CS 333
Introduction to Operating Systems

Class 17 - File Systems

Jonathan Walpole
Computer Science
Portland State University

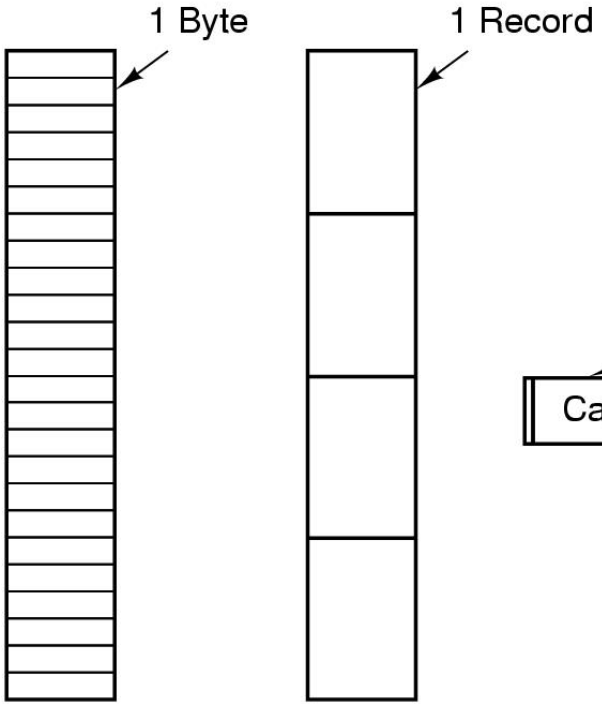
Why do we need a file system?

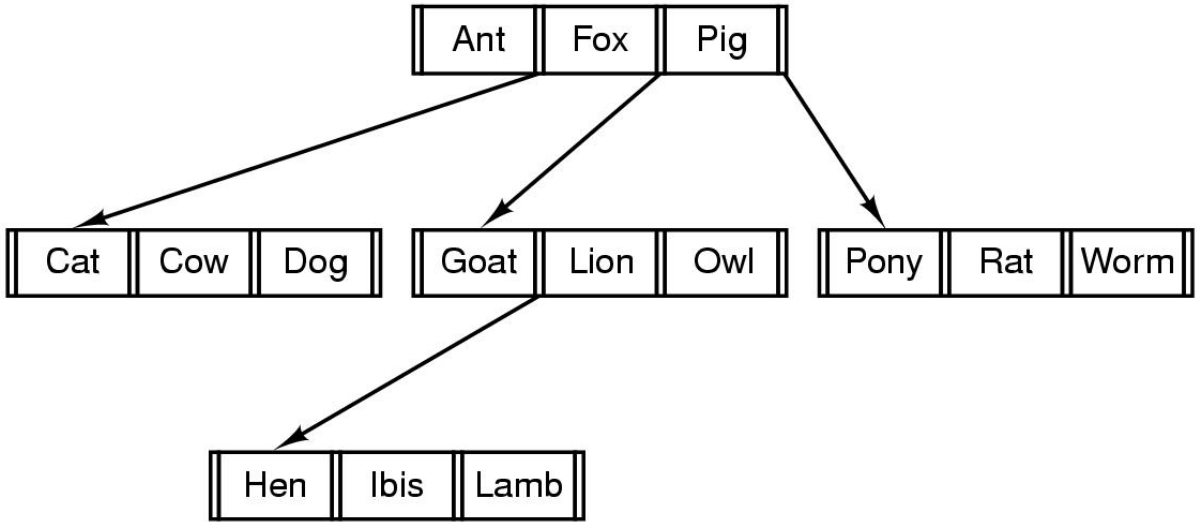
- **Must store large amounts of data**
- **Data must survive the termination of the process that created it**
  - Called "*persistence*"
- **Multiple processes must be able to access the information concurrently**

What is a file?

- **Files can be structured or unstructured**
  - Unstructured: just a sequence of bytes
  - Structured: a sequence or tree of typed records
- **In Unix-based operating systems a file is an unstructured sequence of bytes**

File Structure





Sequence of bytes

Sequence of records

Tree of records

File extensions

- **Even though files are just a sequence of bytes, programs can impose structure on them, by convention**
  - Files with a certain standard structure imposed can be identified using an extension to their name
  - Application programs may look for specific file extensions to indicate the file's type
  - But as far as the operating system is concerned its just a sequence of bytes

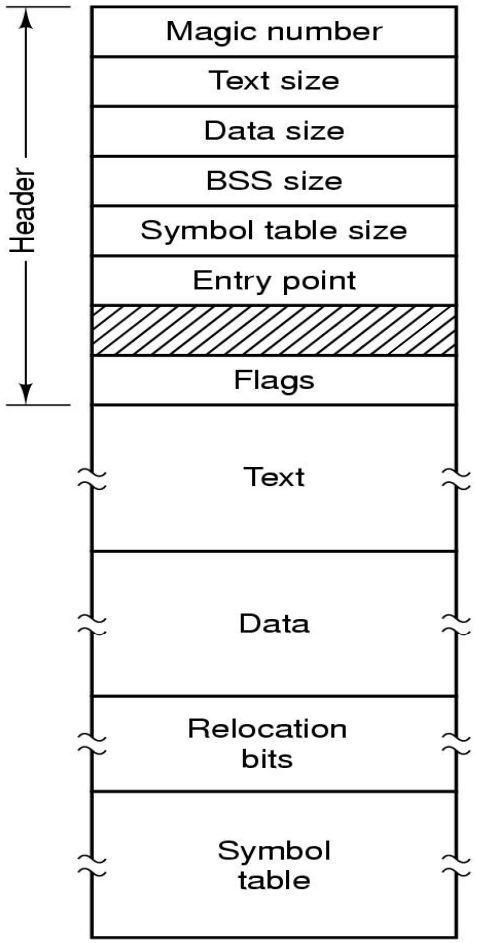
Typical file extensions

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Which file types does the OS understand?

- **Executable files**
  - The OS must understand the format of executable files in order to execute programs
    - **Create process (fork)**
    - **Put program and data in process address space (exec)**

Executable file formats

- 

An executable file

File attributes

- **Various meta-data needs to be associated with files**
  - Owner
  - Creation time
  - Access permissions / protection
  - Size etc
- **This meta-data is called the file attributes**
  - Maintained in file system data structures for each file

Example file attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File access

- **Sequential Access**
  - read all bytes/records from the beginning
  - cannot jump around (but could rewind or back up) convenient when medium was magnetic tape
- **Random Access**
  - can read bytes (or records) in any order
  - essential for database systems
  - option 1:
    - **move position, then read**
  - option 2:
    - **perform read, then update current position**

Some file-related system calls

- Create a file
- Delete a file
- Open
- Close
- Read (n bytes from current position)
- Write (n bytes to current position)
- Append (n bytes to end of file)
- Seek (move to new position)
- Get attributes
- Set/modify attributes
- Rename file

File-related system calls

- fd = open (name, mode)
- byte_count = read (fd, buffer, buffer_size)
- byte_count = write (fd, buffer, num_bytes)
- close (fd)

A "C" Program to Copy a File

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
      int in_fd, out_fd, rd_count, wt_count;
      char buffer[BUF_SIZE];

      if (argc != 3) exit(1);           /* syntax error if argc is not 3 */
```

(continued)

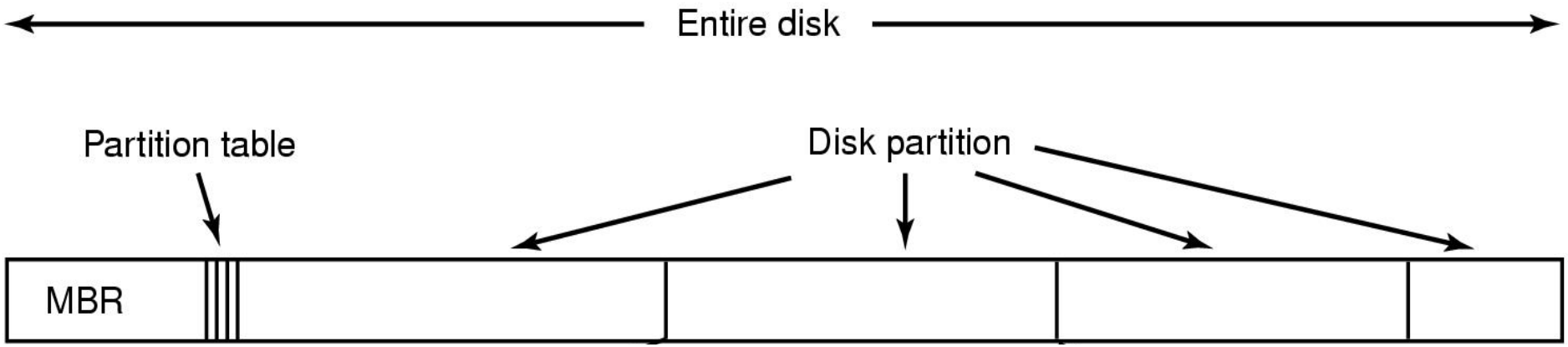
A "C" Program to Copy a File

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);    /* open the source file */
if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);         /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0)                  /* no error on last read */
        exit(0);
else
        exit(5);                    /* error on last read */
}
```

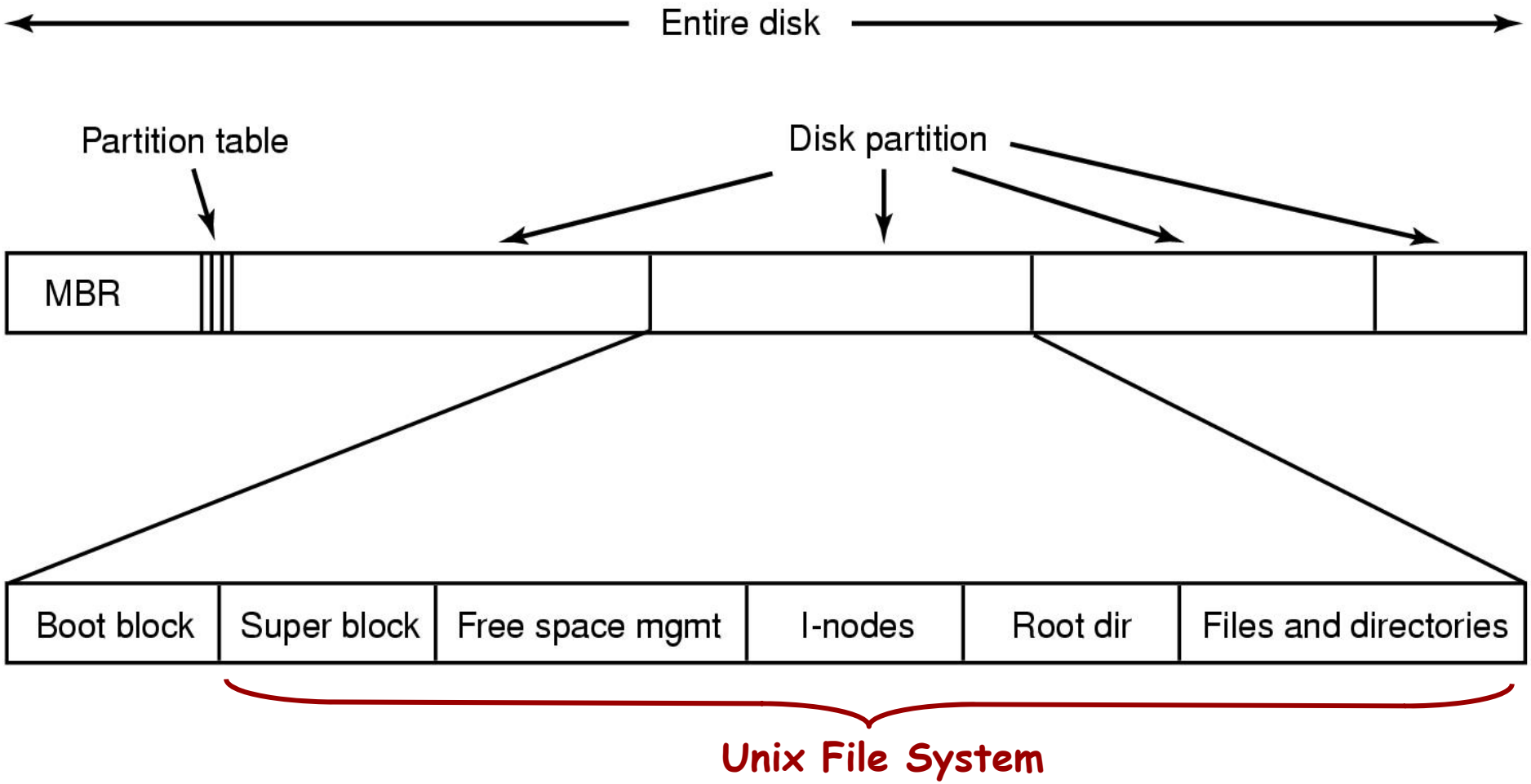
File storage on disk

- **Sector 0: "Master Boot Record" (MBR)**
  - Contains the partition map
- **Rest of disk divided into "partitions"**
  - Partition: sequence of consecutive sectors.
- **Each partition can hold its own file system**
  - Unix file system
  - Window file system
  - Apple file system
- **Every partition starts with a "boot block"**
  - Contains a small program
  - This "boot program" reads in an OS from the file system in that partition
- **OS Startup**
  - Bios reads MBR , then reads & execs a boot block

An example disk



An example disk

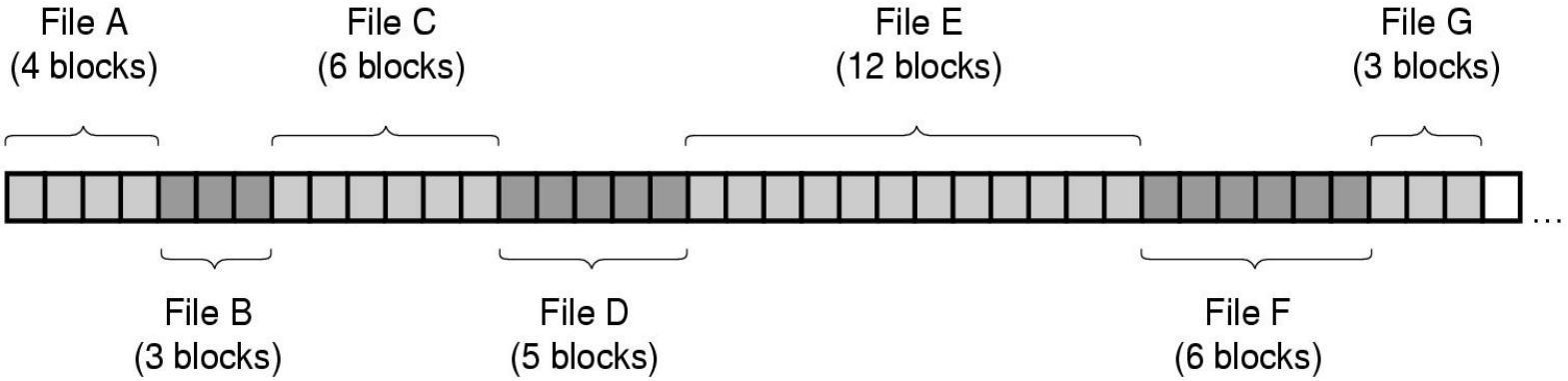


File bytes vs disk sectors

- **Files are sequences of bytes**
  - Granularity of file I/O is bytes
- **Disks are arrays of sectors (512 bytes)**
  - Granularity of disk I/O is sectors
  - Files data must be stored in sectors
- **File systems define a block size**
  - block size = 2^n * sector size
  - Contiguous sectors are allocated to a block
- **File systems view the disk as an array of blocks**
  - Must allocate blocks to file
  - Must manage free space on disk

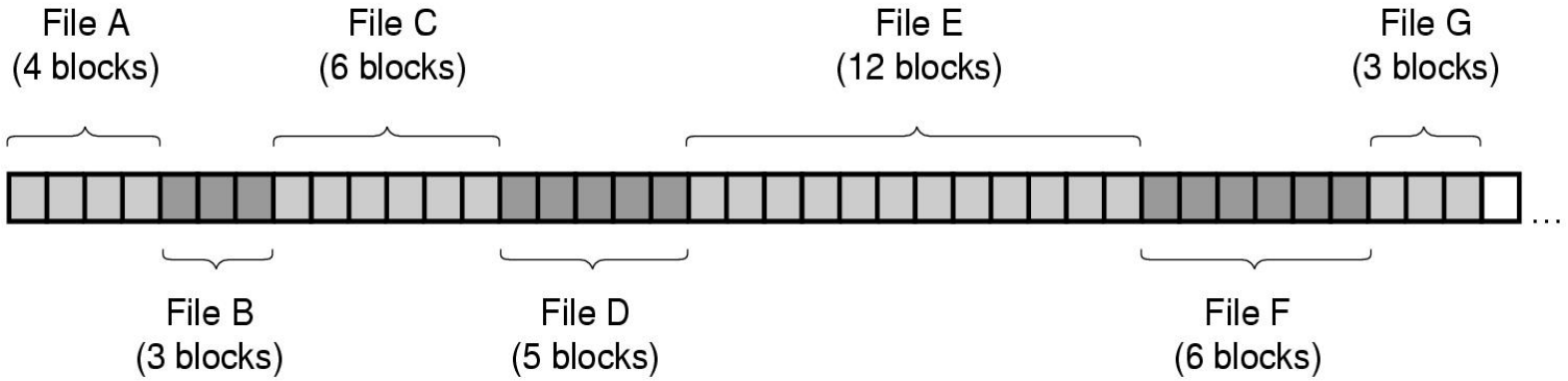
Contiguous allocation

- **Idea:**
  - All blocks in a file are contiguous on the disk

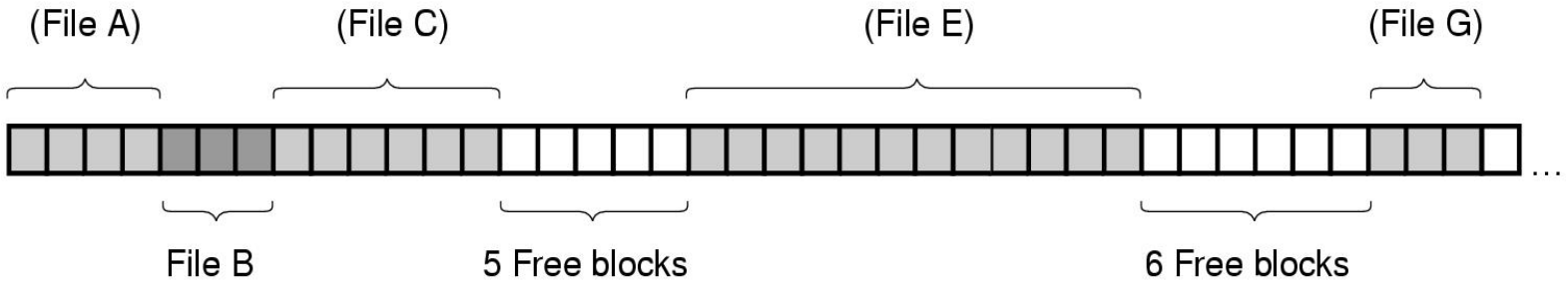


Contiguous allocation

- **Idea:**
  - All blocks in a file are contiguous on the disk.



After deleting D and F...



Contiguous allocation

- ***Advantages:***
  - Simple to implement (Need only starting sector & length of file)
  - Performance is good (for sequential reading)

Contiguous allocation

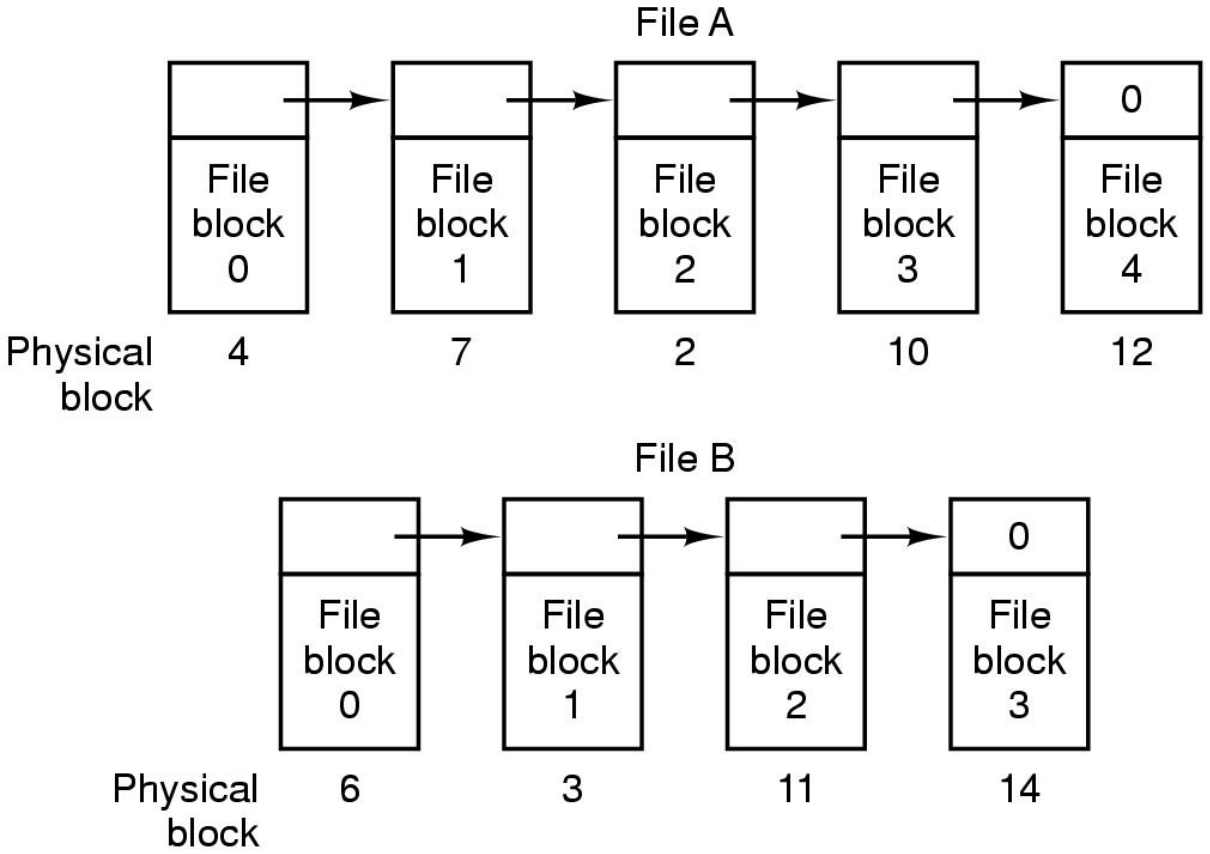
- ***Advantages:***
  - Simple to implement (Need only starting sector & length of file)
  - Performance is good (for sequential reading)
- ***Disadvantages:***
  - After deletions, disk becomes fragmented
  - Will need periodic compaction (time-consuming)
  - Will need to manage free lists
  - If new file put at end of disk...
    - **No problem**
  - If new file is put into a "hole"...
    - **Must know a file's maximum possible size** ... ***at the time it is created!***

Contiguous allocation

- **Good for CD-ROMs**
  - All file sizes are known in advance
  - Files are never deleted

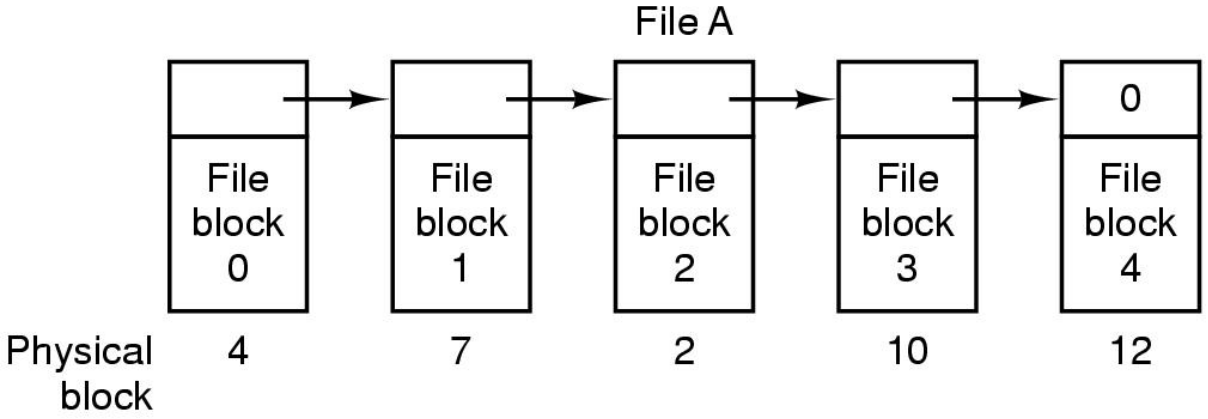
Linked list allocation

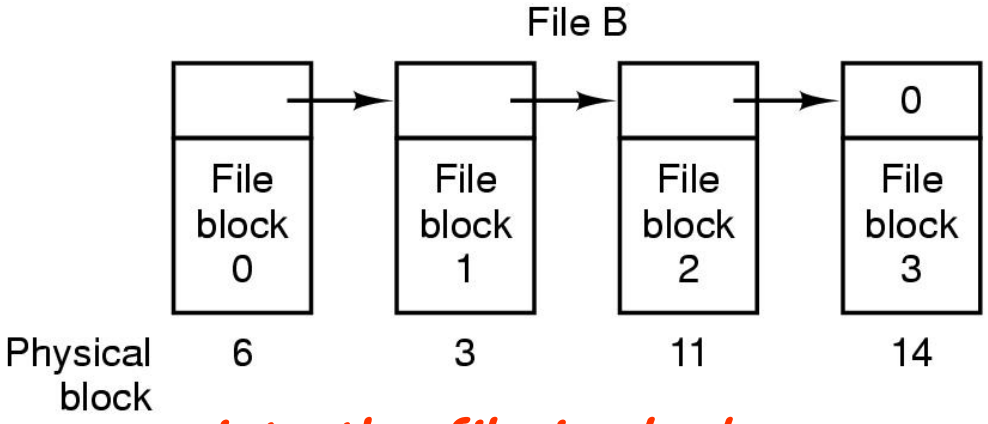
- Each file is a sequence of blocks
- First word in each block contains number of next block



Linked list allocation

- Each file is a sequence of blocks
- First word in each block contains number of next block



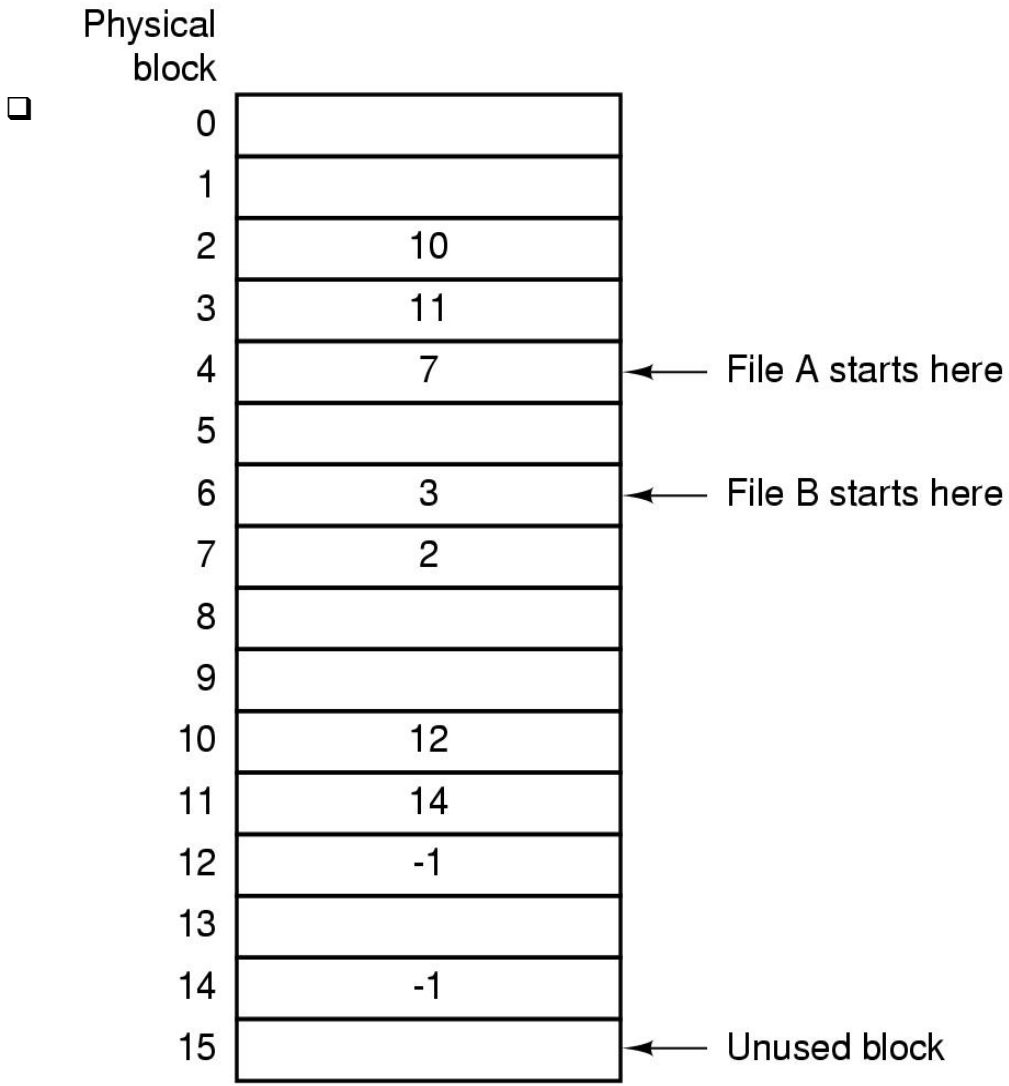


Random access into the file is slow!

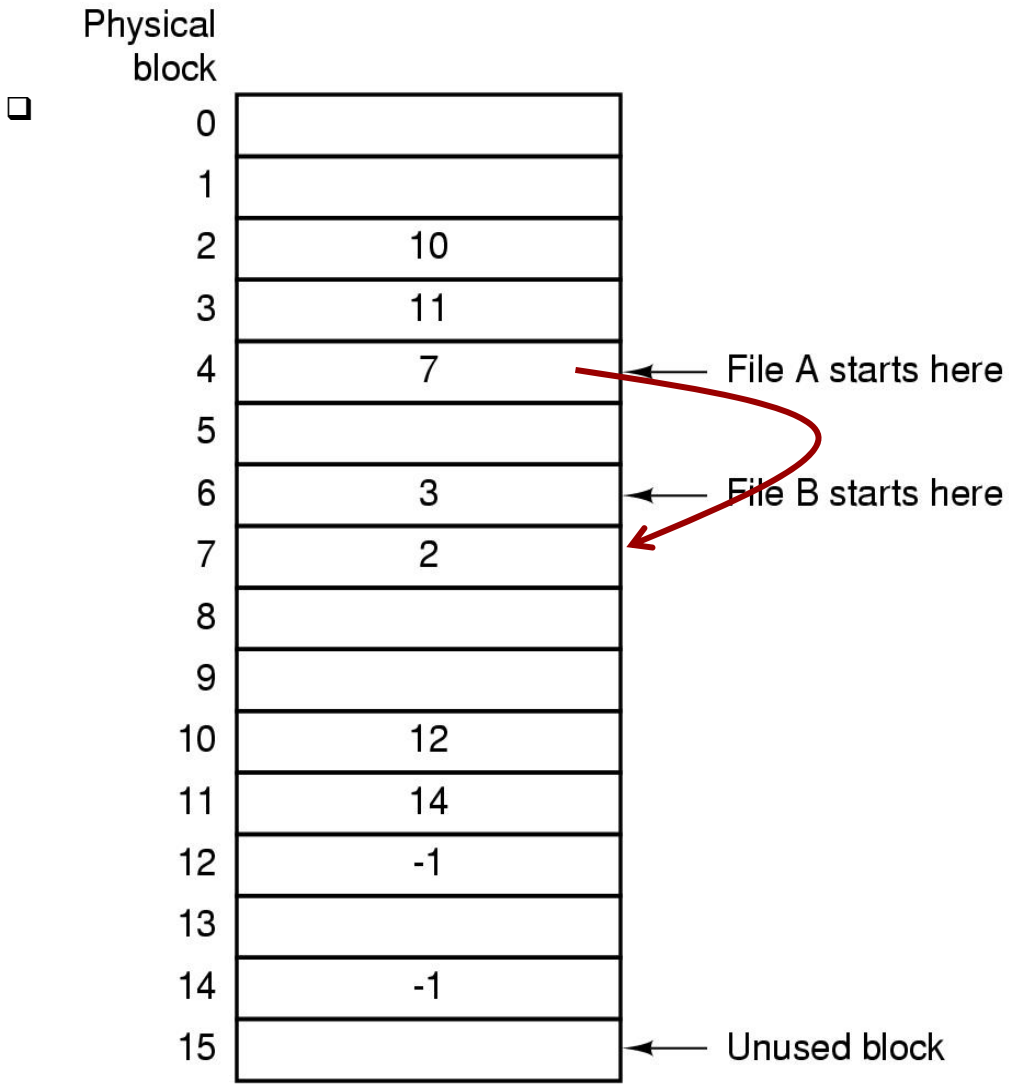
File allocation table (FAT)

- **Keep a table in memory**
- **One entry per block on the disk**
- **Each entry contains the address of the "next" block**
  - End of file marker (-1)
- **A special value (-2) indicates the block is free**

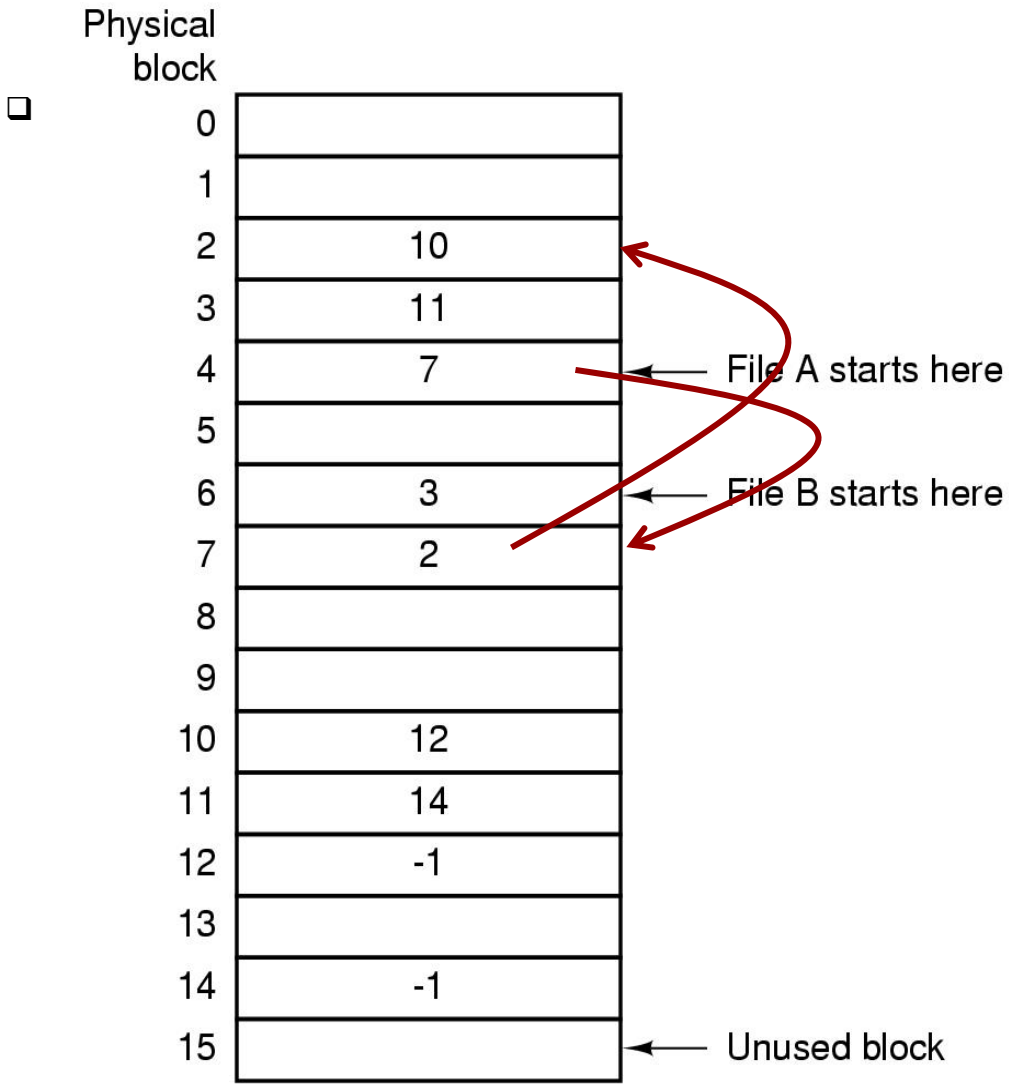
File allocation table (FAT)



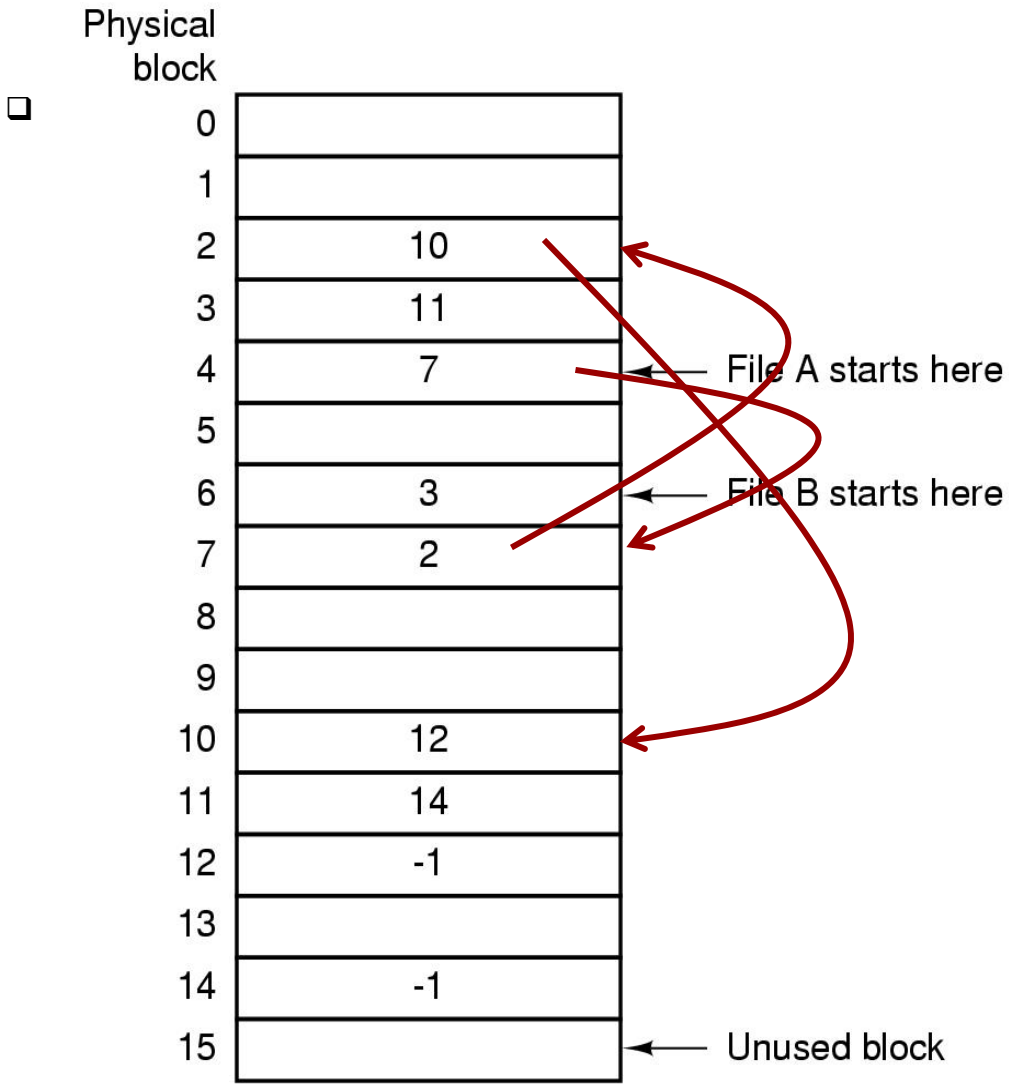
File allocation table (FAT)



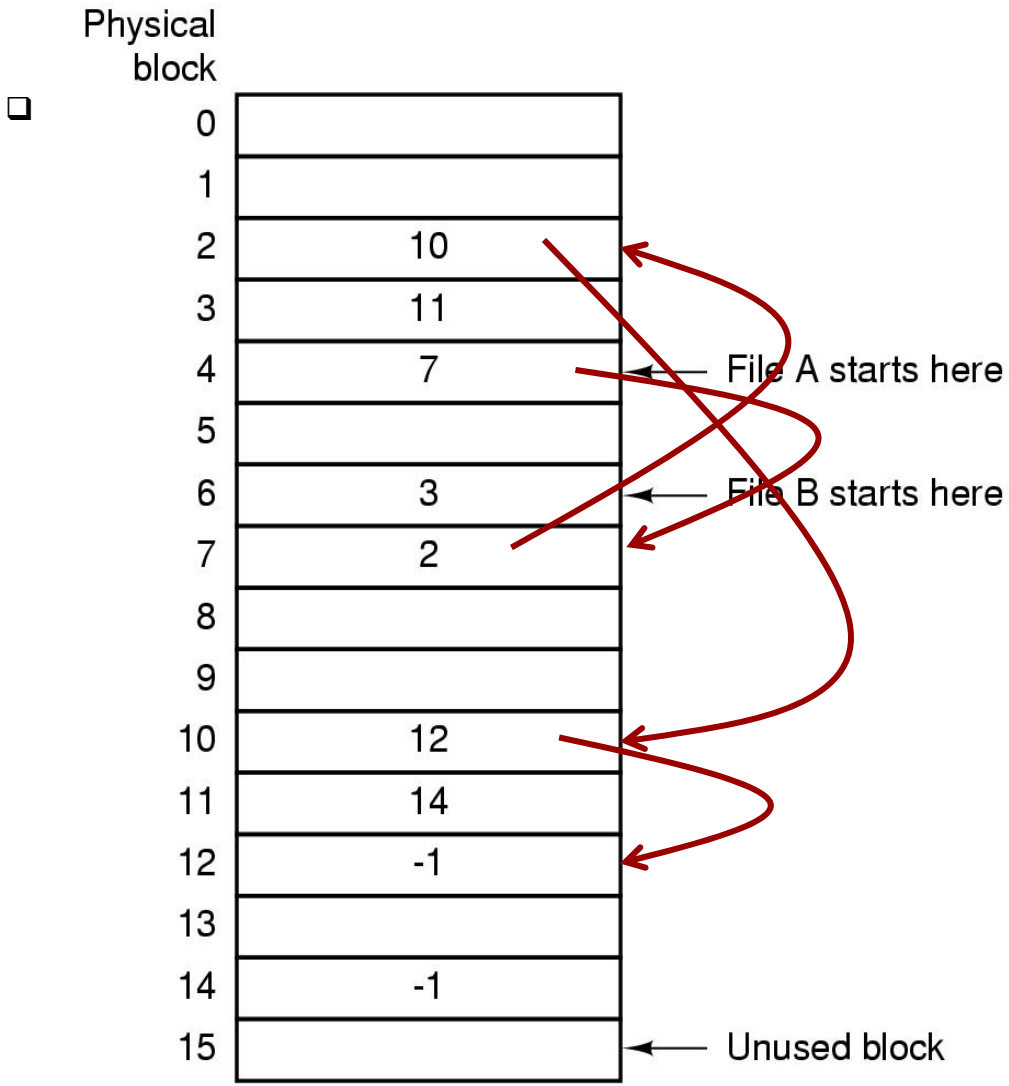
File allocation table (FAT)



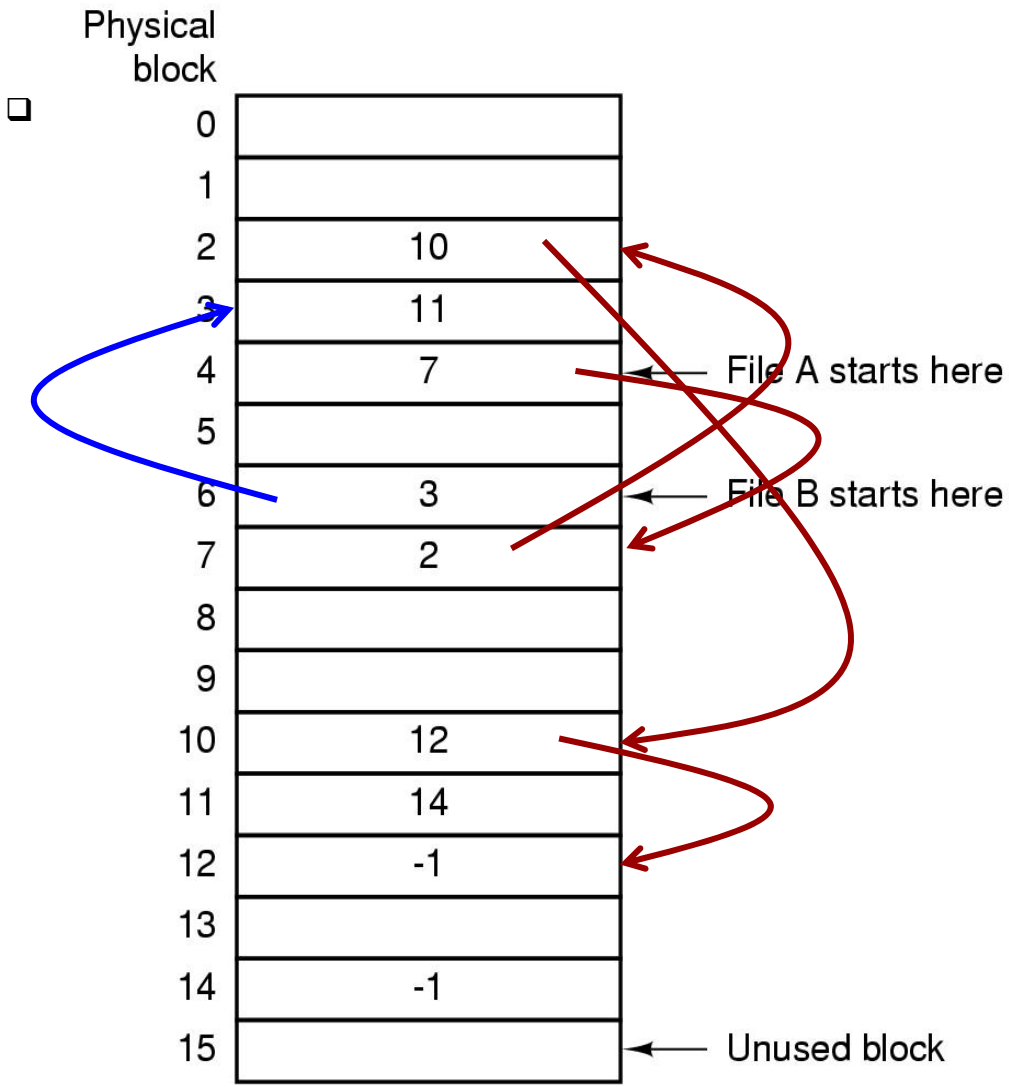
File allocation table (FAT)



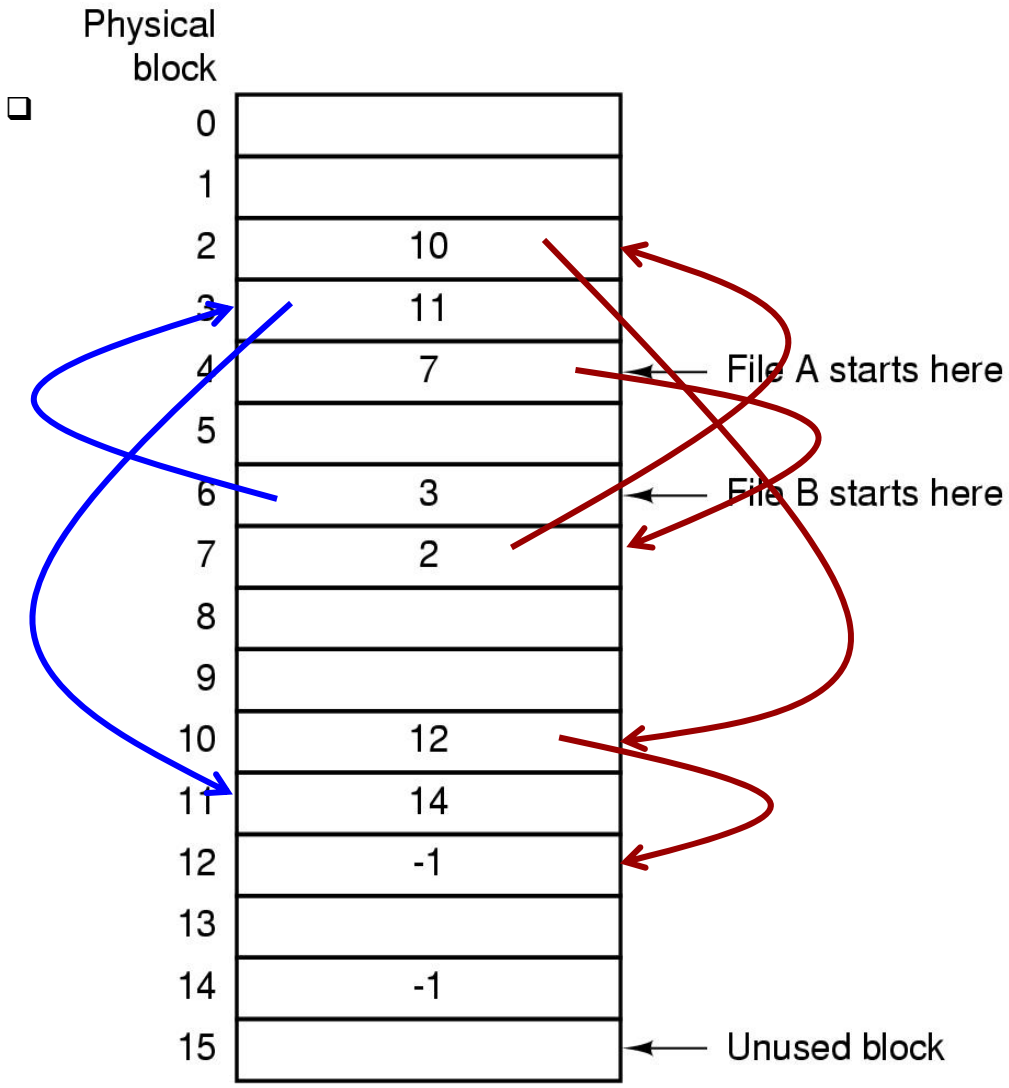
File allocation table (FAT)



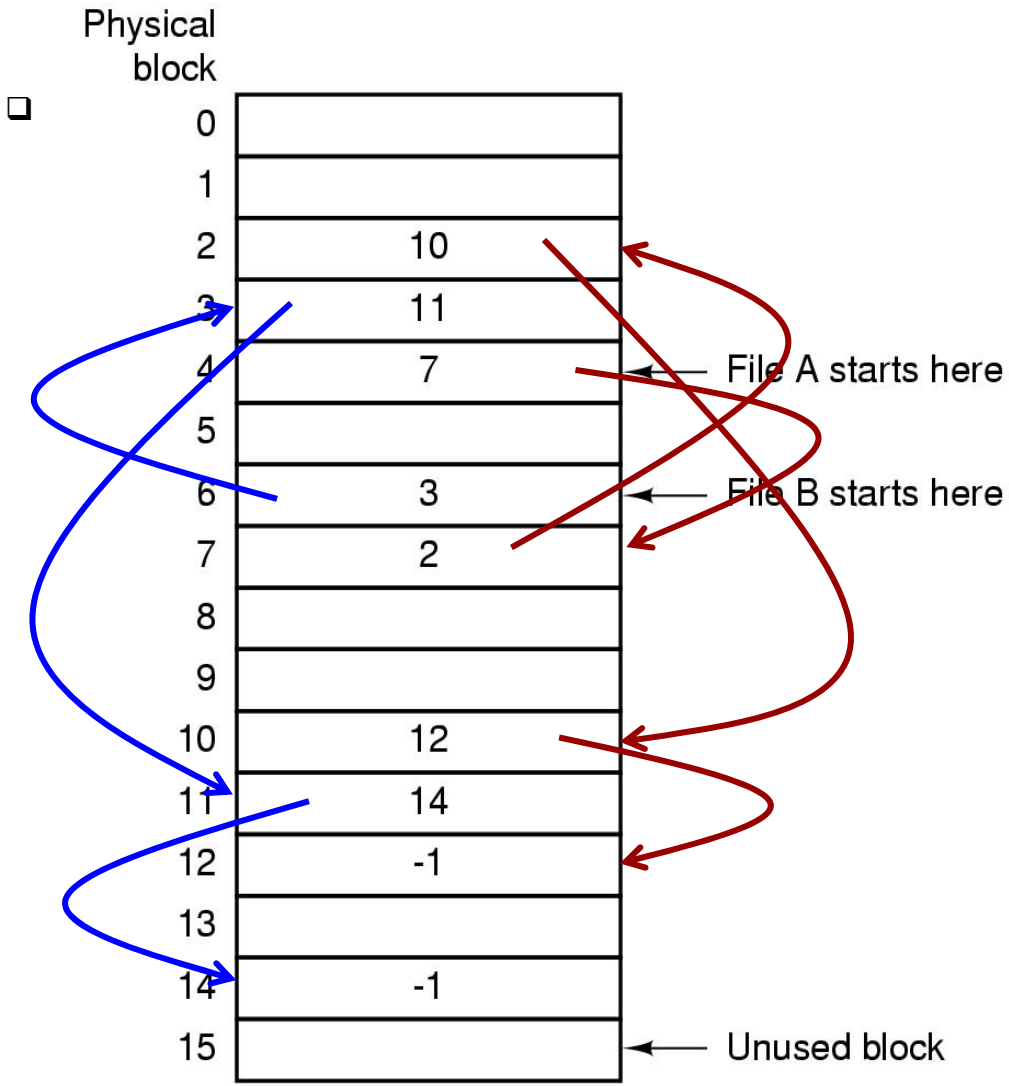
File allocation table (FAT)



File allocation table (FAT)



File allocation table (FAT)



File allocation table (FAT)

- **Random access...**
  - Search the linked list (but all in memory)
- **Directory entry needs only one number**
  - Starting block number

File allocation table (FAT)

- **Random access...**
  - Search the linked list (but all in memory)
- **Directory Entry needs only one number**
  - Starting block number
- **Disadvantage:**
  - Entire table must be in memory all at once!
  - A problem for large file systems

File allocation table (FAT)

- **Random access...**
  - Search the linked list (but all in memory)
- **Directory Entry needs only one number**
  - Starting block number
- **Disadvantage:**
  - Entire table must be in memory all at once!
  - Example:

    20 GB = disk size

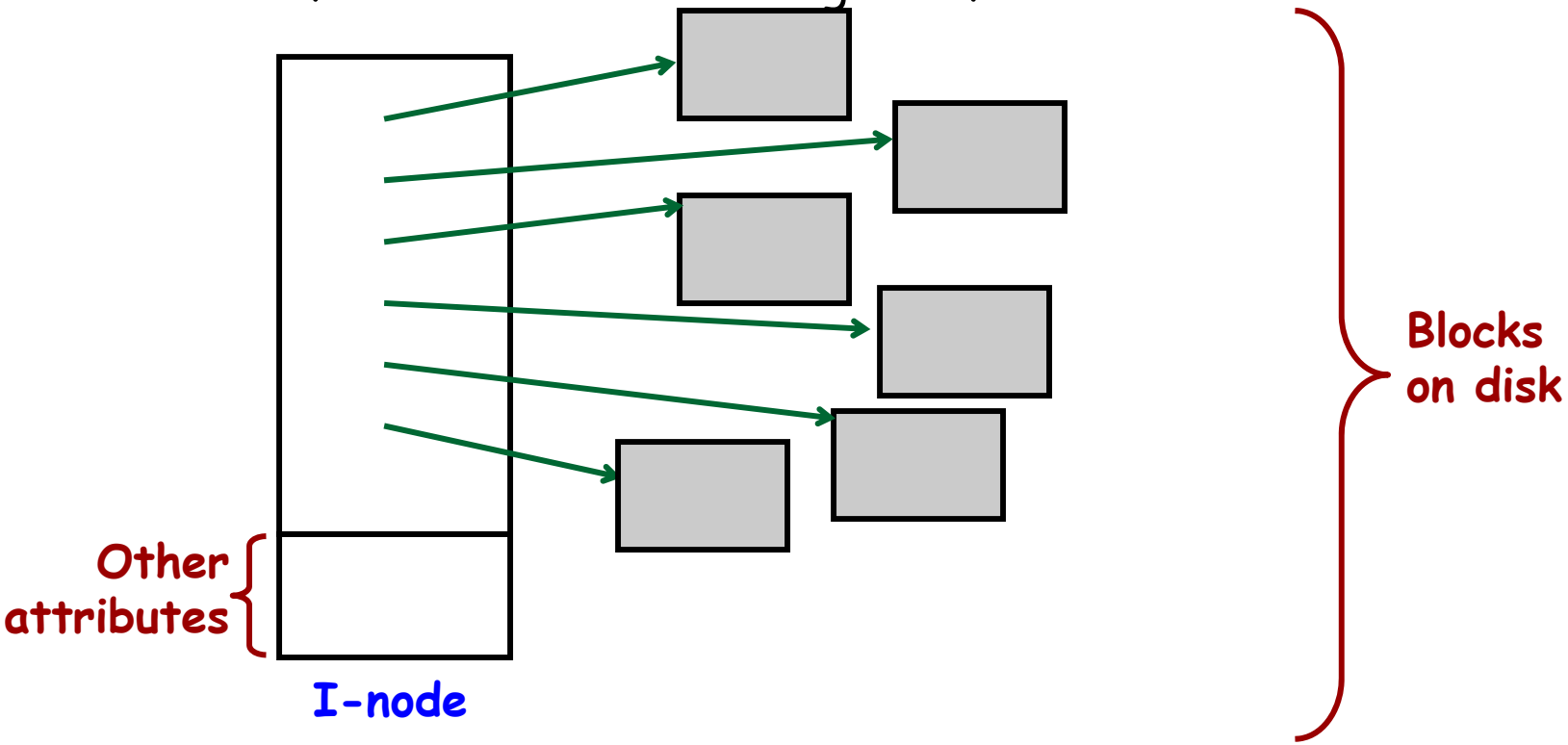
    1 KB = block size

    4 bytes = FAT entry size

    80 MB of memory used to store the FAT

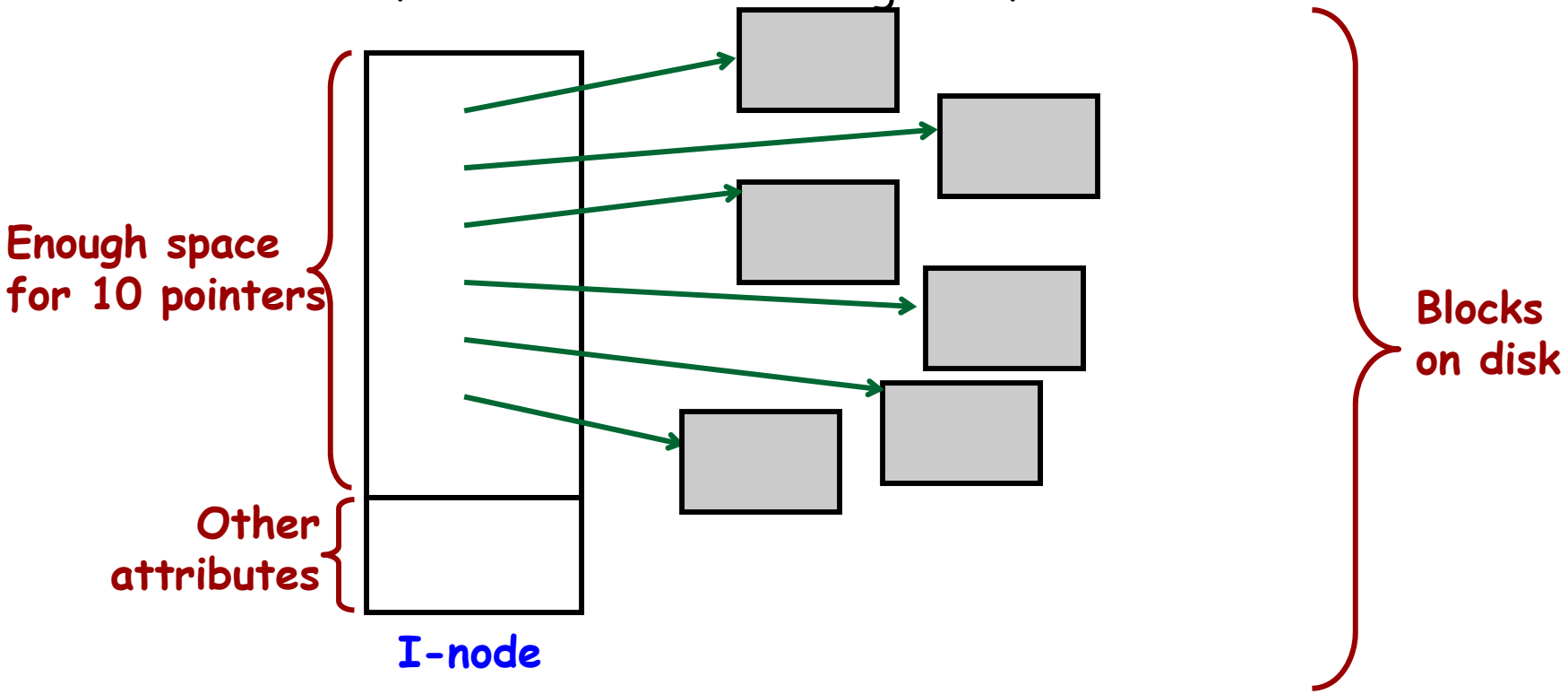
I-nodes

- Each I-node ("index-node") is a structure / record
- Contains info about the file
  - Attributes
  - Location of the blocks containing the file



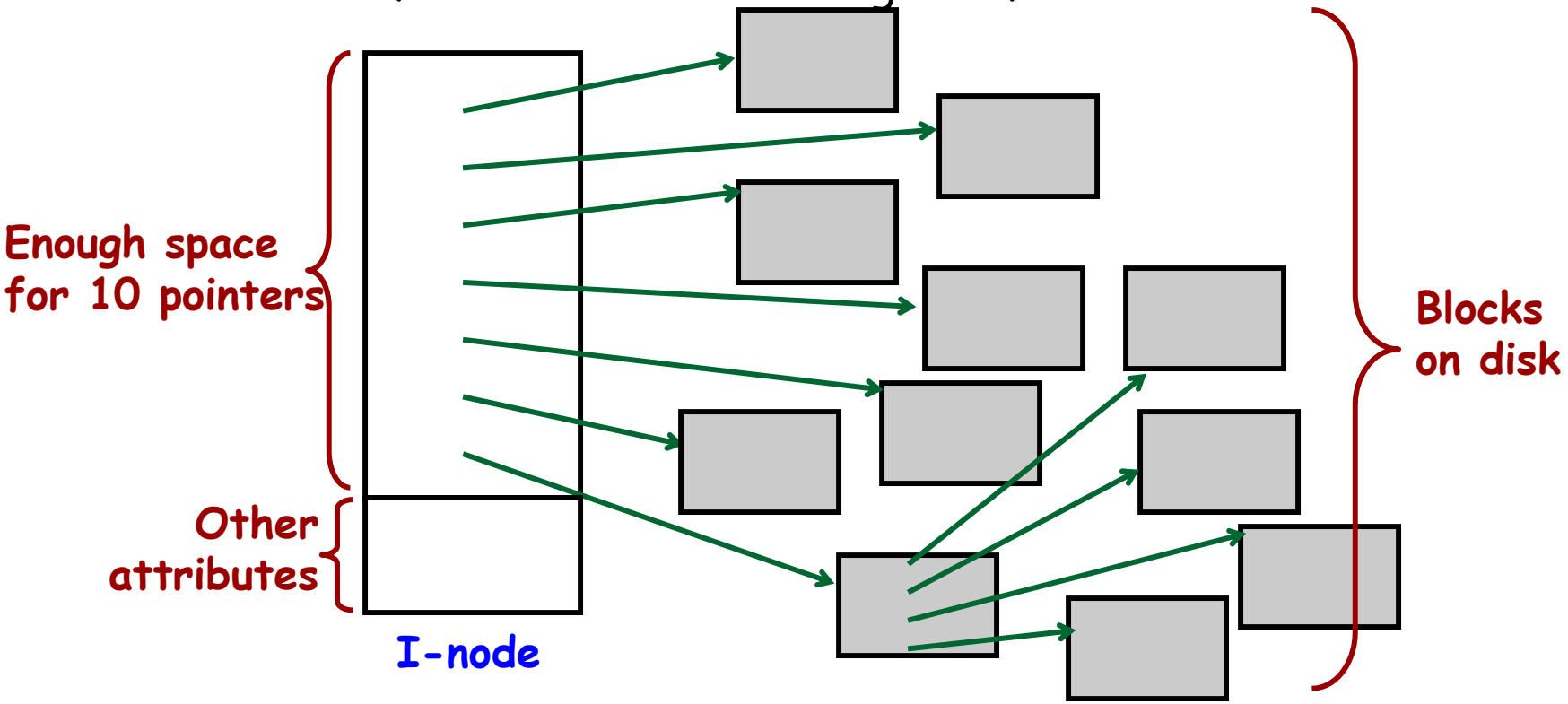
I-nodes

- Each I-Node ("index-node") is a structure / record
- Contains info about the file
  - Attributes
  - Location of the blocks containing the file



I-nodes

- Each I-Node ("index-node") is a structure / record
- Contains info about the file
  - Attributes
  - Location of the blocks containing the file

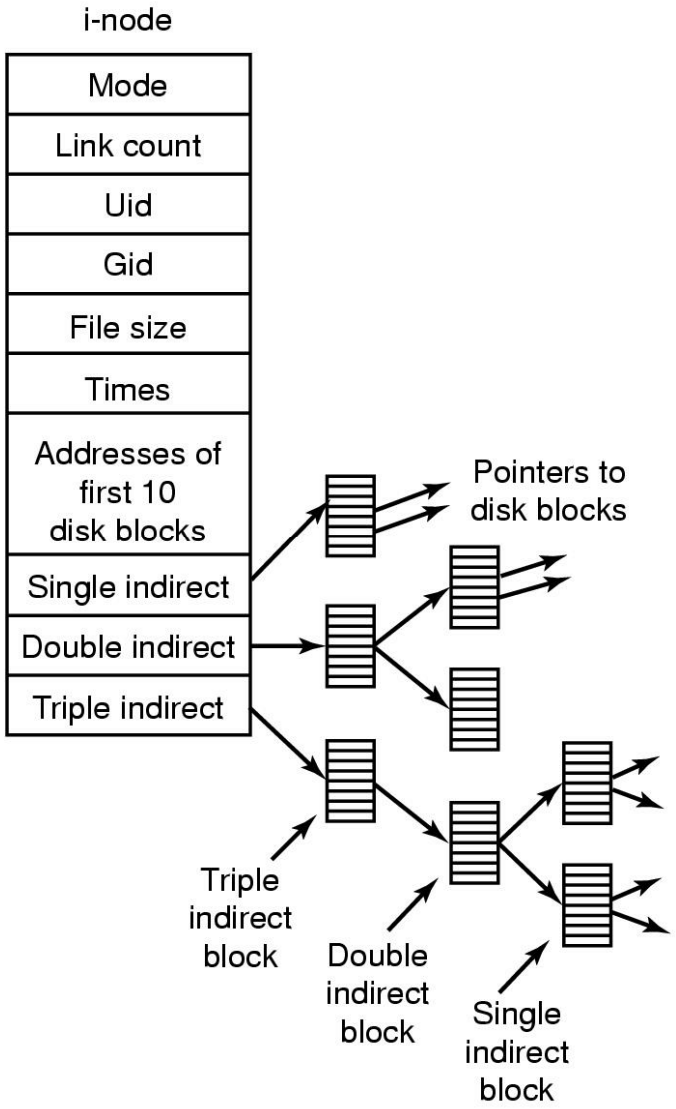


The UNIX I-node entries

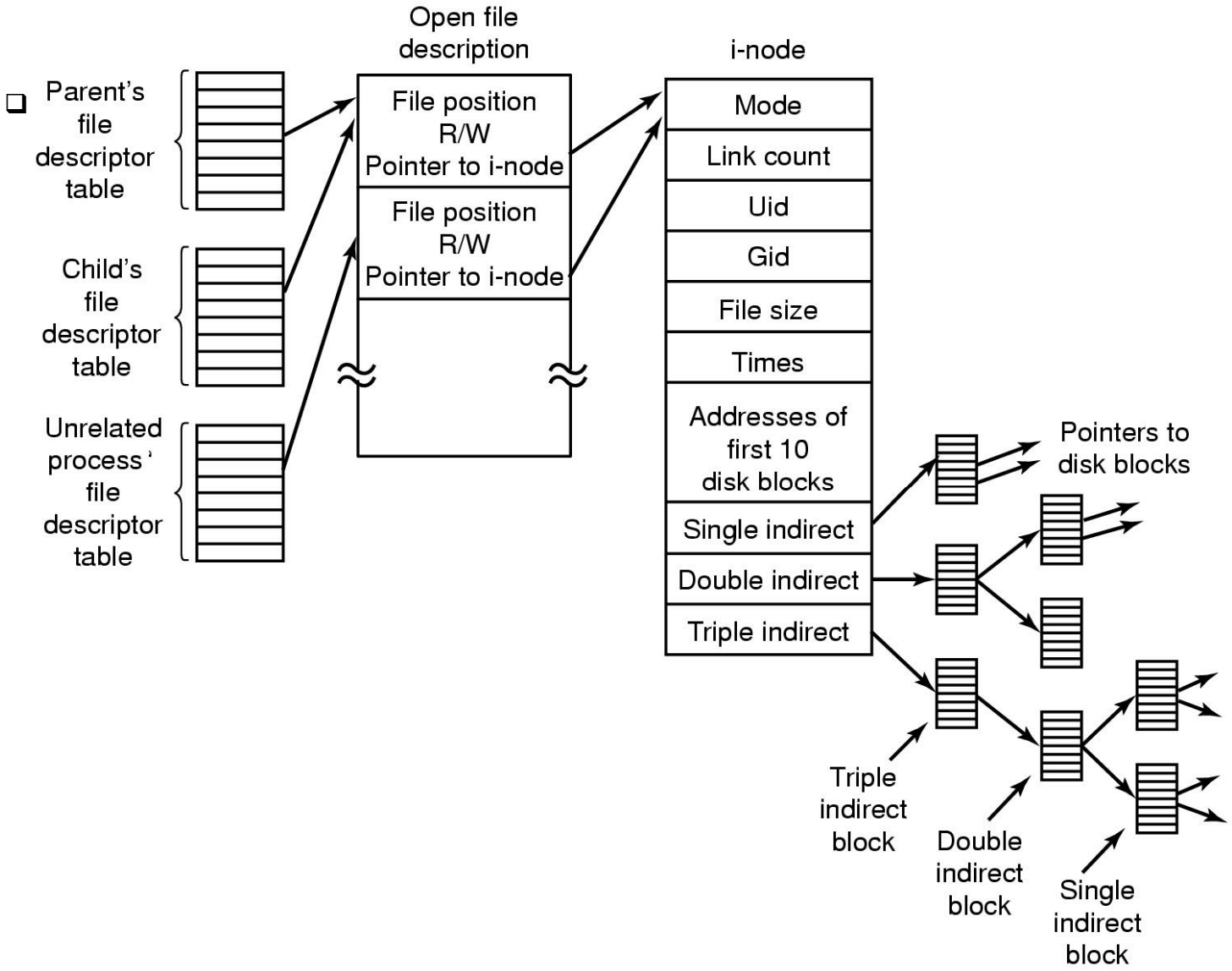
Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Structure of an I-Node

The UNIX I-node

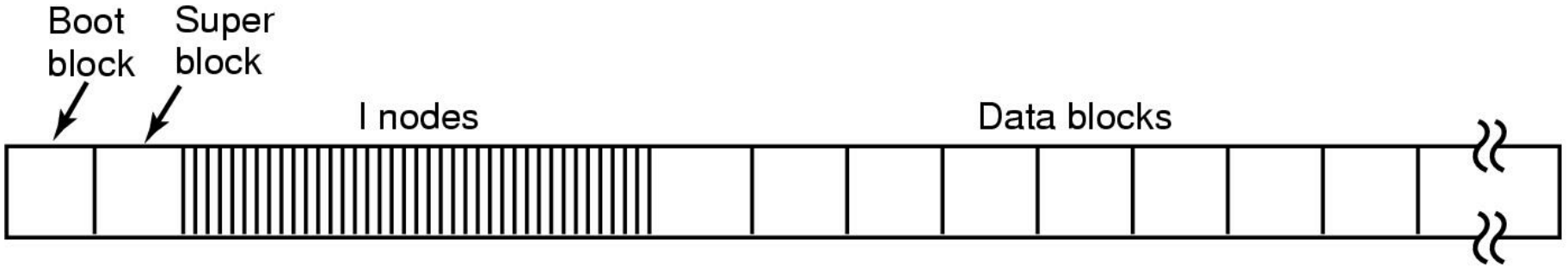


The UNIX File System



The UNIX file system

- *The layout of the disk (for early Unix systems):*

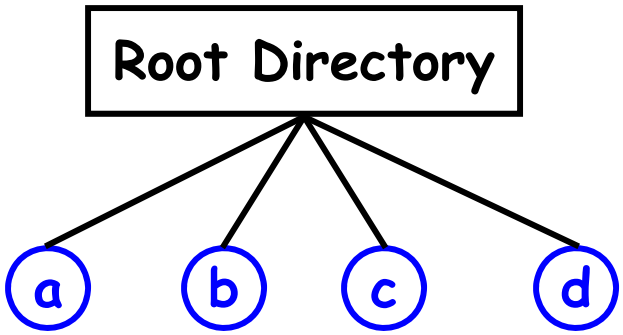


Naming files

- How do we find a file given its name?
- How can we ensure that file names are unique?

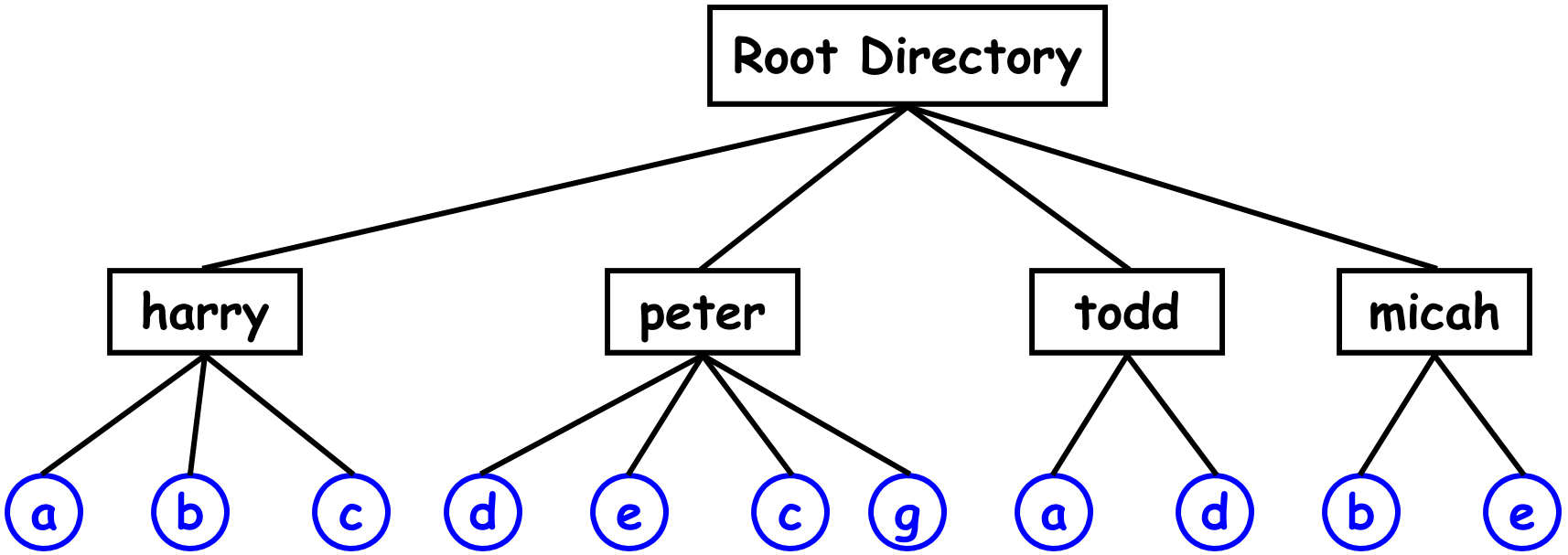
Single level directories

- "Folder"
- Single-Level Directory Systems
  - Early OSs
- Problem:
  - Sharing amongst users
- Appropriate for small, embedded systems



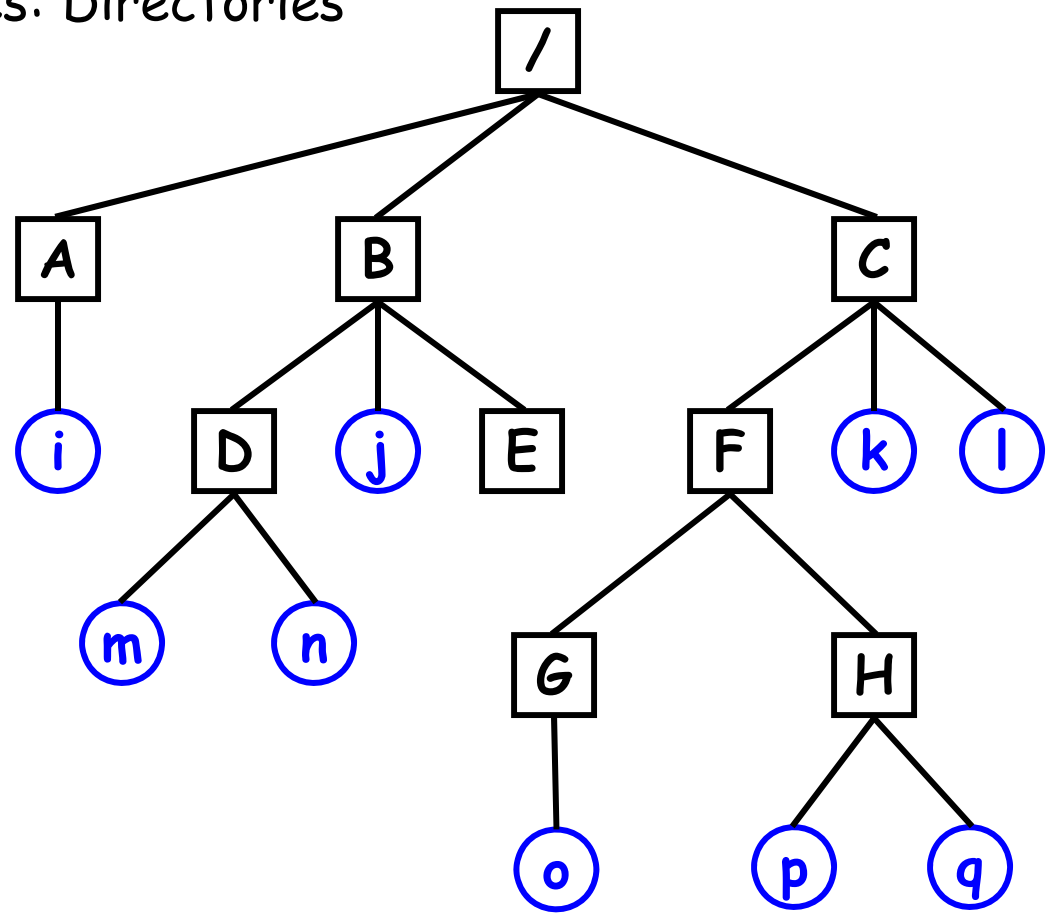
Two-level directory systems

- Letters indicate who owns the file / directory.
- Each user has a directory.
  - /peter/g



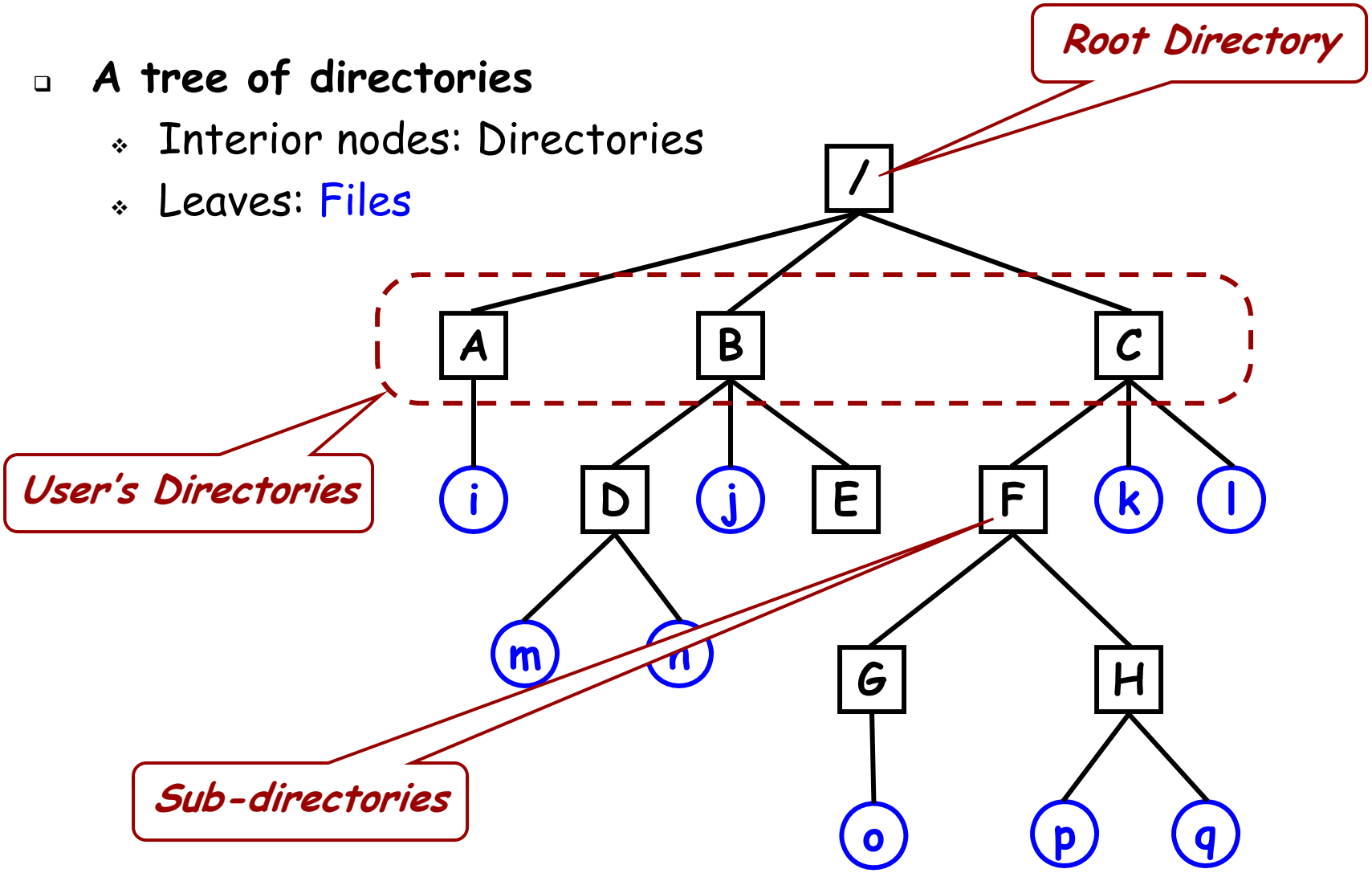
Hierarchical directory systems

- **A tree of directories**
  - Interior nodes: Directories
  - Leaves: Files



Hierarchical directory systems

- **A tree of directories**
  - Interior nodes: Directories
  - Leaves: Files



Path names

- **MULTICS**

  >usr>jon>mailbox
- **Windows**

  \usr\jon\mailbox
- **Unix**

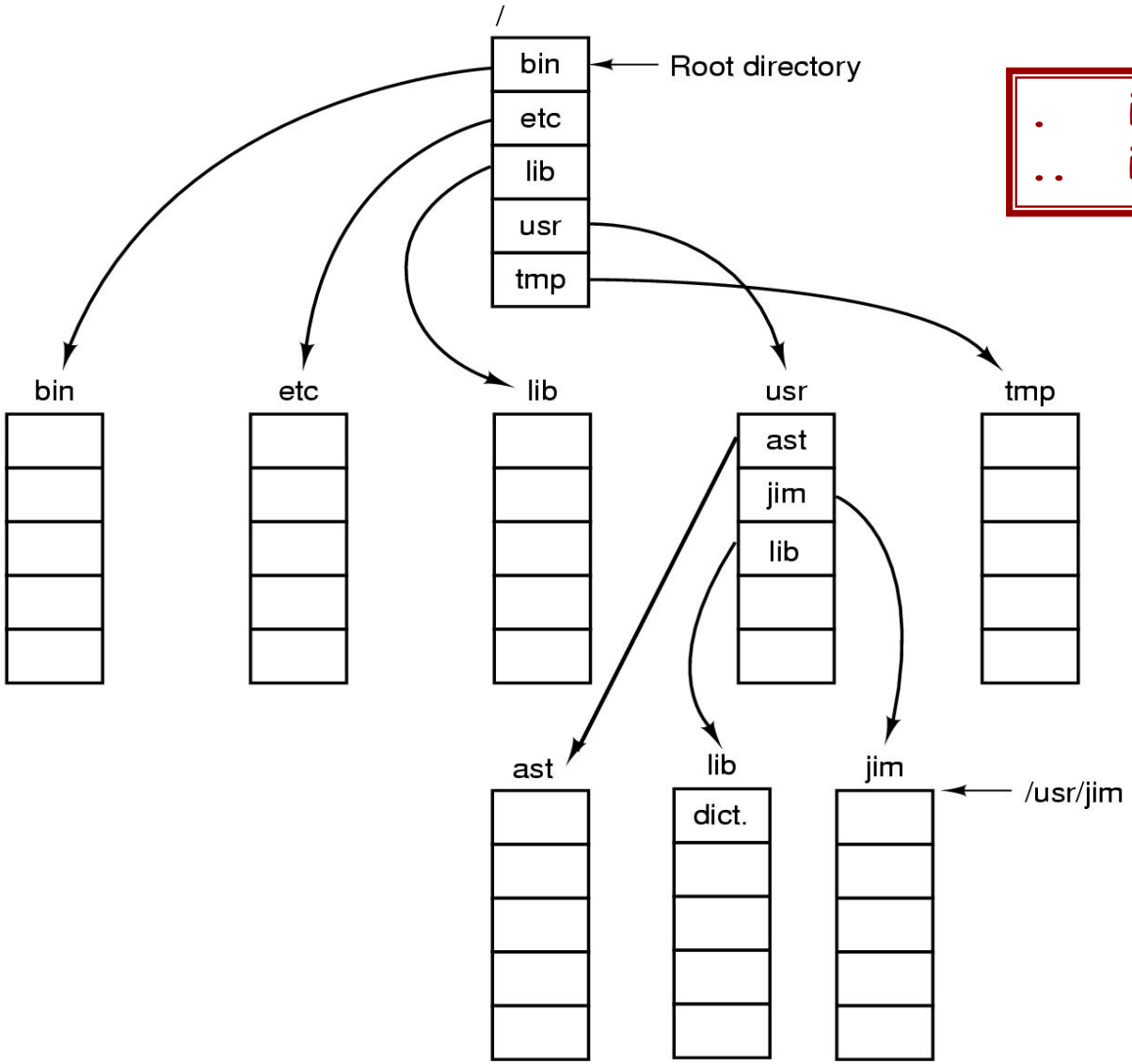
  /usr/jon/mailbox

Path names

- **MULTICS**
  >usr>jon>mailbox
- **Windows**
  \usr\jon\mailbox
- **Unix**
  /usr/jon/mailbox
- ***Absolute Path Name***
  /usr/jon/mailbox
- ***Relative Path Name***
  - "working directory" (or "current directory")
  - mailbox

Each process has its own working directory

A Unix directory tree



. is the "current directory"
.. is the parent

Typical directory operations

- **Create a new directory**
- **Delete a directory**
- **Open a directory for reading**
- **Close**
- **Readdir - return next entry in the directory**
  - Returns the entry in a standard format, regardless of the internal representation
- **Rename a directory**
- **Link**
  - Add this directory as a sub directory in another directory. (ie. Make a "hard link".)
- **Unlink**
  - Remove a "hard link"

Unix directory-related syscalls

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

- s = error code
- dir = directory stream
- dirent = directory entry

Implementing directories

- **List of files**
  - File name
  - File Attributes
- ***Simple Approach:***
  - Put all attributes in the directory

Implementing directories

- **List of files**
  - File name
  - File Attributes
- ***Simple Approach:***
  - Put all attributes in the directory
- ***Unix Approach:***
  - Directory contains
    - **File name**
    - **I-Node number**
  - I-Node contains
    - **File Attributes**

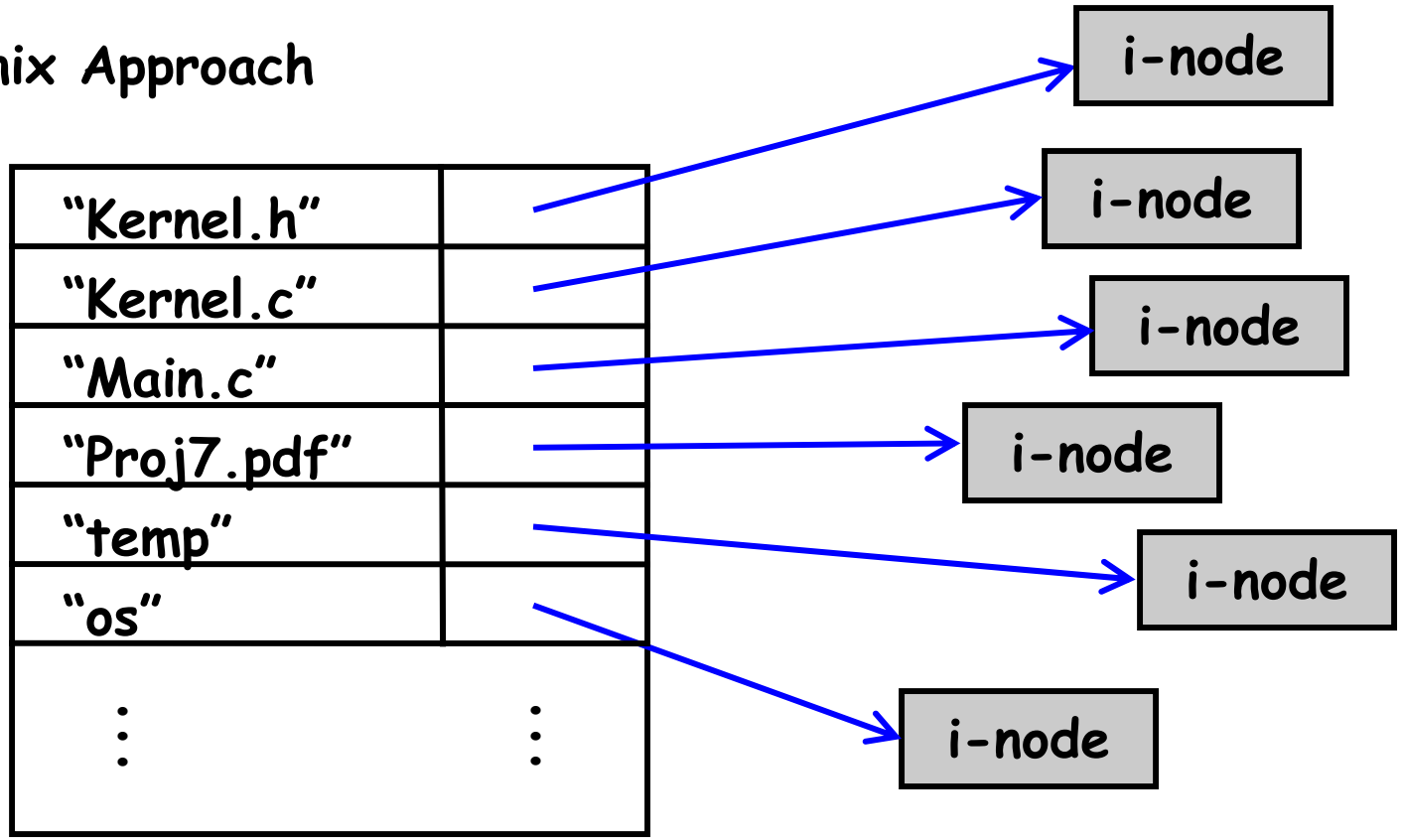
Implementing directories

- Simple Approach

"Kernel.h"	attributes
"Kernel.c"	attributes
"Main.c"	attributes
"Proj7.pdf"	attributes
"temp"	attributes
"os"	attributes
⋮	⋮

Implementing directories

- Unix Approach



Implementing filenames

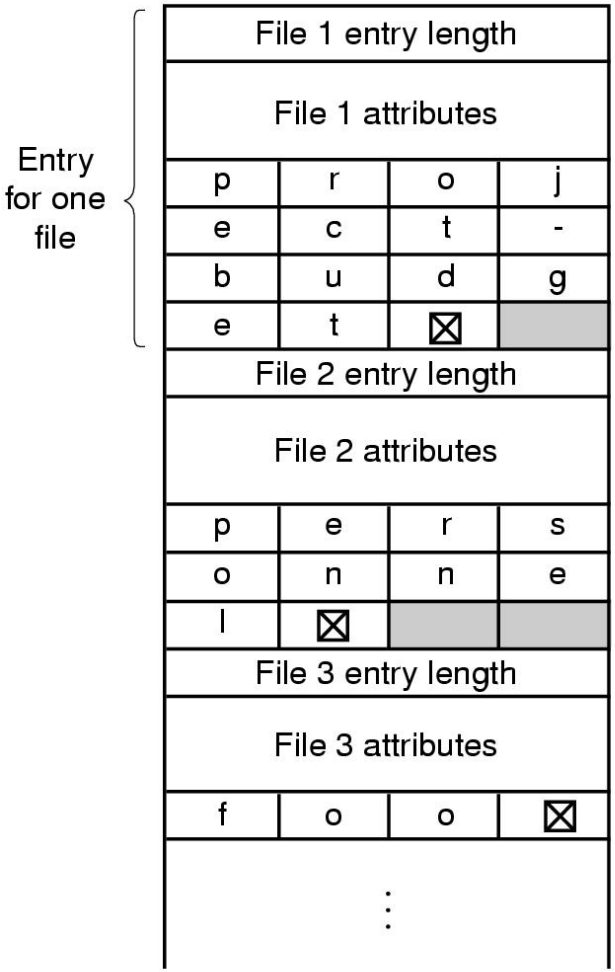
- **Short, Fixed Length Names**
  - MS-DOS/Windows
    - **8 + 3 "FILE3.BAK"**
    - **Each directory entry has 11 bytes for the name**
  - Unix (original)
    - **Max 14 chars**

Implementing filenames

- **Short, Fixed Length Names**
  - MS-DOS/Windows
    - **8 + 3 "FILE3.BAK"**
    - **Each directory entry has 11 bytes for the name**
  - Unix (original)
    - **Max 14 chars**
- **Variable Length Names**
  - Unix (today)
    - **Max 255 chars**
    - **Directory structure gets more complex**

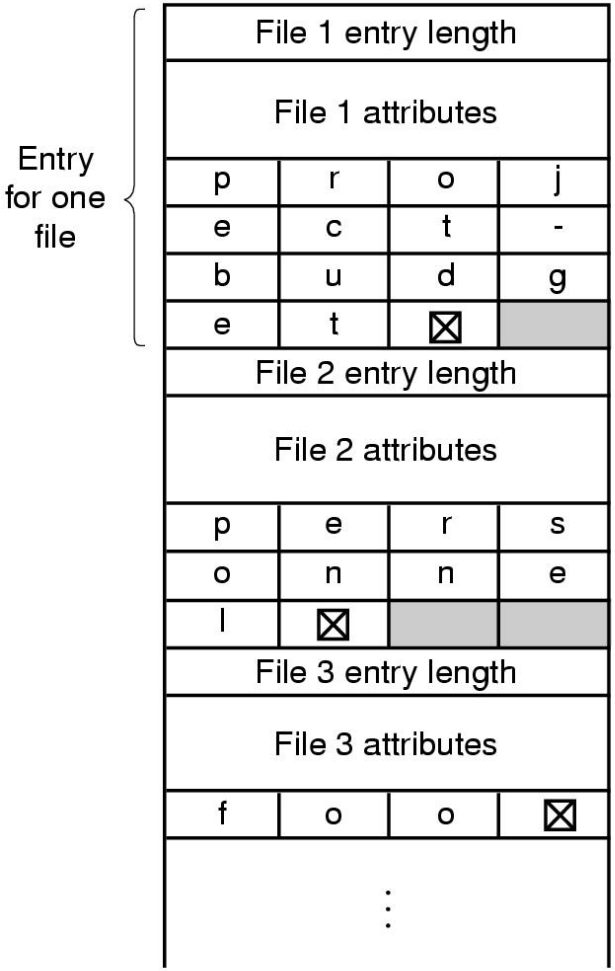
Variable-length filenames

- *Approach #1*

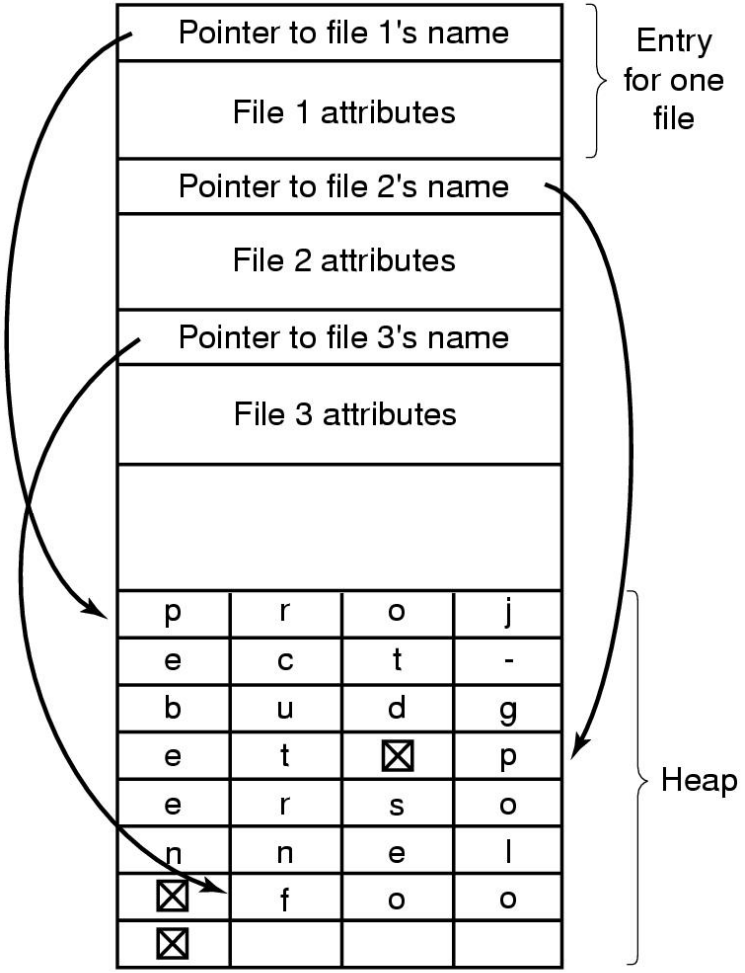


Variable-length filenames

- *Approach #1*

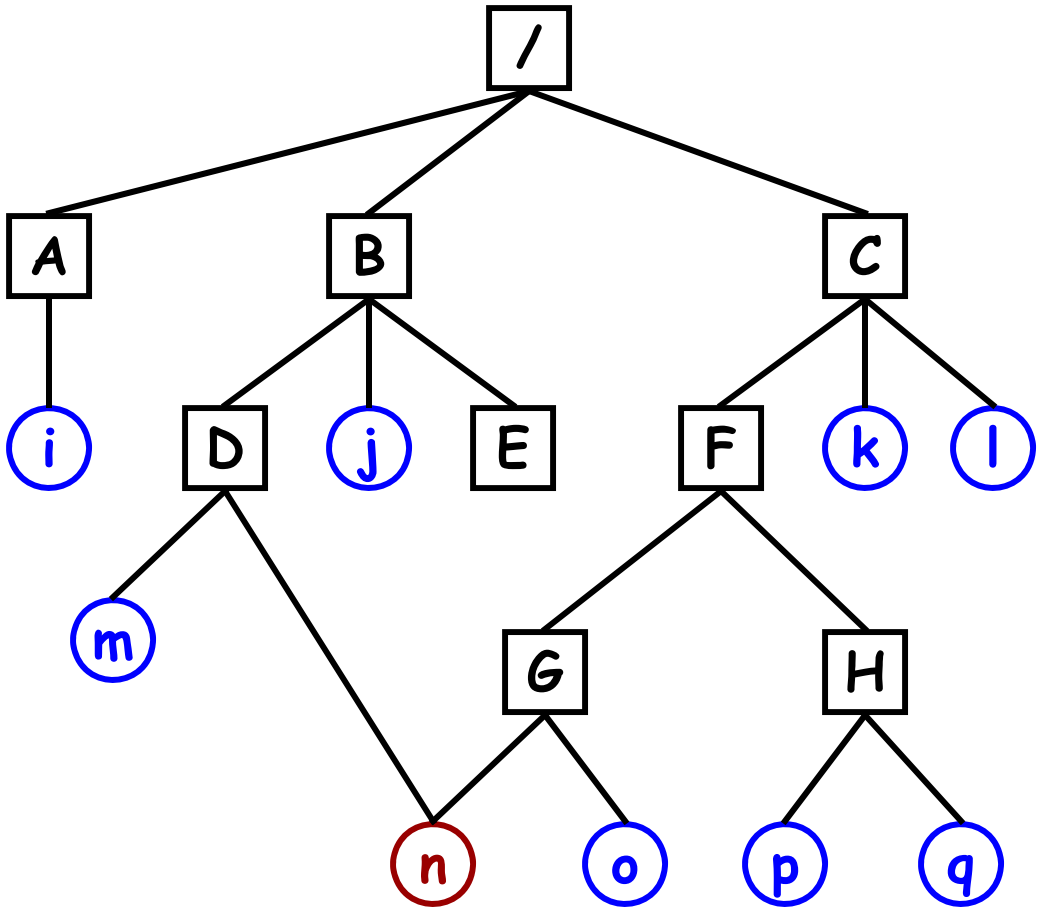


Approach #2



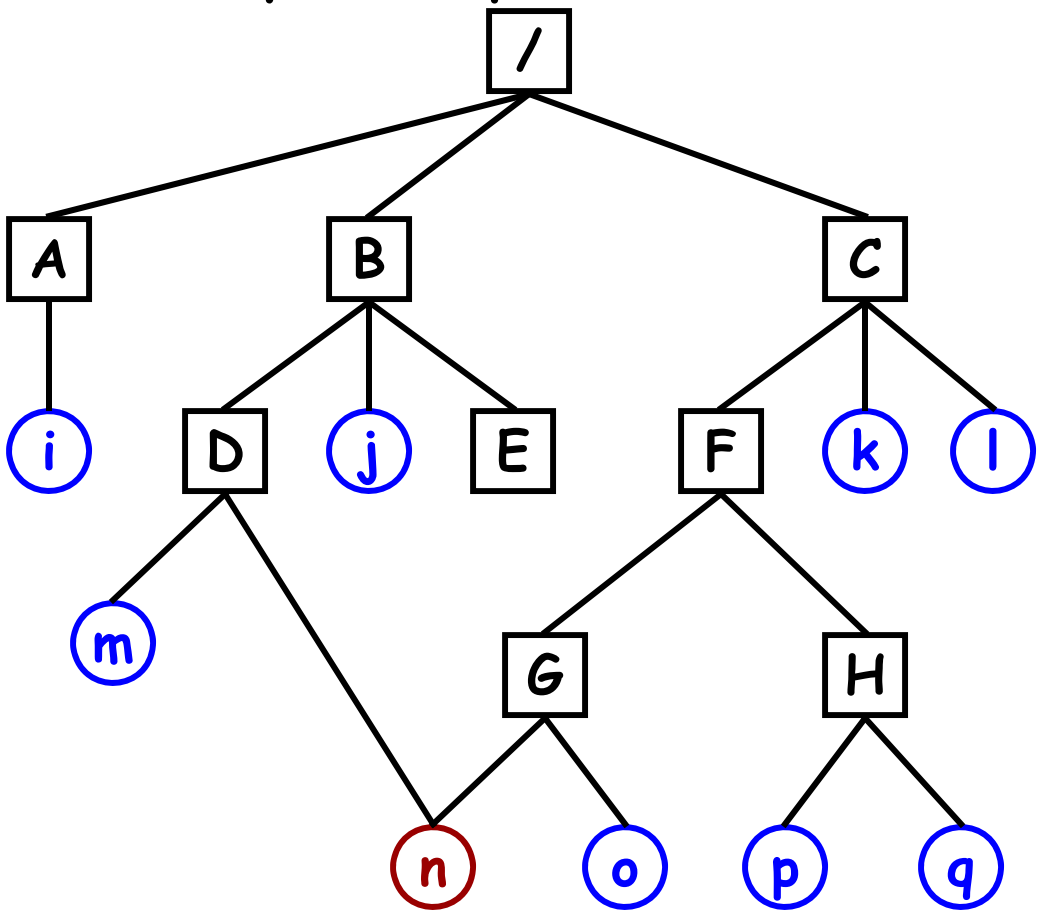
Sharing files

- One file appears in several directories.
- Tree → DAG



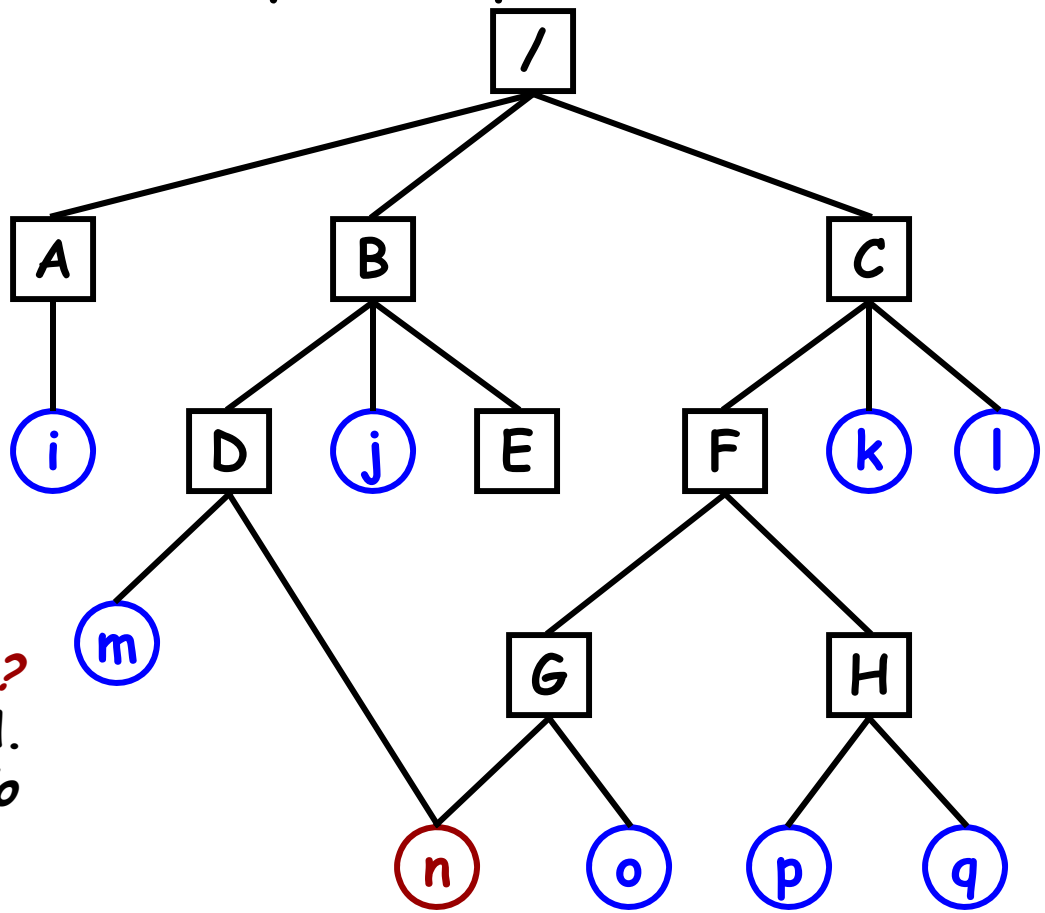
Sharing files

- One file appears in several directories.
- Tree → DAG (Directed Acyclic Graph)



Sharing files

- One file appears in several directories.
- Tree → DAG (Directed Acyclic Graph)

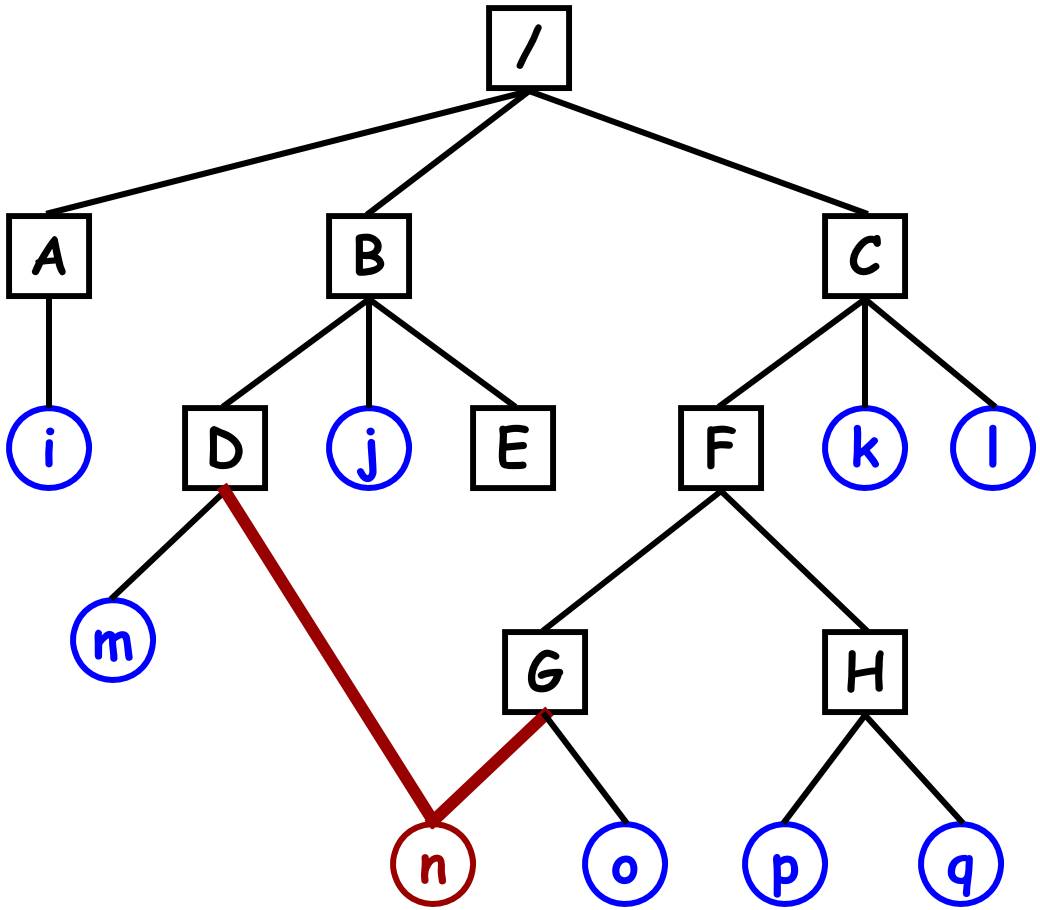


What if the file changes?
New disk blocks are used.
Better not store this info in the directories!!!

Hard links and symbolic links

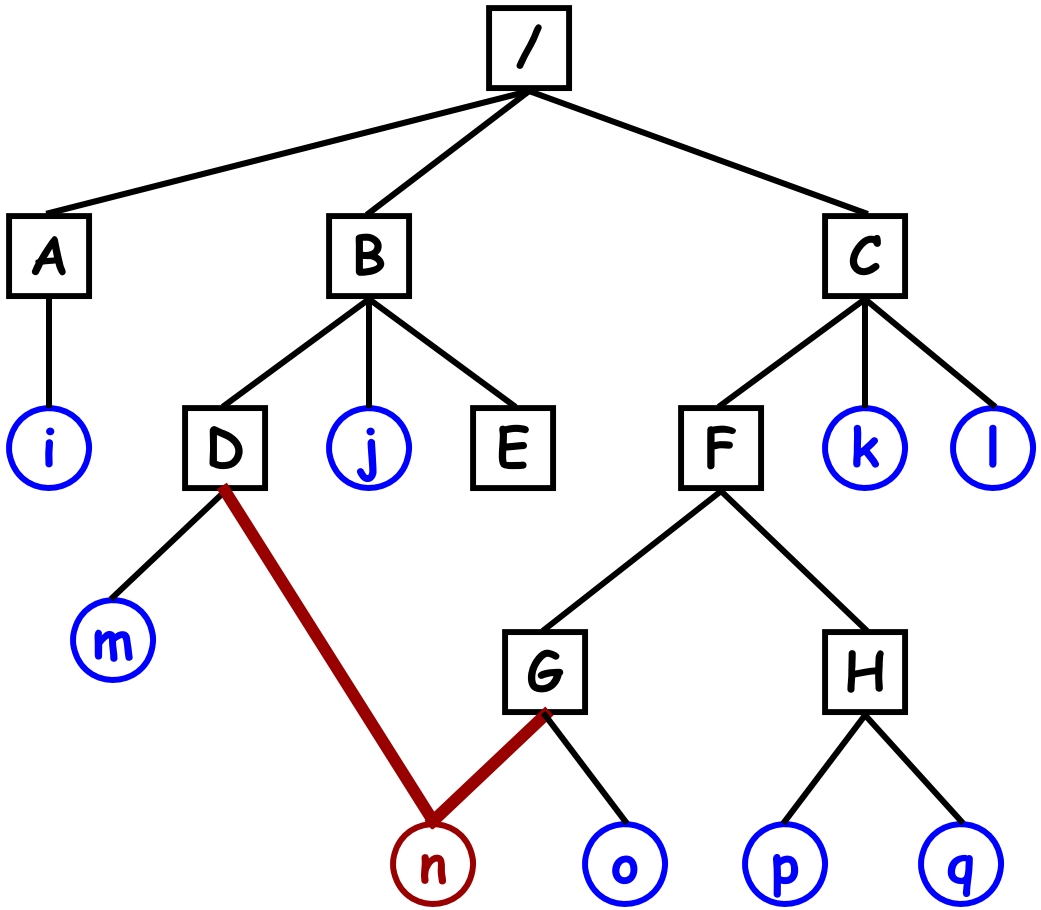
- In Unix:
  - Hard links
    - Both directories point to the same i-node
  - Symbolic links
    - One directory points to the file's i-node
    - Other directory contains the "path"

Hard links



Hard links

- Assume i-node number of "n" is 45.



Hard links

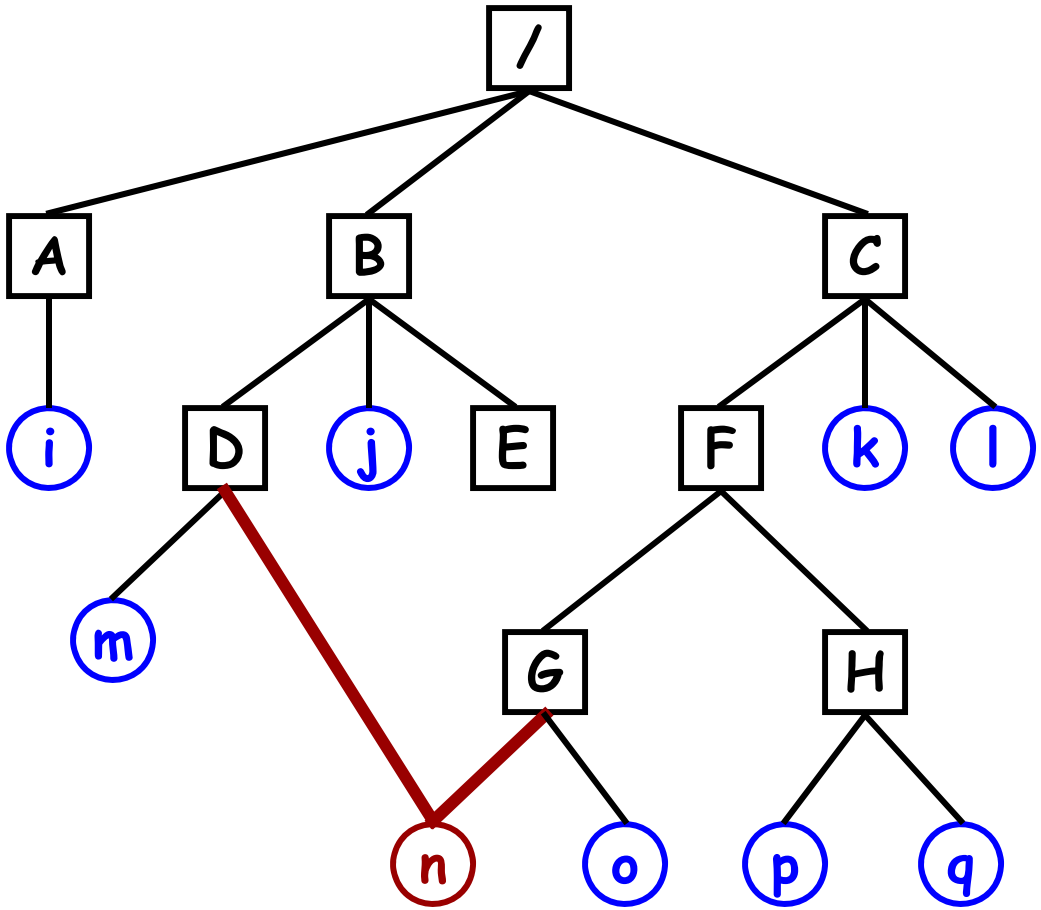
- Assume i-node number of "n" is 45.

Directory "D"

"m"	123
"n"	45
⋮	⋮

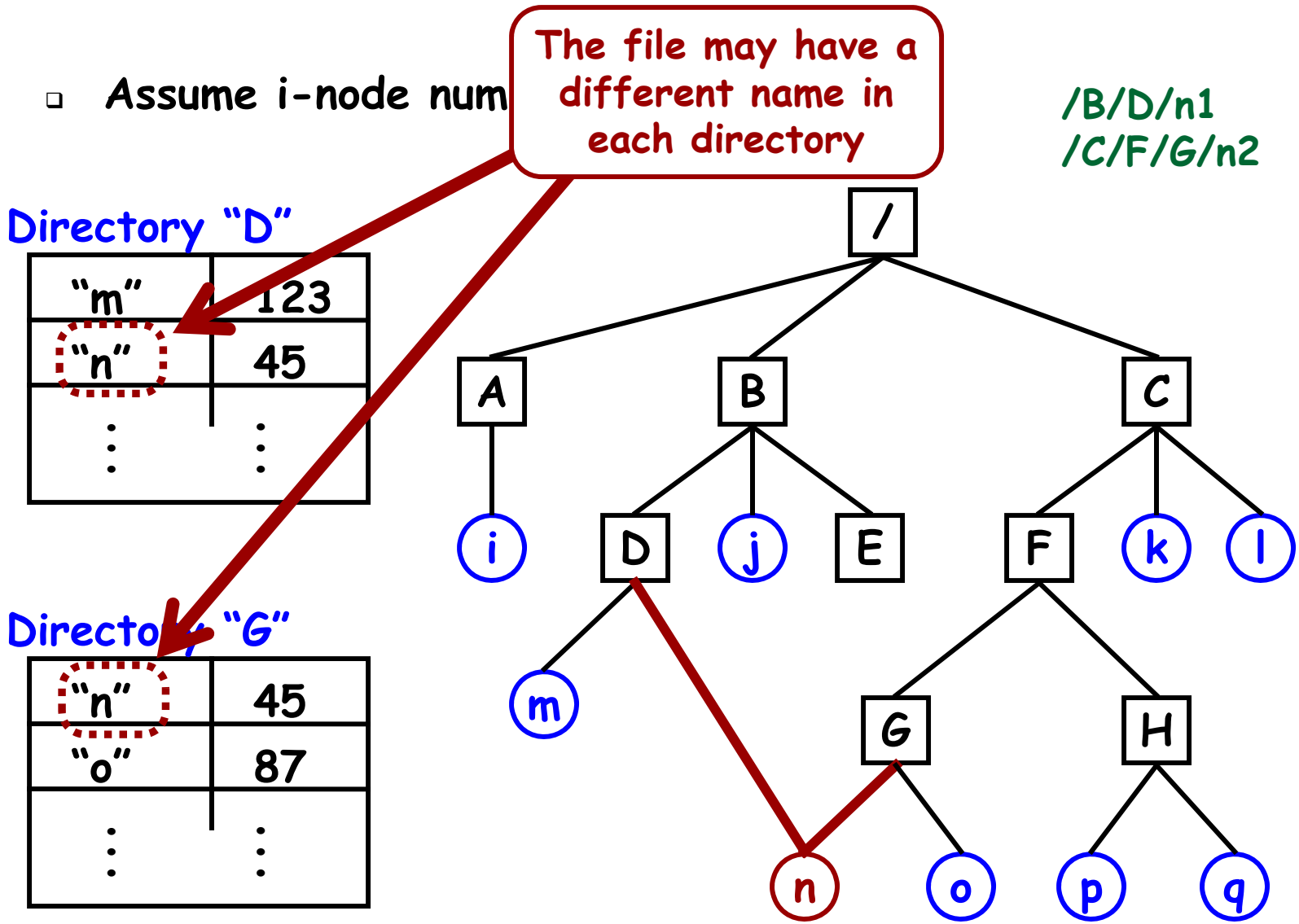
Directory "G"

"n"	45
"o"	87
⋮	⋮



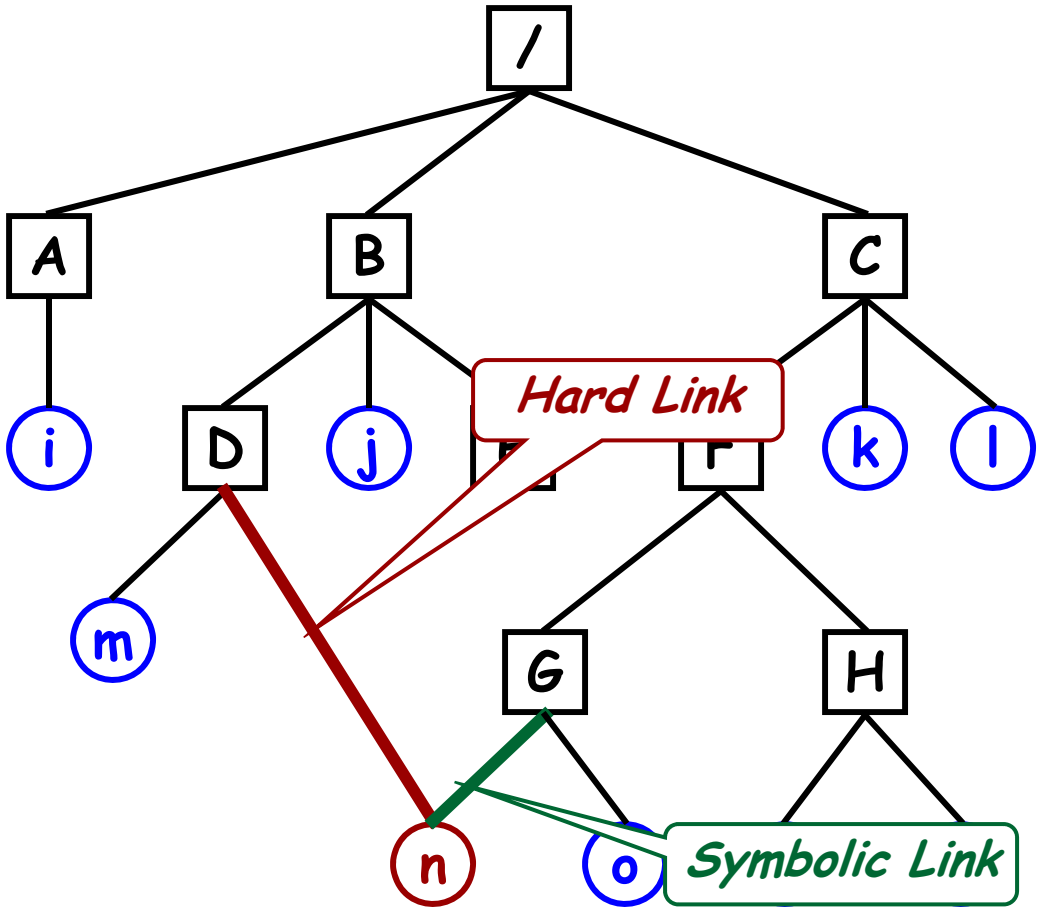
Hard links

- Assume i-node num



Symbolic links

- Assume i-node number of "n" is 45.

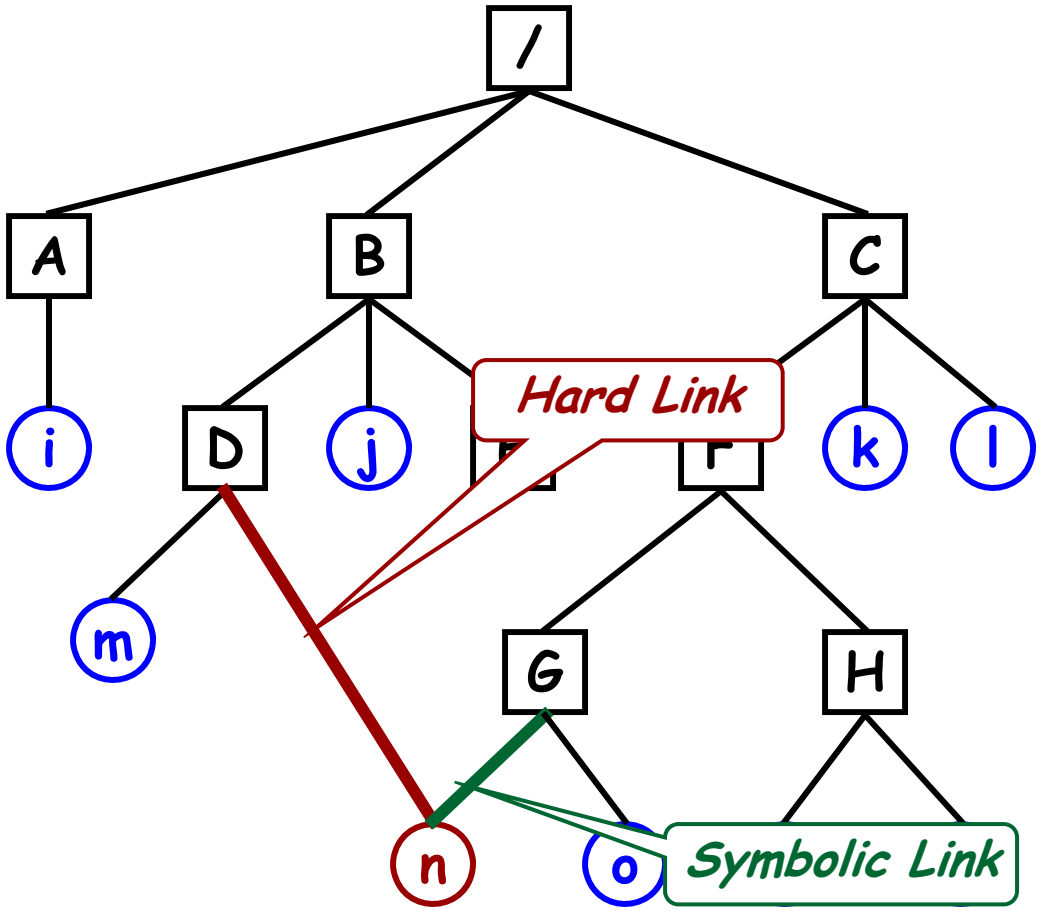


Symbolic links

- Assume i-node number of "n" is 45.

Directory "D"

"m"	123
"n"	45
⋮	⋮



Symbolic links

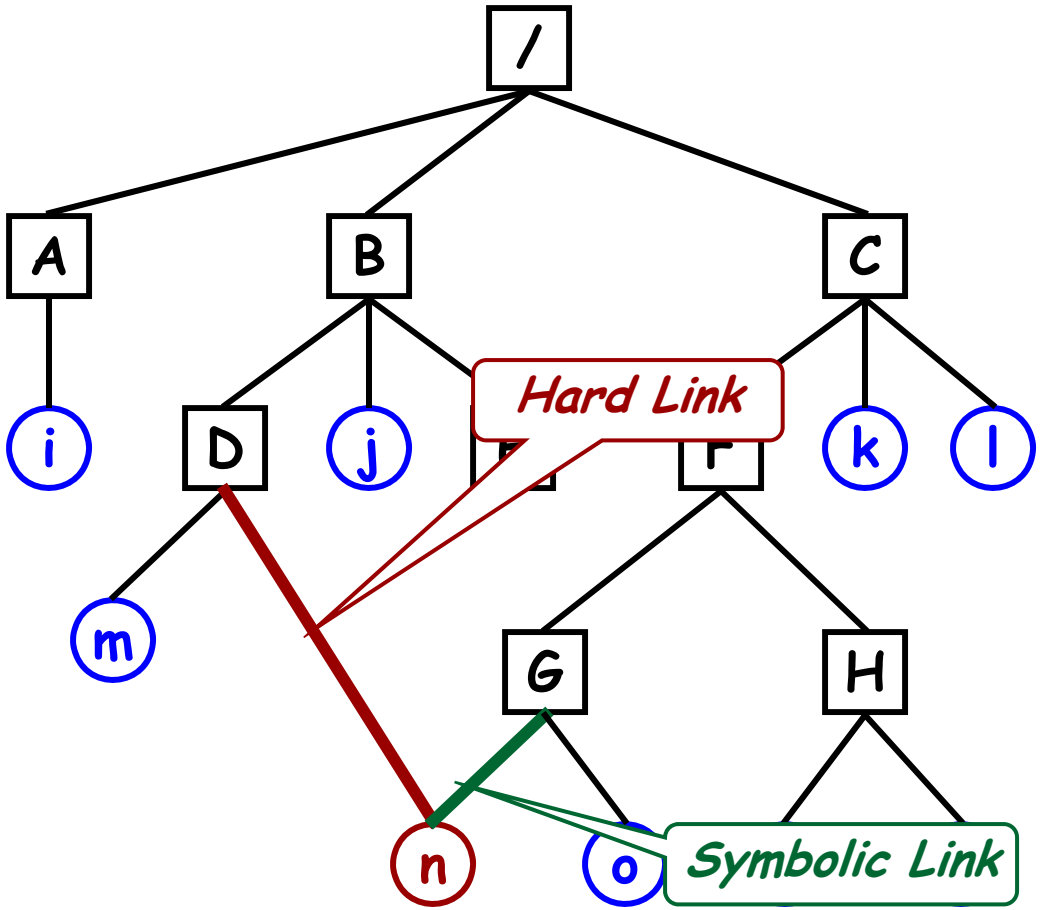
- Assume i-node number of "n" is 45.

Directory "D"

"m"	123
"n"	45
⋮	⋮

Directory "G"

"n"	/B/D/n
"o"	87
⋮	⋮



Symbolic links

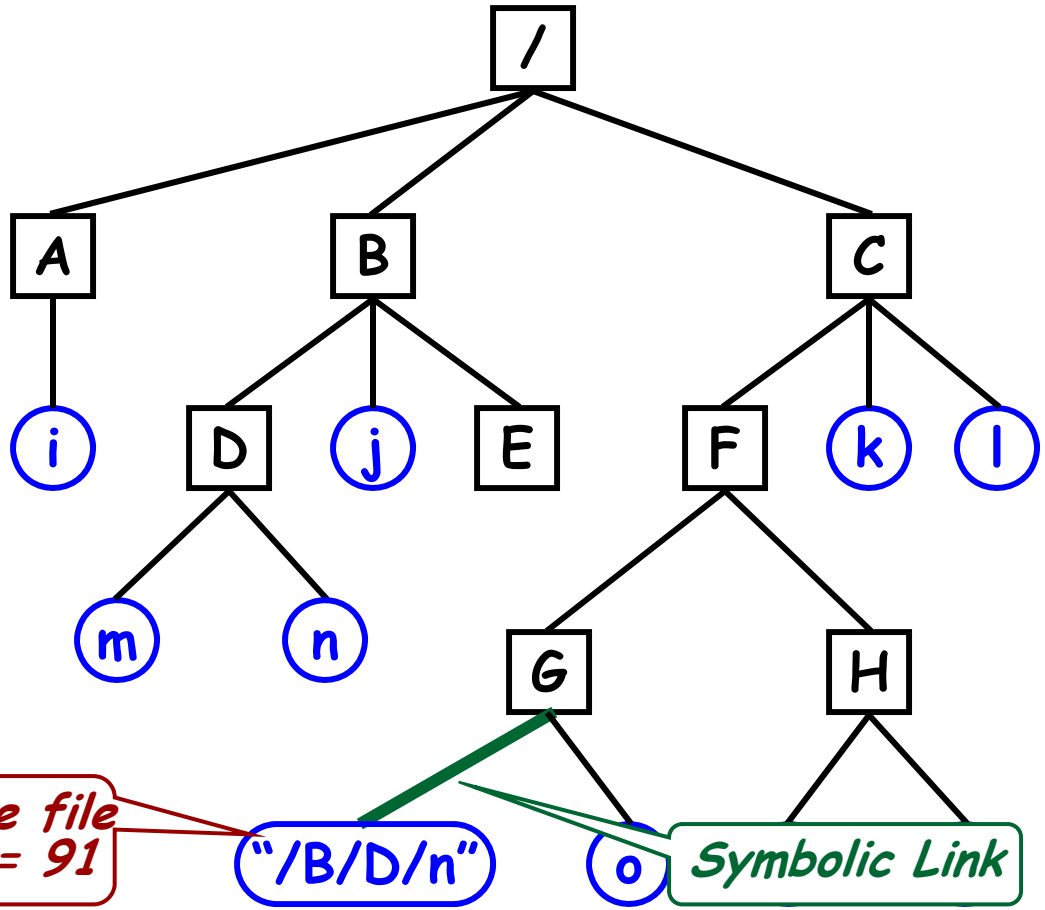
- Assume i-node number of "n" is 45.

Directory "D"

"m"	123
"n"	45
⋮	⋮

Directory "G"

"n"	91
"o"	87
⋮	⋮



Deleting a file

- Directory entry is removed from directory
- All blocks in file are returned to free list

Deleting a file

- **Directory entry is removed from directory**
- **All blocks in file are returned to free list**
- ***What about sharing???***
  - Multiple links to one file (in Unix)

Deleting a file

- **Directory entry is removed from directory**
- **All blocks in file are returned to free list**
- ***What about sharing???***
  - Multiple links to one file (in Unix)
- **Hard Links**
  - Put a "reference count" field in each i-node
  - Counts number of directories that point to the file
  - When removing file from directory, decrement count
  - When count goes to zero, reclaim all blocks in the file

Deleting a file

- **Directory entry is removed from directory**
- **All blocks in file are returned to free list**
- ***What about sharing???***
  - Multiple links to one file (in Unix)
- **Hard Links**
  - Put a "reference count" field in each i-node
  - Counts number of directories that point to the file
  - When removing file from directory, decrement count
  - When count goes to zero, reclaim all blocks in the file
- **Symbolic Link**
  - Remove the real file... (normal file deletion)
  - Symbolic link becomes "broken"

Example: open,read,close

- **fd = open (filename,mode)**
  - Traverse directory tree
  - find i-node
  - Check permissions
  - Set up open file table entry and return fd
- **byte_count = read (fd, buffer, num_bytes)**
  - figure out which block(s) to read
  - copy data to user buffer
  - return number of bytes read
- **close (fd)**
  - reclaim resources

Example: open,write,close

- **byte_count = write (fd, buffer, num_bytes)**
  - figure out how many and which block(s) to write
  - Read them from disk into kernel buffer(s)
  - copy data from user buffer
  - send modified blocks back to disk
  - adjust i-node entries
  - return number of bytes written