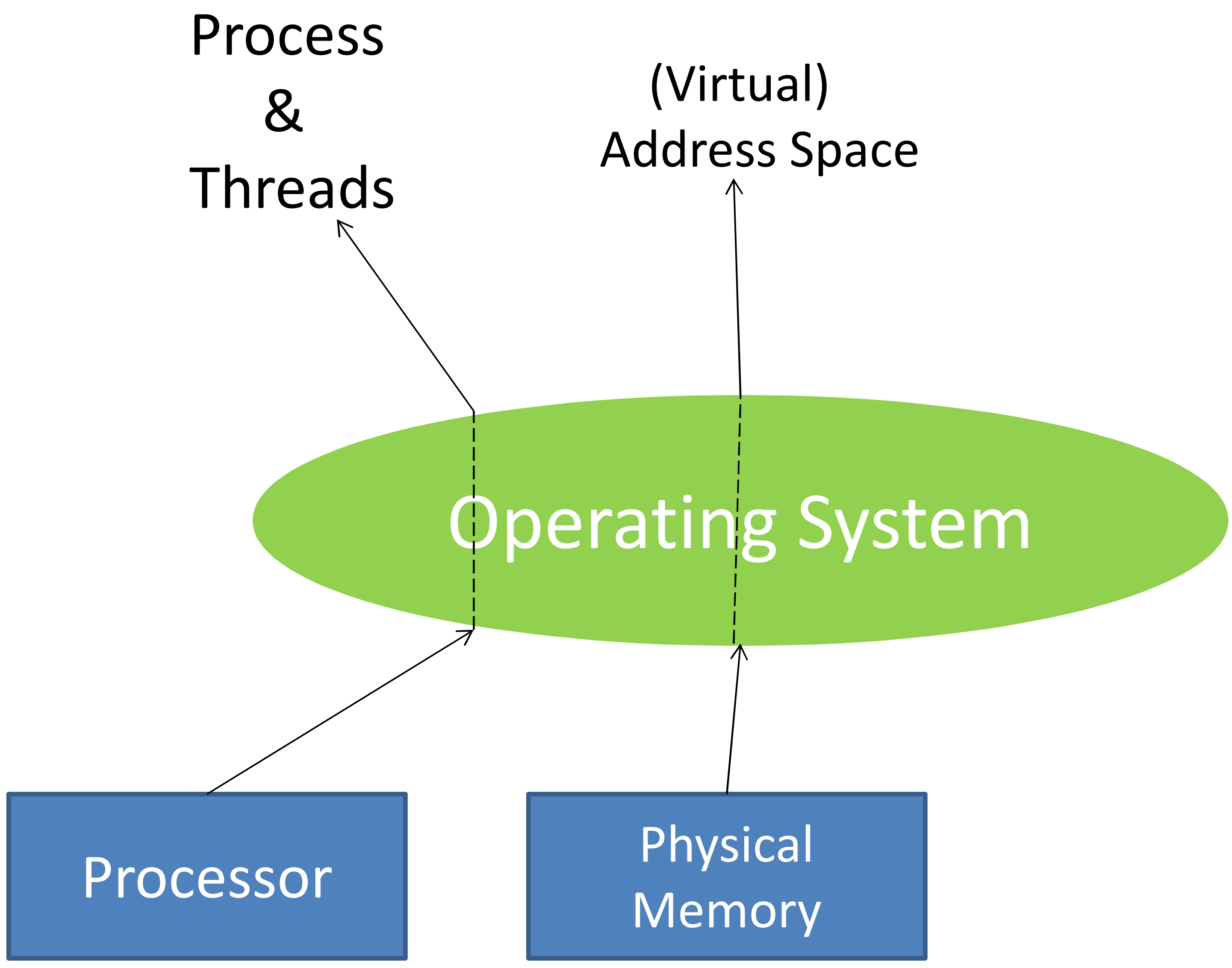
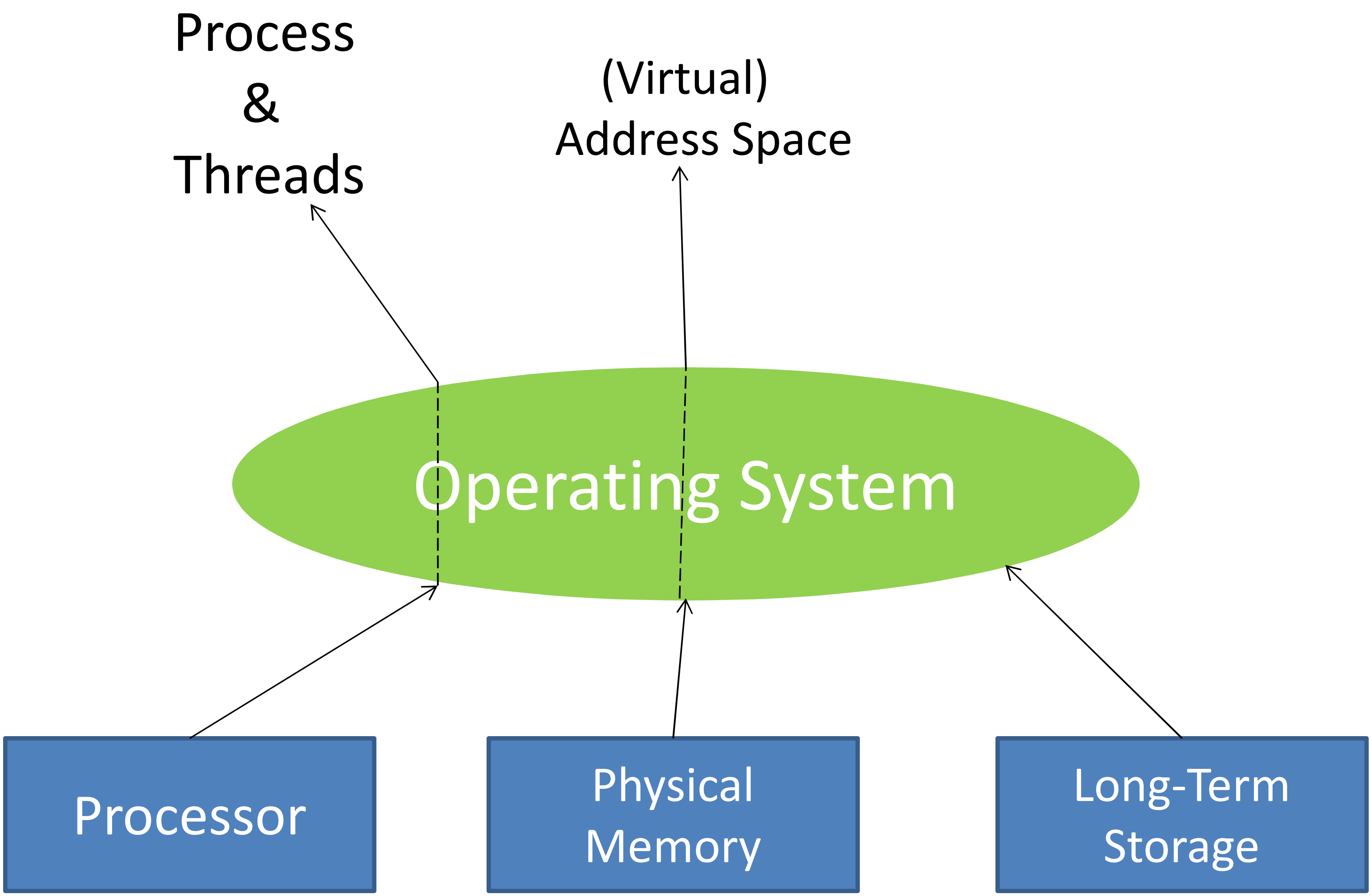
CSCI-GA.2250-001

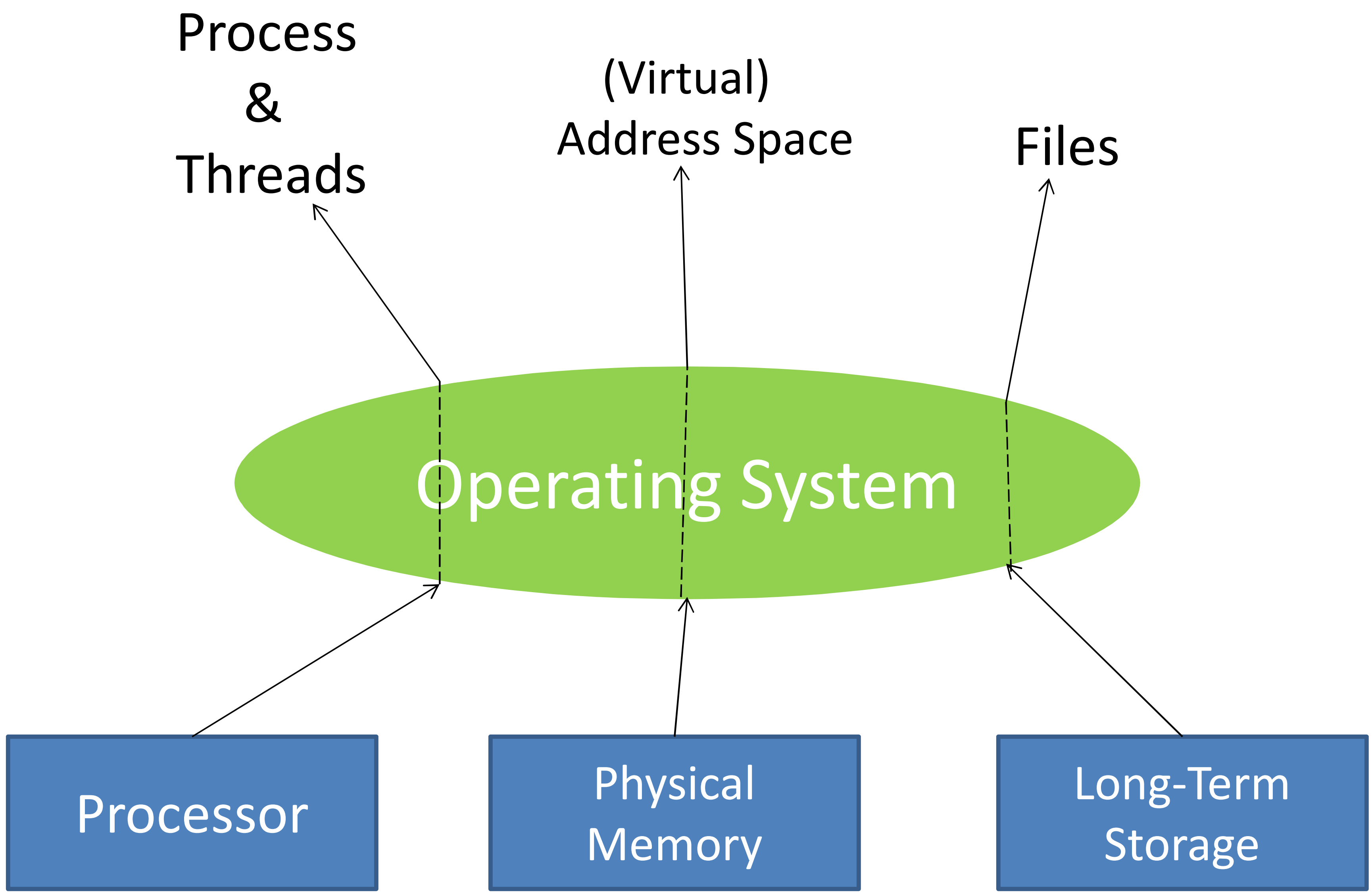
Operating Systems

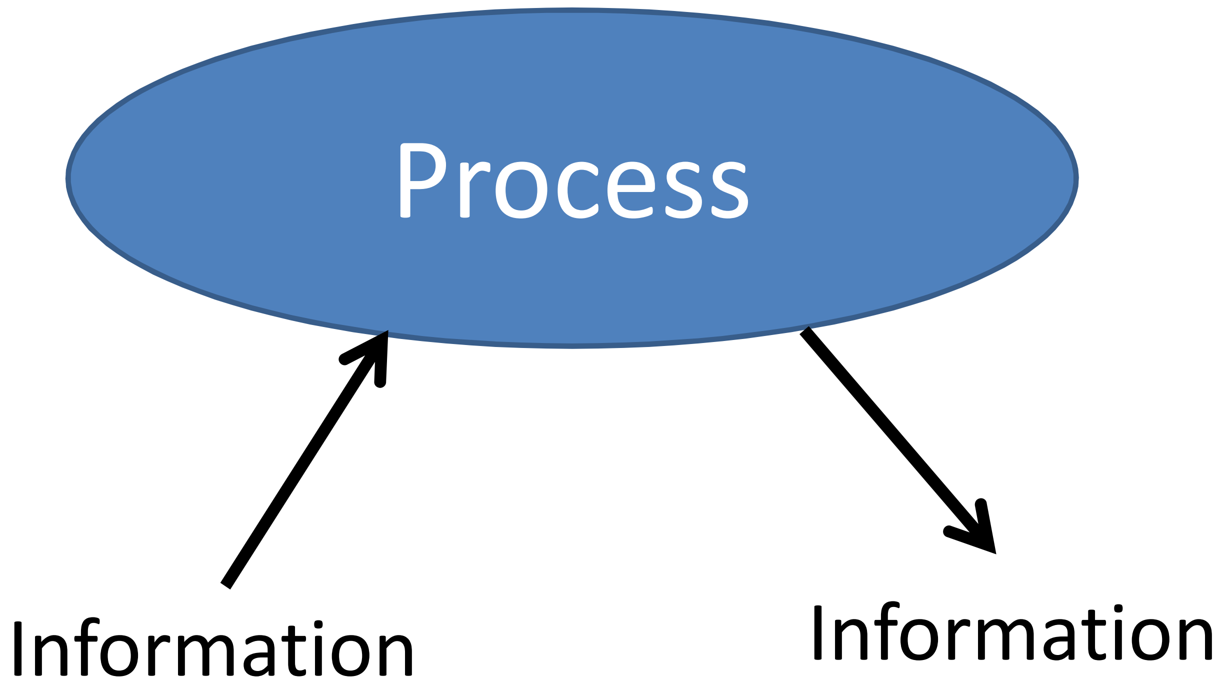
File Systems

Hubertus Franke
frankeh@cs.nyu.edu


Process
&
Threads
(Virtual)
Address Space
Operating System
Processor
Physical
Memory


Process
&
Threads
(Virtual)
Address Space
Operating System
Processor
Physical
Memory
Long-Term
Storage


Process
&
Threads
(Virtual)
Address Space
Files
Operating System
Processor
Physical
Memory
Long-Term
Storage



Is it OK to keep this information only in the process address space?

Shortcomings of Process Address Space

- Virtual address space may not be enough storage for all information
- Information is lost when process terminates, is killed, or computer crashes.
- Multiple processes may need the information at the same time

Requirements for Long-term Information Storage

- Store very large amount of information
- Information must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

Files

- Data collections created by users
- The File System is one of the most important parts of the OS to a user
- Desirable properties of files:

Long-term existence

- files are stored on disk or other secondary storage and do not disappear when a user logs off

Sharable between processes

- files have names and can have associated access permissions that permit controlled sharing

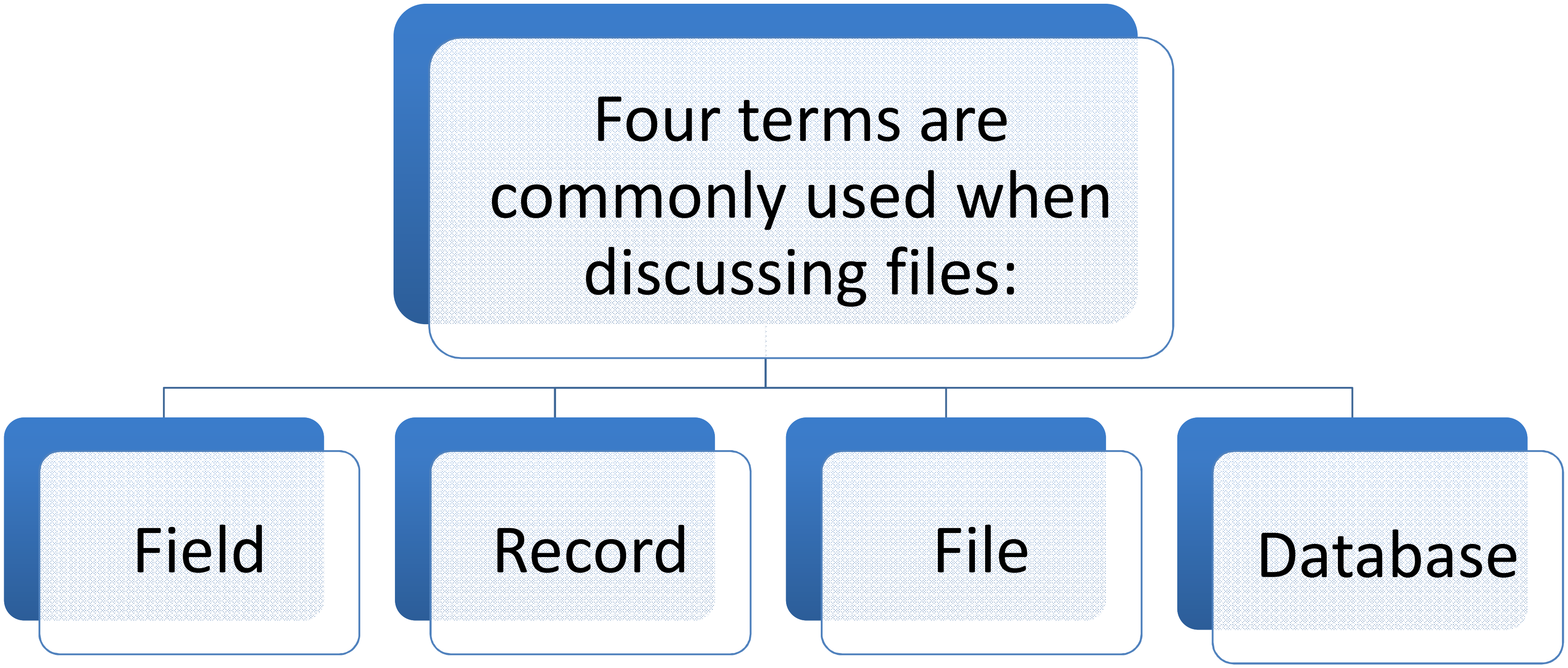
Structure

- files can be organized into hierarchical or more complex structure to reflect the relationships among files

Files

- Logical units of information created by processes
- Used to model disks instead of RAM
- Information stored in files must be persistent (i.e. not affected by processes creation and termination)
- Managed by OS
- The part of OS dealing with files is known as the file system

File Structure



Structure Terms

Field

- basic element of data
- contains a single value
- fixed or variable length

Database

- collection of related data
- relationships among elements of data are explicit
- designed for use by a number of different applications
- consists of one or more types of files

Record

- collection of related fields that can be treated as a unit by some application program
- fixed or variable length

File

- collection of similar records
- treated as a single entity
- may be referenced by name
- access control restrictions usually apply at the file level

Issues

- How do you find information?
- How do you keep one user from reading another user's data?
- How do you know which blocks are free?

Files from The User's point of View

- Files
  - Naming
  - Structure
  - Types
  - Access
  - Attributes
  - Operations
- Directories
  - Single-level
  - Hierarchical
  - Path names
  - Operations

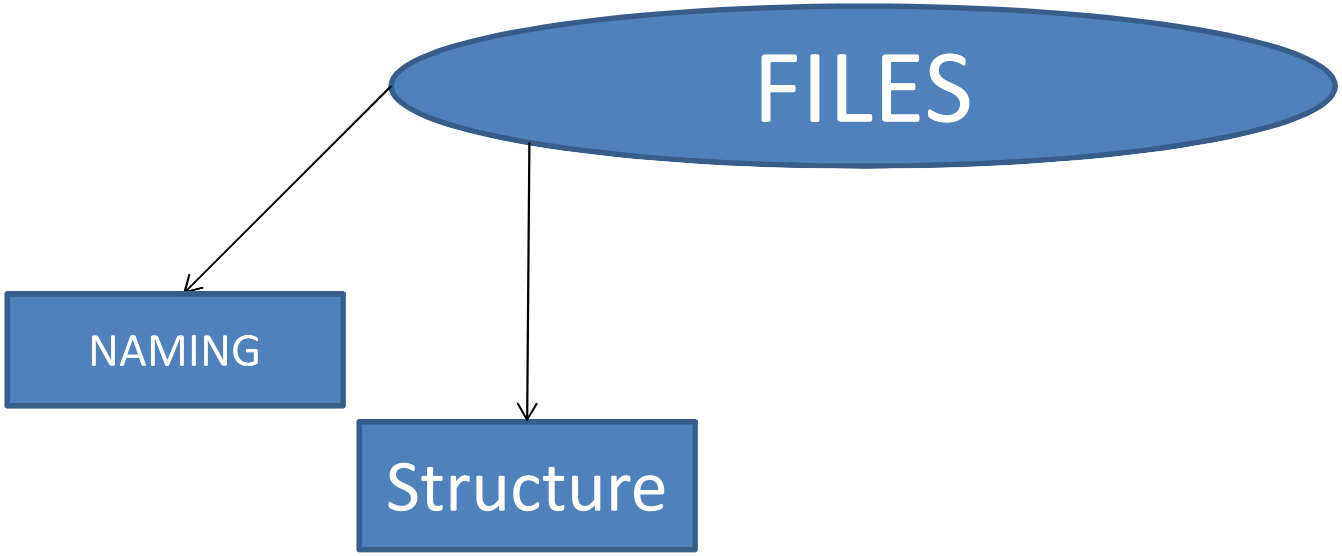
File Systems

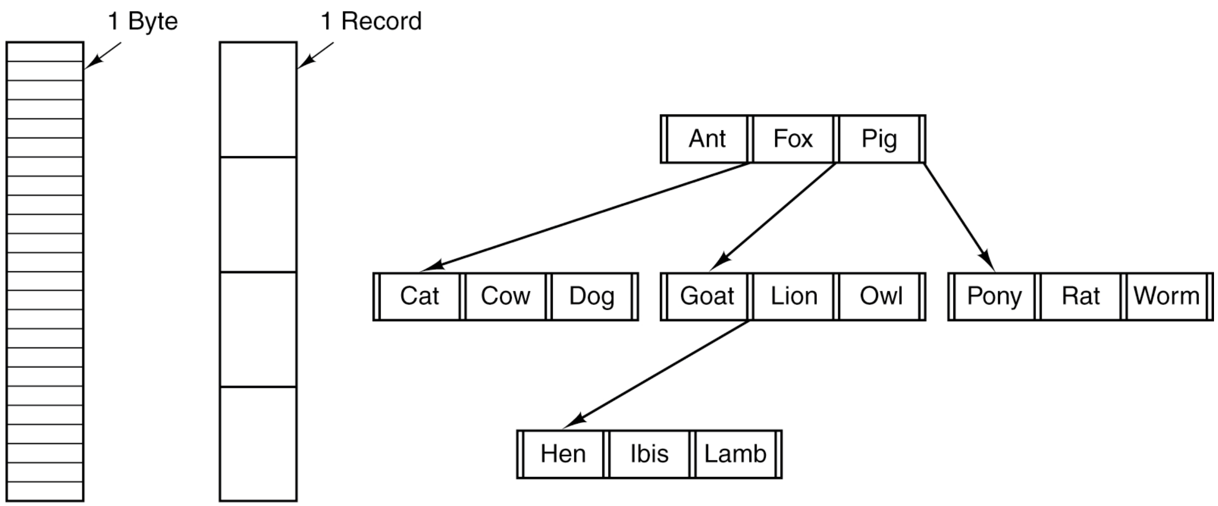
- Provide a means to store data organized as files as well as a collection of functions that can be performed on files
- Maintain a set of attributes associated with the file
- Typical operations include:
  - Create
  - Delete
  - Open
  - Close
  - Read
  - Write

FILES

NAMING

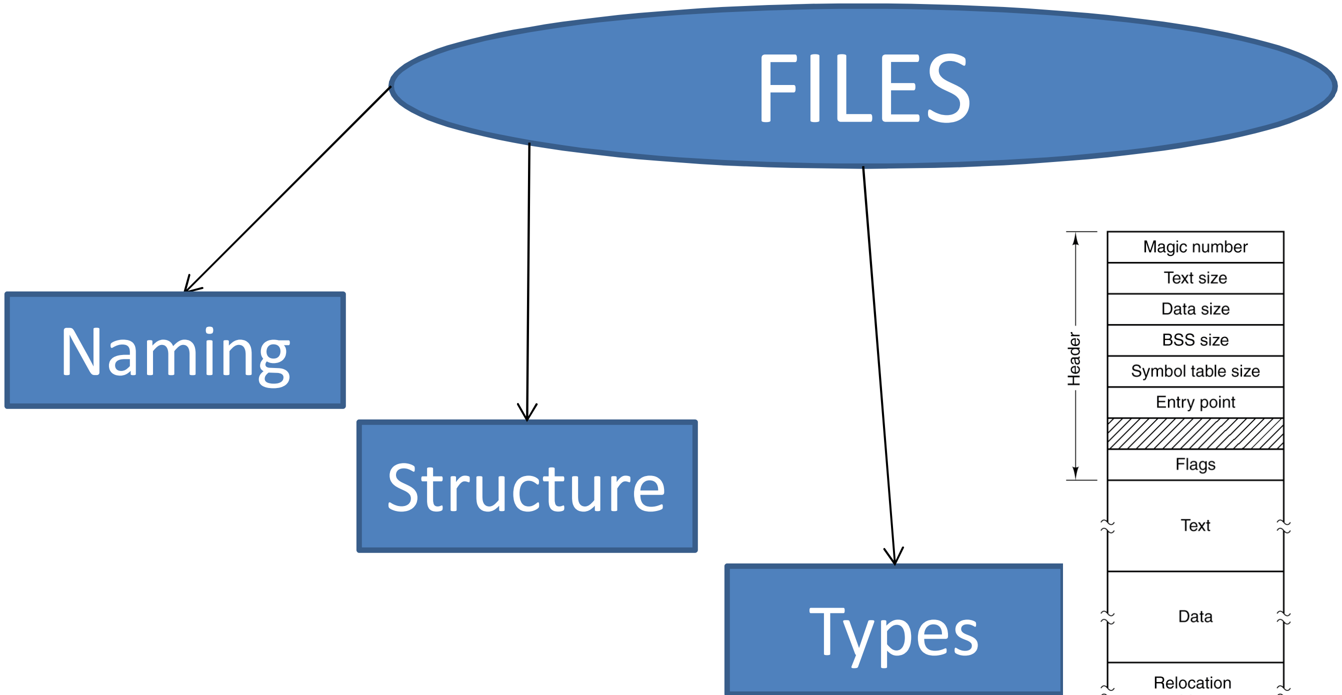
- Shields the user from details of file storage.
- In general, files continue to exist even after the process that creates them terminates.

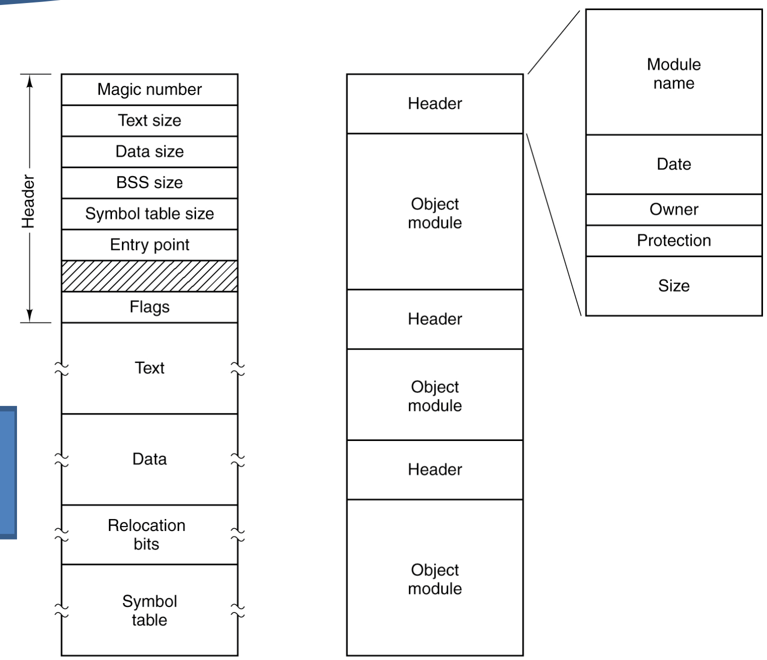




(a) (b) (c)

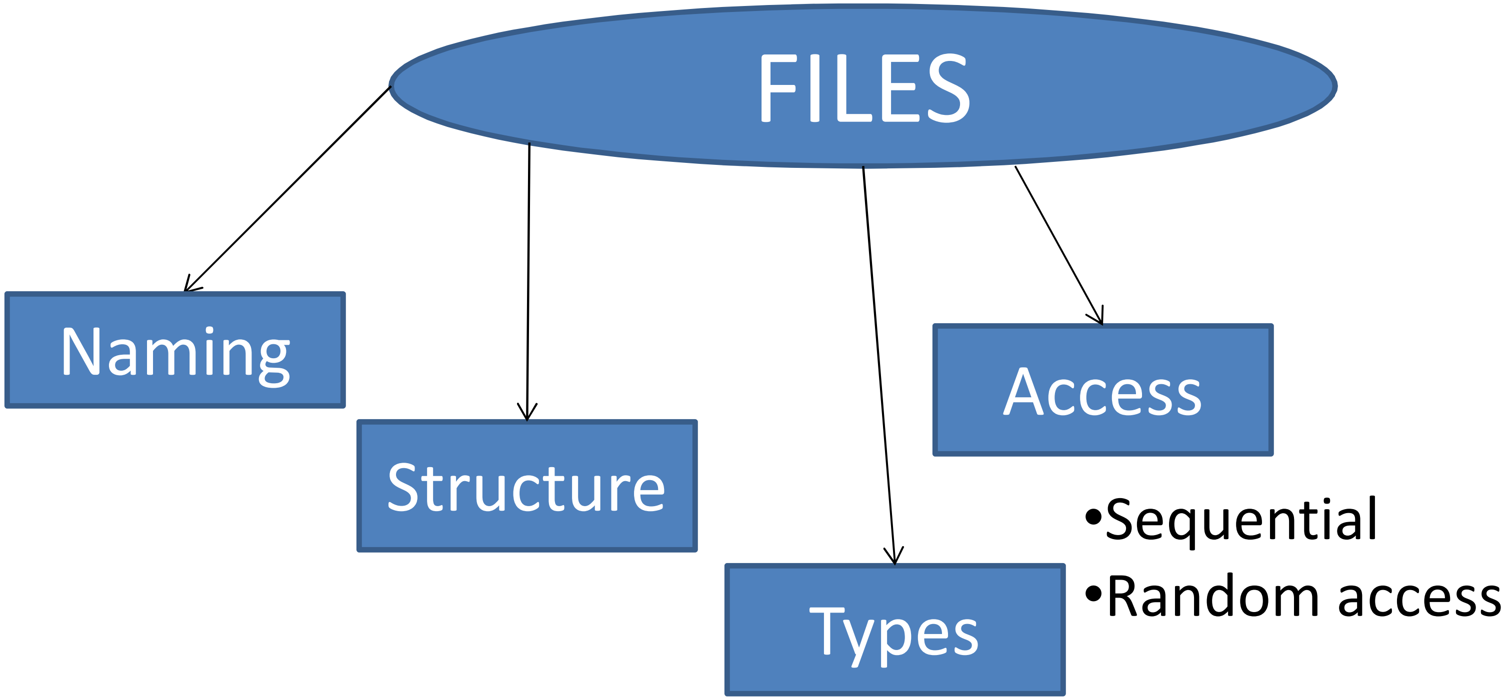
OS view — Historical — Still Used in Some Mainframes

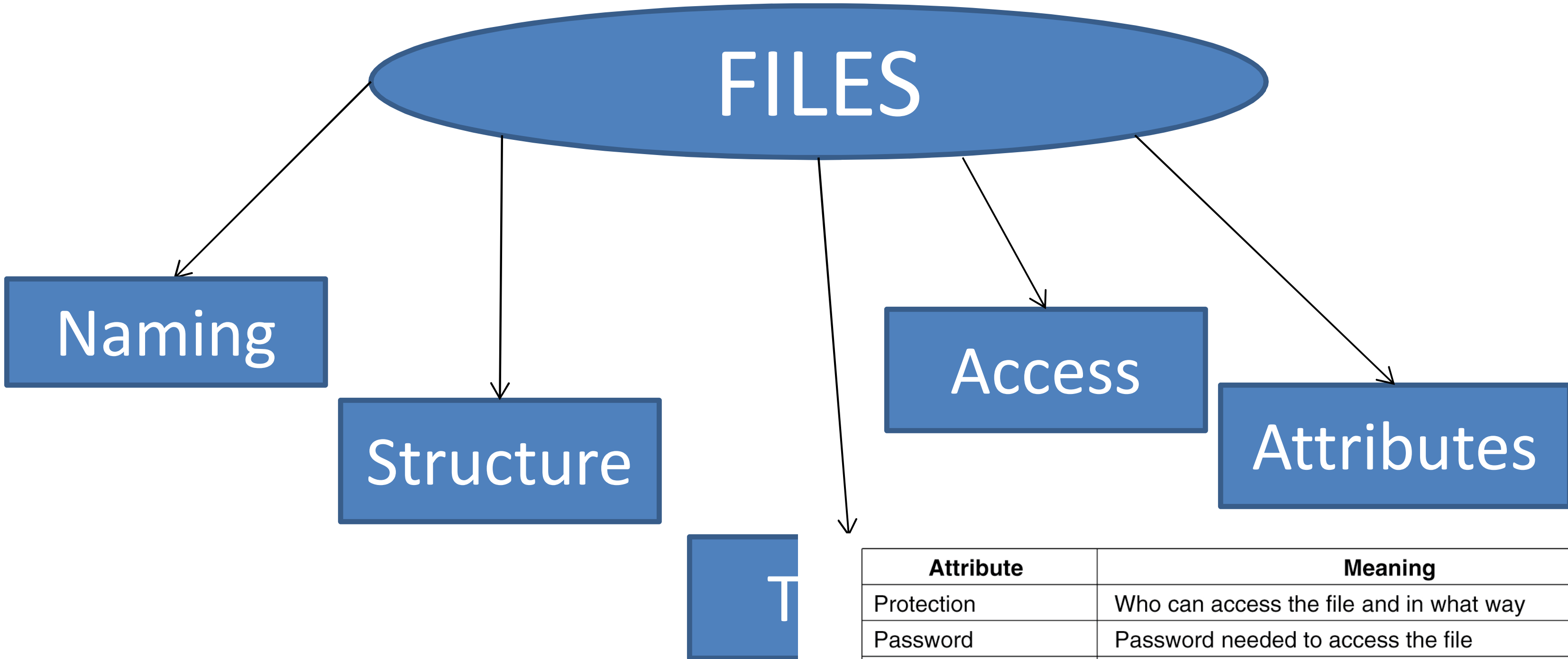




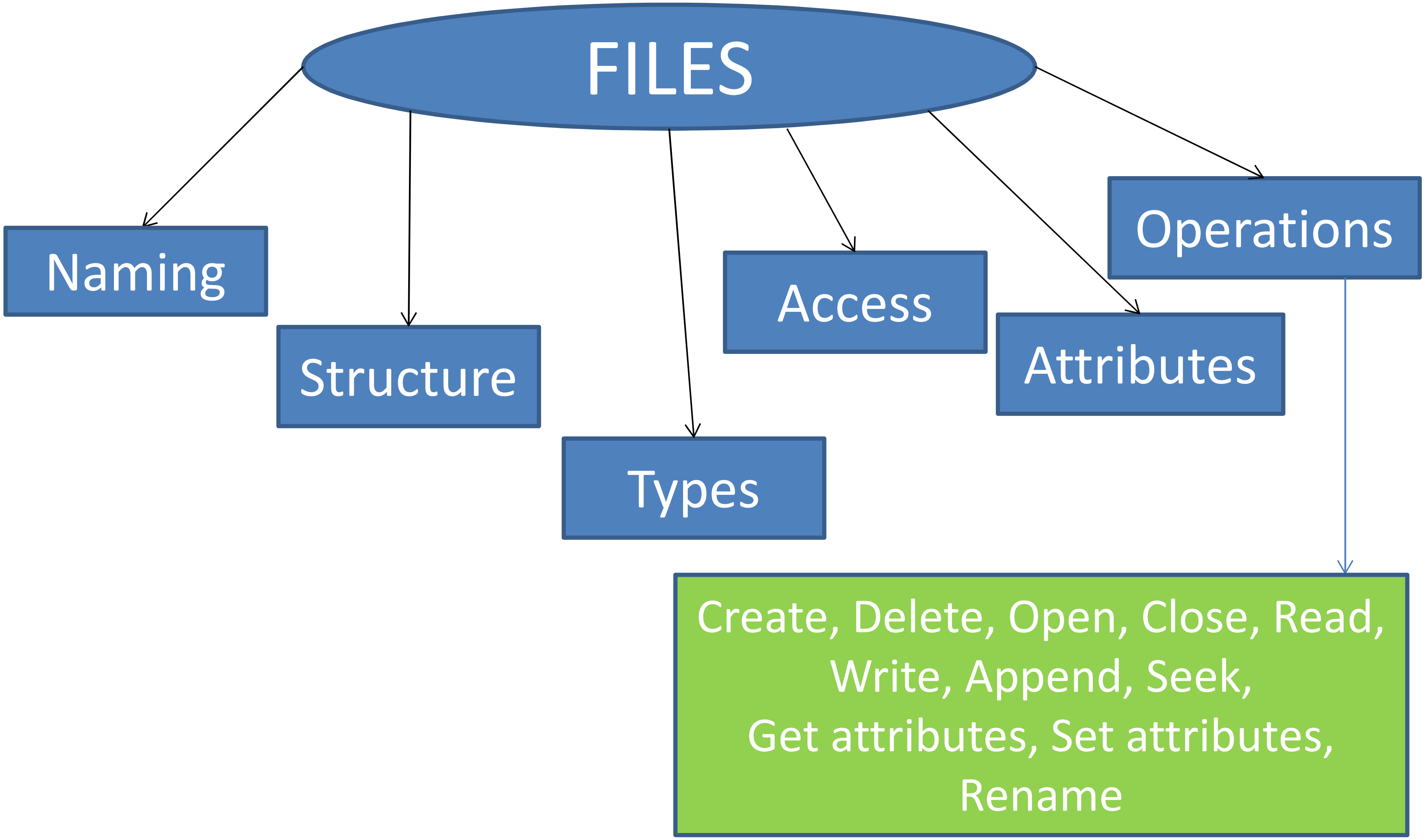
(a) (b)

- Regular files
  - ASCII
  - Binary
- Character special
  - to model serial devices (printers, networks, …)
- Block special
  - to model disks


FILES
Naming
Structure
Types
Access
•Sequential
•Random access

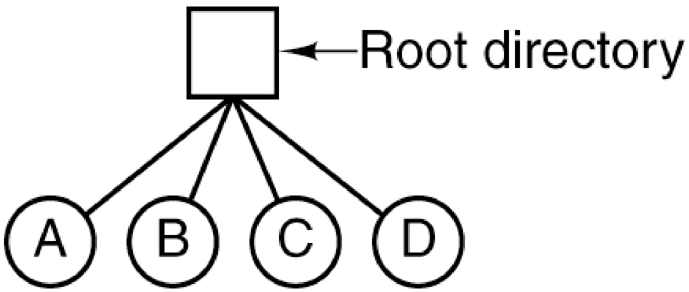


Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 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 was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to


FILES
Naming
Structure
Types
Access
Attributes
Operations
Create, Delete, Open, Close, Read, Write, Append, Seek, Get attributes, Set attributes, Rename

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

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

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

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

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

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

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

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

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

Directories: Single-Level Directory Systems



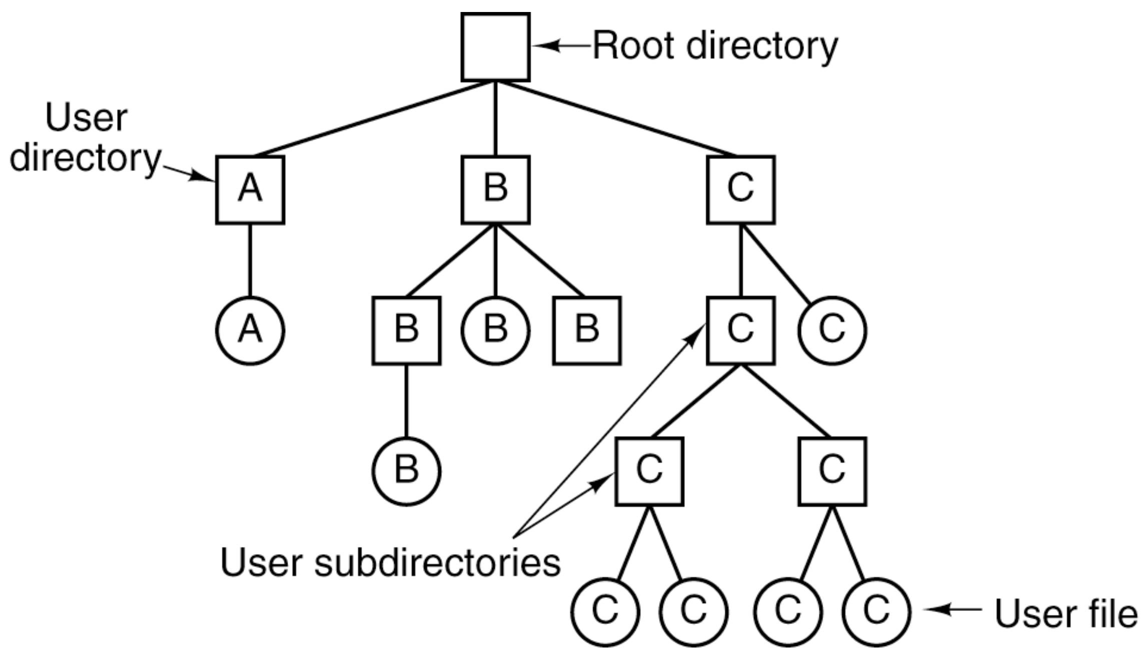
+ Simplicity

-Not adequate for large number of files.

Used in simple embedded devices

Directories: Hierarchical Directory Systems

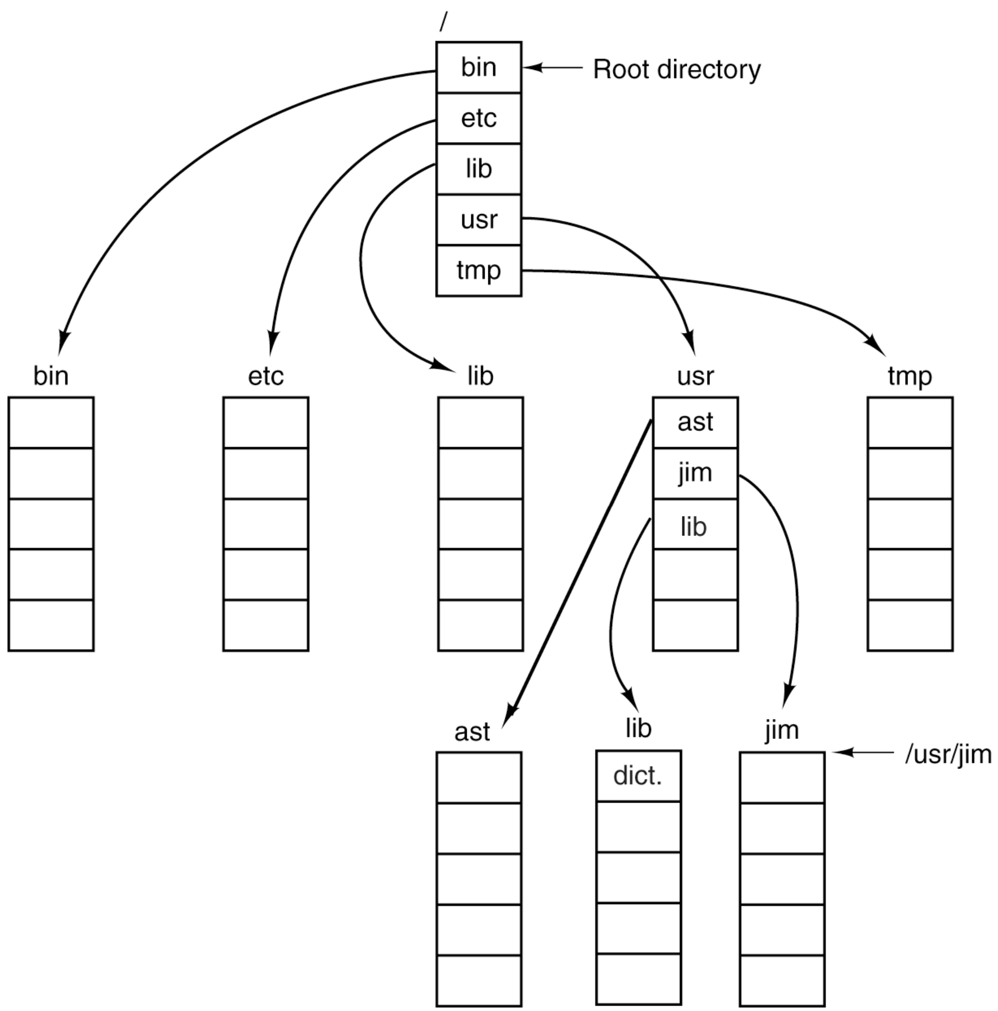
- Group related files together
- Tree of directories



Directories: Path Names

- Needed when directories are used
- Absolute path names
  - Always start at the root
  - A path from the root to the specified file
  - The first character is the separator
- Relative path names
  - Relative to the working directory
  - Each process has its own working directory

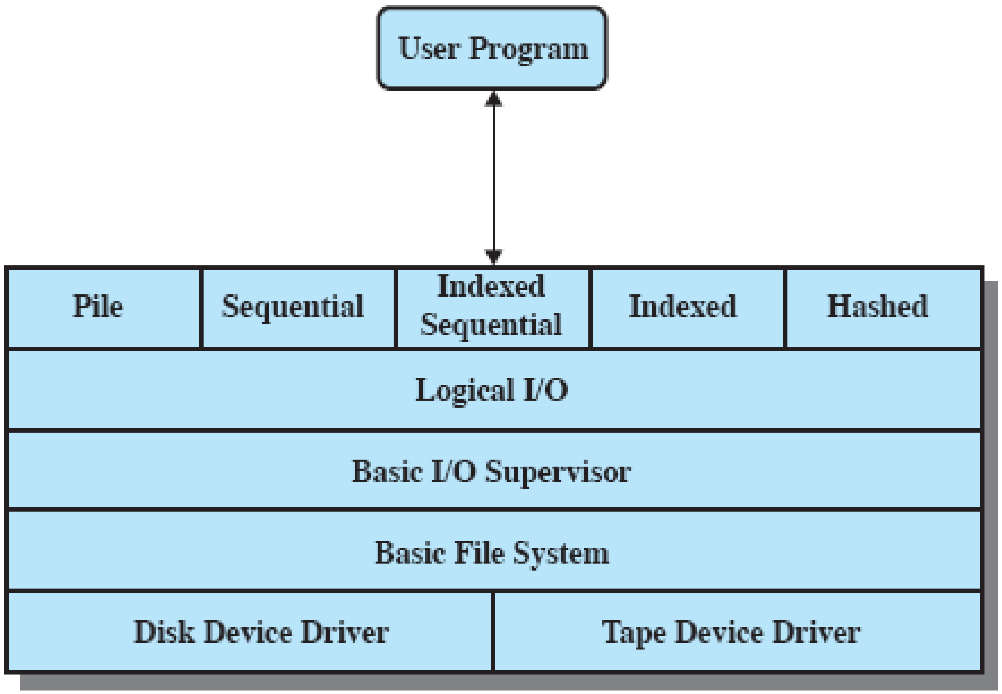
Directories: Path Names



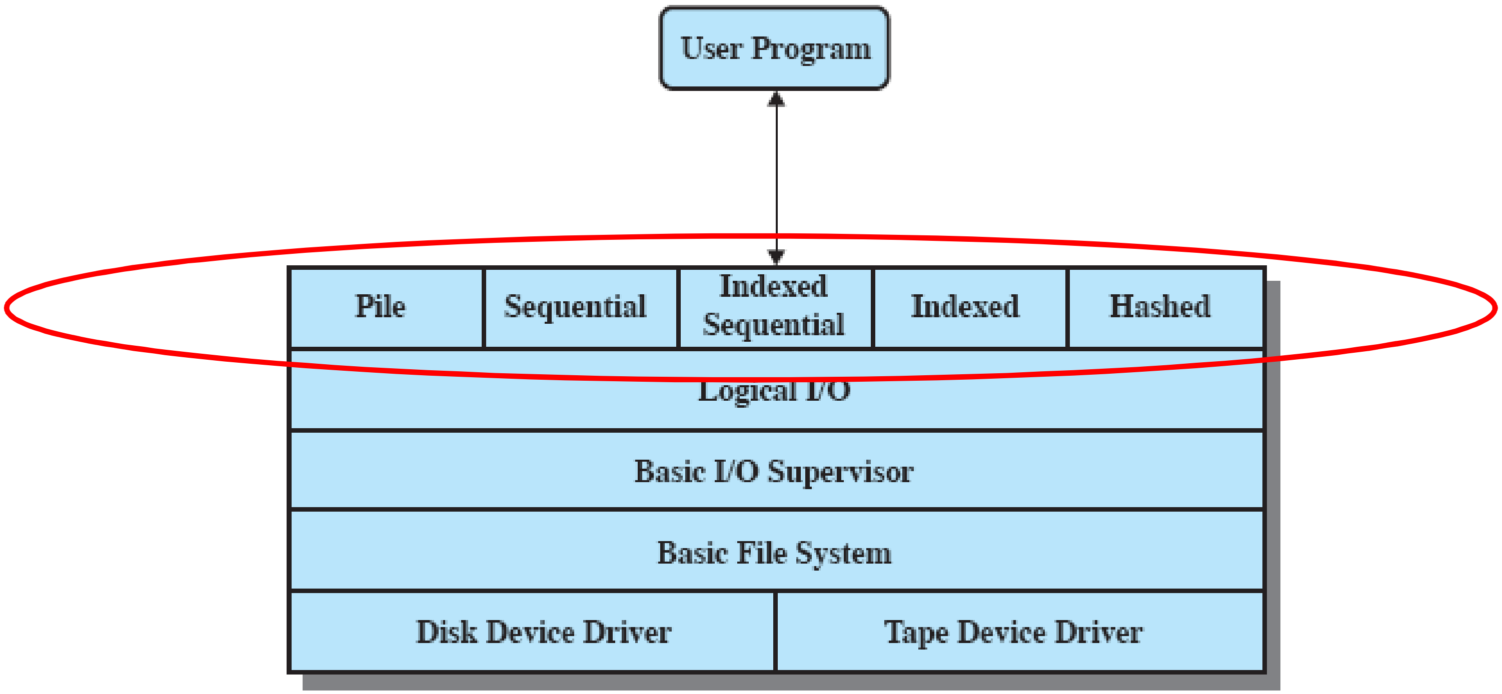
Directories: Operations

- More variations among OSes than file operations
- Examples (from UNIX):
  - Create, deleted
  - Opendir, closedir
  - Readdir
  - Rename
  - Link, unlink

Typical Organization

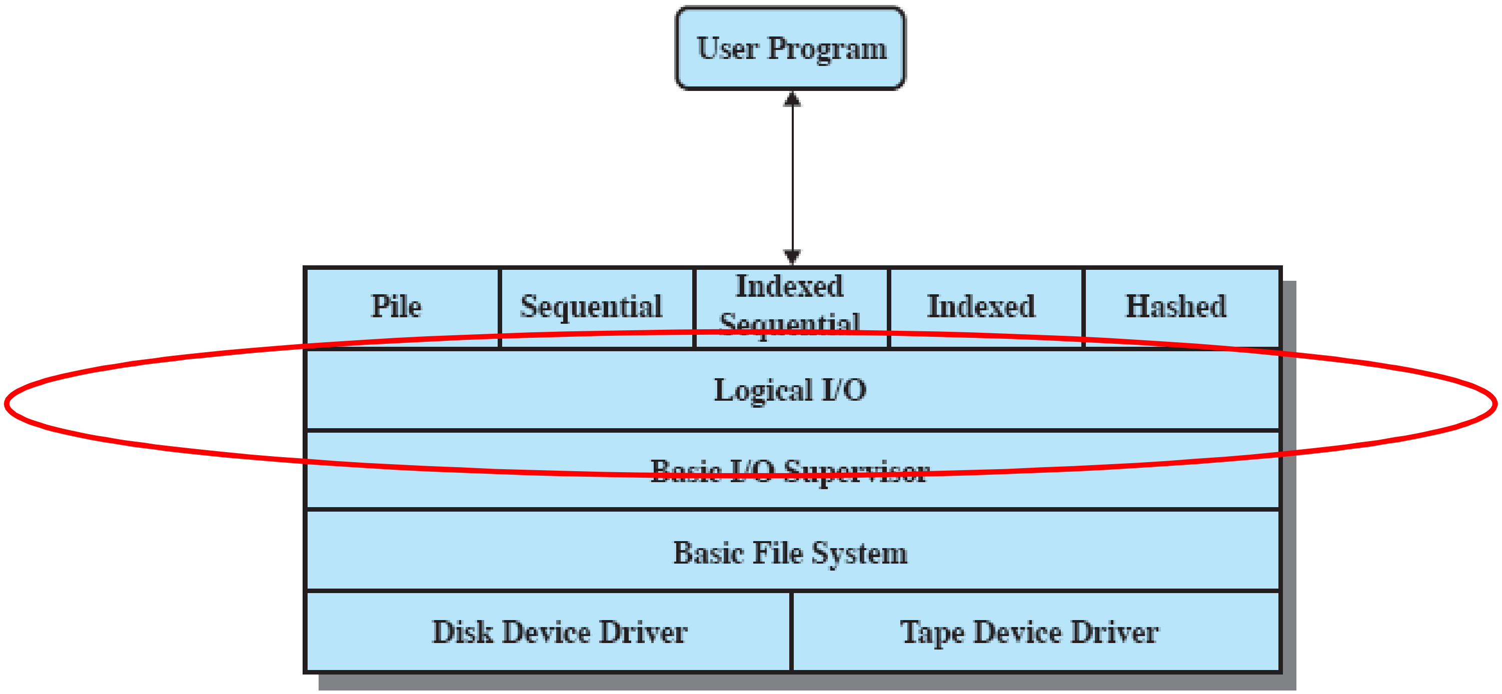


Typical Organization



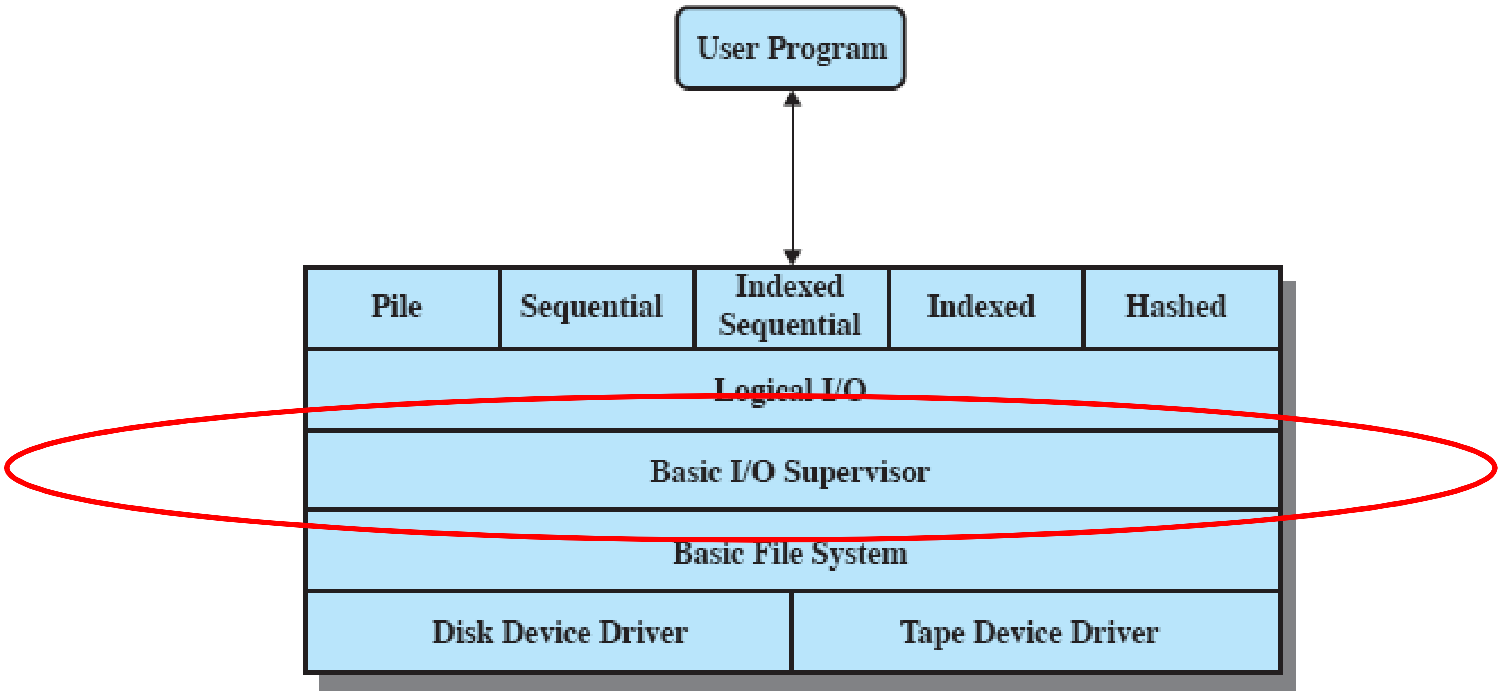
- The level closest to application and provides the access method to file system,
- The figure above shows some access methods, each of which reflects different file structures and different ways of accessing them.

Typical Organization



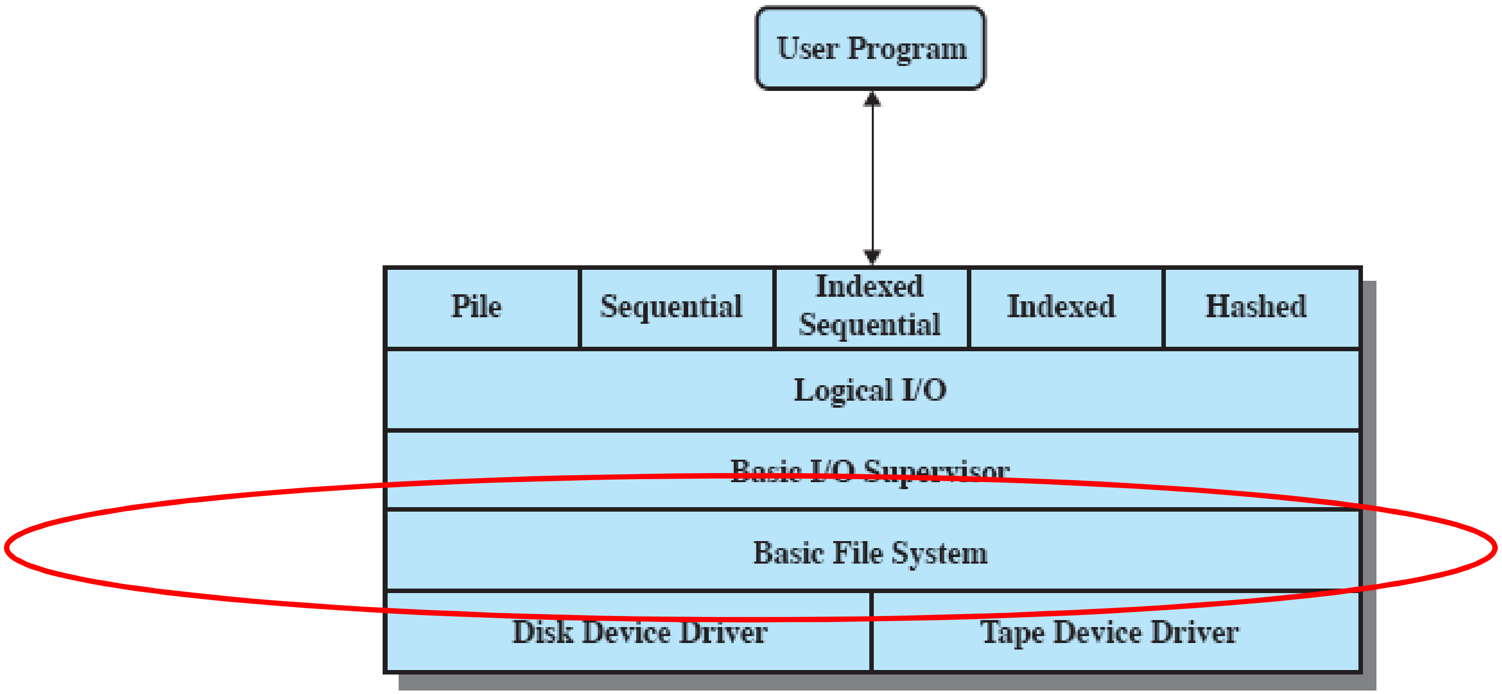
- Enables users and application to access records.
- Deals with file records

Typical Organization

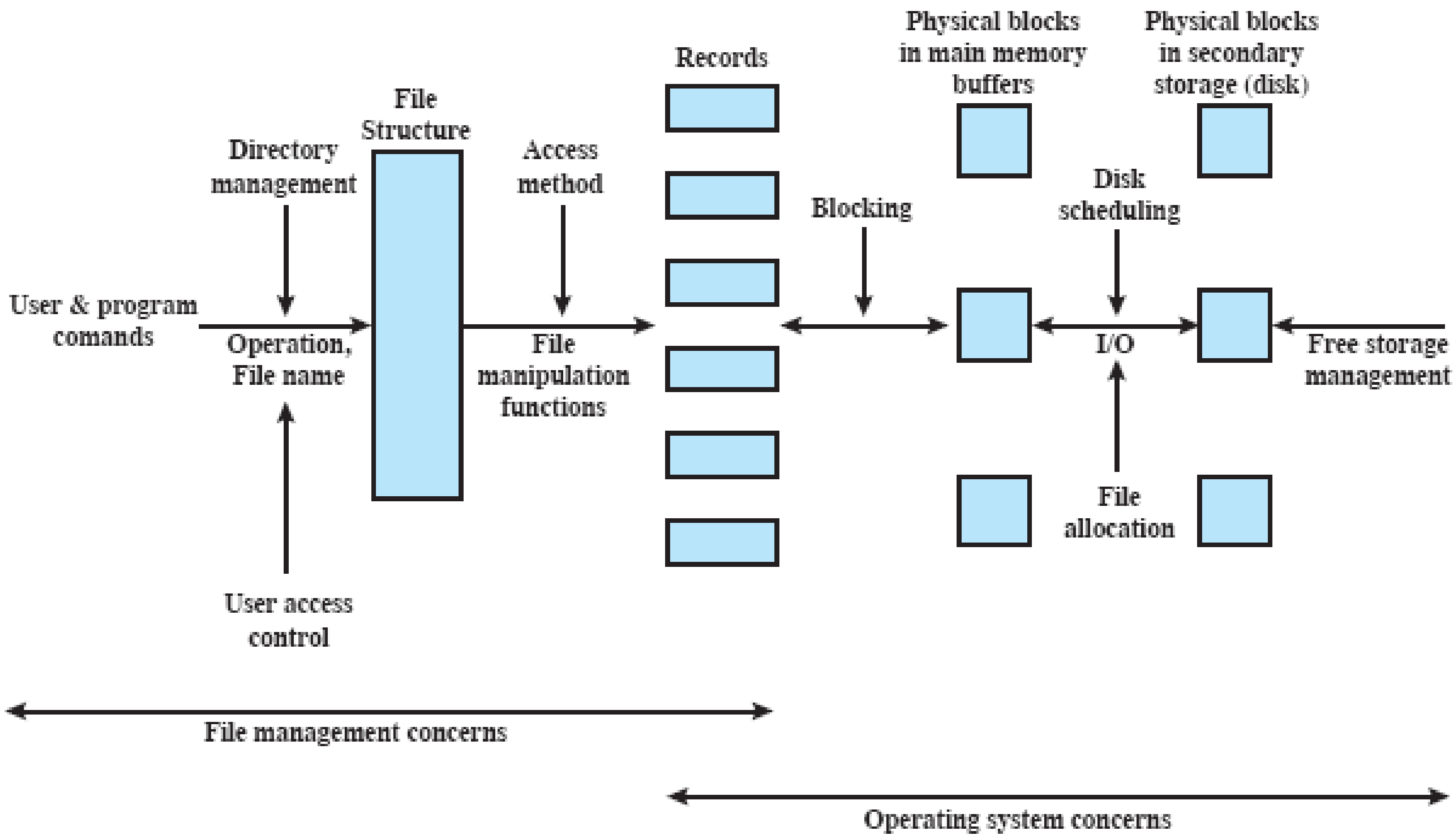


- Responsible for all File I/O initiation and termination
- Selects the device on which File I/O is to be performed.
- Concerned with scheduling disk accesses to optimize performance.
- I/O buffers are assigned at that level.

Typical Organization



- Primary interface with the environment outside the computer system.
- Deals with blocks of data
- Concerned with placement of those blocks in secondary storage
- Concerned with buffering those blocks in main memory.
- Does not understand the structure of data of files involved.



Records
Physical blocks in main memory buffers
Physical blocks in secondary storage (disk)
File Structure
Directory management
Access method
Blocking
Disk scheduling
User & program comands
Operation, File name
File manipulation functions
I/O
Free storage management
File allocation
User access control
File management concerns
Operating system concerns

Intermediate Conclusions

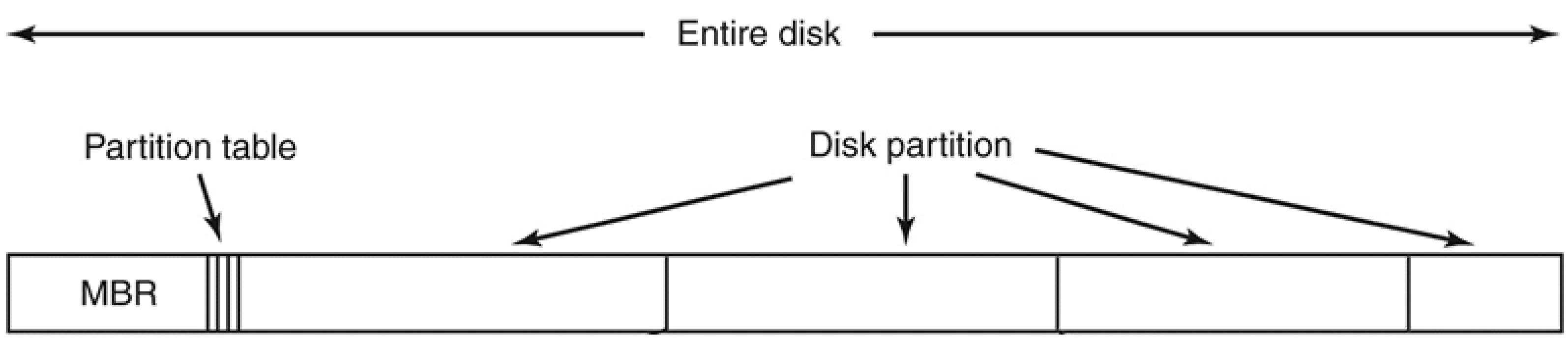
- Files are OS abstraction for storage, same as address space is OS abstraction for physical memory, and processes (& threads) are OS abstraction for CPU.
- So far we discussed files from user perspective.
- Next we will discuss the implementation.

Questions that need to be answered:

- How files and directories are stored?
- How disk space is managed?
- How to make everything work efficiently and reliably?

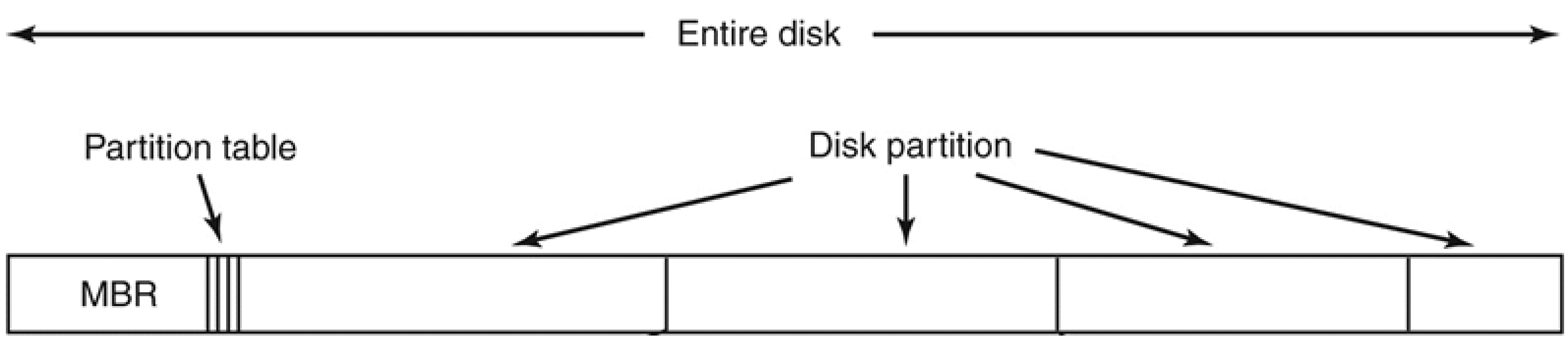
File System Layout

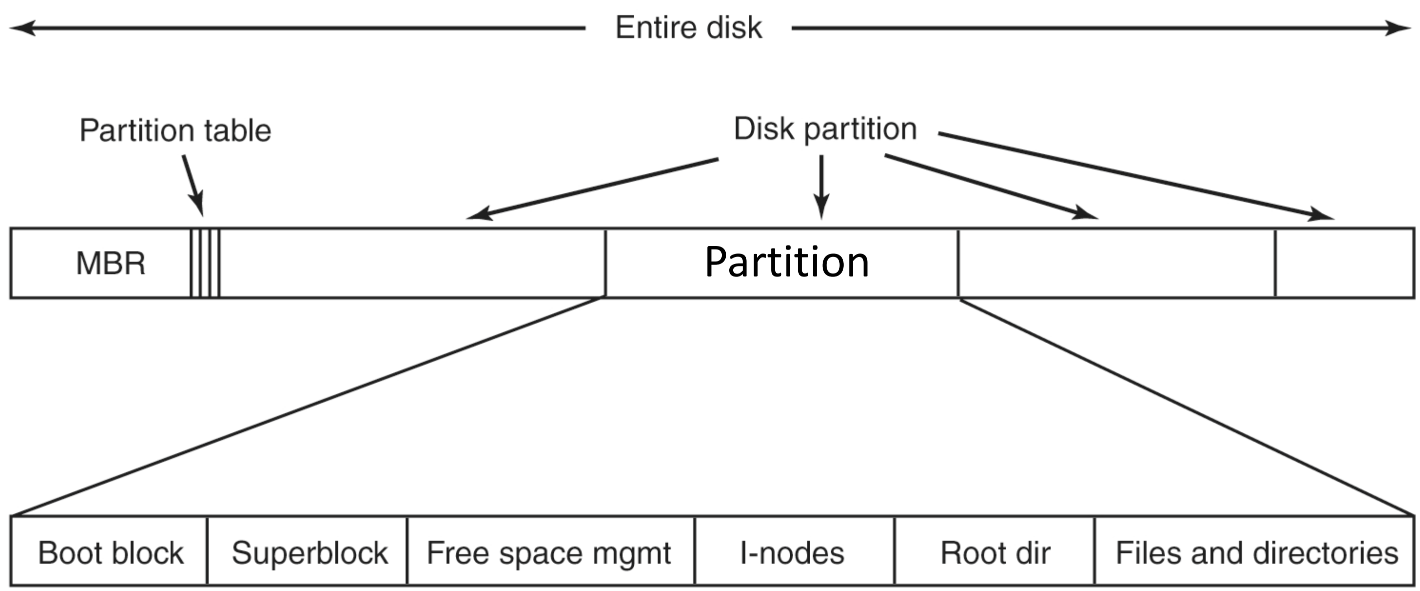
- Stored on disks
- Disks can have partitions with different file systems
- Sector 0 of the disk called MBR (Master Boot Record)
- MBR used to boot the computer
- The end of MBR contains the partition table

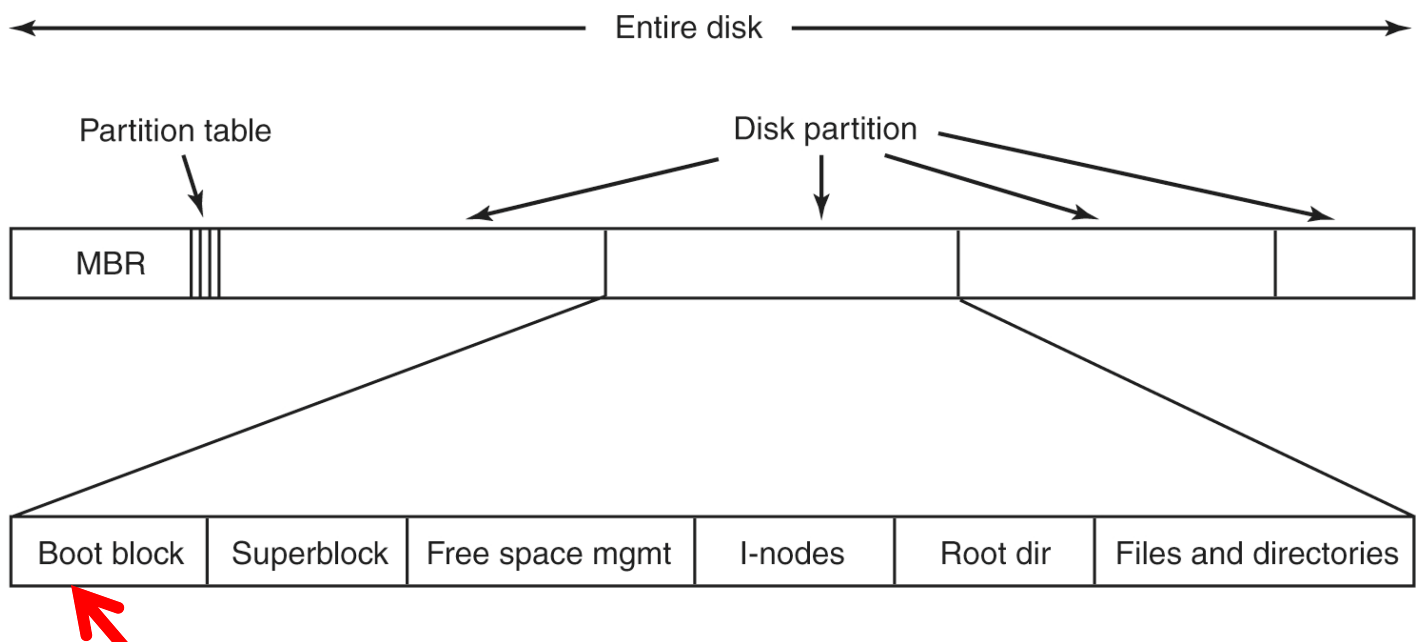


MBR and Partition Table

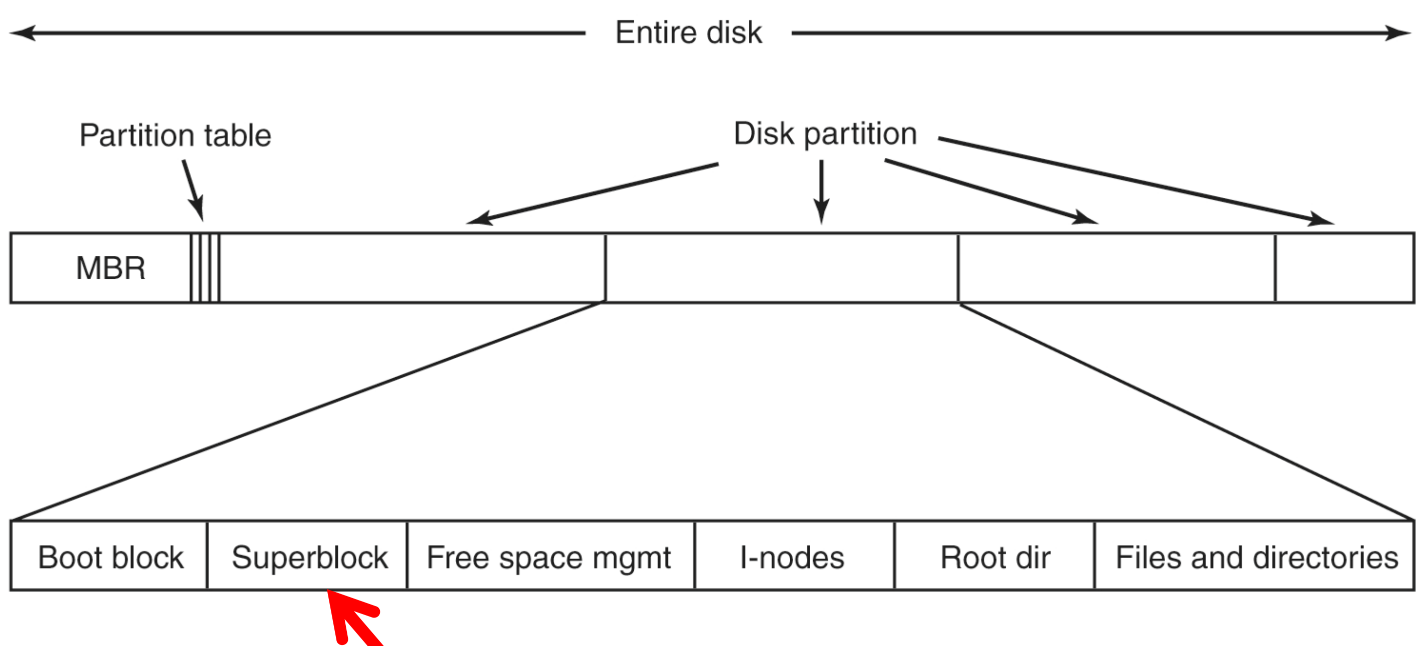
- Gives the starting and ending addresses of each partition
- One partition in the table is marked as active.
- When the computer is booted, BIOS executes MBR.
- MBR finds the active partition and reads its first block (called boot block) and executes it.
- Boot block loads the OS contained in that partition.



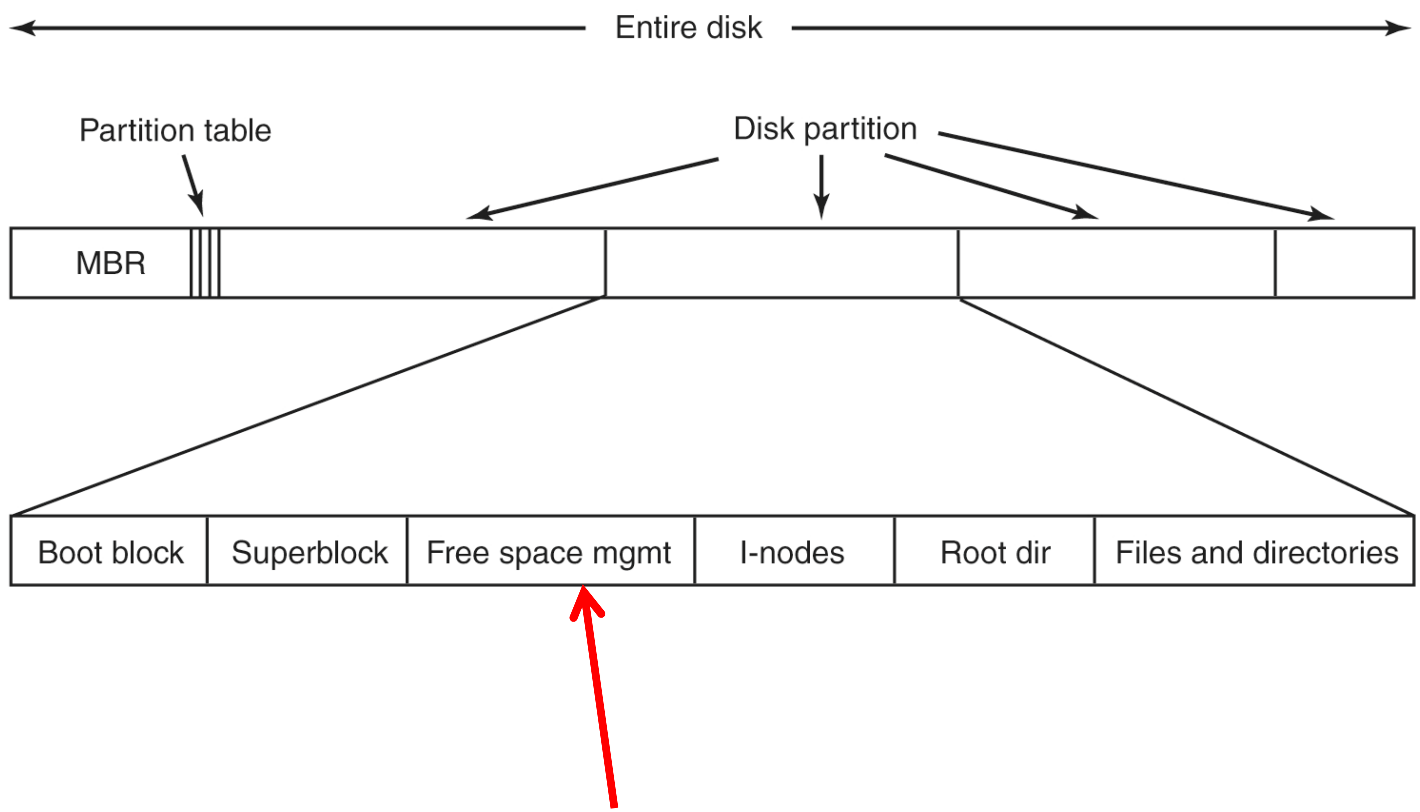

Entire disk
Partition table
Disk partition
MBR
Partition
Boot block
Superblock
Free space mgmt
I-nodes
Root dir
Files and directories



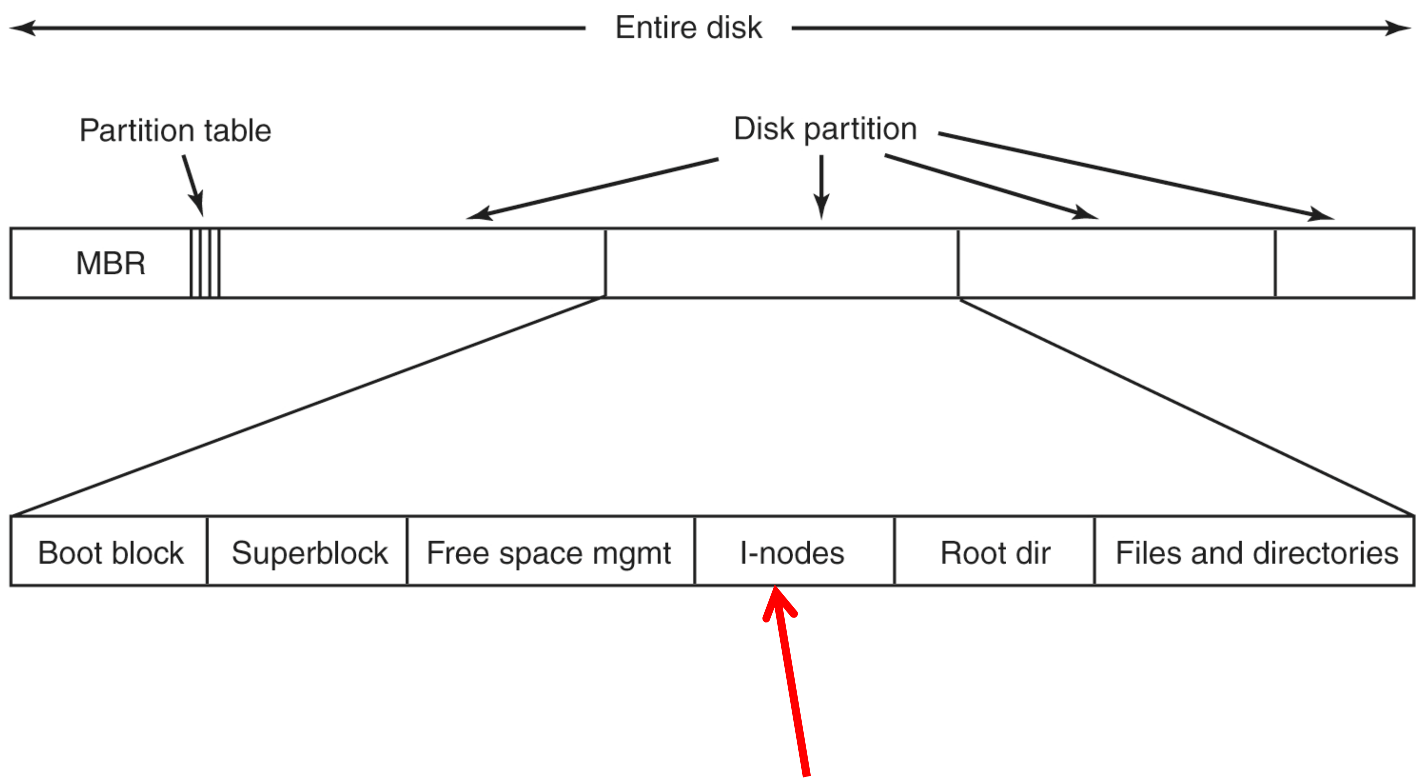
- Contains the bootable code (e.g. operating system)



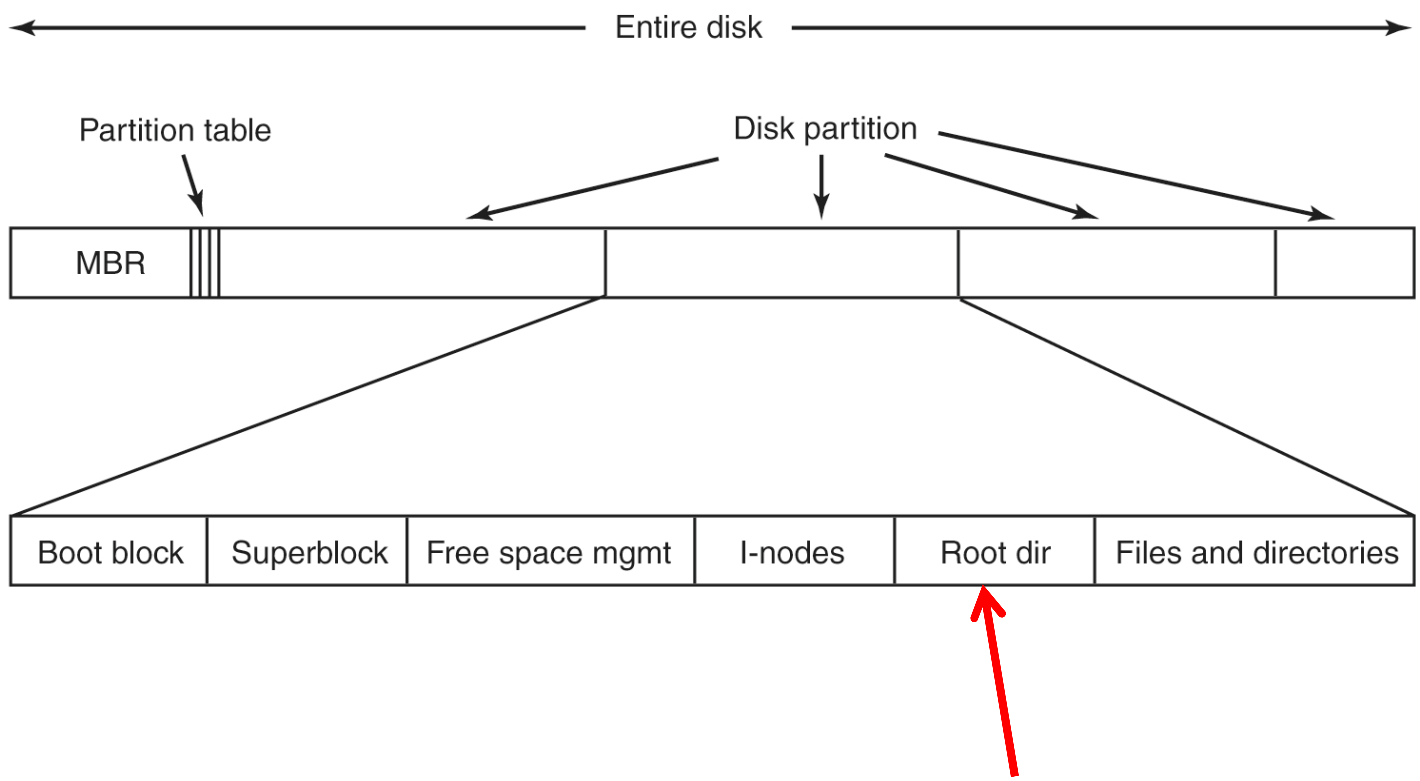
- Contains all key parameters about the file system
  - (e.g. filesystem type (magic, #-blocks, ..)
- Is read into memory when computer is booted or file system is touched.



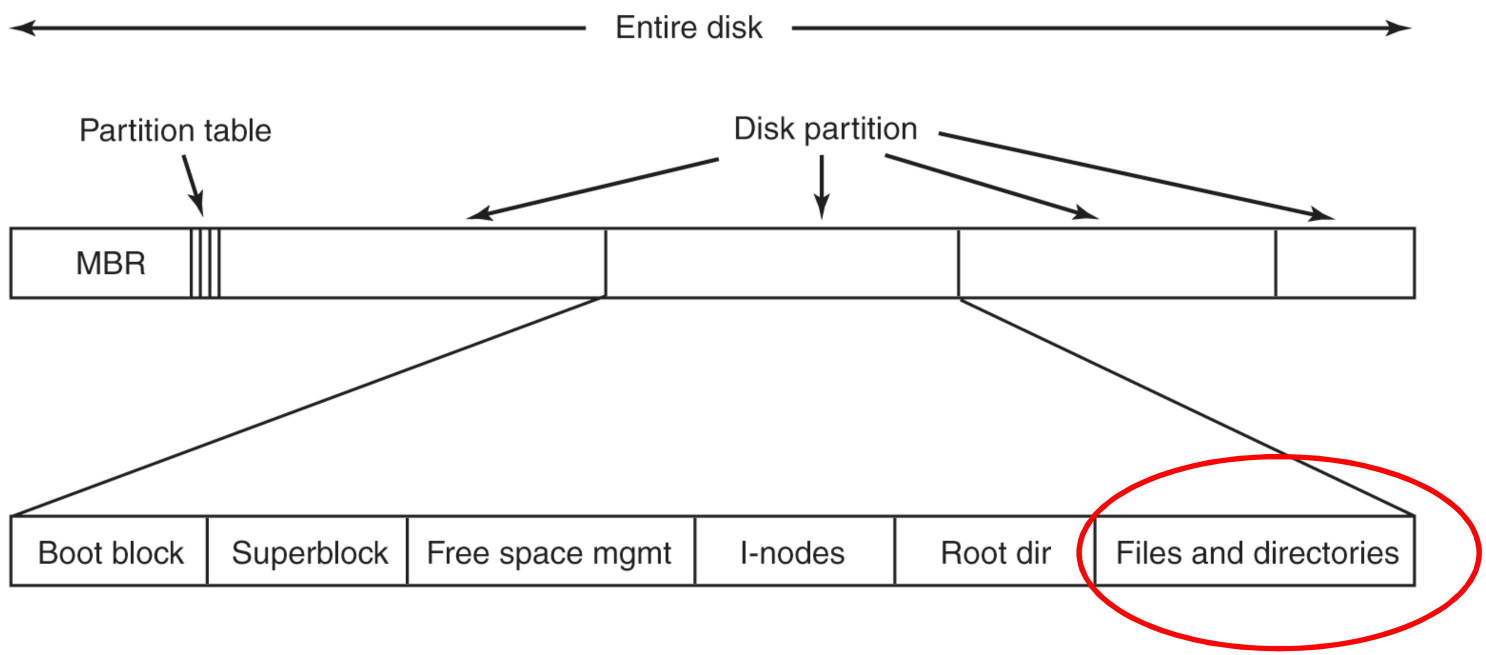
Bitmap or linked list



An array of data structures, one per file, telling about the file



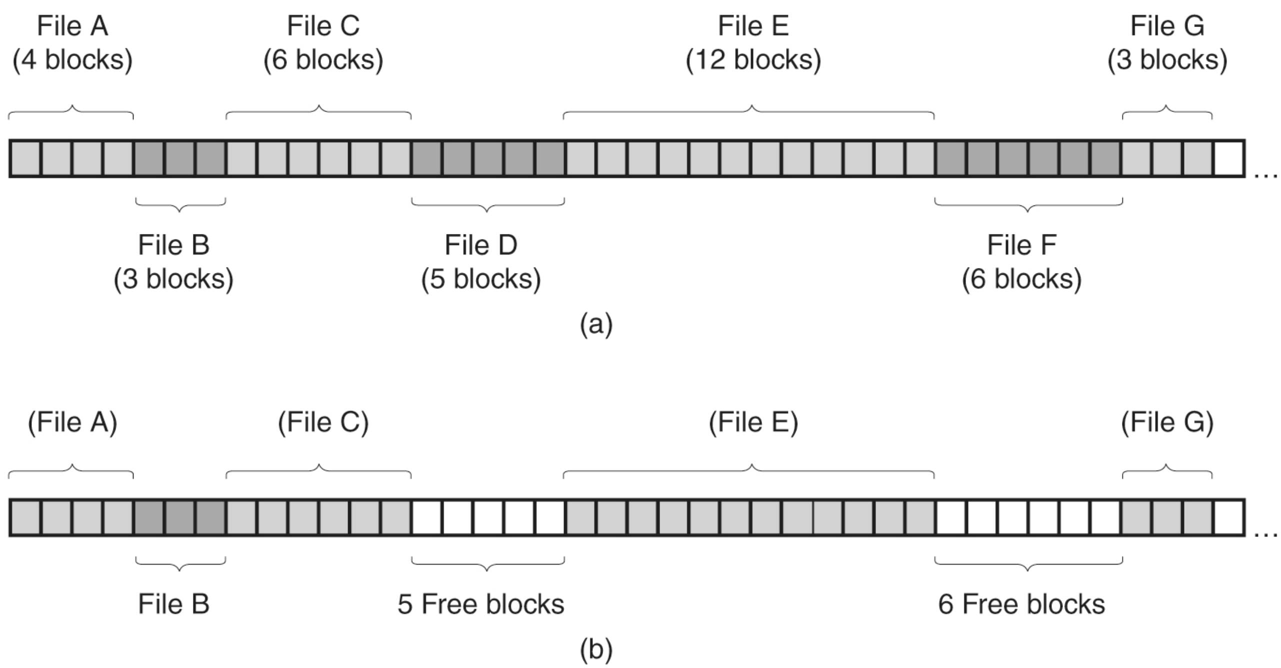
Root Directory .. Think '/' (as in '/home/user/franke')



Which disk blocks go with which files?

Implementing Files: Contiguous Allocation

- Store each file as a contiguous run of disk blocks



After files D and F were deleted

Implementing Files: Contiguous Allocation

+ Simple to implement:

  Need to remember starting block address of the file and number of blocks

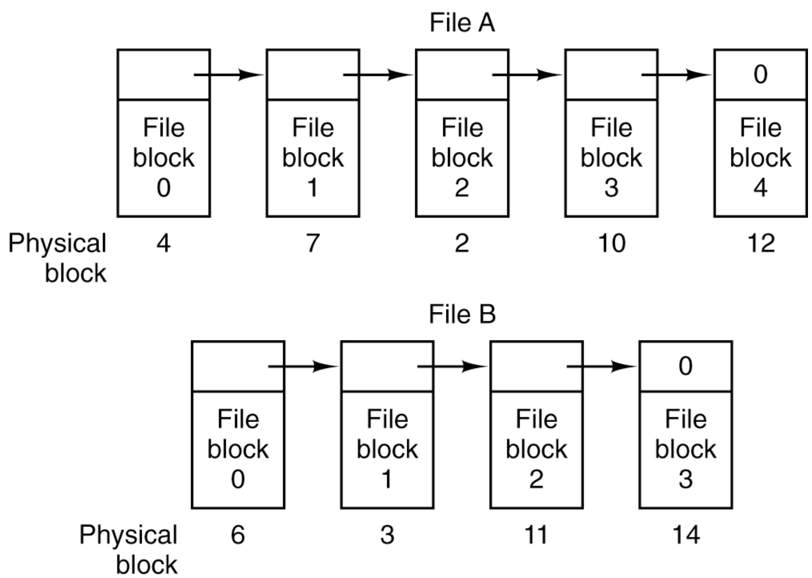
+ Read performance is excellent

  The entire file can be read from disk in a single operation.

- Disk becomes fragmented
- Need to know the final size of a file when the file is created

Implementing Files: Linked List Allocation

- Keep a file as a linked list of disk blocks
- The first word of each block is used as a pointer to the next one.
- The rest of the block is for data.



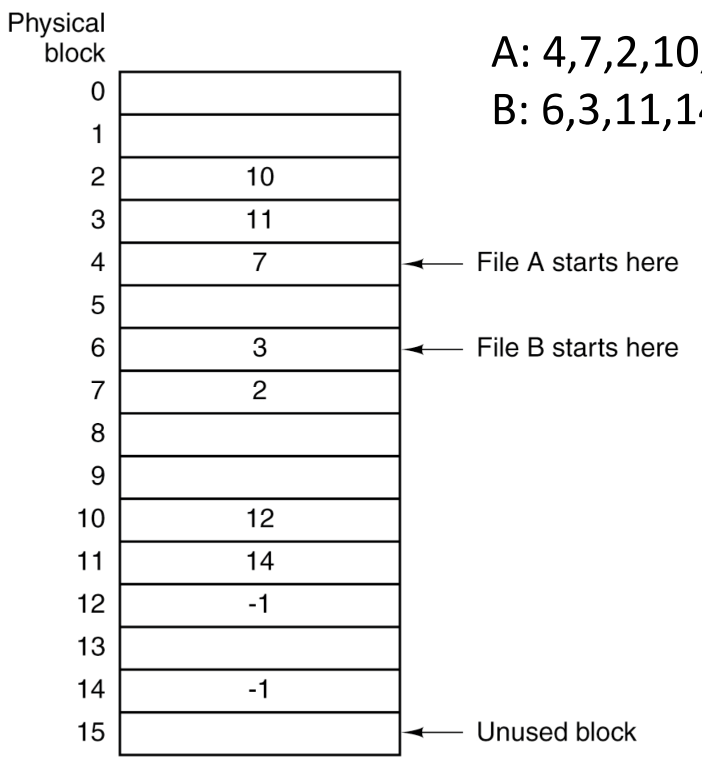
Implementing Files: Linked List Allocation

+ No (external) fragmentation
+ The directory entry needs just to store the disk address of the first block.

- Random access is extremely slow.
- The amount of data storage is no longer a power of two, because the pointer takes up a few bytes.

Implementing Files:
Linked List Allocation Using a Table in Memory

- Take the pointer word from each block and put it in a table in memory.



A: 4,7,2,10,12
B: 6,3,11,14

- This table is called: File Allocation Table (FAT)
- Directory entry needs to keep a single integer: (the start block number)

Main drawback: Does not scale to large disks because the table needs to be in memory all the time.

FAT12, FAT16, FAT32

Limits of FAT

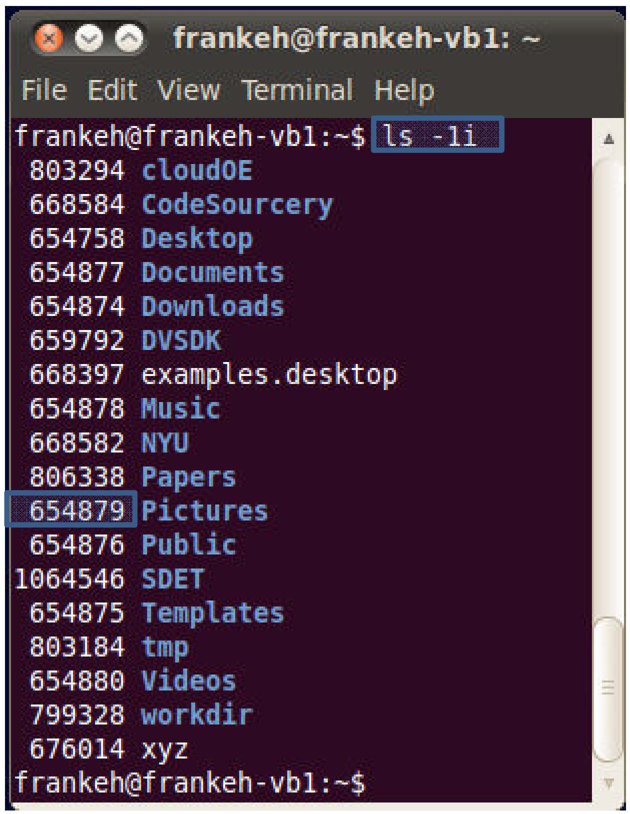
- Limits:
  http://en.wikipedia.org/wiki/Comparison_of_file_systems#cite_note-note-7-14

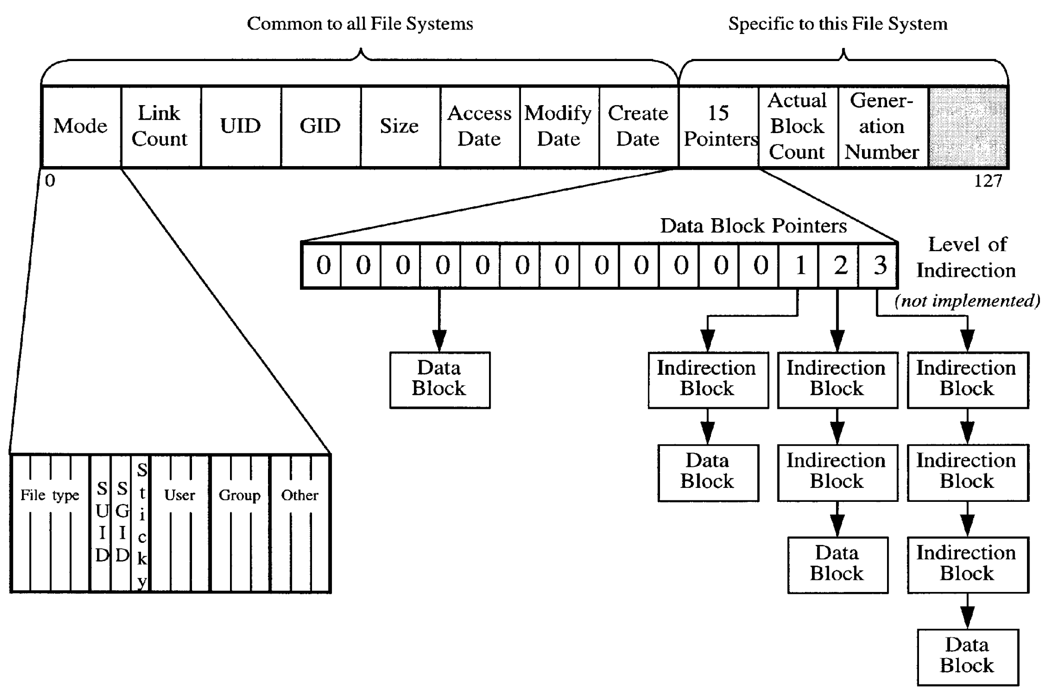
File system	Maximum filename length	Allowable characters in directory entries[5]	Maximum pathname length	Maximum file size	Maximum volume size[6]
FAT12	8.3 (255 UTF-16 code units with LFN)[14]	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \ | + , . ; = [] (lowcase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL[14][15]	No limit defined[16]	32 MB (256 MB)	32 MB (256 MB)
FAT16	8.3 (255 UTF-16 code units with LFN)[14]	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \ | + , . ; = [] (lowcase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL[14][15]	No limit defined[16]	2 GB (4 GB)	2 GB or 4 GB
FAT32	8.3 (255 UTF-16 code units with LFN)[14]	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \ | + , . ; = [] (lowcase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL[14][15]	No limit defined[16]	4 GB (256 GB[22])	2 TB[23] (16 TB)
ext2	255 bytes	Any byte except NUL[15] and /	No limit defined[16]	2 TB[6]	32 TB
ext3	255 bytes	Any byte except NUL[15] and /	No limit defined[16]	2 TB[6]	32 TB
ISO 9660:1988	Level 1: 8.3, Level 2 & 3: ~ 180	Depends on Level[51]	~ 180 bytes?	4 GB (Level 1 & 2) to 8 TB (Level 3)[52]	8 TB[53]
XFS	255 bytes[57]	Any byte except NUL[15]	No limit defined[16]	8 EB[58]	8 EB[58]
ZFS	255 bytes	Any Unicode except NUL	No limit defined[16]	16 EB	16 EB

Implementing Files: I-nodes

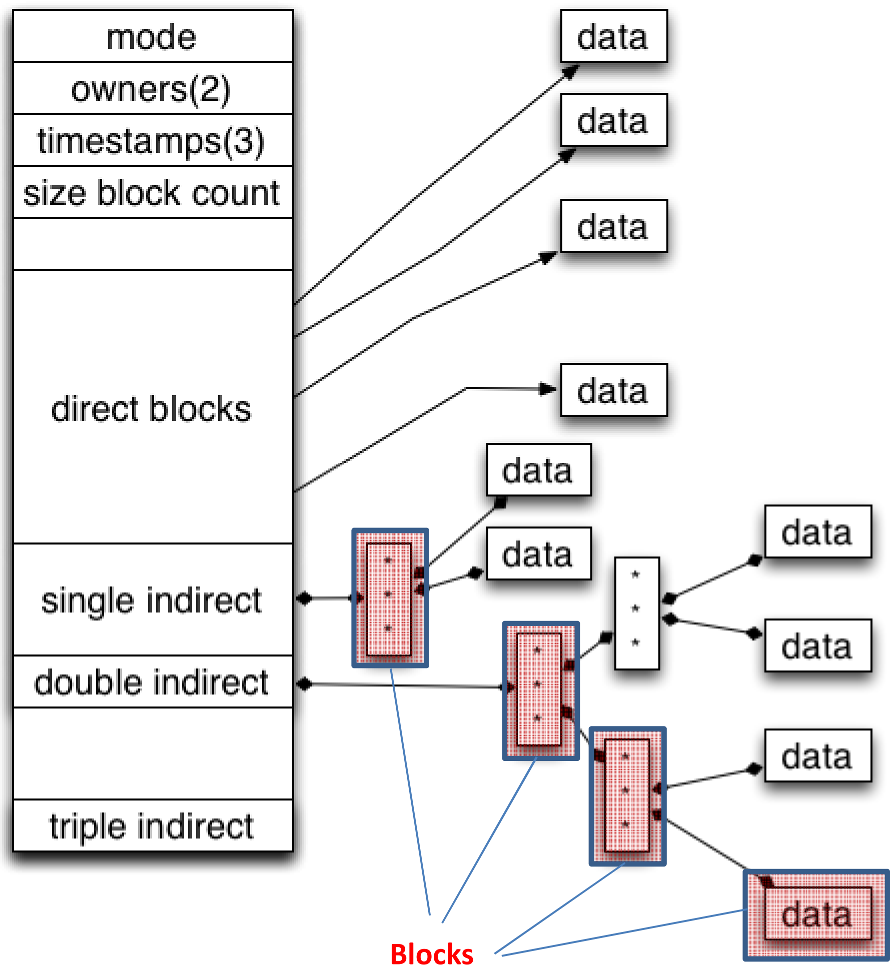
- A data structure associated with each file
- Lists the attributes and disk addresses of the file blocks
- Need only be in memory when the corresponding file is open
- Aka FILE META DATA

i-node=#





Implementing Files: I-nodes



Properties:

- Small file access fast
- Everything a block
- Huge files can be presented

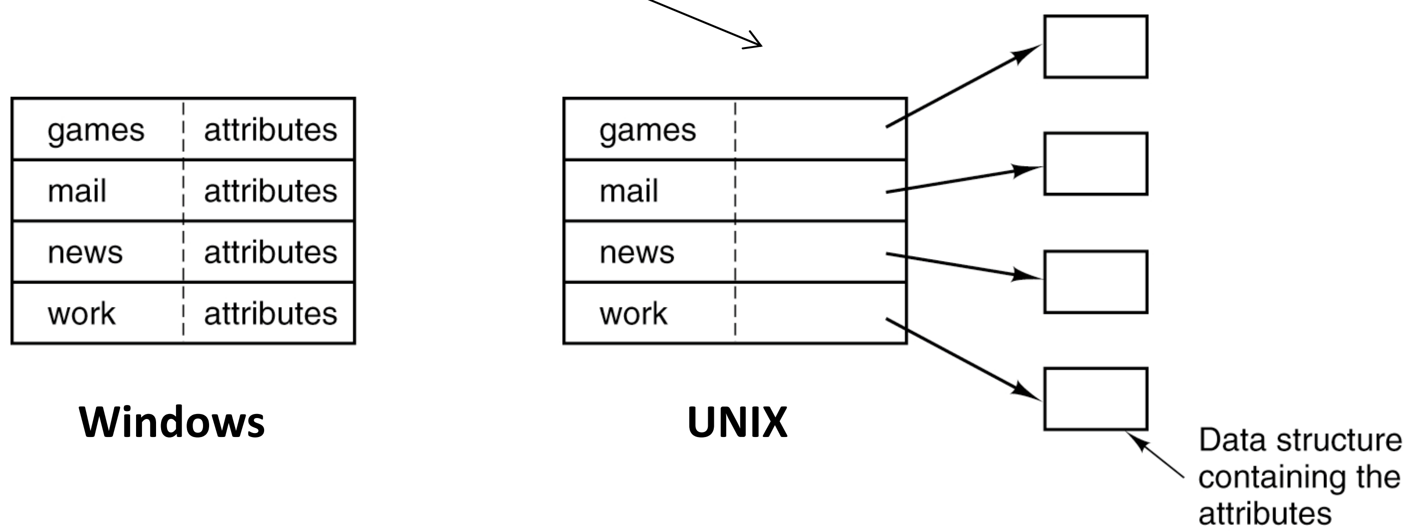
Implementing Directories

Example:

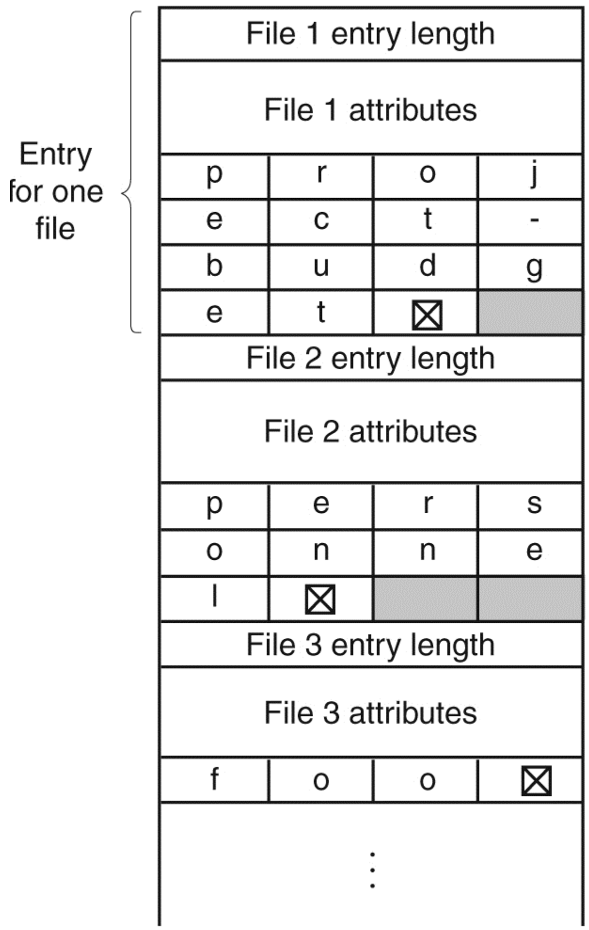
- Disk address of the file (in contiguous scheme)
- Number of the first block (in linked-list schemes)
- Number of the i-node,

Implementing Directories

- Where the attributes should be stored?
  - Directly in the directory entry
  - In the i-nodes



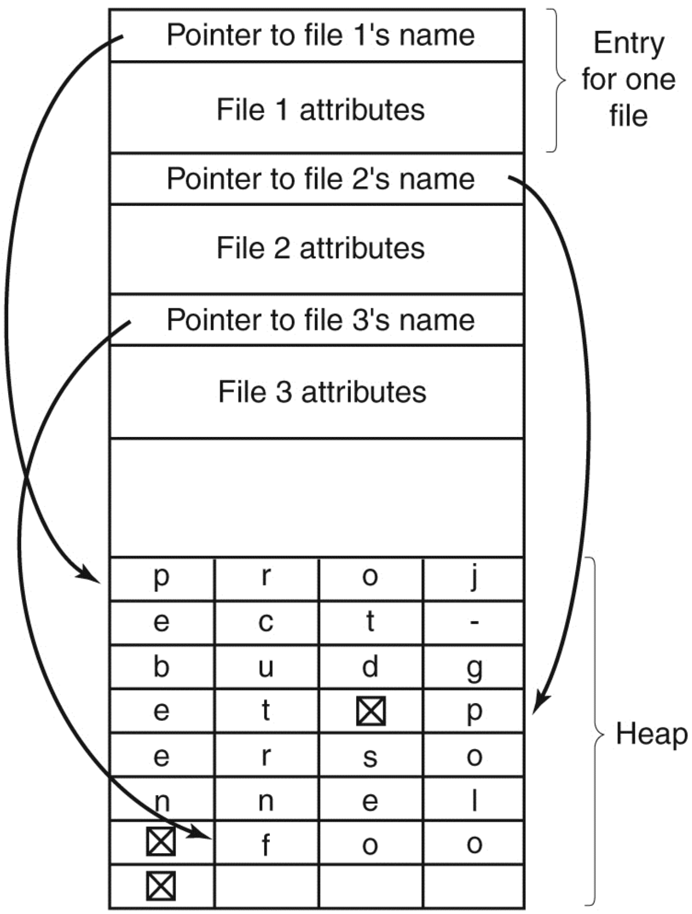
Implementing Directories: Variable-Length Filenames



Disadvantages:

- Entries are no longer of the same length.
- Variable size gaps when files are removed
- A big directory may span several pages which may lead to page faults.

Implementing Directories: Variable-Length Filenames



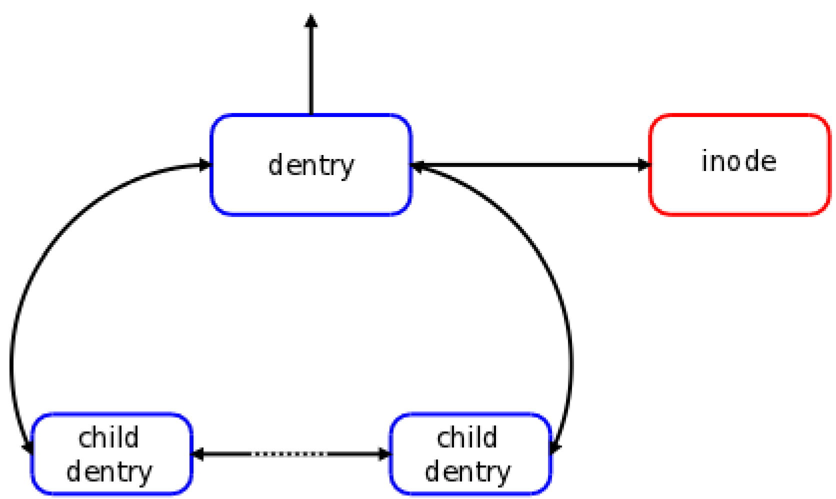
- Keep directory entries fixed length
- Keep filenames in a heap at the end of the directory.
- Page faults can still occur while accessing filenames.

Implementing Directories

- For extremely long directories, linear search can be slow.
  - Hashing can be used
  - Caching can be used

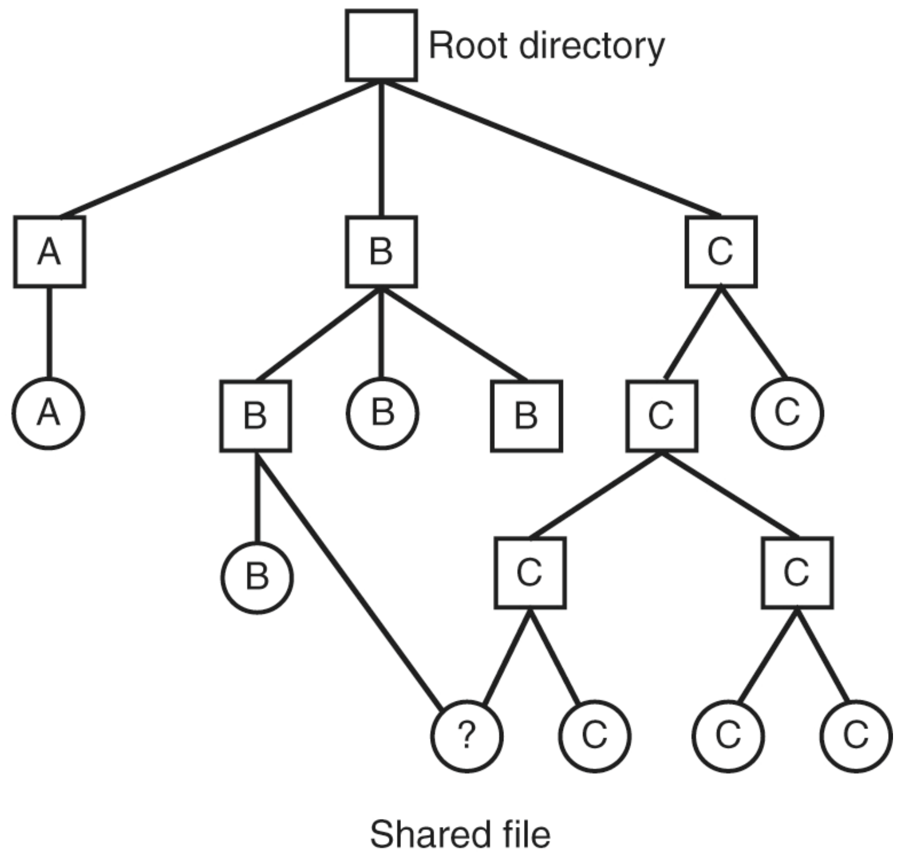
Speeding up

- Continuously going to the disk is expensive
  e.g. "/home/frankeh/nyu/best/class/ever"
- Root -> home -> franke -> nyu -> best -> class -> ever
  multiple dentry (directories need to be read)
- *DENTRY cache*



Shared Files

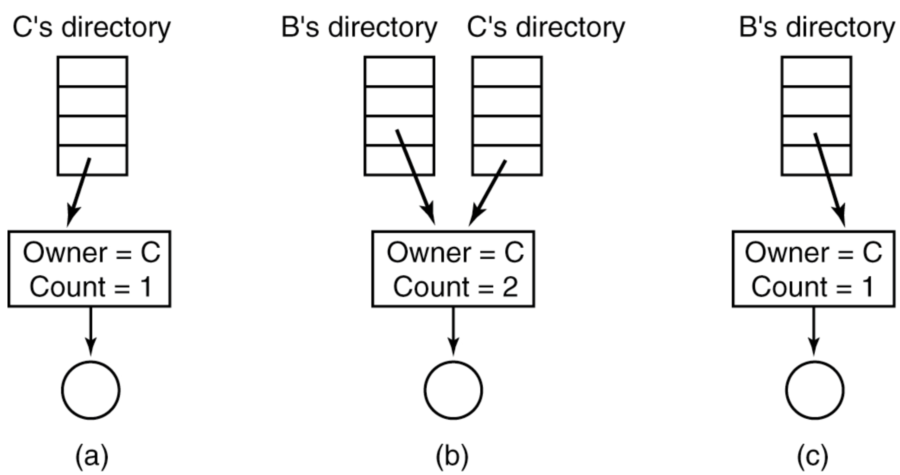
- Appear simultaneously in different directories



Shared Files: Method 1

- Disk blocks are not listed in directories but in a data structure associated with the file itself (e.g. i-nodes in UNIX).
- Directories just point to that data structure.
- This approach is called: static linking

Problematic Scenario



Shared Files: Method 2

- Have the system create a new file (of type LINK). This new file contains the path name of the file to which it is linked.
- This approach is called: symbolic linking
- The main drawback is the extra overhead.

Log-Structured File Systems

- Disk seek time does not improve as fast as relative to CPU speed, disk capacity, and memory capacity.
- Disk caches can satisfy most requests

SO: In the future, most disk accesses will be writes

Log-Structured File Systems

- Structure the entire file as a log
- Periodically (or when in need):
  - All pending writes buffered in memory are collected into a single segment
  - The segment is written in disk in contiguous space at the end of the log
- i-nodes now scattered over the disk
- An i-node map, indexed by i-number, is maintained.
- The map is kept on disk and is also cached.
- A cleaner thread scans log to compact it and discard unneeded information.
- E.g. file created then deleted.

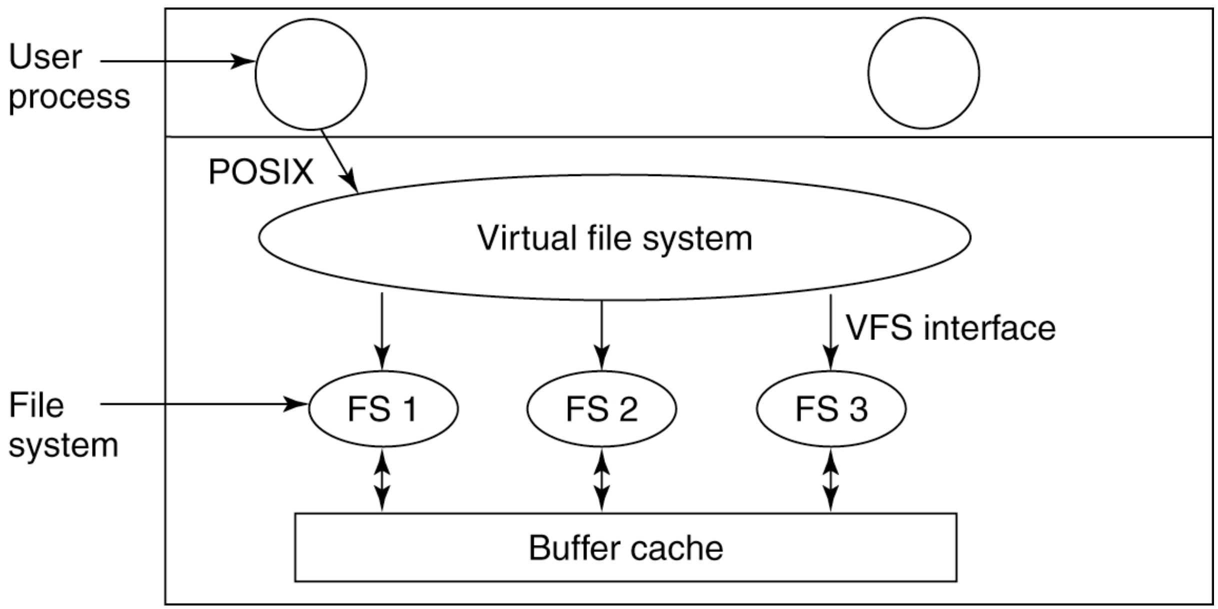
- Disk is circular buffer, where writer add new segments at front and cleaner thread removing old from the back.

Journaling File System

- Keep a log of what the file system is going to do before it does it.
- Commit log to disk (now we have a record)
- If the system crashes before it is done, then after rebooting the log is checked and the job is finished.
- Example: Microsoft NTFS, Linux ext3
- The logged operations must be idempotent (i.e. can be repeated as often as necessary without harm).
- Consider "rm /home/franke/nofreelunch"
  - Remove the file from its directory
  - Release the i-node
  - Release all the disk blocks to the free block pool

Virtual File Systems

- Integrating multiple file systems into an orderly structure.



mount

Example: Linux Internals

current->namespace->list->mnt_sb *see Figure 3*

```
struct super_block {
  struct list_head            s_list;
  unsigned long               s_blocksize;
  struct file_system_type    *s_type;
  struct super_operations    *s_op;
  struct semaphore            s_lock;
  int                         s_need_sync_fs;
  struct list_head            s_dirty;
  struct block_device        *s_bdev;
  ...
};
```

doubly linked list of all mounted filesystems

see Figure 2

```
file_systems

struct file_system_type {
  const char *name;
  int fs_flags;
  struct super_block *get_sb;
  void (*kill_sb);
  struct module *owner;
  struct file_system_type *ne
  struct list_head fs_supers;
}
```

```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*read_inode)(struct inode *);
  void (*write_inode)(struct inode *, int);
  int  (*sync_fs)(struct super_block *sb, int wait);
  ...
};
```

```
struct file_system_type {
  const char *name;
  int fs_flags;
  struct super_block *get_sb;
  void (*kill_sb);
  struct module *owner;
  struct file_system_type *next;
  struct list_head fs_supers;
}
```

Example: Linux Internals

current->namespace->list →

```
struct vfsmount {
  struct list_head mnt_hash;
  struct vfsmount *mnt_parent;
  struct dentry *mnt_mountpoint;
  struct dentry *mnt_root;
  struct super_block *mnt_sb;
  struct list_head mnt_mounts;
  struct list_head mnt_child;
  atomic_t mnt_count;
  int mnt_flags;
  char *mnt_devname;
  struct list_head mnt_list;
}
```

mounted filesystem list

Example: Linux Internals

```
struct inode {
  unsigned long               i_ino;
  umode_t                     i_mode;
  uid_t                       i_uid;
  struct timespec             i_atime;
  struct timespec             i_mtime;
  struct timespec             i_ctime;
  unsigned short              i_bytes;
  struct inode_operations    *i_op;
  struct file_operations     *i_fop;
  struct super_block         *i_sb;
  ...
}
```

```
struct inode_operations {
  int (*create)(struct inode *, struct dentry *,
                struct nameidata *);
  struct dentry *(*lookup)(struct inode *,
                           struct dentry *,
                           struct nameidata *);
  int (*mkdir)(struct inode *, struct dentry *, int);
  int (*rename)(struct inode *, struct dentry *,
                struct inode *, struct dentry *);
  ...
}
```

```
struct file_operations {
  struct module *owner;
  ssize_t (*read)(struct file *, char __user *,
                  size_t, loff_t *);
  ssize_t (*write)(struct file *, const char __user *,
                   size_t, loff_t *);
  int (*open)(struct inode *, struct file *);
  ...
}
```

Example: fs creation and mounting

- Create a new filesystem on a particular device:
  - Allocate inodes and blocks

```
# mkfs -t ext3 /dev/sda6
mke2fs 1.42 (29-Nov-2011)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
1120112 inodes, 4476416 blocks
223820 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=0
137 block groups
32768 blocks per group, 32768 fragments per group
8176 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

- Associating a device with a particular filesystem and providing an access point "mount point"

```
mount -t type device directory
```

```
~]# mount -t vfat /dev/sdc1 /media/flashdisk
```

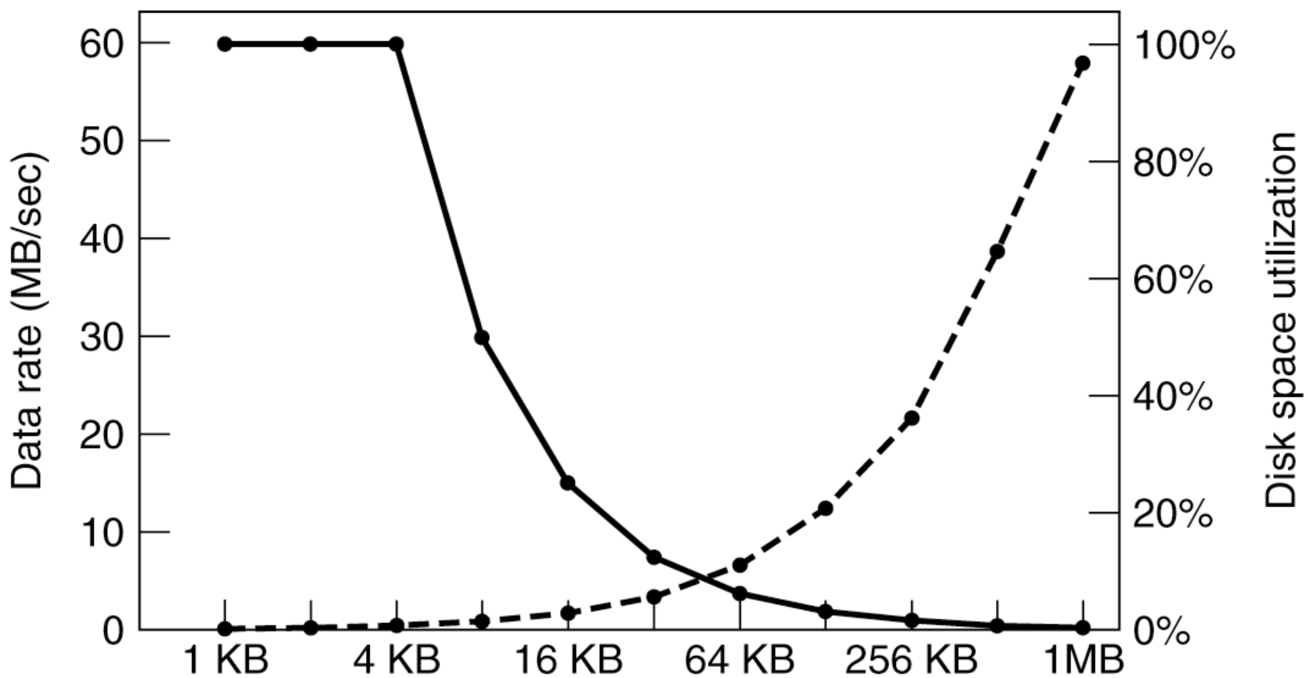
Type	Description
ext2	The **ext2** file system.
ext3	The **ext3** file system.
ext4	The **ext4** file system.
iso9660	The **ISO 9660** file system. It is commonly used by optical media, typically CDs.
jfs	The **JFS** file system created by IBM.
nfs	The **NFS** file system. It is commonly used to access files over the network.
nfs4	The **NFSv4** file system. It is commonly used to access files over the network.
ntfs	The **NTFS** file system. It is commonly used on machines that are running the Windows operating system.
udf	The **UDF** file system. It is commonly used by optical media, typically DVDs.
vfat	The **FAT** file system. It is commonly used on machines that are running the Windows operating system, and on certain digital media such as USB flash drives or floppy disks.

Disk Space Management

- All file systems chop files up into fixed-size blocks that need not be adjacent.
- Block size:
  - Too large -> we waste space
  - Too small -> we waste time

Access time for a block is completely dominated by the seek time and rotational delay.
So … The more data are fetched the better.

(dotted)



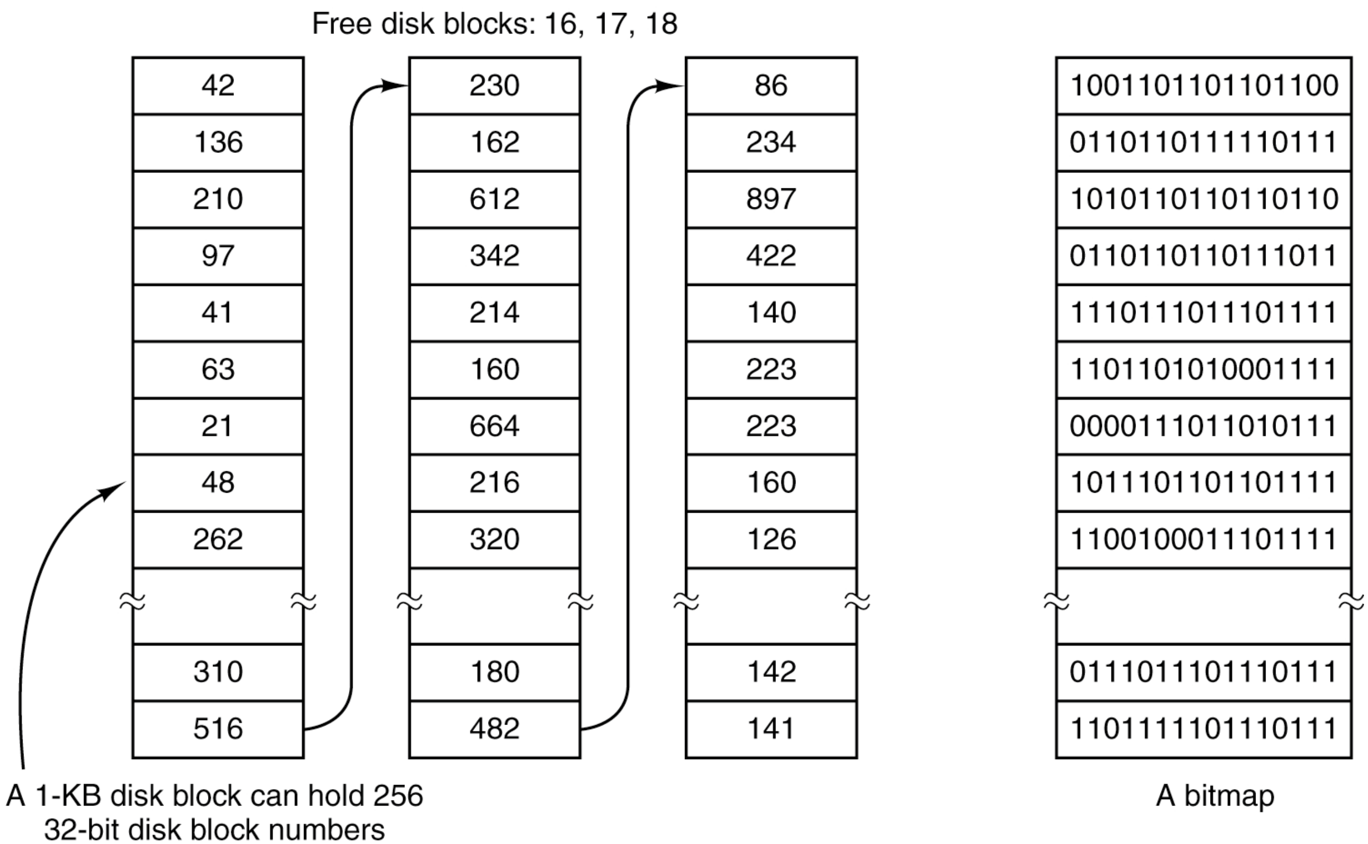
Block Size

Disk Space Management: Keeping Track of Free Blocks

- **Method 1**: Linked list of disk blocks, with each block holding as many free disk block numbers as possible.

- **Method 2**: Using a bitmap

Disk Space Management: Keeping Track of Free Blocks

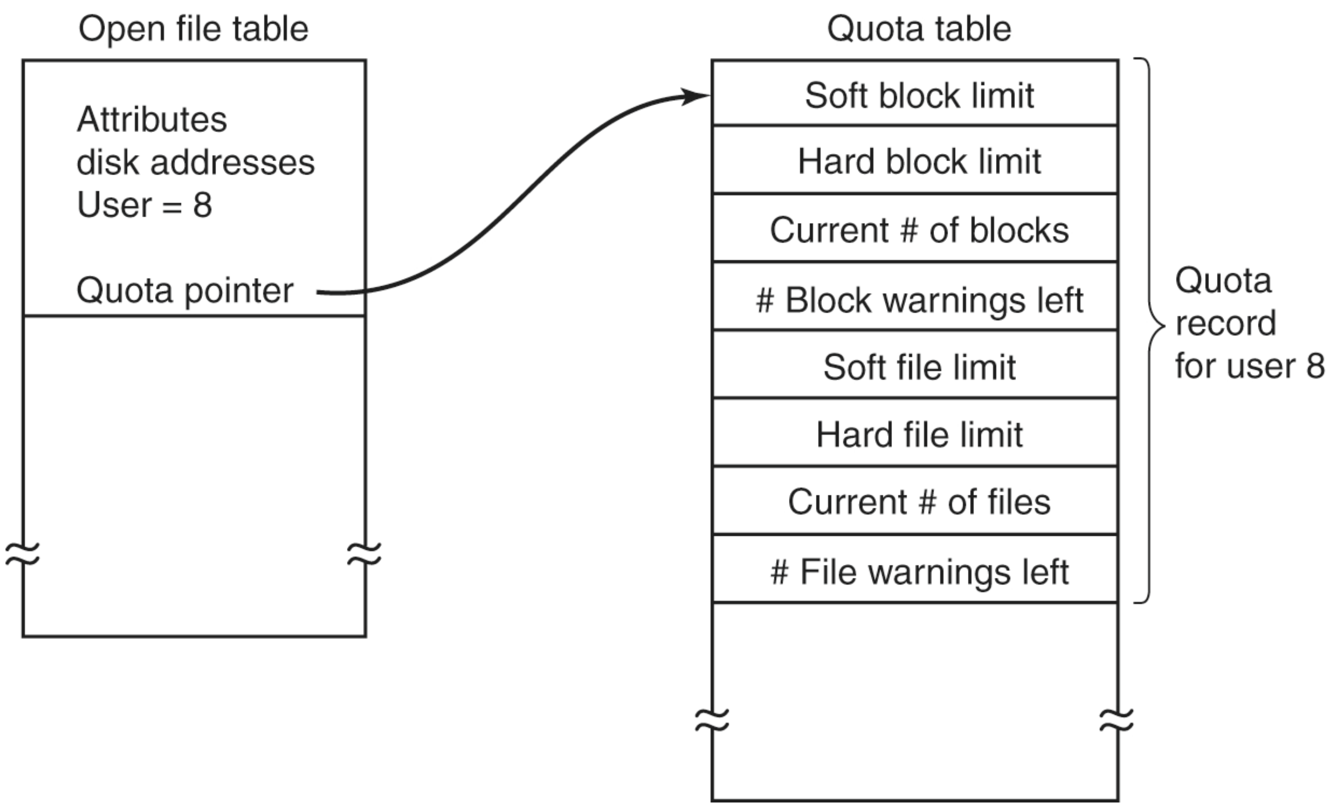


Free blocks are holding the free list.

Disk Space Management: Disk Quotas

- When a user opens file
  - The attributes and disk addresses are located
  - They are put into an open file table in memory
  - A second table contains the quota record for every user with a currently open file.

Disk Space Management: Disk Quotas



File System Backups

- It is usually desirable to back up only specific directories and everything in them than the entire file system.
- Since immense amounts of data are typically dumped, it may be desirable to compress them.
- It is difficult to perform a backup on an active file system.
- Incremental dump: backup only the files that have been modified from last full-backup

File System Backups: Physical Dump

- Starts at block 0 of the disk
- Writes all the disk blocks onto the output tape (or any other type of storage) in order.
- Stops when it has copied the last one.

\+ Simplicity and great speed

\- Inability to skip selected directories and restore individual files.

File System Backups: Logical Dump

- Starts at one or more specified directories
- Recursively dumps all files and directories found there and have changed since some given base date.

File System Consistency

- Two kinds of consistency checks
  - Blocks
  - Files

File System Consistency: Blocks

- Build two tables, each one contains a counter for each block, initially 0
- Table 1: How many times each block is present in a file
- Table 2: How many times a block is present in the free list
- A consistent file system: each block has 1 either in the first or second table

File System Consistency: Blocks

Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a)

Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

(b)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1

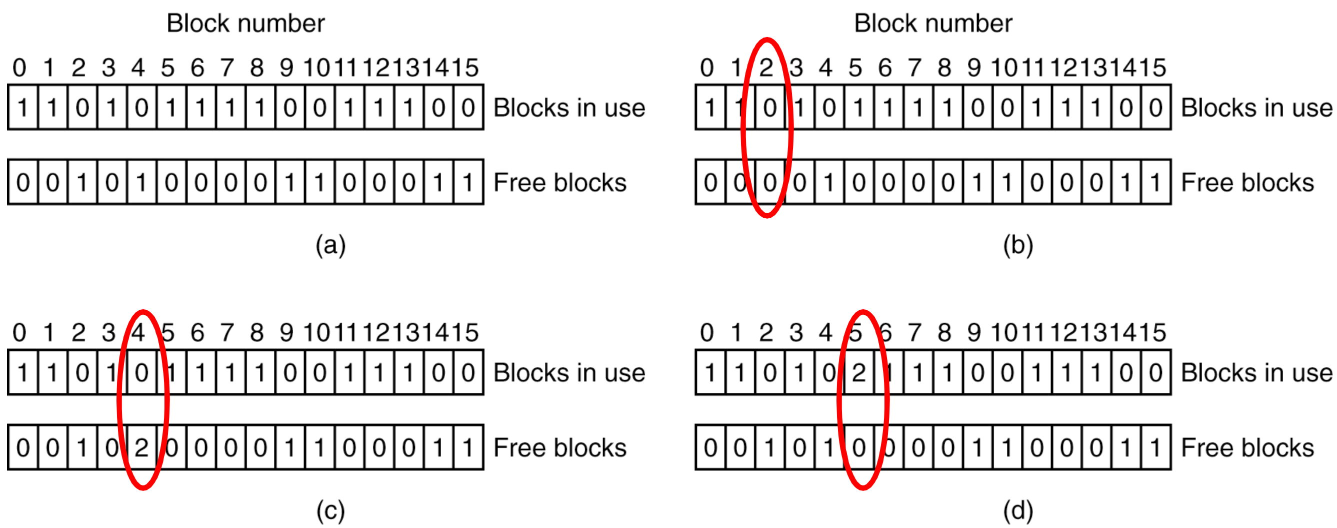
(c)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(d)

File System Consistency: Blocks

Add the block to the free list



Rebuild the free list

Allocate a free block, make a copy of that block and give it to the other file.

File System Consistency: Files

- Table of counters; a counter per file
- Counts the number of that file's usage count.
- Compares these numbers in the table with the counts in the i-node of the file itself.
- Both counts must agree.

File System Consistency: Files

- Two inconsistencies:
  - count of i-node > count in table
  - count of i-node < count in table
- Fix: set the count in i-node to the correct value

File System Performance

- Caching:
  - Block cache: a collection of blocks kept in memory for performance reasons
- Block Read Ahead:
  - Get blocks into the cache before they are needed
- Reducing Disk Arm Motion:
  - Putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder
- Defragmentation

Conclusions

- Files and file system are major parts of an OS.
- Files and File system are the OS way of abstracting storage.
- There are different ways of organizing files, directories, and their attributes.
  - However VFS is a unifying way to represent a common API