Unix Filesystem Organization

"Old" (Original) file system

In the original Unix file system, Unix divided physical disks into logical disks called *partitions*. Each partition is a standalone file system. We will use the term "file system" when referring to a single partition.

Each disk device is given its own *major device number*, and each partition has an associated *minor device number* which the device driver uses to access the raw file system.

The major/minor device number combination serves as a handle into the device switch table. That is, the major number acts as an index, and the minor number is passed as an argument to the driver routines so that they can recognize the specific instance of a device.

Each filesystem contains:

1. a *boot block* located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system.

   Typically, the first sector contains a bootstrap program that reads in a larger bootstrap program from the next few sectors, and so forth.

2. a *super block* describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the inode number of the root directory, magic number, etc.

3. a linear array of *inodes* (short for “index nodes”). There is a one to one mapping of files to inodes and vice versa. An inode is identified by its “inode number”, which contains the information needed to find the inode itself on the disk

   Thus, while users think of files in terms of file names, Unix thinks of files in terms of inodes.

4. *data blocks* blocks containing the actual contents of files

```
----------------------------------------------------------------
|          |            | | | | | | | |        |  |  |  |      |
| B. B.    | S. B.      | Inodes  | | | ...    |  Data Blocks  |
|          |            | | | | | | | |        |  |  |  |      |
----------------------------------------------------------------
```

An inode is the "handle" to a file and contains the following information:

- file ownership indication
- file type (e.g., regular, directory, special device, pipes, etc.)
- file access permissions. May have setuid (sticky) bit set.
- time of last access, and modification
- number of links (aliases) to the file
- pointers to the data blocks for the file
- size of the file in bytes (for regular files), major and minor device numbers for special devices.

An integral number of inodes fits in a single data block.

Information the inode does not contain:

- path (short or full) name of file

Example

Look at `/cs/bin/`

```
< wpi /cs/bin 1 >ls -l
total 192
drwx------   2 mvoorhis csadmin     4096 Jan 16  2001 archives/
-rws--x---   1 root     771        32768 Jan 18  1999 csquotamgr*
-rwx------   1 csadmin  csadmin      162 Jan 12  1998 genQuota*
-rwx------   1 csadmin  csadmin       46 Feb 16  1998 generic*
drwxrwx---   2 mvoorhis csadmin     4096 Oct 29 10:23 gredStuff/
-rwx------   1 mvoorhis 1067         672 Jan 20  2000 list1*
-rwx------   1 mvoorhis 1067         859 Jan 20  2000 list2*
-rwx------   1 csadmin  646          140 Jan 10  2000 reclaim*
-rwxrwx---   1 csadmin  csadmin     1635 Sep 26  1995 stp_create_system.pl*
-rwxrwxr-x   1 csadmin  csadmin      725 Sep 26  1995 stp_default_system.pl*
-rw-rw-r--   1 csadmin  csadmin      114 Feb 10  1995 stp_setup
drwx------  14 mvoorhis csadmin     4096 Oct 30 14:57 tDir/
-rwsr-xr-x   1 mvoorhis 1067      114688 Nov  8 10:10 turnin*
drwxr-xr-x   2 root     771         4096 May 26  1999 utility/
```

Internally, Unix stores directories in files. The file type (of the inode) is marked "directory", and the file contains pairs of name/inode numbers.

For example, when a user issues *open("/etc/passwd", ...)* the kernel performs the following operations:

1. because the file name is a full path name, find the inode of the root directory (found in superblock) and search the corresponding file for the entry "etc"
2. when the entry "etc" is found, fetch its corresponding inode and check that it is of type directory
3. scan the file associated with "/etc" looking for "passwd"
4. finally, fetch the inode associated with *passwd*'s directory entry, verify that it is a regular file, and start accessing the file.

Note: What would the system do when opening "/dev/tty01"?

Eventually, the system would find the inode corresponding to the device, and note that its file type was "special". Thus, it would extract the major/minor device number pair from the length field of the inode, and use the device number as an index into the device switch table.

Getwd()

How to get string of current directory? Have only the inode of the current directory.

```
get current inode
while (inode != root inode) {
    get inode of parent from ..
    search parent's directory file to match our inode number
```

Where should a file's data blocks be physically located?

- to improve performance, we might want to place a file's data blocks in contiguous sectors on disk. However, this leads to inefficiencies in allocating space, or forces the user to specify the size of the file at creation time.

The Unix file system allocates *data blocks* (blocks that contain a file's contents) one at a time from a pool of free blocks. Unix uses 4K blocks. Moreover, a file's blocks are scattered randomly within the physical disk.

Inodes include pointers to the data blocks. Each inode contains 15 pointers:

- the first 12 pointers point directly to data blocks
- the 13th pointer points to an indirect block, a block containing pointers to data blocks
- the 14th pointer points to a doubly-indirect block, a block containing 128 addresses of singly indirect blocks
- the 15th pointer points to a triply indirect block (which contains pointers to doubly indirect blocks, etc.)

```
-------------------------------
| | | | | | | | | | | | | | | |
-------------------------------
 | | | | | | | | | | | | | | |--------------------------
      data blocks        | |-----------|                |
                         |             |                |
                       -----         -----            -----
                       |   |         |   |            |   |
                       -----         -----            -----
                        |||           |||              |||
                        data         -----            -----
                                     |   |            |   |
                                     -----            -----
                                      |||              |||
                                      data            -----
                                                      |   |
                                                      -----
                                                       |||
                                                       data
with 4K blocks:
direct 12x4K = 48K
indirect 1024x4K = 4MB
```

```
double indirect 1024x1024x4K = 4GB
triple indirect 1024x1024x1024x4K = 4TB
```

Advantages:

- data in small files can be accessed directly from the inode That is, one read operation fetches the inode, and another read fetches the first data block.
- larger files can be accessed efficiently, because an indirect block points to many data blocks
- disk can be filled completely, with little wasted space (ignoring partially-filled blocks)

Disadvantages:

- because inode information is kept separately from data, access of data often requires a long seek when file is initially accessed
- inodes of files in a common directory not kept together, leading to low performance when searching directories
- original file system only used 512-byte blocks, an inefficient transfer size
- data blocks of a file are not stored together, leading to poor performance when accessing files sequentially.
- free list quickly becomes scrambled increasing overhead of finding free blocks (seek for each new block)
- original file system used as little as 2% of the available disk bandwidth

The Berkeley Fast File System

The Berkeley Fast File System used the following principles to improve the performance (and reliability) of the file system:

- duplicate the super block, so that it can easily be recovered after a disk crash
- use a large block size to improve throughput
- add the block size to the superblock, so that different file systems could be accessed using different block sizes
- store related data blocks within *cylinder groups*, one or more consecutive cylinders on the disk. Blocks within in a cylinder group can be accessed with only a small seek (if any)
- because large blocks leads to fragmentation, small files (or the remaining bytes of a file) should be stored in *fragments*, where an integral number of fragments (e.g., 4 or 8) fits in a single block. Use 4K/512 or 8K/1K block/fragment size combinations. Inode stores 8-bit mask for fragment use in last data block of the file.

A data structure within each cylinder group contains status information about the blocks stored within that group:

1. a bit map of blocks and fragments indicates which blocks and fragments are free
2. a list of inodes within the cylinder group (why?)
3. duplicate copy of the superblock (stored on a different platter for each cylinder group. why?)

When allocating space, the fast file system uses a *global policy* to determine where to place new directories and files. For example:

- place inodes of files in the same directory in the same cylinder group (makes programs like *ls* faster)
- place new directories in a cylinder group that has a higher than average number of free inodes and the smallest number of directories already in it
- try to place all data blocks for a file in the same cylinder group
- move to a new cylinder group when a file exceeds 48kb, and every megabyte thereafter.

The fast file system also uses a *local policy* to allocate blocks at the lower levels. For instance:

- when adding a data block to an existing file, pick the next block to be rotationally closest
- try to allocate blocks out of the same cylinder, before looking at other blocks in the same cylinder group

The new file system increased the throughput to as much as 30% of the raw bandwidth.

Linux ext2fs

The *2nd extended file system*. Same standard file system as Unix.

Similar to the Berkeley fast file system, but does not use fragments. Rather it uses smaller block sizes (1K, but can be 2K or 4K).

Tries to cluster disk blocks so that a single I/O request can read multiple blocks.

Modern disk technologies pack sectors onto disks at different densities—Linux uses variable size block groups (like cylinder groups in BSD FFS).

Allocation:

- Tries to allocate data blocks in same group as inode.
- Tries to allocate nondirectory inodes in same group as parent directory
- Tries to allocate directory inodes in different group than parent directory

Also has a *proc* file system to allow access to process information through the file system interface.

Also supports other file systems such as FAT and NTFS.

File Mounting

When the system initially boots, the only file system Unix knows about is the root partition from which the system was booted. A special system call:

mount(special, path_name, options)

mounts the filesystem given by *special* at the point *path_name* in the root filesystem, thus allowing multiple file systems to be merged into a single global tree.

Internally, the kernel maintains a *mount table* that keeps information about the mounted file systems. Each entry in the table contains:

- the device number of partition that has been mounted
- a pointer to the buffer containing the super block for the file system
- a pointer to the root inode of the file system
- a pointer to the inode of the directory in which the file system is mounted (e.g., a pointer to the parent directory)

As the kernel is translating a path name, it consults the mount table as needed.

In Memory Data Structures

The kernel maintains a system-wide *file table* that describes open files. Each entry in the file table contains:

- the read/write mark that indicates from where the next data block is to be read
- a pointer to an entry in the *active inode table*, so that the access times can be modified efficiently

Finally, each process maintains a *user file table* that describes the files opened by the process. Entries in the user file table point to the system-wide file table.

Thus, a process can have its own private read/write mark (the default when a file is initially opened), or it can share a read/write mark (as is the case when a new process is created via *fork*).

Caching

Unix relies heavily on caching to improve performance. It keeps in memory:

- recently accessed disk blocks
- the inode of each open file in the system
- a cache of recent name-to-inode mappings
- a directory offset cache. If a process requests a file name in the same directory as its previous request, the search through the directory is started where the previous search ended. Improves the performance of applications that read all files in a directory (like *ls*).