File Management

COMP3231
Operating Systems

References

- Textbook
  - Tanenbaum, Chapter 6

Files

- Named repository for data
  - Potentially large amount of data
    - Beyond that available via virtual memory
      - (Except maybe 64-bit systems)
  - File lifetime is independent of process lifetime
  - Used to share data between processes
- Convenience
  - Input to applications is by means of a file
  - Output is saved in a file for long-term storage

File Management

- File management system is considered part of the operating system
  - Manages a trusted, shared resource
  - Bridges the gap between:
    - low-level disk organisation (an array of blocks),
    - and the user's views (a stream or collection of records)
- Also includes tools outside the kernel
  - E.g. formatting, recovery, defrag, consistency, and backup utilities.

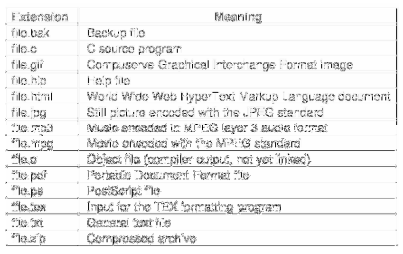
Objectives for a File Management System

- Provide a convenient naming system for files
- Provide uniform I/O support for a variety of storage device types
  - Same file abstraction
- Provide a standardized set of I/O interface routines
  - Storage device drivers interchangeable
- Guarantee that the data in the file are valid
- Optimise performance
- Minimize or eliminate the potential for lost or destroyed data
- Provide I/O support and access control for multiple users
- Support system administration (e.g., backups)

File Names

- File system must provide a convenient naming scheme
  - Textual Names
  - May have restrictions
    - Only certain characters
      - E.g. no '/' characters
    - Limited length
    - Only certain format
      - E.g DOS, 8 + 3
  - Case (in)sensitive
  - Names may obey conventions (.c files or C files)
    - Interpreted by tools (UNIX)
    - Interpreted by operating system (Windows)

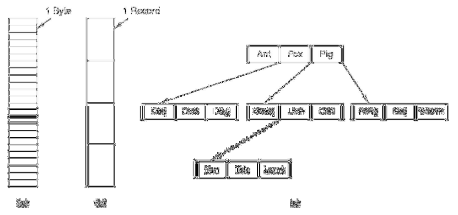
File Naming



Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Typical file extensions.

File Structure
From OS's perspective



- Three kinds of files
  - byte sequence
  - record sequence
  - tree

File Structure

- Stream of Bytes
  - OS considers a file to be unstructured
  - Simplifies file management for the OS
  - Applications can impose their own structure
  - Used by UNIX, Windows, most modern OSes
- Records
  - Collection of bytes treated as a unit
    - Example: employee record
  - Operations at the level of records (read_rec, write_rec)
  - File is a collection of similar records
  - OS can optimise operations on records

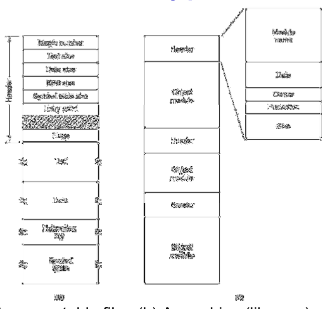
File Structure

- Tree of Records
  - Records of variable length
  - Each has an associated key
  - Record retrieval based on key
  - Used on some data processing systems (mainframes)

File Types

- Regular files
- Directories
- Device Files
  - May be divided into
    - Character Devices – stream of bytes
    - Block Devices
- Some systems distinguish between regular file types
  - ASCII text files, binary files
- At minimum, all systems recognise their own executable file format
  - May use a *magic number*

File Types

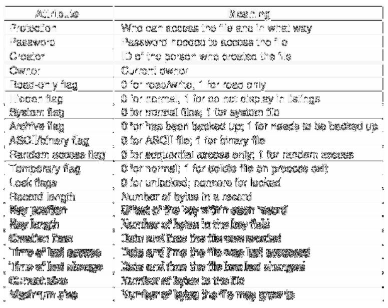


(a) An executable file (b) An archive (libxyz.a)

File Access

- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was mag tape
- Random access
  - bytes/records read in any order
  - essential for data base systems
  - read can be …
    - move file pointer (seek), then read or …
    - each read specifies the file pointer

File Attributes



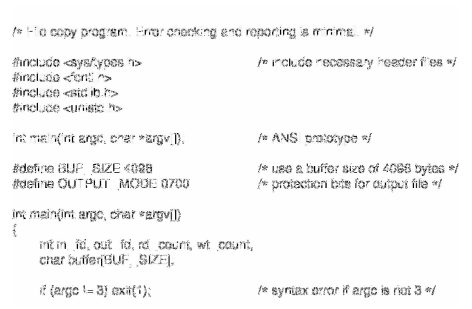
Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Possible file attributes

Typical File Operations

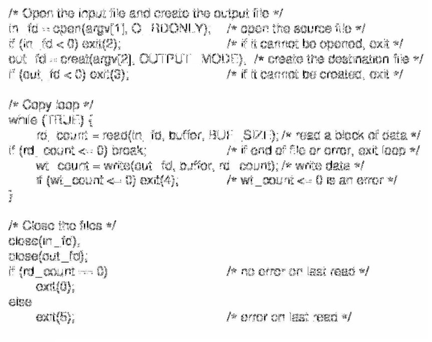
1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set Attributes
11. Rename

An Example Program Using File System Calls (1/2)



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

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

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

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

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

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

An Example Program Using File System Calls (2/2)



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

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

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

File Organisation and Access

Programmer's Perspective

- Given an operating system supporting unstructured files that are a *stream-of-bytes,*

  how should one organise the contents of the files?

File Organisation and Access
Programmer's Perspective

- Performance considerations:
  - File system performance affects overall system performance
  - Organisation of the file system affects performance
  - File organisation (data layout) affects performance
    - depends on access patterns
- Possible access patterns:
  - Read the whole file
  - Read individual blocks or records from a file
  - Read blocks or records preceding or following the current one
  - Retrieve a set of records
  - Write a whole file sequentially
  - Insert/delete/update records in a file
  - Update blocks in a file

Criteria for File Organization

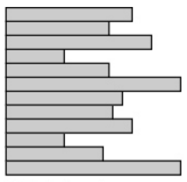
- Rapid access
  - Needed when accessing a single record
  - Not needed for batch mode
- Ease of update
  - File on CD-ROM will not be updated, so this is not a concern
- Economy of storage
  - Should be minimum redundancy in the data
  - Redundancy can be used to speed access such as an index
- Simple maintenance
- Reliability

Classic File Organisations

- There are many ways to organise a file's contents, here are just a few basic methods
  - Unstructured Stream (Pile)
  - Sequential
  - Indexed Sequential
  - Direct or Hashed

Unstructured Stream

- Data are collected in the order they arrive
- Purpose is to accumulate a mass of data and save it
- Records may have different fields
- No structure
- Record access is by exhaustive search

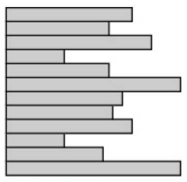

Variable-length records
Variable set of fields
Chronological order

(a) Pile File

Figure 12.3 Common File Organizations

Unstructured Stream Performance

- Update
  - Same size record - okay
  - Variable size - poor
- Retrieval
  - Single record - poor
  - Subset – poor
  - Exhaustive - okay

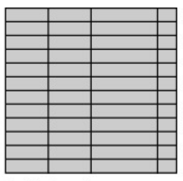

Variable-length records
Variable set of fields
Chronological order

(a) Pile File

Figure 12.3 Common File Organizations

The Sequential File

- Fixed format used for records
- Records are the same length
- Field names and lengths are attributes of the file
- One field is the key field
  - Uniquely identifies the record
  - Records are stored in key sequence

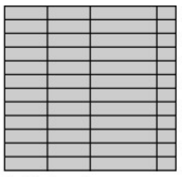

Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential File

Figure 12.3 Common File Organizations

The Sequential File

- Update
  - Same size record - good
  - Variable size – No
- Retrieval
  - Single record - poor
  - Subset – poor
  - Exhaustive - okay



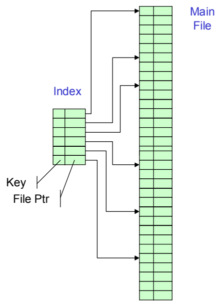
Fixed-length records
Fixed set of fields in fixed order
Sequential order based on key field

(b) Sequential File

Figure 12.3 Common File Organizations

Indexed Sequential File

- Index provides a lookup capability to quickly reach the vicinity of the desired record
  - Contains key field and a pointer to the main file
  - Indexed is searched to find highest key value that is equal or less than the desired key value
  - Search continues in the main file at the location indicated by the pointer

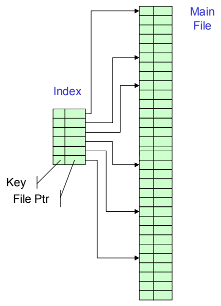


Comparison of sequential and indexed sequential lookup

- Example: a file contains 1 million records
- On average 500,00 accesses are required to find a record in a sequential file
- If an index contains 1000 entries, it will take on average 500 accesses to find the key, followed by 500 accesses in the main file. Now on average it is 1000 accesses

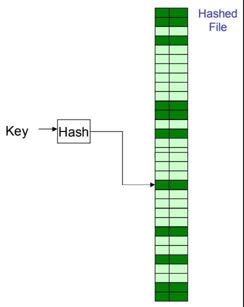
Indexed Sequential File

- Update
  - Same size record - good
  - Variable size - No
- Retrieval
  - Single record - good
  - Subset – poor
  - Exhaustive - okay



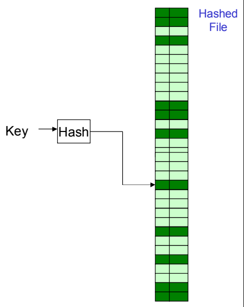
The Direct, or Hashed File

- Key field required for each record
- Key maps directly or via a hash mechanism to an address within the file
- Directly access a block at a the known address



The Direct, or Hashed File

- Update
  - Same size record - good
  - Variable size – No
    - Fixed sized records used
- Retrieval
  - Single record - excellent
  - Subset – poor
  - Exhaustive - poor

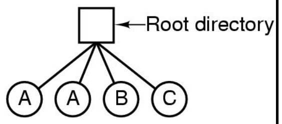


File Directories

- Contains information about files
  - Attributes
  - Location
  - Ownership
- Directory itself is a file owned by the operating system
- Provides mapping between file names and the files themselves

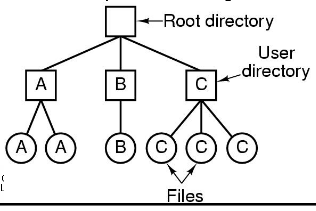
Simple Structure for a Directory

- List of entries, one for each file
- Sequential file with the name of the file serving as the key
- Provides no help in organising the files
- Forces user to be careful not to use the same name for two different files



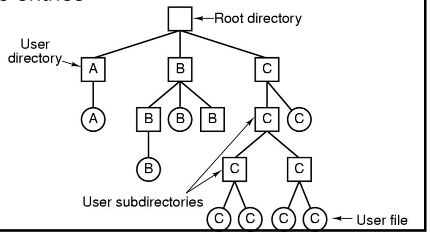
Two-level Scheme for a Directory

- One directory for each user and a master directory
- Master directory contains entry for each user
  - Provides access control information
- Each user directory is a simple list of files for that user
- Still provides no help in structuring collections of files



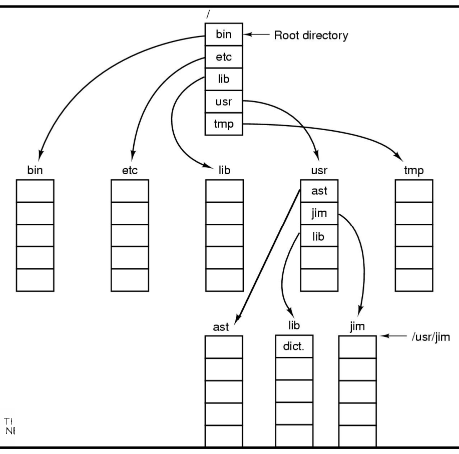
Hierarchical, or Tree-Structured Directory

- Master directory with user directories underneath it
- Each user directory may have subdirectories and files as entries



Hierarchical, or Tree-Structured Directory

- Files can be located by following a path from the root, or master, directory down various branches
  - This is the *absolute* pathname for the file
- Can have several files with the same file name as long as they have unique path names



Current *Working Directory*

- Always specifying the absolute pathname for a file is tedious!
- Introduce the idea of a *working directory*
  - Files are referenced relative to the working directory
- Example: cwd = /home/kevine

  .profile = /home/kevine/.profile

Relative and Absolute Pathnames

- Absolute pathname
  - A path specified from the root of the file system to the file
- A *Relative* pathname
  - A pathname specified from the cwd
- Note: '.' (dot) and '..' (dotdot) refer to current and parent directory

Example: cwd = /home/kevine

```
../../etc/passwd
/etc/passwd
../kevine/../.././etc/passwd
```

Are all the same file

Typical Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

Nice properties of UNIX naming

- Simple, regular format
  - Names referring to different servers, objects, etc., have the same syntax.
    - Regular tools can be used where specialised tools would be otherwise needed.
- Location independent
  - Objects can be distributed or migrated, and continue with the same names.

An example of a bad naming convention

- From, Rob Pike and Peter Weinberger, "The Hideous Name", Bell Labs TR

UCBVAX::SYS$DISK:[ROB.BIN]CAT_V.EXE;13

File Sharing

- In multiuser system, allow files to be shared among users
- Two issues
  - Access rights
  - Management of simultaneous access

Access Rights

- None
  - User may not know of the existence of the file
  - User is not allowed to read the user directory that includes the file
- Knowledge
  - User can only determine that the file exists and who its owner is

Access Rights

- Execution
  - The user can load and execute a program but cannot copy it
- Reading
  - The user can read the file for any purpose, including copying and execution
- Appending
  - The user can add data to the file but cannot modify or delete any of the file's contents

Access Rights

- Updating
  - The user can modify, deleted, and add to the file's data. This includes creating the file, rewriting it, and removing all or part of the data
- Changing protection
  - User can change access rights granted to other users
- Deletion
  - User can delete the file

Access Rights

- Owners
  - Has all rights previously listed
  - May grant rights to others using the following classes of users
    - Specific user
    - User groups
    - All for public files

Case Study: UNIX Access Permissions

```
total 1704
drwxr-x---    3 kevine   kevine       4096 Oct 14 08:13 .
drwxr-x---    3 kevine   kevine       4096 Oct 14 08:14 ..
drwxr-x---    2 kevine   kevine       4096 Oct 14 08:12 backup
-rw-r-----    1 kevine   kevine     141133 Oct 14 08:13 eniac3.jpg
-rw-r-----    1 kevine   kevine    1580544 Oct 14 08:13 wk11.ppt
```

- First letter: file type
  - ***d*** for directories
  - - for regular files)
- Three user categories
  - ***u***ser, ***g***roup, and ***o***ther

UNIX Access Permissions

```
total 1704
drwxr-x---    3 kevine   kevine       4096 Oct 14 08:13 .
drwxr-x---    3 kevine   kevine       4096 Oct 14 08:14 ..
drwxr-x---    2 kevine   kevine       4096 Oct 14 08:12 backup
-rw-r-----    1 kevine   kevine     141133 Oct 14 08:13 eniac3.jpg
-rw-r-----    1 kevine   kevine    1580544 Oct 14 08:13 wk11.ppt
```

- Three access rights per category
  - ***r***ead, ***w***rite, and e***x***ecute

drwxrwxrwx

user group other

UNIX Access Permissions

```
total 1704
drwxr-x---    3 kevine   kevine        4096 Oct 14 08:13 .
drwxr-x---    3 kevine   kevine        4096 Oct 14 08:14 ..
drwxr-x---    2 kevine   kevine        4096 Oct 14 08:12 backup
-rw-r-----    1 kevine   kevine      141133 Oct 14 08:13 eniac3.jpg
-rw-r-----    1 kevine   kevine     1580544 Oct 14 08:13 wk11.ppt
```

- Execute permission for directory?
  - Permission to access files in the directory
- To list a directory requires read permissions
- What about **`drwxr-x—x`**?

UNIX Access Permissions

- Shortcoming
  - The three user categories a rather coarse
- Problematic example
  - Joe owns file **`foo.bar`**
  - Joe wishes to keep his file private
    - Inaccessible to the general public
  - Joe wishes to give Bill read and write access
  - Joe wishes to give Peter read-only access
  - How????????

Simultaneous Access

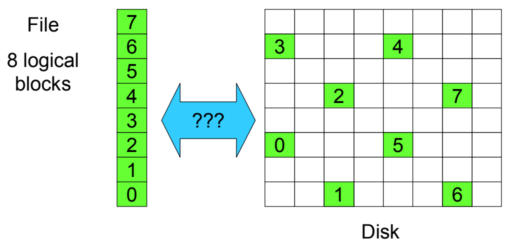
- Most Oses provide mechanisms for users to manage concurrent access to files
  - Example: lockf(), flock() system calls
- Typically
  - User may lock entire file when it is to be updated
  - User may lock the individual records during the update
- Mutual exclusion and deadlock are issues for shared access

File Management II

COMP3231

Operating Systems

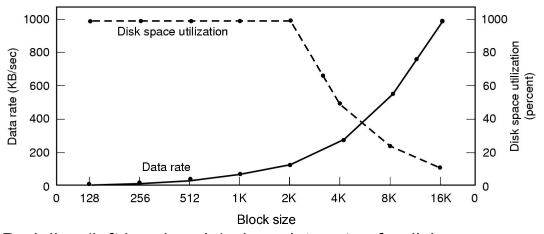
Implementing Files



Trade-off in physical block size

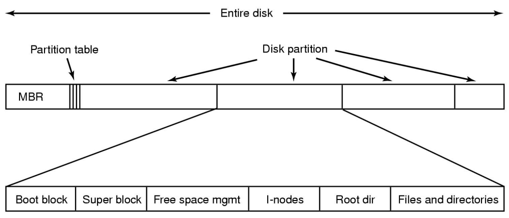
- Sequential Access
  - The larger the block size, the fewer I/O operation required
- Random Access
  - The larger the block size, the more unrelated data loaded.
  - Spatial locality of access improves the situation
- Choosing the an appropriate block size is a compromise

Example Block Size Trade-off



- Dark line (left hand scale) gives data rate of a disk
- Dotted line (right hand scale) gives disk space efficiency
  - All files 2KB (an approximate median file size)

File System Implementation



A possible file system layout

Implementing Files

- The file system must keep track of
  - which blocks belong to which files.
  - in what order the blocks form the file
  - which blocks are free for allocation
- Given a logical region of a file, the file system must identify the corresponding block(s) on disk.
  - Stored in file system *metadata*
    - *file allocation table (FAT)*, directory, I-node
- Creating and writing files allocates blocks on disk
  - How?

Allocation Strategies

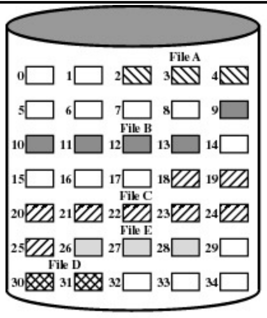
- Preallocation
  - Need the maximum size for the file at the time of creation
  - Difficult to reliably estimate the maximum potential size of the file
  - Tend to overestimated file size so as not to run out of space
- Dynamic Allocation
  - Allocated in *portions* as needed

Portion Size

- Extremes
  - Portion size = length of file (contiguous allocation)
  - Portion size = block size
- Tradeoffs
  - Contiguity increases performance for sequential operations
  - Many small portions increase the size of the *metadata* required to book-keep components of a file, free-space, etc.
  - Fixed-sized portions simplify reallocation of space
  - Variable-sized portions minimise internal fragmentation losses

Methods of File Allocation

- Contiguous allocation
  - Single set of blocks is allocated to a file at the time of creation
  - Only a single entry in the directory entry
    - Starting block and length of the file
- External fragmentation will occur

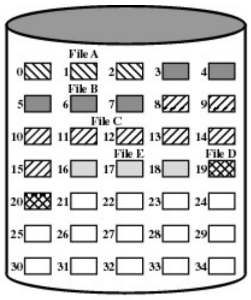


directory

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Figure 12.7 Contiguous File Allocation

- Eventually, we will need compaction to reclaim unusable disk space.



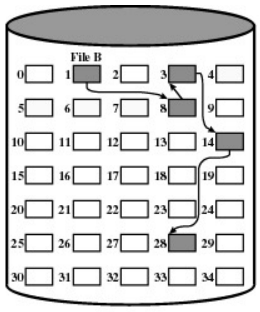
directory

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Figure 12.8 Contiguous File Allocation (After Compaction

Methods of File Allocation

- Chained (or linked list) allocation
- Allocation on basis of individual block
  - Each block contains a pointer to the next block in the chain
  - Only single entry in a directory entry
    - Starting block and length of file
- No external fragmentation
- Best for sequential files
  - Poor for random access
- No accommodation of the principle of locality
  - Blocks end up scattered across the disk



directory

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Figure 12.9 Chained Allocation

- To improve performance, we can run a *defragmentation* utility to consolidate files.

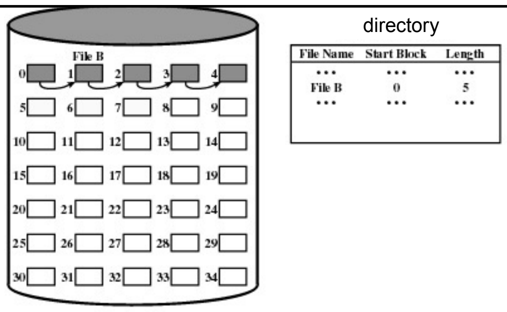


Figure 12.10 Chained Allocation (after consolidation)

Methods of File Allocation

- Indexed allocation
  - File allocation table contains a separate one-level index for each file
  - The index has one entry for each portion allocated to the file
  - The file allocation table contains block number for the index

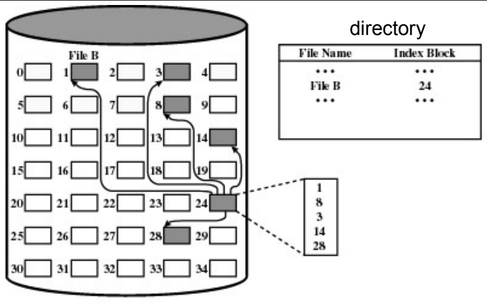


Figure 12.11 Indexed Allocation with Block Portions

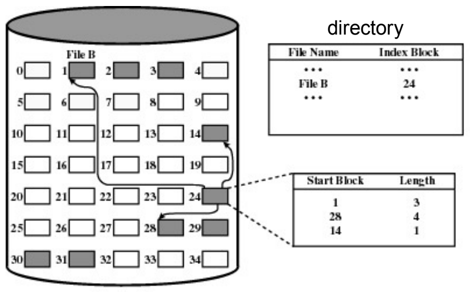
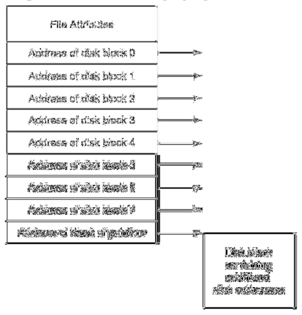


Figure 12.12 Indexed Allocation with Variable-Length Portions

Indexed Allocation

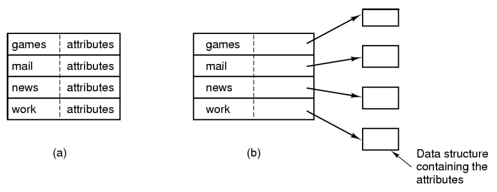
- Supports both sequential and direct access to the file
- Portions
  - Block sized
    - Eliminates external fragmentation
  - Variable sized
    - Improves contiguity
    - Reduces index size
- Most common of the three forms of file allocation

UNIX I-node



An example of indexed allocation

Implementing Directories



- Simple fixed-sized directory entries

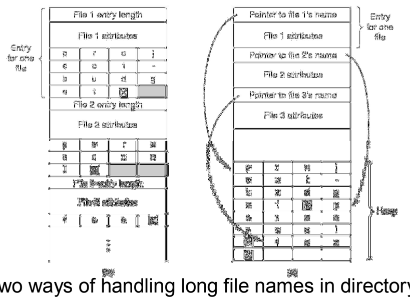
(a) disk addresses and attributes in directory entry
  - DOS/Windows

(b) Directory in which each entry just refers to an i-node
  - UNIX

Fixed Size Directory Entries

- Either too small
  - Example: DOS 8+3 characters
- Waste too much space
  - Example: 255 characters per file name

Implementing Directories



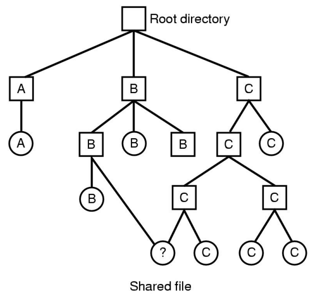
- Two ways of handling long file names in directory
  - (a) In-line
  - (b) In a heap

Implementing Directories

- Free variable length entries can create external fragmentation in directory blocks
  - Can compact when block is in RAM

Shared Files

Files shared under different names

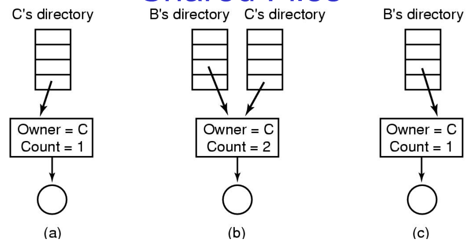


File system containing a shared file

Implementing Shared Files

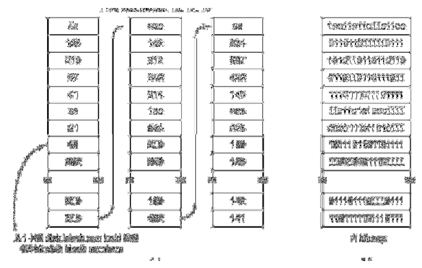
- Copy entire directory entry (including file attributes)
  - Updates to shared file not seen by all parties
  - Not useful
- Keep attributes separate (in I-node) and create a new entry (name) that points to the attributes (hard link)
  - Updates visible
  - If one link remove, the other remains (ownership is an issue)
- Create a special “LINK” file that contains the pathname of the shared file (symbolic link, shortcut).
  - File removal leaves dangling links
  - Not as efficient to access
  - Can point to names outside the particular file system
  - Can transparently replace the file with another

Shared Files



(a) Situation prior to linking
(b) After the link is created
(c)After the original owner removes the file

Free Disk Space Management



(a) Storing the free list on a linked list
(b) A bit map

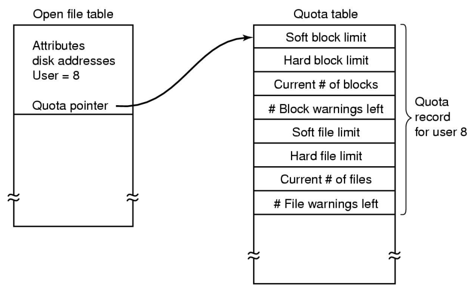
Bit Tables

- Individual bits in a bit vector flags used/free blocks
- 16GB disk with 512-byte blocks →4MB table
- May be too large to hold in main memory
- Expensive to search
  - But may use a two level table
- Concentrating (de)allocations in a portion of the bitmap has desirable effect of concentrating access
- Simple to find contiguous free space

Free Block List

- List of all unallocated blocks
- Manage as LIFO or FIFO on disk with ends in main memory
- Background jobs can re-order list for better contiguity
- Store in free blocks themselves
  - Does not reduce disk capacity

Quotas



Quotas for keeping track of each user's disk use