BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE 11
FILE SYSTEMS - IMPLEMENTATION (CHAPTER 11)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- NFS
- Example: WAFL File System

Objectives

- To describe the details of implementing local file systems and directory structures
- To describe the implementation of remote file systems
- To discuss block allocation and free-block algorithms and trade-offs

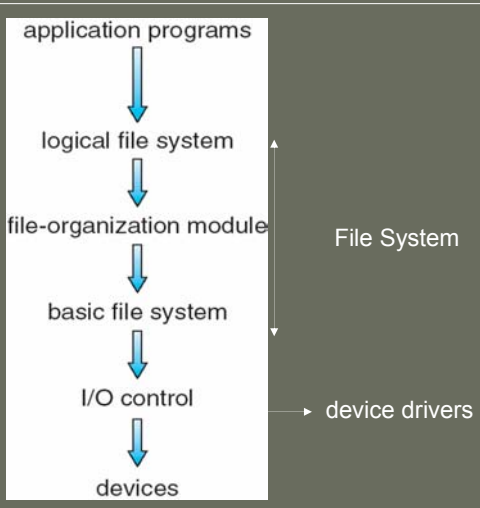
File System Design

- File System Design Involves
  - 1) Defining File System Interface
    - How file system looks to the user
    - What is a file and its attributes
    - What are the operations
    - (logical) directory structure that can be used to organize files
  - 2) How that file system can be implemented
    - Design algorithms
    - Design data structures (in-memory and on-disk data structures)
    - Map logical file system to physical storage device (disk, tape, etc)
      - Storage device dependent

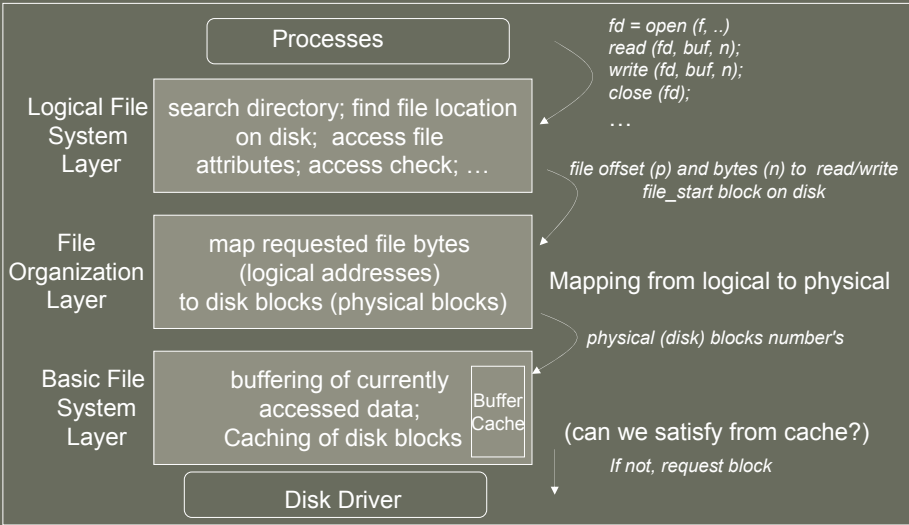
File System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- File system organized into layers
- File system resides on secondary storage (disks)
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
  - Can also sit on other media (USB disk, CD-ROM, etc). Usually need a different file system
- File control block – storage structure consisting of information about a file
- Device driver controls the physical device

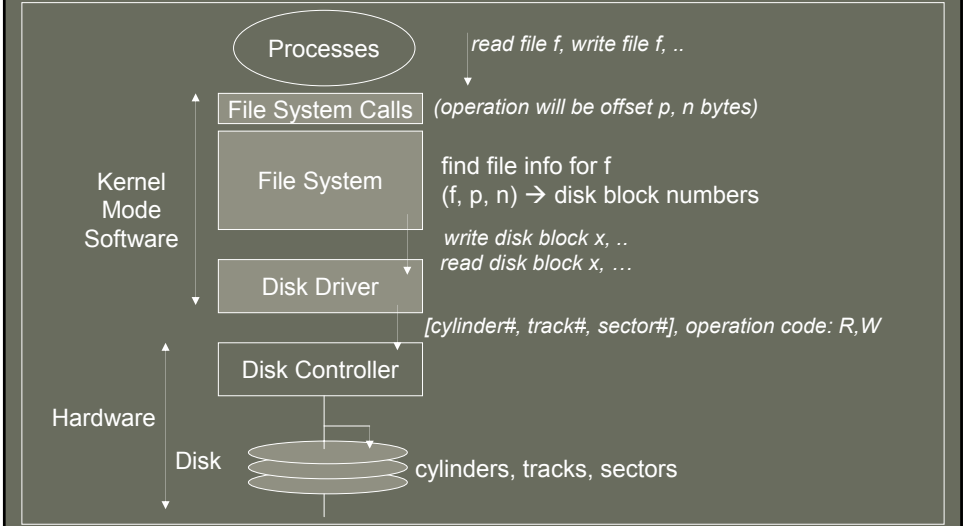
Layered File System



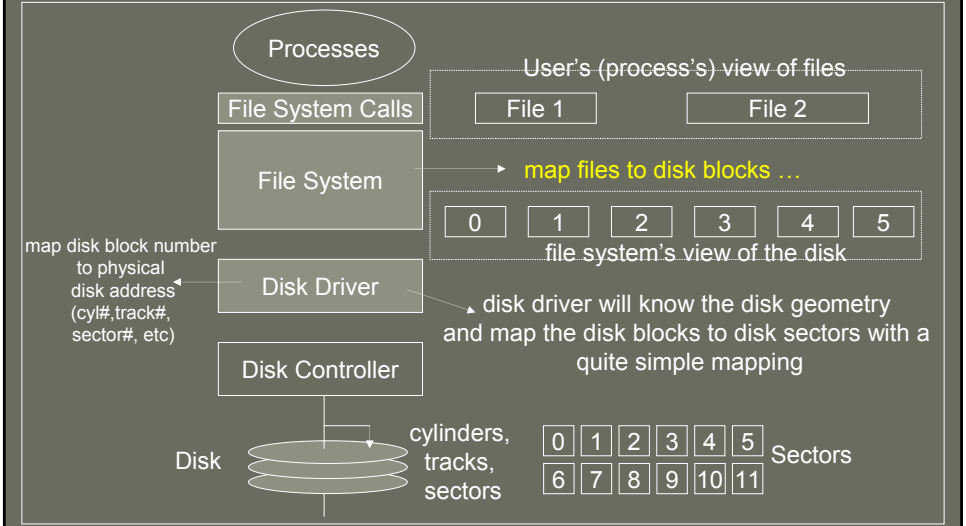
Layering File System



Layered Software

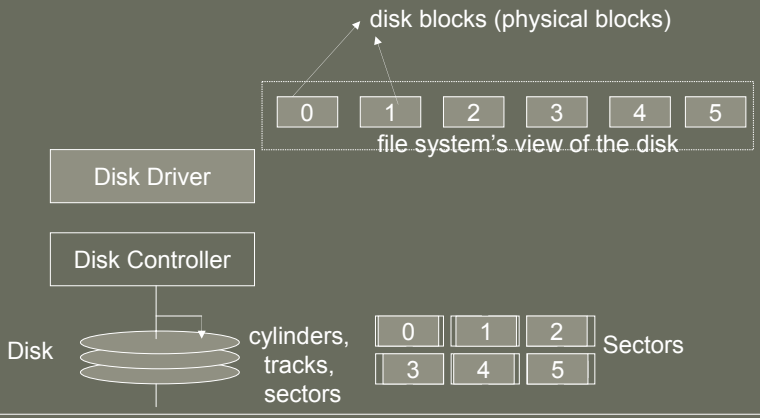


Layered Software

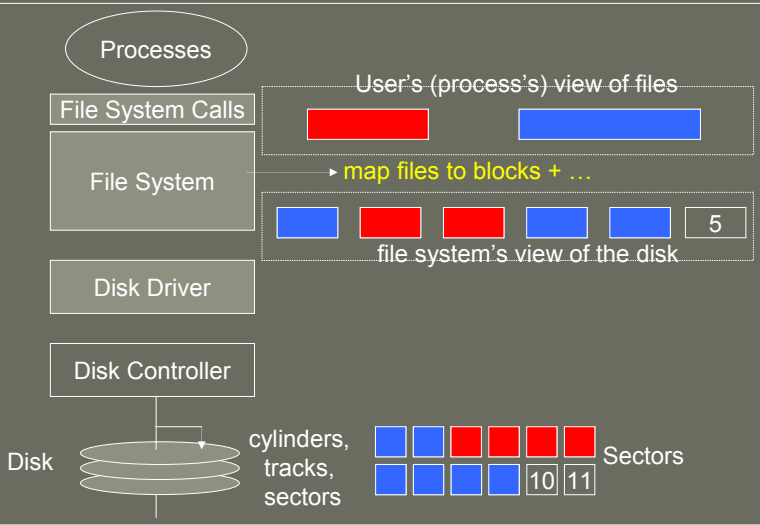


Disk Driver: Mapping disk blocks to physical disk sectors

Block size is a multiple of sector size.
Example: sector size can be 512 bytes; block size can be 1024 bytes, or 4096 bytes.

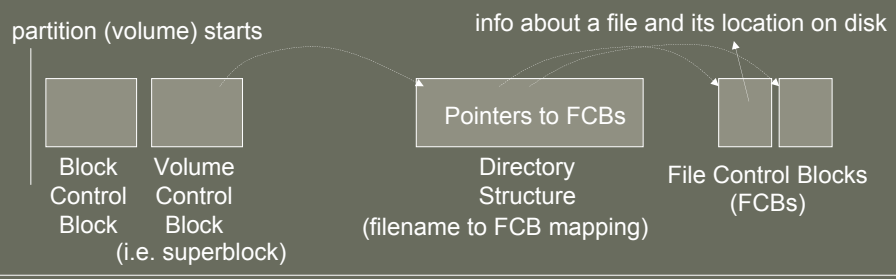


Example mapping files to blocks and sectors

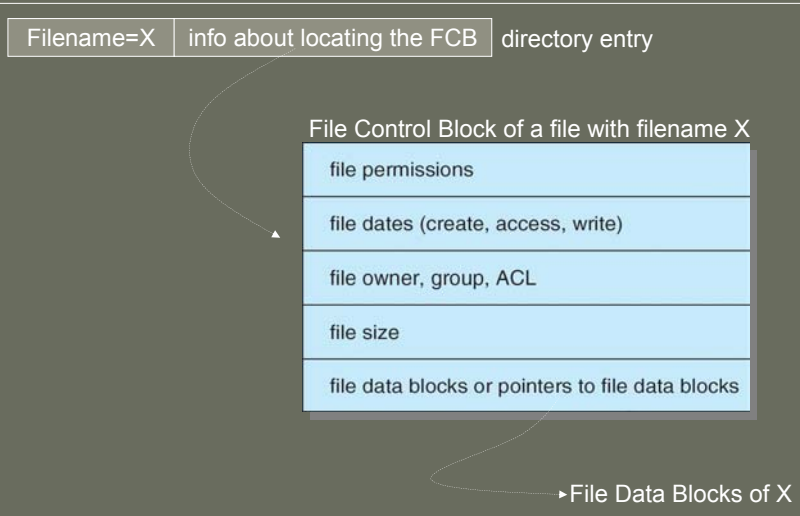


File System Implementation

- Major On-disk Structures (information):
  - Boot control block contains info needed by system to boot OS from that volume
  - Volume control block contains volume details
  - Directory structure organizes the files
  - Per-file File Control Block (FCB) contains many details about the file



A Typical File Control Block



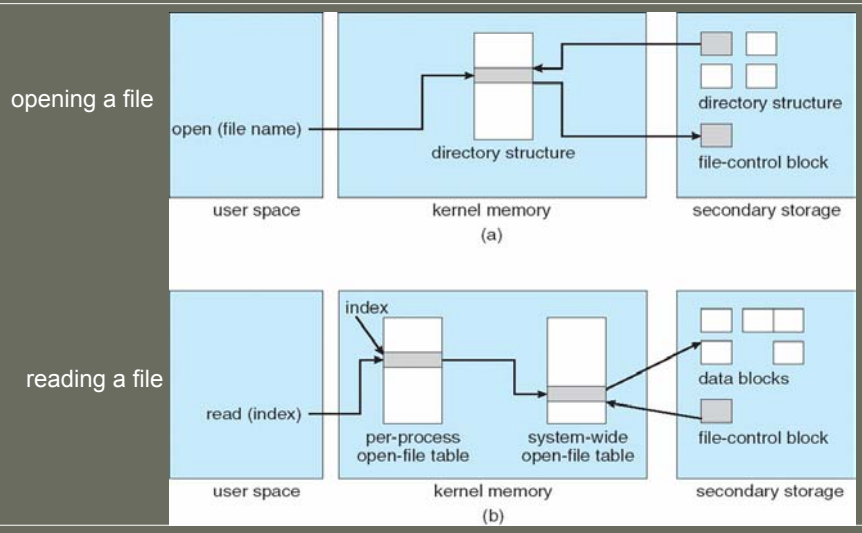
File Types

- Various things can be considered as files:
  - Regular files
    - The ascii files (text files) we use, binary files, .doc files, .pdf files, executable files, etc.
    - Some programs can look to them and understand what they are. They store content
  - Directories
    - A file can store directory information. Hence directories can be considered as special files.
    - we will have a file control block for such a file as well.
  - Device files
    - We can refer to devices in the system with files.
    - Device file “/dev/sda”5 may refer to a hard disk partition.
    - “/dev/fd0” may refer to floppy disk. “/dev/cdrom0” may refer to CDROM.
  - …

In Memory File System Structures

- The following figure illustrates the necessary file system structures provided by the operating systems.

In-Memory File System Structures



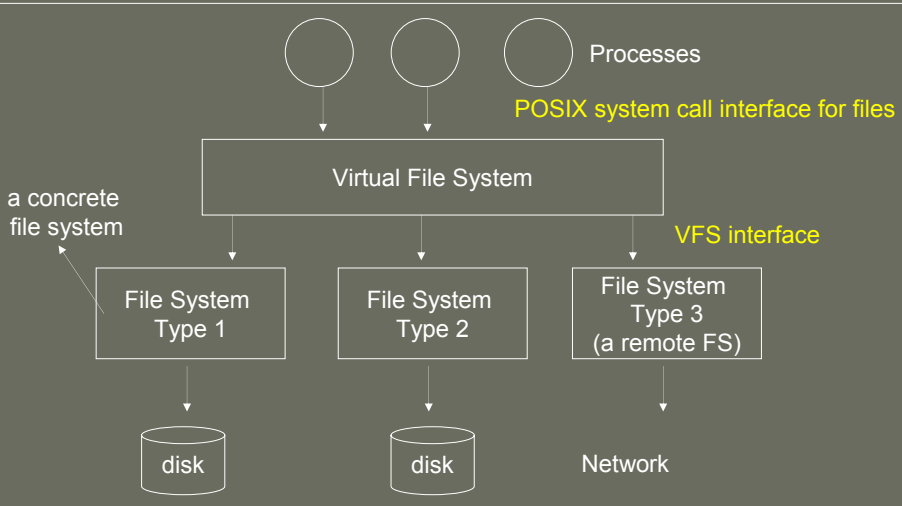
Virtual File System

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.
  - This can be a POSIX system call interface

Virtual File System

- VFS has also an interface to file systems (concrete file systems)
  - This is called VFS interface
- A concrete file system should provide functions developed according to the to the VFS interface (i.e. it should support functions defined in the VFS interface so that VFS layer can call those functions)
- VFS implements the common file system operations that are independent of any specific/concrete file system

Virtual File System



Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute

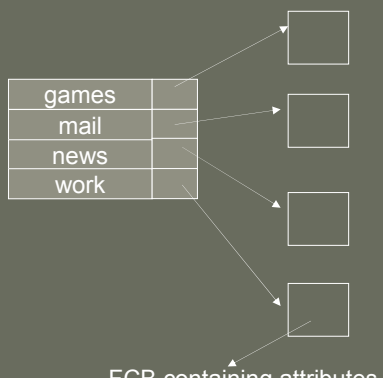
- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - **collisions** – situations where two file names hash to the same location
  - fixed size

Directory Implementation: directory entries

games	attributes
mail	attributes
news	attributes
work	attributes

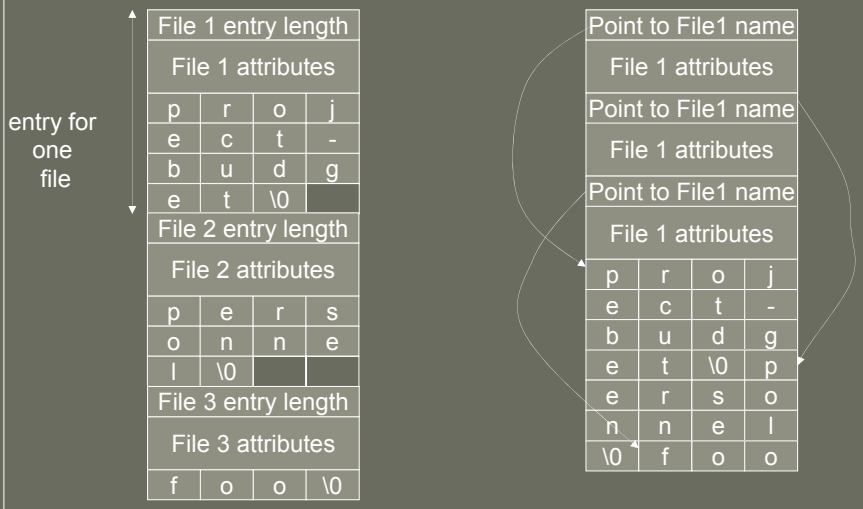
a directory with fixed sized entries

attributes include location info for data blocks of the file



Using fixed sized names

Directory Implementation: handling long filenames

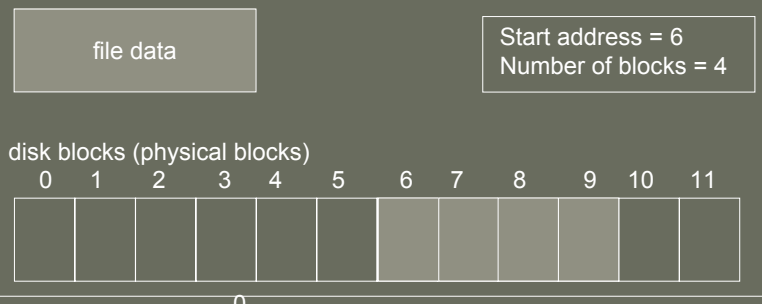


Allocation Methods

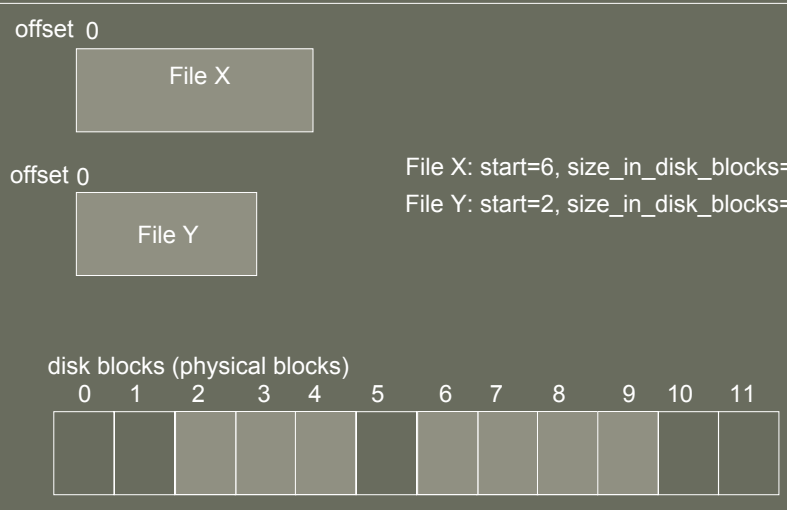
- An allocation method refers to how disk blocks are allocated for files:
- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required to find out the disk data blocks of file
- Random access is fast
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow



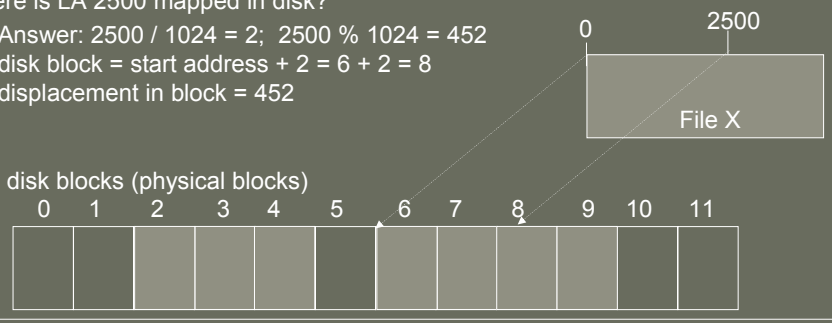
Example



Example

- Assume block size = 1024 bytes
- Which disk block contains the byte 0 of file X (LA = 0)? What is the displacement inside that block?
  - Answer : disk block = 6, displacement (disk block offset) = 0
- Which disk block contains the byte at LA (at file offset) 2500? In other words, where is LA 2500 mapped in disk?

Answer: 2500 / 1024 = 2; 2500 % 1024 = 452
disk block = start address + 2 = 6 + 2 = 8
displacement in block = 452



Contiguous Allocation

LA: logical address into a file: file offset (i.e. address of a byte in file)
(first byte has address 0)

- Mapping from logical (file) address to physical (disk) address

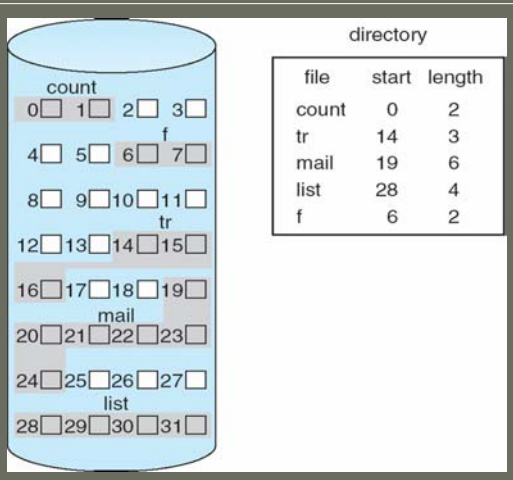
Q = LA div DiskBlockSize

LA/DiskBlockSize

R = LA mod DiskBlockSize

Disk Block to be accessed = Q + starting disk block number (address)
Displacement into disk block = R

Contiguous Allocation of Disk Space



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

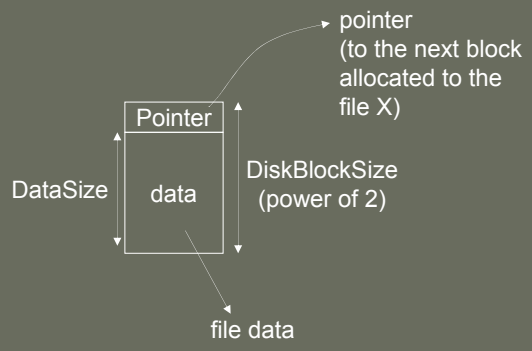
Extent-Based Systems

- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An extent is a contiguous blocks of disk
  - Extents are allocated for file allocation
  - A file consists of one or more extents

Linked Allocation

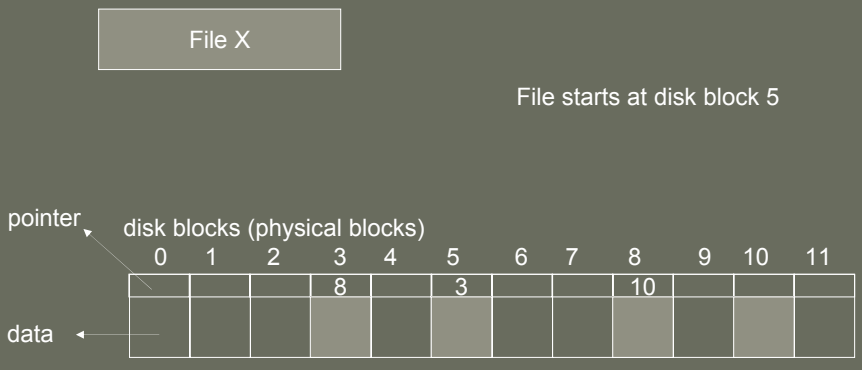
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.

block structure



File data size in a disk block is no longer a power of 2

Linked Allocation



Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access (random access is not easy)

- **Mapping Algorithm**

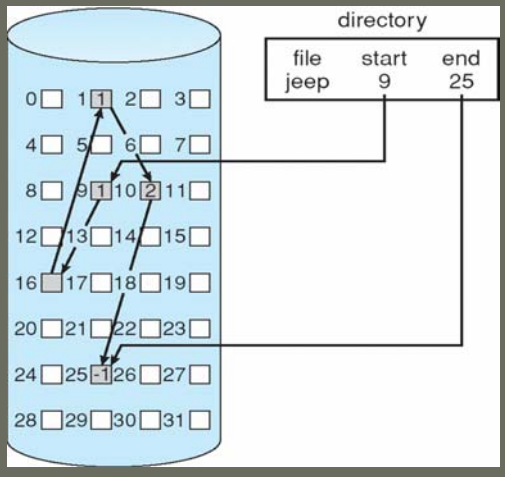
Logical Address (LA) / (BlockSize-PointerSize)

Q (integer division result: quotient)

R (remainder)

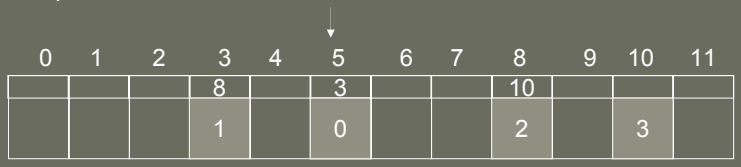
Block to be accessed = the Qth disk block in the linked chain of disk blocks representing the file.
Displacement into disk block = R + PointerSize

Linked Allocation



Linked Allocation: Example

- Assume block size = 1024 bytes
- Pointer size if 4 bytes
- Assume we have a file that is 4000 bytes.
- File data is place as below to the disk blocks; file start at disk block 5



Find out the disk location corresponding to file offset (LA) 2900?

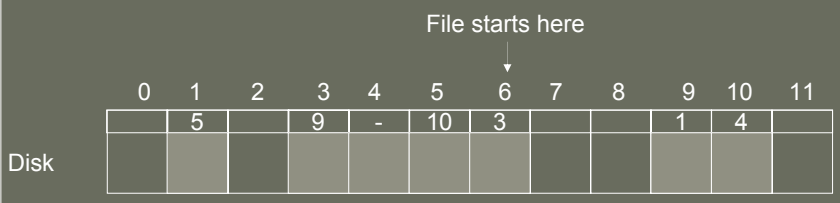
2900 / (1024-4) = 2
2900 % 1020 = 860

Go to the 2nd block in the chain

Second block in chain is disk block 8
Displacement is 860+4 = 864

Linked Allocation: Another Example

We have a file that is 3000 bytes long.
Disk block size = 512 bytes; pointer size = 4 bytes.
We want to access bytes 1000 through 2500 of the file.
Which disk blocks should be retrieved?



Relative Blocks to be accessed 1, 2, 3, 4

Answer: Disk Blocks 3, 9, 1, 5

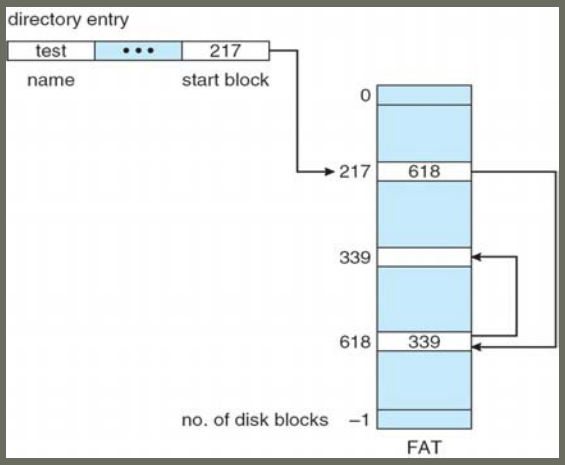
File Allocation Table

File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.

Pointers are kept in a table (FAT)

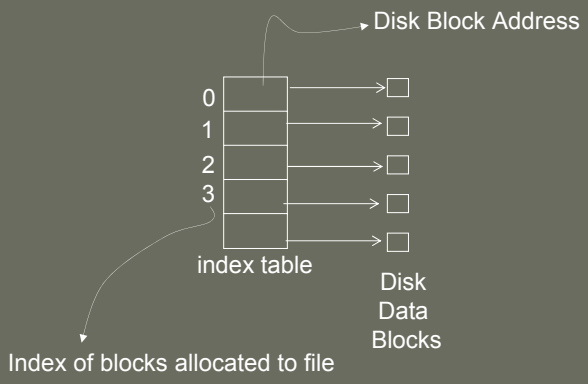
Data Block does not hold a pointer; hence data size is a power of 2.

File-Allocation Table

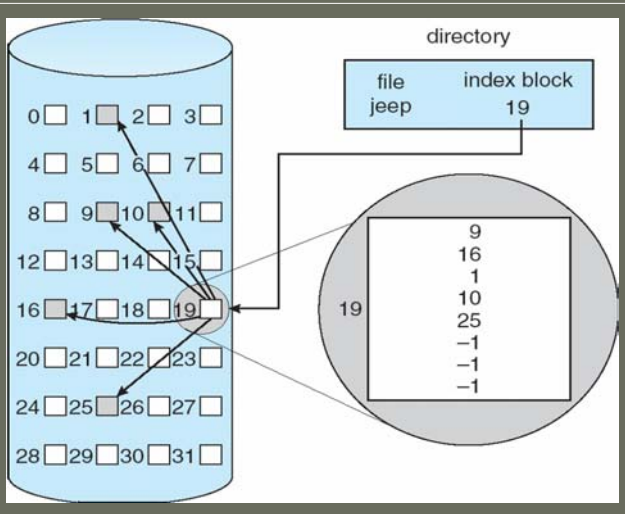


Indexed Allocation

- Brings all pointers together into the index block
- Logical view



Example of Indexed Allocation

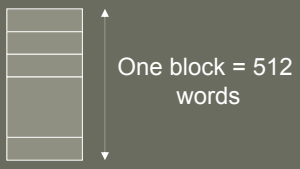


Indexed Allocation (Cont.)

- Need index table
- Random access can be fast
- No external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K (512 x 512) words and block size of 512 words. We need only 1 block for index table

Mapping Algorithm

Q = displacement into index table
R = displacement into block



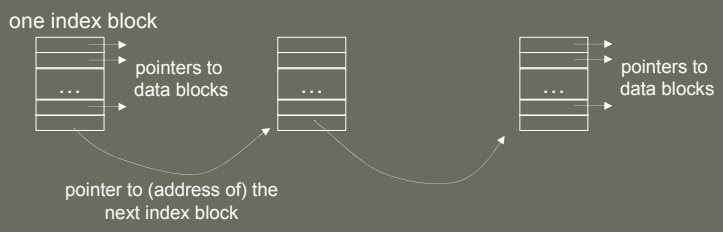
pointer size is 1 word

For larger files, we need other index blocks

Indexed Allocation (Cont.)

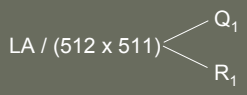
- If index table can not fit into a single block, we can use multiple index blocks and chain them together.

  Linked scheme – Link blocks of index table (no limit on file size)



Indexed Allocation – Mapping (Cont.)

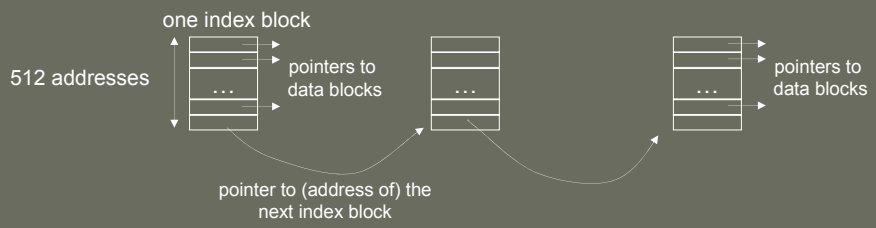
- Mapping from logical to physical in a file of unbounded length (assuming block size is 512 words and 1 pointer occupies 1 word)



Q_1 = block of index table
R_1 is used as follows:

Q_2 = displacement into block of index table
R_2 displacement into block of file:

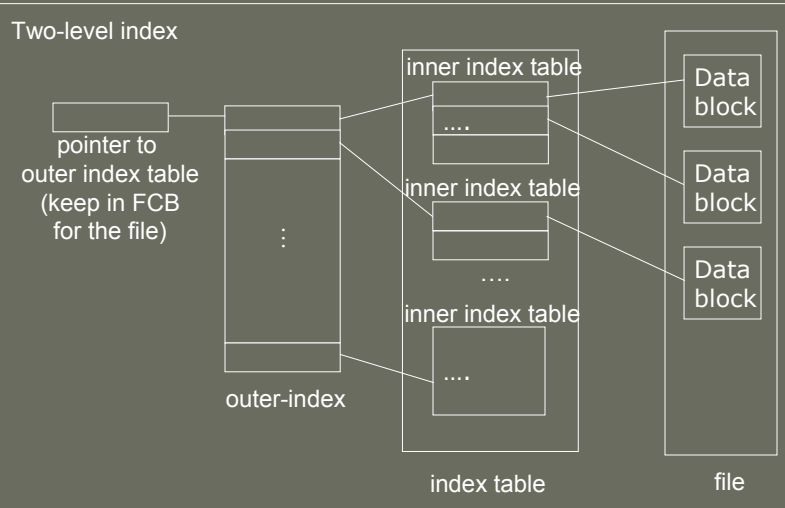
Indexed Allocation – Mapping (Cont.)



In an index block, 511 addresses are for data blocks.
Each data block is 512 words.

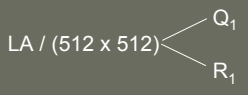
Hence, an index block can be used to map (511x512) words of a file

Indexed Allocation – Mapping (Cont.)



Indexed Allocation – Mapping (Cont.)

- Two-level index (maximum file size is 512^3)

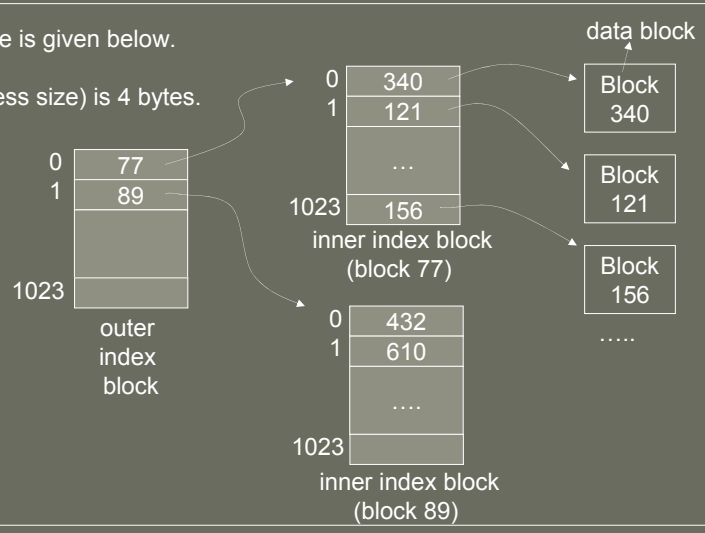


Q_1 = displacement into outer-index
R_1 is used as follows:

Q_2 = displacement into block of index table
R_2 displacement into block of file:

Example

Index table for a file is given below.
Block size is 4KB.
Disk pointer (address size) is 4 bytes.



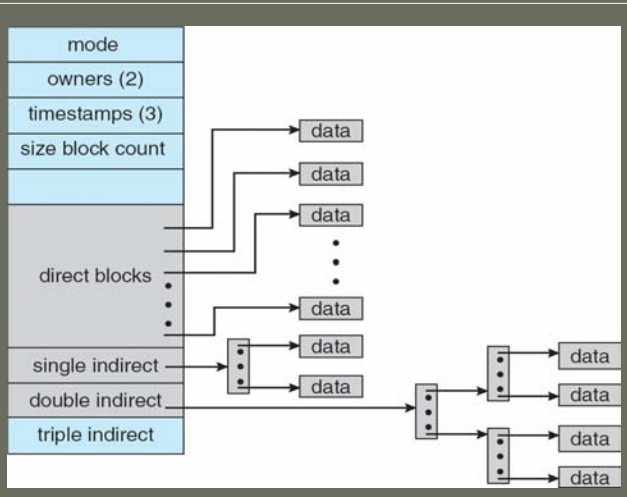
Example

- Where is file offset 5000?
  - 5000 / (1024 x 4096) = 0
  - 5000 % (1024 x 4096) = 5000
  - 5000 / 4096 = 0
  - 5000 % 4096 = 904
  - So it is on disk block 304 (follow outer table entry 0 and then inner table entry 0) and in that block displacement is 904.

Example

- Where is file offset 4198620?
  - 4198620 / (1024 x 4096) = 1.
  - Go to index 1 of outer table. That gives inner index table address: 89; Go to that inner index table (block).
  - 4198620 % (1024 x 4096) = 4316.
  - 4316 / 4096 = 1
  - Go to index 1 in the inner table. There is the data block address: 610.
  - Get that data block.
  - 4316 % 4096 = 220. displacement is 220

Combined Scheme: UNIX UFS (4K bytes per block)



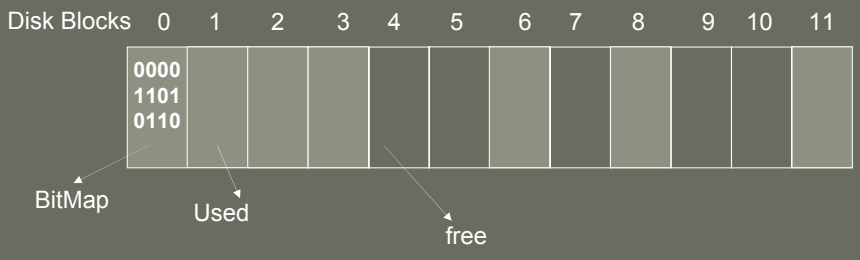
Free Space Management

- How can we keep track of free blocks of the disk?
  - Which blocks are free?
- We need this info when we want to allocate a new block to a file.
  - Allocate a block that is free.
- There are several methods to keep track of free blocks:
  - Bit vector (bitmap) method
  - Linked list method
  - Grouping
  - Counting

Free-Space Management: Bit Vector (Bit map)

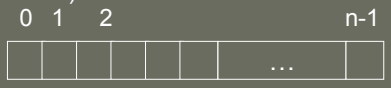
- We have a bit vector (bitmap) where we have one bit per block indicating if the block is used or free.
- If the block is free, the corresponding bit can be 1, else it can be 0 (or vice versa).

Example:



Free-Space Management: Bit Vector (Bit map)

- Bit vector (*n* blocks in disk)



bit[*i*] = { 0 ⇒ block[*i*] used; 1 ⇒ block[*i*] free } (or vice versa)

Finding a free block (i.e. its number)

Start searching from the beginning of the bitmap:
Search for the first 1

(number of 0-value words) * (number of bits per word)
+ offset of first 1-valued-bit

0000000000000000
0000000000000000
0000000000000000
0000000010000000
0000000000000000
0000000000011000
0001100010000000
0000000011110000

3x16+8 = 56

Free-Space Management: Bit Vector (Bit map)

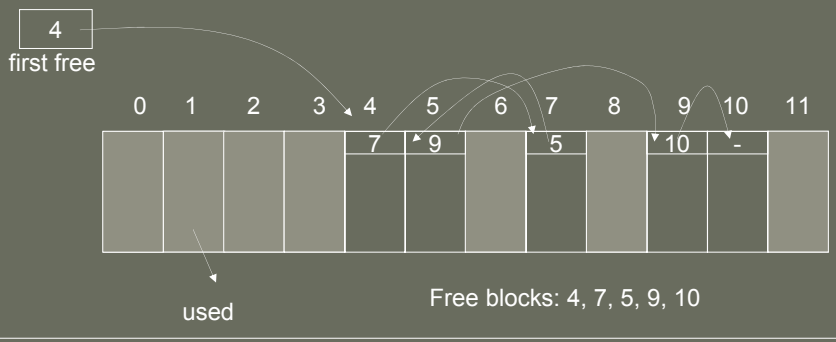
- Bit map requires extra space
  - Example:
    block size = 2^{12} bytes
    disk size = 2^{30} bytes (1 gigabyte)
    $n = 2^{30}/2^{12} = 2^{18}$ blocks exist on disk;

    Hence we need 2^{18} bits in the bitmap.
    That makes 2^{18} / 8 / 1024 = 32 Kbytes.
    32 KB space required to store the bitmap

- Easy to get contiguous files
- Blocks of a file can be kept close to each other.

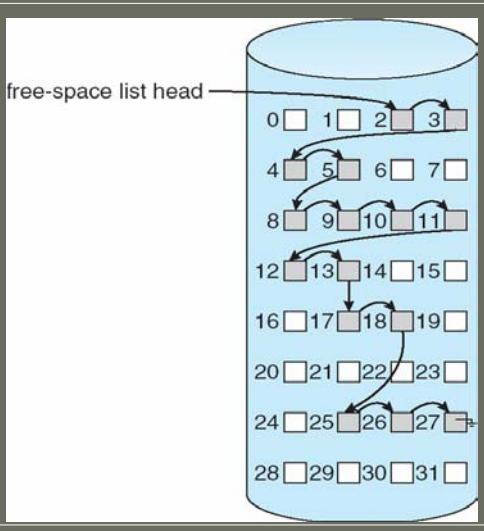
Free-Space Management: Linked List

- Each free block has pointer to the next free block
- We keep a pointer to the first free block somewhere (like superblock)
- Features:

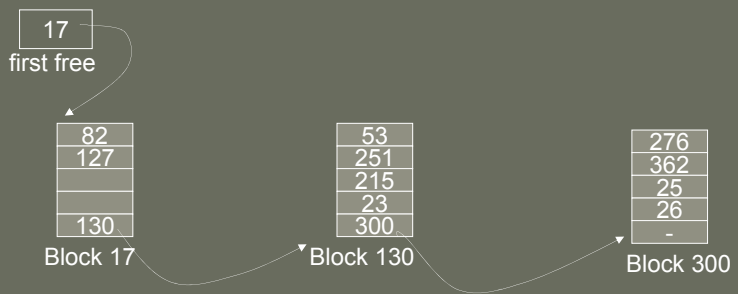


Free-Space Management: Linked List

Linked list (free list) features:
-Cannot get contiguous space easily
- No waste of space



Free-Space Management: Grouping

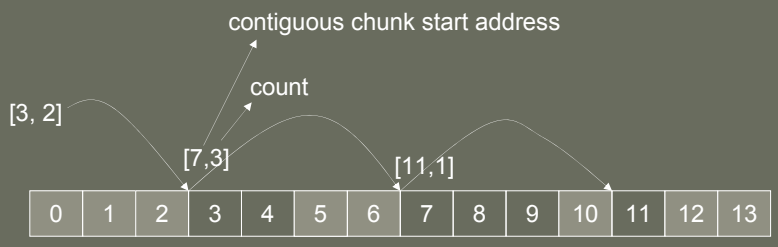


Free blocks are:
82 127 53 251 215 23 276 361 25 26

a block containing free block pointers will get free when those blocks are used.

Free-Space Management: Counting

- Besides the free block pointer, keep a counter saying how many block are free contiguously after that free block



Free-Space Management (Cont.)

- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ
    - Cannot allow for block[*i*] to have a situation where bit[*i*] = 0 (allocated) in memory and bit[*i*] = 1 (free) on disk
  - Solution:
    - Set bit[*i*] = 0 in disk
    - Allocate block[*i*]
    - Set bit[*i*] = 0 in memory

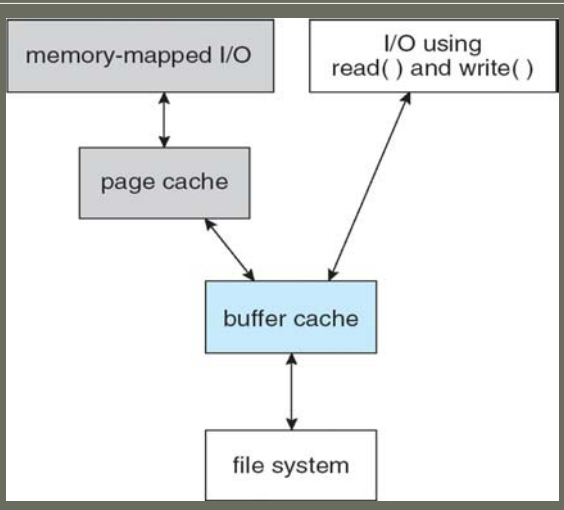
Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

Page Cache

- A page cache caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

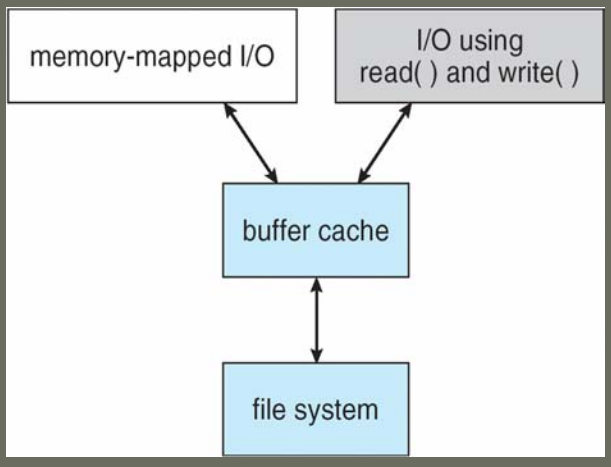
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A unified buffer cache uses the same cache to cache both memory-mapped pages and ordinary file system I/O blocks

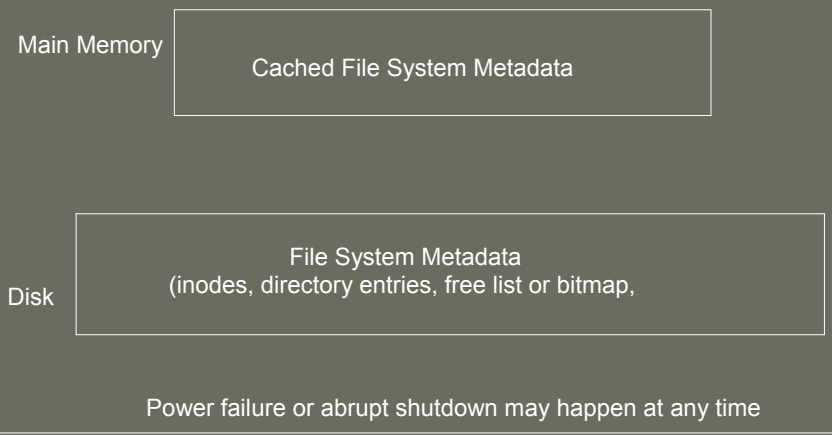
I/O Using a Unified Buffer Cache



Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - is invoked after a power failure
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
  - Recover lost file or disk by restoring data from backup
    - For example after a crash

Journaling File Systems



Power failure or abrupt shutdown may happen at any time

Journaling File Systems

- Example for a modification we can perform on the file system
- We will remove a file; following operations (updates have to be made)
  - 1) Directory entry should be removed (or marked unused)
    - Update directory entry on disk
  - 2) Inode must be marked as free (or removed)
    - Update inode on disk
  - 3) Blocks pointed by inode must be de-allocated
    - Added to the free list or bitmap
    - Update bitmap on disk (or free list).
  - While doing these sequence of operations, a power failure may happen and leave the disk structures in an inconsistent state. We need to recover from this kind of failures.
- Solution: consider these operations as a transaction.

Journaling File Systems

- Journaling file systems record each update to the file system metadata as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet

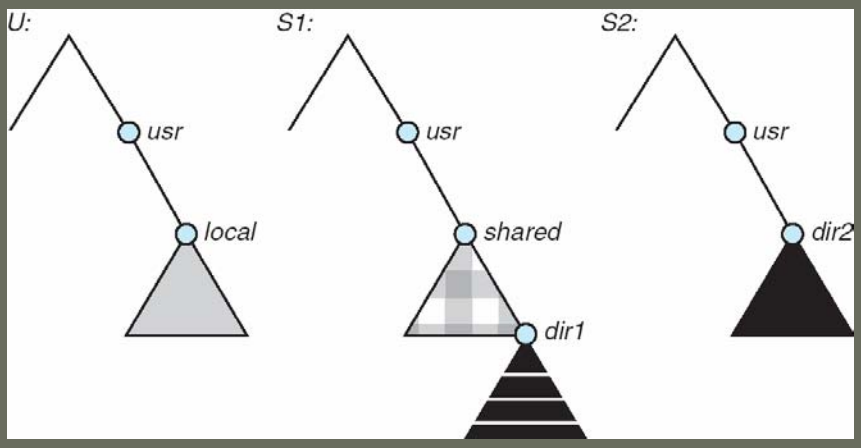
NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

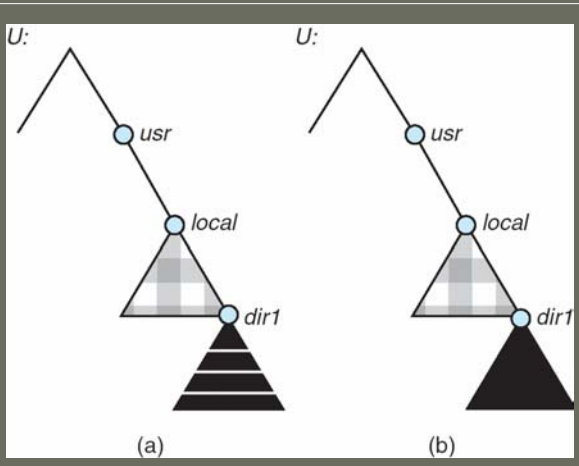
NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by
  - a mount mechanism, and
  - the actual remote-file-access services

Three Independent File Systems



Mounting in NFS



Mounts Cascading mounts

NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – [<a file-system identifier>, and <an inode number>] to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

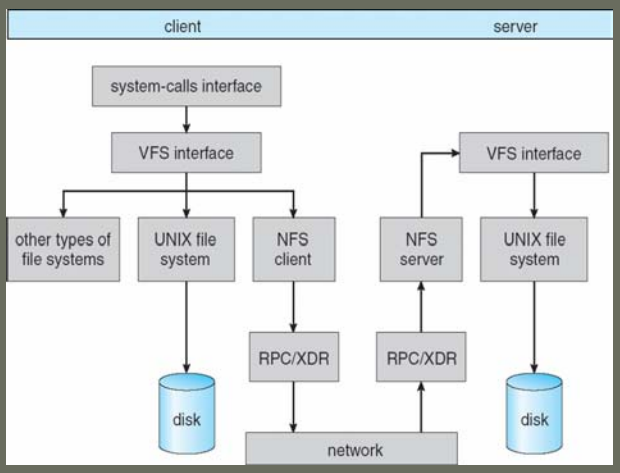
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is just coming available – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors)
- *Virtual File System* (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture



NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names

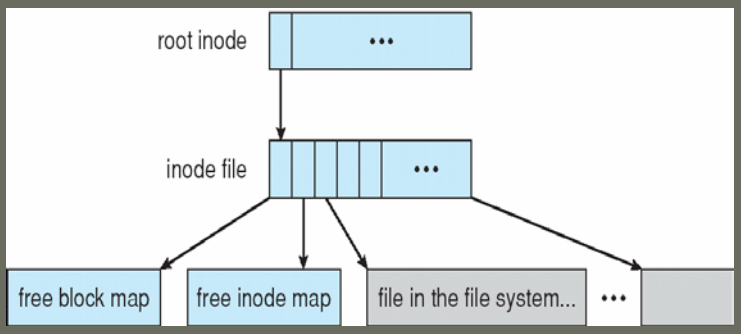
NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk

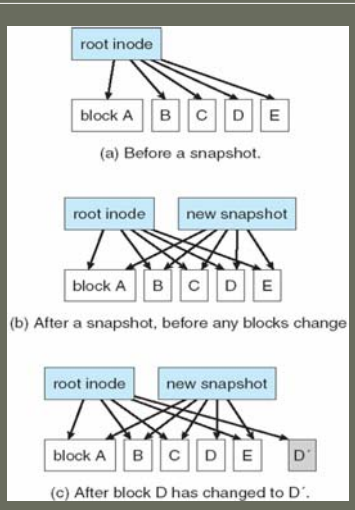
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications

The WAFL File Layout



Snapshots in WAFL

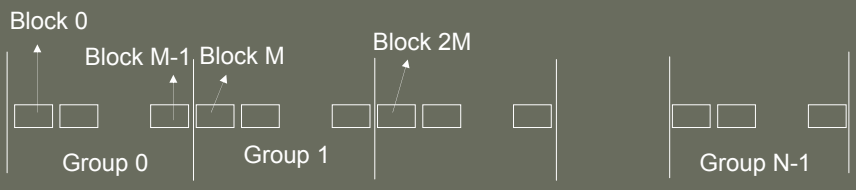


Example File System: Linux ext2/ext3 file system

- Linux ext2 file system is extended file system 2. Derived initially from Minix operating system. Linux is derived from Minix, an educational OS developed by A. Tanenbaum.
- The ext3 file system is fully compatible with ext3. The added new feature is Journaling. So it can recover better from failures.
- The disk data structures used by ext2 and ext3 is the same.

Partition Layout of ext3 (also ext2)

a disk partition before installing ext3 file system: just a sequence of blocks



Ext3 considers the partition to be divided into logical groups.
Each group has equal number of blocks; lets say M blocks/group

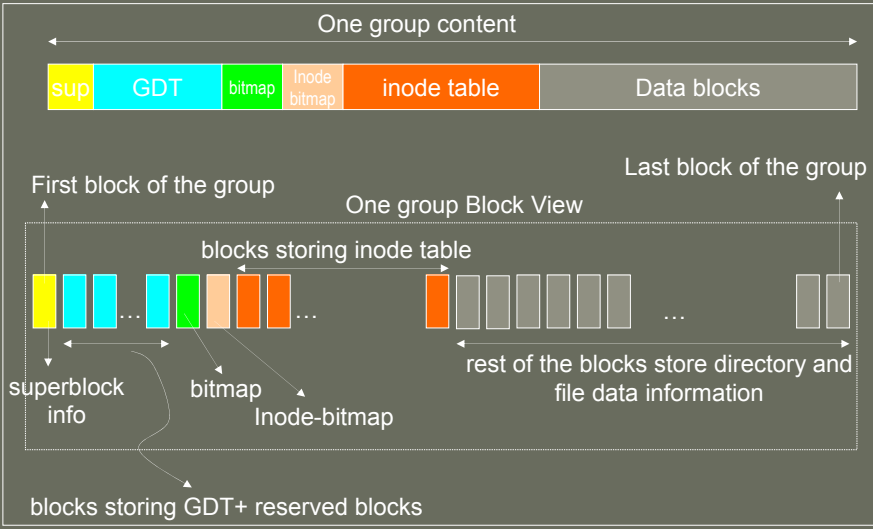
Ext3 file system

- What do we have in a group:
  - Assume block is size if 4 KB.
  - The first block of each group contains a superblock (i.e. superblock info) that is 1024 bytes long.
    - In group 0, superblock info starts at offset 1024 of the first block of the group (i.e. block 0 of the disk)
    - In all other groups, superblock info starts at offset 0 of the first block of the group.
  - Superblock keeps some general info about the filesystem
  - After the first block in a group, a few blocks keeps info about all groups. It means a few blocks store group descriptors table (GDT). Some of these block may be empty and reserved.

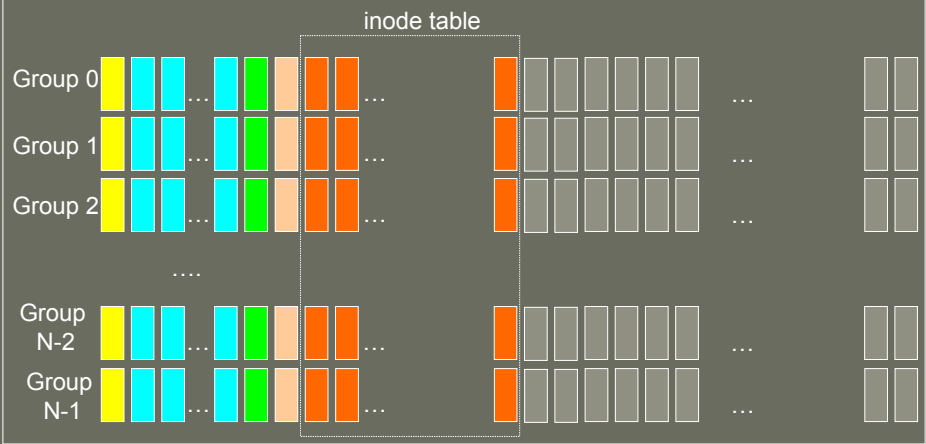
Ext3 file system

- What do we have in a group (continued)
  - After that comes bitmap that occupies one block. It is a bitmap for the group. Each group has it own bitmap.
  - Then comes an inode bitmap; showing which inodes are free.
  - After that comes the inodes (inode table). Each group stores a set of nodes. The number of inodes stored in a group is the same for all groups. So the inode table of the partition is divided into groups: each group stores a portion of the table.

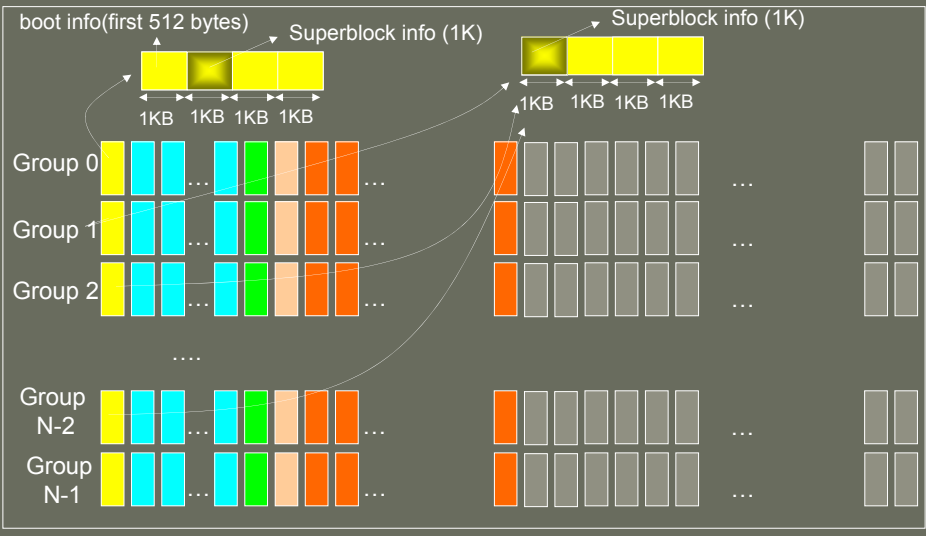
Ext3 file system: group structure



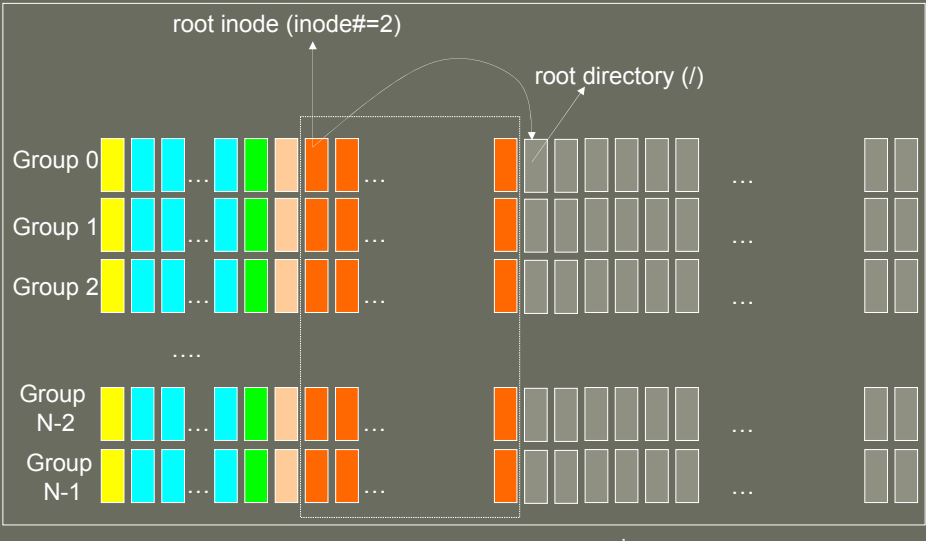
Ext3 file system: all groups and their structure



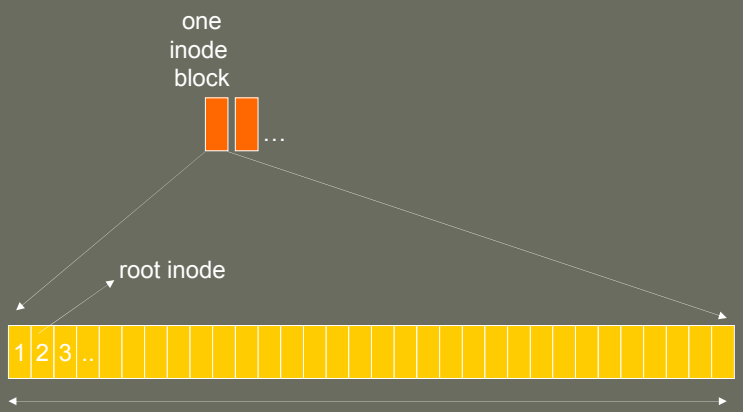
Ext3 file system: structure of the 1st block of each group



Ext3 file system: root inode and root directory



Ext3 file system: root inode



Ext3 file system: a real partition example

- We have a ~28 GB harddisk partition

number_of_groups =
block count / blocks_per_group
= 7323624 / 32768
= 223.49 => 224 groups
Groups from 0 to 223

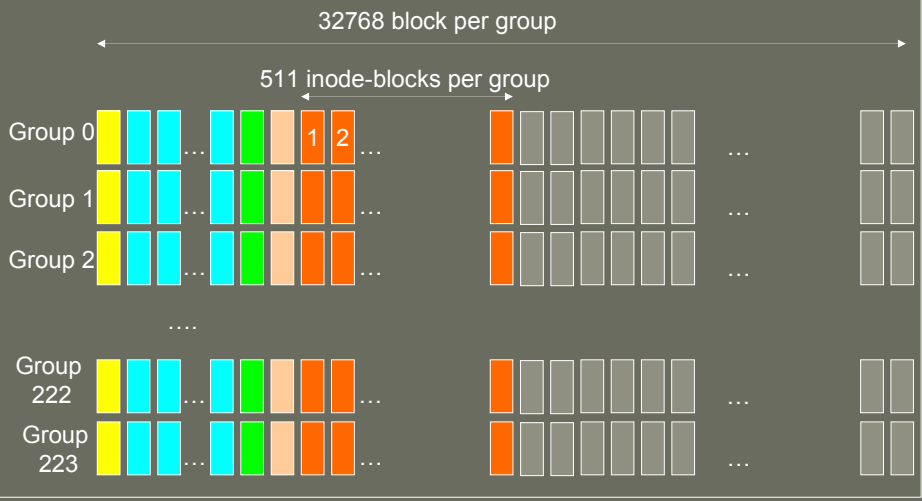
Inode size = 128 KB
Each block can contain 32 indodes
(4 KB / 128 bytes = 32)

There are 16352 inodes per group
16352/32 = 511 blocks required
to keep that many inodes in a group

superblock info

```
…
Filesystem OS type:       Linux
Inode count:              3662848
Block count:              7323624
Reserved block count:     366181
Free blocks:              4903592
Free inodes:              3288736
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      1024
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         16352
Inode blocks per group:   511
….
```

Ext3 file system: a real partition example



Ext3 file system: superblock structure

/usr/include/linux/ext3_fs.h

```
/*
 * Structure of the super block
 */
struct ext3_super_block {
/*00*/  __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count;         /* Blocks count */
        __le32  s_r_blocks_count;       /* Reserved blocks count */
        __le32  s_free_blocks_count;    /* Free blocks count */
/*10*/  __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        __le32  s_log_frag_size;        /* Fragment size */
/*20*/  __le32  s_blocks_per_group;     /* # Blocks per group */
        __le32  s_frags_per_group;      /* # Fragments per group */
        __le32  s_inodes_per_group;     /* # Inodes per group */
        __le32  s_mtime;                /* Mount time */
…
…
}
```

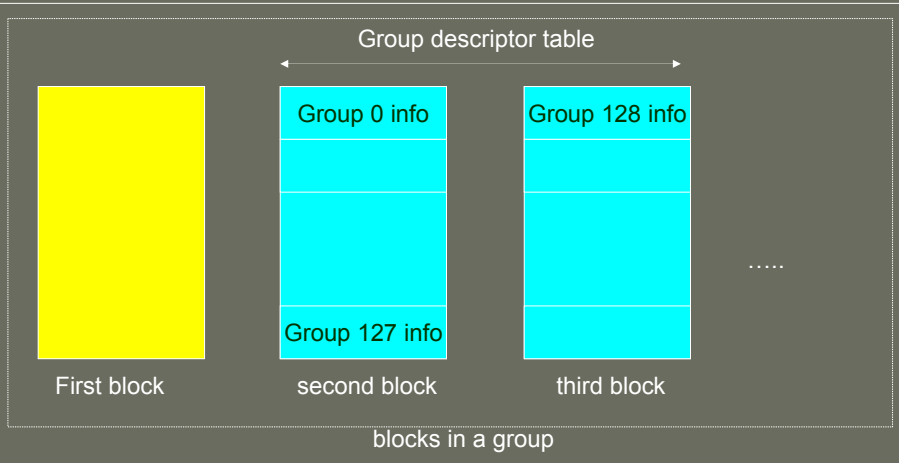
Ext3 file system group descriptors

- The number of blocks allocated for GDT table and reserved blocks may no be the name for each group. Therefore, the group decscriptor for a group tells where the inode bitmap and inode table starts.

```
struct ext3_group_desc
{
        __le32  bg_block_bitmap;        /* Blocks bitmap block */
        __le32  bg_inode_bitmap;        /* Inodes bitmap block */
        __le32  bg_inode_table;         /* Inodes table block */
        __le16  bg_free_blocks_count;   /* Free blocks count */
        __le16  bg_free_inodes_count;   /* Free inodes count */
        __le16  bg_used_dirs_count;     /* Directories count */
        __u16   bg_pad;
        __le32  bg_reserved[3];
};
```

Gives info about a group
Size of group descriptor is 32 bytes

Ext3 file system group descriptors



inodes

- Each inode keeps info about a file or directory
- Inode 2 is the inode for the root directory
- Inode numbers start with 1.

- Given inode number, it is east to compute on which group it is located.

inode structure

```
struct ext3_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
…
…
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
…..
        __le32  i_block[EXT3_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
        __le32  i_file_acl;     /* File ACL */
        __le32  i_dir_acl;      /* Directory ACL */
        __le32  i_faddr;        /* Fragment address */
…..
}
```

Directory entries

- A directory is a file that can occupies one or mode blocks.
- For example, root directory occupies one block.
- Directory is a sequence of entries.
- There is one entry per file
  - The entry points to the inode of the file. Namely it stores the inode number.
  - From the inode number (which is an index to the inode table), it is easy to compute the group# and index into the inode table in that group. The group descriptor also tells where the inode table in that group starts (disk blocks address). In this way we can reach to the disk block containing the inode.
- When we have the inode of a file, we can get further information about the file, like it data block addresses, attributed, etc.

Directory entry structure

/usr/iinclude/linux/ext3_fs.h

```
struct ext3_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT3_NAME_LEN];    /* File name */
};
```

Directory entry structure: file types

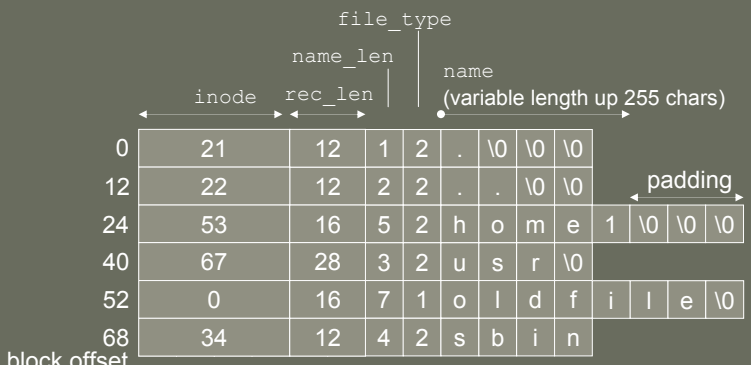
```
#define EXT3_FT_REG_FILE        1  /* regular file */
#define EXT3_FT_DIR             2  /* directory */
#define EXT3_FT_CHRDEV          3  /* char device file */
#define EXT3_FT_BLKDEV          4  /* block device file */
#define EXT3_FT_FIFO            5  /* fifo file */
#define EXT3_FT_SOCK            6  /* socket */
#define EXT3_FT_SYMLINK         7  /* symbolic link */
```

some file types have nothing to do with disk. They correspond to some other objects, like network connections, IPC objects, hardware devices, etc.

Example directory content

```
root directory (/) content
type=2 inode=2          name = .
type=2 inode=2          name = ..
type=2 inode=11         name = lost+found
type=2 inode=915713     name = etc
type=2 inode=1945889    name = proc
type=2 inode=2959713    name = sys
type=2 inode=2534561    name = dev
type=2 inode=1373569    name = var
type=2 inode=3008769    name = usr
type=2 inode=1586145    name = opt
type=2 inode=3270401    name = bin
type=2 inode=1177345    name = boot
type=2 inode=3482977    name = home
type=2 inode=130817     name = lib
type=2 inode=3057825    name = media
type=2 inode=2665377    name = mnt
…
```

Example directory content



There are 6 entries in this directory. Each entry starts at an offset that is multiple of 4.

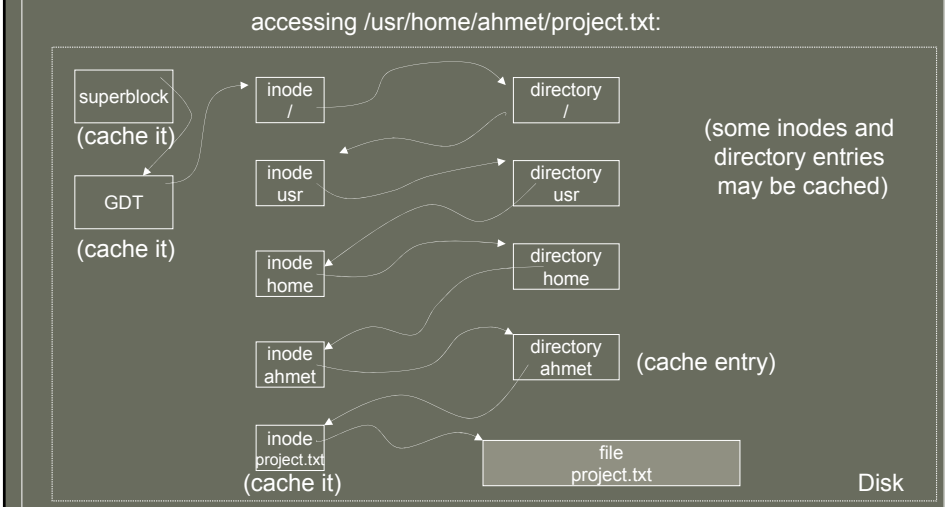
Searching for a file

- Given a filename, /usr/home/ahmet/project.txt, how can we locate it? For example while opening the file and then reading/writing the file.
- The filesystem may do the following steps:
  - Parse the pathname and divide into subdirectory names and file name:
    - /
    - usr
    - home
    - ahmet
    - project.txt
  - From root inode to go to the block that contains the root directory (hence go to the root directory).
  - Search there for an entry “usr”. That entry will tell us the inode number of subdirectory usr (which is also considered a file; therefore has an inode number),
  - Access the inode for “usr” (we can compute the block number containing the inode quite easily).

Searching for a file

- The inode for “usr” will tell us which block(s) contains the “usr” directory.
- Go to that (those blocks) and access the “usr” directory information (a sequence of directory entries).
- There search for entry “home”. That entry will give us inode info for “home”.
- Access inode for “home” and obtain the block numbers containing the “home” directory information.
- Go to those blocks (i.e. to “home” directory).
- Search home directory entries for “ahmet”. The corresponding entry will tell the inode number for directory “ahmet”.
- Access inode for “ahmet” and then access directory information.
- In directory info “ahmet”, search for entry “project.txt”. The entry will tell where the inode for “project.txt” is.
- Access the inode for “project.txt”. It will tell the data block number for the file. Access those block to read/write the file.

Searching for a file



End of lecture