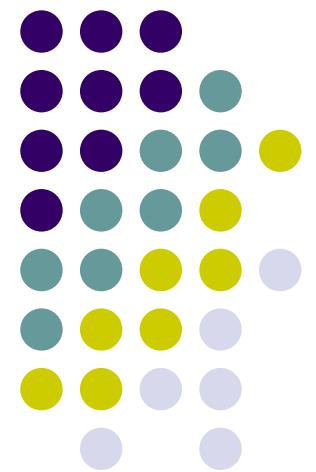
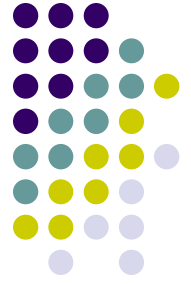


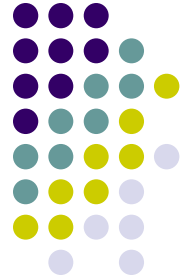
File Management



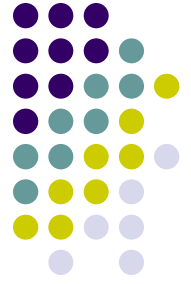


Two Parts

- Filesystem Interface
 - Interface the user sees
 - Organization of the files as seen by the user
 - Operations defined on files
 - Properties that can be read/modified
- Filesystem design
 - Implementing the interface



Filesystem Interface



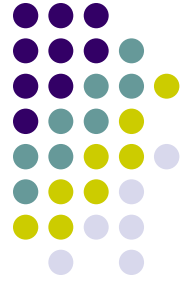
Basic Topics

- File Concept
- Access Methods
- Directory Structure
- File System Mounting
- File Sharing
- Protection



File Concept

- Logical units of information on secondary storage
- Named collection of related info on secondary storage
- Smallest unit of allocation on disk
 - All info must be in at least one file
- Abstracts out the secondary storage details by presenting a common logical storage view



File Types

- Data
 - Text, binary,...
- Program
- Regular files – stores information
- Directory – stores information about file(s)
- Device files – represents different devices



File Structure

- None - sequence of words, bytes
- Simple record structure
 - Lines
 - Fixed length
 - Variable length
- Complex Structures
 - Formatted document
 - Relocatable load file



Important File Attributes

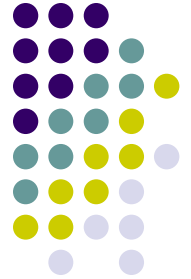
- **Name** – only information kept in human-readable form
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk



File Operations

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

Access Methods



- Sequential Access

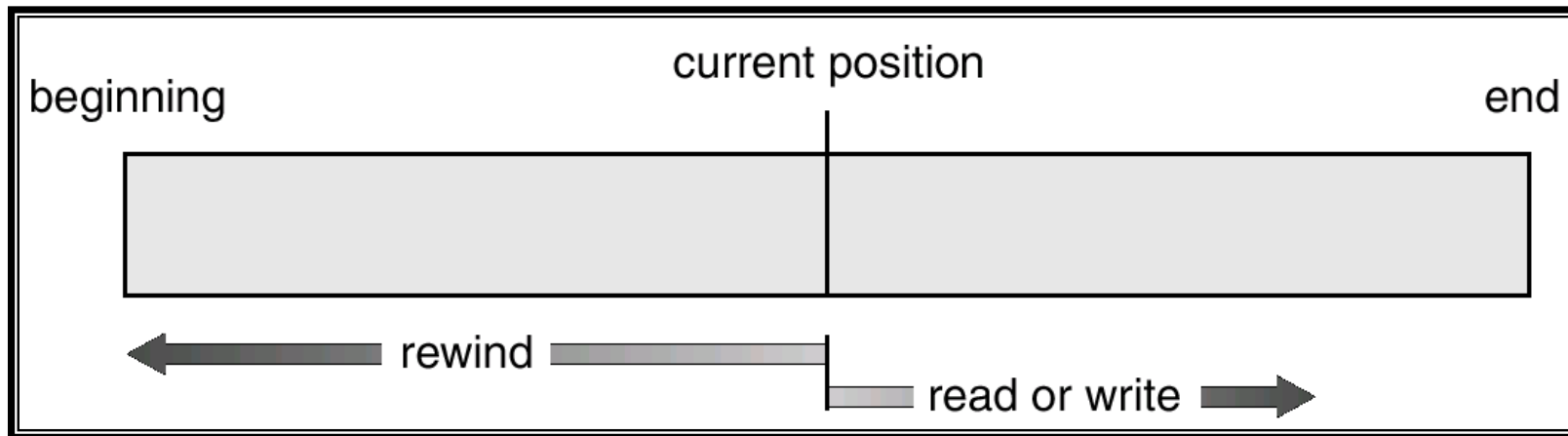
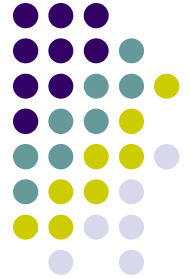
read next
write next
reset

- Direct Access

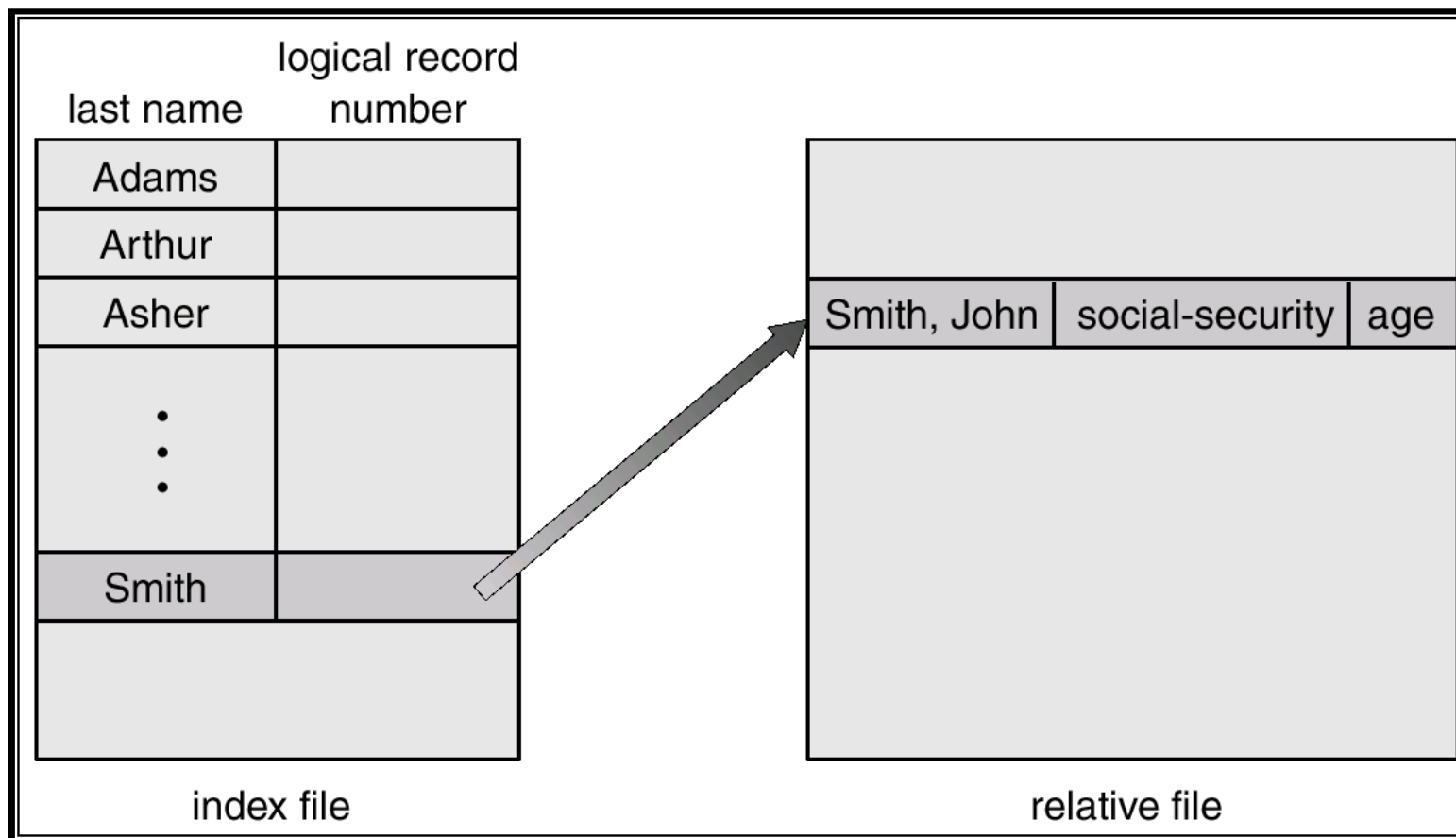
read n
write n
position to n
read next
write next

n = relative block number

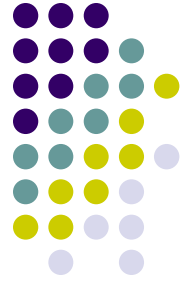
Sequential-access File



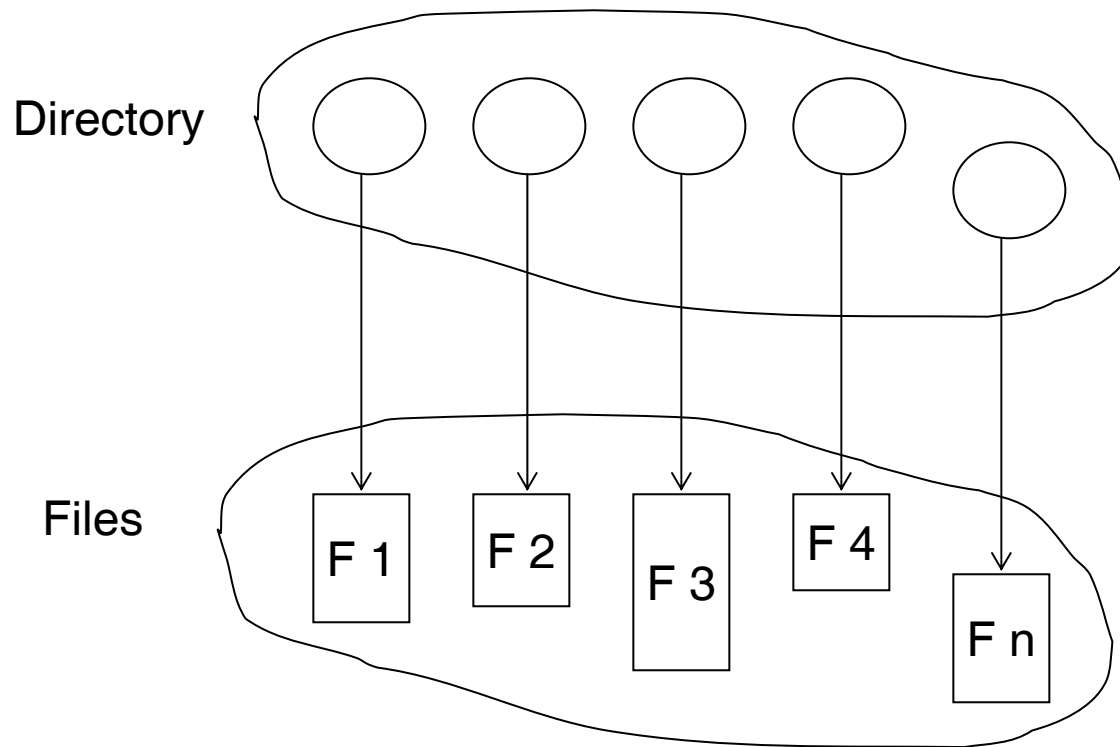
Example of Index and Relative Files



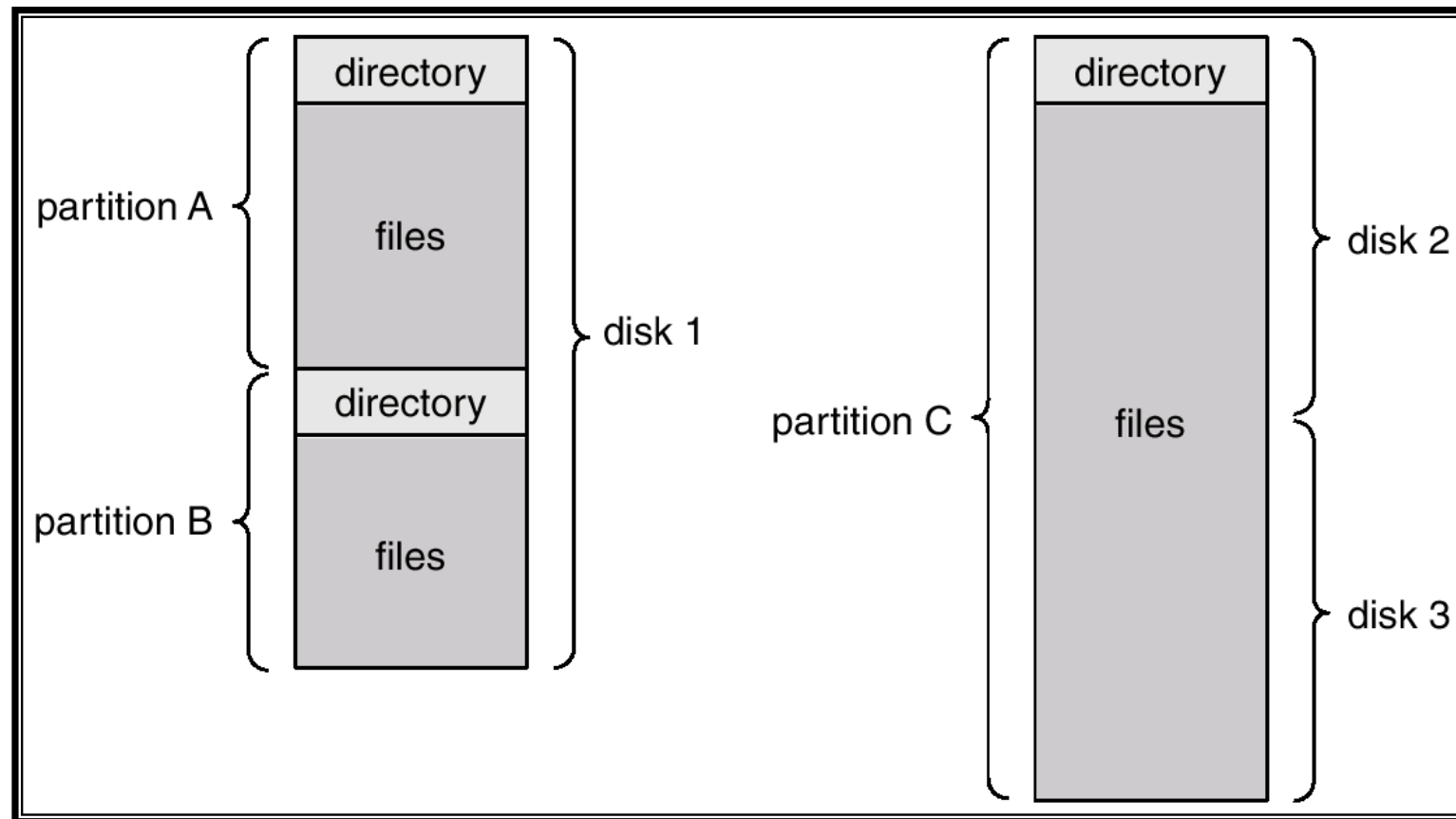
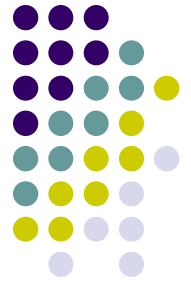
Directory Structure



- A collection of nodes containing information about all files



A Typical File-system Organization

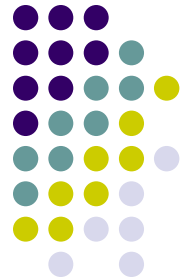


Information in a Device Directory



- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information (discuss later)

Operations Performed on Directory



- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Organize the Directory (Logically) to Obtain

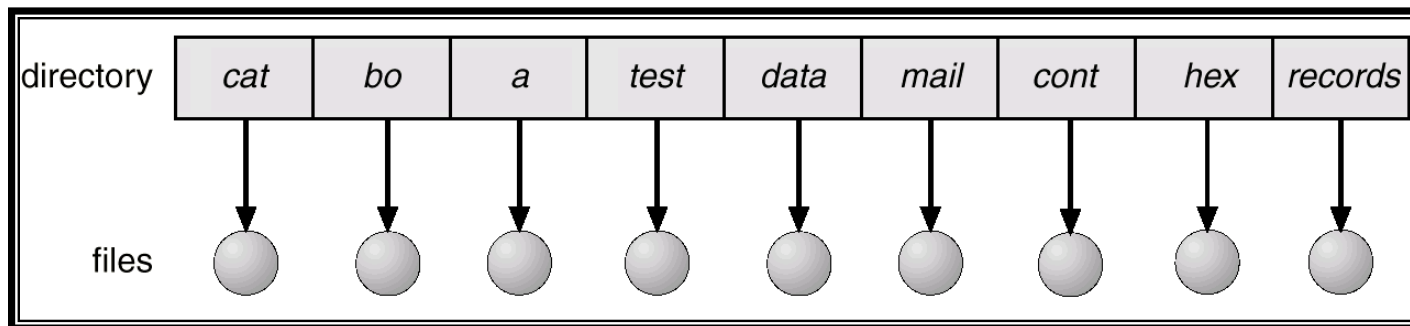


- Efficiency – locating a file quickly
- Naming – convenient to users
 - Two users can have same name for different files.
 - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)

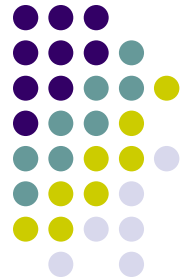


Single-Level Directory

- A single directory for all users

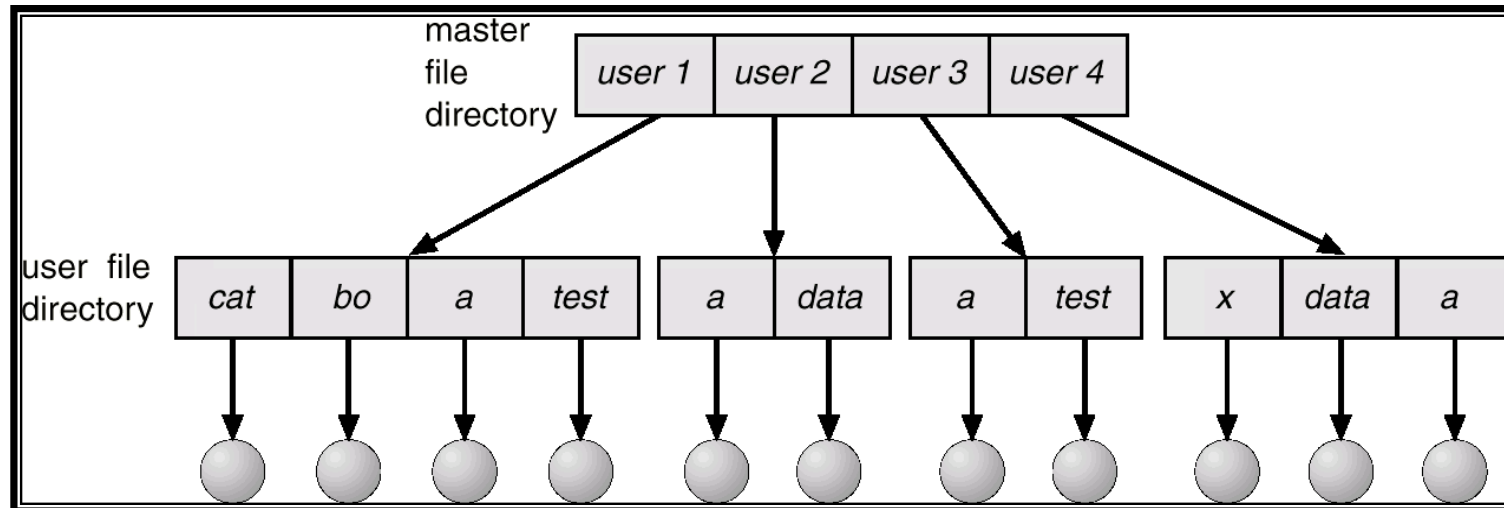


- Problems
 - Naming problem
 - Grouping problem



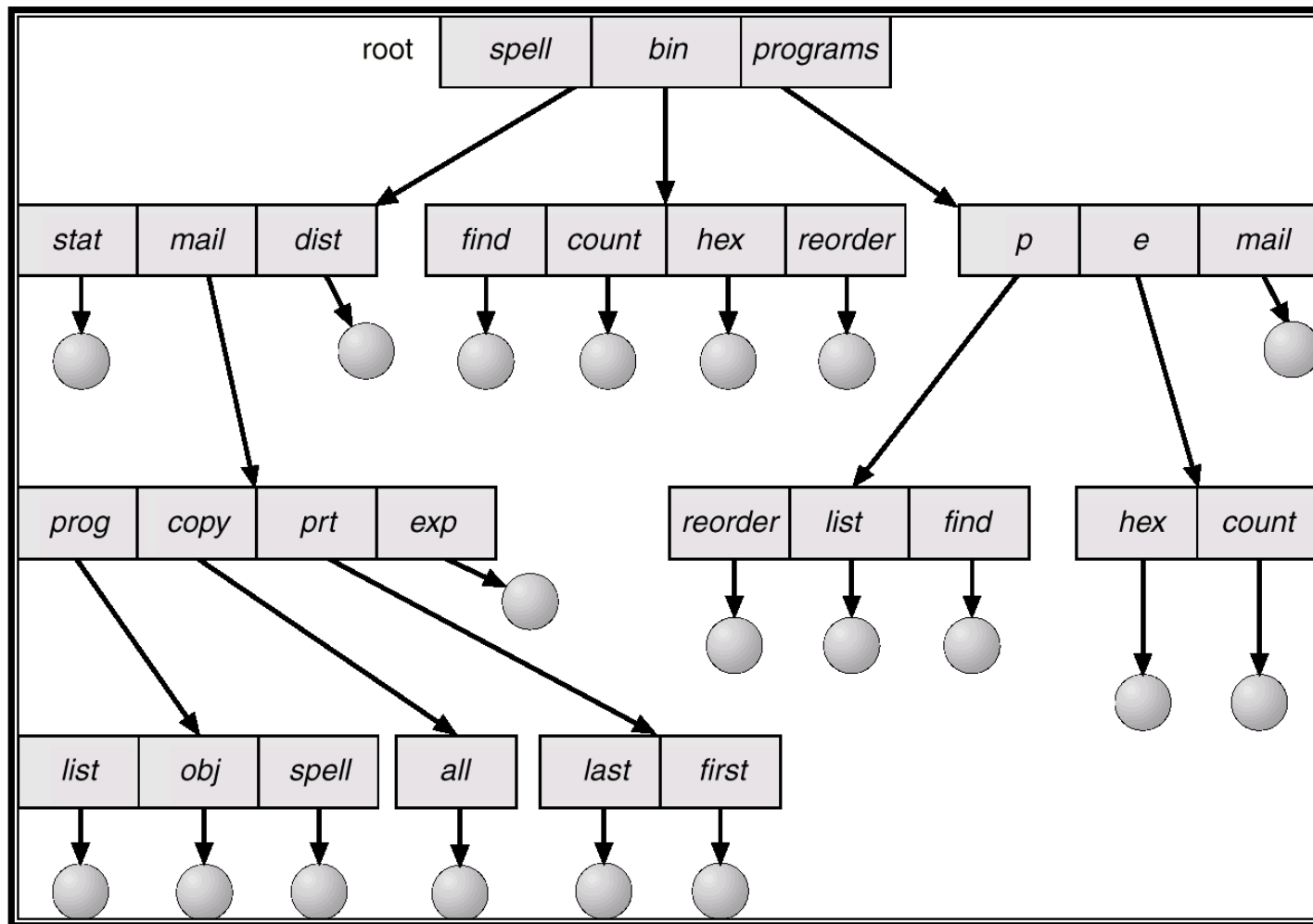
Two-Level Directory

- Separate directory for each user



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories

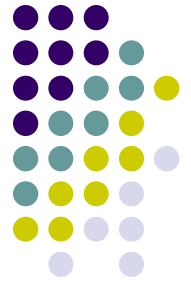


Tree-Structured Directories (Cont.)



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

Tree-Structured Directories (Cont.)



- **Absolute** or **relative** path name
- Creating a new file is done in current directory
- Delete a file
- Creating a new subdirectory is done in current directory.

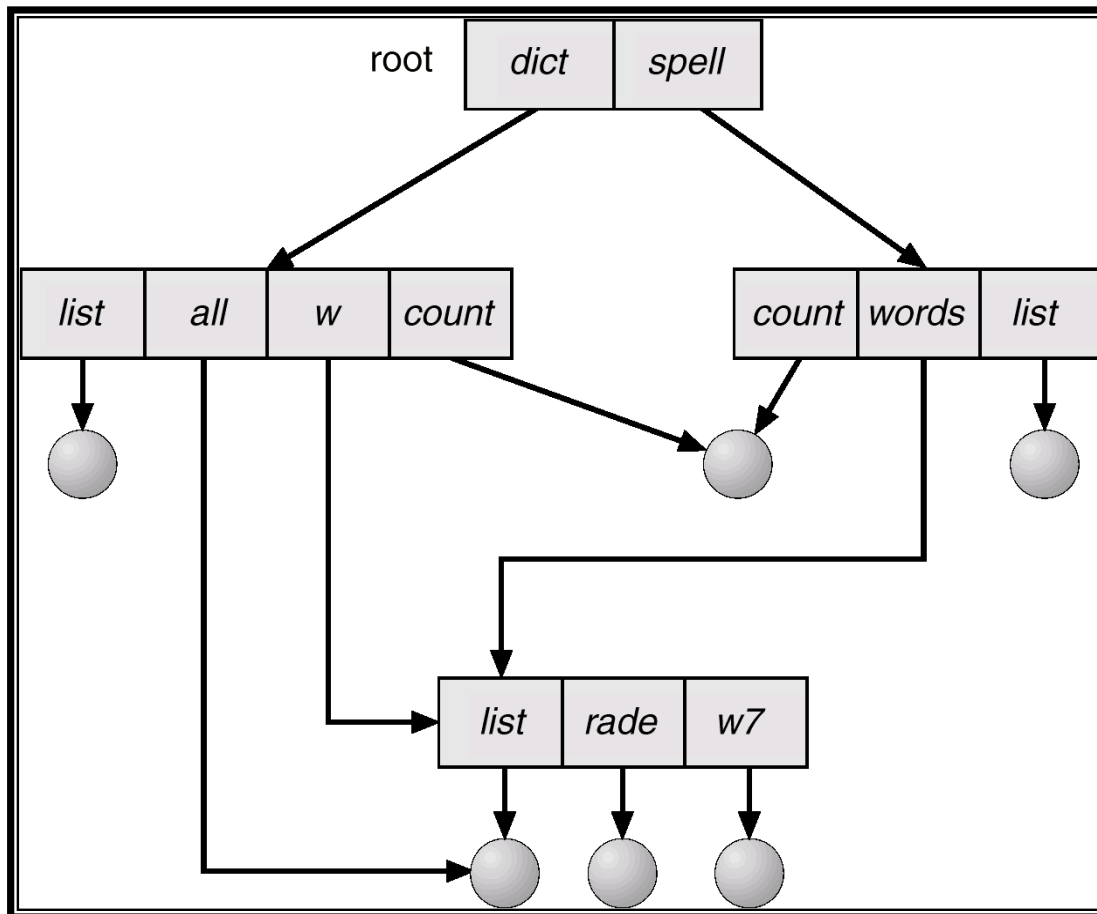
rm <file-name>

mkdir <dir-name>

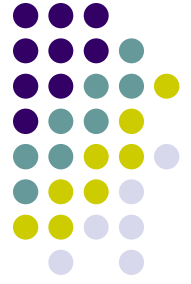


Acyclic-Graph Directories

- Have shared subdirectories and files



Acyclic-Graph Directories (Cont.)

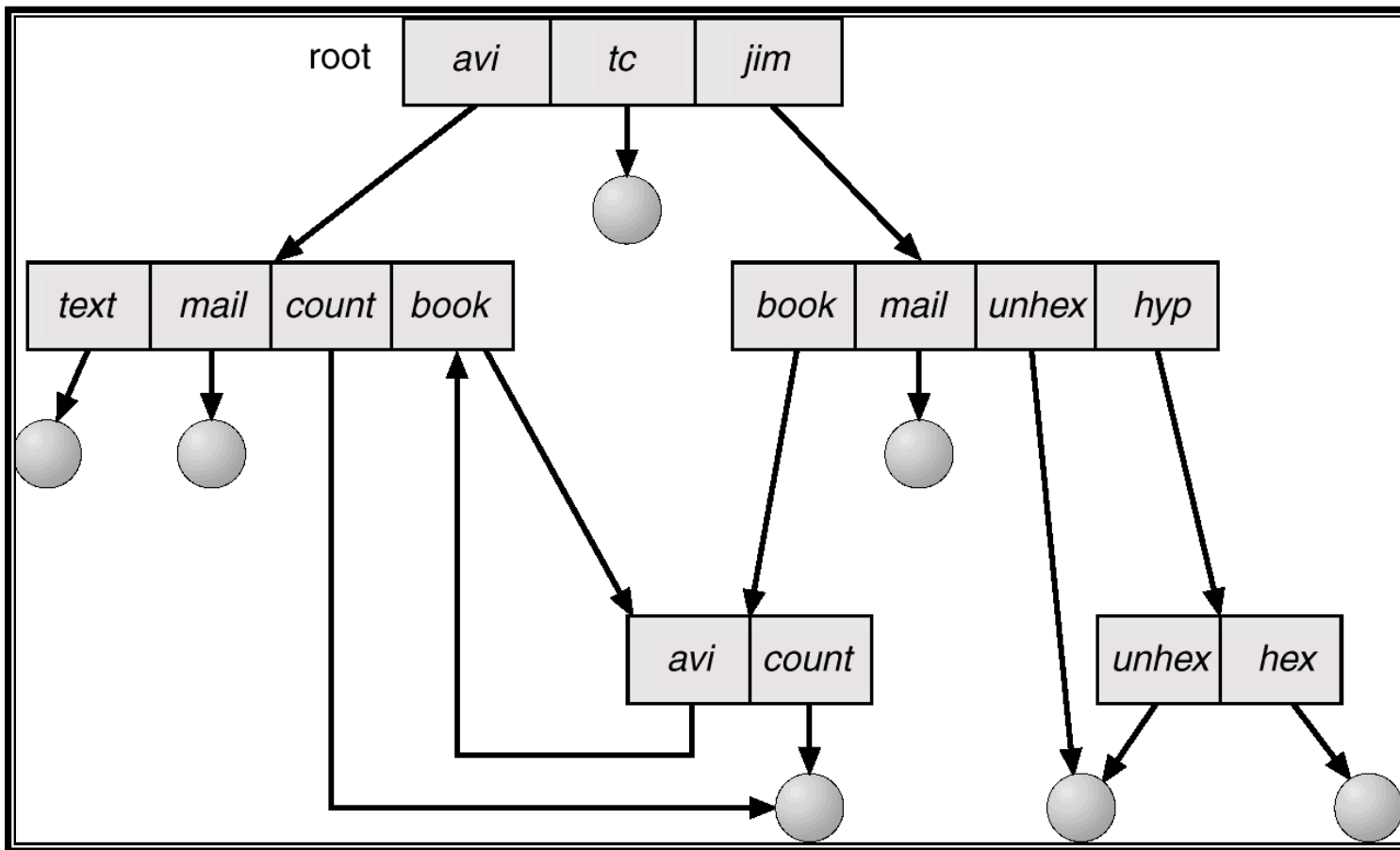
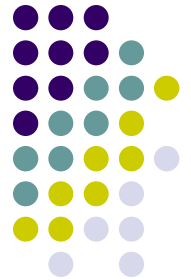


- Two different names (aliasing)
- If *dict* deletes *list* \Rightarrow dangling pointer

Solutions:

- Backpointers, so we can delete all pointers
Variable size records a problem
- Backpointers using a daisy chain organization
- Entry-hold-count solution

General Graph Directory



General Graph Directory (Cont.)

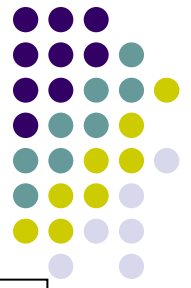


- How do we guarantee no cycles?
 - Allow only links to file not subdirectories
 - Garbage collection
 - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

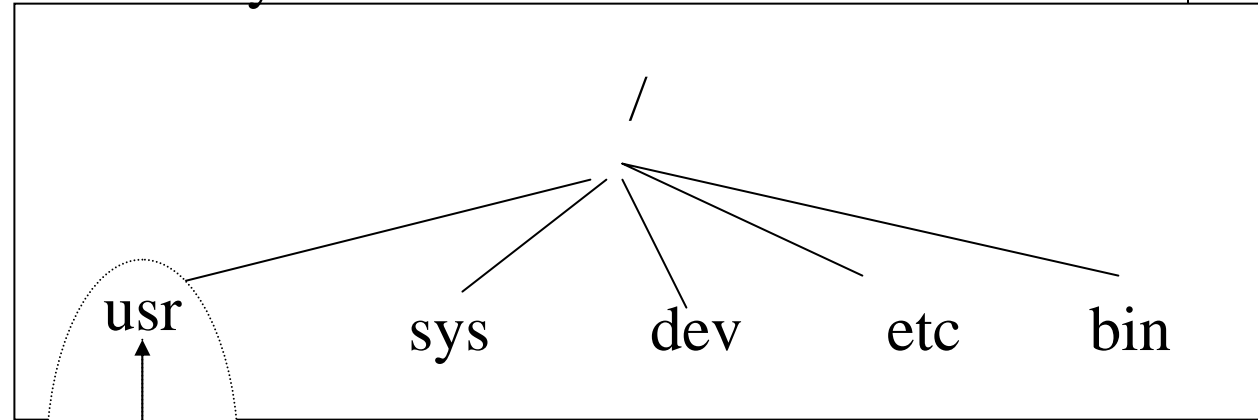


File System Mounting

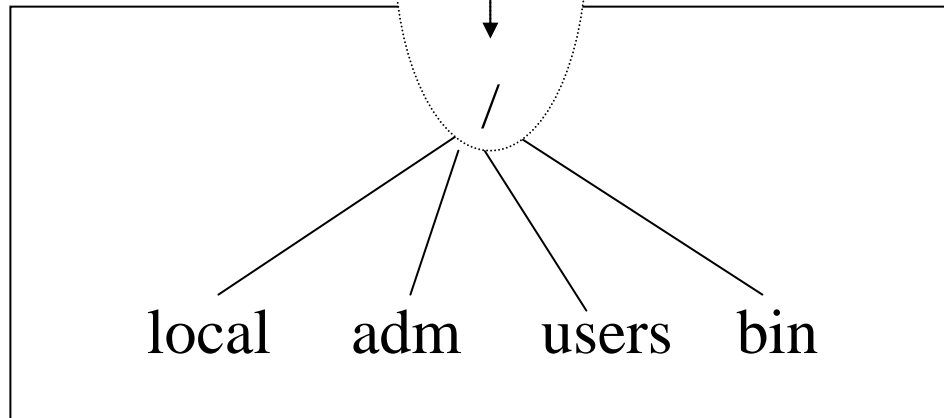
- A filesystem must be **mounted** before it can be accessed
- One file system designated as **root filesystem**
- Root directory of root filesystem is **system root directory**
- Parts of other filesystems are added to directory tree under root by **mounting** onto a directory in the root filesystem.
- The directory onto which it is mounted on is called the **mount point**
- The previous contents of the mount point become inaccessible

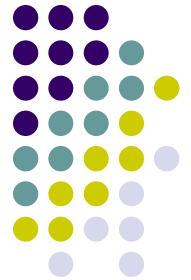


root file system

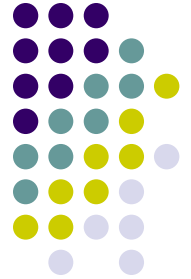


mounted filesystem fs1





- Accessing `/usr/adm/...` now actually accesses `/adm/..` in filesystem `fs1`
- `/usr` in the root file system is the mountpoint
- Anything under `/usr` in the root filesystem becomes inaccessible until `fs1` is unmounted
- Mounting now can be done on any other mountpoint, including any directory on an earlier mounted filesystem
 - Ex. can now mount some other filesystem `fs2` on `/usr/adm`, will hide all files under `/adm` under `fs1` and access to `/usr/adm` will go to corresponding part of `fs2`
- Need not mount `/` always, can mount any subtree of a filesystem on a mountpoint to add only part of a filesystem (but has to be a complete subtree)



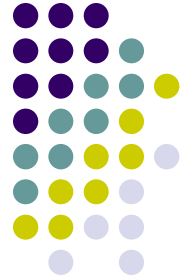
File Sharing

- Create links to files
 - Same file accessed from two different places in directory structure using possibly different names
- Soft Link vs. Hard Links



Protection

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



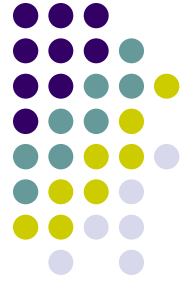
Filesystem Implementation



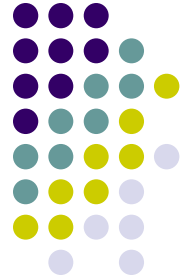
Basic Topics

- Data Structures for File Access
- Disk Layout of Filesystems
- Allocating Storage for Files
- Directory Implementation
- Free-Space Management
- Virtual Filesystems
- Efficiency and Performance
- Recovery

Data Structures for File Access

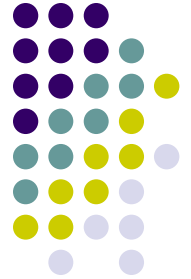


- File Control Block (FCB)
 - One per file
 - Contains file attributes and location of disk blocks of the file
 - Stored in disk, usually brought to memory when file is opened
- Open File Table
 - In-memory table with one entry per open file
 - Each entry points to the FCB of the file (on disk or usually to copy in memory)
 - Can be hierarchical
 - Per-process table with entries pointing to entries in a single system-wide table
 - System-wide table points to FCB of file

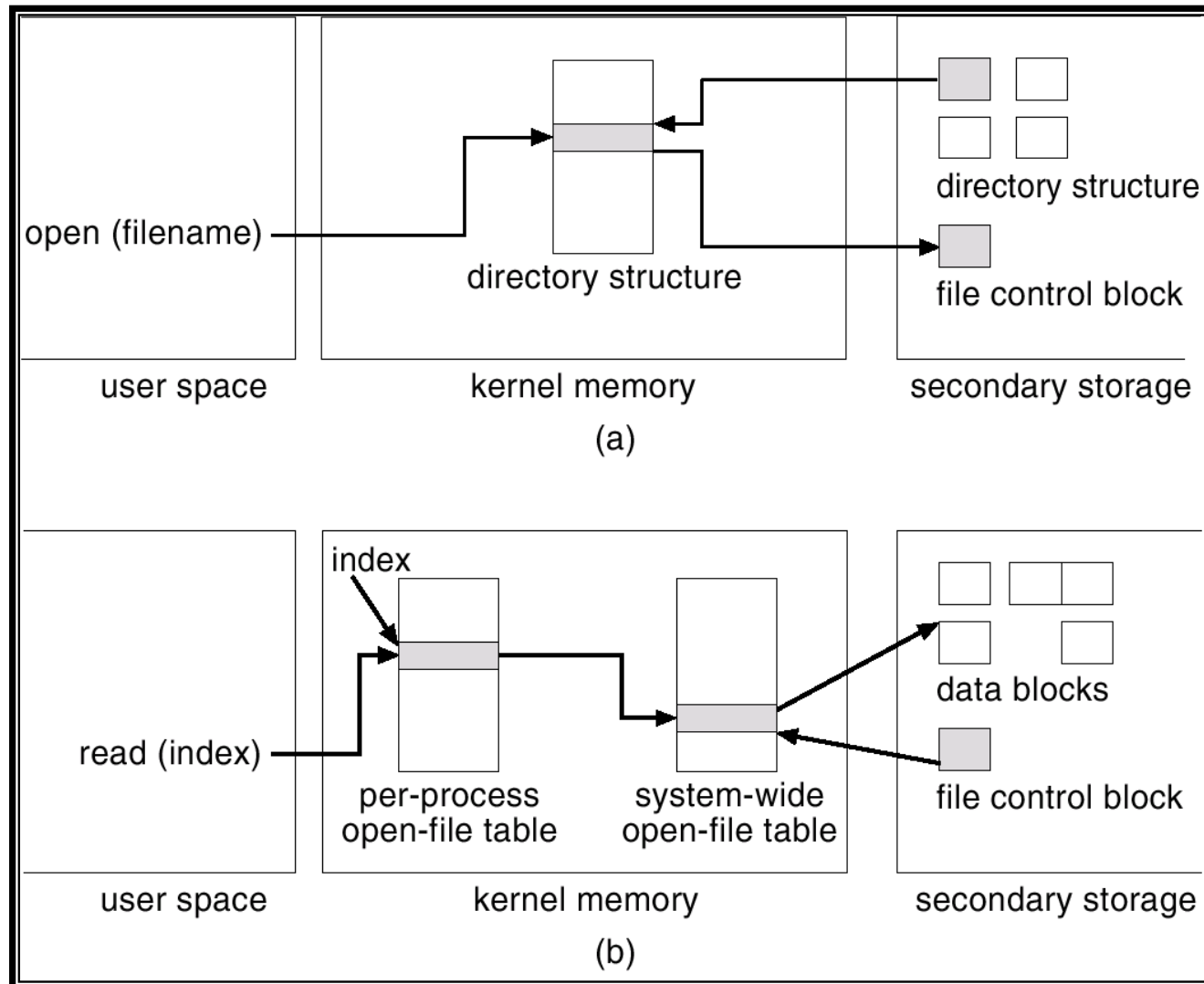


A Typical File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks



In-Memory Open File Tables





Disk Layout

- Files stored on disks. Disks broken up into one or more partitions, with separate fs on each partition
- Sector 0 of disk is the Master Boot Record
- Used to boot the computer
- End of MBR has partition table. Has starting and ending addresses of each partition.
- One of the partitions is marked active in the master boot table

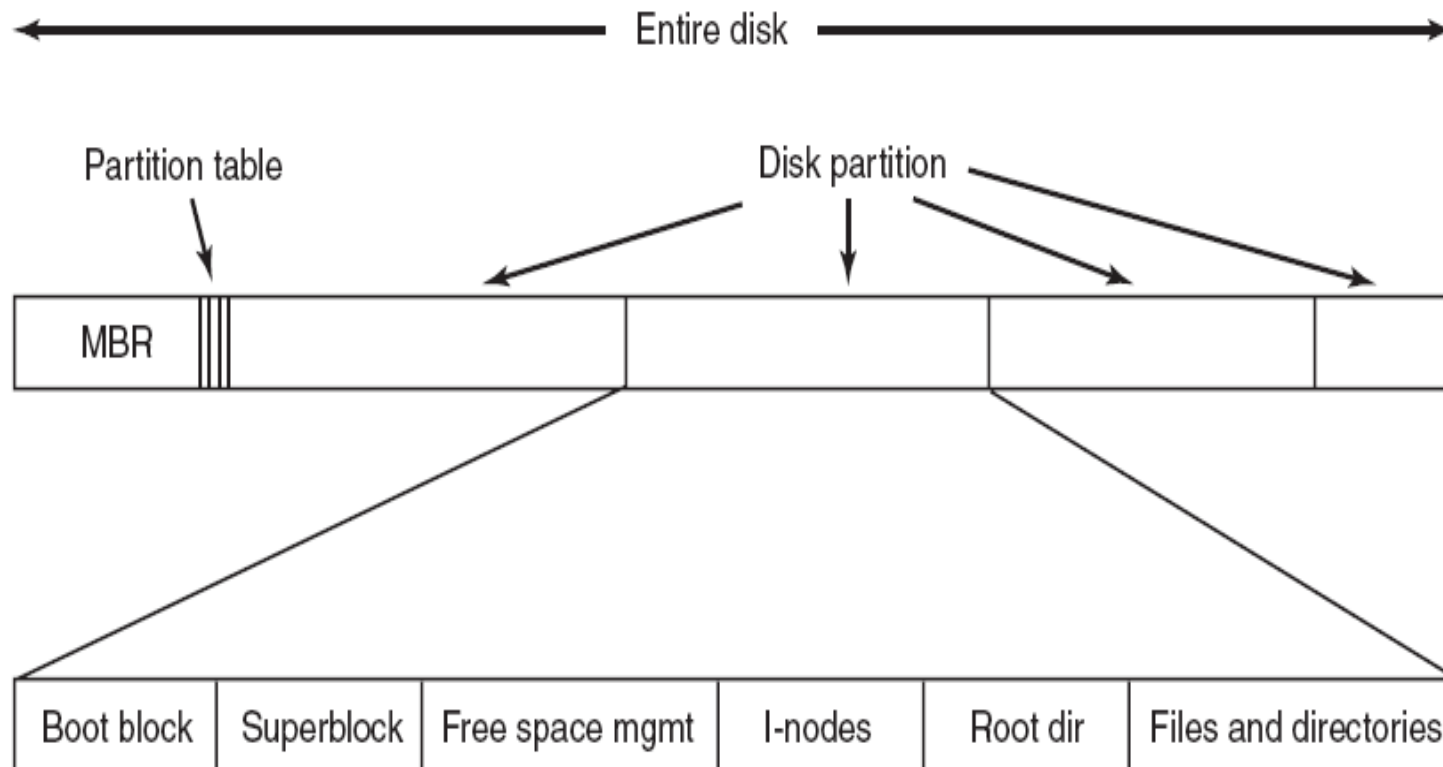


Disk Layout (contd.)

- Boot computer => BIOS reads/executes MBR
- MBR finds active partition and reads in first block (boot block)
- Program in boot block locates the OS for that partition and reads it in
- All partitions start with a boot block

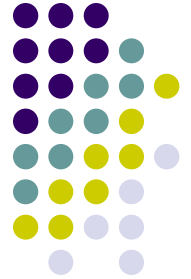


One Possible Example





- Superblock contains info about the fs (e.g. type of fs, number of blocks, ...)
- i-nodes contain info about files
 - Common Unix name for FCB



Allocation Methods

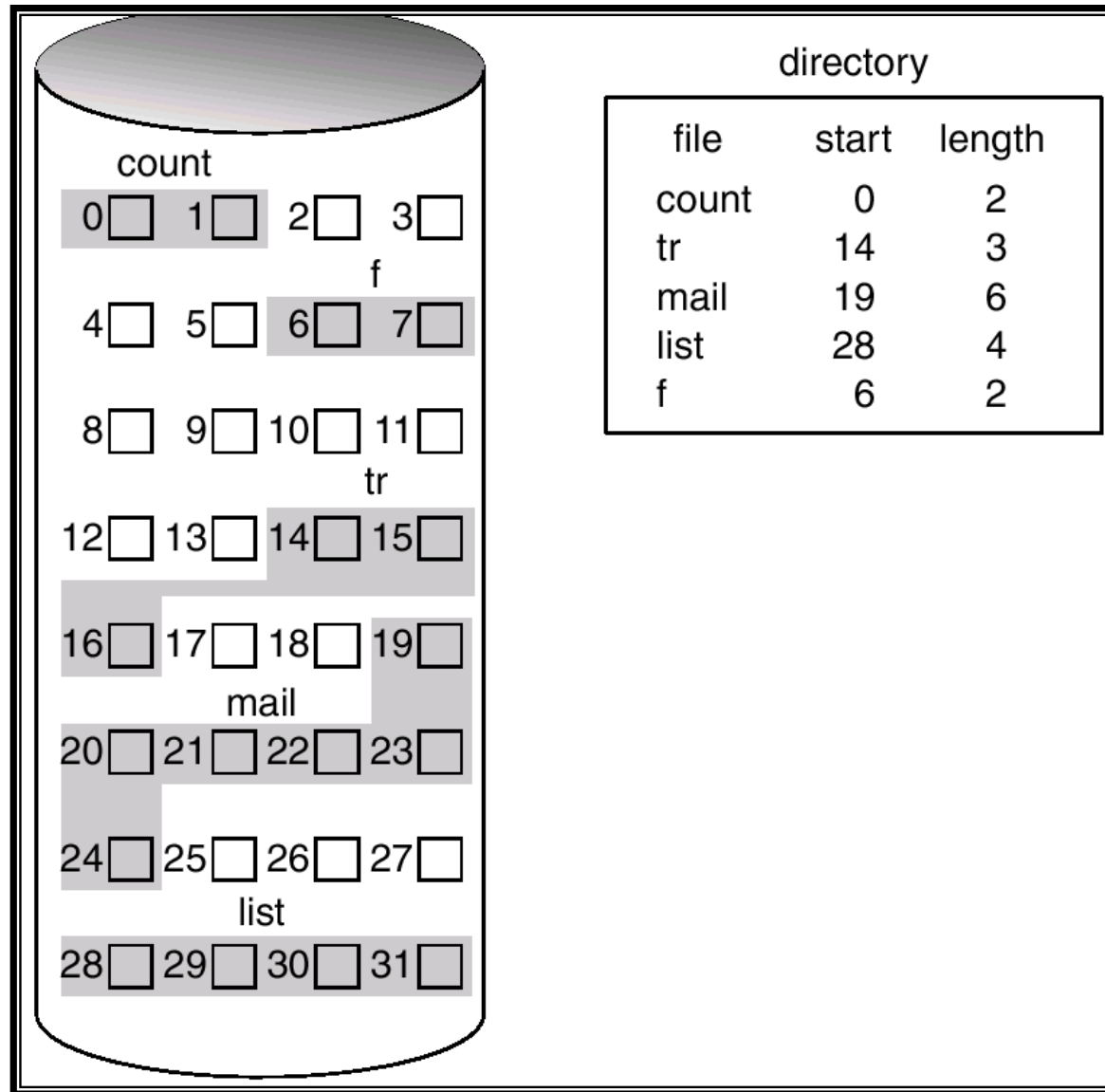
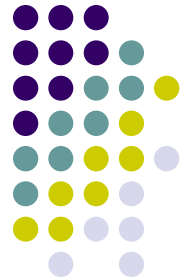
- An allocation method refers to how disk blocks are allocated for files
- Possibilities
 - Contiguous allocation
 - Linked allocation
 - Indexed allocation

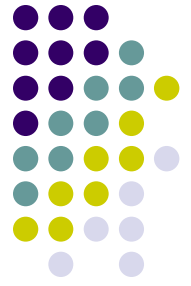


Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Easy to implement – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
 - Fragmentation possible
- Files cannot grow

Contiguous Allocation of Disk Space





Extent-Based Systems

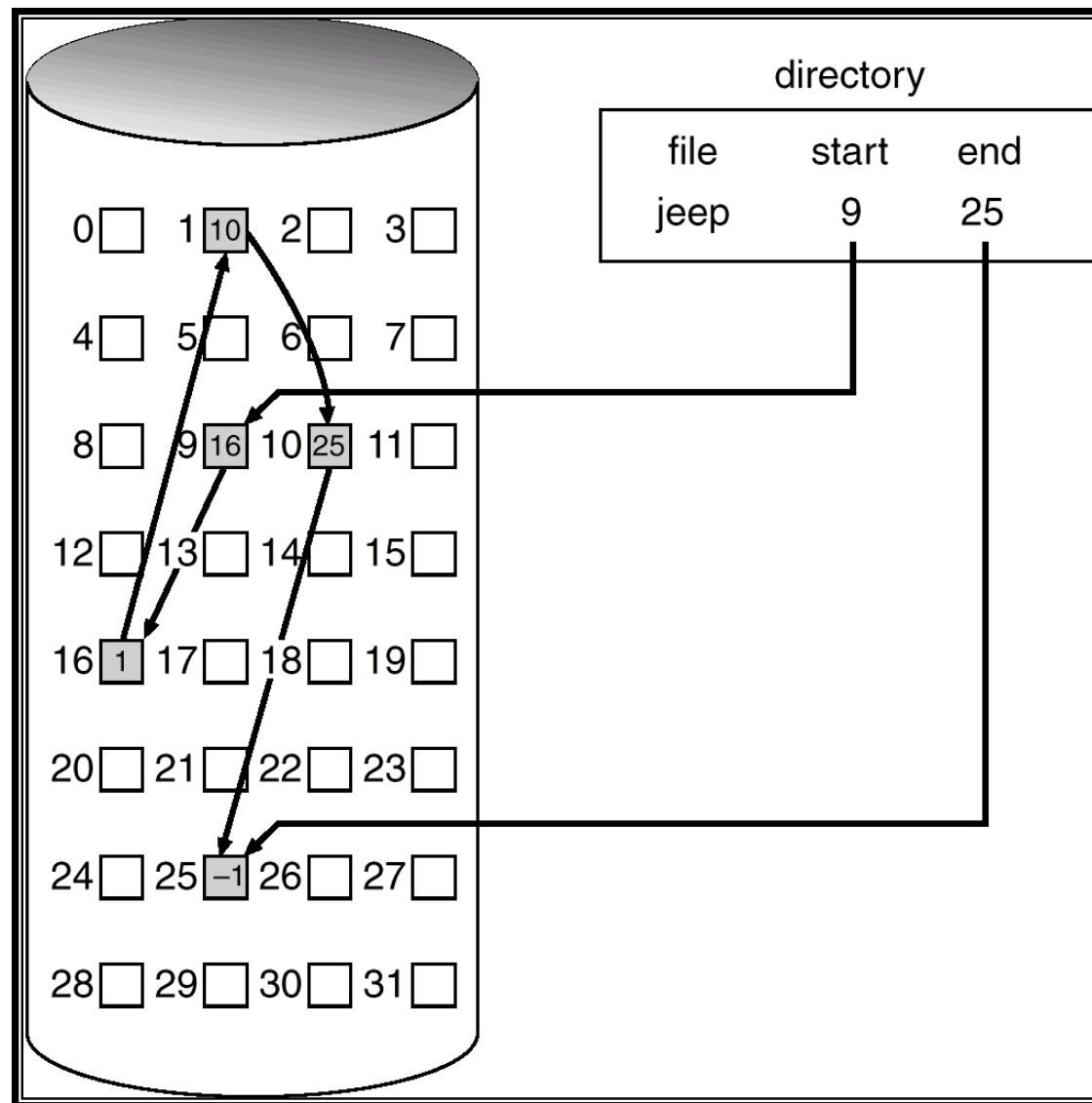
- Many newer file systems use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in **extents**
- An **extent** is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents



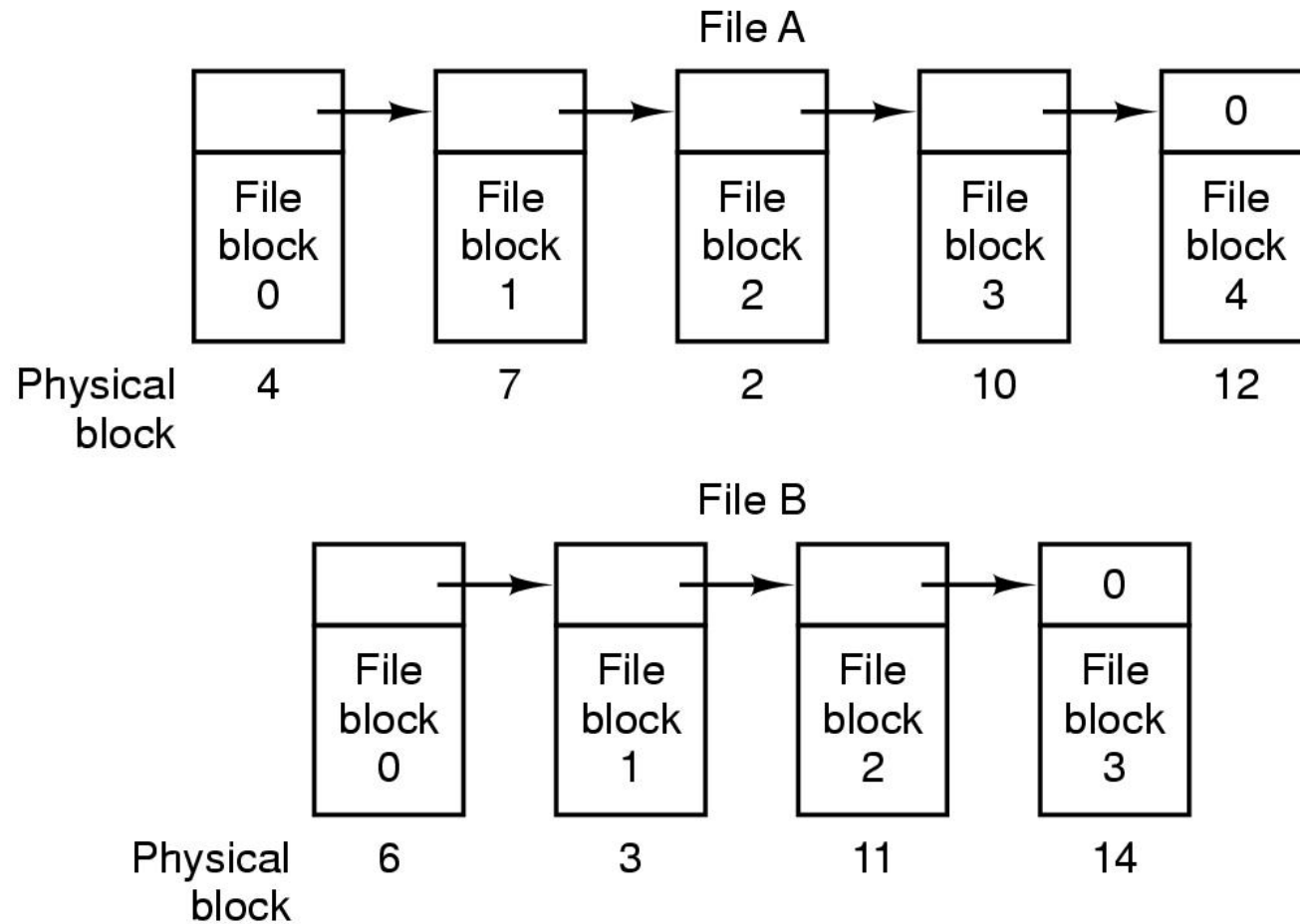
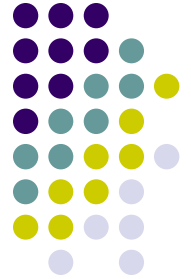
Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Simple – need only starting address
- Free-space management system – no waste of space
- No random access

Linked Allocation



Linked List Allocation

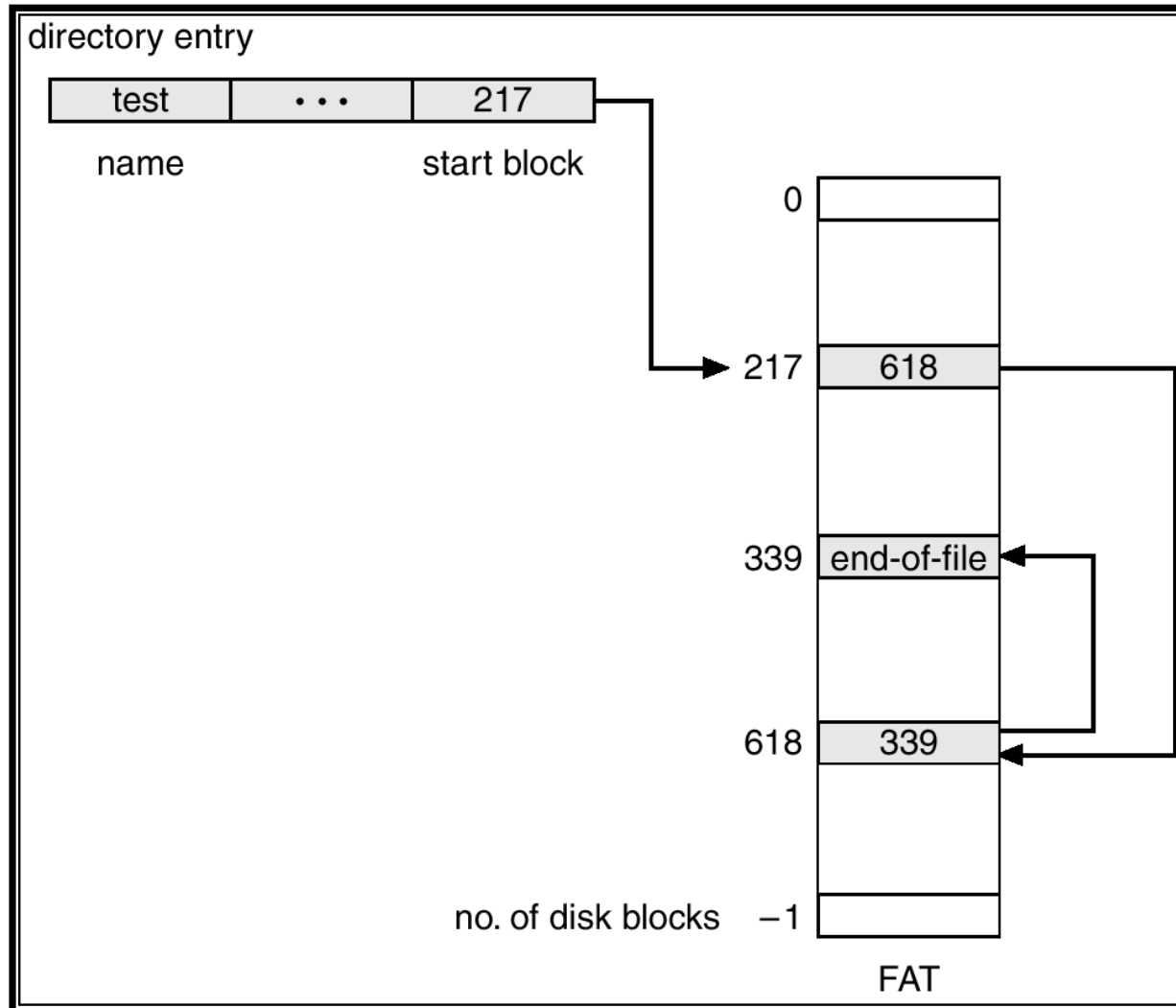
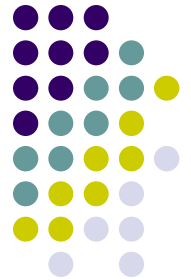


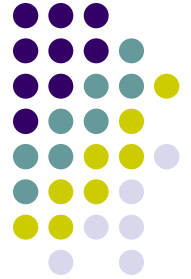
Linked lists using a table in memory



- Put pointers in table in memory
- File Allocation Table (FAT)
- Still have to traverse pointers, but now in memory
- But table becomes really big
 - 200 GB disk with 1 KB blocks needs a 600 MB table
 - Growth of the table size is linear with the growth of the disk size

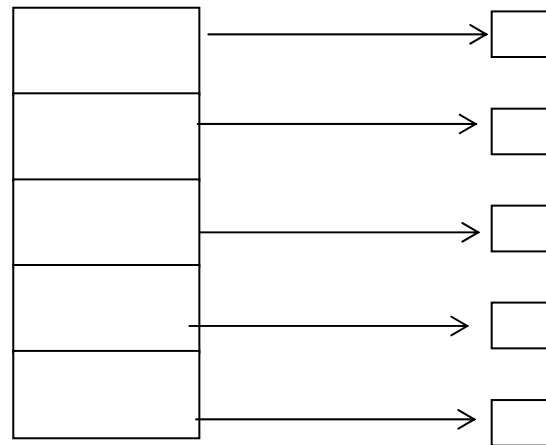
File-Allocation Table





Indexed Allocation

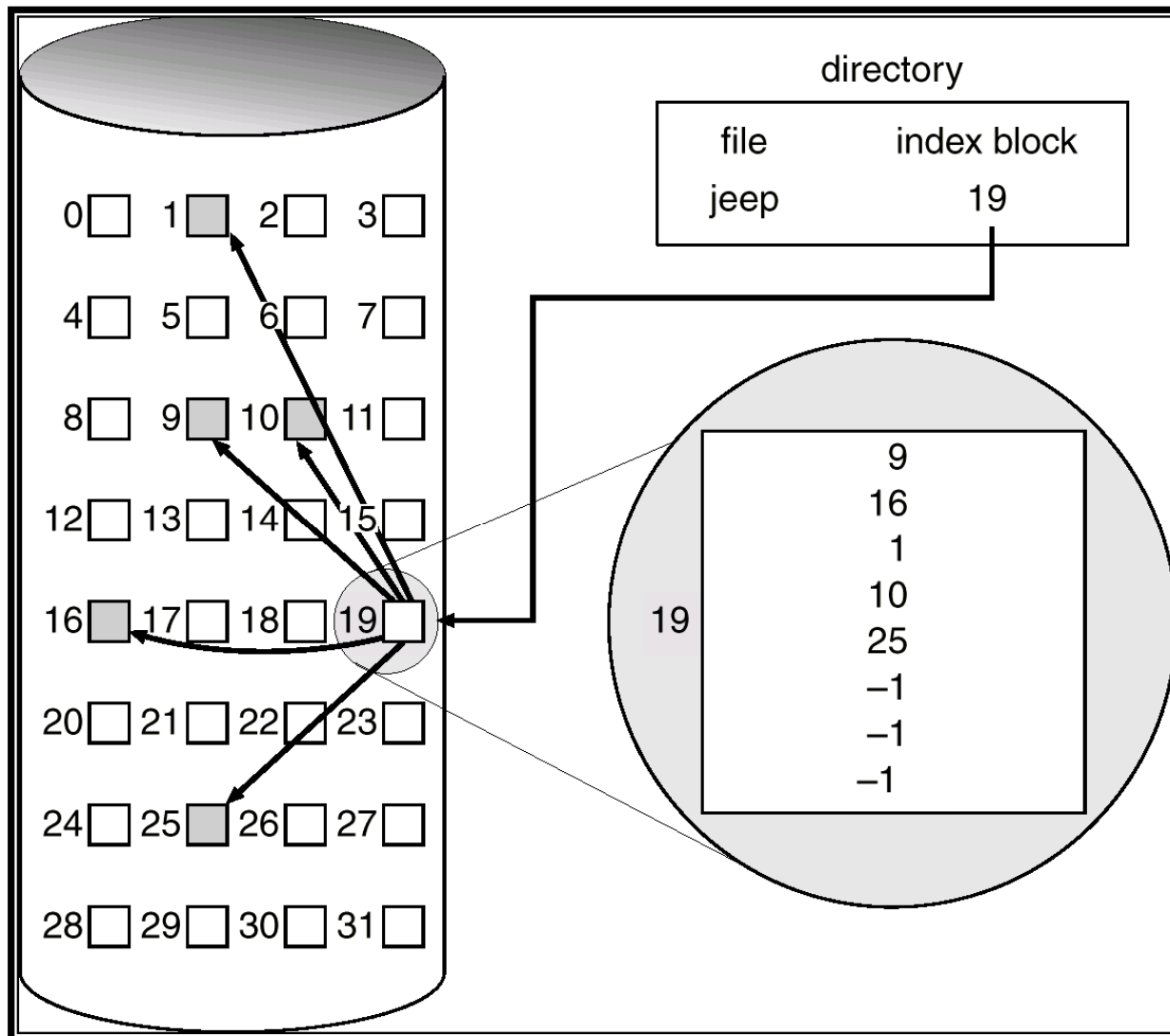
- Brings all pointers together into the **index block**
- Logical view



index table



Example of Indexed Allocation





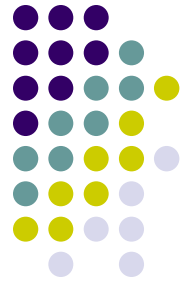
Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table

Indexed Allocation – Mapping (Cont.)

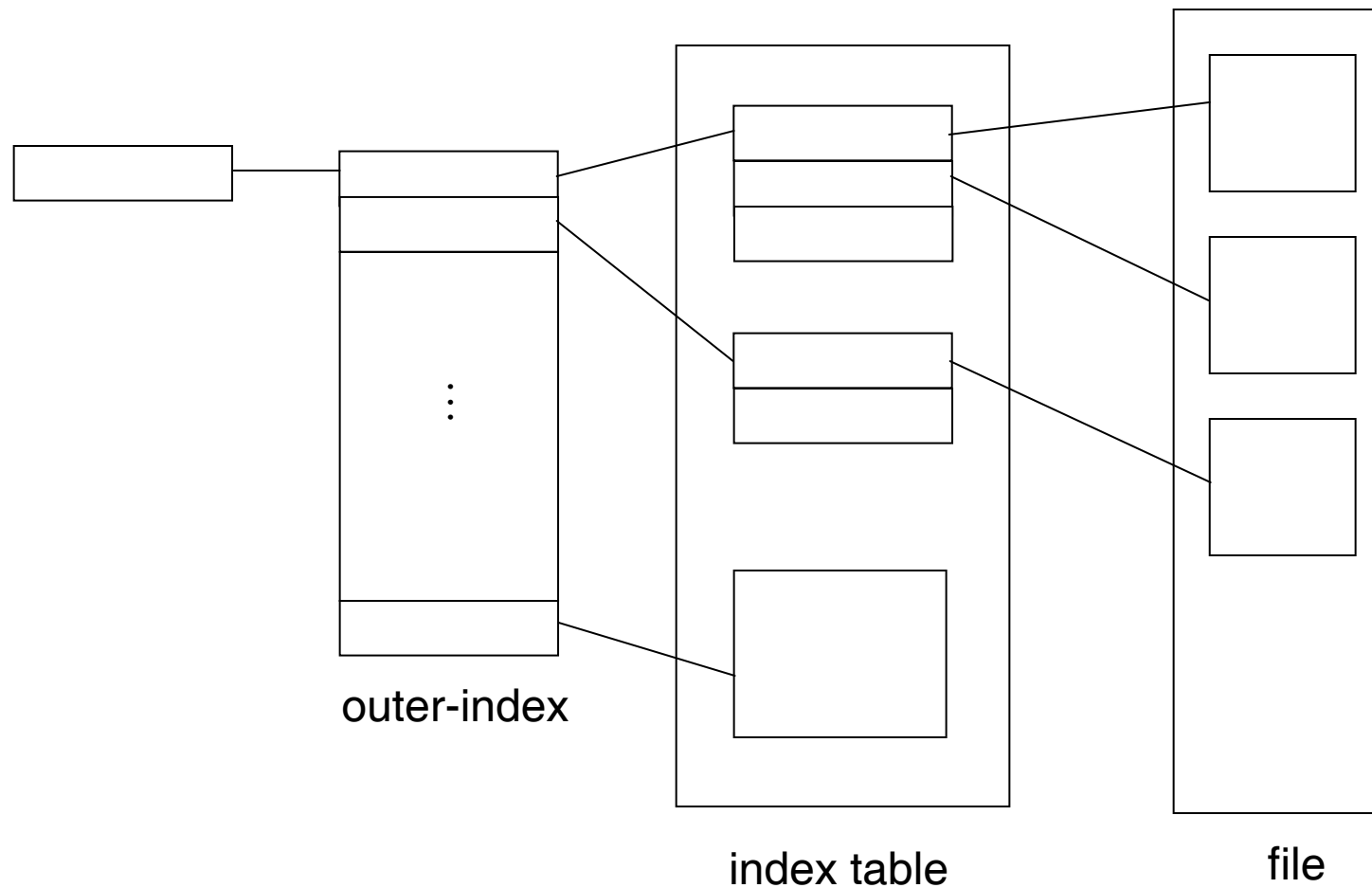


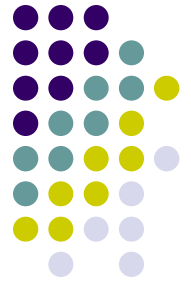
- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- Linked scheme – Link blocks of index table (no limit on size)



Two-level Indexing

- Two-level index (maximum file size is 512^3)

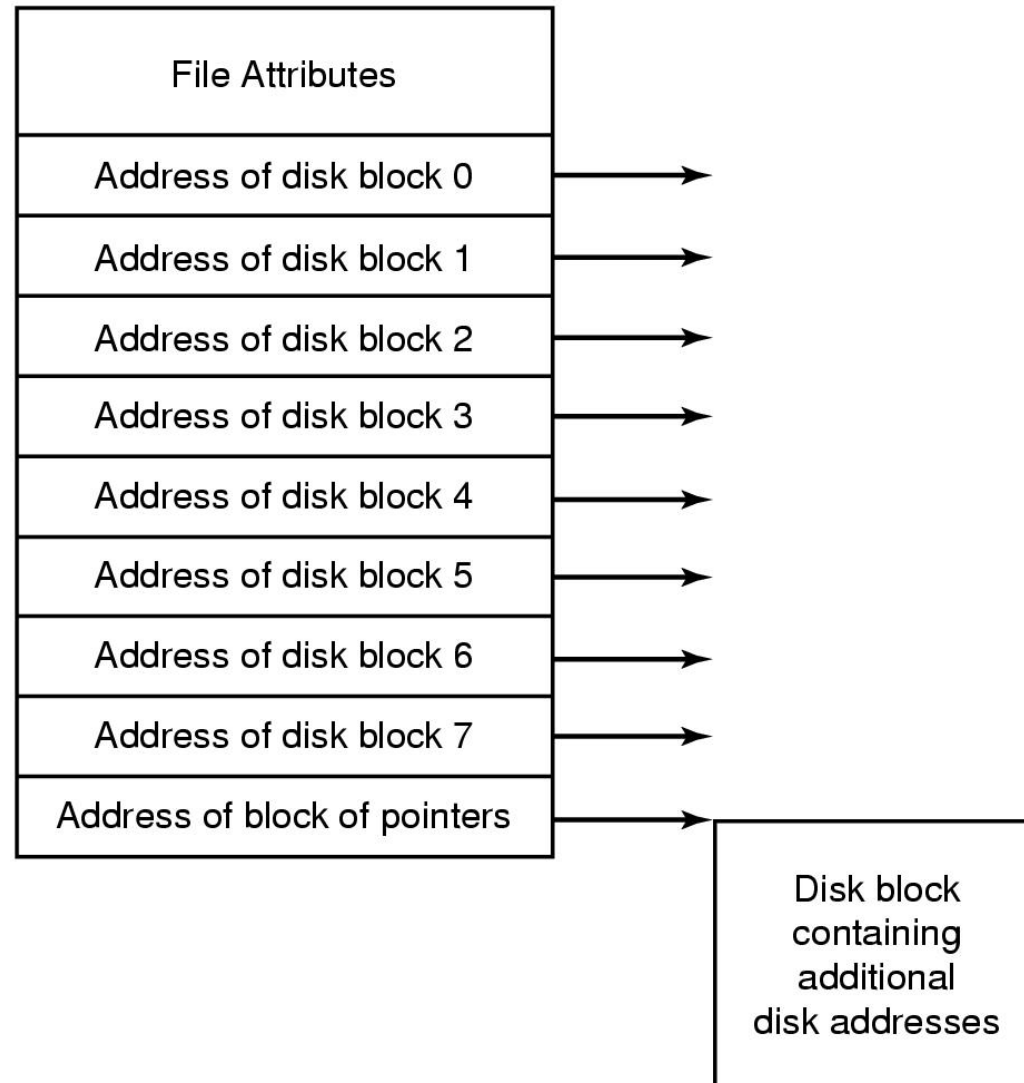
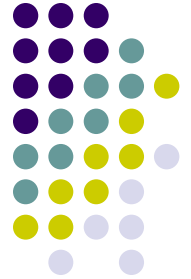




i-nodes

- FCB in Unix
- Contains file attributes and disk address of blocks
- One block can hold only limited number of disk block addresses, limits size of file
- Solution: use some of the blocks to hold address of blocks holding address of disk blocks of files
 - Can take this to more than one level

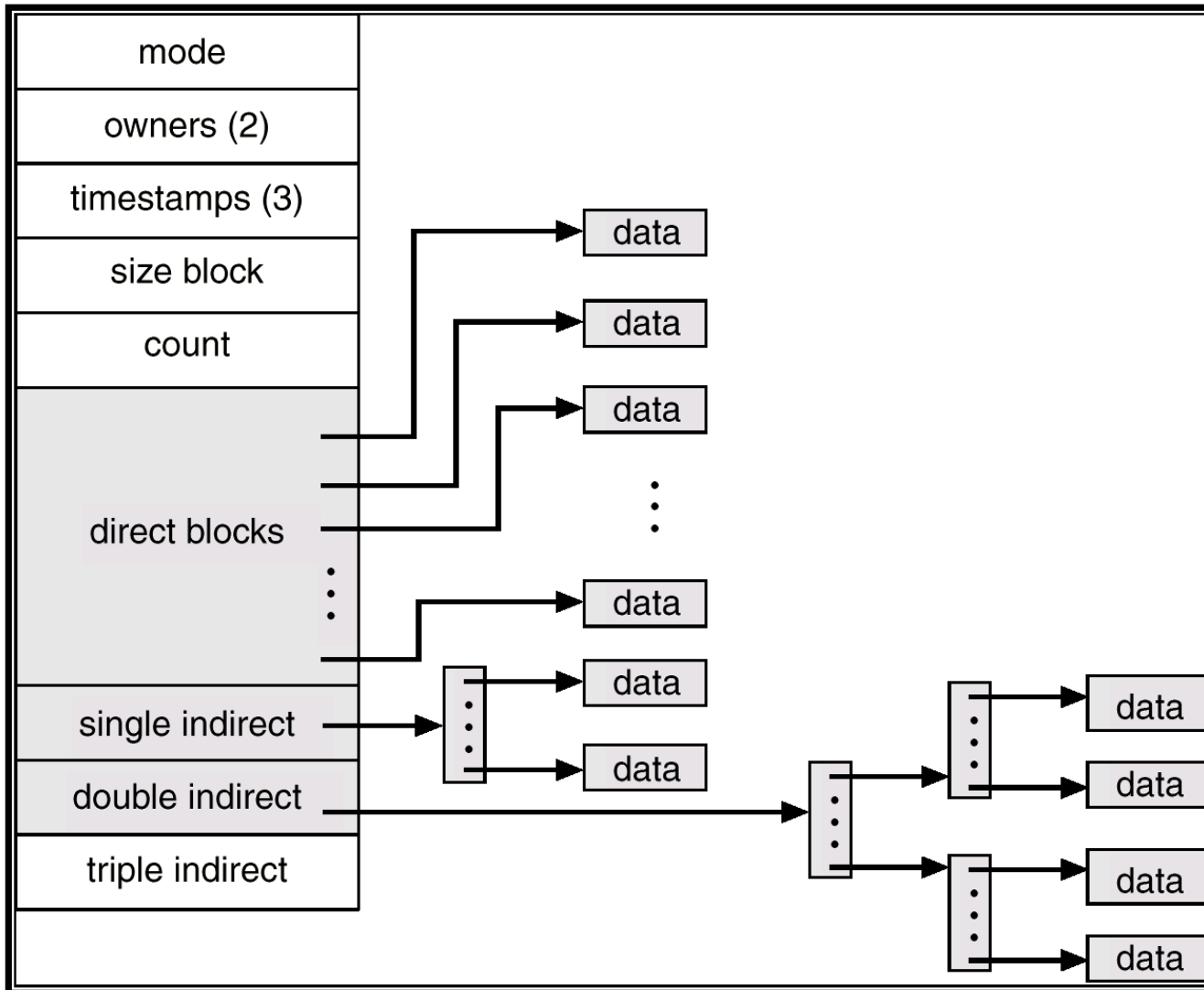
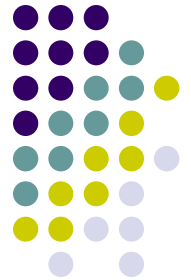
i-node with one-level indirection





Unix i-node

- File Attributes
- 12 direct pointers
- 1 singly indirect pointer
 - Points to a block that has disk block addresses
- 1 doubly indirect pointer
 - Points to a block that points to blocks that have disk block addresses
- 1 triply indirect pointer
 - Points to a block that points to blocks that point to blocks that have disk block addresses
- What is the max. file size possible??





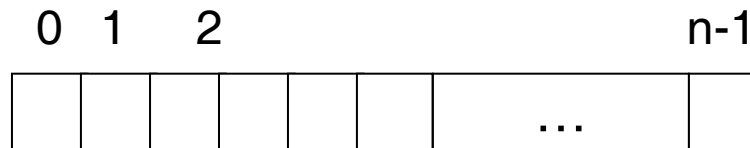
Directory Implementation

- Linear list of file names with pointer to the data blocks
 - Address of first block (contiguous)
 - Number of first block (linked)
 - Number of i-node
- simple to program
- time-consuming to execute
- Hash Table – linear list with hash data structure
 - decreases directory search time
 - *collisions* – situations where two file names hash to the same location
 - fixed size



Free-Space Management

- Bit vector (n blocks)



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation for first free block

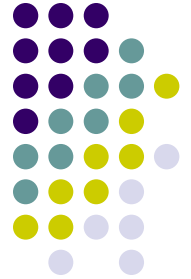
(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

Free-Space Management (Cont.)



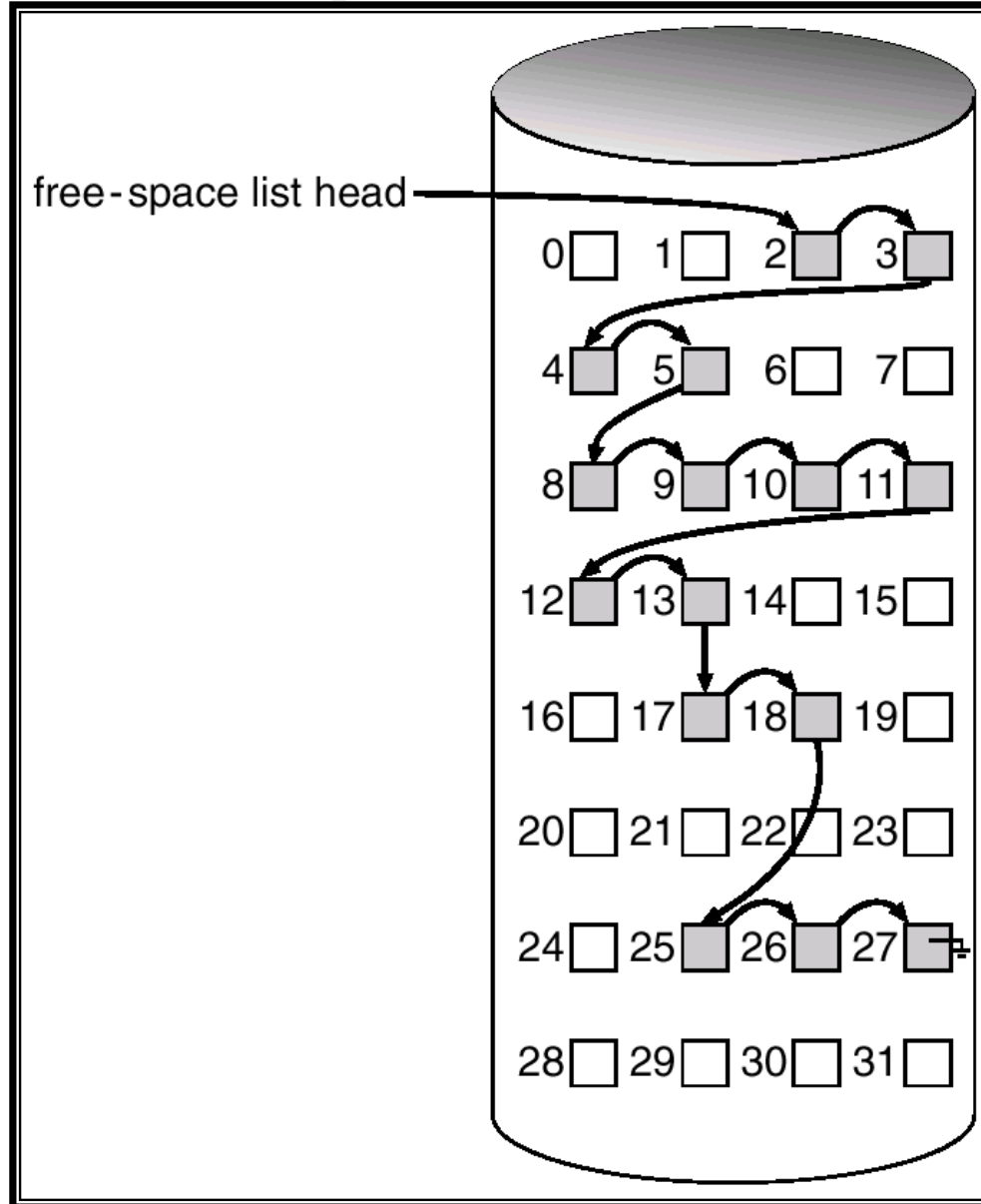
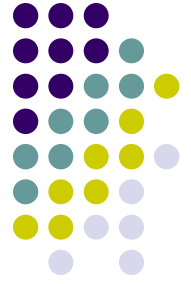
- Bit map requires extra space. Example:
 - block size = 2^{12} bytes
 - disk size = 2^{30} bytes (1 gigabyte)
 - $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files
- Linked list (free list)
 - Cannot get contiguous space easily
 - No waste of space
 - May need no. of disk accesses to find a free block
 - Grouping
 - Counting

Free-Space Management (Cont.)

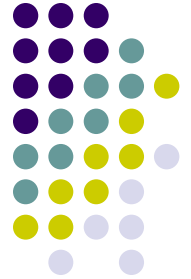


- Need to protect:
 - Pointer to free list
 - Bit map
 - Must be kept on disk
 - Copy in memory and disk may differ.

Linked Free Space List on Disk

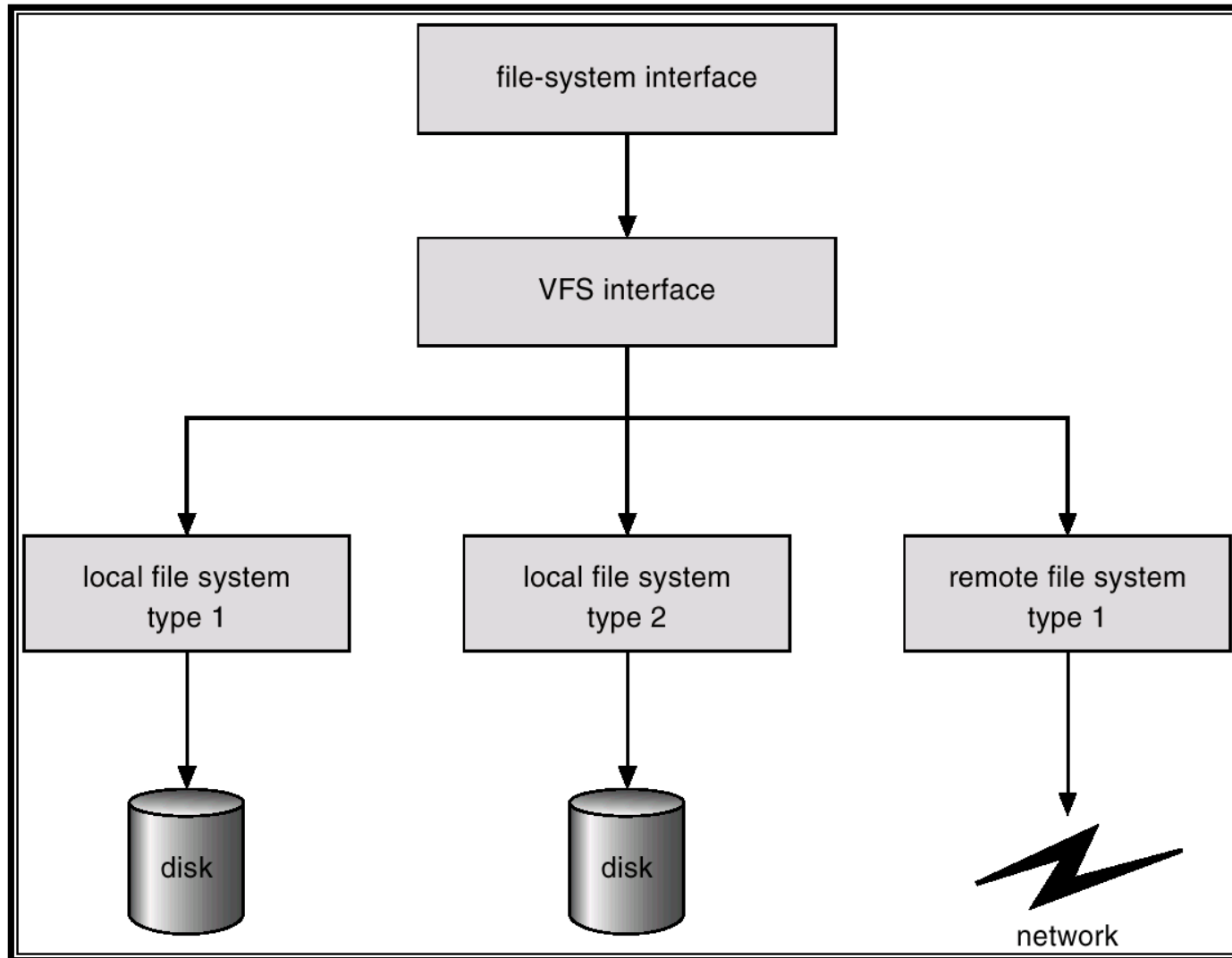
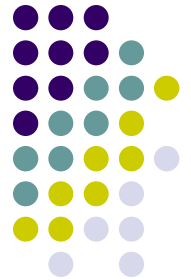


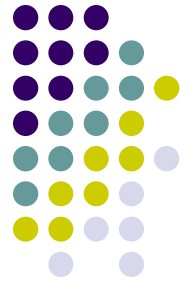
Virtual File Systems



- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

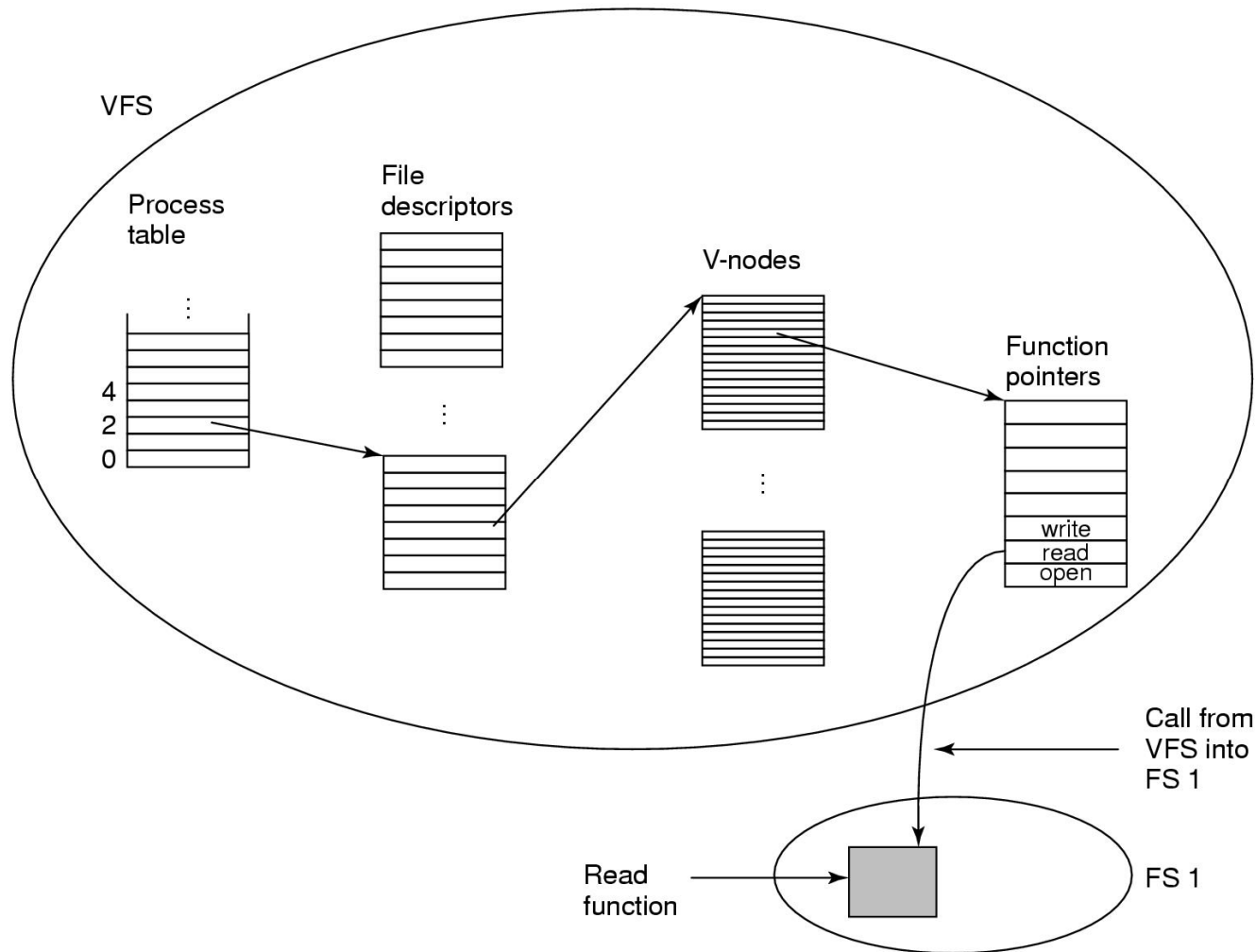
Schematic View of Virtual File System





How VFS works

- File system registers with VFS (e.g. at boot time)
- At registration time, fs provides list of addresses of function calls the vfs wants
- Vfs gets info from the new fs i-node and puts it in a v-node
- Makes entry in fd table for process
- When process issues a call (e.g. read), function pointers point to concrete function calls



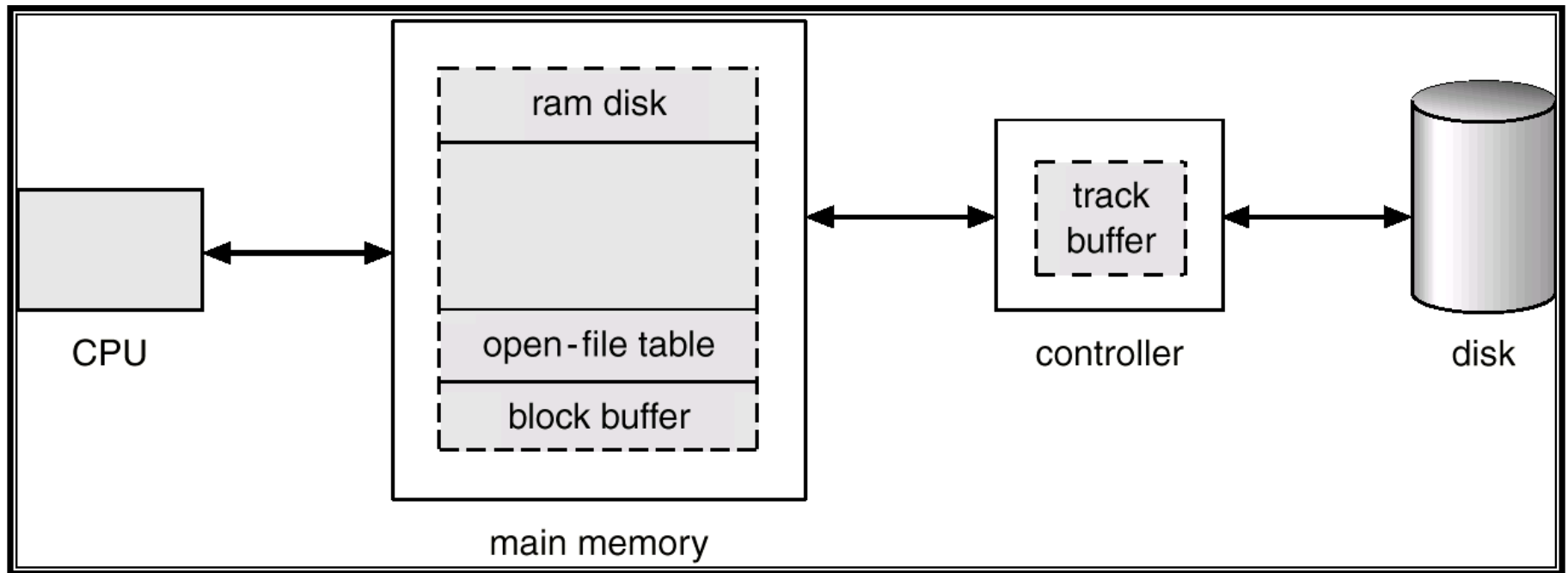
- . A simplified view of the data structures and code used by the VFS and concrete file system to do a read.



Efficiency and Performance

- Efficiency dependent on:
 - disk allocation and directory algorithms
 - types of data kept in file's directory entry
- Performance
 - disk cache – separate section of main memory for frequently used blocks
 - free-behind and read-ahead – techniques to optimize sequential access
 - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.

Various Disk-Caching Locations

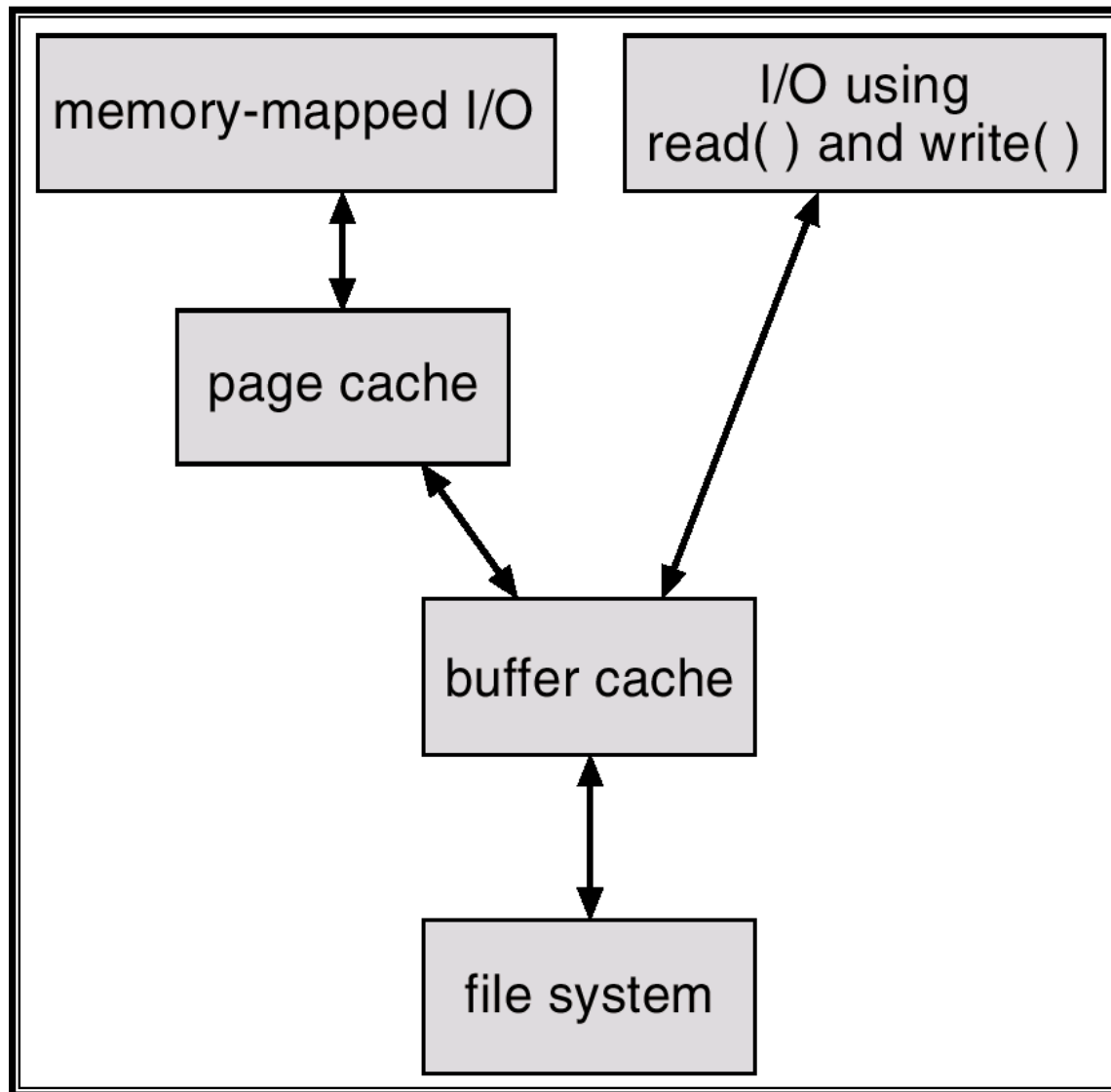
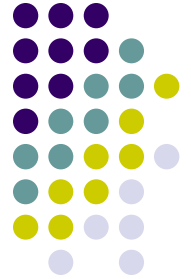




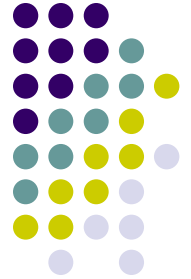
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache

I/O Without a Unified Buffer Cache

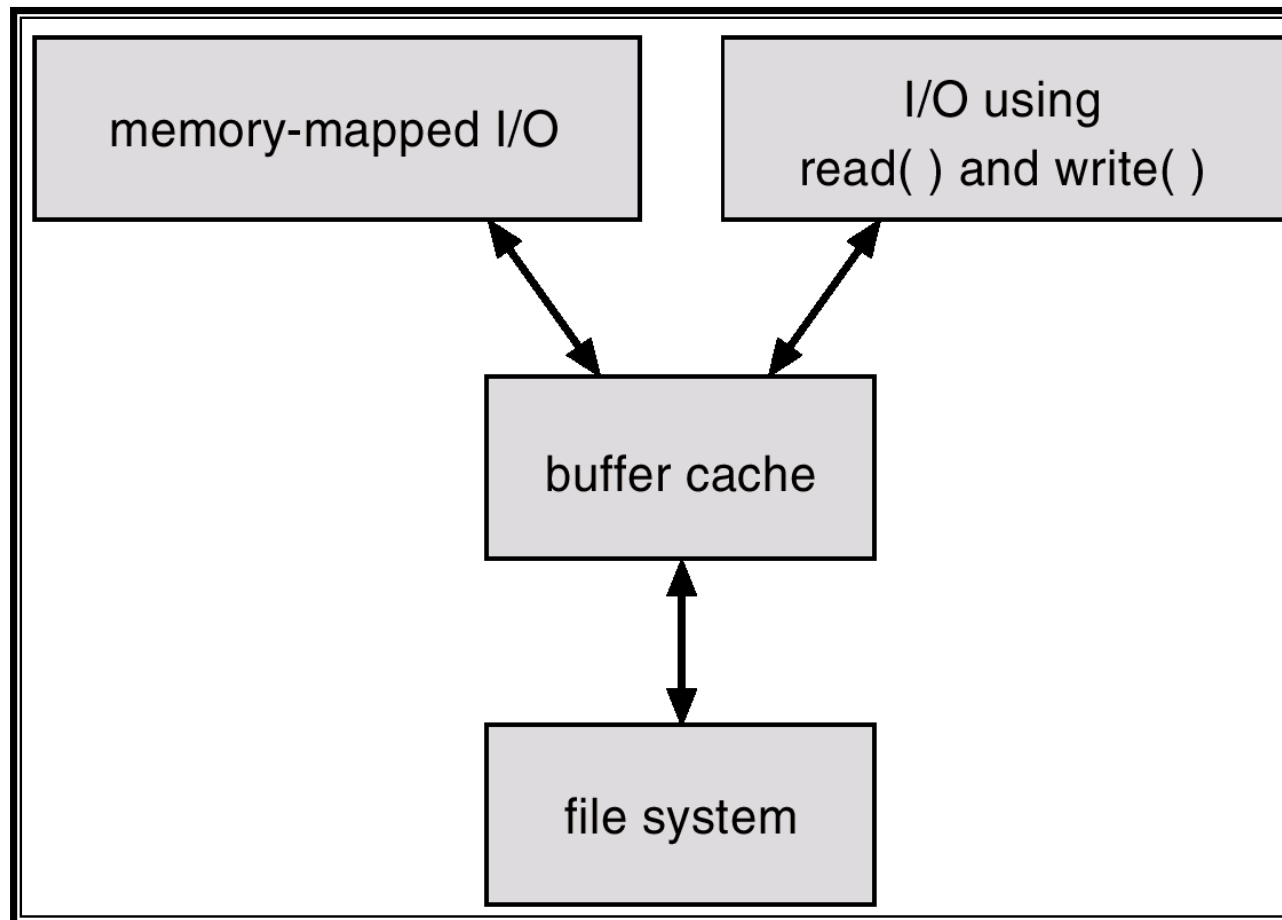


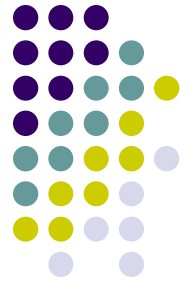
Unified Buffer Cache



- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.

I/O Using a Unified Buffer Cache





Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Use system programs to *back up* data from disk to another storage device (floppy disk, magnetic tape)
- Recover lost file or disk by *restoring* data from backup