

# *Operating Systems*

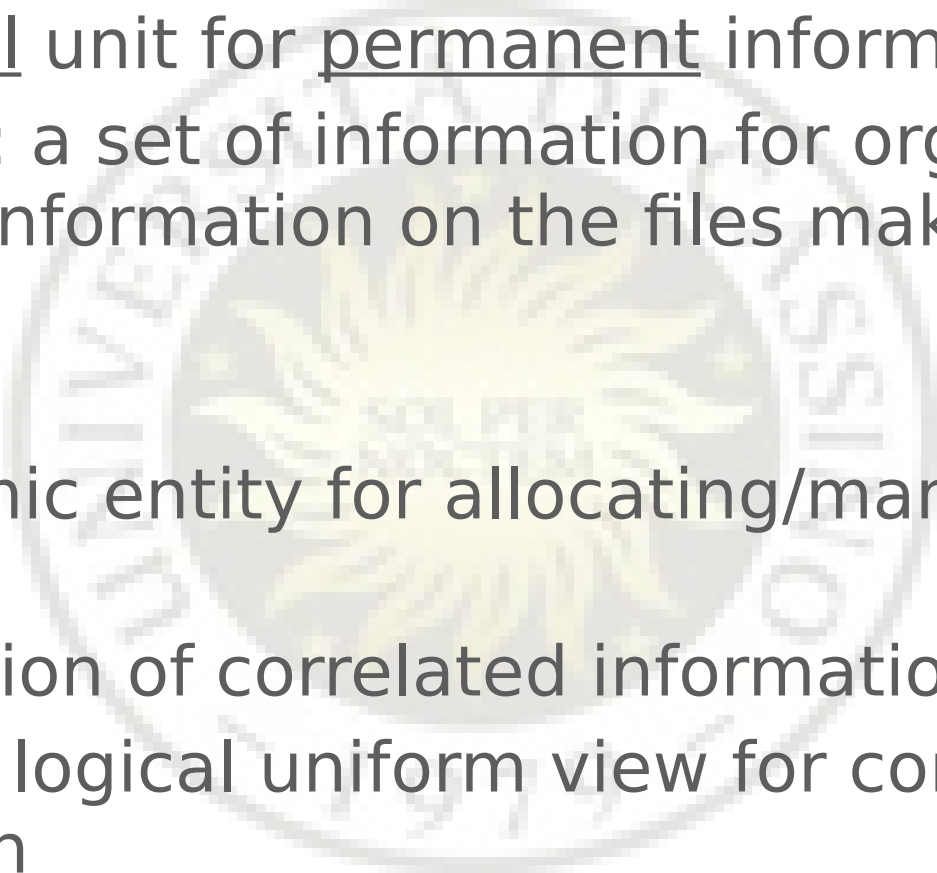
## **File system**

Spring 2016

Francesco Fontanella

# Introduction

- Information can be permanently stored on different media  
**Examples:** hard disks, optical disks, flash drive, etc.
- Each of these media has different physical characteristics
- **file systems** provide an abstract interface which is:
  - ♦ common (to all media)
  - ♦ efficient
  - ♦ easy to use (for users and programmers)

- 
- From the user point of view, a file system consists of:
    - ♦ **file**: logical unit for permanent information storing
    - ♦ **directory**: a set of information for organizing and providing information on the files making up a file system
  - **File concept**
    - ♦ Is the atomic entity for allocating/managing peripheral memory
    - ♦ Is a collection of correlated information
    - ♦ provides a logical uniform view for correlated information

# File attributes

## ■ name:

- Character string allowing users (and OS too) to identify each file in the file system
- Some systems are case sensitive (Ex: UNIX, Linux)

## ■ type:

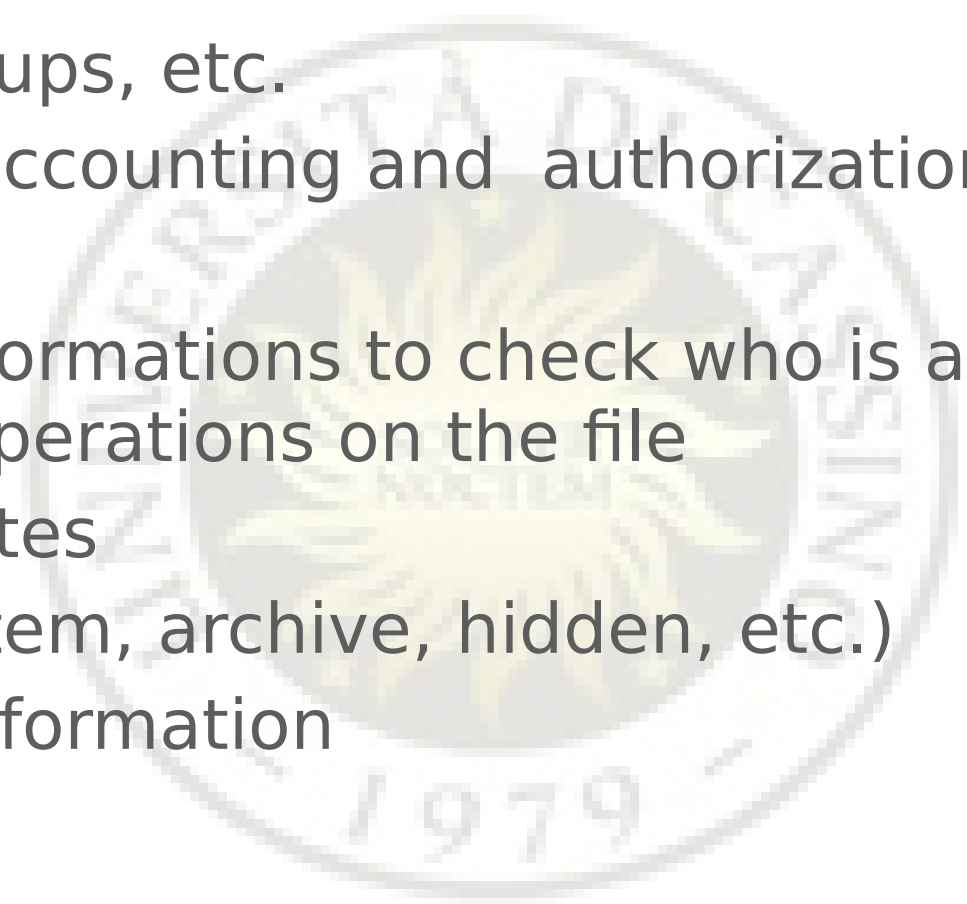
- Specifies the file type. It is necessary in some OS

## ■ Location and size

- Where the file actually is in the peripheral memory and how many bytes it occupies

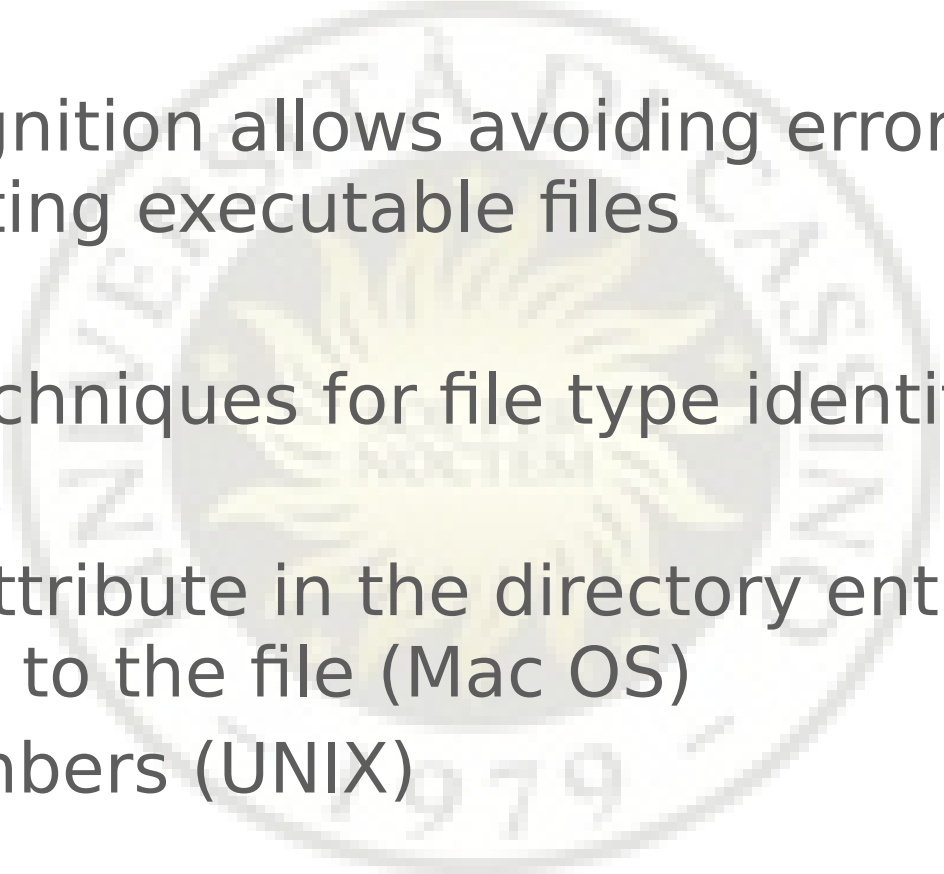
## ■ Date and time:

- Information about file creation and last modification date

- 
- ownership
    - ♦ users, groups, etc.
    - ♦ Used for accounting and authorization
  - protection:
    - ♦ Access informations to check who is authorized to perform operations on the file
  - Other attributes
    - ♦ flags (system, archive, hidden, etc.)
    - ♦ Locking information
    - ♦ etc.

# File types

- According to file internal structure
  - ♦ without format (byte string): text file
  - ♦ with format: file of records, database, a.out,...
- According to the file content
  - ♦ ASCII/binary (displayable or not, 7/8 bit)
  - ♦ source code, object code, ...
  - ♦ executable (active object)

- 
- Some OS support (and recognize) different types of files
  - File type recognition allows avoiding errors like, for example, printing executable files
  - Three main techniques for file type identification:
    - ♦ extensions
    - ♦ A “type” attribute in the directory entry associated to the file (Mac OS)
    - ♦ magic numbers (UNIX)

## ■ **MS-DOS:**

- File name 8+3 (name + extension)
- Extension recognition: .COM, .EXE, .BAT

## ■ **Windows 9x / NT**

- Variable size for file names and extensions
- Extension recognition .COM, .EXE, .BAT
- Extension-program association

## ■ **Mac OS**

- The program which created the file is a file attribute together with the type attribute

## ■ **Unix/Linux**

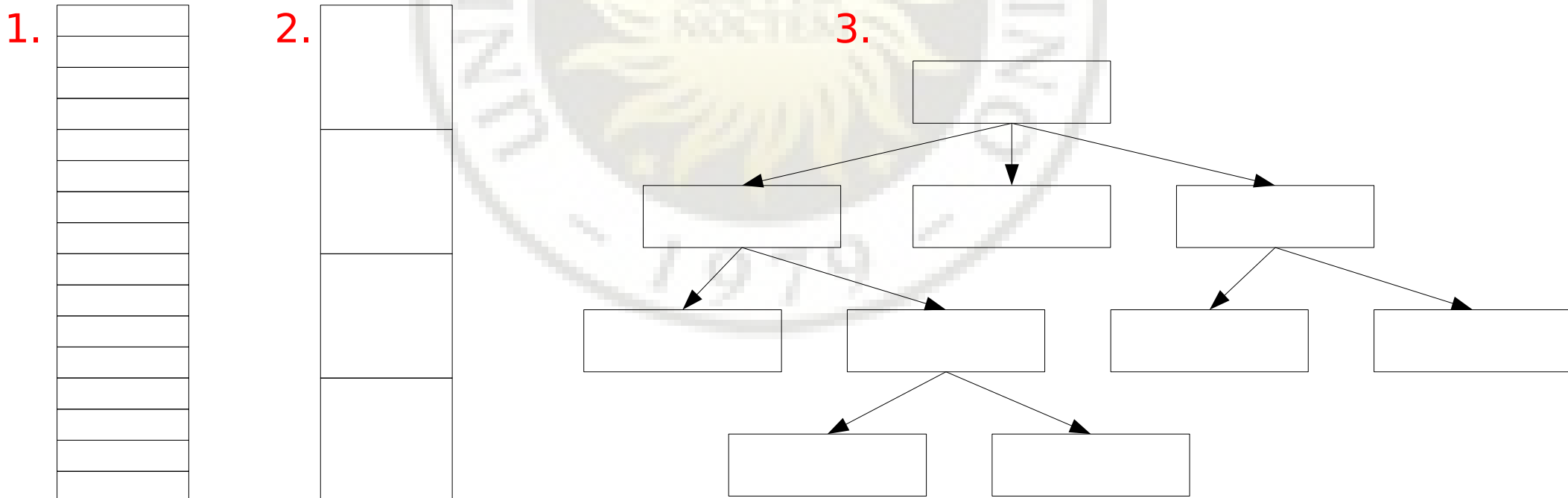
- magic number + extension + heuristic (UNIX).

# File types: further distinctions

- Usually, file systems distinguish among:
  - **regular files**
  - **directories**
    - Are system files for managing file organization
  - **special block files**
    - to model I/O devices like hard disks
  - **special character files**
    - ♦ to model serial I/O devices like terminals, printers, etc
  - **other special files**
    - ♦ Example: pipes

■ Files can be structured in several different ways:

- 1) Byte sequences
- 2) Logical record sequences
- 3) Indexed files (tree structured)



# File structure

- different choices are available for file structures:
  - Minimal choice:
    - Files are managed as simple byte sequences, except the executable files which format is prescribed by the OS
    - e.g., UNIX and MS-DOS
  - ♦ structured part/ user defined part
    - e.g. Macintosh (resource fork / data fork)
  - Different types of predefined files
    - e.g., VMS, MVS

# Structure file support

## ■ trade-off:

- More formats
  - More complex system
  - program incompatibility (accessing files of different formats)
  - Efficient management (and not duplications) for special files
- less formats
  - Simpler system code

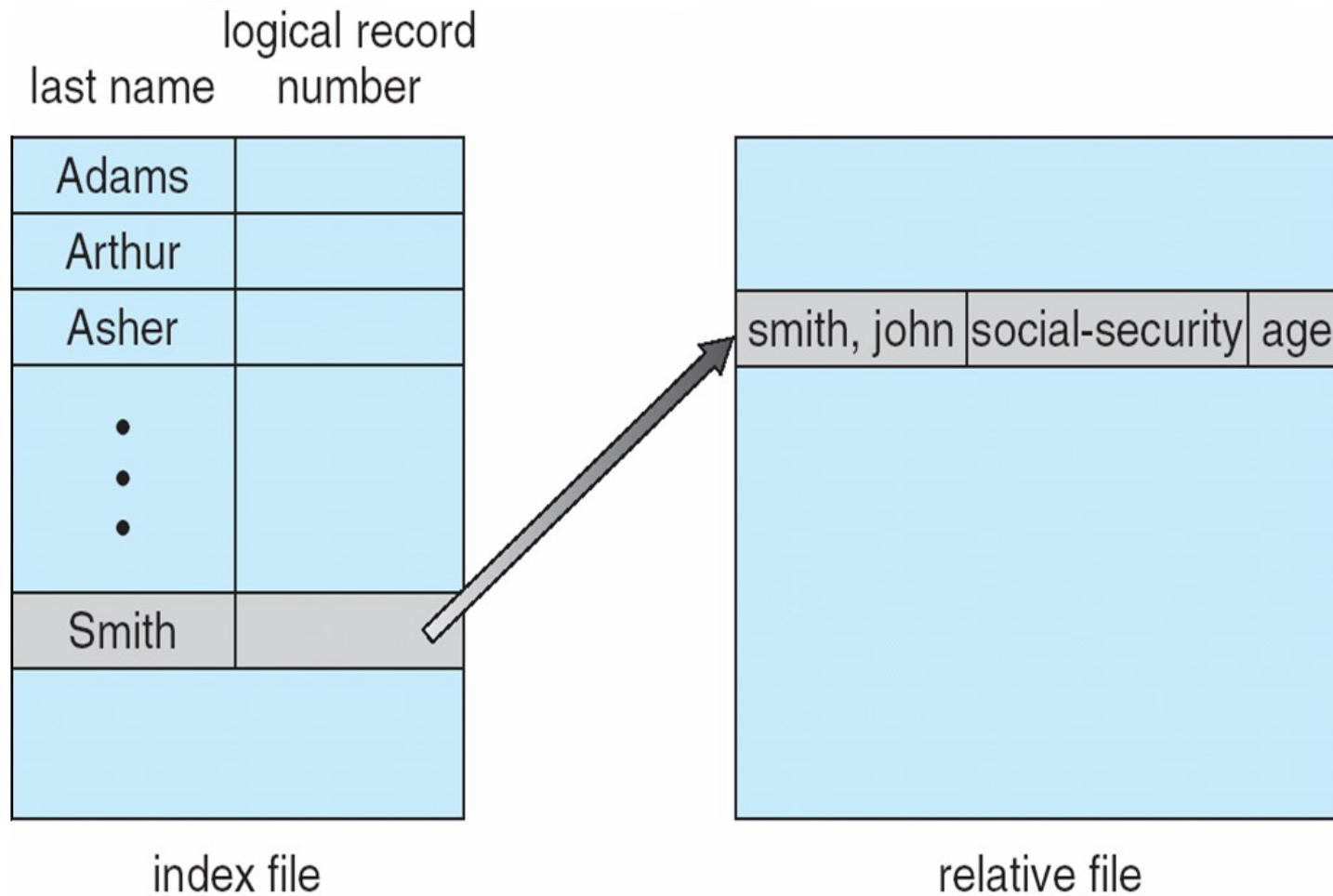
# Access modes

- Sequential
  - ♦ read, write
  - ♦ tapes
- Random access
  - ♦ *read pos, write pos* (or *seek operation*)
  - ♦ disks
- indexed
  - ♦ read *key*, write *key*
  - ♦ database

# Indexed access mode

- There is a table whose entries contain the key-position (location) correspondences
- Once the file is open, the index table can be loaded in main memory:
  - ♦ allows a more efficient access
  - ♦ costly in terms of memory

# Example

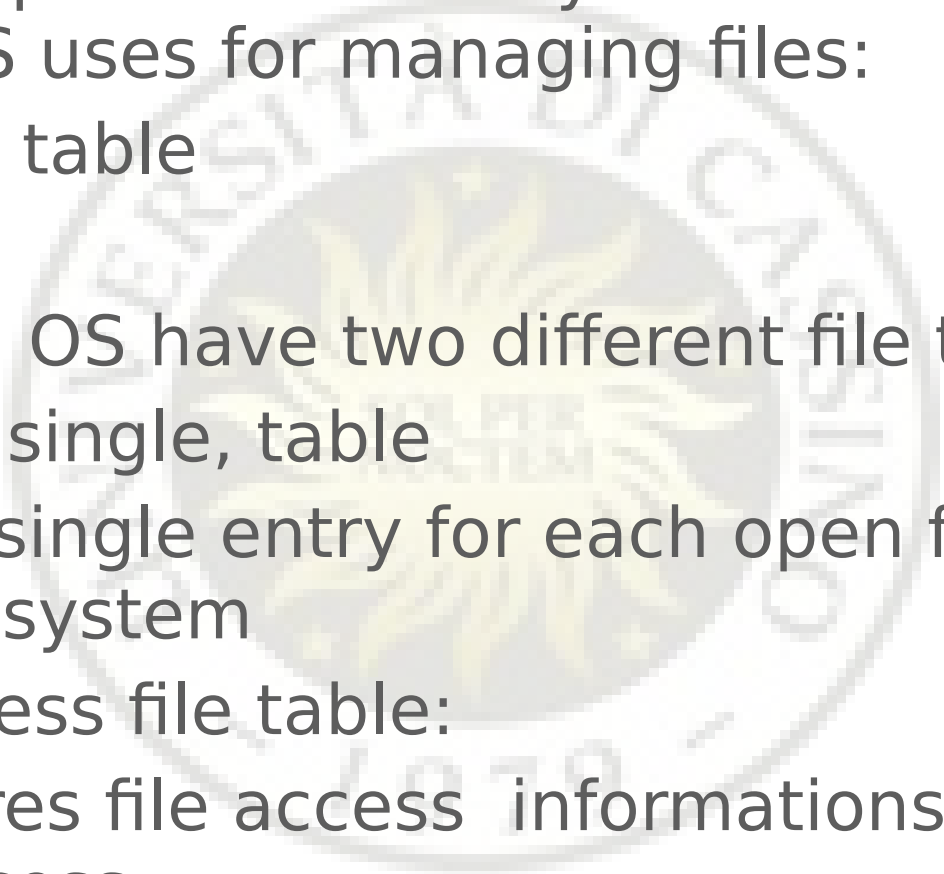


# File operations

- File creation (space allocation directory update)
- Open/close
- Read/write/append (file pointer management)
- Pointer positioning
- Cancellation
- Truncation
- Reading/writing the file attributes (Example: `ls`, `touch`, etc)

# Open/close operations

- The API for file operations is based on the open/close operations:
  - ♦ A file must be opened before it can be managed, and closed at the end of the managing
- The abstraction related to the open/close operations are useful for
  - ♦ Managing file access data structures (e.g. FILE Table)
  - ♦ Checking the access modes
  - ♦ Managing concurrent accesses
  - ♦ Defining a descriptor (i.e. file pointer) for access operations

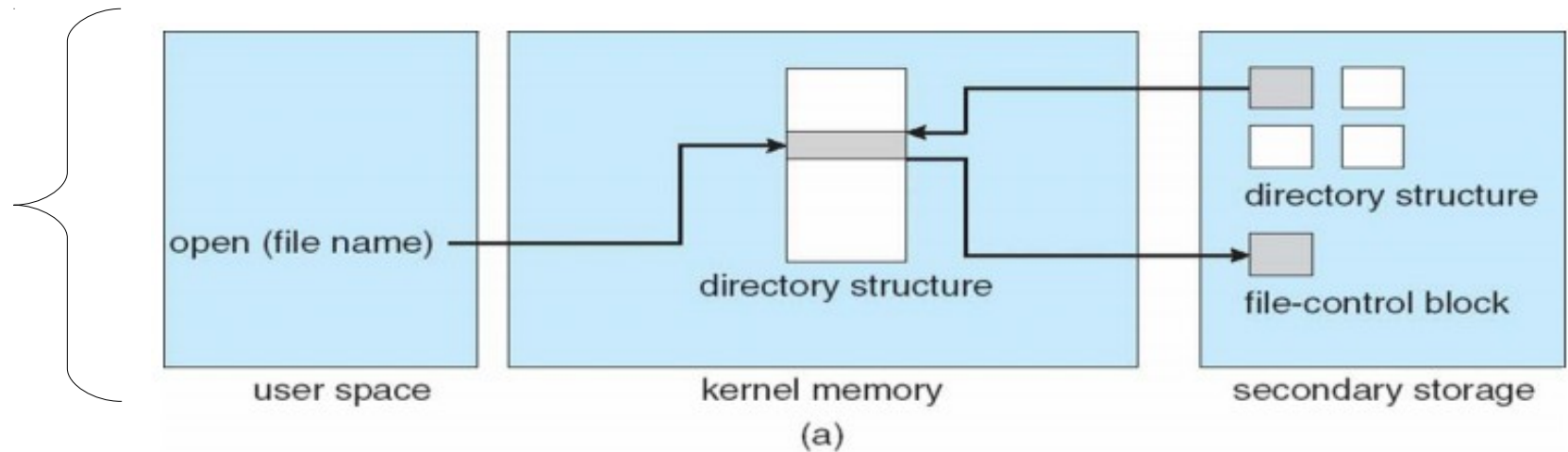
- 
- Open/close operations modify the data structures which the OS uses for managing files:
    - ♦ Open file table
  - Multi-tasking OS have two different file tables:
    - ♦ a global, single, table
      - ♦ Has a single entry for each open file in the whole system
    - ♦ a process file table:
      - ♦ Stores file access informations for that process

# File control block

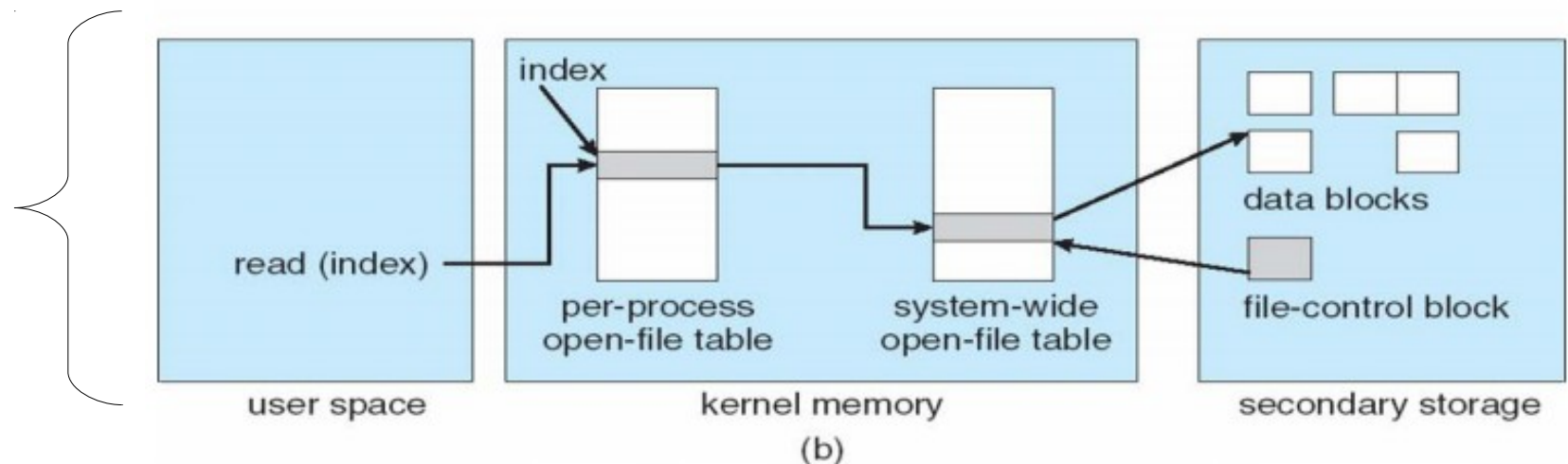
permissions
creation, last access and modification
Owner and group ID
size
data blocks and/or data block pointers

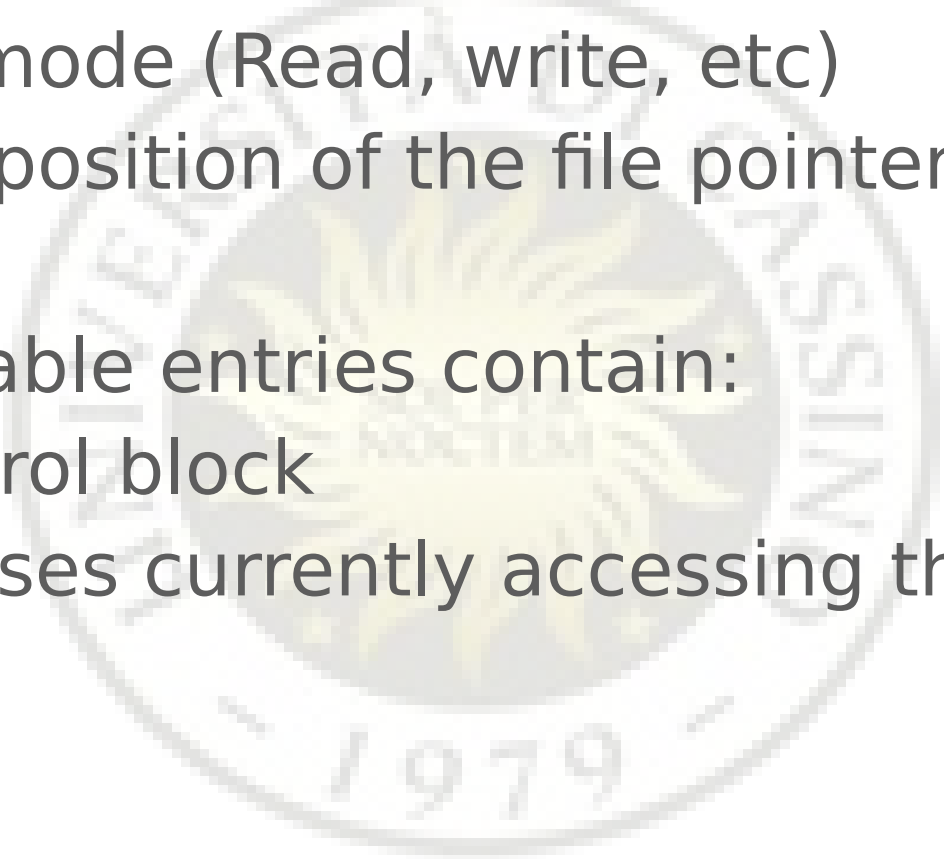
# File system data structures in memory

file  
opening



file  
reading/  
writing



- 
- Process file table entries contain:
    - ♦ Access mode (Read, write, etc)
    - ♦ Current position of the file pointer
  - global file table entries contain:
    - ♦ File control block
    - ♦ #processes currently accessing the file

# Directory

- File system organization is based on the concept of directory:
  - ♦ An abstraction for set of files
- In many systems directories are (special) files
- Directory operations
  - ♦ creation
  - ♦ cancellation
  - ♦ opening
  - ♦ closing
  - ♦ reading
  - ♦ Renaming
  - ♦ link/unlink
  - ♦ Traversal



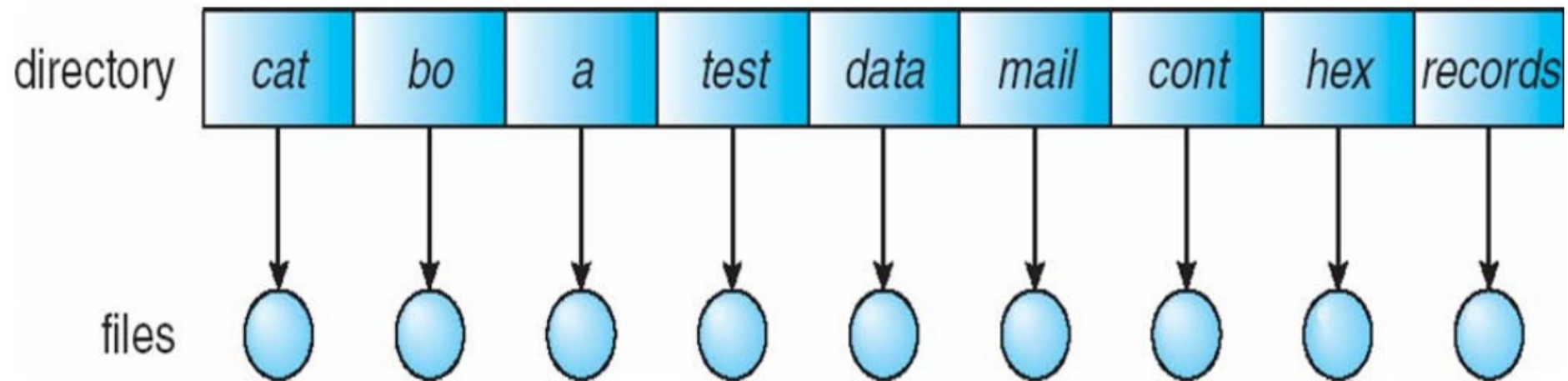
# Directory structures

- Single level
- Two levels
- Tree structured
- Acyclic graph
- Graph



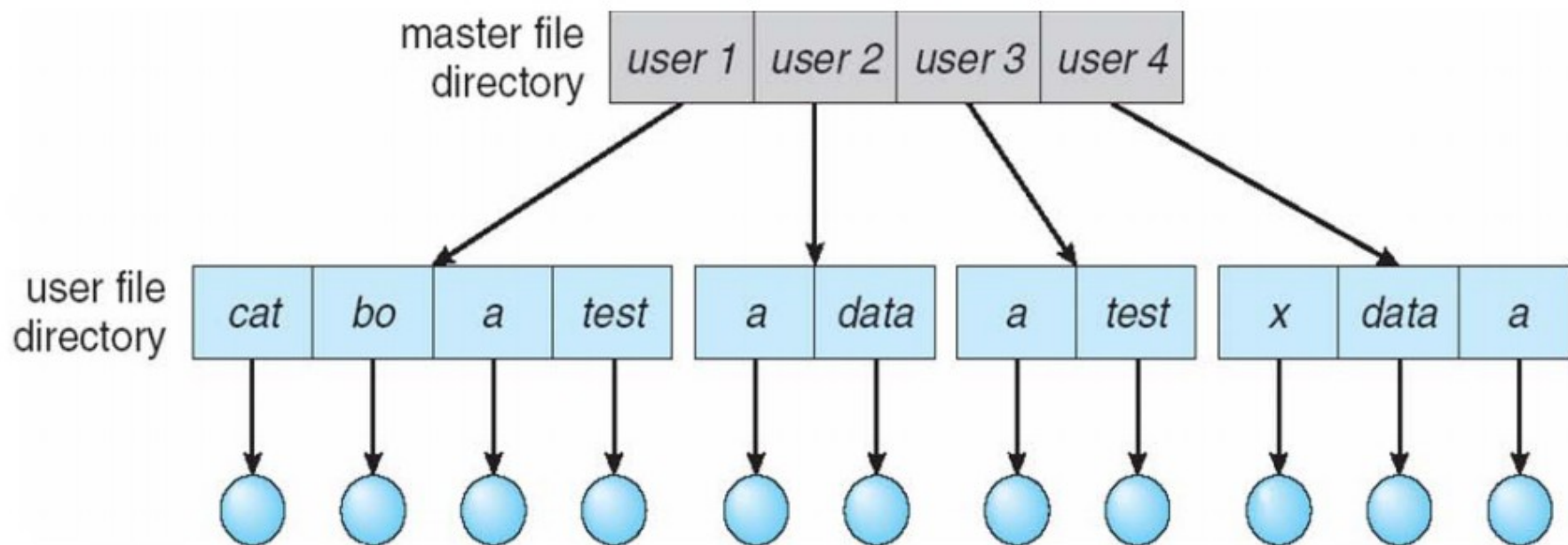
# Single level directories

- A single directory for all users
- Naming and grouping problems

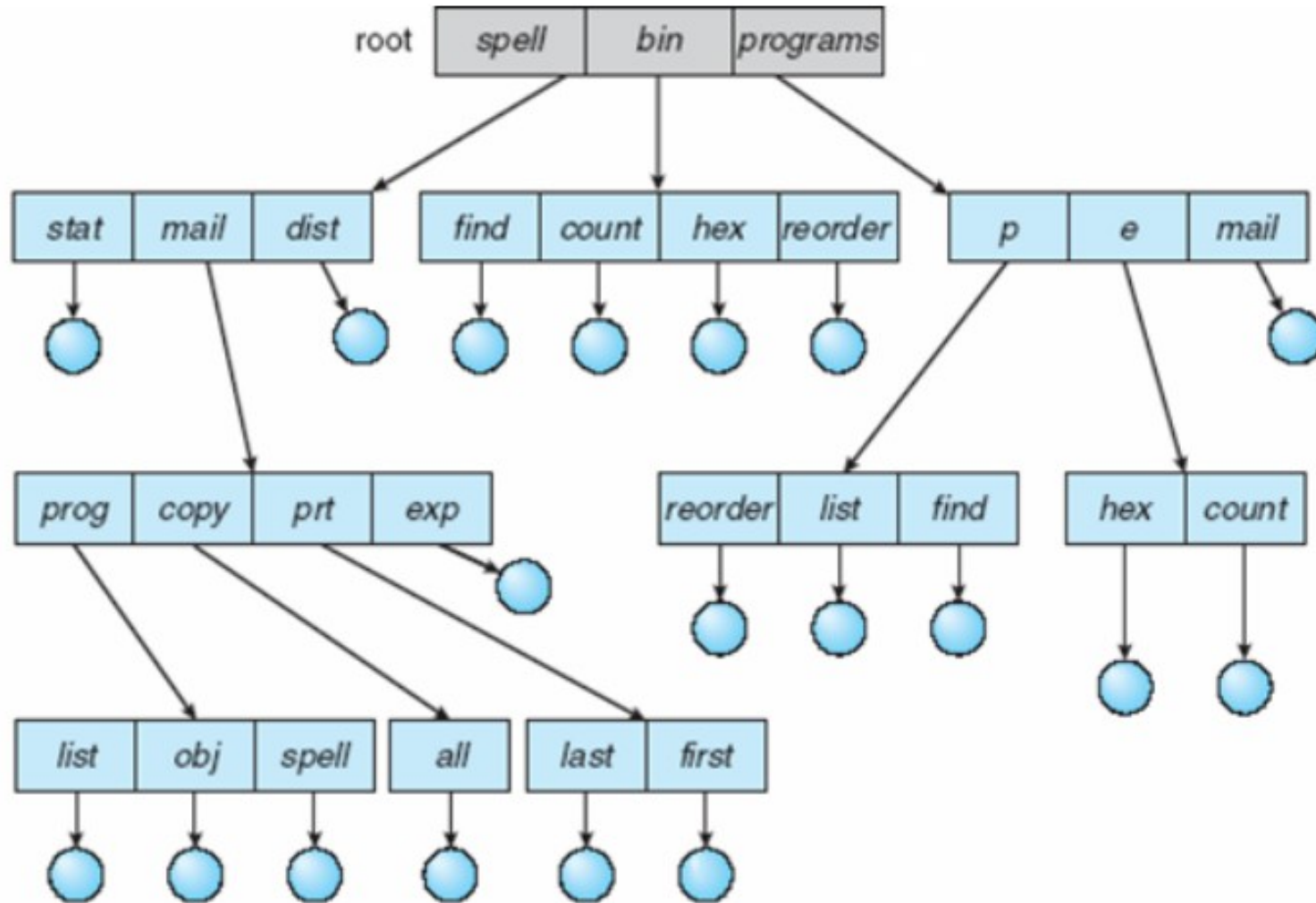


# Two-level directories

- A directory for each user
- Path name
- Different users can have files with the same name
- Efficient search
- No grouping



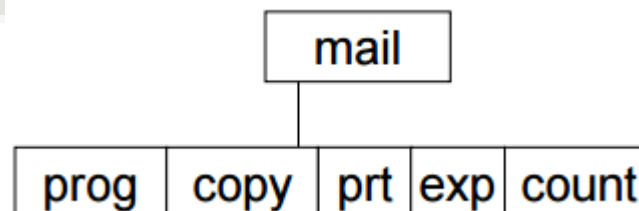
# Tree structured directories



- Efficient searching
- Grouping capability
- Absolute or relative path name
- Current directory (working dir)
  - ♦ `$> cd /home/francesco/OS`
  - ♦ Creating a new file/directory is done in the current directory
- Deleting a directory implies deleting the entire subtree

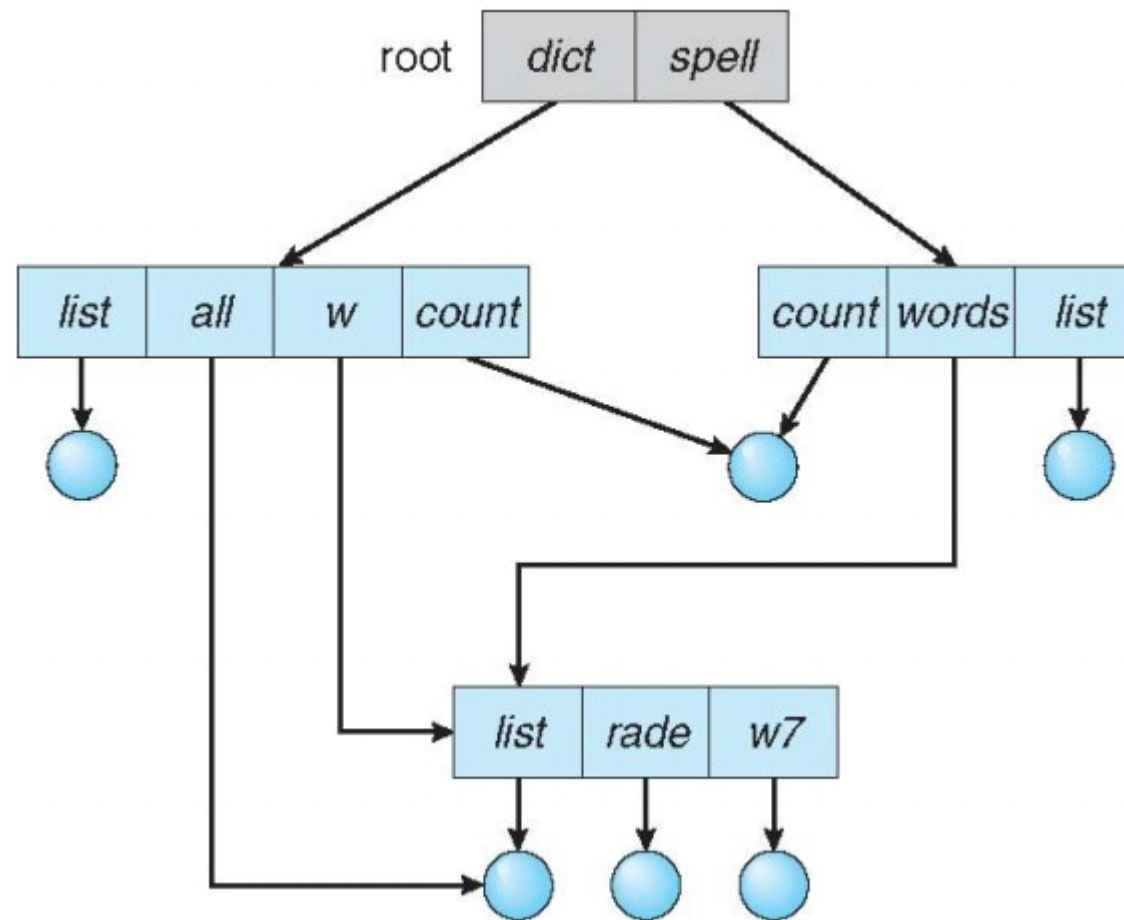
### **example**

deleting the dir mail, also implies deleting the subdirectories  
prog, copy, prt and the file exp

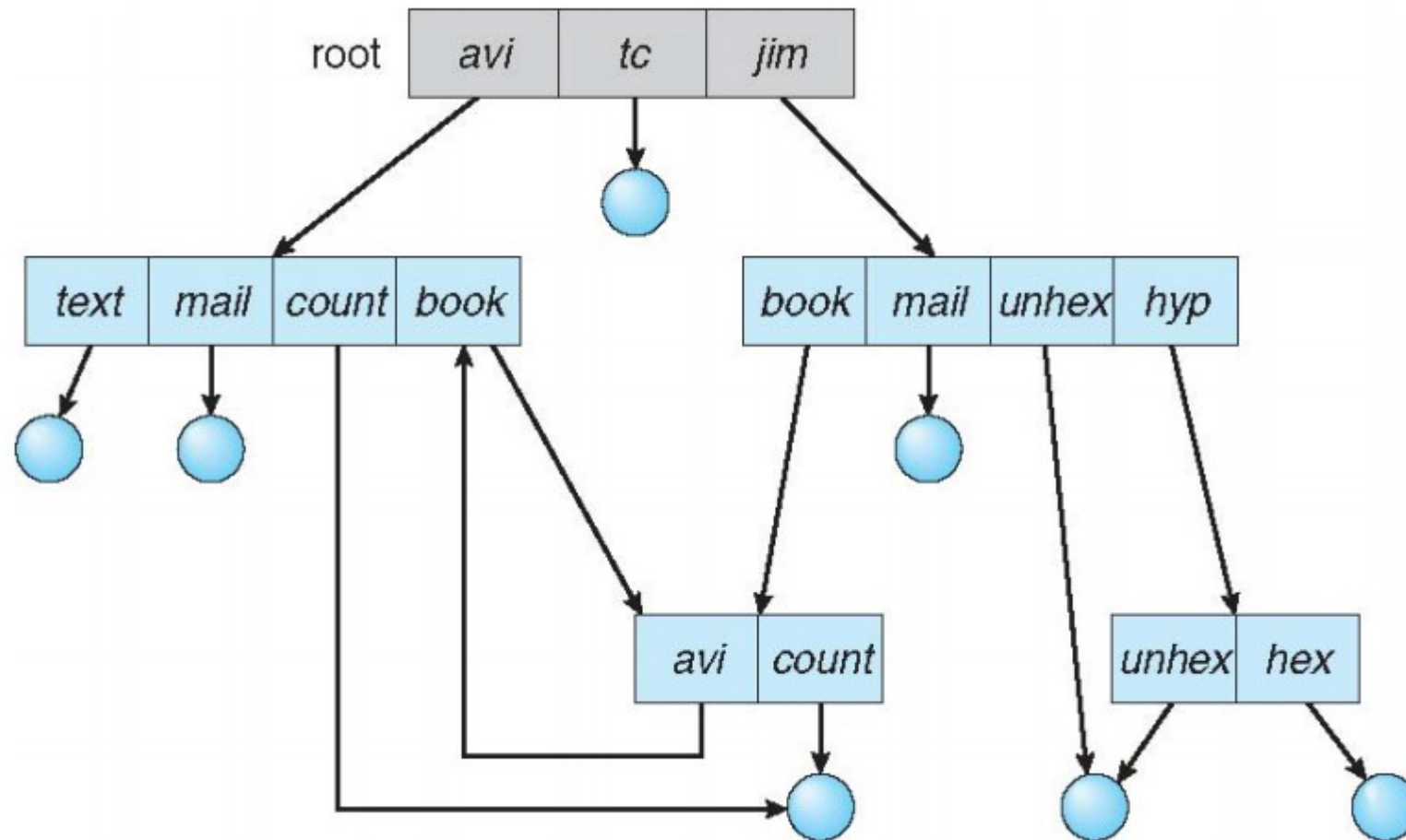


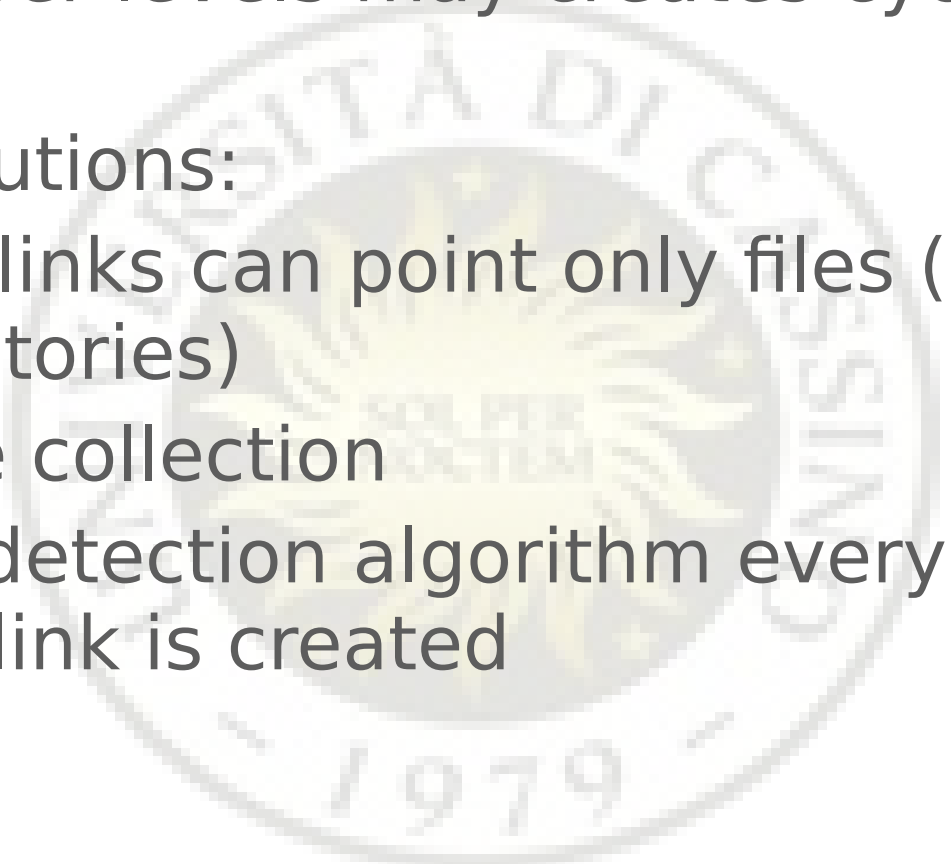
# Acyclic Graph Directories (DAG)

- Have shared subdirectories and files



# General graph structure



- 
- Links to upper levels may creates cycles inside the graph
  - Possible solutions:
    - ♦ Upward links can point only files (no subdirectories)
    - ♦ Garbage collection
    - ♦ A cycle detection algorithm every time a new upward link is created

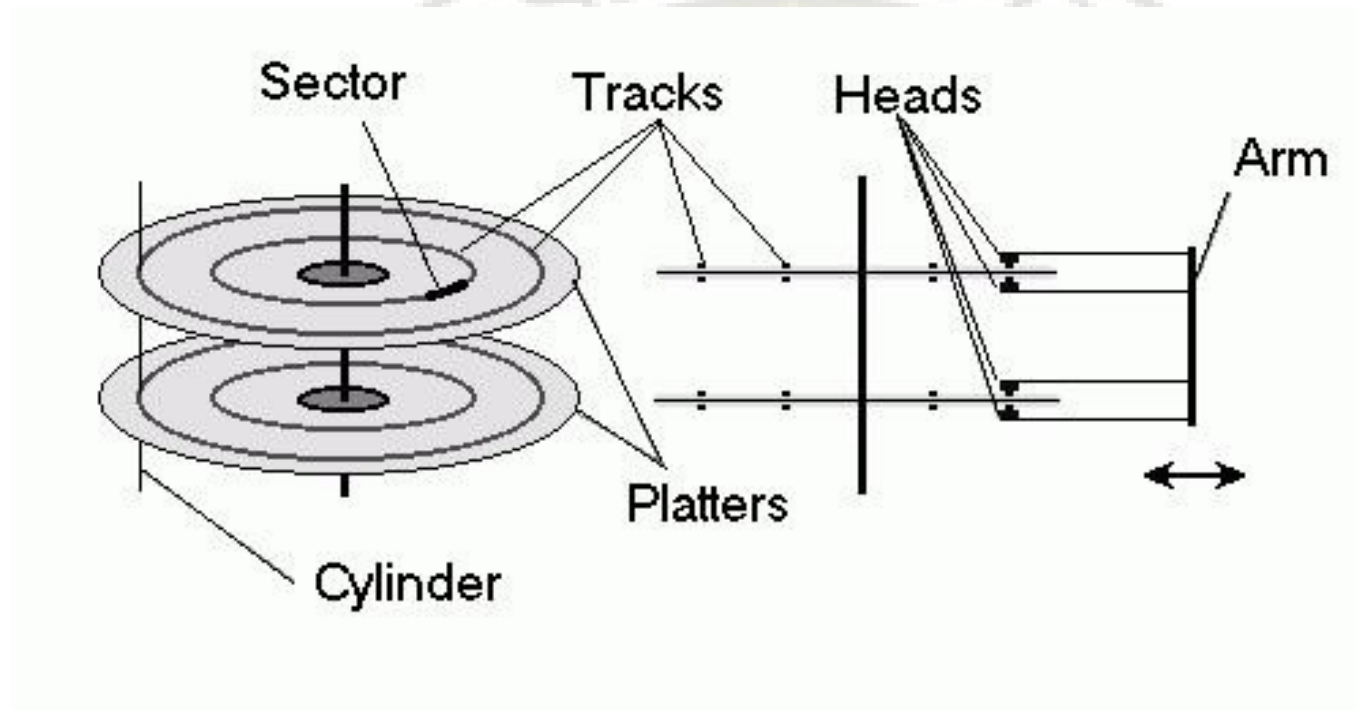
# Coherence semantic

- In a multitasking OS, processes access files independently
- How are managed the file modifications from the various processes?
- In UNIX
  - Content modifications from a process are immediately notified to the other processes
  - Two different types of file sharing:
    - A a single file current position pointer, shared by all accessing processes
    - Each accessing process has its own pointer

# file system implementation



# Hard disks

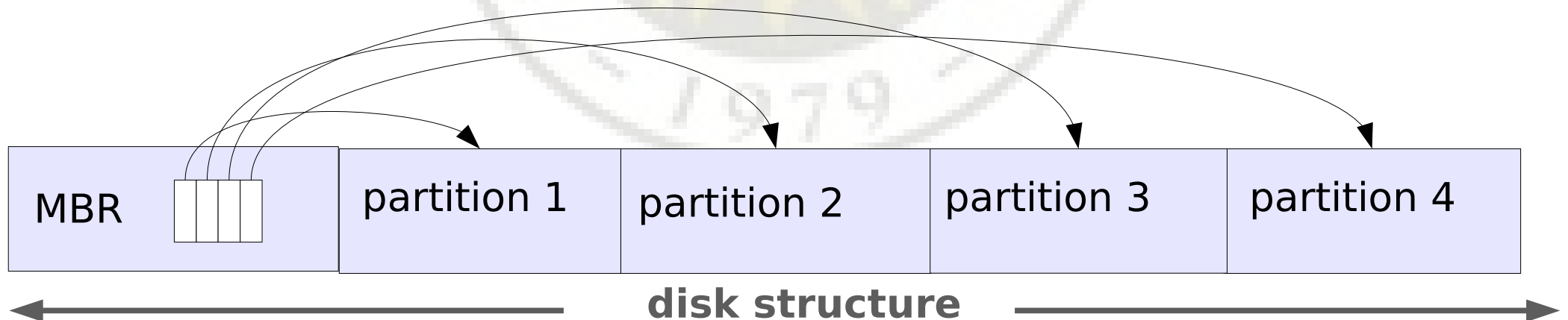


# File system implementation

- Problems to be taken into account
  - Disk is split in blocks
  - Block allocation/deallocation
  - Free block management
  - Directory implementation
  - performance optimizations
  - Techniques for coherence guarantee

# Disk organization

- A disk can be split in one or more **partitions**:
  - ♦ Independent portion of the disk, each of which can host different file systems
- the first sector of the disk is the **master boot record (MBR)**:
  - ♦ is used for the boot
  - ♦ contains the **partition table**, and the active partition
  - ♦ at the boot, it is read and executed



# Master Boot Record

Structure of a master boot record					
Address			Description		Size (bytes)
Hex	Oct	Dec			
0	0	0	code area		440 (max. 446)
01B8	670	440	disk signature (optional)		4
01BC	674	444	Usually nulls; 0x0000		2
01BE	676	446	Table of primary partitions (Four 16-byte entries)		64
01FE	776	510	55h	MBR signature; 0xAA55	2
01FF	777	511	AAh		
MBR, total size: 446 + 64 + 2 =					512

## NOTE

MBR does not belong to any partition

# Partitions

- Two types of partitions
  - **Primary**
    - Can contains a single file system
  - **extended:**
    - Can be split in more **logical partitions**, each with its own file system
    - Can be bootable
    - A disk can contains at most a single extended partition

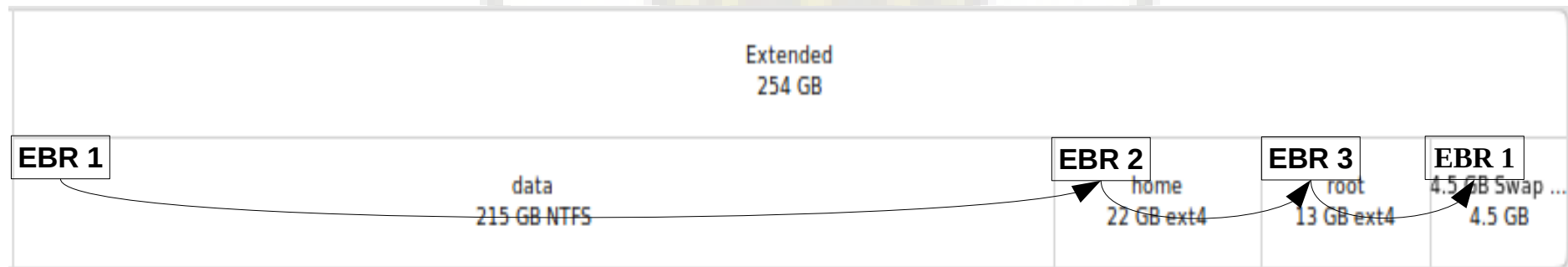
# Partition record

Offsets	Byte Count	Description
0	1	Boot indicator (80h for active; otherwise, 00h)
1 – 3	3	Starting CHS (cylinder, head, sector) values
4	1	<b>Partition type code</b>
5 – 7	3	Ending CHS (cylinder, head, sector) values
8 – 11	4	Starting sector (LBA)
12 – 15	4	Partition size (in sectors)

Specifies the partition type:  
NTFS, EXT3, SWAP, extended, etc.  
A list of file system codes is available [here](#)

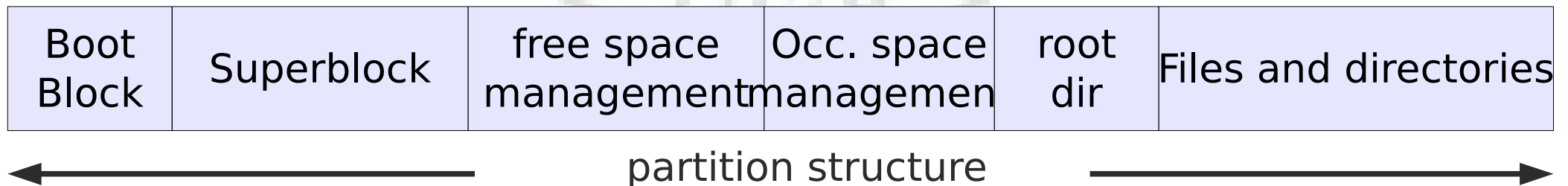
# Extended Boot Record (EBR)

- The Extended Boot Record (EBR) is the descriptor used for logical partitions
- Each logical partition is preceded by the associated EBR
- Each EBR contains a pointer to the next EBR



# Disk organization

- Each partition starts with the boot block
- At the boot the MBR (machine) code loads and executes the boot block of the active partition
- On turn the boot block loads, and executes the OS contained in the partition
- The organization of the rest of the partition depends on the file system

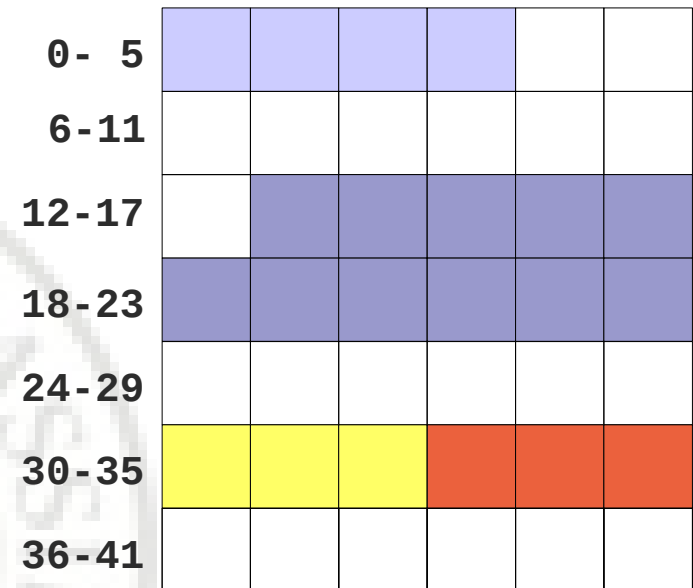


# Allocation problem

- OS disk drivers (and controllers) provide a set of (physical) fixed size blocks
- Starting from this simple organization, how file abstraction can be implemented?
- More precisely:
  - ♦ how disk blocks are chosen for file creation/expansion?
  - ♦ How these blocks are linked in order to model a single (logical) memory space

# Contiguous allocation

- File content is stored in contiguous sequences of disk blocks
- **Vantages**
  - No data structure for linking the blocks
  - Efficient sequential access
    - ♦ Contiguous blocks does not need (slow) arm seek operations
  - Direct access is also efficient:
    - **block= offset/blocksize;**
    - **pos= offset%blocksize;**



0- 5					
6-11					
12-17					
18-23					
24-29					
30-35					
36-41					

directory

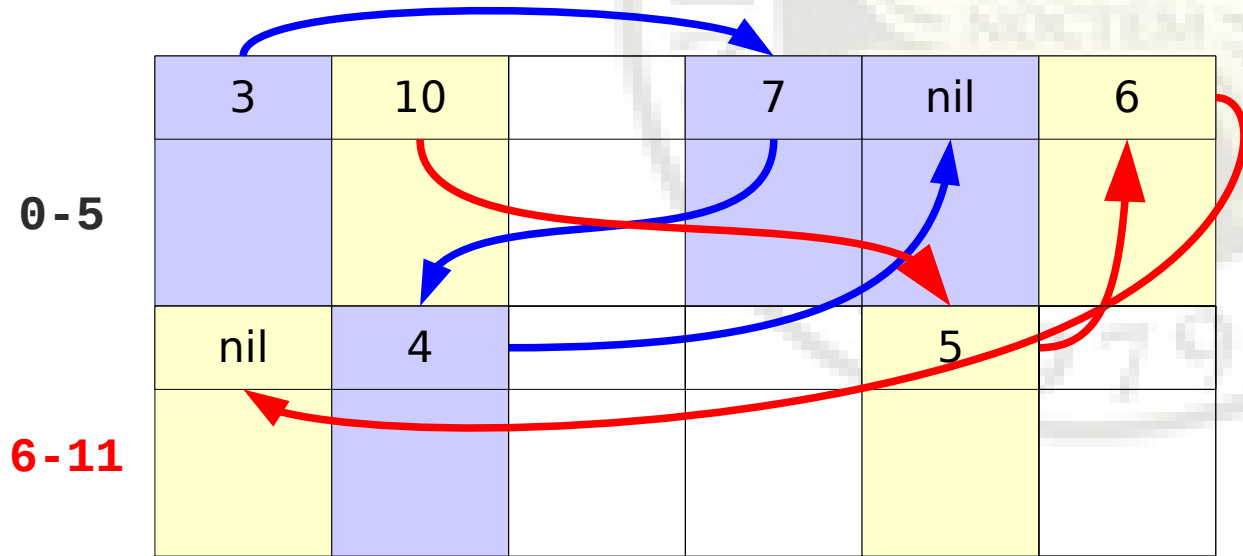
Name	Start	Size
a	0	4
b	13	11
c	30	3
d	33	3

## ■ Disadvantages

- ♦ External fragmentation (as for main memory allocation)
  - ♦ Needs a policy for selecting the disk area: first fit, best fit, worst fit....
  - ♦ Files cannot grow up!
- It is used for the swap partition (virtual memory) in Linux

# Linked allocation

- The content of a file is stored in a list of linked blocks
- Each block contains a pointer to (the index of) the next block
- File descriptor contains the pointers to the first and the last blocks



**Allocatable space**

**directory**

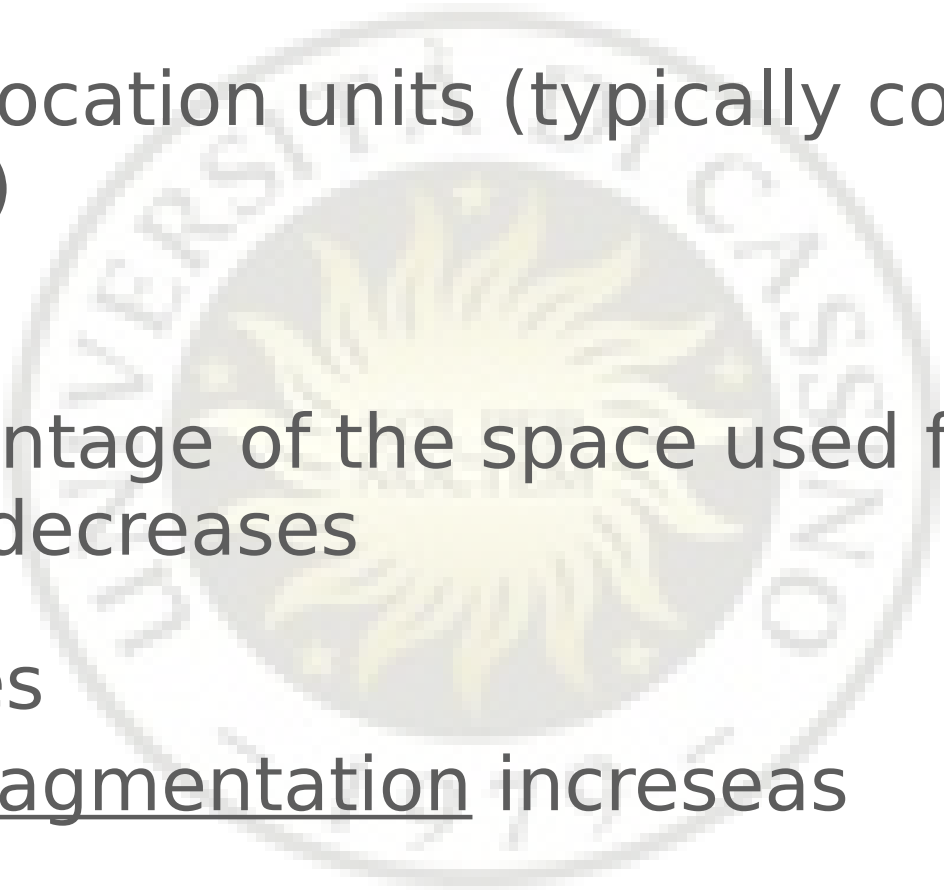
Name	Start	End
a	0	4
b	1	6

## ■ **Advantages**

- ♦ No external fragmentation
- ♦ Efficient “append mode” access

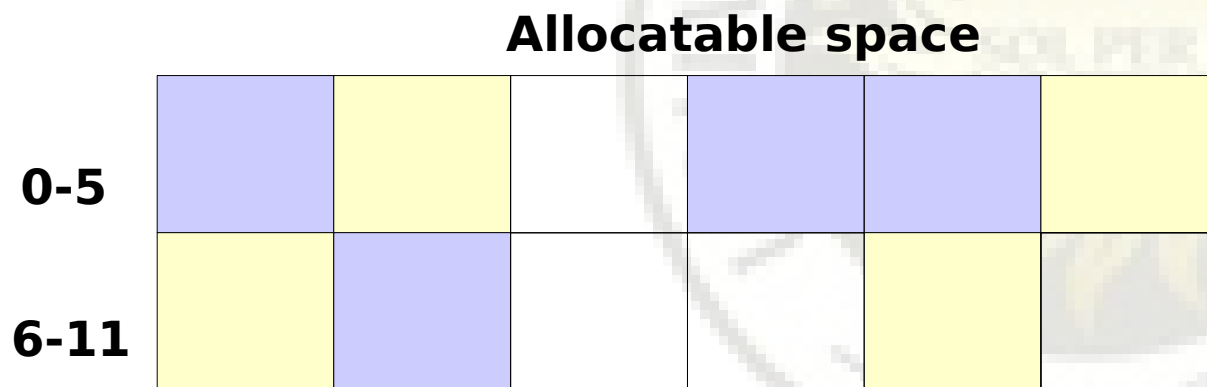
## ■ **disadvantages**

- ♦ Direct access inefficient
- ♦ File system performance tends to decrease:
  - ♦ blocks tend to spread out, (arm seeks increase)
- ♦ Blocks size is not a power of two
- ♦ For small blocks (e.g. 512 bytes) pointer overhead may result significant

- 
- To minimize pointer overhead, blocks are clustered in blocks:
    - ♦ atomic allocation units (typically containing 4, 8, 16 blocks)
  - Advantages
    - ♦ The percentage of the space used for storing pointers decreases
  - disadvantages
    - ♦ internal fragmentation increseas

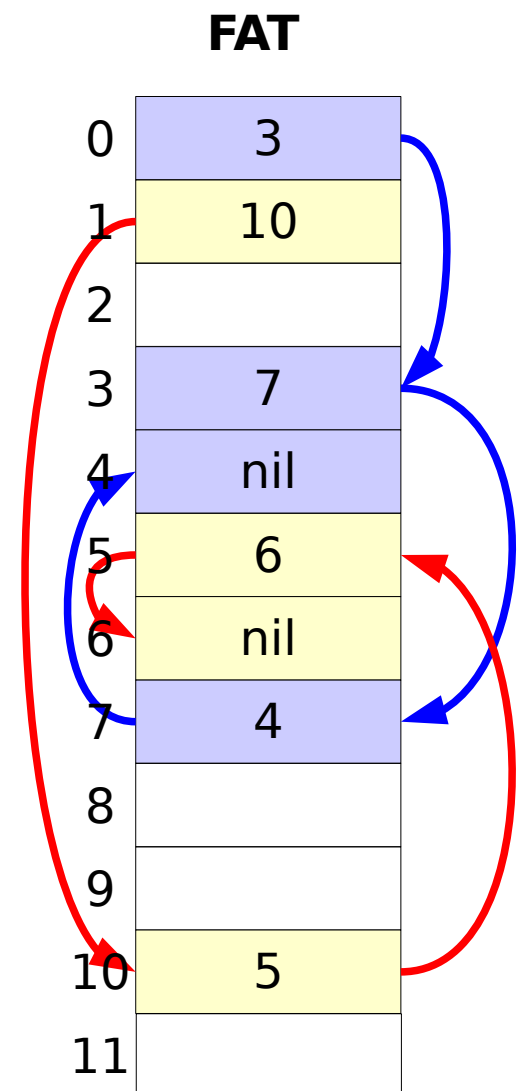
# File Allocation Table (FAT)

- Uses a separate table for storing the pointer to the next block (or cluster)
- An entry for each block (or cluster)
- Similar to linked list



**directory**

Name	Start	End
a	0	4
b	1	6



## ■ Advantages

- ♦ data blocks contain only data

## ■ Disadvantages

- ♦ File scanning also requires reading the FAT:
  - ♦ The number of disk accesses increases

## ■ Advantages

- ♦ FAT blocks can be cached in memory:
  - ♦ Direct access is fast because the pointer list can be scanned in memory
- ♦

## ■ Used by DOS

# FAT32: limitations

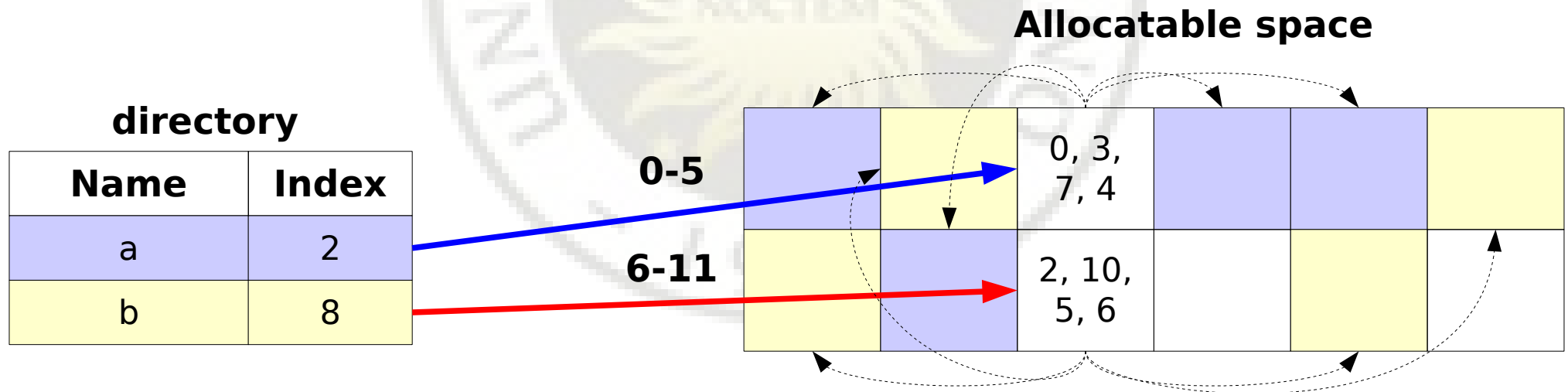
- First FAT32 versions had two important limitations:
  - ♦ A maximum partition size of 128 GB:
    - ♦ Due to the fdisk utility for disk partitioning
  - ♦ Maximum file size of 4GB:
    - ♦ Directory entry has a field of 4 bytes for the file size

# FAT table size (example)

- Suppose you have a 1TB ( $2^{40}$  bytes) hard disk drive and the block size is 1KB ( $2^{10}$  bytes)
- How large is its FAT32 ?
- The disk contains  $2^{30}$  blocks ( $2^{40}/2^{10}$ ):
  - ♦ then table consists of  $2^{40}$  entries
  - ♦ table entries are 4 bytes ( $2^{32}$  bits)
- The size of the FAT32 table is  $2^{32}(2^{32} * 2^2)$ , i.e. 4GB

# Indexed allocation

- The list of the blocks making up a file is stored in a separated block(s)
- To access the file: its index block is loaded in memory, then the blocks index are read



## ■ **Advantages**

- ♦ No external fragmentation
- ♦ Efficient direct access
- ♦ The index block is loaded in memory only when the file is open

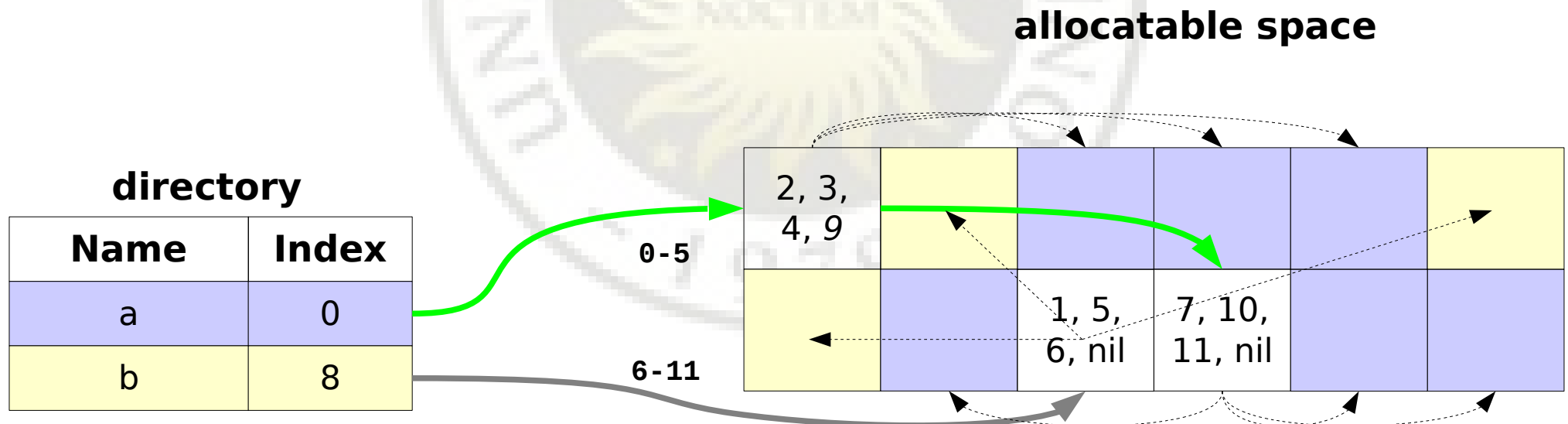
## ■ **Disadvantages**

- ♦ Index block size determines the maximum file size
- ♦ too large index blocks cause a big disk space waste

## ■ A trade-off must be found

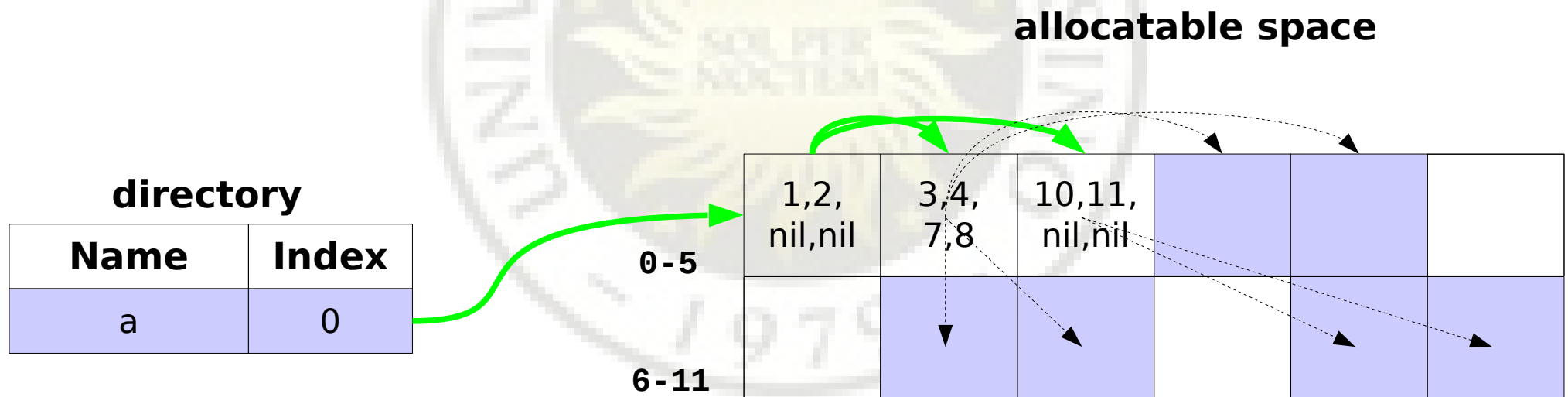
# Solutions

- Index blocks concatenation:
  - ♦ The last element of the index block points to the next index block
- Direct access to big files is (time) costly



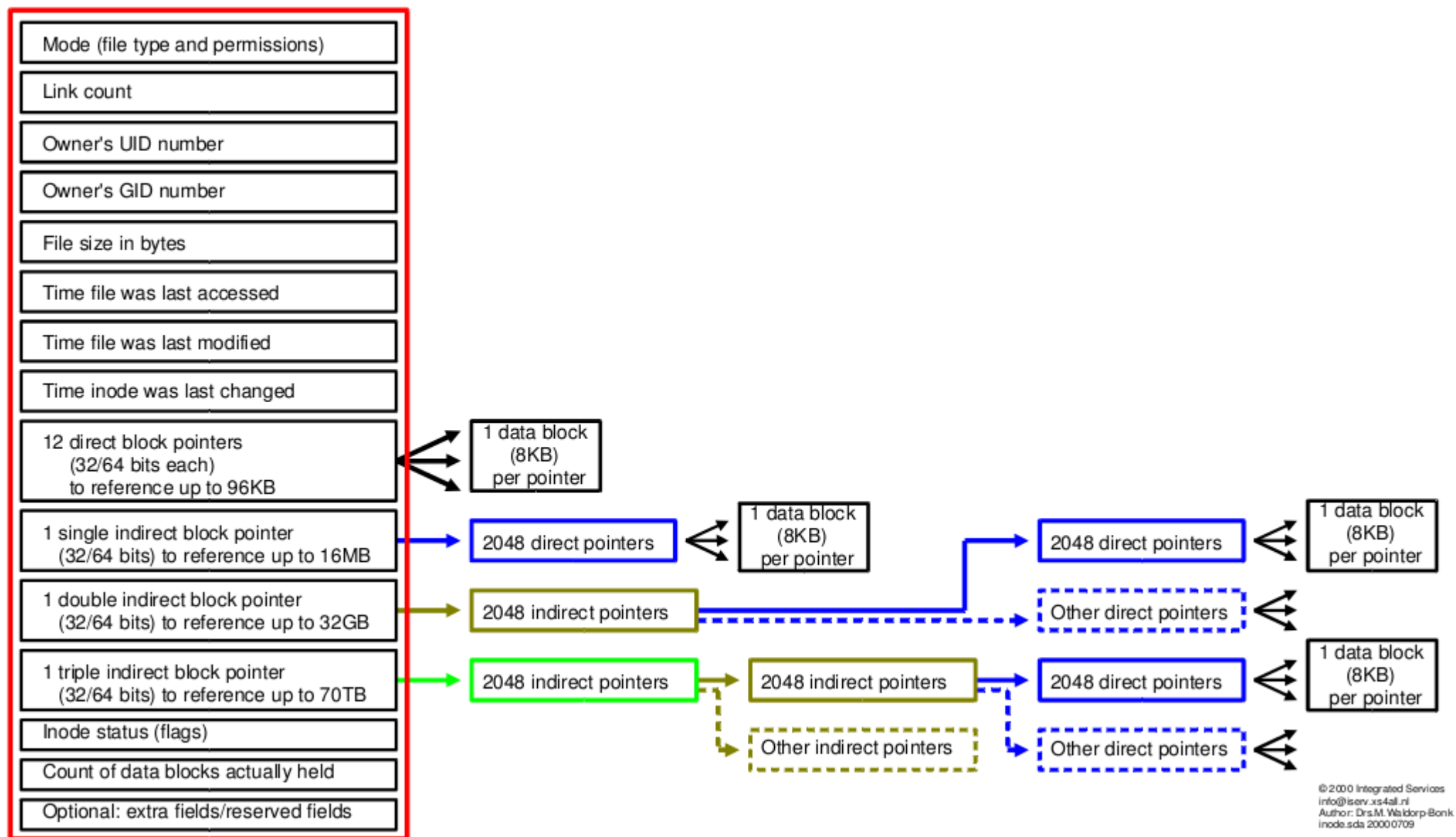
## ■ Multilevel index blocks:

- ♦ An index block for the index blocks
- ♦ Performance decreases, since it requires more disk accesses



# Indexed allocation in UNIX

- Each file is associated to an i-node (index node):
- An i-node is a data structure containing:
  - ♦ file attributes
  - ♦ 12 direct block pointers (the indices in the UNIX jargon)
  - ♦ 1 single indirected block pointers
  - ♦ 1 doubled indirected block pointers
  - ♦ 1 tripled indirected block pointers
- Each file system (partition) has its own i-node table, stored at a fixed location



# I-node: allocation and performance

- Good performance for direct access
- Small files can be accessed fastly and occupy less disk space (do not need pointer blocks)
- Further improvements:
  - Pointer blocks can be loaded in memory before they are actually needed
  - Combining contiguous and indexed allocation
    - Contiguous, when possible, for small files
    - Indexed for larger files

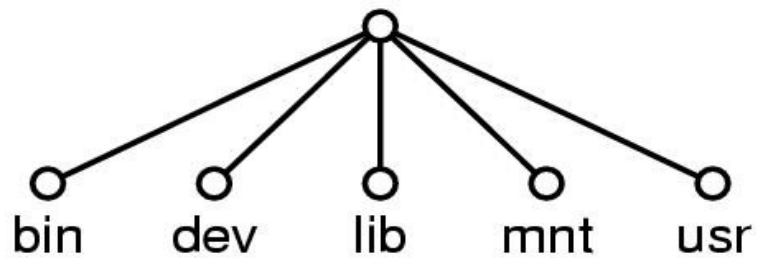
# File system mounting

- Before its file can be accessed, any file system must be **mounted**
- Mounting operation requires two parameters:
  - ♦ The name of the device
  - ♦ Its location (*the mounting point*) in the OS tree (or DAG or graph) structure

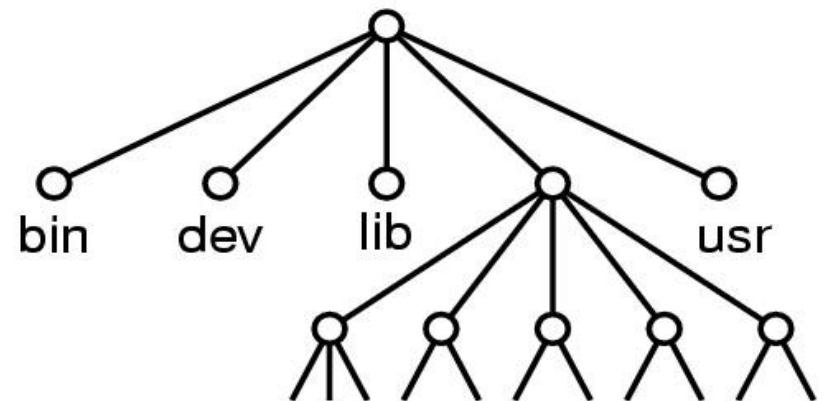
# File system mounting in UNIX

- Mounting point can be anywhere in the structure
- The mounting point can be even not empty:
  - ♦ The contained files and directories are hidden while a file system is mounted there
- Explicit file mounting:
  - ♦ Needs a configuration file for the partitions to be mounted at the boot (`/etc/fstab` file in linux)

**\$> mount /dev/fd0 /mnt**



(a)



(b)

# File system mounting in WINDOWS

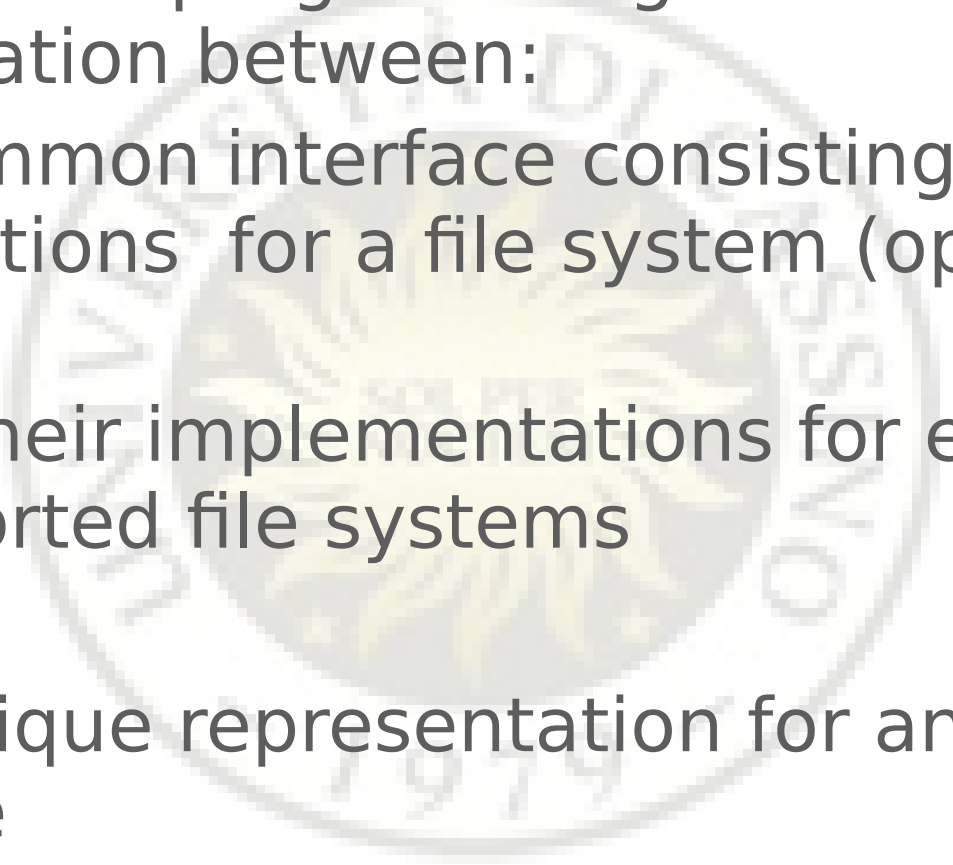
- Directory structure is split in two levels:
  - ♦ a unit letter is associated to each mounted device and/or partition
- Automatic boot mounting

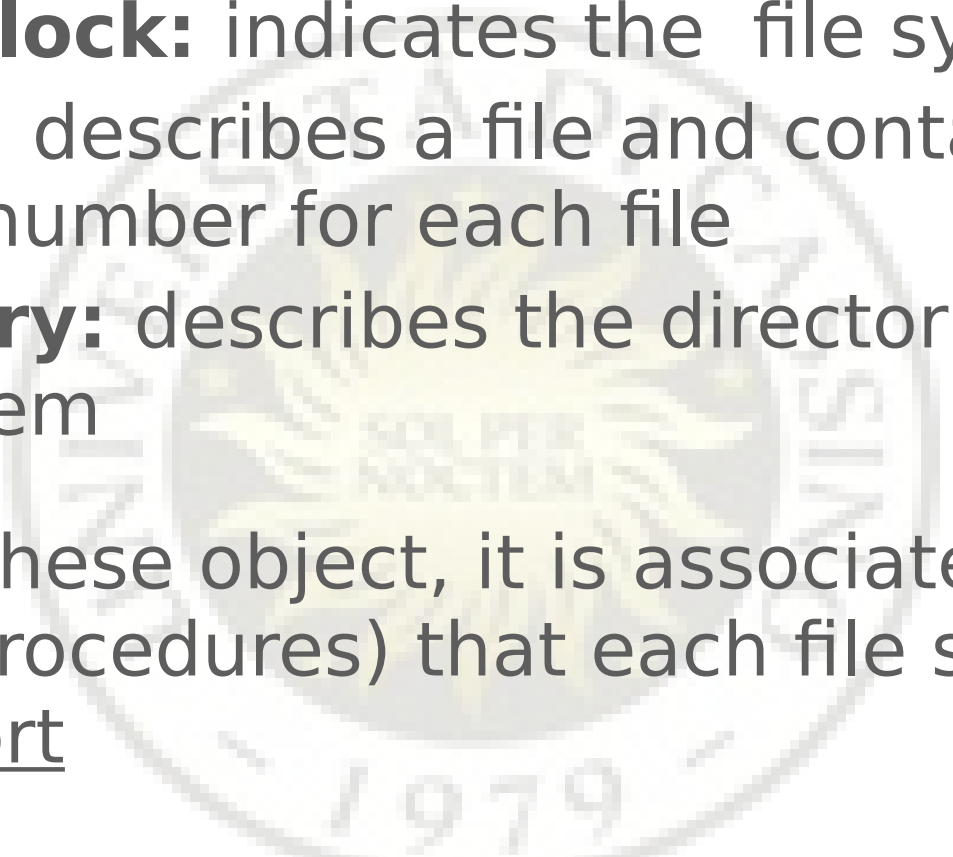
# The mounting table

- Every OS has a data structure called mounting table
- mounting table lists the partitions currently mounted
- The entry of this table specifies:
  - ♦ file system type
  - ♦ device type
- In the UNIX-based systems, the i-node has a field indicating if that directory is a mounting point. If so, the i-node points to the corresponding entry in the mounting table

# Virtual File System

- Multiple file systems may be present on the same computer (ext2, fat32, NTFS, CD-ROM, etc).
- Each file system has its own procedure for accessing files
- Different file systems can be managed using object oriented programming techniques

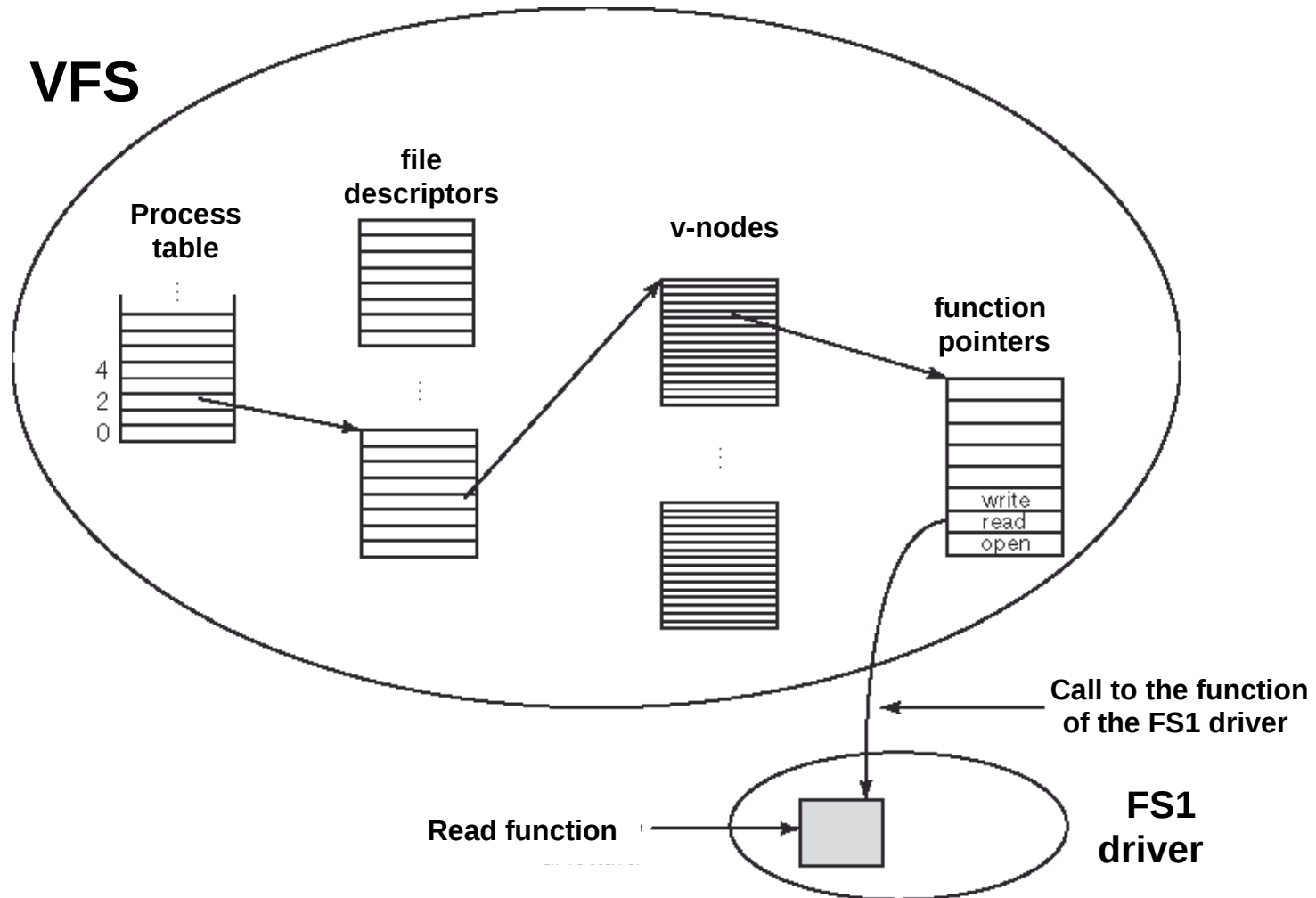
- 
- Object oriented programming techniques allow a clear separation between:
    - ♦ a common interface consisting of the generic operations for a file system (open, read, write, etc.)
    - ♦ and their implementations for each of the supported file systems
  - Allows a unique representation for any local or even network file

- 
- VFS implementation has the following objects:
    - ♦ **Superblock:** indicates the file system type
    - ♦ **v-node:** describes a file and contains a unique number for each file
    - ♦ **directory:** describes the directories in the file system
  - To each of these object, it is associated a set of methods (procedures) that each file system must support
  - The mounting table is inside the VFS

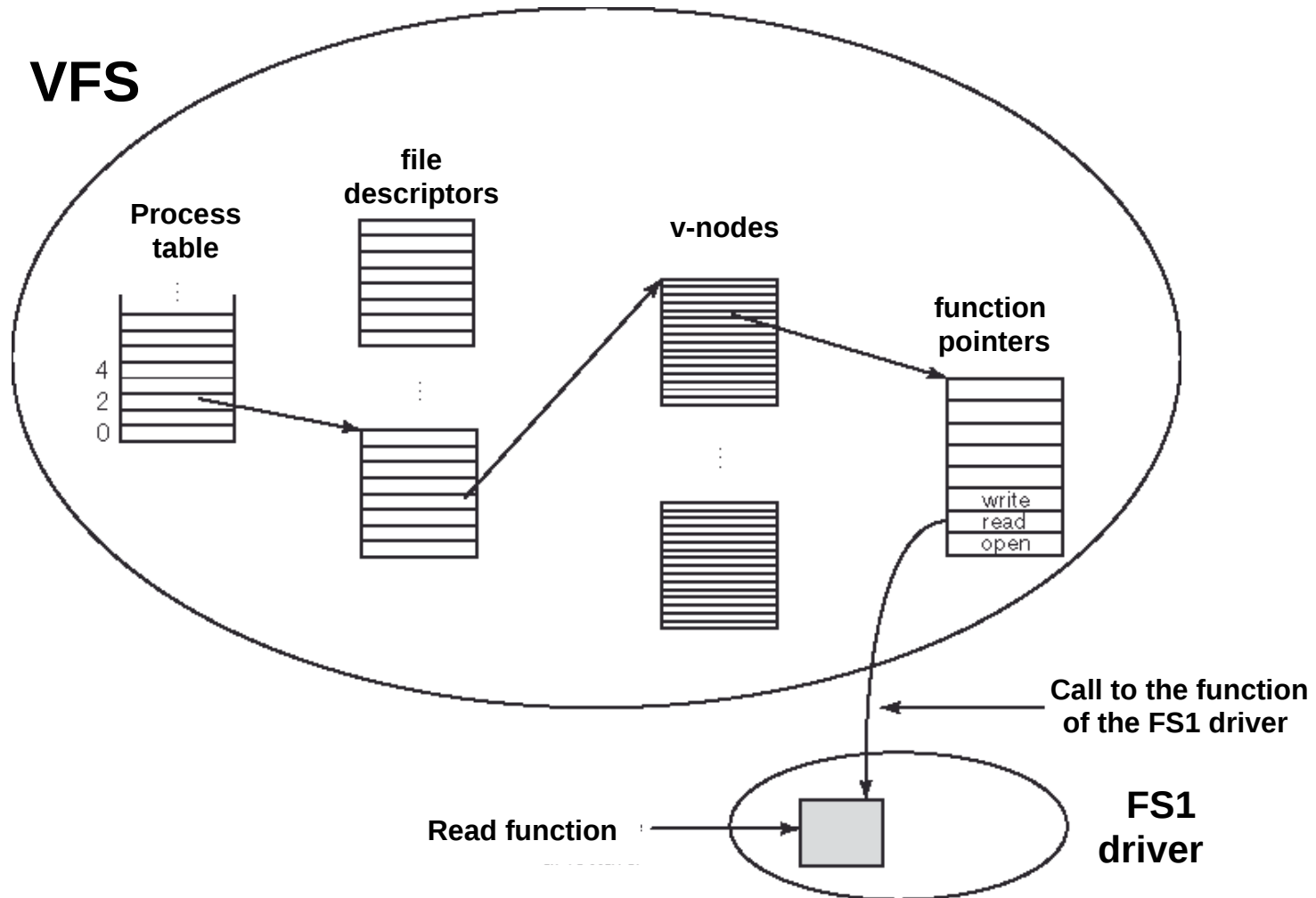
# The mounting phase

- When a file system is mounted, the corresponding driver (module) must provide to the VFS the addresses (pointers) of the procedures specified by the interface
- In practice the driver provides to the VFS the address of a table of function pointers:
  - ♦ At each table entry corresponds a given function (open, write, etc)
  - ♦ Each function has the same table index in any file system driver
- Afterwards, the VFS can access that file system by simply calling those procedures, by using these agreed indices

# Virtual File System implementation



# Virtual File System implementation



# Example

```
fd = open("/usr/francesco/risultatiSO.txt", ORDONLY)
```

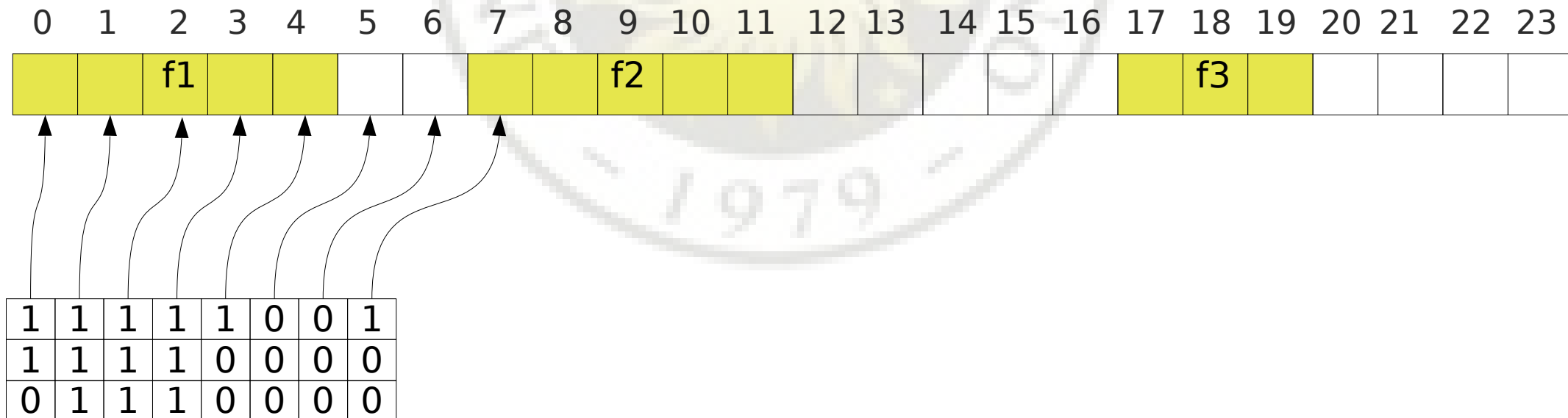
- The VFS scans the file path and figures out that the on the directory `usr` a new file system has been mounted:
  - ♦ locates the superblock object in the list of the file systems mounted
  - ♦ creates a v-node and calls a suitable procedure of the real file system, which copies into the v-node:
    - ♦ the information contained in the file control block
    - ♦ The address of the table of the pointers to the functions implemented for that v-node (open, read, write, etc)
  - ♦ finally, a new entry is added to the global file descriptor table, which points to the just created v-node

# Free space management

- A crucial aspect in any file system concerns the free space management
- The purpose is to reuse the blocks made free by file deletion/reduction
- It needs a data structure for keeping track of the free blocks

# Bitmap

- A bit for each block
- 0 represents free blocks, while 1 represents occupied blocks



## ■ advantages

- simple,
- Easy to select contiguous blocks:
  - ♦ X86 architecture has instructions to find the first on/off bit in a word

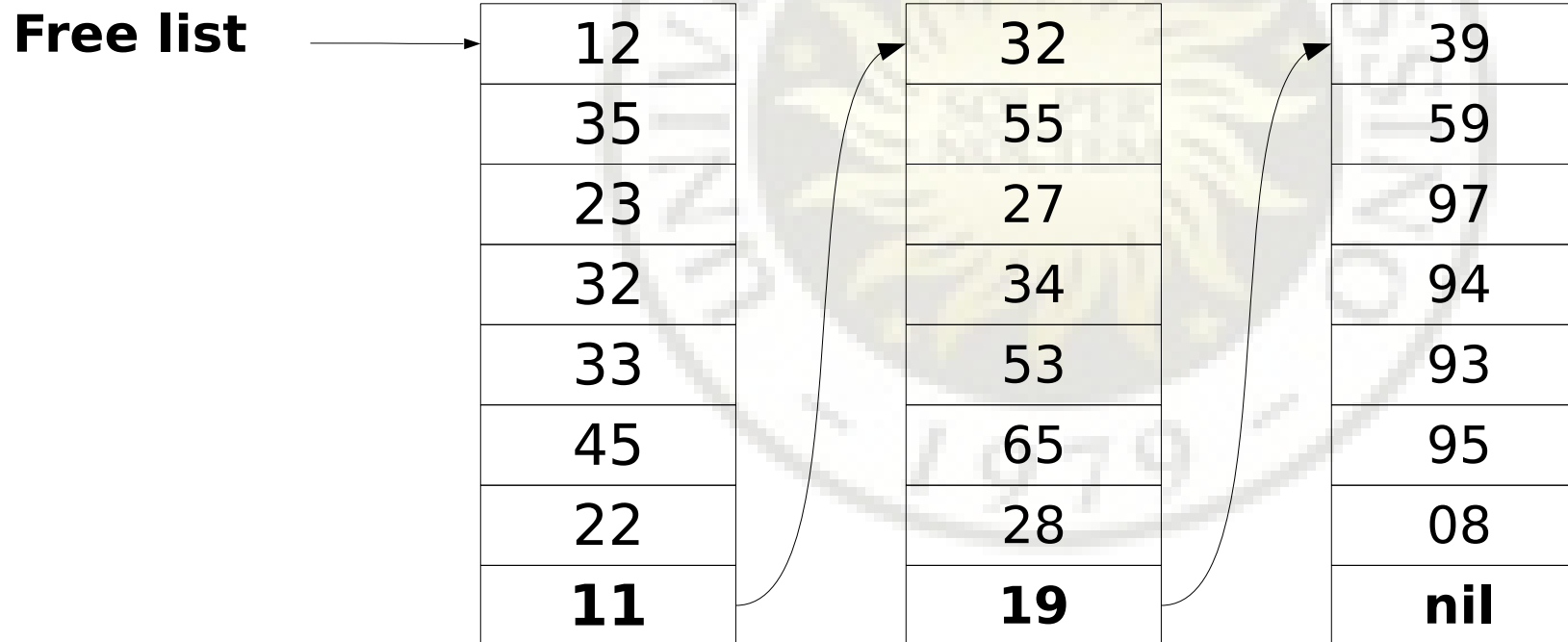
## ■ disadvantages

- Bitmap storage requires extra space  
example

- ♦ block size = 4KB ( $2^{12}$  bytes)
- ♦ disk size = 1 TB ( $2^{40}$  bytes)
- ♦ #blocks =  $2^{40} / 2^{12} = 2^{28}$
- ♦ Bimap size =  $2^{28}$  bits -->  $2^{25}$  bytes (32 MB)

# Linked list

- A linked list of blocks containing the indices (pointers) of the free blocks



## ■ Advantages

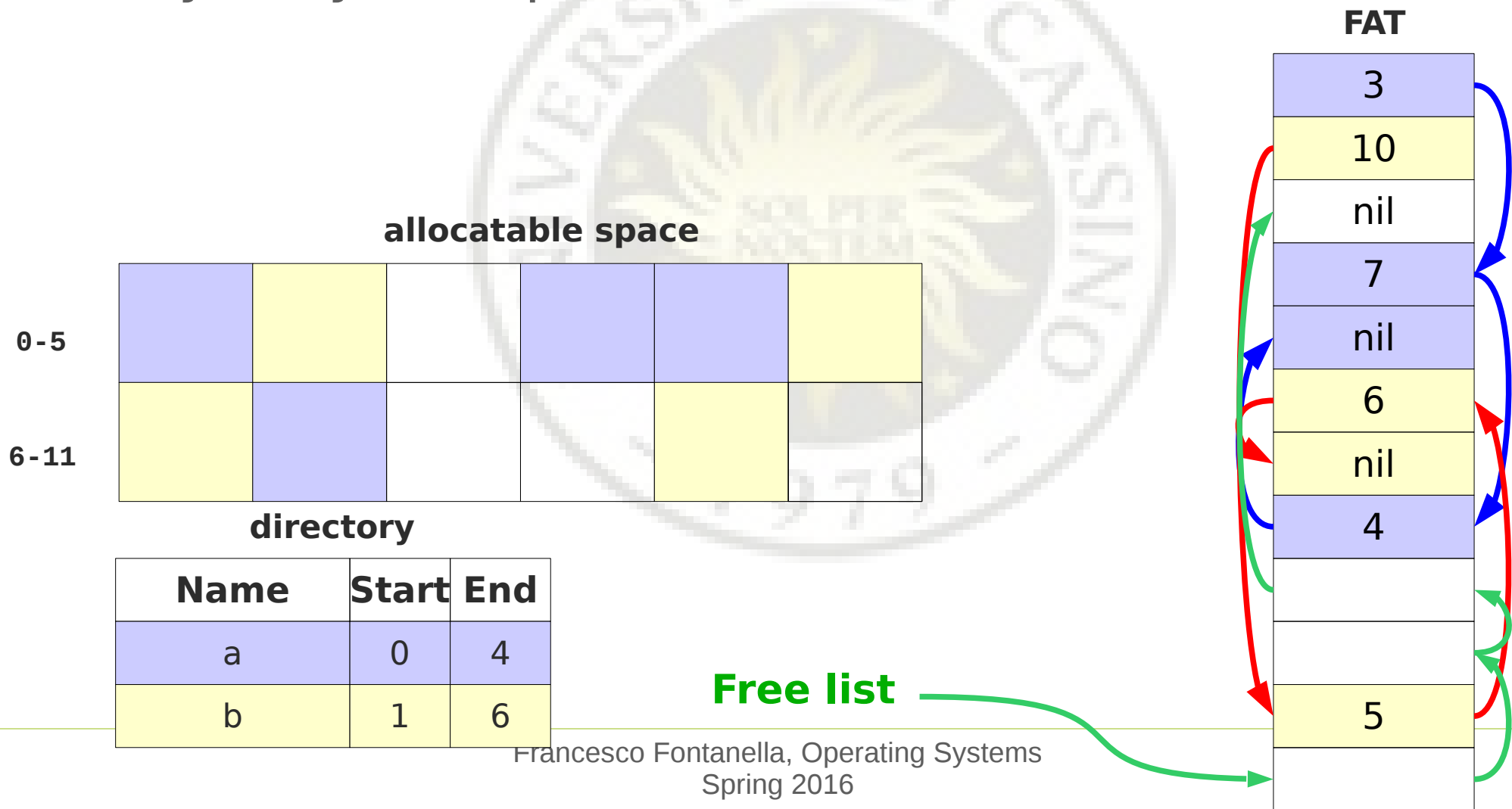
- ♦ needs small space in main memory:
  - ♦ it is sufficient to keep in memory only the first block containing the free block indices
- ♦ does not require a special data structure:
  - ♦ blocks containing the pointers to the free blocks can be stored in free blocks

## ■ disadvantages

- ♦ Allocating large areas is costly
- ♦ It is very difficult to allocate contiguous blocks

# FAT

- Free blocks are stored in a linked list (is a sort of special file)
- Very easy to implement



## ■ Advantages

- ♦ Does not require further space in main memory:
  - ♦ FAT is usually cached in main memory for faster file accesses

## ■ Disadvantages

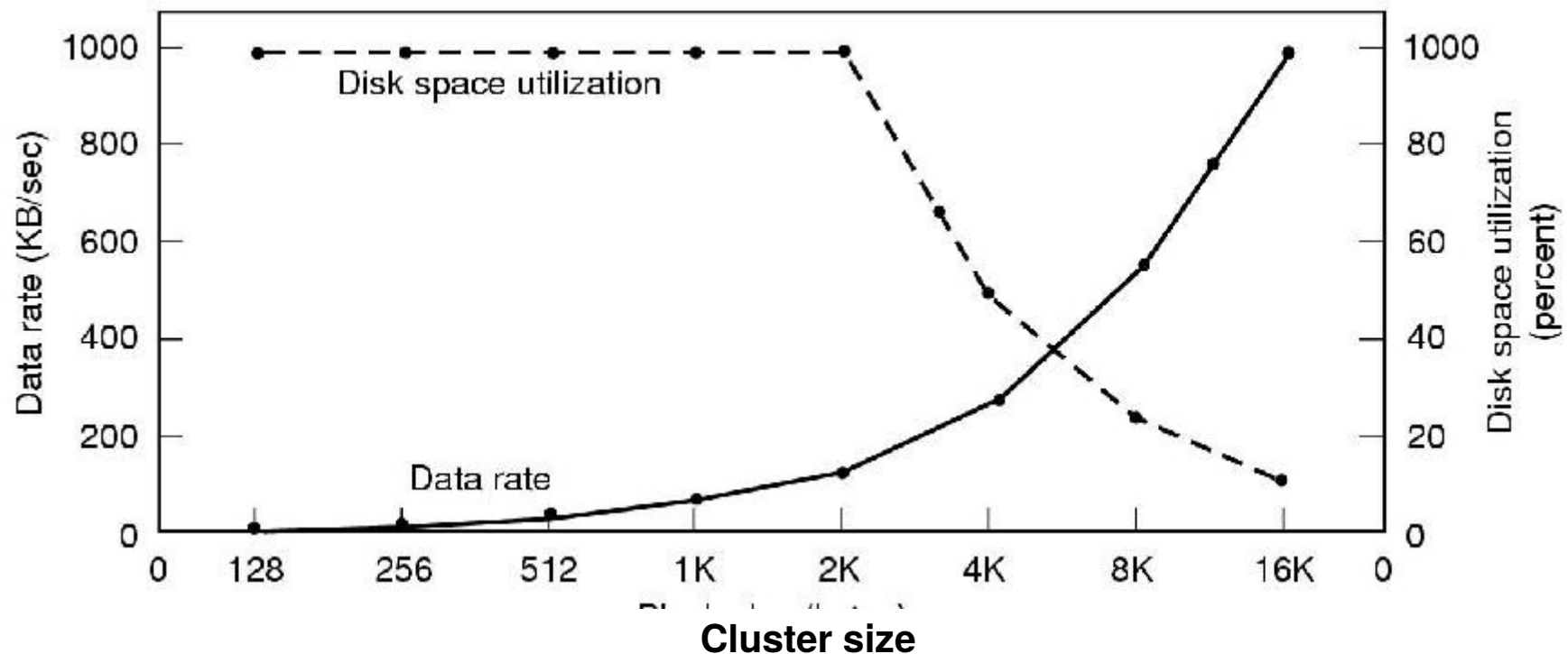
- ♦ Allocating large areas is costly
- ♦ It is very difficult to allocate contiguous blocks

# Cluster size

- Large clusters:
  - ♦ High read/write speed
  - ♦ High internal fragmentation
- Small clusters
  - ♦ Low read/write speed
  - ♦ Low internal fragmentation

# Example

- Suppose that **average** file size is 2 KB



# Directory Implementation

## ■ Directories

- are **special files** containing information about files and directories they contain
- have a certain number of **directory entries**
- Each directory entry contains the information needed to manage the corresponding file/directory:
  - name
  - attributes
  - Allocation information

## ■ Implementation choices:

- Where to store attributes?: directory entry or i-node
- array or hash table?

- directory entries contain all the information of the file/directory
  - ♦ used in MS-DOS

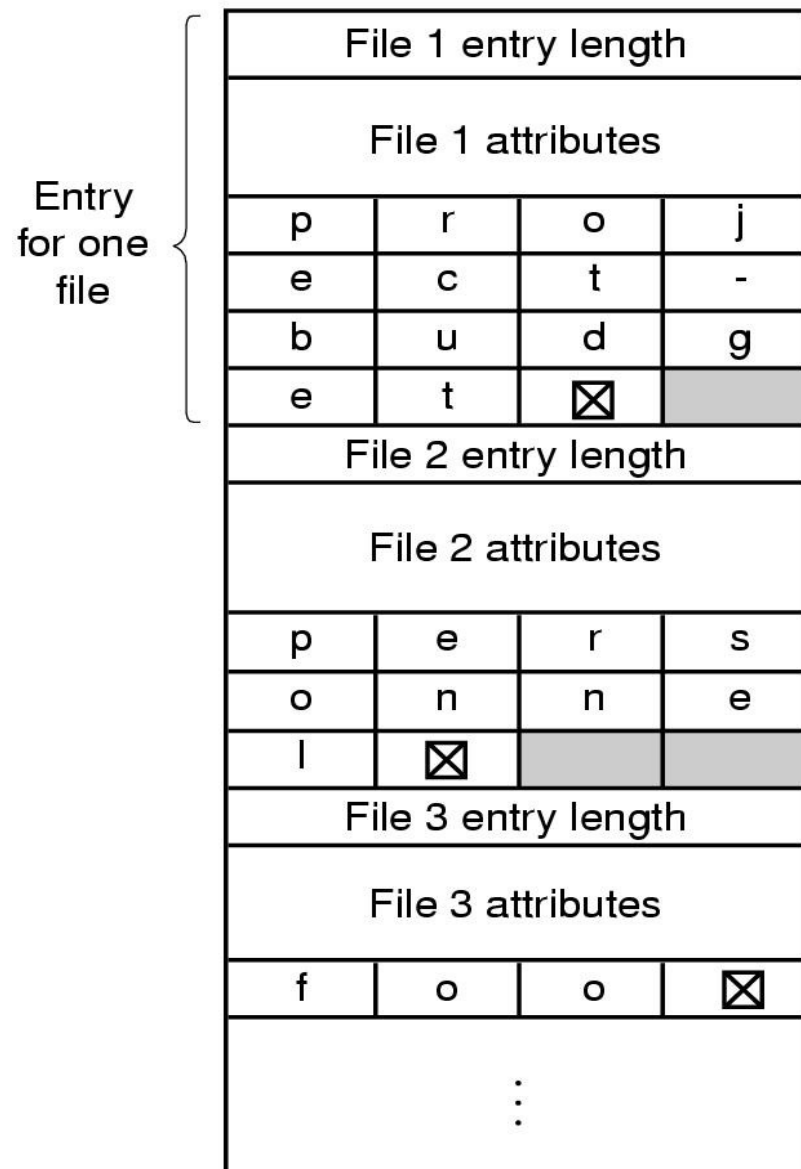
name
attributes
allocation info
name
attributes
allocation info

- file/directory informations are stored in the i-node
  - ♦ used in UNIX

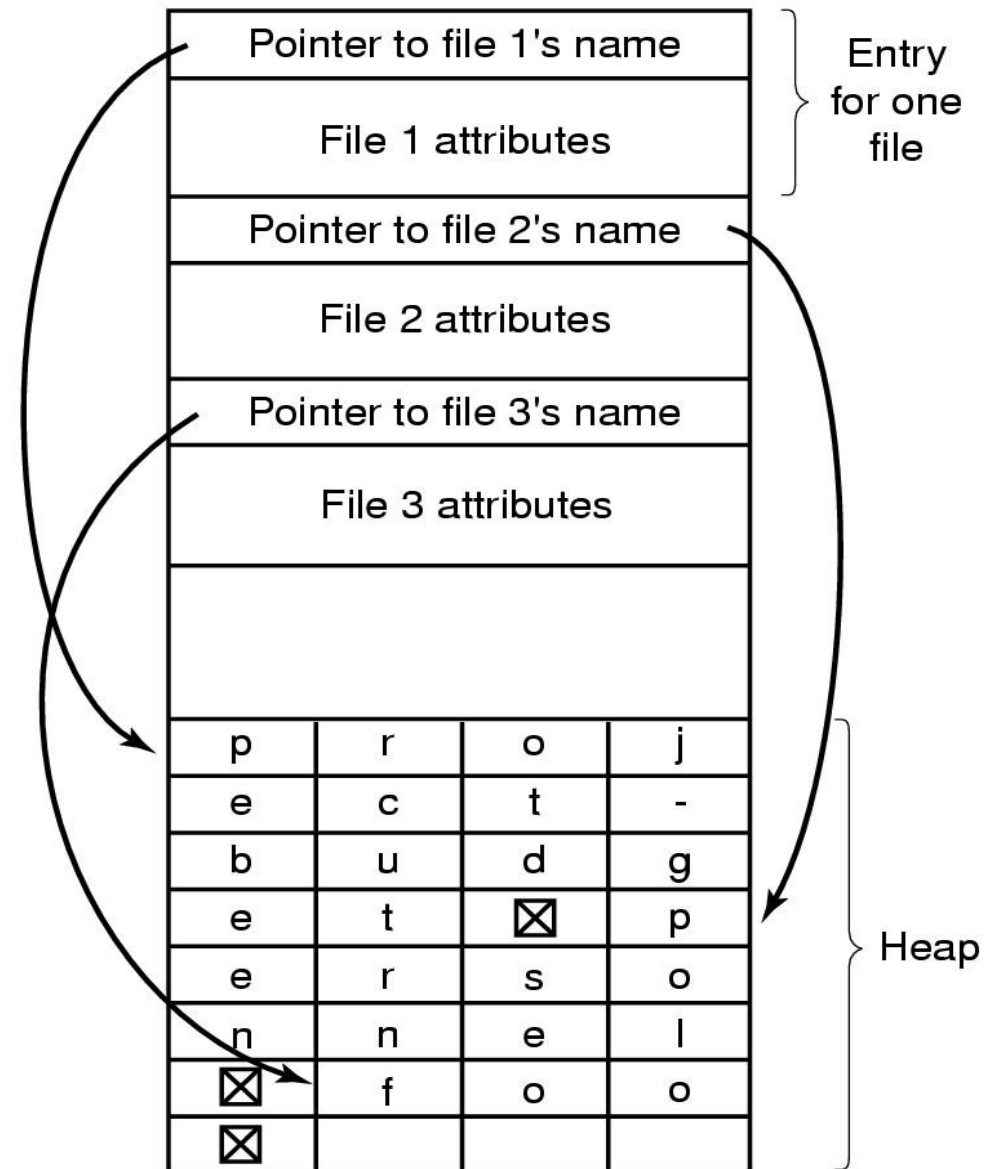
nome	i-node
nome	i-node
nome	i-node
nome	i-node

# Name length problem

- Fixed length
  - Simplest choice
  - Trad-off:
    - Too large reserved space: memory waste
    - Too small reserved space: too short names
- Variable length
  - more complex data structure



(a)



(b)

## ■ Array

- ♦ Simple to implement
- ♦ Inefficient for very large directories

## ■ Hash table

- ♦ More complex implementation
- ♦ Needs to choose some parameters:
  - ♦ Table size
  - ♦ A method for managing collisions

# DAG structure directories

- Two possible implementations:

- ♦ Symbolic links
- ♦ hard links

- **Symbolic links**

- ♦ Special directory entries, contain the absolute path to the file
- ♦ file access:
  - ♦ The file is searched in the directory
  - ♦ It is found that it is a symbolic link
  - ♦ The link is **resolved**: the absolute path is used to access the file

## ■ Hard link

- ♦ File informations are copied in both directories
- ♦ More efficient:
  - ♦ does not need two searches in the file system
- ♦ It is impossible to distinguish the original file from the copies
- ♦ Implementation
  - ♦ Needs i-node like implementation
  - ♦ i-node contain a reference counter

- Several file systems do not implement DAG structures (links)

**Example:** MS-DOS

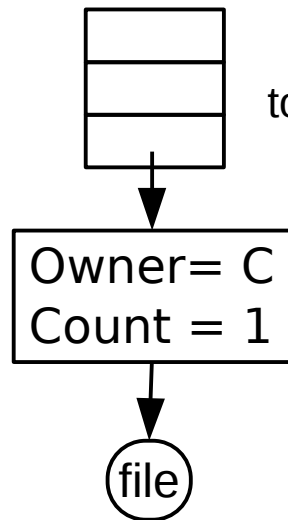
# Hard-link: example

- DAG structures are more flexible than tree structure, but it creates many problems

## EXAMPLE

- an OS has three users, A, B, C

Directory of C

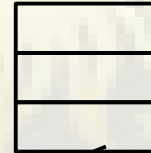


user B creates a hard link  
to a file in the directory of C

directory of C



directory of B

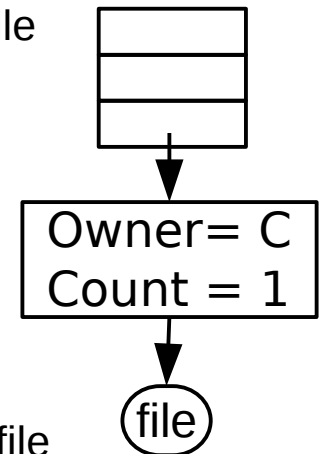


Owner= C  
Count = 2

file

user C deletes the file  
in its directory

Directory di B



now (only) B dir contains a file  
whose owner is C!

# Performance improvements

## Main memory

recently used  
blocks cache

File descriptor table  
of the open files

DMA block buffer

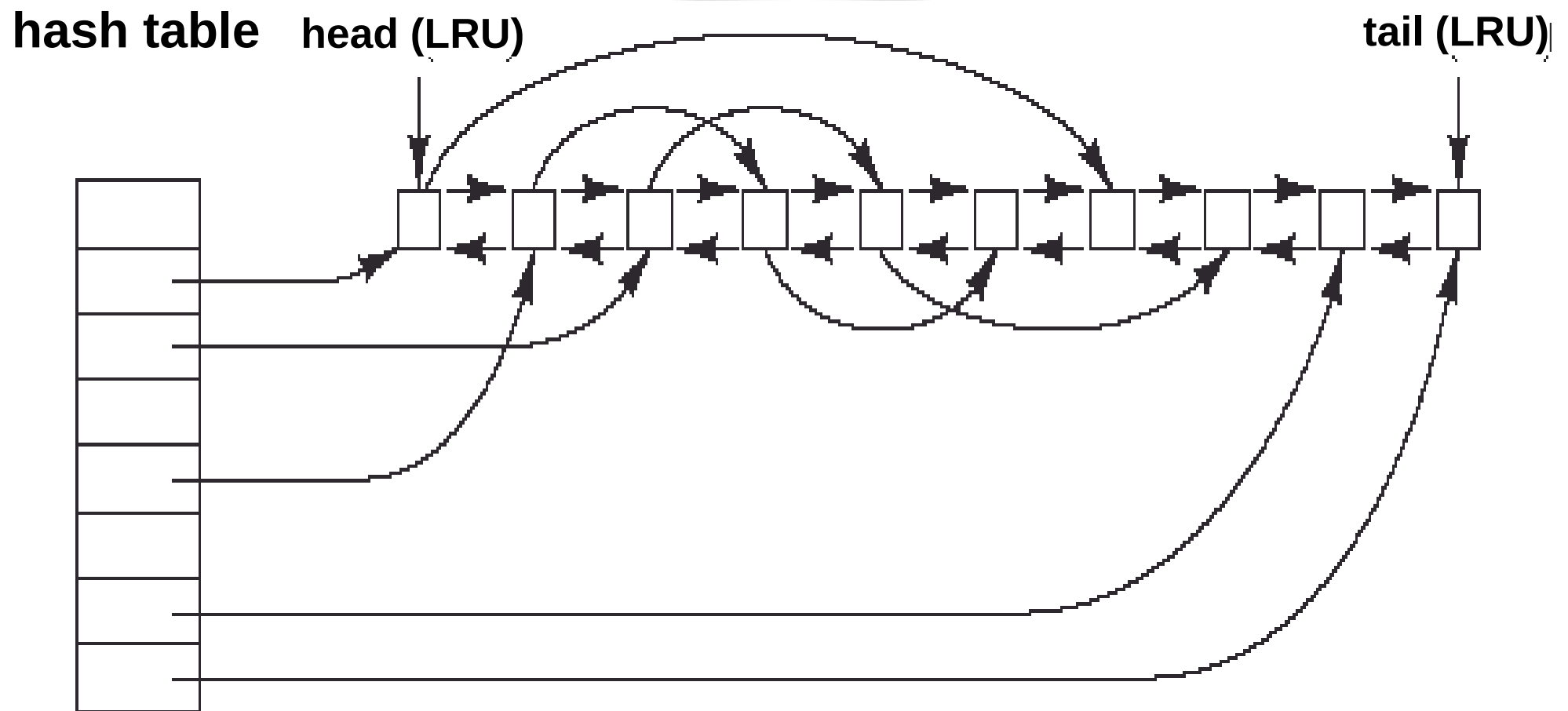
## controller

track buffer

# Performance

- Adding instructions to the execution path to save one disk I/O is a very good solution:
  - ♦ Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
  - ♦ Typical disk drive at 250 I/Os per second
  - ♦  $159,000 \text{ MIPS} / 250 = 630$  million instructions during one disk I/O
  - ♦ Fast SSD drives provide 60,000 IOPS
  - ♦  $159,000 \text{ MIPS} / 60,000 = 2.65$  million instructions during one disk I/O

# Block cache



# Block cache management

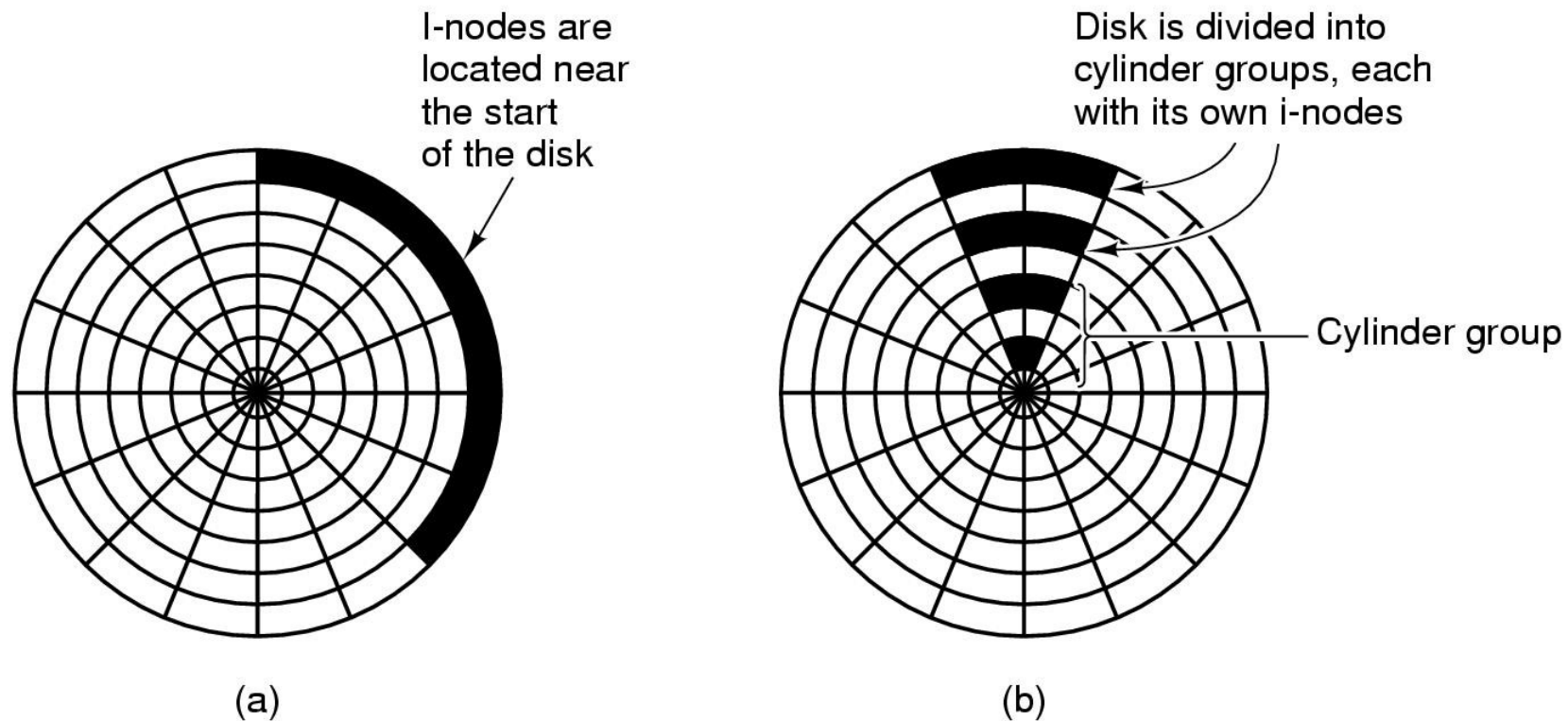
- Different types of blocks:
  - ♦ Data blocks
  - ♦ System blocks (contain i-nodes or directories)
  - ♦ Partially full blocks
- System blocks are usually updated as soon as possible
- Users can force updating (buffer flush operations)
- **UNIX**
  - ♦ A system daemon periodically (some secs) updates data blocks
- **Windows**
  - ♦ Blocks are updated as soon as possible: safe, but requires much more disk I/O operations

# Read-ahead strategy

- Another strategy for performance improvement consists in trying to move up (predict) the loading of the next blocks, before they are actually requested
- Possible strategy:
  - ♦ When the block  $K$  is requested, also loads the block  $K+1$
  - ♦ Works well for sequential access
- for each file, the access pattern could be stored, in order to predict whether the read-ahead strategy is useful or not.

# I-node table positioning

- The optimal positioning of the i-node table allows reducing disk arm movements



# Coherence cache block techniques

- Caching mechanisms may cause inconsistency problems

## Example

Sudden power failures

- This problem is particularly serious for blocks containing FAT, bitmap, i-node and directories
- Two solutions:
  - ♦ healing (file system checker)
    - ♦ **fsck**, **scandisk**
  - ♦ preventing (journaling file system)
    - ♦ ext3, reiserfs

# Block coherence checking

- Two tables (initialized to 0) each entry represents a disk block:
  - **used[i]:**
    - counts how many times the block i is present in the block list of a file
    - Obtained by analyzing all files in the file system to be checked
  - **free[i]**
    - counts how many times the block i is present in the free block list
    - Can be obtained by:
      - scanning the free block list, or
      - copying the bitmap

- **$\text{free}[i] + \text{used}[i] = 1$** 
  - The block is ok
- **$\text{free}[i] + \text{used}[i] = 0$** 
  - Missing block: neither free or used
  - **problem:** wasted space, but no coherence problem
  - **action:** move the block back to the free block list
- **$\text{free}[i] = N \ (N > 1, \text{used}[i] = 0)$** 
  - It appears N times in the free block list
  - **problem:** no coherence problem (at the moment), but the block could be assigned again (up to N times)
  - **action:** keeping only one instance

## ■ **free[i] = 1, used[i] = 1**

- ♦ The block is either free and used
- ♦ **problem:** no coherence problem (at the moment), but the block could be assigned again
- ♦ **action:** remove the block from the free block list

## ■ **used[i] = N ( N > 1) free[i] = 0**

- ♦ the block is being used N times
- ♦ **problem:** coherence problem!
- ♦ **actions:**
  - ♦ create N copies of the block
  - ♦ Assign each block to a different file

# File system and crash

- Deleting a file implies:
  - ♦ removing the associated entry from the parent directory
  - ♦ restoring the i-node in the free i-node list
  - ♦ moving back in the free block list all the blocks used by the file
- What happens if the system crashes (e.g. power failure) while one of these operations is being executed?

# Journaling-based file systems

- Every transaction is stored in a log file (the journal)
- A transaction is considered completed (committed) once it has been stored in the log file. Nonetheless the file system could be not updated yet
- Periodically, all transactions in the log file are actually executed
- Once executed the transactions are removed from the log file

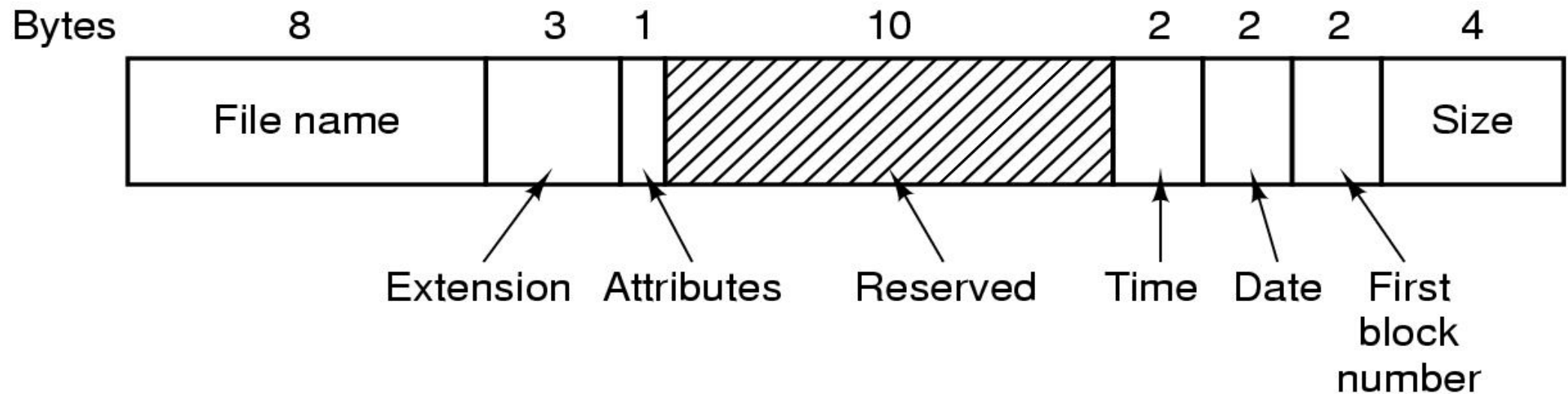
## NOTE

Before being executed, log file transactions are read again in order to check their integrity

# Example: the MS-DOS file system

## ■ File names:

- 8+3 characters
- Block indices: 12-16 bits (MS-DOS, W95), 32 bits (W98)



block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8MB	128 MB	
4 KB	16MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

# Example: the UNIX V7 file system

Root directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up  
usr yields  
i-node 6

I-node 6  
is for /usr

Mode size times
132

I-node 6  
says that  
/usr is in  
block 132

Block 132  
is /usr  
directory

6	.
1	..
19	dick
30	erik
51	jim
26	ast
45	bal

/usr/ast  
is i-node  
26

I-node 26  
is for  
/usr/ast

Mode size times
406

I-node 26  
says that  
/usr/ast is in  
block 406

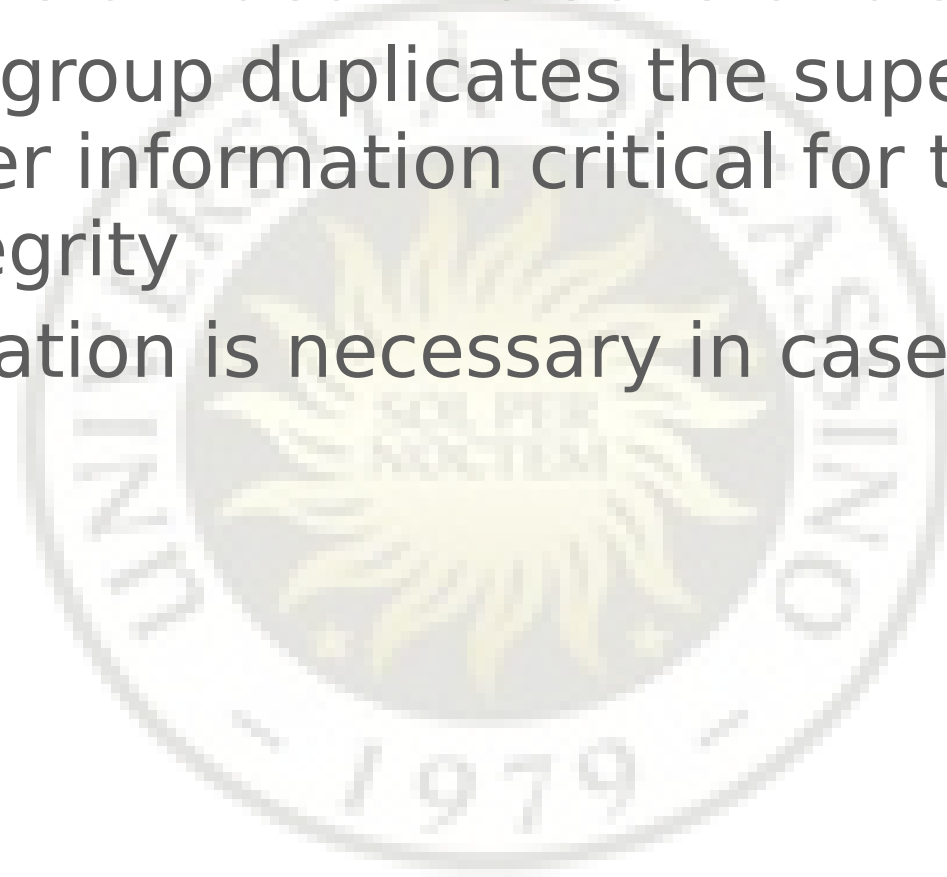
Block 406  
is /usr/ast  
directory

26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

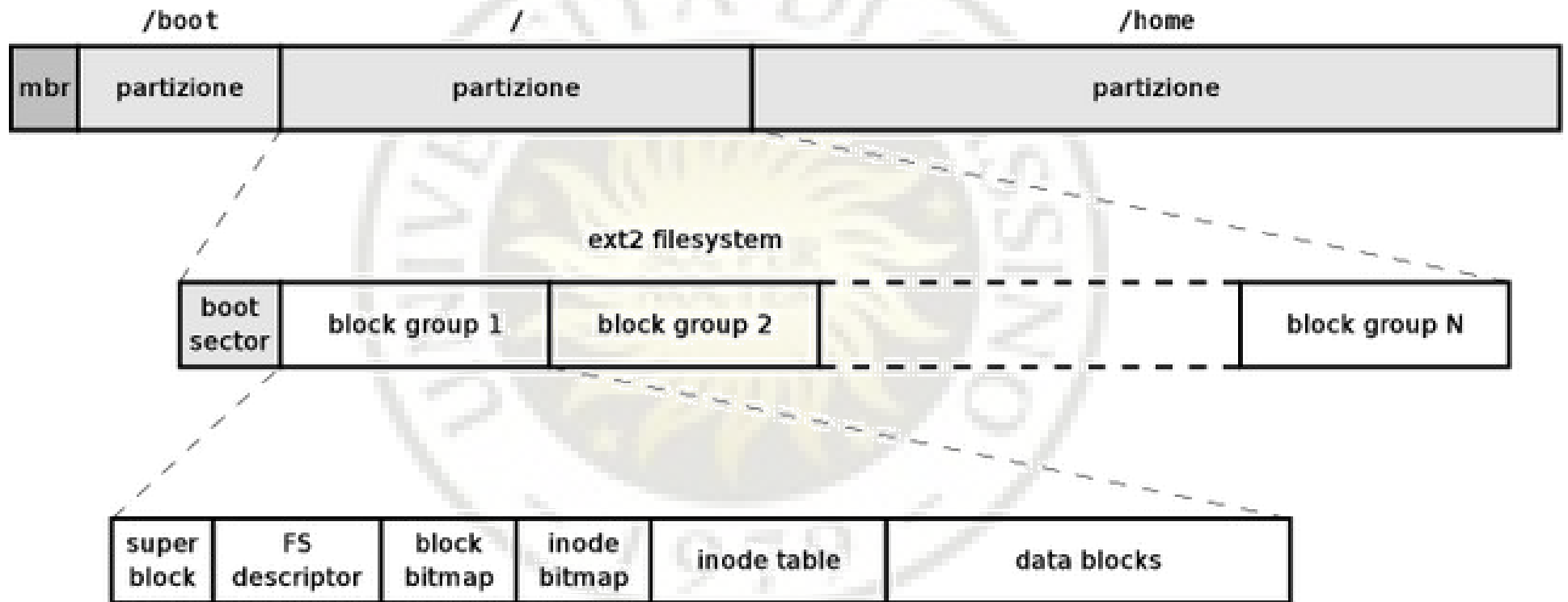
/usr/ast/mbox  
is i-node  
60

# Ext2 file system

- Partitions are divided into several block groups.
- Each block group duplicates the superblock, as well as other information critical for the file system integrity
- This information is necessary in case of disaster recovery



# Ext2 file system



# Superblock

- The superblock contains the following informations:
  - ♦ **Magic number:** allows to check that this is indeed the Superblock for an EXT2 file system. For the current version of EXT2 this is 0xEF53
  - ♦ **Revision Level:** The major and minor revision levels allow the mounting software to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting software to determine which new features can safely be used on this file system

# Block group descriptors

- Each Block Group has a data structure describing it. Like the Superblock, all the group descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption. Each Group Descriptor contains the following information:
  - ♦ **Blocks Bitmap:** The block number of the block allocation bitmap for this Block Group. This is used during block allocation and deallocation
  - ♦ **Inode Bitmap:** The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation,
  - ♦ **Inode Table:** The block number of the starting block for the inode table for this Block Group. Each inode is represented by the EXT2 inode data structure described below.

- ♦ **Free blocks count, Free Inodes count, Used directory count**
- The group descriptors are placed one after another and together they make the group descriptor table. Each Blocks Group contains the entire table of group descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the EXT2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.