Betriebssysteme 13/14

File systems

11. Dezember 2013
Motivation

▸ RAM is not enough (currently)
  ▸ Main memory too small to contain all the data
  ▸ Information must be persistent beyond the lifetime of a process
  ▸ Multiple processes need to share data

▸ Solution:
  ▸ Store stuff on hard disks and similar devices
  ▸ Information becomes persistent – no longer dependent on processes, disappears only if deleted
  ▸ Administration by operating system

▸ ...we need a file system
Users view

- User does not want to see, know and understand
  - where and
  - how
- data is stored
- must be able to refer to data
- we need a naming scheme
types of files

- regular files
  - text (ASCII or UTF-data)
  - binary data (usual with an internal structure)
- directories
- special files (/dev/..., /proc/...)
  - block special files
  - character special files
accessing data

▶ tape drive model
  ▶ sequential access
  ▶ one byte after the other
  ▶ at the end: wind back

▶ introduction of floppy disks: any sequence possible
  ▶ random access
  ▶ seek-system-call added allowing to change access-position within file
directories

- helps organizing files
- directories, folders
- one or multiple levels
One Level

- A, B, C: owners - not names!
- Common on early PC’s, also CDC 6000 (first Supercomputer)
- Simple –
One Level

- A, B, C: owners - not names!
- Common on early PC’s, also CDC 6000 (first Supercomputer)
- Simple – for the system.
- Name clashes?
Two Levels

- one directory per user
- no (less) name clashing problems
- still insufficient
hierarchical system
file structure

- Internal structure different issue
  - for the operating system generally only a byte stream
  - other structures exist, mainly on mainframes
Implementation

Figure: Disk layout, classical example
Implementation („classical Unix“)

- boot block: Boot Loader, to boot system
- super block: Infos on file system, e.g.
  - size of partition
  - block size
  - pointer to free block list
  - inode-number of the root directory
  - „magic number“ (identifies file system type)
  - ...
- index nodes (inodes)
  - 1:1-relation between inodes and files
- data blocks
Implementation („classical Unix“)

- disk organized in block
- blocks group in cylinder groups
  - one copy of the superblock per group
  - data of one file ideally in one group (performance)
- free storage administered using bit maps
inodes
**Inode**

- **Attributes:**
  - **type**
    - file, directory, character special file, block special file
  - **owner**
    - user, group
  - **time values**
    - created, last modified, last accessed
  - **size**
    - in bytes and blocks
  - **mode**
    - permissions (rwx)

- but NO file name
Inode

- 10 (12) direct block pointers
  - sufficient for small files
- 1 indirect block: points to a block containing a list of block numbers
- 1 double indirect block: points to a block containing a list of block numbers
  - each of these block numbers points to a block containing a list of block numbers
- 1 triple indirect block: points to a block containing a list of block numbers
  - each of these block numbers points to a block containing a list of block numbers
    - each of these block numbers points to a block containing a list of block numbers
Inodes

- files may have multiple names
  - directories contain names and inode-number
  - multiple occurrence possible
  - „hard link“
  - inode contains link count

- Inode link-count 0:
  - no more reference within file system exists
  - file can be deleted

- number of inodes limited
  - file system may be full, because
    - no free inode
    - all blocks used
ext/ext2/ext3/ext4

- linux-based file systems
- originally „cross-development“ in minix
- based on the minix file system
  - quite clean implementation
  - but limited disk sizes and file names
- VFS: virtual file system layer
  - interface allowing to use different file systems
- ext: extended file system (1992)
  - first implementation supporting VFS
  - 2 GB disk size
  - 255 Byte file names
- successors: ext2 - ext3 - ext4
VFS
ext2

- first successor
- integrates ideas from the Unix File Systems (UFS) aka Berkeley Fast File System (BSD FFS)
- design principle: extensibility
- until Kernel 2.6.17 volume size limited to 2TB
- uses cylinder-groups, superblock, inodes as described above
ext2

- Attributes (examples):
  - c: compressed
    - file stored in a compressed manner
  - s: secured
    - blocks overwritten with 0 after file is deleted
  - S: synchronized
    - data written to hard disk immediately
  - A: append mode
    - files always opened in append-mode
ext2

- Fast symbolic links
  - symbolic link: indirection - file contains name of another file
  - normally stored on the data blocks of the file
  - fast symlinks: inode contains file name

- “dirty” state
  - after OS-crash: fsck recommended/enforced

- regulars file system checks (fsck), even if clean
ext2 performance

- **Readaheads**
  - more than one block per read requested from disk
    - when accessing files sequentially
    - when reading directories

- **Block-Groups**
  - inodes and data blocks „close“ to each other on hard disk
  - reduces head-seek-times

- **Preallocation**
  - allocating a block to a file results in allocating up to 8 continuous blocks
  - improves write- and read performance
ext2

▶ managing free entities
  ▶ inode allocation bitmap
  ▶ data allocation bitmap

▶ maximum file size
  ▶ depends on block size $b$
  ▶ $\min((\frac{b}{4})^3 + (\frac{b}{4})^2 + \frac{b}{4} + 12) \times b, 2^{32} \times b)$
    ▶ $b=1\text{KB}: 16\text{GB}$
    ▶ $b=4\text{KB}: 2\text{TB}$
ext3

- based on ext2
- journalling file system
- simple
- well tested
- differences to ext2:
  - Journal
  - file systems can grow dynamically
  - HTree for big directories
ext3 Journal

- changes to files stored in a journal
  - in principle a cyclic log
- first change noted in journal
- then executed in file system
- after crash: allows fixing inconsistencies easier
Journal - typical scenario

- file system is consistent
Journal - typical scenario

- file system is consistent
- changes are requested
Journal - typical scenario

- file system is consistent
- changes are requested
- changes noted in journal
Journal - typical scenario

- file system is consistent
- changes are requested
- changes noted in journal
- changes executed in file system
Journal - typical scenario

- file system is consistent
- changes are requested
- changes noted in journal
- changes executed in file system
- similar to stable storage concept
journalling - example

- delete file
  - may need two steps
    1. remove reference from directory
    2. delete inode

- Crash between the two steps:
  - orphaned inode
  - inconsistency

- Change order?
  - directory references non-existing inode
  - later: inode gets reused: may have fatal consequences
Journal

- Without Journal: fsck - file system check at reboot
- Now:
  - Read entries from journal
  - execute changes if required
- file system consistent much faster
- changes become atomic
  - either they have been done completely
    - either they have been completed before the crash
    - or they are executed after the crash from the journal
  - or they are not done at all (not being written into the journal)
implementing journals

- storage:
  - regular file (may grow)
  - hidden file that may not be moved
  - special area on disk
  - separate Device (SSD; NV-Ram)
- journal for the journal
- must be able to check the integrity of the journal
  - checksum
  - ignore entries with incorrect checksum
physical journals

- writes a copy of each block
  - first into the journal
  - then on the disk
- Crash: either
  - neither in journal nor on disk: no change
  - only in journal: copy to disk
  - already on disk: nothing to do
- high overhead
- acceptable for high correctness requirements
logical journals

- only meta-data written to journal
- trades safety against performance
- may lead to asynchronicity between meta-data and data
- example: enlarging a file concerns
  1. inode (size)
  2. free space bitmap (one further block assigned)
  3. new block (data)
- logical journal does not contain step 3: may lead to correct structure but garbage in file
writing journals

- writes to disk optimized by OS
- consequently the journal may be written after changes to disk is committed
- requires synchronisation of kernel, file system and drivers
- journalling-system must enforce flushing the disk-caches
journals in ext3

- three levels
  1. Journal (minimum risk taken)
     - data written to journal
     - data written to disk
     - data has to be written twice
  2. Ordered (medium risk taken)
     - Only meta-data to journal, data directly written to files
     - Changes to meta-data only declared „committed“ if data actually on disk
     - If only data is involved: nothing done in journal
  3. Writeback (highest risk taken):
     - Only meta data covered to journal. Data written to disk „eventually“ (task of sync-process). Risk of data-loss highest.

- Data-structure safe in all three levels. No checksums on journal.
- Content safe only in level 1.
ext4

- successor of ext3
- volume size up to 1 exibyte \((2^{60})\)
- file size up to 16 tebibytes \((2^{40})\)
- uses
  - extents
  - preallocation
  - allocate-on-flush
  - journals with checksum
extents

- Area of adjacent blocks in a file system
- files are extend by multiple blocks
- reduces fragmentation
- up to 128 MiB per Extent
- 4 extents stored in inode
- additional extents: HTree
allocation

- **pre-allocation**
  - allocate files with known file size in advance
    - will likely be stored on adjacent blocks
  - either: write zeroes to file after creation
  - now: use new syscall `fallocate()`
  - useful for media streaming and databases

- **dynamic allocation (allocate on flush)**
  - traditionally: write requires new block - allocate block immediately
  - now: store data in cache, only allocate blocks when committing to disk
  - improves performance
  - ideal combination with extents and multi-block allocator
btrfs

- Features
  - Extent based file storage ($2^{60}$ max file size)
  - Space efficient packing of small files
  - Space efficient indexed directories
  - Dynamic inode allocation
  - Writable snapshots
  - Sub-volumes (separate internal file-system roots)
  - Object level mirroring and striping
  - Checksums on data and meta-data
  - transparent compression (gzip and LZO)
  - Integrated multiple device support
    - RAID-0,1,5,6,10 implementations
  - Efficient incremental backup
  - dynamic resizing
  - SSD-awareness
btrfs

- lowest level manages pool of storage
  - may be multiple block devices
- managed in chunks (1 GiB)
  - multiple files in one Chunk
  - one file on multiple chunks
- RAID (redundant array of independent disks)
  - on chunk-level
  - RAID-1 (mirroring): two chunks belong together
  - RAID-10 (striped mirrors): multiple chunks
Sub-volumes

- correspond to a **posix-filesystem-namespace**
- different to LVM - not a separate block device
- you can mount
  - the top level volume - all sub-volumes visible as subdirectories
  - sub-volume only - you will see only the sub-volume
- can be created at any place within the file system hierarchies
- may be nested
COW

- Copy on Write
  - improvement on journal principle - but no journal used
  - blocks are never overwritten in place
  - when change is made
    - copy to a new location first
    - then meta data is changed to point to the new copy
  - same principle if meta-data on meta-data exist (e.g. superblock)
  - can be turned off (if fragmentation is an issue, e.g. databases)
- supports mirroring and RAID configurations
  - by default meta-data mirrored on two devices, duplicated if only one device available
- checksums to detect errors, CRCs stored separate from the data
Snapshot

- Snapshot: Copy of a sub-volume
  - actually clone of a sub-volume
  - almost no additional space (copy on write!)
  - snapshots can be written to and evolve separately
selected internals

- based on b-trees
- root-tree
  - contains everything
  - other trees are objects within root-tree
- file-system-tree
  - one per sub-volume
  - directories and files
    - inode items
    - extended attributes, access control lists
selected internals

- extents
  - contain file data (no meta-data)
  - outside of trees
  - if small enough: stored within tree
  - 4KB default size, size always multiple of 4K

- snapshots and clones share extents
  - changing part of a extent: up to 3 new extents created
    - small extent with the changed data
    - two extents with unchanged parts of original extent

- extents managed within extent allocation tree
Bitlocker

- Not a file system
- But an important technology for Windows users
- Data on a lost or stolen computer is vulnerable to unauthorized access
Bitlocker

- Not a file system
- But an important technology for Windows users
- data on a lost or stolen computer is vulnerable to unauthorized access what helps?

Encryption!
Bitlocker

- Not a file system
- But an important technology for Windows users
- data on a lost or stolen computer is vulnerable to unauthorized access what helps?
- Encryption!
BitLocker encryption

▶ encrypts
  ▶ the entire Windows operating system volume
  ▶ all user files and
  ▶ system files
  ▶ including swap and hibernation files
▶ available for current Windows system (Vista, W7, W8, Server 2008 and later)
Integrity

- Checking the integrity of
  - early boot components
  - boot configuration data
- requires a Trusted Platform Module (TPM) version 1.2
- BitLocker uses the TPM to ensure that
  - data is accessible only if the computer’s boot components appear unaltered
  - and the encrypted disk is located in the original computer.
other drives

▶ protects fixed and removable devices
▶ encrypts the entire contents of the drive
▶ can be configured to require that BitLocker is enabled on a drive before the computer can write data to the drive

▶ Unlocking drive:
  ▶ automatic (after password or smart card authentication)
  ▶ password required
  ▶ smart card required
  ▶ keys to unlock stored in Active Directory

▶ Can be mixed
  ▶ automatic on home computer
  ▶ password on other computer
hardware and software requirements

- Windows 8 or Server 2012
- TPM 1.2 or 2.0
- TCG-compliant BIOS or UEFI firmware
- boot order must force booting from hard disk
- firmware must be able to access USB stick
- 2 partitions on hard disk: system drive and operating system drive
TPM

▶ stores the decryption key on the chip
▶ measures
  ▶ BIOS
  ▶ MBR
  ▶ OS loader
▶ Only if these are unchanged the TPM will provide the key to unencrypt the data
▶ but no real protection against BIOS or MBR-viruses