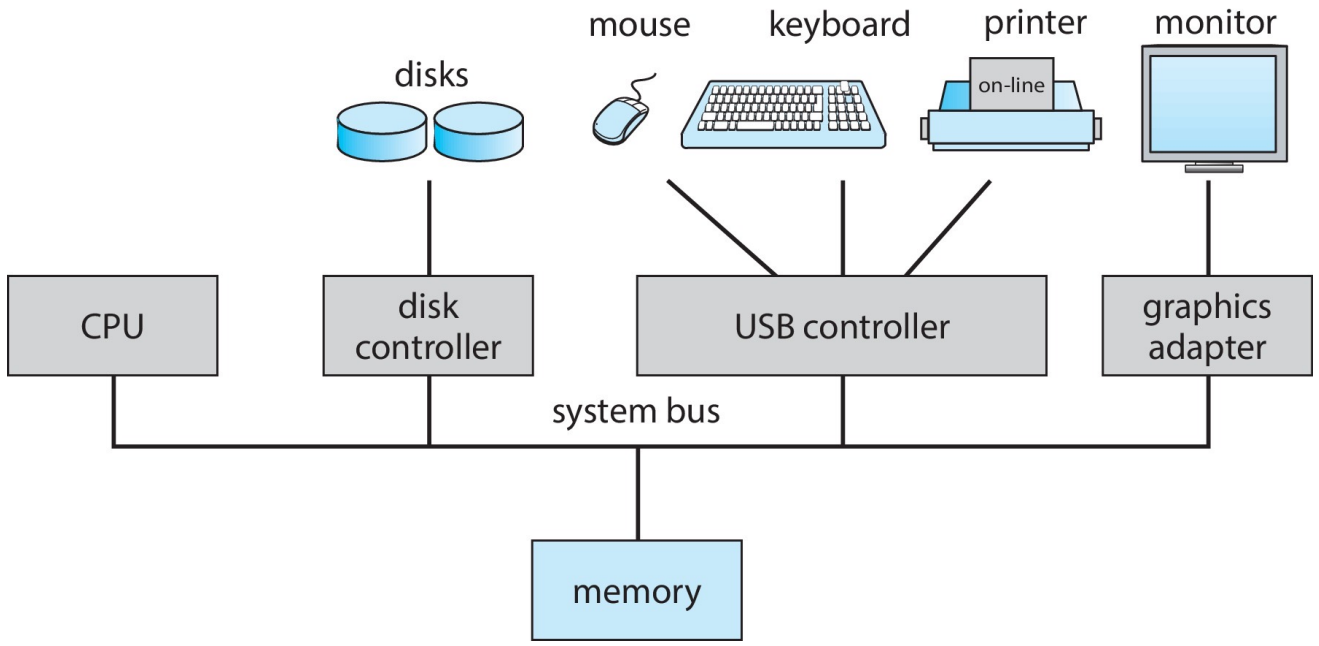
File systems

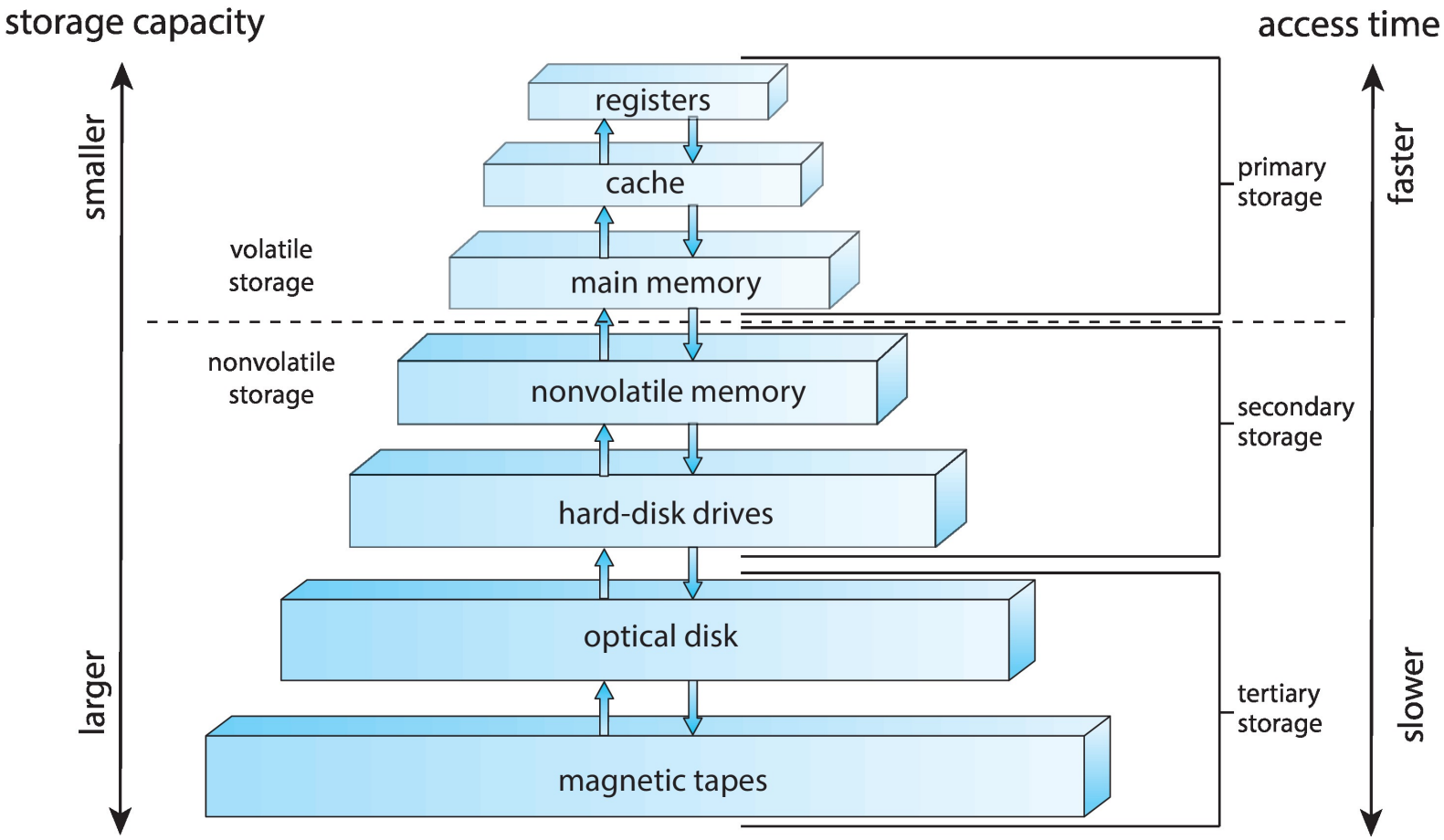
- Secondary storage devices
  - Types of storage devices
  - Disk structure
  - Address mapping
  - Disk formatting
  - Boot time
- Files
- Directories
- File system implementation
- File system example: MS-DOS

Computer System Hardware

- Computer-system hardware
  - One or more CPUs, device controllers connect through common bus providing access to the memory and other I/O devices



Storage-device hierarchy



File-system Management

- OS abstracts physical properties to logical storage unit - file
- File-System management
  - Files usually organized into directories
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage

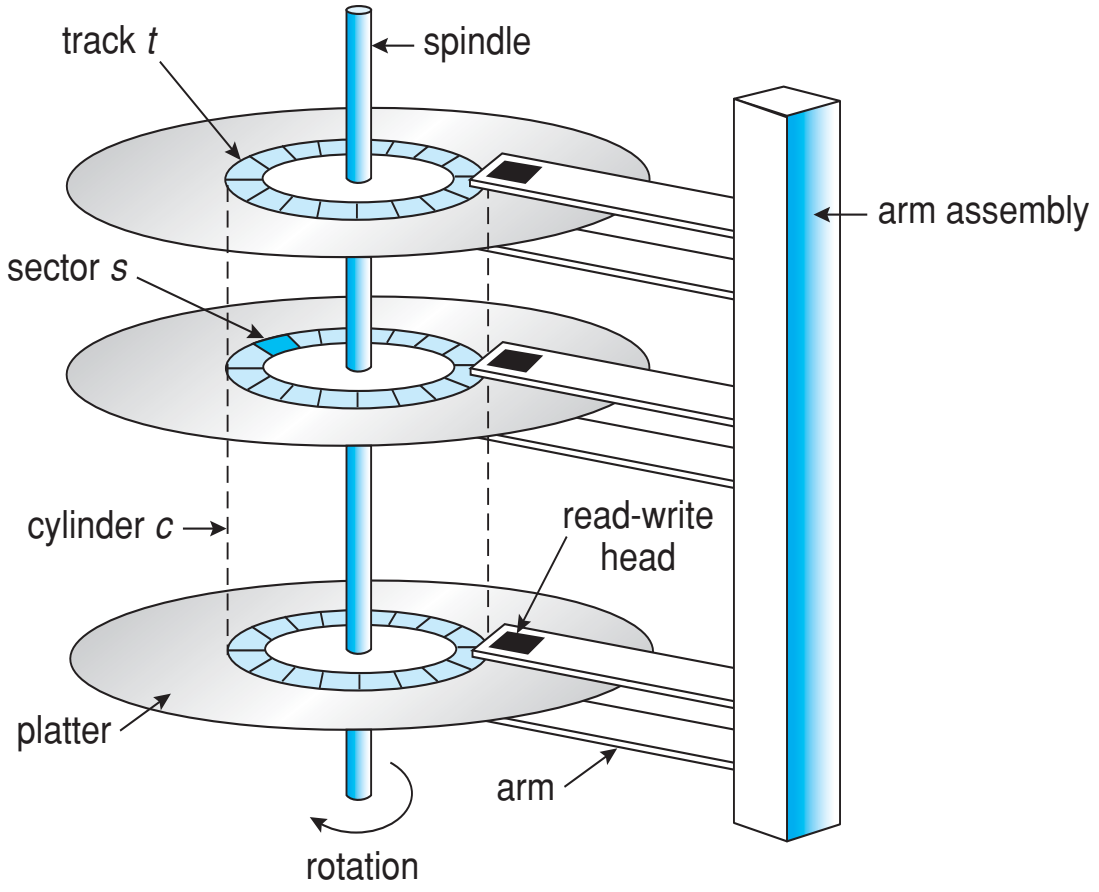
Mass storage devices

- Bulk of secondary storage for modern computers is hard disk drives (HDDs) and nonvolatile memory (NVM) devices, mainly solid-state disks (SSDs)

Hard disk drives

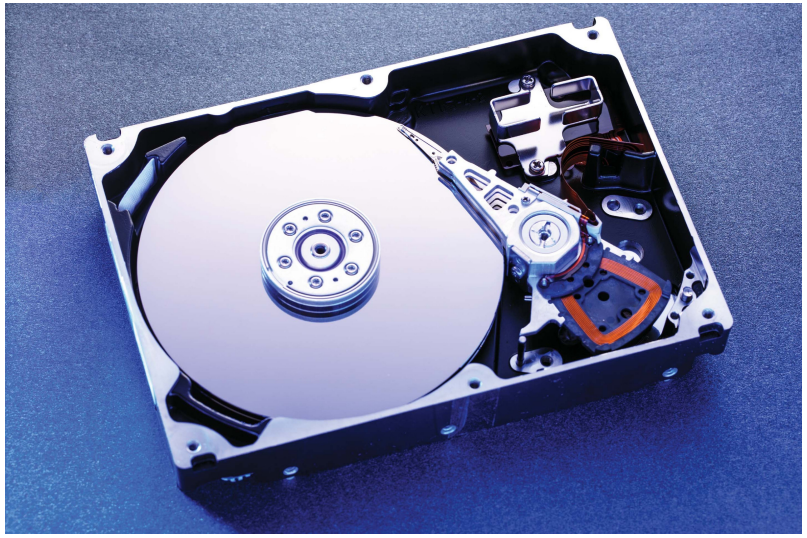
HDDs spin platters of magnetically-coated material under moving read-write heads

- Drives rotate at 60 to 250 times per second
- Transfer rate is rate at which data flow between drive and computer
- Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency)
- Head crash results from disk head making contact with the disk surface -- Not good!



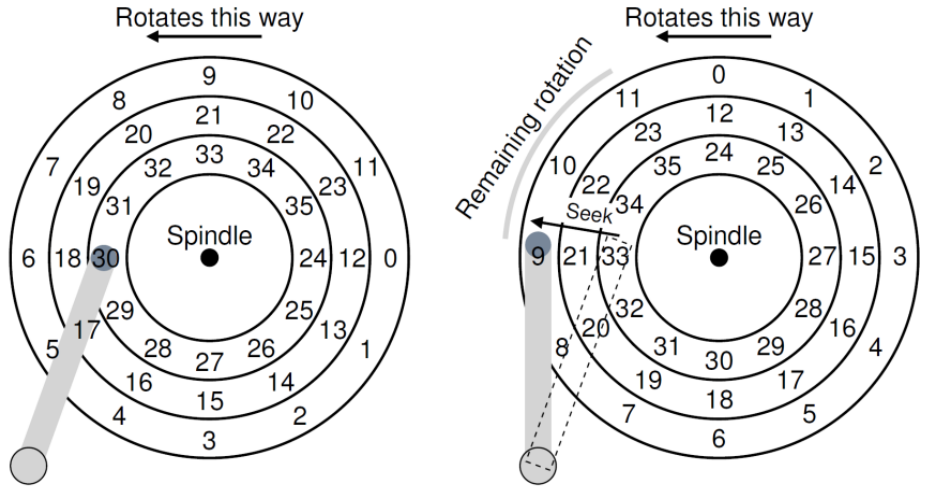
Hard Disk Drives

- Platters range from .85” to 14” (historically)
  - Commonly 3.5”, 2.5”, and 1.8”
- Range from 30GB to 3TB per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - Latency based on spindle speed
    - 1 / (RPM / 60) = 60 / RPM
  - Average latency = ½ latency

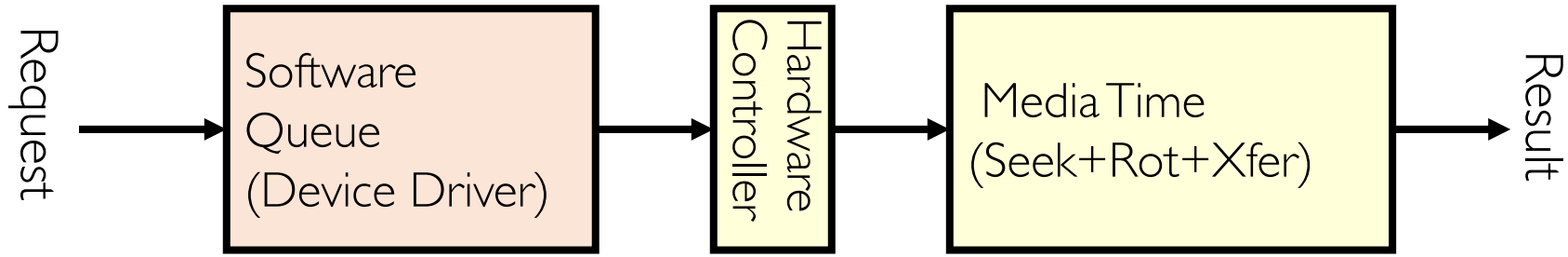


Performance of HDDs

- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector) under r/w head



Request Time = Queueing Time + Controller Time + Seek + Rotational + Transfer



Example of Current HDDs

- Seagate Exos X18 (2020)
  - 18 TB hard disk
    - 9 platters, 18 heads
    - Helium filled: reduce friction and power
  - 4.16ms average seek time
  - 4096 byte physical sectors
  - 7200 RPMs
  - Dual 6 Gbps SATA /12Gbps SAS interface
    - 270MB/s MAX transfer rate
    - Cache size: 256MB
  - Price: $ 562 (~ $0.03/GB)

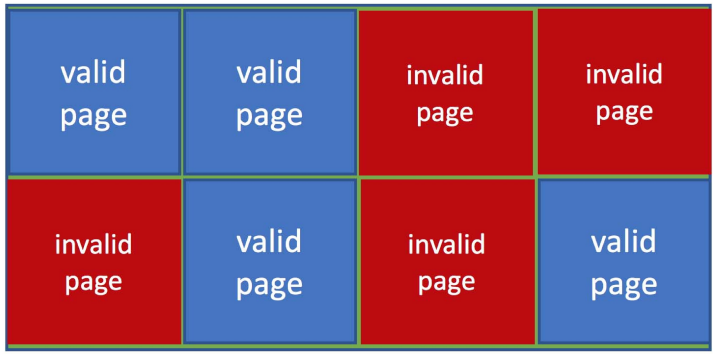
- IBM Personal Computer/AT (1986)
  - 30 MB hard disk
  - 30-40ms seek time
  - 0.7-1 MB/s (est.)
  - Price: $500 ($17K/GB, 340,000x more expensive !!)

Nonvolatile memory (NVM) devices

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

Nonvolatile Memory Devices

- Based on NAND semiconductors technology, have some characteristics that present their own storage and reliability challenges
- Read and written in “page” increments (think sector) but can’t overwrite in place
  - Must first be erased, and erases can only be executed in ”block” increments
  - Can only be erased a limited number of times before worn out – ~ 100,000
  - Life span measured in **drive writes per day** (**DWPD**)
    - A 1TB NAND drive with rating of 5DWPD is expected to have 5TB per day written within warrantee period without failing
- With no overwrite, pages end up with mix of valid and invalid data



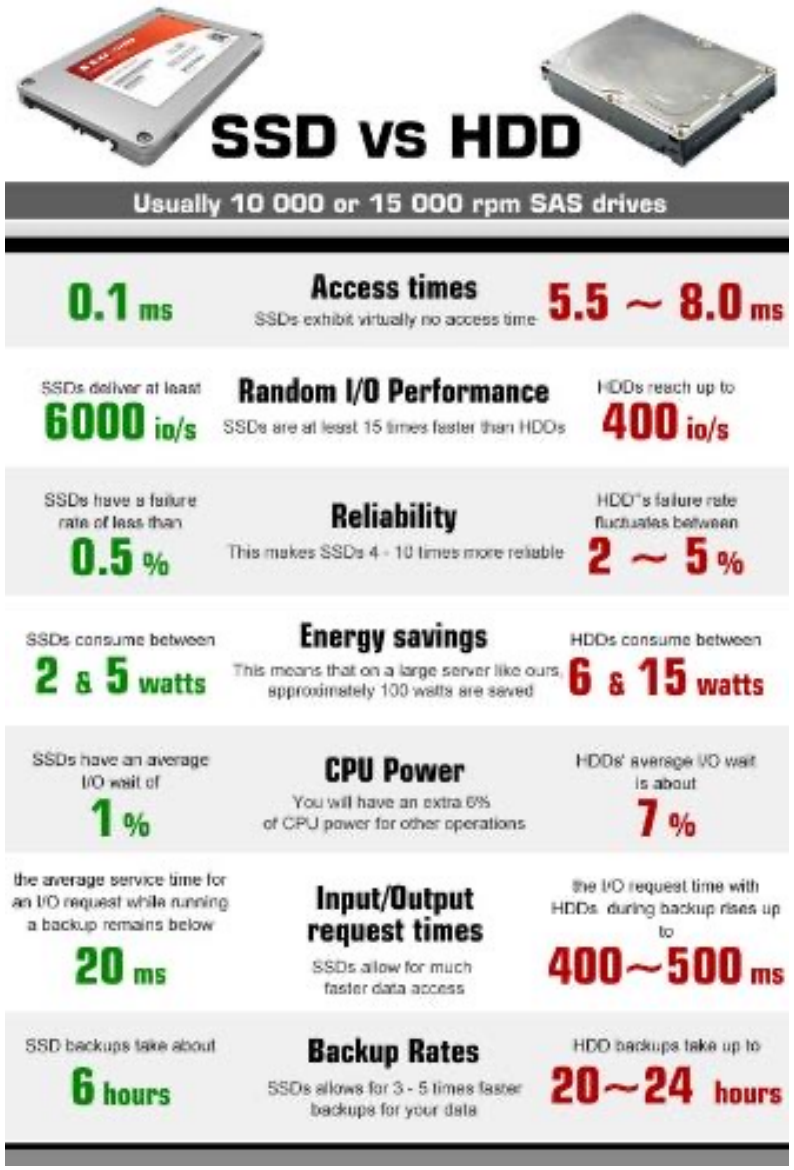
NAND block with valid and invalid pages

Some “Current” (large) 3.5in SSDs

- Seagate Exos SSD: 15.36TB (2017)
  - Seq reads 860MB/s
  - Seq writes 920MB/s
  - Price (Amazon): $5495 ($0.36/GB)

- Nimbus SSD: 100TB (2019)
  - Seq reads/writes: 500MB/s
  - Random Read Ops (IOPS): 100K
  - *Unlimited writes for 5 years!*
  - Price: ~ $40K? ($0.4/GB)
    - However, 50TB drive costs $12500 ($0.25/GB)

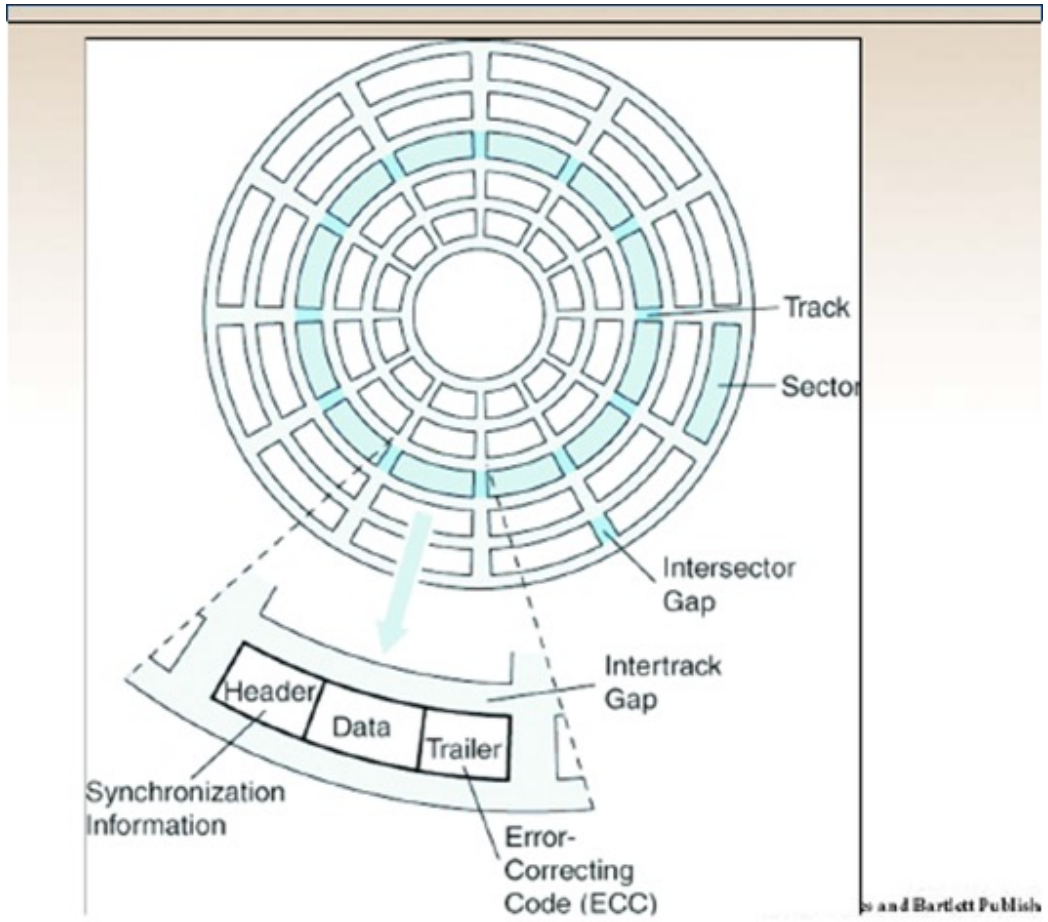
HDD vs. SSD Comparison



HDD	SDD
Require seek + rotation	No seeks
Not parallel (one head)	Parallel
Brittle (moving parts)	No moving parts
Random reads take 10s milliseconds	Random reads take 10s microseconds
Slow (Mechanical)	Wears out
Cheap/large storage	Expensive/smaller storage

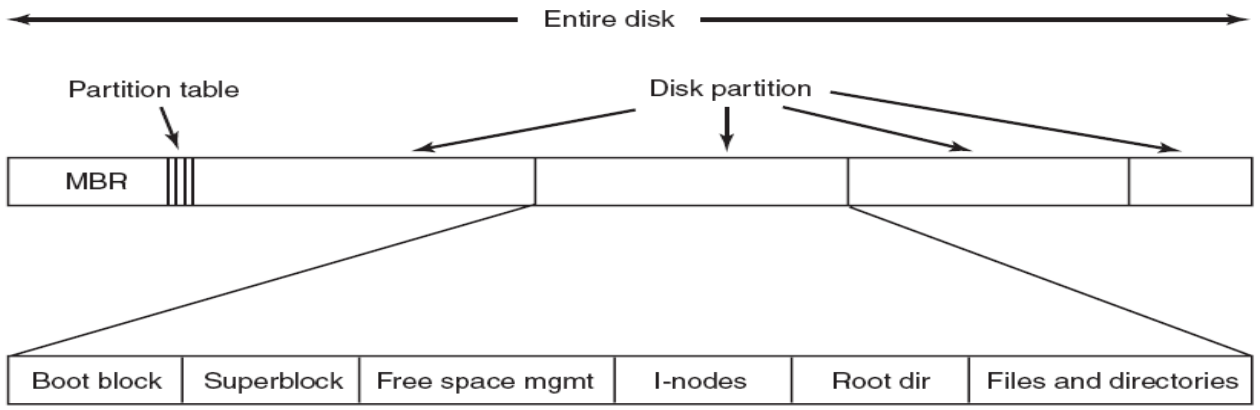
Disk formatting

- A new storage device is a blank slate: it is just a platter of a magnetic recording material (HDD) or a set of uninitialized semiconductor storage cells (SSD)
- Before a storage device can store data, it must be divided into sectors that the controller can read and write, this is called low-level formatting
- **Low-level formatting, or physical formatting** — Dividing a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (**ECC**)
  - Usually 512 bytes or 4KB
- Most drives are low-level-formatted at the factory as a part of the manufacturing process.



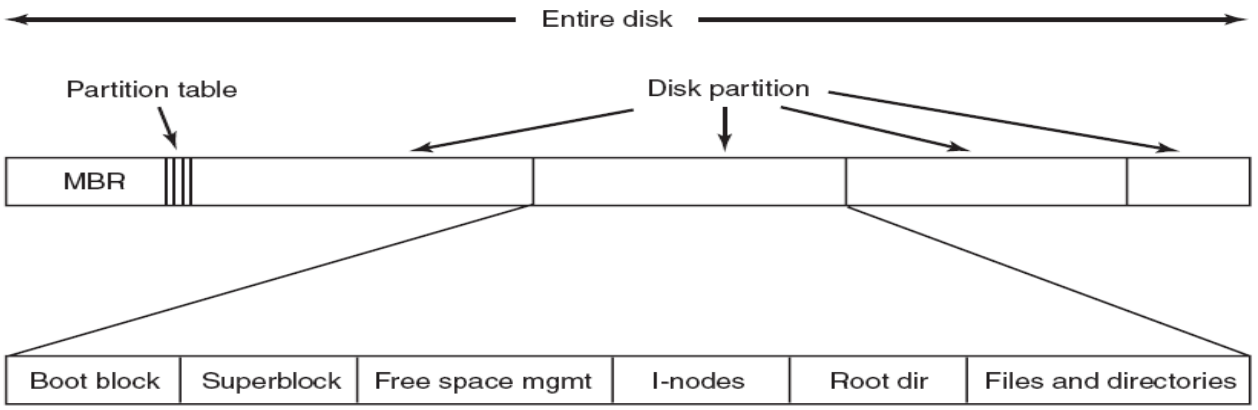
Disk formatting

- The next step is disk partition.
- **Partition:** the disk is partitioned into one or more groups of cylinders, each treated as a logical disk
- For instance, one partition can hold a file system containing a copy of the operating system's executable code, another the swap space, and another a file system containing the user files
- The partition information is written in a fixed format at a fixed location on the storage device



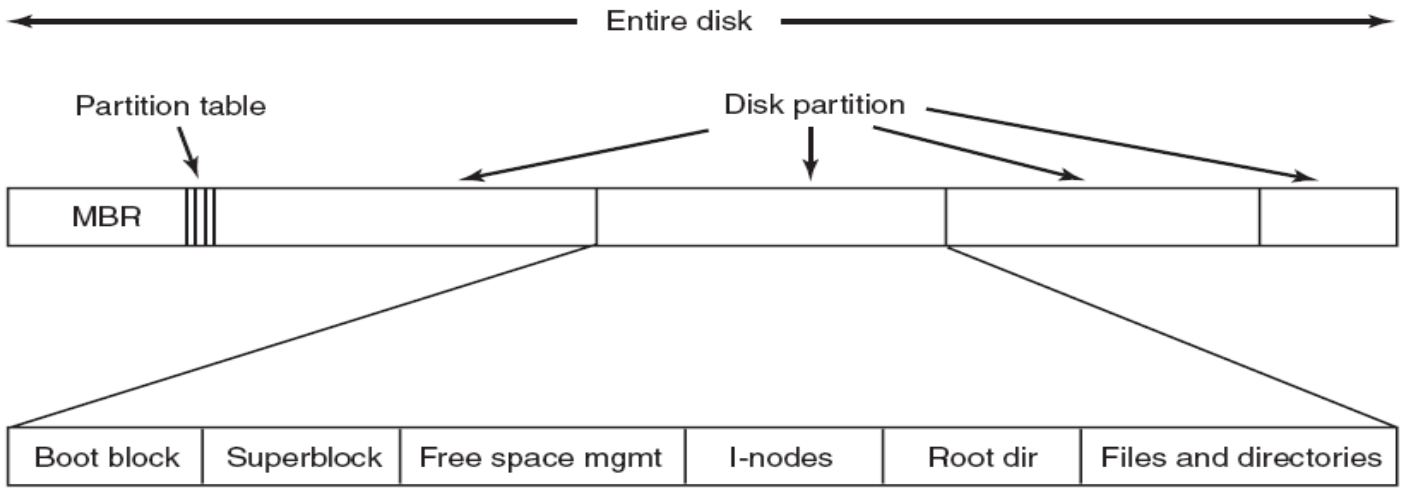
Disk formatting

- The last step is logical formatting or creation of a file system
- In this step, the operating system stores the initial file-system data structures:
  - **Superblock**. Contains a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information
  - free blocks in the file system in the form of a bitmap or a list of pointers
  - i-nodes
  - Root directory of the file system



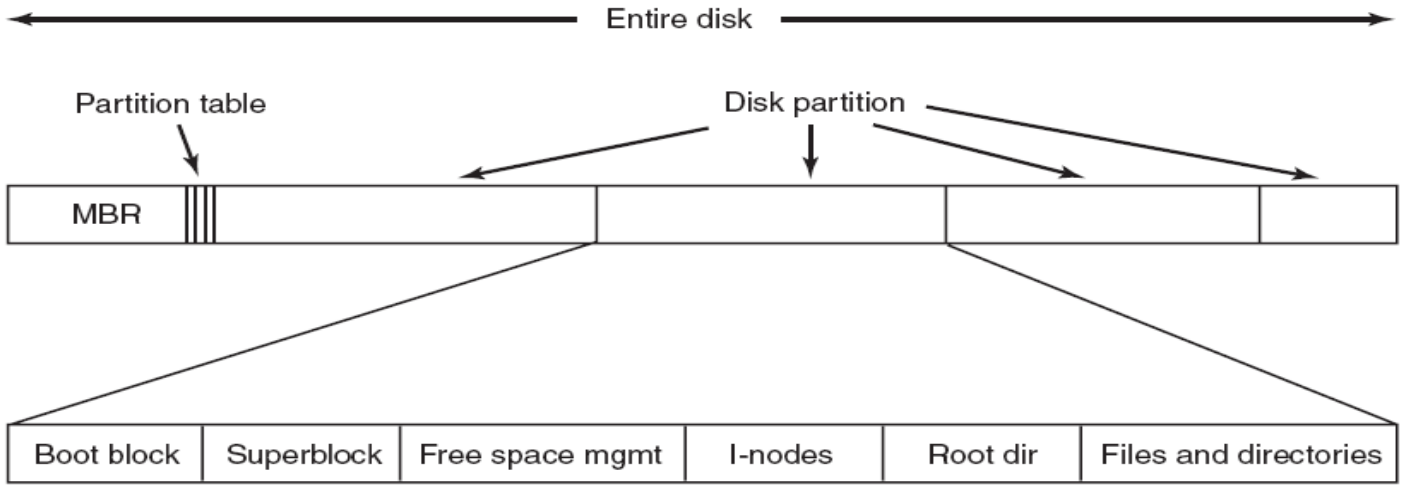
Boot time

- Each partitions can have a different file system.
- Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer.
- The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition.



Boot time

- One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR.
- The first thing the MBR program does is locate the active partition, read in its first block, which is called the **boot block**, and execute it.
- The program in the boot block loads the operating system contained in that partition
- Every partition starts with a boot block, even if it does not contain a bootable operating system.



Files

- **Files** are logical units of information created by processes.
- When a process creates a file, it gives the file a **name**.
- When the process terminates, the file continues to exist and can be accessed by other processes using its name.
- So, files provide a way to store information on the disk and read it back later.

File naming

- The exact rules for file naming vary from system to system, but all current operating systems allow strings of one to eight letters as legal file names.

- *andrea*, *bruce*, and *cathy* are possible file names.
- Digits and special characters are also permitted, like *2*, *urgent!*, and *Fig.2-14*
- Many file systems support names as long as 255 characters.
- Some file systems distinguish between upper- and lowercase letters (UNIX), whereas others do not (MS-DOS)

File naming

- Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*.
- The part following the period is called the **file extension** and usually indicates something about the file.
- In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters.
- In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions: *homepage.html.zip*
- UNIX file extensions are just conventions and are not enforced by the operating system
- Windows is aware of the extensions and assigns meaning to them. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter.
    - For example, double clicking on *file.docx* starts Microsoft *Word* with *file.docx* as the initial file to edit.

File types, extensions

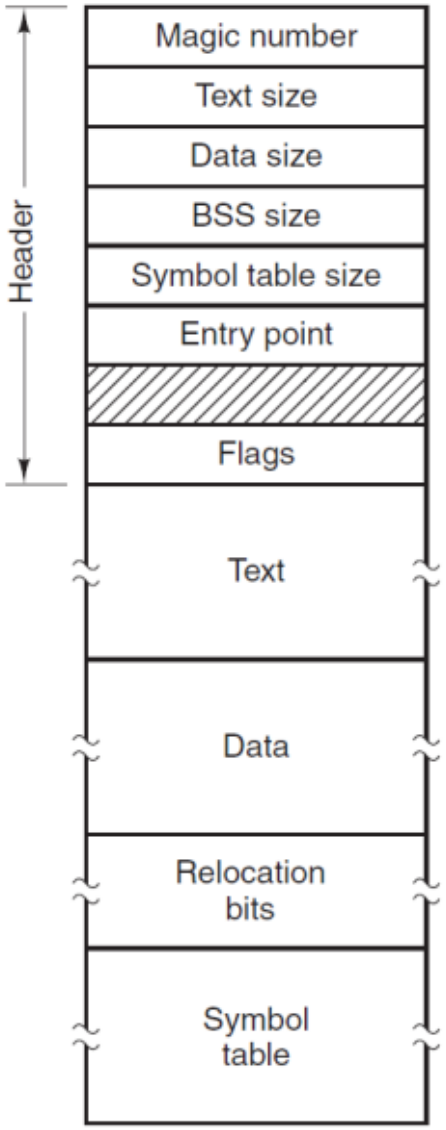
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Structure

- None - sequence of bytes
- Simple record structure
  - Lines (directories)
  - Fixed length
  - Variable length
- Complex Structures
  - Formatted document (Word, Latex, binaries)
  - Relocatable load file (libraries)
- Who decides:
  - Operating system
  - Program

File types

- Regular- contains user information (ASCII, binary)
- Directories, are system files for maintaining the structure of the file system



File access

- **Sequential Access**

```
read next
write next
reset
```

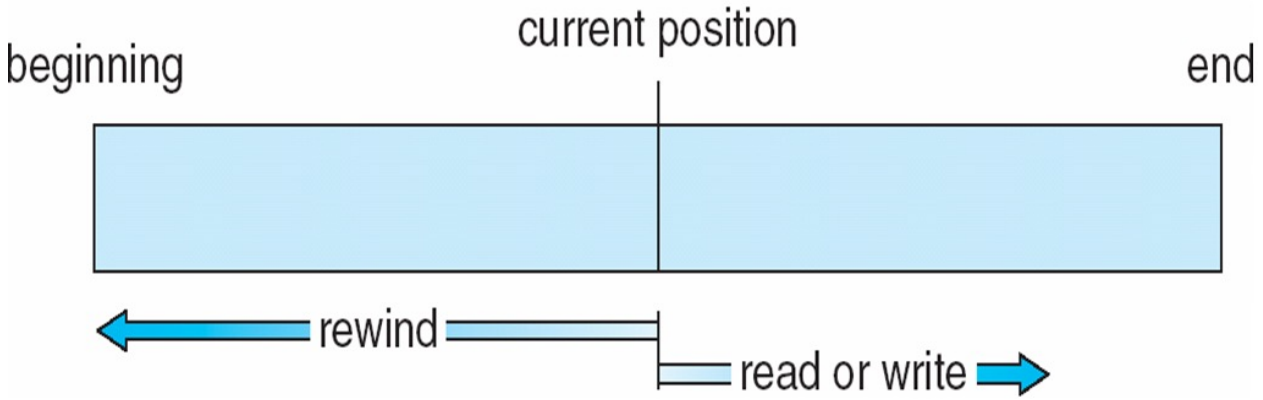
no read after last write (rewrite)

- **Direct Access –** file is fixed length logical records

```
read n
write n
position to n
      read next
      write next
rewrite n
```

n = relative block number

Sequential access file



File attributes

- Every file has a name and its data.
- In addition, all operating systems associate other information with each file, the date and time the file was last modified and the file's size.
- These extra items the file's **attributes** or **metadata**.
- The list of attributes varies considerably from system to system

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

System calls for files

- Create. The file is created with no data. It set some of the attributes.
- Delete. When the file is no longer needed, deleted to free up disk space.
- Open. Fetch attributes and disk addresses into main memory for rapid access
- Close. Free up internal table space used by attributes and addresses
- Read.
- Write.
- Append. Add data only to the end of the file..
- Seek. Repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

Open Files

- Several pieces of data are needed to manage open files:
  - Open-file table: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

Copyfile abc xyz

```
/* 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 */
```

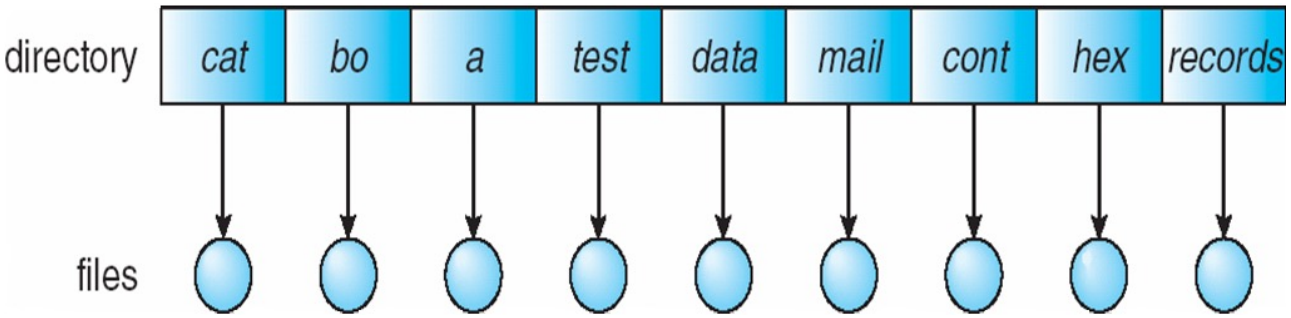
Copyfile abc xyz

```
        /* 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 */
}
```

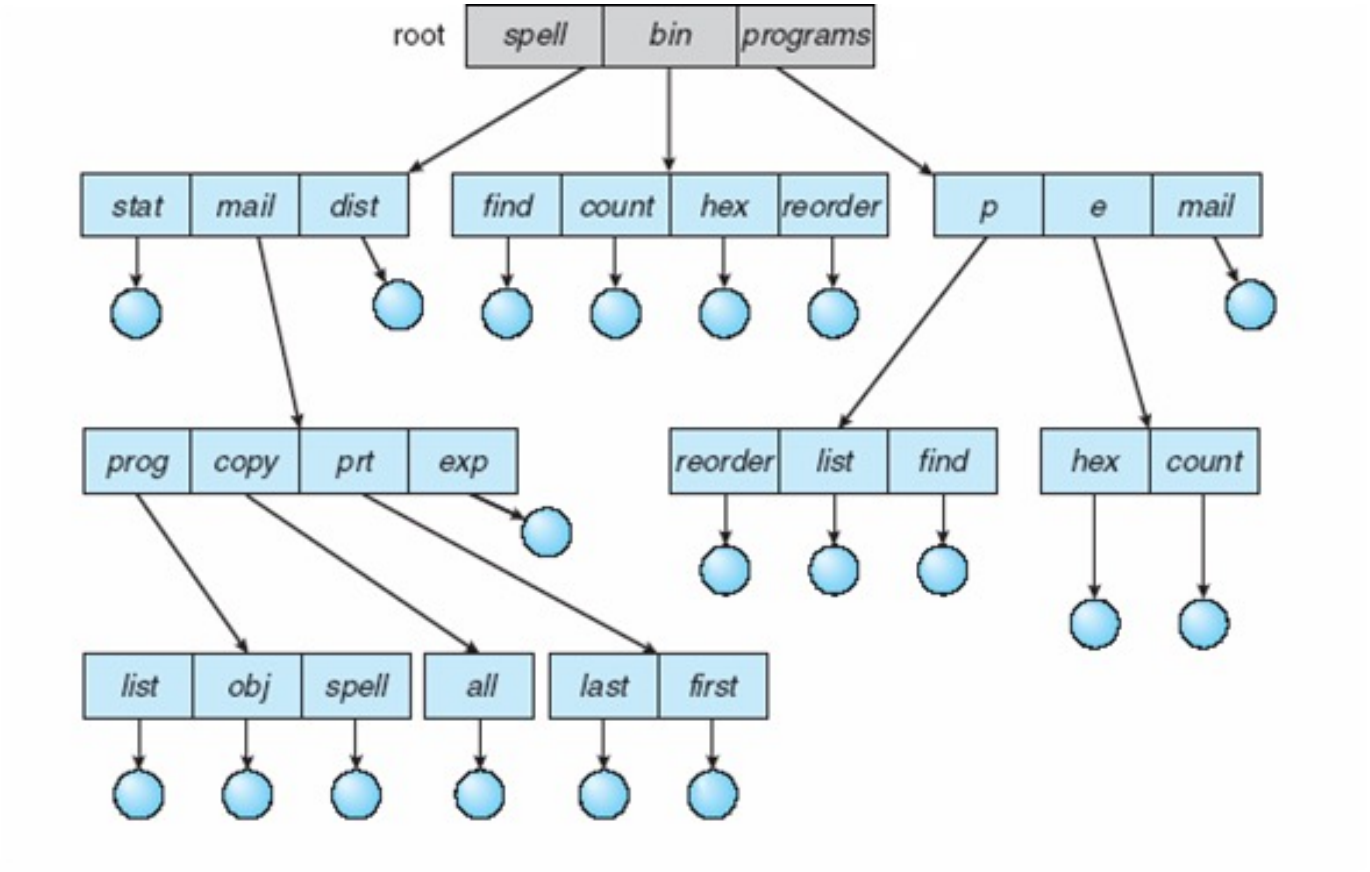
Directories

- To keep track of files, file systems normally have **directories**, which are themselves files.
- A directory is a file with a special structure
- Two types of directories: **Single-Level Directory and Hierarchical directory**
- **Single-Level Directory Systems:** one directory containing all the files
  - Was common in early personal computers



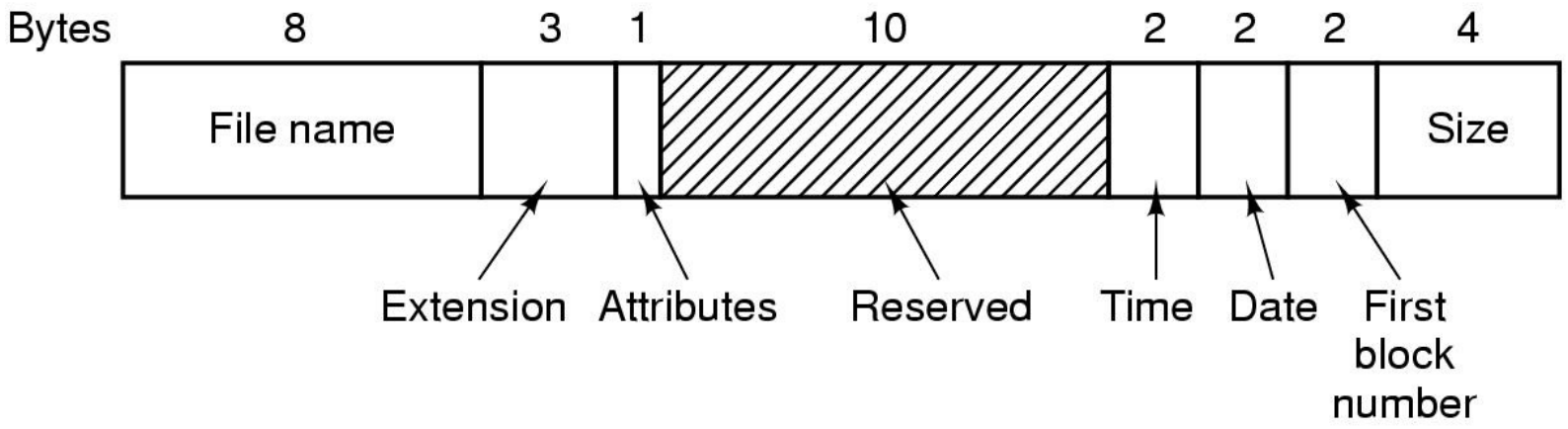
Hierarchical directory systems

- It is a tree where leaves are data-files and internal nodes are directory-files
- Each directory-file contains a mix of data-files and subdirectories
- The tree has, as a root node which is a directory, the root directory



MS-DOS directory entry

- MS-DOS directories have variable length, but each entry is 32 bytes long
- The *Attributes* field contains bits to indicate that a file is read-only, needs to be archived, is hidden, or is a system file.



- For Linux, a directory entry is a string of characters (file name) and a number (i-node), ls -i

Path names

- When the file system is based on a directory tree, a file is identified using its name and a path in the tree.
- Two different methods:
  - **Absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox*
  - **Relative path name**. In conjunction with **working directory** (also called the **current directory**). Path names beginning with the working directory

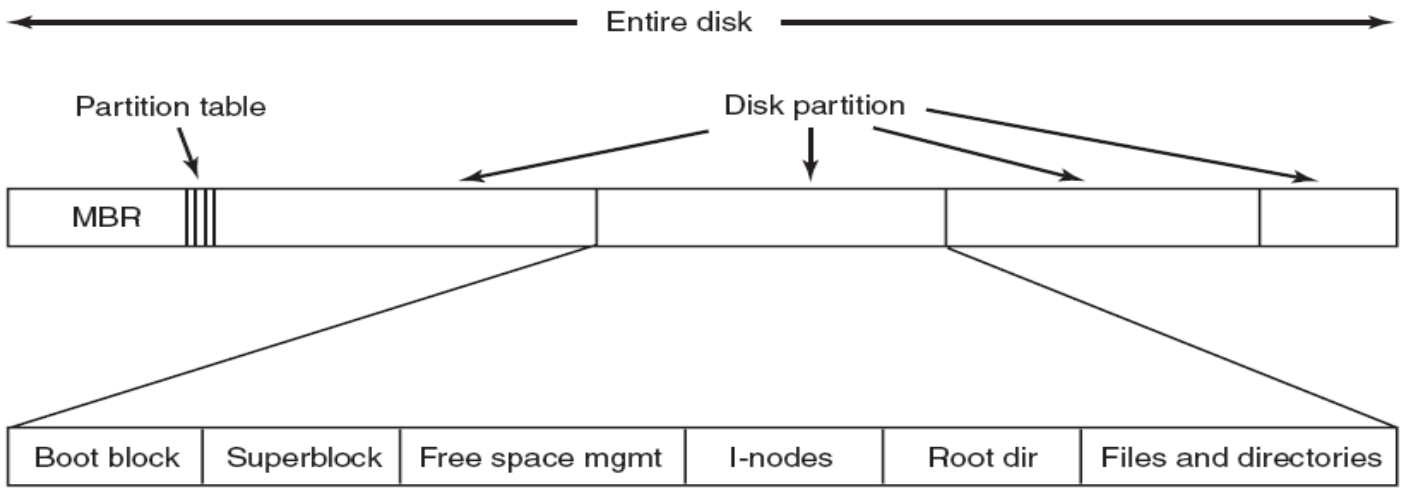
Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox

System calls for directories

- Create. Empty except for dot and dotdot (mkdir in Linux)
- Delete. Only an empty directory can be deleted (same as deleting a file, in Linux “rm –option directory”)
- Opendir. Directories can be read. For example, to list all the files
- Closedir. When a directory has been read, it should be closed to free up internal table space.
- Readdir. This call returns the next entry in an open directory.
- Rename. Directories are just like files and can be renamed the same way files can be.

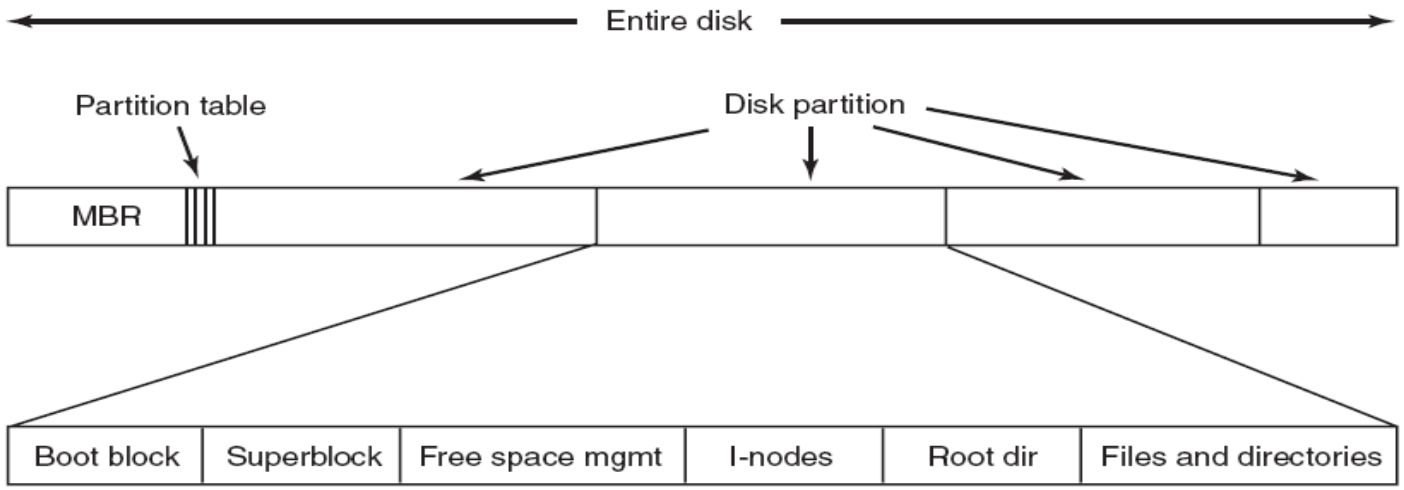
File system implementation

- The file system is stored on a disk
- **Superblock**: contains all the key parameters about the file system, it is read into memory when the computer is booted or the file system is first touched.
- Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other administrative information.



File system implementation

- Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers.
- This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file.
- After that might come the root directory, which contains the top of the file-system tree.
- Finally, the remainder of the disk contains all the other directories and files.

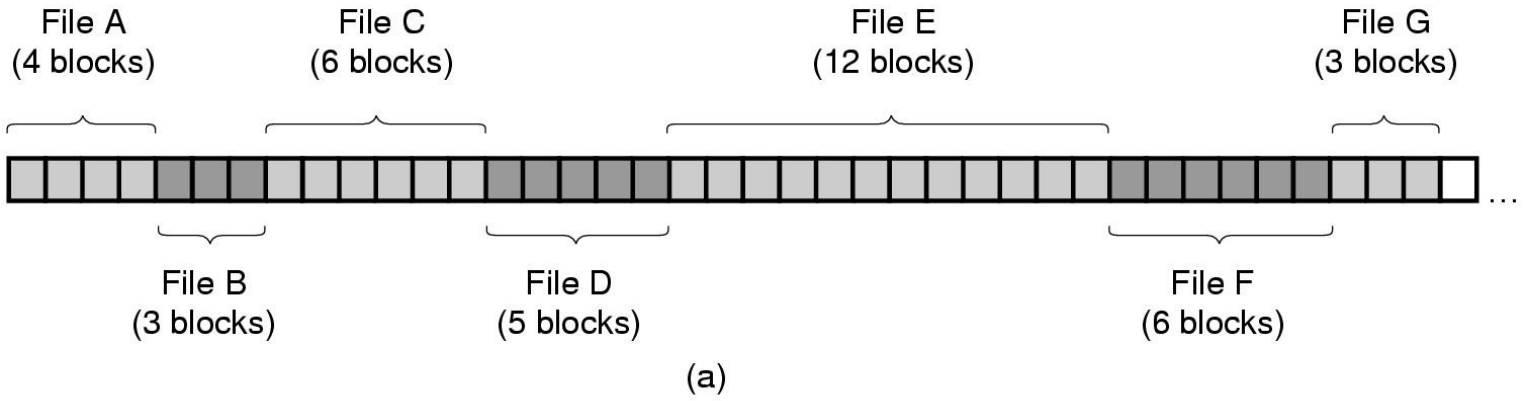


Storing files

- Files are stored in blocks (sectors) of the disk, so there must be a policy to allocate blocks to files (similar to allocate main memory to processes)
- Allocation methods:
  - Contiguous blocks
  - Linked list of blocks
  - Linked list using table
  - I-nodes

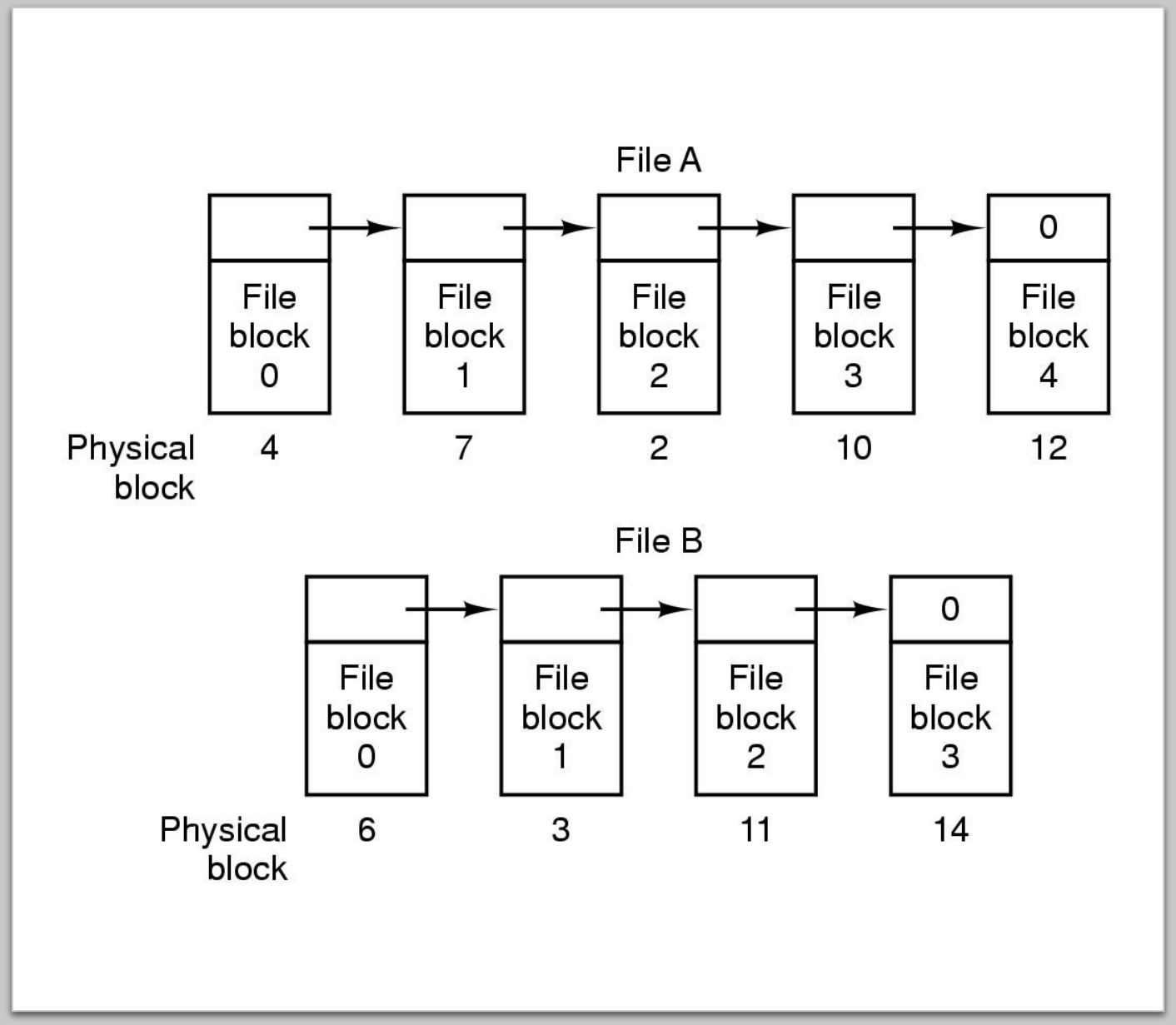
Contiguous allocation

- The simplest allocation scheme is to store each file as a contiguous run of disk blocks.
- Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks
- The directory entry only need the address of the first block
- Drawback: over time, the disk becomes fragmented.



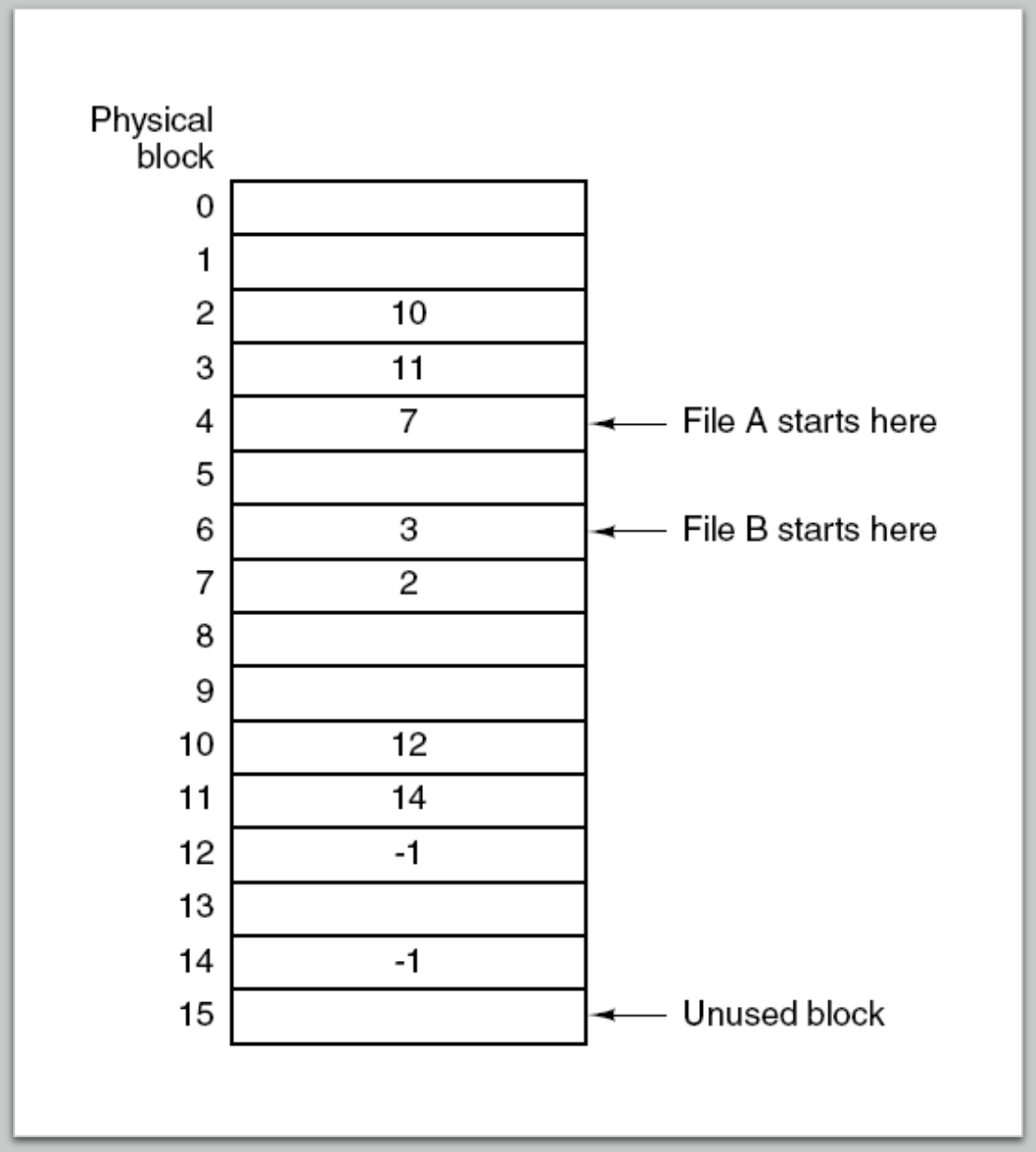
Linked list allocation

- Here each is stored 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.
- The directory entry only need the address of the first block
- Random access is slow, need to chase all the pointers



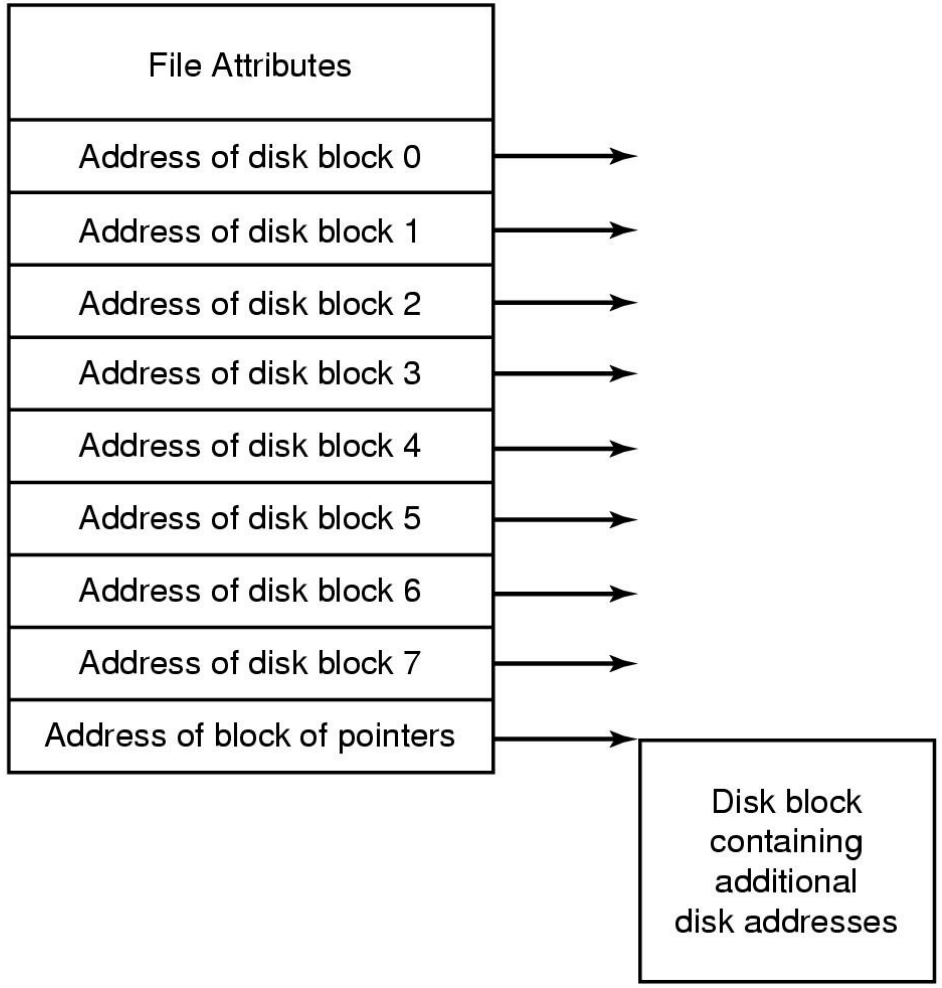
Linked list allocation using table

- Have a table in main memory of the same size as the number of blocks on the disk
- Store the address of the next block in the tale
- Inconvenient: Table too big, with a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks
- This “file allocation table” (FAT) come in different forms in MS_DOS and Windows, FAT-12, FAT-16, FAT-32



I-nodes

- Lists the attributes and disk addresses of the file's blocks
- There is one i-node per file, i-nodes are stored on a specific set of blocks on the disk, when a file is opened, they are loaded in main memory
- i-nodes contain a fix number of disk addresses, if file is big, one address is kept to store the address of a block that contain more disk address
- i-nodes is the method in Unix/Linux operating systems, Windows has a similar system called NTFS

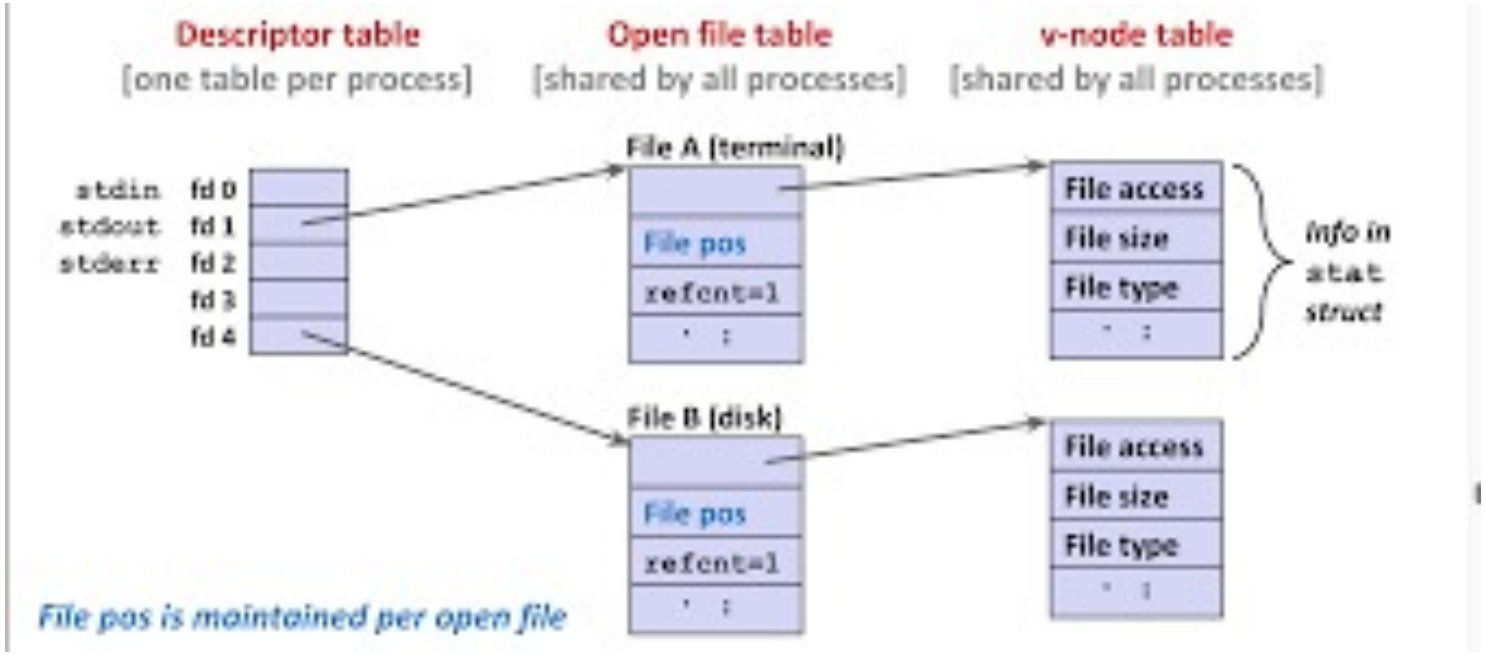


Implementing directories

- Before a file can be read, it must be opened.
- When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk.
- Then the directory is loaded in main memory where it can be read.
- The directory entry provides the information needed to find the disk blocks of the file.
- Depending on the system this information may be:
  - the disk address of the entire file (with contiguous allocation),
  - the number of the first block (both linked- list schemes),
  - or the number of an i-node.
- In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

File descriptor (fd)

- File descriptors index into a per-process **file descriptor table**, which indexes into a system-wide table of files opened by all processes, called the **file table**.
- The file table records the *mode* with which a file has been opened: reading, writing, appending and indexes into a third table called inode table (v-node)
- When a file is opened, the path name must be used to the file on disk. After that the read or write operations only need the fd to find the file.
- Linux file descriptors of a process can be accessed in `/proc/PID/fd/`



Opening a file

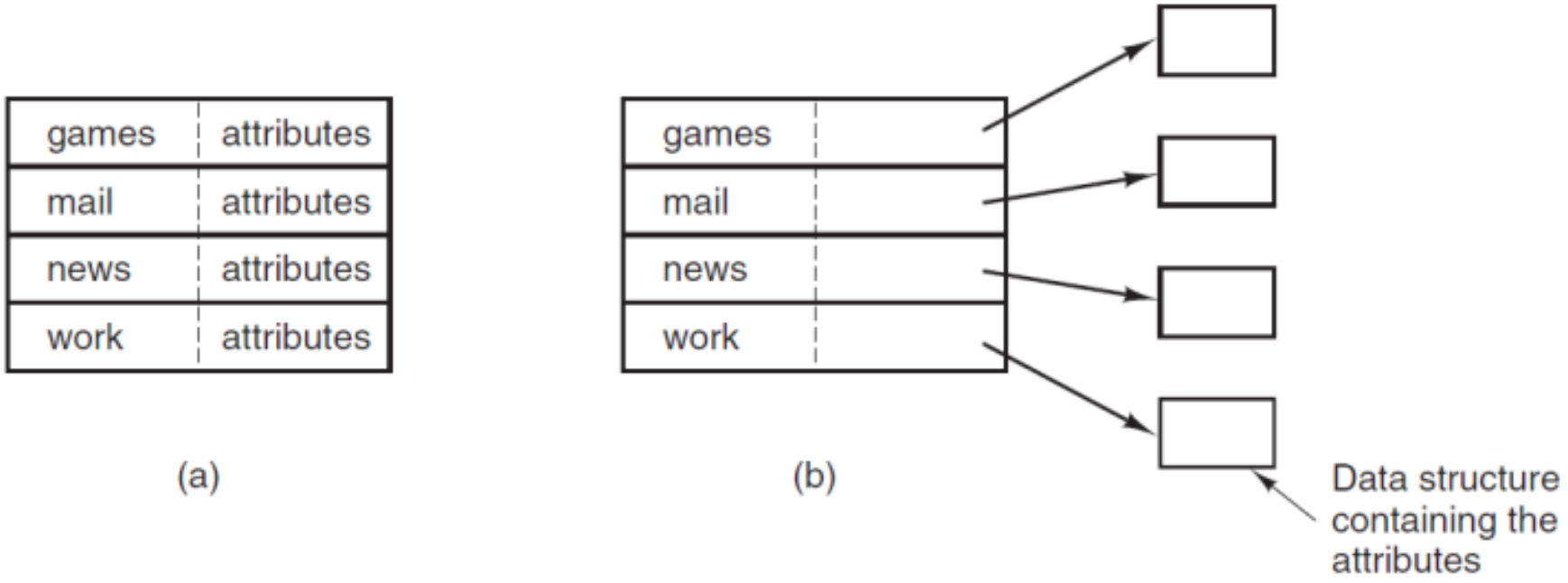
- The open() system call passes a file name to the file system.
- The open() system call first searches the open-file table to see if the file is already in use by another process.
- If it is, an entry is created into the process file descriptor table pointing to the existing open-file table.
- If the file is not already open, the directory structure is searched for the given file name.
- Once the file is found, its i-node is copied in main memory and an entry in the open-file table is created pointing to the new i-node
- Next, an entry is made in the process file descriptor, with a pointer to the newly created entry in the open-file table
- The open() call returns a pointer to the appropriate entry in the file descriptor of the process.
- All other file operations (read, write, ect.) are then performed via this pointer.

Opening a file: example

- Open(/usr/me/mailbox), not already open
  - 1-find entry of “usr” in / and get the i-node of usr on disk
  - 2-load i-node in main memory
  - 3-find blocks of the directory file usr on disk
  - 4-load the directory usr in main memory
  - 5-read usr until find entry of directory “me” and read the number of the corresponding i-node
  - 6-find i-node of “me” on disk and load it in main memory
  - 7-read the directory “me” until string “mailbox” is found
  - 8-read the i-node number of file “mailbox”
  - 9-find the i-node on disk and load it in main memory
  - 10-create an entry in the open file table and makes it point to the i-node for the mailbox file
  - 11-create an entry in the file descriptor table and makes it point to the corresponding entry in the open file table
  - 12 returns the entry number to the calling process
- The system usually don’t need the absolute path, rather the relative path from the current working directory is enough

Implementing directories

- Another issue is where the attributes should be stored.
- Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere.
- One obvious possibility is to store them directly in the directory entry.
- For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries

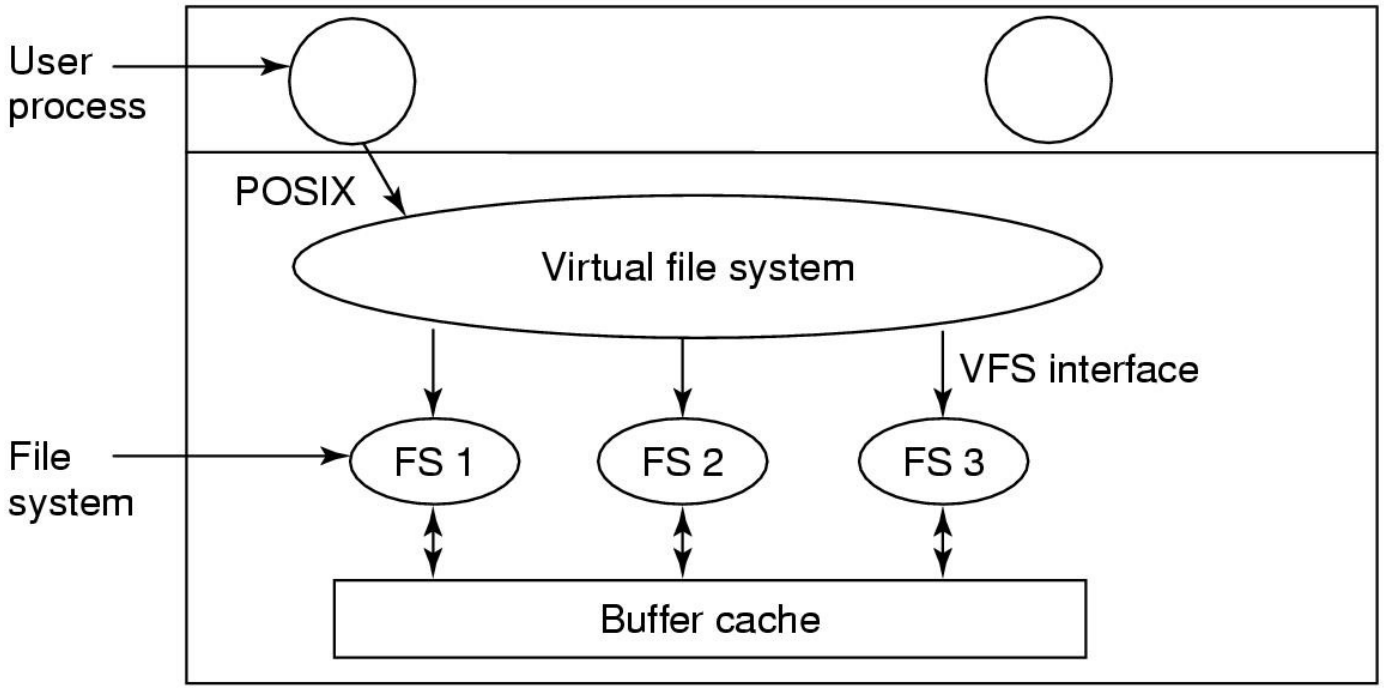


Multiple file systems

- A computer or an operating system could use different file systems
- Windows may have a main NTFS file system, but also a legacy FAT -32 or FAT -16 drive or partition that contains old, but still needed, data, and from time to time a flash drive, an old CD-ROM or a DVD
- Windows handles these disparate file systems by identifying each one with a different drive letter, as in *C:*, *D:*, etc.
- When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to.
- Windows does not have a virtual file system

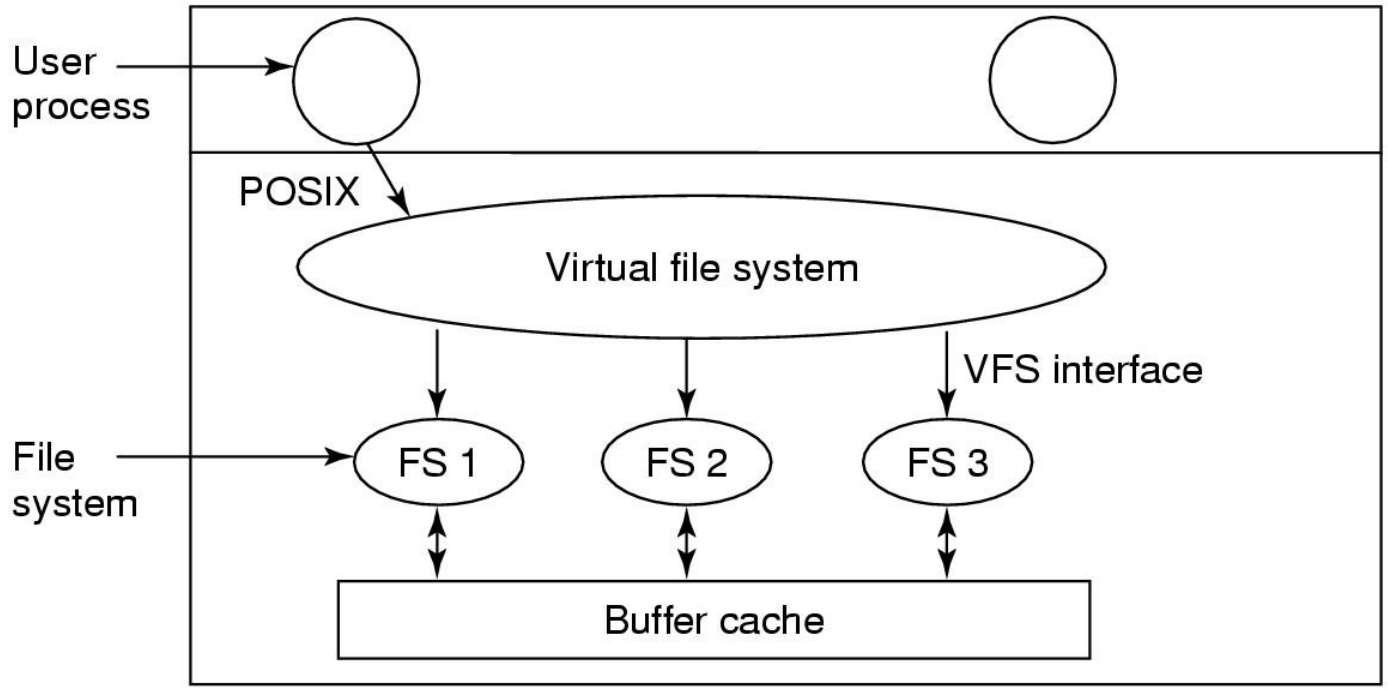
Virtual file systems

- UNIX systems integrate multiple file systems into a single VFS structure.
- For example, a Linux system could have ext2 as the root file system, with an ext3 partition mounted on */usr* and a second hard disk with a ReiserFS file system mounted on */home* as well as an ISO 9660 CD-ROM temporarily mounted on */mnt*.
- POSIX system calls such as open, read, write, lseek, are directed to the virtual file system for initial processing



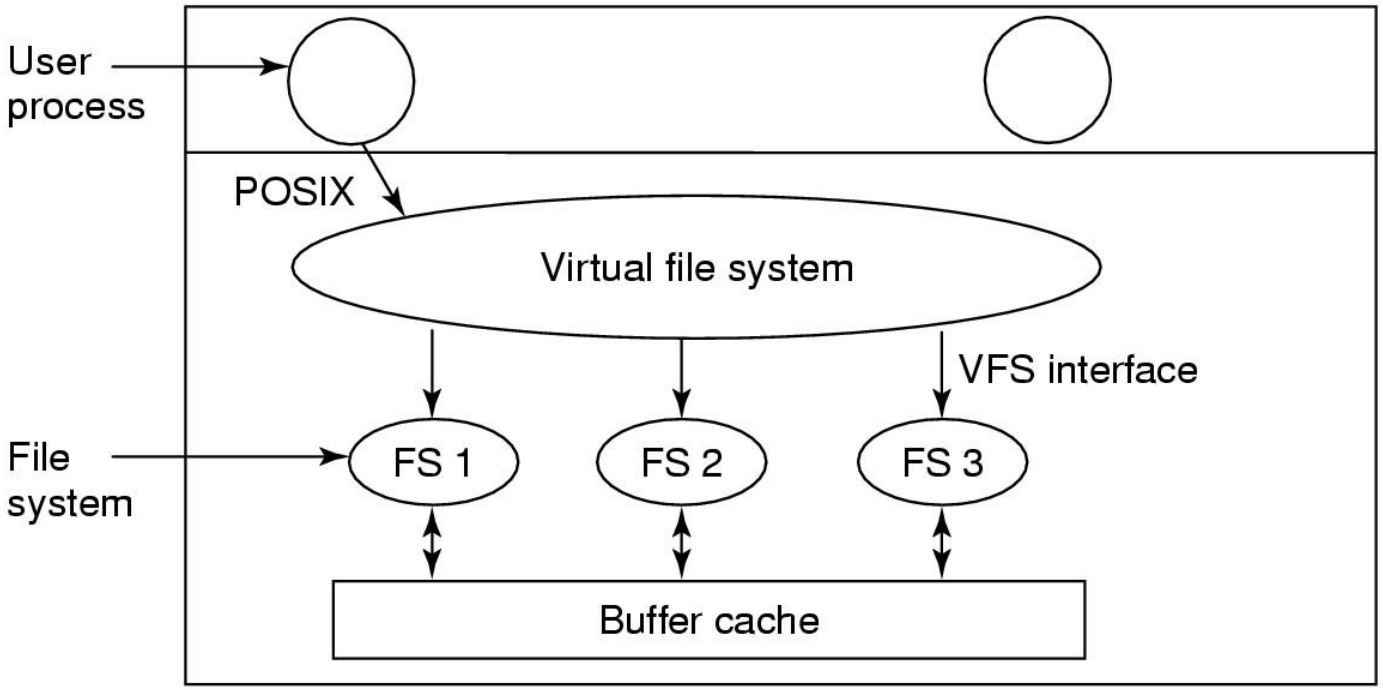
Virtual file systems

- The VFS also has also a “lower” interface to the concrete file systems
- This interface consists of several dozen function calls that the VFS can make to each file system to get work done.
- To create a new file system that works with the VFS, the designers of the new file system must make sure that it supplies the function calls the VFS requires.



Virtual file systems

- Most file systems under the VFS are partitions on the local disk.
- They can also be a remote file systems using the **NFS** (**Network File System**) protocol.



How VFS works

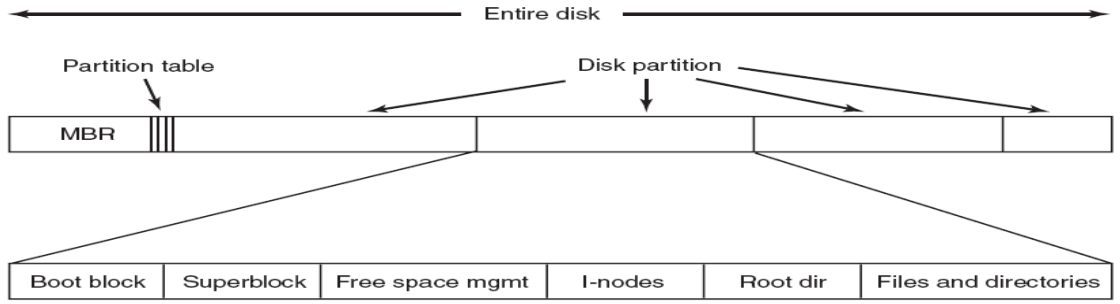
- When the system is booted, the root file system is registered with the VFS.
- When other file systems are mounted, either at boot time or during operation, they, too must register with the VFS.
- A file system registers by providing a list of the addresses of the functions the VFS requires, either as one long call vector (table) or as several of them, one per VFS object, as the VFS demands.

How VFS works

- After a file system has been mounted, it can be used. For example, if a file system has been mounted on */usr* and a process makes the call

  open("/usr/include/unistd.h", O RDONLY)

- Parsing the path, the VFS sees that a new file system has been mounted on */usr* and locates its superblock by searching the list of superblocks of mounted file systems.
- The VFS find the root directory of the mounted file system and look up the path *include/unistd.h* there.

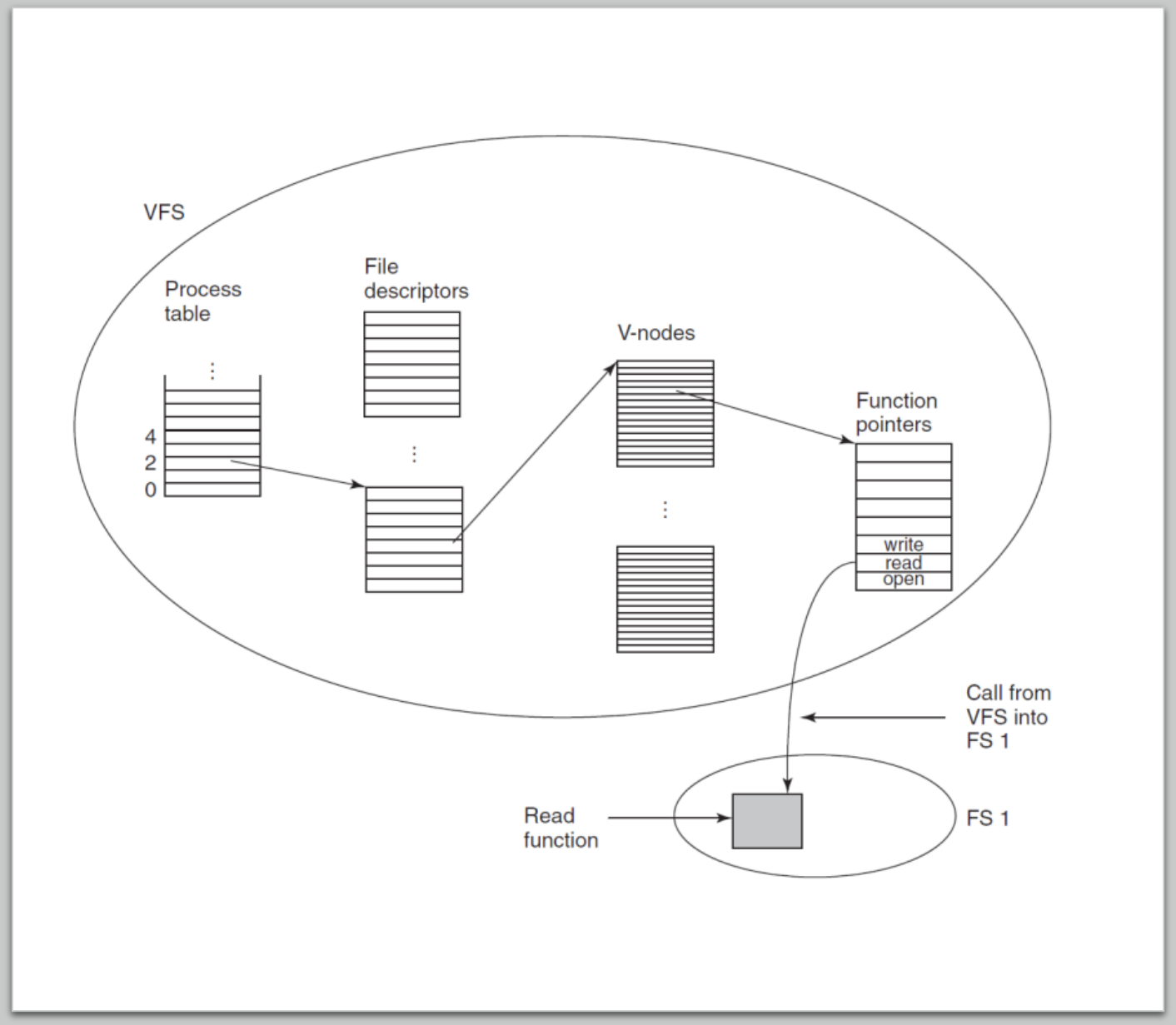


How VFS works

- The VFS then creates a v-node and makes a call to the concrete file system to return all the information in the file's inode.
- This information is copied into the v-node (in RAM), along with other information, most importantly the pointer to the table of functions to call for operations on v-nodes, such as read, write, close, and so on.
- After the v-node has been created, the VFS makes an entry in the file-descriptor table for the calling process and sets it to point to the new v-node

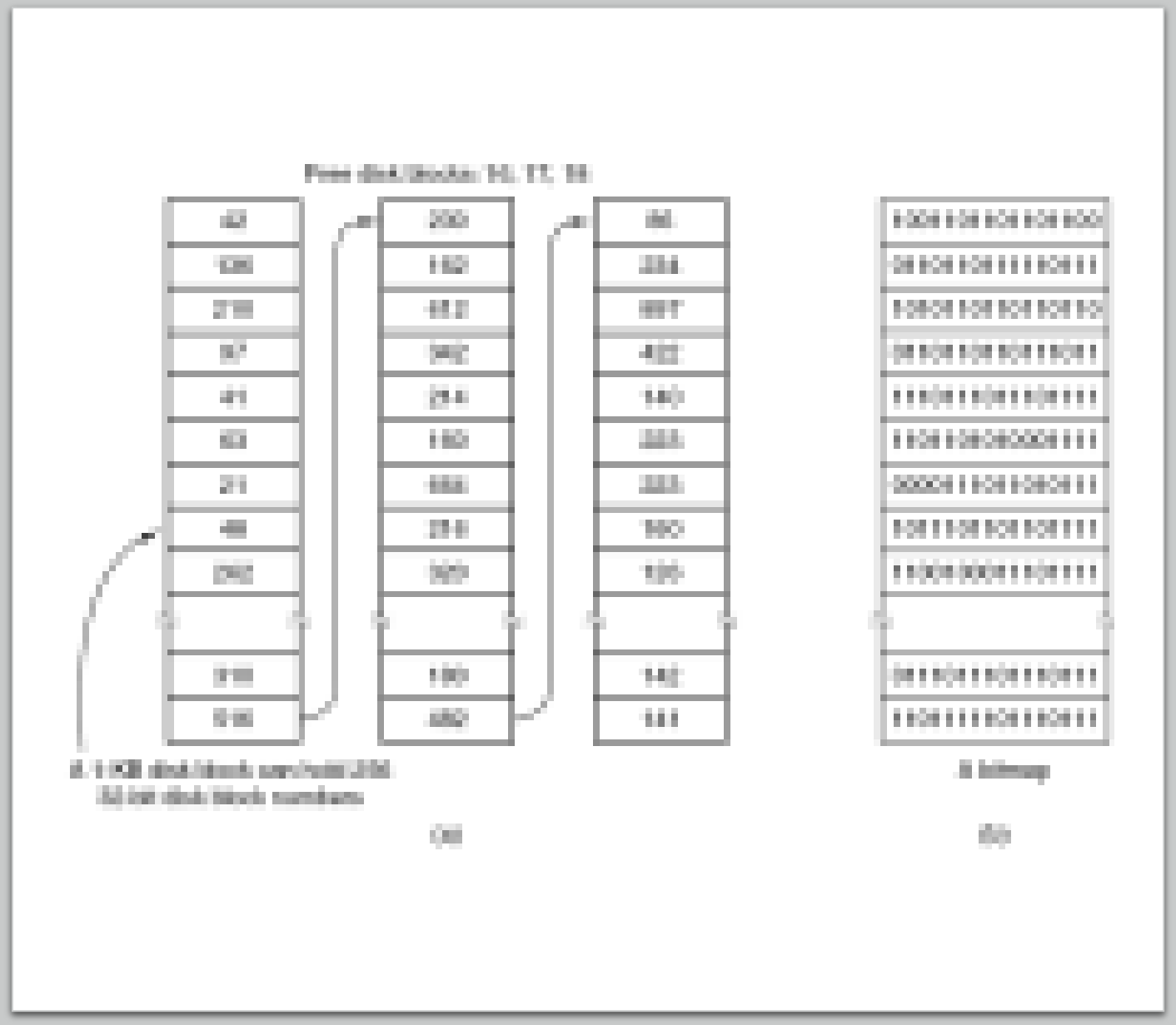
How VFS works

- Later when the process does a read using the file descriptor, the VFS locates the v-node from the process and file descriptor tables and follows the pointer to the table of functions, all of which are addresses within the concrete file system on which the requested file resides



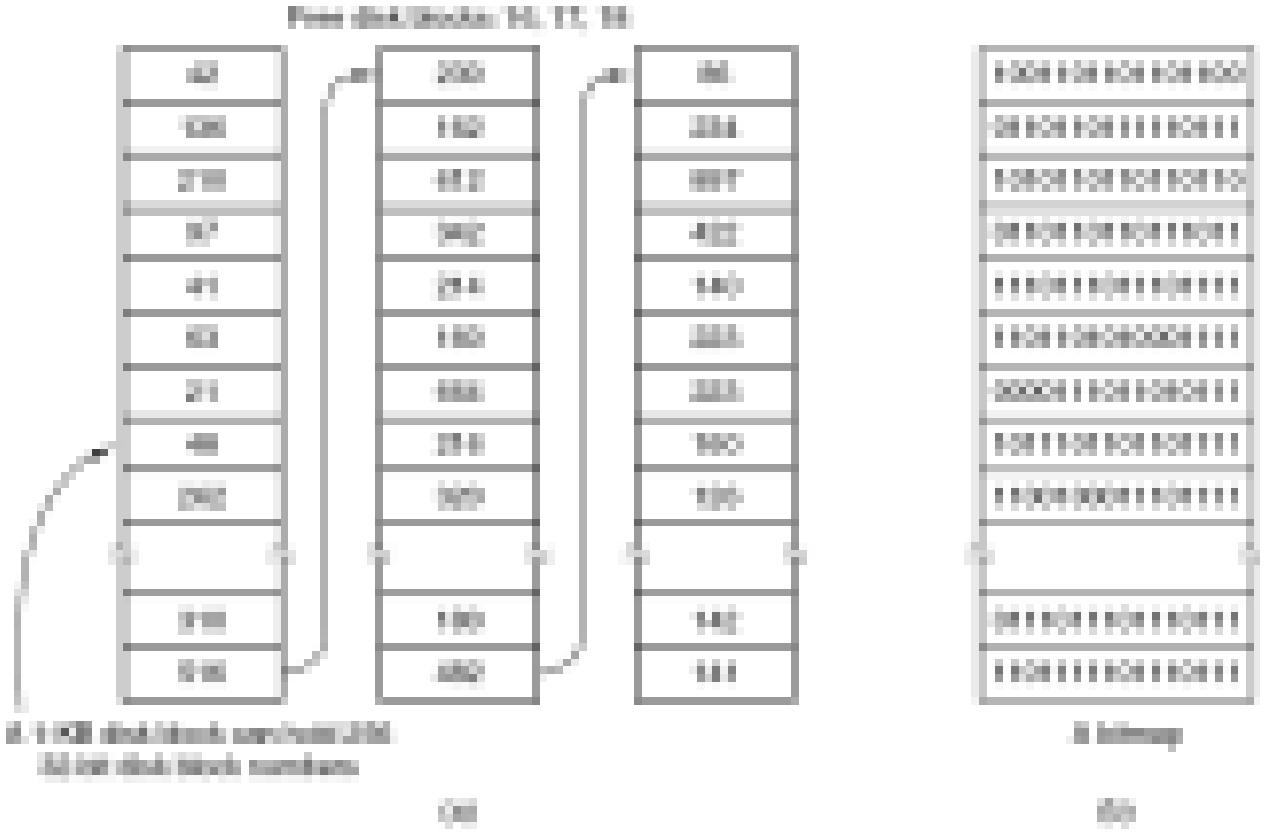
Keeping track of free blocks

- Two methods.
- One: use a linked list of disk blocks, with each block holding as many free disk block numbers as will fit:
  - With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.)
  - A 1-TB disk requires about 4 million blocks.



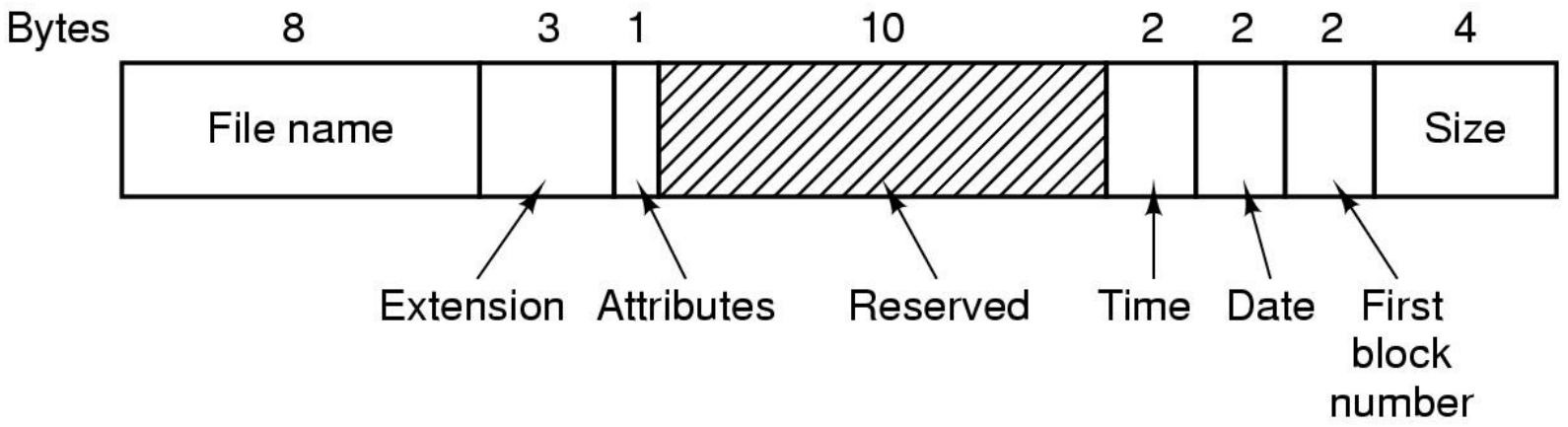
Keeping track of free blocks

- Two methods.
  - Two: bitmap. A disk with *n* blocks requires a bitmap with *n* bits. Free blocks are represented by 1s in the map, allocated blocks by 0s
  - 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store



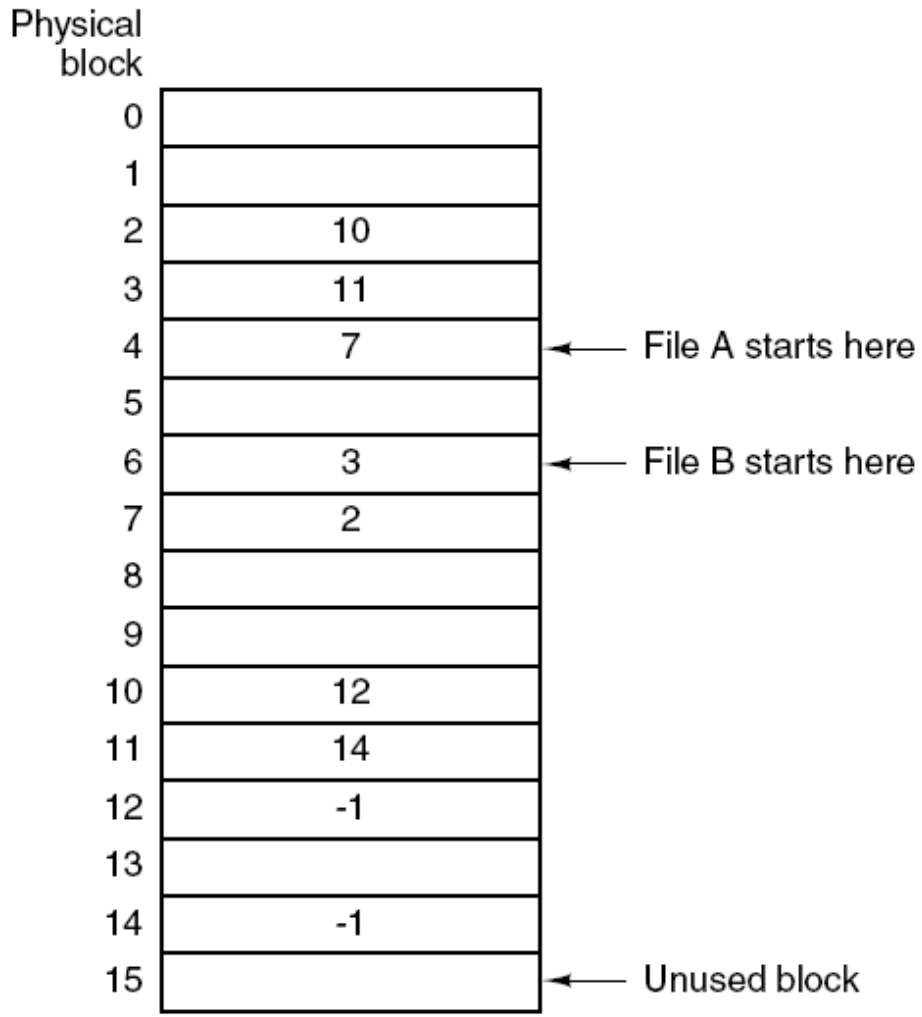
Examples of FS: MS-DOS

- To read a file, an MS-DOS program must first make an open system call to get a handle for it.
- The open system call specifies a path
- The path is looked up component by component until the final directory is located and read into memory. It is then searched for the file to be opened.
- MS-DOS directories have variable length, but each entry is 32 bytes long
- The *Attributes* field contains bits to indicate that a file is read-only, needs to be archived, is hidden, or is a system file.



Examples of FS: MS-DOS

- MS-DOS stores the file size as a 32-bit number, so in theory files can be as large as 4 GB. However, other limits restrict the maximum file size to 2 GB or less.
- MS-DOS keeps track of file blocks via a file allocation table in main memory.
- The directory entry contains the number of the first file block. This number is used as an index into a 64K entry FAT (file allocation table) in main memory. By following the chain, all the blocks can be found.
- The FAT file system comes in three versions: FAT -12, FAT -16, and FAT -32, depending on how many bits a disk address contains



Examples of FS: MS-DOS

- For all FATs, the disk block can be set to some multiple of 512 bytes
- With blocks of 512 bytes and block address size 12 bits, a partition cannot be larger than 2MB.
- The size of the FAT table in memory is 4096 entries of 2 bytes
- MS-DOS supported four disk partitions per disk drive, so when disks with larger capacity came, blocks were made bigger

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB