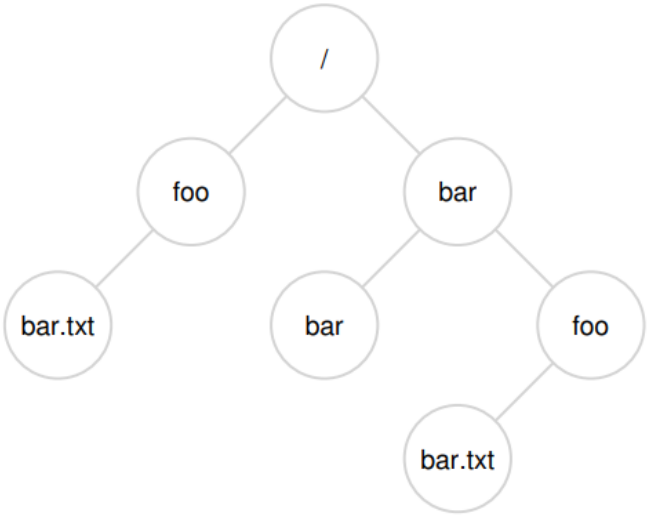
The Abstraction

The key abstraction to virtualize the persistent storage is

- **file** : is a linear array of bytes each of which can be read or written. A file has a name (users abstraction at higher level) and usually a number (i-node for the OS at lower level). The OS does not know about the structure of a file (a document or a C - program or a picture). The responsibility of the OS is to allow read and write (other than opening, closing etc. as we shall see later) and make the file available whenever needed.
- **Directory**: Is like a file with a name (as well as a lower level identifier – a number) and each entry of the directory is either a file or another directory – essentially creating a tree structure



The Abstraction ...contd

Considering a variety of devices available for secondary and archival storage we may abstract our requirements as read a character (or block) and write a character (or block) [like primary storage] in the storage mediums (devices) used at this level.
The abstraction is certainly an oversimplification of the need from user's perspective and the following are important

- Naming
- Structure
- Types
- Access
- Attributes
- Operations

Naming

The minimum requirement to recognize the existence of a file, in the user level, is cetainly a string (the name) associated with the file.

- We name (arbitrary string) a file when it is created
- There may be
  - variation distingushing small and capital letters (In Unix CST, Cst, cst are all different strings and in MS-DOS they are considered same)
  - restriction of length (8 charcaters, as in MS-DOS, only or practically no restriction; as in UNIX)
  - effort to have more meaning by allowing one or more extensions (.c is a C program text) i.e., the two part (or more) naming convension.

[Note: The use of extensions is extended in WINDOWS where the corresponding processing software is automatically called if you double click on the file icons – thus word s/w is automatically called if you select a file whose extension is .doc/.docx]

Structure

The contents of a file may be viewed as having A) no structure, B) a block structure; C) or a bit extended structure; such as a tree

- A) It is a series of bytes – OS does not know or care. User routine must impose any meaning. For example in a UNIX text file we may look for *newline* and such a text file is nothing but 'n' consecutive byte for the OS. [UNIX and WINDOWS follow this logic]
- B) It is a sequence of records (fixed length) - in old days when a file is created and stored in secondary memory by taking input from a card reader (each card could hold 80 characters) – the file was 'n' such records.
- C) It consists of a tree of records of variying length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key. [used in some commercial data processing]

Types

Practically all operating systesms offer different types of files as per the contents, or some internal structure, or management or control needs. Here is the UNIX file classifications

- Ordinary files : An ordinary file is a file on the system that contains data, text, program instructions, executables and availble either as ASCII text or Binary. Does not contain any other file and available under a directory
- Directory : Directories are files that contain other files and sub-directories. Directories are used to organize the data by keeping closely related files in the same place
- Special or Device Files : These files represent the physical devices. Files can also refer to computer hardware such as terminals and printers. UNix has two different sub-types for device files; i) Character (used to imitate keyboard/printer and ii) block (used to imitate disks)

Unix supports other special files like; Symbolic link; named pipe and socket

Access Types

In the old system tape was predominant used as secondary memory device and the access allowed is only sequential

- Bytes or records in a file in order, starting at the beginning, is read or written.
- Sequential files could be rewound, however, so they could be read as often as needed.
- Sequential files were convenient when the storage medium was magnetic tape rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position.

- Files whose bytes or records can be read in any order are called random-access files.
- Random access files are essential for many applications

Two methods can be used for specifying where to start r/w. i) Mention the start position or ii) Seek the current position and then r/w the file sequentially.

Attributes

For the user as well as OS to manage the files numerous attributes (also known as metadata) is associated with the files.

- These are protection, ownership, acess right,
- file type (Text, or Binary or some application related special type)
- time; e.g., time created, time last accesses
- size (current size and other things like no. of records, no of bytes in a record etc.)
- flags; RO, hidden, system, archive

List of attributes vary with OS but some of them are common.

Operations

Other than the obvious read and write OS supports many operations on the file for obvious reason.
The operations other than read and write are as follows

- create (touch command in UNIX creates an empty file) : Space must be allocated for this file in the file system and an entry is to be made in the directory
- open (The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls)
- close (when it is done a file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes.)
- append : An old file may be appened with new data at the end of the file.

Operations contd.

- rename : This operation can be done in two steps i) copy the file ii) delete the old file; however this is provided in every OS.
- get/set atributes
- Reposition : (For random access files we may define the starting position wherefrom we may read or write; done using seek())
- delete – This operation releases the space and the directory is updated. However, if a file being shared cannot be deleted outright until all the links are relinquished

The file system interface

Through the file system interface (system calls) we deal with the basic tasks like create a file, acess (r/w) a file and delete etc. operations.
Create : open() call may be used to create a file. Here a file named foo is created (if it is not already available) for writing only. If foo is already present then the contents will be fully truncated and it will be ready to take new input.

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
```

This call returns a file descriptor (fd) that is used for other file operations like close the file; like close(fd);
Reading and writing: operations may be traced by

```
UNIX> echo hello > greet
```

The string "hello" is wriiten (redirected to the file greet instead of the monitor) to the file "greet"

The file system interface ...

To find the system calls used (including read and write) we may use *strace* command and get the following output (shown after the command)

```
unix>strace cat greet
...
open("greet", O_RDONLY|O_LARGEFILE) = 3
read(3, "hello\n", 4096)            = 6
write(1, "hello\n", 6)              = 6
hello
read(3, "", 4096)                   = 0
close(3)                            = 0
...
unix>
```

Note that open() call returns the file descriptor and in this case it is 3 (Why?). We also see the fd for the write call is 1 (why?).

Non sequential read/write

The example we have seen is for sequential read and write; starting from the beginning to the end.
Sometimes, however, it is useful to be able to read or write to a specific offset within a file; for example, if you build an index over a text document, and use it to look up a specific word, you may end up reading from some random offsets within the document. To do so, we will use the lseek() system call. Here is the function prototype:

```
off_t lseek(int fildes, off_t offset, int whence);

The first argument is familiar (a file descriptor).
The second argument is the offset, which positions the file
 offset to a particular location. The third is called whence --
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

Non sequential read/write ... contd.

We see

- for each file a process opens, the OS tracks a "current" offset, which determines where the next read or write will begin reading from or writing to within the file.
- Thus, part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways.
  - i) when a read or write of N bytes takes place, N is added to the current offset; thus each read or write implicitly updates the offset.
  - iii) explicitly with lseek, which changes the offset as specified above.

Note that this call lseek() has nothing to do with the seek operation of a disk, which moves the disk arm. The call to lseek() simply changes the value of a variable within the kernel; when the I/O is performed, depending on where the disk head is, the disk may or may not perform an actual seek to fulfill the request.

Write Immediate

Most times when a program calls write()

- it is just telling the file system to write this data to persistent storage, at some point in the future.
- The file system, for performance reasons, will buffer such writes in memory for some time; at that later point in time, the write(s) will actually be issued to the storage device.

From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g. machine crash just before the write to disk is done) will data be lost.
In some applications (e.g., DBMS) writing must be completed when demanded – this is essential for recovery of data if required.
In UNIX the fsync(int fd) system call forces immediate write of all the dirty pages back to the disk.

Write Immediate ... contd.

Here is a simple example of how to use fsync().

- The code opens the file foo, writes a single chunk of data to it, and then calls fsync() to ensure the writes are forced immediately to disk.
- Once the fsync() returns, the application can safely move on, knowing that the data has been persisted (if fsync() is correctly implemented, that is).

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to fsync() the directory that contains the file foo. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory. Not surprisingly, this type of detail is often overlooked, leading to many application-level bugs

Rename and file metadata

When we rename file the OS gurantees completion otherwise there would be inconsistencies due to a crash, does it happen, while the renaming operations is being completed. We can see the renaming operation

```
mv oldname newname
```

by using strace. However, the rename() call works as an atomic call with respect to the system crash. uses rename(char *old, char *new)
We keep a lot of information abaout a file for different purpose. We call it a metadata (you may use the command stat to ghet the metadata). A portion of the metadata is shown below:

```
struct stat {
dev_t st_dev; /* ID of device containing file */
ino_t st_ino; /* inode number */
mode_t st_mode; /* protection */
nlink_t st_nlink; /* number of hard links */
uid_t st_uid; /* user ID of owner */
gid_t st_gid; /* group ID of owner */
```

file metadata...contd.

```
dev_t st_rdev; /* device ID (if special file) */
off_t st_size; /* total size, in bytes */
blksize_t st_blksize; /* blocksize for filesystem I/O */
blkcnt_t st_blocks; /* number of blocks allocated */
time_t st_atime; /* time of last access */
time_t st_mtime; /* time of last modification */
time_t st_ctime; /* time of last status change */
};
```

Directory

Directories (or Folders) are the natural way of dividing a huge collection of assorted item to several classes of, possibly, related items.

- **Single level Directory** : Here all the files are availble under one directory (technically in a device without any further partition/division). Structure is simple but with numerous files locating a particular file is time consumingh. This was the case for early PC OS as well as the first supercomputer CDC 6600. This is still a preferred choice in some embedded system.
- **Hierarchical Directory** : This is typically a tree – we have several directories under the root directory and each directory may have files and sub-directories. From the organisation point of view this is the natural and preferred choice where a file (or directory) can be located easily and quickly by following the absolute path starting with the root directory (or using the relative path if we know where we are.

With the sharing of files/directories we establish links and the direcory-tree becomes DAG.

Directory Operations

A directory, like a file, has got operations like create, destroy, rename etc. for obvious reasons. Two special operations are elaboarted below:

- Link : allows a file to appear in more than one directory. It creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a hard link.
- Unlink : A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain.

A variant of the hard link is a soft-link where a name can be created that points to FILE naming another file.

File system implementation

A user is concerned about the file name, its attribute and the operations permitted. The OS on the other hand would create a system through which all the files are accessed and manipulated reliably and efficiently considering the characteristics of the devices on which it is implemented. A file system is stored on disk and may have multiple partition to accommodate different file systems.

- Sector 0* of the disk is called the MBR (Master Boot Record)
- The end of the MBR contains the partition table containing the start and end addresses of each partition. One of the partitions in the table is marked as active.

On power-on 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 (boot block) and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable OS.

[* Where the r/w head positioned after power-on – cylindar 0, head 0, sector 1 (the first sector)]

File system implementation contd.

Each partition is divided into

- Boot block – the program that loads the OS
- Superblock – all important parametrs (administrative) about the file system (read into memory at booting); often contains a magic number to identify the file-system type, no. of blocks etc.
- Free space management – information in the form of bit-map or list of pointers
- i-nodes – an array of data structures containing information about the file/directory
- Root directory–contains the top of the file system tree
- Files and directories – stored in the remaining areas of the partition.

Disk block allocation to files

There are many possible methods that are used in different OSs

- Contiguous
  - Very simple to implement – provided the requirement is known in advance. Initially when the disk is empty we have no problem – however with time it would create a big problem
  - Access performance is very good – only one seek
  - DISK fragmentation (external) – offsets advantages – as compaction is not a prudent choice.
  - In case of write-once (file sizes known) read many devices this allocation is perfect (e.g., CD).
- Linked List
  - Every block containing a pointer to the next
  - No external fragmentation
  - Slow access and amount of data is no longer in powers of 2 (due to the presence of the pointer) so reading full block data requires handling two blocks and some manipulations.
  - Solution to the problem cited above is keeping the pointers in memory in the form of a table. Such a table is known as FAT.

Disk block allocation to files

- FAT, while successful for a small system (used in MSDOS and still supported by WINDOWS) with small memory and disk, is not suitable due to the requirement of
  - keeping the entire FAT in memory all the time.
  - For a 512 GB disk and 2 KB block size it would be 1/4 billion entries, 4-bytres each. And that FAT would take 1 GB space all the time.
- I-node Index-node) is a data structure used in UNIX systems
  - I-node contains the file attributes and disk addresses of the file's blocks (usually 12 blocks – enough for small files)
  - For larger files 2nd level and 3rd level indirection is done by using pointers pointing to disk block that contains pointer to data blocks.
  - the i-node need be in memory only when the corresponding file is open. So, if we have k files open and each i-node requires n-bytes then the total memory requirement would be kn bytes. To limit the value of k most operating systems allow a maximum of m files that can be handled by a process [for MSDOS the default is 10 – can be increased though].

Directory

From the pathname we reach the directory entry for a particular file which will be opened for any operation. The directory entry leads to the disk blocks where the file is stored. This information could be

- Address of the first block – for contiguous allocation and linked list
- i-node number

In any case it is a mapping of the ASCII filename to the desired block address on the disk. Storing the attibutes (metadata like owner, creation time etc.) of the files need be considered as well.

- Can be stored with the directory entry. Simplest is a list of fixed size per file entries each containing file name, a structure of attributes and a limited number of disk bock addresses.
- For OS using i-node the directory entry can be short; filename and the i-node number while the atrributes are stored in the i-nodes. This is a better proposition.

For short file names (say, MSDOS with 8 character in the name and 3 in extension storing the name is not a problem. However, most modern OSs allow variable length file names.

Variable length file names

- The simplest approach is to allow, say, 255 characater file name that accommodates all practical cases
- However, most of the file names are short and a fixed length approach wastes a lot of directory space

To save the directory space each directory entry contains a fixed portion, typically starting with the length of the entry,

- and then followed by the fixed length header (Metadata) This fixed-length header is followed by the actual file name (however long it may be padded with blanks for word alignment. And then about the next file, in the same order. The problem is that with file removal we get gaps in the directory space

A better alternative is to have pointer to the file name and attributes in the directory space

- and maintain the list of files in a heap at the end of the directory – in this case no alignment is necessary.

Shared Files

It is often required to share a file by a group of people and the file to be visible in the directory of different users sharing the file. In UNIX this is done

- by establishing a hard link between one file (say, the original) [actually sevral copies of the file being shared with the same i-node number]. However, the link-count field in each copy is increased to indicate the sharing.
- Any change done by a user in his/her file is relfected in others. However, If one such copy (including the original) is deleted others remain.
- note that with links the directory structure takes new form; from tree to a DAG.

Another type of link is a soft or symbolic link which can work across filesystem. However, if you delete the original file (which is being shared) the link would would be dangling and sharing stops. Also, note that in case of soft link the link count field is not increased. Removing the symbolic link does not affect the original file.

LOG structured file Systems

The raionale behind the LFS are

- CPU are getting faster and faster
- Memories (core and secondary) are getting bigger
- Plus it is now possible to comply most of the read requests from the file system cache without any disk access (seek time is the bottleneck)
- Hence, most of the access would be disk writes – thus prefetching of blocks in anticipation, as done in many OSs, will no longer improve system performance.

Now, most of the writes are in small chunks that affects the performance as a 50 usec disk write is often preceded by a 10 ms seek and 4 ms rotation delay
One of the sources of small write is say creating a new file on UNIX To write this file, the i-node for the directory, the directory block, the i-node for the file, and the file itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

LOG structured file Systems contd.

In LFS the disk is being treated a great big log in such a way as to achieve the full bandwidth of the disk, even with small random writes. This is done by

- Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log
- A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together.
- At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

LOG structured file Systems contd.

To summarize

- all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log.
- Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it.
- All of the blocks will themselves be in segments, somewhere in the log

For any finite disk ultimately it would be full and no new segments can be written. However, many existing segments will have unused blocks and they will be identified and used (For this job LFS uses a cleaner thread that scans the log to compact it).

Journalling File System

A conventional file system may be made very robust by keeping a log which is very handy in case of system crash.

- The log registers what the file system is going to do before carrying out the actual operation.
- During reboot the system can check the log before the crash and finish the job.

To understand how is it beneficial let us consider the tasks we need for a trivial operation, namely, file removal. The steps are i) Remove the file from its directory; ii) release the i-node to the pool of free i-nodes, and iii) return all disk blocks to the pool of free disk bloks. Normally, the order of these 3 operations does not matter. However for a system class it does. Suppose that after step i the system crashes. During reboot there would be no information available on i-node or the blocks and they cannot be reassigned. Similar problems exist if we changed the order and the crash happens before completing the actions.

Virtual File Systems

On the same computer many different FS may exist even for the same OS. For example WINDOWS may have NTFS as well as

- old FAT-16 or FAT-32 drive or partition containing old but useful information
- occassional use of Flash drive, CD-ROM oe DVD each with its unique file system are laso supported

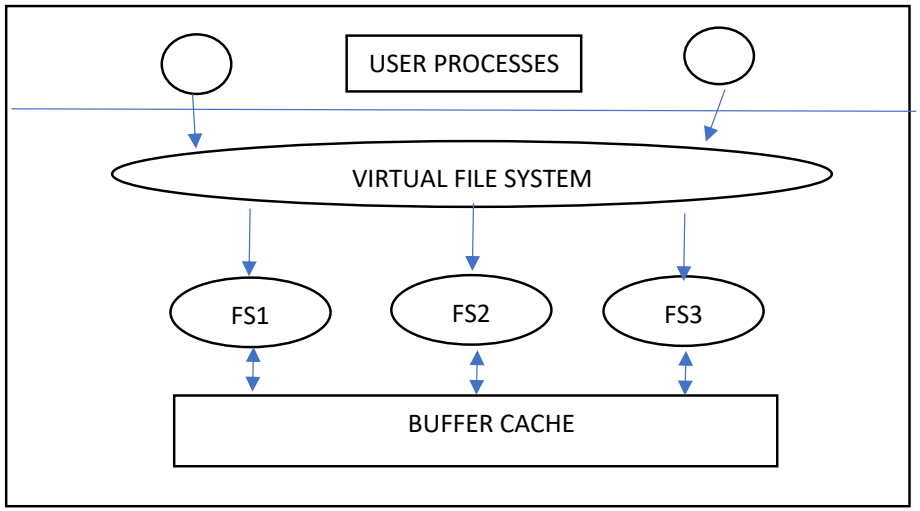
WINDOWS did not try to integrate these diverse system it uses drive letter C: D:; etc. The drive letter indirectly informs the WINDOWS to use the required file system.

In all modern UNIX systems, however, there is an attempt to integrate multiple file system to a single structure. For example, a LINUX system may have *ext2* as the root FS; with an *ext3* partition on /usr as well as CD-ROM temporarily mounted on /mnt. Thus the multiple (incompatible) FSs are not visible to the users or processes. However, it is very much visible to the FS implementation and a true virtual file system is supported in all UNIX installations.

Virtual File Systems contd.

UNIX tries to unify using a concept of a VFS

- The key idea is to abstract the common part of all FSs and put the code in a separate layer that calls the underlying actual FSs to manage the data.
- All system calls (POSIX) related to the files are directed to VFS for initial processing.
- The VSF has a lower level interface (shown by the downward arrows towards the actual FS) to manage the data by the actual FS



Implementation Issues: DISK BLOCK SIZE

File system is implemented on a disk. There are several issues to be addressed; these include block size, free block tracking, Disk quotas, etc. The choice of disk block size is very important so is a page size in any paging environment.

- Large block waste a lot of disk space though enjoy the higfher data transfer speed
- Small blocks waste less space bu higher access time (More access time as most of the files would spread accross multiple random blocks leading to high seek and latency).

It is imperative to select a block size which would contain most (say, 60–70%) of the files without wasting too much space. In a study at a University in 2005 it has been found that

- 60% of the files are 4 KB or less; 91% are 64 KB or less; while the median size is around 2.5 KB
- with a 4 KB block 93% of the disk blocks are used by 10% of the largest files – thus small files has got no significant effect as the disk is filled up with a small number of large files.

DISK BLOCK SIZE contd.

From the study of the utilisation of disk space (thus less wastage) and an effort to keep higher data-rate a graph is drawn from the research data shows

- High block size (after 128 KB) the data rate is linearly increasing
- for Small blocks while the utilisation increases lineraly (after 4 KB) and the data rate also follows the path.
- The curves met at 64 KB size with a data rate of 6.7 MB only. This is not acceptable considering the low data rate.

For these reasons most of the OSs has used block size as 4KB. However, with disk storage capacity reaching TB and beyond calls for larger size and we are moving towards 64 KB in near future

Tracking the FREE BLOCKS contd.

Two methods may be used i) Linked list of disk block numbers; ii) BIT MAP

i) Linked list

- Each block holding the address of as many blocks as possible
- 1 KB block and 4-bytes (32-bit) block number – can store 255 free block numbers – the last being used to point to the next 1 KB block holding the next set of free block numbers.
- for 1 TB disk with 1 KB block size the number of blocks would be 1 billion – so to store the free block linked list (each node holding 255 free block numbers) we need around 4 million blocks.

ii) BIT MAP

- n number of blocks is represented using n no. of bits in a bitmap (1=free; 0 = in use)
- For the same 1 TB disk with 1 KB block thus we need a billion bit bit-map that requires around 13,42,17,728 no. of 1-KB block to store
- for 1 TB disk with 1 KB block size the number of blocks would be 1 billion – so to store the free block linked list (each node holding 255 free block numbers) we need around 4 million blocks.

Tracking the FREE BLOCKS contd.

Free list system for tracking can be improved when there is a frequent run of consecutive free blocks

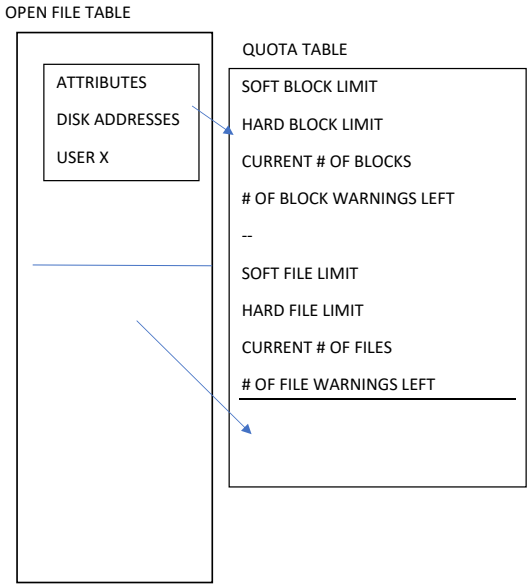
- Where we could store the address of the first block and the run-lngth
- However, severly fragmented disk may not have wider runs and keeping the start address and the count will not be advantageous.

Note that in free list method only one block of pointers need be kept in memory; thereafter more may be read from the disk. When a file is deleted the blocks being freed are added to the block of pointers in main memory.

DISK QUOTA

Multiuser OS usually provides a way of preventing blocking of unlimited disk space by enforcing quotas – both on the no. of open files and the disk blocks. The OS keeps track of these quotas and forces the users not to exceed them.

- When a user opens a file, the attributes and disk addresses are copied into an open-file table in memory.
- Any increase in file's size would be charged from the user's quota.
- A second table (quota-table) contains the quota record for every user with a currently open file.



DISK QUOTA contd.

. Every time a block is added to a file, the total number of blocks charged to the owner is incremented, and a check is made against both the hard and soft limits.

- The soft limit may be exceeded, but the hard limit may not.
- An attempt to append to a file when the hard block limit has been reached will result in an error.
- That is in a session when you log off quota should be within the soft limit.

Analogous checks also exist for the number of files to prevent a user from hogging all the i-nodes. When a user attempts to log in, the system examines the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks. If either limit has been violated, a warning is displayed, and the count of warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in.

FILE SYSTEM BACKUP

The real value of a computing system is the non-tangible the information contained in its file system. Destruction of the FS is a real disastar unless the back-up is ready.

- There cannot be an absolute protection for the physical damage
- However, FS back-up may be taken at regular intervals to prevent the fiasco.

Making a backup takes a long time and occupies a large amount of space, so

- First, should the entire file system be backed up or only part of it?
- At many installations, the executable (binary) programs are kept in a limited part of the file-system tree. It is not necessary to back up these files if they can all be reinstalled from the manufacturer's Website or the installation DVD.
- Also, most systems have a directory for temporary files. There is usually no reason to back it up either.

In short, it is usually desirable to back up only specific directories and everything in them rather than the entire file system.

FILE SYSTEM BACKUP contd.

It is wasteful to back up unchanged files since the previous back-up. This leads to incremental dumps.
The simplest incremental dumping is to take a full back-up (dump) periodically may be an interval of a month and take daily dumps of the files that changed after the last full dumps
Since an immense amount of data is to be dumped it may be desirable to take a dump in *compressed* form. However, a single r/w error of the compressed dump may lead it useless.

Also, it is difficult to take back-ups on active file system where the files and directories do change frequently. So, back-ups can be taken during the night – not acceptable in many cases. For this reason some algorithms are employedto take rapid copy of the FS by copying important data structures – we may call it a snap-shot and then make required changes on the files and directories at leisure afterwards.

FILE SYSTEM BACKUP contd.

Two strategies are used to take a dump; i) A physical dump or ii) a logical dump. Physical dump: Simple strategy where the source block 0 is copied to destination (dump disk/tape) 0 and continues until the end of soruce block (say 'n'). The points to ponder are

- No point copying unused disk blocks. This can be improved if the dumping program can get the free list.
- Note that skipping the unused block woud require to write the block number in each block as there will be no guarantee that the k-block of the disk is going to k-block of the back-up.
- Dumping of bad blocks – some of them are always present after formatting. It is managed by creating a file consists of the BAD blocks to gurantee that they may not appear in the list of free blocks. If the bad block information can be passed to the disk-controller for remapping tehn it is fine. Else, they should be visible to the OS in the form of bad-block list or bit maps for proper back-up.

FILE SYSTEM BACKUP contd.

Logical dump: It starts at one or more specified directories and recursively dumps all files and directories that have been changed since the last dump

- Dump gets a series of carefully chosen files and directories
- And is easy to restore

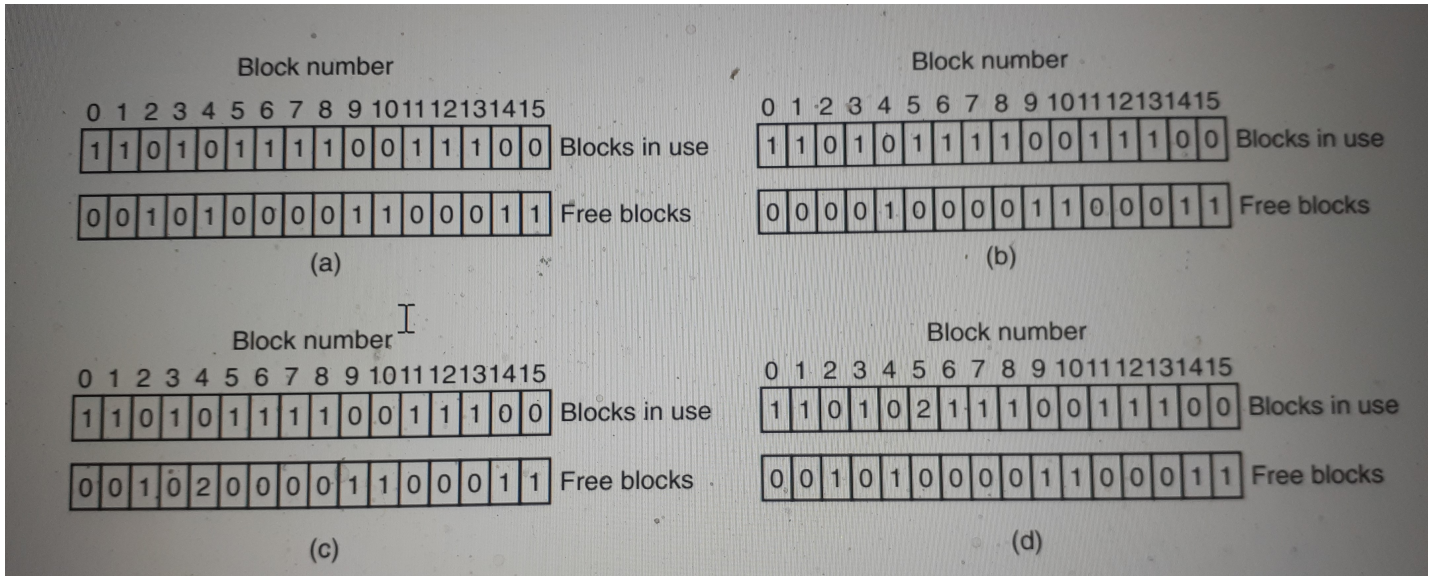
Logical dumping is the most common form and used in most OS. The algorithm used for dumping takes dumps of all the directroies (even the unmodified one) on the path to a modified file or directories – i) if the FS is restored in another computer; or ii) incrementally restore a single file on a path where the directory right in the middle of the path was removed.

FILE SYSTEM CONSISTENCY

OS read blocks, modifes them and for performance reason make delayed write. Any crash before the write might make the system inconsistant; particularly for blocks containing i-nodes, directories, or the free list – if they are not written back to disk.
OS provides utilities; e.g, UNIX has *fsck* and WINDOWS offer *sfc*. This utility can run during booting or after a system crash. Consistency checks are made on i) blocks or ii) files

- blocks
  - Two tables are maintained - each entry is a counter (initially, 0) for each block.
  - One is counting how many times each block is present in a file; and the other counts how often each block is present in the free list.
  - If the FS is consistent then the each block count in one table would be 0 (or 1) and the same in the other table is 1 (or 0). i.e., the count os used block and the corresponding free block would be complementary.

FILE SYSTEM CONSISTENCY contd.



Case (a) in the figure shows a consistent FS. However in case (b) we see block 2 is missing; present neither in used list or free list.

- This scenario is not harmful except wastage of space
- the FS checker puts the missing blocks on the free list as a remedy

Another scenario depicts (case (c)) the presence of a block (number 4) in the free list twice.

- Here the free list is rebuild to remove duplicates.

However, case (d) – we see that block 5 count is 2 in the used list – that cannot

FILE SYSTEM CONSISTENCY contd.

Removing both the files with the same block

- would make the block count as 2 in the free list

If one of the files is removed then

- would make the block present (count as 1 in both the lists) in both the used and free list

The appropriate action is

- to allocate a free block,
- copy the contents of block 5 into it and insert that copy to one of the files.
- The consistency would be intact with the
- chance of some garbage in one or both the files — so the owner must be informed to repair the damage; if any.

FILE SYSTEM PERFORMANCE

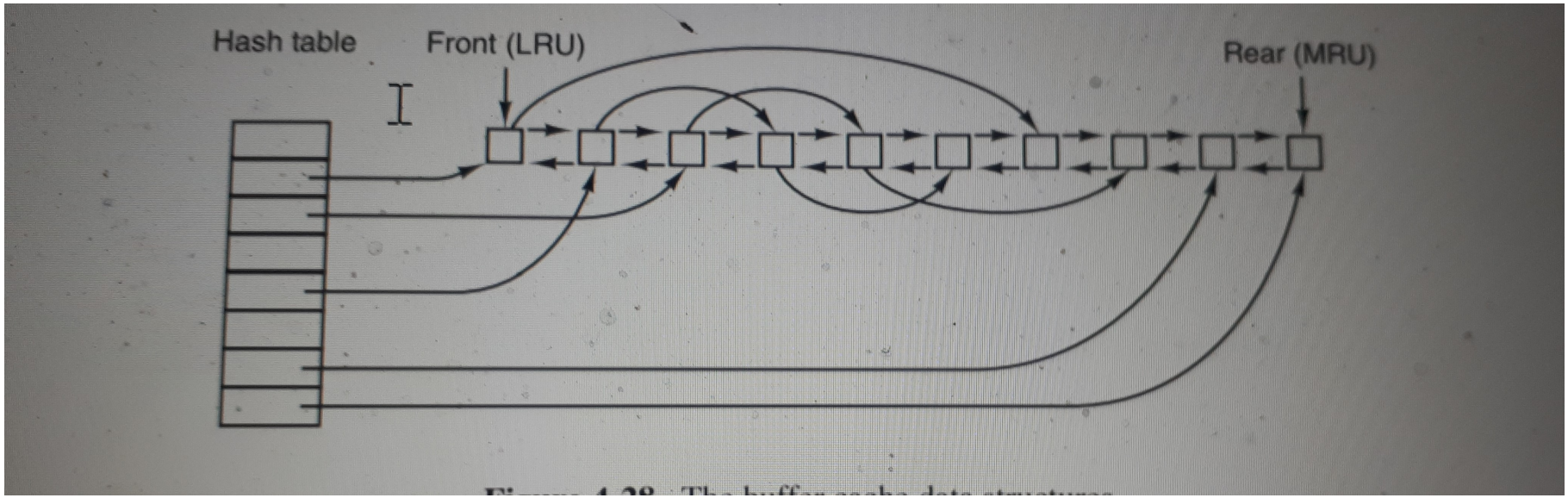
Avoiding disk access, if we could, improves FS performance. Several measures that are taken on this aspects include

- disk caching
- block read ahead
- rducing disk arm motion

The most common technique is to maintain a block or buffer cache in the memory

- it is checked to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access.
- If the block is not in the cache, it is first read into the cache and then copied to wherever it is needed.

FILE SYSTEMS PERFORMANCE



- With thousands or more active blocks in the cache faster access is an important issue.
- Usually hashing on the device and disk address to find out the required block in the cache.
- If the block need be brought to the full-cache an existing block need be written back and the situation is like paging – so a modified LRU kind of algorithm may be used with the checking "is the block essential to the consistency of the file system"

BLOCK READ AHEAD: This technique is also used to improve the performace where the some blocks adjacent to the requested block is read and place in the cache to increase the hit rate.

- when block 'k' from a file is read – the cache checks if the adjacent block k+1 is already available; if not
- it is read and placed in the cache
- Considering the locality of reference this read-ahead strategy may increase hit rate and improve file system performance.
- This strategy works if the access is sequential (Note that even for random access many blocks are stored in consecutive locations in the disk)

FILE SYSTEMS PERFORMANCE

REDUCING DISK ARM MOTION: Latency and seek time in general and seek in particular takes a very high access time (with SDD this is not a problem) so allocation strategy to reduce the disk-arm movement may be used for performance boosting

- by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder.
- If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block.
- With a free list, part of which is on disk, it is much harder to allocate blocks close together.
- However, even with a free list, some block clustering can be done by taking a group of consecutive blocks as an allocation unit.
- by putting i-node blocks on the middle of the disk could reduce seek time
- Even with SSDs which has got limitations on the number of writes certain strategies to evenly distribute disk block write is important

DISK DEFRAGMENTATION

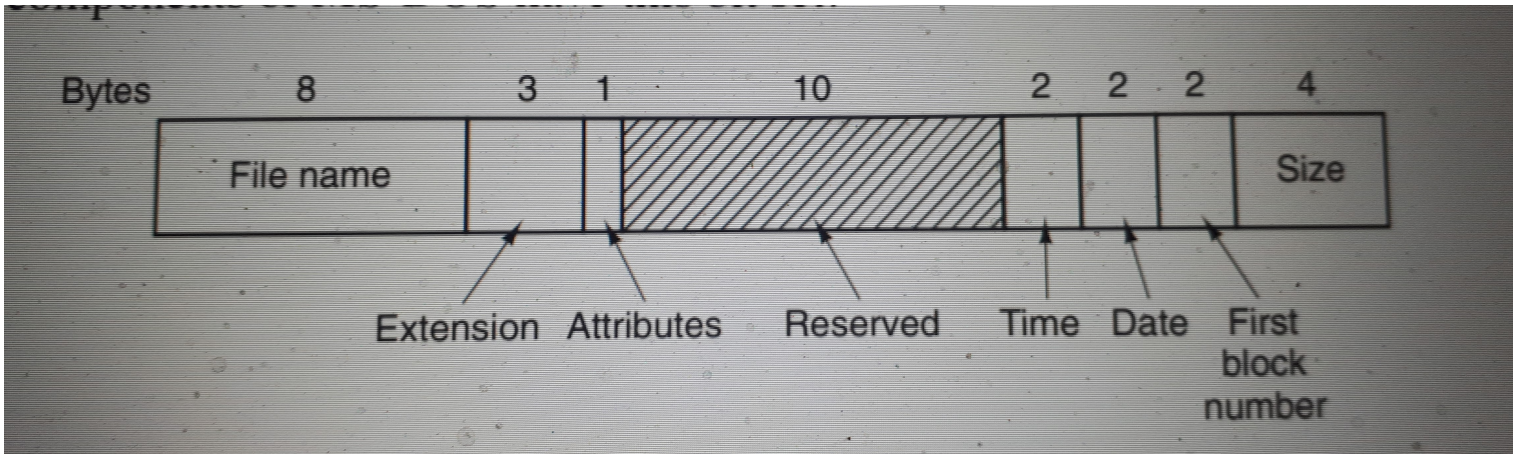
After installing the OS all free disk space is a single contiguous unit. Thus furher allocation to user files are also contiguous and you have a high performing FS. However with time due to file removal and new files that does not fit in the hole the disk becomes badly fragmented. So, the blocks of a file spread are random and spread all over the disk.
Performance cen be boosted by moving the files around to make them contiguous.

- Defragmentation works better if you have a lot of free space at the end of the partition.
- Some files cannot be moved like journaling log etc. as the overhead of moving them is too high.
- In some system most important data-strucre is kept in fixed size contiguous block – they are not fragmented and need not be moved.
- With a free list, part of which is on disk, it is much harder to allocate blocks close together.
- However, even with a free list, some block clustering can be done by taking a group of consecutive blocks as an allocation unit.

EXAMPLE FILE SYSTEM: MSDOS

MSDOS FS which was offered in the earliest PCs are still supported in WINDOWS system. Morover, it (and its extension FAT 32) is still used in many embedded systems; such as digital camera, MP3 player etc.
MSDOS directroy has a variable size but uses a fixed size (32 bytes) entry .

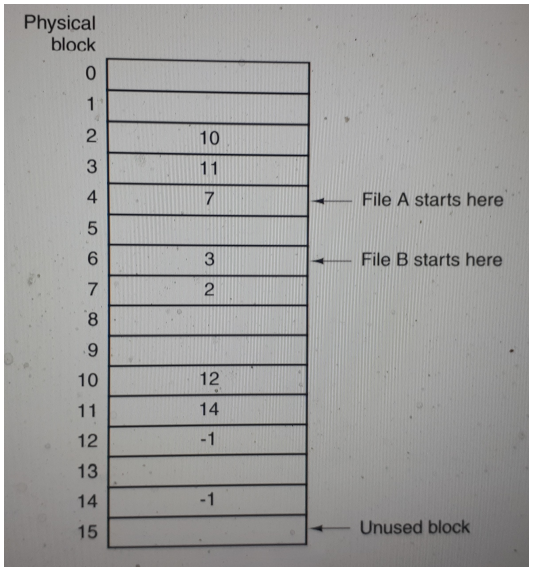
- The name and extensions are padded on the right; if required
- Attribute bits include RO, ARCHIVE, HIDDEN, SYSTEM. Using the ARCHIVE bit allows a user level application can reset the bit after archiving the file– while other program can set it indicating modification and the need for further archive.



The time field has second(5-bits), minutes (6-bits) and hours (4-bits) sub-fields. The date field subdivides to day(5-bits), month (4-bits) and year (7-bits). The base year is 1980 – so you can go p to 1980 + 128 -1 = 2107 after that it would be a Y2108 problem (like y2K).

- SIZE is expressed in 32 bit – so theoretically it is 4 GB
- For all FATs (FAT-12, FAT-16 and FAT-32) , the disk block can be set to a multiple of 512 bytes. Note that the numbers 12, 16 etc are indicating the disk block address length in the FAT and are progressively made bigger to accommodate larger disks.

Note that the 2 byte or 16-bit directory entry is indexed to a 64KB FAT in memory



Here in the FAT all the blocks (4, 7, 2, 10, and 12) of File A can be accessed using the FAT.

The FAT to keep track of free disk blocks as well. Free blocks are marked with a special code. When MS-DOS needs a new disk block, it searches the FAT for an entry containing this code. Thus no bitmap or free list is required.

FAT-12 with 512 bytes blocks were fine for the floppy disk era. Thereafter to accommodate larger HDD FAT -12 allowed a blocksize of 1, 2, and 4 KB as well. In order to cope up with the larger HDDs FAT 16 was introduced and finally FAT 32 with bigger allowed block sizes.

Block Size(KB)	FAT-12 (MB)	FAT-16 (MB)	FAT-32 (TB)
0.5	2		
1	4		
4	16	256	1
32		2048	2

So, the maximum partition size becomes 16 MB. FAT-16 could have a 2048 MB (with block size 32 KB) and the same for FAT32 is 2 TB. Note that in FAT-32 we could have 4KB block for a 2 GB partition and bigger blocks for other partition storing bigger files offering a flexible solution.

FS – UNIX V7 contd.

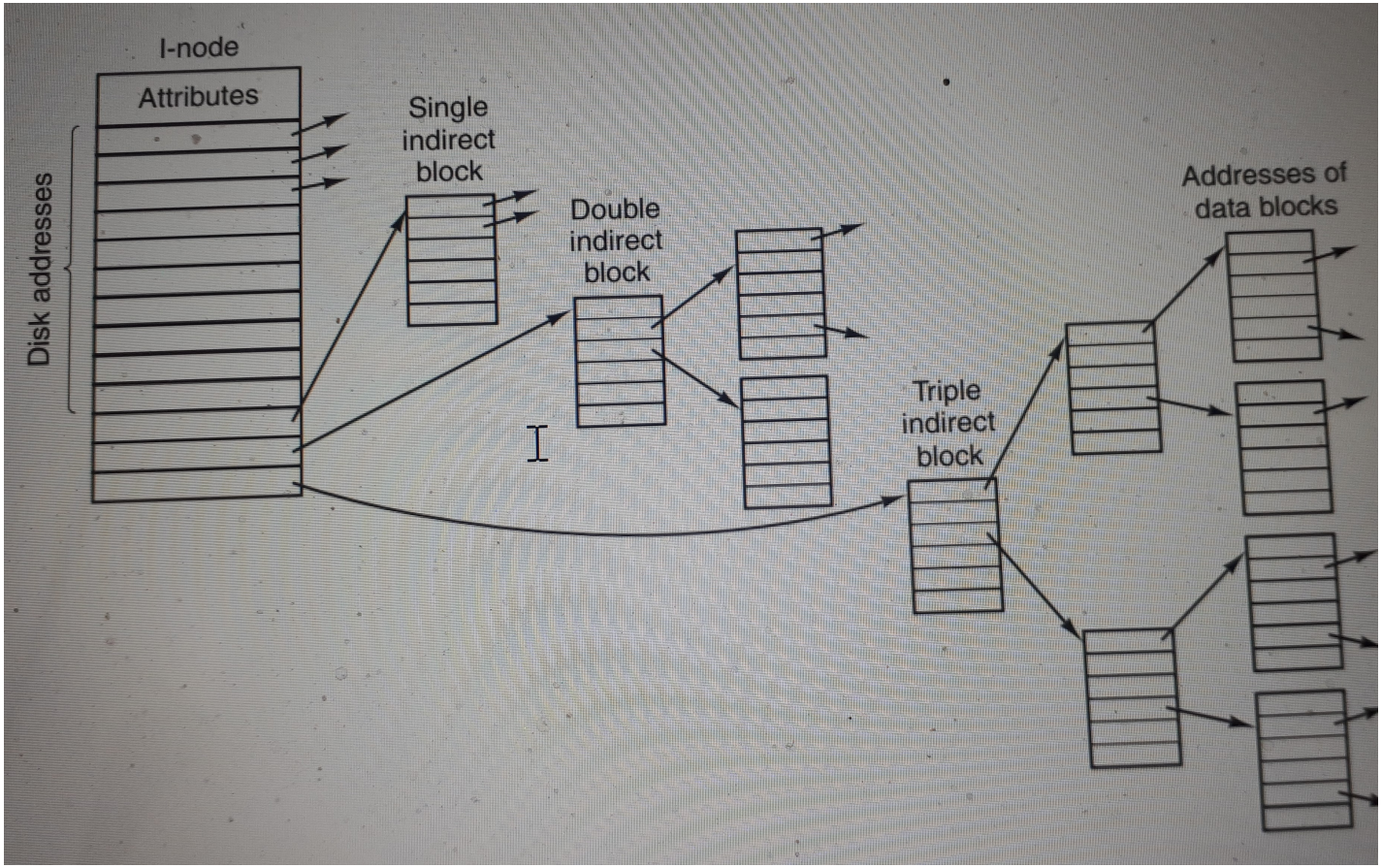
UNIX has eveolved from a solid technical foundation and for that reason even the early UNIX FS like V7 (used in DEC PDP 11 – most popular mini-computer in any Institute during 70s) is still relevant as an approach to a long standing standard of a FS.

- It is a tree structured FS that took the form of a DAG if we add the links.
- File names can be up to 14 ASCII charcaters except '/' and 'null'
- Directory entry contains entry for each file in that directory
- Directory entry (16 bytes) has a very simple form: A 2-byte i-node number (so, the maximum number of files in a file system is 64K) and a ii) 14-byte filename

i-node (2)	File name string (14)

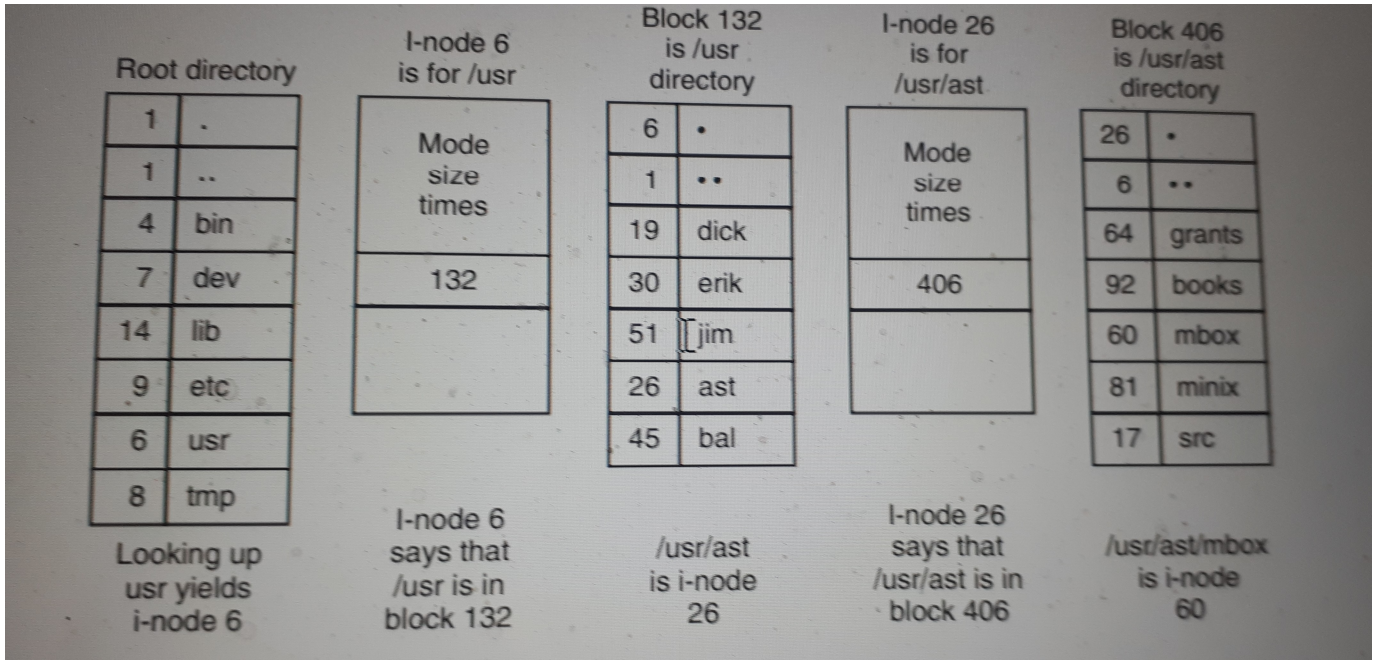
UNIX i-node contains attributes that includes file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory entries that point to the i-node. The rest of the i-node contains block address in a 3-levels of indirection.

FS – UNIX V7 contd.



i-node directly holds 10 disk block address to access small files – to cater to the need of much larger files – it holds pointer to a disk block that holds the addresses of disk blocks (single indirect) ... this is continued to 2 more levels (i.e., double and triple indirect)

FS – UNIX V7 contd.



Let's see how the path name /usr/ast/mbox is looked up. First the FS locates the root directory which is available at a fixed place on the disk.

- then the first component of the path '/usr' is located to find the i-node no. of the same (locating an i-node from its number is straightforward, since each one has a fixed location on the disk)
- Next the component 'ast' is looked up – when we get the component we get the i-node for /usr/ast – from here we finally i-node (consequently the disk blocks) for the file mbox

The i-node for this file, mbox, is then read and kept in memory until it is closed. Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of from the root directory.

- Every directory has entries for . and .. The entry '.' has the i-node number for the current directory, and the entry for '..' has the i-node number for the parent directory.
- Thus, a procedure looking up ../dick/prog.c simply looks up '..' in the working directory, finds the i-node number for the parent directory, and searches that directory for dick. No special mechanism is needed to handle these names.
- As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

Note that the '..' in the root directory points to itself as a special case.

File System Implementation: Revisited

The most visible items of a computer system from the user's perspective are files (users, executable, system files etc.) stored in the persistence memory (disk: hard or SSD). For the ease of access starting from the root the files are organised logically as a tree – where the leaf nodes are usually the files.
For a better understanding of the FS we would try implementing an experimental small and simple FS based on UNIX FS. We have to think of

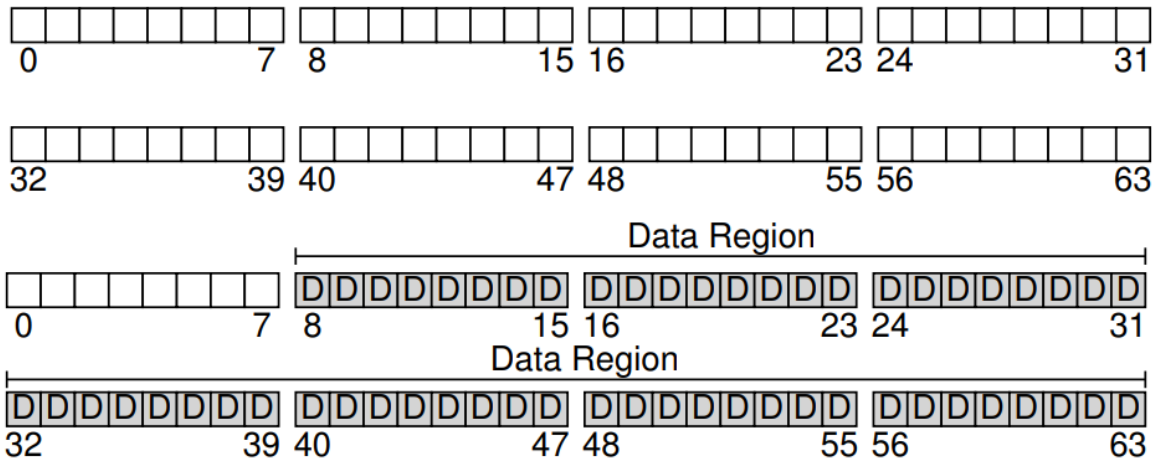
- The data structure of the FS — structure to keep the metadata and the data of a file (This could be a simple array of blocks in our case)
- Access methods; i.e., how do we map the open(), read, write and other calls interacting with the FS onto its structures? Which structures are read during the execution of a particular system call? Which are written? How efficiently are all of these steps performed?

File System Implementation: An example

The first thing to implement the FS is a logical structure that resembles a disk. Let us consider the following

- We have 64 4K-blocks (really a small hypothetical disk – however to get the idea it is enough) these steps performed?
- It is quite natural that most of these blocks would be used to store i) data (D) and others for storing the metadata; ii) information about the files stored (inodes etc.) and and iii) information about the FS itself (also known as superblock)

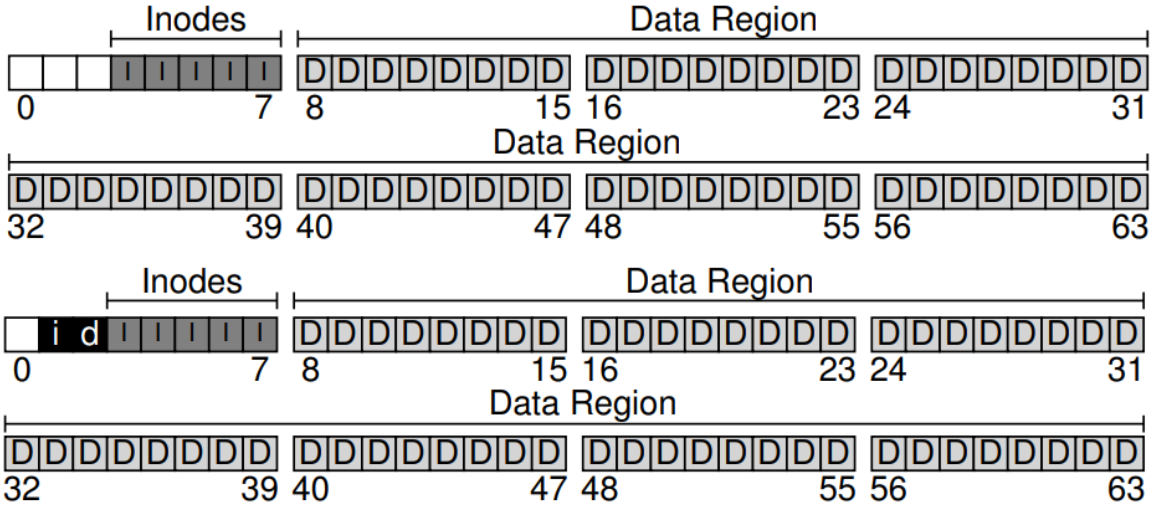
The figures shows 64 disk blocks of which 56 would be used to store data (files actually). 8 blocks (0 to 7) are kept for administrative purpose (inode, free block list and super blocks)



File System Implementation: An example ...contd.

Explicit information about each file. This information (i-node) is a key piece of metadata, and tracks things like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, etc. Let's call this portion of the disk the inode table, which simply holds an array of on-disk inodes. Thus, our on-disk image now looks like this picture, assuming that we use 5 of our 64 blocks for inodes (denoted by I's in the diagram)

- for 256 byte i-node we have a maximum of $5 \times 4096/256 = 80$ files.
- To store allocation information on Data and I-node blocks we need, say, 2 more blocks ('i' and 'd') indicating free or allocated i-node and data blocks.

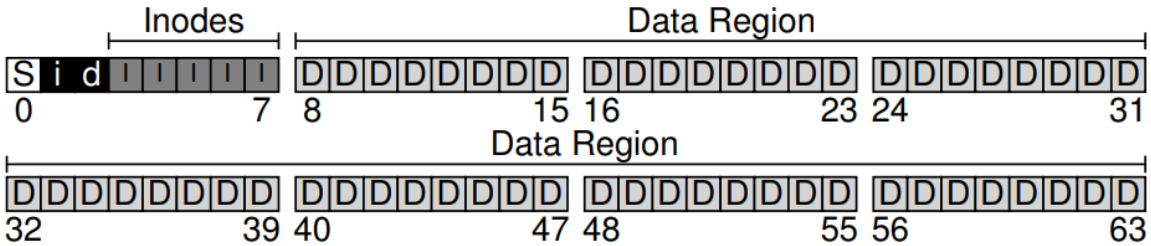


File System Implementation: An example ...contd.

Many allocation-tracking methods are possible.

- a free list that points to the first free block, which then points to the next free block, and so forth.
- Bitmaps, one for the data region (the data bitmap), and one for the inode table (the inode bitmap) may be used.

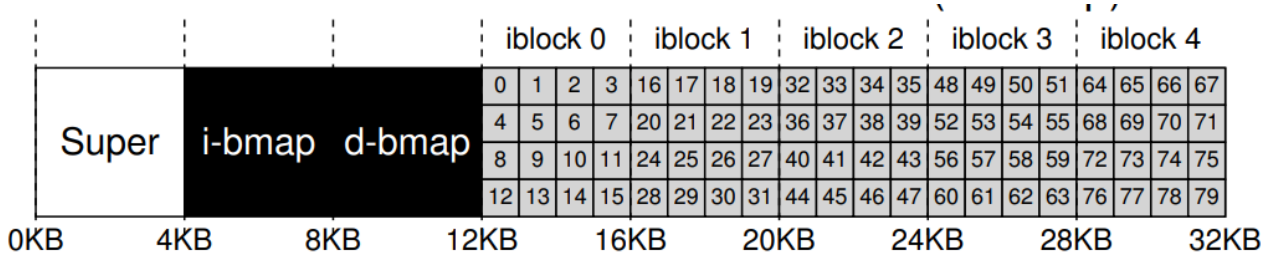
In bitmap each bit is used to indicate if the block is free (0) or not (1). We have an inode bitmap (i) and a data bitmap (d). Note that for 80 i-nodes we are using full 4 KB – for simplicity. Finally, one block is used as superblock (S) that contains information about this particular, including, how many inodes and data blocks are in the file system (80 and 56, respectively in this instance), where the inode table begins (block 3), and so forth. It will likely also include a magic number (MS DOS EXE uses 4A 5D) of some kind to identify the FS.



I-node Details

One of the most important on-disk structures of a file system is the inode (index node); virtually all file systems have a structure similar to this.

- Each inode is implicitly referred to by a number (called the i-number), or the low-level name of the file.
- from a given an i-number we may get the disk location of corresponding i-node.



To read i-node no. 32. We get the desired block address 32 * sizeof(i-node) + i-node block start address = 32 * 256 + 12 K = 8KB + 12KB=20KB. Note that disks are not byte addressable rather sector addressable and for a sector size of 512 bytes the sector address is 20 /(1/2) = 40. So, reading sector no. 40 gives us the details of i-node number 32. The i-node conatins all essential information about the file like *type*, *size*, no. of blocks allocated, protection information etc.

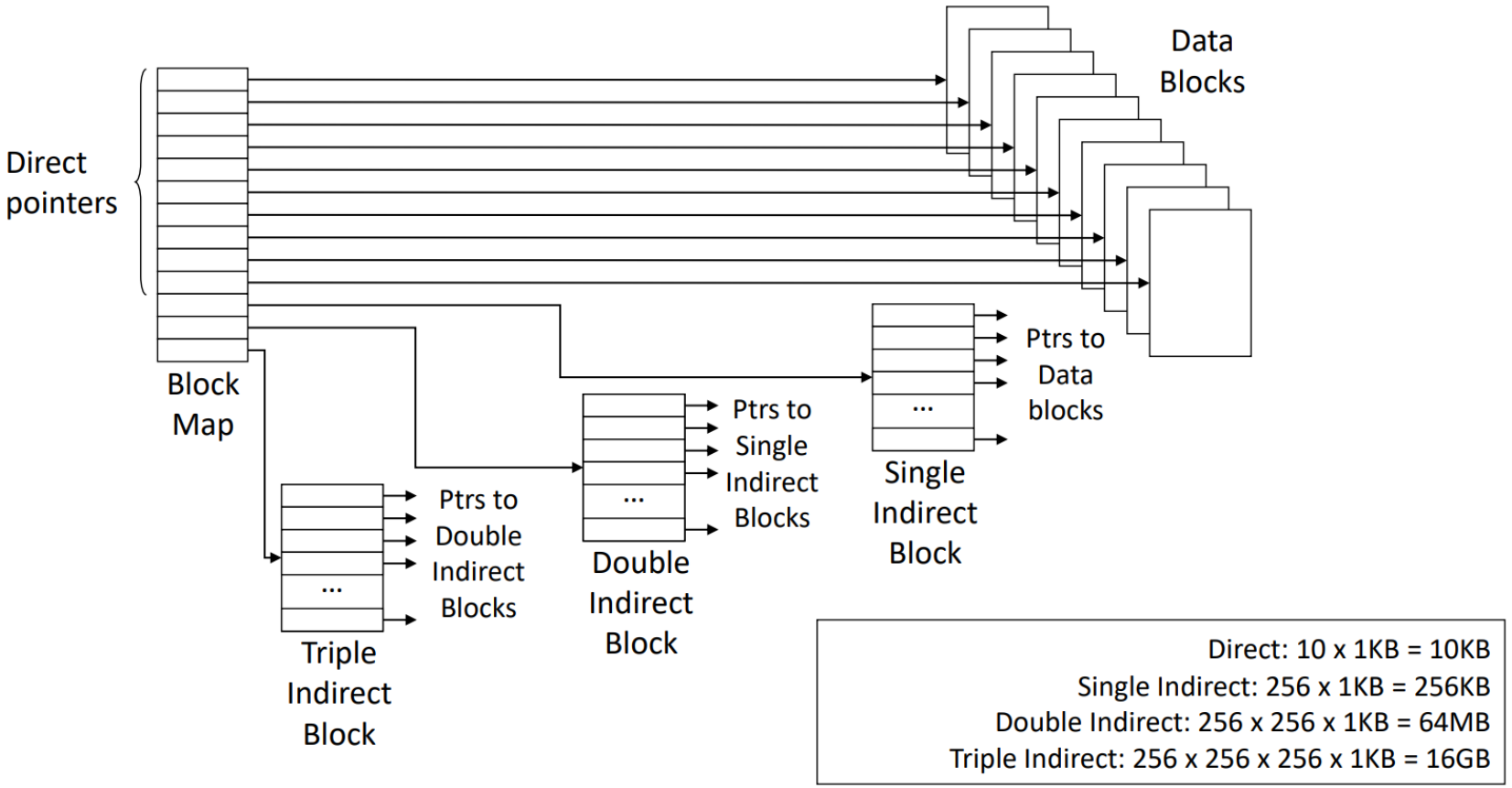
I-node fields

Here is an example of the i-node fields for the ext2 i-node (used in Linux)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

I-node fields ...contd.

One of the most important decisions in the design of the inode is how it refers to where data blocks are. The unix-inode approach is phenomenal as it has got a provision to support very small file to very big one. Note that it has got 10 direct data block pointer. One single indirect block pointer. One double indirect block pointer and one triple indirect block pointer. See the bottom right to get the address capacity using 1 KB block.



i-node fields ...contd.

Of course, in the space of inode design, many other possibilities exist; after all, the inode is just a data structure, and any data structure that stores the relevant information, and can query it effectively, is sufficient. As file system software is readily changed, you should be willing to explore different designs should workloads or technologies change.
Reagrding some statistics on the files which would have some bearing in designing a file system and using a particular data structure for i-node see the table below.

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Directory Organisation

In many file systems directories have a simple organization. A directory

- just contains a list of (entry name, inode number) pairs.
- For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory.
- For each string, there may also be a length (assuming variable-sized names).

For example, assume a directory dir (inode number 5) has three files in it (foo, bar, and foobar), and their inode numbers are 12, 13, and 24 respectively. The on-disk data for dir might look like this:

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

The directory contains dot and dot dot entries – the dot indicates current directory (dir in our case) and dot dot indicates the parent (root in our case).

Directory Organisation ... contd.

Some pertinent points reharding the directory:
Deleting a file (e.g., calling unlink()) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g., with a reserved inode number such as zero). Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within.
Often, file systems **treat directories as a special type of file**. Thus, a directory has an inode, somewhere in the inode table (with the type field of the inode marked as "directory" instead of "regular file"). The directory has data blocks pointed to by the inode (and perhaps, indirect blocks); these data blocks live in the data block region of our simple file system. Our on-disk structure thus remains unchanged. We should also note again that this **simple linear list of directory entries is not the only way to store such information**. As before, any data structure is possible. For example, XFS [S+96] stores directories in B-tree form, making file create operations (which have to ensure that a file name has not been used before creating it) faster than systems with simple lists that must be scanned in their entirety.

Directory Organisation ... contd.

Free space Management: A FS must keep track of the i-nodes and data blocks that are free or not so as to allocate blocks and i-nodes whenever you create a file/directory. We may have simple bitmaps for the job. For example, when we create a file

- we will have to allocate an inode for that file. The file system will thus search through the bitmap for an inode that is free, and allocate it to the file;
- the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information. A similar set of activities take place when a data block is allocated

We may have some **pre-allocation** policy as well, like ext2 FS where a contiguous 8-free blocks are found for data blocks during file creation so that some portion is guranteed to be available on the disk in contiguous blocks for better performance.

Disk read/write operations

Let us see the micro operations to carry out open a file, read it and close. Also assume that the FS is mounted and superblock is read and ready in the memory. Steps for open("/foo/bar", O_RDONLY)

- The file system must traverse the pathname (/foo/bar) and thus locate the desired inode.
- First thing the FS will read from disk is the inode of the root directory whose i-number is fixed (2 in unix system) Thus, to begin the process, the FS reads in the block that contains inode number 2 (the first inode block).
- Once the i-node is read it can then see the pointers to data blocks that contains the contents of the root directory .... and finally finds i-node number for 'foo'; say it is 44.
- This way finally the i-node number of 'bar' is found
- The open() call reads the i-node of 'bar' checks the permission and allocate a file descriptior (fd) in the per-process open-file table and returns.
- Now after opening read() call(s) are made to read the contents of 'bar'

Disk read/write operations

The first read (at offset 0 unless lseek() has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block; it may also update the inode with a new last accessed time. The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc. The steps are shown in the following table.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
open(bar)			**read**			**read**				
				read			**read**			
					read					
read()					**read**					
					write			**read**		
read()					**read**					
					write				**read**	
read()					**read**					
					write					**read**

Disk read/write operations

The write operation is similar; at first it must be opened and then the application can issue write() calls to update the contents of the file and finally closes it.

- Unlike reading, writing to the file may also allocate a block (unless the block is being overwritten, for example).
- When writing out a new file, each write not only has to write data to disk but has to first decide which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap).

Thus, each write to a file logically generates three I/Os:

- (i) one to read the data bitmap, which is then updated to mark the newly-allocated block as used,
- (ii) one to write the bitmap (to reflect its new state to disk), and one
- (iii) to write the actual block itself. So, quite an involved task. However, creating a file is more complex,

Disk read/write operations to create a file

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
create (/foo/bar)		**read** **write**	**read**	**read** **write**	**read** **write**	**read**	**read** **write**			
write()	**read** **write**				**read** **write**			**write**		
write()	**read** **write**				**read** **write**				**write**	
write()	**read** **write**				**read** **write**					**write**

Disk read/write operations to create a file

Figure (previouse slide) shows what happens during the open() (which creates the file) and during each of three 4KB writes.

- In the figure, reads and writes to the disk are grouped under which system call caused them to occur, and the rough ordering they might take place in goes from top to bottom of the figure.
- You can see how much work it is to create the file: 10 I/Os in this case, to walk the pathname and then finally create the file.

How can a file system accomplish any of this with reasonable efficiency? Number of I/O operations scattered over different racks/sectors must be reduced to improve FS performance. The answer is exploiting the disk storage structure as well as caching and buffering.

Caching and buffering

Reading and writing files can be expensive, incurring many I/Os to the (slow) disk. To remedy, what would clearly be a huge performance problem, most file systems

- Use system memory (DRAM) to cache (so called, RAMDISK: usually 10% of the system memory is kept for disk-caching during booting) important blocks
- Modern systems integrate virtual memory pages and file system pages into a unified page cache. In this way, memory can be allocated more flexibly across virtual memory and file system, depending on which needs more memory at a given time.

Effect of caching on write Note that with large cache I/O may be largely avoided – however write I/O is a must to maintain consistency with the disk-data. So, write cache is not as effective as read-cache. To improve **write-buffering** is used to improve write I/O. The write buffering has two advantages i) by delaying write – the FS can batch some I/O operations; ii) the FS can batch a number of writes and then schedule the subsequent I/Os' improving performance. Write operations are thus buffered from 5 to 30 seconds. [Note Database systems may not have the luxury of write buffering]