File Systems

Anandha Gopalan
(with thanks to D. Rueckert, P. Pietzuch, A. Tannenbaum and R. Kolcun)
axgopala@imperial.ac.uk

Imperial College London

Long term non-volatile, online storage → e.g. programs, data, text, photos, music, ...

Sharing of information or software → e.g. editors, compilers, applications, ...

Concurrent access to shared data → airline reservation system, ...

Organisation and management of data → e.g. convenient use of directories, symbolic names, backups, snapshots, ...

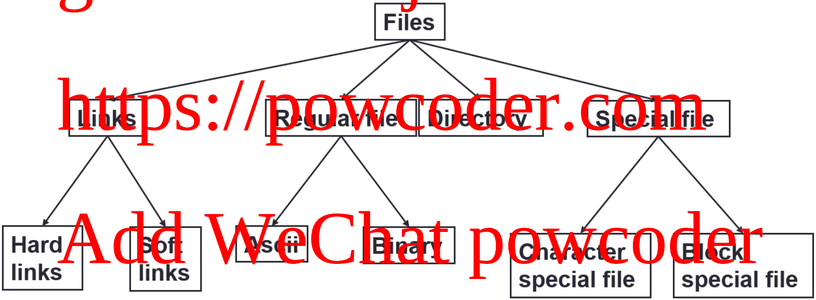
File: Named collection of data of arbitrary size

File Naming
Typical file extensions

File Type	Usual Extension	Function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, py, hs	source code in various languages
batch	bat, sh	commands to the command interpreter
...	...	...
...	...	...

File Types

What is a file?



Create	Create empty file. Allocate space and add to directory
Delete	Deallocate space. Invalidate or remove directory entry
Open	Search directory for file name. Check access validity and set pointers to file
Close	Remove pointers to file
Read	Access file, update current position pointers
Write	Access file, update pointers
Reposition/seek	Set current position in file to given value
Truncate	Erase contents but keep all other attributes
Rename	Change file name
Read attributes	e.g. creation date, size, archive flag, ...
Write attributes	e.g. protection, immutable flag, ...

System Call	Description
`open (file, how, ...)`	Open a file for reading/writing
`close (fd)`	Closing an open file
`read (fd, buf, nbytes)`	Read data from file to buffer
`write (fd, buf, nbytes)`	Write data from buffer to file
`lseek (fd, offset, ...)`	Move file pointer
`stat (name, &buf)`	Get file's meta-data
`fcntl (fd, cmd, ...)`	File locking and other operations

Logical name to physical disk address translation

- i.e. /homes/axgopala/.vimrc $\rightarrow$ disk 2, block 399

Management of disk space

- Allocation and deallocation

File locking for exclusive access

Performance optimisation

- Caching and buffering

Protection against system failure

- Back-up and restore

Security

- Protection against unauthorised access

File Attributes I

Basic information

file name	symbolic name; unique within directory
file type	text, binary, executable, directory, ...
file organisation	sequential, random, ...
file creator	program which created file

Address information

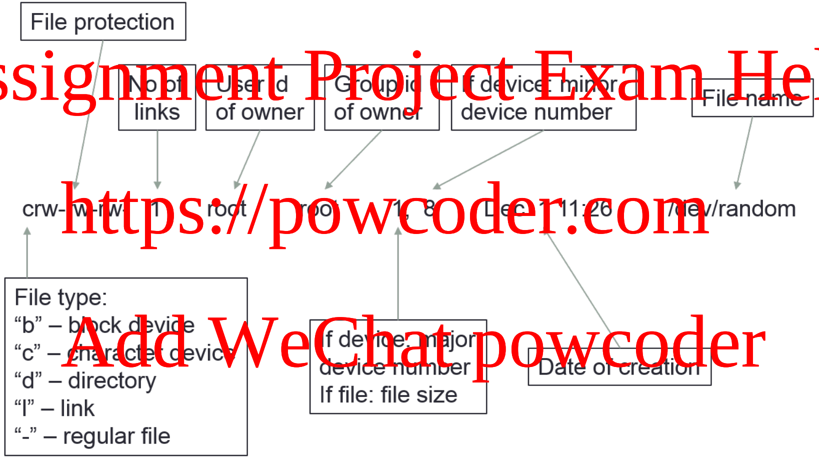
volume	disk drive, partition
start addresses	cyl, head, sect, LBA
size used	
size allocated	

Access control information

owner	person who controls file (often creator)
authentication	password
permitted actions	read, write, delete for owners/others

Usage information

creation timestamp	date and time
last modified	
last read	
access activity counts	


File protection
No of links
User id of owner
Group id of owner
If device: minor device number
File name
crw-rw-rw- 1 root root 1, 8 Dec 1 11:26 /dev/random
File type:
"b" – block device
"c" – character device
"d" – directory
"l" – link
"-" – regular file
If device: major device number
If file: file size
Date of creation

Unix/Linux stat System Call

File attributes can be accessed using system call `stat(2)` (`man 2 stat`)

- Return information about specified file in `struct stat`


```
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 */
  ...
  off_t     st_size;        /* total size, in bytes */
  struct timespec st_atim;  /* time of last access */
  struct timespec st_mtim;  /* time of last modification */
  struct timespec st_ctim;  /* time of last status change */
};
```

File size naturally variable

Space allocated in blocks (typically 512 – 8192 bytes)

Choosing block size

- Block size too large → wastes space for small files (remember memory management ☺!)
  - More memory needed for buffer space
- Block size too small → wastes space for large files
  - High overhead in terms of management data
  - High file transfer time: seek time greater than transfer time

Which allocation works the best?

Contiguous file allocation

Block chaining

File allocation table

Index blocks

Contiguous File Allocation I

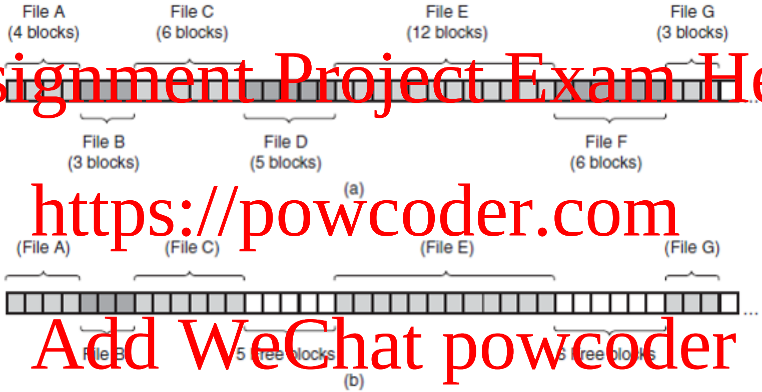
Place file data at contiguous addresses on storage device

Advantages

- Successive logical records typically physically adjacent

Disadvantages

- External fragmentation
- Poor performance if files grow and shrink over time
- File grows beyond size originally specified and no contiguous free blocks available
  - Must be transferred to new area of adequate size
  - Leads to additional I/O operations

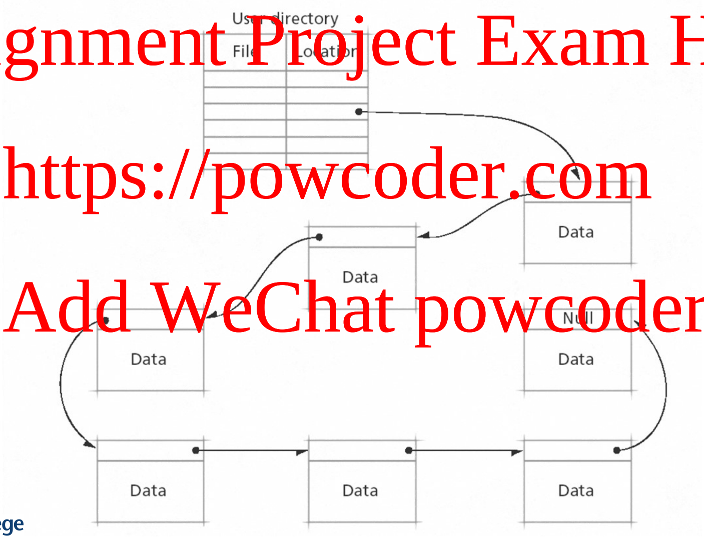


(a) Contiguous allocation of disk space for seven files

(b) The state of the disk after files D and F have been removed

Block Linkage (Chaining) I

Place file data by linking them together ⇒ insertion/deletion by modifying pointer in previous block



Need to search list to find data block

- Chain must be searched from beginning
- If blocks dispersed throughout disk, search process slow
  - Many seeks can occur
  - Block-to-block seeks occur

Wastes pointer space in each block

Store pointers to file blocks

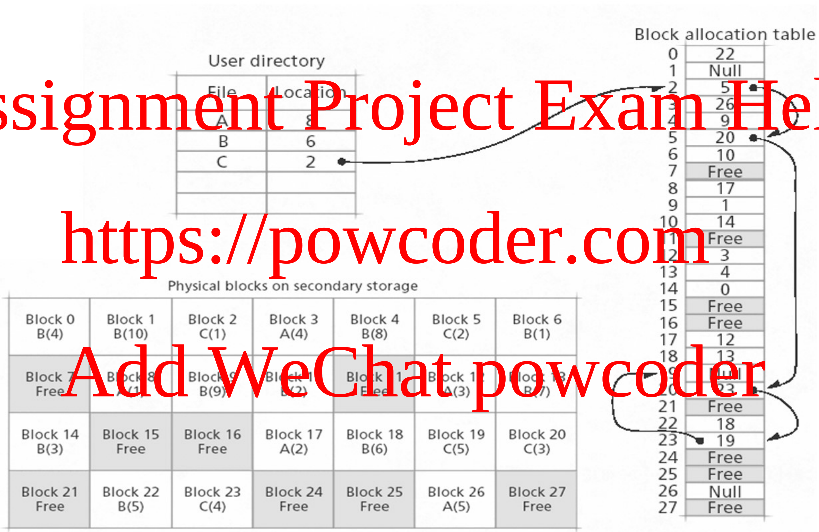
- Directory entries indicate first block of file
- Block number as index into block allocation table
  - Determines location of next block
  - If current block = last block, set table entry to null

File Allocation Table (e.g. MSDOS/Windows (FAT16/32)) → akin to Block Allocation Table

- Stored on disk but cached in memory for performance

Reduces number of lengthy seeks to access given record

- But files become fragmented → periodic defragmentation
- Table can get very large


User directory
File
Location
A
B 6
C 2
Block allocation table
0 22
1 Null
2 5
3 26
4 9
5 20
6 10
7 Free
8 17
9 1
10 14
Free
3
13 4
14 0
15 Free
16 Free
17 12
18 13
23
21 Free
22 18
23 19
24 Free
25 Free
26 Null
27 Free
Physical blocks on secondary storage
Block 0 B(4)
Block 1 B(10)
Block 2 C(1)
Block 3 A(4)
Block 4 B(8)
Block 5 C(2)
Block 6 B(1)
Block 14 B(3)
Block 15 Free
Block 16 Free
Block 17 A(2)
Block 18 B(6)
Block 19 C(5)
Block 20 C(3)
Block 21 Free
Block 22 B(5)
Block 23 C(4)
Block 24 Free
Block 25 Free
Block 26 A(5)
Block 27 Free

Block Linkage vs. FAT

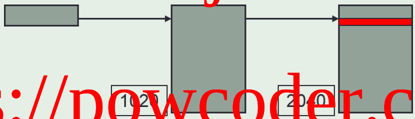
Consider a disk with a block size of 1024 bytes. Each disk address can be stored in 4 bytes. Block linkage is used for file storage, i.e. each block contains the address of the next block in the file.

1. How many block reads will be needed to access: the 1022^{nd} data byte and the 510100^{th} data byte?
   Hint: $500 \times 1020 = 510000$ and $498 \times 1024 = 509952$

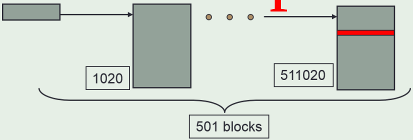
2. How does this change if a file allocation table (FAT) is used?

Answer: Block Linkage

There are 1020 data bytes per block. The 1022^{nd} byte is resident on the 2^{nd} disk block $\rightarrow$ 2 reads are required



The 510100^{th} data byte is resident in the 501^{st} disk block $\rightarrow$ 501 reads are required



Answer: FAT

There are 1024 data bytes per block. Each block of the FAT can represent $\frac{1024}{4} = 256$ data blocks

1. The 1020^{th} byte is on the 1^{st} block and requires 1 read for the FAT and 1 read for the data block, for a total of 2 reads

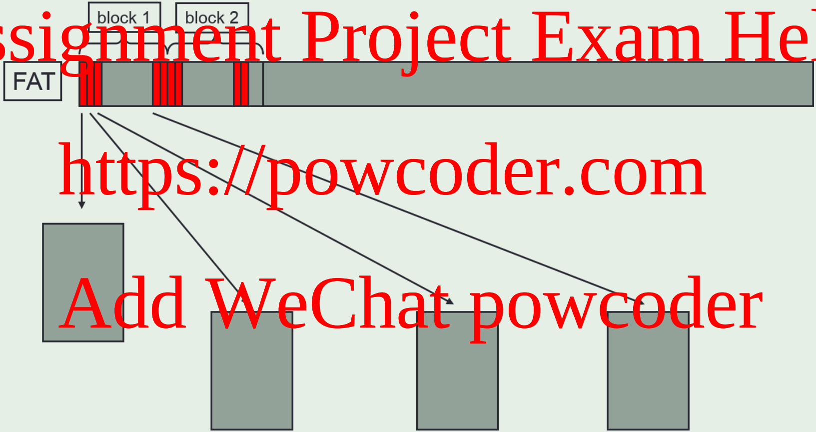
2. The 510100^{th} data byte is on the 499^{th} data block
   - At best, all of the first 499 blocks of the file can be represented in 2 FAT blocks
   - At worst, 499 reads could be performed for the FAT

Either case requires 1 extra read for the data. Hence,

- Best case requires 3 reads
- Worst case requires 500 reads

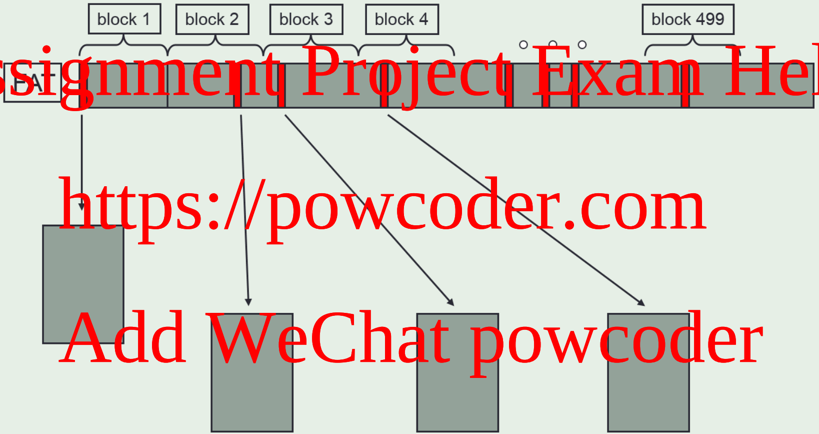
Example Problem

Answer: FAT



Example Problem

Answer: FAT



How can we improve?

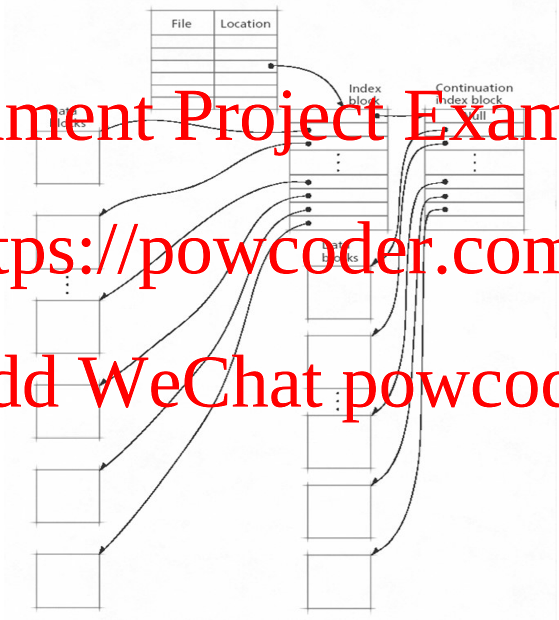
Index Blocks I

Each file has one (or more) index blocks

- Contain list of pointers that point to file data blocks
- File's directory entry points to its index block
- Chaining $\rightarrow$ may reserve last few entries in index block to store pointers to more index blocks

Advantages over simple linked-list implementations

- Searching may take place in index blocks themselves
- Place index blocks near corresponding data blocks $\rightarrow$ quick access to data
- Can cache index blocks in memory for faster access


File
Location
Index
block
Continuation
index block
Data
blocks
Data
blocks

Index blocks called inodes (index nodes) in UNIX/Linux

On file open, OS opens inode table → inode entry created in memory

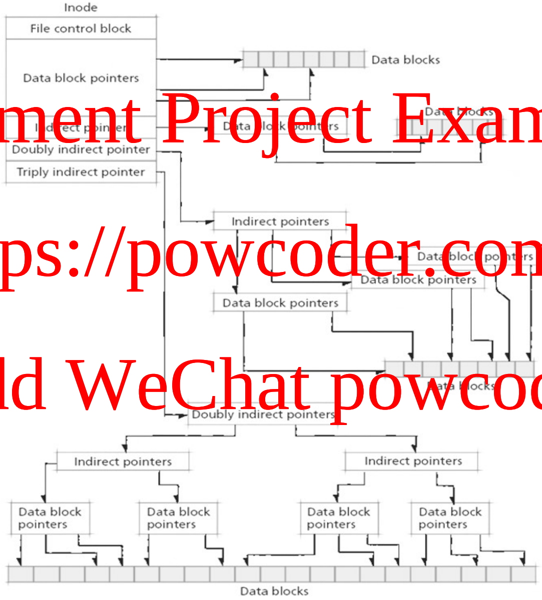
Structured as inode on disk, but includes:

1. Disk device number
2. Inode number (for re-write)
3. Num of processes with opened file
4. Major/minor device number

inode

Type and access control
Number of links
User ID
Group ID
Access time
Modification time
Inode change time
Direct pointer
Direct pointer
...
Direct pointer
Indirect pointer
Double indirect pointer
Triple indirect pointer

Inodes


Inode
File control block
Data block pointers
Doubly indirect pointer
Triply indirect pointer
Data blocks
Indirect pointers
Data block pointers
Data block pointers
Doubly indirect pointers
Indirect pointers
Indirect pointers
Data block pointers
Data block pointers
Data block pointers
Data block pointers
Data blocks

Inodes I

In a particular OS, an inode contains 6 direct pointers, 1 pointer to a (single) indirect block and 1 pointer to a doubly indirect block. Each of these pointers is 8 bytes long. Assume a disk block is 1024 bytes and that each indirect block fills a single block.

1. What is the maximum file size for this file system?
2. What is the maximum file size if the OS would use triply indirect pointers?

Example Problem

Answer: Inodes I

1. The maximum file size is:

$$
\begin{array}{ll}
6 \text{ x } 1024 & \text{(data directly indexed)} \\
+\ 128 \text{ x } 1024 & \text{(data referenced by single indirect)} \\
+\ 128^2 \text{ x } 1024 & \text{(data referenced by double indirect)} \\
= 16.13 \text{ MB} &
\end{array}
$$

2. The maximum file size is:

$$
\begin{array}{ll}
6 \text{ x } 1024 & \text{(data directly indexed)} \\
+\ 128 \text{ x } 1024 & \text{(data referenced by single indirect)} \\
+\ 128^2 \text{ x } 1024 & \text{(data referenced by double indirect)} \\
+\ 128^3 \text{ x } 1024 & \text{(data referenced by triple indirect)} \\
= 2.02 \text{ GB} &
\end{array}
$$

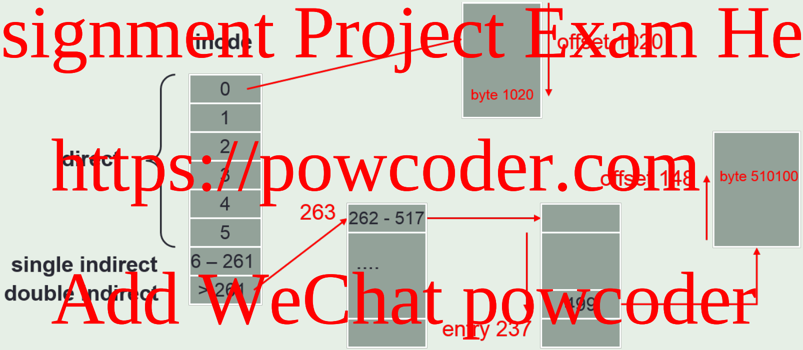
Inodes II

In a particular OS, an inode contains 6 direct pointers, 1 pointer to a (single) indirect block and 1 pointer to a doubly indirect block. Each of these pointers is 4 bytes long. Assume a disk block is 1024 bytes and that each indirect block fills a single disk block.

How many disk block reads will be needed to access:

1. the 1020^{th} data byte?
2. the 510100^{th} data byte?

Answer: Inodes II



1020 / 1024 = 1^{st} block

510100 / 1024 = 499^{th} block

	Block Chaining	**FAT**	**Inodes**
Byte 1020	2	2	2 (assuming inode not yet in memory)
Byte 510100	501	best case: 3 worst case: 500	4 (assuming inode not yet in memory)

How do we manage a storage device's free space?

Need quick access to free blocks for allocation

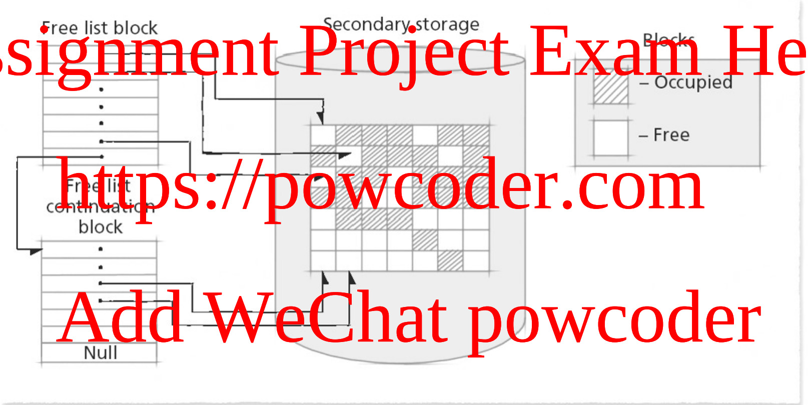
Use free list

- Linked list of blocks containing locations of free blocks
- Blocks are allocated from beginning of free list
- Newly-freed blocks appended to end of list

Low overhead to perform free list maintenance operations

Files likely to be allocated in noncontiguous blocks → increases file access time

Free List



Free list block
Secondary storage
Blocks
– Occupied
– Free
Free list continuation block
Null

Example Problem

Free List

Block size: 1 KB

Disk block number precision: 32-bit

Number of free blocks each block can hold: 255 (one block is required for pointer to the next block)

Hard drive size: 500 GB

Number of blocks: 488 million

Number of blocks required to store all addresses: 1.9 million $\left(\frac{488}{255}\right)$

Bitmap contains one bit (in memory) for each disk block

- Indicates whether block in use
- i^{th} bit corresponds to i^{th} block on disk

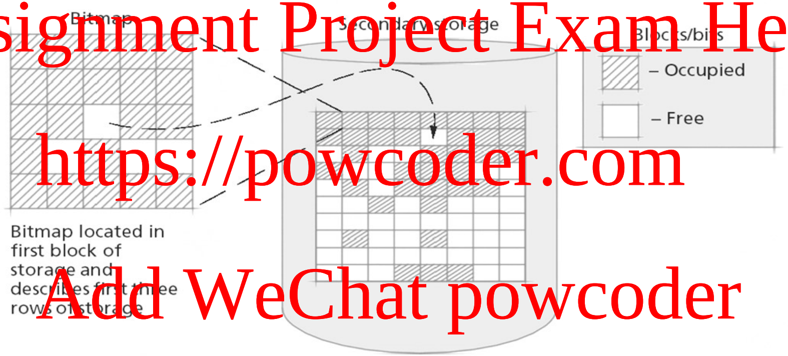
Advantage of bitmaps over free lists

- Can quickly determine available contiguous blocks at certain locations on secondary storage

Disadvantage

- May need to search entire bitmap to find free block, resulting in execution overhead

Bitmap II


Bitmap
Secondary storage
Blocks/bits
– Occupied
– Free
Bitmap located in first block of storage and describes first three rows of storage

Bitmap

Block size: 1 KB

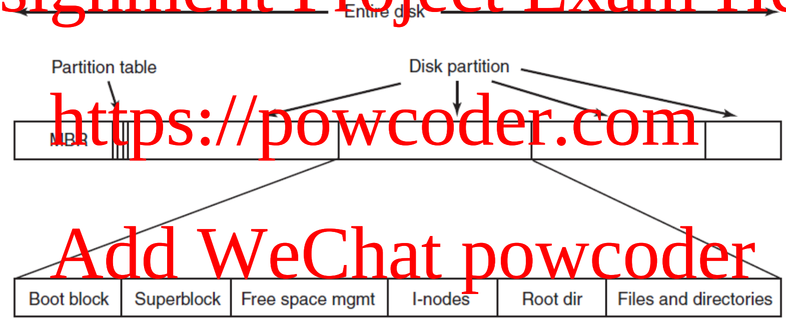
Hard drive size: 500 GB

Number of blocks: 488 million

Number of bits required: 488 million

Number of blocks required to store the bitmap: 60,000 $\left(\frac{488000000}{(1024x8)}\right)$

A possible file system layout



Fixed disk layout (with inodes)

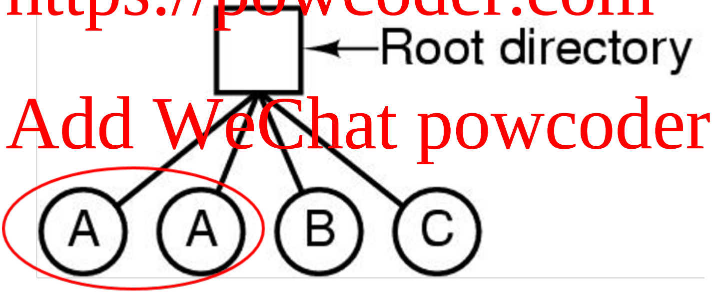
- boot block
- superblock
- free inode bitmap
- free block (zone) bitmap
- inodes + data

Superblock (contains crucial info about FS)

- no of inodes
- no of data blocks
- start of inode & free space bitmap
- first data block
- block size
- maximum file size, . . .

Directory

- Maps symbolic file names to logical disk locations (e.g. blah.txt → disk 0, block 2 (LBA))
  - Helps with file organisation
  - Ensures uniqueness of names



Single-Level File System

Single-level (or flat) file system

- Simplest file system organisation
- Stores all files using one directory
- No two files can have same name

FS often performs linear search of directory to locate file

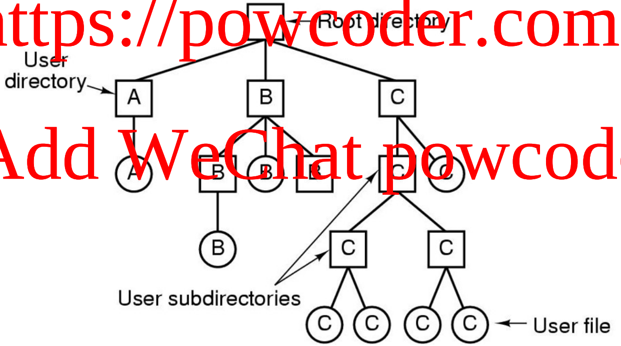
- Leads to poor performance

Little flexibility in terms of file organisation

MultiLevel (Tree) Directory Structure

Hierarchical file system

- UNIX, Linux, Windows, Mac, . . .
- **Root** indicates where on disk root directory begins
- **Root directory** points to various directories
  - Each of which contains entries for its files
  - File names need be unique only within given directory



Pathnames

- File names usually given as path from root directory to file

Absolute pathnames

- Unix/Linux: /homes/axgopala/foo
- Windows: \homes\axgopala\foo

Relative pathnames

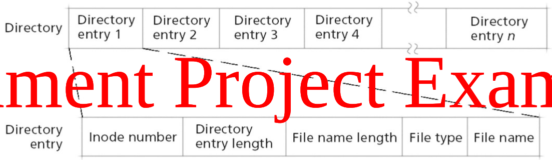
- Relative to working (or current) directory
- Can be changed using `cd` command
- Displayed with `pwd`
- Current directory: .
- Parent directory: ..

Directory Operations

open/close	Open or close a directory
search	Find file in directory system using pattern matching on string, wildcard characters
create/delete	Create or delete files/directories
link	Create link to file
unlink	Remove link for file
change directory	Opens new directory as current one
list	Lists or displays files in directory → implemented as multiple read entry operations
read attributes	Read attributes of file
write attributes	Change attributes of file, e.g. protection information or name
mount	Creates link in directory to directory in different file system, e.g. on another disk or remote server

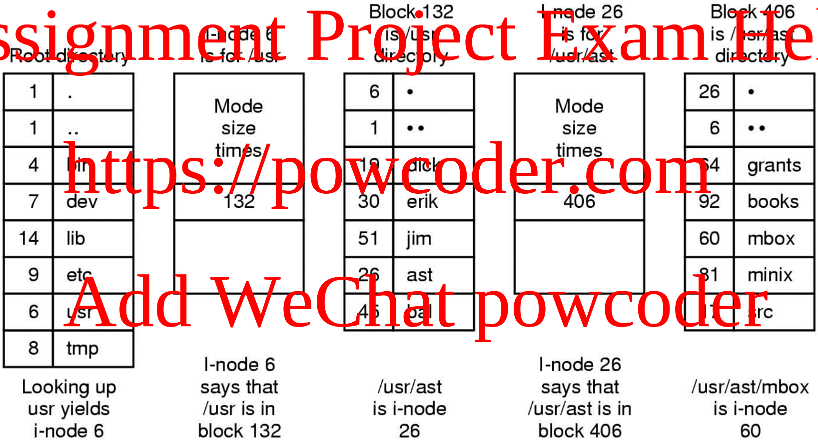
Unix/Linux: Directory System Calls

System Call	Description
`s = mkdir (path, mode)`	Create a new directory
`s = rmdir (path)`	Remove directory
`s = link (oldpath, newpath)`	Create a new (hard) link
`s = unlink (path)`	Unlink a path
`s = chdir (path)`	Change working directory
`dir = opendir (path)`	Open directory for reading
`s = closedir (dir)`	Close directory
`dirent = readdir (dir)`	Read one entry from directory
`rewinddir (dir)`	Rewind directory to re-read




```
struct dirent
{
  long d_ino;                 /* inode number */
  off_t d_off;                /* offset to next dirent */
  unsigned short d_reclen;    /* length of this d_name */
  unsigned char d_type;       /* file type; not supported */
                              /* by all file system types */
  char d_name [NAME_MAX+1];/* file name (null-terminated) */
};
```

Steps in looking up **/usr/ast/mbox**



Links

Link: Reference to directory/file in another part of FS

- Allows alternative names (and different locations in tree)

Hard link: Reference address of file

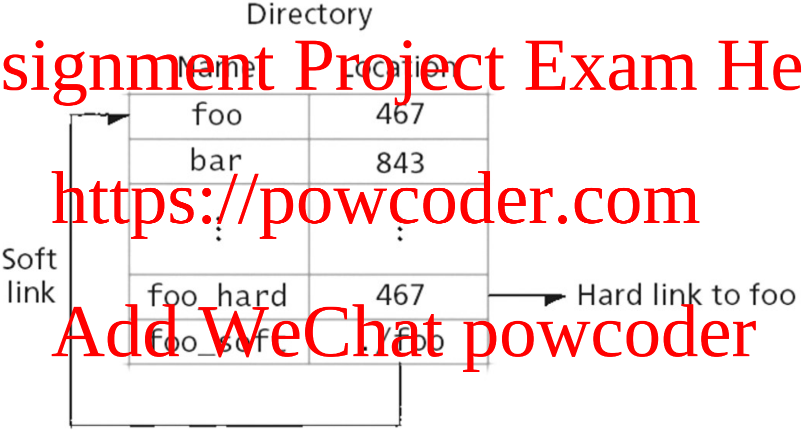
- Only supported for files in Unix

Symbolic (soft) link: Reference full pathname of file/dir

- Created as directory entry

Problems

- File deletion: search for links and remove them
  - Leave links and cause exception when used (symbolic links)
  - Keep link count with file → delete file when count = 0 (hard links)
- Looping: directory traversal algorithms may loop


Directory
Name
Location
foo
467
bar
843
Soft
link
foo_hard
467
Hard link to foo
foo_soft
/foo

Mounting I

Mount operation

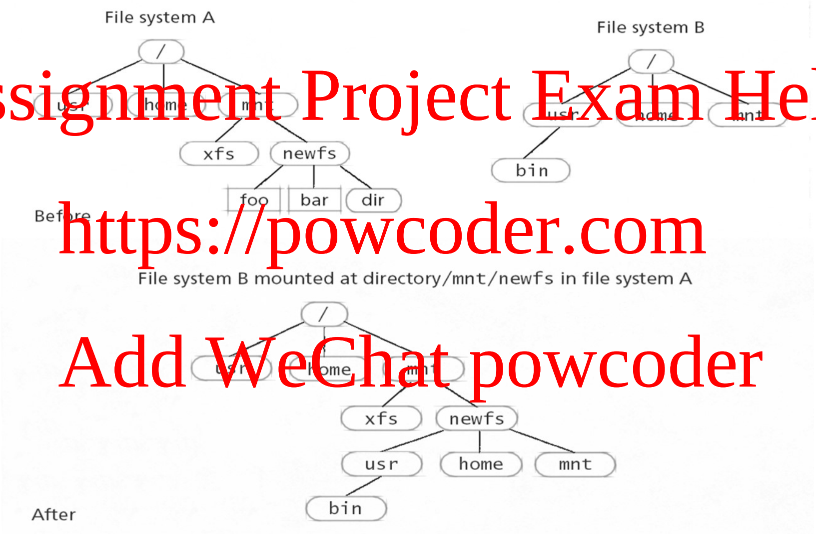
- Combines multiple FSs into one namespace
- Allows reference from single root directory
- Support for soft-links to files in mounted FSs but not hard-links

Mount point

- Directory in native FS assigned to root of mounted FS

FSs manage mounted directories with mount tables

- Information about location of mount points and devices
- When native FS encounters mount point, use mount table to determine device and type of mounted FS


File system A
/
usr
home
mnt
xfs
newfs
foo
bar
dir
File system B
/
usr
home
mnt
bin
Before
File system B mounted at directory/mnt/newfs in file system A
/
usr
home
mnt
xfs
newfs
usr
home
mnt
bin
After

Linux ext2fs

Second extended file system (1993)

Goal: high-performance, robust FS with support for advanced features

Typical block sizes: 1024, 2048, 4096 or 8192 bytes

Safety mechanism: 5% of blocks reserved for root

- Allow root processes to continue to run after malicious/errant user process consumes all FS disk space

Represents files and directories in ext2 FS

Stores information relevant to single file/directory → e.g. time stamps, permissions, owner, pointers to data blocks

ext2 inode pointers

- First 12 pointers directly locate 12 data blocks
- 13^{th} pointer is indirect pointer
  - Locates block of pointers to data blocks
- 14^{th} pointer is a doubly-indirect pointer
  - Locates block of indirect pointers
- 15^{th} pointer is triply-indirect pointer
  - Locates block of doubly indirect pointers

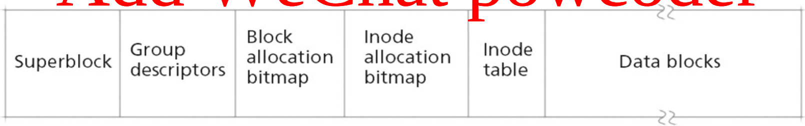
Provides fast access to small files, while supporting very large files

Block groups

- Clusters of contiguous blocks
- FS attempts to store related data in same block group
- Reduces seek time for accessing groups of related data

Block group structure

- Superblock: Critical data about entire FS
  - e.g. total num of blocks and inodes, size of block groups, time FS was mounted, ...
  - Redundant copies of superblock in some block groups



Block group

Inode table: Contains entry for each inode in block group

Inode allocation bitmap: Inodes used within block group

Block allocation bitmaps: Blocks used within group

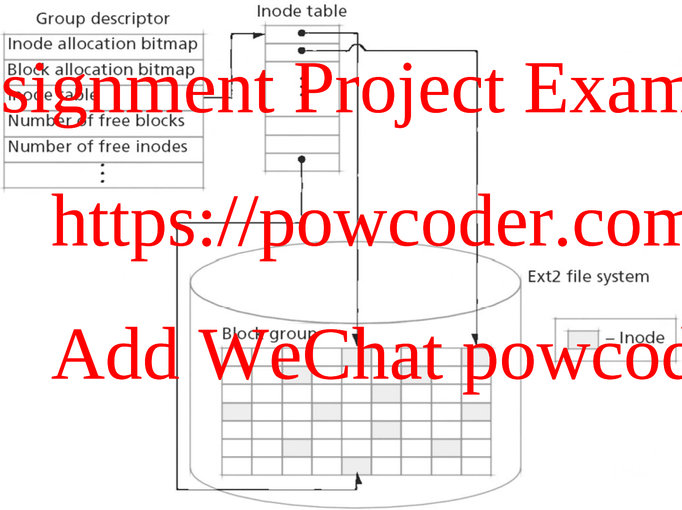
Group descriptor: block numbers for location of:

- inode table
- inode allocation bitmap
- block allocation bitmap
- accounting information

Data blocks: Remaining blocks store file/directory data

- Directory information stored in directory entries
- Each directory entry is composed of: inode number, directory entry length, file name length, file type, file name

ext2fs Block Groups III



ext2 vs. ext3 vs. ext4

Feature	ext2	ext3	ext4
Year	1993	2001	2008
Kernel	0.99	2.4.15	2.6.19
Journaling	N	Y	Y
Max file size	16 GB – 2TB	16 GB – 2TB	16 GB – 16 TB
File system size	2 GB – 32 GB	2 GB – 32 GB	1 EB (Exabyte)