



Learning objectives for this week:

- Understand the major concepts we've covered, including:
 - files and file systems
 - directory structures
 - security
- Apply these concepts to reasoning about alternative file system designs
- Apply these concepts to reasoning about alternative directory structure designs
- Understand where and how security is an issue

1. Concepts

- a. Why is it necessary to store files noncontiguously?
Files are stored non-contiguously for similar reasons to why memory is allocated non-contiguously. Fragmentation would occur with contiguously allocated files (where you wouldn't be able to create a file of a certain size even though you had enough free space to do so), and it wouldn't allow you to easily extend the size of files if there was fragmentation. Further, extending a file would be impossible if it didn't have free space after its end.
- b. Why are directories necessary?
Directories are necessary because otherwise every file would have to have a unique name – impractical for multi-user systems and for people to remember the names of every file that they've created. Directories are a convenient way of organizing protection as well, and provide a convenient level of granularity for mounting file systems.
- c. Why are access lists more general than the standard UNIX protection scheme?
With the UNIX protection scheme, you can only set permissions for User, Group, and Other, so you'd only be able to set permissions for at most three sets of users (which would correspond to User, Group, and Other). Access lists are more general so that you can set permissions for arbitrary combinations of users. For example, an ACL would allow you to have 1 group of users who could read a file, and another who could write it, which would be difficult – possibly impossible – to achieve with UNIX permissions.

2. File Access and Storage

- a. Explain why it's easy to fake a sequential file on a direct-access file, but not vice-versa.
It's easy to fake sequential file access using direct-access files by just keeping track of the "next" position to read/write blocks from. It is much more difficult to simulate random file access using sequential files as the physical device may not support it, causing each operation to be in linear time. For example, with a magnetic tape drive, you may have to wind the tape (a very slow operation) to read each block.
- b. In the original Macintosh file system, files were structured as follows. Each file had a *resource fork* and a *data fork*, either of which could be empty. A resource fork contained a structured collection of code, user interface elements and data useful for initializing a run of the program. A directory entry included a file's type and creator (represented as 4-byte codes, usually representing 4 capital letters. File names were limited to 31 characters, but could contain any characters except a "." (which was used as a path separator). Compare this arrangement with the naming and structuring conventions of UNIX – what is made easier or harder in the Macintosh approach?
UNIX files do not contain a comparable resource fork – they only contain the file data. It may be possible to simulate this using a custom file format defined by the application developer. Resource forks allows easy identification of the file type, however if you know what the type of the file is (although it may be missing the proper data in the resource fork), an application may refuse to open it. With UNIX files, it can be much harder to determine the file type, unless it is stored elsewhere (such as part of the file name as an extension). Copying files from the Mac to UNIX, and back, may cause issues as some of the metadata may be conflicting when it is imported back to the Mac file system. It's a common convention in UNIX to use the first 2 bytes as an indicator of the type of contents (e.g., a Postscript file is supposed to start with "%!"). But such conventions in effect result in representing metadata (information about a file, as opposed to its contents) either as part of the contents or as part of the name, which makes it easy to modify the metadata by mistake. On the other hand, the Mac approach has drawbacks in portability of files. A UNIX file, provided the data format is not tampered with, can be stored on any file system without translation, whereas a Mac file has to be stored using some convention for splitting the data and resource forks (e.g., in AppleDouble format). Mac OS X uses a different scheme: for interest, try to find out how it addresses the problem of storing metadata and application resources.
- c. Compare the FAT approach (p 425) to the UNIX approach (p 428).
 - i. Is there any difference in the total overhead (extra space needed) to store a small file, e.g., 1 disk block? Explain your answer.
With FAT, you always require the entire file allocation table, which can become quite large. UNIX files only require the inode which can point straight to the data block for a single block file, without the overhead of a large table. Thus for small files on relatively empty drives, UNIX files using inodes can be far more efficient. When the drive is comparatively full, the file allocation table will still be the same size under FAT, so the relative overhead is much smaller. However, the UNIX file still requires the same amount of overheads (which can be larger than with FAT). FAT also places a hard limit on the number of blocks you can allocate, since it uses in-memory data structures and pointers. If for example you have a disk big enough that you need more than 2^{32} blocks, FAT would have to be reimplemented to support bigger pointers.
 - ii. For a very large file, e.g., 20GB, which scheme is better? Explain your answer.
FAT uses a fixed allocation table size irrespective of the size of the stored files, whereas the UNIX files use inodes that contain tables which have 0 – 3 levels of indirection. For a 20 GB file, a UNIX file would require the use of triple-indirection tables, i.e. requiring many more tables beyond the initial inode (thus having a larger overhead). FAT, on the other hand,

requires more work to get to a particular random location, since blocks are organized as a list. Reaching a block relatively far from the start of the file with the UNIX scheme is better than a linear search, as it is more like descending a tree. Also note that point in (i) about problems with pointer sizes in using a FAT for very large files (or partitions).

- iii. Which of the two schemes in general is likely to result in more disk accesses?

Looking at it simplistically, it depends whether or not the entire file allocation table is cached in RAM. If it is, then traversing the linked lists in the table do not require I/O operations, so only one operation is required to read/write the block. However, if the table is not entirely in RAM (which is possible with very large hard drives) then traversing the table may require many I/O operations in addition to accessing the block. With UNIX inodes, in the worst case the triple-indirection tables would need to be traversed (causing three I/O operations), in addition to the operation to access the block. Thus for cached allocation tables, FAT requires fewer I/O operations, and without caching, UNIX files require fewer operations. However, in the UNIX model, the indirect pointers point to either a specific block, or a block of pointers, which is why the worst case is correct. Of course inodes can also be cached.

- d. Explain why managing free space is important from the point of view of performance.

Many file operations that occur frequently, such as creating and append data to files, require free space. Therefore an efficient management scheme will greatly impact performance. Also, it is important to try to keep files contiguous wherever possible, to avoid a problem called fragmentation, which we didn't cover in lectures, where logically sequential file accesses result in more-or-less random patterns of disk head movement.

3. Directory Structures and Mounting

- a. Explain why a tree-structured directory does not meet all of the following design goals for naming files:

- it should be possible to name a file only considering others used in a like context
- it should be possible to reuse names for different purposes
- it should be possible to give more than one name to the same thing
- names should be convenient for the user

Tree-structured directories do not allow users to give more than one name to the same file. If they did, then it would no longer be a tree (as several nodes could point to one node).

As a follow-up question, consider why a tree-structured directory does meet the other design goals.

- b. What mechanism solves any problems you have identified in (a) – for example, in UNIX?

Hard (direct) links.

A hard link allows a file to have a different name, which can be in another directory (or the same name repeated in another directory).

- c. Explain how NFS supports the concept of using any machine on a network as if it was your own machine.

You can mount a part of a remote filing system onto a local name so that it appears that those files are on the local machine.

Typically, this will include your home directory and any applications or libraries you are likely to use. NFS does not specifically define a consistent distributed name space, but system administrators will usually design an NFS mounting strategy which makes all machines look consistent.

4. Security and Dependability

- a. If you would like Tom, Don and Sue all to have read-only access to a file, but Don and Sue should have write access to another file, which Tom should not be able to read or write, how would you achieve this using:

- i. UNIX-style permissions?

A simple solution would be to put Tom & Don in a group, and give the group read/write access to the second file. Both files should allow read only access for "other", but this answer assumes there are no other users. You would have to have at least 2 groups, e.g., the first file belongs to group1, with group permissions "r" for the file, and all the named users would be in that group. Another group, e.g., group2, would own the second file, with group permission set to "w", and owner permission to no read or write. Then, Tom could be the owner of the second file (or you could make "other" permission no read or write, and not have Tom as owner), and all 3 users would be in group1, but only Don and Sue in group2.

- ii. an access list?

You can use the access matrix to specify permissions for each of the users. An access control list lists users and what permissions they have (it can also in some implementations include negative rights). So in this case, the ACL for the first file would list users as follows:

Tom:read, Don:read, Sue:read

and for the second file:

Don:write, Sue:write

- b. In general, what are points for and against the two mechanisms for access control?

For access lists:

- you can have a complex set up of permissions

Against access lists:

- space: directories entries aren't a fixed size
- if a new user is added, you may have to add access permissions for them to every object
- it may be too fine grained and complex to use

For UNIX permissions: simple

- permission data can be stored with the directory entries

Against UNIX permissions:

- very coarse grained, can only set permissions for User, Group, and Other

In general: In practice some ACL-based systems only support permissions at directory level. Solaris compromises by allowing both ACLs and UNIX-style permissions. The general trade-off is ACLs are more general, at the cost of less predictable (generally greater) space required to store them.

- c. What performance overheads would you expect of a journaling file system? Do you consider those overheads to be justified?

You need to keep track of every operation that occurs in the log file so that if a failure occurs you can roll back. Whether these overheads are justified depends upon how important the data is. However, much of the time overhead can be hidden because asynchronous I/O will prevent the CPU from seeing the extra I/O. However, if a read follows a write, it will have to wait until the write had been both logged and written to disk. In that case, the CPU will see a speed drop. Another justification is that the lengthy file system check (e.g., fsck in UNIX) needed after a crash is eliminated.