

File Systems

Goal

To learn how files are represented both in memory and on the secondary storage devices.

File Systems

A file is a collection of data. There are two aspects of it:

- *virtual*: this is how the user (process) sees the file
- *physical*: this is how the file is represented to the hardware and operating system.

A file's name often reflects something about the file.

example: in TOPS-20, file names are *name.ext*, where *ext* is a three-character extension describing the file; “bas” for BASIC, “for” for FORTRAN, “bli” for BLISS, “obj” for object, “exe” for executable, “txt” for text, and so forth. On Linux, FreeBSD, and MINIX, the last letter may designate something; for example, C source files end in “.c” and C++ source files in “.cc”.

Directories

Files can be organized into *directories* (“folders” to the Mac) to make organizing them easier. A directory contains pairs of
(name, location)

The location may be a physical location (disk address) or an index into an array containing those locations or any other datum used to locate files.

There are several main types of directory organizations; in historical order, they are:

- a one-level (flat) directory in which all files are in the same, single directory.
 - no two files can have the same name (so to keep users having to worry about collisions, the system could make the user name a component of each file name)
 - to find a file, one must search the whole directory
- hierarchical directories impose a tree structure on directories; typically there is a master directory, and then user directories for each user.
 - do absolute and relative path names, current working name.
- graph-structured directory systems are basically hierarchical systems, but with the ability to *alias* files.
 - *direct* aliasing occurs when one (file) location appears twice (or more) in directories, often with different names.
 - *indirect* aliasing occurs when a special type of file containing a path name is created; it is said to be an indirect alias for the file it names. When you refer to the indirect alias, the operating system interpolates the name of the file being aliased.

issues:

- naming: there is no such thing as a "true" name now
- deletion: If a file is deleted under one alias, is it inaccessible using the other aliases?
yes: must find all other aliases and delete them; expensive
no: don't delete file until all aliases deleted; use a *link count* to track how many aliases a file has.
- accounting: usually, the owner of a file pays for storage (and other related charges), but if another user aliases to the file, the owner might no longer be able to delete all references to it!
solution: have each person owning a link to the file (*ie.*, owning a directory containing a link to the file) pay a percentage of the cost of the file.

Information kept in a directory (or indicated by it) is the name, file type, *etc.*

Access Control

Typical protection modes are: *read*, *write*, *append*, *delete*, *privilege* (allows modification of others' rights), *owner* (indicates owner of file), and *search* (grants permission to search directory).

example: UNIX; note difference in meaning of execute for files and directories.

implementation: describe access lists, abbreviation

association of rights: are privileges associated with a *name* or a *file*?

That is, if *x* is an alias for *y*, can a user have read permission on *x* but not on *y*?

Process View of File

Processes operate on files using the following commands:

- *create*: find space for the file, allocate it, and make an entry in the directory
- *open*: begin operations on a file
- *close*: end operations on a file
- *read*: transfer information from the file
- *write*: transfer information to the file
- *rewind*: move to the beginning (or a random point) in the file
- *delete*: remove the file

Access Methods

How can processes access files?

- *sequential*: one block after the other. The process keeps track of a *read/write pointer* (part of the PCB) indicating where the next action is to be done; the pointer always advances.
- *direct*: the read/write pointer can move freely.
- *mapped*: map the file into a virtual segment, and return the segment number rather than the file descriptor; then treat the file as part of the process' virtual store. On closing, just release the storage.
example: TOPS-20, MULTICS, FreeBSD
- *structured*: the file consists of a sequence of records; often the operating system knows about the file type.
example: ISAM (Indexed Sequential Access Method). In this, a small master index points to blocks in a secondary index, which in turn point to real file blocks. Thus, it takes at most 2 reads to locate any record

Information in disk directory file

A *disk directory* is like a directory for a disk; it describes what blocks are in use and which are free. This means it must keep track of what blocks are not in use; such a list is a *free list*. Several representations:

- a bit map, with 1 bit per block
- a linked list of blocks
- like linked list, but in each block of size n on the free list, store $n-1$ numbers of free blocks; the n -th is the address of the next block making up the list
- pairs of (block number, number of free blocks from that block on); if there is more than one contiguous block free, this usually saves some space

The latter three are often called *file maps* because each free block is represented by 1 word (pointer).

Allocation of Disk Blocks to Files

This is done in one of three ways:

- *contiguous allocation*: here, blocks are allocated sequentially (contiguously)
advantages:
 - minimal head motion for sequential reading of file*problems*
 - need to find space for it (using the usual algorithms: first-fit, best-fit, ...). Compaction is possible but usually requires copying almost everything on the disk
 - how much space should be allocated for the file? It might grow beyond its initial allocation.
 - there may be room to increase the allocation;
 - the program may be terminated; in this case, people tend to ask for as much room as possible (wasting space)
 - the file may be moved elsewhere (very slow)

Note that files may grow for years, so even if you know the maximum size a file will ever get, you may waste lots of space for a long time.
- *linked allocation*: the directory contains pointers to the first and last blocks of the file, and the last n bytes of each block in the file point to the next block in the file.
advantages:
 - this scheme eliminates the need to know the size of files in advance
 - again, it is great for files accessed sequentially*disadvantages*:
 - it is poor for direct access files as the operating system must follow links to get to the desired block.
 - it wastes n bytes of disk space per block
 - it is unreliable: if 1 pointer is deleted or changed, the file is garbled; a doubly-linked list, which would ameliorate this, uses more memory.
- *indexed allocation*: this scheme brings all pointers together into one block.
advantages:
 - compact and easy to reference blocks*disadvantages*:
 - wastes more space as an entire block is pointers rather than just 1 word per block (so a 511 block file and a 2 block file use the same number of pointers)

implementation issue If you need more than 1 index block, link them together. Or, use indirection: if you can have 256 pointers/block, 2 levels of indirection allows $256^2 = 65,536$ blocks.

example: UNIX scheme: the first 12 blocks of a file are data, the 13th is an index block, the 14th is a doubly-indexed block (*ie*, it contains pointers to index blocks), and the 15th is a triply-indexed block (*ie*, it contains pointers to doubly-indexed blocks)

Network File Systems

These require that the system know where the file is kept, and be able to communicate with the server.

- *centralized server*: the system determines where the file is kept using a table that shows where it is
Example: NFS and mount points; use the file system to determine which server to talk to
- *distributed file data*: a file contains *metadata*; when you request a file, the system locates this file and uses it to acquire the contents of the file. BitTorrent does this.

NFS protocol

- To the kernel, it's just another file until you get to the mount point, at which point a lower part of the kernel acts as a client to the server.
 1. Mount the file system; this exchanges messages to make the file system available to the client; access modes controlled by various configuration files. Common options:
 - a. *soft*: file system calls that fail after a certain number of retries fail rather than continue retrying forever
 - b. *rdonly*: read only
 - c. *nosuid*: ignore setuid bits
 - d. *nodev*: ignore device files
 2. On open, system walks down directory tree to mount point, then uses file handles to get the "pointer" to the file.
 - a. Handle is all that is needed for access
 - b. Handle includes generation number to detect conflicts
 - c. All accesses use the handle