R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه بیست و سوم – مدیریت فایل

جلسه‌ی گذشته

SSD

Nonvolatile Memory Devices

- If disk-drive like, then called **solid-state disks** (**SSDs**)
- Other forms include **USB drives** (thumb drive, flash drive), DRAM disk replacements, surface-mounted on motherboards, and main storage in devices like smartphones
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span – need careful management
- Less capacity
- But much faster
- Busses can be too slow -> connect directly to PCI for example
- No moving parts, so no seek time or rotational latency

Nonvolatile Memory Devices

- Have characteristics that present challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - *Must first be erased, and erases happen in larger ”block” increments*
  - *Can only be erased a limited number of times before worn out – ~ 100,000*
  - *Life span measured in **drive writes per day (DWPD)***
    - A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warrantee period without failing

RAID Structure

- **RAID** – **redundant array of inexpensive disks**
  - *multiple disk drives provides reliability via **redundancy***
- Increases the **mean time to failure**
- **Mean time to repair –** exposure time when another failure could cause data loss
- **Mean time to data loss** based on above factors
- If mirrored disks fail independently, consider disk with 100,000 **mean time to failure** and 10 hour mean time to repair
  - *Mean time to data loss is* $100,000^2 / (2 * 10) = 500 * 10^6$ *hours, or 57,000 years!*

RAID Levels

- **RAID 0**: High speed, no redundancy.
- **RAID 1**: High fault tolerance (mirroring), low storage efficiency.
- **RAID 2**: Obsolete; bit-level striping with Hamming code.
- **RAID 3**: Byte-level striping, single parity drive (low write performance).
- **RAID 4**: Block-level striping, single parity drive (write bottleneck).
- **RAID 5**: Block-level striping, distributed parity (best overall balance of speed, efficiency, and redundancy).

FREE BLOCK

Disk Space Management

- The OS must choose a disk “block” size...
  - *The amount of data written to/from a disk*
  - *Must be some multiple of the disk’s sector size*

- How big should a disk block be?
  - *= Page Size?*
  - *= Sector Size?*
  - *= Track size?*

Disk Space Management

- Large block sizes:
  - *Internal fragmentation*
  - *Last block has (on average) 1/2 wasted space*
  - *Lots of very small files; waste is greater*

- Small block sizes:
  - *More seeks; file access will be slower*

Block Size Tradeoff

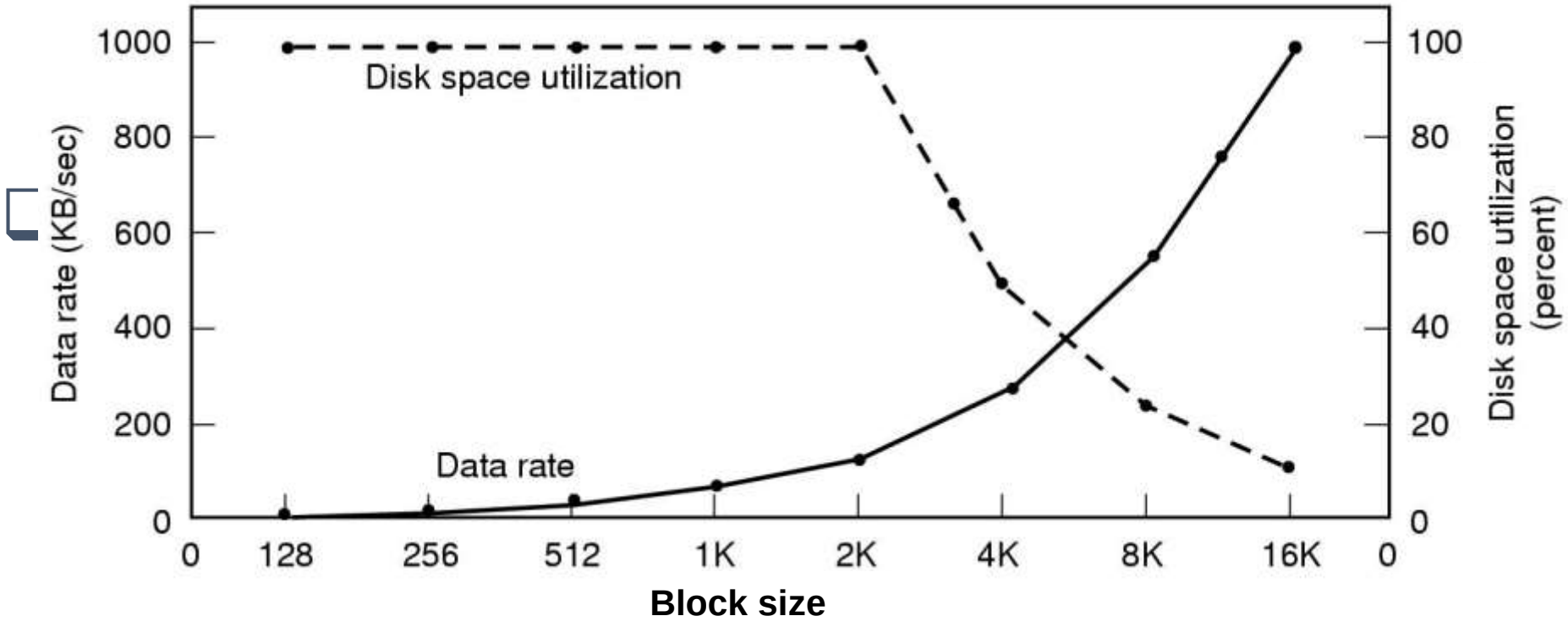
- Smaller block size?
  - *Better disk utilization*
  - *Poor performance*
- Larger block size?
  - *Lower disk space utilization*
  - *Better performance*

Simple Example

- A Unix System
  - *1000 users, 1M files*
  - *Median file size = 1,680 bytes*
  - *Mean file size = 10,845 bytes*
  - *Many small files, a few really large files*

- For simplicity, let's assume all files are 2 KB...
  - *What happens with different block sizes?*
  - *The tradeoff will depend on details of disk performance*

Block size tradeoff



Assumption: All files are 2K bytes
Given: Physical disk properties
Seek time=10 msec
Transfer rate=15 Mbytes/sec
Rotational Delay=8.33 msec * 1/2

Managing Free Blocks

- Approach #1:
  - *Keep a bitmap*
  - *1 bit per disk block*

- Approach #2
  - *Keep a free list*

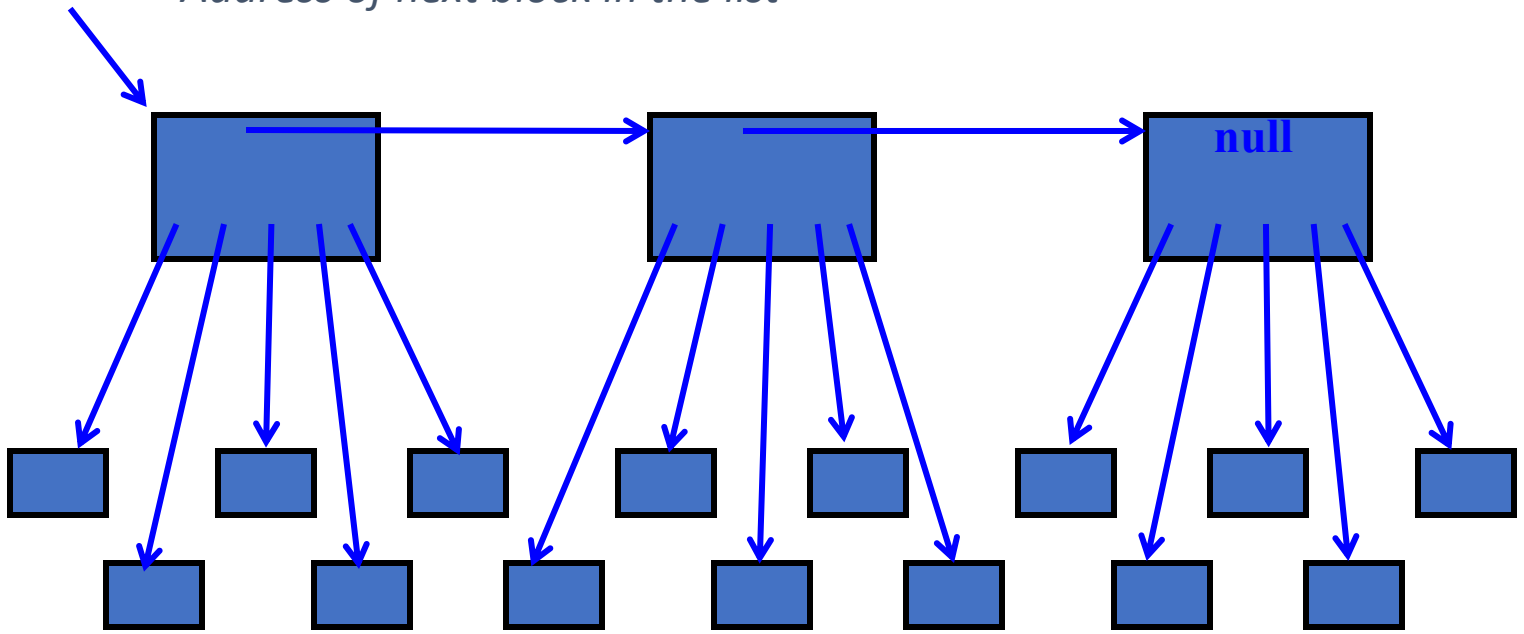
Managing Free Blocks

- Approach #1:
  - *Keep a bitmap*
  - *1 bit per disk block*
    - Example:
      - *1 KB block size*
      - *16 GB Disk $\Rightarrow$ 16M blocks = 2^{24} blocks*
    - Bitmap size = 2^{24} bits $\Rightarrow$ 2K blocks
      - *1/8192 space lost to bitmap*

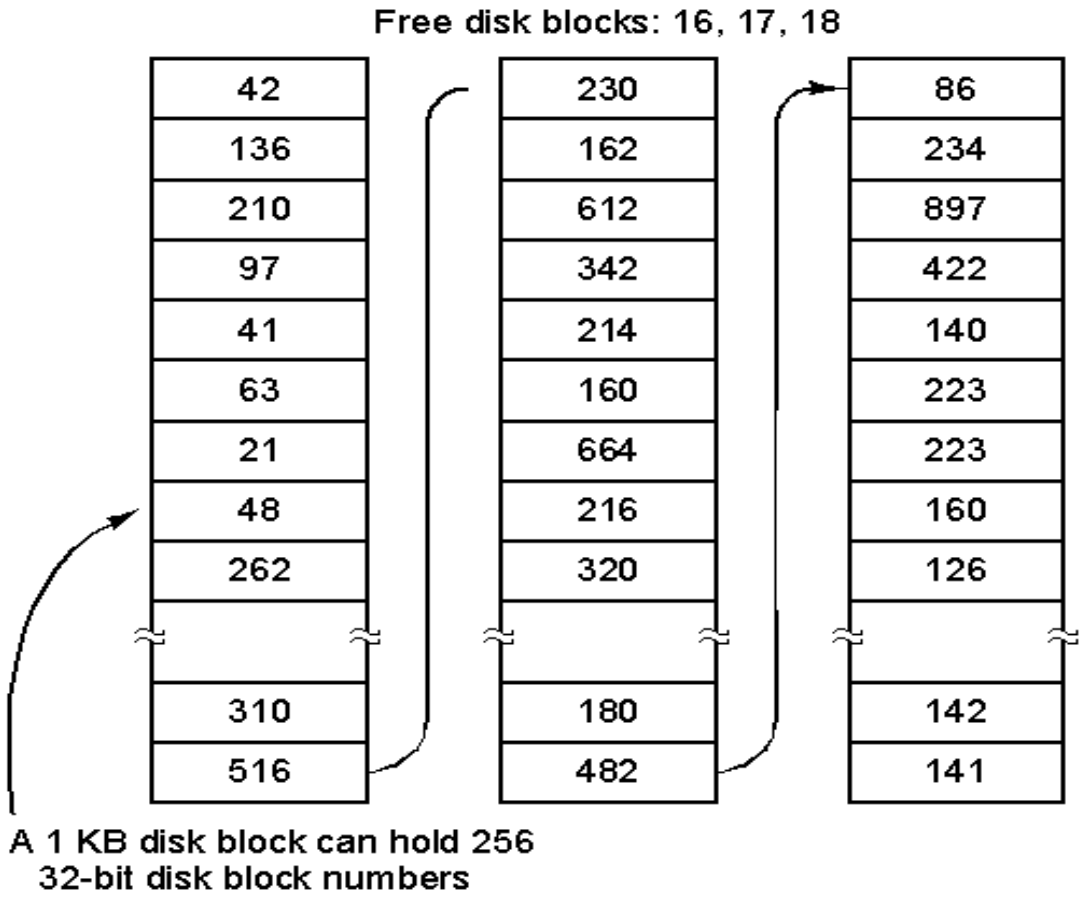
- Approach #2
  - *Keep a free list*

Free List of Disk Blocks

- Linked List of Free Blocks
- Each block on disk holds
  - *A bunch of addresses of free blocks*
  - *Address of next block in the list*



Free list of disk blocks



Assumptions:
- Block size = 1K
- Each block addr = 4bytes
- Each block holds
  - 255 ptrs to free blocks
  - 1 ptr to the next block

This approach takes more space than bitmap…
But "free" blocks are used, so no real loss!

Free List of Disk Blocks

- Two kinds of blocks:
  - *Free Blocks*
  - *Block containing pointers to free blocks*
- Always keep one block of pointers in memory
  - *This block may be partially full*
- Need a free block?
  - *This block gives access to 255 free blocks*
  - *Need more?*
    - Look at the block's "next" pointer
    - Use the pointer block itself
    - Read in the next block of pointers into memory

Free List of Disk Blocks

- To return a block (X) to the free list
  - *If the block of pointers (in memory) is not full, add X to it*

Free List of Disk Blocks

- To return a block (X) to the free list
  - *If the block of pointers (in memory) is not full, add X to it*
  - *If the block of pointers (in memory) is full*
    - Write it to out to the disk
    - Start a new block in memory
    - Use block X itself for a pointer block
      - *All empty pointers except for the next pointer*

Free List of Disk Blocks

- Scenario:
  - *Assume the block of pointers in memory is almost empty*
  - *A few free blocks are needed.*
    - This triggers disk read to get next pointer block
  - *Now the block in memory is almost full*
  - *Next, a few blocks are freed*
  - *The block fills up*
    - This triggers a disk write of the block of pointers
- Problem:
  - *Numerous small allocates and frees, when block of pointers is right at boundary results in lots of disk I/O*

Free list of disk blocks

- Solution
  - *Try to keep the block in memory about 1/2 full*
  - *When the block in memory fills up...*
    - Break it into 2 blocks (each 1/2 full)
    - Write one out to disk
- A similar solution
  - *Keep 2 blocks of pointers in memory at all times*
  - *When both fill up*
    - Write out one
  - *When both become empty*
  - *Read in one new block of pointers*

Comparison: Free List vs Bitmap

- Desirable:
  - *Keep all the blocks in one file close together*
- Free Lists:
  - *Free blocks are all over the disk*
  - *Allocation comes from (almost) random location*
- Bitmap:
  - *Much easier to find a free block "close to" a given position*
  - *Bitmap implementation:*
    - Keep 2 MByte bitmap in memory
    - Keep only one block of bitmap in memory at a time

جلسه‌ی جدید

فایل

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 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*

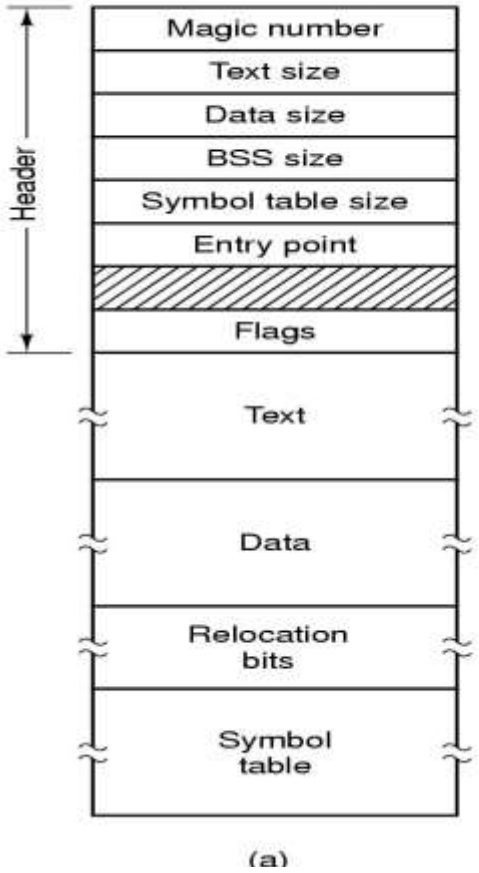
Typical File Extensions

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

Executable Files

- Executable files
  - *The OS must understand the format of executable files in order to execute programs*
  - *The exec system call needs this*
    - Put program and data in process address space

Executable File Format



An executable file

Executable File Format - Details

- **Magic Number**:
  - *A unique identifier that indicates the file type (e.g., executable, object, or script).*
  - *It helps the operating system recognize the format of the file.*
- **Text Size**:
  - *Specifies the size of the **text segment** (the machine instructions or code) in the file.*
- **Data Size**:
  - *Specifies the size of the **data segment**, which contains initialized global and static variables.*
- **BSS Size**:
  - *Indicates the size of the **BSS segment**.*
  - *BSS stands for **"Block Started by Symbol"** and contains uninitialized global and static variables. This space is allocated at runtime.*

Executable File Format - Details

- **Symbol Table Size**:
  - *Specifies the size of the **symbol table**.*
  - *The symbol table contains information about symbols (like function and variable names) used for linking and debugging.*
- **Entry Point**:
  - *This is the memory address where the program begins execution.*
  - *It marks the start of the **main function** or the first instruction to execute.*
- **Flags**:
  - *Contains **status information** or settings about the file, such as execution permissions or format specifications.*

Executable File Format - Details

- **Text**:
  - *The **code segment** of the program where the compiled machine instructions reside.*
  - *This is the part of the file that will be loaded into memory for execution.*
- **Data**:
  - *Contains the **initialized global and static variables**.*
- **Relocation Bits**:
  - *These are used for **relocation**, ensuring that the program can execute correctly regardless of where it is loaded in memory.*
- **Symbol Table**:
  - *This table contains symbolic information, like:*
    - Function names
    - Variable names
    - Debugging information
  - *It is particularly useful during linking and debugging.*

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 Models

- 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*
  - *Move position (seek), then read*

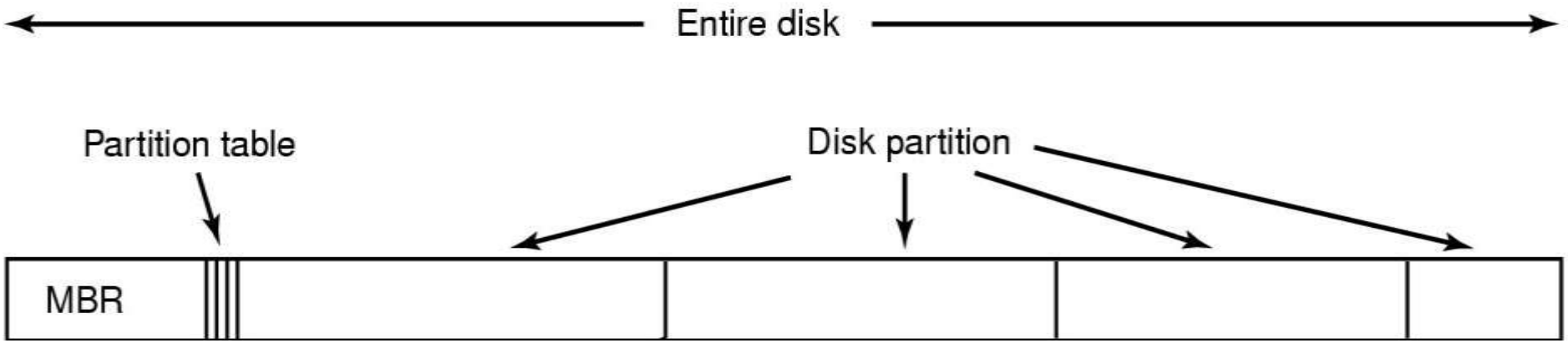
File-Related System Calls

- 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 System Call Examples

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

File Storage on Disk



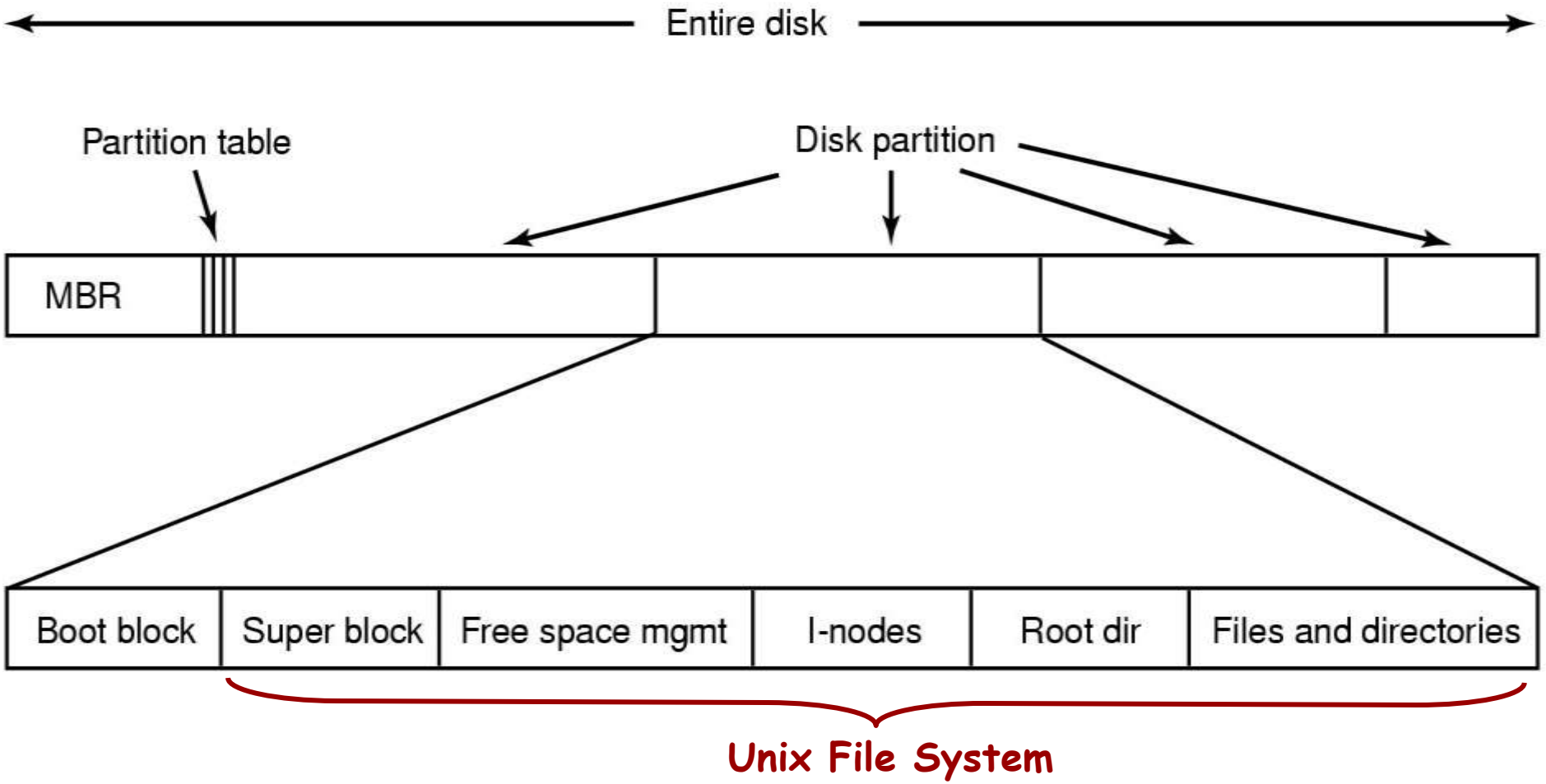
File Storage On Disk

- 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*

File Storage On Disk

- 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*

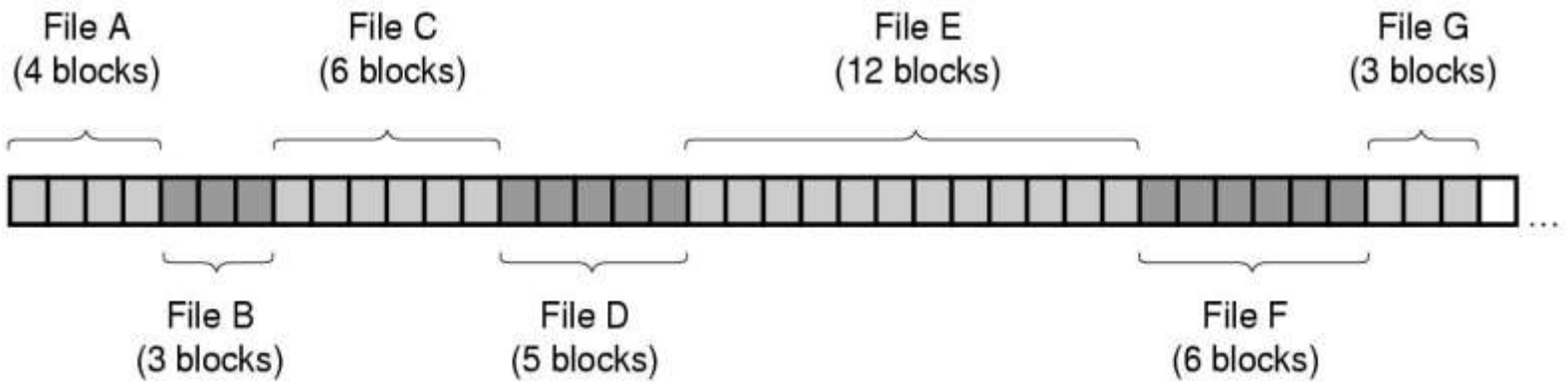
An Example Disk



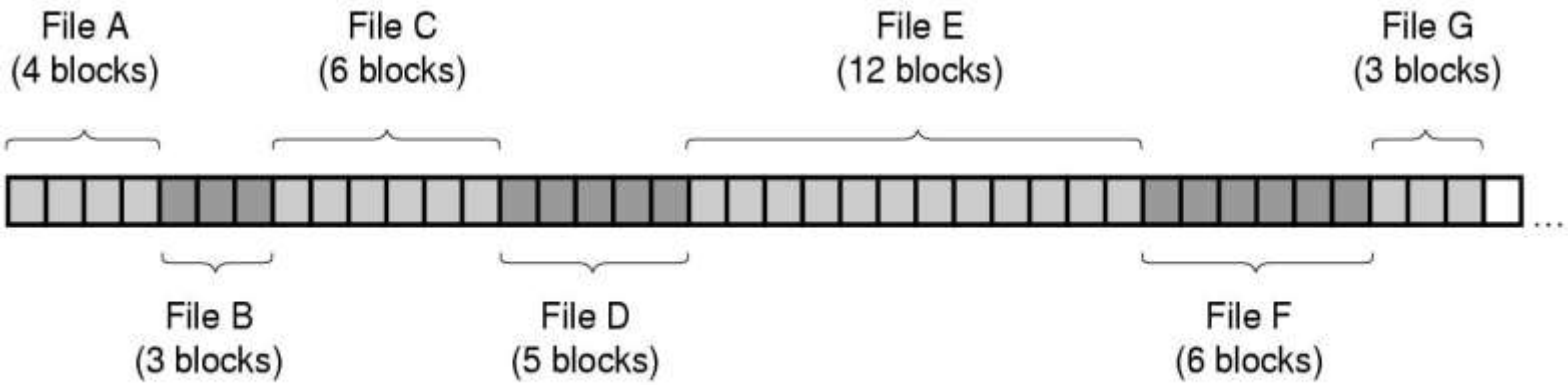
File Bytes vs Disk Sectors

- 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*

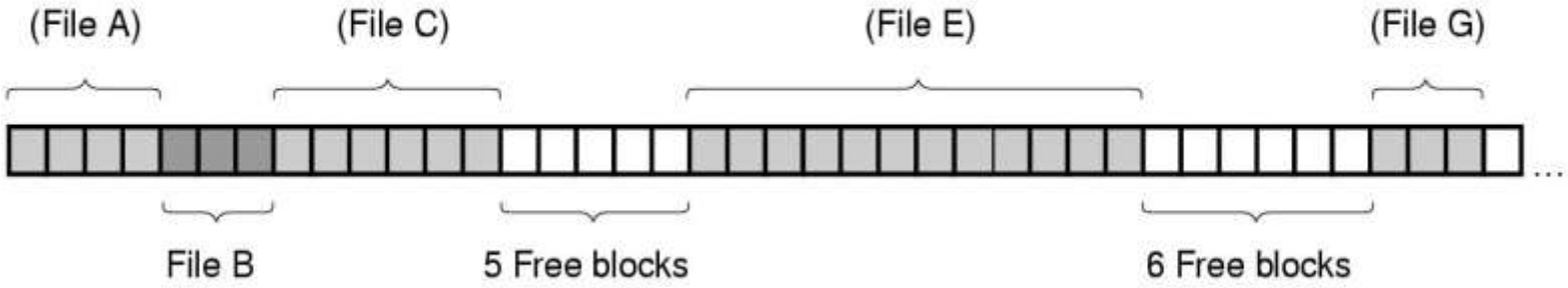
Contiguous Allocation



Contiguous Allocation



After deleting D and F…



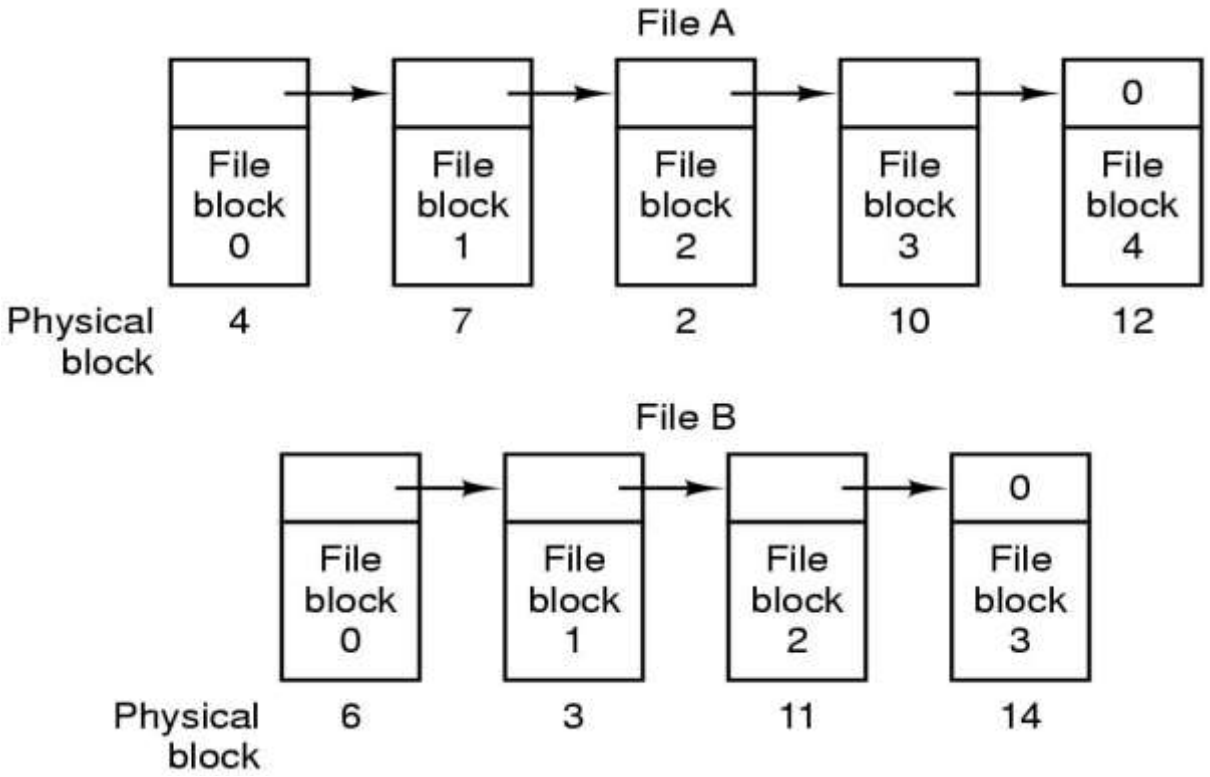
Contiguous Allocation

- 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

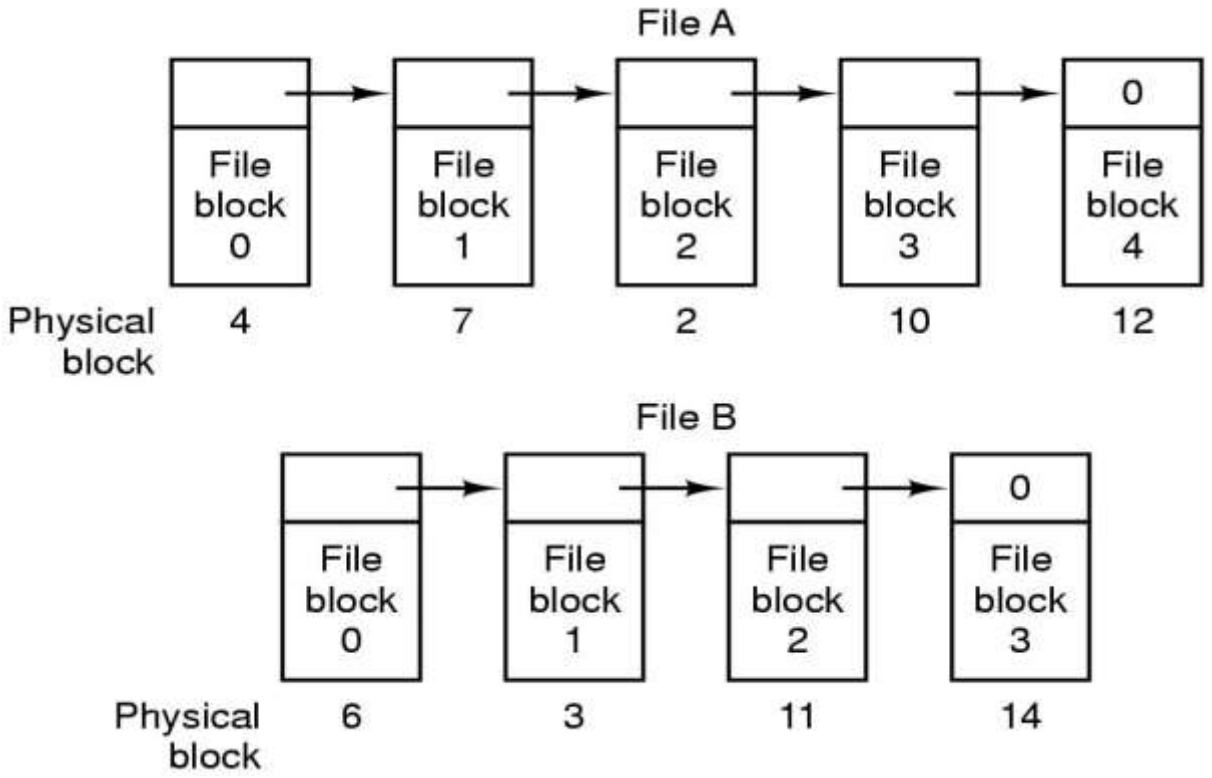
- Good for CD-ROMs
  - *All file sizes are known in advance*
  - *Files are never deleted*

Linked List Allocation



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

Linked List Allocation

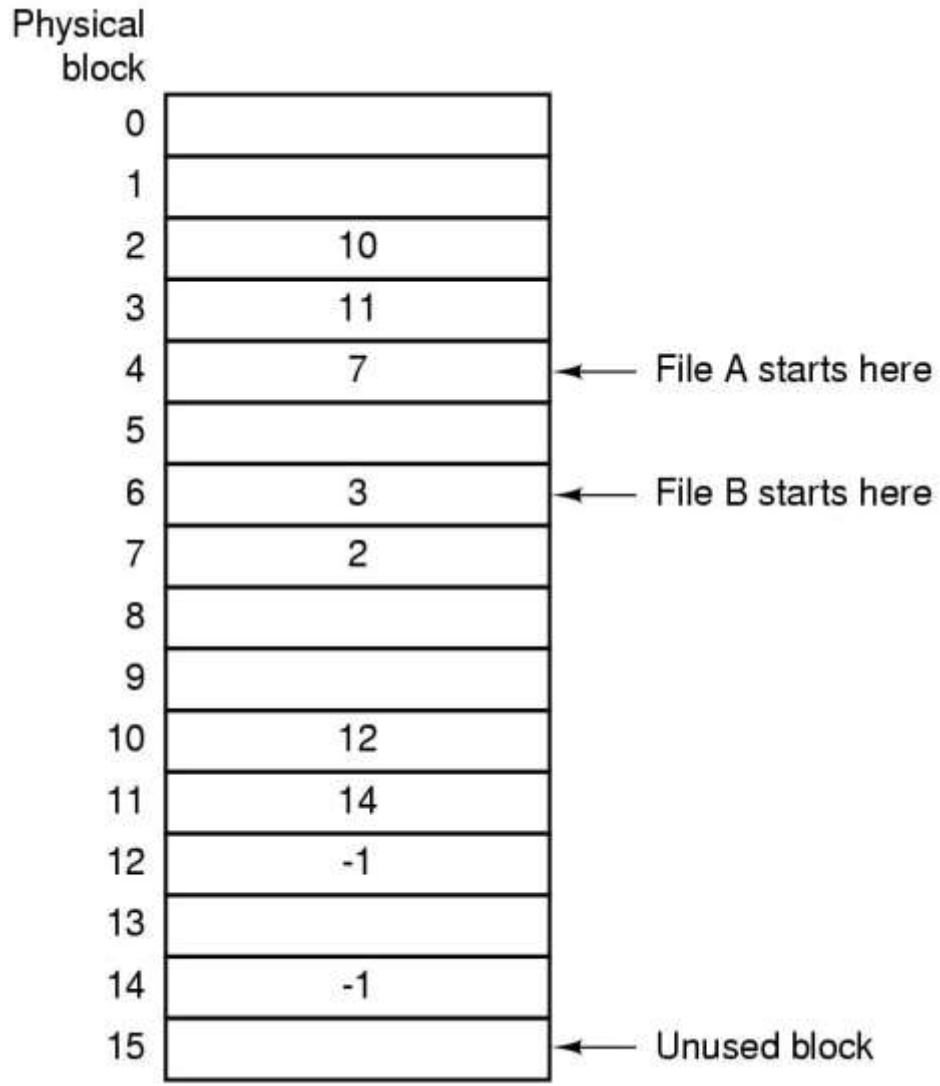


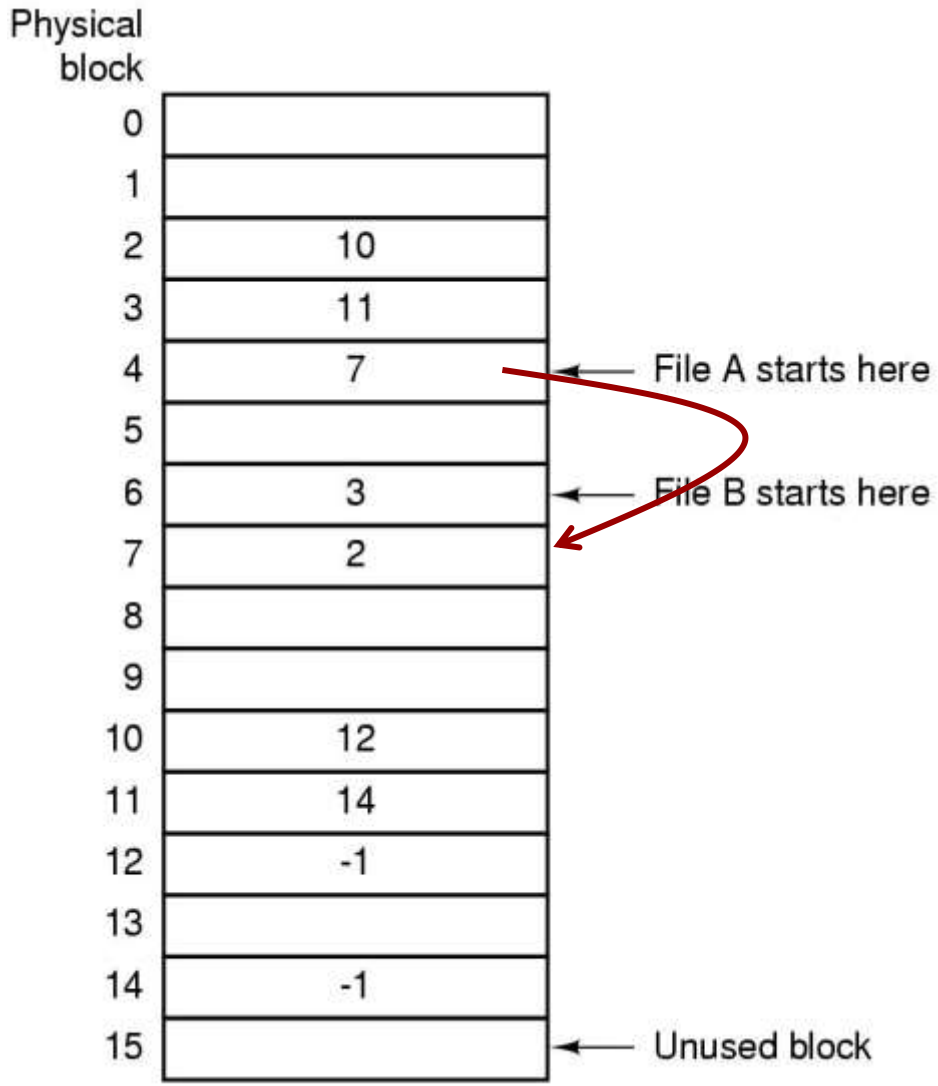
- Random access into the file is slow!

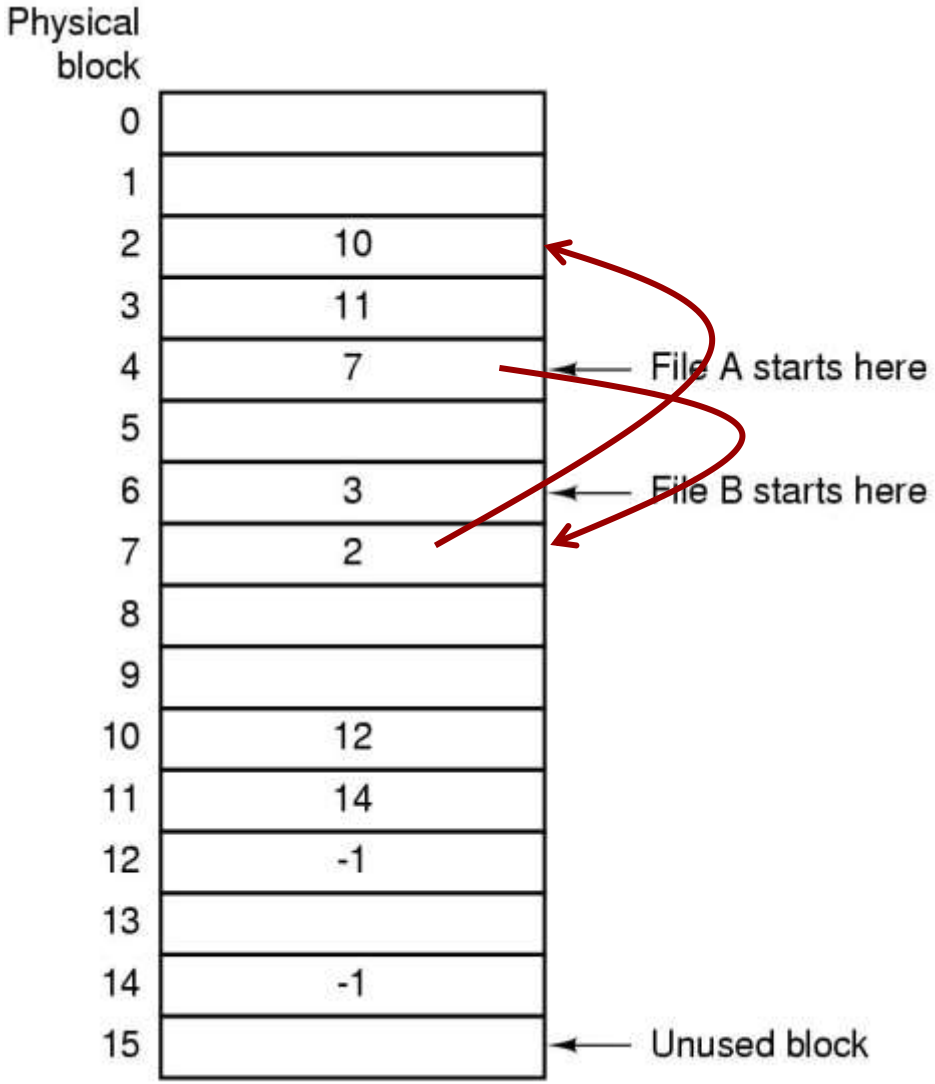
FAT

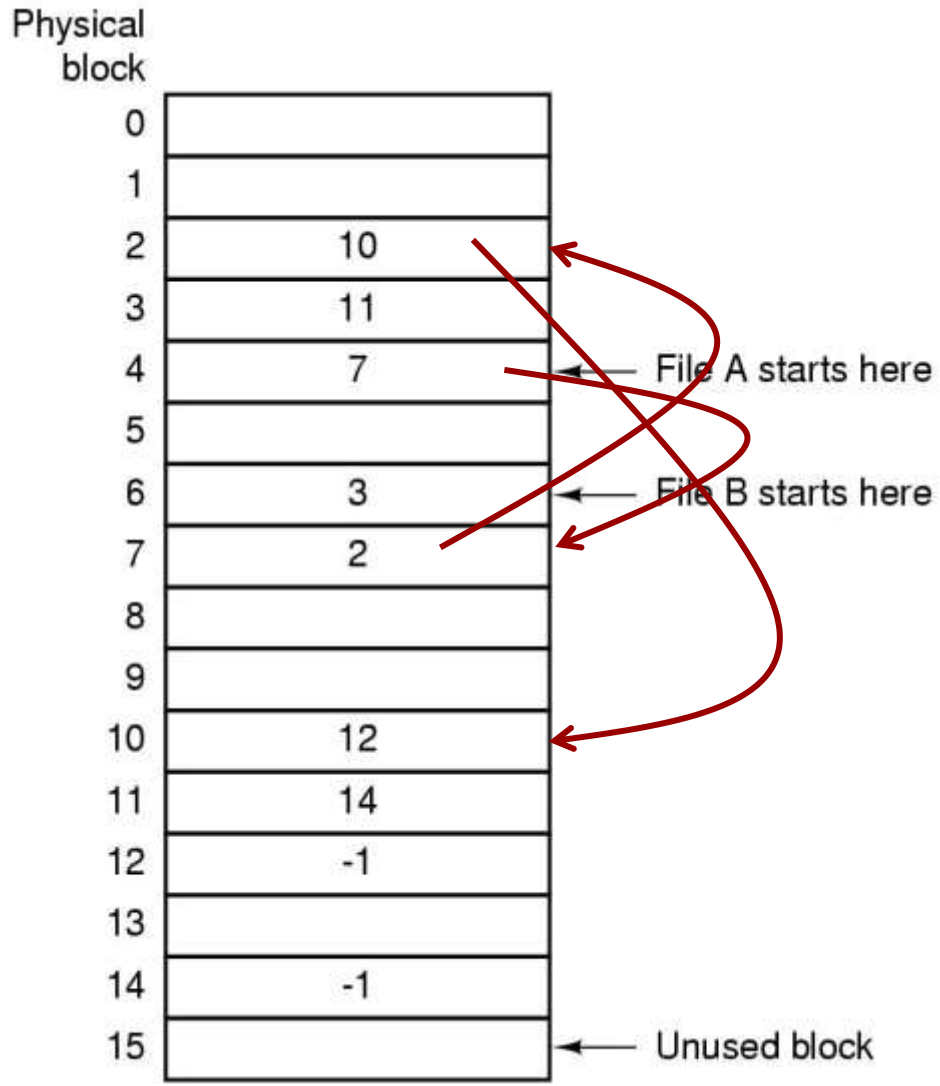
File Allocation Table (FAT)

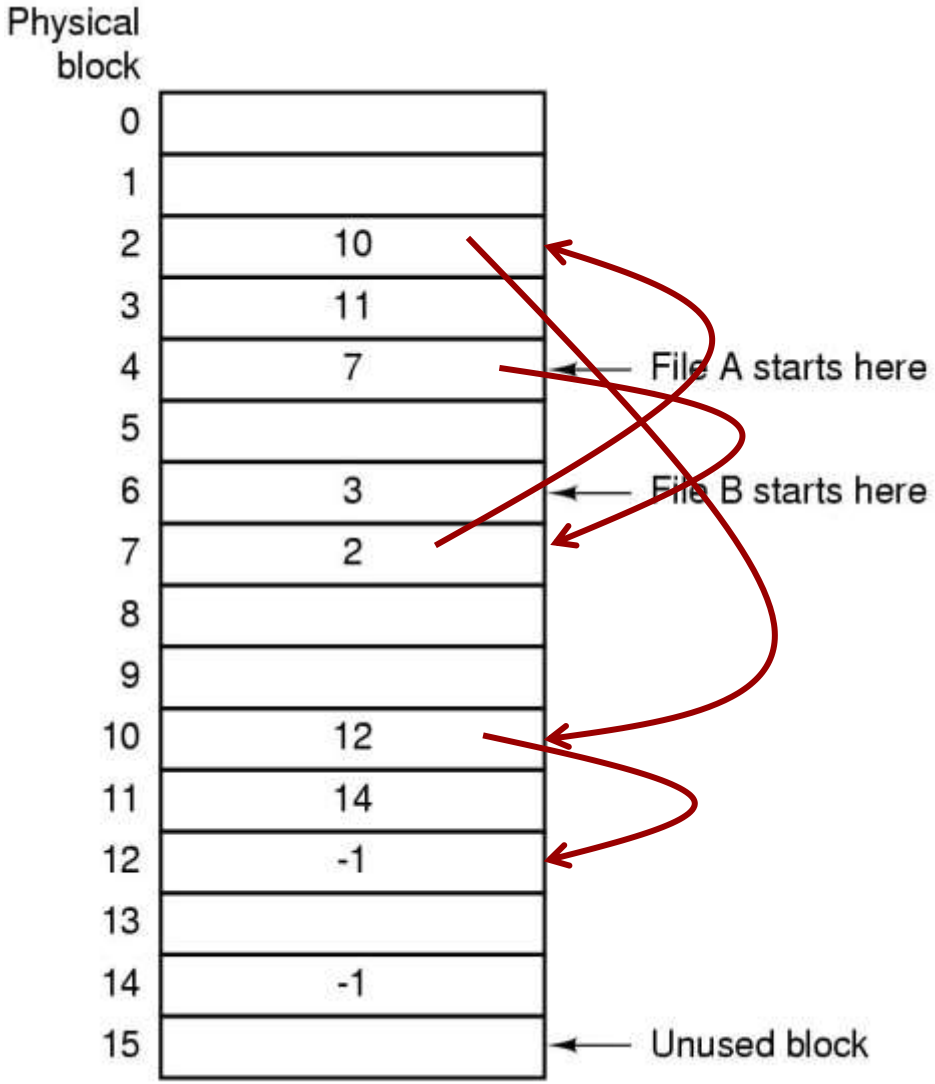
- Keep a table in memory
- One entry per block on the disk
- Each entry contains the address of the “next” block
  - *End of file marker (-1)*
  - *A special value (-2) indicates the block is free*

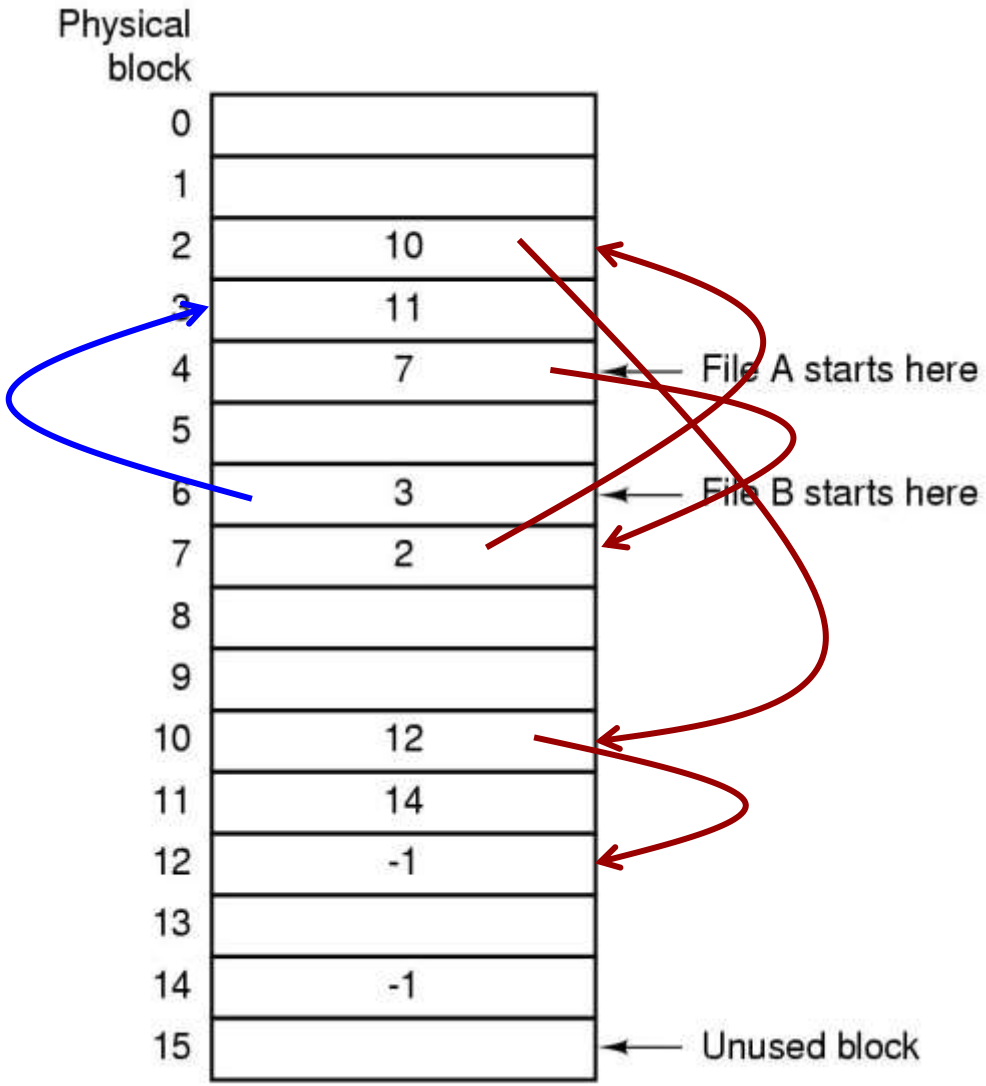

Physical block
0
1
2
10
3
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block

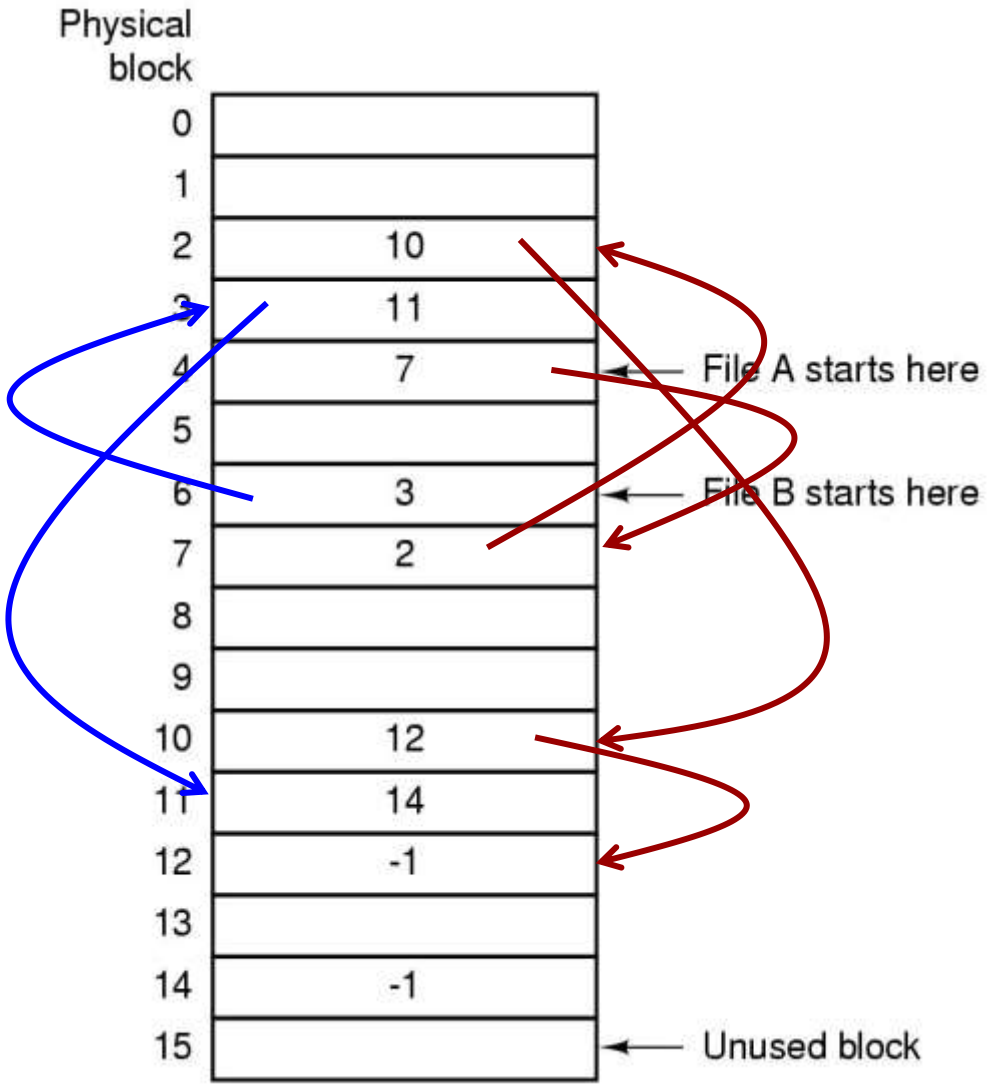

Physical block
0
1
2
10
3
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block

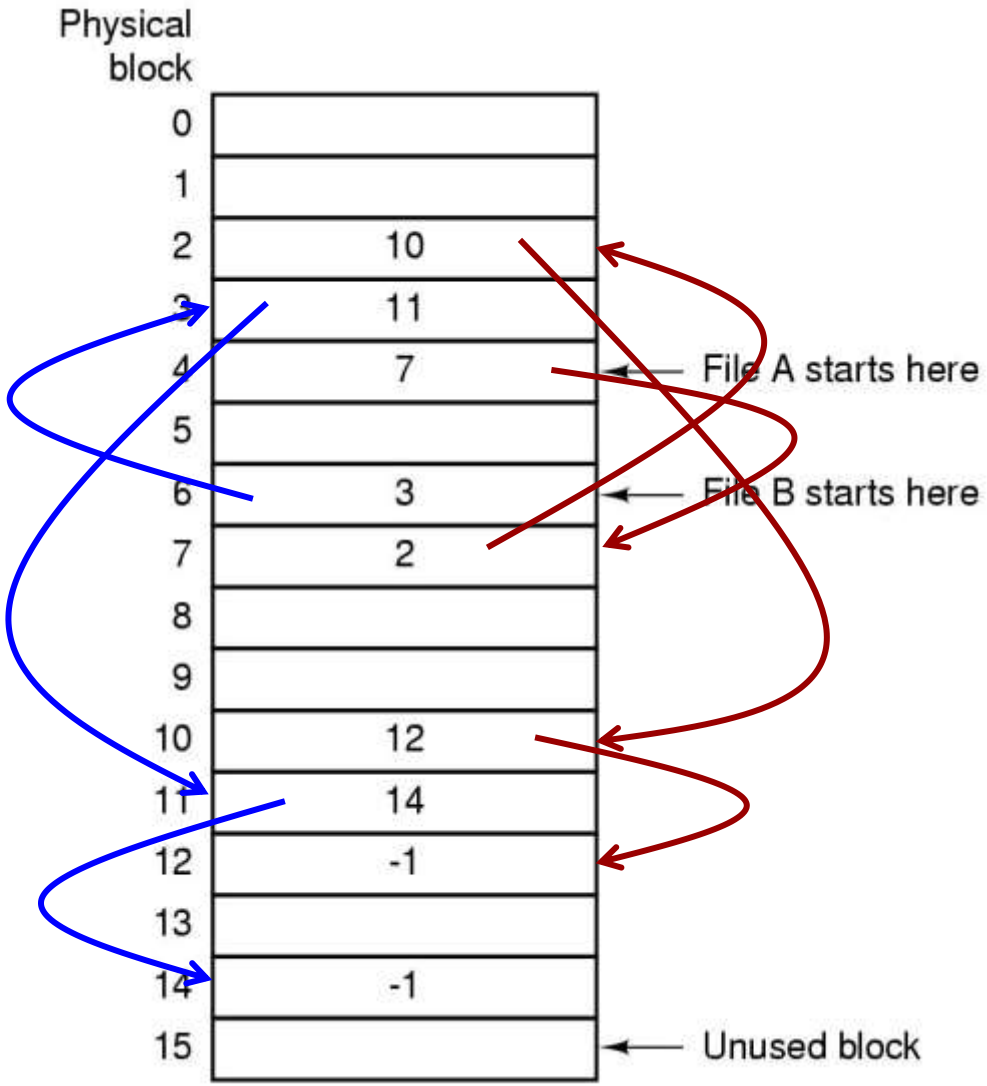

Physical block
0
1
2
10
3
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block


Physical block
0
1
2
10
3
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block


Physical block
0
1
2
10
3
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block


Physical block
0
1
2
10
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block


Physical block
0
1
2
10
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block


Physical block
0
1
2
10
11
4
7
File A starts here
5
6
3
File B starts here
7
2
8
9
10
12
11
14
12
-1
13
14
-1
15
Unused block

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!*
  - *Example:*
    - *200 GB = disk size*
    - *1 KB = block size*
    - *4 bytes = FAT entry size*
    - *800 MB of memory used to store the FAT*

FAT Size

- **200 GB = disk size**
- **1 KB = block size**: Each block on the disk is 1 KB.
- **4 bytes = FAT entry size**: Each block entry in the FAT table takes 4 bytes.
- **Memory used**:
  - Total number of blocks = $\text{Disk size}/\text{Block size}$

$$\text{Total blocks} = \frac{200 \times 1024 \times 1024\,\text{KB}}{1\,\text{KB}} = 200 \times 1024 \times 1024\,\text{blocks}$$

  - FAT size = Total blocks × 4 bytes.

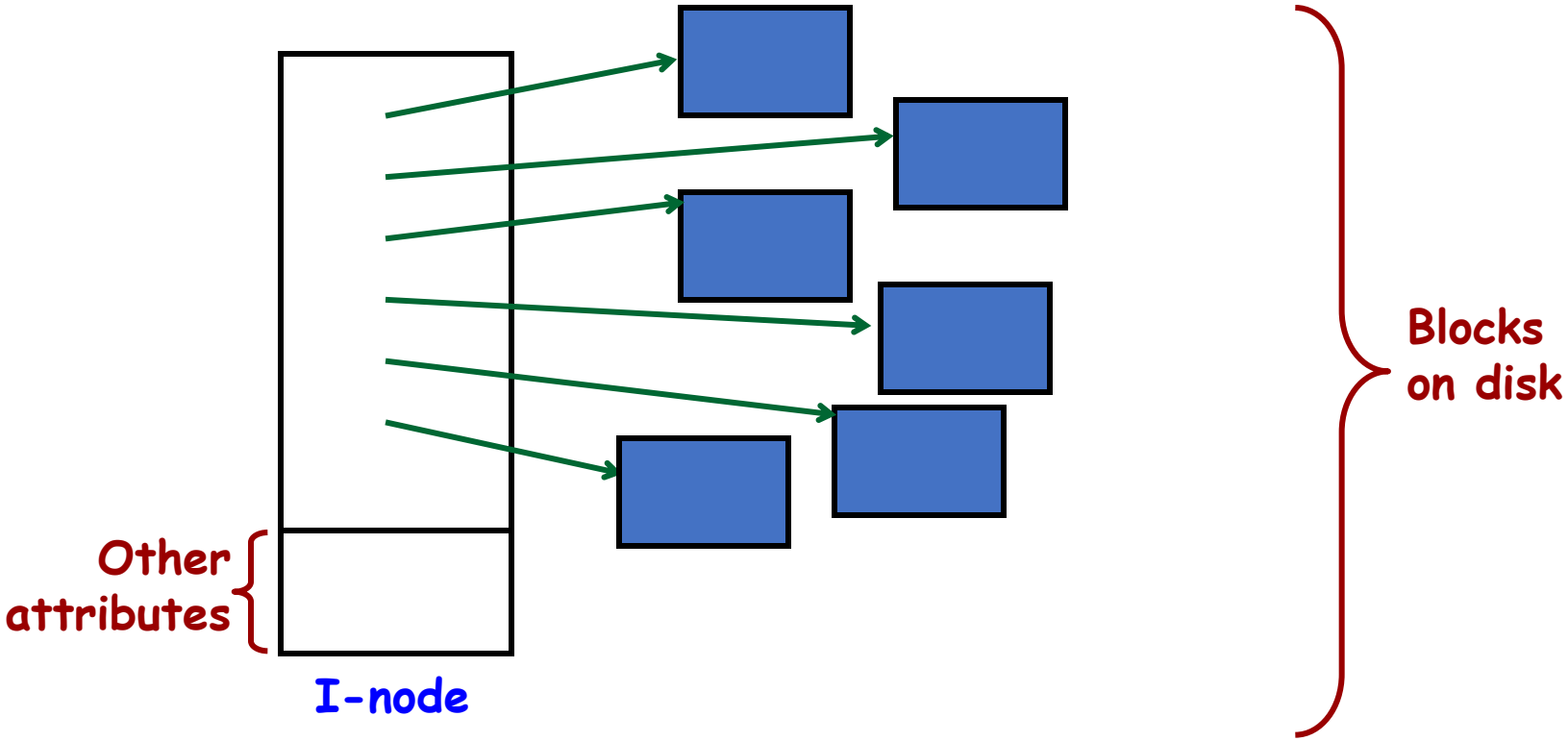
$$\text{FAT size} = 200 \times 1024 \times 1024 \times 4\,\text{bytes} \approx 800\,\text{MB}$$

Thus, **800 MB of memory** is required just to store the FAT table!

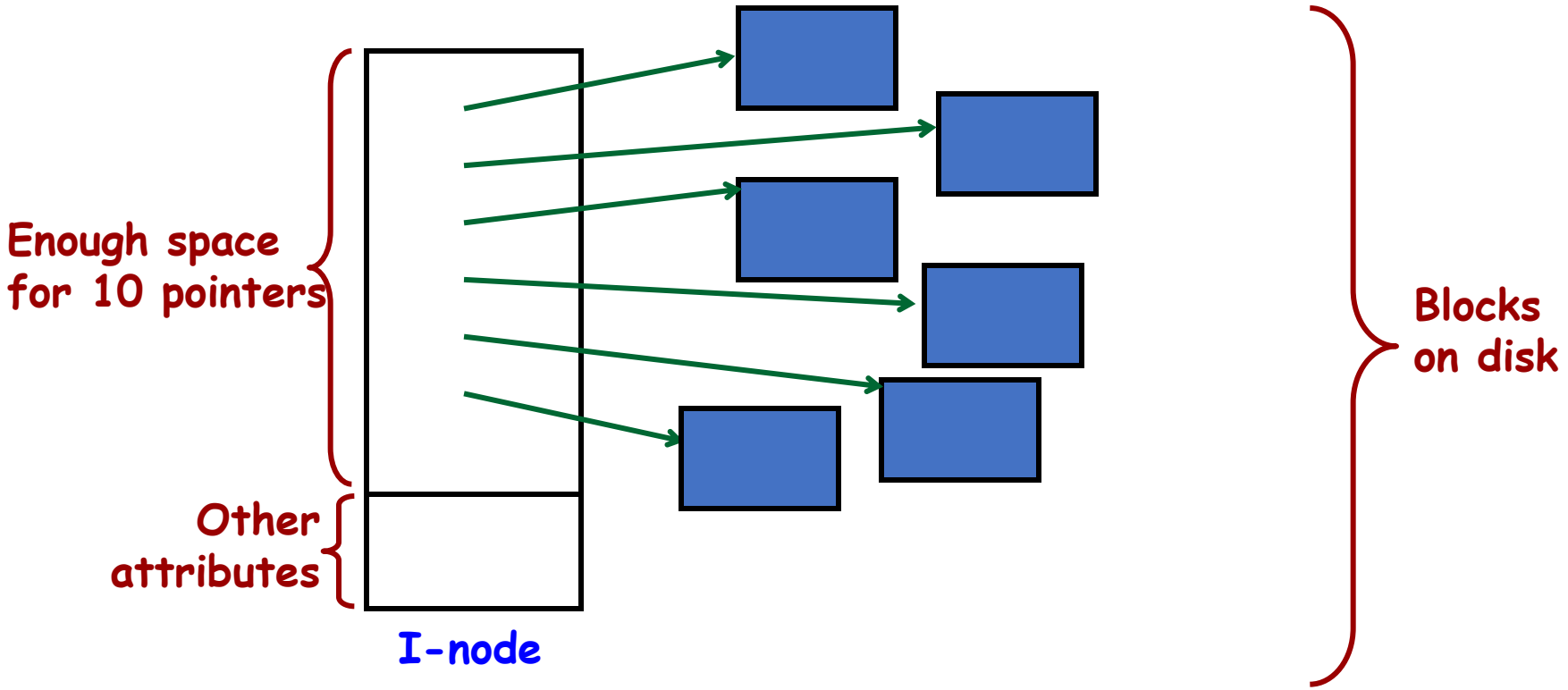
I-NODES

I-nodes

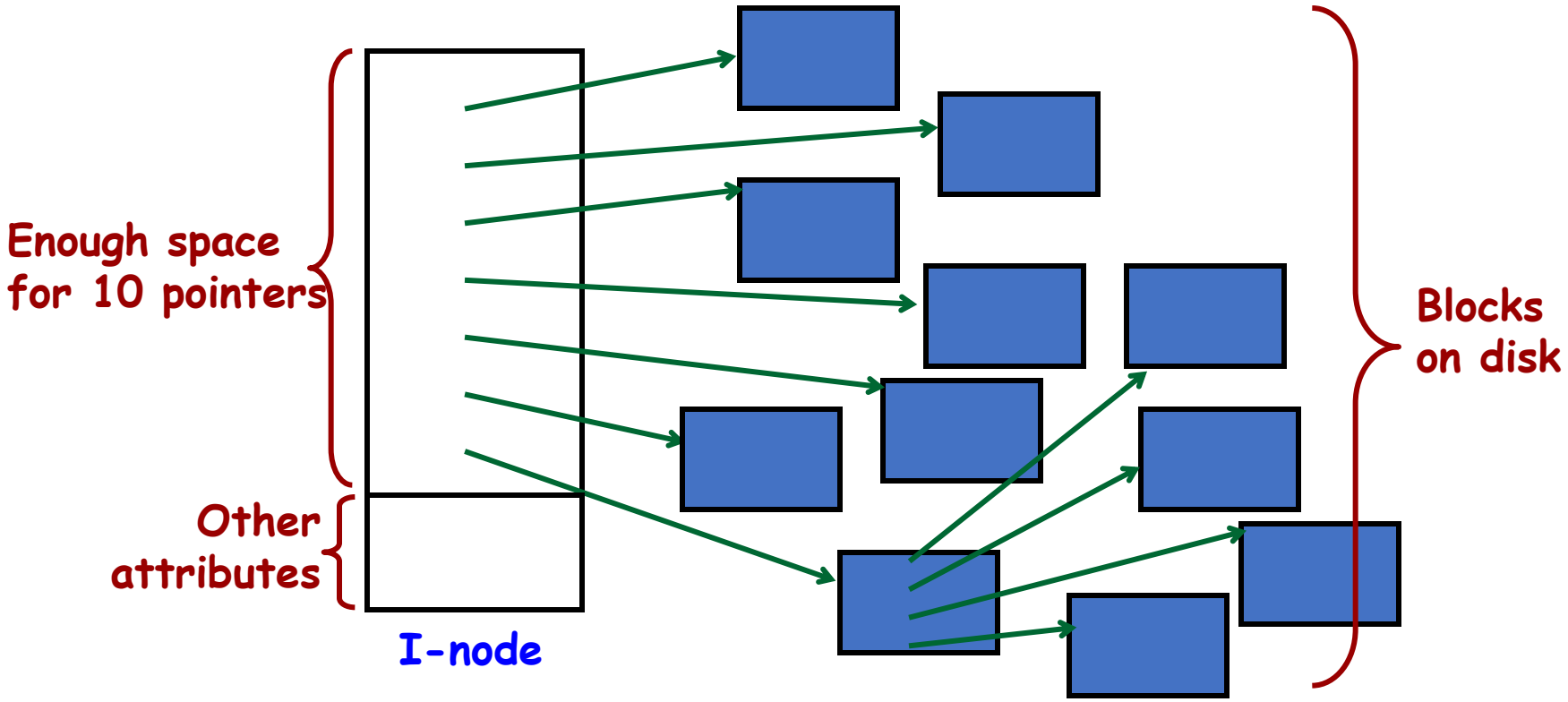
- Each I-node ("index-node") is a structure containing info about the file
  - *Attributes and location of the blocks containing the file*



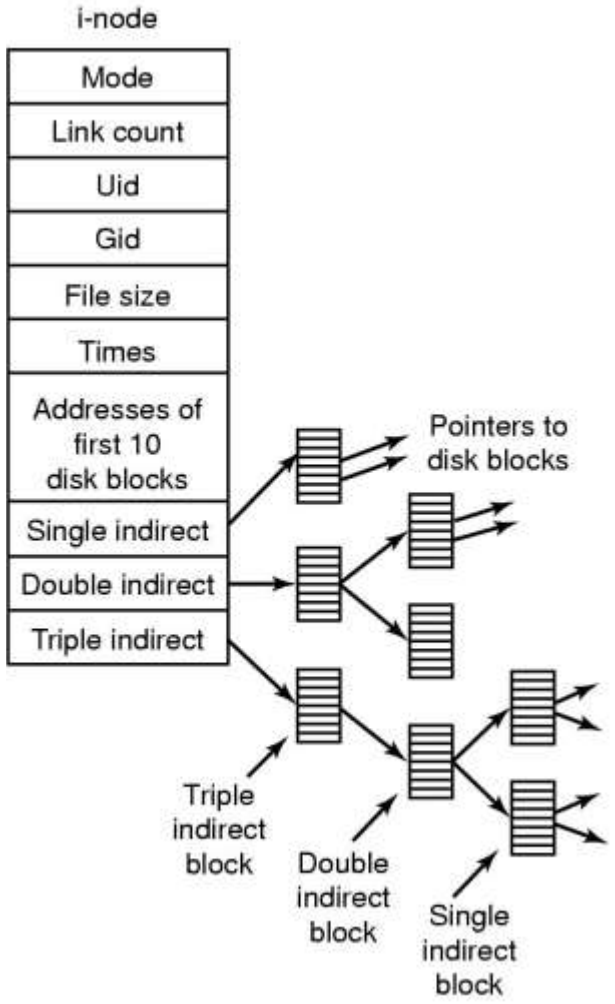
I-nodes


Enough space
for 10 pointers
Other
attributes
I-node
Blocks
on disk

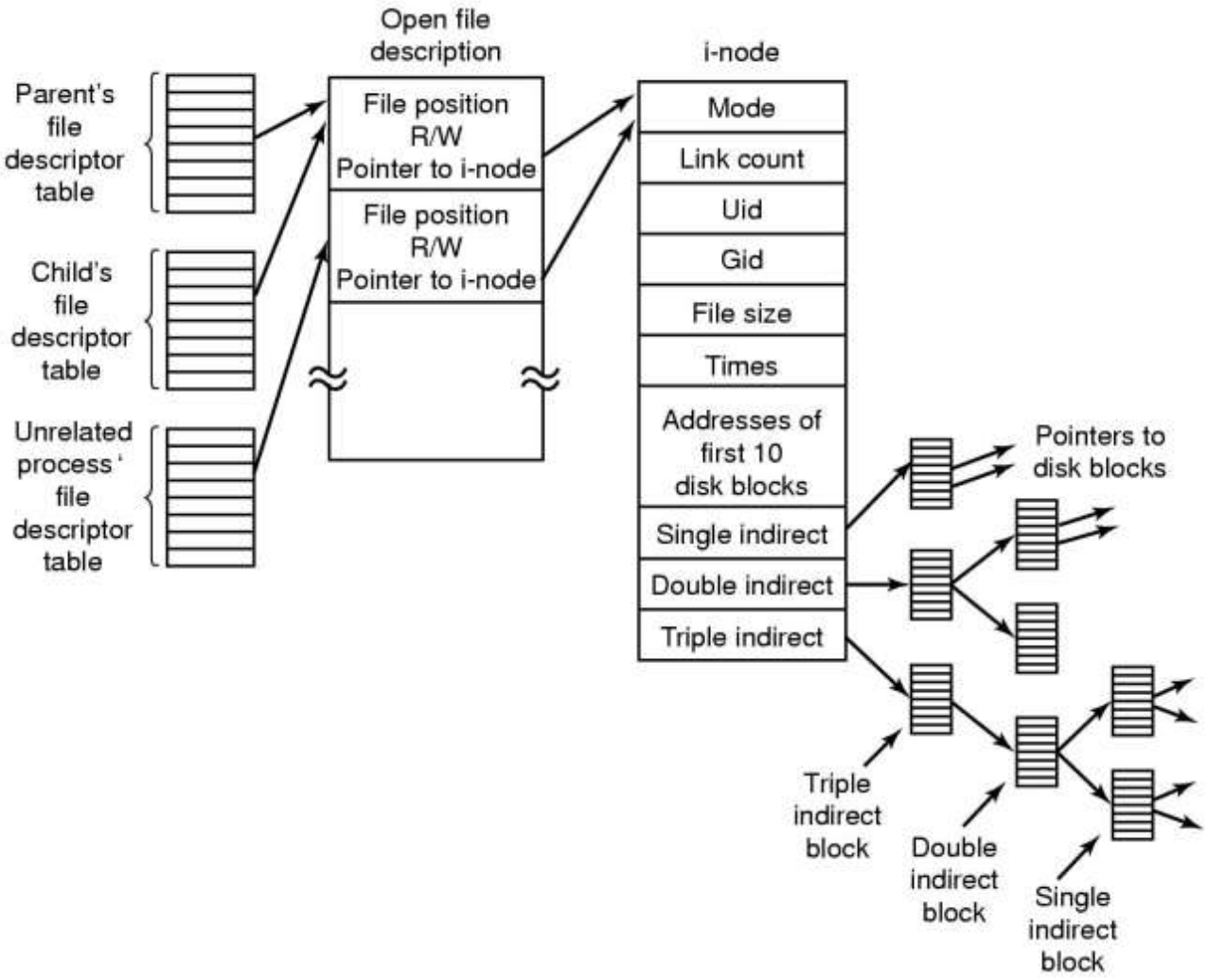
I-nodes


Enough space
for 10 pointers
Other
attributes
I-node
Blocks
on disk

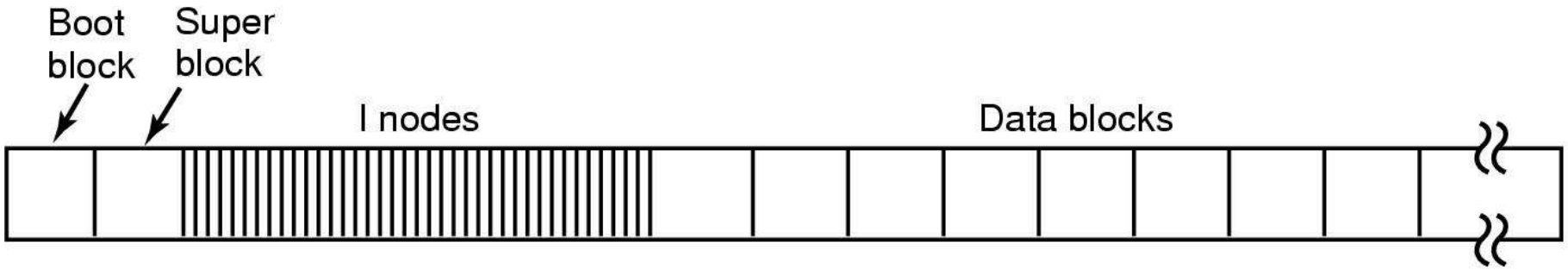
The UNIX I-node



The UNIX File System



The UNIX File System

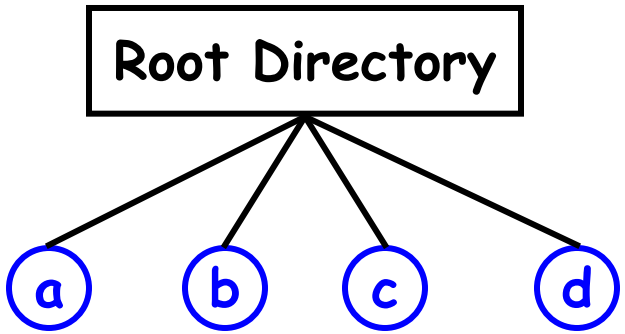


Naming Files

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

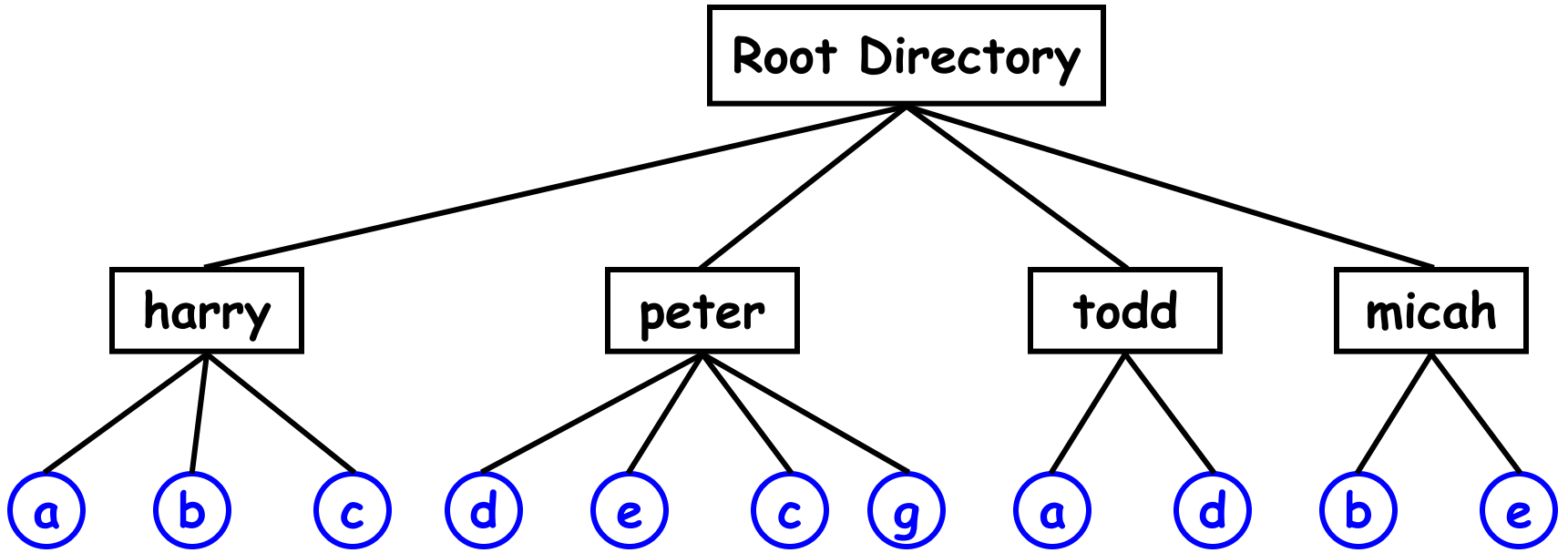
Single Level Directories

- Sometimes called folders
- Early OSs had single-Level Directory Systems
  - *Sharing amongst users was a problem*
- Appropriate for small, embedded systems



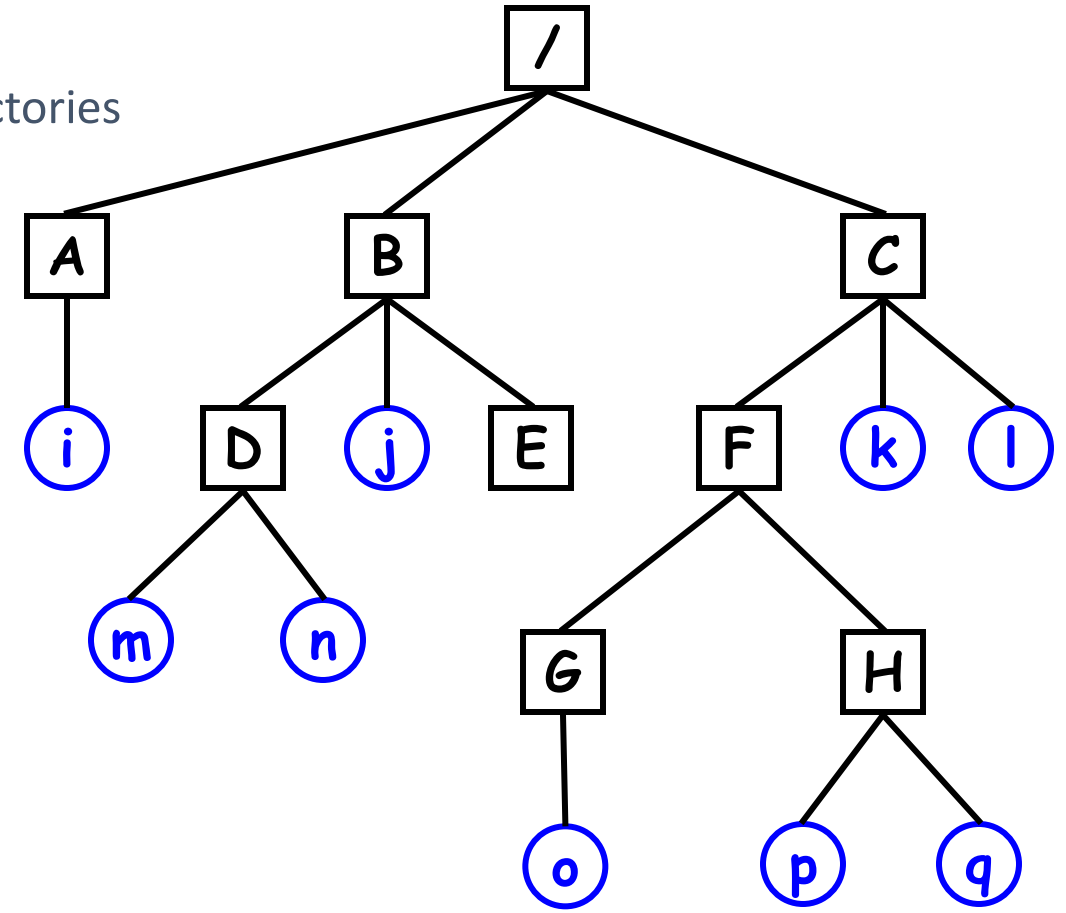
Two-Level Directories

- Each user has a directory
  - */peter/g*



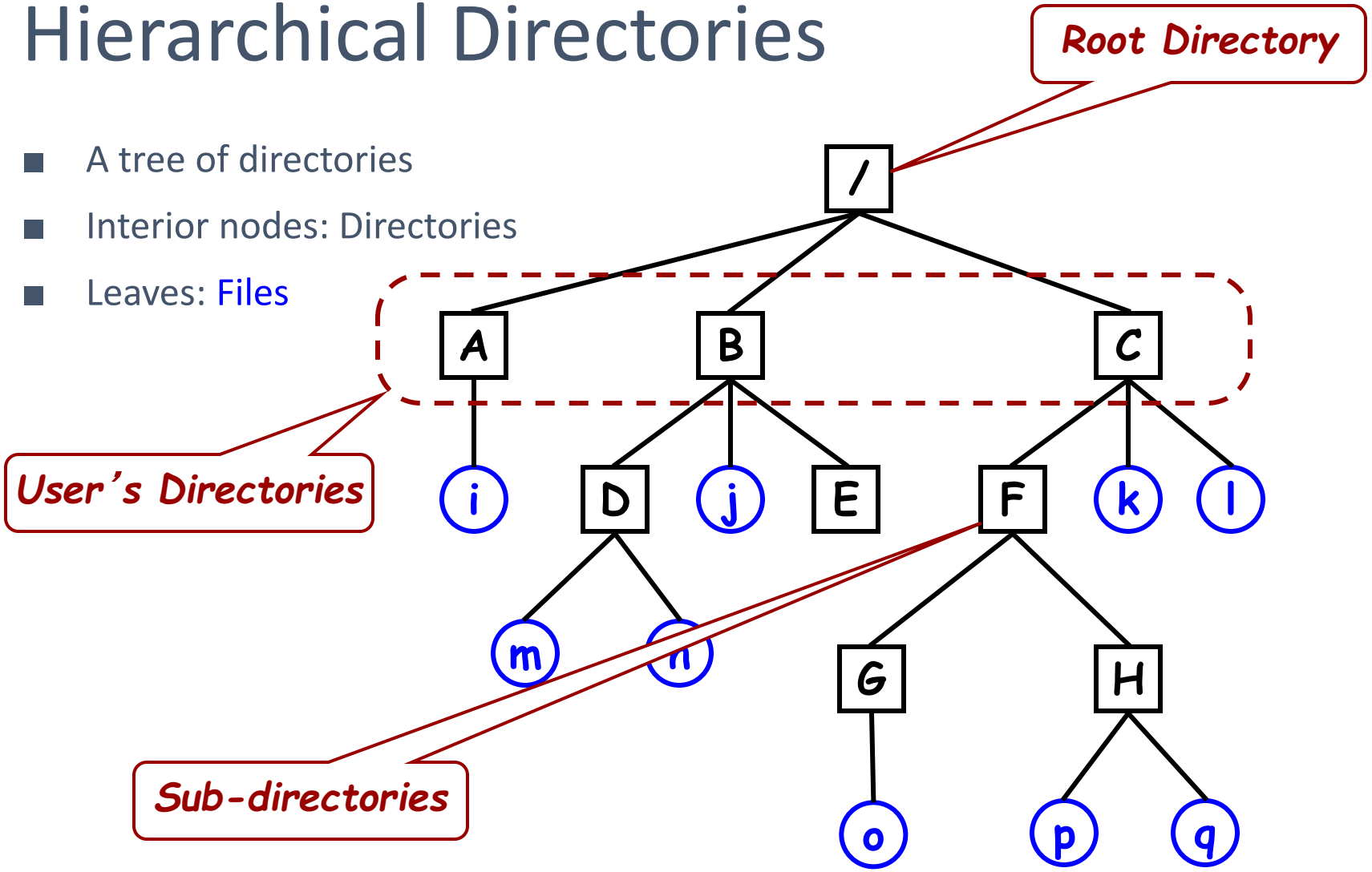
Hierarchical Directories

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



Hierarchical Directories

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



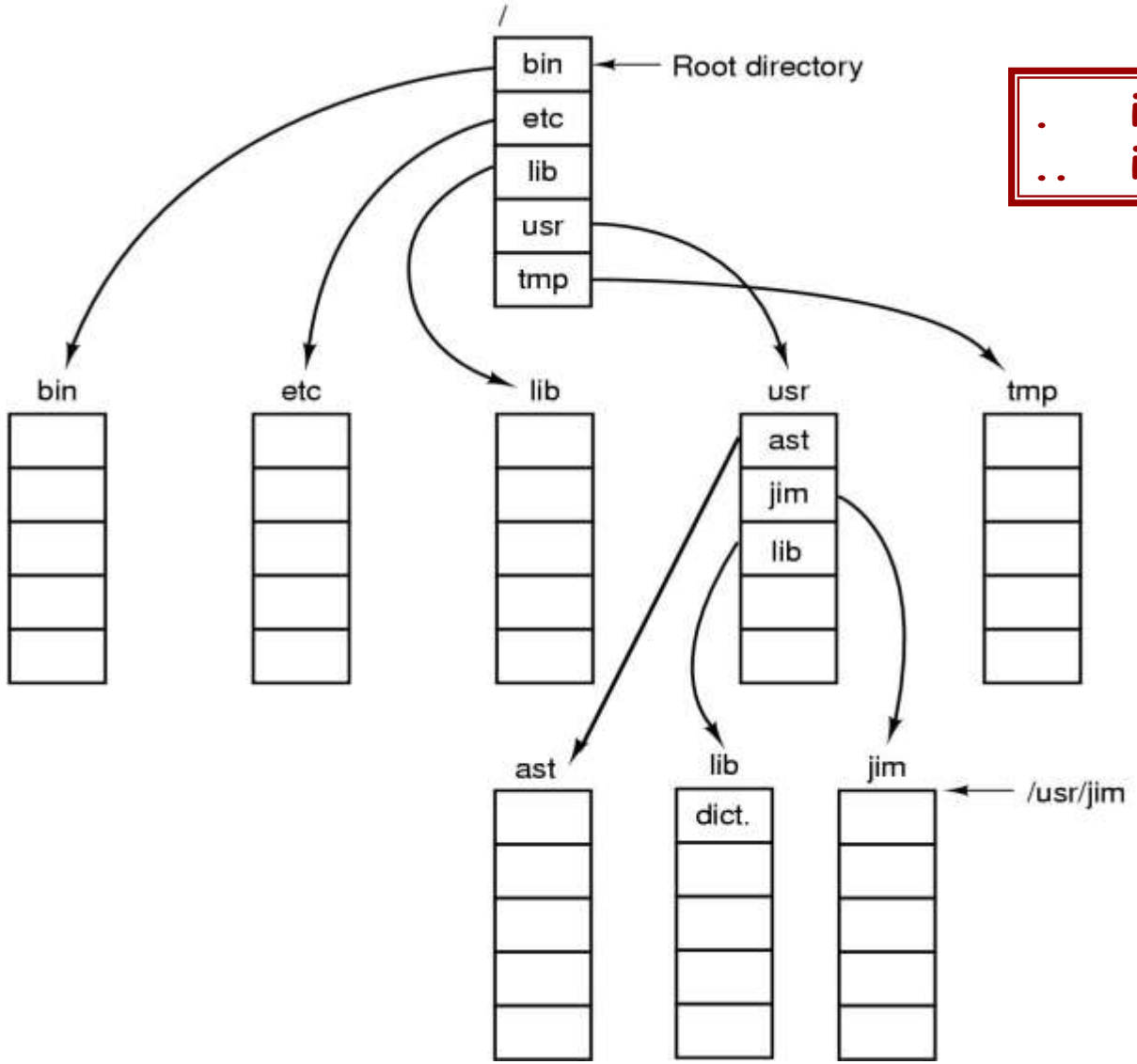
Path Names

- Windows
  \usr\jon\mailbox
- Unix
  /usr/jon/mailbox

- Absolute Path Name
  /usr/jon/mailbox

- Relative Path Name
  - *"working directory" (or "current directory")*
  - *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
- Rename ... a directory
- Link ... add this directory as a sub directory in another
- Unlink ... remove a “hard link”

Implementing Directories

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

Implementing Directories

- Simple approach

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

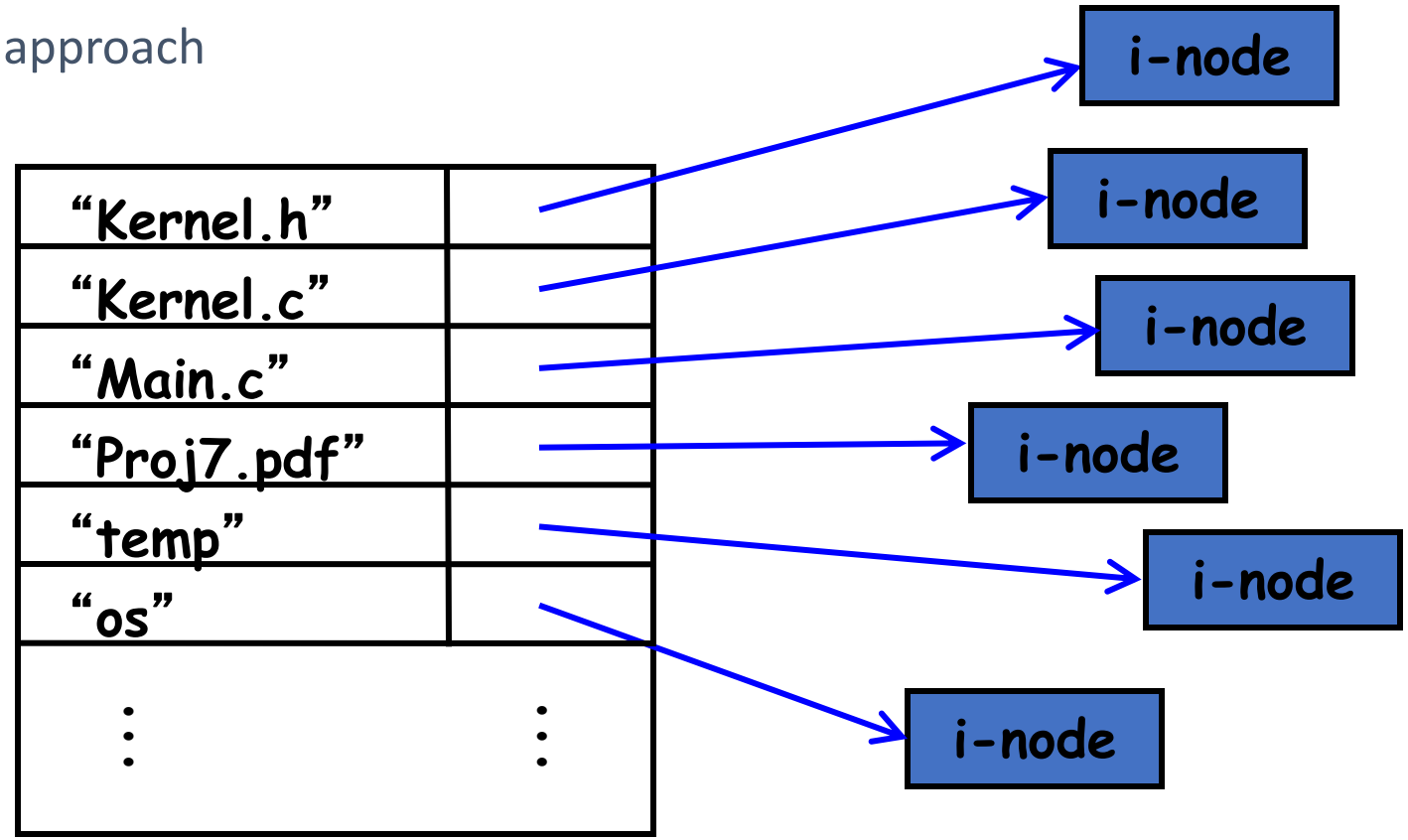
Implementing Directories

- List of files
  - *File name*
  - *File Attributes*

- Unix Approach:
  - *Directory contains*
    - File name
    - I-Node number
  - *I-Node contains*
    - File Attributes

Implementing Directories

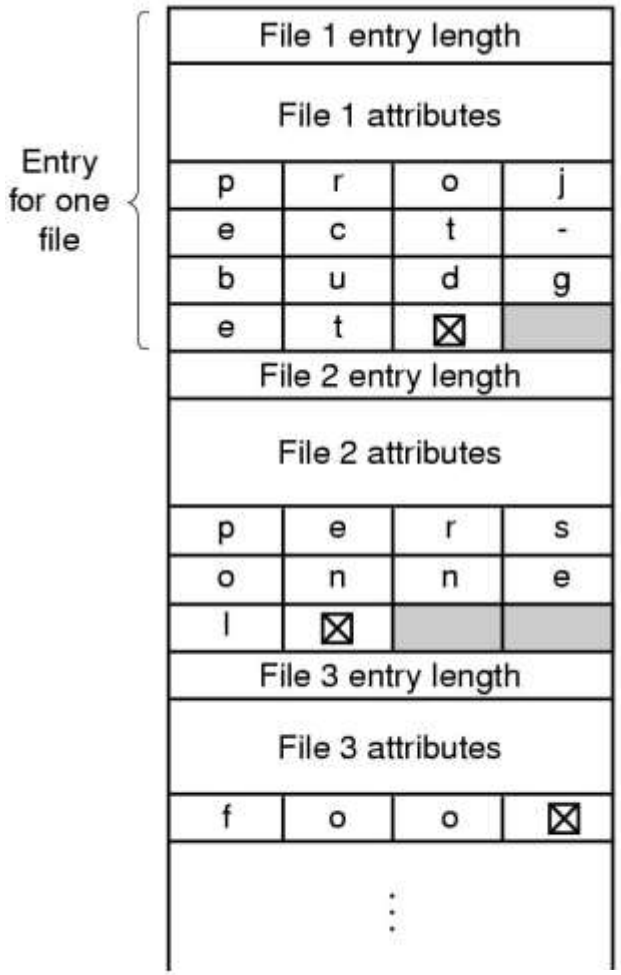
- Unix approach



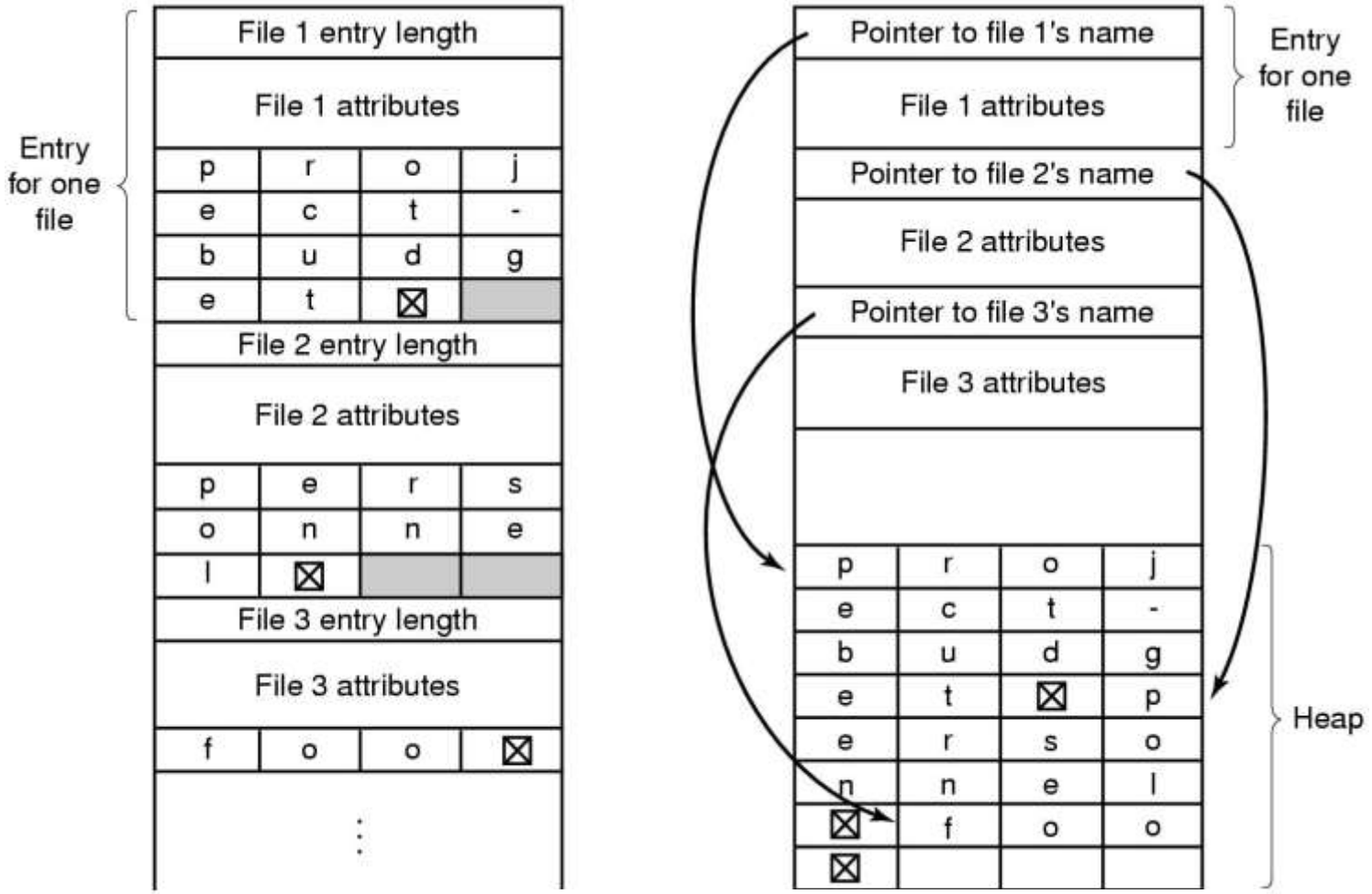
Implementing Filenames

- 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*
- Variable Length Names
  - *Unix (today)*
    - *Max 255 chars*
    - *Directory structure gets more complex*

Fixed-Length Filenames



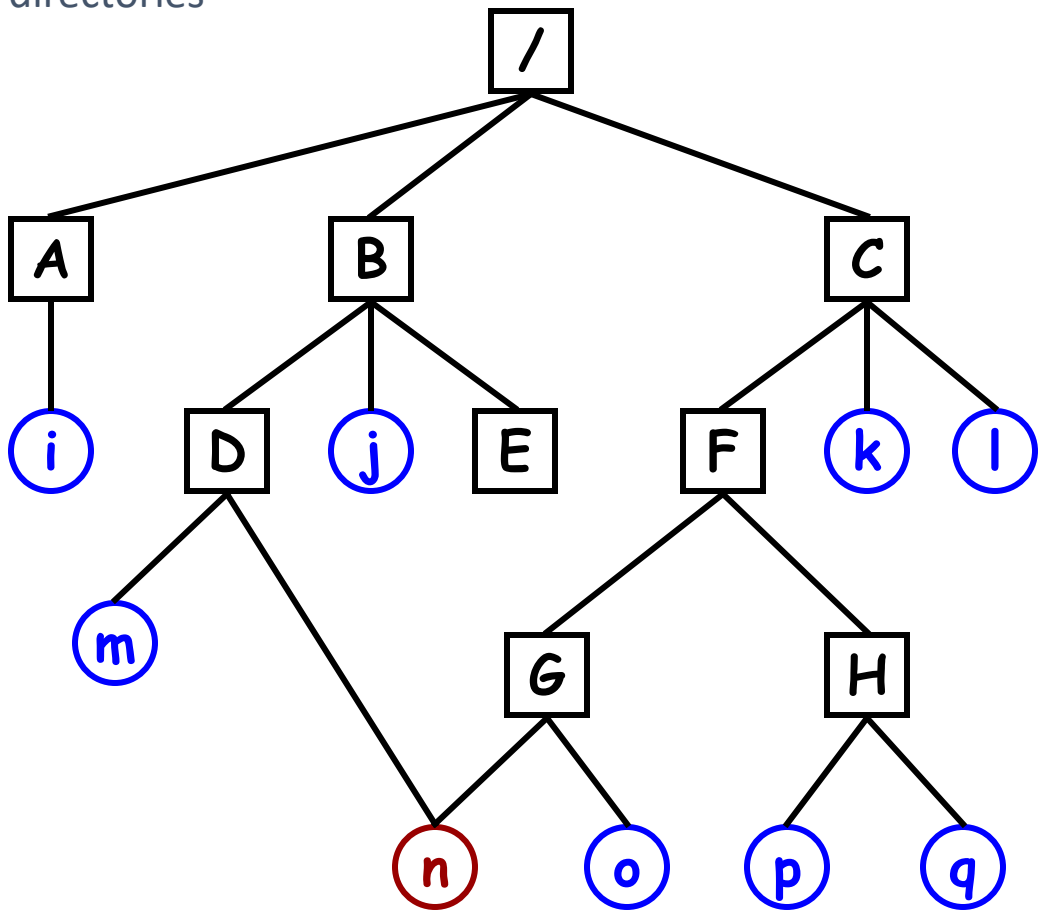
Variable-Length Filenames



SHARING FILES

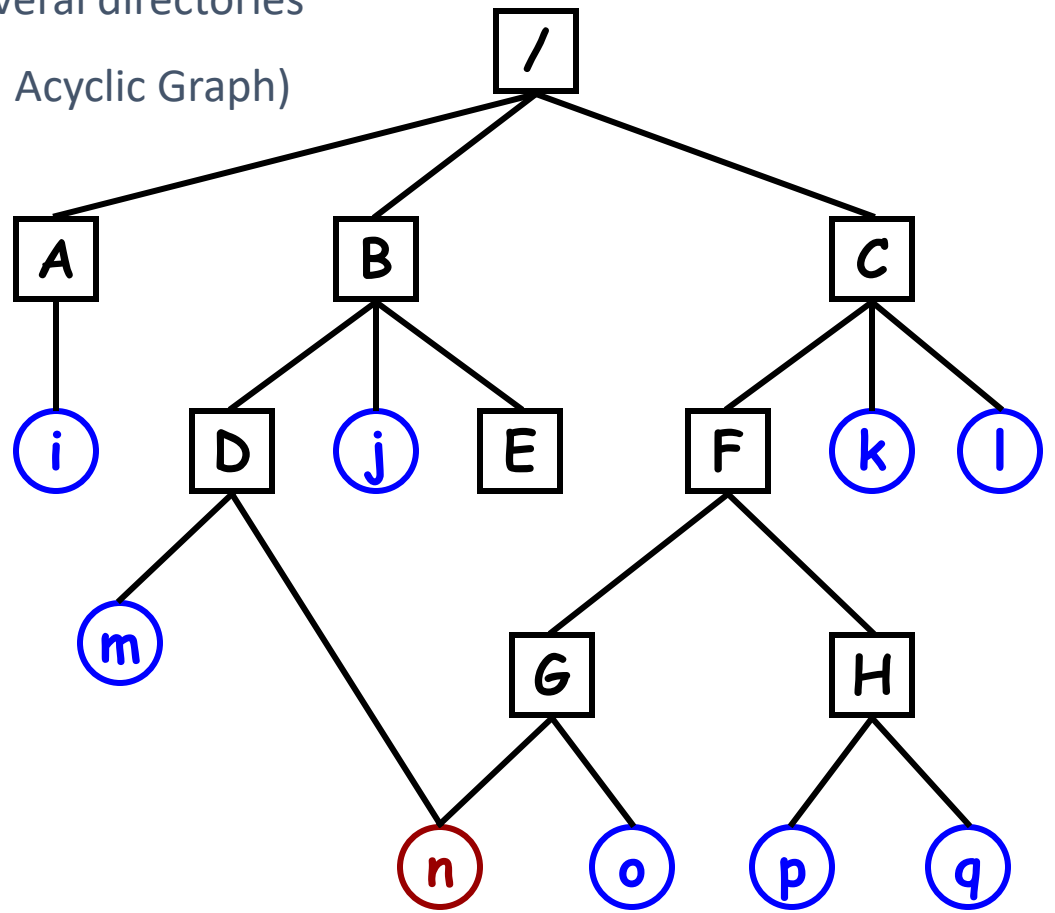
Sharing Files

- One file appears in several directories
- Tree → DAG



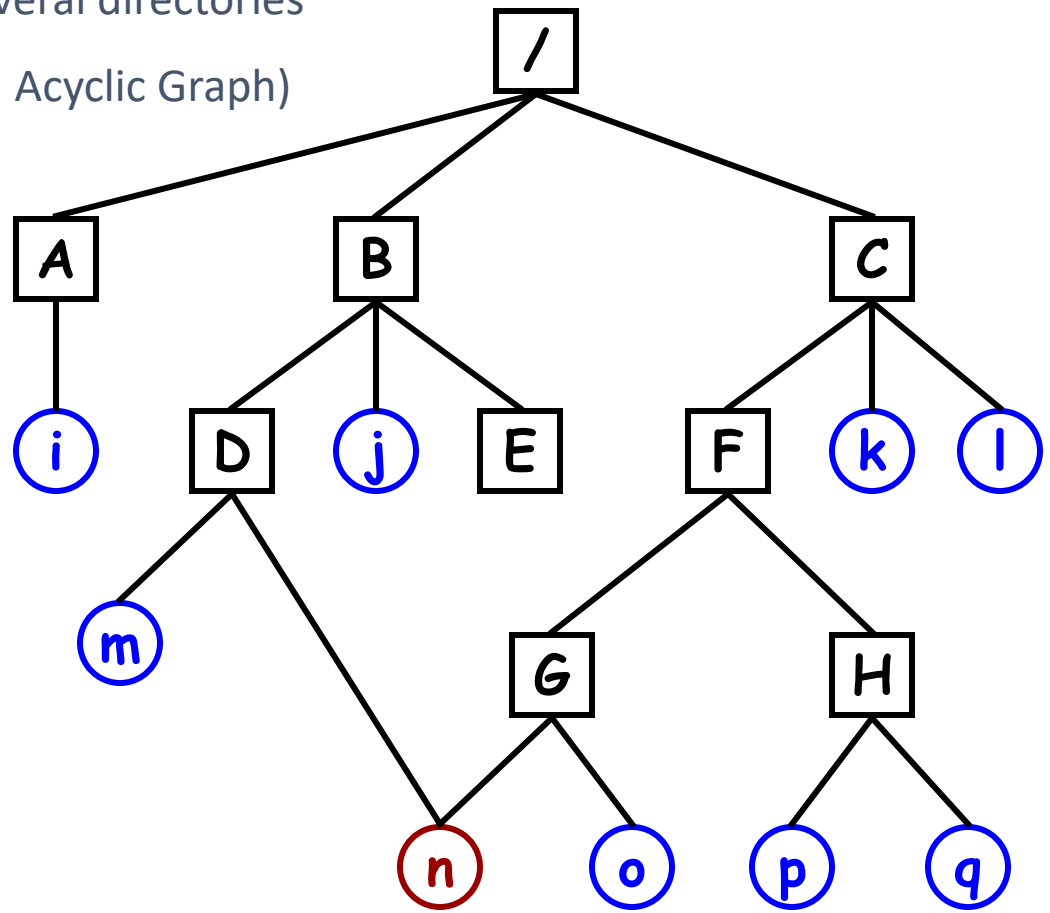
Sharing Files

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



Sharing Files

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

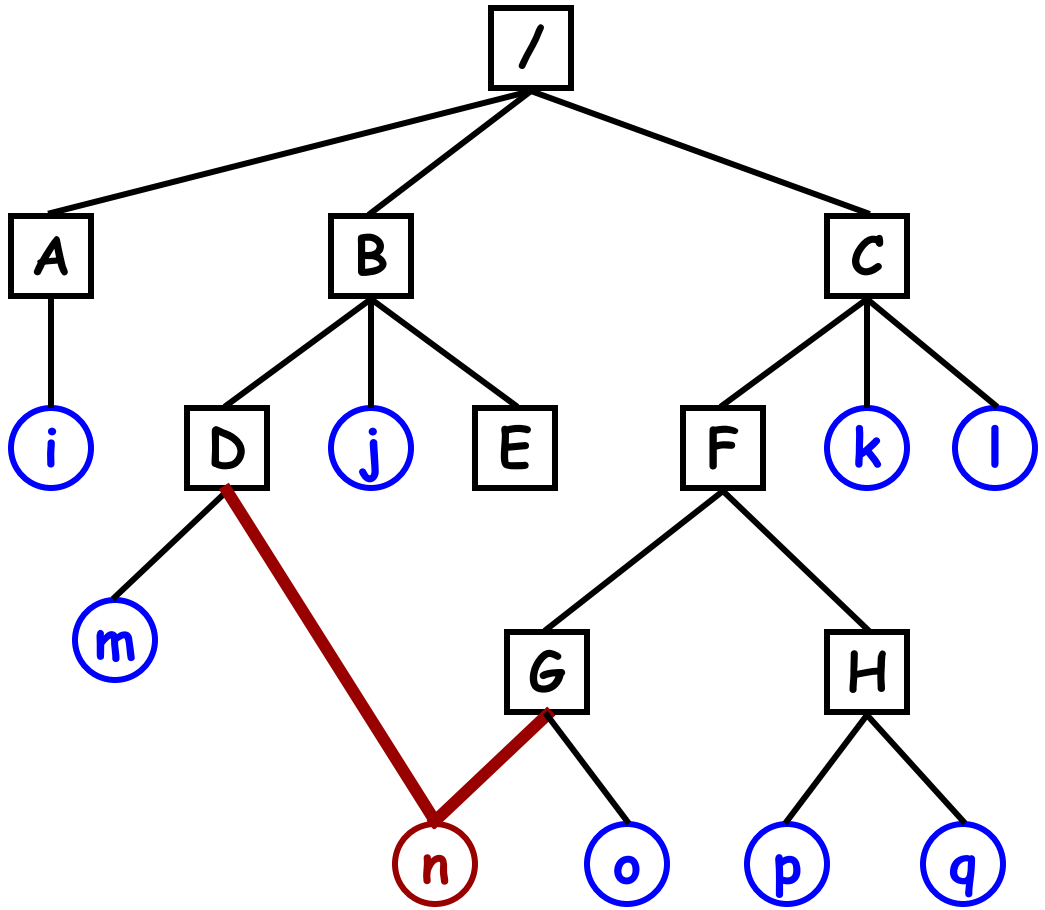


What if the file changes?
New disk blocks are used.
Better not store this info in the directories!!!

Hard Links and Symbolic Links

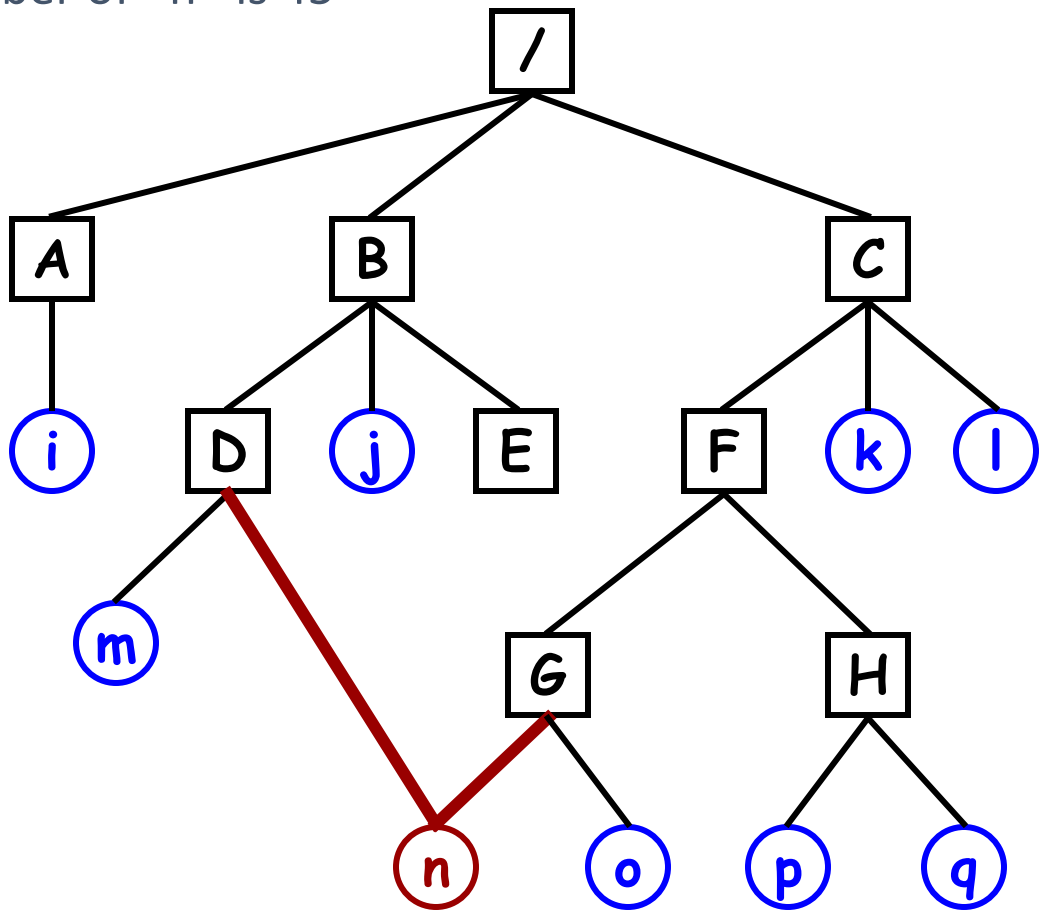
- In Unix:
  - *Hard links*
    - Both directories point to the same i-node
  - *Symbolic links*
    - One directory points to the file's i-node
    - Other directory contains the "path"

Hard Links



Hard Links

- Assume i-node number of “n” is 45



Hard Links

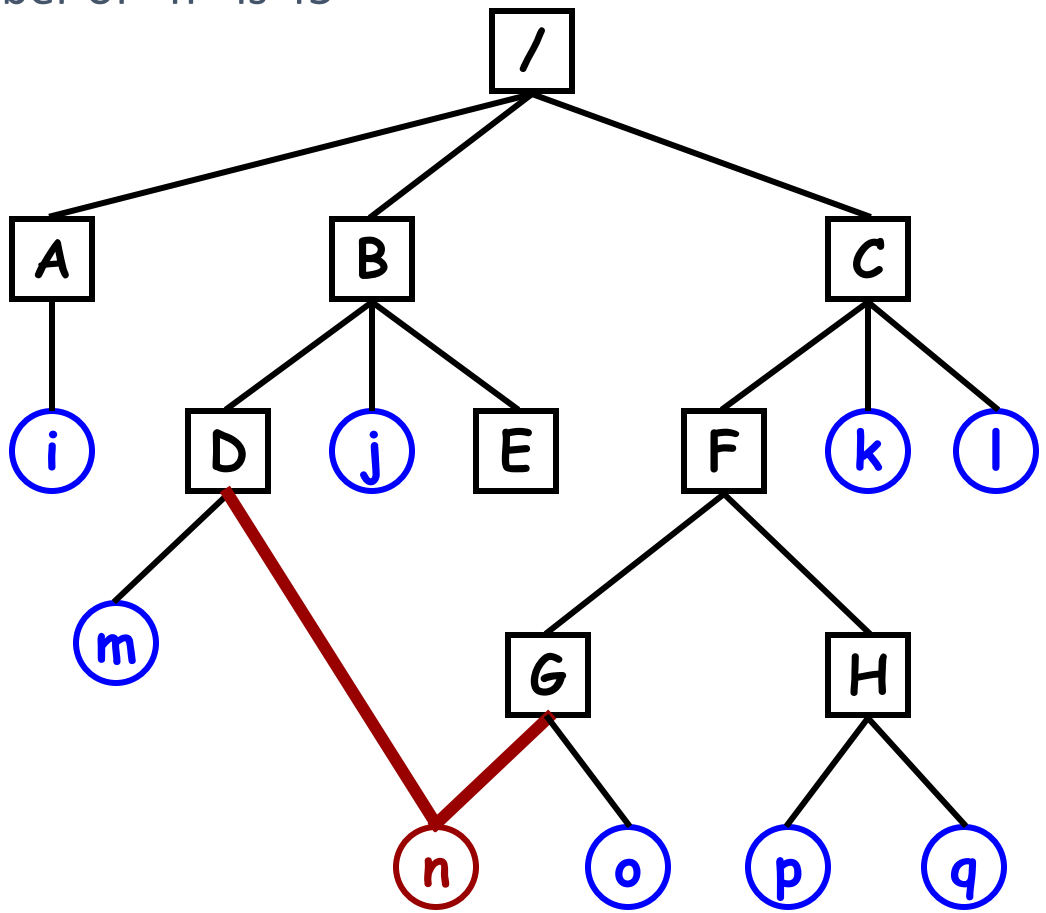
- Assume i-node number of "n" is 45

Directory "D"

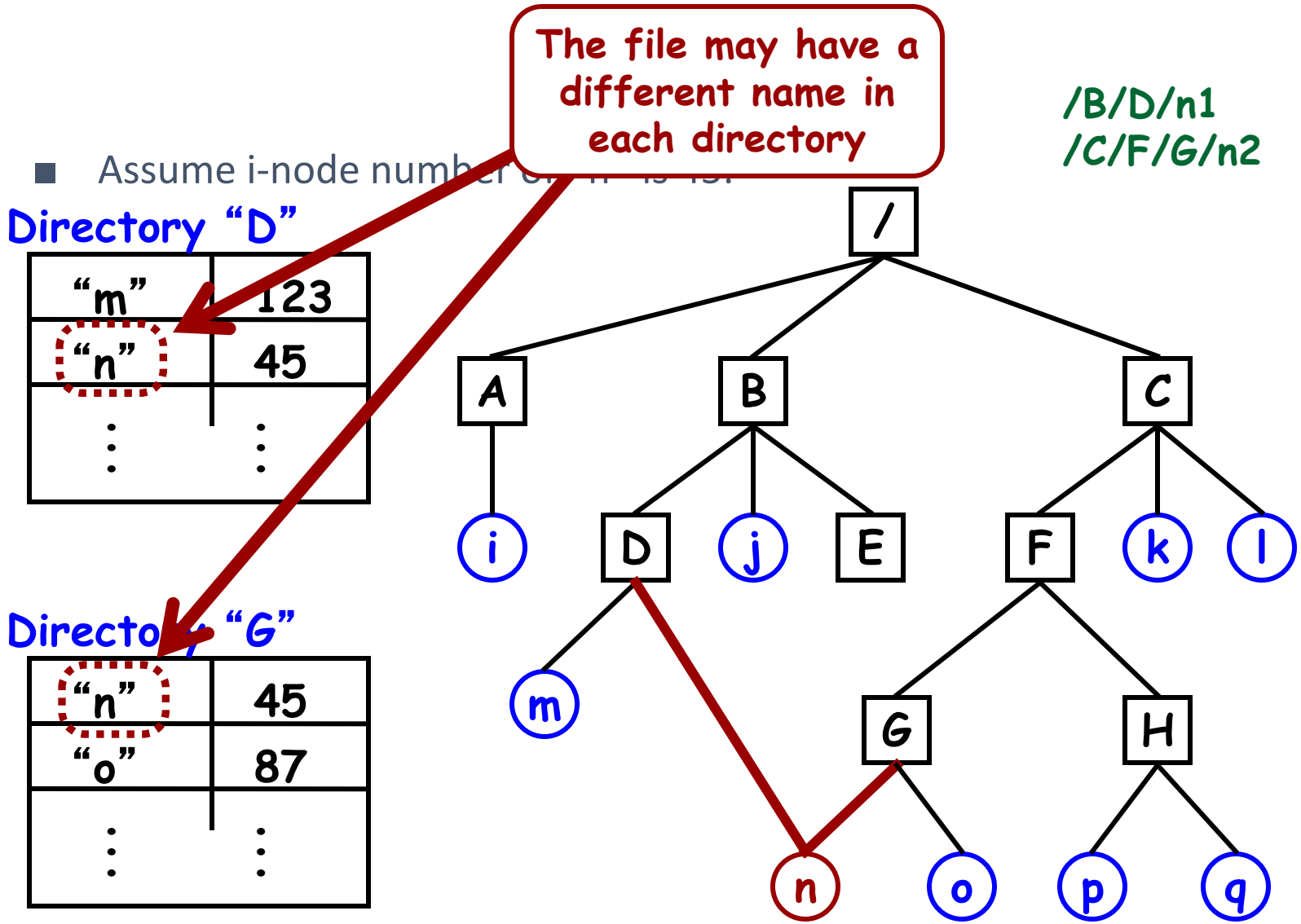
"m"	123
"n"	45
⋮	⋮

Directory "G"

"n"	45
"o"	87
⋮	⋮

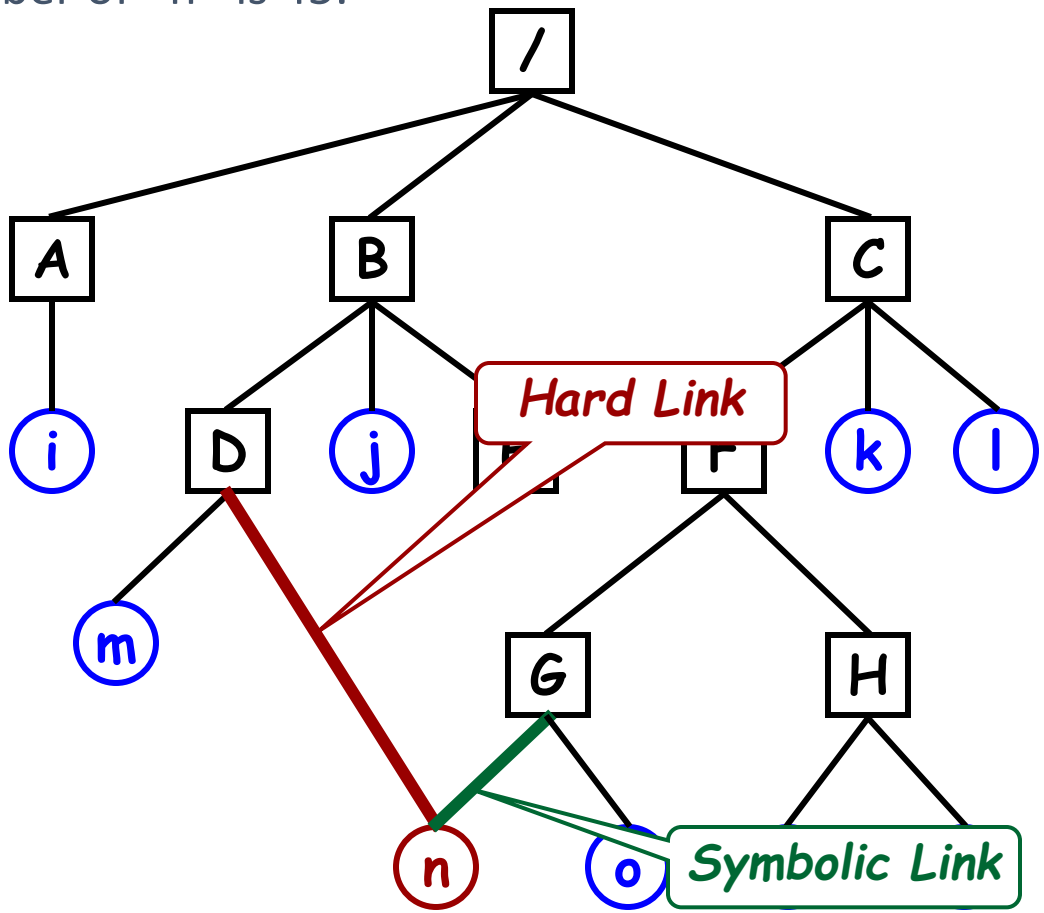


Hard Links


The file may have a different name in each directory
/B/D/n1
/C/F/G/n2
Directory "D"
"m" 123
"n" 45
Directory "G"
"n" 45
"o" 87
/
A
B
C
i
D
j
E
F
k
l
m
G
H
n
o
p
q

Symbolic Links

- Assume i-node number of “n” is 45.

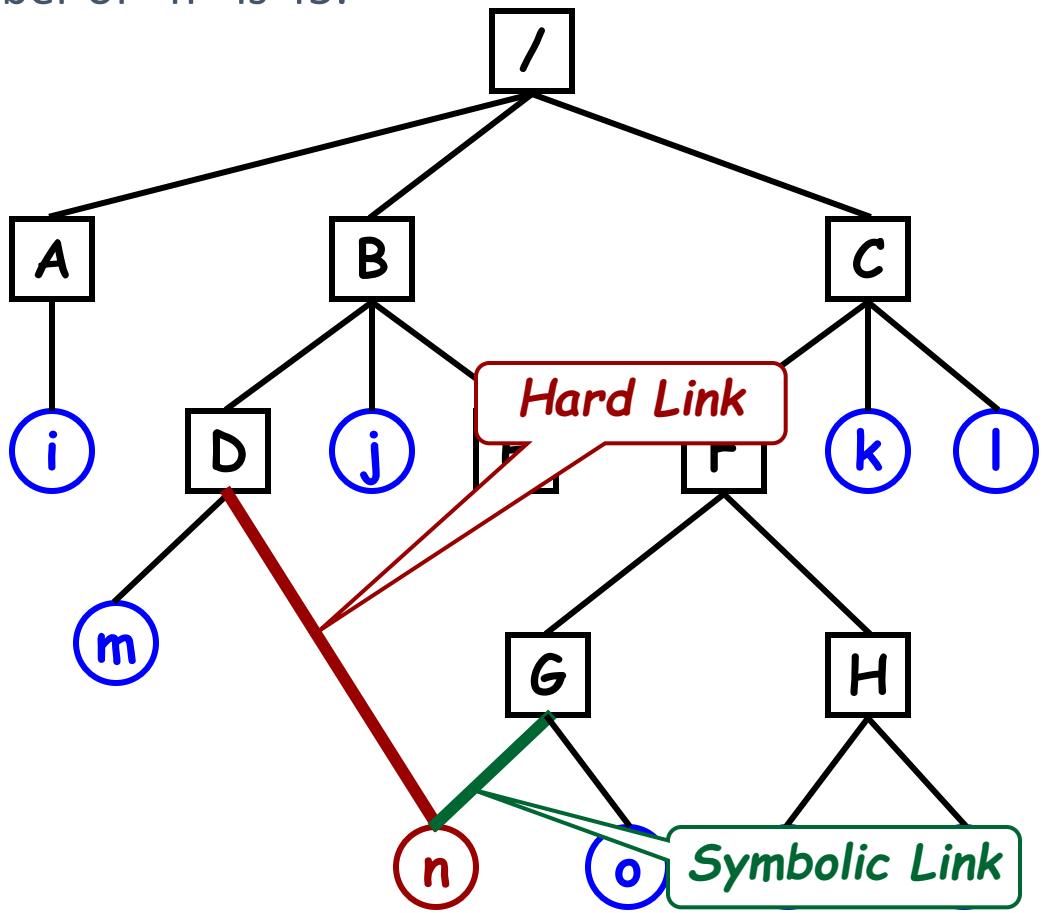


Symbolic Links

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

Directory "D"

"m"	123
"n"	45
⋮	⋮



Symbolic Links

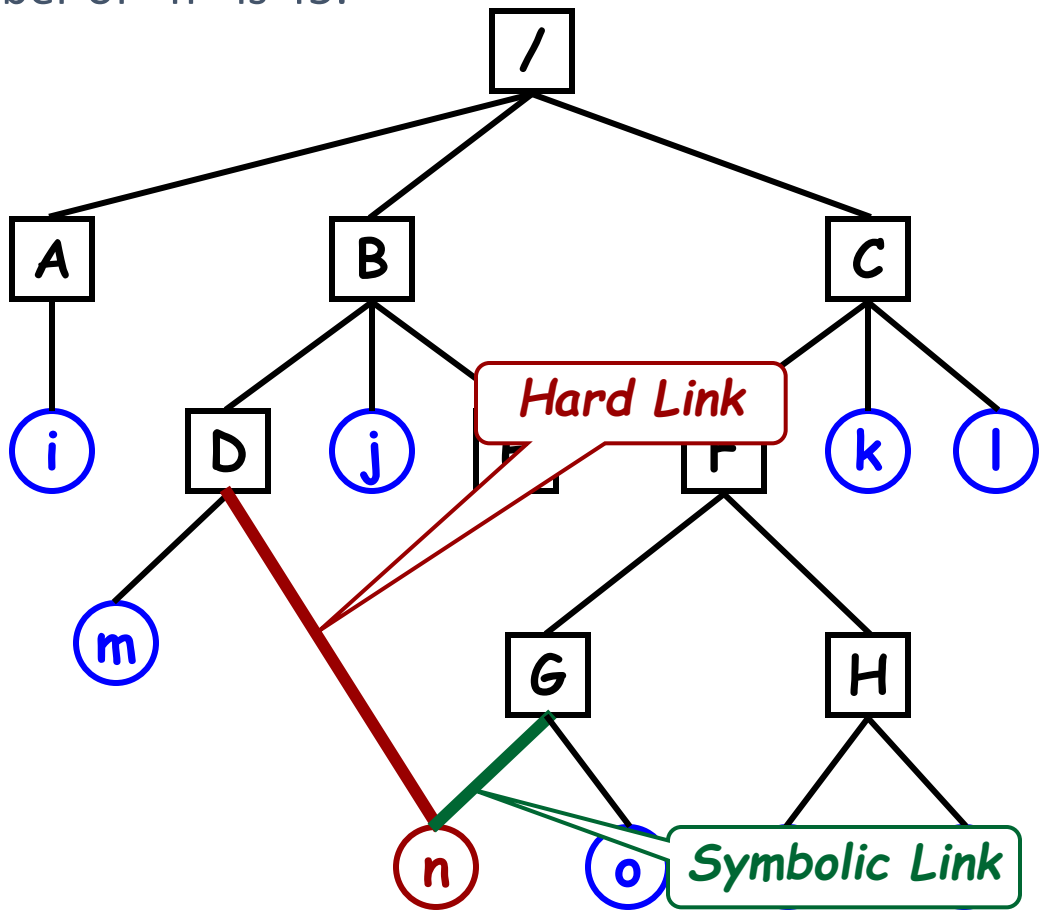
- Assume i-node number of "n" is 45.

Directory "D"

"m"	123
"n"	45
⋮	⋮

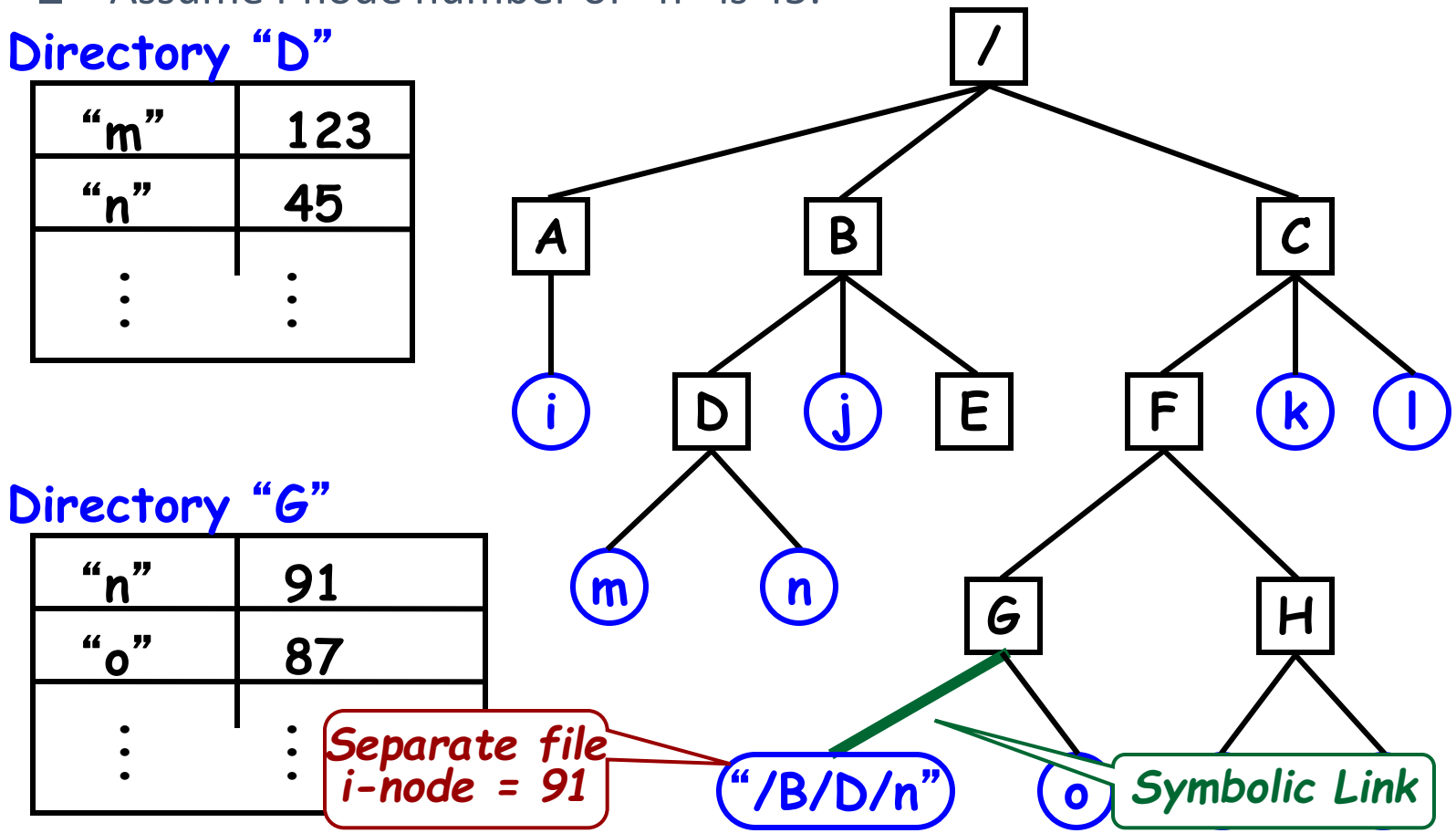
Directory "G"

"n"	/B/D/n
"o"	87
⋮	⋮



Symbolic Links

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



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*