Local filesystems

The operations defined for local filesystems are divided in two parts:

1. Common to all local filesystems are hierarchical naming, locking, quotas attribute management and protection.
2. The other category is concerned with organization and management of data on the storage medium.

Vnode-operations for hierarchical filesystem operations are shown in table 8.1

Inode

- The central data structure in the Unix filesystem is the inode (fig. 8.1).
- For open files, the inode is cached in main memory.
- In addition to the inode there is a vnode (fig. 8.3)
- To be able to quickly locate cached inodes, a hash table is used that hash on $<$*inumber,devicenumber*$>$ (fig. 8.4).

Open

- The open system call translates a path name to a vnode.
- The translation is performed by calling the VFS-level *lookup()* routine for every component in the name.
- If the name is found in the namecache, the vnode is returned otherwise *ufs_lookup()* is called.

Ufs_lookup:

- Look up the name in the directory and return the inumber.
- Look up the inumber in the hash table. If hit return otherwise:
  - → Allocate a new vnode
  - → Locate the disk block for the inode and read it into a kernel buffer.
  - → Allocate a new inode in the inode-cache and copy the buffer to the new cache entry.
  - → Connect the vnode and the inode.

Directories

- Disk space for directories is allocated in units called *chunks*.
- The size of a *chunk* is chosen so that it can be written to disk in one operation (requirement to make disk operations atomic).
- A chunk is divided in directory *entries* of variable size (fig. 8.6).

A directory *entry* consist of:

1. Inumber (0 if empty chunk)
2. The size of the entry in bytes
3. The type of entry (file, directory, etc.)
4. The length of the filename in bytes
5. A variable length null-terminated filename (max 255 characters)

Directory operations

User programs can read chunks of a directory with the system call *getdirentries*.

However, the common way to read directories is the following routines:

opendir() Open a directory. Returns a pointer to a DIR struct that is used by the other directory operations.

readdir() Returns the next directory entry.

closedir() Closes an open directory.

rewinddir() Repositions the read pointer to the start of the directory.

telldir() Returns information about the current read position.

seekdir() Sets the read position to a position previously obtained with *telldir()*.

Finding of Names in Directories

- System calls such as open and stat use the *lookup()* routine to find a name in a directory.
- In principle, the directory is sequentially searched until the name component is found or the end of the directory is reached.
- To improve performance the namecache is used for both positive and negative caching.
- A common operation is *'ls -l'* that reads the directory sequentially and calls *stat* for every component.
- To improve the performance of this command the kernel saves the directory offset of the last successful lookup. The next lookup in the directory is started at this offset.
- This optimization have a drawback in that each cache miss will cause the directory to be searched twice (once from the middle to the end and once from the beginning to the middle).
- To improve the searching of big directories, FreeBSD 5.2 dynamically builds a hash table in main memory the first time a directory is searched.

Pathname Translation

- The translation from path name to inumber requires alternately reading of directories and inodes.
- The translation is illustrated by fig. 8.5 and fig. 8.7

File Flags

Chflags() and **fchflags()** can set flags in a 32-bit user-flags word in the inode.

The following flags are defined:

- *Immutable* - The file cannot be moved, changed or deleted.
- *Append_only* - as *immutable* but data can be appended to it.
- *not_needing_to_be_dumped*

Security levels are defined as follows:

-1. *Permanently insecure mode* - Always run system in level 0 mode.

0. *Insecure mode* - Immutable and append-only flags can be turned off.

1. *Secure mode* - Immutable and append-only flags cannot be cleared. Special files for mounted filesystems and /dev/mem and /dev/kmem are read-only.

2. *Highly secure mode* - As secure mode, but all special files for disks are read-only.

That /dev/mem is not writable in secure mode has the side effect that a X-server do not work because it uses mmap on /dev/mem for communication with the graphics card.

Safety Aspects

- In safe modes, the security level can only be decreased by the init process (when rebooting into single user mode).
- The result is that the security level can only be decreased by somebody with physical access to the machine or the system console.
- Typically programs like *su* and *login* are marked immutable.
- The append-only flag is normally used for system logs.

The Local Filestore

- Every filesystem is stored on a disk partition.
- The data storage is handled by a part of the filesystem code that is separate for every filesystem.
- The following storage methods existed in 4.4BSD:
  - → FFS - The fast Berkley filesystem
  - → LFS - The log based filesystem
  - → MFS - The memory-based filesystem
- In FreeBSD 5.2 LFS is not implemented (because the internal interface has been changed).
- The memory-based file system is implemented by *memory disk* in FreeBSD.
- The data storage code manages a flat namespace where the names are inumbers or something similar.

Data Storage - Overview

- A number of interface routines are defined for communication between the hierarchical level and the data storage level (fig. 8.11).
- All these operations are called via the vnode interface.
- The real name of the routines is filesystem dependent (for example ffs_valloc).
- Two operations are defined for creating/releasing objects (files, directories):

Valloc() Create a new object (i.e. allocate an inode)

Vfree() Release an object (i.e. an inode)

Read/Write-Operations

Operations for manipulating an existing object:

Vget() Read inode from disk and if needed allocate cached inode+vnode

Read() Copy data from the disk to the process according to a description in an *uio* struct

blkatoff() Copy data from the disk to a kernel buffer

Write() Copy data from a process to the disk

Fsync() Write all cached data associated with an object back to disk

Truncate() Decrease or increase the size of an object

Handling of Blocks

- The system calls read/write can address single bytes.
- Disks can only address blocks (sectors).
- Thus, I/O operations to disks must be translated to a sequence of blocks.

Writing an (arbitrary) sequence of bytes to the disk is illustrated in fig. 8.30:

1. Allocate a buffer
2. Determine the disk address for the data
3. Read the needed block from the disk to a kernel buffer
4. Copy data from the process to the correct part of the buffer
5. Write the complete buffer to the disk

Wasted Space and Block Size

- Data transports to a disk have to use large blocks in order to obtain a high average speed on the data transports.
- Unfortunately, big blocks may give high fragmentation losses (Table 8.12)

UFS2

UFS2 is a reimplementation of UFS.

The following additions/changes is included in UFS2:

- Extended attributes
- 64 bit file pointers to support files bigger than 1 terabyte
- Somewhat more dynamic allocation of inodes than in older filesystems
- access control lists
- Many filesystems now support so called “*extended attributes*”
  - → “*Extended attributes*” can be used to store extra information that is not part of the data in the file.
  - → Such extra information may be name of author or character encoding.
  - → In UFS2 the kernel uses *extended attributes* to store access control lists.

UFS2 - inodes

- Increasing the size of a file pointer to 64 bits had the side effect that 128 bytes was not enough to store an inode anymore.
- The inodes in UFS2 are 256 bytes big.
- the Inodes in UFS2 are extended with two pointers to blocks of extended attributes.
- The formate of extended attributes is shown in fig. 8.2

UFS2 - allocation of inodes

- Older filesystems allocated a fixed number of inodes in each cylinder group when the filesystem was created with *mkfs*.
- This lead to a certain waste of memory space because some extra space needed to be allocated to inodes as a safety margin.
- In UFS2 only 64 inodes are allocated for each cylinder group.
- However, extra space for inodes is reserved in each cylinder group and cannot be used for data blocks until the filesystem runs out of space for data blocks.
- If the preallocated inode space is insufficient, an extra block for inodes is allocated.

Memory Based Filesystems

- Disk memories implemented with RAM memory, so called RAM-disks have been available for a long time.
- These memories have better access time and bandwidth than normal disk memories but the cost per stored byte is higher.
- A problem is that RAM-disks loose the memory at a power failure and for this reason they are only suitable for temporary storage.
- In principle permanent storage is possible if battery backup is used, but this is expensive.
- Temporary storage with short access time is useful for compilers and other programs that store large amounts of data for a short time.

RAM-Disks

- Due to the rather high price, RAM-disks are seldom a good alternative because in most cases the improvement is bigger if the money is spent on extra primary memory.
- An alternative is to simulate the RAM-disk in software.
- If the program reserves memory for use only by the RAM-disk this leads to poor resource utilization.
- The memory will give better improvement if it is used by the buffer cache.
- The 4.4BSD memory based filesystem avoided the problem by building the filesystem in virtual storage.
- Memory based file systems are typically used for /tmp

Memory Based filesystem in FreeBSD5

- FreeBSD has a *md* (memory disk) driver that implements a memory based virtual disk.
- New *memory disk* units are created with the *mdconfig* command.
- Then a filesystem is created on the unit with *mkfs* and it is mounted (for example on /tmp).