File Systems

Operating Systems

Oleg Goldshmidt

`ogoldshmidt@computer.org`

Lecture 10

File Concept I

- back to general principles: OS hides complexity from users
- how information is stored on devices is none of the user's business
- present the user with a logical view of stored information
- file: a named collection of related information recorded on a storage device
  - the smallest logical information unit: all stored data are in files
  - an example of “raw” data not using files: databases

File Concept II

- (normally) on high-capacity non-volatile storage
  - maintain data past program termination or failure
  - manipulate large quantities of data (larger than virtual memory)
  - sharing data between processes
- files may contain programs and/or data
- data files: numeric, alphabetic, alphanumeric, binary
- formatted or unformatted data
- file types: source code, object code, executables, text, graphics, sound, etc.

File Types

- different file types may be supported differently
  - complicates the OS implementation considerably
- type specified by extension, or by a combination of filesystem tests, “magic number” tests, and language tests (`file(1)`)
  - filesystem tests (`stat(2), sys/stat.h`): empty or special files (sockets, symlinks, pipes, etc.
  - “magic number” tests: stored in a particular place near the beginning of the file, usually describes a binary format
  - if not special or binary, it is either “text” (ASCII etc.) or “character data” (EBCDIC etc.)

Determining File Type

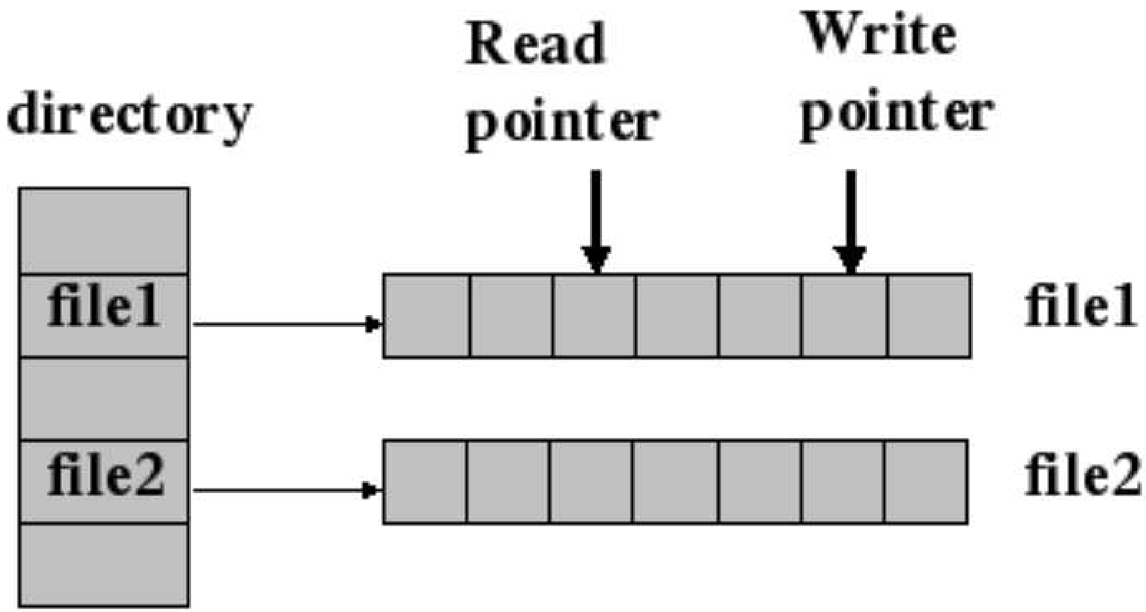
```
#include <sys/types.h>
#include <sys/stat.h>
struct stat  buf;
char         *s;
if (lstat(filename, &buf) < 0)
  exit(EXIT_FAILURE)
if      (S_ISREG(buf.st_mode))  s = "regular";
else if (S_ISDIR(buf.st_mode))  s = "directory";
else if (S_ISCHR(buf.st_mode))  s = "character special";
else if (S_ISBLK(buf.st_mode))  s = "block special";
else if (S_ISFIFO(buf.st_mode)) s = "fifo";
else if (S_ISLNK(buf.st_mode))  s = "symbolic link";
else if (S_ISSOCK(buf.st_mode)) s = "socket";
else                            s = "unknown";
printf("%s\n", s);
```

File Attributes

- name: case-sensitive or not
- type: if different types are supported
- location: storage device and location on the device
- size: in bytes, words, or blocks; possibly also the maximal allowed size
- access control information
- time, date, user: for creation, modification, access
  - security
  - usage monitoring and statistics
  - audit

Directory

- the attributes of all files are kept in a directory
- directory must also be kept on non-volatile storage
- on many systems (e.g., UNIX) directory is also kept in file(s), on others it is a special data structure



File Operations I

- a file is an abstract data type
- basic file operations
  - create: allocate space, make a directory entry, assign some of the attributes (e.g., access permissions)
  - write: a system call specifying the file name and the data to write; the filesystem provides the storage location to write to, must keep a write pointer per file
  - read: a system call that specifies the file name and the memory location to put the data in; the directory is searched, and the system needs a read pointer per file
    - usually a file is either read from or written to — one current position pointer is enough

File Operations II

- basic file operations (cont.)
  - seek: the current position pointer is set to the given value; no actual I/O is performed
  - delete: release the space and erase the directory entry
  - truncate: sometimes we want to keep the file attributes but erase the contents of a file; instead of deleting and then recreating the file we reset the length to zero
- other common operations
  - rename: keep the data and the attributes, change the name
  - get/set attributes

File Operations III

- examples of compound operations
  - append: seek the end, write
  - overwrite: truncate, write
  - copy: create a new file, read from old, write to new
- optimizations
  - open: avoid searching the filesystem directory each time a file is accessed
    - keep an "open file table", use the table index ("file descriptor") throughout
    - some systems may open a file on first reference
    - usually there is `open(2)` and `fopen(3)` that returns a file descriptor or a pointer to the open file table entry
  - close: removes the file from the open file table

File Operations IV

- open and close in multiuser environments (e.g., UNIX)
  - several users may open a file at the same time
  - 2 levels of file tables
    - per-process table containing the files that the process has open; stores the usage information on each file (e.g., the current position)
    - each entry in the process file table points to a global open file table that contains process-independent information: location, size, access times, etc.; also has open count
- other operations
  - lock: whole files or sections thereof (`flock(2)`)
  - map: map file to virtual memory (`mmap(2)`, `munmap(2)`)

Access Methods

- sequential access — record by record in order
  - by far the most common
- direct (a.k.a. relative, random) access
  - fixed length logical records, a program can skip a number of records forward or backward — similar to block access to disk
  - either include block number in `read()` and `write()` or use `seek()` to position correctly
  - user usually deals with blocks numbered relative to the beginning of the file
- indexed access
  - search the index, go directly to the record
  - index may be kept in memory (if small enough)

Directories And Directory Operations

- we store huge amounts of data — need some structure
- physical disks and partitions, or logical volumes
- each logical partition stores information on its files
- operations
  - search (by name or pattern) (`find(1)`)
  - create/delete (`mkdir(1)`, `rmdir(1)`)
  - list a directory (`ls(1)`, `readdir(2)`)
  - rename a file (`mv(1)`, `rename(2)`)
  - traverse the file system (`ftw(3)`, `nftw(3)`)

Reading Directory Contents

```
#include <unistd.h>
#include <limits.h>
#include <sys/types.h>
#include <dirent.h>

char                 buf[PATH_MAX];
DIR                 *dp;
struct dirent       *dirp;

if ((dp=opendir(getcwd(buf,PATH_MAX)) == NULL)
        exit(EXIT_FAILURE);
while ((dirp=readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);
closedir(dp);
```

Directory Structure I

- single level directories
  - all files lumped together in one directory
  - not scalable
  - not suitable for multiple users (unique names etc.)
- two-level directories
  - let each user have a directory
  - still not scalable
  - what if users want to share files
  - other directories are needed for system files

Directory Structure II

- directory tree
  - root directory and subdirectories
  - current directory (`pwd(1)`)
  - changing directories (`cd`, `chdir(2)`)
  - directory stack (`pushd`, `popd`)
  - home directory for user
  - absolute and relative paths
  - do we delete non-empty directories?
  - how do we search for executables?

Directory Structure III

- acyclic graph directories
  - linking files and directories
    - sharing a directory between two users
    - using different implementations
    - etc.
  - hard links — duplicating information
  - soft (symbolic) links — the directory entry contains the target
- multiple names per file (aliases)
- deleting files — dangling links, hard links especially problematic
- reference counting for hard links (`unlink(2)`)

Directory Structure IV

- general graph directory
  - acyclic graphs are simple
    - easy to traverse
    - easy to count references
  - general graphs may have self-referencing structures
  - garbage collection
    - traverse the entire file system marking everything that can be accessed
    - make a second pass, freeing everything that is not marked
    - similar to garbage collection in Lisp, Java, etc.

Filesystem Hierarchy Standard I

- `http://www.pathname.com/fhs/`
- requirements and guidelines for file and directory placement for UNIX-like OS
- support for interoperability, system administration, documentation
- root filesystem: enough to boot, restore, repair
- `/boot`: static files for bootloader (e.g., kernel)
- `/bin`: essential command binaries (for all users)
- `/dev`: device files
- `/etc` host-specific configuration (scripts, but no binaries)

Filesystem Hierarchy Standard II

- `/home`: user home directories (optional)
- `/lib`: essential libraries and kernel modules
- `/mnt`: for temporary mounts
- `/opt`: add-on software and data
- `/tmp`: temporary files
- `/sbin`: system binaries (not for regular users)
- `/usr`: shareable, read-only data
  - `/usr/include`: system headers
- other (non-UNIX) systems have their own rules that may or may not be observed

Access Control

- traditional UNIX
  - owner, group, all
  - read, write, execute
  - `chmod(1)`, `chown(1)`, `chgrp(1)`
  - directories must be executable for `chdir(2)`
  - watch write permissions on directories!
  - default permissions (`umask`)
- other operations may be controlled
  - append, delete, list, rename, copy, edit, etc.
  - on many systems these operations are implemented via read, write, execute, and control is exercised at the lower level only

Access Control II

- accell control lists
  - list allowed operations on a per-user basis
  - allows very fine-grained control
    - e.g., all members of group `students` except for users `john` and `jane` can read this file
  - difficult to use, maintain
  - directory entry is now of variable size — more complicated space management
- other approaches
  - password protection for files or trees, possibly different passwords for different operations

Consistency Semantics

- what happens when users access a file simultaneously?
  - especially if multiple users modify the same file
- session: the series of file accesses between `open` and `close`
- UNIX semantics
  - writes to an open file are visible immediately to all the other users who have the file open
  - all users share the pointer to the current location in a file
  - single file image → contention → processes may be delayed

Consistency Semantics II

- session semantics (Andrew Filesystem)
  - writes to an open file are not visible to other users who have the file open
  - once a file is closed the changes made during the session are visible only in sessions starting later; already open sessions do not see the changes
  - multiple images → no contention → no delays
- immutable shared files
  - once a file is declared shared by its creator it cannot be modified
    - neither contents nor name may be changed
  - simple implementation

Mapping And Mounting

- Windows: mapping a drive
  - files on different devices have different namespaces
  - the full path name always contains the physical device where the file is stored
- UNIX: mounting
  - single directory tree, single namespace
  - a filesystem must be mounted before it can be accessed (like a file must be opened)
  - attach the root of the filesystem on a given device to a particular node of the mail filesystem tree
  - verify that there is a valid filesystem on the device

Filesystem Internal Organization I

- device drivers and interrupt handlers do basic block I/O
- the basic file system issues the appropriate I/O commands to the device driver
- the file organization module maps the file's logical blocks to the physical blocks on the disk
  - knows the location of the file
  - knows how the disk space was allocated
  - manages free space
- logical filesystem
  - works with the directory structure given a symbolic file name
  - handles access control, etc.

Filesystem Internal Organization II

- the layered structure allows
  - using more than one filesystem on a single machine
  - replace the physical filesystem with a layer calling a remote system
    - NFS on UNIX, Linux
    - CIFS (samba) on Windows, Linux
    - AFS, GPFS, etc.
  - implement "virtual" filesystems such as `/proc`, shared memory segments, etc.
- Linux VFS layer
  - presents a common interface to the upper software layers, specific filesystems override default file operations (like base and derived classes in OOP)

Metadata

- "data about data"
  - data location on disk
  - creation date/time
  - last modification date/time
  - last access date/time
  - ownership information
  - access control information
- the above info is often held in a specialized data structure (UNIX: inode) which is a part of `dirent`
- the file name, the parent directory, etc. are included in the directory entry directly

Data Layout

- designed with two criteria in mind
  - availability
  - performance
- availability in presence of failures
  - minimal: power failure should not result in data loss
  - stronger: how much time is needed to "restart" a filesystem
  - metadata are stored differently from data, for availability
- performance — through clever space allocation and caching

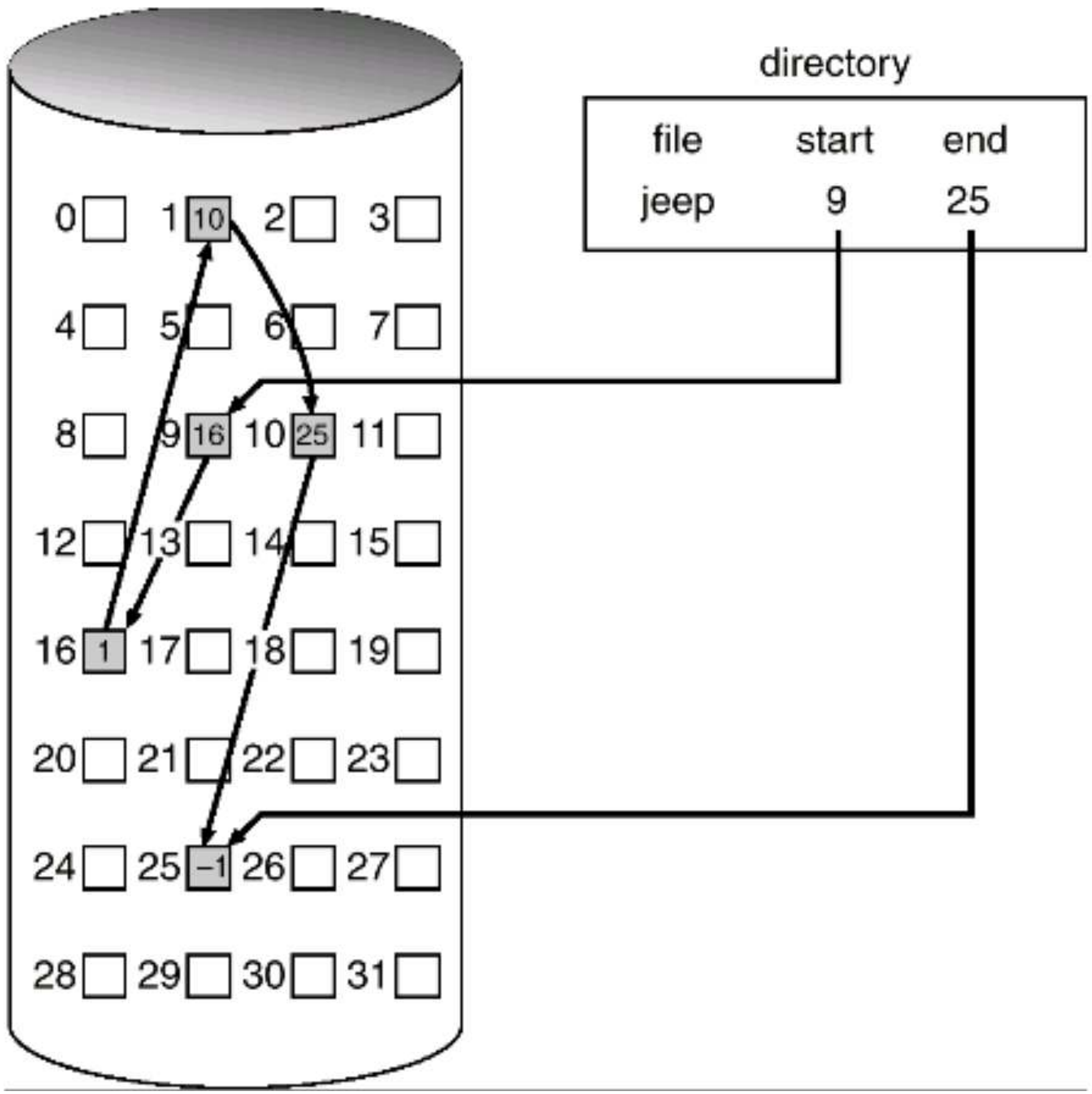
Space Allocation I

- contiguous allocation
  - each file occupies a contiguous set of blocks on disk
  - linear ordering $\rightarrow$ no head movement for seeks when access is sequential $\rightarrow$ good performance
  - algorithms similar to memory allocation (same problem)
  - external fragmentation is a problem
  - how much space will be needed for a file?
    - can relocate files dynamically into a larger hole
    - internal fragmentation
  - modification: allocation in extents

Space Allocation II

- linked allocation
  - each file is a linked list of blocks
    - a bit of overhead — the address of the next block
    - reduce overhead by clustering blocks, at the cost of internal fragmentation
  - directory entry has a pointer to the first block — initially `nil` (empty file)
  - free space management finds new blocks to add
  - efficient only for sequential access (random access to a linked list is lousy)
  - pointers can be damaged by bugs
    - doubly-linked lists may help, but overhead is larger

Linked Allocation



Space Allocation III

- FAT: File Allocation Table
  - a variant of linked allocation (DOS, OS/2)
  - a table at the beginning of a partition
  - table indexed by block number, the value is the next block in the file
  - unused blocks have 0 value
  - a lot of disk seeks, unless the table is cached
  - better random access time

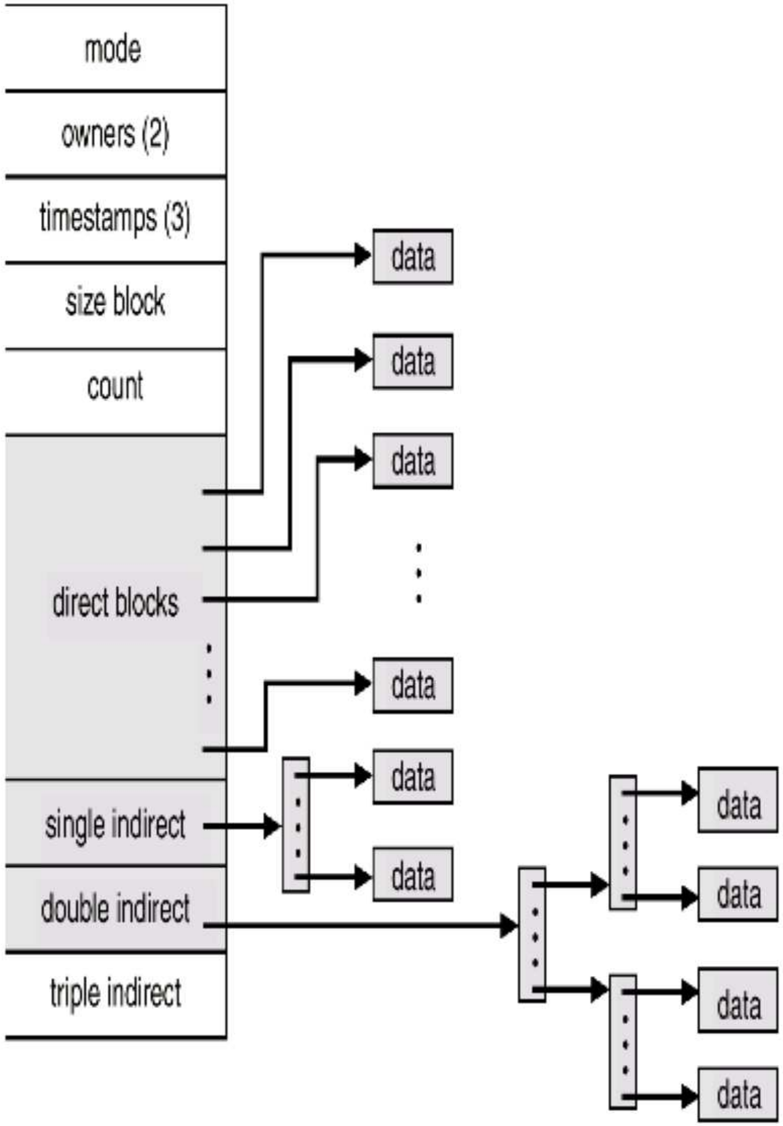
Space Allocation IV

- indexed allocation
  - like linked allocation, but bring all pointers together into the index block
  - each file has an index block — an array of disk block addresses
  - the i-th element contains the address of the i-th block
  - similar to paging
  - random access without external fragmentation
  - overhead is larger than for linked allocation
    - consider a small file — how much space is needed for the block pointers?
  - the table may be multi-level

Inodes I

- combined single level and multi-level indexing
  - inode structure
  - contains some metadata
  - contains n (e.g., 12) direct block pointers
  - pointers to single, double, and triple indirect blocks
  - small files (up to $48\,\mathrm{K}$ for $4\,\mathrm{K}$ blocks) can be accessed directly, no need for a separate index block
  - larger files will use the index tables
  - can be cached in memory
  - data blocks scattered over the disk
- loss of a directory entry can be disastrous — cache inodes for reads, but write inode to disk before the newly allocated data blocks

Inodes II


mode
owners (2)
timestamps (3)
size block
count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

Performance Considerations

- space vs. speed
  - speed favors allocation in large chunks
  - space favors allocation in small chunks
  - know thy workload!
  - new types of data (e.g., video) shift the balance
  - type of access (sequential or random) may be declared when the file is created
    - use linked allocation for sequential access
    - use contiguous allocation for direct (or sequential) access
      - maximal size must be declared
    - can convert from one type to another
  - combine contiguous (for small files) and indexed allocation

Free Space Management

- bitmap
  - bit per block: 1 if free, 0 if allocated
  - many architectures have instructions to find the first 0 or 1 in a bit sequence
  - must be kept in memory (and occasionally written to disk for recovery purposes)
  - need $20\,\mathrm{M}$ for $80\,\mathrm{G}$ disk
- linked list
  - normally sequential access is sufficient
  - FAT incorporates it as is
- grouping — list free blocks in the 1st free block
- counting — keep the number of consecutive free blocks following the current one