File Management (Continued)

COMP 229 (Section PP) Week 10

Prof. Richard Zanibbi
Concordia University
March 20, 2006

Last Week...File Management

File System Types (pp. 514-528)

- Low-Level File System *(for Byte Stream Files)*
- Structured File System *(e.g. for records, .mp3 files)*

Low-Level File System Implementations (pp. 529-544)

- Low-level file system architecture
- Byte stream file *Open* and *Close* operations
- Block Management (in part)

Files and the File Manager

Files As Resource Abstraction

Files are used to represent data on storage devices (e.g. disk)

Files As a Central Abstraction of Computation

- Most programs read one or more files, and eventually write results to one or more files
- Some view programs as filters that read data, transform the data, and write the result to a new file (is this an "extreme view"?)
  - e.g. UNIX shell applications using the pipe operator (|)
- In C: by default, stdin, stdout, stderr are defined

File Manager (OS Service Responsible for Files)

- Implements file abstraction ***(including APIs for file types)***
- Implements directory structure for organizing files
- Implements file systems for organizing collections of directories (e.g. on separate disks)
- File management involves operations on external devices & memory

File Descriptors, Revisited

Field	Purpose
External Name	A string identifier
Sharable	Can processes share the file? For read/write/execution?...
Owner	User who created the file
Protection Settings	Which users can read/write/execute the file
Length	Number of bytes in file
Time of Creation	When file was created
Time of Last Modific.	Date when file was last written to
Time of Last Access	Date when file was last used
Reference Count	# directories in which file appears
Storage Device Details	File block organization (on device)

File System Types

External View of File Manager

Part of the System Call Interface implemented by the file manager (see Fig. 13.2)

Low vs. High-Level (Structured) File Systems

See Fig. 13.3

- Low Level File System implements only Stream-Block Translation and Byte-Stream Files (e.g. WIndows, Unix)
  - Applications must translate byte streams to/from abstract data types used in programs
- Structured (High-Level) File Systems also implement Record-Stream translation and Structured Record files (e.g. MacOS, systems for commercial applications, e.g. some IBM systems)
  - Have a language for defining record types, keys for searches

Marshalling: producing blocks from records (“flattening”)
Unmarshalling: producing records from blocks

See Fig. 3.16 (marshalling a record into a byte stream)

Structured File Types

(Record-Oriented) Structured Sequential Files

Records organized as a list

Record attributes encoded in file **Header**

Indexed Sequential Files

- Records have an integer index in their header
- Records contain one or more fields used to index records in the file (e.g. student #)
- ***Applications of File Manager*** **define tables associating record attributes with index values**
- Representation: just index values in records, linked lists (one per *key*), or stored index table (used by file manager)
- Popular in business, human resources applications

Inverted Files

- Generalized external (system) index tables used by file manager: allow entries to point to groups of records or fields
- New record fields are extracted and placed in the index table, with pointer in the table to the new record where appropriate
- Records accessed using the index table rather than location in file

```
struct message {
  address to;
  address from;
  line subject;
  address cc;
  string body;
}

struct message *getRecord(void) {
  struct message *msg;
  msg = allocate(sizeof(message));
  msg->to = getAddress(...);
  msg->from = getAddress(...);
  msg->cc = getAddress(...);
  msg->subject = getLine();
  msg->body = getString();
  return(msg);
}

putRecord(struct message *msg) {
  putAddress(msg->to);
  putAddress(msg->from);
  putAddress(msg->cc);
  putAddress(msg->subject);
  putString(msg->body);
}
```

Abstract Data Type (for email messages)

Retrieves next record

- Programmer-defined abstract data types
- Programmer-defined record read/write methods (e.g. using standard, predefined access function names)
- File Manager invokes programmer routines

Appends record to file

Example: ("Record-Oriented") Structured Sequential File for an "Inbox"

Opening Files

Fig. 13.10

- Opening a file
- Note difference between internal and external file descriptor
- Also, note that an additional process-specific data structure is needed (Process-File Session)

Fig. 13.11

- Opening a Unix file
- Note that Process-File Session is represented using two Data Structures (Open File Table, File Structure Table)

Closing a File

When Issued:

- All pending operations completed (reads, writes)
- Releases I/O buffers
- Release locks a process holds on a file
- Update external file descriptor
- Deallocate file status table entry

This week...File Management (Cont'd)

File Management (pp. 534-558)

- Block Management, Continued
- Reading/Writing Byte Streams
- Supporting High-Level File Abstractions
- Directories (Structures, Implementations)
- File Systems

Block Management, and Reading and Writing Byte Streams

Block Management

Purpose

Organizing the blocks in secondary storage (e.g. disk) that hold file data

(blocks containing file data are referenced from the "Storage Device Detail" field in a file descriptor; see p.530)

Computing number of blocks for a file

See page 534 in text

Block Management Strategies:

1. Contiguous allocation
2. Linked lists
3. Indexed allocation

Examples of Block Management

Contiguous Allocation

See Fig. 13.12

All blocks are adjacent (contiguous) on storage device: fast write/read

Problems with external fragmentation (space between files)

- must be space for the whole file when file is expanded; if not, the whole file must be relocated to another part of storage (e.g. the disk)

NOTE: "Head Position" is the location of the file pointer (current byte)

Linked Lists (Single and Doubly-Linked)

See Figs. 13.13 and 13.14

Blocks have fields defining the number of bytes in a block and link(s) to the next (also previous, if doubly-linked) block in the file

Blocks do not have to be contiguous, allowing us to avoid the fragmentation problems with contiguous allocation

Indexed Allocation

See Fig. 13.15

Size/link headers for blocks separated from the blocks and placed in an index

Index is stored with the file, and loaded into memory when a file is opened

Speeds searching, as all blocks are stored within the index table (no link following to locate blocks in the file)=

Contiguous Allocation Strategies

Best Fit

Find smallest free area large enough for the file

First Fit

- Use first unallocated block sequence large enough for the file
- Found using linear search

Worst Fit

- Use largest available contiguous space
- Divide space in 2, one part for the file, the rest for free space

Block Representation in Unix File Descriptors

UNIX File Descriptors

See Table 13.1

UNIX File Block Representation

See Fig. 13.16

- 12 direct links to data blocks
- 3 index block references, which are singly, doubly, and triply indirect, respectively

Block Types: Data or Index

Index Block: list of other data or index blocks

The Unix block representation can represent more locations than most machines can store (example numbers given in text)

UNIX File Descriptors (inode)

Field	Purpose
Mode	Access permissions
UID	ID for user that created the file
Group ID	ID identifying group of users with access to the file
Length in bytes	Number of bytes in file
Length in blocks	Number of blocks implementing file
Time of Last Modific.	Time when file was last written to
Time of Last Access	Time when file was last *read*
Time of Last inode mod.	Time when inode was last changed
Reference Count	# directories in which file appears
Block references	Pointers, indirect pointers to file block

DOS File Allocation Tables (FAT)

Represents Linked List Allocation

- Each location in the table is a logical block address
- Content of each table cell is a pointer to the next block in a file, or an “end of file” marker

“Modern” FAT Alteration

- Table locations point to *clusters* (a group of contiguous blocks) rather than individual blocks

Unallocated Blocks

"Free List" representation

A linked list of unallocated blocks

Very large, particularly if disk is mostly empty

Hard to efficiently manage so that adjacent free blocks can be easily found

Block Status Map

Represents every block on a disk using one bit

If bit is on, block is used ('0' if free)

Can be kept in memory; much faster to find adjacent free blocks vs. a free list

Useful for checking file system contents

- e.g. comparing blocks that files on a disk reference vs. those that are indicated as used in the Block Status Map, as part of disk recovery

Packing/Unpacking Blocks

Reading/Writing

- File system reads/writes one block of data at a time
- Blocks of data are copied into or out of the byte stream for the file
- On file open, first block is copied into memory (as byte stream), and file pointer located at the start of the first block.
- Processes operate on the copy of the data that is in memory (the byte stream)
- Write operations pack (marshall) bytes into copies of device storage blocks into memory. When blocks become full, the write operation will transfer the blocks to the device
- Note that when we open for writing, the block is still read in, and overwritten. When the program finishes writing out a block, the write operation will arrange for the data to be written to disk

Packing/Unpacking Continued

Location of a Byte on Disk

Assuming we have fixed-sized blocks (of size k),

then we can compute the block containing a byte using $N = \lfloor i/k \rfloor$, where N is the block number

See Fig. 13.18

- Inserting/Deleting bytes in the interior of a byte-stream file
- Leads to internal fragmentation (unused block bytes)

Buffering (see page 544)

To improve performance, file manager tries to read ahead blocks in a file (into a “buffer”), and “write behind” the blocks that have been filled in the buffer used to store bytes in the file

Supporting High-Level File Abstractions

Structured Sequential Files

Implementation

Logically same as for byte stream file manager

– record rather than byte-based

– insertion: similar issues per bytes in low-level files

Additional File Descriptor Fields:

File type (e.g. relocatable object file, postscript)

Access methods (abstract data type function.)

- May be fixed (e.g. OS-provided)
- May be programmer-defined (e.g. email example)

Other: relationships with other files, minimum version number of file manager handling file type, etc.

Indexed Sequential Files

Index Field

Normally indicates order of records (from 0)

Implementation

- File manager manages mapping from records to blocks (I/O requests are record index-based)
- Inserting records causes complications (e.g. fragmentation issues)
- Optional index table
- Buffers unlikely to be very helpful for many usage scenarios (e.g. servicing bank customers)

Database Management Systems

Storage Manager

Replaces file manager

Low-level interface, allow direct manipulation of storage devices

Database Administrator

Chooses organization or records across and within files

Common Database Types

Relational: concerned with efficient searches on relationships

Object-Oriented: application defines access methods

*Conventional file system interfaces do not support these

*Note

Usually data stored in a database cannot be accessed by the "normal" file interfaces of the File Manager

Multimedia Documents

Storage Demands

Similar to databases (OO databases often support multimedia files)

Performance Issues

- Block allocation affects streaming/bandwidth
- File Manager: normally assume that applications will not request multiple blocks simultaneously
- *No widely-used OS's currently provide high performance support for multimedia files

Directories (Structures, Implementations)

File Directories

File Directory

A set of files and other (sub)directories

Principle function: help people find their way around data on a system

Implementation

Directories are stored as additional files

Operations for Directory Management (key parameter: file/directory name)

Enumerate

List contents of a directory (files, subdirectories)

- e.g. using ‘ls’ in Unix, ‘dir’ in Windows

Copy

Make a copy of a file (e.g. within the directory)

Rename

Change file name

Delete

Remove file from current directory; release file descriptor and all blocks in the file

Traverse

Change current directory by indicating new directory to go to

– e.g. ‘cd’ in Unix, Windows

Motivation for Structuring Directories (e.g. organizing lecture notes)

Small Number of Files (e.g. 5 lectures)

- Small enough to easily sort through

'Medium' Number of Files (e.g. 50 lectures)

- Now probably easier to manage if we organize by course, for example (one group of subdirectories)

Large Number of Files (e.g. database, Millions)

- Becomes impractical to manage manually: start employing additional subdirectories
- Perhaps even patterns of subdirectories
  - all courses also have subdirectories for each year
  - each year has subdirectories for each term, etc.

(Pure) Hierarchical File Organization in the "Real" World

File

Paper Document (e.g.)

Folder

Group of Files, with an additional document summarizing the files in the folder (the *directory)*

"Super" Folder

Contains a group of folders and files, with an additional directory summarizing the folder contents

File Drawer...
File Cabinet...
Filing Room...
Filing Building...
Filing Complex...
Filing Nation (?....)

Directory Structures

...refers to how directories are organized and related in a system

Root Directory

Initial node in a hierarchical file system

– e.g. “/” in Unix

Directory Tree

Most common directory structure: purely hierarchical organization (creating a tree)

File/Directory Sharing

– Example: two or more users wish to refer to the same file from within their own home directories

– Reference count in file descriptors used to prevent a shared file from being deleted when still being used

– Tree becomes a directed (and hopefully acyclic) graph

Directory Structures

See Fig. 13.19

Note that a “Strict Hierarchy” is often called a “directory tree”

Absolute and Relative Paths

Absolute Path

Path from the root directory to the desired directory/file

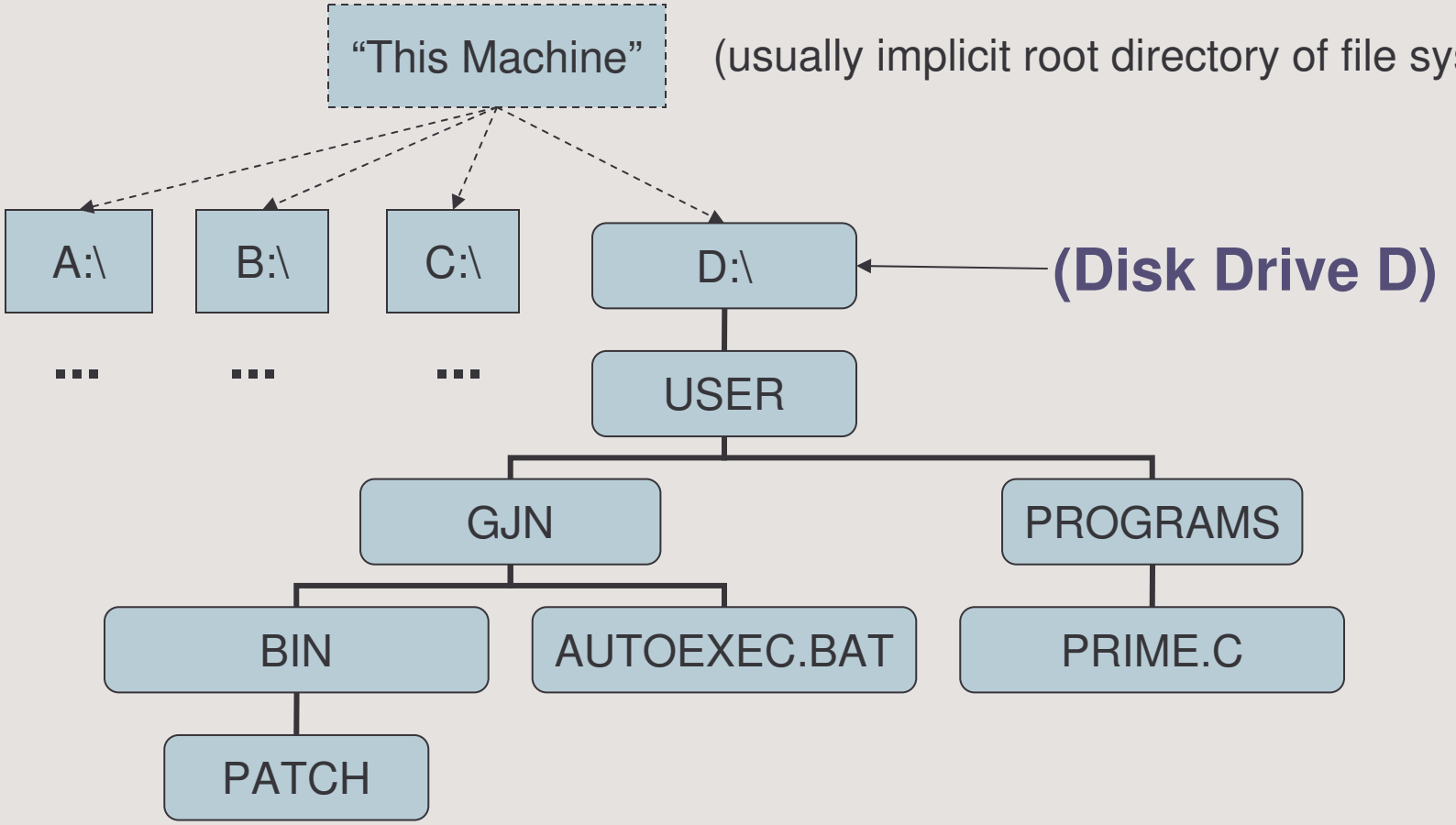
- e.g. “/home/zanibbi/comp229/slides.ppt”
- e.g. “D:\myfiles\zanibbi\comp229\slides.ppt”

Relative Path

Path from (“relative to”) a given directory

- (usually current)
- e.g. : “comp229/slides.ppt” (from /home/zanibbi)
- e.g. : “comp229\slides.ppt” (from D:\myfiles\zanibbi)

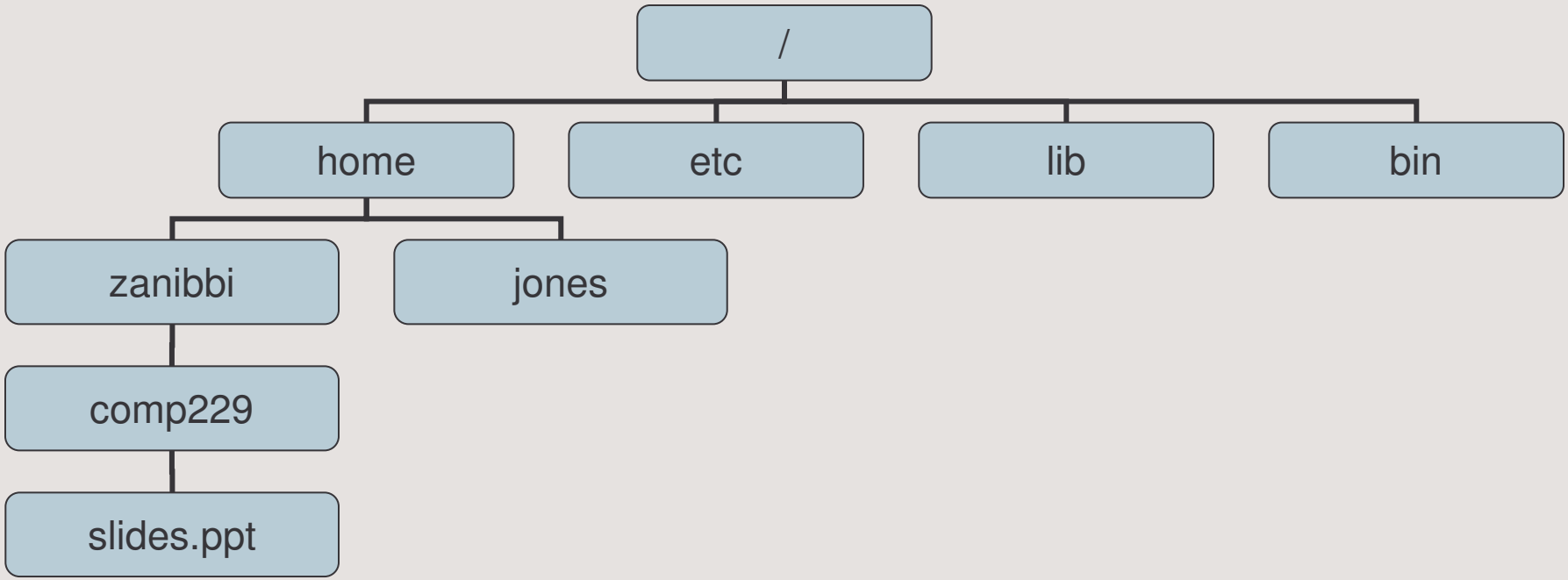
Example: DOS File Directory



Example: Unix File Directories

Directed Acyclic Graphs

Root Directory: “/” (e.g. “cd /”)



Implementing Directories

Directory (as File)

Set of structured records (files/directories)

Directory Entries (Records)

Each record refers to an external file descriptor, provides information for locating the external file descriptor

- **Linux:** Filename, disk address for external file descriptor
- **MS-DOS:** Filename, extension (type), creation time, file size, address of first file block

Implementing Directories, Cont'd

Directory Operations

Operate on directory (record) entries

If file attributes stored on disk, must be retrieved (e.g. Linux)

Long File Names

Old days: Usually 8 character length names (max)

~1990: "long" file names appear

- MS-DOS: extra characters stored in additional directory entry
- Unix: Create a heap in each directory, used for extra characters when needed

Directories On Disk

Directory file has a file descriptor, indicating blocks where directory data is located on disk.

Opening Files, Searching Paths

Path Representation

- Directories, directory entries organize files on disk
- To locate a file, directory entries must be used

Path Searching

Recursive, from the initial directory (root directory for absolute paths)

- e.g. /usr/gjn/books/opsys/chap13 **– 5 directories**
- e.g. books/opsys/chap13 (from /usr/gjn) **– 3 directories (must read *current directory* (“gjn”) to start)**

Caching Directory Information

To speed up searches, some file managers store directory data in memory

File Systems

File Systems

Disk Partitioning

For organization and performance, disks are often partitioned to act as two or more separate disks

Removable Storage

Floppy disks, Zip disks, CD-ROMS, etc.

– must combine directories for use by system

File System

– Hierarchical file collection associated with a removable or permanent storage medium

– Single root directory

– Describes set of files on a removable storage medium

Volume

The contents of a file system

Example: ISO 9660 File System (see Figs. 13.21-13.23)

Creation of Standard

Derived from “High Sierra” specification created by industry experts in 1986

Features

- Focus: reading files (CD’s originally non-writable): simplifies block allocation
- CDs store data in a single helical track
- Sectors (blocks) are 2048 bytes
- Each file/directory is stored in a contiguous set of sectors (contains file descriptor and file)

Mounting File Systems

System Boot Disk

- Logical device (e.g. disk partition) from which the master boot record is loaded
- Also the device that will contain the root directory for the system

Other Disks/Storage

Must be “grafted” onto the base directory structure

This “grafting” is often called “mounting” (from Unix)

Mounting File Systems, Cont'd

mount() – See Fig. 13.24

- Unix system call requesting a removable device be added to the file hierarchy
- Root directory of the mounted system will be treated as a subdirectory in the base hierarchy
- Mounted files can be used as though part of the file system on the system boot disk
  - e.g. "mount /dev/cdrom", "cd /media/cdrom"

umount()

"Unmounts" a device from the base hierarchy

- e.g. before removing a cd-rom

Heterogeneous File Systems

Definition

A file system comprised of files systems of different types (e.g. ISO9660, ext3, FAT32)

Virtual File System (Linux – See Fig. 13.25)

- File manager has file system dependent and independent parts
- **File System Dependent:** reads/writes data
- **File System Independent:** implements general file manager algorithms (e.g. traversal, delete, copy, etc.)
- File Manager defines its own internal file descriptor (need to translate those from different file systems), in Linux, based on inodes (Unix)

Next Week...

Device Management

pp. 152-174

Compilers (*back to Part I of text*)

pp. 225-242

Basic Compiler Functions

Lexical Analysis using Finite State Automata