CMPT 300

Introduction to Operating Systems

File systems

Outline

- Files
- Directories
- File system implementation

File System

- Physical reality
  - Physical sector is unit of storage
  - Block oriented
  - No protection among users of the system
  - Data might be corrupted if machine crashes
- File system model
  - File is a unit of storage
  - File is a sequence of bytes
  - Users protected from each other
  - Robust to machine failures

What is a ‘file’?

- A file is
  - an abstraction to describe stored information.
  - A logical unit of information created by a process.
- A file has (from users point of view)
  - A name (usually conveys meaning about the contents of the file)
    - May have an extension to denote the type of contents or associated application (that created or uses the file).
  - Stored information
  - A size
  - Information on ownership

File systems

- Files should be
  - Persistent
    - Exists regardless of if processes are using it
  - Shareable between processes
  - Have a consistent and clearly defined structure (how they are stored)
    - This is important to the OS that manages the files

File naming

- Naming rules are dependent on the OS.
- Naming rules specify
  - Characters legal within the name
  - Maximum / minimum number of characters
  - Whether name is case sensitive
  - Structure of name (are there extensions, etc.)
  - Are file names fixed length or variable length
  - …

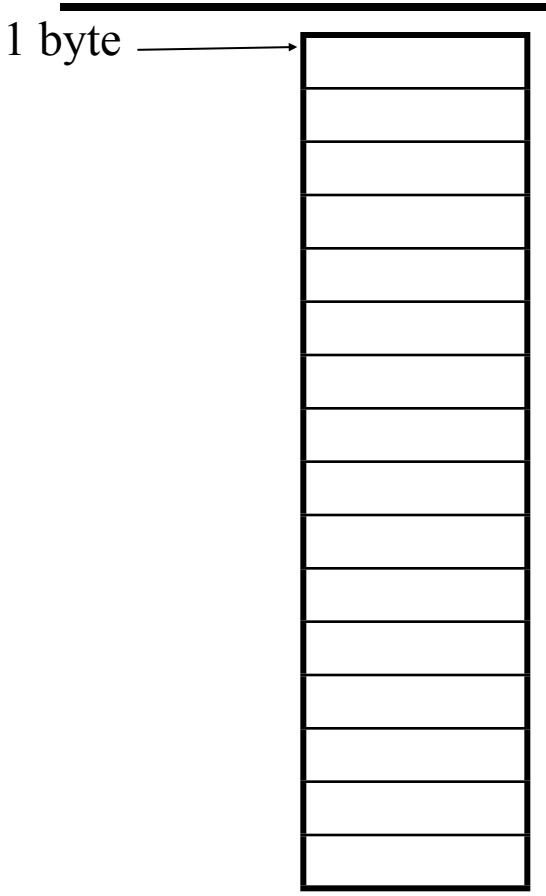
Some typical file name extensions

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

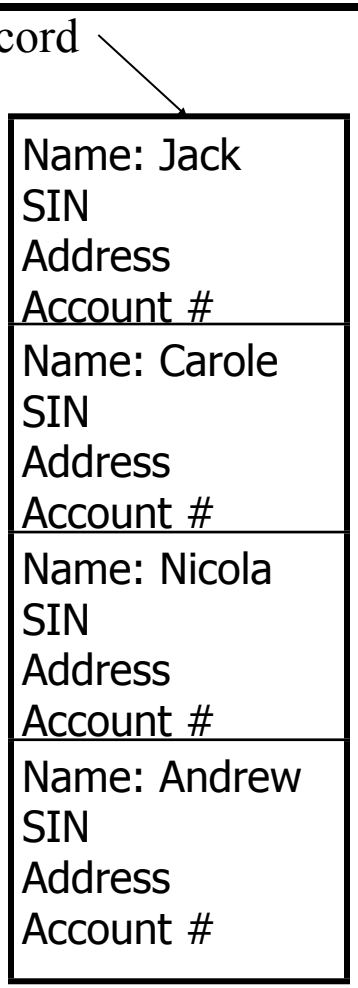
File Structure

- An unstructured sequence of bytes
  - Most widely used, e.g., UNIX and Windows
  - User programs impose meaning of files
- A sequence of fixed-length records
  - Records have internal structure
  - Read/write in records
  - Not used in any current general-purpose system
- A tree of records
  - Search records by keys
  - Used on some large mainframe computers

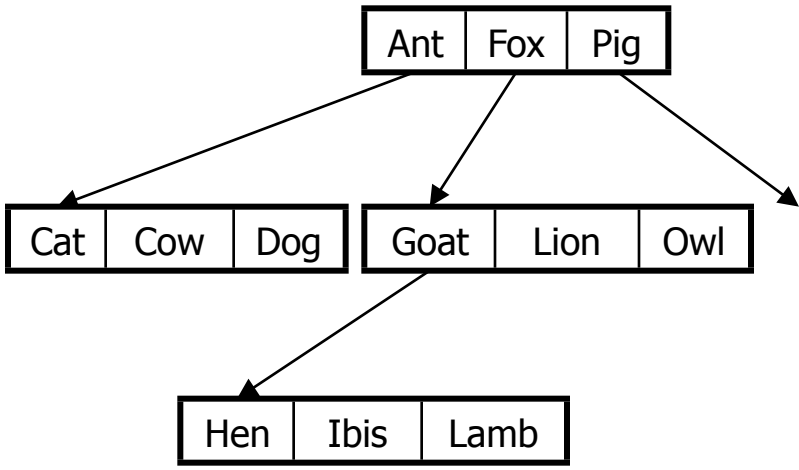
Examples of File Structures



Byte sequence



Record sequence



Tree

File types

- Regular files – store user's information
  - ASCII files (text file): lines of text
    - Can be displayed and printed as is.
    - E.g., source code file (*.cpp, *.h)
  - Binary files: binary streams
    - Internal structure know to programs
    - e.g., Object file, executable code (*.o, *.exe).
- Directories – maintained by system
  - Maintaining the structure of the file system
- UNIX special files: modeling I/O devices
  - Character special files: serial I/O devices
  - Block special files: disks/block devices

File Access

- Sequential access
  - Read all the bytes in order from the beginning
  - Rewind if read again
- Random access files
  - Read the bytes/records by specifying positions
  - Applications: database, etc.
  - All the files are random access nowadays
- How to specify the starting point for reading
  - Use seek operation to set the current position
    - Roll forward/backward for *n* bytes

File Attributes

- File protection and access
- Flags control/enable some specific property
- Used in file with records having a key
- Time stamps
- Size

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	Read/write or read only
Hidden flag	Normal file or the file does not display in listings
System flag	Normal file or system file
Archive flag	The file has been backed up or not
Random access flag	Sequential access only or random access
ASCII/binary flag	ASCII file or binary file
Temporary flag	Normal file or file will be deleted on process exit
Lock flags	Unlocked or 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	**Data 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 was last changed**
Current size	**Number of bytes in the file**
Maximum size	**Number of bytes the file may grow to**

Operations on files

The most common system calls relating to files:

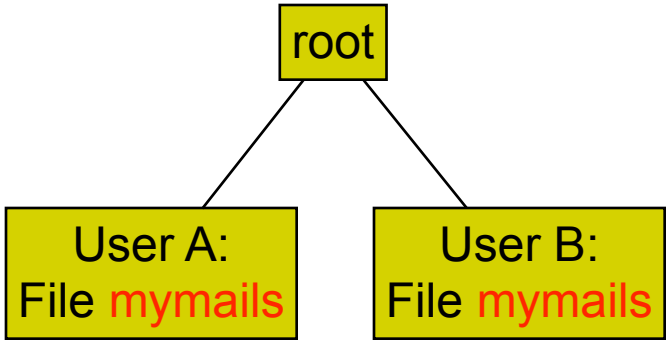
- Create
- Delete
- Open
- Close
- Read
- Write
- Append
- Seek
- Get Attributes
- Set Attributes
- Rename

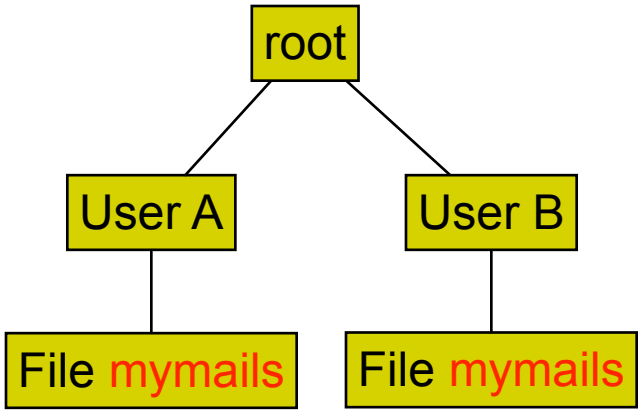
Directory

- A collection of files and/or other directories
- Also called “folder” on Windows machines.

Organization: Single-level / Two-level

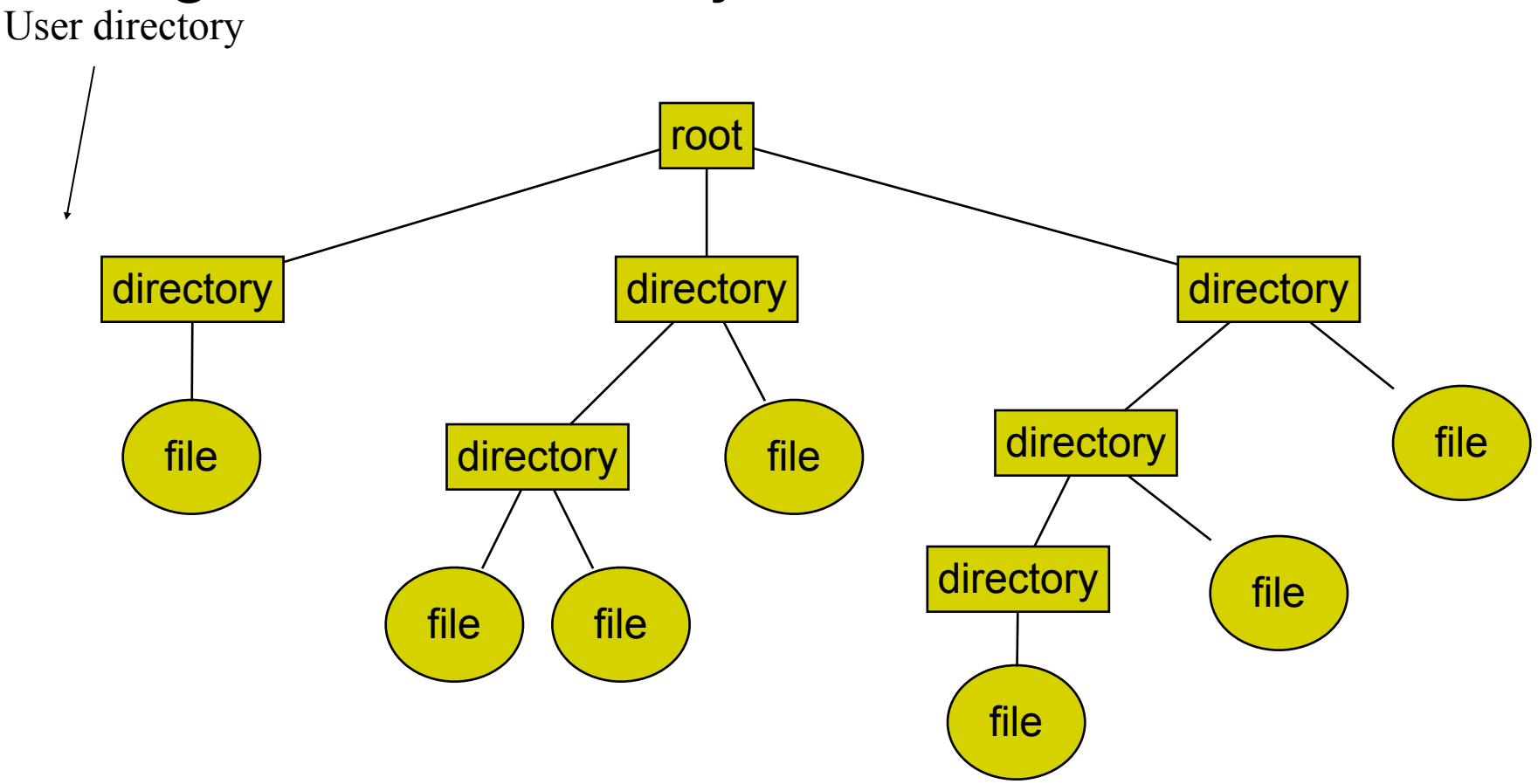
- Single-level: one directory for all the files
  - Not good for huge amount of files
  - Not good for multi-user system
- Two-level: user→directory
  - A large number of files from one user, inconvenient



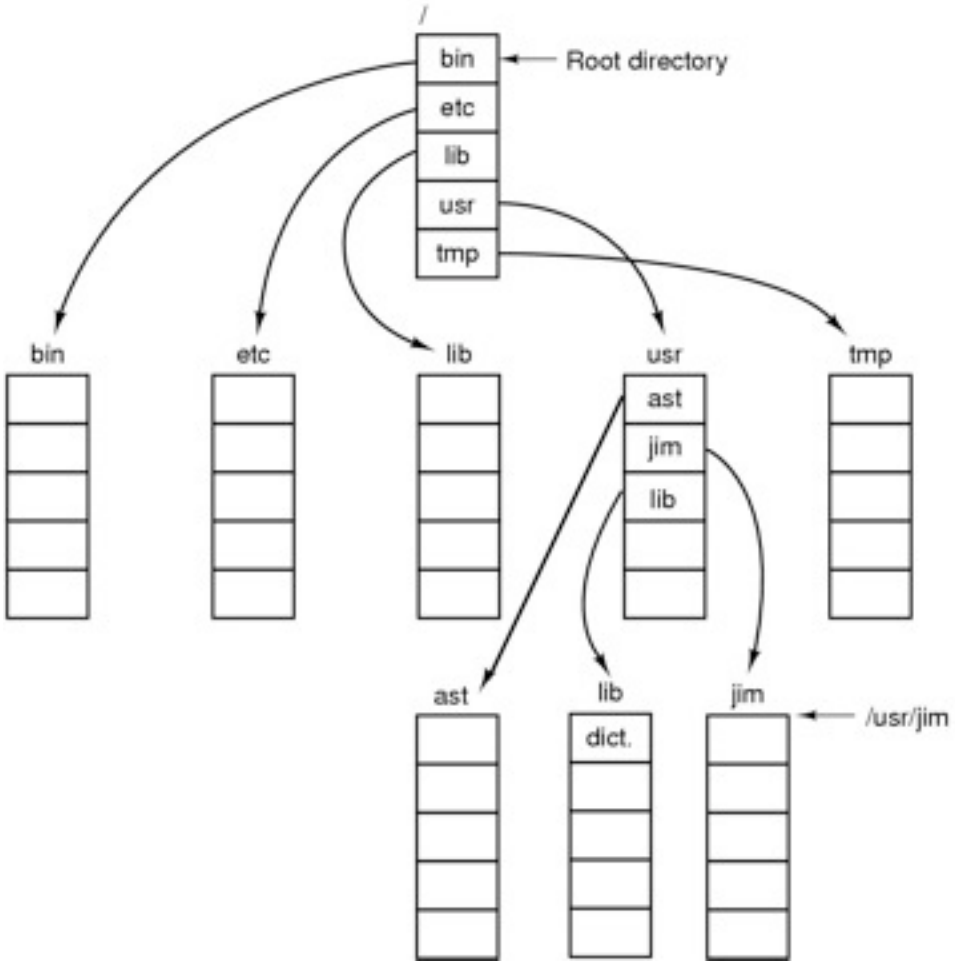


Hierarchical Directory Systems

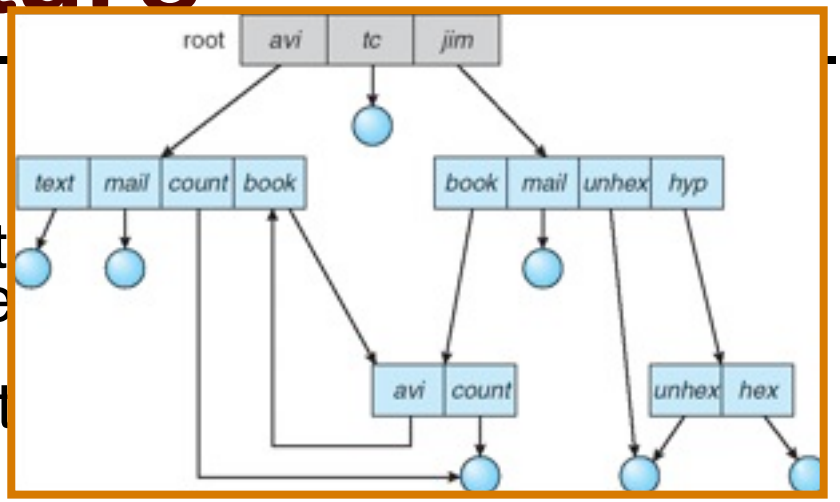
- A general hierarchy: a tree of directories



A UNIX directory tree.



Directory Structure



Not really a hierarchy!

- Many systems allow directory st
  an acyclic graph or even a (pote
- Hard Links: different names for t
  - Multiple directory entries point at
  - 'ln' command in UNIX
- Soft (symbolic) Links: "shortcut" pointers to other files
  - Implemented by storing the logical name of actual file
  - 'ln –s' command in UNIX

Name Resolution: The process of converting a logical name into a physical resource (like a file)

- Traverse succession of directories until reach target file

Path Names

- Mechanism to locate files
- Absolute path name
  - Path starting from the root directory
  - E.g., */usr/fran/mailbox*. '/' is path separator ('\" on Windows)
- Relative path name
  - Relative to the current working directory
    - E.g., if working directory is /usr/fran, then /usr/fran/mailbox = mailbox
  - Each process has its own working directory
  - Current directory "." and parent directory ".."
    - E.g., ../cindy/mailbox, ./mailbox

Directory Operations (UNIX)

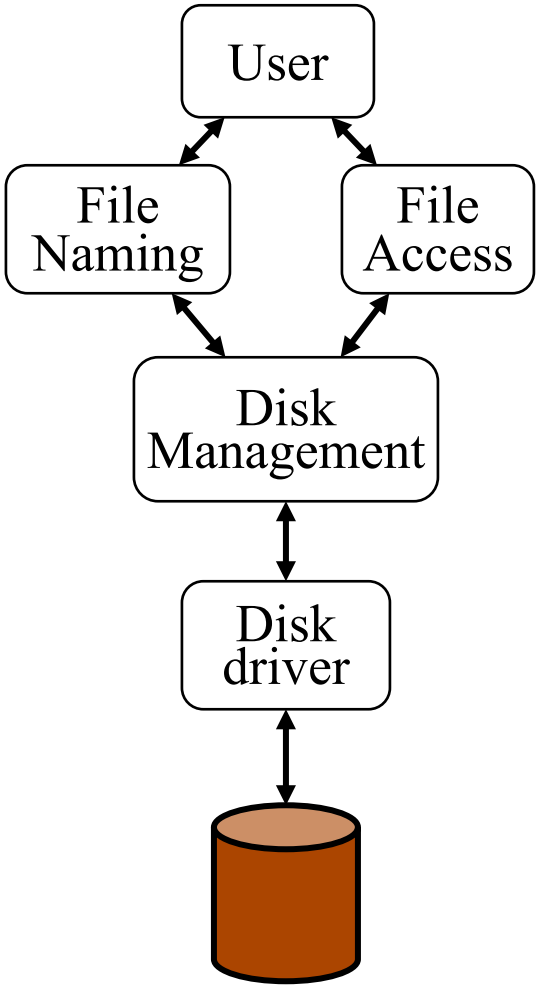
- Create: a directory is created
  - Empty except '.' and ".." entries
  - Delete, rename a directory
- Link (hard link)
  - Allow a file to appear in more than one directory
  - One copy of a file, multiple directory entries
- Unlink
  - A directory entry is removed
  - Link count == 0
    - Yes: remove the file (free the i-node and data blocks)
    - No: keep the file

File Management System

- System software to provide I/O services to users
  - Meet needs of user, access and organize files and directories, providing standardized interface
    - Each user can create, modify, delete their own files and directories and have controlled access to files of other users
    - Each user may control access by others to their files
    - Each user should be able to organize their files for efficient use, and refer to them by symbolic names
  - Verify validity of files, minimize lost/damaged data
  - Optimize throughput and system usage for I/O to files
  - Provide support for a variety of devices

File System Components

- Disk management
  - Arrange collection of disk blocks into files
- Naming
  - To locate file data, user provides to file system file name, not track or sector number
- Access Security
  - Keep information secure, don't leak, don't allow someone else to modify file
- Reliability/durability
  - When system crashes, may lose data in main memory (volatile), but want files to be durable



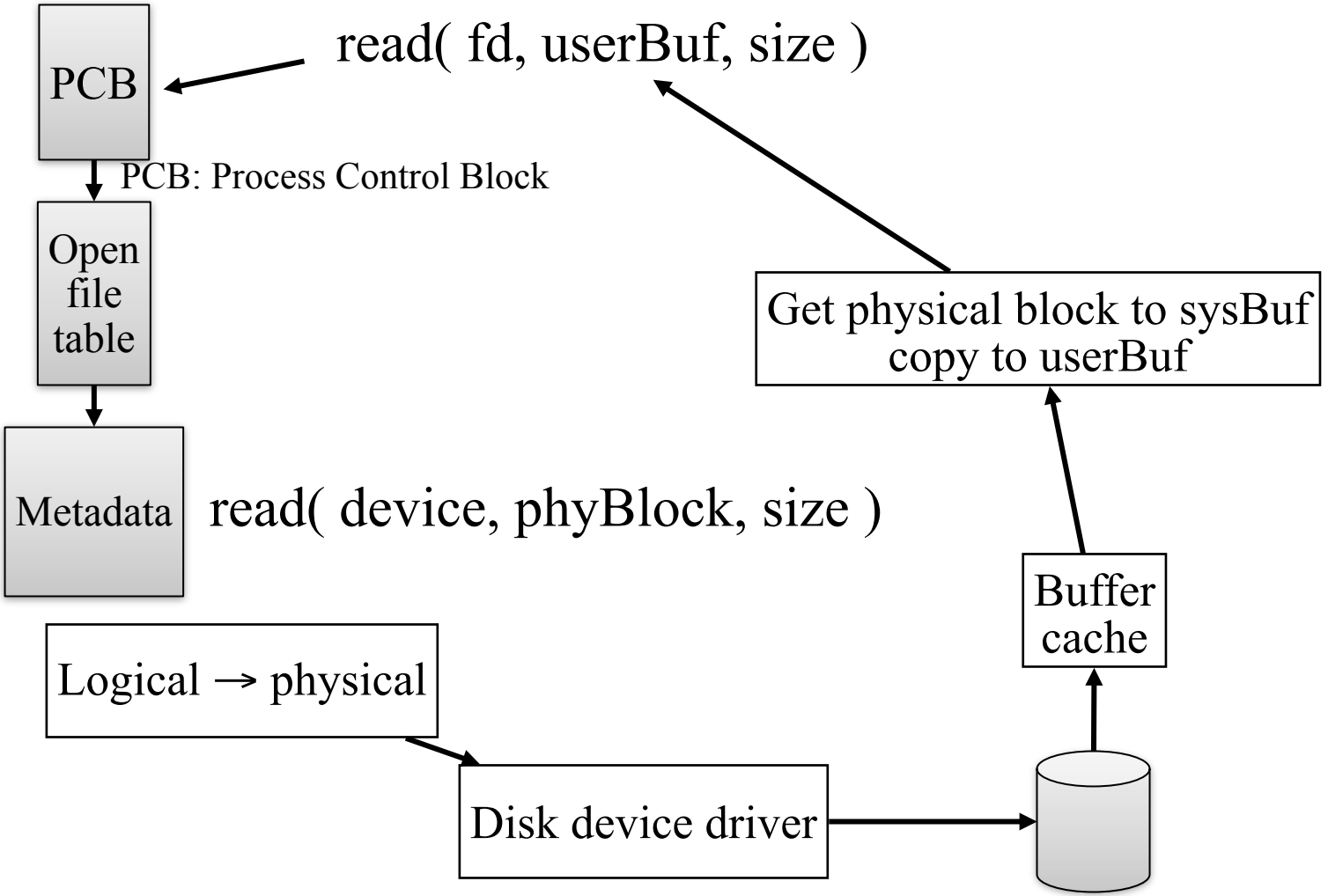
File Descriptor

File descriptor (fd)

- The user process side
  - Before reading or writing a file, user process has to call open (filename, mode): mode is either r, rw, w, …
    - open(…) checks if access is valid (Unix has fopen(…))
  - fd=open(…) returns a unique integer called the **file descriptor**
  - User process needs to use fd for all future operations on that file read(fd, buff) or write(fd, buff)
  - When user process done with that file, it calls close(fd)

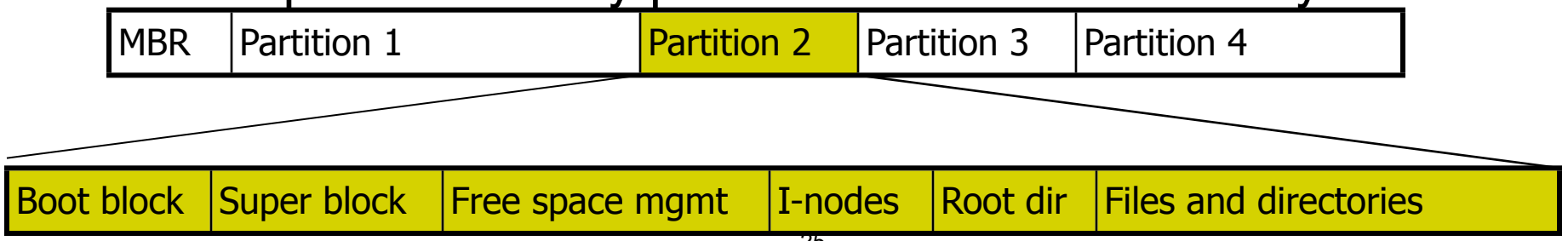
- The file system side
  - File system maintains an internal data structure for each open file, i.e., for each valid file descriptor. Created on open(…), deleted on close(…).
  - Open file table : system-wide list of file descriptors in use

Reading A Block



File System Layout

- Disk is divided up into several partitions
  - Each partition has one file system
- MBR – master boot record
  - Boot the computer & contain the partition table
  - Partition table
    - Starting & ending addresses of each partition
    - One partition is marked as active
- Within each partition
  - Boot block – first block, a program loads the OS
  - Superblock – key parameters about the file sys.



Implementing Files

- Key issue: how to keep track of which disk sectors go with which file?
  - E.g., block size= 512B, file size=2014B, so where are these 2014/514=4 blocks on disk?
- Many methods
  - Contiguous allocation
  - Linked list allocation
  - I-nodes
  - Each one has its own pros and cons

26

File systems Implementation Challenges

- Files grow and shrink in pieces
- Little *a priori* knowledge of this dynamism
- Several orders of magnitude in file sizes – smallest files are 0B large, largest are a few GB's-TeraBytes's (that's 1024 GB's)
- Need to overcome/hide/mask disk performance behavior
- Desire for efficiency
- Coping with failure (of devices, or by users)

Contiguous Allocation

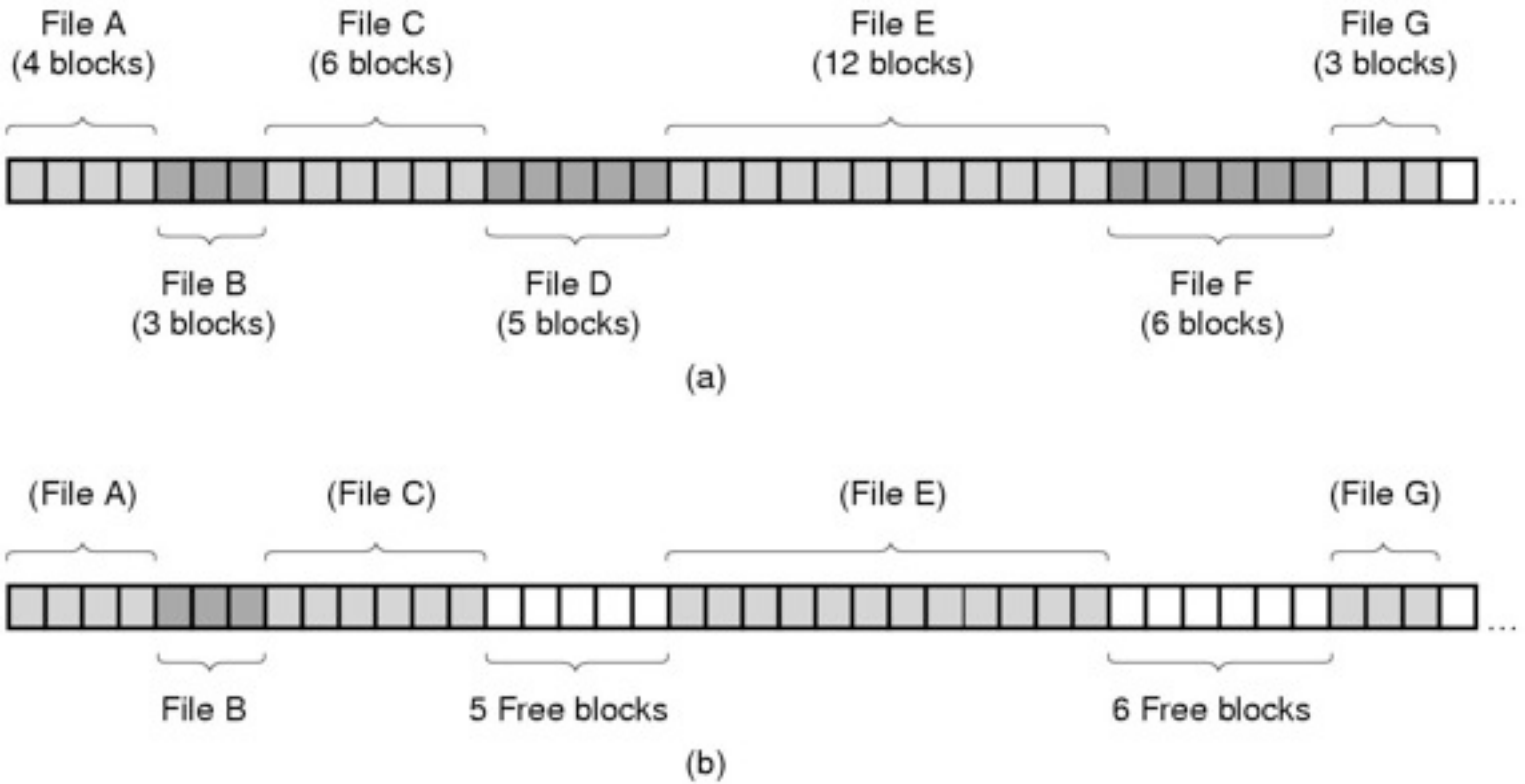
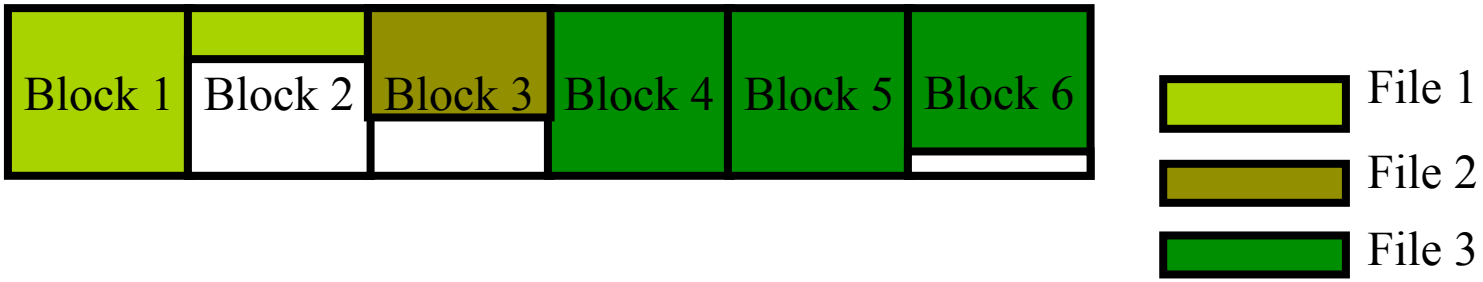


Figure 4-10. (a) Contiguous allocation of disk space for 7 files. (b) The state of the disk after files D and F have been removed.

Internal vs. External Fragmentation

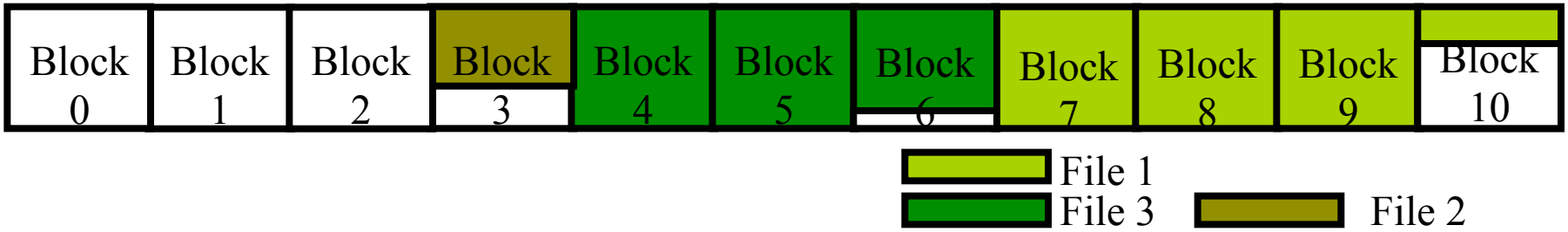
- ***Internal fragmentation:*** space wasted at the end of each block



- ***External fragmentation***: Free blocks are scattered throughout the disk, instead of forming a few large contiguous sets of free blocks

Enlarging a File

- What happens if file2 in Block3 grows by two blocks?
- Cannot allocate next contiguous block (block 4) because it is already in use
- To assure the file is contiguous
  - Find a large enough series of empty blocks to hold the extended file
  - Copy existing portion to this series of block, then append the new blocks (copy block 3 to block 1, then append new blocks, block 3 is now available)



Contiguous Allocation Pros and Cons

- Pros
  - Simple to implement
    - Each file has two numbers, starting address & length
  - Read performance is excellent
- Cons
  - Expensive to grow a file if relocation is necessary
  - Deletion of files may cause external fragmentation

Linked-List Allocation

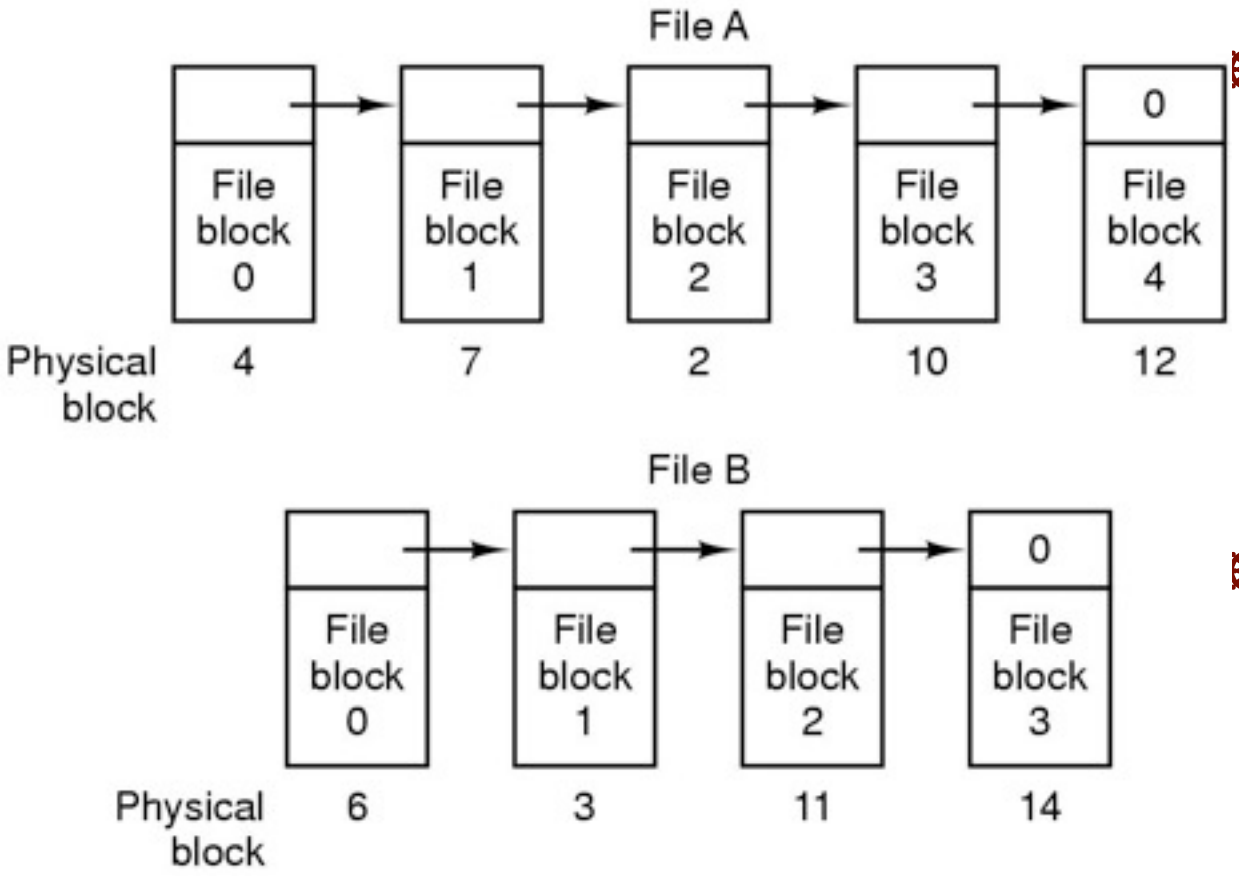
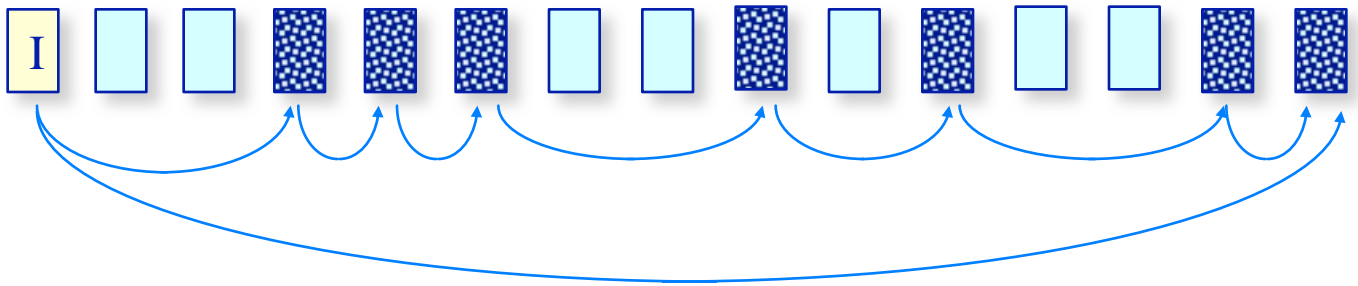


Figure 4-11. Storing a file as a linked list of disk blocks.

- File A uses disk blocks 4, 7, 2, 10, and 12, in that order
- File B uses disk blocks 6, 3, 11, and 14, in that order.

File Allocation Methods

Linked allocation



- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks
  - Pluses
    - Easy to create, grow & shrink files
    - No external fragmentation
  - Minuses
    - Impossible to do true random access
    - Reliability
      - Break one link in the chain and…

Linked List Allocation Pros and Cons

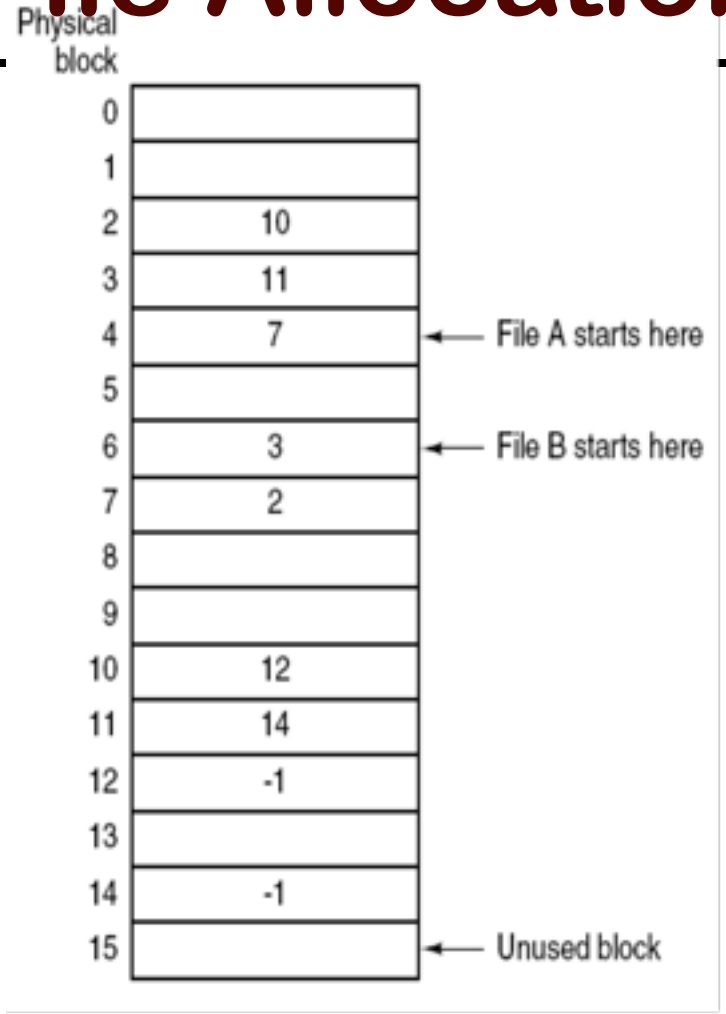
- Pros
  - No space is lost due to disk fragmentation (except for internal fragmentation in the last block)
    - Makes expansion/contraction of file simple
  - File is still referenced by a single pointer to its first block
  - Sequential access of file is efficient
    - Just follow the pointers!
- Cons:
  - Random access of file not efficient
    - Must chase pointers from the first block
  - The pointer takes up a few bytes, so data blocks no longer 2^N long
    - Less efficient for many programs that expect 2^N block sizes

File Allocation Methods

Linked allocation – File Allocation Table (FAT) (Win9x, OS2)

- Maintain linked list in a separate table
  - A table entry for each block on disk
  - Each table entry in a file has a pointer to the next entry in that file (with a special "eof" marker)
  - A "0" in the table entry ➔ free block
- Comparison with linked allocation
  - If FAT is cached ➔ better sequential and random access performance
    - How much memory is needed to cache entire FAT?
      - 400GB disk, 4KB/block ➔ 100M entries in FAT ➔ 400MB
    - Solution approaches
      - Allocate larger clusters of storage space
      - Allocate different parts of the file near each other ➔ better locality for FAT

File Allocation Table (FAT)



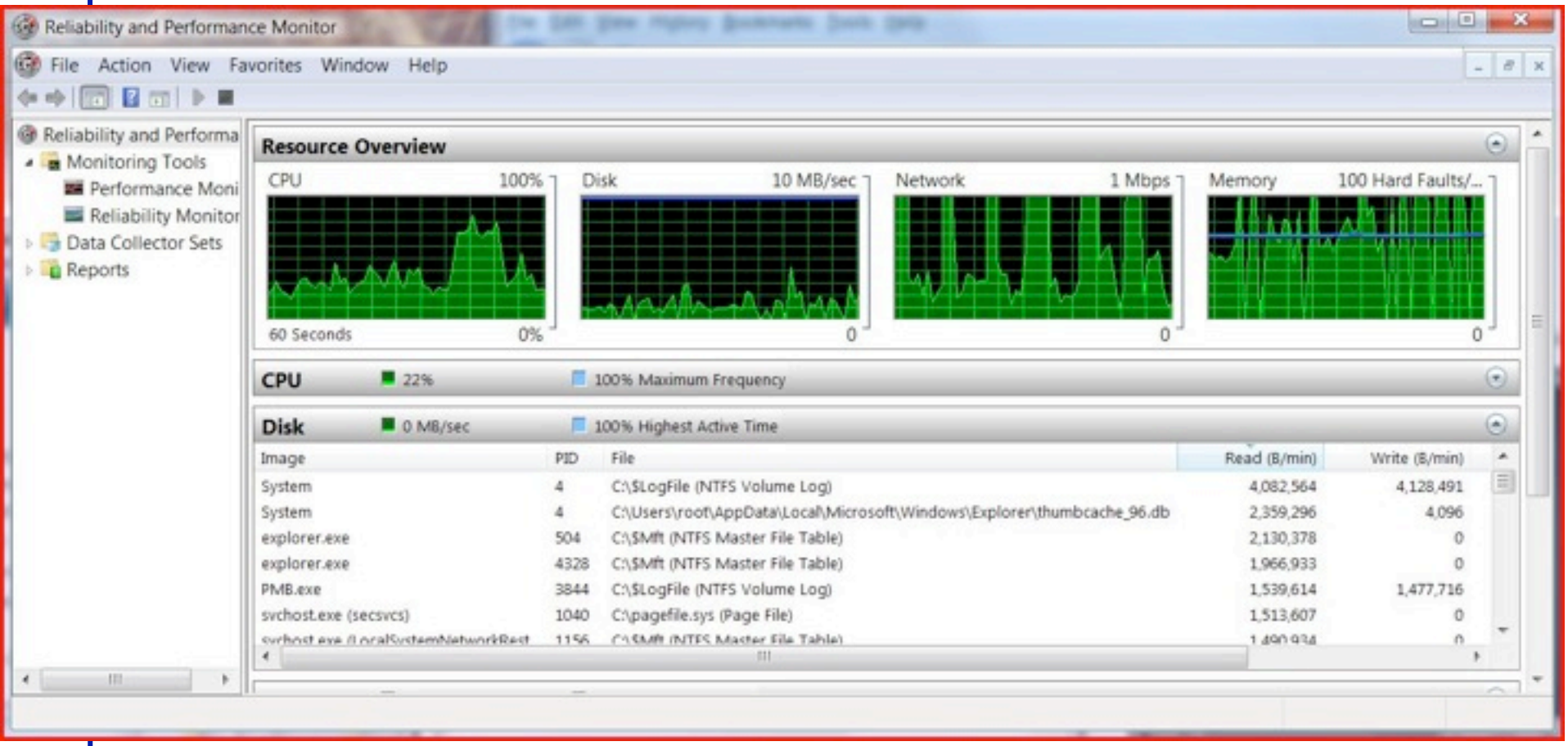
- Same example as before:
- File A uses disk blocks 4, 7, 2, 10, and 12, in that order
- File B uses disk blocks 6, 3, 11, and 14, in that order.
- Used by MS/DOS and early Windows

FAT Pros and Cons

- Pros:
  - No pointers in data blocks → data blocks are 2^N long
  - Random access is easy, since pointer-chasing is done on the table, no need to access disk blocks
- Cons:
  - Entire table must be kept in memory; can get large
    - With a 200-GB disk and a 1-KB block size, the table needs 200 million entries. Each entry has to be a minimum of 3 bytes. Thus the table will take up 600 MB of main memory

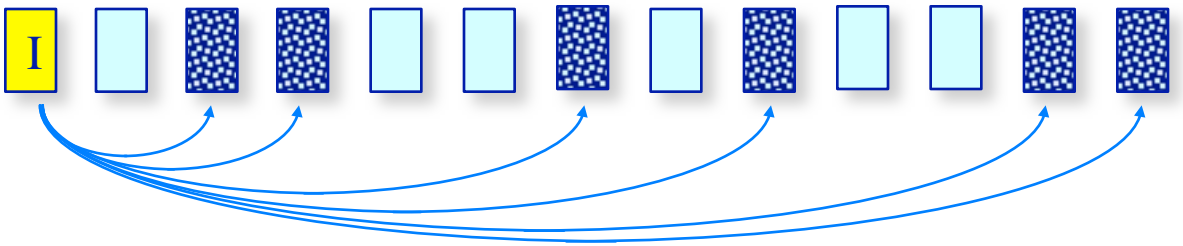
Vista reading the master file table

MFT contains a record for each file, inlines small files



File Allocation Methods

Direct allocation



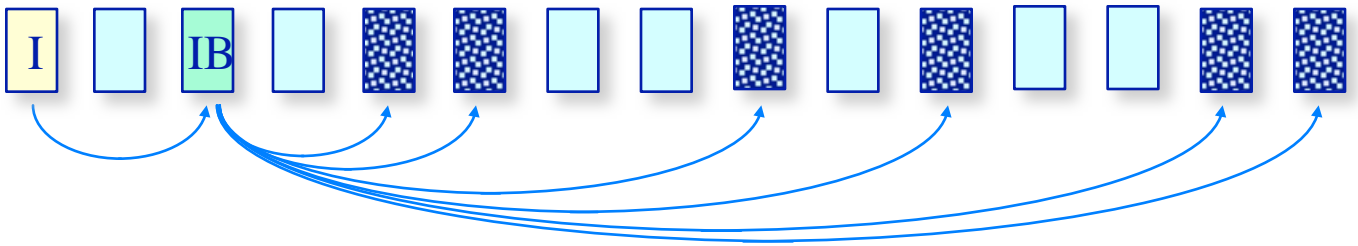
- File header points to each data block

- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access

- Minuses
  - Inode is big or variable size
  - How to handle large files?

File Allocation Methods

Indexed allocation



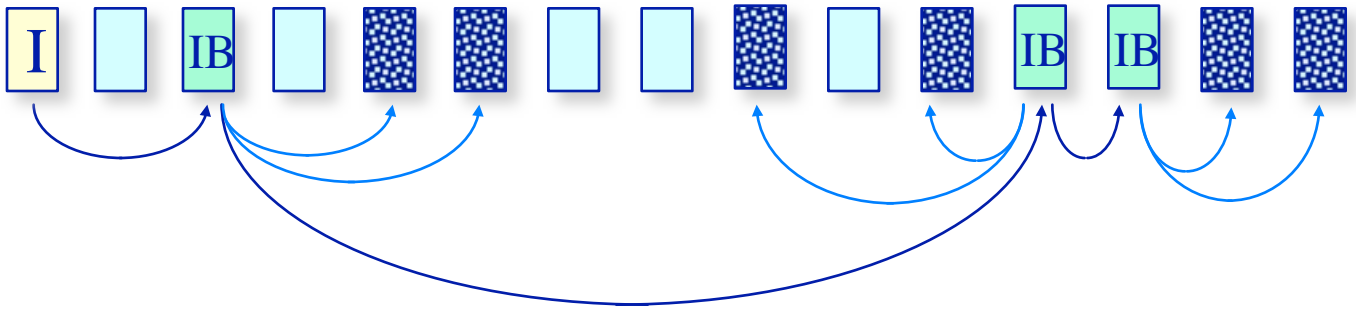
- Create a non-data block for each file called the *index block*
  - A list of pointers to file blocks
- File header contains the index block

- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access
- Minuses
  - Overhead of storing index when files are small
  - How to handle large files?

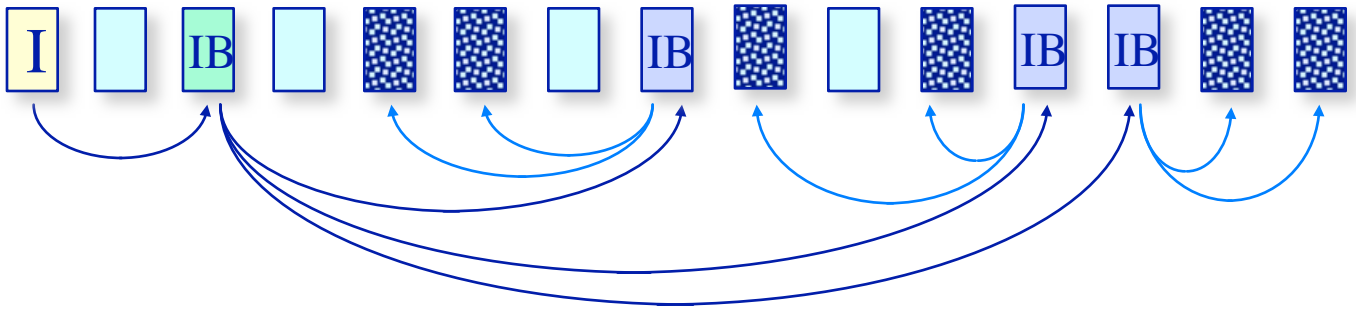
Indexed Allocation

Handling large files

- Linked index blocks (IB+IB+…)

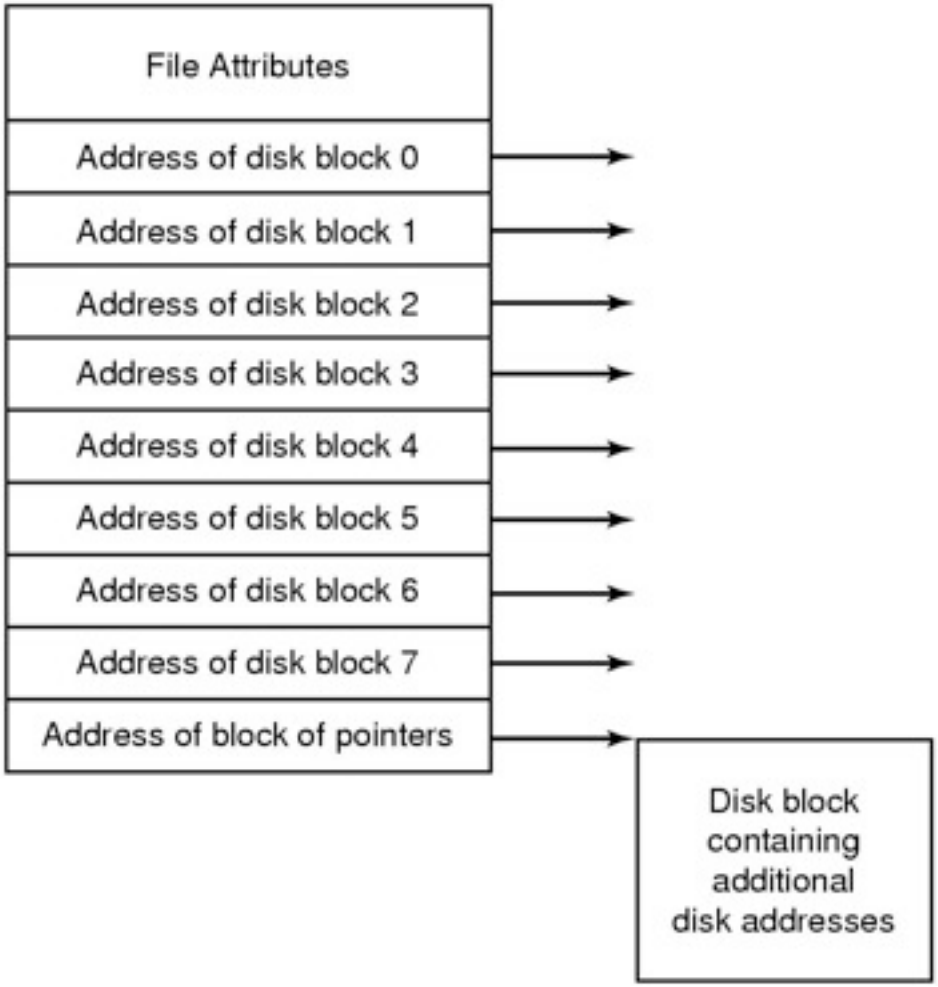


- Multilevel index blocks (IB*IB*…)



- Why bother with index blocks?
  - A. Allows greater file size.
  - B. Faster to create files.
  - C. Simpler to grow files.
  - D. Simpler to prepend and append to files.
  - E. Scott Summers is the X-men's Cyclops

I-nodes



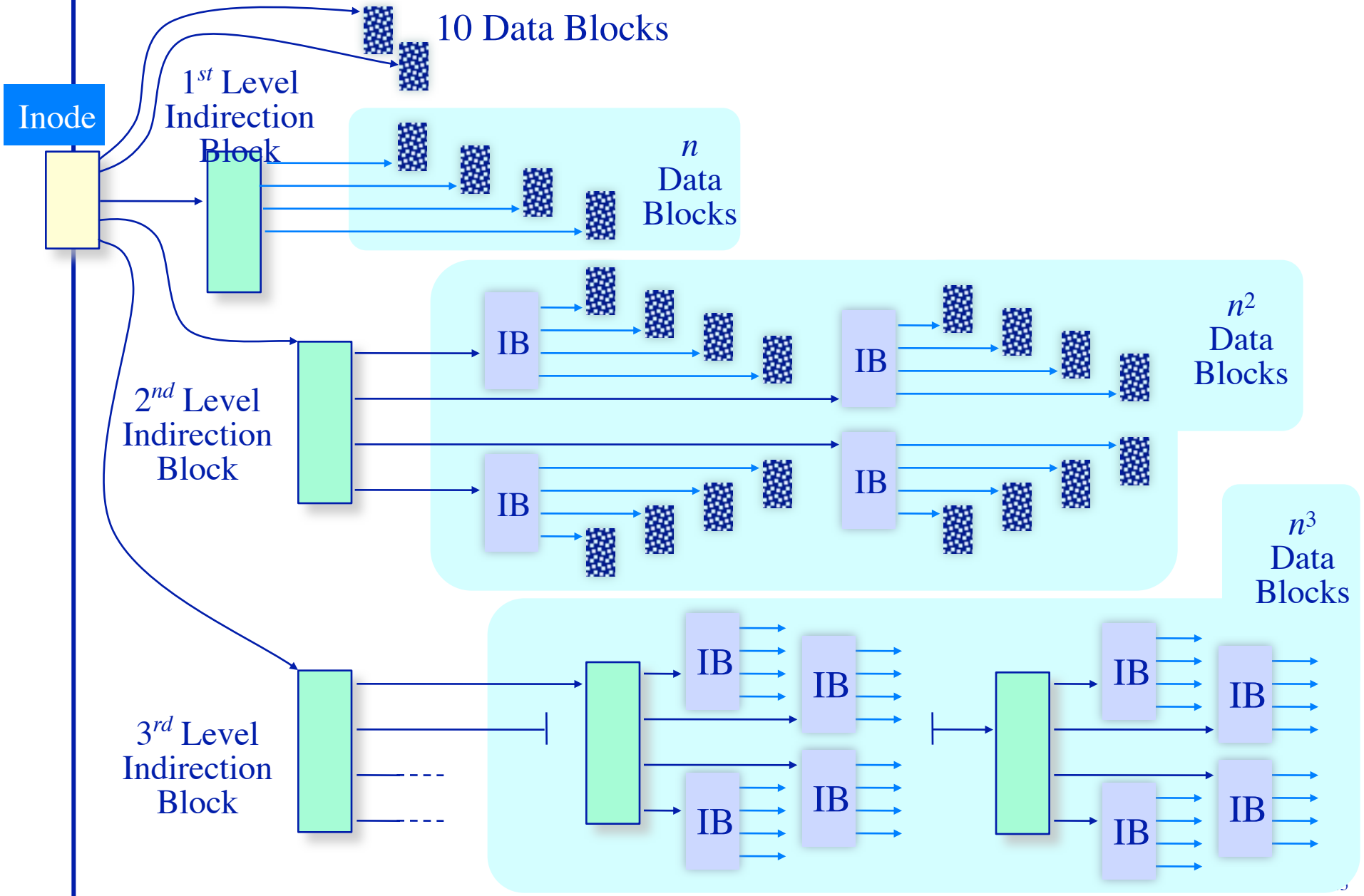
- I-node (index-node) lists disk addresses of the file's blocks
- Used in UNIX

Multi-level Indirection in Unix

- File header contains 13 pointers
  - 10 pointes to data blocks; 11th pointer → indirect block; 12th pointer → doubly-indirect block; and 13th pointer → triply-indirect block
- Implications
  - Upper limit on file size (~2 TB)
  - Blocks are allocated dynamically (allocate indirect blocks only for large files)
- Features
  - Pros
    - Simple
    - Files can easily expand
    - Small files are cheap
  - Cons
    - Large files require a lot of seek to access indirect blocks

Indexed Allocation in UNIX

Multilevel, indirection, index blocks



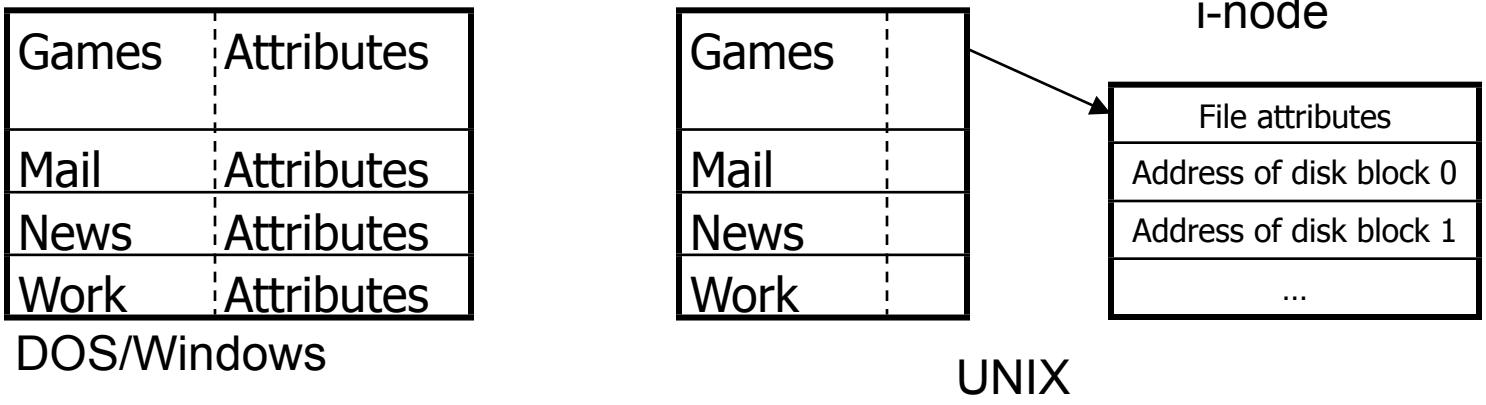
I-nodes Pros and Cons

- Pros:
  - Small size: an i-node need only be in memory when the corresponding file is open, hence total size of i-nodes is proportional to max number of files that are open simultaneously
    - In contrast, FAT table size is proportional to total number of disk blocks
- Cons:
  - Each i-node has a fixed number of disk addresses; a file may grow beyond the limit
    - Solution: reserve the last disk address for the address of a block containing additional disk block addresses (indirect blocks)

- How big is an inode?
  - A. 1 byte
  - B. 16 bytes
  - C. 128 bytes
  - D. 1 KB
  - E. 16 KB

Implementing Directories

- Directory system: map the ASCII file name onto the info needed to locate the data
  - Directory entry
- Where are the attributes stored?
  - In the directory entry (MS-DOS/Windows)
  - In the i-nodes (UNIX)



Locate A File: /usr/ast/ mbox

root

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*

Attr.
132
.....

/usr is in block 132

Block 132 is */usr* dir.

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*

Attr.
406
.....

/usr/ast is in block 406

Block 406 is */usr/ast* dir.

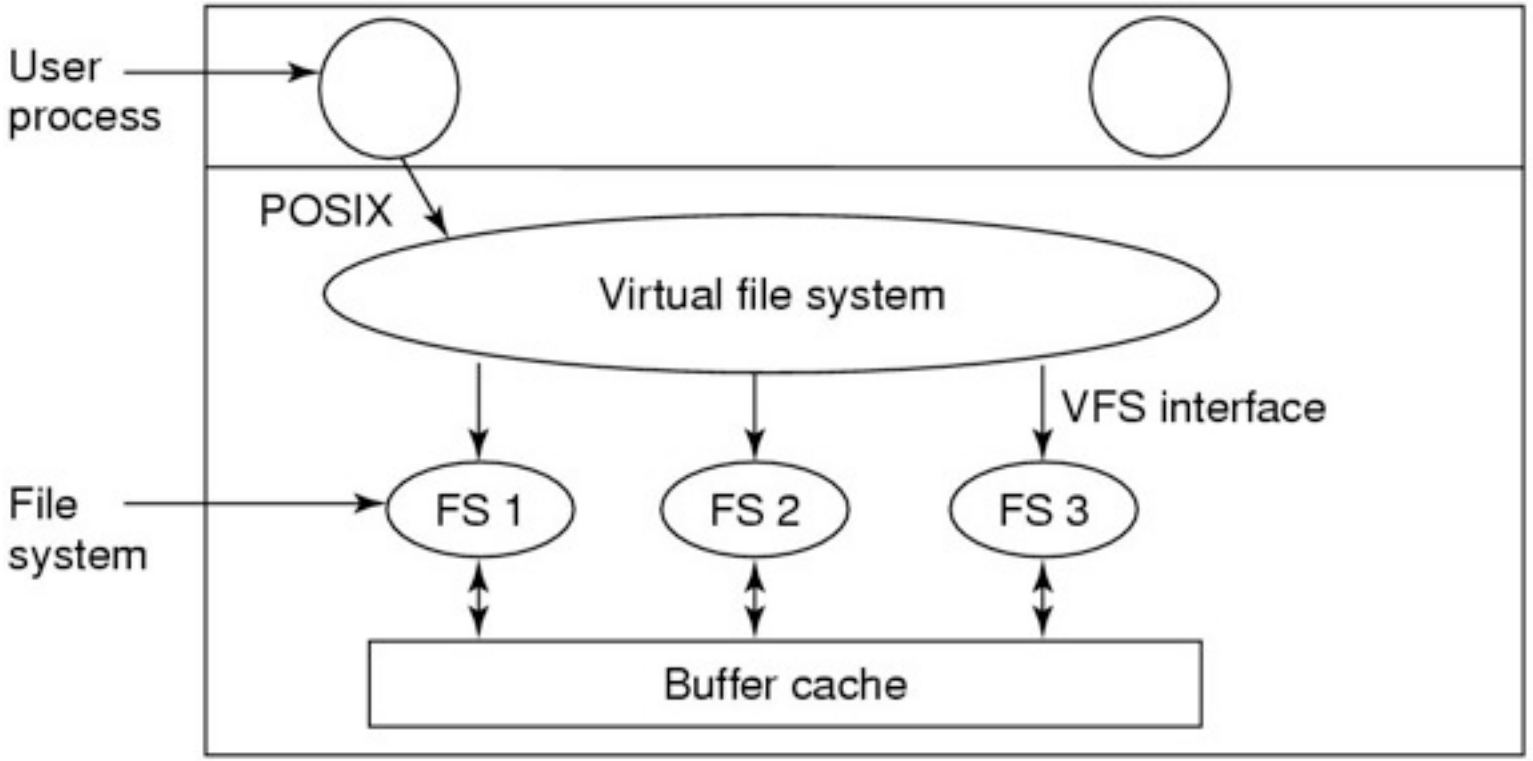
26	.
6	..
64	grants
92	books
60	mbox
81	simix
17	src

/usr/ast/mbox is i-node 60

- Sequence of disk accesses to resolve "*/usr/ast/mbox*"?
  - Read in inode for root (fixed position on disk)
  - Read in first data bock for root; search for "usr" to get address of its inode.
    - Table of file name/index pairs. Search linearly – ok since directories typically very small
  - Read in inode for "usr" to get addresses of its data blocks
  - Read in first data block for "usr"; search for "ast" to get address of its inode.
  - Read in inode for "ast" to get addresses of its data blocks
  - Read in first data block for "ast"; search for "mbox" to get address of its inode.
  - Read in inode for "mbox" to get addresses of its data blocks

- Current working directory: Per-address-space pointer to a directory (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (if CWD=""*/usr/ast/*", then can resolve "mbox" without absolute path.)

Virtual File System



- An additional layer of software to hide differences among different file systems and present a uniform interface (e.g., POSIX) to the user

Approaches: Summary

- Contiguous storage:
  - Excellent access time (sequential and random)
  - Poor space usage with external fragmentation
  - Simple management, no need for data structures to relate non contiguous blocks
  - VERY difficult and inefficient to extend existing files !!!!
- FAT
  - Good sequential access (2 dereferences per block)
  - Good random access (order N)
  - Good space usage with some internal fragmentation
  - One pointer per block must be stored in memory and on disk, size increases as size of disk increases
  - Not efficiently scalable to large disks
- I-nodes
  - Good sequential access (2-4 dereferences per block)
  - Better random access (order logN)
  - Good space usage with some internal fragmentation
  - One pointer and one I-node per file on disk
  - One pointer and one I-node per OPEN file in memory

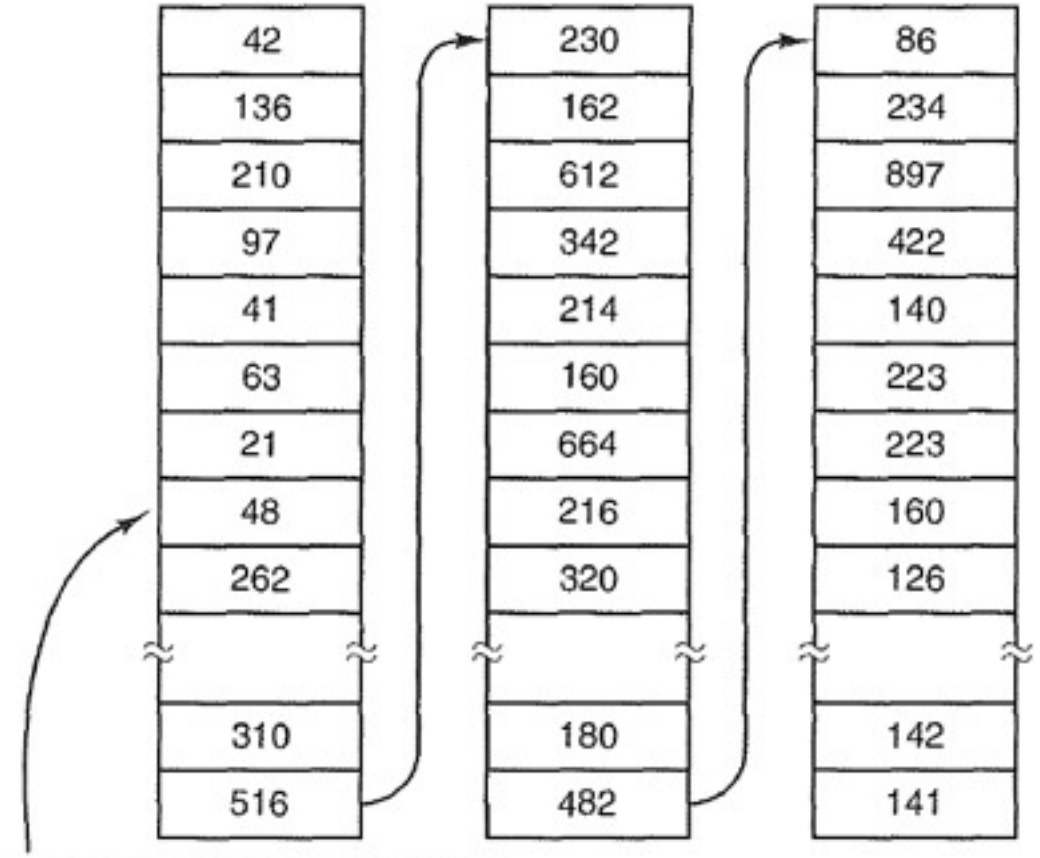
Choosing a block size

- Large block size means
  - Large amount of internal fragmentation, decreased disk space utilization (less of the disk actually being used)
  - Wasted space!
- Small block size means
  - Most files occupy multiple blocks, thus need multiple seeks and rotational delays to access
  - Reduced performance!

Free list

- Need to keep track of which blocks are free.
- Two common approaches
  - Linked list of disk blocks holding addresses of free blocks
  - Bitmap, 1 bit for each block, 0 if not allocated, 1 if allocated.
- The list or bitmap is kept on the disk (not in memory) to prevent data loss upon system crash

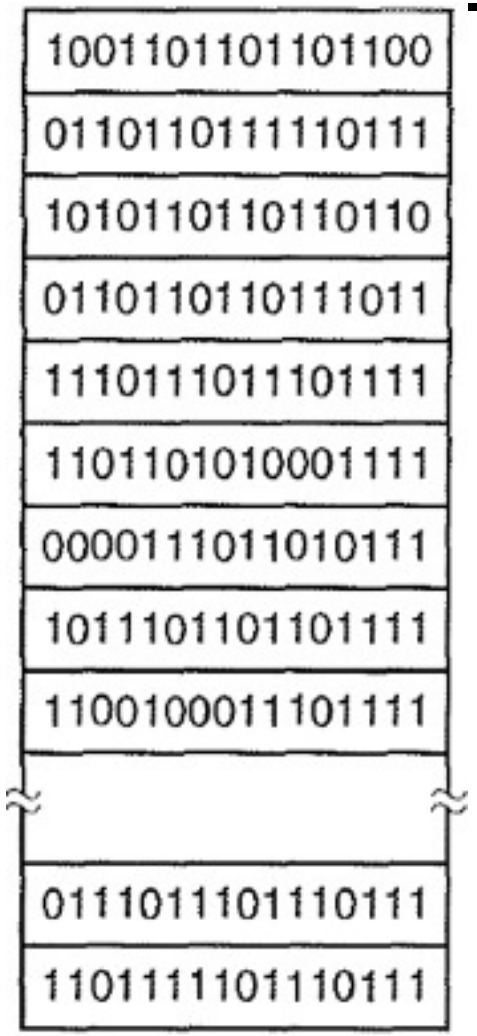
Free list management: linked list



- Addresses of free blocks are kept in a list of disk blocks
- Suppose each disk block address is 32-bits (4 Bytes), then each 1KB-disk block holds 255 addresses for free blocks (plus one address for the next block)

Figure 4-22. (a) Storing the free list on a linked list

Free list management: bitmap



- A disk with n blocks requires a bitmap with n bits.
- Each bit in the bitmap refers to a block
  - 1 indicates free blocks
  - 0 indicates allocated blocks
- 1-bit per block, on contrast to 32-bits per block for the linked list

Free list management: bitmap cont'

- How much space is required?
  - One bit for each block on the disk
  - Disk size in bytes / (8 * block size in bytes)
    - Dividing by 8 is for converting from # bits to # bytes
  - Example for an 8GB disk with 2KB blocks
  - $8*2^{30}/(8*2*2^{10})=0.5*2^{20}=0.5$ MB (250 disk blocks)
- Less space
  - But more time searching
- Used by MacOS, NTFS(Windows)
- Q: When will linked-list scheme require less space than bitmap scheme?
- A: When the disk is nearly full (with few free blocks)

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain “dirty” blocks (blocks yet on disk)
  - Example, ‘Use Once’:
    - File system can discard blocks as soon as they are used

- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - Works very well for name translation
    - Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - Fails when some application scans through file system, thereby flushing the cache with data used only once
    - Example: `find . -exec grep foo{}`
- Other Replacement Policies?
  - Some systems allow applications to request other policies

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced

- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - Too many imposes delays on requests by other applications
    - Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk (similar to write-back cache)
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - If some other application tries to read data before written to disk, file system will read from cache
    - Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

- Flushed to disk periodically (e.g. in UNIX, every 30 sec)
- Advantages:
  - Disk scheduler can efficiently order lots of requests
  - Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
- Disadvantages
  - What if system crashes before file has been written out?