CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2004

Lecture 18: Naming, Directories, and File Caching

18.0 Main Points

How do users name files? What is a name?

Lookup: given a name, how do you translate it into a file header?

18.1 Abstractions: File Systems, Directories, and Names

The user is given the view of a single namespace for files, but this namespace can be implemented over multiple physical devices or even over multiple systems.

Likewise, it is useful to split a single physical device into several logical devices, such as ones for swap space, different types of uses etc (for example, my laptop has on logical device for regular files and one for multimedia applications). Can vary the file-system parameters per logical device (e.g., block size, various policies).

1. Logical file system
2. Multiple physical file systems
3. Logical devices
4. Physical devices

In UNIX, these file systems must be "mounted" to be used. Mounting a file system into the hierarchy of the *root file system*. A file system contains a boot block in the first sector (if it is a boot-able file system) and a *superblock*, which contains the static parameters of the file system such as its allocated size, block sizes, allocation policies, etc. It can also contain a free block list.

In the kernel, a file is identified according to its <*logical device number, inode number*> pair. Inodes in a file system are numbered sequentially. In some (early) versions of UNIX the inodes are kept in a single array, and the inode number is just the index into that array.

Note, as discussed below, a key service of the file system is the mapping of human-readable file names to locations on physical devices. This includes the notion of "directories", which are just files containing mappings of names to inodes (of data files or other directories).

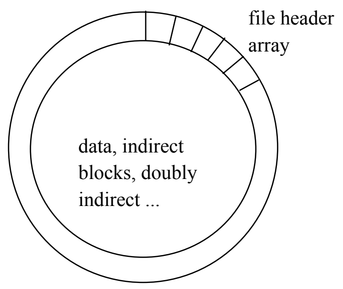
Typically, a human-readable file name is *bound* to an actual file through the use of an "*open*" command. On open, the file path name is traversed and the inode is located. Permissions and other constraints can be checked at this time.

System maintains an *open file table* in order to cache the mapping for use by other file operations (read, write, etc.). After opening the file, these other operations refer to the file by using the index into this table. The index points to a file descriptor that keeps important information about the file such as its *current file position pointer*, which indicates (at least in UNIX) which byte of the file is to be read or written next.

In a multi-user system there are typically two levels of open file table: a global table and a *per process* table. The *per process* table keeps track of the files currently opened by that process and process-specific information such as the current file position. An entry in this table points to the relevant entry in the global (system-wide) table. The global table has one entry per open file (regardless of how many processes have opened it). It tracks the number of processes that have opened the file and closes the file when this number goes to zero.

18.2 File Header Storage

Where is file header stored on disk? In (early) UNIX and DOS/Windows' FAT file system, it is stored in a special array in the outermost cylinders.



UNIX refers to file by index into array – tells it where to find the file header

UNIX-isms:

- "i-node" – file header
- "i-number" – index into the array

Original UNIX file header organization, seems strange:

1. Header not stored anywhere near the data blocks. To read a small file, seek to get header, seek back to data.
2. Fixed size, set when disk is formatted. Means maximum number of files that can be created.

Later versions of UNIX moved the header information to be closer to the data blocks – typically, the inode for a file would be stored in the same "cylinder group" as the parent directory of the file" (makes *ls* of that directory run fast).
+ Reliability: whatever happens to the disk, you can find all of the files

+ UNIX BSD 4.2 puts portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc. in same cylinder => no seeks!

+ File headers are much smaller than a whole block (a few hundred bytes), so multiple file headers fetched from disk at same time

Question: do you ever look at a file header without reading the file? If not, put the file header as the first block of the file!

Turns out that fetching the file header is something like 4 times more common in UNIX than reading the file (ls, make).

18.3 Naming

18.3.1 Options

1. Use index (ask users specify i-node number). Easier for system, not as easy for users.
2. Text name
3. Icon

With icons or text, still have to map name -> index

18.3.2 Directories

Directory maps name -> file index (where to find file header)
Directory is just a table of file name, file index pairs.

General idea: **relation**. Table associating things together.
Directories just a special kind of a relation, connecting file name to index (ditto with password file, caches, etc.)

Each directory is stored as a file, containing a list of "name", index pairs.

But, only OS is permitted to modify directory.

Any program can read the directory file. This is how "ls" works.

Problem: means hard to change file directory structure!

18.3.3 Directory Hierarchy

Directories organized into hierarchical structure

```
/joe/abcde/file1
^ root
  ^ subdir joe
       ^subdir abcde
```

Top-level directory has pair: <joe, #>. joe has pair <abcde, #>, etc.

How many disk I/O's to access first byte of file1?

1. Read in file header for root (always at fixed spot on disk).
2. Read in first data block for root.
3. Read in file header for joe
4. Read in first data block for joe.
5. Read in file header for abcde
6. Read in first data block for abcde.

7. Read in file header for file1
8. Read in first data block for file1

Current working directory: short cut for both user and system. Each address space stores file index for current directory. Allows user to specify relative filename, instead of absolute path (if no leading "/").

Thus, to read first byte of file, just last 4 steps above.

How can this possibly be efficient? *Caching* (of course!)

18.3.4 Not really a hierarchy…

Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph.
UNIX does this through the concept of "links". Two flavors:

1) **Hard links** – different names for the same file.
- All names are equally valid
- Implemented by having multiple directory entries point to same inode
- Question: how to know when you can delete a file?
- Can only be used with non-directory files.
- Can't cross file system boundaries

2) **Soft links –** "shortcut" pointer to other file
- Implemented by simply storing the logical name of the actual file
- Fewer restrictions: can point to directories, cross file systems, etc.
- No protection from deletions – may be "dangling" or point to wrong file!
- Cycles are possible – Question: how does system avoid infinite loops when following a path?

18.4 File Caching and Related Topics

- Use caching and prefetching to achieve good performance in a file system.
- Three key ideas:
  - Caching of disk blocks read into memory
  - Prefetching of disk blocks expected to be needed soon
  - Delayed writes

18.5 Caching

Key idea (as usual): exploit locality of use in file systems by caching disk blocks in memory.

Use an LRU replacement scheme.

Easy to do since we can afford the overhead of maintaining timestamps for each disk block being cached.

Advantages:
- Works very well for name translation,
- Works well in general as long as memory is big enough to accommodate a host's working set of files.

Disadvantages:
- LRU loses when some application scans a big enough part of the file system, thereby flushing the cache with data that is used only once; for example
```
find . –exec grep foo {} \;
```

Some systems allow applications to provide the file system with hints about which replacement policy to use. For example, an application might indicate that it will not use a file more than once. The file system would then know to discard any disk blocks of the file once they have been used.

Question: how much memory should the OS allocate to the file system cache vs. the VM paging store?

If we allocate too much memory to the file system cache then we won't be able to run many applications in parallel.

If we allocate too little memory to the file system cache then many applications may run slowly.

Solution: let the boundary between the two vary so that the disk access rates for paging and file access are balanced.

18.6 Prefetching

Key idea: exploit the fact that the most common form of file access is sequential by prefetching subsequent disk blocks ahead of the current read request (if they're not already in memory).

How much should one prefetch?
- Request too many blocks and we start imposing unnecessary delays on concurrent file requests by other processes.
- Request too few blocks and too many seeks (and rotational delays) will occur among concurrent file requests.

18.7 Delayed Writes

Key idea: Batch writes to optimize disk scheduling and allocation.

Unix systems use a 30 second write-behind policy:

- Writes only copy data from a user process to kernel disk block buffers,
- Dirty disk block buffers are only flushed to disk once every 30 seconds.

Advantages:

- Disk scheduler can efficiently order lots of requests.
- Disk allocation algorithm can be run with correct size value for a file.
- Some files need never get written to disk! (E.g. temporary scratch files written in **/tmp** frequently don't exist for 30 seconds.)

Disadvantages:

- What if the system crashes before your file has been written out? Worse yet, what if the system crashes before a directory file has been written out?

Topic of the next lecture!

18.8 Protection and Access Control

Use access control lists and capability lists to control access to resources such as files.

18.9 Protection

Goals:

- Prevent accidental and maliciously destructive behavior.
- Ensure fair resource usage.

A key distinction to make: policy vs. mechanism.

Mechanism: how something is to be done.
Policy: what is to be done.

18.10 Access Control

18.10.1 Domain structure

Access/usage rights associated with particular domains

Example: user/kernel mode => two domains

Unix:

- Each user is a domain
- *Super-user* domain
- Groups of users (and groups)

18.10.2 Types of access rights

What kinds of access rights do we need for files?

- Read
- Write
- Execute

For directories:

- List
- Modify
- Delete

For access rights themselves:

- Owner (I have the right to change the access rights for some resource)
- Copy (I have the right to give someone else a copy of an access right I have)
- Control (I have the right to revoke someone else's access rights)

18.11.3 Access control matrix

Conceptually, we can think of the system enforcing access controls based on a giant table that encodes all access rights held by each domain in the system.

For example:

	File 1	File 2	File 3	Dir 1	Dir 2	…
User A	rw	r	rwx	lmd	l	…
Group B		r	rw		lm	…
…	…	…	…	…	…	…

The access control matrix represents the policy we want to enforce.

There are two principal means of providing a mechanism to do so:

- Access control lists
- Capability lists

Access control lists: keep lists of access rights for each domain with each object.

```
File3:
     User A: rwx
     Group B: rw
     ...
```

Capability lists: keep lists of access rights for each object with each domain.

```
User A:
     File1: rw
     File2: r
     ...
```

Which is better?

- ACLs allow easy changing of an object's permissions.
  Example: add Users C, D, and F with rw permissions.
- Capability lists allow easy changing of a domain's permissions.
  Example: you are promoted to system administrator and should be given access to all system files.

Combination approach:

- Objects have ACLs
- Users have capabilities, called "groups" or "roles"
- ACLs can refer to users or groups
- Change permissions on an object by modify its ACL
- Change broad user permissions via changes in group membership

18.10.3 Revocation

How does one revoke someone's access rights to a particular object?

Easy with ACLs: just remove entry from the list. Takes effect immediately since the ACL is checked on each object access.

Harder to do with capabilities since they aren't stored with the object being controlled:

- Not so bad in a single machine: could keep all capability lists in a well-known place (e.g. the OS capability table).
- Very hard in distributed system, where remote hosts may have crashed or may not cooperate. (This topic will be covered in greater detail in a future lecture.)
- Various approaches possible:
  - Put expiration dates on capabilities and force reacquisition.
  - Put epoch numbers on capabilities and revoke all capabilities by bumping the epoch number (which gets checked on each access attempt).
  - Maintain back pointers to all capabilities that have been handed out. (Tough if capabilities can be copied.)
  - Maintain a revocation list that gets checked on every access attempt.