Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Miloslav Trmač
# zlomekFS over FUSE

Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Study Program: Computer Science

I would like to thank Doc. Ing. Petr Tůma, Dr., my supervisor, for his advice and his extraordinary patience.

I declare that I have written this master thesis on my own and listed all the used sources. I agree with lending of the thesis.

Brno, August 5 2007 Miloslav Trmač

Contents

Abstract

Název práce: *ZlomekFS Over FUSE*
Autor: *Miloslav Trmač*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *Doc. Ing. Petr Tůma, Dr.*
e-mail vedoucího: *tuma@nenya.ms.mff.cuni.cz*
Abstrakt: *ZlomekFS (ZFS) je distribuovaný systém souborů, který umožňuje práci se soubory i v případě odpojení počítače od sítě. Změny provedené v době, kdy je síť nedostupná, jsou po připojení k síti automaticky synchronizovány.*

Původní implementace ZlomekFS používala vlastní ovladač systému souborů pro operační systém Linux. Tato práce tento ovladač nahrazuje použitím rozhraní FUSE, které bylo přidáno do Linuxu až po dokončení původní implementace ZlomekFS. Pro implementaci ZlomekFS pomocí FUSE bylo rozhraní FUSE rozšířeno o potřebné operace pro správu cache v jádru operačního systému. Tato rozšíření rozhraní FUSE byla implementována pro operační systémy Linux a FreeBSD.
Klíčová slova: *systém souborů, správa cache*

Title: *ZlomekFS Over FUSE*
Author: *Miloslav Trmač*
Department: *Department of software engineering*
Supervisor: *Doc. Ing. Petr Tůma, Dr.*
Supervisor's e-mail address: *tuma@nenya.ms.mff.cuni.cz*
Abstract: *ZlomekFS (ZFS) is a distributed file system, which allows working with files even if a computer is disconnected from the network. The changes performed while the network is not available are automatically synchronized after connecting the computer to the network.*

The original ZlomekFS implementation was using its own file system driver for the Linux operating system. This work replaces the driver with use of the FUSE interface, which was added to Linux only after the original ZlomekFS implementation was finished. To implement ZlomekFS using FUSE, the necessary kernel cache management operations have been added to the FUSE interface. These FUSE interface extensions were implemented for the Linux and FreeBSD operating systems.
Keywords: *file system, cache management*

Chapter 1

Overview

1.1 ZlomekFS

ZlomekFS[1] is a distributed file system. Unlike a simple network file system used on a local network, ZlomekFS supports local caching of volume contents. When a computer is disconnected from the network, it can use its local volume cache and continue to use the file system, reading it as well as modifying it.

After the computer is reconnected to the network, the local volume cache is synchronized with the primary volume copy, propagating the file system changes in both directions. If the file system changes conflict, the conflict 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.

ZlomekFS was implemented as an user-space daemon and a kernel module. The daemon communicates with other computers over the network and implements all file system operations. Volume caches are stored as directories located on other file systems (e.g. `ext3`) mounted on the local computer.

The kernel module is a Linux file system driver, which also implements a character device (usually `/dev/zfs`). It implements file system operations by serializing them (using a modified version of the ZlomekFS network protocol) and making them available to user-space. The daemon opens the `/dev/zfs` device, handles the operation requests, and returns results to the kernel.

1.2 FUSE

Implementing most of a file system in user space in a manner similar to the above-described ZlomekFS implementation has several important advantages over a pure kernel-space implementation:

- The whole operating system is significantly more resilient to bugs in the file system implementation.
- Debugging of the file system driver is easier in user-space because most debugging and process inspection tools cannot be used on kernel-space tasks.
- Most of the file system implementation is not Linux-specific and can be quite easily ported to any POSIX-compliant operating system.

Therefore, Linux since kernel version 2.6.14rc1 provides FUSE[2], a generic interface for implementing file systems in user-space. FUSE consists of a kernel module, an user-space library for communicating with the kernel module, and a utility which allows mounting user-space file systems by unprivileged users (if permitted by the system administrator).

1.3 Goals

The primary goal of this thesis is to remove the ZlomekFS-specific kernel module and to modify the user-space daemon to use FUSE for processing file system operations. This has the following advantages over a ZlomekFS-specific kernel module:

- It reduces the long-term effort necessary to maintain ZlomekFS. The Linux kernel is well-known for its explicit decision not to keep backward compatibility (not even an API compatibility) for kernel modules[3], while the FUSE interface has been revised several times, always without breaking binary compatibility.
- The FUSE kernel module is widely used (the FUSE wiki currently lists 82 file systems using FUSE), so it will be likely maintained and enhanced by developers not participating in ZlomekFS development.
- FUSE provides the mechanisms necessary for using ZlomekFS without system administrator cooperation, and for using several ZlomekFS file systems on a single computer (the ZlomekFS kernel module provides a single `/dev/zfs` device, which cannot support two separate ZlomekFS mounts). Although the ZlomekFS kernel module could in principle be extended to add these features, they are already available in FUSE and porting ZlomekFS to FUSE will automatically make them available to ZlomekFS users.

The current FUSE interface is insufficient for ZlomekFS implementation; therefore, the thesis is focused mainly on the design of the FUSE interface extensions. In addition to the unavoidable interface extensions necessary for implementing ZlomekFS, other possible FUSE interface extensions are examined to make the FUSE interface general enough for implementing a wide range of different file systems.

Finally, the possibility of implementing ZlomekFS on operating systems other than Linux is considered. The port of ZlomekFS to FreeBSD is described, and possibilities of porting ZlomekFS to some other operating systems are described as well.

Chapter 2

File System Interface Overview

To be useful in practice, an user-space file system interface should fulfill the following requirements:

- The interface should support all operations customarily provided by file systems on the target platform. The common operations provided by UNIX operating systems are standardized by POSIX and described in more detail in section 2.2. In addition to these operations, some other operations are available on several operating systems.

- The interface should support, or at least not prevent, performance enhancements, like caching or sending requests to remote network servers in batches.

- The interface should use an efficient data representation which is, if possible, easy to work with, both in the kernel and in the user-space driver.

- The interface must allow a secure implementation: e.g. the interface should not require exposing of kernel pointer values to the user-space driver. Interfaces which allow mounting of arbitrary user-space file system drivers by unprivileged users, like FUSE does, also need to protect the kernel and applications run by other users against malicious file system drivers. Thus, the interface must not allow arbitrary access to memory of a process which performs operations on a file system with an user-space driver, for example.

Fulfilling the above requirements by FUSE is strongly influenced by the kernel file system interface: a file system operation or a caching mechanism that cannot be supported by the kernel is useless. Therefore, this chapter reviews several widely used or otherwise significant file system interfaces, focusing on operations they support and the caching mechanisms they use, and the ways they might be represented in FUSE or used to enhance FUSE.

Network file systems are more suitable for user-space implementation than other file systems because the network latency is usually at least comparable to, if not significantly greater than, the overhead of the user-space file system interface. That's why the most commonly used network file systems are also briefly examined to make sure that FUSE can be used to implement efficient clients for these network file systems, reusing the long-term effort to optimize these protocols and their caching mechanisms.

2.1 Data Representation

Before reviewing specific file system interfaces, some common issues related to data representation are described in this section. The data that need to be represented by any file system interface include:

- Raw data, i.e. the bytes transferred using the `read` or `write` system calls. This data should, obviously, be transmitted unchanged.[1]

- Operation flags and parameters (e.g. the `O_*` flags for the `open` system call, or `struct flock`).

  This data can usually be transmitted without major modification, using the same ABI which is used for system calls on the platform. This avoids the necessity to translate data between different formats in the kernel. For user-space file system interfaces this guarantees the interface will remain sufficiently stable, because an user-space file system driver which can run on a specific platform can also use the kernel interface.

  Using "the same ABI" for an user-space file system interface is, unfortunately, ambiguous on systems with support for more than one ABI in the kernel. Other than genuinely different ABIs on a single platform, this includes "multilib" systems, running 32-bit user-space processes on a 64-bit kernel. On those systems, the kernel contains a layer which translates the system calls and data structures using the legacy ABI to the current one. To maintain the correspondence between system call ABI and the user-space file system interface, it would therefore be necessary to implement a similar translation layer in the kernel file system

[1]Representing raw data might be more difficult on platforms which don't use 8-bit bytes, but such platforms are extremely rare. 8-bit bytes are required by POSIX since the 2001 revision.

driver, and to enable or disable its use depending on the ABI used by the user-space driver. This removes the main advantage of using the system ABI for the user-space file system interface: if the kernel file system driver must contain a translation layer, it would be simpler to use a translation layer unconditionally.

- File[2] names. Unlike other operation parameters, which are comparatively uniform, the capabilities of the underlying file system affect the available file system names and the semantics of their representation.

- "Kernel" data structures. Apart from data structures directly related to file systems, this may include other data structures (e.g. a representation of processes, in order to check whether the process invoking the operation is allowed to perform a file system operation). In a typical kernel-internal file system interface, these data structures are directly referenced by passing pointers, which is obviously unsuitable for an user-space file system interface.

  The main data structures that need to be represented in a file system interface are a representation of files (whether open or not), and a representation of the state associated with open files (which usually includes at least the current position within the file data).

2.2 POSIX File System Interface

The fundamental document for FUSE interface design is the POSIX specification[4], which describes the file-system operations that should be supported by an operating system.[3]

2.2.1 POSIX File System Operations

Although POSIX defines several file system interfaces at higher levels of abstraction (e.g. `FILE`, `glob`), they can all be implemented using the following low-level operations:

[2]POSIX explicitly includes directories in the meaning of "file". The same convention is used throughout this thesis.

[3]POSIX does not address the issue of file systems in wide-spread use (e.g. FAT) which cannot support all the requested operations. Such file systems therefore must not be used on a system installation that claims POSIX compliance. Nevertheless, applications must be able to handle such file systems in practice, and FUSE should support implementation of such file systems.

- Operations on whole files: `access`, *`chmod`, *`chown`, `link`, `mkdir`, `mknod`, *`pathconf`, `readlink`, `rename`, `rmdir`, *`stat`, *`statvfs`, `sync`, `symlink`, *`truncate`, `unlink`, `utimes`.
- Operations on opened files: `aio_`*, `closedir`, `close`, `fcntl` file locking, `fsync`, `ioctl`, `lio_listio`, `lseek`, `mmap`, `msync`, `munmap`, `opendir`, `open`, `poll`, `posix_fadvise`, `posix_fallocate`, *`read`*, `readdir_r`, `seekdir`, *`select`, `telldir`, *`write`*.
- Kernel-internal data manipulation: `dup`*, `fcntl` (except for locking), `getcwd`, *`chdir`, `umask`. Implementation of these operations doesn't require any support from file system drivers.
- Kernel-local operations: `pipe`*, the socket API. These operations are handled in other parts of the operating system unrelated to the file system interface. To the extent these operations may affect file system contents (e.g. `mkfifo`, binding `AF_UNIX` sockets to local paths), they can be implemented using other file system interfaces (e.g. `mknod` for both of the examples above).

2.2.2 POSIX Data Representation

Portable file names in POSIX contain only characters in the *portable filename character set*, which includes ASCII letter and digits, period, underscore and hyphen. File name length limit is file-system specific, but not less than 14. The special file names "`.`" and "`..`" refer to the "current" directory and its parent directory, respectively. File names in a path name are separated by slashes. Path names starting with exactly two slashes are unportable, and may be interpreted using completely different rules (even using a different file name separator character).

Open file states are represented using *file handles*, integers allocated by the operating system (using a specified algorithm). Each process has its own file handle name space, but a single open file state may have several file handles referring to it; the file handles may all be defined within one process, or they may be in several separate processes. File handles in multiple processes that refer to a single open file state are usually created using the `fork` system call. UNIX also supports a mechanism to transfer an open file state to a different process, using an `AF_UNIX` socket: the file handle of the source process is resolved to an open file state, and a new file handle in the target process referring to this open file state is created. Curiously, POSIX requires a definition of the `SCM_RIGHTS` ancillary data message type used for such open file state transfer, but doesn't specify any semantics of the data.

References to files within a file system are usually performed using path names or file handles. In addition, each file has an integer *inode number*[4], returned in `st_ino` by the `*stat` system calls. Each file within a file system must have a different inode number. Inode numbers can't be used to reference files in system calls; the application must use a path name or a file handle. Thus, inode numbers are usually used to detect symbolic link loops or races when interpreting path names in user space, and less often to detect hard links (because the inode number is available in `struct dirent` returned by the `readdir` function, but to check `st_nlink` in `struct stat` the application would have to use a separate `*stat` system call).

2.3 Linux File System Interface

This section describes the Linux[5] file system interface. It is neither the first generally used file system interface, nor the most widely used interface, nor an user-space file system driver interface; nevertheless, it is described first because Linux is the primary platform of ZlomekFS, it is the base of the FUSE interface and because describing the Linux interface first will simplify the description of the FUSE interface.

2.3.1 Linux File System Operations

Linux file system drivers provide several sets of functions that operate on kernel objects. Some of the functions are mandatory; many are optional and work as hooks for the file system layer to handle less usual file systems.

The file system interface is tightly integrated with the kernel "caches".[5] At the lowest level lies the *inode cache*. A `struct inode` represents a single file, whether currently used or not. Each file in a file system has at most one entry in the inode cache. Besides data used by the generic file system layer, it caches information that is returned by the `*stat` system calls. Thus, for a typical block device-backed file system, the information would be read once when initializing the inode, and `*stat` only copies the data to user space. The inode cache layer handles writing out the data when it is modified and manages the amount of unused inodes that are kept in the cache, depending

[4]Inode number is the traditional UNIX name of the identifier. It is called a *file serial number* in the POSIX specification.

[5]The word "cache" usually means a system component that can in principle be removed without impacting the functionality of the system; the Linux "caches" contain currently used data together with currently unused data, and it is not possible to remove any of the "caches".

on memory pressure. To do so, the inode cache uses functions (e.g. read inode from disk, write inode to disk) provided by the file system driver in `struct super_operations`. Optionally, the file system driver may replace the default implementation of inode allocation and deallocation (usually to embed the `struct inode` within a larger data structure and use the extra space to store file system-specific data), or it may be notified when inode in memory becomes modified (e.g. to preallocate space in a modification journal). Finally, it may be notified when the inode is no longer used; the file system driver can e.g. immediately evict the inode from the inode cache.

Because the file system driver may be notified whenever the metadata of an inode is read, whenever an inode is modified and whenever an inode stops being used, the file system driver can prevent most of the caching. The most important function of the inode cache is thus not caching of the inode metadata, but unique file identification. The core kernel code, with very few exceptions, does not track any file system-specific inode identification data; currently referenced files within the kernel and on the interface between the kernel and the file system are simply identified using the `struct inode *` or a higher-level data structure that references a `struct inode *`.

A similar mechanism is used for whole file systems: a file system is identified by a `struct super_block *`[6], the file system layer tracks modifications of the file system metadata and uses `struct super_operations` to write the modifications out. The only difference is that file system metadata is not cached when the file system is unmounted, thus there is no "super block cache".

Above the inode cache is a *dentry cache*, which caches name lookups. Each dentry represents one directory entry; the dentries form a tree which represents a subset of the possible absolute paths (not containing symbolic links) that can be used to designate a file. The cached information is either an inode reference, or an information that no file exists with that name. Because each dentry contains a link to a dentry representing its parent directory, a reference to a dentry implicitly describes "the" absolute path that refers to the file within the containing file system.[7] The availability of the path information makes dentry references more useful than inode references in some situations, so dentries are often used instead of inodes to refer to files.

The file system driver may customize the dentry cache behavior by implementing several hooks. Most significant is the ability to replace the hash func-

[6]The name is derived from a "super block" on the first UNIX file systems, a disk block with a fixed location on the block device that contained file system-global metadata. The unintuitive name `struct super_operations` is a shorthand for "super block operations".

[7]The names in dentries are modified by the `rename` system call, so the path might be different from the path used to open the file.

tion and name comparisons, which allows e.g. case-insensitive name lookups. Similarly to the inode cache, the lookup caching may be restricted because the file system is notified when a lookup is satisfied from the dentry cache (with the possibility to override the cached results) and when a dentry reference count becomes zero.

Each inode refers to a `struct inode_operations`, a set of functions provided by the file system driver to perform operations on the inode as a whole. Typically, a file system driver has separate `inode_operations` structures for different file types (e.g. one for regular files, one for directories, one for symbolic links). Most of the provided operations closely match the POSIX operations on whole files, with the following exceptions:

- Two lower-level operations are provided: `lookup`, used during path name lookup to fill in a dentry for an entry of a directory, and `create`, used to create regular files.

- The *`chmod`, *`chown` and *`utimes` system calls are provided by a single `setattr` operation.

- The *`truncate` system calls are implemented using two operations: First, `setattr` is called with the new file size. Then the affected page cache pages are removed, and, if provided by the file system driver, a `truncate` operation is called. The `truncate` operation deallocates disk blocks on block-based file systems, and is called after the page cache does not refer to the blocks to make sure the page cache does not contain data for a single disk block on two different pages. The `setattr` operation, called before purging the page cache, can be used by journaling file systems to create a “truncate” transaction to make sure the file system will remain consistent even in the event of a system crash while removing page cache pages (which might block until I/O in progress is completed).

- The `getxattr`, `listxattr`, `removexattr` and `setxattr` operations are provided to support extended attributes of files. ACLs, if supported by the file system, are manipulated using these operations on attributes within the reserved `system.*` name space. Enforcing ACLs is performed in the driver’s `permission` operation, corresponding to the `access` operation in POSIX. If the file has a SELinux context label, it is stored as an extended attribute as well.

- The `follow_link` and `put_link` operations are provided for resolving symbolic links. The generic `readlink` operation copies the path contained in a symbolic link to a caller-provided buffer. The `follow_link`, put_link operation pair allows the file system driver to simply return a pointer to the path in its internal data structures, which eliminates unnecessary copying of the data, and more importantly allocating a buffer (a whole page) for the data on each lookup. For example, a disk-based file system may use the system page cache to cache symbolic link paths like any other data, and simply point to the data in the page cache.

In addition, the follow_link operation may, instead of returning a pointer to a path, directly resolve the symbolic link to a dentry reference. This is used for the symbolic links in /proc/*/fd/, which can be used to view opened files of a process. readlink on such a symbolic link returns the path to the opened file, as recorded by the structure of the dentry cache. If the file was unlinked, the returned path cannot be used to access the file (the path returned by readlink actually has a ␣(deleted) suffix), but the follow_link operation can still obtain a valid dentry reference and opening the symbolic link will succeed.

When a file system is mounted, the file system driver allocates a dentry for the root file (usually a root directory) of the file system. This allows using the lookup operation to reach any file within the file system.

File systems that can be served over NFS must implement two additional operations and provide them in a struct export_operations:[8] encode_fh creates an NFS file handle for a specified dentry, and decode_fh returns a dentry for a specified NFS file handle. File systems which store the inode number and a generation counter in the NFS file handle can use the provided default implementation, and only implement creation of a dentry with a specified inode number.

Open file states are represented by a struct file, which contains a dentry reference, kernel state such as file position, a place for file system private data, and a reference to a struct file_operations, provided by a file system driver for performing operations on opened files. To open the file, an initial struct file_operations is provided by the file system driver in the inode. The open function from these operations may replace the struct file_operations pointer in a struct file by a more specific set of operations (e.g. when opening a named pipe, different file operations are used for named pipes opened for reading and for named pipes opened for writing). The struct file_operations pointer may be replaced even after the file is opened, although this is done very rarely.

[8]The operations could easily be provided directly in struct super_operations without the additional indirection. Splitting them to a separate data structure, apart from a minor implementation simplification, allows checking whether a file system supports NFS simply by checking whether a pointer to the struct export_operations is not NULL.

The operations provided by `struct file_operations`, are, again, quite similar to the POSIX operations they are used to implement, although the interface differences are somewhat larger. The significant differences are:

- The `readdir` operation provided by `struct file_operations` actually implements a `getdents` system call, filling a memory buffer with as many directory entries as possible.[9] The `closedir`, `opendir`, `seekdir` and `telldir` operations are not provided by Linux file system drivers because the functionality is available using the ordinary `close`, `open` and `lseek` system calls.

- Three different functions can be provided to implement the `ioctl` system call. The simplest function is `unlocked_ioctl`. For convenience, a file system driver may provide an `ioctl` function instead, which is called with the big kernel lock[10] held. Originally, only the `ioctl` function was provided. If the `ioctl` operation for a specific file type turns out to be performance critical, the implementation is modified to use a more fine-grained locking mechanism and used as `unlocked_ioctl` instead.

  The third function, `compat_ioctl` is used on "multilib" systems, running 32-bit applications on a 64-bit kernel. For most other system calls, the generic kernel code can translate between the 32-bit and 64-bit data structures, but driver-specific `ioctl` operations need driver-specific data translation. The `compat_ioctl` function is called without the big kernel lock.

- As described above, there is a separate `create` operation provided by the file system driver in `struct inode_operations`, so the `open` operation does not handle the `O_CREAT` flag.

- Two operations are used to implement the `close` system call. The split is necessary to correctly handle open file states with more than one associated file descriptor or other reference (e.g. a memory-mapped region of the file). The first operation, `flush`, is called on every `close` call. A file system driver can write out all modified data to ensure

[9]`readdir`, which returns one entry at a time, is implemented in GNU libc using `getdents`. This decreases the total number of system calls necessary to transfer the data to user space.

[10]A recursive spin lock or mutex (depending on the architecture), which was used to make sure only one task is running kernel code at a time when Linux was being ported to multiprocessor platforms. Nowadays it is mostly used in legacy code where replacing the big kernel lock use by a more local synchronization mechanism is not worth the effort.

that if the data cannot be written to the underlying file system, the application calling `close` will receive an error code—there would be no other opportunity to notify the application about the error if this were the only file descriptor available to the process calling `close`.

The second operation, `release`, is called when there are no more references to the open file state and the open file state is being freed. If the file system driver is keeping private data in the open file state, it can provide this operation to deallocate the data.

- A `fasync` operation notifies the file system that a value of the `FASYNC` flag has changed for the file state. If supported by the file system, the process "owning" a file state with the `FASYNC` flag receives a signal when the underlying file can be used without blocking[6]. Whether a file is considered usable without blocking should be consistent with the values returned by `poll`.

- The `sendfile` and `sendpage` operations are variants of the `read` and `write` operations for files using the page cache. While `read` copies the read data to user space, `sendfile` calls a specified callback function with a reference to the memory page containing the data. Similarly, `sendpage` writes data specified by a reference to a memory page instead of copying data from user space. These operations can be used by the `sendfile` system call to copy data from one file to another (usually from a regular file to a network socket) without copying it to user space and back.

  The `splice_read` and `splice_write` operations provide a similar functionality, using a pipe as an intermediary buffer. The `splice_read` operation "reads" data from a file by collecting the relevant pages from the page cache and attaching references to them to a destination pipe. The `splice_write` operation writes data contained in a pipe to a destination file. These operations are used to implement the `splice` system call. Compared to `sendfile` and `sendpage`, the `splice_read` and `splice_write` operations can, using the intermediate pipe buffer, operate more than one page at a time (possibly reading from the input file and writing to the output file in parallel, using separate threads). For file systems that implement `sendpage` and can't benefit from operating several pages at a time, a generic version of `splice_write`, implemented using `sendpage`, is available.

  There is one more difference between the `sendfile` / `sendpage` and `splice_read` / `splice_write` operations. The `sendfile` system call is expected to fail with `EINVAL` if it cannot perform the operation without copying the data, so the `sendpage` operation should not be implemented by file systems which would copy data from the supplied page. On the other hand, such file systems may implement `splice_write`, by using a generic implementation which copies the data.

- The `get_unmapped_area` operation can be provided by file systems with additional requirements on the addresses passed to `mmap`. The main user of this hook is `hugetlbfs`, used to map physical memory (reserved in advance for the purpose) using higher-level page table entries.

- The `check_flags` operation can be used to impose additional restrictions on open file state flags modified using the `fcntl(..., F_SETFL, ...)` operation.

- The `dir_notify` operation is called when an user-space process requests to be notified about modifications to a directory. `dnotify`, the user-space interface to this feature, has been obsoleted by `inotify`, mainly because to use `dnotify`, an application has to keep the watched directory open, which makes the watch intrusive because file systems with open files cannot be unmounted. `inotify` does not need any support from the file system driver because it does not report directory modifications performed on a remote file system by the server or other clients.

- The `lock` operation can be used to override the default (local) implementation of the `fcntl` file locking. The `flock` operation can similarly override implementation of the `flock` system call.

The following POSIX operations cannot be implemented by file system drivers:

- The `posix_fadvise` operation, which advises the kernel about future use of a specified part of a file. Linux uses this information to tune read-ahead for file systems that use the page cache, uniformly across all file system types. All file systems therefore automatically benefit from the advisory information without any additional code.

- The `posix_fallocate` operation is not implemented in any released Linux kernel, although an implementation has been merged and will probably be available in Linux 2.6.23.

- The pathconf operation is not implemented in Linux. A subset of the information can be obtained using the *statfs* system calls.

The unavailability of the posix_fallocate and pathconf operations does not automatically mean Linux cannot be POSIX compliant, it would be sufficient to implement them in the C library. GNU libc provides an inefficient, but correct implementation of the posix_fallocate operation. The pathconf operation is provided as well, but not all requested information is available to the calling application.

Quota files, if supported by the file system, are handled specially: they are not cached, and on journaling file systems modifications to the quota database should be performed in the same transaction as the related file system operation. The file system driver therefore must provide, at minimum, specialized operations to read and write data from quota files. It also must cooperate with the quota database before allocating and after freeing data blocks and inodes.

Linux file systems usually cache file data in memory, using the *page cache*: data contained in each inode is logically split into page-sized chunks, and each chunk may be present in the page cache. The file operations used by the page cache code to read and write data are provided by the file system driver in a separate struct address_space_operations[11]. At minimum, a read-write file system driver needs to implement the following operations:

- The readpage operation starts (possibly asynchronously) reading file data to a page.

- The writepage operation starts (possibly asynchronously) writing a modified page to the underlying file.

- The prepare_write and commit_write operations are used to implement the *write* system calls. The prepare_write operation is called before copying data from user space, and the commit_write operation after copying the data. Both operations are called with an exact specification of the byte range to be modified, not only a reference to the affected page.

  On block device-backed file systems, the prepare_write operation usually makes sure the relevant blocks are allocated, and if the write operation partially affects blocks which contain older data, it reads them

[11]The page cache implementation is somewhat independent of the file system layer, referencing "address spaces" instead of inodes. In practice, there is exactly one non-inode address space, used to keep track of anonymous pages that are both stored in the swap space and present in memory.

to the page. The `commit_write` operation marks the relevant blocks dirty.

Using these operations, instead of only `readpage` and `writepage`, allows overwriting existing files without having to read old data from the file system if the *`write`* system calls are suitably aligned. File systems drivers can also use the more specific information, e.g. to write only the affected blocks to a journal instead of writing the whole page, or to send only the modified data to a network server instead of sending the whole page.

Other operations may be implemented by the file system driver if a very tight integration with the page cache is necessary.

2.3.2 Linux Data Representation

Operation flags and parameters use the system call ABI, to the extent it is relevant; many data structures and constants are used only for the Linux file system interface. If the platform supports several system call ABIs, or several versions of a particular interface (the layout of `struct stat` is a notorious example: four versions of `struct stat` are supported on the x86_64 platform, three of them only for compatibility with i386), the generic kernel code converts the data to a single representation which is used in the file system driver interface.

File names are strings of arbitrary bytes, terminated by the NUL character. The special file names "`.`" and "`..`" are handled in the generic code, so the file system driver does not need to implement them, and it cannot override their semantics. The file system driver still must return them among the entries returned by the `readdir` operation, though.

File names in a path are separated by slashes and no special handling is performed for the POSIX-reserved path names starting with two slashes. The file system driver interface does not use path names; path name lookup is performed in the generic code, using the `lookup` operation to resolve separate path components. The result of a path name lookup is a dentry reference, which uniquely identifies the path within a single file system.[12] All file system operations reference files using dentries and inodes. Path names for files which might not exist yet are represented as a dentry referencing the parent directory and a file name within that directory. The `symlink` and `readlink` operations are an exception: the symbolic link contents are passed through

[12]To identify the path within the whole set of valid absolute paths, a reference to a `struct vfsmount` representing the mount point of the file system is necessary because a single file system can be simultaneously visible under more than one mount point.

the generic file system code as an uninterpreted NUL-terminated string. This matches the UNIX practice, where arbitrary data can be stored in a symbolic link, even if it does not currently form a valid path name. It also allows special symbolic link semantics within a file system (e.g. “variables” within the path, which are substituted by the host name or the architecture of the system).

Other than handling slashes, NUL, “`.`” and “`..`”, the generic code does not try to interpret the file names in any way, and it makes no assumptions about the character encoding used for the file names. On some systems, two or more encodings might be used at the same time, possibly within a single file system, or even within a single path.

This is simple and easy to implement for local, UNIX-like file systems, but network file systems, file systems such as NTFS or VFAT which store file names in Unicode, and file systems with case-insensitive file name matching need to know the used character encoding. In Linux, this was traditionally solved by using a file system-specific mount option to specify the encoding. Because the only practical way a generic application can handle file names is to assume they are all using the character encoding specified by the `LC_CTYPE` locale category, the character encoding should be a system-wide (or at least chroot-wide), not a per-file system parameter. This is supported in Linux using the `CONFIG_NLS_DEFAULT` compile-time option, which is used to provide a default character encoding if no mount option is specified.

As described above, files within a file system are identified using paths or dentry references. The generic file system code does not work with inode numbers at all, it only copies them from data provided by a file system driver to user-space. Providing unique inode numbers on some file systems is very hard, though: some file systems, including ZlomekFS, use file identifiers too large to fit within in the inode numbers, and on some file systems, e.g. a virtual file system which allows “mounting” a remote FTP server, files can be identified only by their paths. If there is no way to algorithmically derive a unique inode number from a file identifier, the file system would need to maintain a mapping of file identifiers to inode numbers.

If maintaining the mapping requires allocating storage for each mapped inode number, the POSIX requirement implicitly limits the number of files on a supported file system by the storage space necessary to store the inode number mapping (after a single (`find /`*mount_point*) command, the inode number for each file is selected and it must be preserved until the file system is unmounted). This makes the POSIX requirement impractical for very large file systems (in the extreme case, using such a file system would require using additional disk space for storing the mapping because it can’t fit in the system memory), therefore some file systems don’t provide the required guarantee.

On Linux, a subset of the requirement is typically implemented by using the inode cache to maintain the mapping between files and inode numbers: as long as the file is in the inode cache, its inode number is constant. When a file is not in the inode cache, it is added into the cache and assigned an inode number which doesn't conflict with any other file in the inode cache. As long as a single application does not store inode numbers for a very long time, and the system is not under extreme memory pressure, applications will usually not see two different files with a single inode number; on the other hand, it is likely that some application will eventually notice a single file with two different inode numbers. This implementation is used e.g. in the Linux `smbfs` and `vfat` file systems.

2.4 FUSE Kernel Protocol

FUSE uses `/dev/fuse`, a character device, for communication between user and kernel space. Before mounting a FUSE file system, the `fusermount` utility opens `/dev/fuse`, which creates an unassociated device context. If this succeeds, `fusermount` mounts a `fuse` file system to the desired mount point, passing the `/dev/fuse` file descriptor as a string among other mount options, which associates the device context with the mounted file system. The file descriptor is then transferred over a socket to the main user-space file system driver.

File system requests from the kernel are queued in the device context and the user-space driver can read them using the `read` system call. Each `read` system call copies exactly one request to user space (if the request is larger than the supplied buffer, the request is aborted). After processing the request, the user-space driver uses `write` to pass a response to kernel; again, each response must be copied using exactly one `write` system call.

The user-space driver may read more than one request from the kernel before sending any response, and it may return responses out of order. Each request and response contains a 64-bit request identifier to allow matching responses to replies. If the task which created the request is interrupted by a signal, a pending request may be canceled by sending an interrupt request to the user-space driver.

2.4.1 FUSE File System Operations

The FUSE request types reflect the Linux file system interface, with operations that assume direct access to kernel data structures replaced by simpler, although perhaps less efficient, operations (e.g. `readpage`, `readpages`, `read`, `aio_read` are all represented by a single "read" request type). The following Linux functionality is not supported in FUSE:

- As described below, no write caching is supported by FUSE to prevent denial of service by malicious file system drivers. For the same reason, read-write `mmap` operations (with both `PROT_WRITE` and `MAP_SHARED`) are not supported.

- `ioctl`. Instead of being a specific operation, `ioctl` is a way to multiplex file-specific system calls with a single parameter. The parameter is often a pointer to a structure in memory, which further describes the request, and may point contain other pointers to more data structures. To implement `ioctl` in full, a FUSE file system driver would have to be able to arbitrarily read and write to the memory of the process calling `ioctl`. Because the FUSE file system drivers are expected to be potentially malicious, this capability is not provided.

  Although it is possible to design a safe subset of `ioctl` (e.g. similar to the Windows `DeviceIoControl` call), this is not necessary in practice because `ioctl` is very rarely used on regular files and directories, and other file types are handled in the kernel proper without forwarding requests to FUSE. The primary `ioctl` used on regular files and directories is `FIBMAP`, used by low-level tools to retrieve the block number used by the file system to store a specific block of a file on the underlying block device. This single `ioctl` is supported by FUSE, as a special "bmap" operation.

- `posix_fadvise`, `posix_fallocate` and `pathconf`, because these operations are not exposed by Linux to file system drivers.

- `dir_notify`, an obsolete interface specific to Linux.

- `poll`, `*select` and the `FASYNC` flag. This functionality is traditionally only used on sockets, pipes and character device files, which are all implemented in the kernel proper without help from file system drivers. Although implementing `poll` or `pselect` for regular files might be useful if file operations have high latency (which is often true for network file systems implemented using FUSE), POSIX, consistent with historical practice, currently requires that state of regular files is always reported as ready for both reading and writing.

- Disk quotas.

2.4.2 FUSE Data Representation

Operation flags and parameters use the local ABI for constants (e.g. the `O_*` flags). Composite data structures (e.g. the `struct stat`-like data returned by the "getattr") operations use FUSE-specific layout, using platform-independent types, instead of the local ABI. This reflects the Linux multilib implementation, which shares the constant values to reduce the necessary complexity of the ABI translation layer, but may use different data types and structure layouts in the 32-bit and 64-bit ABI. The user-space FUSE library shields the user-space drivers from the FUSE-specific layout, using the local ABI for representation of all data and converting the data between the local ABI and the FUSE-specific layout.

File names are represented by arbitrary NUL-terminated sequences of bytes, without specifying their encoding.

Open file states are identified by a 64-bit file handle, chosen by the user-space driver when opening the file, and passed to the driver on operations performed using the handle. Because the file handle is never revealed to other processes, the user-space driver can use a pointer to its internal data structure as the file handle. The operations on open files all also pass the necessary inode numbers and the current position within the file data (if necessary), so user-space drivers may be implemented without keeping any state for open files.

Files within a file system are identified by a single 64-bit inode number, chosen by the user-space driver. As noted in section 2.3.2, this is not sufficient to identify files on all file systems. Because the FUSE interface notifies the user-space driver when an inode is removed from the inode cache, FUSE file system drivers can implement a mechanism similar to using the kernel inode cache to maintain a subset of the mapping between file identifiers and inode numbers. Although implementing the mapping in user space is more difficult than using the inode cache by kernel-space file system drivers, the user-space driver may be able to perform the mapping more efficiently (e.g. by directly using the inode numbers of the underlying file system if they fit) and the storage necessary to store the mapping can be swapped out under memory pressure. The FUSE library also provides a simplified API that identifies files only by full paths from the file system root, keeping inode numbers completely hidden from the file system driver.

Preliminary support for serving FUSE file system over NFS is available in the FUSE repository, but it has not been merged to the Linux kernel yet. An NFS file handle consists of (only!) the lower 32 bits of the FUSE inode number and a generation number. The FUSE kernel protocol transfers the generation number, but it is not exposed in the FUSE library API. File system drivers therefore cannot specify the generation number, and cannot avoid accepting stale file handles if the inode has been deallocated and allocated to a new file. The above-described simplified API, which hides even inode numbers from the file system driver, keeps the generated inode numbers small enough to fit in 32 bits, and uses the inode generation number as higher 32 bits of the inode number. But because the mapping between file paths and inode numbers is maintained only as long as the inode is present in the kernel inode cache, NFS operations may fail with spurious `ESTALE` errors if a file has not been used for a while.

By default, FUSE file systems can be only used by the unprivileged user who mounted them. If allowed by the system administrator, files on FUSE file systems can be accessible by other users. In that case, all access decisions are performed in the user-space driver, based on the user ID, primary group ID and process ID of the process attempting to perform the operation, which are transferred as a part of each request. Note that supplementary group IDs are not passed to user space this way; there may be as many as 65536 supplementary groups, which would impose an unacceptable overhead on request processing. If necessary, the user-space driver can gather the supplementary group IDs for the known user ID from the group database and keep them in a local cache to avoid the per-request overhead. File systems implementing the usual UNIX permission model may optionally leave the access decisions to the kernel proper, using the `st_mode`, `st_uid` and `st_gid` file attributes; this handles supplementary group IDs automatically.

2.4.3 Caching in FUSE

Because FUSE request handling has considerably larger overhead than calling a kernel-mode operation implementation, results of the most often used operations are cached in the kernel. The most important case is the "getattr" operation, used to populate the kernel `struct inode` with file attributes and to update the attributes for the `*stat` operation. Therefore, the returned file attributes include a timeout specifying how long to use the returned data without performing another "getattr" request.

The second important case is the "lookup" request, used to find the inode number for a file identified by its parent directory and name. Because it would be immediately followed by a "getattr" request, the "lookup" response includes the file attributes. The "lookup" responses are automatically cached in the Linux dentry cache. The Linux dentry cache contents are dropped only in response to memory pressure, which suits the block device-backed file systems where the kernel is the only entity modifying the data. For network file systems, where the "lookup" results may become invalid without any activity of the client computer, the Linux dentry cache uses a `revalidate` callback, which allows the file system driver to declare the dentry cache entry stale. FUSE cannot forward this callback to user space, because the overhead would be very similar to simply using a "lookup" request; the dentry cache would provide negligible benefit in that case. FUSE therefore uses a simpler mechanism, again allowing the user-space driver to define a timeout for the validity of the lookup results.

File data is automatically cached in the Linux page cache. While opening a file for reading or writing, the user-space driver may specify that the page cache should be completely bypassed and the "read" and "write" requests should be passed unmodified to the user-space driver. Otherwise, the page cache is used, without any associated timeout, and the user-space driver can request invalidation of the cache for the whole file while the file is being opened. This is sufficient for basic network file systems with little or no changes to files (e.g. "mounting" anonymous FTP server), where the page cache is invalidated if the file modification time has changed since the previous open.

The Linux implementation of FUSE passes all writes to the user-space driver immediately ("write-through") to prevent a denial of service: if the operating system attempts to write out a dirty cache page, allocation of additional memory should be minimized while writing the data out. This can be handled in kernel-space file system drivers or cooperating user-space drivers, for example by preallocating enough memory in advance. Because FUSE is designed to allow untrusted user-space drivers, it cannot rely on the user-space driver to avoid such allocations. A malicious user-space driver might intentionally create separate processes ("memory hogs") that try to allocate even more memory. While completely avoiding memory exhaustion in a general-purpose operating system without significant performance sacrifices is difficult, the operating system always has the option to kill an user-space process which has allocated excessive amount of memory. With FUSE, killing the memory hogs would not solve the problem because the dirty cache pages would still remain allocated. The guilty user-space driver can avoid being killed by holding a very small amount of memory allocated; the kernel would eventually have to start killing completely unrelated processes in order to free memory. An implementation that limits the number of outstanding dirty pages per file system was proposed in February 2007, but it did not prevent users from mounting hundreds of FUSE file systems to avoid the limit.

2.5 ZlomekFS Kernel Protocol

The ZlomekFS kernel module uses a subset of the ZlomekFS network protocol to relay file system operations to the user-space server. In addition to the basic operations used in the kernel protocol, the network protocol also supports connection authentication, merging of changes on conflicts, and volume configuration changes.

Similarly to FUSE, requests to the user-space driver are queued and made available to user space by a character device. The user-space driver can read requests from the device, and write replies. Multiple requests may be outstanding at once, with a request ID used to match replies to requests. Mounting ZlomekFS is much simpler than mounting FUSE file systems because ZlomekFS is not designed to be mounted by unprivileged users and because only one ZlomekFS mount can be mounted at a time, which makes the privileged `fusermount` utility and the complex mechanism for associating mounts and instances of opening the character device unnecessary.

The ZlomekFS protocol operations are even more similar to Linux operations than the FUSE operations because the extra flexibility to handle different kinds of file systems is not necessary. The operations not supported by FUSE are not supported by the ZlomekFS protocol either, with the exception of write caching and writable `mmap` operations—again, ZlomekFS is not designed to be mounted by unprivileged users, so there is no danger of a malicious driver intentionally exhausting system memory. In addition, the ZlomekFS protocol does not support extended attributes and exporting over NFS.

2.5.1 ZlomekFS Data Representation

Like the ZlomekFS network protocol, the kernel protocol uses platform-neutral data structures and constants, independent of the local ABI, with two exceptions: The file permission and mode bits (corresponding to the `st_mode` member `struct stat`, except for `S_IFMT`), and the *flags* parameter of the "open" and "create" operations, use the local ABI to define flag values. The UNIX values of the permission and mode bits are well-known and rather commonly hard-coded in applications, so using the local ABI instead of translating the values bit by bit is safe in practice. This unfortunately cannot be expected of the `O_*` flags (the non-`O_ACCMODE` flag values differ significantly between Linux and FreeBSD, for example).

File names are specified by arbitrary NUL-terminated sequences of bytes, without specifying their encoding.

Files within a ZlomekFS file system are identified by their *file handle*, a quintuple of integers specifying the server storing the master version of the file, the volume on the master server, the device and inode on the underlying file system on the master server, and a generation number. Reserved values for some of the integers are used to carve out name space for virtual directories and directories representing merging conflicts. The ZlomekFS kernel module uses the kernel inode cache to assign inode numbers for files currently represented in the inode cache, as described in section 2.3.2.

The ZlomekFS server does not keep any state about open files for the client (in this case, the kernel): the file position is stored by the client and the "read" and "write" operations specify the position. Capabilities formed by concatenating a file handle, the granted access mode, and an authenticator, are used in the protocol to verify the client can perform the requested operation. A client receives a capability in a reply to a "create" or "open" request, and it can release it using the "close" request. If a client has more than one capability for the same file with the same access mode, they can be used interchangeably. For the client, capabilities are stateless, at least until invalidated by closing them.

The ZlomekFS server, as currently implemented, treats ZlomekFS capabilities as stateful: for each capability handed to a client, it stores a capability used for accessing the master server. The authenticator in the capability is not a cryptographic authenticator, but merely a string of random data, and a capability is considered valid if the server is currently holding state for a capability with the same authenticator.

As a network file system using the UNIX permission model, ZlomekFS needs a single name space of user and group identifiers over the whole set of interconnected systems. It implements such a name space by maintaining a shared configuration file which maps user and group names to integer identifiers, and by specifying the mapping between local UIDs and GIDs and ZlomekFS user and group names locally on each connected system. The UID and GID fields in the ZlomekFS protocol contain the identifiers maintained in the global database in structures transferred over the network, and local UIDs and GIDs in structures transferred between the kernel and the local server.

2.5.2 Caching in ZlomekFS

The basic caching mechanisms in the ZlomekFS kernel module are similar to FUSE: results of many operations return file attributes along with their main result, the page cache is used to store file data, and the data returned by the "getattr" and "lookup" requests is cached in the kernel inode and dentry cache, respectively. The timeout after which data stored in the inode and dentry cache expires is fixed, not specified by the user-space server.

Usually it is acceptable if modifications performed on a network file system by another client are visible locally only after a few seconds, but ZlomekFS uses directories to present a "user interface" for file conflict resolution: when the user selects one of the presented file versions and removes the other, the user has effectively given a command to the "file system application", and expects some response by the application—in this case, the conflict resolution directory should be replaced by the user-designated "correct" file. If the file system creates a conflict directory *X*`/file` containing versions `file-A` and `file-B`, the user removes `file-B` and returns to directory *X*, the inode and dentry cache entries relevant for `file` should be immediately invalidated; otherwise, the user would still see that *X* contains a subdirectory `file` (although in the ZlomekFS internal state, `file` already refers to `file-A`, the chosen file verson), and even worse, attempts to enter that subdirectory would fail with a "not a directory" error. Such a user experience would be very confusing and frustrating.

That's why the ZlomekFS protocol supports an asynchronous "invalidate" notification. Unlike operation requests made by the kernel to the user-space server, the "invalidate" notification is sent by the server to the kernel whenever the server modifies attributes of a specific file. The kernel module looks up the affected file by the specified file handle, and marks the cached "getattr" and "lookup" results invalid.

On "cached volumes" (volumes which have a remote master server, but store a local copy of the data), additional care is necessary to safely synchronize data modifications between the local system and the remote server. This synchronization is performed by the user-space daemon, which contacts the remote server to determine if a synchronization is necessary when looking up the file, while opening it, and before closing it (if the file is open multiple times, the check is performed on each close by a client). Before synchronizing regular files, there must be no dirty data left in the page cache, and the page cache contents must be treated as invalid after the synchronization. The ZlomekFS kernel module automatically writes out all modified data before each close of a file, and invalidates the page cache after closing the file. Invalidating the page cache after each close of the file is not strictly necessary, the page cache needs invalidating only after the local copy of the file is modified by a synchronization.

2.6 The Vnode Interface

The file system interfaces used on of BSD-derived UNIX operating systems are derived from the vnode interface[7], which was originally implemented in SunOS[8]. This section describes the variant of the vnode interface currently implemented on FreeBSD[9, 10].

The references to the implementation of the single supported file system by the rest of the original UNIX kernel were replaced by references to a "virtual file system" in the vnode interface, and references to inodes were replaced by "virtual inodes", or vnodes. The "virtual file system" code calls functions provided by file system drivers to provide the requested functionality.

Compared to the Linux file system interface, the vnode interface is simpler because the generic FreeBSD code handles only the necessary minimum (e.g. vnode reference counts and mandatory access control). Other commonly used routines are optional and used only if explicitly called from the file system driver. The only mandatory "cache" in the Linux sense is the vnode allocation pool, which lets file system drivers keep unused vnodes in memory. In Linux, on the other hand, as much as possible is implemented in the generic code, and only the truly file system-specific actions are implemented in file system drivers, even at the cost of more complex interface. For example, the Linux dentry cache is mandatory and the interface of directory operations uses dentry references; FreeBSD has a similar cache, but its use is optional and the file system driver must take care to correctly invalidate cache entries while performing directory modifications. Even basic UNIX permission checking is done in the generic code in Linux, but in the file system drivers in FreeBSD. The FreeBSD file system drivers necessarily contain a lot of duplicate, or nearly duplicate, code.

2.6.1 Vnode File System Operations

The file system driver provides file system-level operations as function pointers in a `struct vfsops`. Apart from mounting and unmounting a file system, and operations corresponding to POSIX operations, the following operations are provided:

- The `root` operation returns the vnode representing the root of the file system.
- The `quotactl`, `extattrctl` and `sysctl` operations are used for file system-specific operations on the file system invoked by system calls with corresponding names, like `ioctl` is used for file-specific operations. The generic code only translates a path parameter into a vnode

reference. Note that quotas, if supported, must be completely implemented in the file system; no generic quota database maintenance code is provided.

- The `checkexp` and `fhtovp` operations are used for serving the file system over NFS. The `fhtovp` operation returns a vnode representing the specified file handle, and the optional `checkexp` operation can be used to prevent accessing the file system over NFS.
- The `vget` operation returns a vnode for the specified inode number. This operation is necessary for the NFS server (to look up all vnodes in a directory in $O(N)$—reading names of all entries and looking up vnodes by name would take $O(N^2)$ time). Many FreeBSD file system drivers call the `vget` operation from their own `struct vfsops` instead of calling their implementation of the `vget` operation directly. Because the `vget` pointer is never modified after registering the file system, there appears to be no reason for this indirection, and it seems likely it was simply copied from one file system driver to another.

Operations on single vnodes are provided in a `struct vop_vector` referenced by the vnode. The POSIX file and directory operations are provided, with the following differences and additional operations:

- The `O_CREAT` flag is handled using a separate `create` operation.
- Two operations are used for file name lookup: the `lookup` operation is called by the generic kernel code to return a vnode for a file, given its parent directory and a file name. This operation must handle the "`.`" and "`..`" entries (except for "`..`" in a mount point). To use the optional lookup cache, the file system driver moves the implementation of `lookup` to the `cachedlookup` operation, and uses a generic implementation of the `lookup` request.[13]
- A single `setattr` operation is used to implement the *`chmod`, *`chown`, *`truncate` and *`utimes` operations.
- An `advlock` operation implements the `flock` system call and `fcntl` file locking.
- The `readdir` operation implements a `getdirentries` system call. Like `getdents`, it writes as many directory entries to the user-space buffer as possible. In addition, it returns a file offset of the first returned directory entry.

[13]Thus, the `cachedlookup` function is called when the lookup is *not* cached!

- The `whiteout` operation allows creation and removal of *whiteout* directory entries. A whiteout directory entry stores only the file name and an indication of the entry type in the directory; it has no associated content or attributes. It is used to stop lookups of a file name in an `unionfs` directory without searching other underlying file systems. Creation of a whiteout entry "deletes" a file from the `unionfs` view, and removal of the whiteout entry "undeletes" the file.

- The `lock1`, `unlock` and `getwritemount` operations can be implemented to completely replace the main vnode locking mechanism, and to associate a different mount point with file system modifications. `unionfs` and similar file systems implement these hooks to make sure the files on the underlying file system are locked as well as the virtual files managed by the virtual file system.

- The `kqfilter` operation is used to add a "watch" on file events, so that the events are reported to user space via the kqueue interface[11].

- The `revoke` operation implements a system call with the same name, making all file descriptors currently referring to a character device invalid.

- There are three operations for access to and interpretation of ACLs, and six operations for access to extended attributes. If a file has a MAC label, it is stored as an extended attribute, but modification of the label is performed using a separate `setlabel` operation.

- To implement `mmap`, the `getpages` operation reads data into the specified pages, and `putpages` writes data from the specified pages to the underlying file. Generic implementations are available for disk-based file systems, using a `bmap` operation to find which disk blocks to read to the pages, and a `write` operation to write out dirty pages. Note that the memory-mapped pages are not used as a file data cache, and file system's `read` and `write` operations use only the buffer cache.

- To use the buffer cache, disk buffers are attached to the vnode, indexed by the "logical" block number (within vnode's data blocks), and a `strategy` operation of the vnode is used to perform I/O on the buffers. This operation translates the logical block number to a physical block number (within the underlying block device) if the translation is not already cached for the buffer, and forwards the I/O request to the block device.

- To make the `read` and `write` operations more efficient, the I/O may optionally be performed in clusters of more than one block. To do this, the `bmap` operation returns, along with the physical block number, the number of blocks before and after this block that are allocated on the disk continuously (as far as it is easy to determine this information, e.g. until encountering an indirect block boundary). The file system driver can also support a limited form of online defragmentation by implementing the `reallocblks` operation, which attempts to reallocate specified blocks to be contiguous on disk.
- The `vptofh` operation returns an NFS file handle for the specified vnode.

2.6.2 Vnode Data Representation

Operation flags and parameters reuse parts of the system call ABI. The most significant exception is the representation of vnode attributes using a `struct vattr` instead of `struct stat`.

File and path names use the traditional UNIX convention, described in detail in previous sections.

The generic file system code in FreeBSD uses vnode references for file identification. Inode numbers and NFS file handles are available only if the file system can be exported over NFS.

Open file states are represented using a `struct file`. Each open file state has an associated `struct fileops` pointing to implementations of file-specific operations. A file system driver may associate an `struct fileops` with the file state while the file is being opened. Most file system drivers, however, use the default implementation of `struct fileops`, which simply calls the corresponding operations from `struct vop_vector`. Curiously, the relevant `struct file` is not available to any of the called vnode operations except for `open` and `close`, even though the `struct file` is available to `struct fileops`.

2.7 9P2000

9P2000[12] is a file system protocol used in the Plan 9 and Inferno operating systems. Unlike the traditional UNIX file system, which uses device files, even if stored on a remote file server, only as a way to communicate with device drivers on the local kernel, 9P was designed to provide a general interprocess communication mechanism on a microkernel-like operating system. When resources accessible to a process are represented as files within a single name space, manipulating the name space allows transparent resource sharing (e.g. replacing a “mouse pointer” device by a connection to a virtual device provided by a window system) and distributed operation[13].

The protocol provides only operations to create and delete files, read and modify their attributes, open and close them, and read or write data. Although the protocol is small and simple, it presents an unconventional approach to file system interface design.

The most important difference of 9P2000 from the other file system interfaces described in this thesis is its implementation of path lookup and file identification. Other interfaces are based on a file identifier which, although possibly short-lived, refers to a single file throughout its lifetime: an inode number, pointer to an inode cache member, or a file path (either absolute, or relative to a specific directory). 9P2000 uses “cursors” instead: during the initial connection to the file system server, the client obtains a *file ID* for the root directory of the file system. To reach other files, it uses a `walk` operation to obtain a file ID for a file identified by a path (not necessarily a single file name) relative to the root directory. The path operand of the `walk` operation is limited to 16 segments, for larger paths it is necessary to use the `walk` operation repeatedly. The repeated `walk` operations can be instructed not to create a new file ID for the result, but to *move* the base file ID to the new path.

The `open` operation has no file name or path parameter. It only checks client’s permission to access a file specified by its file ID and prepares the file for ID later `read` or `write` operations. The file ID is then used to identify the open file state, and it cannot be used for `walk` operations.

This decoupling of path lookups from other operations simplifies modification of the directory structure (either by writing an intermediate server, or within the Plan 9 kernel) because it is necessary to intercept only the `walk` operation, and maybe the `wstat` operation (used to modify file attributes) to intercept file renames.

In addition to the volatile file IDs, each file has a 64-bit identifier, which should be unique not only among all currently existing files on the file system, but also among deleted files and files which will be created in the future. The path ID is returned by the the `create`, `open` and `stat` operations. In addition, `walk` returns path IDs not only for the target file, but for all directories along the specified path. Like UNIX inode numbers, the path ID can only be used to check whether two paths refer to the same file, it is not possible to look up a file by its path ID.

The 9P2000 protocol supports read-only caching of data by using a “version” attribute for all files, and providing it to the client whenever a path ID is provided (the path ID, file version and file type identification together form a “qid”). In particular, the client can compare the version number returned by the `open` operation with the version number of cached data, and invalidate the cache if the version numbers differ.

Although the 9P2000 protocol is lightweight, with no obvious “warts” (e.g. compared to the large portion of the CIFS protocol which is kept or defined only to preserve backward compatibility), it has several significant deficiencies:

- On error, the server returns only an error message (a string in UTF-8). The contents of the string are completely arbitrary, so the client cannot automatically handle specific errors (for example, a logging system cannot automatically delete old log files if the file system runs out of space). Because the file server has no information about the language used by the ultimate human user, it cannot return an appropriate error message.

  In the worst case, both the client application and the user only know that something has failed, and neither has any information about the cause of the problem.

- There are only two ways to open files: the `create` operation creates a new file, and fails if a file already exists, and the `open` operation opens an existing file, and fails if it is not present. The protocol does not provide a way to atomically open a file, creating it if it does not exist. The Plan 9 kernel attempts to support the functionality by trying to use the `open`, `create`, `open` requests in order, but that can still fail if another process is concurrently creating and deleting a file with the specified name. In that case, the request to open a file, and create if it is not found, would fail with a nonsensical “file not found” error.

- File IDs, used to identify the “cursor”, are allocated by the *client.* This might require additional code on the server because the server cannot use “natural” identifiers (e.g. an index within a static table) for shared state, and it is forced to use a data structure storing a mapping between state identifiers and server’s data.

  In theory, allocating identifiers by the client should simplify the client’s code: some very simple clients might use only two or three file IDs during the whole lifetime of the process, using compile-time constants instead of variables to refer to file ID values. In practice, simplification of client code is unlikely.

  The most important 9P2000 client in Plan 9, the kernel, mounts many separate 9P2000 servers to a single path name space. The state kept for a file object visible from user space must therefore at minimum specify the server, and storing a server-generated file ID along with server identification would be trivial. An user-space client connected to a single server can use compile-time file identifiers, but only if the application does not share the connection with third-party library code. To share a server connection between independent libraries, it would be necessary to implement a shared mechanism for allocating file identifiers between the libraries, which would be unnecessary if the identifiers were allocated by the server.

- The protocol does not provide any way to move a file from one directory to another.

Finally, the protocol does not implement some widely used features. They are arguably not really necessary in a file system protocol, and the functionality can be provided by other means (e.g. using the Plan 9 `bind` system call to alter name space of a process instead of using symbolic or hard links, or using a specialized server or a file system with efficient support for very small files to implement a database library instead of implementing server-side byte range locking), but considering the large amount of software that relies on these features, 9P2000 is unlikely to replace the currently used network protocols.

To support some UNIX features missing in 9P2000, an extension called 9P2000.u[14] was implemented. It still does not support moving a file between directories and byte range locking.

2.8 CIFS

The CIFS protocol[15], known also as SMB, is the primary remote file access protocol of the Microsoft Windows operating system. Because it is designed to primarily support the Windows API, the detailed specification of the supported operations differs significantly from the POSIX and traditional UNIX interfaces. During the more than two decades the protocol is used, the protocol has changed significantly, new flags were defined to identify support for newer features and new operations were defined to replace their obsolete variants. Neither the API differences between UNIX and Microsoft Windows nor the evolution of the protocol over time are discussed in this thesis; the following section focuses on features of the protocol that are relevant to FUSE or could be used to design a more efficient interface.

Files within a file system are identified using paths, which are relative to the root directory of the file system, or relative to an arbitrary previously opened directory. A client can therefore open a directory and then use paths relative to the directory instead of retransmitting the full path each time, and unlike the POSIX current working directory, a process can use more than one base directory.[14]

To support client-side data caching, a client can request an *oplock* on a file while opening it. If the server grants the oplock, the client may safely cache data. The server may at any time ask the client to synchronize its state with the server and to give up the oplock. Oplocks are granted purely at the discretion of the server[15], a client must be prepared to give up the oplock at any time, and to use files for which the server did not grant the requested oplock at all.

The weakest oplock is called “level II”: it guarantees no data is written to the file while the oplock is held, so the clients holding a level II oplock can cache data read from the server. A client holding an “exclusive” oplock is guaranteed no other clients have the file open, so it is safe to cache both read and written data and to manage file locks locally without consulting the server. A “batch” oplock has the additional guarantee that other clients won’t affect the file in any other way that could prevent its reopening (e.g. remove the file); it can be obtained if the client intends to close the file and reopen it later. Because an oplock is automatically released when the file it was granted for is released, the client which was granted a batch oplock must not send the “close file” and “reopen file” operations to the server.

To minimize the number of round-trips necessary to perform a simple operation, several operations can be transmitted as a single protocol request. For example, a request could contain operations to open a file, read data from it, and to close the file. The server would send a single reply, containing the results of all three operations, in order. If any of the operations fails, request processing is terminated and the server reply contains results of the successful operations and an error code for the unsuccessful operation. Note that the “read data” and “close file” operations operate with an unknown file handle; the server implicitly substitutes the result of the first operation in the request in place of the relevant handle parameter in all further operations. To make this possible, the request is transmitted as a special variant of the first operation (`*_ANDX`) and the other operations are formally transmitted as data within the first operation.

The protocol further reduces the overhead of some operations by implementing some high-level operations completely at the server side. The most

[14]A similar mechanism (`openat` and related operations) was implemented in Solaris and will be specified by the next version of the POSIX standard.

[15]“Oplock” means “opportunistic lock”—the server may revoke the lock whenever it is convenient.

important of these operations is copying and moving complete files (without copying data to the client and back). Several operations can use wildcards to specify a group of files, so the client does not have to transfer the complete contents of a directory over the network. Finally, the protocol supports creating a temporary file with an unique name. This does not result in a significant improvement if the client-specified file name is unique, but it could save many round-trips if the sequence of file names proposed by the client were not random enough.

2.9 NFS

Network File System is the most commonly used remote file access protocol used by UNIX operating systems. Version 4 of the protocol[16] was designed quite recently, based on the experience with both previous versions of NFS and other protocols, such as CIFS. Because it is not compatible with previous versions of NFS, version 4 provided an opportunity to design a cleaner implementation of the ideas developed for other protocols.

Unlike other file systems interfaces described in this thesis, the NFS protocol is designed to transparently recover from server crashes or other server failures that have not corrupted the storage device. The RFC contains a detailed description of the used mechanisms, their rationale and implementation advice. Although the mechanisms are non-trivial and interesting in themselves, they are not described in this thesis because its author cannot possibly describe them better than the protocol designers, and because they are not relevant to design of FUSE (the FUSE transport mechanism is reliable, the user-space server has only one client, the kernel-space client is never uncertain whether a server has crashed, and a crash of the client implies a crash of the server). The reader is encouraged to refer to the RFC to learn about these aspects of the NFS protocol.

Files on an NFS file system are primarily identified by their full path, using UTF-8 and specified string canonicalization rules (without canonicalization, it is possible that visually equivalent strings which differ e.g. in use of composing characters versus precomposed characters would be considered different file names). This is a significant change from earlier versions of NFS, which uses opaque file handles that are valid even after server reboot. As described in section 2.3.2, it is not always possible to provide such file handles, so NFS version 4 allows the use of “volatile” file handles, and it allows using two or more different file handles to refer to a single file. A volatile file handle may become invalid (even without a server crash); when it does, the client needs to obtain a new handle using a full path to the file. The server may provide a “fileid” attribute that can be used by clients to check whether two file handles refer to the same file, but supporting this attribute is not mandatory.

Because all paths resolved by an NFS client are relative to a file handle, the server provides two file handles: a file handle for the root directory of the file system, and a file handle for a root of the “public” subtree of the file system (for compatibility with WebNFS[17]).

To support a wide variety of client and server platforms, the set of file attributes supported by the protocol is not completely fixed. Only a small number of attributes is mandatory (file size is the only modifiable mandatory attribute), most defined attributes are optional. All operations dealing with file attributes use a bitmap to specify a subset of attributes which is transferred, and the result of operations that modify file attributes always includes a bitmap of attributes that are supported and were actually modified. The client can therefore determine which attribute values will be preserved by the server, and maybe use an alternate mechanism for storing unsupported attributes. In addition to the file attributes defined by the NFS protocol, the server may support other attributes (identified by string names) and/or user-defined extended attributes; both types are accessed as by associating a virtual directory of attributes to each file.

Like CIFS, NFS supports performing more than one operation per a roundtrip. Unlike CIFS, this mechanism is not limited to requests starting with a special operation nor to operation sequences in which all operations use a single file handle. The only available server request RPC is designed to transfer a sequence of operations, so a request with a single operation is simply a multi-operation request that contains only a single operation.

Instead of automatic rewriting of file handles in the requested operations, the protocol provides two file handle variables (a *current* and a *saved* file handle), and most operations implicitly use the current file handle, or both the current and saved file handle. For example, the `open` operation stores the file handle of the opened file as the current handle, a following `read` operation uses the current handle, and a final `close` operation closes the current handle. Another important example is the `lookup` operation, which resolves a file name within the directory specified by the current handle and replaces the current handle by a handle of the found file: the operation does not support multi-component path names, but a client can chain many lookup operations in a single request.[16]

[16]Note the difference between chaining of NFS `lookups` and the 9P2000 `walk`: `walk` modifies a *single* file ID to point to a different file, NFS `lookups` change the value of the “current handle” variable to a *different* file handle.

The `getfh` and `putfh` operations can be used to read or write the current handle if the client needs to work with more than one file at a time, and the `savefh` and `restorefh` operations allow access to the saved handle.

The CIFS oplocks are called *delegations* in NFS.[17] The delegations are not coupled to an open file state, the client can close the file and reopen it only before synchronizing state with the server. NFS introduces three improvements to write delegations (corresponding to exclusive oplocks in CIFS):

- An NFS client usually sends all file modifications to the server when an application on the client closes a file to make sure some common errors (running out of space on the file system or out of the assigned quota) can be detected and reported to the application. A client that holds a write delegation for a file might be able to avoid sending the modifications to the server until the delegation is revoked and to send only the final version of the file if detecting these errors were not necessary. Therefore, when the NFS server grants a write delegation, it may promise the client that any writes up to a specified maximum size will not fail for other reasons than a media error.

- A client that owns a write delegation does not have to notify the server about file attribute changes, so the server would have to revoke the delegation to know the current attributes of a file. Because reading of file attributes is very common (every `readdir` operation returns a client-specified subset of file attributes) and there is no way to re-grant a revoked delegation, NFS defines a callback which can be used by the server to ask the client for the current state of the file attributes without breaking the delegation.

- When an NFS server revokes a write delegation because another client is requesting to open the file and truncate it to zero bytes, the server can disclose this to the delegation owner. In that case the delegation owner can simply discard its modified data and avoid sending them to the server.

The NFS standard also requires clients that do not own a delegation to at minimum check the file change time when opening a file and to invalidate its caches if the file was modified since the cached data was obtained, and to send all file modifications to the server before closing the file. Although a client that does not hold a lock or a delegation on a file can never guarantee the data

[17]By granting the delegation, the server delegates performing operations to the client without requiring the client to cooperate with the server.

it uses is current, these requirements define at least minimum standard of data consistency, in line with the most popular implementations of previous versions of the protocol.

Chapter 3

Extensions to FUSE

This chapter discusses features of operating system interfaces described in the previous chapter that are currently not available in the FUSE kernel protocol, discusses possible extensions of the protocol to support these features, and proposes a set of extensions to the FUSE kernel protocol.

3.1 Extensions Necessary for ZlomekFS

As described in section 2.5.2, two functions not provided by FUSE are necessary to support ZlomekFS:

- The directory cache must immediately reflect creation and changes in the virtual conflict directories.

  This can be implemented either by using a very short validity timeout for the dentry cache (effectively bypassing the cache and giving up its benefits), or by allowing the user-space driver to notify the kernel that a specific dentry is not valid anymore.

- The page cache must support file synchronization without becoming inconsistent with the user-space driver: all modified data must be written to user-space before last close (`release`) of a file, and the page cache contents must not be considered valid after a synchronization that modifies the local copy of the file.

  The trivial solution, to completely disable the page cache, is not correct in this case, because the page cache (support for `mmap`, in particular) is necessary to support executing programs stored on ZlomekFS. The necessary behavior can be implemented primarily in kernel space, by directly copying the ZlomekFS kernel module behavior (write data before last close and unconditionally discard the cache after closing the file) in the FUSE kernel module, and using it if it is enabled by a mount option or by a flag in the data returned by the “open” operation.

Alternatively, the user-space driver can ask the kernel to discard the cache when returning from the “release” operation. Similarly asking the kernel to write data when returning from the “release” operation wouldn’t work because the user-space driver would not be notified when no modified data is left; the kernel would have to call a “prepare-release” operation, optionally write modified data, then call the “release” operation, and optionally invalidate the cache.

Finally, the page cache changes can be completely decoupled from the “release” operation by allowing the user-space driver to ask the kernel to write out all modified data for a specific file, or to invalidate the cache of a specific file, whenever the user-space driver considers it necessary.

3.2 Other Functionality Extensions

The descriptions of various file system interfaces above contain many features that are not currently supported by FUSE. The primary constraint for FUSE extensions is the requirement that it must be possible to use the new functionality using the available system calls, without modifying applications to explicitly communicate with the user-space driver. This currently excludes the `posix_fadvise` and `posix_fallocate` operations, and support for many features and properties of the CIFS protocol.

Features of other file system interfaces that are not currently supported by FUSE are described in the following list:

- The `dnotify` interface, already abandoned by the Linux applications it was designed for (GUI file managers), is probably not useful enough to implement. In addition, the client processes are notified about changes in a directory by a signal, and the user-mode driver, an untrusted process, should not be allowed to flood processes of other users with signals.

- The FreeBSD kqueue interface is, like `poll`, not very useful for regular files. On directories it can be used to receive notifications about changes in the directory, similarly to `dnotify` above.

- The FreeBSD `revoke` system call is only used on character devices, so it is not relevant to FUSE.

- The FUSE interface cannot enforce standardization of the character encoding used for paths. Like any other kernel driver, the FUSE kernel module does not know the encoding used by incoming data, so the paths transferred over the FUSE kernel interface cannot have a specified encoding. On the other hand, the user-space driver can, at least for file systems that are private to the user that mounted them, and usually even for file systems available system-wide, use the locale specification in its environment to determine the correct encoding. This knowledge can be used either when implementing clients for other file systems that specify file name encoding, or to implement a "filter" file system driver that forwards all request to another file system, converting file names on the fly. A "module" that performs the automatic conversion is actually already distributed with FUSE; unlike a complete file system driver, the module is a shared library that runs in the address space of the underlying file system driver.

- File system quotas could be useful if implemented by specific trusted file system drivers, even though they do not make any sense for untrusted drivers. File system quotas can currently be enforced by a FUSE file system driver (except for notifying the user on the TTY on which the restricted application is running), if the file system driver implements access to quota data files (in particular, user-space drivers cannot use the quota handling code present in the Linux kernel).

  It is currently not possible to extend the FUSE protocol to portably support transparent quota management (e.g. changing and recomputing quotas) using the platform's standard user-space tools. These tools, at least on Linux and FreeBSD, access quota data files (or even the block device underlying the file system) directly, and the quota data file cannot have multiple formats at once. Therefore, each user-space file system driver needs to be accompanied by its own set of file system-specific quota management tools.

- The current implementation of serving FUSE file systems over NFS can, as described in section 2.4.2, return spurious `ESTALE` errors. Although support of persistent file handles can be avoided in NFS version 4, it is necessary to support older clients.

  To support UNIX-like file systems with persistent inode numbers and a generation number stored in the inode, it is only necessary to add support for the `st_gen` field of `struct stat`, as described later in this section. No protocol extension is necessary to use more than 32 bits of the inode number to represent file handles if the NFS file handle is large enough. Unfortunately, the current design of Linux file handles limits the size of directory file handles (inode and generation number together) to ten bytes on NFS version 2, which is not enough to store a full 64-bit inode number and a 32-bit generation number.

In general (to support file systems with persistent file identifiers that cannot fit in 64 bits), it is necessary to delegate the interpretation of NFS file handles to the user-space driver. To interpret file handles, the protocol could be extended either by a "return inode number for this file handle" operation, or by providing an "getattr for this file handle" operation (which returns the inode number along with other attributes). Returning only an inode number might be simpler if the mapping between file handles and inodes is trivial, and it avoids transferring attributes if they are already cached by the kernel. On the other hand, if the inode is not in the cache, two user-space requests are necessary to use the inode. Because most of the overhead of an user-space request (e.g. context switching and memory allocation) does not depend on the size of transferred data, it would probably be more efficient to return all attributes along with the inode number: if the file is not in the cache, there is one fewer request to make, and if it is in the cache, the cache can be updated. To further reduce the number of user-space requests, the FUSE kernel module could implement a hash table of NFS file handles for files currently present in the inode cache.

The user-space driver could return an NFS file handle whenever it returns other file attributes. Alternatively, a separate "get file handle" protocol request could be used: this would save the overhead of creating and transferring file handles for FUSE mounts that are not served over NFS, at the cost of additional requests for exported file systems. Again, most of the overhead does not depend on the size of transferred data, so transferring the NFS file handle along with other attributes would be the better choice for file systems that are served over NFS. The vast majority of FUSE mounts are *not* served over NFS, though, and "getattr" is a very commonly performed request (unlike both operations proposed in the previous paragraph that use a file handle to identify a file, which are both used only on NFS-exported file systems), and it is possible that using a separate operation would result in smaller average overhead, when measured over all FUSE users. When comparing worst-case behavior, it is necessary to consider an NFS "readdir" operation, which reads a directory and returns file names and a specified subset of file attributes, usually including a file handle. It would be necessary to perform a "get file handle" request for each returned directory entry, and user-space drivers that do not use natural inode numbers would have to look up the file using its inode number, which could be very inefficient when compared to simply returning the file handle along with other data returned by "readdir".[1]

If the NFS file handle were returned along with other file attributes, the file system driver could choose not to provide a valid handle if the file system would never be mounted over NFS; the user could provide such a guarantee by using a mount option. This can be more efficient if the overhead of mapping between "natural" file identifiers and NFS file handles is significant, although that is unlikely. No additional extension of the FUSE kernel protocol would be necessary to support such a mount option. In addition, if the file attribute structure used an variable-length byte array to represent the NFS file handle, using such a mount option could even avoid the cost of transferring unused NFS file handles to kernel space. A file system driver could indicate it does not support NFS at all by returning an invalid NFS file handle for the root directory of the file system.

Although the above paragraphs suggest how the FUSE kernel protocol could be extended for more general NFS support, the value of the feature is rather limited. A malicious FUSE file system accessible over NFS can be used to deny NFS service to other users by accessing the file system over NFS and not responding to any file system requests, causing the NFS server's thread to block and to stop replying to other NFS clients. This makes serving arbitrary FUSE file systems over NFS undesirable on a multi-user machine.[2]

Fundamentally, in most cases it is possible, and probably more efficient, to mount a FUSE file system on a client directly instead of exporting it over NFS.

- Whiteout directory entries can be easily supported without extending the FUSE protocol on host operating systems which support whiteout entries, in particular the `S_IFWHT` file type constant. The `mknod` operation can be used to create the entries, and `unlink` to remove them. A portable file system driver can check whether the `S_IFWHT` constant is available during build configuration, and enable or disable support for

[1]It would be possible to replace the sequence of "get file handle" requests by a single batch request, but the lookups by inode number are not easily avoidable.

[2]It is still possible to serve user's home directories over NFS safely because the default NFS configuration does not make file systems mounted in the subtree of an exported file system visible to NFS clients.

whiteout directory entries depending on the result.

- Similar approach can be used by file system drivers to support additional file attributes if they are present in `struct stat`. In this case, however, a new protocol version would be necessary to actually transfer the attributes between the kernel and the user-space driver.

  Compared to POSIX and Linux `struct stat`, the FUSE file attribute structure is missing a value of the `st_blksize` attribute. The file system driver of a block-device backed file system can specify a block size while mounting the file system; other file systems use `PAGE_SIZE`, the default value.

  Three more attributes are supported on FreeBSD: `st_flags`, `st_gen` and `st_birthtime`. The `st_flags` and `st_gen` attributes are useful only for system software. The `st_birthtime` stores information which could be useful to the user, and a similar attribute is supported by several widely used file systems.

- An NFS client implemented using FUSE should be able to fulfill the cache management requirements on clients that do not own a delegation. Currently, the only way to fulfill the requirements is to completely disable data caching for the file. To use the page cache, the file system driver must be able to optionally invalidate the cache when opening a file, and to write out modified data before closing the file.

  Because this is quite similar to the ZlomekFS page cache requirements, the possible implementations are similar as well. The driver could use a flag in the data returned by the "open" operation to request immediate cache invalidation without disabling data caching, which is already implemented in FUSE, or to request modification write out before closing the file. (A mount option cannot be used because the server grants or declines delegations for each file separately, not for the whole file system.)

  Alternatively, the driver could send explicit requests to the kernel to invalidate or write out the caches. The semantics of these requests is equivalent to the semantics of the requests proposed for ZlomekFS.

- Beyond the semantics required by NFS, it might be useful to invalidate the cache whenever a change of the last modification time is detected (e.g. when the user calls `fstat` on the file and updated attributes are fetched from a remote server). This cannot be implemented using a flag returned by the "open" operation. The FUSE kernel module could in principle invalidate the caches automatically when the last modification time changes, but it is better to delegate the decision to the user-space driver, which can use protocol-specific information to evaluate the trade-off between caching obsolete data and rereading it. Again, the "invalidate cache" request proposed for ZlomekFS can be used by the driver if it decides the obsolete data should not be held in the cache.

When implementing file system driver in user-space, the FUSE kernel interface is not the only constraint. Some facilities available to kernel-space drivers are simply not available in user space, usually for security reasons:

- Direct access to hardware (`/proc/driver/nvram` on Linux)
- Access to the address space of the calling process (`ioctl`, `hugetlbfs` on Linux)
- Access to the address space of other processes (`procfs`)
- Access to arbitrary kernel data structures: although the data may be available in `/dev/kmem`, the user-space driver cannot use kernel's locks to safely access it.
- Deep cooperation with kernel's caches, either because the overhead of communication about cache changes would be prohibitive, or because the design of data structure locks does not allow blocking for a response from user space.

3.3 Performance Enhancements

FUSE is not suitable for implementing file systems tuned for maximum possible performance. Even if the FUSE kernel protocol had the best possible design and the kernel module, user-space library and user-space driver were all implemented in the most efficient way, the interface between user and kernel space imposes some overhead that can be avoided by implementing a kernel-space file system driver. By imposing unavoidable overhead, FUSE limits the extent to which it makes sense to optimize the interface and implementation: savings of two machine instructions while processing a request are insignificant compared to the overhead of transferring the request and response between kernel and user space. This section describes some possible improvements of the FUSE kernel protocol that can bring significant performance enhancements without imposing large implementation costs.

At the lowest level, it might be possible to reduce the amount of data copied between kernel and user space by changing the layout of data structures used by the protocol (removing various padding, or using the smallest possible data type to represent the request type). This is a good example of an optimization that is probably not worth the cost, saving a few cycles at the cost of introducing a new version of the protocol and extending the library to handle both versions.

Another way to reduce the amount of copied data would be to allow the driver to directly map pages from the kernel's page cache, avoiding one copy of the data. The decrease in the amount of copied data could be significant, but mapping pages to the address space of the driver, together with the necessary TLB flushes, impose additional overhead. As described in appendix A.1, direct access to the page table from a file system driver would actually cause a net slowdown.

At a higher level, the overhead could be reduced by avoiding some requests altogether. The operations are specified by system calls from user-mode applications, so to reduce the number of requests it is necessary to reduce the average number of requests per operation.

In general it is not possible to delay executing operations, gather several operations from applications, and to send several operations as a single "batch request" to the user-space driver. First, any of the operations could fail and there would be no way to report this to the application. Second, an application can sometimes observe the external effects of its actions because a single file system often can be observed using several independent paths (e.g. over NFS, FTP, HTTP, as an IMAP mailbox), and the application would observe an inconsistent state.

The only case in which partially inconsistent state might be acceptable (because it is expected by applications) is reading obsolete data and writes that do not propagate to a remote file server until the file is closed. FUSE already allows file systems to use the page cache for data caching (delayed writes are not supported for security reasons described in section 2.4.3). File system drivers that want to avoid providing obsolete data if possible could benefit from data caching if they obtain an oplock from a remote server, or a similar guarantee that the file will not be concurrently modified. To benefit from oplocks, the kernel module must support cached and uncached data access, and the file system driver must be able to switch between the two and to invalidate the page cache. The "invalidate cache" request proposed for ZlomekFS is usable for handling oplock revocation as well, but an "invalidate cache and switch to uncached data access" request would be better. Enabling cached data access can be done using a flag returned with other results of the "open" operation because both NFS and CIFS grant oplocks only when opening a file.

The FUSE kernel module caches file data, but it does not cache directory contents returned by `readdir` at all. The cache validity times returned for each entry can be used to determine the cache validity time associated with the whole batch of `readdir` results. No extension to the FUSE kernel protocol is necessary to support `readdir` result caching.

To avoid file system requests for other operations, it would be necessary to make the sequence of file system operations asynchronous with respect to the application that requests the operations. An interface that could be used to specify such an asynchronous sequence of operations (similar to the way a "channel program" would be started on an S/360 I/O channel processor) was proposed for Linux[18], but it does not seem to be worked on currently. Finally, even if the application can specify such a sequence of operations to the kernel, the Linux file system interface would probably not represent this directly to the file system drivers, preventing the use of the operation sequence for avoiding FUSE requests.

The other way to reduce the average number of requests per operation is to literally implement each operation using fewer requests. Both FUSE and ZlomekFS return file attributes along with other results of operations that create new files and of the "lookup" operation; otherwise a "getattr" operation would follow immediately when setting up an inode for the created dentry.

A path lookup is split by the generic file system layer into a sequence of file name lookups, and the FUSE kernel module simply forwards each file name lookup in turn if the results are not present in the dentry cache. The current Linux and FreeBSD file system interfaces do not allow any batching of the requests, the `lookup` operation is no different from other operations in this regard. If changing the kernel's file system interface is possible, the sequence of lookups could be replaced by a single user-space request if the following properties can be maintained:

- The atomicity of directory operations must be preserved. For example, if the file system's implementation of `rename (`A`, `B`)` performs a separate operation to remove B before renaming A, all other local applications must be prevented from observing the state where no file named B exists. The generic kernel code guarantees this by locking each directory before looking up a file name in it. If the path lookup were performed on a remote server, the local locks would be ineffective and the result could indicate a missing B.

- Mount points must be interpreted, even if the requested path exists on a file system that was hidden by the mount point.

- Symbolic links must be correctly resolved, even if they point outside of the file system in which they are stored.
- The untrusted file system driver must not be able to cause anomalies in file lookup, such as interpreting "`..`" differently. Similarly, the untrusted file system should not even be passed paths that it otherwise could not observe, e.g. looking up `../../secret-name` in a FUSE mount point should not make `secret-name` available to the user-space driver.[3] Note that it is acceptable to make `symlink/secret-name` available to the file system driver if the contents of `symlink` are currently "`../..`" because the untrusted file system driver can at any time observe `secret-name` by changing the contents of `symlink` to be "`.`".

On Linux, mount points are always present in the dentry cache, so no file system support for lookup operations is necessary if file name lookup batching is performed only for path names that cannot be resolved using the dentry cache, and the path does not contain any "loops" of the form X`/..`, where X is a directory not present in the dentry cache. To implement the atomicity requirement, it is necessary to make the results of the full path name batch only advisory, and to discard them if the relevant directories are concurrently modified by other processes. Lookup batching could be implemented on Linux by adding an optional "path tail" parameter to the `lookup` operation. The file system driver could ignore the path, or it could use it to lookup the remaining path components as well, and to receive whatever file attributes are conveniently available from the underlying storage mechanism or remote server. After preparing a dentry to return to as a result of the `lookup` operation, it could use the data about additional path components to pre-create inodes and dentries in the kernel's caches, if not prevented by concurrent modification by other processes. The lookup of the remaining path components by the generic kernel code would then be satisfied from the dentry cache without requesting repeated lookups from the file system driver.

This design preserves the atomicity guarantee, mount point handling and "`..`" semantics because implementation of the lookup in the generic kernel code does not change, it still performs the same locking and it still interprets each file name separately. To keep sensitive file names secret from untrusted file system drivers, no path tail hint would be passed if the tail contains a

[3]Some file names, for example "Personal bankruptcy filing", are naturally sensitive. It is also possible to use file names as a crude replacement for ACLs by storing data in X`/secret-name`, making X publicly accessible but not publicly readable, and revealing `secret-name` only to the intended users of the data.

"`..`" component; this restriction need not apply to kernel space file systems, it can be implemented in the FUSE kernel module.

To support symbolic links, the underlying file system must be able to return as much data as possible, terminating at an invalid lookup: if the tail is X`/symlink/`Y, the underlying file system should be able to return data about X and `symlink` without failing because `symlink` is not a directory. Otherwise, looking up a long path ending with `symlink/`Y could result in looking up $O(L^2)$ file names, where L is the number of components in the path name. The 9P2000 `walk` operation and the possibility of requesting several `lookup` operations in a single request in NFS are suitable for lookup batching; CIFS supports specification of files using path names, but cannot return partial results if the path does not exist.

At the highest level, most of the overhead could be eliminated by using high-level operations, such as "return the complete contents of a file at this path", "replace the contents of a file at this path by this data", or "copy this file to another path on the same file system". These requests cannot be expressed using POSIX operations, so they are out of scope of FUSE. A file system interface that supports such requests could, though, support file access over FUSE as well. This approach is planned for the GVFS library (a replacement for GNOME's `libgnomevfs`): it will provide a native interface for applications that use the GVFS API, and it will make the same files simultaneously accessible as a mountable FUSE file system.

3.4 Summary of Proposed Extensions

The above description of various possible extensions to the FUSE protocols also helps choosing a suitable extension to support ZlomekFS: the possibility to invalidate the page cache or to write out modified data under complete control of the user-space driver (instead of only implementing a specific behavior which can be enabled or disabled from user space) can be used for other purposes as well.

The following extensions are proposed to support ZlomekFS, to fulfill the NFS requirements on clients that are not granted a delegation, and to allow cache invalidation whenever a file modification is detected:

- Add a way to invalidate a dentry and file attribute cache of a specific file from user space.
- Add a way to invalidate page cache of a specific file from user space.
- Add a way to write out all modified data for a specific file from user space.

In addition to the above page cache management operations, the following extensions are proposed to allow data caching only if an oplock is granted:

- Add a way to invalidate page cache of a specific file from user space, switching future data accesses to uncached.
- Add a flag to the data returned from an "open" request that switches future data accesses for the file to cached.

The following extensions are proposed for better NFS support, if an incompatible change to the protocol is acceptable:

- Add NFS file handle to the attributes returned by the user-space driver in the reply to "getattr" and various other requests. The NFS file handle should be represented as a variable-length array.
- Add a variant of the "getattr" request that identifies a file by its NFS file handle.

If the set of attributes returned by the user-space driver is changed incompatibly, members necessary to support the `st_blksize` and `st_birthtime` could be added.

Finally, to avoid repeated "lookup" requests, adding of an optional "path tail" parameter to the "lookup" operation (both in the kernel file system interface and in the FUSE kernel protocol) is proposed.

Chapter 4

Implementation

The cache management extensions proposed in the previous chapter (the first two groups described in section 3.4) were implemented in the FUSE user-space library and the Linux kernel module.

4.1 Protocol Extensions

A new `FOPEN_NO_CACHING` flag value was defined for the `open_flags` member of `struct fuse_open_out` to support switching between cached and uncached file access. The "no caching flag" is associated with the file, not the single open file state, and each `open` or `create` result updates the value of the flag. The default value of the flag is 0 for compatibility with previous versions of the protocol.

The other protocol extensions (invalidating dentry and file attribute cache entries, invalidating page cache and optionally disabling caching, and writing out all modifications) are all "reverse" requests, originating in user-space. The FUSE interface did not support any such requests before, and all data written to the character device was supposed to be request results. To support reverse requests, the request identifier 0 was reserved, and request result header with request identifier 0 are used to identify reverse requests. The reverse request itself follows, starting with a 32-bit command code and followed by other operands; in our case, all reverse requests need only a single operand, inode number of the file operand.

When requests are originated in the kernel, their results are asynchronously supplied by the user-space driver by writing a reply structure to the character device. Similar handling of reverse request results, letting the user-space driver read a reply structure from character device, would cause implementation difficulties because the user-space driver might have to read several requests from the device (and defer their processing) only to reach the reply structure; even if the reply structure were placed at the head of the read queue, multi-threaded user-space drivers would have to add an exclusion mechanism to make sure the reply structure is handled by the thread that has sent the reverse request.

Because it is not necessary to return any data from the kernel for these requests, except for an error code, a simpler solution was used instead: the error code is returned directly as the error code of the *`write`* system call used to perform the reverse request.

The implementation of the kernel protocol extensions in the FUSE user-space library is straightforward.

4.2 ZlomekFS

The kernel listener implementation in the ZlomekFS daemon was rewritten from scratch to use the FUSE protocol. The ZlomekFS command-line parsing code was rewritten to use the FUSE command-line parsing API to make usage of `zfsd` consistent with other FUSE file system drivers.

A `mlock` configuration option was added for the `config` configuration file. It can be used to prevent the default `mlockall()` system call and to allow running the ZlomekFS daemon by unprivileged users.

Unlike the original ZlomekFS implementation, the new version cannot be used as a server-only daemon without mounting the ZlomekFS file system locally. The original implementation allowed only one ZlomekFS mount, so it was simple to start `zfsd` and mount the file system later. FUSE supports running a single file system driver several times at once, but mounting a file system requires access to the file handle used to open the `/dev/fuse` device, so deferral of the mount is not possible without implementing a protocol to transfer the file handle from the file system driver over a socket. It is difficult to justify this implementation effort when the ZlomekFS file system can simply be mounted in a special-purpose directory that is never accessed.

4.3 Linux

Only small modifications of the FUSE kernel module were necessary to support the protocol extensions on Linux.

When invalidating dentries for a single file, some dentries may be referenced and thus it is not possible to immediately discard them. Instead, an indication that the metadata was invalidated is added to the inode. When the generic kernel code attempts to use the dentry in a lookup, it uses the `revalidate` operation provided by the FUSE kernel module to check the validity of the dentry; if the dentry references an inode which has invalidated metadata, the dentry is considered invalid.

Because the Linux kernel module never caches file modifications, the “no caching” flag affects only file reads. Reading data using the *`read`* system calls handles the “no caching flag”; mapping a file to memory and reading the memory ignores the flag because there is no way to catch every memory read (by removing page table entries after exactly one memory access). Besides, the overhead of rereading the whole page when any single byte of the page is modified would be prohibitive.

4.4 FreeBSD

The protocol extensions necessary to support ZlomekFS were also implemented on `fuse4bsd`[19], a FreeBSD implementation of the FUSE kernel protocol.

The implementation revealed a significant difference between low-level aspects of the Linux and FreeBSD file system interfaces. On Linux, a process that uses an inode only increments the reference count, and the inode mutex is used only to serialize specific operations; most of the file system driver's operations are called without locking the inode mutex. Although FreeBSD vnodes have a reference count, the vnode's read-write lock is locked almost whenever a reference to the vnode is held; even looking up a vnode by its inode number in the global hash table returns a locked vnode. This introduces extra complexity to the generic file system code because it must often handle vnodes locked both exclusively and non-exclusively. Because vnodes are locked while processing calling driver's operations, the FUSE kernel module cannot avoid keeping them locked while the user-space driver is processing requests. If the user-space driver performs a reverse request for the same file, the kernel cannot process it because locking the vnode again would cause a deadlock.

A reverse request to invalidate the file attribute cache needs to be processed soon enough not to degrade user's experience, but it is not necessary to process it immediately: the user-space driver is not affected by attribute caching and the reverse request cannot fail in a way that needs to be handled by the user-space driver. Therefore, a queue of invalidation requests pending in the kernel was added to each FUSE mount: invalidation requests that cannot be processed immediately because the destination vnode is locked[1] are

[1]This is possible because the desired locking operation can be specified to the above-

put on the queue and success is reported to the user-space driver. Whenever the user-space driver answers a request and no other requests are pending (so no vnodes can be locked by processes that block until the user-space driver does something), the queue is processed and invalidations for unlocked vnodes are performed; invalidation requests for locked vnodes are kept on the queue. The length of the queue is artificially limited to `MAX_INVAL_QUEUE_LEN` (1024) entries to make sure the user-space driver cannot cause unbounded kernel memory allocations.

Execution of the "write modified data" request cannot be delayed like this, because the data needs to be written out before closing the file; in general, the "write modified data" request inevitably results in a deadlock. The kernel module was modified to support at least writing modified data while processing the "release" request. As described in [19], the FUSE open file states are independent of the open file states used by the kernel interface, and sometimes the kernel module releases several FUSE open file states at once—while the vnode is kept locked. The code was modified to gather the open file states in a list, temporarily unlock the vnode, and release the file handles. An extensive rework of locking (to use vnode's "interlock", a spinlock, instead of relying on the read-write lock) was necessary to safely maintain a list of the open file states to be released.

4.5 Other Work

The ZlomekFS-specific kernel module was updated to work on newer kernel releases, eventually up to Linux 2.6.22.1, to serve as a base against which the FUSE port can be compared (some benchmark results are presented in appendix A.2).

Testing of ZlomekFS on the FreeBSD port of FUSE has revealed a problem in the design of the ZlomekFS protocol: the `readdir` operation returns an inode number, not a file handle, for each directory entry. The ZlomekFS daemon returns the inode number of the file underlying the directory entry in the local volume cache (on a local file system), if present; otherwise it returns the inode number returned by a remote ZlomekFS server. These inode numbers are obviously inconsistent with inode numbers that are assigned by the `zfsd` kernel listener and available using the `*stat` system calls. The tree search functions provided by the FreeBSD C library, unlike the GNU C library implementation, abort the search upon encountering such inconsisten-

mentioned hash lookup by inode number, and the desired locking operation can be non-blocking; of course, support of non-blocking locking adds some complexity to the hash lookup function.

cies because they assume the directory tree is being unpredictably modified and continuing the search could result in an infinite search or a search that continues outside the originally specified subtree.

There seems to be no unobtrusive fix to this problem because the file handles that should be returned by the `readdir` operation are not available along with file names: whenever a `lookup` operation is performed, `zfsd` can attempt to synchronize the destination file and optionally create a conflict directory with a new, synthetic, file handle. Changing the network protocol to replace the inode number with a file handle would not help because the decision whether to synchronize a file happens locally. If the FUSE `readdir` operation must return inode numbers consistent with the results of a subsequent `lookup`, a simple `readdir` to gather only names of files present in a directory must trigger synchronization of all files in the directory. Nevertheless, the implicit `lookup` of all directory entries returned by `readdir` seems to be the only possible solution, even though it increases the cost of the `readdir` operation.

4.6 Future Directions

This thesis opens opportunities for future work in several areas:

- Several FUSE kernel protocol extensions where proposed in chapter 3, but not implemented. Most important is probably the proposed addition of a "path tail" argument to `lookup`, which could be used by several Linux file systems drivers. The generalization of NFS support and support for additional attributes in the FUSE kernel protocol are comparatively less useful.

- The FUSE kernel protocol allows caching of `readdir` results, but neither Linux nor FreeBSD kernel modules currently implement it.

- The ZlomekFS implementation could use the more flexible possibilities of invalidating the page cache to invalidate it only when a file synchronization actually occurs, not after each file close.

  The current implementation of the `readdir` FUSE request in ZlomekFS performs an implicit `lookup` for each returned directory entry. The result of a `lookup` operation includes the full set of file attributes, but the only attribute returned to the kernel is the file handle. Better integration of the file handle lookup to the `readdir` implementation might be more efficient.

- ZlomekFS could be ported to other operating systems. Operating systems that support NFS must already support the necessary data cache management operations, and support of file attribute cache invalidation is likely as well, so a kernel-space implementation (if such a distinction makes sense for the operating system in question) would probably be possible.

  A more practical way to port ZlomekFS is to port FUSE instead, and to implement the necessary FUSE kernel protocol extensions. The split between kernel and user space in FUSE makes some assumptions about the interface that do not necessarily hold for the native file system interface of the target operating system. The `fuse4bsd` author has encountered non-trivial implementation difficulties[19], and larger difficulties can be expected on non-UNIX operating systems. The design of locking and resource "ownership" in the native file system interface can also have large impact on the porting FUSE; the reverse requests necessary for ZlomekFS place even more requirements on the locking design. It would not be surprising to discover that it is impossible to port FUSE with the ZlomekFS extensions to some operating systems without extensive modification of the native file system interface.

Chapter 5

Conclusion

The main goals of this thesis were fulfilled successfully: The FUSE kernel protocol was extended to support ZlomekFS, ZlomekFS was ported to FUSE, and the FUSE kernel protocol extensions were implemented on Linux. The FUSE protocol extensions were implemented on FreeBSD in a manner that allows running ZlomekFS on FreeBSD, although the implementation is not fully general.

In addition, the FUSE protocol was extended further to let user-space file system drivers benefit from oplocks and similar mechanisms that make local data caching possible, and several other extensions of the FUSE protocol were proposed.

Appendix A

Performance Measurements

The measurements were performed on an otherwise idle computer with an Intel Pentium M processor and 512 MB of system memory. The automatic CPU frequency scaling was disabled and the frequency was fixed at 600 MHz.

A.1 Mapping Page Cache to User Space

As described in section 3.3, copying of data between user space and the kernel-space page cache can be avoided by mapping page cache pages into the driver's address space, at the cost of additional overhead necessary to modify the page tables. To evaluate this trade-off, a simple proof-of concept program was written.

The program writes 160 MB of data (bytes with value `0x01`) to a file, one page at a time, using a single page-sized and page-aligned buffer. Using conditional compilation, it can write the data using one of the following methods:

- `write`: The buffer is filled with data, and a `pwrite` system call writes its contents at the desired position. Although the data does not change between different pages, it is always overwritten again to be consistent with the other methods, which must write data to the mapped pages.
- `mmap`+`munmap`: The contents of the file at the desired position are mapped over the buffer, the buffer is filled with data, and the memory mapping is torn down.
- `mmap`-only: The contents of the file at the desired position are mapped over the buffer (overriding a previous mapping, if any), and the buffer is filled with data. Before closing the file, the final memory mapping is torn down.

The `write` method corresponds to the behavior of a FUSE file system driver responding to a "read" request in the current implementation: the driver prepares the data, uses a `writev` system call, and the kernel copies it to the page cache page. The `mmap`+`munmap` method represents a hypothetical FUSE modification, in which the kernel maps the destination page cache page to the driver's address space, the driver prepares the data directly in the page cache page, and the page is unmapped.

The `mmap`+`munmap` method tears down a memory mapping only to create another mapping in the following iteration. If the hypothetical FUSE interface preallocates address space for the purpose of mapping page cache tables, it is unnecessary to unmap memory pages from the address space as long as the page cache pages are not reused for another purpose. The `mmap`-only method was prepared to evaluate this improvement.

The test program was run ten times for each method. Before each test run, the destination file (placed on an `ext3` file system) was already allocated and the compiled test program was run at least once. Results are summarized in table A.1.

Table A.1: `write` vs. `mmap`

Method	Wall Time	User Time	System Time	User + Sys
`write`	8.4 ± 2.3	0.107 ± 0.006	0.562 ± 0.029	0.669 ± 0.026
`mmap`+`munmap`	7.5 ± 1.2	0.220 ± 0.010	0.521 ± 0.024	0.741 ± 0.023
`mmap`-only	7.5 ± 1.4	0.208 ± 0.020	0.493 ± 0.034	0.701 ± 0.036

All times are in seconds.

As expected, replacing `write` by `mmap` decreases system time because the data does not have to be copied by the kernel, and increases user time due to TLB misses and less efficient cache use (accessing various pages in the page table instead of a single physical page containing the buffer). Overall, when measured as a sum of user and system time, the `mmap-only` method causes a slight slowdown over `write`, and `mmap`+`munmap` is slower significantly. The results are reversed when comparing the wall time, but the variance of the wall time measurements is so large that the results are not statistically significant.

A.2 Comparison of ZlomekFS Implementations

This section provides a performance comparison of the original ZlomekFS implementation and the port to FUSE. The Bonnie++ benchmark utility was used to compare the speed of the implementations. This benchmark was chosen because it tests both raw data access speed and directory operations, while being more transparent than an "application" benchmark such as a web server benchmark or software compilation. Bonnie++ reports throughput (measured in kilobytes or operations per second) and CPU usage; the CPU usage figures were omitted because they measure only CPU usage attributed to the benchmark process, not CPU usage of the user-space driver.

ZlomekFS was tested in a "single-node" configuration, on a local volume backed by an `ext3` file system. The following three variants were tested:

- module: The original implementation with a ZlomekFS-specific kernel module.
- FUSE: The port of ZlomekFS to FUSE.
- direct: The port of ZlomekFS to FUSE, but the "no caching flag" is set for all files. This change approximates the behavior of a simpler port of ZlomekFS that completely disables page cache use instead of extending the FUSE kernel protocol. Note that setting the "no caching flag" does not simulate the effects of using a very small file attribute and lookup cache validity timeout instead of explicit flushing of cache entries.

To provide at least a rough indication of the total overhead of ZlomekFS, a single run of each benchmark was performed on an `ext3` file system,

Data access throughput tests were performed using 1 GB of data, using three test runs. The results in table A.2 show the FUSE port is usually faster than the original ZlomekFS implementation, up to 23% in the "rewrite" test. A possible cause was revealed by comparison of the source code: FUSE copies data directly between user space and the page cache, the ZlomekFS kernel module uses an intermediate data buffer.

Another perhaps surprising result is that the uncached variant of FUSE is faster than the cached variant, although only slightly. The page cache cannot provide any benefit in this benchmark because the file used for the benchmark is more than twice as large as the memory available for caching, and it is read and written sequentially using page-aligned system calls. The uncached variant copies data directly between the benchmark process and the user-space driver; the cached variant must perform one more copy to keep the page cache updated.

Directory operation tests were performed using 10240 files in a directory, using ten test runs. The `ext3` file system performs these operations so fast that it could not be accurately measured; operation speed for `ext3` was

Table A.2: ZlomekFS Data Access Throughput

Test	module	FUSE	direct	`ext3`
`putc`	7.60 ± 0.18	7.78 ± 0.07	7.86 ± 0.06	10.76
`write`	13.99 ± 2.67	15.84 ± 0.39	15.60 ± 0.35	16.71
Rewrite	6.15 ± 0.26	7.57 ± 0.15	7.62 ± 0.13	7.95
`getc`	8.07 ± 0.70	10.54 ± 0.06	10.58 ± 0.04	10.47
`read`	16.43 ± 3.84	16.15 ± 0.67	15.27 ± 1.75	18.56
Seek	75.83 ± 1.57	84.73 ± 1.57	86.03 ± 1.59	101.10

Throughput is measured in megabytes per second. Seek performance is measured as number of seeks per second.

Table A.3: ZlomekFS Directory Operations per Second

Test	module	FUSE	direct	`ext3`*
Seq. create	828 ± 35	851 ± 12	825 ± 23	10955
Seq. read	9427 ± 635	4419 ± 118	4329 ± 142	102624
Seq. delete	956 ± 40	830 ± 24	828 ± 22	19754
Rand. create	841 ± 31	824 ± 22	832 ± 23	10592
Rand. read	10864 ± 587	8410 ± 274	8468 ± 139	158647
Rand. delete	957 ± 51	931 ± 26	923 ± 30	20325

* The values for `ext3` were measured in a directory with ten times as many files.

therefore measured in a directory with ten times as many files. The results are summarized in table A.3. The extreme slowdown of the sequential read test in FUSE implementations can be attributed to the implicit “lookup” operations performed on `readdir`, as described in section 4.5.

Bibliography

[1] Zlomek J., Shared File System for a Cluster, 2004.

[2] FUSE: File System in Userspace, releases 2.6.0–2.7.0.
`http://fuse.sourceforge.net/`

[3] Kroah-Hartman, G., The Linux Kernel Driver Interface.
`http://www.kroah.com/log/linux/stable_api_nonsense.html`

[4] IEEE Std 1003.1-2001: Standard for Information Technology—Portable Operating System Interface (POSIX®).
`http://www.opengroup.org/austin`

[5] Linux 2.6 GIT repository, various intermediate versions between 2.6.18 and 2.6.23.
`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git`

[6] BSD Sockets Interface Programmer's Guide, Edition 6.
`http://www.docs.hp.com/en/B2355-90136/index.html`

[7] Kleiman S. R., Vnodes: An Architecture for Multiple File System Types in Sun UNIX. USENIX Association: Summer Conference Proceedings, Atlanta, 1986, pp. 237–247.

[8] Rosenthal D. S. H., Evolving the Vnode Interface. Proceedings of the Summer USENIX Conference, June 1990, pp. 107–117.

[9] Free BSD CVS, the HEAD branch after FreeBSD release 6.2.
`http://www.freebsd.org/developers/cvs.html`

[10] FreeBSD Architecture Handbook.
`http://www.freebsd.org/doc/en/books/arch-handbook/`

[11] Lemon J., Kqueue: A generic and scalable event notification facility. Proceedings of the FREENIX Track (USENIX-01), California, June 2001, pp. 141–154.

[12] Plan 9 Programmer's Manual, Volume 1, Fourth Edition, section 5. `http://www.cs.bell-labs.com/sys/man/5/INDEX.html`

[13] Pike R., Ritchie D. M., The Styx Architecture for Distributed Systems. Bell Labs Technical Journal, Vol. 4, No. 2, April–June 1999, pp. 146–152.

[14] Van Hensbergen E., Plan 9 Remote Resource Protocol Unix Extension. `http://v9fs.sourceforge.net/rfc/9p2000.u.html`

[15] Common Internet File System (CIFS) Technical Reference, Revision 1.0. SNIA CIFS Technical Work Group, 2002.

[16] Shepler S., Sallaghan B., Robinson D., Thurlow R., Beame C., Eisler M., Noveck D., Network File System (NFS) version 4 Protocol. RFC 3530, 2003.

[17] Callaghan B., WebNFS Server Specification, RFC 2055, 1996.

[18] Molnar I., "Syslets", generic asynchronous system call support, `http://redhat.com/~mingo/syslet-patches/async-v1-ANNOUNCE.txt`

[19] "Fuse for FreeBSD" documentation. `http://fuse4bsd.creo.hu/doc/html_single_out/doc.html`