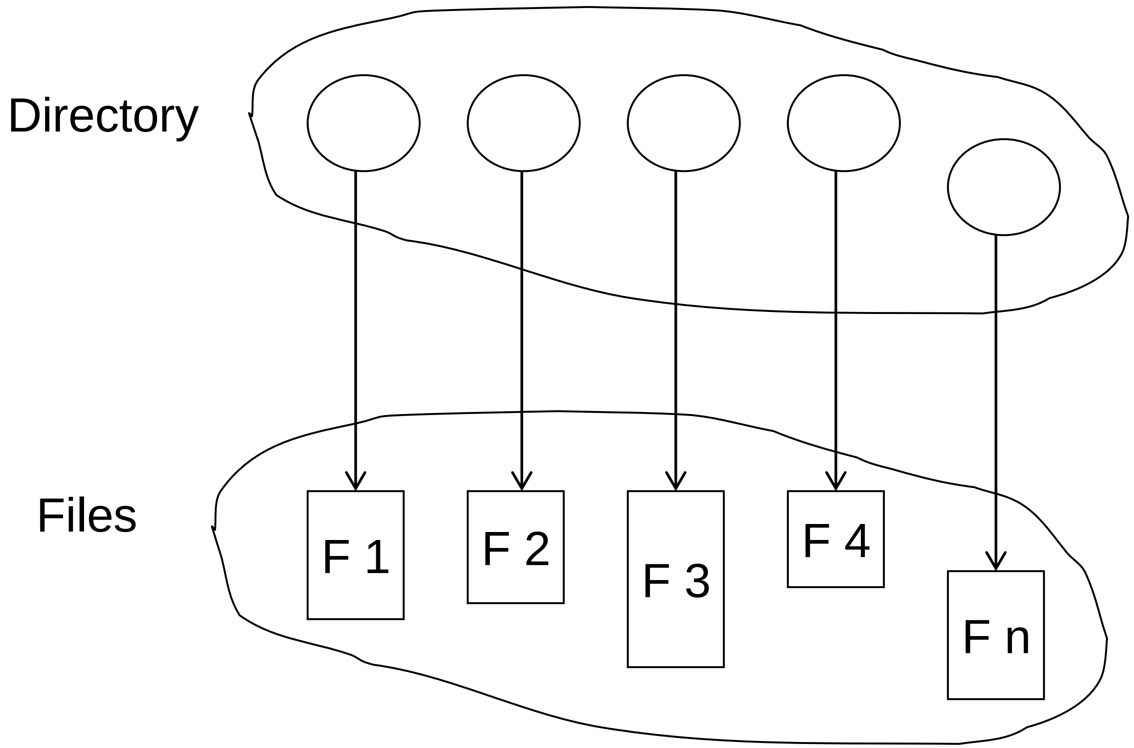
- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions also known as minidisks, slices
- Entity containing a file system known as a volume
- Each volume containing file system and also tracks that file system's info in device directory or volume table of contents

# A Typical File-system Organization



# Directory and Files

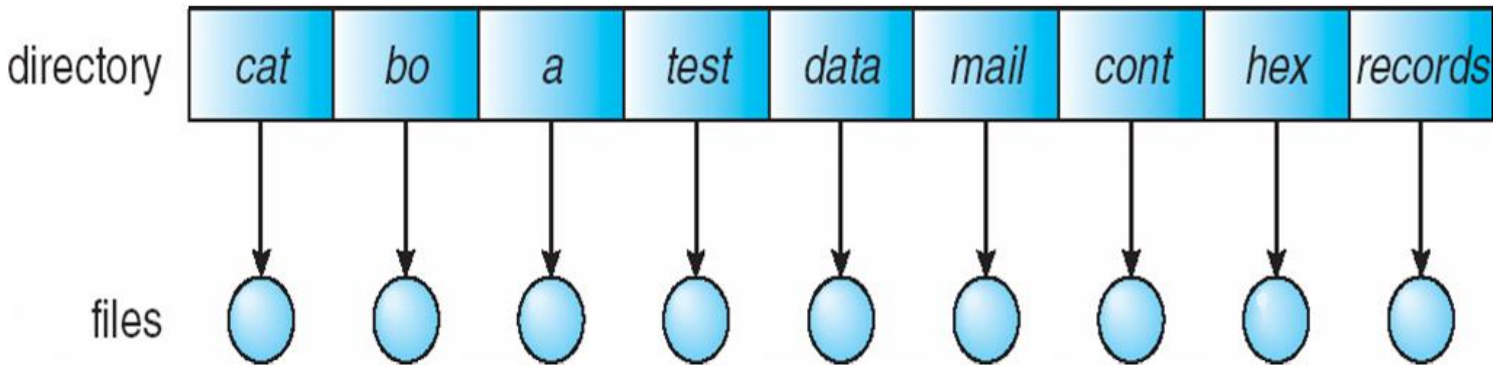
- A collection of nodes containing information about all files



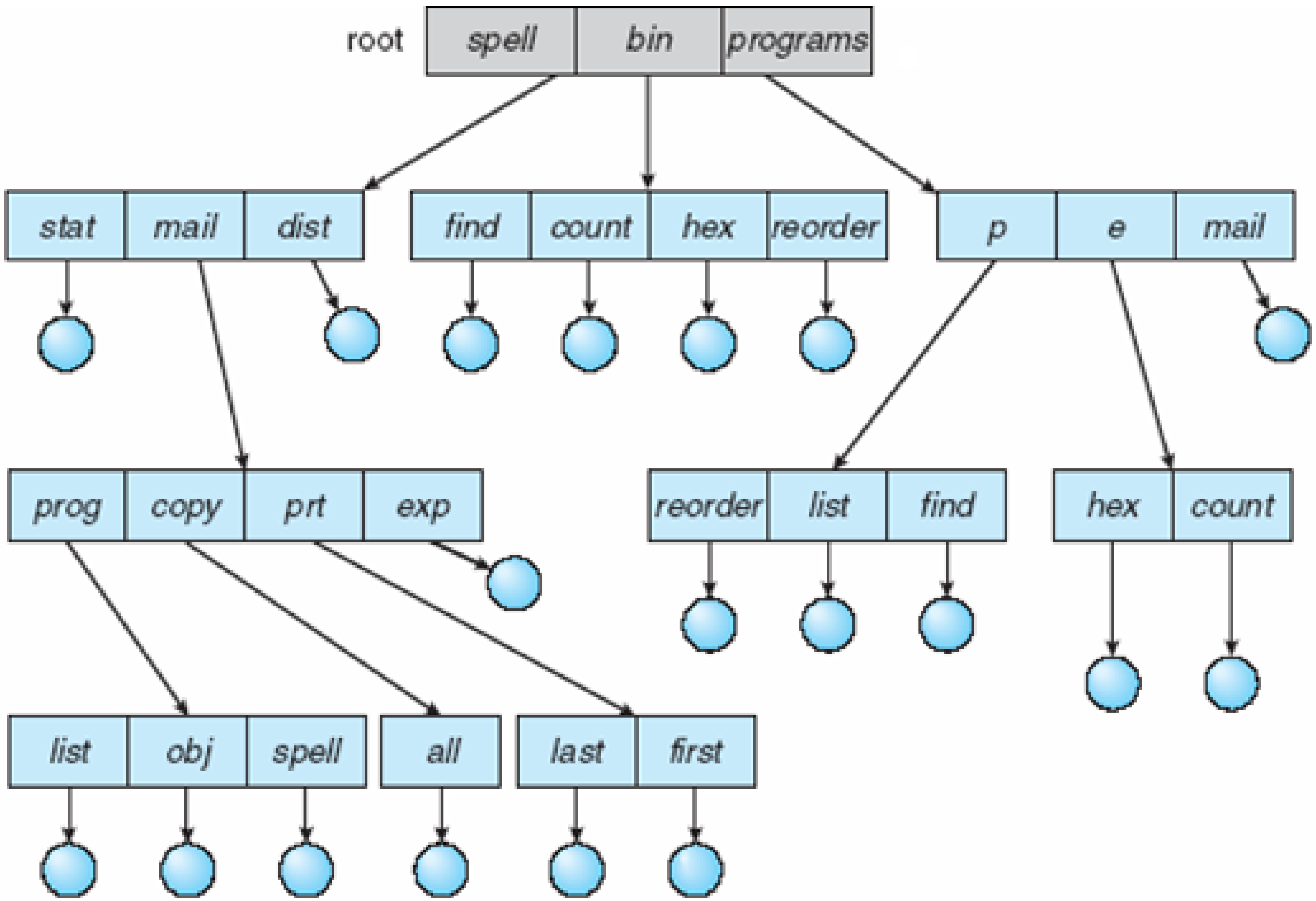

Both the directory structure and the files reside on disk.

# Single-Level Directories

- A single directory for all users



# Tree-Structured Directories

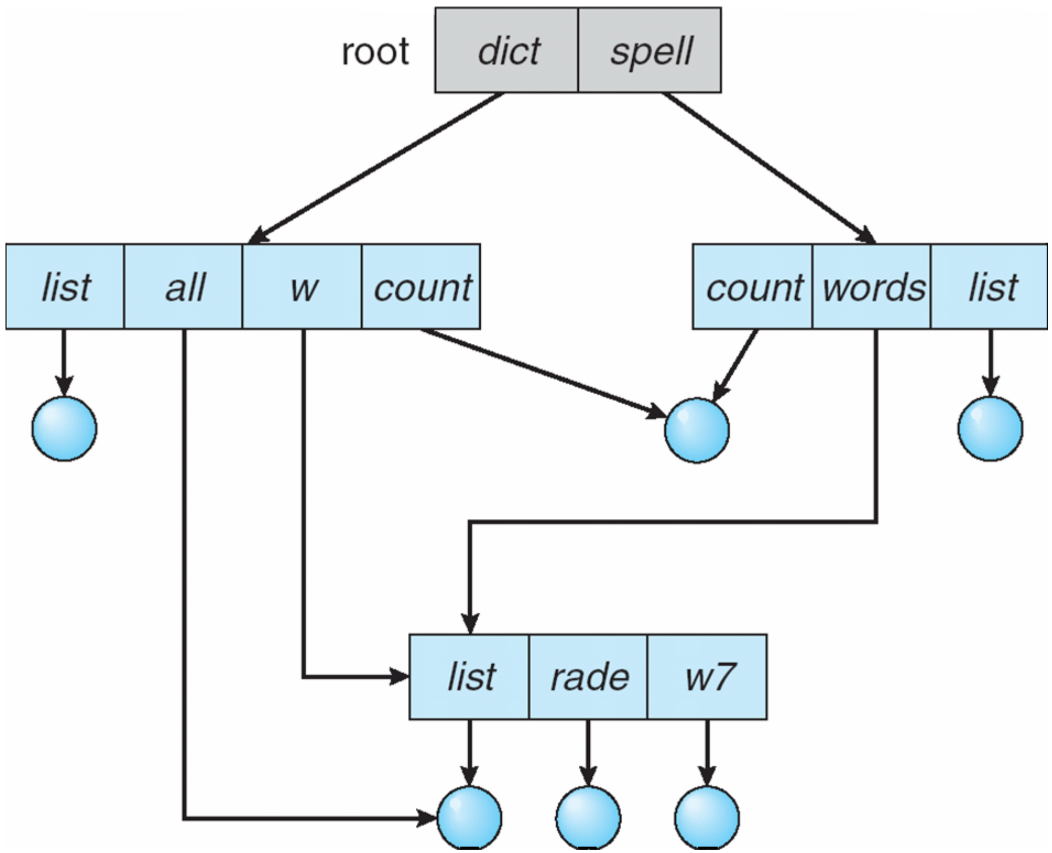


# Tree-Structured Directories

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - cd /spell/mail/prog
  - type list

# Acyclic-Graph Directories

- Have shared subdirectories and files

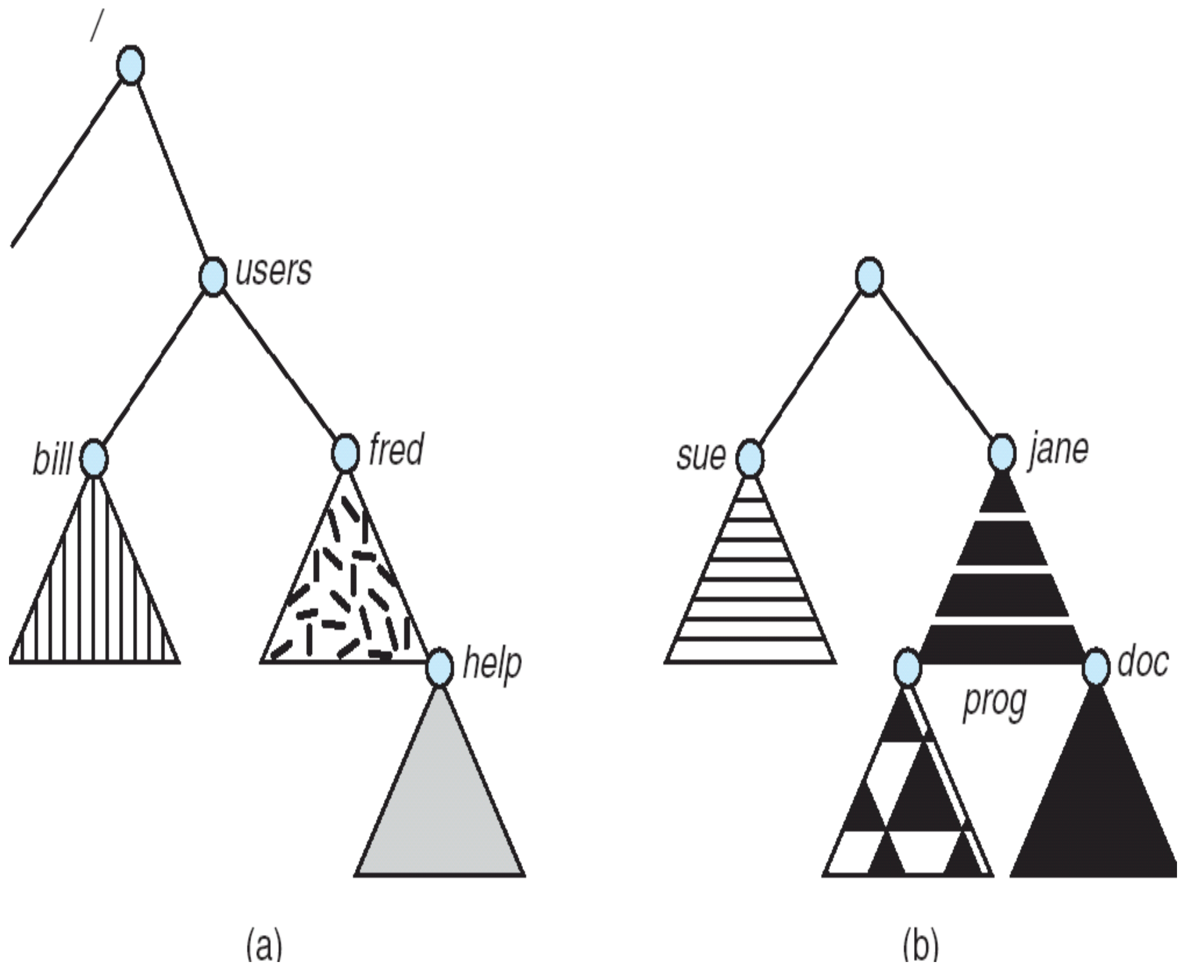


# Acyclic-Graph Directories

- Two different names (aliasing)
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

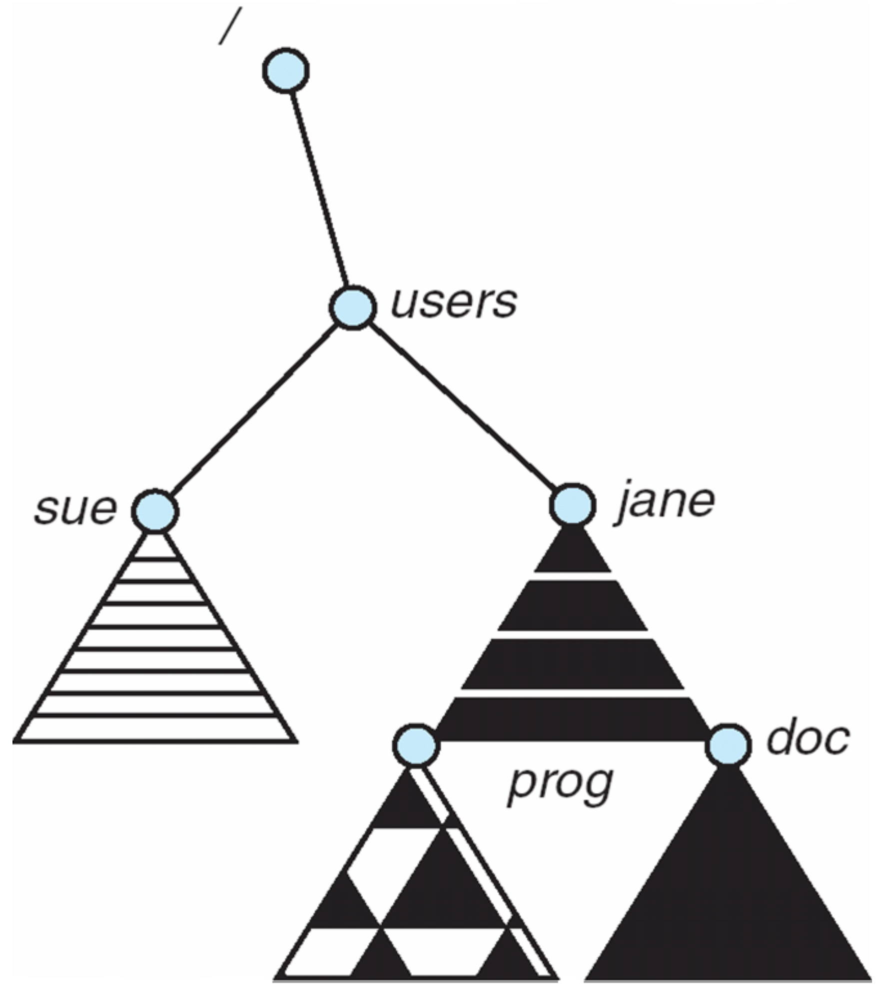
# File System Mounting

- When a system bootstraps, only the root file system is accessible
- A (non-root) file system must be **mounted** before it can be accessed
- An un-mounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point**



# After (b) is mounted to /users

- When a system bootstraps, only the root file system is accessible
- A (non-root) file system must be **mounted** before it can be accessed
- An un-mounted file system (i.e. Fig. 11-11(b)) is mounted at a **mount point**



# Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

# Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

| | | | |
|---|---|---|---|
| | | | RWX |
| a) **owner access** | 7 | $\Rightarrow$ | 1 1 1 |
| | | | RWX |
| b) **group access** | 6 | $\Rightarrow$ | 1 1 0 |
| | | | RWX |
| c) **public access** | 1 | $\Rightarrow$ | 0 0 1 |

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.

owner group public

chmod 761 game

# A Sample UNIX Directory Listing

| | | | | | | |
|---|---|---|---|---|---|---|
| -rw-rw-r-- | 1 | pbg | staff | 31200 | Sep 3 08:30 | intro.ps |
| drwx------ | 5 | pbg | staff | 512 | Jul 8 09.33 | private/ |
| drwxrwxr-x | 2 | pbg | staff | 512 | Jul 8 09:35 | doc/ |
| drwxrwx--- | 2 | pbg | student | 512 | Aug 3 14:13 | student-proj/ |
| -rw-r--r-- | 1 | pbg | staff | 9423 | Feb 24 2003 | program.c |
| -rwxr-xr-x | 1 | pbg | staff | 20471 | Feb 24 2003 | program |
| drwx--x--x | 4 | pbg | faculty | 512 | Jul 31 10:31 | lib/ |
| drwx------ | 3 | pbg | staff | 1024 | Aug 29 06:52 | mail/ |
| drwxrwxrwx | 3 | pbg | staff | 512 | Jul 8 09:35 | test/ |