# File Systems


Carsten Griwodz
University of Oslo


(includes slides from Pål Halvorsen, Kai Li, A. Tanenbaum and M. van Steen)

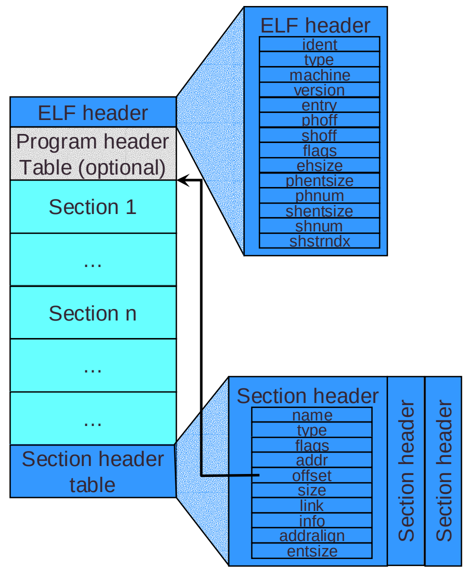
# File Examples

- Text file
  - Example ASCII

```
Tags Tables
===========
A "tags table" is a description of how a multi-file program is
     broken up into files.  It lists the names of the component
     files and the names and positions of the functions (or other
     named subunits) in each file.  Grouping the related files
     makes it possible to search or replace through all the files
     with one command.  Recording the function names and positions
     makes possible the `M-.' command which finds the definition of
     a function by looking up which of the files it is in.

Tags tables are stored in files called "tags table files".  The
     conventional name for a tags table file is `TAGS'.

Each entry in the tags table records the name of one tag, the name
     of the file that the tag is defined in (implicitly), and the
     position in that file of the tag's definition.

Just what names from the described files are recorded in the tags
     table depends on the programming language of the described
     file. They normally include all functions and subroutines, and
     may also include global variables, data types, and anything
     else convenient.  Each name recorded is called a "tag".
* Menu:
* Tag Syntax::                Tag syntax for various types of code
     and text files.
* Create Tags Table::         Creating a tags table with `etags'.
* Select Tags Table::         How to visit a tags table.
* Find Tag::                  Commands to find the definition of a
     specific tag.
* Tags Search::               Using a tags table for searching and
     replacing.
* List Tags::                 Listing and finding tags defined in a
     file.


File: emacs,  Node: Tag Syntax,  Next: Create Tags Table,  Up: Tags
```

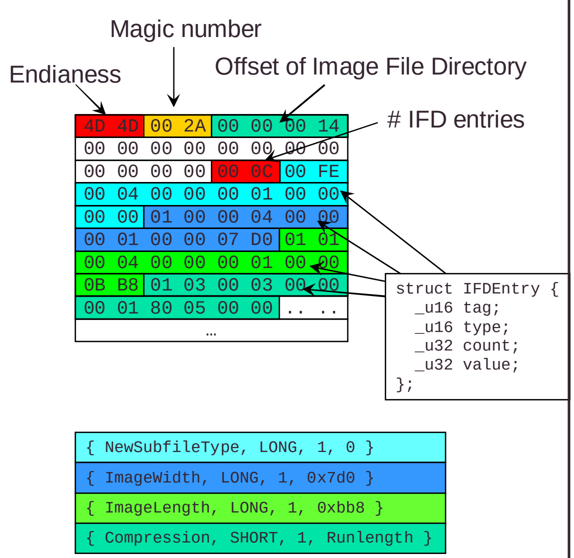
# File Examples

- Text file
  - Example ASCII
- Program file
  - Example ELF




# File Examples

- Text file
  - Example ASCII
- Program file
  - Example ELF
- Image file
  - Example TIFF



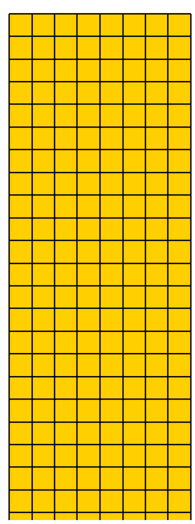
# File Examples

- Text file
  - Example ASCII
- Program file
  - Example ELF
- Image file
  - Example TIFF
- Archive file
  - Example tar
- Video file
  - Example MPEG
- Database
  - Example Berkeley DB format

- Files have *types* and *structure*

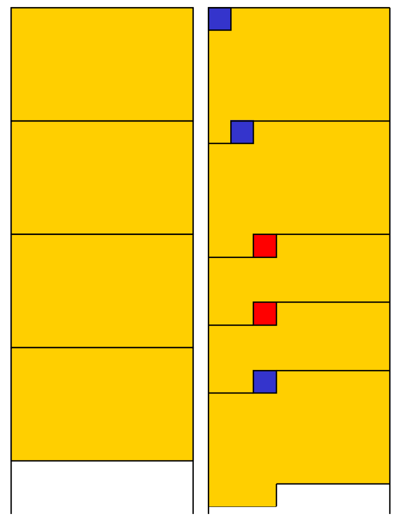
# Files

- Unstructured files
  - Low-level files
- Structured files



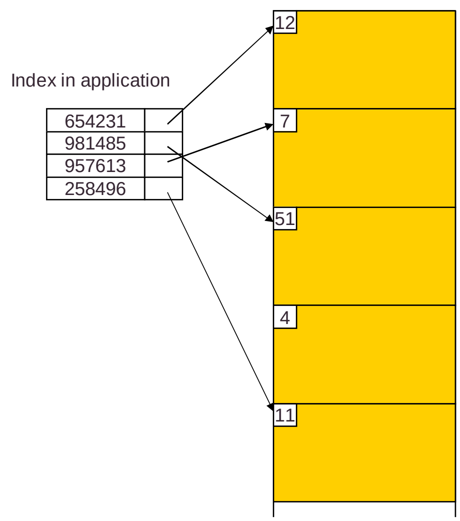
# Files

- Unstructured files
  - Low-level files
- Structured files
  - Record-oriented sequential files



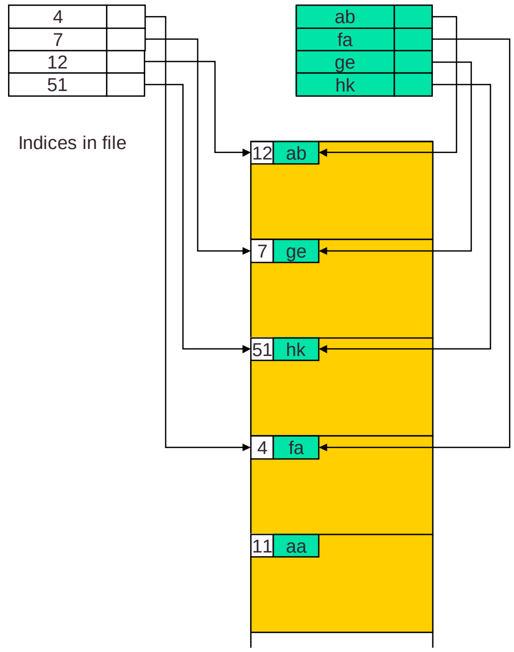
# Files

- Unstructured files
  - Low-level files
- Structured files
  - Record-oriented sequential files
  - Indexed sequential files



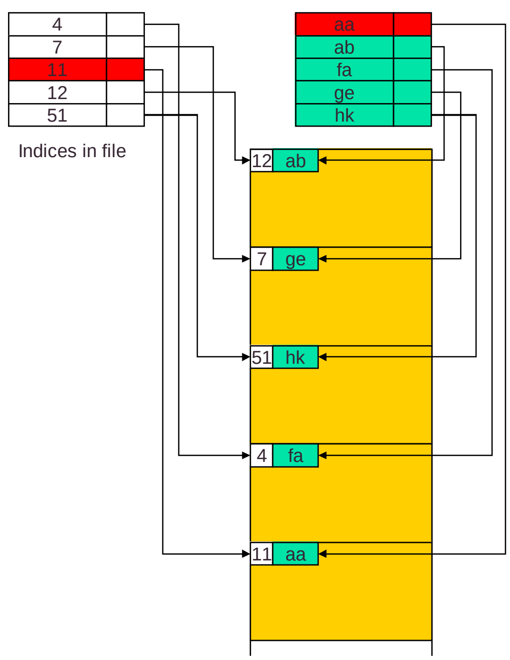
# Files

- Unstructured files
  - Low-level files
- Structured files
  - Record-oriented sequential files
  - Indexed sequential files
  - Inverted files




# Files

- Unstructured files
  - Low-level files
- Structured files
  - Record-oriented sequential files
  - Indexed sequential files
  - Inverted files



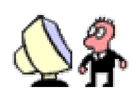
# Files

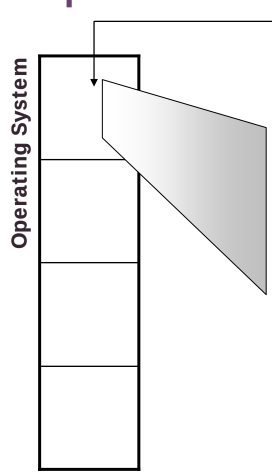
- Unstructured files
  - Unix
  - Windows
- Structured files
  - MacOS (to some extent)
  - MVS

- In this course we consider unstructured files

# Open

`open(name, mode)`



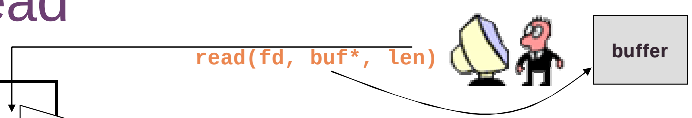


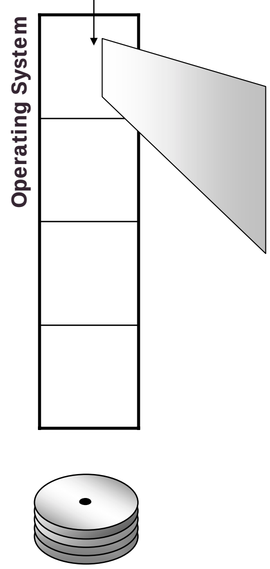

sys_open() → vn_open():

1. Check if valid call
2. Allocate file descriptor
3. If file exists, open for read. Otherwise, create a new file. Must get directory inode. May require disk I/O.
4. Set access rights, flags and pointer to vnode
5. Return index to file descriptor table

**fd**

# Read



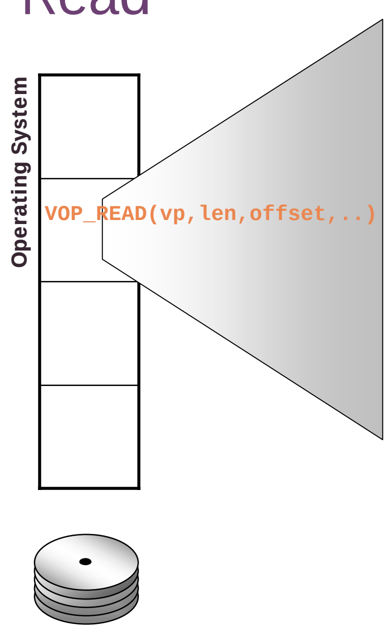




sys_read() → dofileread() → (*fp_read==vn_read)():

1. Check if valid call and mark file as used
2. Use file descriptor as index in file table
   to find corresponding file pointer
3. Use data pointer in file structure to find vnode
4. Find current offset in file
5. Call local file system

`VOP_READ(vp,len,offset,..)`

# Read



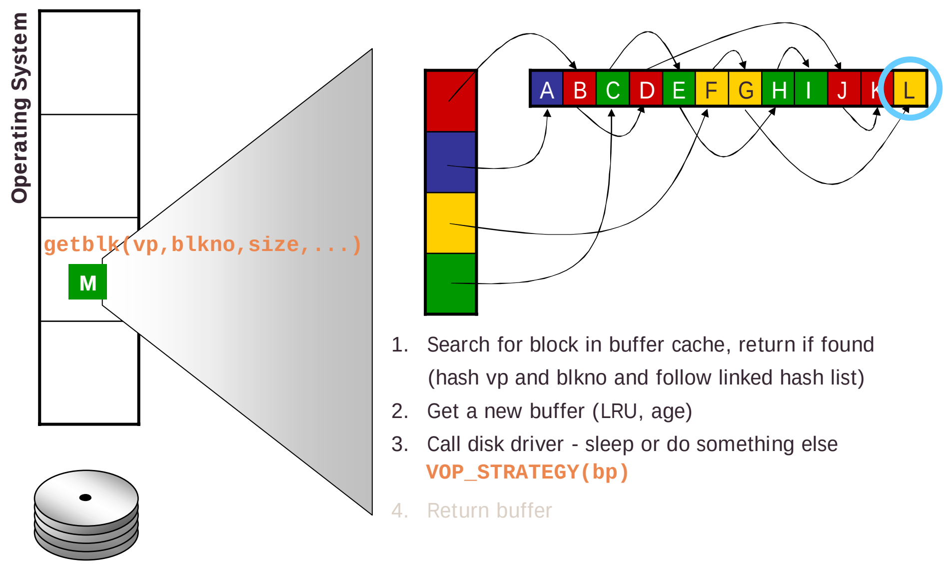

VOP_READ(...) is a pointer to a read function in the corresponding file system, e.g., Fast File System (FFS)

READ():

1. Find corresponding inode
2. Check if valid call - file size vs. len + offset
3. Loop and find corresponding blocks
   - find logical blocks from inode, offset, length
   - do block I/O, fill buffer structure
     e.g., bread(...) → bio_doread(...) → getblk()

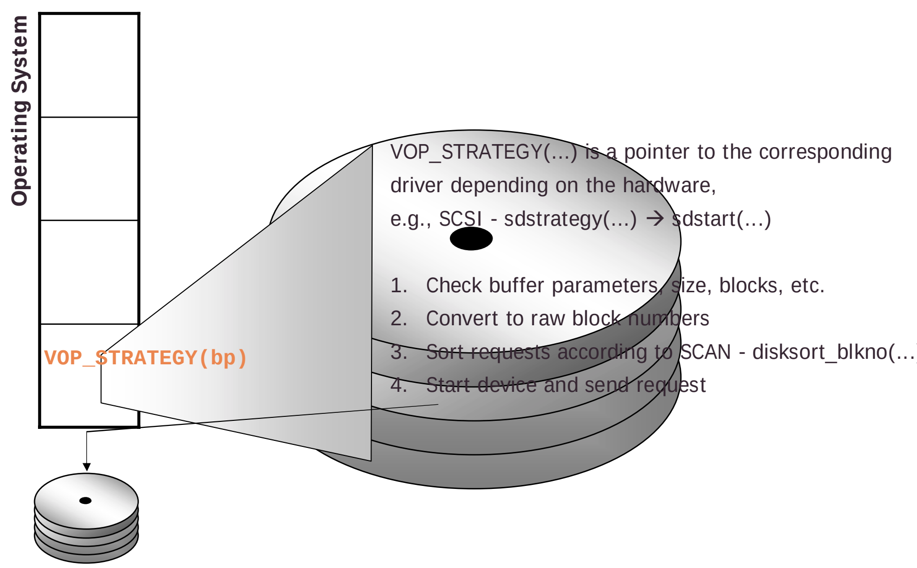
     `getblk(vp,blkno,size,...)`
   - return and copy block to user

# Read




1. Search for block in buffer cache, return if found (hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else **VOP_STRATEGY(bp)**
4. Return buffer

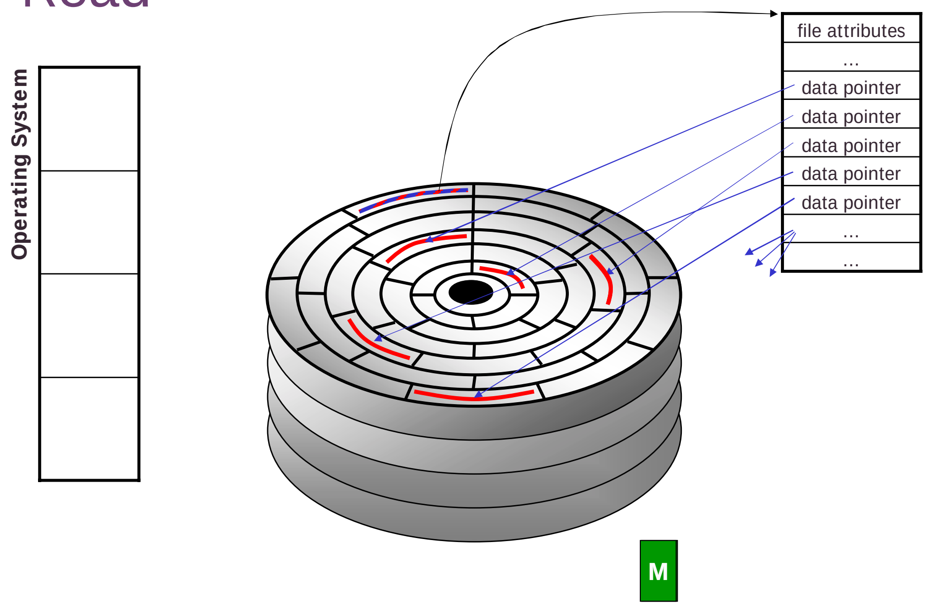
# Read




VOP_STRATEGY(...) is a pointer to the corresponding driver depending on the hardware, e.g., SCSI - sdstrategy(...) → sdstart(...)

1. Check buffer parameters, size, blocks, etc.
2. Convert to raw block numbers
3. Sort requests according to SCAN - disksort_blkno(...)
4. Start device and send request

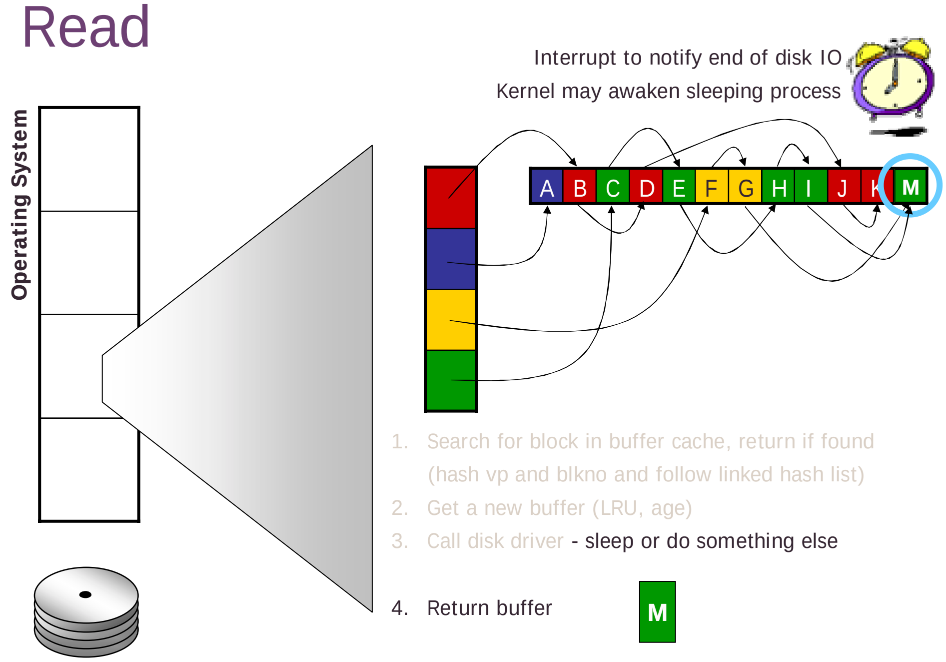
# Read




# Read

Interrupt to notify end of disk IO

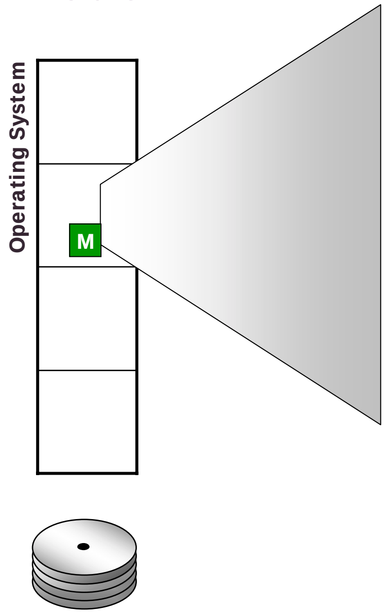
Kernel may awaken sleeping process

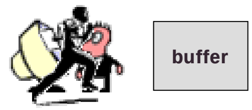



1. Search for block in buffer cache, return if found (hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else

4. Return buffer

## Read




READ():

1. Find corresponding inode
2. Check if valid call - file size vs. len + offset
3. Loop and find corresponding blocks
   - find logical blocks from inode, offset, length
   - do block I/O,
     e.g., bread(...) → bio_doread(...) → getblk()
   - return and copy block to user

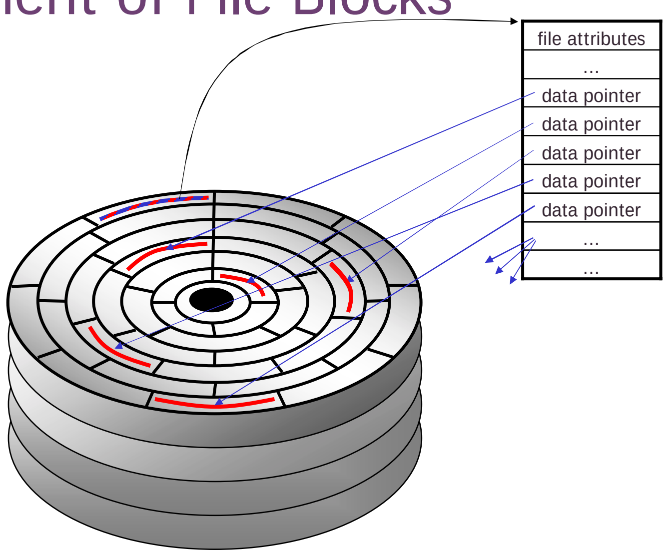
## Files

- Regular files
- Special files
  - Directories
  - Hard links
  - Soft links
  - ...

# File Systems

- Handle files on disk
  - Directly
    - Diskettes, CDs
  - In partitions
    - Typical
  - In logical volumes
    - Abstraction layer between
      - One or more disks
      - Partitions
- Have representations in
  - User space
  - Kernel space
- User space representation
  - File system API
    - E.g. VFS
  - File handle
  - Function calls
    - File: Create, delete, read, write, open, close, seek
    - Directory: Create, delete, list
- Kernel space representation
  - Map of file handle to file information
    - File attributes
    - Buffers in memory
    - Information about placement on disk

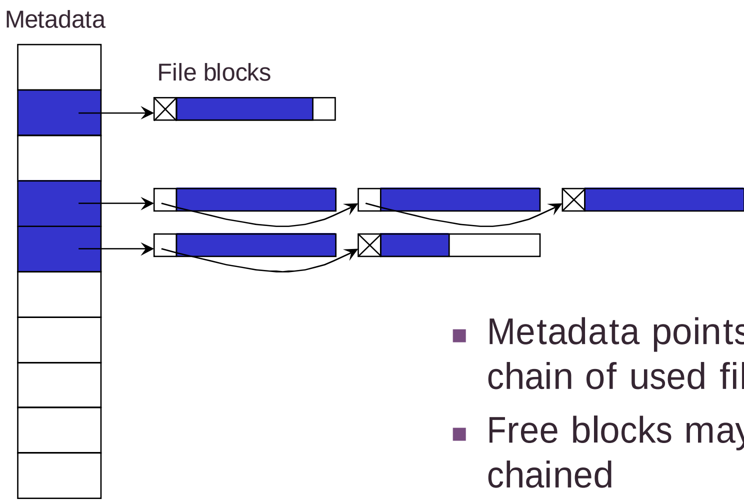
# Management of File Blocks



# Management of File Blocks

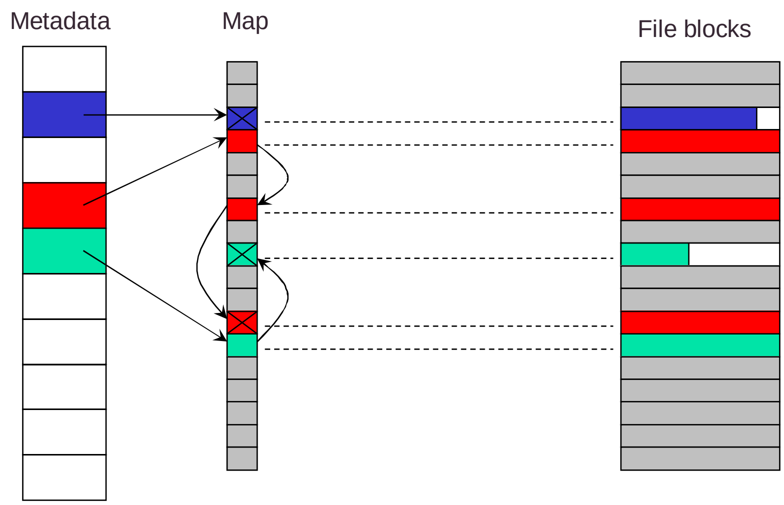
- Many files consist of several blocks
  - Relate blocks to files
  - Maintain order of blocks
- Approaches
  - Chaining in the media
  - Chaining in a map
  - Table of pointers
  - Extent-based allocation

# Chaining in the Media



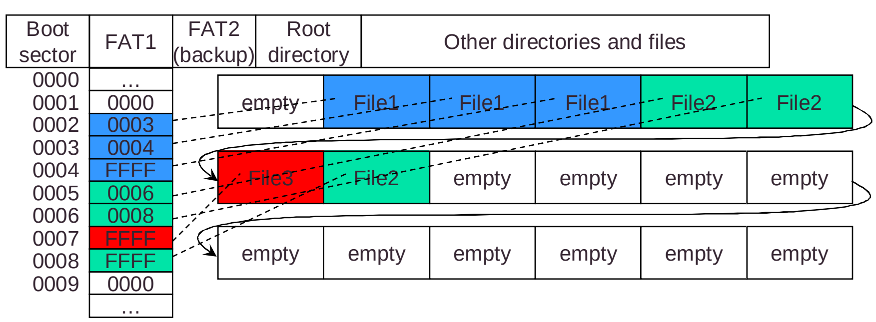

- Metadata points to chain of used file blocks
- Free blocks may also be chained

## Chaining in a Map

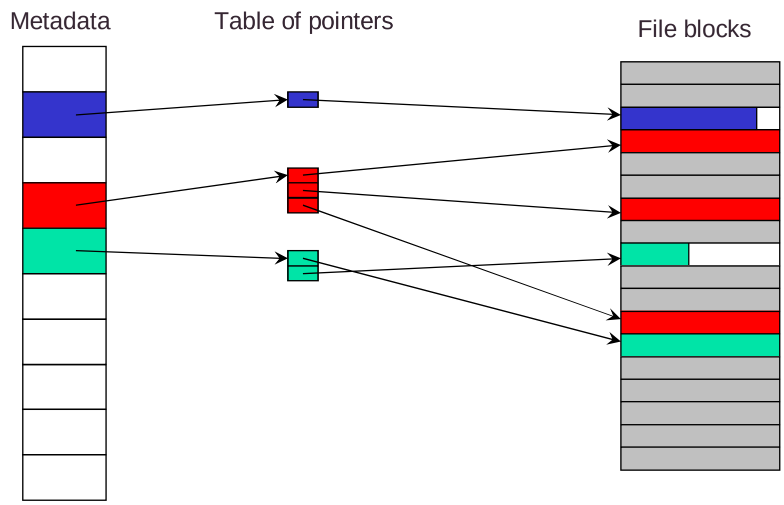



## FAT Example

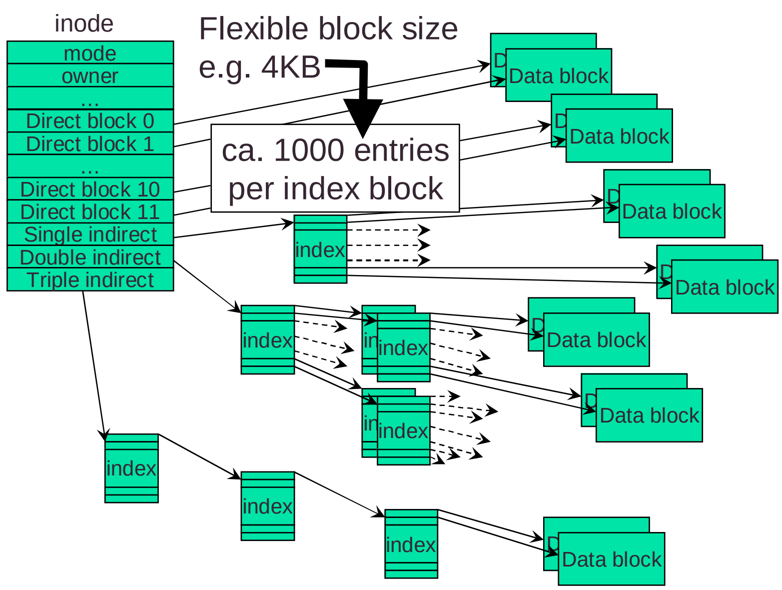
- FAT: File Allocation Table
- Versions FAT12, FAT16, FAT32
  - Number indicates number of bits used for identifying blocks in partition ($2^{12}$, $2^{16}$, $2^{32}$)
  - FAT12: Block sizes 512 bytes – 8 KB: max 32 MB partition size
  - FAT16: Block sizes 512 bytes – 64 KB: max 4 GB partition size



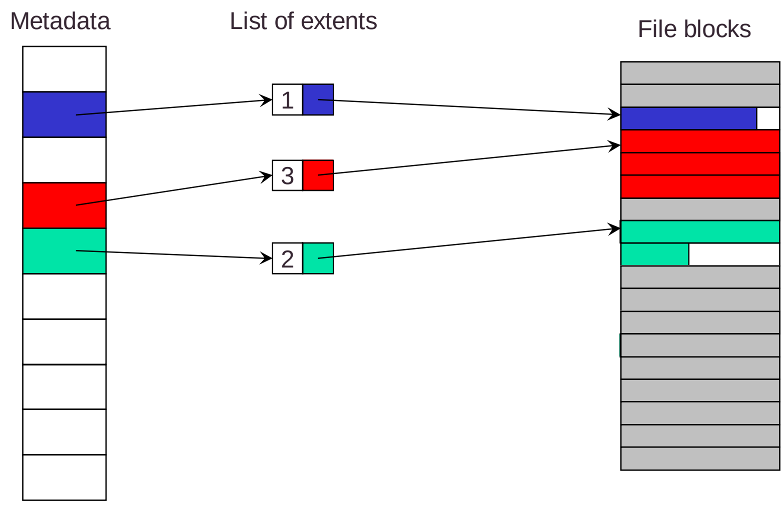
# Table of Pointers



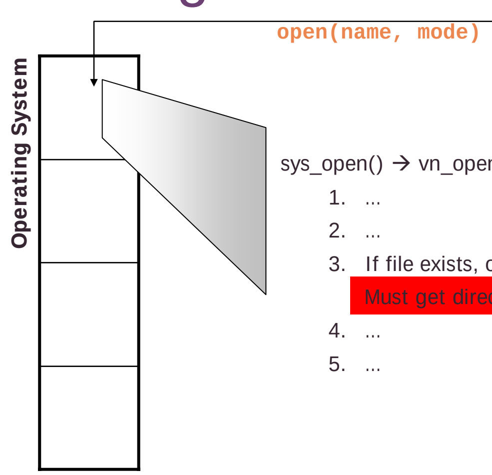

# Unix Examples

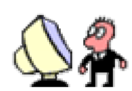


# Extent-based Allocation




# Finding Files



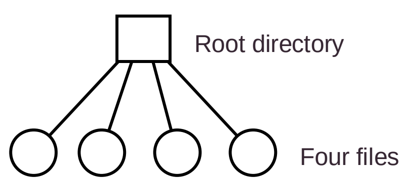
sys_open() → vn_open():

1. ...
2. ...
3. If file exists, open for read. Otherwise, create a new file. Must get directory inode. May require disk I/O.
4. ...
5. ...

## Arrangement of Directories

- Single-level directory systems
- Hierarchical directory systems
- Shared files
  - Hard links
  - Soft links

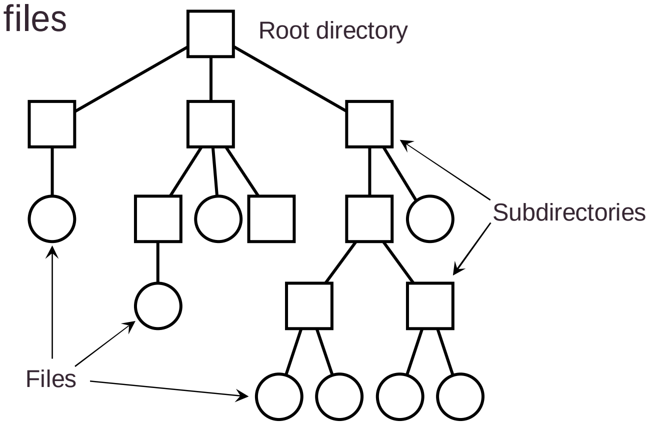
## Single-level Directory Systems




- CP/M
  - Microcomputers
  - Single user system
- VM
  - Host computers
  - “Minidisks”: one partition per user

# Hierarchical Directory Systems

- Tree structure
  - Nodes = directories, root node = root directory
  - Leaves = files

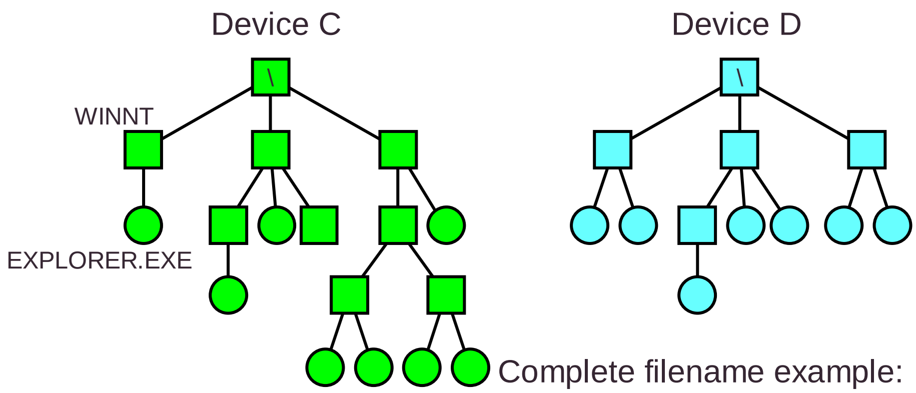



# Hierarchical Directory Systems

- Directories
  - Are stored on disk
  - Need attributes just like files
  - Subdirectories need names
- To access a file
  - Must test all directories in path for
    - Existance
    - Being a directory
    - Permissions

## Hierarchical Directory Systems

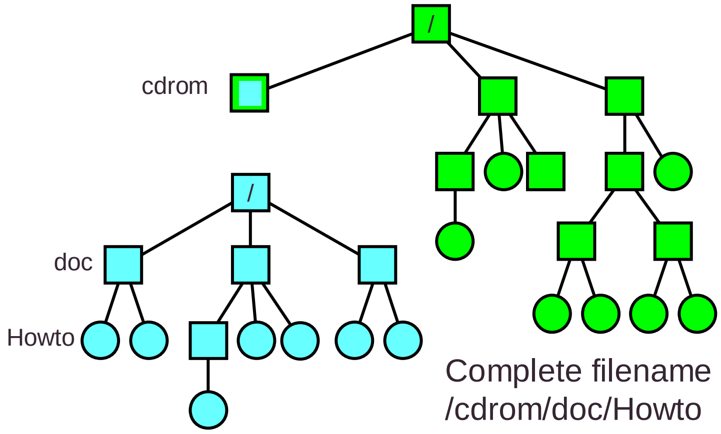
- Windows: one tree per partition or device




Complete filename example:
C:\WinNT\EXPLORER.EXE

## Hierarchical Directory Systems

- Unix: single acyclic graph spanning several devices




Complete filename example:
/cdrom/doc/Howto

# Directories

- Map names to file indices

- Data structures
  - Linear list
  - Trees
  - Hash tables

- Trade-off
  - Complexity
  - Efficiency

# Linear list

- Method
  - Store (filename, I-node) pairs linearly in a file
  - Create, delete a file
    - Search for the file name
    - Add a file (to the unused slot of the end)
    - Remove a file from the directory (with or without compaction)
- Pros
  - Relatively simple
  - Create effort is O(1)
- Cons
  - Linear search effort is O(n)

# Tree data structures

- Method
  - Sort the file by name
  - Store in a tree data structure such as B-tree
  - Create, delete, search in the tree data structure
- Pros
  - Efficient for a large number of files
  - Worst case effort is O(log n)
- Cons
  - Complex
  - Not necessarily efficient for a small number of files
  - Requires more space
  - Create effort is O(log n)

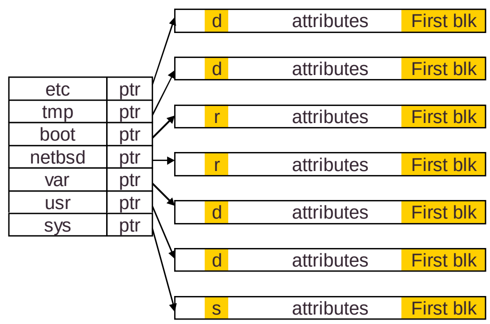
# Hashing

- Method
  - A linear list stores the directory entries
  - A hash table hashing a name to an i-node (standard implementation plus space management for directory entries)
- Pros
  - Fast searching and relatively simple
  - Hash function and few files make average search effort O(1) possible
  - Create effort is O(1)
- Cons
  - Not as efficient as trees for very large directory
  - Worst case search effort is O(n)

# Naming and Attributes

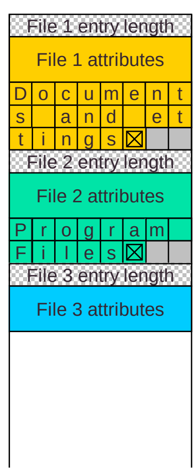
- Directory entries must
  - Allow fast name matching
  - Refer to attributes
- Attributes
  - File type
  - Ownership
  - Access rights
  - Sizes: Current size, max size
  - Flags: System, hidden, temporary, archived (dirty)
  - Times: Creation, last modified, last accessed
- Attributes
  - Stored as part of the file itself
  - Stored in additional structure

| | | | |
|---|---|---|---|
| etc | d | attributes | First blk |
| tmp | d | attributes | First blk |
| boot | r | attributes | First blk |
| netbsd | r | attributes | First blk |
| var | d | attributes | First blk |
| usr | d | attributes | First blk |
| sys | s | attributes | First blk |



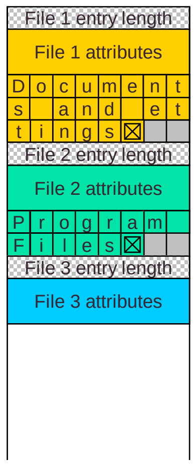

# Naming and Attributes

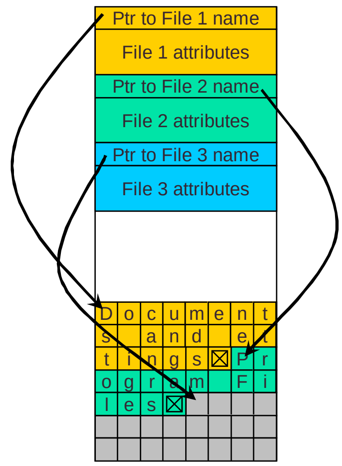
- Two ways of handling long file name in directory
  - In-line



# Naming and Attributes

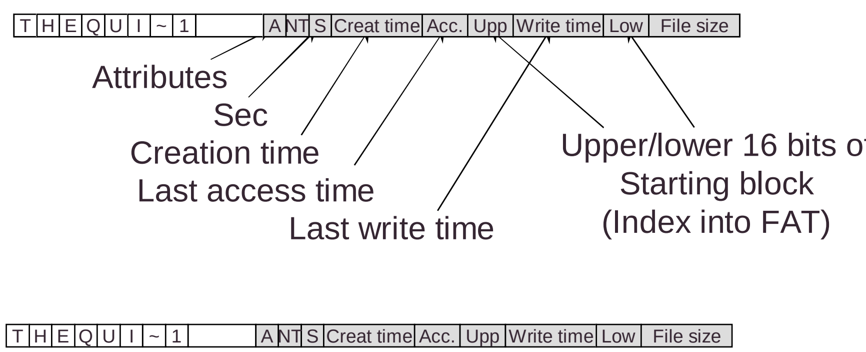
- Two ways of handling long file name in directory
  - In-line
  - In a heap




# Naming and Attributes

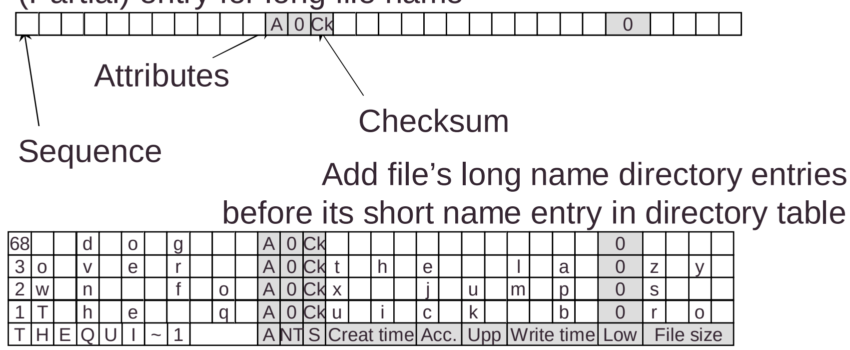
- Windows FAT file names
  - Example file name
    "The quick brown fox jumps over the lazy dog"



# Naming and Attributes

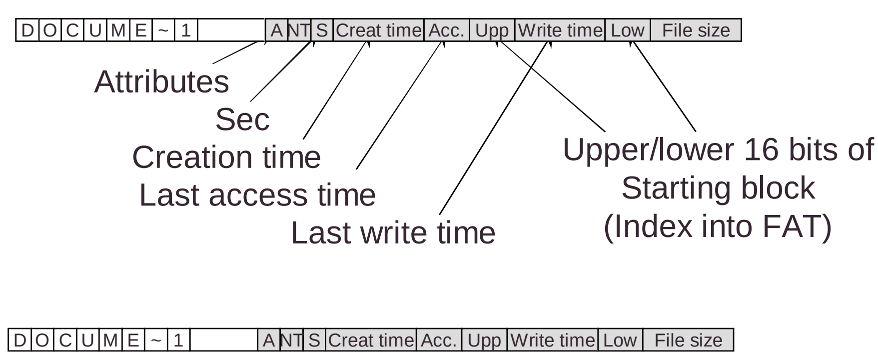
- Windows FAT file names
  - Example file name
    "The quick brown fox jumps over the lazy dog"

(Partial) entry for long file name




# Naming and Attributes

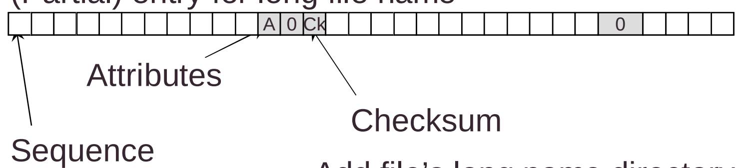
- Windows FAT file names
  - Example file name
    "Documents and Settings"



# Naming and Attributes

- Windows FAT file names
  - Example file name
    "Documents and Settings"

(Partial) entry for long file name

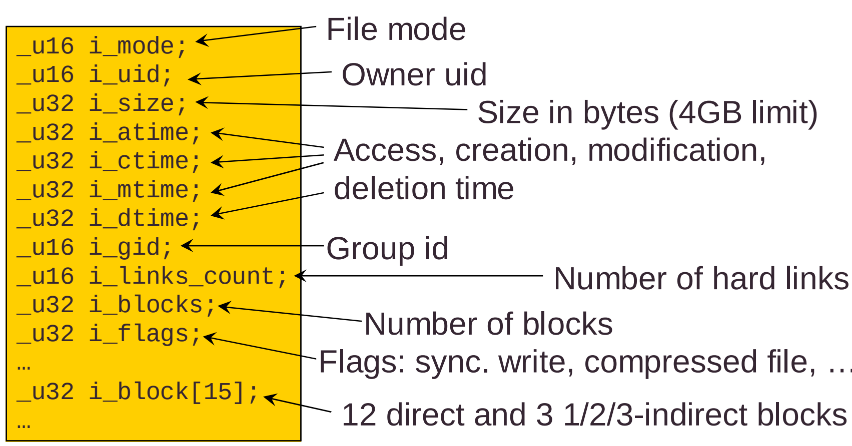



Add file's long name directory entries
before its short name entry in directory table

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 68 | | | S | | e | | t | | t | | A | 0 | Ck | i | | n | | g | | s | | | | | | 0 | | | | | |
| 1 | D | | o | | c | | u | | m | | A | 0 | Ck | e | | n | | t | | s | | | | a | | 0 | | n | | d | |
| D | O | C | U | M | E | ~ | 1 | | | | A | NT | S | Creat time | | | | Acc. | | Upp | | Write time | | | | Low | | File size | | | |

# Naming and Attributes

- Linux EXT2
  - Inode are similar to standard Unix inodes
  - Access to file of subdirectory requires access to data block



# Naming and Attributes

- Linux EXT2
  - Directory inodes refer to data blocks containing linear list of entries

File's (or subdirectory's) inode

Entry size (aligned)

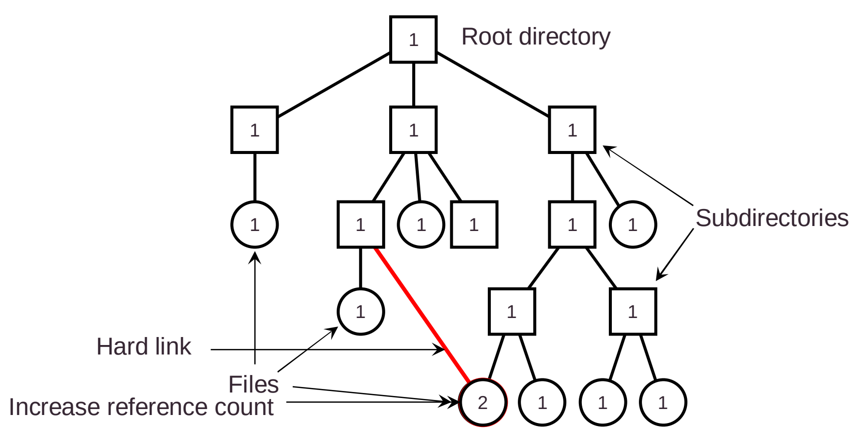
```
_u32 inode;
_u16 rec_len;
_u8  name_len;
_u8  file_type;
char name[255];
```

File's type, such as regular file, directory, symbolic link, …

File's name
max filename length is 255 bytes

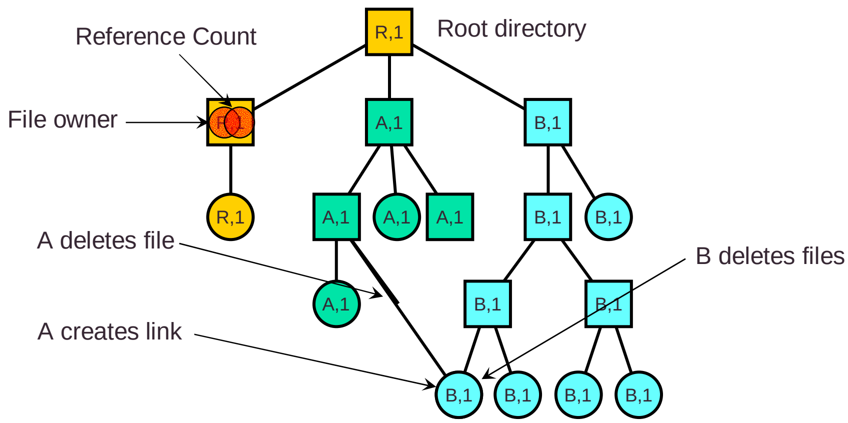
# Shared Files: Hard Link

- Hard Link
  - Break tree structure - maintain acyclic graph



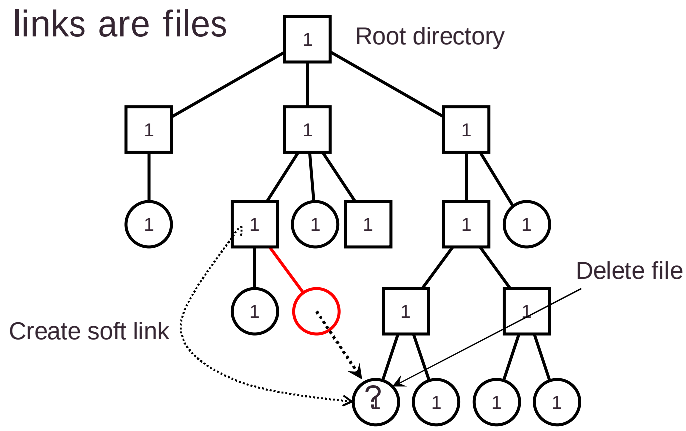
# Shared Files: Hard Link

- Hard Link
  - Files remain until all hard links are deleted




# Shared Files: Soft Link

- Soft Link
  - Tree structure not broken
  - Soft links are files



## Virtual File System Operations

- Many OSes support several file systems
  - Windows
    - FAT, NTFS, UFS, ...
  - Linux
    - Ext2, Ext3, ReiserFS, JFS, XFS, FAT, ...
- Same user space functions for all
  - POSIX API
  - Win32 API

```
file_operations {
        llseek
        read
        write
        readdir
        poll
        ioctl
        mmap
        open
        flush
     ...
};

inode_operations {
        create
        lookup
        link
        unlink
        symlink
        mkdir
        rmdir
        mknod
        rename
        readlink
        follow_link
        truncate
        permission
     ...
};
```

## Summary

- File types
  - Unstructured and structured files
- Abstraction layers
  - User space
  - System call layer
  - Virtual file system layer
  - Local file system layer
  - Disk driver
- File structures
- Directories
  - Naming and attributes
  - links

# Directory examples

- Flat (CP/M)
  - All files are in one directory
- Hierarchical (Unix)
  - /home/inf3160/prosjekter
  - Directory is stored in a file containing (name,i-node) pairs
  - The name can be either a file or another directory
- Hierarchical (MS-DOS, NT)
  - C:\windows\temp\foo
  - Use the extension to indicate whether the entry is a directory

# Example file system implementations

- CP/M
- Windows
- Linux VFS
- Specific Linux file systems
  - Ext2
  - JFS

## File systems on disk

- Partitioning
- Logical volume management

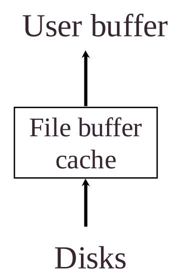
## superblock?

Memory mapped files?

Buffer cache?

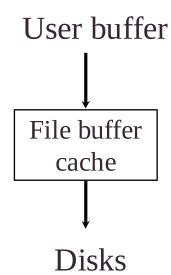
# File Caching

- Cache files in main memory
- read( fd, buf, n)
  - On a hit
    - copy from the buffer cache to a user buffer
  - On a miss
    - replacement if necessary
    - read a file into the buffer cache




# Maintain Consistency

- write( fd, buffer, n )
  - On a hit
    - write to buffer cache
  - On a miss
    - read the file to buffer cache if the file exists (possible replacement)
    - write to buffer cache
- When do you write the buffer cache to disk?
- In what order?

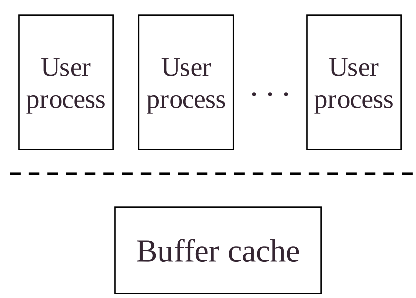


## What to Cache and Replace?

- Things to consider
  - I-nodes and indirect blocks of directories
  - Directory files
  - I-nodes and indirect blocks of files
  - Files
- A reasonable strategy?
  - Cache i-nodes and indirect blocks if they are in use
  - Cache only the i-nodes and indirect blocks of the current directory

## Where Is the Buffer Cache?

- Kernel
  - All processes share the same buffer cache
  - Global LRU
  - Each process use a different replacement strategy
- Can we move the buffer cache to the user level?
  - Duplications



## Relationship with Virtual Memory

- Memory mapped file
  - Use the file as the backing store for mapped pages
- Should we do this for all files?
  - Difficult to tell the size of the file
  - VM typically don’t care about writing back frequently
  - Huge files require huge VM space