UNIX FILE SYSTEM

NEZER J. ZAIDENBERG

AGENDA

- UFS
- EXT2
- EXT3,EXT4 – NEW FEATURES
- /PROC FILE SYSTEM
- VIRTUAL FILE SYSTEM

REFERENCES

- UNIX FILESYSTEMS – S. PETE
- ADVANCED PROGRAMMING IN THE UNIX ENVIRONMENT CHAPTERS
- UNDERSTANDING LINUX KERNEL

WHAT PURPOSE FILE SYSTEM SERVE

- Manage used and free blocks on the disks
- Manage multiple files
- Manage multiple devices
- User permissions
- And more (wear leveling, links, devices)

Something about physical disks drives and logical partiton

- Hard drive - where data is kept
- JBOD – just bunch of disks. (several hard drives that do nothing really special)
- RAID – Redundant array of independent disks = several disks that operate in special way to improve read/write/reliability performance usually at cost of disk space or reliability(for example, mirroring = using 2 disks data is saved to both disks. Double read performance, much better availability, same write speed, data takes twice the space. Striping = using two volumes where data is saved on both. Doubles performance of read and write but reliability is damaged.)
- Hard drive may have multiple partitions each is treated as a separate disk for most OS related issues.
- Today high end storage project (big iron from vendors such as IBM (Shark/ESS), EMC (Symmetrix, Clarion), HDS, SUN(STK) etc.) have many physical drives that are usually not visible to the User. Instead the machine exports several logical partition, each may be mapped to one or several disks.
- In this course when I use the term disk I refer to logical partition

Hard drives

- Mechanical – What most of you have in your PC
  - Spinning heads over metal plates
  - Slow compared to memory
  - Slow seek time
  - Relatively fast sequential read
  - Tend to be unreliable compared to other hardware components
  - In this course I assume standard hard drives
- Solid State – New technology –
  - Uses solid state technology
  - Slow compared to memory
  - Seek time = identical to sequential read time
  - Relatively reliable
  - Require wear leveling (some solid state disks include wear leveling in hardware)
- Other types of hardware
  - CDROM – fast sequential read, very slow seek, ROM
  - Tapes drive – usually don’t have “file system” – very very slow seek

Types of file system

- Physical file system – we take a disk (or partition!) and we want to arrange files on it.
- Logical file system – file system that demonstrate some logical state of the system such as /proc /dev or /sys (those file systems demonstrate running processes or devices detected by the system or system info.) – Those file system don’t deal with real file and are beyond scope. (but we acknowledge their existence)
- Virtual file system – we take several physical and logical merge them into one file system.

Some definition

- Disk (n), partition(n) – where I put the files on. (I don’t care about type of disk or disk/partition semantic. I also ignore for now network file system logical file system etc.)
- Mount(v) – the action in which I make a file system usable by the system (occur automatically in windows and some unices)
- Unmount (v) – making the file system no longer usable to the system – for example if I want to eject it
- File (n) – unless noted otherwise I would refer to a real file! (not socket, pipe etc. those are not written on disk)

Disk based file system

UFS (early design + Not very accurate)

- UFS was first available starting version 5-version 7 Linux from AT&T. (early 1980's)
- UFS (UNIX file system) is the modern name of the Berkeley fast file system (FFS)
- UFS was first described in a USENIX letter from 1984 titled "A fast file system for UNIX" (by Mckusick et al)
- UFS derived file system exist and improved in most modern UNIX box. (indeed the Linux ext2 file system is almost direct extension.)
- Today UFS implementation (found in Solaris for example) have many additional features, beyond our scope. Here we describe the some of the basic 1984 implementation. (It is easier to understand UFS first then ext2)
- This review – which by no means attempts to be historically accurate or describe any specific version in any way – is helpful to understand the idea's that UNIX file system implement. (note that not all ideas were introduced in one version and with new ideas also came new optimization concepts that complicate things that I left out)
- I ignore (as beyond the scope or no longer relevant) many consideration that were made regarding physical positioning of the data on the disks.

Basic building block of ufs

- Block and fragments
- Inode
- Superblock

Block

- Place to store data.
- 512 bytes (version 7) and 4096 bytes and up (BSD 4 versions) (I ignore fragments intentionally.)
- Each block is identified by unique address it can be used or not
- Files are saved on discrete number of blocks. (and its file either use a block or not)
- Each block is identified by unique
- Nothing smart here

Inode

- Reference to a file
- Points directly and indirectly to blocks
- Contain the OS info on a file
- Does not contain the file name
- Each Inode is identified by unique number

The inode strcture

- Permissions, user id, group id etc.
- Timers
- Everything we can get in stat/fstat
- Direct referances to block that the file is made of
- Indiret (reference to block containing references) reference to blocks
- Indirect^2 (reference to references to references) reference to block
- Etc. (modern UNIX system have indirect^4 references)

The super block

- The file system catalog
- General information about the file system such as
  - Number of Inodes
  - Number of blocks
  - Number of used and free inodes and blocks

Maximum file size?

- Maximum file size=total number of blocks that we can point to.
- Derived from the number of indirect levels of pointing to blocks
- In most cases it is practically unlimited in modern UNIX boxes (but old versions had limit of 2GB to couple of terabytes)

Filenames and directories

- A directory type of file is a file containing list of i-Nodes (specified by I-node number) and names that are contained in the directory
- (The directory can contain other directories)
- The I-Nodes are the files that are contained in the directory
- For each I-Node we have the name that will be used to access it. (A file with several hard links can have several names)
- Permissions for directory we have = read permission = I can read the directory (ls(1)) write = I can create files in the directory (touch(1)) execute = I can cd into the directory

Hard links

- Hard links are two I-Nodes pointing to the same file
- Usable when two users want to work on the same file (data) each from his own directory
- Also when one binary is used (such as bzip2, gcc) and decides based on how it is called what to do (check argv[0] is it bunzip2? Is it gcc? g++?)
- When hard link is deleted the file is not deleted (but the inode count on the inode is reduced by 1)
- When the last (and only) Inode is deleted the blocks are marked free

Soft (Symbolic) links

- Windows : Short cuts
- Those files contain a path where another file is located
- When UNIX reads the file it moves to the other file and operate on it. (so open (unless op on symbolic links actually calls open on the file it points)

Broken links

- Symbolic links are not counted in the I-node
- That means that if the file the symbolic link points is deleted we have “broken link”
- Homework - not for submission – create a broken link

Ideas from Berkeley

- Some ideas for improvement was added in UFS
  - Blocks were too speed on the disks that caused many seek for the next block and low throughput. Therefore, UFS has larger block size (with continuous data)
  - Fragments were added to support partial usage of blocks
  - Super block is now replicated several times on the disk (stability and reliability as well as performance – faster seek time for nearest superblock)
  - Many new features are added (but I didn’t made distinction)

Fragments

- In an effort to reduce waste, and maintain low seek times, UFS allowed blocks to be broken to fragments to store odd ends of a file
- When new data was appended to files with fragments the new data was either filled in the fragment block (filling the block) or copied to a new block.

Catalog based file system

- Most file system that resides on disk are catalog based.
- There exist a catalog (such as the superblock) with info regarding the file system
- The catalog is in specific place
- Catalog based file system can be mounted easily (one only needs to read the file system catalog and know what's up)
- Catalog based file system are not suitable for devices that require wear leveling (The catalog is written to and accessed to much more often then other parts of the filesystem)
- Catalog based file systems are suitable for mechanical hard drives are less suitable for Solid state devices (some solid state disks implement wear leveling internally so catalog based file system can be used)
- Catalog based file system are used now days in UNIX, Windows, IBM mainframes and most computer systems. They are not used in SS devices which explains why the OS has to read the entire disk on key when you plug it in (why it takes long to recognize)

Problems with the UFS model

- No log – incase of crash we don’t know what happened with last I/O and may have problems in recovering
- Fragmented file – as we have seen (also from berkeley) we have the problem of fragmented files – when file is broken to many blocks that span all over the disk we need to seek for each block. This greatly reduce throughput. Berkeley allocation algorithms and larger block sized improved performance by factor of 10 (i.e. 1000%!) when first implemented (compared to version 7 UFS measured as ability to use disk throughput!) however Berkeley still achieved only 40-50% of disk throughput
- Wear leveling – the catalog is written much more then other parts of the file system

EXT2

- Ext2 contains some logical performance extensions over Berkeley
  - Multiple block size
  - Disk is implemented as several block groups each contain superblock, inodes and data and block and inode bitmap (to assist in finding free block/inode) – Using block groups helps to reduce fragmentation as files are extended to nearby blocks
  - 8 blocks at a time are allocated at write to further minimize file fragmentation
  - Ext2 added other enhancement (long file names, 4TB file system, large files (indirect^3), reserved space (for root), periodic file system check etc. that are beyond scope)

Ext2 i-node 1/2

- struct ext2_inode {
- __le16 i_mode; /* File mode */
- __le16 i_uid; /* Low 16 bits of Owner Uid */
- __le32 i_size; /* Size in bytes */
- __le32 i_atime; /* Access time */
- __le32 i_ctime; /* Creation time */
- __le32 i_mtime; /* Modification time */
- __le32 i_dtime; /* Deletion Time */
- __le16 i_gid; /* Low 16 bits of Group Id */
- __le16 i_links_count; /* Links count */
- __le32 i_blocks; /* Blocks count */

Ext2 i-node 2/2

- __le32 i_flags; /* File flags */
- union {__le32 } osd1; /* OS dependent 1 */
- __le32 i_block[15];/* Pointers to blocks */
- __le32 i_generation; / * File version (for NFS) */
- __le32 i_file_acl; /* File ACL */
- __le32 i_dir_acl; /* Directory ACL */
- __le32 i_faddr; /* Fragment address */
- union {} osd2; /* OS dependent 2 */
- }

Ext2 super block (important fields)

- Inode, blocks, count, size, free count etc.
- Timers (mount time, write time)
- Block group
- User id/group id
- How many blocks to pre-alloc on each write
- Magic number (to identify ext2 file system)
- Following the ext2 superblock (on serperate blocks) we will fine the ext2 block bitmap and ext2 inode bitmap

```
struct ext2_super_block {
00    __le32  s_inodes_count;         /* Inodes count */
      __le32  s_blocks_count;         /* Blocks count */
      __le32  s_r_blocks_count;       /* Reserved blocks count */
      __le32  s_free_blocks_count;    /* Free blocks count */
10    __le32  s_free_inodes_count;    /* Free inodes count */
      __le32  s_first_data_block;     /* First Data Block */
      __le32  s_log_block_size;       /* Block size */
      __le32  s_log_frag_size;        /* Fragment size */
20    __le32  s_blocks_per_group;     /* # Blocks per group */
      __le32  s_frags_per_group;      /* # Fragments per group */
      __le32  s_inodes_per_group;     /* # Inodes per group */
      __le32  s_mtime;                /* Mount time */
30    __le32  s_wtime;                /* Write time */
      __le16  s_mnt_count;            /* Mount count */
      __le16  s_max_mnt_count;        /* Maximal mount count */
      __le16  s_magic;                /* Magic signature */
      __le16  s_state;                /* File system state */
      __le16  s_errors;               /* Behaviour when detecting errors */
      __le16  s_minor_rev_level;      /* minor revision level */
40    __le32  s_lastcheck;            /* time of last check */
      __le32  s_checkinterval;        /* max. time between checks */
      __le32  s_creator_os;           /* OS */
      __le32  s_rev_level;            /* Revision level */
50    __le16  s_def_resuid;           /* Default uid for reserved blocks */
      __le16  s_def_resgid;           /* Default gid for reserved blocks */
      __le32  s_first_ino;            /* First non-reserved inode */
      __le16   s_inode_size;          /* size of inode structure */
      __le16  s_block_group_nr;       /* block group # of this superblock */
      __le32  s_feature_compat;       /* compatible feature set */
60    __le32  s_feature_incompat;     /* incompatible feature set */
      __le32  s_feature_ro_compat;    /* readonly-compatible feature set */
68    __u8    s_uuid[16];             /* 128-bit uuid for volume */
78    char    s_volume_name[16];      /* volume name */
88    char    s_last_mounted[64];     /* directory where last mounted */
c8    __le32  s_algorithm_usage_bitmap; /* For compression */
      __u8    s_prealloc_blocks;      /* Nr of blocks to try to preallocate*/
      __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
      __u16   s_padding1;
d0    __u8    s_journal_uuid[16];     /* uuid of journal superblock */
e0    __u32   s_journal_inum;         /* inode number of journal file */
      __u32   s_journal_dev;          /* device number of journal file */
      __u32   s_last_orphan;          /* start of list of inodes to delete */
ec    __u32   s_hash_seed[4];         /* HTREE hash seed */
fc    __u8    s_def_hash_version;     /* Default hash version to use */
      __u8    s_reserved_char_pad;
      __u16   s_reserved_word_pad;
100   __le32  s_default_mount_opts;
      __le32  s_first_meta_bg;        /* First metablock block group */
      __u32   s_reserved[190];        /* Padding to the end of the block */
}
```

Log based file system

- In attempt to improve stability we implement a file system log (similar to database log)
- We will record operation we are about to take in the log
- The log will help recreate

Ext3/4 file system

- The ext3 file system is basicly a log added to the ext2 file system
- Ext3 is currently the default file system in Linux
- Ext4 is the next (experimental file system)
- Both file system add additional features that are beyond the scope of this course (and the usability requirements of most users)

Other disk file systems

- Linux has many file system projects
  - ReiserFS – very fast and stable log based file system that lost popularity after its author Hans Reiser was arrested for allegedly killing his wife.
  - Xfs – yet another log based file system by SGI
  - Jffs (and varients) – file system for solid state disks
  - Cdrfs – cdrom file system

Logical file system

/proc

- /proc is a speical file system that contain some info regarding the system (for example max size of shared memory etc.)
- There is also directory for each running process containing information about the process (CPU accounting information, open file descriptors etc.)
- /proc is used by performance monitors and other programs that manipulate or monitor processes
- Other logical file system are implemented

Virtual file system

The virtual file system

- Several file systems are accessed by the same host (the HD, maybe another HD (with dos partition maybe?), a DVD, CD-R, USB disk on key and a network share or two)
- Each file system is MOUNTED and is assigned in a specific place.
- UNIX also puts some “special files” in place – sockets, pipes etc.
- All those files have a name and are accessed by UNIX
- All those files are part of the Virtual file system interface

The need for VFS

- We want to use files from many different file system each has (or maybe has not got) different super block and different properties
- Each file system driver has to support several methods that are supposed to be common to all file system (also when we create a new file system we register the new method and the new file system name for mount to use)
- When we call mount(1), unmount(1), open(2), read(2), write(2) etc. The kernel calls the VFS interface methods implemented by the file system driver (the piece of kernel code that make us able to read the files on the file system)

The VFS interface

- Vfs_mount – mount a file system
- Vfs_unmount – ubmount a file system
- Vfs_root – return the root vnode for the file system (what is vnode? Bare with me)
- Vfs_statfs – return file system specific info (answer to statfs(2))
- Vfs_sync – flush data to disk
- Vfs_fid, vfs_vget – beyond the scope (used by network file system)

So what is vnode

- Vnode is a kernel struct that points to a file (if the file is implemented on a UFS or similar file system a v-node will point on i-node)
- All file operation are done using the vnode operation vector which contain pointers to function that can handle the specific vnode (based on the file system that vnode points on. Obviously a v-node pointing to ext2 file system will be different from v-node pointing to msdos file system.)
- Not all vnode functions has to be implemented for every type of file system (for example one may implement file system that does not support hard links)

Vnode operation vector functions (partial list)

- Vop_select – implement select(2)
- Vop_rdwr – read to or write from file
- Vop_link – implement link(2)
- vop_rename – obvious
- Vop_mkdir – make directory
- Vop_rmdir – remove directory
- Vop_symlink – implement symlink(2)

Linux specific stuff

- Linux file system driver are implemented as kernel module (remember from class 1?)
- A file system driver inform the system he is a driver
- A file system driver supply the system with list of functions to call when a file operation is done on said file system (a struct is given with pointers to functions) and a name given to mount. When mounting a file system from that specific type the specific API will be called.
- A file system specific api is used with the new driver.

Other file systems

- Linux also support network file systems (file systems that are received via the network from windows or UNIX hosts), distributed file systems (file are saved on several computers and accessed by group of computers)
- Modules that are (below file system layer) that implement software RAID products
- File system interface written by several programs. – all those are considered beyond the scope