UNIX File System

The UNIX file system has a hierarchical tree structure with the top in **root**.

- Files are located with the aid of directories.
- Directories can contain both file and directory identifiers.
- The user identifies files with absolute or relative **path names**.
- Example on absolute names: /home/terry/notes/apr22.txt
- Each user has a **login directory**. The user can create his own subdirectories within the login directory.
- The system has information about a users *current directory*. At login, the home directory is set as *current directory* but this can be changed with the command *cd*.
- The operating system uses inodes as the internal name for files. An i-number is an index in a table of inodes.
- An entry in the inode table contains complete information about a certain file. A directory only contains a translation from *path nam*e to *i-number*.

UNIX File System

Every directory always has two standard entries.

The name "." in each directory refers to the directory itself.

The name ".." in each directory refers to the parent directory.

The same file may be present in several directories. This is called a **hard link**.

A file that is removed do not disappear until the last link to it is removed.

External Devices

- External devices may be reached via **special files** in the file system.
- By convention the special files are located in the /dev directory.
- Internally, the kernel identifies external devices by it's **device number**.
- A device number consists of a **major device number** and a **minor device number**.
- The *major device number* identifies the device driver that handles the device.
- The *minor device number* is used by the device driver to distinguish between different logical units handled by the same driver.
- Every special file have a unique device number and is only an interface to the device driver.
- From the user point of view, special files are read and written in the same way as normal files.

Mountable File Systems

A UNIX system may have several physical file systems, but only one file system tree.

- The *mount* system call is used to connect the different file systems to a single tree.
- The *mount* system call takes two parameters: The name of a directory in the file system and a special file structured as a file system.
- The effect of mount is that the file system in the special file is added as a subtree under the specified directory.
- The system administrator can use the command *mkfs (make file system)* to create an empty file system at a special file.
- The only limitation with mounted file systems is that it is not possible to create hard links across mount boundaries.

UNIX File Permissions

- Every file has an *owner*, usually the user that created the file.
- Every file also belongs to a *group*. The files owner may change the group but a file may only belong to one group at a time.
- The system administrator may give a user membership in some groups.
- Every file has 10 protection bits. Nine of these bits specifies independent read, write or execute access to the owner, group and others.
- The tenth bit is the set-user-id bit. If a file with the set-user-id bit set is executed, the effective user id during the execution of the file is set to the owner of the file.

File System Topology

- **Tree** - Simple and rather general method, but prevents sharing of files.
- **Acyclic graph** - Allows sharing of files. Cycles can be prevented by allowing links to files only (not to directories).
  - → To assure that a file is removed only than the last link to it is removed, *reference counters* are used.
- **Cyclic graph** - Most general topology, but difficult to implement.
  - → May create indefinite looping in recursive algorithms.
  - → When deleting a file, the *reference count* may not be 0, even then there is no way to refer to a file or directory. (Requires garbage collection).

Block Allocation Methods

A method is needed to keep track of which blocks are allocated to a certain file.

- **Contiguous allocation** - All blocks in a file are placed in a sequence on the disk. Give high bandwidth but difficult to locate free space.
- **Linked allocation** - Each block may be located anywhere on the disk and every block contains the address to the next block. Easy to locate free blocks, but very inefficient for sequential access.
- **File allocation table (FAT)** - The disk keeps an area reserved for the *file allocation table* (FAT). The table has an entry for every block in the file system. Free blocks are marked with 0. For blocks belonging to a file, every entry in the FAT gives the entry for the next block in the file. The last block in every file contains an end marker.
- **Indexed allocation** - The block pointers are kept in special index blocks. A 512 byte block contains pointers to 128 data blocks. Several variants exists.

Disk Free Space Management

The file system also need to keep track of the free blocks at the disk.

- **Linked list** - All free blocks are linked together by a link field in each block. Inefficient as every new block needed requires a disk operation.
- **Linked list of address blocks** - Every address block have pointers to n-1 free blocks and a pointer to the next address block. Gives access to n-1 free blocks per disk access. Furthermore no pointers need to be stored in the data blocks.
- **Bit vector** - Every disk block corresponds to a bit in the bit vector. Free blocks are represented by 1 and used blocks by 0. Can be used to locate blocks at a wanted position on the disk.

UNIX File System Implementation

Files and directories are described by a data structure called an *inode*.

When the file system is created, a fixed number of *inodes* are created at a known location on the disk. An *inumber* have a 1-1 mapping to a disk address.

An *inode* is allocated for every new file or directory.

An *inode* contains:

- A type code that tells if the inode describes a file, directory, special file or is free.
- Time for creation, last access and last change.
- Id and group id for the owner of the file.
- Protection bits.
- File size.
- Number of links to the file (reference counter).
- Addresses to data blocks.

Placement of Data Blocks

- The block size in the original UNIX file system was 512 bytes.
- A block address on a disk usually was 32 bits (today 48 bits). Thus, a 512 byte block fits 128 block pointers.
- The address to the first 10 blocks in a file is stored in the inode.
- In addition to these blocks , the inode also has pointers to an indirect block, a double indirect block and a triple indirect block.
- This gives a maximum file size of $(10 + 128 + 128^2 + 128^3) \cdot 512 = 1$082 201 088 bytes

Open

- The purpose of open is to translate from a path name to an inumber by searching the directories.
- If the specified file exists, a *file descriptor* and a *file struct* is allocated.
- The *file descriptor* points to the *file struct* that points to the cached *inode*.
- The *file struct* contains the position pointer that indicates the current read/write position.
- Open files inherited at *fork* gets the same *file struct* as it's parent process and thus the read/write position is shared between child and parent in this case.
- Open is a time expensive operation since it requires reading of many disk blocks from different locations on the disk.

When a file is open, read or write commands can locate the inode and the disk blocks via the file descriptor.

File Creation and Removal

- The creation of a new file consists of allocating and updating an inode and creation of a directory entry in specified directory.
- To create a *hard link* means to to create a new directory entry and incrementing the reference counter in the inode.
- To remove a file with several hard links means that the directory entry is removed and the reference counter in the inode is decremented.
- If the reference counter was decremented to 0, it was the last link to the file and it's data blocks and inode is freed.

Symbolic Links

- Contrary to *hard links* which are implemented at the inode level, *symbolic links* are implemented at the file system tree level.
- A *symbolic link* is a file of type LINK that contains the path name to another file or directory.
- If the system finds a symbolic link during translation of a path name, the path name in the link file is concatenated with the remaining components in the path name under translation. The translation process continues with the new path name.
- Contrary to *hard links*, *symbolic links* may cross mount points and may point to directories.
- Infinite recursion is prevented by limiting the number of directories in a path name.
- Symbolic links do not create disk allocation problems, because the symbolic links are not affected if a file is removed. It is up to the user to remove invalid symbolic links.
- Symbolic links was first implemented in BSD4.2

Caches

Caches are used to improve the performance of the file system.

inode_cache - Copies of the latest referenced inodes are kept in a the inode cache in primary memory.

Block_buffer_cache - Read and write disk operations transfer data between the disk and a *block buffer cache* in primary memory. From the buffer cache the data is copied to or from the process address space.

- The buffer cache saves disk accesses when repeated operations are to the same block.
- When writing, the write system call completes when the data is written to the buffer cache. This makes it possible to use a disk scheduling algorithm to optimize the data transfers to the disk.
- The drawback with the *buffer cache* is that if the system crashes, all data in the cache is lost unless it has already been written to the disk.
- To reduce the data loss in case of a system crash, the **sync** system call is periodically executed (usually every 30 seconds) to write unsaved data in the *buffer cache* to the disk.

Unified Buffer Cache

- Caching of file data is in many systems done both by the file system and the virtual memory.
- Pages from executable files are always stored in the page cache.
- When memory mapped files are used data from the same file may be stored both in the *page cache* and in the file system *buffer cache*. This leads to double buffering (fig. 11.11) and may lead to inconsistent caches.
- In most newer operating systems the *page cache* and the *block buffer cache* have been unified to a single cache.