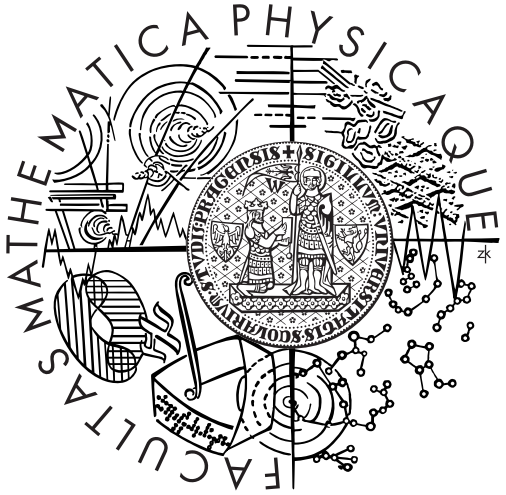
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Rastislav Wartiak

History and Backup Support for zlomekFS

Department of Software Engineering

Supervisor: Mgr. Vlastimil Babka

Study Program: Computer Science

2010

I would like to thank my supervisor, Vlastimil Babka, for guiding me through existing zlomekFS implementation and for the patience with my constant lack of time dedicated to this work.

I hereby declare that I wrote the thesis myself using only the referenced sources. I agree with lending the thesis.

Prague, April 15, 2010 Rastislav Wartiak

Contents

List of Tables

List of Figures

Název práce: Podpora historie a verzování v zlomekFS
Autor: Rastislav Wartiak
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Vlastimil Babka
e-mail vedoucího: babka@ksi.mff.cuni.cz

Abstrakt: zlomekFS je distribuovaný souborový systém, který umožňuje práci se sdílenými soubory i bez aktivního připojení, s využitím lokální kopie. Během synchronizace lokálních změn nabízí jednoduchý mechanizmus řešení konfliktních změn. Dalším vývojem se stal souborovým systémem bez specifického kódu v jádru. Může si tak v budoucnu získat popularitu mezi uživateli.

Jelikož může být obsah tohoto souborového systému aktualizován více uživateli, může být ukládání historie změn užitečnou vlastností. Tato práce realizuje verzování souborů v zlomekFS, vyřešením otázek jako způsob uložení verzí a přístup k nim. Systém je dále rozšířen o možnost zálohování s využitím verzování.

Nová funkcionalita je odvozena z analýzy jiných soborových systémů s obdobnými možnostmi a výběru nejvhodnějšího způsobu pro zlomekFS.
Klíčová slova: distribuovaný, souborový systém, verzování, zálohování

Title: History and Backup Support for zlomekFS
Author: Rastislav Wartiak
Department: Department of Software Engineering
Supervisor: Mgr. Vlastimil Babka
Supervisor's e-mail address: babka@ksi.mff.cuni.cz

Abstract: zlomekFS is a distributed file system that supports disconnected operation using local cache. During synchronization of local changes it offers easy-to-use conflict resolution mechanism. Further improved it became a file system with no specific kernel code. It has therefore a good potential in future public use.

As the content of this file system can be updated by many users, keeping history of the changes can be a useful feature. This thesis implements file versioning in zlomekFS, answering the questions such as how to store and access the history. On top of the versioning, the possibility of consistent backup is introduced into the file system.

New functionality is derived from the analysis of other file systems with similar features and selection of the most suitable approach for zlomekFS.

Keywords: distributed, file system, versioning, backup

Chapter 1

Introduction

1.1 zlomekFS

The zlomekFS[1] project built a network file system with support for disconnected operation inspired by CODA. The file system creates a local cache of remote files that can be used to operate without network connection. Changes done without network connection are integrated back to remote files when network connection becomes available. Conflicting change is represented as a directory in the file system and the user can resolve the conflict simply by deleting one of the file or directory versions present in the conflict directory, without using specialized tools. When caching the data, zlomekFS can operate in three modes, based on the connection to the node providing the data:

- On-line, with fast connection (low latency): All data is synchronized and local copy is kept up-to-date.
- On-line, with slow connection (high latency) – so called "mobile operation": Only the data that is currently accessed in synchronized, keeping the rest of the modifications for later synchronization.
- Off-line – so called "disconnected operation": Local copy is used and updated. When a low-latency connection is established, local changes are reintegrated to the master data.

1.2 History

First version, created by Josef Zlomek[1], set up most of the functionality it has today. His thesis set up fundamental design decisions as volumes, caching and conflict resolution. zlomekFS was split between a small kernel-space wrapper and user-space daemon. Most of the code developed in this version is still used.

Later changes, done by Miloslav Trmač[2], replaced zlomekFS-specific kernel-space code with a standard wrapper. This was allowed by the existence of FUSE framework[4] and made systems using zlomekFS independent on kernel versions and simplified development and testing of new zlomekFS functionality. With

these changes, it was possible to run more zlomekFS mounts on one system. It still, however, required changes in non-zlomekFS code – it relied on expanded FUSE interface.

Last versions so far, published by Jiří Zouhar[3], included support for regression testing and added new logging facility. His work was mostly focused out of the zlomekFS code, but its result is important for a wider use of zlomekFS. It can ensure the file system is reliable. In my opinion, this is more important than features a file system offer.

1.3 Goals

The goal of this thesis is to extend the support for integrating changes done without network connection so that a history of recent changes can be accessed, especially for backup and recovery purposes. That includes:

- Detailed design of the support for history, based on both practical usability and technical options. Questions such as how long to store the history, how to access the history, etc. should be answered.
- Working implementation of the support for history, well documented and tested and integrated within the existing project.

After zlomekFS was ported to FUSE, new version of FUSE offered functions that were missing in the implementation before and there is no longer need to patch FUSE to add there functions. During the implementation of the changes designed in this thesis, zlomekFS will be updated to leverage these new functions.

1.4 Structure of the thesis

In chapter 2, first, some of existing file systems that support file versioning are introduced with their approach to file versioning. Interesting details of these selected file systems are described. Later, issues to solve in this thesis are outlined. Next, chapter 3 analyses ways to take consistent file system backup and add to the list of decisions to be made created for the file versioning defined earlier. Chapter 4 presents approach taken in the implementation and contains implementation details of designed file versioning and backup. Apart from coding details, it contains brief overview of problems encountered during the development and testing. Chapter 5 contains description of test performed on the implementation and results of these tests. Finally, chapter 6 concludes the thesis and shows how the goals were met. Ideas for future work are presented there too.

Chapter 2

File history - versioning

This chapter analyses file versioning in some of the existing file systems and summarizes decisions to be made before implementation of versioning in zlomekFS.

Most of the text discussing possible approach (starting with section 2.2) focuses on file versions. Unless explicitly specified, this applies to directories as well as they can be viewed as a regular file containing list of file names and its attributes.

2.1 Versioning in existing file systems

2.1.1 File-11

File-11 is one of the first implementations of file system with versioning support[1]. This is the file system used in OpenVMS operating system[9]. In addition to a file name and file type, every file has a version number. When the system displays file specifications, it displays a semicolon in front of the file version number. When accessing a file, it is possible to specify a version by adding a semicolon or a period to the file name. Versioning did not apply to directories.

Unless user specifies a version number, the system uses the highest existing version number of that file. When a file is modified and a new version of the file is created, the file name remains the same but the version number is incremented by one.

User can refer to versions of a file in a relative manner by specifying a zero or a negative version number. Specifying zero locates the latest (highest numbered) version of the file. Specifying -1 locates the next-most-recent version, -2 the version before that, and so on. To locate the earliest (lowest numbered) version of a file, user can specify -0 as the version number. Note that is not possible to create files with a version number higher than 32767. An attempt to create a new file with a version number higher than 32767, will result in an error.

The `/VERSION_LIMIT` qualifier for the `CREATE/DIRECTORY`, `SET DIRECTORY`, and `SET FILE` commands controls the number of versions of a file. If the version

[1] VMS v1.0 with this file system was released to public in October 25, 1977 and its design started in 1975.

limit is exceeded, the system automatically purges the lowest version file in excess of the limit. For example, if the version limit is 5 and user creates the sixth version of a file (`ACCOUNTS.DAT;6`), the system deletes the first version of the file (`ACCOUNTS.DAT;1`). To view the version limit on a file, user can enter the `DIRECTORY/FULL` command.

The `PURGE` command can be used to manage the number of versions in a specific directory. By default, all but the highest numbered versions of all files in the current directory will be deleted. This behavior can be overridden with the `/KEEP=N` switch and/or by specifying directory path(s) and/or file name patterns.

2.1.2 Wayback

Wayback[5] is an implementation of a versioning file system for Linux. It provides the ability to remount any already mounted file system with versioning support under a different directory. Because of this, Wayback can be used on any block device with any base file system (ext3, ReiserFS, FAT, etc).

In Wayback, versioning is done automatically at the write level: each write to the file creates a new version. It implements versioning using an undo log structure. It is a user-level file system built on the FUSE framework that relies on an underlying file system for access to the disk.

Existing directory can be mounted as a new directory, with versioning support. All changes done to the files in the new directory are recorded by Wayback. At any time, user can list version of a file by `vstat` utility and revert to an old version by `vrevert` utility. Reverting can be done to a specific date/time or version and even reverting can be undone. Old version of a file can be saved as a new file by `vextract` utility or deleted by `vrm` utility.

Every versioned file has a hidden log file and every directory has a hidden catalog. Log contains timestamp, write offset, size, file enlarged flag and data. When a file is deleted, it is first truncated to zero bytes. This creates a backup of the file data. Deleting a directory actually renames it and omits it from directory listing. This way logs for files in the directory are not lost.

2.1.3 ext3cow

ext3cow[10] is a stand-alone disk file system based on the ext3 file system. Versioning is implemented through block-level copy-on-write. That is where the name of the file system comes from, as COW stands for Copy-On-Write.

When a change to a file is made, instead of writing the new data over the old data, ext3cow preserves the old data, and allocates new data blocks for the new data. Only the changed data is given a new allocation, allowing similar data blocks to be shared between versions – this minimizes the amount of data required to support versioning.

File versions are not visible in a directory listing, but can be accessed by appending `@` character followed by an epoch number – number of seconds that passed since The Epoch (00:00:00 UTC, January 1, 1970). There are plans to

support some common, for example: `@yesterday`, `@lastweek`, `@lastmonth`, etc. Currently, there is no mechanism to delete old versions of a file, but one existed in a version for old kernel versions.

New versions are created based on snapshot. A snapshot updates the file system's epoch counter, located in the super block. It copies no data and only performs a sync to flush dirty buffers. The actual version creation is done when an IO operation that modifies a file is performed. Multiple changes made between two snapshots are therefore merged into one.

2.1.4 CopyFS

CopyFS[11] is a simple versioning file system build on the FUSE framework. It is especially useful to store configuration files as the new versions are based on user-named snapshots called *tags*. User can tag files with arbitrary version name and move back-and-forth between versions.

List of version can be retrieved by `copyfs-fversion` utility. Individual versions of a file are accessible by locking an older version of a file as a current one by this utility and then access the file by its name.

2.1.5 HTFS

High Throughput File System (HTFS)[13] is the file system used by SCO OpenServer. When mounting a HTFS file system it is possible to specify maximum number of file versions kept, or specify zero to disable versioning. Another option specifies minimum time before a file is versioned. If set to zero, a file is always versioned (as long as maximum number of file versions is greater than zero). If set to a value greater than zero, the file is versioned after it has existed for that number of seconds.

Versioning is enabled on a per-directory basis by setting the directory's setuid bit, which is inherited when subdirectories are created. If versioning is enabled, a new file version is created when a file or directory is removed, or when an existing file is opened with truncation. Old versions remain in the file system, under the name of the original file but with a suffix attached consisting of a semicolon and version sequence number. All but the current version are hidden from directory reads (unless the `SHOWVERSIONS` environment variable is set), but versions are otherwise accessible for all normal operations.

2.1.6 Write Anywhere File Layout

Write Anywhere File Layout (WAFL)[18] is a file system designed specifically to work in an NFS appliance. It offers snapshots, read-only clones of the active file system. WAFL uses a copy-on-write technique to minimize the disk space that snapshots consume. Using snapshots it eliminates the need for lengthy file system consistency checking after an unclean shutdown.

WAFL creates and deletes snapshots at predefined times. By default, it creates four hourly snapshots during the day and nightly snapshot every midnight.

It keeps hourly snapshots for two day and nightly snapshots for a week. Saturday-to-Sunday snapshot is kept even longer, for two weeks. It keeps up to 255 snapshots on-line. Every directory in the file system contains a hidden sub-directory `.snapshot`. This sub-directory enables access to older version of files in the directory. It is not contained in directory listing.

Snapshot is created by duplicating root inode. It contains link to the same data blocks as current root inode, no data except for root inode itself is copied. When data is changed, new block is created and links in the root inode are updated. Old data blocks are then not in the current tree, only the snapshot tree. This applies to all file and indirect blocks too, up to the tree root. As this would cause a lot of I/O operations, commonly modified blocks, such as indirect blocks and inode file blocks are written only once-in-a-while, instead of after every request.

Such caching can result in file system inconsistencies after an unclean shutdown. WAFL therefore creates special snapshots called *consistency points* every ten seconds or faster. These snapshots are accessible by user, they are used only after restart to revert the file system to a consistent point (most recent of them). Between consistency points WAFL writes only to unused blocks, so blocks used by the last consistency point remain unchanged. Log of all operations performed since the last consistency point is kept in extra piece of hardware - non-volatile RAM. After restart, WAFL replays any request that is left unprocessed in it. When shut down normally, it creates consistency point after it closes its NFS interface. In that case, there are no requests in the log and disks contain up-to-date copy of the file system. WAFL uses two logs. Only one of them is used to store new requests and once it gets full, new consistency point is written, while the other log is used. Logging only requests instead of data blocks, as with regular disk cache, less storage is needed. Most modifications are smaller than a block size, e.g. modifications of meta-data.

WAFL uses a bitmap for every data block in the file system. Each bit represents one snapshot, if it is set then the block is used in that snapshot. Same rule applies to current state, it has dedicated one bit. File system then easily knows if the block is in use or not. Deleting a snapshot means to clear appropriate bit in the bitmap for all used blocks.

2.1.7 Log-Structured File System

Log-structured file system is a storage mechanism which stores all data in circular log using sequential disk access. It is similar to today's journaling file system, except for the fact that it uses log (or journal) as a final storage for the data, not only as a temporary write-ahead log. It was first implemented in Sprite LFS[19]. The idea derives from the expectation that as hardware performance increases, it creates a gap between the speed of CPU and seeking time of magnetic storage. Having more RAM, most read requests are served from the cache, whereas write requests must still be written to disks.

Log created by the file system is written throughout the disk and contains

inodes, indirect and data blocks and all other meta-data. By the time it reaches the end of the disk, free space is created in the log by deletion of the data. It has to therefore utilize this free space by either directly writing new data or moving and compacting surrounding data to create bigger free segments. Most of the file system implementation deals with this free space fragmentation.

System start-up and crash recovery use checkpoints that are regularly written to the log and rolling-forward mechanism to get up-to-date state of the file system in the memory cache. Even though Sprite LFS itself does not expose older versions of files stored in the system or snapshots, it has all data to do so in the log.

2.1.8 NILFS

NILFS[12] is an implementation of a log-structured file system supporting continuous snapshotting. It creates a number of checkpoints every few seconds or per synchronous write basis (unless there is no change). Each checkpoint represents a consistent state of the file system. Users can select significant versions among continuously created checkpoints, and can change them into snapshots which will be preserved. There is no limit on the number of snapshots until the volume gets full.

A snapshot is the checkpoint marked not to be deleted by garbage collector. The garbage collector is separately implemented as a user space daemon and can be executed manually. The recent checkpoints are protected from garbage collector during the period given by a parameter `protection_period`; garbage collector never deletes checkpoints whose age from its creation time is less than the value given by the parameter.

Checkpoints can be listed with `lscp` utility, each checkpoint has its identification number. It is possible to make a snapshot of the current state by `mkcp` utility, and is also possible to change an existing checkpoint into a snapshot and back by `chcp` utility. Checkpoints can be invalidated by `rmcp` utility.

Each snapshot is mountable as a read-only file system. It is mountable concurrently with a writable mount and other snapshots. This feature is convenient for on-line backup.

2.1.9 Elephant

Elephant file system[14] is a prototype of stand-alone versioning file system, implemented in the FreeBSD kernel. Old versions are created using copy-on-write and can be accessed based on a timestamp. Elephant manages storage at the granularity of a file or groups of files using user-specified retention policies. Apart from these policies, file system uses file-edit histories to determine which old versions of a file to retain and for how long.

Retention policies offered in this file system are: keep all, keep one and keep landmarks. Keep all and keep one are simple (non-)versioning algorithms. Interesting approach is taken to automatically detect versions that are important in

the file history, called *landmark versions*. It is based on the assumption that as the file versions become older, lower granularity of changes to store is important. Elephant therefore keeps all versions up to the certain point in time, when it starts retiring versions created in a short time interval, while keeping only the last version from that interval. More into the past it goes, larger interval it applies to file versions stored at that time. Apart from automatic landmark detection, user can explicitly mark specific versions as landmarks.

Deleting version of individual files can cause problems for the user to retrieve consistent set of files from the history. A utility to find consistent file set versions is provided. Based on specified timestamp, it offers the user closes available versions creating a consistent file set. Elephant has to keep information about all versions, even those that were deleted, to find such moments in the history. Authors plan to extend the functionality of the file system with user defined file sets and to apply versions and landmarks to whole sets.

Old versions of files are accessible by adding `#date:time` to the file name. If applied to a current working directory, all relative paths reference files from that moment in history. Process can even define that moment by newly added `setepoch` system call. Versioning access is then applied to all file access, including absolute paths. Another system call, `readinodelog`, is implemented to list versions of specified file.

2.1.10 CVFS

Comprehensive Versioning File System (CVFS)[15] is a stand-alone versioning file system that focuses on storage space efficiency. It performs comprehensive versioning, where every modification (i.e. even every write operation) creates a new version of a file. This was a design requirement as its intended use is as a self-securing storage, to enable post-intrusion diagnosis by providing a record of what happened to stored files before, during, and after an intrusion. Users can access the file system via NFSv2 mount.

Creating a new version for every change can generate a huge number of versions. As every new version requires new storage space, main of objective of CVFS project was to reduce the space required to a minimum. It uses two approaches: journal-based meta-data to store inodes and indirect blocks and multiversion B-trees to store directories. Journal eliminates the need to allocate a new block for every change in file meta-data, e.g. change in a single pointer in inode or indirect block to a modified data block. This slightly increases retrieval time for older versions of files, but significantly reduces space required to store a new version. File system keeps the retrieval time on an acceptable level by storing snapshots of data in regular intervals. This limits the amount of journal that is replayed during the retrieval of an older version.

Directory entries are stored in multiversion B-trees. This is a variation on standard B-tree that keeps old version of entries in the tree. Entries consist of a regular key/data pair and a time range for which the entry is valid. They are sorted first by the key and then by the time range. As the entries are unique,

all standard B-tree operations apply. This structure is optimized for lookups (logarithmic for both current and old versions), while listing becomes slower as the number of versions increase.

Access to current versions has the performance as the access without any versioning. Older versions are retrieved by a combination of snapshot data and journal roll-back, access time depends on how often the snapshot is taken and clustering of write operations. Clustering refers to the physical distance between relevant journal entries. If several changes are made to a file in a short span of time, then the journal entries are most likely to be read together, thus speeding up journal roll-back.

Versions that are older than specified amount of time are cleaned from the file system by a separate daemon.

2.1.11 PersiFS

PersiFS[7] is a stand-alone versioned network file system that is exposed to the users as a standard NFS mount. Each change to the file system is stored as a new version. These version are then accessible using timestamp as a separate directory, containing read-only versions of the whole file system as it existed at that point in time. Current version of the file system is accessible in a directory called `now`, older versions as directories in format `yyyy-mm-dd-hh-mm-ss`. To improve the usability, the user interface includes a separate tool to display a log of modification times for a particular file.

Each write to the current version of the file system writes the data to its destination and appends the change to the modification log. When enough changes are made to the file system, a snapshot of the data is created and modification logs are discarded. This allows to read current version of the data directly, while having reasonable fast access to older versions, as only logs since the last snapshot was taken need to be replayed.

Files are divided into chunks and file system handles them in chunks, instead of blocks. Chunks are stored separately and file data is actually only a list of chunks. Every chunk has a fingerprint, hash of its contents. Whenever chunk is added it is first checked, whether chunk with the same data already exists. If so, existing version is used, otherwise it is inserted into an on-disk B^+-tree. This efficiently stores multiple versions of a file. Note that chunk boundaries are not placed at regular intervals but instead use content-sensitive chunking, based on file contents.

PersiFS is unusual in creating new versions, as it is not notified of file close, because of the NFSv3 protocol. It has to create a new version for every write operation or for a series of operations which it tries to group together in a five-second time. This can possibly create inconsistent file version as the version can be created during a file modification.

2.1.12 Versionfs

Versionfs[6] is a lightweight user-oriented versioning file system. It works on top of any file system and provides a host of user-configurable policies: versioning by users, groups, processes, or file names and extensions; version retention policies and version storage policies. Versionfs creates a new version, every time the user modifies a file. Older versions are accessible using regular file operations and a library wrapper. They are identified by `;Xn` postfix in the file name, where X stands for a storage policy type (f, c, s, d) and n for a version number. Along with versions, meta-data file with suffix `;i` is stored, that contains minimum and maximum version numbers and storage method for each version.

Versionfs stores current version of a file as a regular file. Older versions are stored based on a file storage policy: full, compressed and sparse mode and special storage policy for directories. In full mode, the entire file is copied to an old version after the first change in the file. Further writes modify only the current version, until the file is closed. Compress mode behaves in the same manner, except that the old version is compressed when copied. In sparse mode, only changed data blocks are stored in old versions. When a block is modified in the current version, it is copied into old version, to its appropriate place. Versionfs uses sparse files to store these versions, to save storage space. As it would not be possible to decide whether the version file contains zero-filled block or it is not stored in that version, extra meta-data is stored at the of the version file. Meta-data contains the original file size and a bitmap that indicates which parts of the file version are valid.

New versions are created using copy-on-change policy. This differs from common copy-on-write in that writes that do not modify data (i.e. write the same data over the old one) do not result in a new version. Versionfs compares the old and new data blocks. File delete is either replaced by rename or data is compressed (resulting in higher I/O), depending on the storage policy. Truncate is replaced by rename and new empty file is created instead, to save I/O.

Old version of a file is retrieved either directly from version file or uncompressed from it or multiple spare version files are used until whole file is created. This can result in traversing all versions, up to the current one. Versionfs allows converting old version between storage formats. Administrator can turn off read or read-write access to old versions.

Retention policies determine how many versions are stored. Minimum and maximum can be specified for a number or age of versions or for a storage space they consume. A version is discarded if it exceeds at least one of the maximums and does not violate any of its minimums. Retention policies applied to individual files, thus administrator can set defaults and limits for values for each file size, file name, file extension, process name and time of day.

Manipulating multiple files stored in a regular file while creating or modifying file versions is not atomic. In case of a system crash, meta-data file can be fully regenerated from file versions, because storage method and file version is a part of file version name. File system checker is provided to reconstruct meta-data files.

2.1.13 CheckFS

CheckFS[20] implements checkpoints (snapshots) and block-level incremental backup features in Linux common ext3 file system. It uses copy-on-write technique to store data changed are creating the checkpoint. During this process all pending and current I/O operations are finished first and CheckFS starts to monitor write operations. Creating a checkpoint does not require any data to store in the file system.

It works on top of ext3 file system, it is enough to load two kernel modules. When a write occurs to a block, it is first copied to a free block and information is stored in special checkpoint block. Subsequent writes are performed in regular way, until next checkpoint is created. If a hole in a sparse file is modified no data is copied, thus saving storage space.

CheckFS directly supports incremental backups by storing only the data that is referred to in checkpoint blocks, it therefore offers easy block-level incremental backup. Multiple checkpoints can be created and release of a snapshot results in merging the data into the next snapshot. Utilities are provided to backup checkpointed data and to restore it to the file system into separate directory subtree.

2.1.14 Volume Shadow Copy Service

Volume Shadow Copy Service[21] allows to create a snapshot of NTFS file system. It is provided by Microsoft in all systems running Windows since versions XP and Server 2003. VSS provides a unified way to take block level backup copy by solving problem with open files by operating system itself, instead of backup software. When snapshot is creating all applications that have any files open receive a signal from VSS to put all files into consistent state and stop making any file modifications. Once the data blocks are copied, another signal is sent and all applications resume its normal operation.

This service is provided primarily for backup purposes. However, snapshots can be mounted to any directory within existing file systems. This makes VSS a versioning tool. As snapshots can be created using copy-on-write technique, they are space-efficient. With a support from third-party application, NFTS can offer versioning support, though with a more complicated way of retrieving older versions.

2.1.15 Version control systems

Concurrent Versions System (CVS)[16] and Subversion[17] (and many others) are version control systems for keeping track of modifications to project (usually source code) files. File versions are stored on separate file system (usually remote) called repository and files are versioned using specific tools. These systems can add simple versioning support to almost any regular file system. However, they do not work transparently with all applications.

CVS uses separate version number for individual files, called revision number. While this is enough for separate files, version control systems are primarily used to store source code. And even for simple development project there are multiple files in the repository. As they are interconnected, it is not easy to find corresponding versions of other project files for an older version of specific file.

Most of the shortcomings of CVS are solved in Subversion. Files can be committed to the repository in sets (called changesets) and uses one global revision number for the whole repository, instead of versioning individual files. User can always retrieve consistent set of files from the history, as long the changes were committed in logically-interconnected groups. Among other improvement, Subversion can keep track of files when renamed or moved.

Mature version control systems provide tools supporting distributed work on a stored file. It is possible to lock a file for a longer time, without a need to be connected to the repository and they offer tools for merging conflicting changes.

2.2 Use of versioning

Versioning support can serve different purposes in the file system. Preferred usage can be taken into account during the designing the versioning.

2.2.1 Undo user change

One of the reasons to use versioning file system is to offer users the ability to restore older versions of their files. Files are deleted or modified by mistake or the user wants to revert changes made to a file since particular moment in time. Versioning file system offers functionality that is otherwise, and only to some extent, done manually by users themselves - taking backups or storing versions under different names.

2.2.2 Backup

Many file system backup solutions rely on the functionality of accessing all files within the file system in a single point in time. As this is not possible on a regular non-versioned file system[2] without delaying user operations during the whole backup, versioned file systems are used. This area is covered in more detail in chapter 3.

2.2.3 History analysis

Versioning can be very useful in intrusion detection. Attackers are usually trying to hide their activity on penetrated systems. They are deleting or modifying log files and wiping out all their traces. File system that does not allow users to delete or modify old versions of files for a time long enough to be reviewed by the administrator can be useful in such situations.

[2]Snapshots are considered a versioning feature.

2.3 New versions

First decision to be made when implementing a versioning in a file system is when new file versions are created.

2.3.1 File level

Most detailed versioning is to create new version after every operation that modifies the file. This is closed *on-change* versioning. Most of existing file systems do not expose individual changes to the user, even though journaling file systems store this level of information.

Common approach is *on-close* versioning, to create a new version every time a file that was changed is closed. This way all versions of a file, as they are perceived by the user, are stored. User can revert any change made to the file, as long as old versions of that file are stored. As this may have undesired performance impact, especially on files that are frequently modified, it may be possible to limit the time between two versions of a single file are created. If a modified file is closed and less then specified amount of time has elapsed since that last version was created, modifications are written directly into the last version, instead of creating a new one.

Some file systems create new versions only on user request. This is called *tagging*. When a file is tagged, first modification to it creates a new version that is being updated since then, until next tag. Once the user finishes his/her modifications to a state that is worth preserving (at least for some time), he/she tags the file. Tagging can be done on individual files, on multiple files, directories or on a whole file system. Using a file system that does not support versioning can the user achieve this either by copying to another directory, by renaming the file to include the version number or to store the file in a version control system (see subsection 2.1.15).

File level versioning offers better customization of versioning, as versions of each file is created and maintained separately. It is possible to apply versioning only to some files and thus reduce storage space required for versioning support or to store versions for a longer period of time.

2.3.2 Block level

Another option is to tag files automatically. File system takes *snapshots* of files regularly or per user request and creates new version for every file that has been changed since the last snapshot was taken. These snapshots are particularly useful when taken for the whole file system, as they represent a consistent state of it. This can be used for backup purposes. Most file system snapshots are done this way, as they are implemented using block-level copy.

Block-level versioning applies to all files equally. It is not is possible to store multiple versions of one file and discard or even not create versions of other file. This can be an advantage, as all files within one snapshot are from single point

in time. Having file set consisting of many files, it may be complicated to retrieve all matching versions of different files using file level versioning.

2.4 Modified data

User sees individual file versions or snapshots[3] as separate sets of data that may or may not differ between versions. Creation and storage of version data differs in creation and access speed and storage space required.

2.4.1 Copy

Simplest way of creating a new version or a snapshot is to copy all data representing particular file or file system respectively. Its obvious drawback is that each version requires same amount of storage space even when only small part of a file or file system is modified. This applies to snapshots in particular. Full copy versioning method is in fact a backup method. Copied data are completely separate from its original and can be stored on other storage media or easily transported.

Huge amount of data is read and written with each version and this process may negatively affect the file system. If new versions are created during non-business hours (e.g. during the night), it may be an advantage over other methods. Once the process is finished, no change in the file system results in extra data copy, as it seen in other methods. This can result in absolutely no performance impact on the file system operations, when compared to non-versioning approach.

2.4.2 Copy-on-write

Many versioning file systems create versions and snapshots using copy-on-write technique. When a write request resulting in a new file version is processed, either old data is copied to separate location prior overwriting it with the new data or new space is allocated for the data. Pointers to the data (inode and indirect blocks) are updated for old or new version respectively. Write request resulting in a modification of the last snapshot is processed in a similar way.

Using copy-on-write reduces storage space required for versioning. Depending on the size of modified data between versions or snapshots, the size of the version varies from zero to the size of the file or file system. This approach has a performance impact on the write requests that result in data copy, response time is longer.

[3]This section mentions both types of versioning in most of the text, as it may not be obvious where only one of them applies.

2.4.3 Copy-on-change

Applications do not have any information whether underlying file system is versioned or whether it uses copy-on-write to store modified data. When modifying a file they can truncate the file and write the new content that differs only in a small part. Copy-on-change technique copies the data between versions only when they differ from what is already stored in the file or file system.

Each block of the new data in a write request is compared to the stored data. Byte comparison is resource consuming, it is better first compare hash computed from both blocks and use byte compare only if they match. Further optimization can be achieved by storing the hash value along with the data. New data is in cache, thus no I/O is required to compute the hash. If the result of the comparison is to write the data block, computed hash can be immediately reused.

This technique can save storage space and I/O for the cost of extra CPU power. Effectiveness differs depending on the applications and their write algorithms used.

2.4.4 Journal

Many changes, to meta-data in particular, modify only a small portion of data block. Using a new data block with each change requires unnecessary amount of storage space. File system can use *journal* to store information about the change and directly modify the data. Retrieval of an older version of data then consists of using current data and replaying (or *rolling-back*) the journal until the requested point in time. The journal therefore has to contain identification of the change and old data. Once the journal is implemented, adding new data to the journal entry creates write-ahead log that can be used for meta-data consistency, as in regular journaling file system. Or, if versioning is to be implemented in a journaling file system, it requires only small changes in the journal to add versioning.

As more changes are in the journal, replaying it may become slow and resource consuming. This can be avoided by creating *checkpoints*, snapshots of current data. Replay is then performed only since the first newer checkpoint as it contains all data of requested file at the time of the checkpoint. File system implementation has to decide how often checkpoints should be made. Every checkpoint shortens the time needed to retrieve old versions since the last checkpoint, but it requires more storage space. Creating checkpoint after every change behaves like a system without journaling.

2.5 Storage optimization

Versioning requires more storage space as multiple versions of data is stored. If file versions differ only in part of the data, it can be effectively shared between versions.

2.5.1 Data block sharing

Unmodified data blocks can be shared between versions. When a new version of a file or snapshot is created, file system shares unmodified the data blocks with previous version(s). Using copy-on-write mechanism, data from previous version is not overwritten but a new data block is allocated and written to. Meta-data is updated to point to the new block.

Data block sharing is commonly implemented in snapshotting file systems. In fact, block-level copy-on-write and copy-on-change are based on this technique. Sparse files can be used to store modifications in file systems that implement file-level versioning. Shared data is not written to the version file resulting in a hole in the file. Additional meta-data has to be stored along with the file, as it is not possible to distinguish between a hole and a zero-filled block of data[4]. This meta-data contains a block map, recording used blocks and holes in the version file.

2.5.2 Data range differencing

Improvement of data block sharing is to share variable ranges of file data. Instead of keeping data blocks with the same contents, unmodified data ranges are kept. This method gives better results for insertions or deletions of parts of the file. Text files are commonly modified this way.

This technique is not possible in block-level versioning. File-level versioning can again use sparse files, but meta-data is a bit more complicated. It does not store a bitmap of file blocks, but a list of ranges. Interval tree is a suitable structure to storage such information.

2.5.3 Hash-based data storage

Another improvement of data block sharing is to share data only between version, but system-wide. Some applications modify file contents by truncating the file on open and writing new data afterwards. Even if there is only a small change in the file all file systems using copy-on-write duplicate the data. This method can handle these unnecessary truncates efficiently.

File system builds a tree of data blocks with block data hash value as a key. When a new data block is to be written, it is first searched in the tree. If it exists, it is reused in the operation. It if does not yet exist, new block from the storage is allocated and the block is added to the tree. Every block has a usage counter associated, so the file system known the block is no longer used in any file and can be released with the last file that was using it.

This technique can save storage space even when multiple copies of the same file are stored in the file system. Disadvantage of this technique is the need of extra storage and CPU time.

[4]It is possible to find holes in a file by truncating it block-by-block and comparing number of disk blocks used. But it is a lengthy and complicated way.

2.6 Access to versions

As versions are created, user has to be able to access them. This can be done in more or less transparent way.

2.6.1 Within the file system

Older file versions can be accessible directly within the file system. Using regular file operations is more convenient for the user and older versions can be used directly in applications. Versions can be present in directory listings, either along the current versions or in separate directories. File systems relying on a simple copy versioning method can offer the access to such versions without any extra code.

These directories can be included in every file system directory or a separate tree can be provided. Separate tree is used only for snapshots, as all files are from a single point in time and the timestamp can be a part of the tree name. These trees are either an integral part of the file system or are mounted separately.

2.6.2 Separate utility

Some file systems do not offer the possibility to access the files directly. Instead, they provide tools to manipulate with files: list versions, delete version, copy version to a file within the file system and sometimes revert file to an older version, as a special case of previous operation. Only very simple file systems are limited in this way, most of the file systems more user-friendly.

2.6.3 Read-write access

Depending on the file system, it may be possible to modify contents of older versions. Modification of the data that differs from previous and next version is straight-forward. Making changes to shared data, e.g. when copy-on-write is used to create versions requires more actions to be performed. Except for journal-only file systems (described in subsection 2.4.4), the ability to do so is only a design decision.

2.7 Version names

There are three common version naming schemes: numbers, timestamps and tags.

2.7.1 Numbers

Numbered versions are found in file-level versioning file systems. First time file is created, it has no versions, apart from current. Once a modification occurs resulting in a new version, old version is stored with a number 1. As new versions come, they are assigned numbers in an increasing order.

Unless explicitly specified, file system offers the user the latest available version. Access to older versions requires specifying version number. Version number is a part of the file name, if specified. It is often used as a suffix, separated from the file name by a semicolon or another non-conflicting symbol. File system has to prevent users to create files with names that can conflict with selected naming schema.

Apart from absolute numbering, file system can offer relative version numbering too. Users can specify negative version numbers and access versions more or less recent version. Version -1 represents version prior the current one and lower numbers (higher in absolute value) go more into the history.

2.7.2 Timestamps

Individually numbered versions are not suitable for snapshotting file systems. All files within a snapshot are from a single point in time and snapshots can be identified by the timestamp of that moment. Old version can be therefore accessible using this timestamp. Again, version identifier can be a part of the file name. At symbol (`@`) is used.

Snapshots can leverage from the fact that all files have the same timestamp and instead of adding version identification to the file name, it can be a part of the path to the file. Subtree of such a path then represent the file system or its appropriate part as it existed in the specified moment in time. Applied to current working directory, it offers transparent back-in-time view of the file system.

Timestamps of successive snapshots do not create a full sequence, the time-line consists mostly of holes between the moments a snapshot was taken. Specifying exact timestamps is complicated for the user. File system usually offers to specify any moment in the history and then first snapshot before specified moment is presented. This is the data that existed in that moment in time.

2.7.3 Tags

Tags are user defined version names. They are used in combination with numbers or timestamps or are used instead of them. When used in combination, they add extra layer to the version identification. Version meta-data then contains identification of a file version(s) it points to. If used separately, the methods to identify the version are the same, it just replaces the original identifier.

Common use of tags is to create a single version of multiple files within the file-level versioning file system. All files contained in the tagged set are from a single point in time and this basically created the illusion of a snapshot.

2.8 Version limits

Storage space is finite, so most be the amount of space occupied by old versions. File system can either leave the burden of removing old versions to the user or

it can apply *retention policies*. This is a set of rules specifying how long versions are stored. Versions that violate any of the rules are deleted.

2.8.1 Number of versions and age

Basic retention policy is to specify how many versions are kept or for how long. When a new version is created and the limit for the number of versions is reached, oldest version is removed. Traversing a directory during version creation can be costly and this process may be postponed. When a time interval is specified in retention policy, separate *cleaning daemon* has to be implemented. It regularly checks whether any version exceeds the specified duration.

2.8.2 Landmark versions

Oldest version is not always the best candidate for deletion. User can modify a file for a long time, creating versions every day. As the time passes, it is less important to store all versions from every hour and day and it is possible to remove some of the granularity of versions stored.

File system can first remove all but one version within each hour. The version that is left is a landmark version. Later, the interval can be a day, even further a week and so on. This allows storing versions over a long period of time, without requiring a significant amount of storage space, even for often modified files.

User should be able to specify landmark versions explicitly. If tagging is used with a combination of other automatic versioning method, tagged files can be treated as landmark versions, thus not deleted.

2.8.3 File sets

Users often modify multiple files that are logically interconnected, parts of multi-file projects. If a retention policy automatically landmarking versions is applied, all files within such a file set must have their landmark version from the same time. If a landmark version of one file relies on a version of other file, this must be a landmark version too.

File system has to either offer multi-file tagging or the possibility for the user to request such a behavior for a file set. This can be done using a configuration file or simply by existence of a specific file in the directory (see section 2.10).

2.8.4 Combination of rules

File system can apply multiple retention policies and delete a file version when breaking any of the specified limits. Retention policies can specify not only the maximum values, but the minimum ones too. These minimums ensure that at least some of the versions are preserved, even though they would by deleted otherwise according to other policy.

Common combination of rules is a number of versions and age. Having set only maximums, it is possible to end up with many but very recent versions for

a file that is modified often or with no old version at all for a file that was not modified for a long time. Specifying a minimum number of versions to keep and minimum age a file version to be when deleted avoids such situations.

2.8.5 Retired versions

So far in text, old versions exceeding a retention policy were deleted. Storage space occupied by them can be, at least partially, freed by compressing them instead. This approach can prolong the time old versions can be stored within the file system, prior having a significant impact on the storage capacity available for current versions. Compressing can be implemented in the cleaning daemon and specified in the retention policy, thus creating a third value.

In cases, when it is important to store old version primarily as a backup and regular access to them is not required, they can be moved to other storage. This process can be a part of backup procedure - first backup current status, then backup any old versions that are going to be deleted.

2.9 Speed and storage optimization

Versioning file system is designed to access current versions most of the time and speed optimization focuses on these operations.

2.9.1 Undo log

Copy-on-write, copy-on-change and journaling are used to store the versions effectively and some performance degradation is allowed in access to them. Current versions are therefore stored as if any versioning is not implemented at all. Reconstruction of an old version of a file starts from the current version and version data is then traversed back in time, until the specified version is reached. This modified data is called undo log.

Every time, only parts of the file are modified, unless the whole file content was previously modified. File system then, in fact, discards all the data it was already prepared. If it can detect such a version before the reconstruction starts, it can speed up the process by starting at the last complete version of a file that is newer than the specified version.

Complete versions of a file can be deliberately stored by the file system itself. This can limit the amount of time and I/O needed to reconstruct an old version, but requiring more storage space.

2.9.2 Redo log

Journaling file systems use write-ahead technique, to store the modified data in the log (journal) before it is directly modified. It helps them to recover from a system crash[5]. Write operations are either interrupted during the log write and

[5]Many databases use this technique too.

the file system is still in consistent state, or during the write to its final location. In that case in is read from the log and written to the final location again.

Using such log for versioning requires storing oldest version of a file and reconstructing newer versions by applying logged data. This is called redo log. Performance impact of the access to current versions can be avoided by storing the current version along the oldest. These two versions are them accessible directly, those in between are reconstructed.

Instead of deleting the oldest version when it breaks the retention policy, only the appropriate log is applied to it, resulting in an oldest preserved version according to the policy. The log contains the modifications of all files, so the age retention policy can be easily applied. As the oldest entries from the log are deleted, they are first applied to the files they belong to. Implementation of other policies is more complicated and requires selectively deleting the data from the log.

2.10 Customizable versioning

File-level versioning does not need to version all files within the file system. It can offer the user to specify which files version and which not. This can be achieved by a configuration file, utility or using special flag in the file or its parent directory attributes. Configuration file does not have to be system-wide, file system can detect a presence of a particular file in the directory, for example. All files within such a directory and its subdirectories are then versioned.

2.11 Features summary

Tables 2.1 and 2.2 summarize analyzed file systems with respect to the versioning features mentioned in this chapter. In case I was not able to confirm specific feature from public information, a question mark is used.

Additional column in both tables shows features that are implemented in zlomekFS. Implementation details are described in chapter 4.

	File-11	Wayback	ext3cow	CopyFS	HTFS	WAFL	Sprite LFS	NILFS	zlomekFS
File-level	X	X		X					X
Block-level			X		X	X	X	X	
Copy-on-write	?	?	X	X		X	X	X	
Copy-on-change	?	?							X
Version data sharing	?	X	X		?	X	X	X	X
Direct version access	X		X		X				X
Separate utility for versions		X		X		X		X	
Version names - numbers	X			X					
Version names - timestamps			X			X		X	X
Number of versions retention	X				X				
Age retention									X
Landmark versions								X	
File sets				X					
Undo log	?	X							
Redo log	?					X	X		X
Customizable versioning	X				X				

Table 2.1: Features of analyzed file systems (part 1)

	Elephant	CVFS	PersiFS	Versionfs	CheckFS	VSCS	SVN	zlomekFS
File-level				X			X	X
Block-level	X	X	X		X	X		
Copy-on-write	X	X	X		X	X	X	
Copy-on-change				X				X
Version data sharing	X	X	X	X			?	X
Direct version access	X	?	X	X				X
Separate utility for versions	X	?	X		X	X	X	
Version names - numbers				X			X	
Version names - timestamps	X	X	X		X			X
Number of versions retention				X				
Age retention		X		X				X
Landmark versions	X							
File sets	X						X	
Undo log		X						
Redo log			X	X				X
Customizable versioning							X	

Table 2.2: Features of analyzed file systems (part 2)

Chapter 3

File system backup

One of the reasons to implement file versioning in zlomekFS is to offer consistent backup of the file system. This chapter analyses different ways to take backup of a file system and discusses the possibilities of using file versioning.

3.1 Objectives

Designing file system backup requires setting objectives to be met. Different expectations have impact on the ways a backup can be done. This section covers two most common criteria for backup.

3.1.1 Recovery point objective

Recovery point objective (RPO) is the moment in history to which data can be restored. It defines the possible time interval that might not be covered by the backup and gets lost in case of an accident that requires restoring the file system. Shorter the time allowed to lose data, more complicated backup strategy has to be used.

Ideally, there is a zero-loss scenario. This requires making synchronous copy of the file system, separated from the source file system. Synchronization can occur to separate storage within one hardware system, to separate system in local environment (LAN) or to remote separate system (WAN or fiber). Such backup covers failure of specific parts of the storage (usually separate hard disks), failure of the whole storage or system hardware or accidents on the site (e.g. fire or flood), respectively.

3.1.2 Recovery time objective

Recovery time objective (RTO) is the time needed to restore the data. One of the prerequisites of short RTO is a well-described restore plan. This consists of procedures to be performed and tools to be used.

3.2 Methods

Backup can be performed using different methods. They differ by the complexity of the backup process itself, storage space required for the backup and, of course, by their RPO and RTO.

3.2.1 Full backup

Basic backup method is to periodically copy the contents of the whole system to backup media. This method can be simply applied to any file system. Compared to other methods, it offers best recovery time among all methods. All data is directly accessible during recovery and it consists of simple copy, same as taking the backup. Disadvantage of this method is that it requires most storage space of all methods.

3.2.2 Incremental backup

Incremental backup is trying to lower the requirement for storage space, comparing to full backup. Initial full backup is taken, followed by periodical incremental backups. During incremental backup, only changed data (files, data blocks) since the last full or incremental backup is copied. This minimizes the amount of data that is transferred with each backup, especially for data that is changed only occasionally. Depending on the detection of changes, it decreases time needed to take such a backup as well, comparing to full backup.

More backups result in higher recovery time, as all incremental backups have to be processed apart from initial full backup. Data that was changed multiple times are overwritten with each incremental backup that covers such modification. The solution is to take full backup after a number of incremental backups. Usual strategy is to take full backup during weekend and incremental backup each working day.

3.2.3 Differential backup

Differential backup is similar to incremental backup, it relies on initial full backup and copies only changed data. Instead of copying only changes since that last "fast" backup, it copies all changes since the full backup. This approach shortens recovery time only to process initial full backup and only one differential backup. Drawback of it is that requires more storage space as the data changed early since the initial full backup and not later are copied every time to differential backup. Same strategy, with a regular combination of full and differential backups is usually applied.

3.2.4 Synchronous copy

Methods mentioned so far rely on copying data from the file system in specific moment of time. These techniques do not offer zero-loss data storage in case

of an accident. Nevertheless, this can be achieved, using synchronous copy. It is used to mirror data storage content to a separate storage. Primary storage sends all write requests to a secondary storage, apart from processing them itself. Only when both storages store written data, either to permanent storage, non-volatile memory or battery-backed-up cache, response is returned. This method thus creates backup copy that is always up-to-date and offers best recovery point possible.

3.2.5 Asynchronous copy

Asynchronous copy process write requests similar to synchronous copy, but primary storage responds once the request is stored by this storage, without waiting for secondary storage. All non-processed requests are stored in local cache. This method offers same recovery possibilities as synchronous copy, but continues to work even when secondary storage is not accessible (e.g. network disconnection). Once the storages are connected again, all waiting requests are processed and up-to-date copy is again created. It is usually possible to define maximum time for such a disconnection, based on recovery point objective.

Another advantage of this method is that it is faster compared to synchronous copy, as network latency add up to the processing time on the secondary storage.

3.2.6 Versioning

Versioning is not a backup method per se. It can be used to cover backup needs, if they consist only of the ability to recover from user errors or retrieve older versions, as it is sometimes used. It does not storage data more reliably than current versions are stored, in respect to software or hardware failure.

3.3 Rotation schemes

Every backup uses a part of the storage that is eventually reclaimed for another backup copy. Unless WORM is used to store backups or infinite age of the data is required in the backup, backup rotation scheme is defined. It specifies how long individual backups are kept.

3.3.1 Grandfather-father-son

Grandfather-father-son is a very common backup rotation scheme. It is used in many modifications, but it basically defined three different time intervals to keep backups. One backup (grandfather) is typically a monthly full backup, another (father) is a weekly full backup and the last (son) is an incremental daily backup. Grandfather copies are not used in data recovery, its sole purpose is to have old version of data available. More such copies from different months can be stored, even at remote locations (they do not have to be available immediately).

3.3.2 The Tower of Hanoi

More complex, but cost-effective, rotation schema is called The Tower of Hanoi. It is based on the puzzle with the same name. It uses only full backup method and offers to preserve history up for 2^{n-1} sessions with only n copies. It effectively uses first copy every other session, second copy every fourth session, third copy every eighth session and so on. It shares the same advantages and disadvantages with full backup method.

3.4 Extraction of data

Every backup operation copies selected data from the file system to backup storage. Data can be access at different levels, with or without the support of the file system or underlying storage.

3.4.1 File copy

Copying data on a file level is the simplest way of taking backup. Selected files, depending on backup method, are copied to backup storage. Backup storage sometimes uses other format than source file system, e.g. stores backup to tar archive.

Incremental backup relies on file attributes, either modification time or special archive flag, if it supported by the file system. Such flag is set by the file system every time file is modified. Backup software then clears the flags, once the file is copied, either during full or increment backup. Differential backup does not clear the flag as it will copy the file again, until the next full copy.

Both, incremental and differential backups can copy only modified part of files. File system that stores modification log can make such logs accessible to backup software. This approach can be useful for text files, but makes backup and restore process more complicated.

During copy, read lock is acquired. Until the file is completely copied to the backup storage, applications have to wait for exclusive (or write) locks. This can affect the performance of the system backup is taken.

File copy has to deal with open files. Copying its contents might use inconsistent version of the file and thus making file restoration impossible. Easiest way is to avoid the problem by scheduling the copy to later time, e.g. after other files are copied. As this may work for short-timed opens, e.g. when an application is writing new version of the file, it does not work for applications that keep files open for a longer time. Typical such applications are databases, their backup is discussed in subsection 3.4.5.

Unless combined with versions or snapshots, it is not possible to take consistent backup of mounted file system with file copy method. *Consistent backup* contains data of all files from the very same moment in time. This requirement can be refined to be valid only for logically-connected group of files, e.g. project source files.

3.4.2 Block copy

Instead of copying individual files from the file system, whole file system is copied block-by-block. This copy is called *disk image*. It is created without the knowledge of the file system, thus it has to be unmounted during the copy. Block copy is faster than file copy, but it can be used only where file system does not have to be available continuously.

Incremental and differential backup method can be applied to block copies. Blocks are either marked by the file system, similarly to file archive flag or hash values are computed and compared with the copy.

3.4.3 Versions

File system that supports on-close file versioning can be backed up consistently combining older version of files. Open-file problem is less frequent in this case and simple file copy can be performed. Depending on the version naming (numbers or timestamps) backup software can select versions directly or has to list all available versions and select appropriate one based on its timestamp.

Even though timestamp version names seem to be better for backup purposes, this applies only to full backup. Incremental and differential backup is easier with numbered names as they can be simply compared with initial full backup versions.

3.4.4 Snapshots

Snapshots are a special case of versioning. Here, snapshots with block-level access are discussed. File systems with such a feature can be backed up using block copy, while the file system is still mounted. File system can make separately available modified data and allow using incremental or differential backup method.

As snapshots are file-system-wide, they suffer from the open-file problem and it cannot be solved using technique described in subsection 3.4.5.

3.4.5 Database backup

Database files are often backed up while they running, thus using file copy that does not require to unmount the file system. May files are open while the database is running. This makes them difficult to copy in consistent state. Backup software may copy the file contents during multi-block update, which would make the copy useless for restoration. If the database can be shut down during the backup, files can be copied freely. This is called *cold backup*.

Most of the databases are backup-aware. That allows the backup software requesting the files from the database for a limited period of time and backup files in consistent state. Oracle database, for instance, allows putting individual tablespaces into backup mode. Database commits all cached writes to the file and closes it. Further modifications are stored only in redo log files. Once the backup software releases the file, database replays the redo log into the tablespace. This is called *hot backup*.

3.5 Special files and meta-data backup

Backup software should recognize files such as soft links, hard links, sparse files, devices, sockets and pipes to back up and restore them correctly. Some of them can be excluded from backup by design.

Files do not consist only of data, they have meta-data associated with them, such as permissions, owner and group identification and ACLs. Meta-data that is stored in directories or in separate files is backed up via these entities.

3.6 Optimization

Backup copies contain a significant amount of data, full copies in particular. Reducing their size and the time needed to take a backup can result in taking backup more often and thus improving RPO.

3.6.1 Compression

Compression is a common method applied to backup data storage. It reduces storage space required and depending on the hardware and storage location it can shorten the time needed to take a backup. It can be applied all abovementioned backup methods.

3.6.2 Encryption

Access to file contents is controlled by the file system. Once the data is copied to the backup storage, file system loses the control of them. If a backup is mounted to the same or another file system with the same level of security, file access can be applied again, using meta-data stored in the backup copy. If it is, however, accessed without applying the access control (e.g. mounting a file copy on another system where the user has administrator rights), it poses a security risk. Backup copies can be encrypted to avoid this problem. Encryption can be implemented in the backup software or storage hardware, as it requires inconsiderable system resources.

3.6.3 Data sharing

Backup copy can contain blocks of data with identical contents or individual backup copies can contain identical files or data blocks. Backup software can take advantage of this and store such data only once. Deciding whether similar data already exists in any of considerable backup copies can take some time and this can make backup process slower. Backup software may therefore create a regular copy to a temporary storage and perform the optimization while copying the data to the final backup storage.

3.6.4 Sparse files

Incremental and differential file copy methods store only changed data. Sparse files can be used to store the changes, if only parts of modified files are backed up. This approach is similar to storing versions in sparse files, described in subsection 2.5.1.

3.6.5 Performance impact

Taking a backup requires to read data from the file system it is backed up. This utilizes resources on the system and affects regular operations. Apart from a slowdown, applications can suffer from file locks created by the backup software or are required not to work with their (database) files. Ideally, the file system offers file versions or snapshots that can be used over longer period of time and backup copy operations can be performed with lower priority. Otherwise, backup process should take as less time as possible.

3.7 Restore

Backup copies are useful only when it is possible to restore the data if needed. It gives a false sense of security otherwise. Storing backup on the same storage device as the source file system, within the same system, room or building can result in losing backup copy along with the original data.

Restore process is performed on files and file systems that are not in use. This simplifies the whole process. Full copy along with incremental or differential copies, if created, is restored to defined location, not necessarily the same as the source. Restore uses the same method as backup, either file or block copy.

Chapter 4

Implementation

This chapter describes how versioning is implemented in zlomekFS, based on the text in previous chapters. It outlines the important parts of the implementation of the changes. It can be used as a reference for future improvements of zlomekFS.

4.1 Versioning in zlomekFS

Chapter 2 offered multiple approaches in many details in the file system versioning support. Analysis of the possibility to incorporate them into zlomekFS and suitability for the thesis goals, resulted in implementation of some of them in zlomekFS.

4.1.1 Limitations

zlomekFS is not a stand-alone file system, it stores its data as files within the underlying file system. File data blocks are not accessed directly, requests for data are transformed into operations with regular files of the underlying file system. Only the file file-level versioning can be reasonably implemented.

4.1.2 Versions

Every time a file data or attributes are changed, a new version file is created. It contains file data and has file attributes of the file before the change occurred. Multiple writes within one open-close pair are grouped together into one version file. When the first write into file occurs and the new data differs from the data in the file, zlomekFS creates a version file and copies original data to the version file (copy-on-change technique), unless data was already copied and current write rewrites data already written since the last open. This version file is created on the underlying file system and is accessible to the user.

Version file does not always contain all file data. It contains only copy of the data that was modified, even though its size is set to the size of original file before the first write occurred. If only part of the file was modified, the version file contains parts of data that were not written into the file. Such a file is called

sparse[1]and these unwritten parts are called *holes*. This reduces the amount of disk space required to store version files. Details of modifying data and creating of version file is described in Figures 4.1 and 4.2.

Write into a file that was truncated before opening does not generate a new version, as it was already created during truncate. This operation is described in subsection 4.1.6.

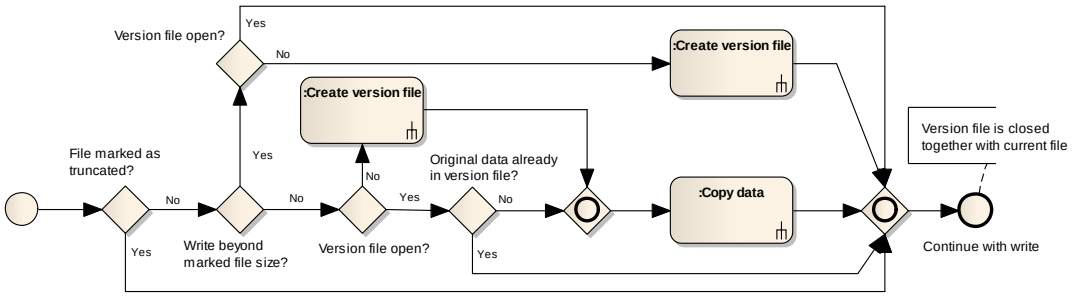


Figure 4.1: Write data

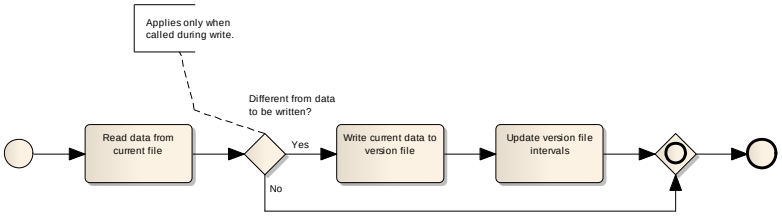


Figure 4.2: Copy data into version file

When the file is closed, zlomekFS closes corresponding version file as well. Along with the version file, version interval file is created. It contains list of intervals that are valid in the version file. As there is no easy way to detect holes in the file, version interval file supplies this information. This helps to read old version data, because only version interval files are read to find version files containing requested version of data. Interval trees were already implemented in zlomekFS to store modified data intervals, I used this code for version intervals. If the version file covers whole file, i.e. version file contains all original data, interval file is not stored to disk. Details of synchronizing version data to disk is described in Figure 4.3.

4.1.3 Version naming

My first idea was to use suffix with a semicolon and a number to identify a version. It is a very straightforward solution and easy to use. Problem aroused with

[1]Sparse file contains empty data blocks that were not written into the file. File system does not store these empty data to disk, it only stores information about empty blocks in the file metadata. This reduces disk space required by the file.

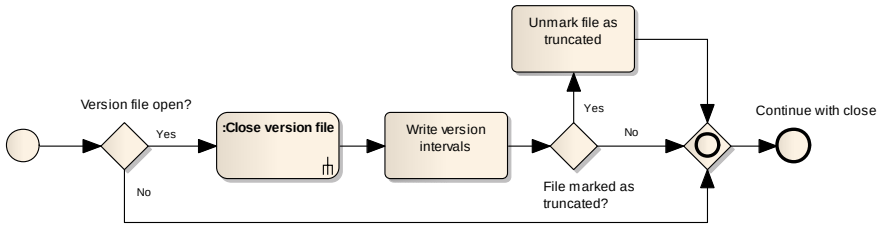


Figure 4.3: Close file

local changes in disconnected mode. New versions would have been numbered in context of a local node and then renumbered during reintegration, if the file had been changed on other node since the last connection. This could confuse the user and complicate the access to old versions.

Due to this problem I decided to implement version names using a time stamp. Every version file is internally identified by a Unix timestamp[2] of the moment it was created. This time stamp is a part of the version file name. Time stamps define the moment in time when the file data has changed and the corresponding version file contain the data as it existed prior to that moment.

Current version is accessible by its sole name, old versions require a suffix of the at-symbol (`@`) and a time stamp in `yyyy-mm-dd-hh-mm-ss` format. User time stamp is converted to a Unix timestamp and back. Partial time stamp, with the rest of the detail missing, is allowed. First month/day/hour/minute/second of particular year/month/day/hour/minute is used in such a case. Specifying `2009-10`, for example, equals to `2009-10-01-00-00-00`. File version that existed in specified moment in time is provided to the user.

Time stamp with one-second granularity can create ambiguity, as more file version could be created within one second. zlomekFS therefore stores only the last version from within one second. Such versions would be indistinguishable from each other to the user anyway[3]. If a new version has to be created and a version file from within the same second already exists, it is opened and version data is added. Details of version name generation are described in Figure 4.4.

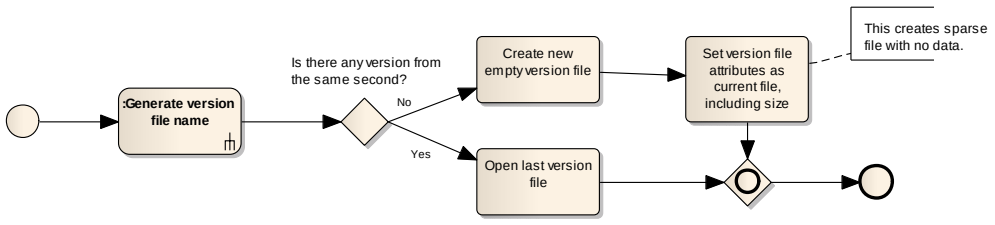


Figure 4.4: Create version file

[2]Number of seconds since the Epoch - 00:00:00 UTC, January 1, 1970.

[3]Implementation can be easily altered to offer better granularity, e.g. one microsecond.

4.1.4 File attribute modifications

Directory data modifications, i.e. creation or deletion of a file within a directory or a change of its attributes, are versioned with the files themselves. File attribute modifications, except for the file size change, create sparse version file with no data that has attributes of the original file.

4.1.5 File size modifications

File size modifications are processed in three different ways. If the new size is larger than the current size, only sparse version file with original size is stored. If the new size smaller but non-zero, all data beyond new size are copied to the version file. Truncation of a file, i.e. setting its size to zero, results in rename of current file to version file and new empty current file is created instead, with rest of the attributes same as the original file. This is faster than copying all file data to the version file. Details of file size changes are described in Figure 4.5.

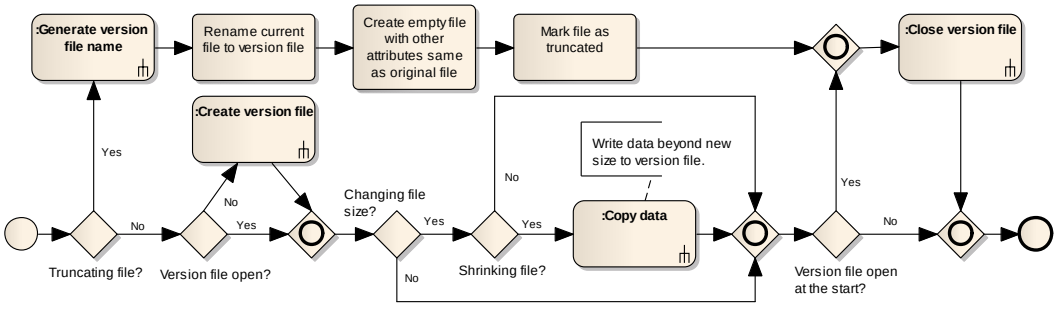


Figure 4.5: File size modification

4.1.6 File truncate

During testing of versioning in zlomekFS I found out that many applications that modify the file contents start with file truncation and then write new data. In fact, I had to create simple testing application that was writing only into some part of a file to test functionality of interval files. zlomekFS does not receive `O_TRUNC` flag with `open` call, this is handled in kernel. Interaction between kernel and a file system during `open()` call is described in Figure 4.6. This sequence of separate `setattr` and `open` calls would create two versions. First containing all the data before truncate and second empty version file. It is therefore important to detect file truncation and when a file is open after truncate, it should not generate additional version file. Truncate operation marks the directory entry (Figure 4.5) as truncated and until the next close all the following write operations do not generate a version file (Figure 4.1).

Another aspect of a truncate is that version file always contains complete file data. Data in the version file and in current file are identical up to the start of the first modification. This could be detected and version file can be rewritten to

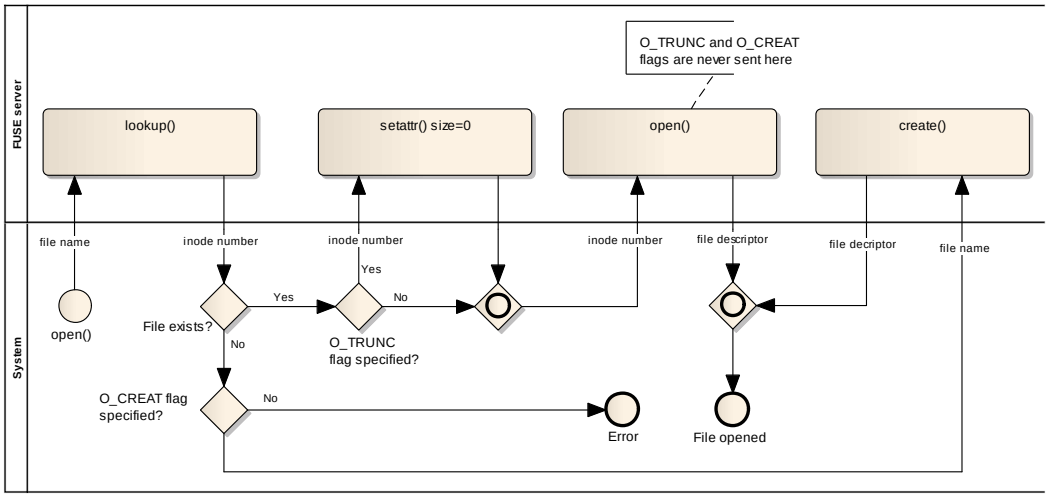


Figure 4.6: System `open()` call

create hole in the beginning of the version file. This could save disk space opposed to more resources required to create a version file. Current implementation of versioning in zlomekFS replaces truncate with rename to version file and create of an empty file, as described in Figure 4.5. Data is not compared afterwards.

4.1.7 Rename and delete

Similar approach as for truncate is used for delete (`unlink`) and rename of a file. No data is copied during delete, current file is simply renamed to a version file, as described in Figure 4.7. During rename, source file is always versioned the same way as if deleted. If destination file exists, it is versioned as deleted as well. Details of rename are described in Figure 4.8.

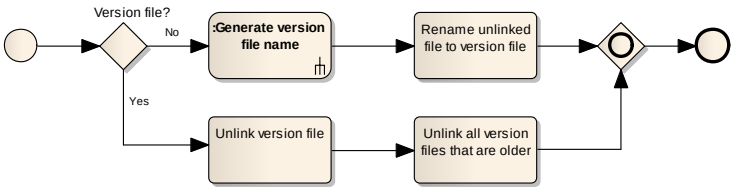


Figure 4.7: File delete

4.1.8 Directory listing

File versions can be present in directory listing. This depends on system-wide configuration as one of `zfsd` parameters. If not set there, it can be enabled locally for a specific directory by presence of special file `.verdisplay`. Version files are then displayed along current versions of files. Details of reading directory data are described in Figure 4.9.

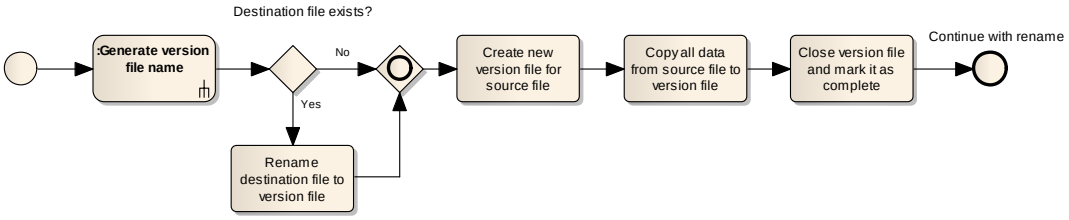


Figure 4.8: File rename

Directory versions can be accessed using the same way as files. All files and directories within the directory tree are then implicitly from the same moment in time as was specified for the directory. Directory content does not display version times as opposed to directory listing with versions. Every file is present only with its appropriate version. Accessing subdirectories results in applying the same time as specified for the parent. Specifying a time stamp to a directory creates virtual snapshot of that directory tree as it existed in the specified moment in time. This snapshot contains valid[4] data and is accessible as long as version data from that moment in time still exists.

Whole directory content is not always returned as a result of one system call. Once the communication buffer between the kernel and the file system is full, it is returned together with an identification of the position in the directory data that was already returned. This value is called *cookie* and it is passed as an input for successive call. First call for the directory listing passes zero value for the cookie. This value is used during listing of versioned directory to detect whether prepared list of files can be returned or version files have to traversed again.

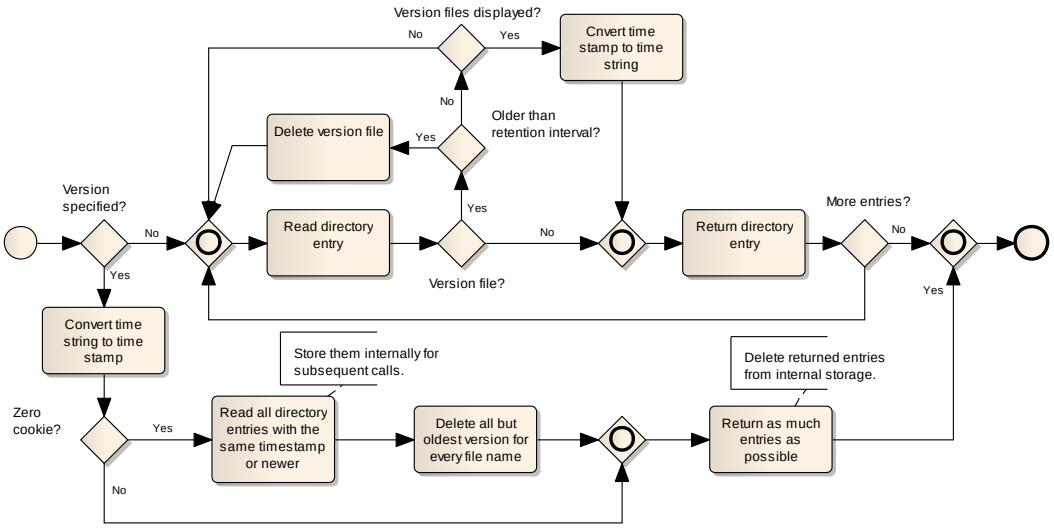


Figure 4.9: Directory listing

[4]in respect to its accuracy

4.1.9 Old version data

Version files can be accessed by specifying additional time stamp suffix. Version file with exact time stamp does not have to exist. zlomekFS looks for the version files that have the same or newer time stamp and uses the one with the oldest time stamp, as described in Figure 4.10. Lookup will succeed if current file with the specified name or any version file matching this condition exists.

Search for appropriate version file is always based on the version file time stamp. File modification time is not used for versioning, as it can be modified by the user. Modification time, if not altered, specifies when the data in the version file was created and the versioning time stamp specifies when the data was overwritten.

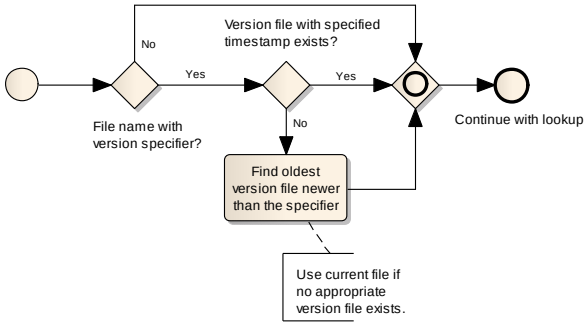


Figure 4.10: File lookup

Version files does not always contain all file data. If a version file is open, its interval file is read as well. If the version file does not cover whole file size, first newer version file is checked. Its intervals are added to the data coverage and this process continues until the file is covered with version data. Current file is treated as a newest and complete version file. This is described in Figure 4.11.

Opening of a version of a file always start with exact version file, as this is found during lookup. Open is called with inode of the version file that matches time stamp specification.

Once version intervals are read and stored during open, read calls can follow. Read of requested data can consist of multiple reads from different version files, according to version intervals, as described in Figure 4.12. Writing to version files is not allowed.

4.1.10 Directory attributes and removal

Versioning does not apply to directory attribute modifications and to their removal. User operation `rm -r` *`directory`* is translated into directory listing (`readdir`), followed by deletion of individual files (`unlink`) and finally by delete of the directory (`rmdir`). When deleting a directory that contains version files, those have to be deleted prior the directory itself. zlomekFS does not delete them automatically.

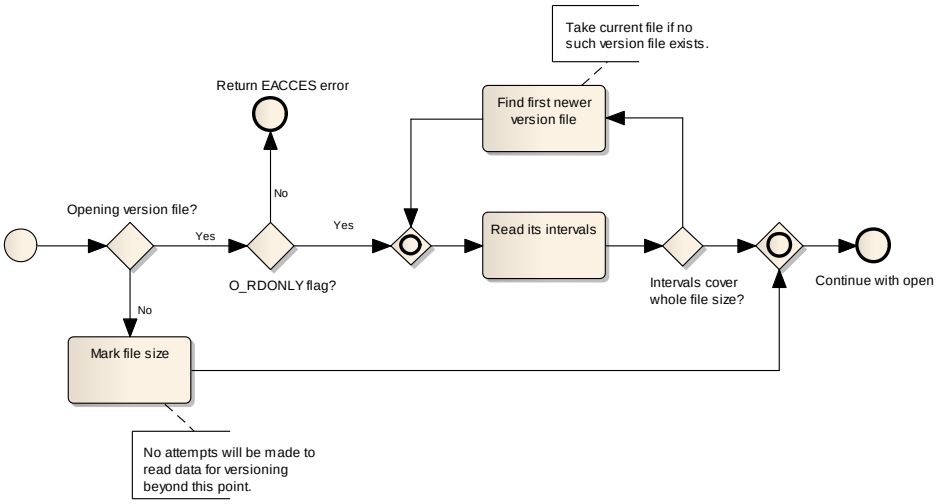


Figure 4.11: Open file

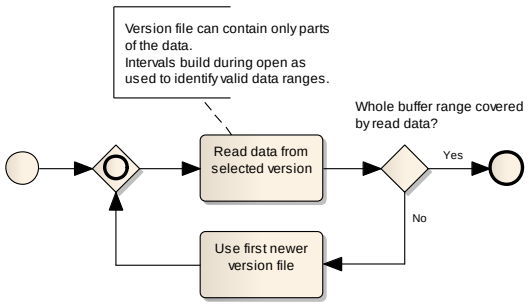


Figure 4.12: Read version data

If version file display is not set, version files are not visible and are not removed. Such a directory cannot be removed. To access and delete versions even when version files are not visible, zlomekFS offers `@versions` time stamp that displays all files that have a version file in the directory. The oldest version of every file is displayed. Calling `rm` *`directory`*`@versions/*` deletes these versions of every versioned file. When no versions exist anymore, directory itself can be deleted.

4.2 Version retention

Versioning in zlomekFS implements age version retention. Retention period is specified as a `zfsd` parameter. Retired versions that should be deleted are detected during directory listing, as shown in Figure 4.9. They are immediately deleted. If a directory is commonly accessed, no additional actions are required to check for retired version files. To apply retention policy for a whole file system, simply run `ls -Ra` *`rootdir`* regularly.

When a version file is removed, older version file either does not exists or it

is deleted as well. Automatic version retention cannot create inconsistent combination of version files when some of the versions in the middle of the version list is missing. After retention is applied, every file looks like it had no changes older than the retention period. Oldest accessible version is therefore the first newer version than the retention period.

4.3 Distributed editing

zlomekFS allows to connect to remote nodes and access remote data. When the data is not cached and all operations are performed on remote node, versioning is done by the node that stores the data (master node) and no further modifications are required to implement versioning. This differs in cached mode and this section depicts this situation.

4.3.1 Cached mode versioning

Multiple versions of a file can be created in distributed environment that allows caching and disconnected modifications. Two possible approaches of cached data versioning follow:

Approach A

Create versions only on master node, disconnected modifications are not versioned. Reintegration occurs into current master version. Conflict are handled the same as without versioning.

Approach B

Create versions on disconnected local copy too. In this case reintegration can reorder master copy versions. Reintegration of old file versions when no other modification of the same file exists since the modifying node disconnected from the master node is straightforward, versions are only copied to the master node. Combining it with versions created on other node during the disconnection would create a sequence of versions that do not link to each other.

Versions simply ordered according to their timestamps without resolving conflicts between them would not make a sense. However, resolving conflicts for all versions and not only for the current version of a file is not sensible. Such versions are therefore renamed and node name is added to the version name. Resulting suffix is `@nodename:yyyy-mm-dd-hh-mm-ss`. It is still possible to use a suffix without a node name, but this extended format searches only in file versions that existed on the specified node.

Conflict can be only in the current version. Old versions are kept from all nodes with their original timestamp, without any conflict resolution.

4.3.2 Impact of versioning

Versioning implemented in zlomekFS as described in section 4.1 replaces file with a new one in case of truncate. However, inode numbers of the underlying file system are used to identify the file. File handles consist of node ID, volume ID, device, inode and generation numbers. Implementing versioning in cached mode would require one of the following modifications:

- resign to renaming current files to versions and copy all data when a file is truncated
- change zlomekFS file handles

First modification results in worse performance, especially when most of the modifications are done using truncate. Modification of a file using truncate and write of similar amount of data as the original file triples the amount of data transferred - current file is read first, it is written into version file afterwards and this all is followed by the new data write.

Second modification requires to design new file handle and to make many changes in zlomekFS code.

4.3.3 Result

Replacing truncate with data copy would make zlomekFS very slow when versioning is active and it will degrade its performance dramatically. Changing zlomekFS file handle requires a lot of time. Current implementation of versioning therefore does not support caching of data volumes when versioning is active.

4.4 Backup in zlomekFS

Time-stamp versioning makes backup easy. Backup software decides to which moment in time it should backup the data, opens *`rootdir@timestamp`* directory and copies all files and directories in it. Specifying a time stamp creates a virtual snapshot of requested directory tree.

In most cases backup software will backup current file system contents. zlomekFS offers `@now` time stamp specifier for such purposes. It is replaced by current time and still works as a directory version. Backup software then gets consistent state of a directory tree, as it existed when the backup procedure started.

4.5 Other changes in the project source tree

4.5.1 Versioning modifications

All modifications related to versioning are enclosed by `#ifdef` precompiler directive. They are compiled into `zfsd` only when `VERSIONS` constant is defined. When the user does not use versioning, it can be left out of the resulting binary.

4.5.2 New FUSE interface

At the time of zlomekFS transformation to use FUSE interface[2], FUSE lacked some of the functionality required for complete transformation - directory invalidation and file page cache synchronization. FUSE implementation was therefore patched to add such functions. Current[5] implementation of FUSE offers this functionality. zlomekFS does not depend on patched FUSE anymore, it calls library function `fuse_lowlevel_notify_inval_inode` to invalidate file/inode data and function `fuse_lowlevel_notify_inval_entry` to invalidate directory entry.

4.5.3 Syplog

Logging in zlomekFS depends on syplog library that was implemented during one of the zlomekFS modifications. It offered only logging to file. New medium called `print` was added. It prints log messages to standard output and can be used for debugging purposes.

Individual messages sent to syplog are identified by level and facility. They can be filtered according to these identifiers. New facility called `VERSIONING` was added for messages that are generated with versioning code.

[5]since kernel version 2.6.31

Chapter 5

Evaluation

This section describes performance and space requirement comparison between zlomekFS with and without versioning. Test results contain data for the same actions performed directly on the underlying file system as well.

5.1 Setup

In the following tests, the system was Gentoo Linux 2.6.31 in VMWare environment with both cores of Intel Core2 1.83 GHz CPU and 1 GB of RAM assigned to the virtual machine. Disk operations were performed on external IBM Travelstar hard-drive connected via USB. Underlying file system used was ext3.

5.2 FUSE library build

Build of FUSE library version 2.8.1 was selected for testing. Test consisted of the following actions:

- extract the archive to the file system (`tar xzf`)
- run `configure`
- make
- make clean (not measured)
- make
- list the directory tree (`ls -R`)
- list versioned directory tree (`ls -R @now`) - only with versioning enabled

5.3 Wait time

`time` tool was used to measure real, sys and user time. As the test focuses on the performance of zlomekFS server that is called from kernel when accessing the file system, wait time for individual actions was selected as the metric. It was calculated as a difference between real time and sum of sys and user time. Every action was tested 5 times and results were averaged. The file system was remounted before every action to clean all caches. Measured wait times are shown in Figure 5.1.

Most of the overhead of zlomekFS is contained in the core code, versioning does not add significant overhead, when comparing to the original zlomekFS. This applies when manipulating with new files, with no version history. However, this overhead increases as version files accumulate. Successive make wait times did not differ when versioning was not used (either on ext3 or zlomekFS). With versioning enabled, wait times increased during library rebuild.

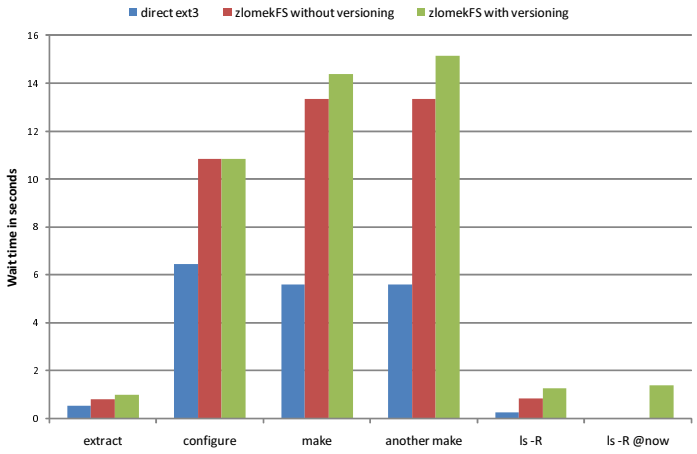


Figure 5.1: Wait time during individual actions

5.4 Disk space

Disk space required to store the files from the test build after each of the test actions finished is shown in Figure 5.2. zlomekFS without versioning generates little overhead comparing to the underlying file system. Space requirements for successive builds stay the same as for the first one. Versioning capabilities require more disk space, because it has to store additional data in version files. This amount increases as more files are modified. Version retention policy cleans retired data to keep the space overhead within user-defined limits.

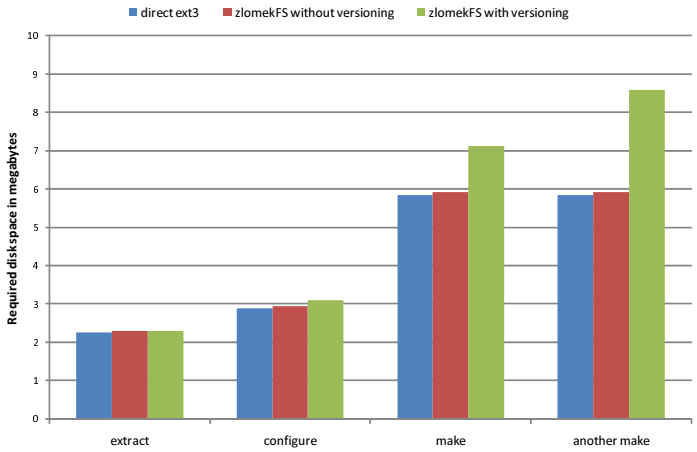


Figure 5.2: Disk space required after individual actions

Chapter 6

Conclusion

This thesis implements versioning into existing zlomekFS file system. Versioning capability was designed after analysis of existing versioning file systems. It adds similar versioning functionality to zlomekFS as is implemented in other file systems. New versions of files are created when file data or attributes are changed, using copy-on-change technique. Multiple writes between one open-close pair are grouped together into one version. Implementation uses as little disk operation overhead as possible. In the most common file modification case, truncation, it does not copy any data. It only creates one empty in place of the truncated file.

User can specify a moment in time for files and directories and zlomekFS presents them as they existed in that moment. Version files can be displayed along current files in directory listing. Age retention policy is implemented and zlomekFS removes old version files automatically.

Implemented versioning offers file system snapshots that contain consistent set of files and their content as it existed in the specified moment in time. This can be used for backup purposes. Any change in data within requested directory tree will not be presented to the backup software during performing a backup.

Implementation of versioning does not support cached mode and modifications in disconnected mode. Existing zlomekFS implementation relies on local inode numbers to identify files and this is in conflict with reasonable performance impact of versioning. Extending versioning to disconnected mode will require redefining this file identification and modifications throughout zlomekFS code. This thesis allows to combine versioning with local or direct remote access only, but describes possible versioning schema for conflicting modifications and access to such versions.

6.1 Future work

Natural continuation of this thesis is the extension to cached mode and modification of file identification, as mentioned above. Possible version naming schema is described in this thesis. Version files can be created on data-caching nodes and later reintegrated into master data. Current version naming is designed to support mixing of version data from multiple nodes.

Another improvement would be to detect file modification when data is inserted in/deleted from the file body and the rest of the file is shifted to the end/start of the file. This improvement can decrease the amount of disk space required for versions. This applies to source code in particular, where most of the file modifications are such a category.

Bibliography

[1] Josef Zlomek: Shared File System for a Cluster, http://shiva.ms.mff.cuni.cz/svn/zzzzzfs/trunk/doc/Zlomek-SharedFileSystem.pdf

[2] Miloslav Trmač: zlomekFS Over FUSE, https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/trunk/doc/Trmac-zlomekFSoverFUSE.pdf

[3] Jiří Zouhar, Regression Testing For zlomekFS, https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/trunk/doc/Zouhar-RegressionTesting.pdf

[4] Filesystem in Userspace (FUSE), http://fuse.sourceforge.net/

[5] Wayback: User-level Versioning File System for Linux, http://wayback.sourceforge.net/

[6] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, Erez Zadok: A Versatile and User-Oriented Versioning File System, http://www.eecs.harvard.edu/~kiran/pubs/versionfs-fast04.pdf

[7] Austin T. Clements, Dan R. K. Ports, Ben A. Schmeckpeper, Hector Yuen: PersiFS: A Continuously Versioned Network File System, http://pdos.csail.mit.edu/6.824-2005/reports/amdragon.pdf

[8] Versioning file system, Wikipedia, http://en.wikipedia.org/wiki/Versioning_file_system

[9] OpenVMS User's Guide , Second Edition, HP Technologies, Digital Press, 1998

[10] Ext3 Copy-On-Write File System (ext3cow), http://www.ext3cow.com/

[11] CopyFS, A copy-on-write, versioned file system, http://n0x.org/copyfs/

[12] NILFS, a log-structured file system, http://www.nilfs.org/

[13] SCO OpenServer Documentation, http://osr507doc.sco.com/en/Navpages/index.html

[14] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch: Elephant: The file system that never forgets, http://www.hpl.hp.com/personal/Alistair_Veitch/papers/elephant-hotos/elephant.pdf

[15] Craig A.N. Soules, Garth R. Goodson, John D. Strunk, Gregory R. Ganger: Comprehensive Versioning File System (CVFS), `http://www.hpl.hp.com/personal/Craig_Soules/papers/fast03.pdf`

[16] CVS - Concurrent Versions System, `http://www.nongnu.org/cvs/`

[17] Version control with Subversion,`http://subversion.tigris.org/`

[18] Dave Hitz, James Lau, & Michael Malcolm: File System Design for an NFS File Server Appliance, Network Appliance, `http://www.netapp.com/library/tr/3002.pdf`

[19] Mendel Rosenblum and John K. Ousterhout: The Design and Implementation of a Log-Structured File System, `www.cs.berkeley.edu/~brewer/cs262/LFS.pdf`

[20] Checkpoint and Block Level Incremental Backup, LinSysSoft Technologies, `http://checkfs.linsyssoft.com/`

[21] How Volume Shadow Copy Service Works, Microsoft TechNet,`http://technet.microsoft.com/en-us/library/cc785914.aspx`

[22] Backup Encyclopedia, Genie-Soft, `http://www.backupencyclopedia.com/glossary/default.html`

Appendix A

Enclosed CD

Data on enclosed CD is structured to directories as follows:

- src - current source code of zlomekFS including versioning retrieved from source repository
- thesis - text of this thesis (PDF and source files - LyX and graphics)

All this data can be found in my branch of zlomekFS SVN repository at `http://shiva.ms.mff.cuni.cz/svn/zzzzzfs/branches/wartiak`.