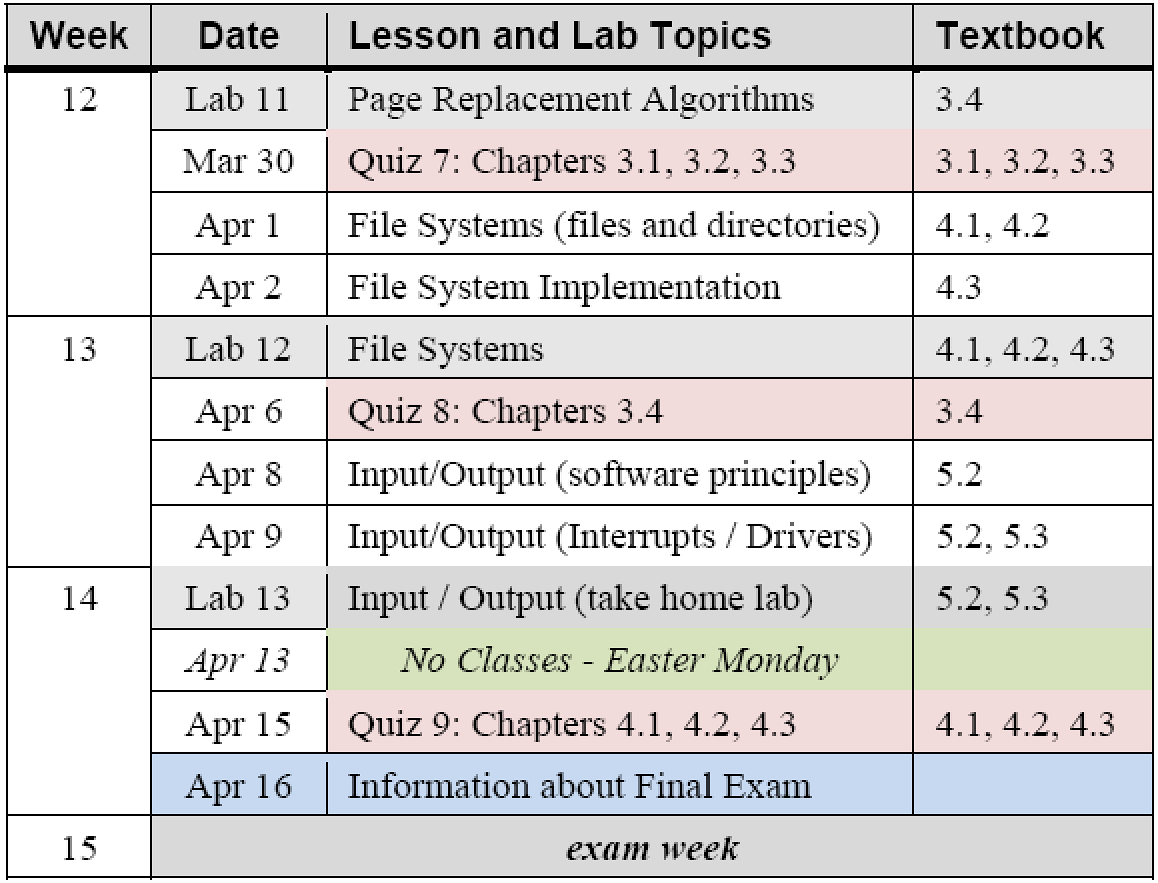
COMP 4735: Operating Systems Concepts

Lesson 13: File Systems

Rob Neilson

rneilson@bcit.ca

Schedule for remaining classes …



Week	Date	Lesson and Lab Topics	Textbook
12	Lab 11	Page Replacement Algorithms	3.4
	Mar 30	Quiz 7: Chapters 3.1, 3.2, 3.3	3.1, 3.2, 3.3
	Apr 1	File Systems (files and directories)	4.1, 4.2
	Apr 2	File System Implementation	4.3
13	Lab 12	File Systems	4.1, 4.2, 4.3
	Apr 6	Quiz 8: Chapters 3.4	3.4
	Apr 8	Input/Output (software principles)	5.2
	Apr 9	Input/Output (Interrupts / Drivers)	5.2, 5.3
14	Lab 13	Input / Output (take home lab)	5.2, 5.3
	Apr 13	*No Classes - Easter Monday*	
	Apr 15	Quiz 9: Chapters 4.1, 4.2, 4.3	4.1, 4.2, 4.3
	Apr 16	Information about Final Exam	
15	***exam week***		

File Systems

Why do we need a file system?

- There is a need to store large amounts of data
  - on disk, or tape, or cd, or other type of storage media
- The file system gives us a way or organizing the files on the storage media
- 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 of typed ***records***

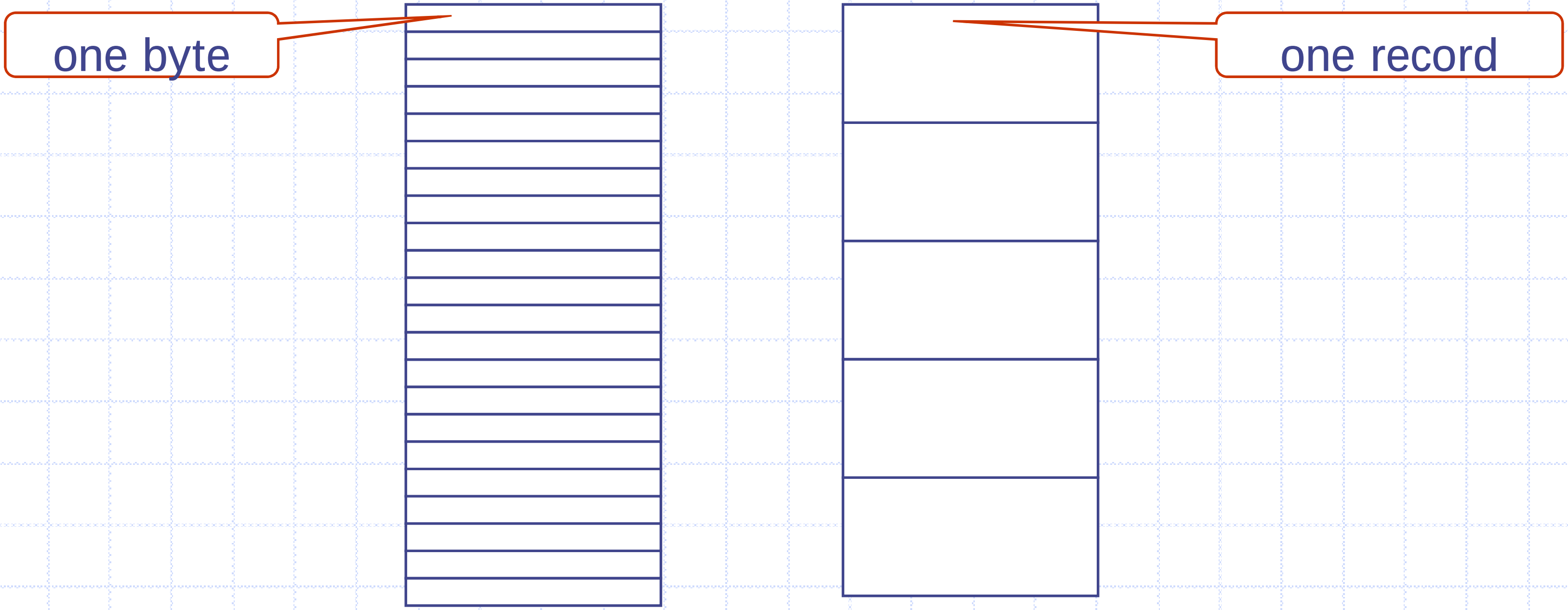
- In Unix-based operating systems a file is an unstructured sequence of bytes

- Typically, the OS organizes files into groups, called directories or folders

- The part of the OS that manages all the files and directories, and which implements the design and organization of them is known as the file system

File Structures

- The file structure is the internal organization of data within the file
- There are two basic types of file structure …



Sequence of bytes

Sequence of records

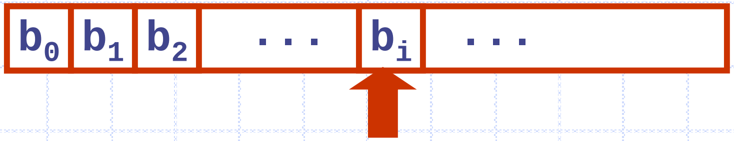
Fill in the blanks …

- Some systems implement files as “series of ________”
  - programmer can define whatever structure they want
  - file can have a structure overlaid on it
  - often thought of as a continuous “stream”
  - this is low level

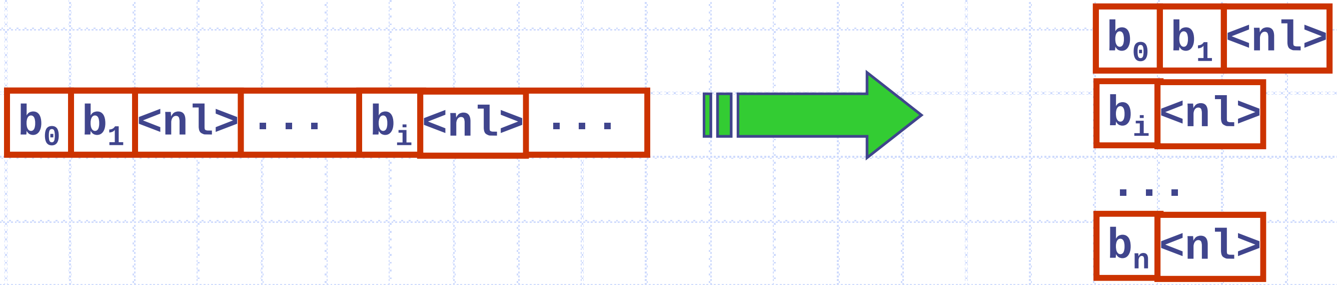
- Some systems implement files as “series of ________”
  - data is organized fields of data, much like a table in a DB
  - programmer defines the format of the data in advance, and the file stores the data in that format
  - suitable when you know the structure of the data in a file
  - this is a higher level approach

Series of Bytes: “byte-stream” file structures

- this is what unix (and most modern OSs) use
- files are essentially continuous streams of bytes, with no pre-determined internal structures or format

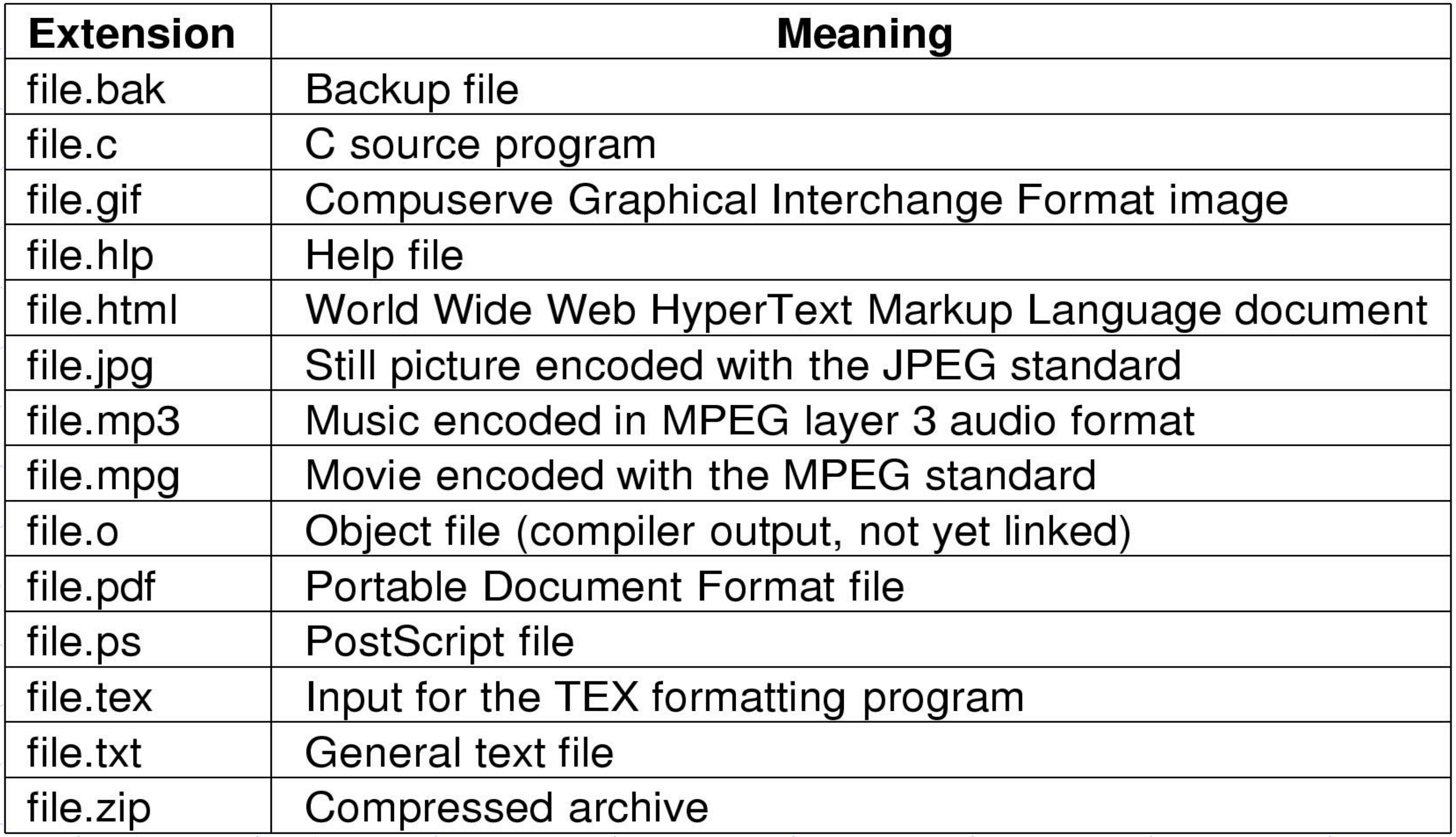


- the application that uses the file needs to have knowledge of the internal file structure
  - for example, a text editor would associate semantic meaning with the newline character ...



Byte stream file structures (cont)

- Some OSs imply semantic meaning to unstructured data as a part of the file naming strategy (file extensions), DOS/Windows for example:



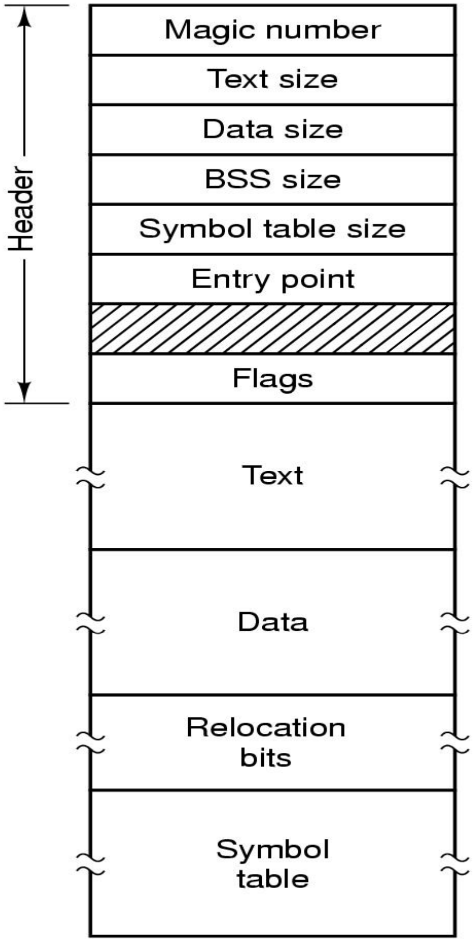
Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
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

File extensions

- Consider a file.doc file
- by associating the “.doc” extension with a file, we are essentially saying
  - “the contents of this file are structured in MS Word format”
  - the OS has mechanisms for you to register the programs that understand MS Word format, and therefore can read the file
- Unix also “allows” file extensions, but they are not built into the file naming strategy
  - you can add an extension if you like (for example “.c”) – but you do not have to add the extension

Which approach is better?

Example of files with internal data organization



Header

Magic number

Text size

Data size

BSS size

Symbol table size

Entry point

Flags

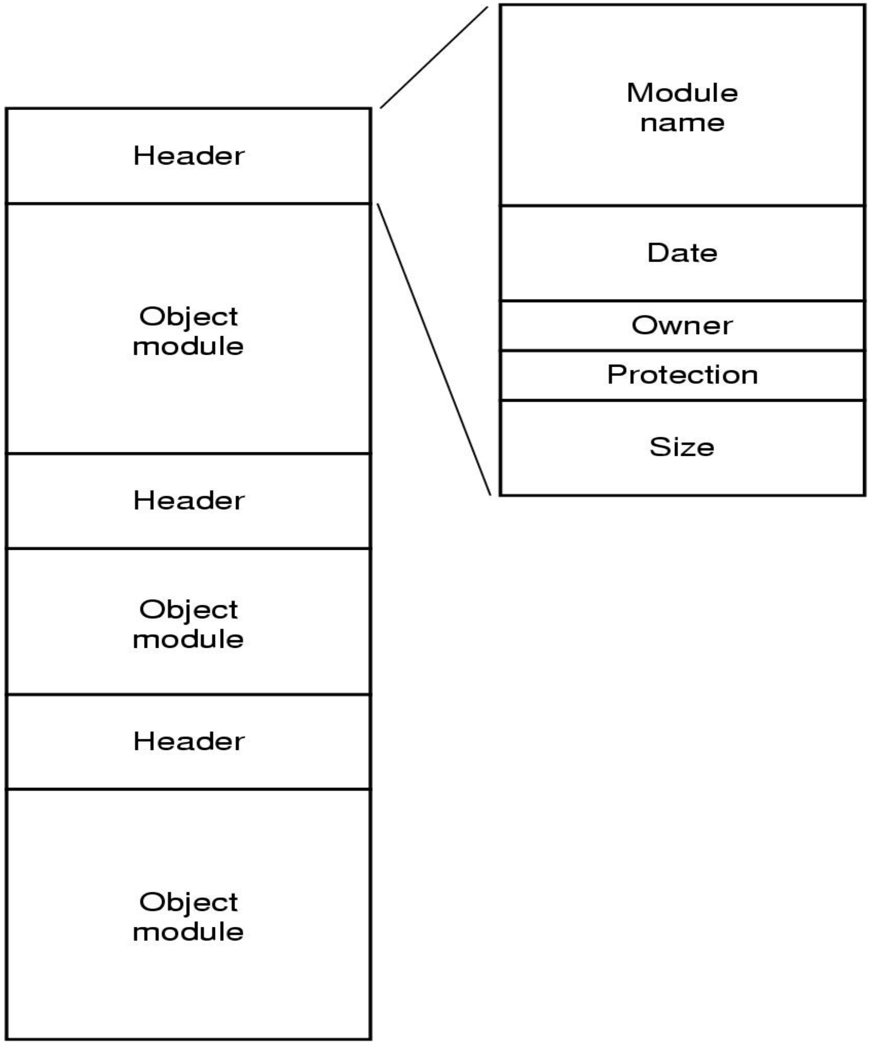
Text

Data

Relocation bits

Symbol table

An executable file



Header

Object module

Header

Object module

Header

Object module

Module name

Date

Owner

Protection

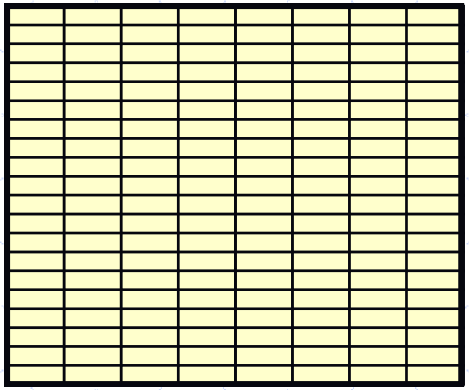
Size

An archive

Series of “records”

- a record is an ordered collection of fields that is known to an application
  - for example, a record might be have the following format:

```
01 StudentDetails.
   02 StudentId        PIC 9(7).
   02 StudentName.
      03 FirstName     PIC X(10).
      03 Initial       PIC X.
      03 SurName       PIC X(15).
   02 DateOfBirth.
      03 BirthDay      PIC 99.
      03 BirthMon      PIC 99.
      03 BirthYear     PIC 9(4).
   02 CourseCode       PIC X(4).
```



and the file is organized as a series of records

  - so the record is stored as …

StudentId	Firstname	Initial	SurName	BirthDay	BirthMon	BirthYear	CourseCode
7 bytes	10 bytes	1 byte	15 bytes	2 bytes	2 bytes	4 bytes	4 bytes

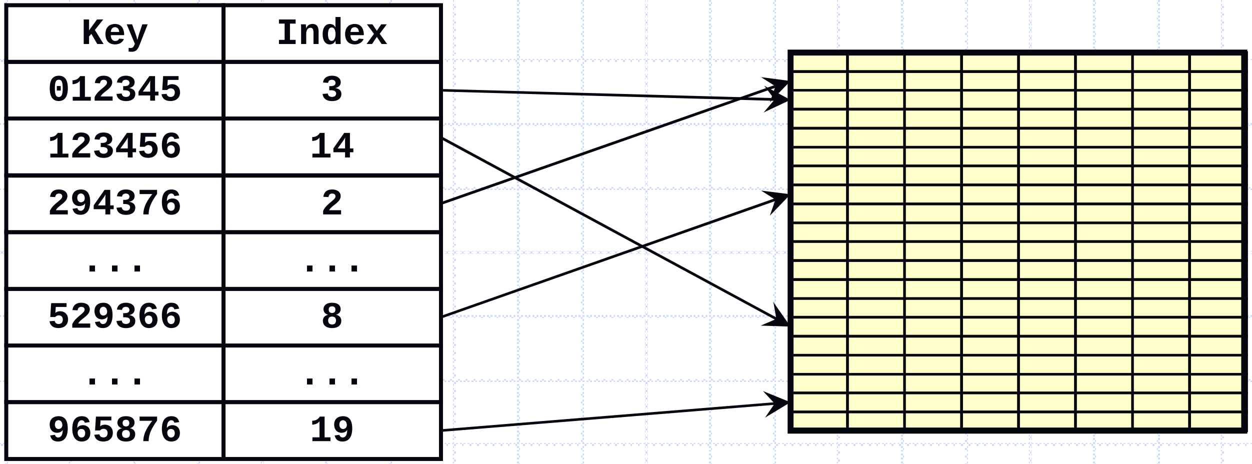
Record-oriented sequential files

- in most modern systems the record oriented files are still stored sequentially
- individual records are separated by a header
  - header contains the meta-data (field defns etc) that define the record
- the data is accessed at the record level (as opposed to byte level), for example:

```
fileID = open(fileName)
close(fileID)
getRecord(fileID, record)
putRecord(fileID, record)
seek(fileID, position)
```

Indexed sequential files

- in the previous examples, records are always accessed sequentially, in a stream
- there are many cases where we would prefer to have random access to the records
- to do this we define an "index" on the file
  - the index is a lookup table (ie: hashmap?) that maps a key value to each record in the file system



Key	Index
012345	3
123456	14
294376	2
...	...
529366	8
...	...
965876	19

ISAM

- probably the most ubiquitous indexed-sequential file structure is ISAM (*indexed sequential access method*)
- developed by IBM (1970's)
- still used today
  - MySQL is based on MyISAM – which is the same technology

  - ➢ **data is organized into fixed length fields**
  - ➢ **fields are groups into records**
  - ➢ **records are stored sequentially on the storage device (tape, disk etc)**
  - ➢ **a hashmap is defined as the index to the sequentially stored records**

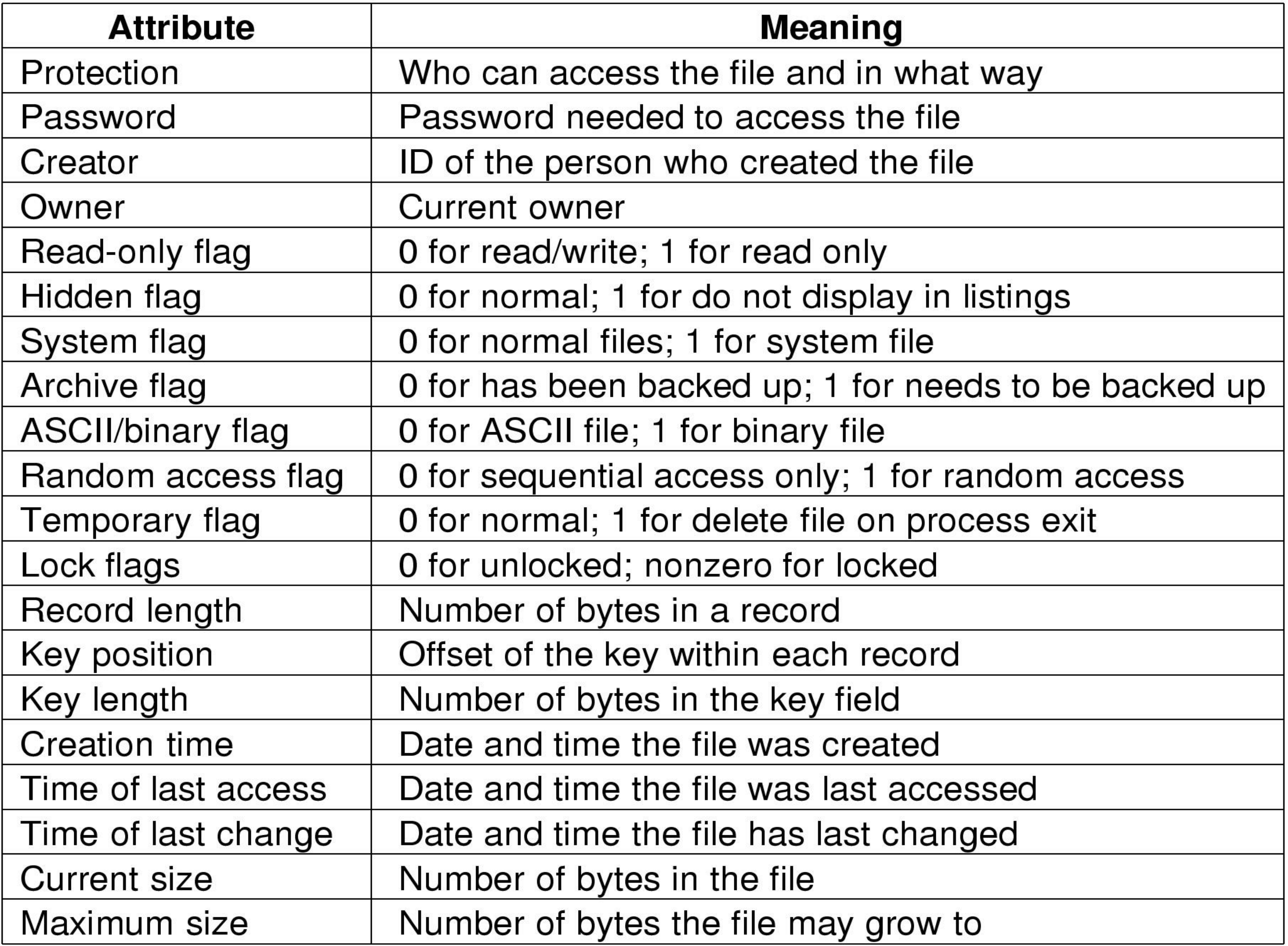
- today we typically use
  - byte-stream files when we need files in a program
  - dbms when we need structured data (records) in a program
    - internally the dbms will define some form of isam or vsam storage mechanism, but it is transparent to the programmer/user

File attributes

- the file structure must be able to store file meta-data
  - ie: attributes that describe each file, such as
    - 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 methods

- Sequential Access
  - read all bytes/records from the beginning
  - cannot jump around (but could rewind or back up) convenient when medium was magnetic tape
  - the current position (byte or record) is updated after each access (read/write operation)

- Random Access
  - can read bytes (or records) in any order
  - became the standard access method as disk drives became default storage media
  - Two ways to access records|bytes
    - option 1: two system calls
      - move position, eg: seek(), then read()
    - option 2: one system call
      - read system call requires position to start read at

File system programming interface (system calls)

- A typical file system interface will include functions to:
  - 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 (an example)

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

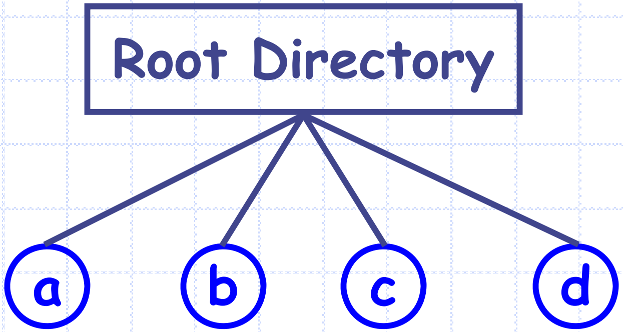
Directories

Directories

- so far we have just considered single files,
- we need a way to keep track of all the files, so we put them in directories
- directories are implemented as files that store file names and pointers to the first block of the data in the file
- there are a two basic ways to use directories to organize a file system
  1. use a flat file system (all files in one folder)
  2. use a hierarchical file system, where directories can have links to other directories

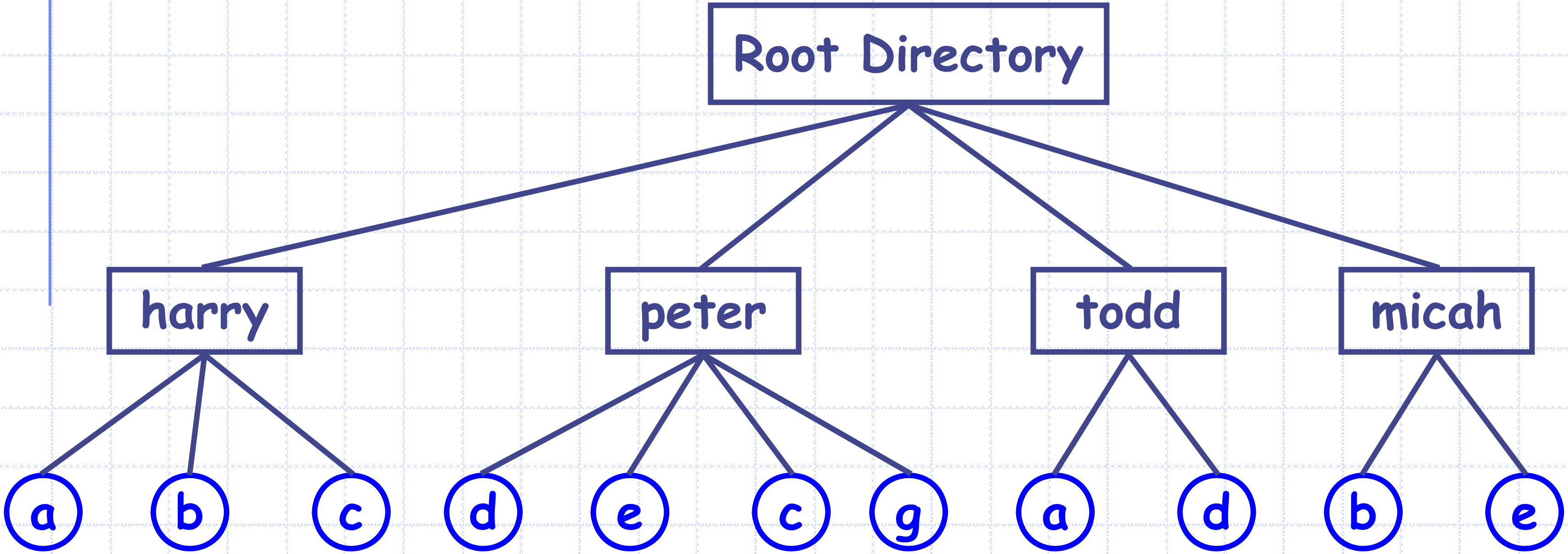
Single level directories

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



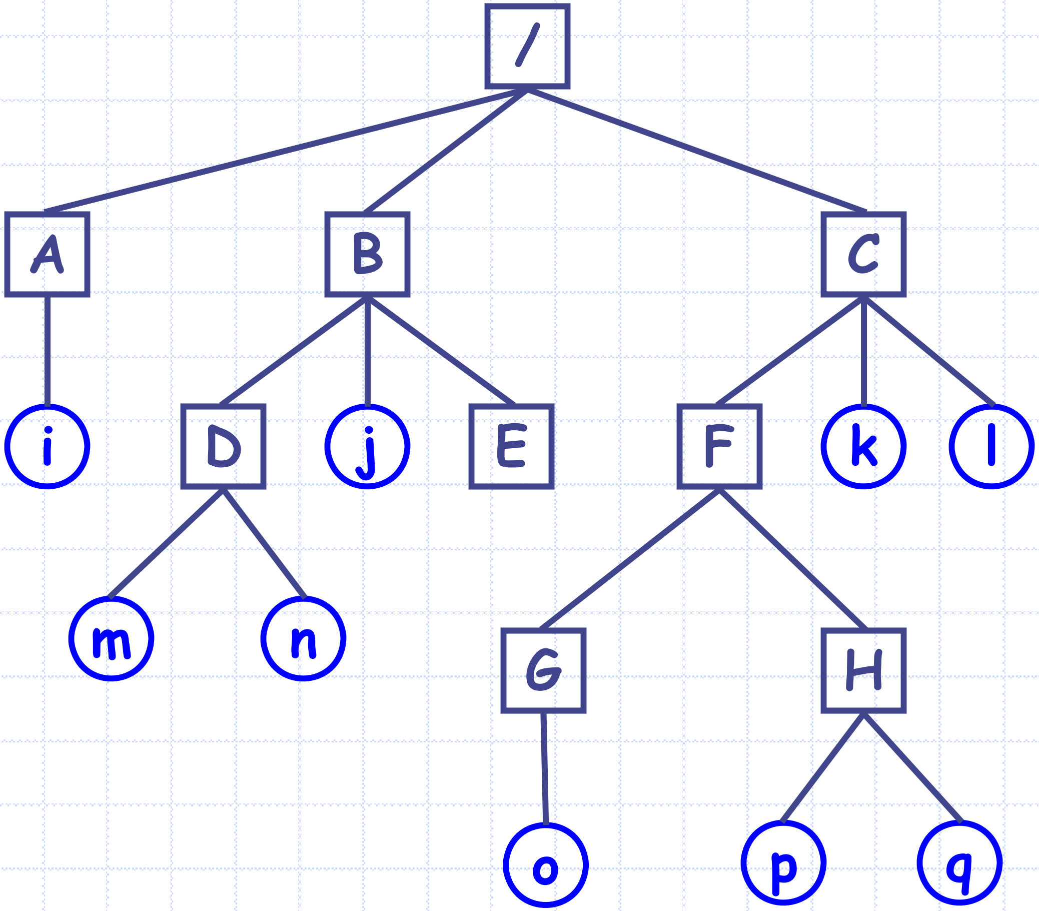
Two-level directory systems

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



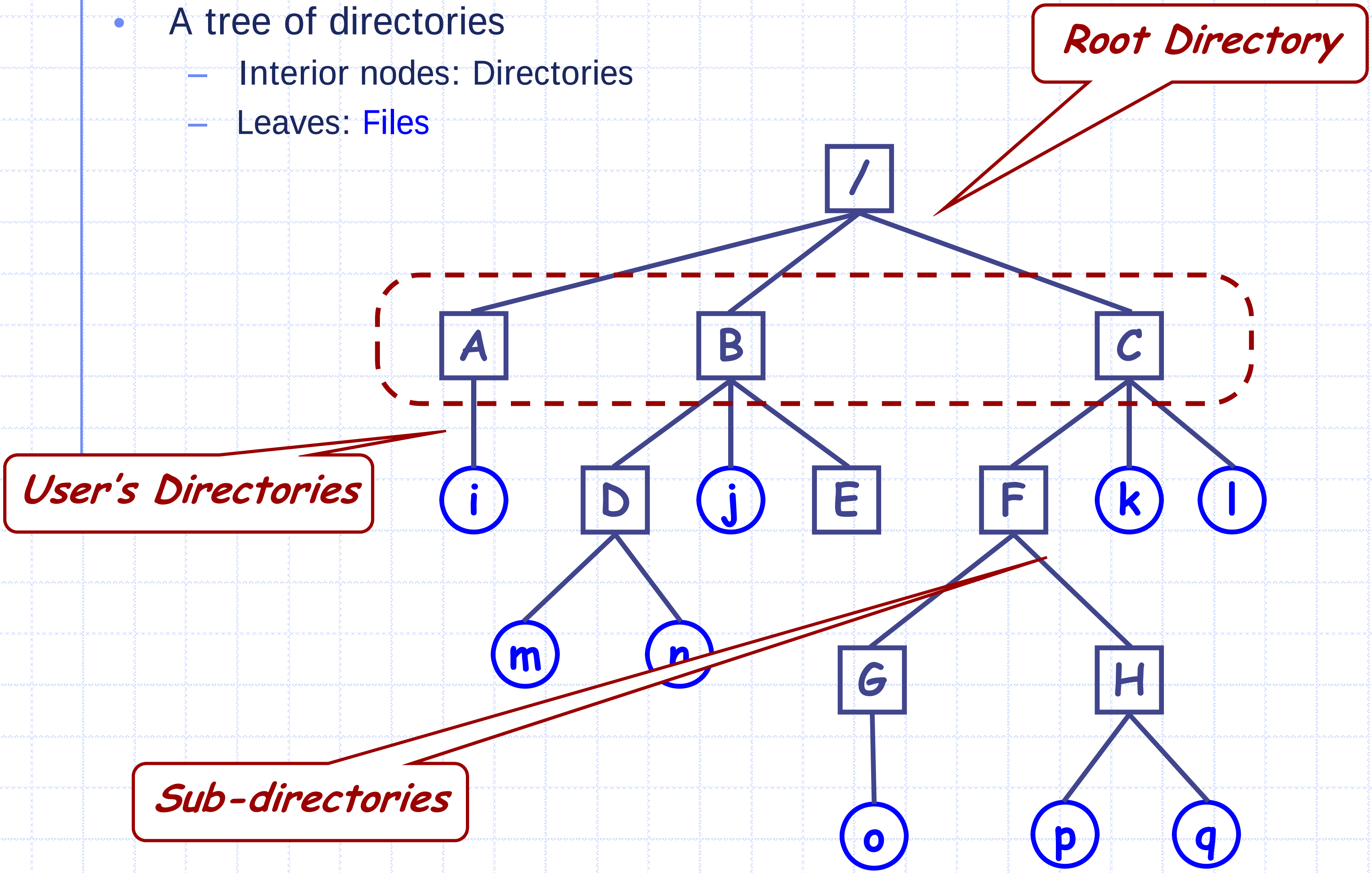
Hierarchical directory systems

- 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>harry>mailbox
- Windows
  \usr\harry\mailbox
- Unix
  /usr/harry/mailbox

Path names

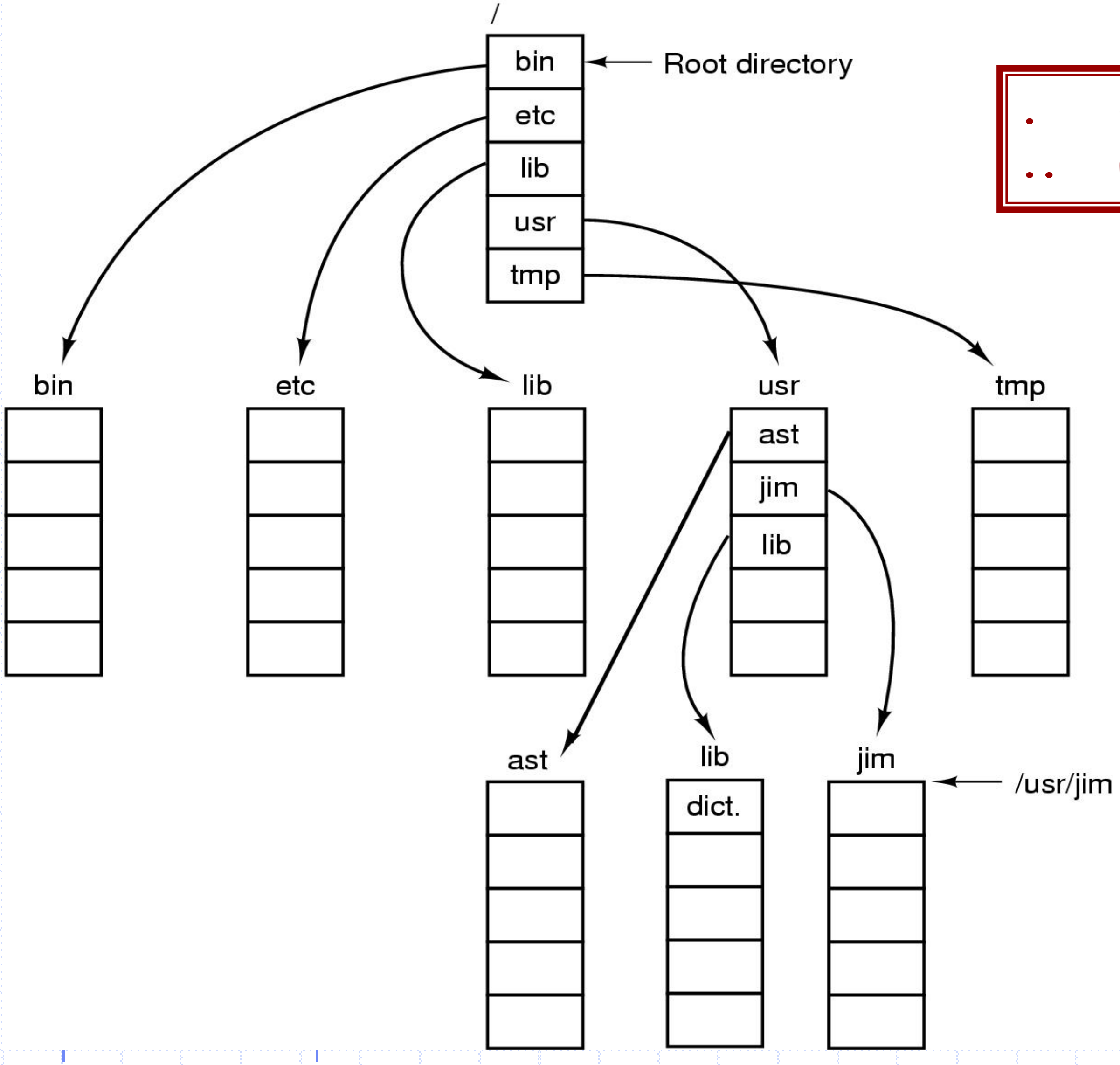
- MULTICS
  >usr>harry>mailbox
- Windows
  \usr\harry\mailbox
- Unix
  /usr/harry/mailbox

- *Absolute Path Name*
  /usr/harry/mailbox

- *Relative Path Name*
  - “working directory” (or “current directory”)
  - mailbox

Each process has its own working directory

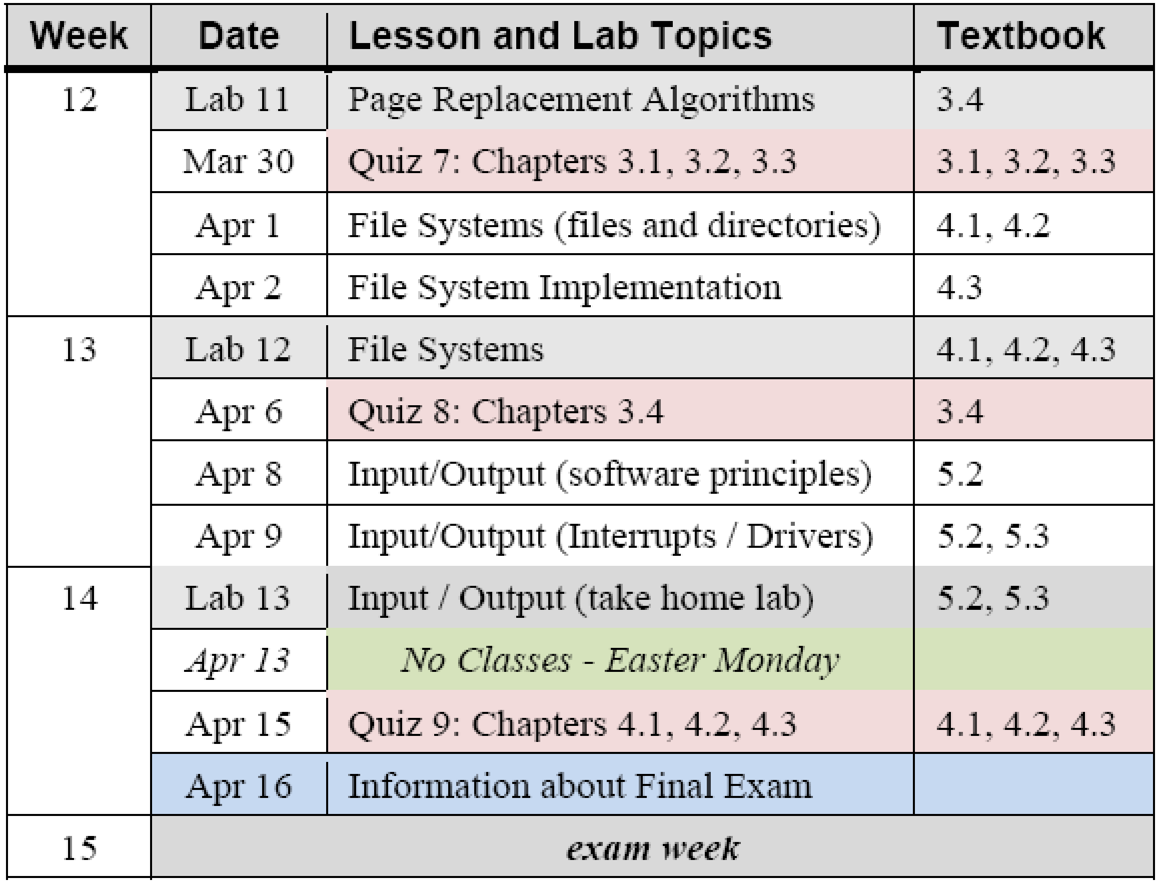
A Unix directory tree



. 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”



Week	Date	Lesson and Lab Topics	Textbook
12	Lab 11	Page Replacement Algorithms	3.4
	Mar 30	Quiz 7: Chapters 3.1, 3.2, 3.3	3.1, 3.2, 3.3
	Apr 1	File Systems (files and directories)	4.1, 4.2
	Apr 2	File System Implementation	4.3
13	Lab 12	File Systems	4.1, 4.2, 4.3
	Apr 6	Quiz 8: Chapters 3.4	3.4
	Apr 8	Input/Output (software principles)	5.2
	Apr 9	Input/Output (Interrupts / Drivers)	5.2, 5.3
14	Lab 13	Input / Output (take home lab)	5.2, 5.3
	Apr 13	*No Classes - Easter Monday*	
	Apr 15	Quiz 9: Chapters 4.1, 4.2, 4.3	4.1, 4.2, 4.3
	Apr 16	Information about Final Exam	
15	*exam week*		

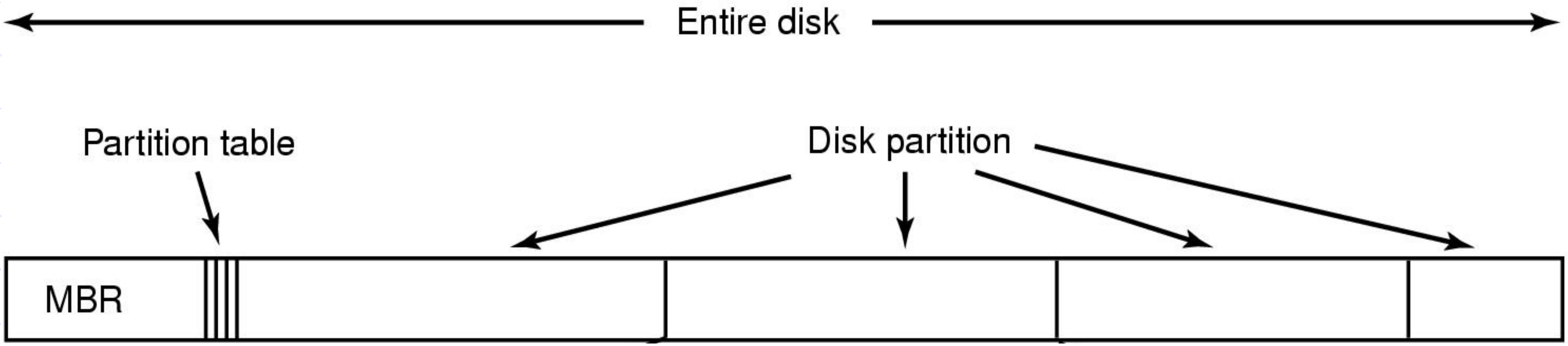
File Storage on Disk

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
- 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 , determines active partition, then reads & execs a boot block in the active partition

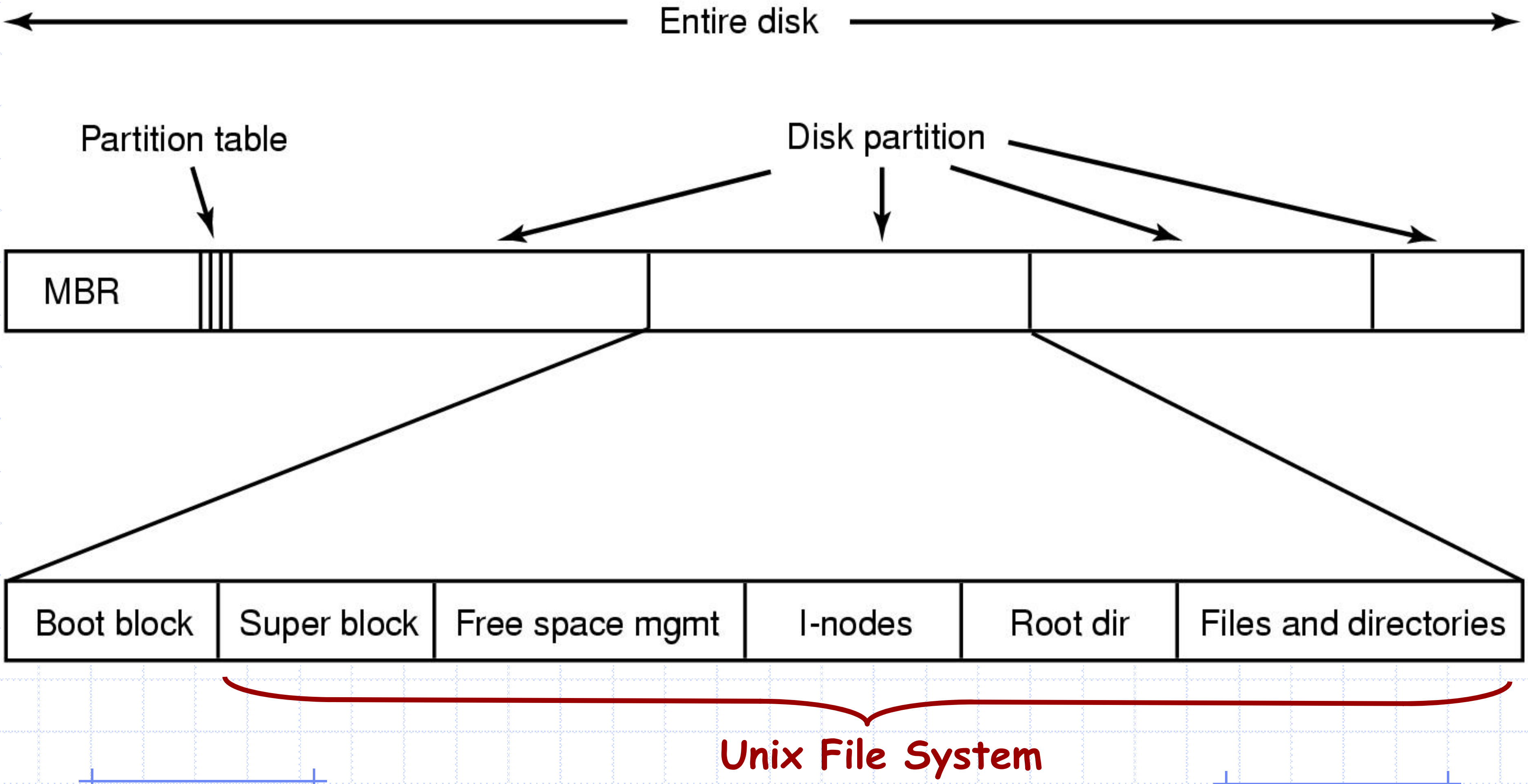
An example disk

- master boot record contains the partition table
  - partition table defines the start and end of each partition on disk



An example disk

- each partition holds a file system
  - different partitions can have different types of file systems



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
  - File data must be stored in sectors
  - Disk drives typically have different numbers of sectors per track
- File systems define a block size
  - block size allows us to access the disk without worrying about number of tracks or sectors
  - block size = 2^n * sector size
  - Contiguous sectors are allocated to a block
- File systems view the disk as an array of blocks
  - *File Systems needs a way to allocate blocks to file*
  - *File Systems must manage free space on disk*

Implementing Files

- from the preceding discussion we know that files are made up of a series of “blocks”
  - these blocks need to be allocated and linked together whenever a file is created or updated
- in addition to space for the data, we also need to maintain some information about the file itself, for example
  - file name
  - owner
  - protection settings
  - length
  - time of creation
  - time of last modification
  - time of last access
- this information can be stored in one of two places
  - in the directory entry (more on this later), or
  - with the data … maybe in the first block of the file

Allocating blocks to files

- assuming that a file is a series of blocks, what we need is some way to allocate blocks to files, and to keep track of the blocks that make up a file

- there are 3 well known ways that this is done:

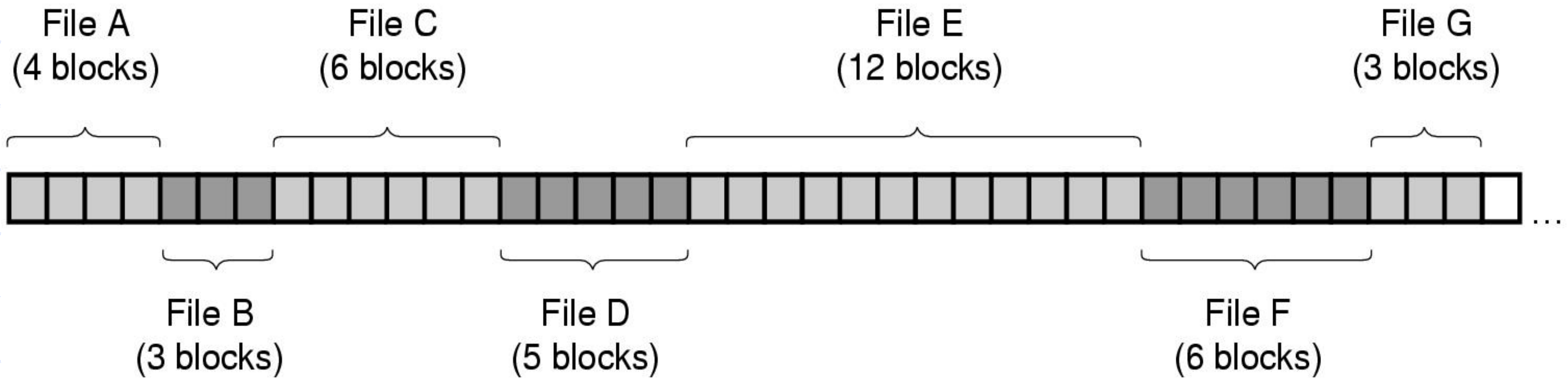
  1. maintain the file as a contiguous set of blocks on the storage device
  2. maintain the file as a list of blocks interconnected with links
  3. maintain the file as a collection of blocks interconnected with some kind of index file

  – now we will examine each of these methods …

Contiguous allocation (1)

Idea:

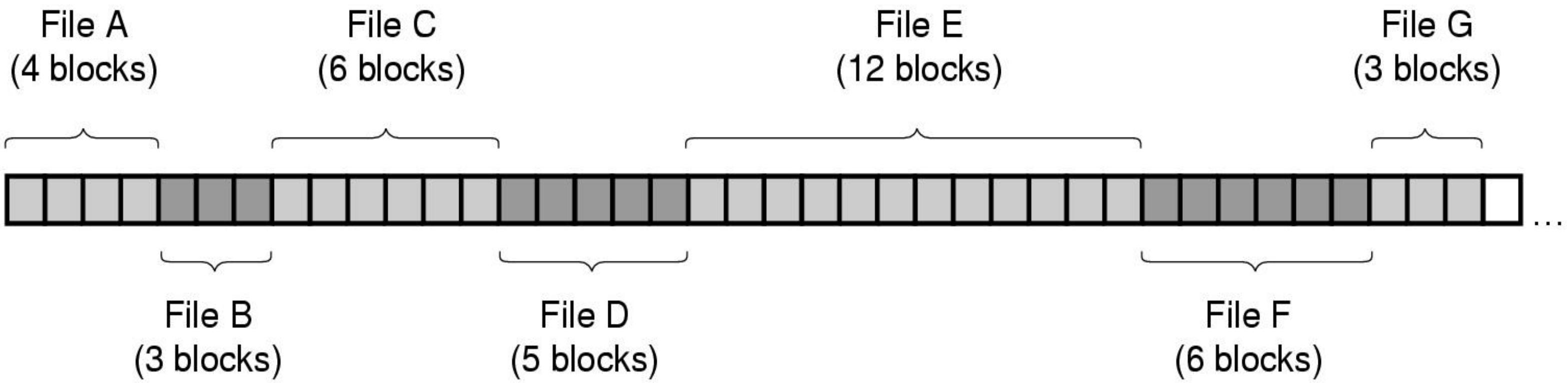
- all blocks in a file are contiguous on the disk
- this method treats the random accessed device (hard disk) as a sequentially accessed device (such as a tape)



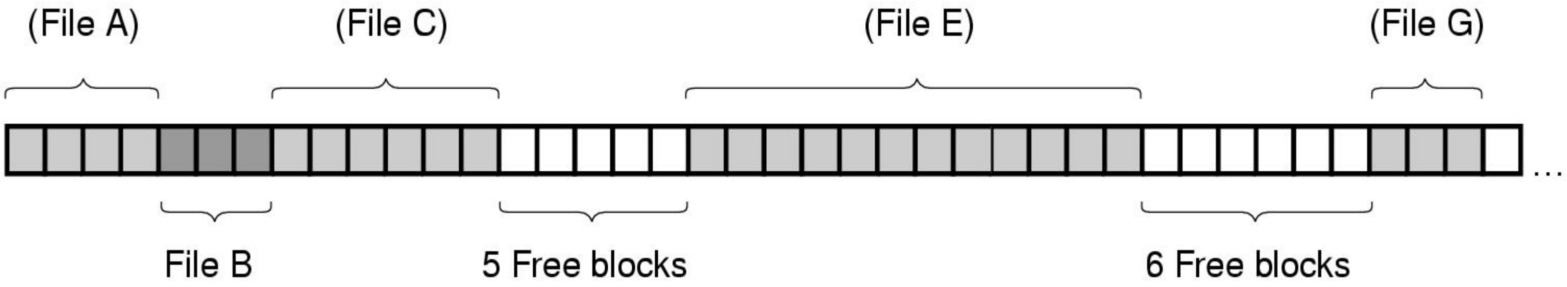
Contiguous allocation (2)

Idea:

- since files are always stored in sequential blocks, we can end up with external fragmentation on the disk



After deleting D and F...



Contiguous allocation (3)

- one of the problems with contiguous allocation is “where to place the file”

- this same problem was addressed in the section on memory management

- we will have to rely on a placement algorithm, such as
  - best fit
  - worst fit
  - first fit

- we also have the problem of what to do if the file grows. We either need to
  - add the next contiguous block (which may or may not be available), or
  - move the file to a different location on disk

Contiguous allocation (4)

Advantages:

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

Disadvantages:

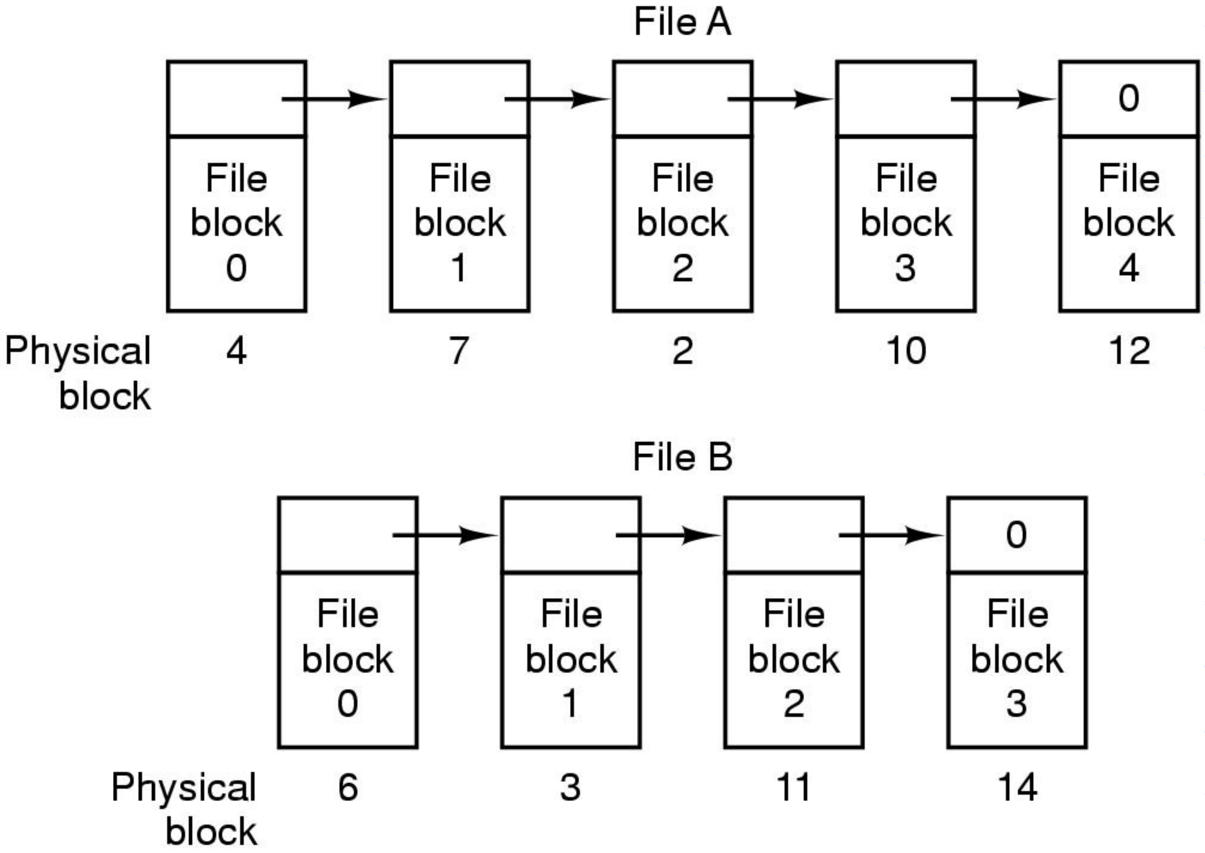
- After deletions, disk becomes fragmented
- Will need periodic compaction (time-consuming)
- Will need to manage free lists
- If new file put at end of disk…
  - No problem
- If new file is put into a “hole”…
  - Must know a file’s maximum possible size *… at the time it is created!*

Contiguous allocation (5)

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

Linked list allocation (1)

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



Random access into the file is slow!

Linked list allocation (2)

Advantages:

- Scalability (it is easy to add or remove blocks as the file grows/shrinks)
- Disk fragmentation is not an issue

Disadvantages:

- File could be stored all over the disk, requiring more seek time when reading or writing multiple blocks
- Random access is slow, as we may need to traverse the entire list
- Susceptible to corruption (if one block gets messed up we can lose a large part of the file and have no way to find it again)

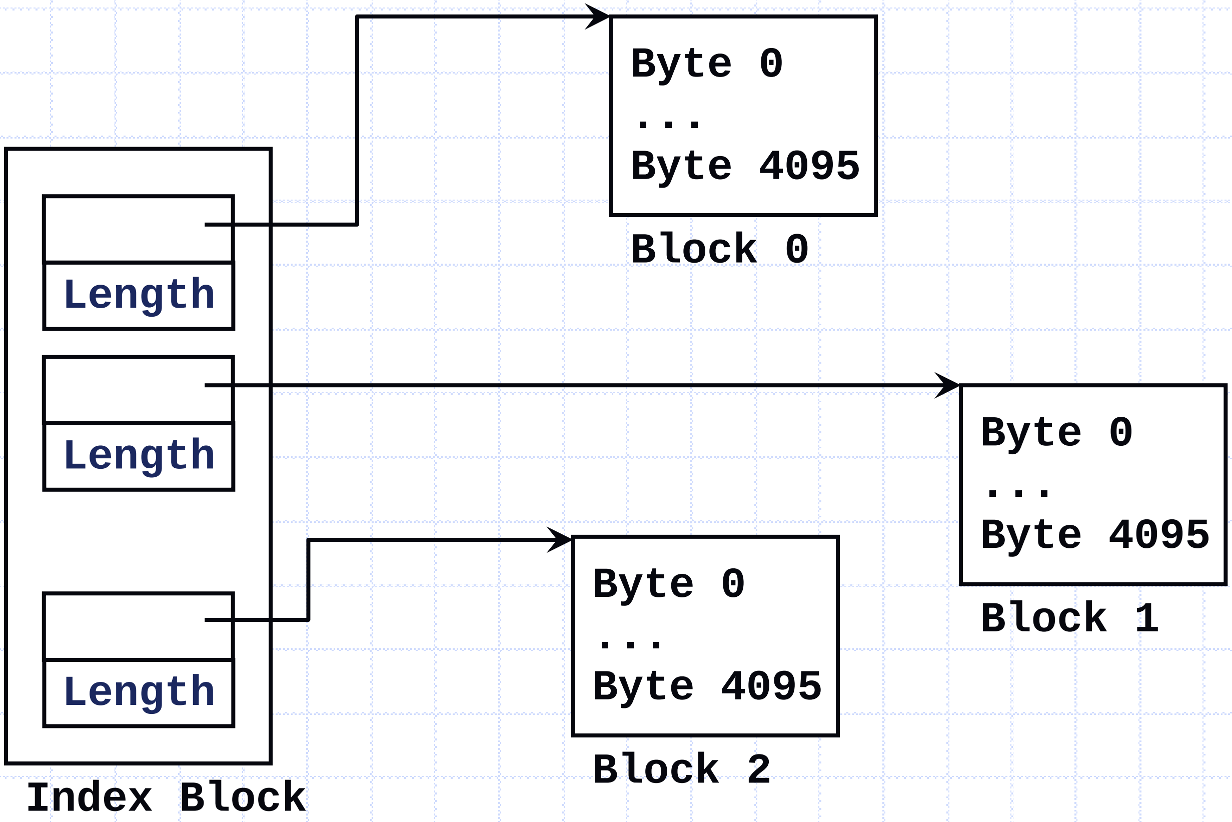
Variation:

- Implement as a list of pointer to blocks, and store in main memory
- This makes traversal much faster as we don't have to seek all over the disk

Indexed allocation (1)

Idea:

- use one of the blocks in the file (index block?) as an index to all the other blocks in the file



Indexed allocation (2)

Advantages:

- fast file access (no list to search)
- can grow/shrink
- does not result in external fragmentation

Disadvantages:

- can waste space (internal fragmentation) if there are only a few blocks but the table in the index block has additional (unused) entries
- not as scalable as a linked list (there are a fixed number of index entries)

Implementing directories

- A directory is simply a list of files, with each entry containing
  - 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
    - Number of an index block (i-node)
  - I-Node contains
    - File Attributes

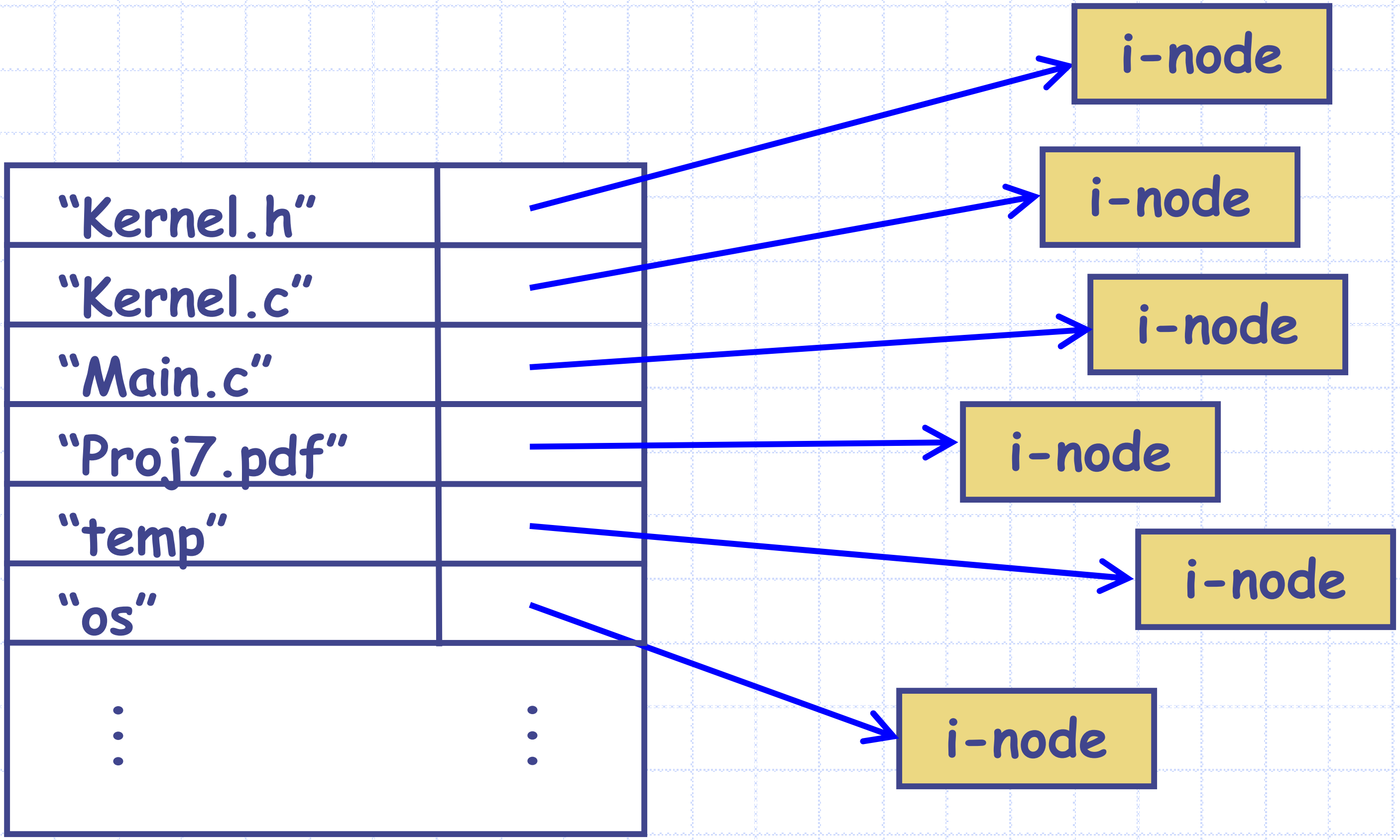
Implementing directories

- Simple Approach

“Kernel.h”	attributes
“Kernel.c”	attributes
“Main.c”	attributes
“Proj7.pdf”	attributes
“temp”	attributes
“os”	attributes
⋮	⋮

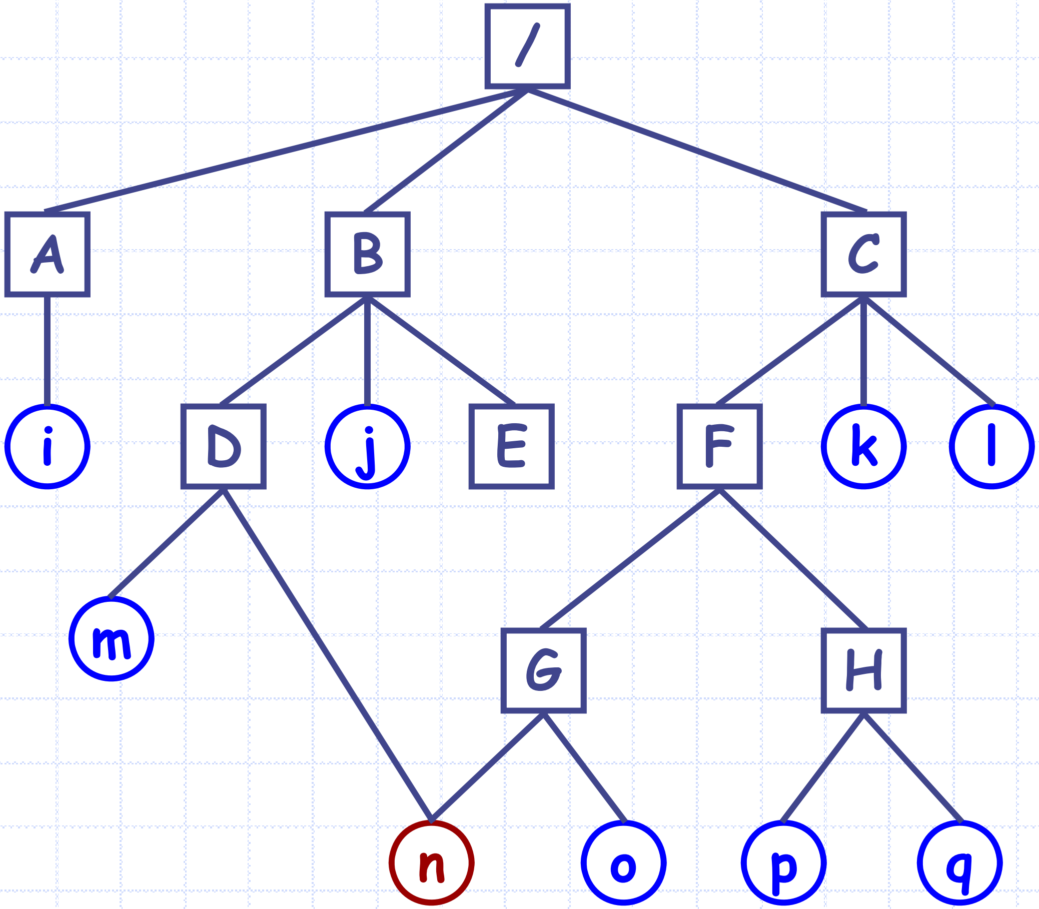
Implementing directories

- Unix Approach



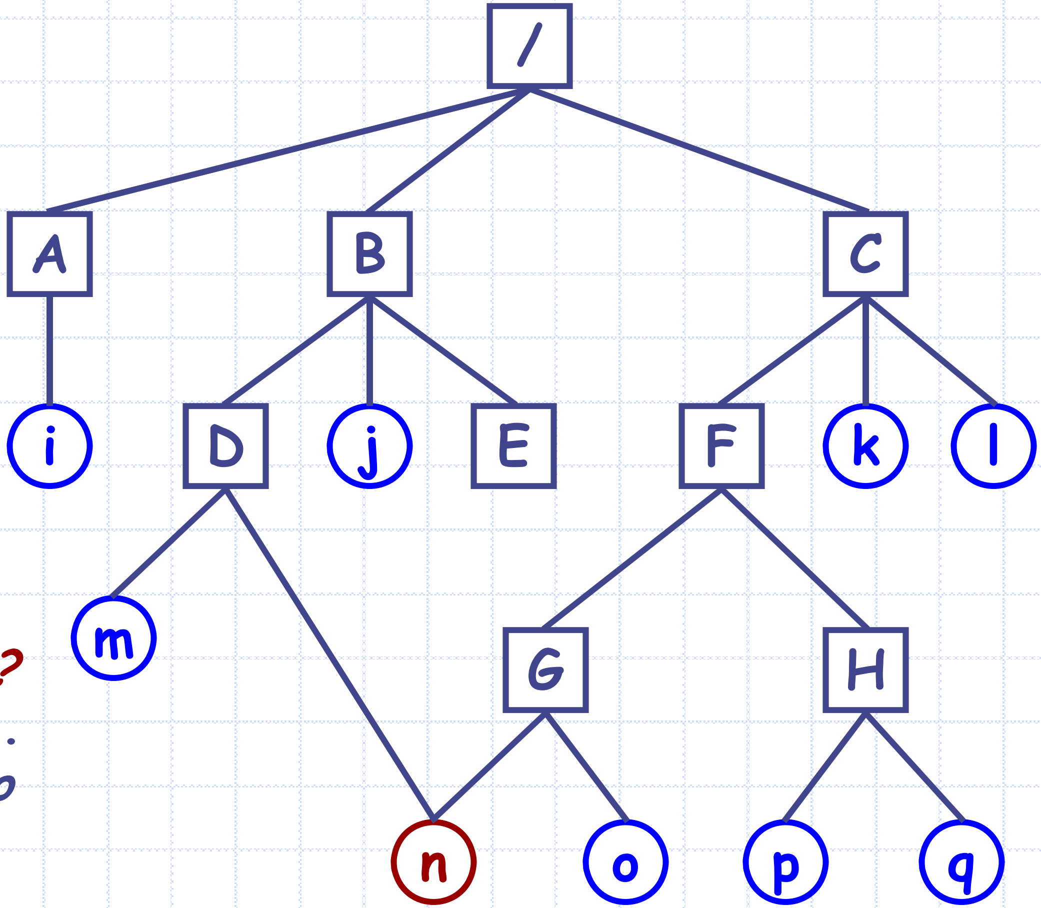
Sharing files

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



Sharing files

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



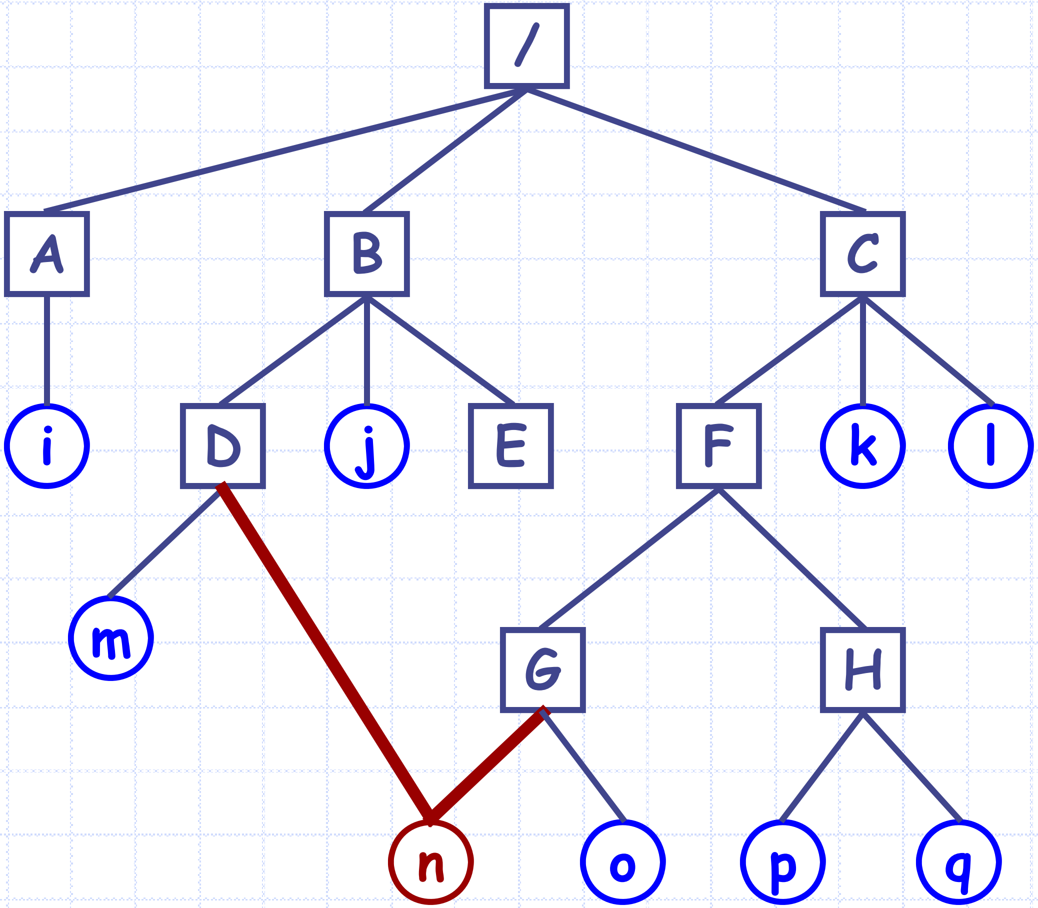
What if the file changes?
New disk blocks are used.
Better not store this info
in the directories!!!

Hard links and symbolic links

In Unix:

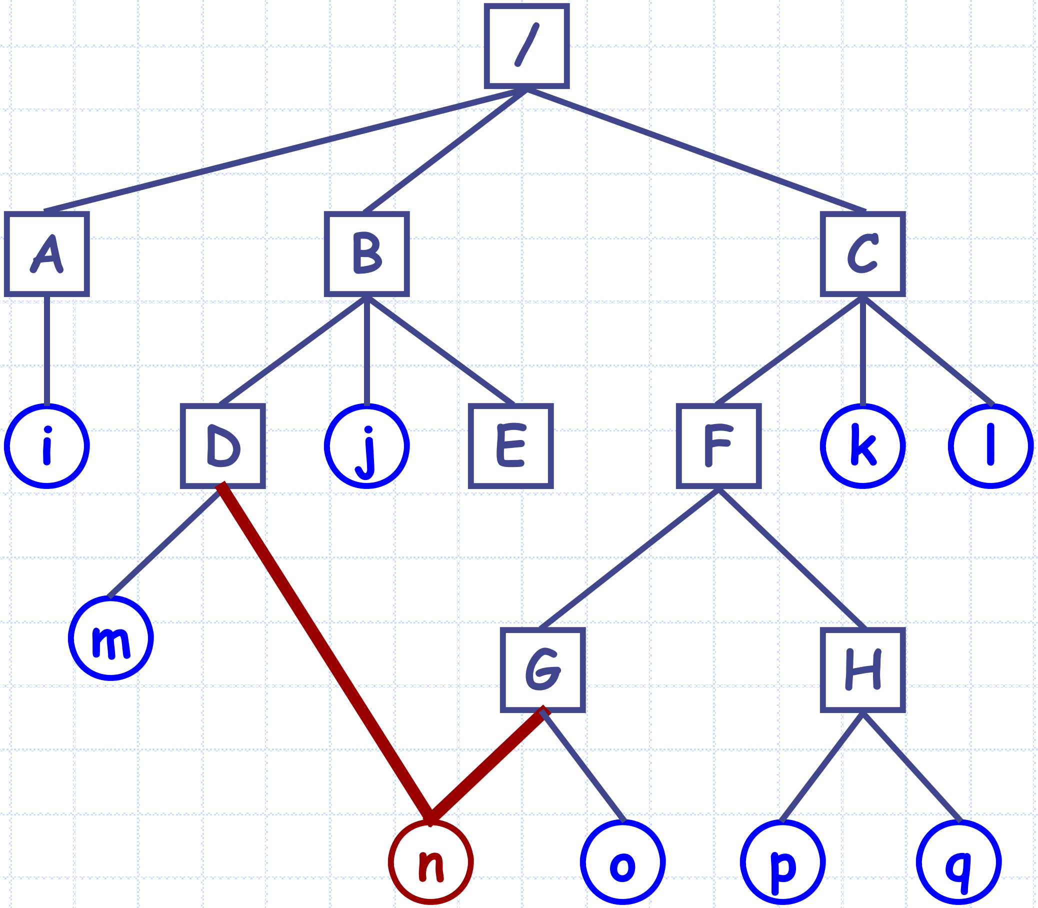
- Hard links
  - Both directories point to the same i-node
- Symbolic links
  - One directory points to the file’s i-node
  - Other directory contains the “path”

Hard links



Hard links

- Assume i-node number of “n” is 45.



Hard links

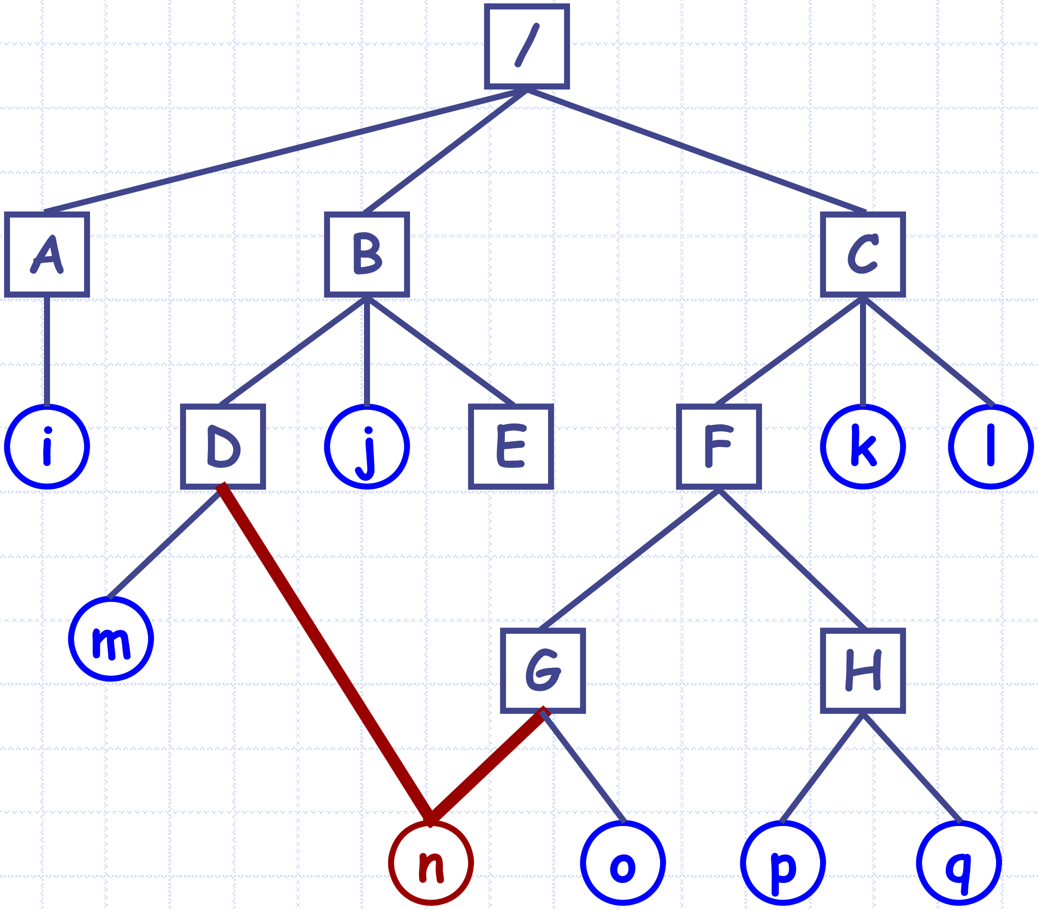
- Assume i-node number of “n” is 45.

Directory “D”

“m”	123
“n”	45
⋮	⋮

Directory “G”

“n”	45
“o”	87
⋮	⋮



Hard links

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

The file may have a different name in each directory

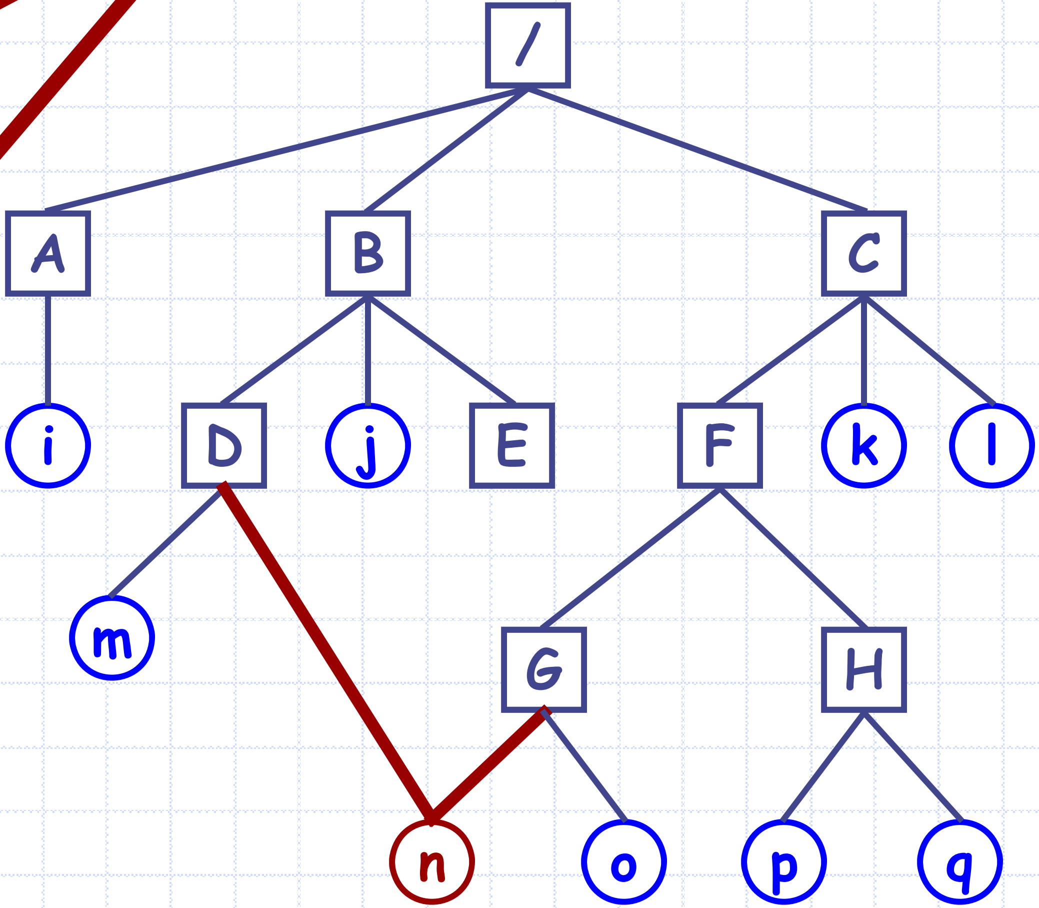
Directory "D"

"m"	123
"n1"	45
⋮	⋮

Directory "G"

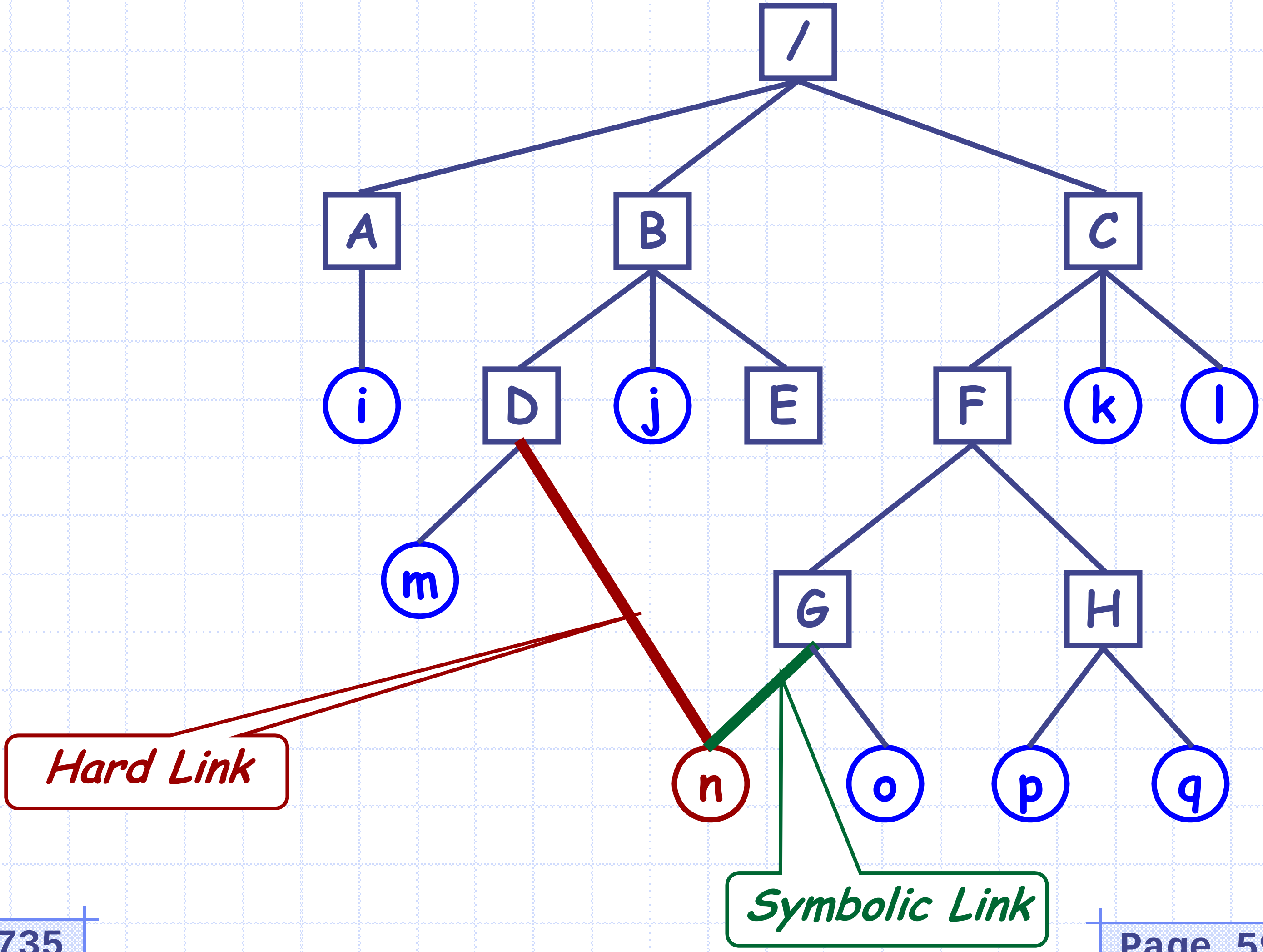
"n2"	45
"o"	87
⋮	⋮

/B/D/n1
/C/F/G/n2



Symbolic links

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

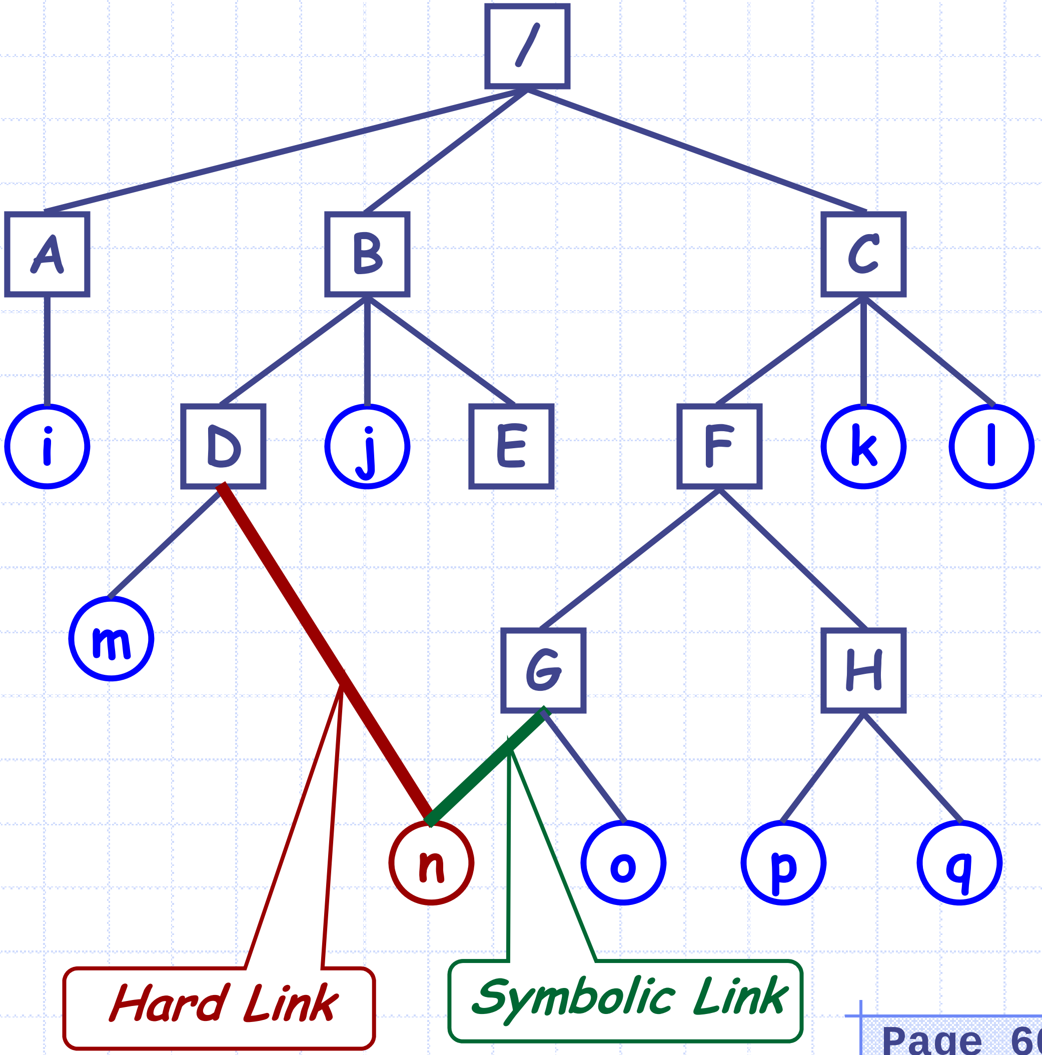


Symbolic links

- Assume i-node number of “n” is 45.

Directory “D”

“m”	123
“n”	45
⋮	⋮



Hard Link

Symbolic Link

Symbolic links

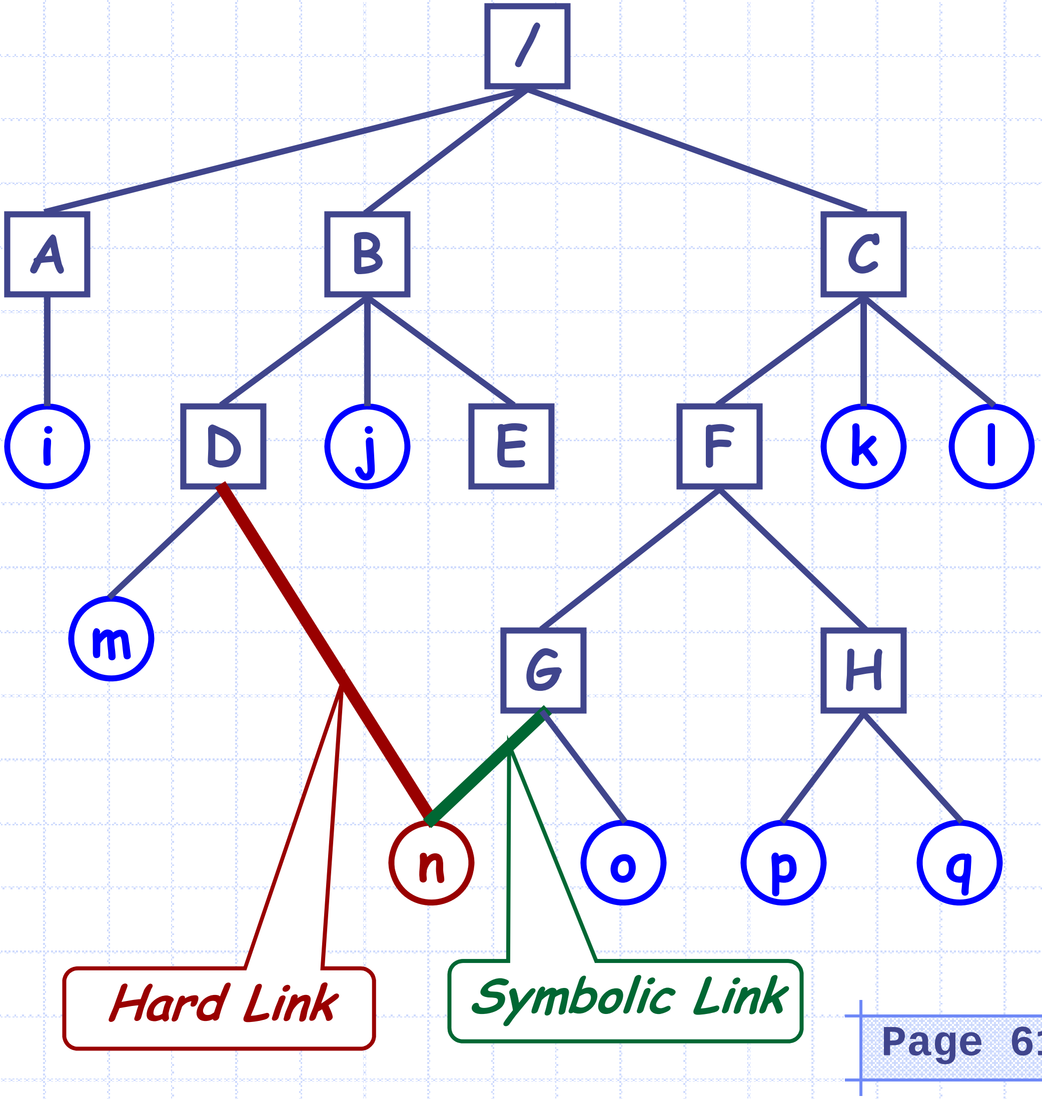
- Assume i-node number of “n” is 45.

Directory “D”

“m”	123
“n”	45
⋮	⋮

Directory “G”

“n”	/B/D/n
“o”	87
⋮	⋮



Symbolic links

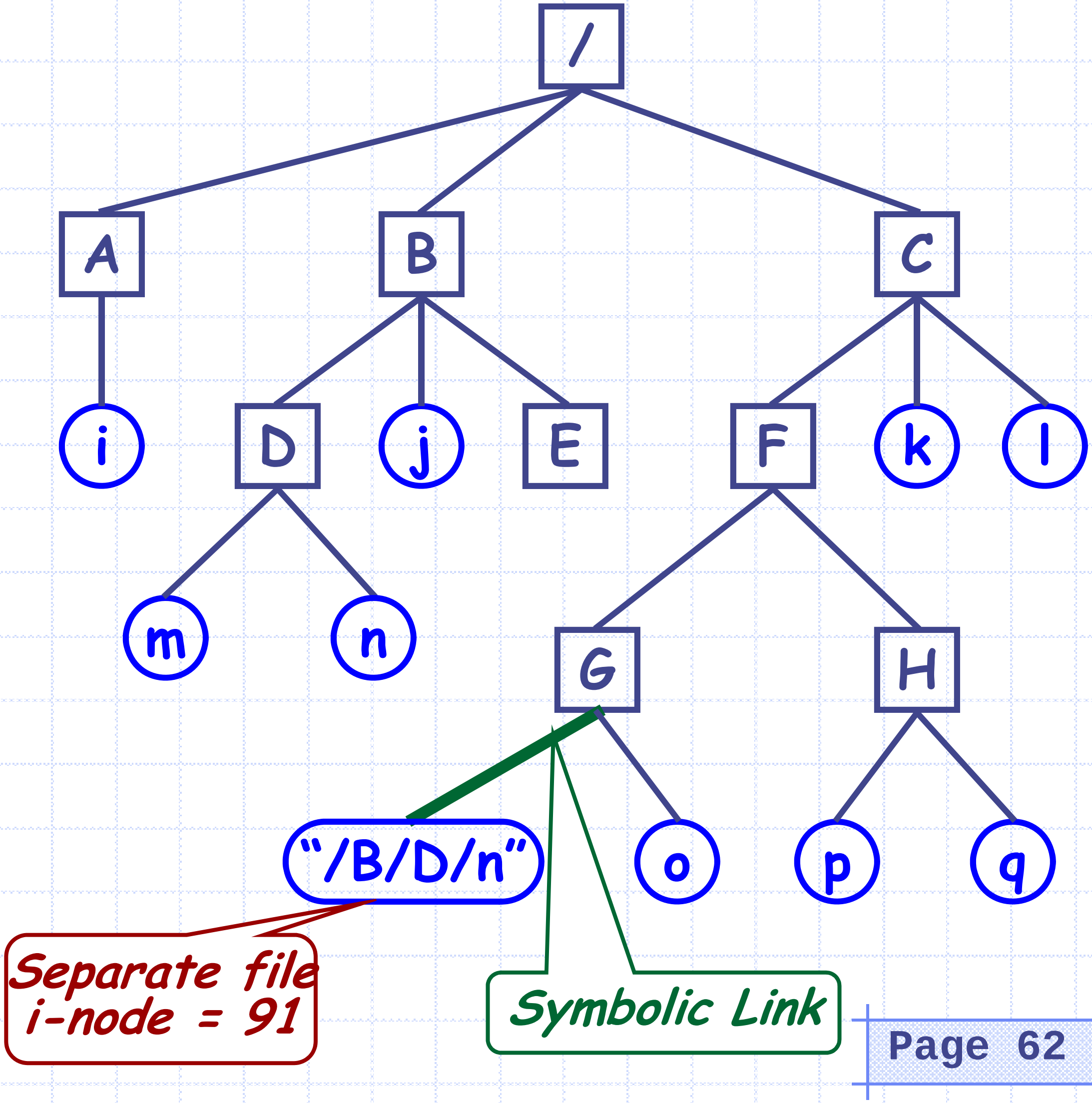
- Assume i-node number of “n” is 45.

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"

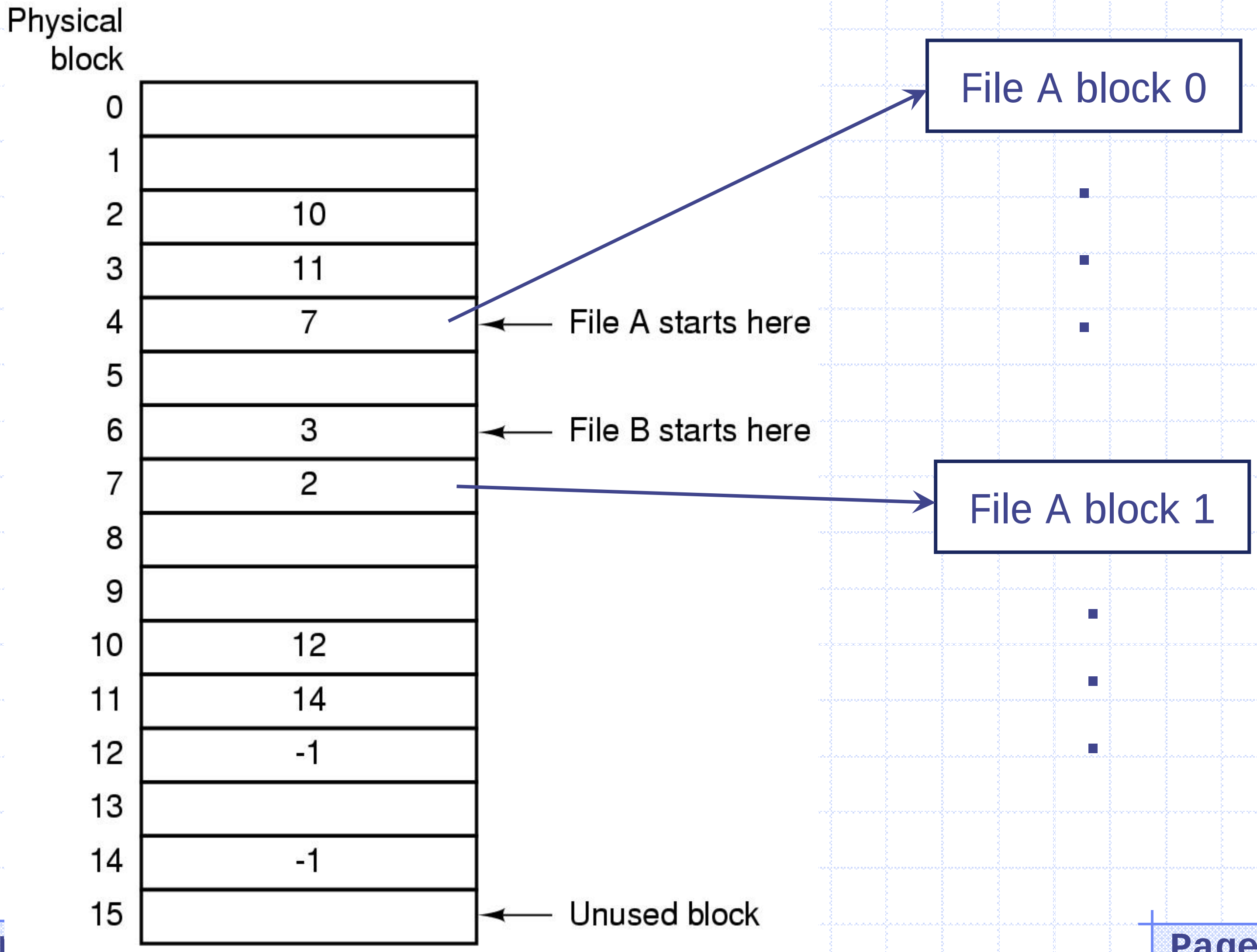
Example File Systems

- FAT
- I-node

File allocation table (FAT)

- Keep a table in memory
- The table essentially *contains* "linked lists" of the blocks in each file
- One FAT entry per block on the disk
- Each entry contains the address of the "next" block
  - the directory has a pointer to the first FAT entry for the file
  - End of file marker (-1)
- A special value (-2) indicates the block is free

File allocation table (FAT)

- 

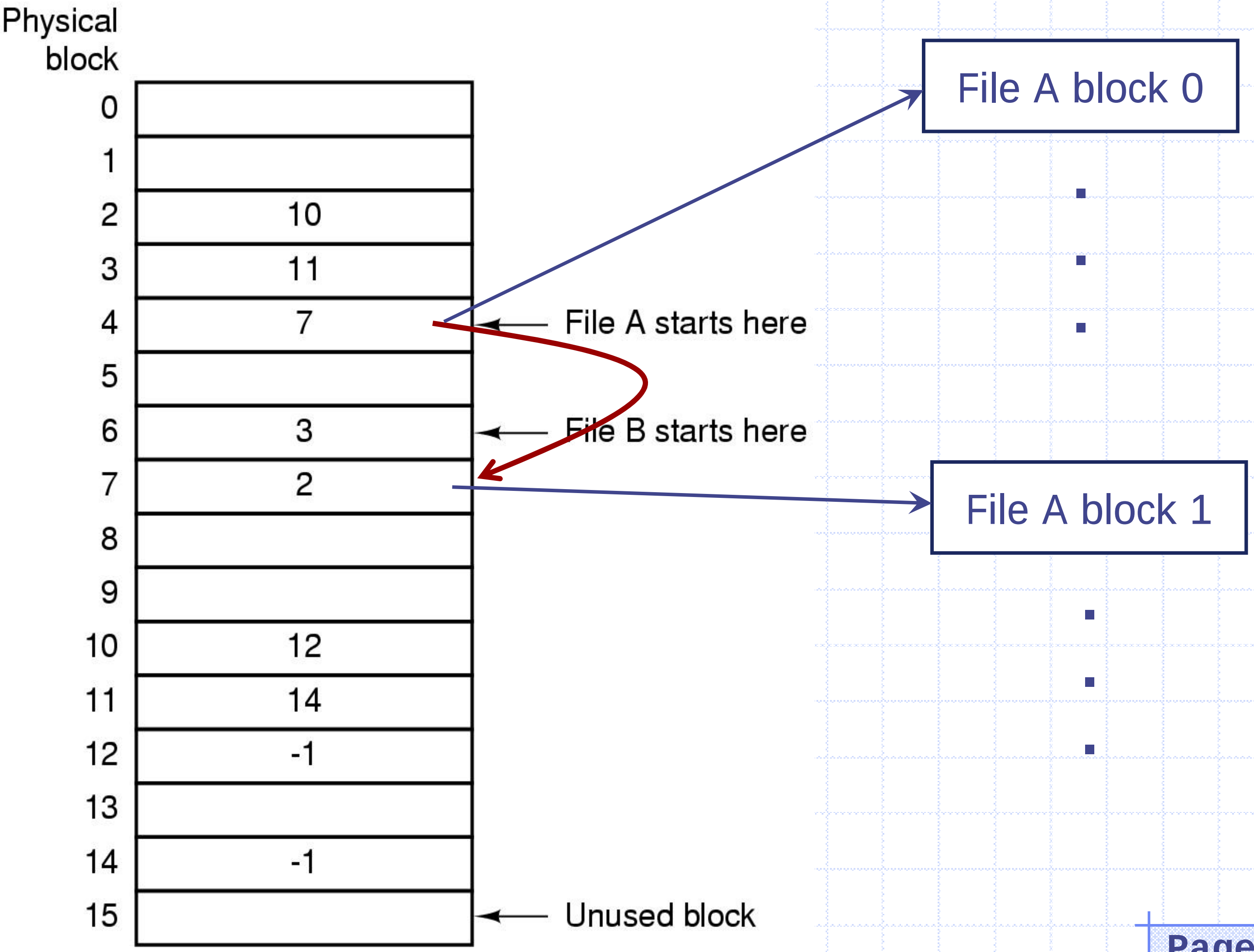
Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

File A starts here

File B starts here

Unused block

File allocation table (FAT)

- 

Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

File A starts here

File B starts here

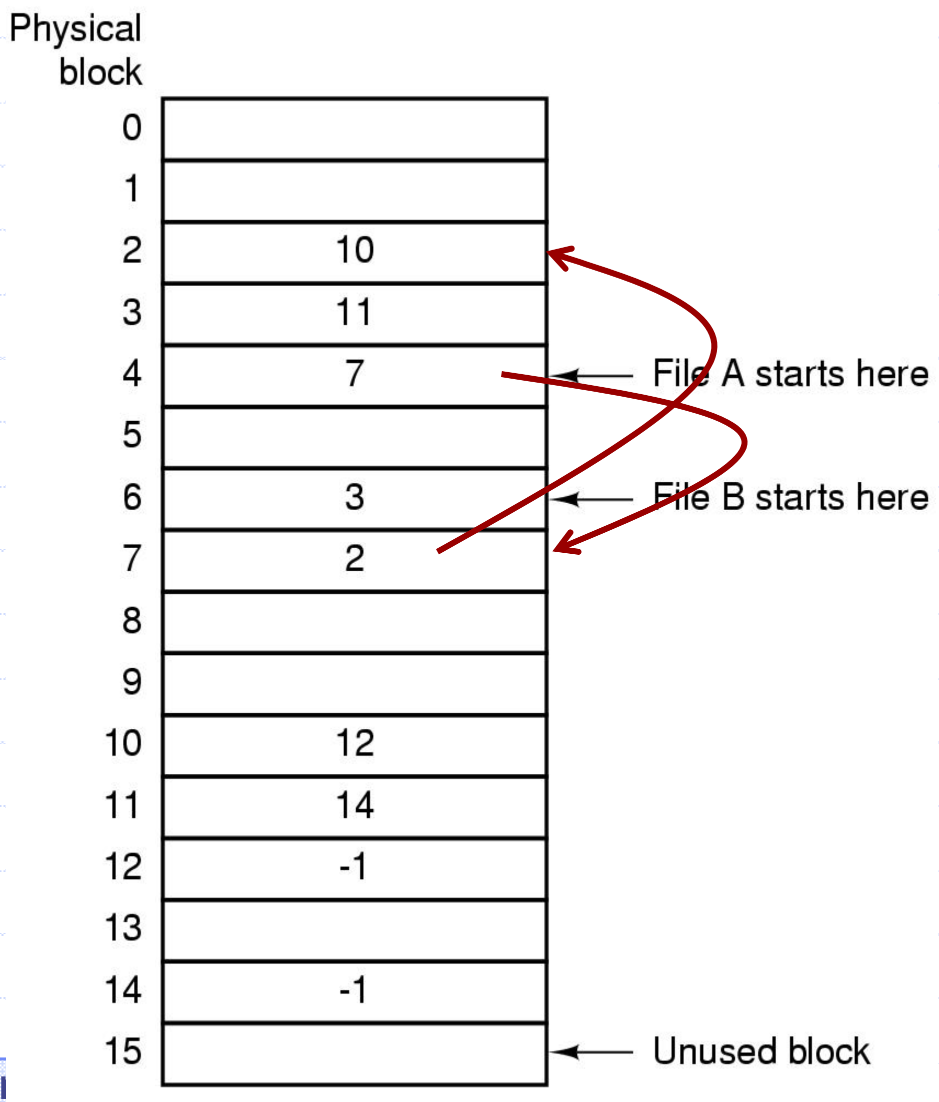
Unused block

File A block 0

File A block 1

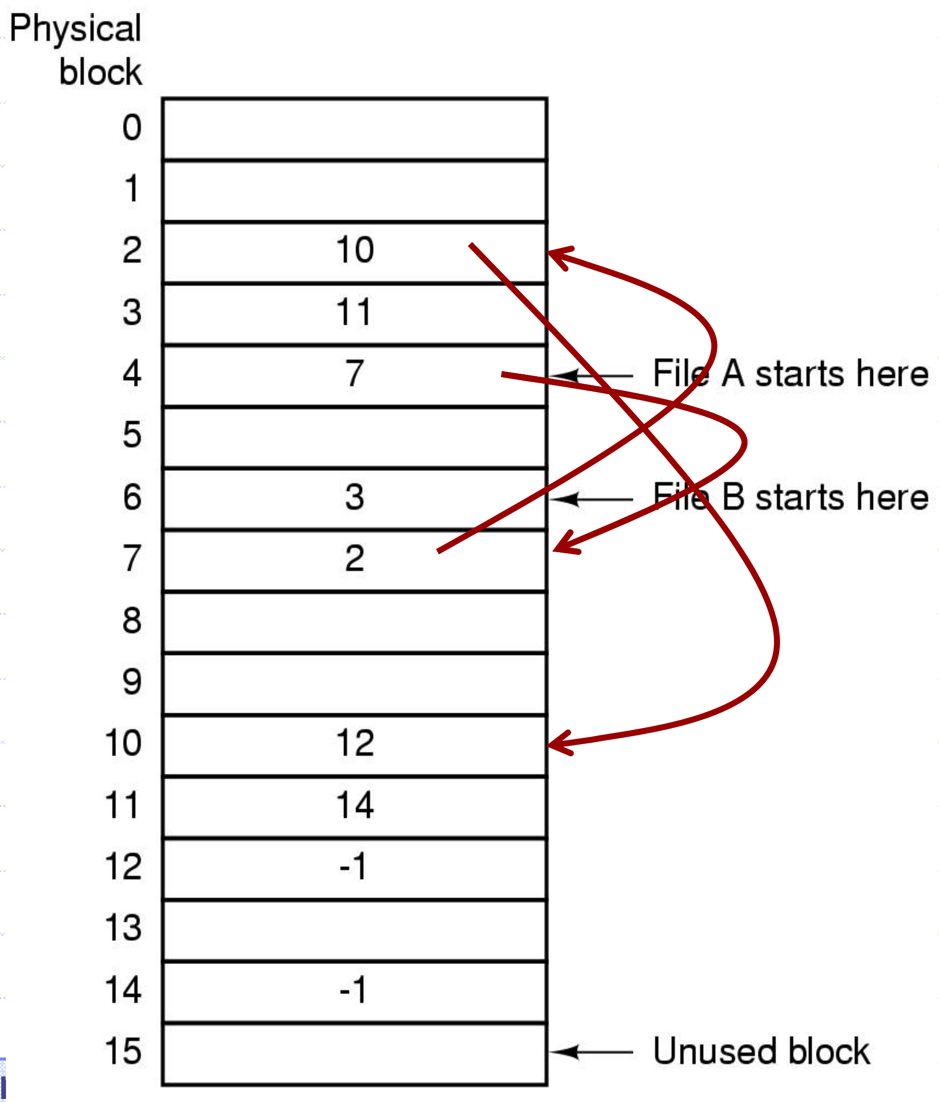
File allocation table (FAT)

-



Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

File allocation table (FAT)



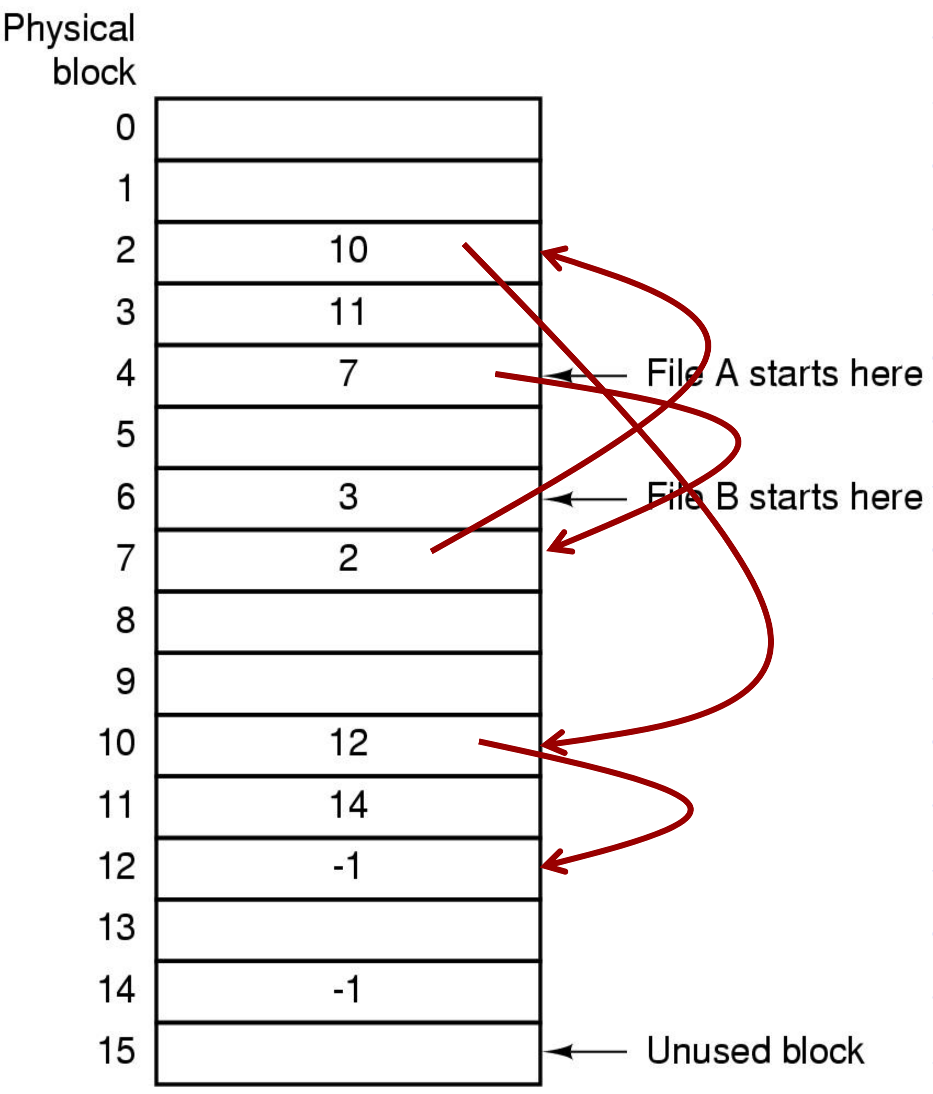
Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

File A starts here

File B starts here

Unused block

File allocation table (FAT)

- 

Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

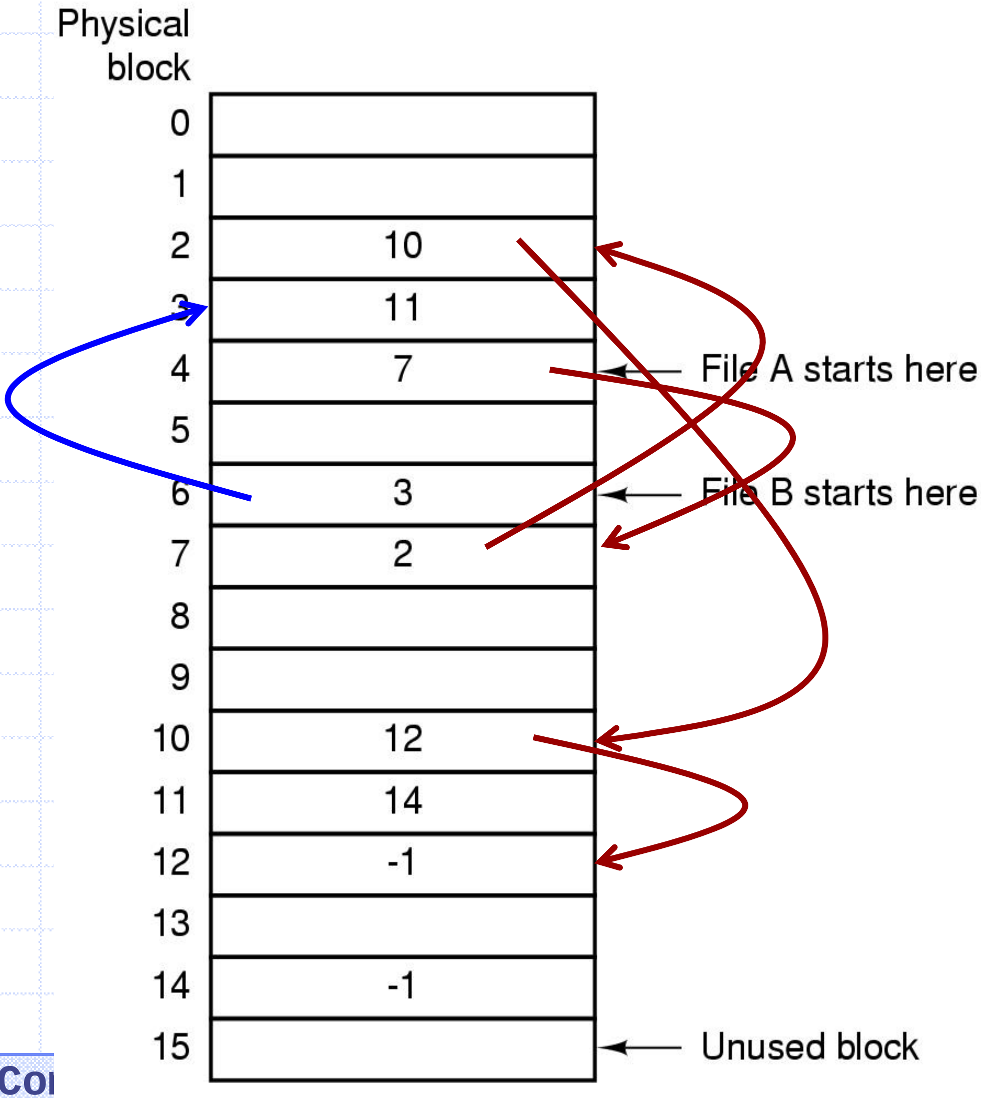
File A starts here

File B starts here

Unused block

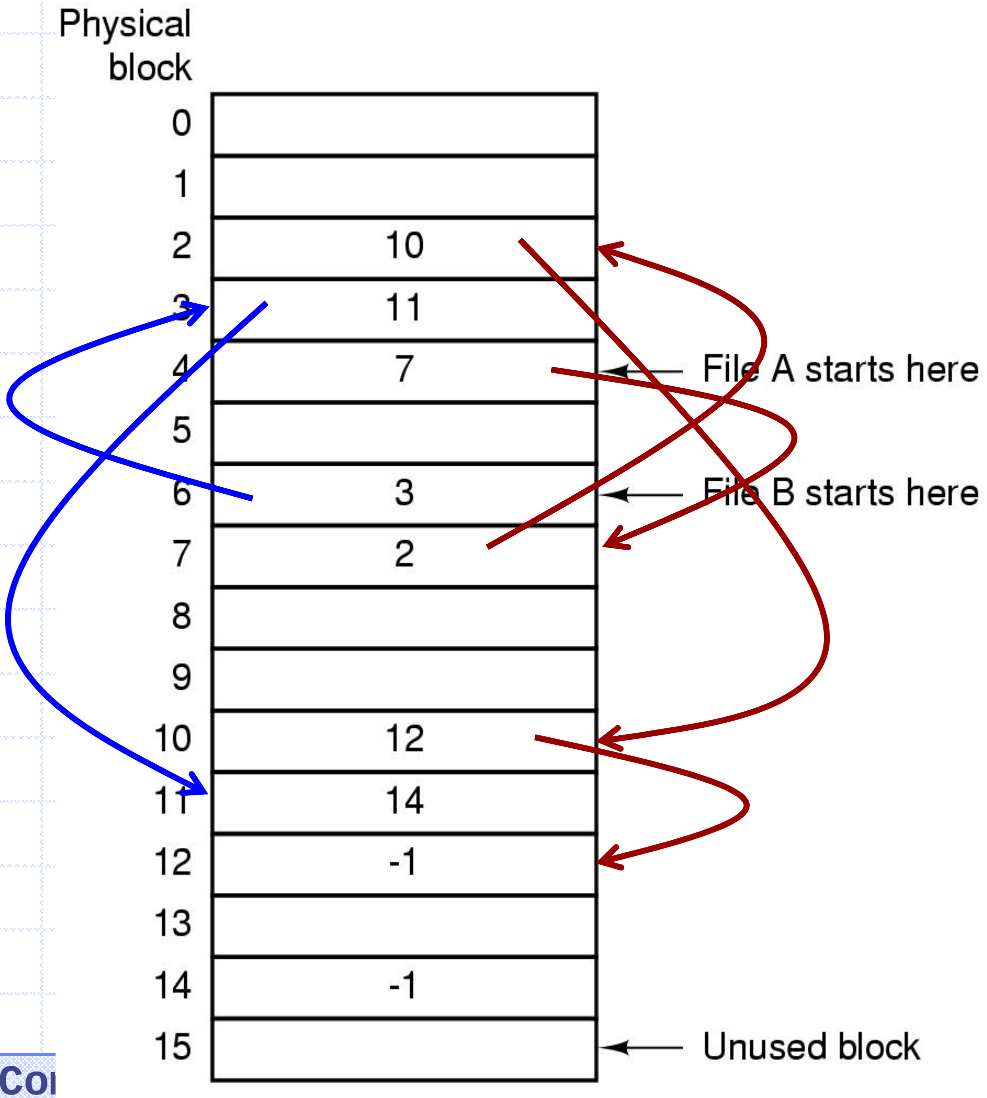
File allocation table (FAT)

-



Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

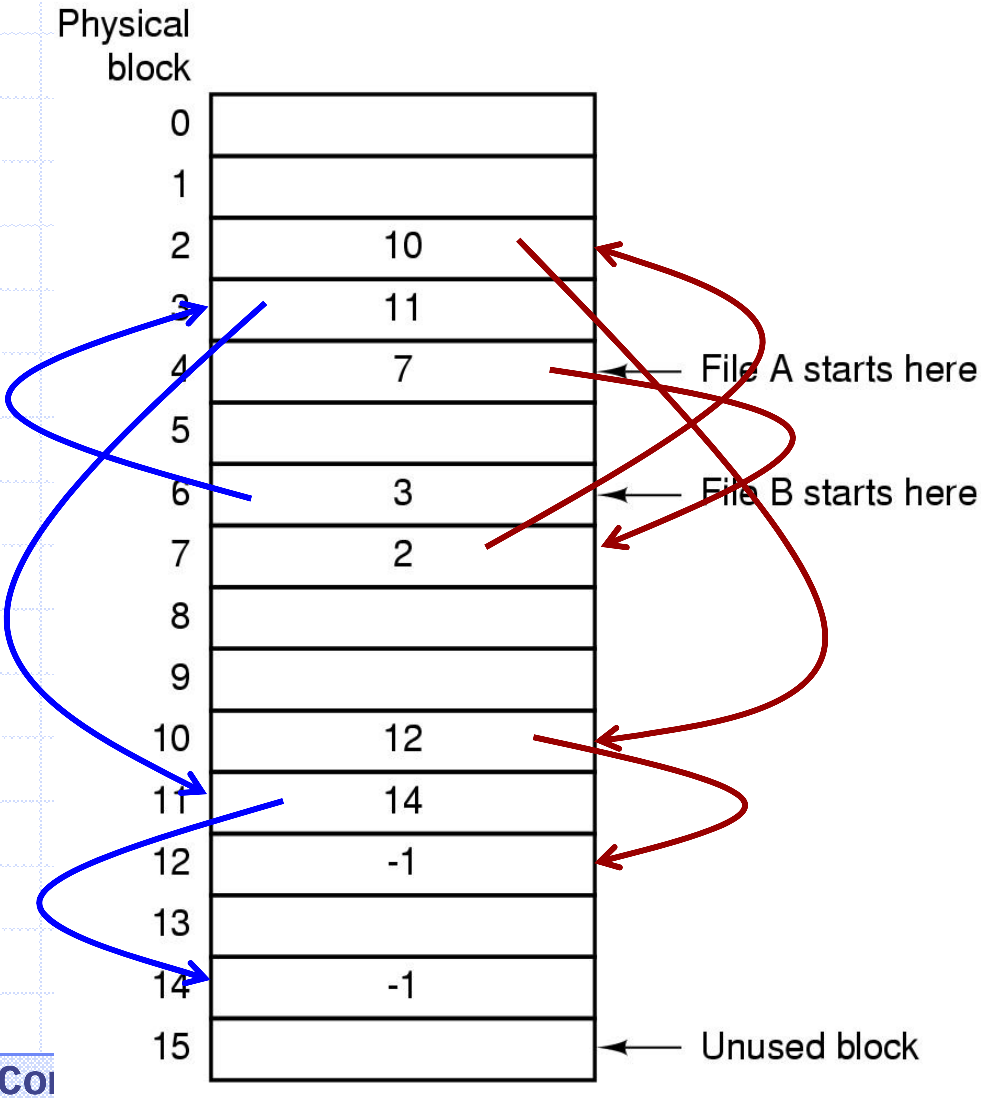
File allocation table (FAT)



Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

File allocation table (FAT)

-



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

File allocation table (FAT)

- To support 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

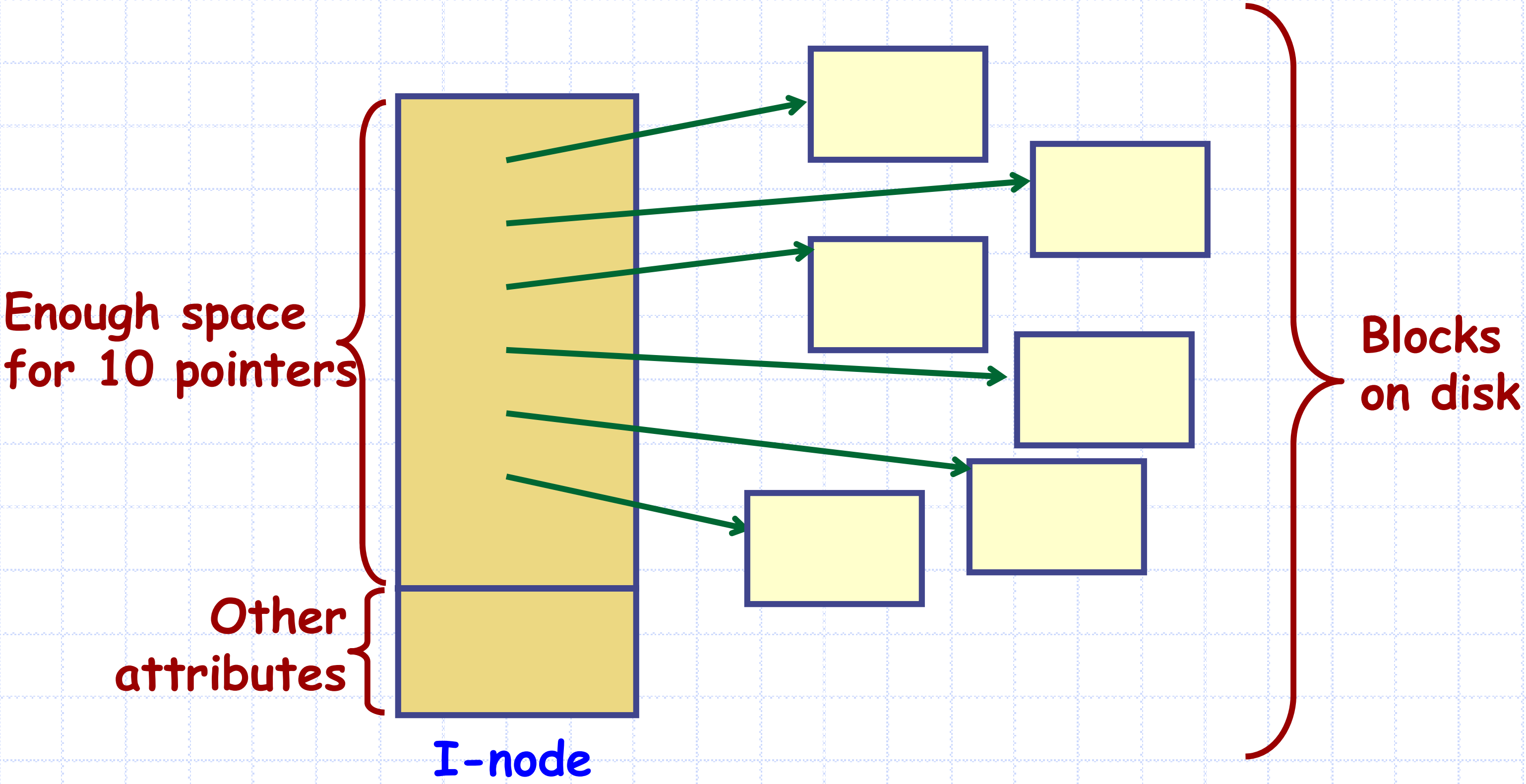
    1 KB = block size

    4 bytes = FAT entry size

    80 MB of memory used to store the FAT

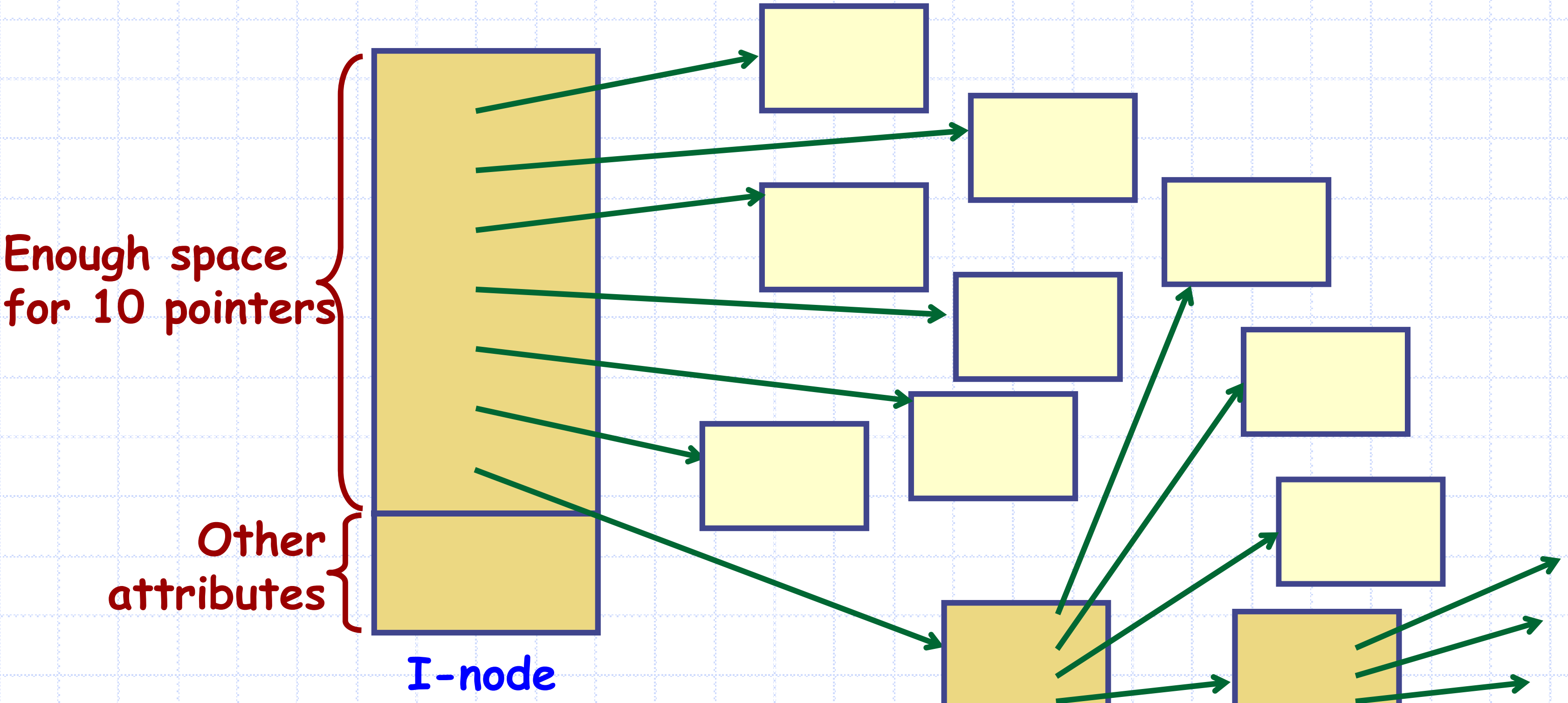
I-nodes

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



I-nodes

- If the index structure fills up, we allocate another inode that can contain additional block links
  - so in a way the I-node model uses an index based allocation, as well as a linked list allocation (for the index blocks)



Allocating blocks to files

- when a file is first created it will have a directory entry and maybe an index node (depends on file system type), but it will not have any data blocks

- as data is added to the file, we need to allocate data blocks
  - there is a file pointer that points to the current position in the file
  - if we try to write when the pointer is at the end of a block, the driver will detect this and allocate a new block

- *How do we choose the new blocks?*

Allocating blocks

- we need to maintain a collection of unallocated blocks

1. we could put them on a free-list (linked list)
   - this can exacerbate fragmentation, as we have no information about which blocks are adjacent

2. we can maintain a free-block map (disk status map)
   - this is the preferred method, as it takes very little space, and allows the OS to allocate contiguous blocks where ever possible

status	1	1	0	0	0	0	1	1	0	...	0	0
block #	0	1	2	3	4	5	6	7	8		n-1	n

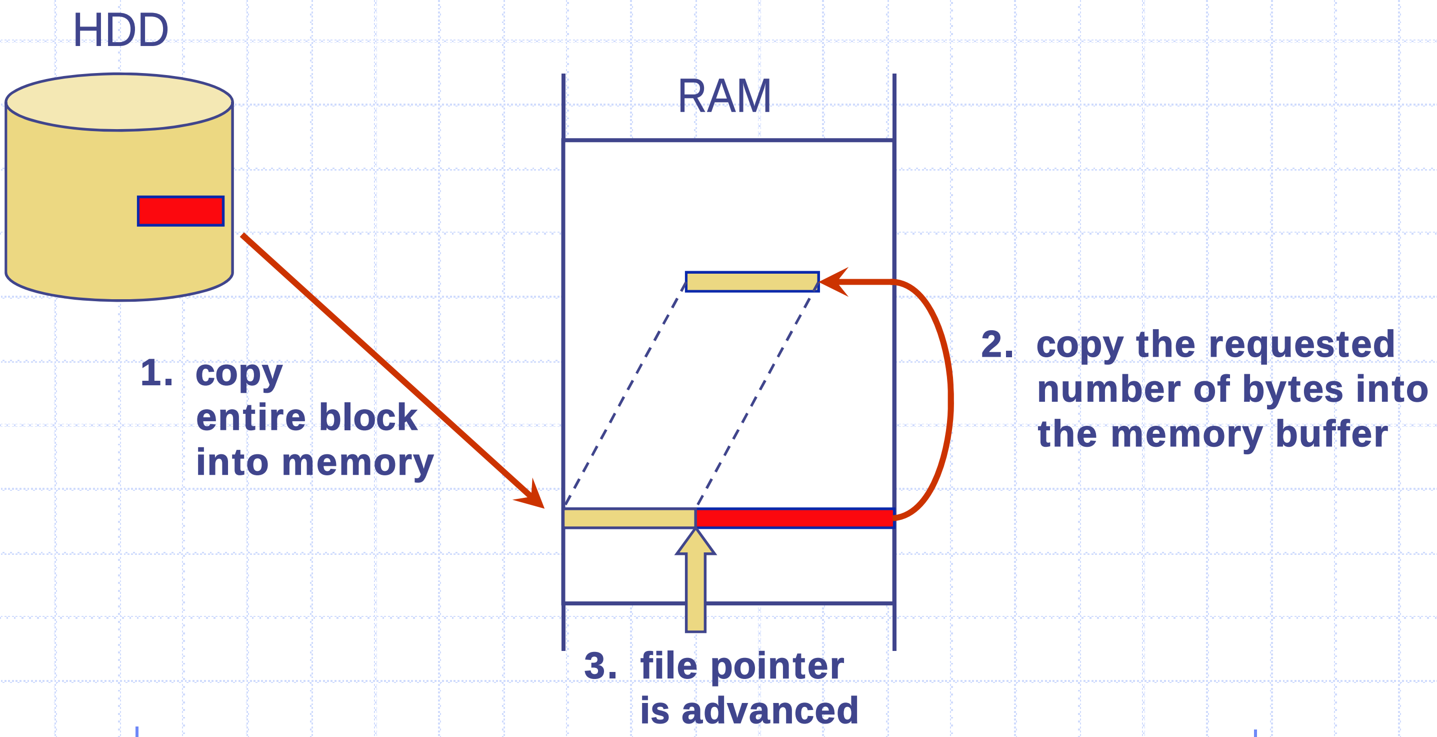
- this strategy uses very little space (a 1GB disk w/ 4KB blocks has 256KB disk map entries – which needs 32KB storage)

Allocating blocks

- Where do the blocks go once they are allocated to the file?
  - they are added to the data structure that represents the file
    - ie: linked to the list, or the index file or whatever
- The OS maintains a list of file descriptors, one per file, that point to all the open files in the system (more on this later)
- This means that a block is either connected to a file descriptor, or it is available (and marked as so in the disk status map)

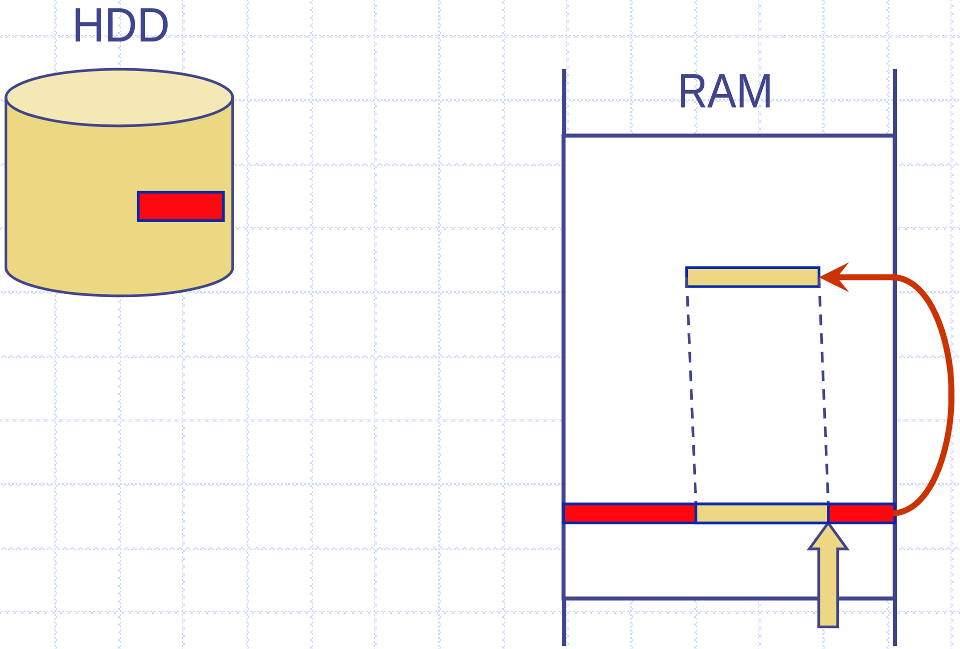
Read operations

- assume a command `read(fd, buffer, num_bytes)`
- assume also that this is the first read for this file
  - ie: the file pointer is at the beginning of the file
- first, the driver will determine the disk block to fetch, and …



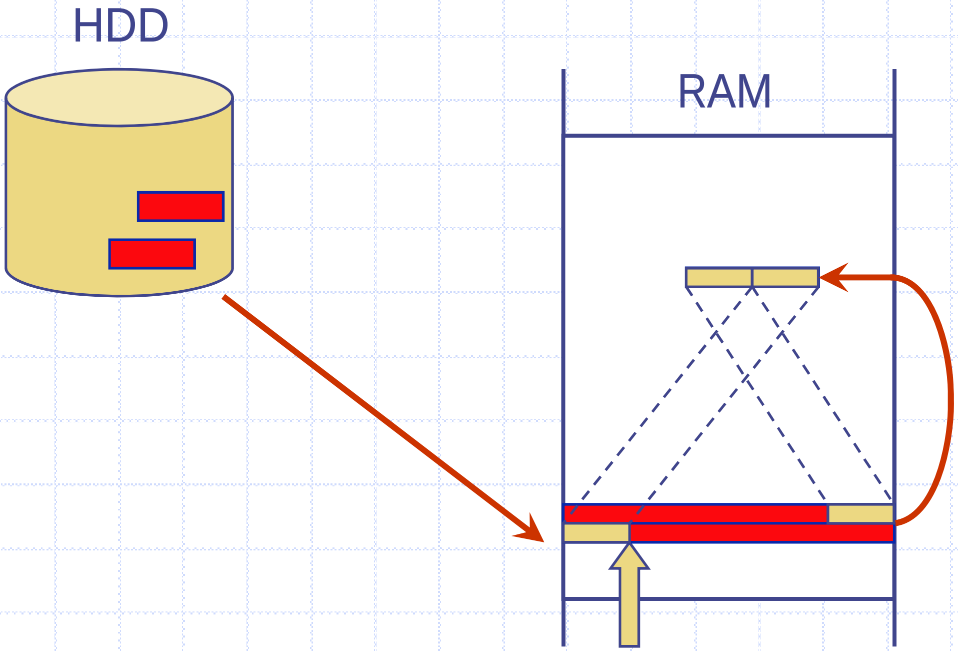
Read operations (2)

- the next read will start at the file pointer … this time we do not have to go to disk (in this example) because the block is still in memory



Read operations (3)

- a subsequent read will may require the next block to be loaded from disk …

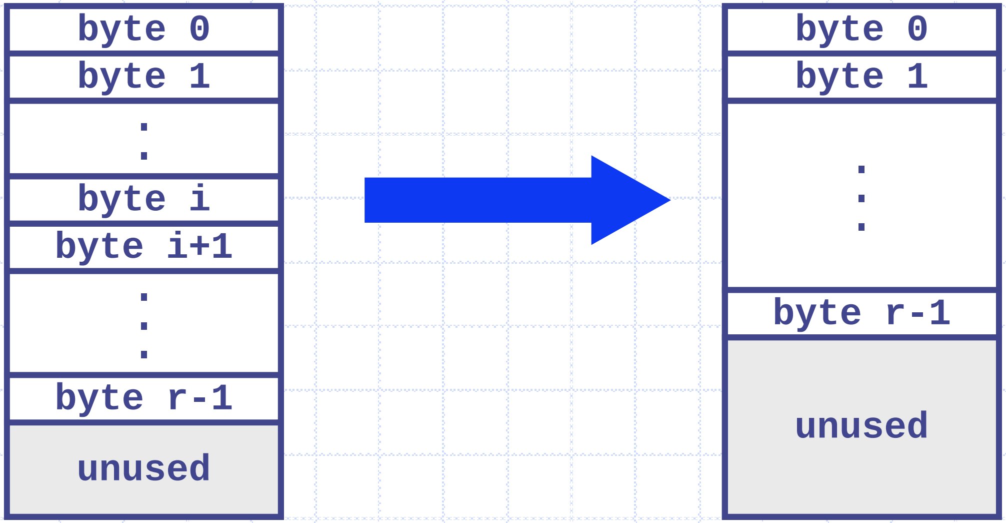


Write operations

- write operations are basically the reverse of the reads
  - each write over-writes an entire block on disk
- the OS maintains a copy of the entire block (the one the file pointer is referencing) in RAM
- the write operations update the RAM copy of the block until the file pointer moves to the next block
  - at this point (ie: the pointer moves to a byte in the next block)
    - write the block we just finished updating out to disk
    - read in the next block so the file pointer is pointing to a memory copy

Delete operations

- if we delete some bytes from a block, the memory copy needs to be "compacted" before it is written back to disk
- assume we have the block shown below, and we delete bytes i and i+1

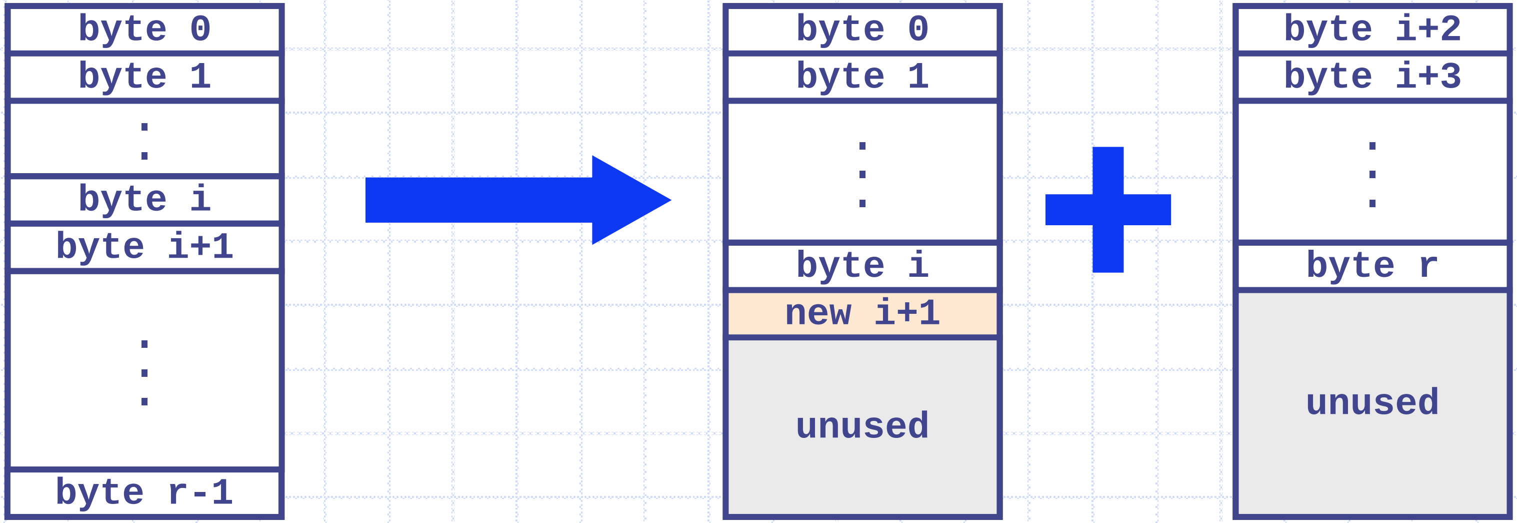


- if there are a lot of deleted blocks in a file, this will lead to a significant amount of internal fragmentation (inside blocks)
  - this means there are more blocks in a file than are actually needed – which slows down the entire system as we are spending too much time on IO

Inserting data into blocks

What happens if we try to "insert data"?

1. if the block is not full, we can "insert it" in the memory copy, and write to disk
2. if the block is full, we need to allocate a new block and split the data
   - assume we have the file shown below and we want to insert a new byte between i and i+1



- again, this leads to fragmentation and to an excessive number of allocated blocks (which can be spread all around the disk!)

Managing existing files

- when a file is created, a file descriptor is allocated for it
- the file descriptor information is saved with the file on disk
  - we call this the ***external file descriptor***
  - in unix this information is in the inode itself
  - in dos it is an entry in the FAT

- when a file is opened, the OS creates an ***open file descriptor*** – which is stored in the kernel
  - every open file has a descriptor in kernel memory

Opening files

- the generalized sequence of steps in the open operations are:

1. Locate the on-device (external) file descriptor
2. Extract info needed to access the file from the external file desc
3. Authenticate that process can access the file
4. Create an open file descriptor in primary memory
5. Create an entry in a “per process” open file status table
6. Allocate resources, e.g., buffers, to support file usage

Example File Systems

- Now let's consider some real file systems:
  - ISO 9660 file system
  - CPM file system
  - DOS file system
  - Unix file system

ISO 9660

- this is the file system used on CD-ROMs
- these are relatively simple file systems
  - designed for write-once media
  - no provision for keeping track of free blocks ... why?
  - blocks cannot be allocated to an existing file
  - file deletion is not supported

CD-ROM Media

- CD-ROMs do not have cylinders
  - data is all stored on one continuous track, which sort of forms a “spiral” on the disk
  - the “spiral” is divided into logical blocks
  - each block contains 2048 bytes of payload
  - tracks (files) are typically accessed sequentially, however it is possible to seek across the spiral
  - position of blocks is given in minutes and seconds
    - 1 sec == 75 blocks

Volumes and Sets

- ISO 9660 supports “CD-ROM Sets”
  - can include up to 2^{16}-1 CDs

- Each disk can be partitioned into logical volumes (partitions)

File System Organization

- Each file system is organized as follows:
  1. first 16 blocks (32MB) are unused in the standard
     - typically used to include a bootstrap program
  2. block 16 contains Primary Volume Descriptor
  3. if there is more than one volume, additional blocks with the secondary volume descriptors will follow block 16
  4. the root directory follows the volume descriptors
     - contains an entry for each file
  5. the files as well as any subdirectories follow the root directory

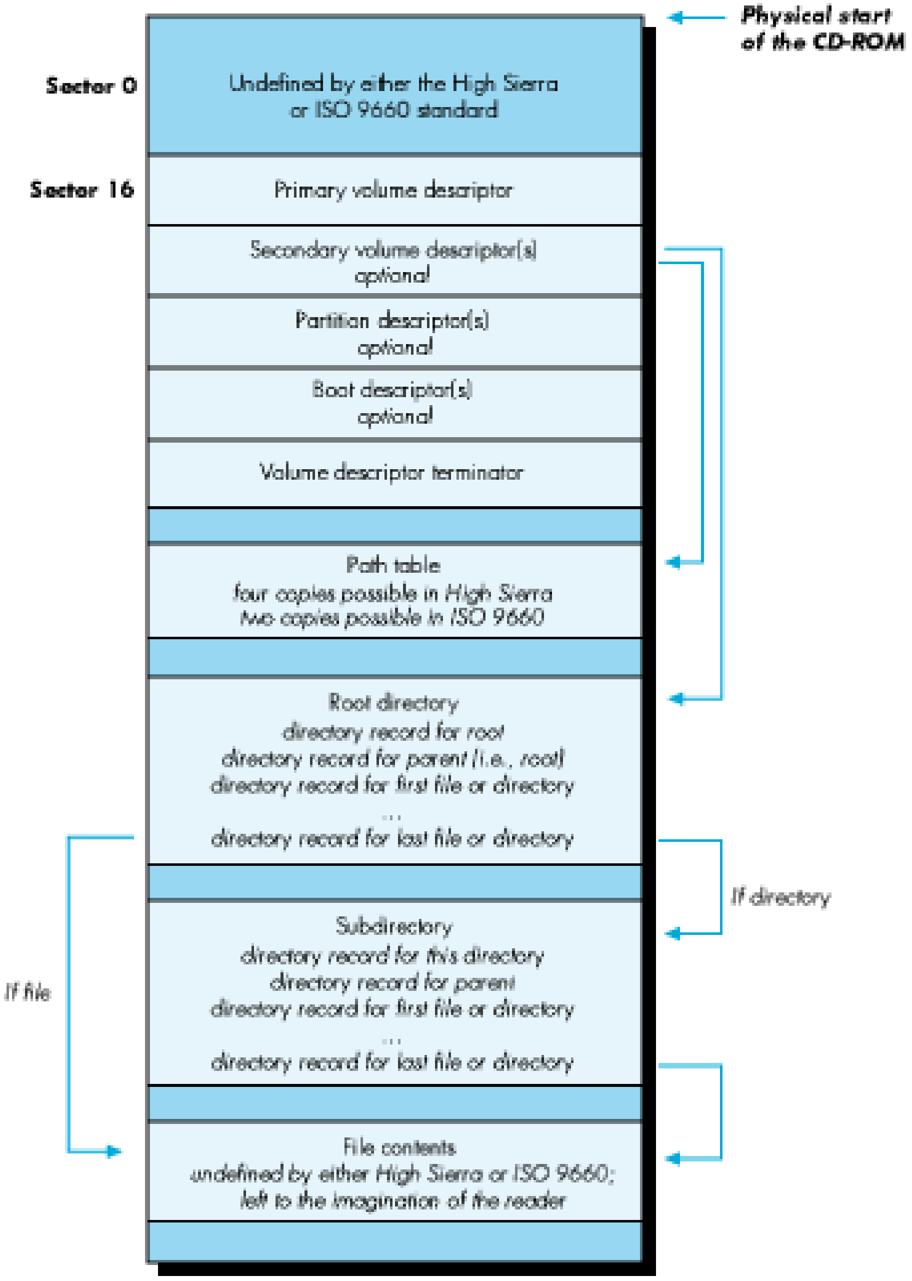

Physical start of the CD-ROM
Sector 0
Undefined by either the High Sierra or ISO 9660 standard
Sector 16
Primary volume descriptor
Secondary volume descriptor(s) optional
Partition descriptor(s) optional
Boot descriptor(s) optional
Volume descriptor terminator
Path table
four copies possible in High Sierra
two copies possible in ISO 9660
Root directory
directory record for root
directory record for parent (i.e., root)
directory record for first file or directory
...
directory record for last file or directory
If directory
If file
Subdirectory
directory record for this directory
directory record for parent
directory record for first file or directory
...
directory record for last file or directory
File contents
undefined by either High Sierra or ISO 9660;
left to the imagination of the reader

Image courtesy of: http://www.mactech.com

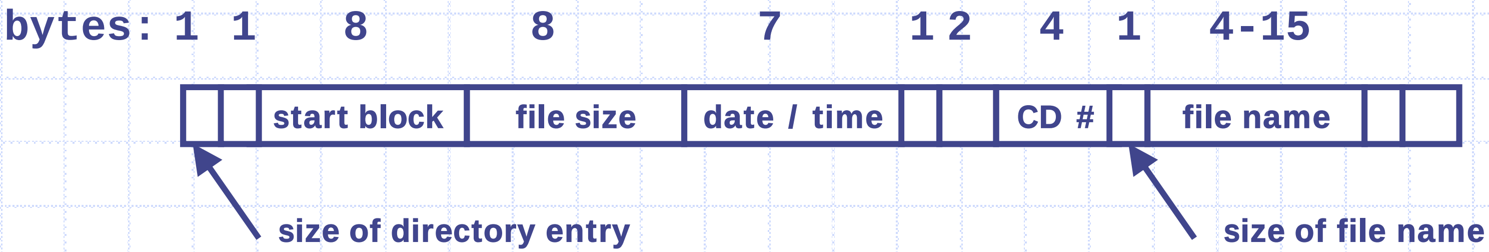
Primary Volume Descriptor

- contains information describing the file system, including:
  - logical block size
  - number of blocks
  - a directory entry for the root directory
    - points to the root directory
  - a path table
    - contains links to all subdirectories

Directory Entries

- each directory has a variable number of entries
- directory entries are variable length

- each entry contains the starting block and length of each file
  - note: files are always stored sequentially



bytes: 1 1 8 8 7 1 2 4 1 4-15

start block

file size

date / time

CD #

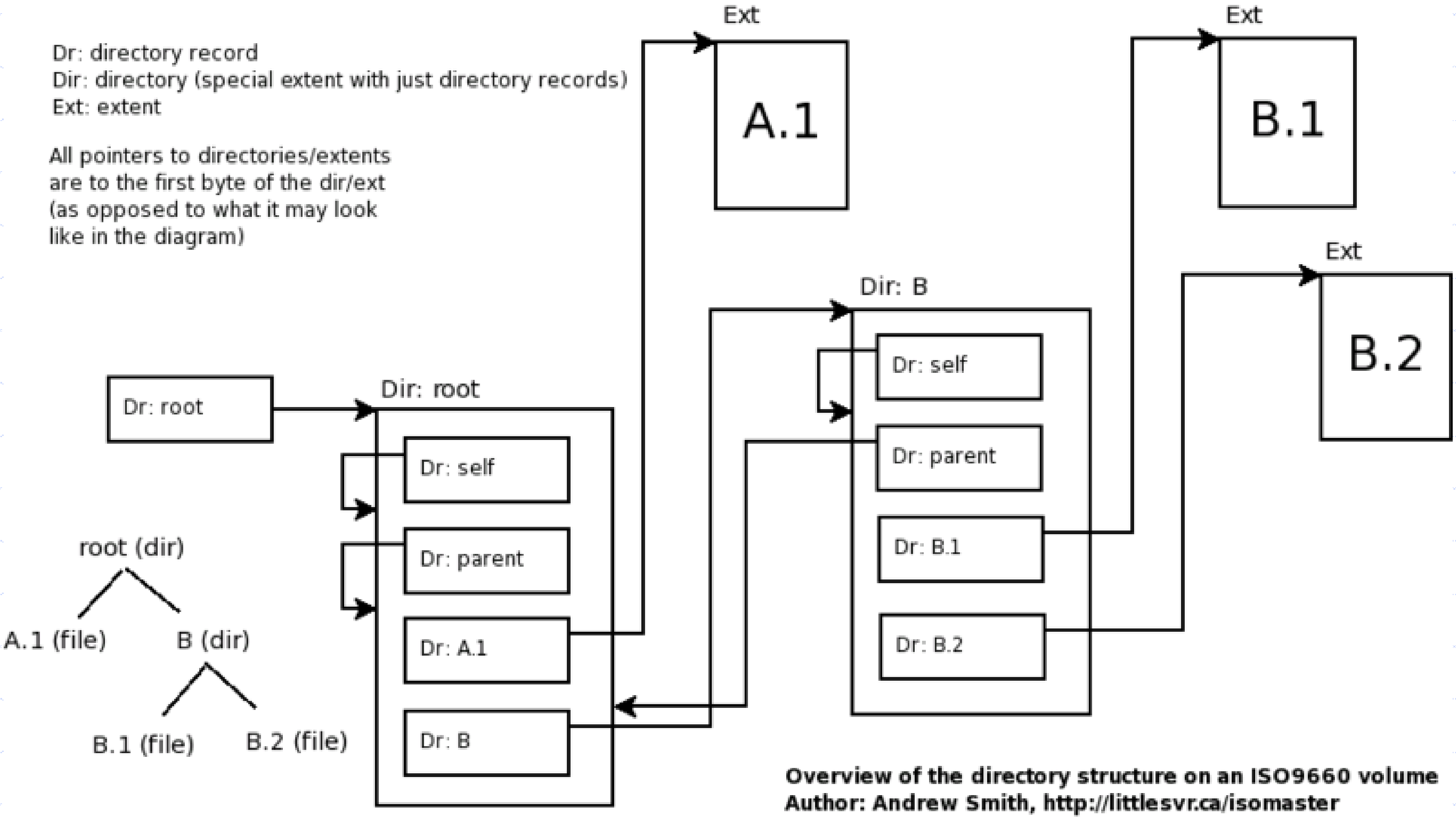
file name

size of directory entry

size of file name

- files an be stored on different CDs in the set

Directories (cont)



Dr: directory record
Dir: directory (special extent with just directory records)
Ext: extent

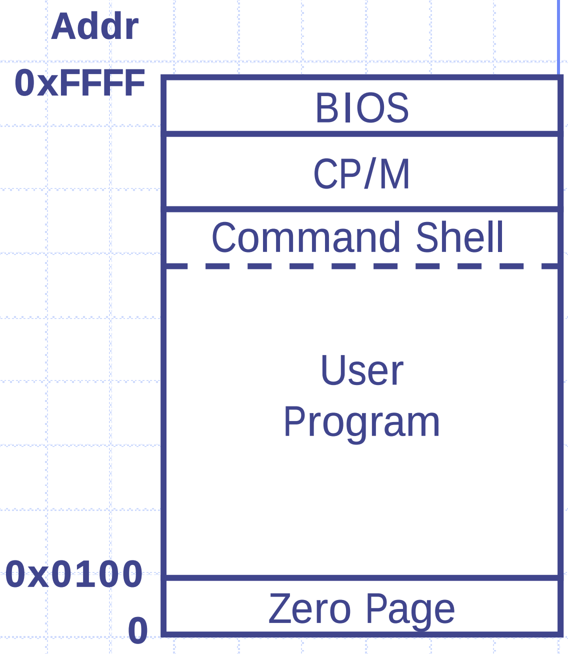
All pointers to directories/extents are to the first byte of the dir/ext (as opposed to what it may look like in the diagram)

Overview of the directory structure on an ISO9660 volume
Author: Andrew Smith, http://littlesvr.ca/isomaster

CP/M File System

What is CP/M

- popular OS from early 1980’s
- ran on 8080 and Z80 micro-computers
- ran quite well in 16MB of RAM
  - kernel uses 3584 bytes
  - command shell uses 2KB
- bios is loaded into RAM
  - all h/w interaction is via bios calls
- zero page holds:
  - interrupt vectors
  - system variables
  - current command buffer (holds arguments for current command)

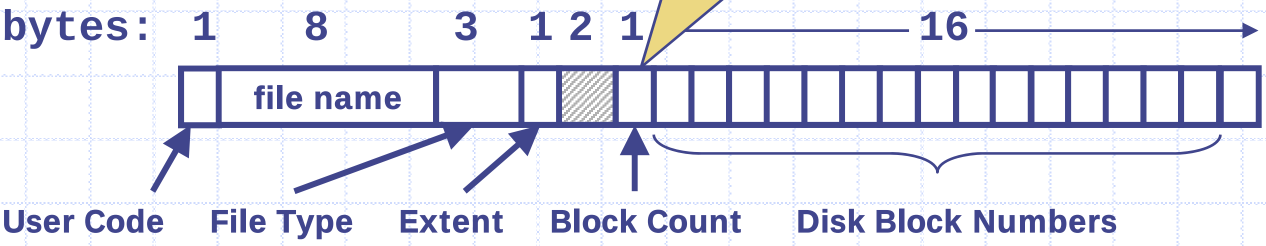


CP/M Directories

Note: file size is not stored ... just the block count.

Also, there is no end of file marker (this is left to the programmer).

- CP/M has exactly one directory
  - fixed size 32 byte entries
  - all files are listed in the directory
- data block size is 1KB



- user code
  - the name of the owner
  - only one person can log in at a time
  - you only see files that have your own user code
- file type
  - 3 byte file extension (like DOS)
- extent
  - files may need more than 16 blocks (floppy disk holds 180KB)
  - allow multiple directory entries for each file
  - extent is the sequence number for the multiple directory entries

CP/M File Operations

- free list of blocks is stored as a bitmap
  - requires 23 bytes for 180KB disk
  - created at system startup (kept in RAM)
    - directory is loaded into RAM and scanned; blocks not in files are added to the free list
  - discarded at shutdown
- directory is not kept in memory
  - read from disk when needed (such as on an open command)
  - has all the disk block numbers when the directory entry is found
- files are not stored in contiguous blocks on disk
  - the CPU is too slow
  - cannot process the interrupt and start reading the next block as fast as the disk spins
  - blocks in a file are interleaved to allow multiple blocks to be read on one revolution

DOS File System

DOS is a Better CP/M

- still no multi-programming
- more shell features
- more system calls

- OS is basically responsible for
  - loading programs
  - handling keyboard and screen
  - managing file system

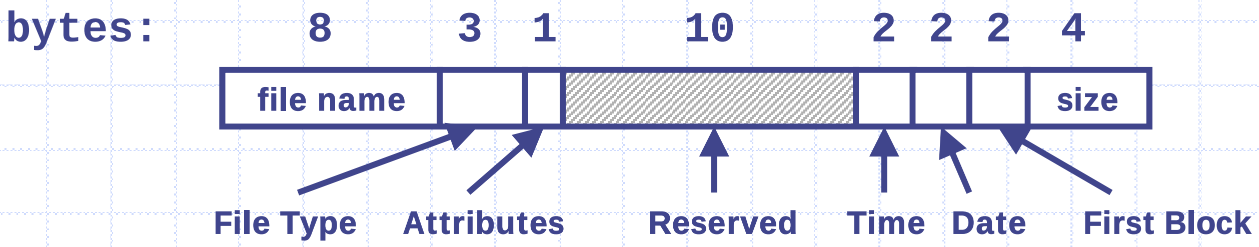
DOS File System

- patterned after CP/M
- uses 8+3 file names
- DOS 1 had a single directory
- DOS 2 introduced hierarchical file system
  - no links, so file system is a rooted tree
- no concept of users
  - logged in user has access to all the files
- supports hidden files (to prevent OS files from being deleted
- provides flags for read-only

Opening Files

- to use a file, programs must make **open( )** system call
  - returns a file handle (pointer to a descriptor)
- files can be specified by absolute or relative pathname
  - OS parses the path name and locates top directory in the name
    - it is either the root directory, who location is known, or the current working directory, whose location is also known
- **open( )** searches directory for next level directory, and so on, until it finds the directory entry for the file
  - the directory entry contains the first block number
- the first block number is used as an index into the File Allocation Table (in main memory)
  - programs can follow the chain of pointers in the FAT to access the entire file

DOS Directory Entries

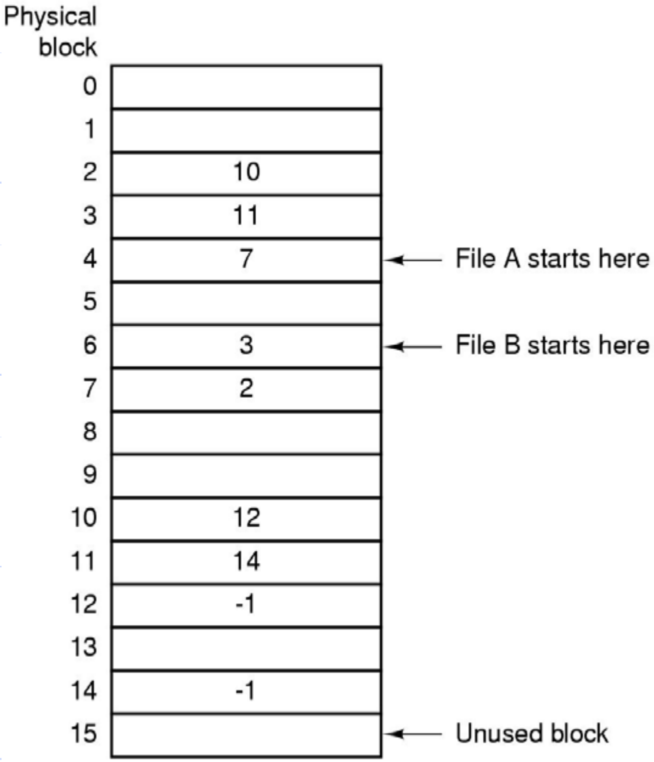


- attributes: read-only, archive, hidden, system file
  - archive is not used by the OS / can be read or set by application programs
- reserved: this space is unused
- date and time:
  - time can hold 65536 seconds (there are 86400 seconds in a day)
  - date = day (5) + minutes (6) + year-1980 (7 bits)
    - date is good until 2107
- first block
  - number of the first block in the file
  - this number is used as an index into the File Allocation Table

DOS FAT

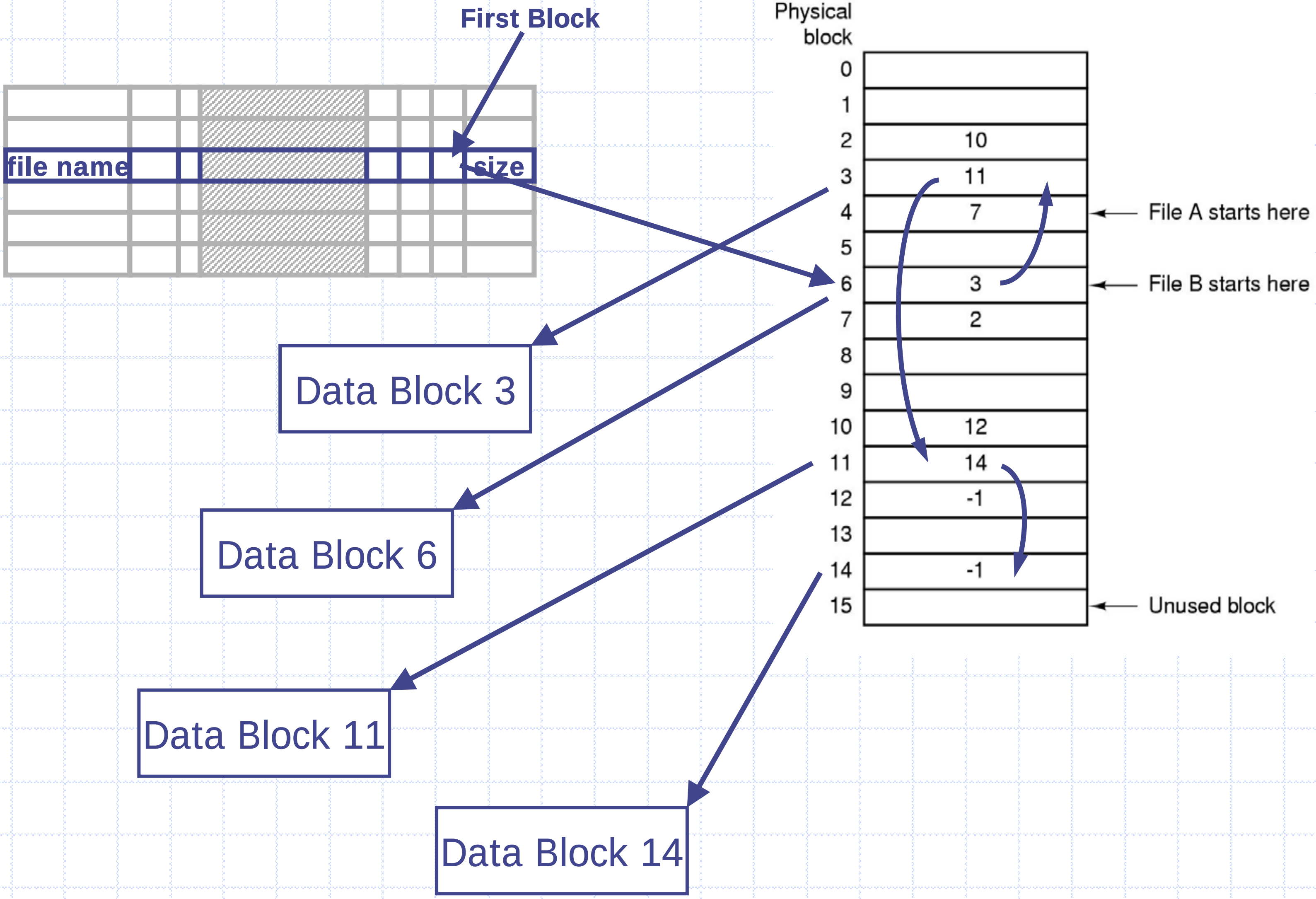
- FAT is a linked list of blocks
- each FAT entry contains
  - block number of next block in list
- -1 marks “last block in file”
- -2 denotes a free block

- FAT contains 64K entries (FAT-16)
  - loaded in main memory
  - every block in the partition has an entry in the FAT
  - block size is set for each partition
    - 512B, 1BK, 2KB, 4KB, 8KB, 16KB, or 32KB are allowed



Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

Putting it all together ...



First Block

file name

size

Physical block	
0	
1	
2	10
3	11
4	7
5	
6	3
7	2
8	
9	
10	12
11	14
12	-1
13	
14	-1
15	

File A starts here

File B starts here

Unused block

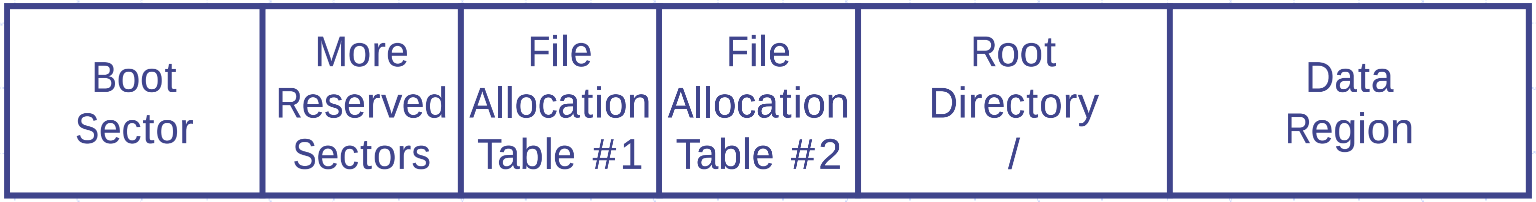
Data Block 3

Data Block 6

Data Block 11

Data Block 14

FAT Partitions



Boot Sector

More Reserved Sectors

File Allocation Table #1

File Allocation Table #2

Root Directory /

Data Region

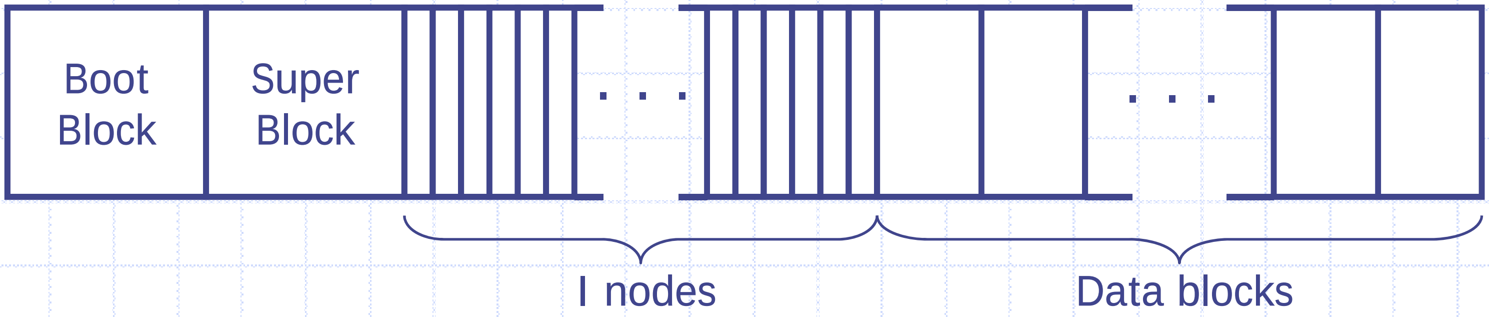
- boot sector
  - contains all he file system attributes such as sector size, number of tracks etc
  - also contains a jmp instruction to boot loader in case this partition is set as default boot partition
- more reserved sectors
  - contains the boot loader code (if partition is bootable)
- FAT region
  - contains two copies of the FAT table (for redundancy)
- Root dir
  - the directory table for teh / directory
  - there are a fixed number of entries in the / directory
- Data region
  - data and directory blocks for all the files in the system

Unix File System

- a Unix file is a stream of 0 or more bytes containing arbitrary information
- no distinction between types of files
- file names up to 255 chars, special chars allowed in file names
- as in CP/M & DOS, files can be referenced
  - by absolute path (fully qualified pathname, starting at root (/)
  - by a relative path, starting at current working directory of the process

File Systems / partitions

- Unix maintains 1 file system for each partition
- The layout of the regions of the partition are:

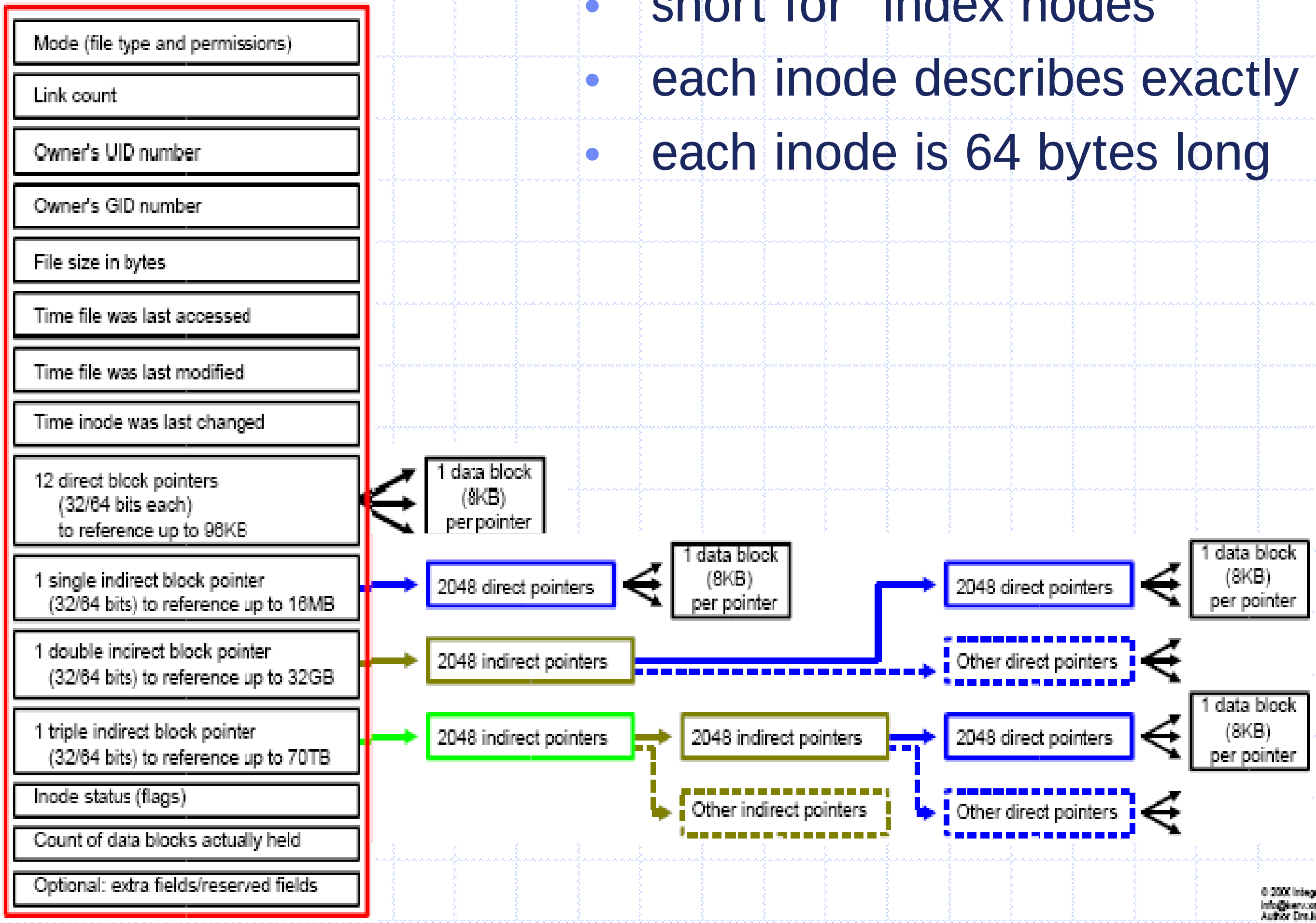


- Boot Block:
  - not used by the file system; may contain code to boot the OS

- Super Block:
  - file system parameters such as
    - number of i-nodes
    - number of disk blocks
    - start of free list

I Nodes

- short for “index nodes”
- each inode describes exactly one file
- each inode is 64 bytes long


Mode (file type and permissions)
Link count
Owner's UID number
Owner's GID number
File size in bytes
Time file was last accessed
Time file was last modified
Time inode was last changed
12 direct block pointers (32/64 bits each) to reference up to 96KB
1 single indirect block pointer (32/64 bits) to reference up to 16MB
1 double indirect block pointer (32/64 bits) to reference up to 32GB
1 triple indirect block pointer (32/64 bits) to reference up to 70TB
Inode status (flags)
Count of data blocks actually held
Optional: extra fields/reserved fields
1 data block (8KB) per pointer
2048 direct pointers
1 data block (8KB) per pointer
2048 indirect pointers
2048 direct pointers
1 data block (8KB) per pointer
Other direct pointers
2048 indirect pointers
2048 indirect pointers
2048 direct pointers
1 data block (8KB) per pointer
Other indirect pointers
Other direct pointers

Data Blocks

- blocks follow inodes on disk

- all files and directories are made up of data blocks

- blocks do not need to be contiguous on disk

- block size varies by implementation
  - originally 1 block == 1 sector
  - increased to 1KB in 4.0BSD
  - increased to 8KB for Unix Version 7

File Descriptors

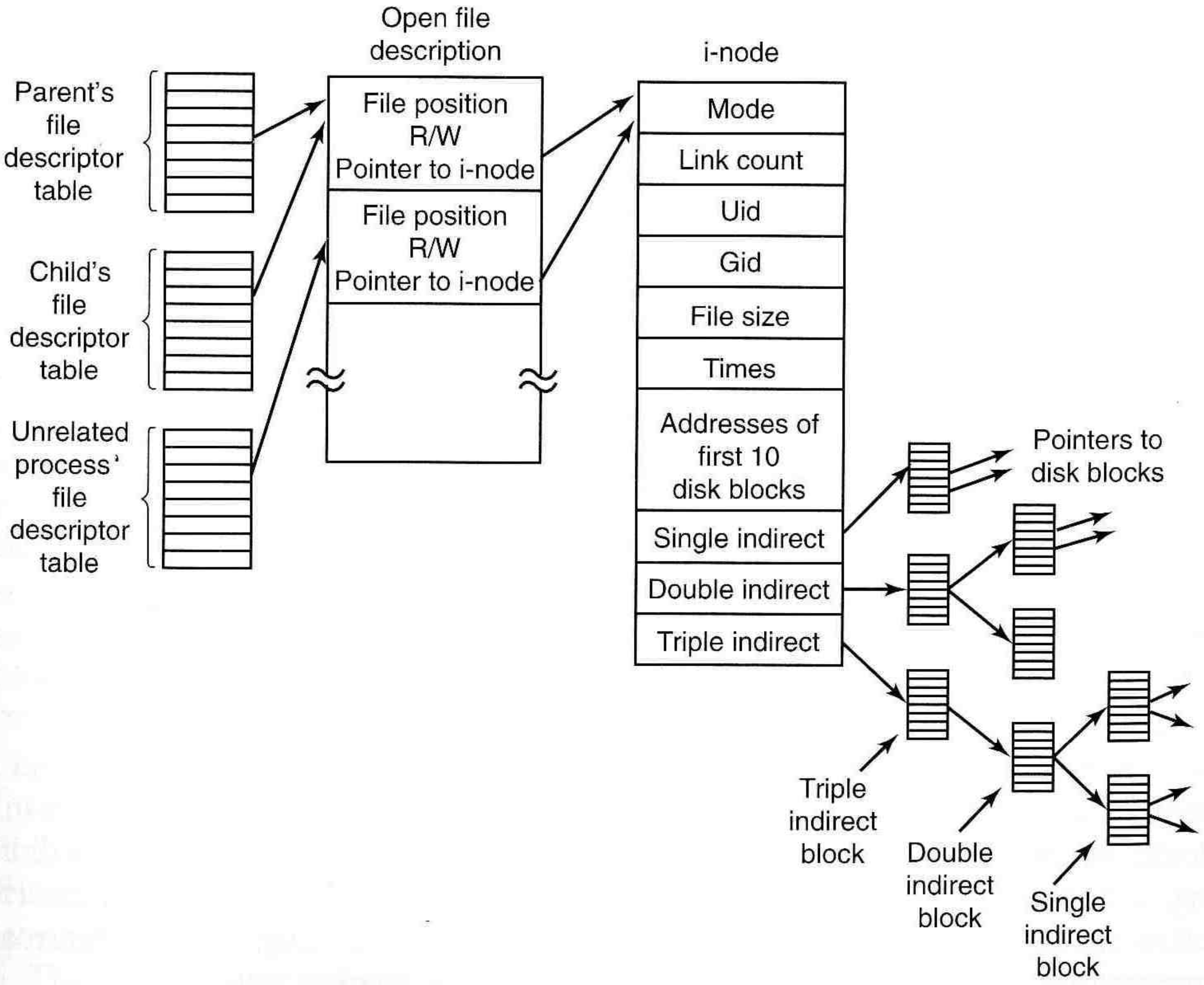


Figure 10-33. The relation between the file descriptor table, the open file description table, and the i-node table.

Unix Directory format

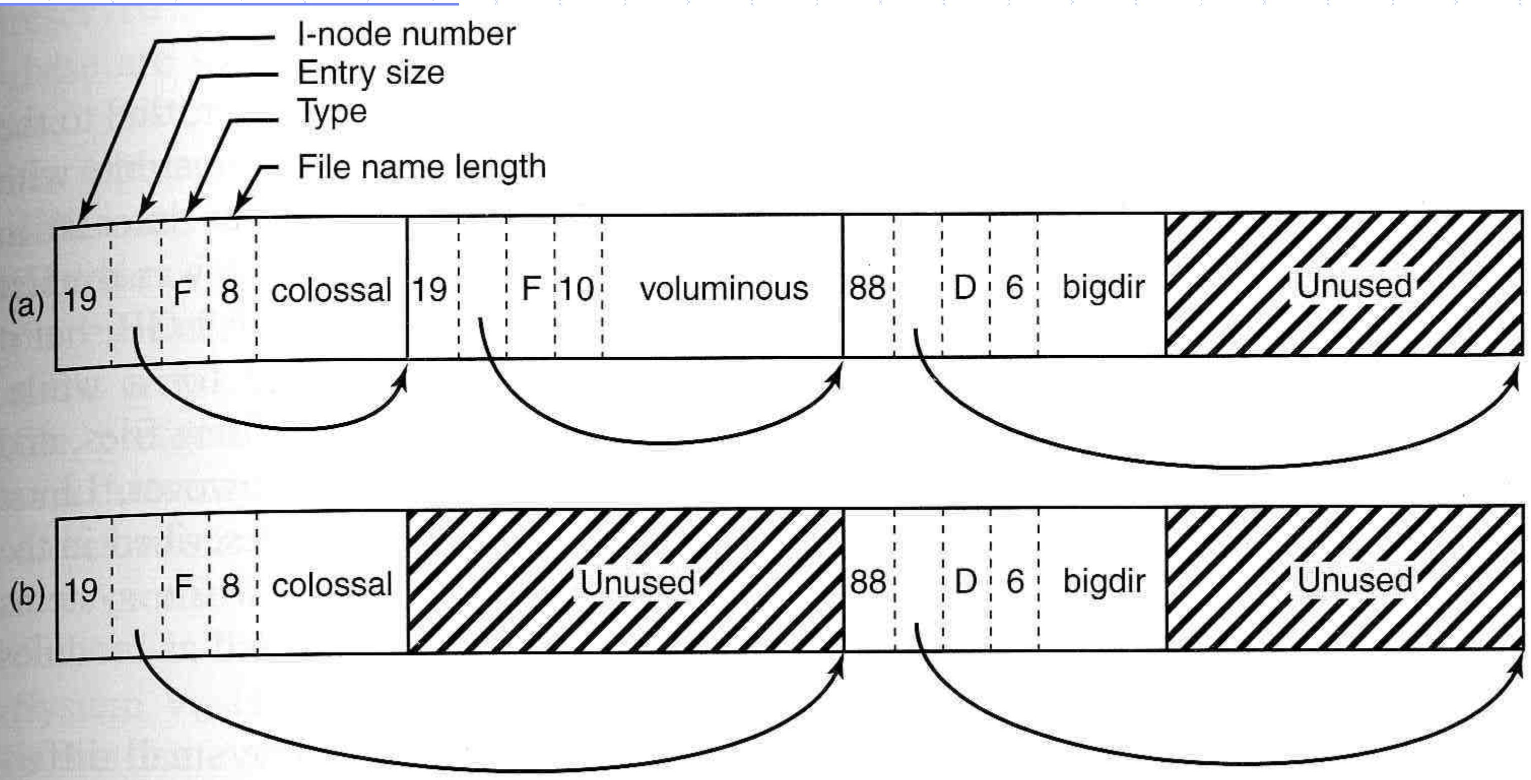
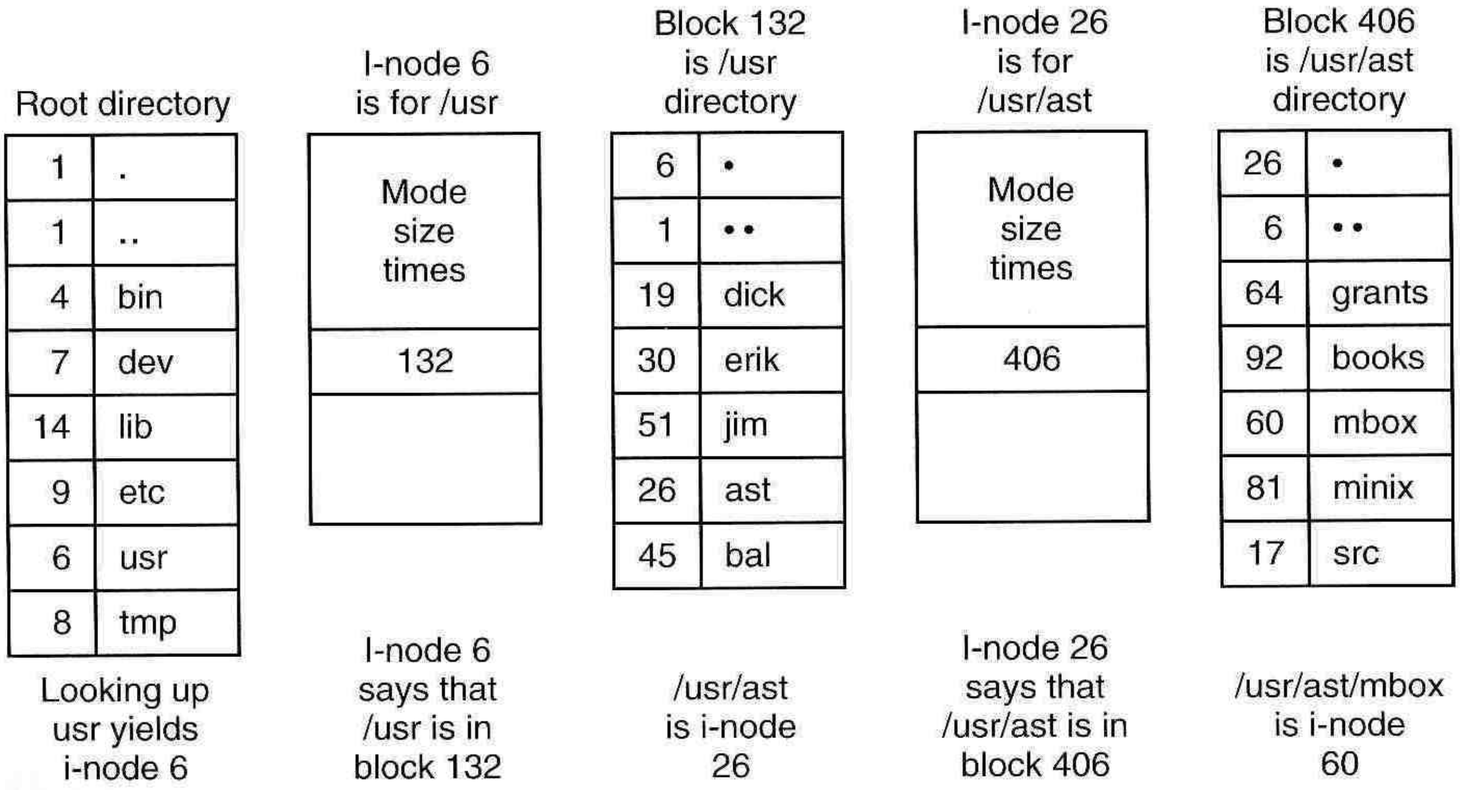


Figure 10-34. (a) A BSD directory with three files. (b) The same directory after the file *voluminous* has been removed.

Unix Directory (file lookup example)



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

Figure 6-39. The steps in looking up */usr/ast/mbox*.

The End