Comparison of kernel and user space file systems

— Bachelor Thesis —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Kira Isabel Duwe
E-Mail-Adresse: 0duwe@informatik.uni-hamburg.de
Matrikelnummer: 6225091
Studiengang: Informatik

Erstgutachter: Professor Dr. Thomas Ludwig
Zweitgutachter: Professor Dr. Norbert Ritter

Betreuer: Michael Kuhn

Hamburg, den 28. August 2014
Abstract

A file system is part of the operating system and defines an interface between OS and the computer’s storage devices. It is used to control how the computer names, stores and basically organises the files and directories. Due to many different requirements, such as efficient usage of the storage, a grand variety of approaches arose. The most important ones are running in the kernel as this has been the only way for a long time. In 1994, developers came up with an idea which would allow mounting a file system in the user space. The FUSE (Filesystem in Userspace) project was started in 2004 and implemented in the Linux kernel by 2005. This provides the opportunity for a user to write an own file system without editing the kernel code and therefore avoid licence problems. Additionally, FUSE offers a stable library interface. It is originally implemented as a loadable kernel module. Due to its design, all operations have to pass through the kernel multiple times. The additional data transfer and the context switches are causing some overhead which will be analysed in this thesis. So, there will be a basic overview about on how exactly a file system operation takes place and which mount options for a FUSE-based system result in a better performance. Therefore, an overview is given on how the related operating system internals work as well as a detailed presentation of the kernel file systems mechanisms such as the system call. Thereby a comparison of kernel file systems, such as tmpfs and ZFS, and user space file systems, such as memfs and ZFS-FUSE is enabled.

This thesis shows the kernel version 3.16 offers great improvements for every file system analysed. The meta data operations even of a file system like tmpfs raised by a maximum of 25%. Increasing the writing performance of memfs from about 220 MB/s to 2 600 MB/s, the write-back cache has an enormous impact with a factor of 12.

All in all, the performance of the FUSE-based file systems improved dramatically, transforming user space file systems in an alternative for native kernel file systems although they still can not keep up in every aspect.
Acknowledgements

My warmest appreciation goes
to Professor Ludwig and Professor Ritter for enabling this thesis,
to Michael Kuhn for his kind supervision, helpful advice and especially the proofreading,
to Fabian, Max and Pablo for forming a 'Bachelor thesis writing group',
to Daniel P. Bovet and Marco Cesati for writing their amazing book 'Understanding the Linux Kernel',
to Florian and Jacob for proofreading,
to my parents for always supporting me and never doubting my wish to study informatics
though I had no previous knowledge,
to my dear little sister for going grocery shopping when I was ill or tired,
to my not so little brother for providing entertainment!
I thank Enno for enduring nights of FUSE compiling and kernel debugging!
# Contents

1 Introduction/Motivation ......................................................... 6  
1.1 Motivation ................................................................. 6  
1.2 Related work .............................................................. 7  
1.3 Own contribution .......................................................... 8  

2 Fundamentals of operating systems ........................................ 9  
2.1 Kernel ................................................................. 9  
2.2 Protection ring ......................................................... 10  
2.3 Address translation ...................................................... 11  
2.4 Four-level paging ....................................................... 12  
2.5 Context switching ....................................................... 13  
2.6 System call and mode transition ...................................... 14  
2.7 Kernel module .......................................................... 17  
2.8 Cache ................................................................. 18  
2.9 Processes ............................................................... 21  

3 Kernel file systems ........................................................... 23  
3.1 Virtual File System (VFS) ............................................. 24  
3.1.1 VFS internals ..................................................... 25  
3.2 Special file systems ..................................................... 30  
3.2.1 /proc-system .................................................... 30  
3.2.2 Temporary file system (tmpfs) .................................. 31  

4 User space file systems ....................................................... 32  
4.1 Filesystem in Userspace (FUSE) ....................................... 32  
4.1.1 System call with FUSE ......................................... 33  
4.1.2 Mount options .................................................. 34  
4.2 Implementation of a Filesystem in Userspace (FUSE)-based file system . 35  

5 Comparison ................................................................. 38  
5.1 Tools ................................................................. 38  
5.1.1 Monitoring ....................................................... 38  
5.1.2 Benchmarking .................................................. 39  
5.2 Test environment ...................................................... 39  
5.3 Performance of volatile memory file systems ....................... 40  
5.3.1 Meta data overhead .......................................... 40  
5.3.2 Iozone ........................................................ 43
5.4 Performance of disk-based file systems ................. 48
  5.4.1 Reading ........................................ 48
  5.4.2 Writing ........................................ 49
5.5 Time of a context switch .............................. 50

6 Conclusion .............................................. 52

Bibliography ............................................. 53

Acronyms .................................................. 56

List of Figures ........................................... 57

List of Tables ............................................ 58

Appendices .................................................. 59

A Usage Instructions ....................................... 60
  A.1 FUSE .............................................. 60
    A.1.1 FUSE version 29 ................................ 60
    A.1.2 FUSE version 30 ................................ 60
  A.2 File systems ....................................... 60
    A.2.1 tmpfs .......................................... 60
    A.2.2 memfs .......................................... 60
    A.2.3 ZFS-FUSE ..................................... 61
    A.2.4 ZFS ........................................... 61
  A.3 Benchmarking ....................................... 61
    A.3.1 IOzone .......................................... 61
  A.4 Tracing ............................................. 62
    A.4.1 Git and DTrace ................................ 62
    A.4.2 DTraceToolkit ................................ 62
    A.4.3 mpstat/ sysstat ................................ 62
    A.4.4 perf ........................................... 63
1. Introduction/Motivation

This chapter provides the necessary information to understand why this thesis was written. First, a general overview is given of the requirements towards a file system and the reasons why user space file systems arose. Afterwards, the related work is presented shortly. Finally, the aims of this thesis are deduced.

1.1. Motivation

File systems are an important component of the operating system as their behaviour influences the overall performance of the OS remarkably. In today’s operating systems, the file I/O is almost always the bottle neck. Therefore, optimizing file system operations and an efficient usage of the storage results also in better system performance.

The chosen file managing model also has a direct impact on the available functionalities of the system.

Network based systems provide the opportunity to manage a distributed system whereas special file systems, such as the temporary file system (tmpfs), enable fast access as their data are only stored in the RAM.

Not only performance but also the system’s integrity as well as data consistency plays an important role for an OS.

The kernel and also the file system itself are responsible for checking access rights or enabling encryption.

As there are different demands on what a system and therefore the file system should be capable of, there can not be one solution to all those problems. Therefore, a grand variety of file systems arouse and the desire for developing an own and exactly fitting one.

File systems and the related operations are all part of the operating system’s kernel.

The kernel source code is neither trivial and nor easy to understand since it is complex due to several optimizations and safety instructions. This makes it hard for most user to just write a new kernel file system.

Additionally, license problems, with Linux developed under the GNU General Public License, complicate the development, especially for companies. Hence, no file system can be included in the kernel unless the source code is opened to public.

Furthermore, changing the kernel requires recompiling and also rebooting of the system, which is not just inconvenient but also impossible for non privileged users in an administered network.

The debate in the 1980 was impelled by such needs and resulted in the microkernel concept which reduces the kernel size to a minimum and relocates the remaining func-
tionality into the user space. This led to Linux adapting this concept within a monolithic kernel by introducing modules which enable similar capabilities [BC05]. Finally, this gave rise for the FUSE project as it is today. It provides a frame work for writing user space file systems. Despite his dislike Linus Torvalds agreed, after long discussion with the FUSE developer Miklos Szeredi, to include the FUSE module in the kernel 2.6.14-rc1 [Sta06]. FUSE consists of a kernel module, a user space library offering a stable API and a mounting tool, to allow non privileged users to mount their user space file system. Due to all those offered abilities, FUSE is widely used for all kinds of file system types. Prominent systems with a great performance are Ceph and GlusterFS. Ceph is a scalable high performance distributed file system which separates the data and meta data management [WBM+06]. It provides data replication as well as failure detection and is capable of supporting over 250 000 meta data operations per second. In this distributed file system the client’s code is entirely located in user space and it is mounted via FUSE. GlusterFS is a parallel network file system which enables accessing NAS (Network Attached Storage) via Infiniband [Com11]. Seen from the user’s perspective, GlusterFS resembles NFS although it has a totally different internal design. Again, the client side uses FUSE for the file system.

1.2. Related work

Rajgarhia and Gehani analyse the performance of user space file systems in [RG10]. Therefore, they explain the inner structure of such a file system and why a performance overhead is induced. Also, the different mount options and their influence on the performance are presented, finally followed by the results of the Bonnie 2.0.6 benchmark as well as the PostMark benchmark. In [IMOT12], Ishiguro et al. describe the possibilities to optimize the local file access in a distributed storage based on FUSE. They introduce a mechanism to reduce context switches and show the related result with the GFarm file system. Unfortunately, they need to modify the FUSE library as well as the FUSE module to improve the system’s behaviour. As one of the goals of FUSE is to make the implementation of file systems easier, changing the related kernel module discards a major part of the provided facilitations. The analyse is also very specific to this distributed file system and does not enable an overall evaluation of FUSE.
1.3. Own contribution

Due to the few analyses of today’s FUSE performance and the overhead introduced by the user space library, there is the need for further inspection. Additionally, the kernel version 3.15 finally includes the support for the write-back cache policy, making new measurements to evaluate the new functionality necessary. Therefore, this thesis aims to answer the following questions:

- **How fast can FUSE possibly be?**
  In order to rate the different performances, also with respect to the own capabilities, it is important to know the best possible maximum of a concept. For that reason detailed knowledge of the internals of each system is required. This gave rise to the next questions:

- **What exactly happens in a native kernel file system?**

- **How does FUSE master those tasks?**
  To provide the necessary information to understand both concepts, an overview will be given about operating system internals. Afterwards both the user space and the kernel space approach are looked at thoroughly.
  They are analysed especially to present the difference in their behaviour regarding an issued system call.
  It is of special interest, how a switch is performed from user mode to kernel mode and how exactly a context switch takes place, as those are the most powerful concepts to ruin a system’s performance.

- **To what extend do kernel file systems performance differ from those of a FUSE-based file system?** For the kernel’s side, the temporary file system (tmpfs) will be regarded as the current optimum. Furthermore, some benchmarking of the meta data overhead is necessary to offer the same information for FUSE.
  To evaluate the best performances, the greatest part will cover volatile based systems to detach the analysis from the capabilities of the disk’s capabilities. However, these aspects are regarded with ZFS on Linux as well as ZFS-FUSE. The FUSE-based system will also be tested with special interest in the mount options. Therefore the new write-back cache will be analysed, as well as the most promising parameters such as big writes and large reads.
2. Fundamentals of operating systems

This chapter offers an overview of operating system internals. Therefore, an introduction about the kernel in general, the virtual memory management, the address translation and the system call mechanism will be given. As the system calls have a great influence of a file systems performance they are discussed in detail. The different approaches to perform a mode transition such as the \texttt{sysenter} assembly language instruction and the \texttt{int $0x80} assembly language instruction are presented. Lastly, the page cache policies are explained as FUSE makes use of them in the newest version and the kernel’s process management are outlined.

2.1. Kernel

To define the \textit{operating system} is a complex task. An approach to determine what is meant consists of describing the basic functions. [Tan09] distinguishes between two essential views; top-down and bottom-up. On the one hand there is the application and the need for a clear and tidy abstraction of the hardware. On the other hand the operating system has to manage the computer’s resources.

The \textit{kernel} is the most basic and central part of an operating system. As Wolfgang Maurer explains in [Mau08] the kernel basically is a layer between the hardware and the software. Its purpose is to mediate between applications and hardware. This does not mean the kernel is solely an interface. As Tanenbaum mentioned, the kernel is also responsible for the administration of rights, allocate resources due to current requirements.

Now to distinguish between the kernel and the OS is difficult. The opinions about what exactly is included in which concept vary widely. There is the approach to set them both equal which happens in several of books about operating systems as they only describe the kernel’s tasks [Tan09].

In this thesis the term kernel is used for describing the core functionality as a subset of an operating system, which includes the sole access to the hardware, whereas the rest of the OS also provides the environment to utilise the provided options. Despite the discussion about the distinction the tasks of the kernel are obvious. They consist of memory management, process management and scheduling, devices drivers, modules and the file system.

The \textit{virtual address space} is the term for describing the fact, that the maximum address space is not affected by the physical RAM which is actually available[Mau08]. It is
divided into two separate parts which are referred to as user space and kernel space. The last is reserved for the kernel.

Each user process possesses its own virtual address space. Due to this model each process thinks it is the only one residing in this address space as it can not see any other process. Nevertheless, the kernel space, which a user process can not access, is always the same, regardless what process is running. The virtual address space is parted to protect the processes from interfering with each other.

### 2.2. Protection ring

The operating system provides different privilege levels to run the CPU as shown in figure 2.1. Standard Unix kernels use only Kernel Mode and User Mode as described by Bovet and Cesati [BC05, p. 19].

The operating system is executed in Kernel Mode (supervisor mode) and has access to the complete instruction set. An application is normally running in User Mode hence not allowed to access the kernel data or the kernel programs. When it requires some kernel services, a switch from User Mode to Kernel Mode is necessary. After performing the request a switch vice versa takes place. This is accomplished by the construct of a system call.

The Current Privilege Level (CPL) is set in a 2-bit field in the code segmentation register code segement (CS). This register is a segmentation register used to hold one of the Segment Selectors which is needed by the Segmentation Unit performing the logical address translation [BC05, p.37] as shown in 2.2.

The value 3 depicts the lowest privilege level (User Mode), while the value 0 shows the highest one (Kernel Mode). The wanted CPL has to be supported by the CPU. There are CPUs providing two execution states whereas for example the 80x86 microprocessors have four levels [BC05, p. 19].
2.3. Address translation

As context switches and system calls have a strong impact on a file system’s performance knowledge about their inner structure gives valuable insight. To understand the procedure of a context switch and a system call a short summary about address translation, paging and the corresponding registers is given.

The logical address translation is part of the virtual memory management. The Memory Management Unit maps logical addresses into physical ones as shown in 2.2. First, the logical address which is used to address an operand or an instruction in the machine language instructions is translated by hardware circuit called Segmentation Unit into a linear address [BC05, p. 36]. A linear address is also referred to as virtual address. Thereafter, a second hardware circuit named Paging Unit transforms the linear address into a physical address which are utilised to address memory cells in the chips. The interpretation of the linear address depends on the PG-Bit in the cr0 control register. Only when paging is enabled the transformation by the Paging Unit takes place. Otherwise the linear address is interpreted as the physical address in the main memory [Mär94, p.143].
A logical address is divided into two parts: the Segment Selector and an offset specifying the relative address in the segment. To simplify the search for segment selectors the processor has segmentation registers to hold them. They are called cs, ss, ds, es, fs, and gs. Of these the first three are used for specific storing [BC05, p.37]:

- **cs** - code segment register: pointing to a segment holding program instructions
- **ss** - stack segment register: pointing to a segment holding current program stack
- **ds** - data segment register: pointing to a segment holding global and statistic data

To describe the segment characteristics an 8-byte Segment Descriptor is used, which is either stored in the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT). Normally only one GDT is built whereas each process might have its own LDT. To restrict the access to the segment the Descriptor Priviledge Level (DPL) is used. It describes the minimal CPL which is necessary to access this segment. With a DPL set to 0 the segment is accessible only with a CPL equal to 0 while a DPL set to 3 allows access with every CPL value.

## 2.4. Four-level paging

To understand the concept of the four-level paging model a short recap on Paging. Pages are grouped linear addresses in fixed-length intervals to increase efficiency [BC05, p. 46]. The translation of linear address to the physical addresses was performed with two translation tables, the Page Directory and the Page Table. Those two steps were used to reduce the necessary amount of RAM since Page Tables were only required for the memory actually occupied. Further basic information about the paging mechanism are given in [Tan09, p. 242 ff]

With Linux version 2.6.11 a four-level paging model has been introduced. There are now four different types of page tables namely:
• Page Global Directory
• Page Upper Directory
• Page Middle Directory
• Page Table

Now, as seen in 2.3, the linear address consists of five parts. The Page Global Directory includes several Page Upper Directories which hold several Page Middle Directories which point to several Page Tables. Each process uses its own Page Global Directory as well as an own set of Page Tables. A control register is process register which influences the behaviour of the CPU. The cr3 is such a control register. It stores the address of the Page Global Directory and thereby enables the translation of the linear address to the physical address.

```
GLOBAL DIR  
UPPER DIR   
MIDDLE DIR  
TABLE       
OFFSET
```

![Figure 2.3.: The Linux paging model](image)
Based on [BC05, p. 58]

To speed up the linear address translation a special cache named Transition Lookaside Buffer (TLB) is used. The first look-up fails like with every cache. Hence the corresponding physical address has to be computed and is stored in the TLB so further use of this address can be dealt with more quickly. Each CPU has its own local TLB.

### 2.5. Context switching

*This subsection as well as the next one on system calls is based on Bovet’s *Understanding the Linux Kernel*[BC05, ch.2,3,7,10].*

To manage the system’s resources the kernel must be able to control the running process. Therefore it needs to be able to suspend and resume the execution of a process. This is named *process switch*, *task switch*, or *context switch*[Mau08]. Since the scheduler uses *Time Division Multiplex* (TDMA), also called time-sharing, to enable economic usage of the CPU, context switches are necessary.

Every process uses its own address space but all share the same CPU registers. Therefore, the kernel must ensure to save the content of the registers, so that after suspension the process has the same environment it had before. This set of data is called
the hardware context.

A context switch is made up of the following steps:

- Installing the new address space by switching the Page Global Directory. Linux saves the current cr3 control register in the process descriptor and loads the cr3 related to the process executed next.

- Switching the hardware context.

- Also switching the Kernel Mode stack to supply all the necessary information to execute the new process.

Sometimes, people confuse context switches with the change of the privilege level in system call. They are not the same. A context switch always occurs in the Kernel Mode[BC05]. When an application invokes a system call, it is running in User Mode. To execute the required service the kernel has to perform a mode transition. Although, they often occur short after another, they provide different functionalities and therefore should not be mixed up.

### 2.6. System call and mode transition

The system call mechanism is an extra layer between the application and the hardware providing some advantages.

No knowledge of hardware characteristics is needed, as the system call is an abstraction of the actual system, so programming gets significantly easier. The system security is increased since the validity of the call is checked [BC05]. Also, the system becomes more portable as a program can be compiled on every kernel with this set of interfaces.

Every system call returns an integer value. In the kernel a value greater or equal to 0 represent a successful termination, while a negative value indicates an error. The errno variable, which contains a specific error code, is set by the wrapper after returning to User Mode.

The execution order is described in 2.4.
The application in User Mode invokes a library wrapper function. Since the kernel implements many system calls the system call number is a necessary parameter to pass internally. The wrapper uses the eax register for this. Users do not need to concern themselves with the system call number, they just use the system call provided by the library.

Afterwards, the function calls a trap. Depending on your Kernel version and hardware there are two possibilities. Either executing the int $0x80 assembly language instruction or executing the sysenter assembly language instruction.

What ever is used, the result is a jump to the system call handler which is an assembly language function. Thereby, the processor is switched to Kernel Mode.

The system call is taken care of by the system call handler which invokes the system call service routine.

At that point the real action is performed wherefore this whole work is done.

After exiting the service routine, the hardware context is restored in the Kernel Mode stack and thereby the CPU switches back to User Mode.

Hereafter the differences in the two ways of issuing a system call are explained based on the 'Intel Software Developers Manual' [Gui10, ch. 5] and [BC05, ch.4, 10].

The handling of errors and exceptions while executing is left out to simplify this section as it is already complex.

In older version of the Linux kernel the int $ 0x80 instruction and the iret instruction were the only possibility to perform a switch from User Mode to Kernel mode and vice versa. The sysenter instruction and sysexit instruction are supported since the Linux 2.6. kernel.
During kernel initialization the Interrupt Descriptor Table is set up. The vector 0x80 is related to the system call function. Now the control unit looks up the base address of the GDT to get to the Segment Descriptor to get to the base address of the segment that includes the interrupt handler. To make sure the issuing sources has the necessary authority the CPL (section 2.2) and the DPL (section 2.3) are compared. An exception is invoked if the CPL is lower than the DPL. If the CPL is higher than the DPL the control unit has to start using the stack related to this new privilege level. This is performed by reading the Task State Segment (TSS) used to save the contents of the processor registers and loading the ss and esp with the proper values. The new stack saves the logical address of the old stack by storing the previous values of ss and esp. Afterwards, the eflags, cs, and eip are saved in the stack. Next, the logical address of the first instruction of the interrupt handler is loaded which is defined by cs and eip.

At this point the process switches from User Mode to Kernel Mode and starts executing the instructions from the system_call() function. First the system call number and all necessary CPU registers are saved. Then the macro for interrupt handling loads the Segment Selector of the kernel data segment to ds and es. Thereafter the data structure of the current process is stored in ebx. It follows a check on whether tracing is enabled. If this is the case, do_syscall_trace() is invoked two times, right before and right after the execution of the system call routine. A further check is performed to ensure the passed system call number is valid. An invalid number leads to termination of the system call handler storing the -ENOSYS value in eax. Finally, the service routine related to the valid system call number in eax is invoked.

As shown above the int $0x80 instruction is very complex and inherently slow due to those several consistency and security checks. Under the Pentium IV, this method suffered a strong performance penalty [LaF].

Therefore, the sysenter instruction referred to as the 'Fast System Call' by the Intel Documentation was introduced [Gui10].

In contrast to the int $0x80 instruction which requires different descriptors to get the address of the interrupt service routine, the sysenter instruction includes a definite address the CPU knows and is able to jump directly to that section which saves time [She13].

The sysenter instruction uses 'Model-Specific Registers' short MSR which denotes that register is present only in some models of 80x86 processors (e.g. SYSENTER_CS_MSR). When issuing a system call via the sysenter instruction, there is a wrapper routine in the standard library used to handle the call. First, the wrapper loads the system call number in eax and then issues the __kernel_vsyscall() function. This function is the solution to the compatibility problem that a library can only use the sysenter instruction if both the Linux kernel and CPU support
Either the int $0x80 instruction is called or the sysenter instruction depending on what is supported. To use the sysenter instruction, the function builds the vsyscall page. Afterwards the ebp, edx, and ecx are saved on the User Mode stack. Next, the user stack pointer is stored in ebp.

The CPU switches to Kernel Mode due to the Segment Selector loaded in ss and starts executing the sysenter_entry() assembly language function held in the SYSENTER_-EIP_MSR which is the kernel entry point.

sysenter_entry() sets up the stack pointer, enables local interrupts, saves eflags and the user related registers such as the current user stack pointer and the Segment Selector of the user data segment as well as the address of the instruction to be executed when returning from the system call.

After restoring the ebp, the system call handler is invoked by executing the system_-call() function as described above.

There have been different comparison of those two instructions regarding the necessary clock cycles. They differ concerning the absolute numbers as there are different testing environments but they all show that the 'Fast System Call' is as twice as fast. Schneider found that the sysenter takes about 500-800 cycles whereas the int $0x80 instruction needs 1350-1750 [SB05]. LaForest presents 181 cycles for the sysenter instruction and 330 cycles for the int $0x80 instruction [LaF]. For further information and studies see [VYC05] and [NSJ+09].

2.7. Kernel module

As the FUSE project is implemented as a loadable kernel module a short summary of the abilities of a module is given.

A kernel module is the Linux solution to the missing flexibility that comes with a monolithic kernel without changing the whole architecture to a microkernel. The advantages of both concepts are combined and therefore lead to a stable and fast system.

Basically, a module is a file whose code can be linked dynamically to the kernel [BC05, p. 11]. It is an efficient way to add file systems, device drivers and other components at runtime without the need to recompile the kernel or even to reboot the whole system [Mau08, p. 474]. This allows to link experimental code just by loading the related module and therefore test new features more easily. Modules are also convenient collections of drivers which can be pre-compiled and added into the kernel after automatic hardware detection selects the corresponding module. Thereby, all kind of users are enabled to install drivers as there is no need for building a new kernel which results in a greater acceptance for Linux systems.

Another major advantage is the ability to resolve licence issues. As the Linux kernel is available under the Open Source licence GNU General Public License every driver whose
code is not revealed to the public cannot be included in the kernel. This problem is avoided when using modules for the compiled drivers which can be linked to the kernel and therefore do not have to be included. Nevertheless, this is only a technical solution for the ideological debate regarding open code.

2.8. Cache

As the performance of the different caches in an operating system plays an important role in the performance of a file system, the cache in general and writing policies in specific are discussed in the following section which is mainly based on the book 'Computer Architecture' by Hennessy and Patterson [HP12].

Accesses to disk are very slow which is why there are different techniques to reduce the number of accesses and speed up the performance. The most common one is the cache. Typical access times for the different level of the memory hierarchy are shown in table 2.1.

<table>
<thead>
<tr>
<th>Access times</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ns</td>
<td>Register</td>
</tr>
<tr>
<td>2 ns</td>
<td>Cache</td>
</tr>
<tr>
<td>10 ns</td>
<td>RAM</td>
</tr>
<tr>
<td>10 ms</td>
<td>hard-disk</td>
</tr>
</tbody>
</table>

Table 2.1.: Access times for the different memory levels (rough estimate) based on [Tan09, p. 55]

Until the Linux kernel version 2.4. there were two different caches namely a page cache and a buffer cache which had nothing to do with each other. Starting with version 2.4. those two were combined for reasons of performance and resulted in one cache called page cache. There are still buffers in the kernel but now they describe the mapping of disk-block to a page [Lov05, p. 348].

Caches are optimized for reading fast, as every instruction access is a read, but most instructions do not write data. Unfortunately, writes also take longer than reads, as the tag whether the address is a hit has to be checked first and not at the same time [HP12, p.B-10].

There are two different policies regarding when a page is written to disk:

- **write-through**: writing information both to the cache and the memory block
- **write-back**: writing information only to the cache. The modified block is not written to disk until it is replaced in the cache.

The *dirty bit* is used to reduce the frequency of writing blocks back. A dirty block is one that is modified while in the cache whereas a clean block is not modified. When a
cache miss occurs, the clean block does not need to be written back to disk, as the same information is stored in the lower levels. Thereof the advantages and also disadvantages of each concept arise.

Write-through is easier to implement as the cache is always clean and a miss does not result in writing to the lower memory. Also the data coherency is simplified as the most current copy of the data is already stored to disk.

In contrast write-back enables writing at the speed of the cache and multiple writes in one block need only one write to disk. Since the writes do not got to disk, unless a replacement is necessary, the write-back policy uses less memory bandwidth. Therefore it also saves power which can be a vital criterion for embedded systems. But this better performance comes at the price of inconsistent memory which is a great problem when the system crashes.

Therefore dirty pages are flushed to disk periodically, if either a replacement occurs or a system call such as `sync` is invoked [BC05, p. 622].

There are also two options how to deal with write miss:

- **Write allocate**: On a write miss the block is loaded from disk and the appropriate bytes are changed.

- **No-write allocate** (write-around): With this option a write miss does not affect the cache because the block is modified directly in the lower memory. Hence the block is not in the cache until the block is read by the program. It is mostly used for application which write their data and then never touch it again. Thus, the cache is not filled with unused blocks replacing data which might be important. Therefore, the miss rate is reduced.

Usually, the write-back cache uses write allocate to manage repeated writing to the same block efficiently. Write-through caches often use no-write allocate as writes must go down to the disk in any case, so there is nothing gained by write-allocate.

The sequence of actions for those combinations are shown in figure 2.5 and figure 2.6. Having a write-back cache with write allocation results in handling write and read request the same way, whereas the write-through cache manages only data blocks that are read. All blocks modified are directly written to disk and not cached until they are read.
Figure 2.5.: Write-back cache with write allocation
Based on [wik14] and [HP12, B 11]
2.9. Processes

As processes are fundamental to today’s multiprocessor machines a short overview of the structure which represents a process is given based on Bovet [BC05, ch. 3].

To manage processes the kernel needs to keep track of their attributes such as the priority and the assigned address space. Thus, the process descriptor whose fields hold all the information concerning one process was introduced. It consists of a large number of fields containing the attributes and also pointers to resources owned by this process. In table 2.2 the most important ones regarding the topic of this thesis are presented since the overall number is about 80 fields. The structure for the process descriptor is called task_struct, as processes are often referred to as threads or tasks.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm_struct</td>
<td>Pointers to memory area descriptors</td>
</tr>
<tr>
<td>fs_struct</td>
<td>Data representing the interaction of the process and a file system</td>
</tr>
<tr>
<td>files_struct</td>
<td>Files currently opened by the process</td>
</tr>
</tbody>
</table>

Table 2.2.: Fields of the task_struct
- Based on [BC05, p. 82 ff]

The fs_struct structure contains the current working directory, the root directory, the system’s root file system as well as the file system mounted in the current working directory. Also a bit mask to set the file permissions when opening a file is included.
A file descriptor is an integer assigned by the kernel to identify an opened file. The file descriptor is only valid within a process and used as a parameter for all file-related operations.

The system’s root file system is the file system that holds the most essential system programs and initialisation scripts. It is mounted by the kernel during the booting phase. As every file system is a tree of directories each one has its own root directory. In contrast to Unix, in Linux every process can have its own tree of mounted file systems which form the name space of the process although usually most processes share the name space beginning at the system’s root file system. The mount point is the directory the file system is mounted on. In Linux it is possible to mount one file system several times. It is even possible to stack mounts on a single mount point whereby the new mount hides the previous mount and the related file system.

The files_struct structure holds an array named File Descriptor (FD) whose elements point to the file objects used by the virtual file system (see section 3.1.1). The file descriptor is used as the array index of FD. The first element of the array is reserved for the standard input of the process, the second for standard output and the third for standard error. Further data held in files_struct are the maximum numbers of file objects and file descriptors and a set of file descriptors to be closed on exec().
3. Kernel file systems

This chapter is about kernel file systems. First, an overview over the different types is given and later on the virtual file system with its inner structure is explained in detail. Also, the execution order of a system call including the virtual file system is shown. Lastly, special file systems as the temporary file system are presented.

A file system is part of the operating system and defines an interface between OS and the computer’s storage devices. It is used to control how the computer names, stores and basically organises the files and directories. Due to many different requirements such as efficient usage of the storage, a grand variety of approaches arose.

There are three general classes of file systems [Mau08]:

- **Disk-based file systems**, storing files on non-volatile memory.

- **Special file systems**, or pseudo file systems, are sometimes referred to as virtual file systems but those must not be confused with the Virtual File System (VFS). In the following they are called special file systems to distinguish those fundamentally different concepts which will be explained in detail later on. The temporary file systems live only in the RAM whereas the /proc file system does not require any memory at all.

- **Network file systems** are a combination of disk-based file systems as well as virtual file systems. They permit access to data via a network. Due to the abstraction a user space application sees no difference between local and network file systems as the file related operations are provided. On the other hand the kernel is not concerned whether the storage location is on a different system.

In the following especially the concept of a virtual file system is explained in detail with focus on the temporary file system (**tmpfs**).

The main task of file system is to store data and the related meta data. A file can be each and any information the user wants to save.

The meta data consists of things like the file name, the ownership, permissions to access the file, timestamps to keep track when the file was created or modified and the file size.

Following the approach of UNIX where mostly everything is considered a file [Mau08, p.524], the kernel subsystems as character and block devices, pipes, sockets for network protocols or even a terminal communicate via the file interface.

There are a lot of system calls related to file systems. In 3.1 the most important
ones for a basic file system are shown. The interface semantics are defined by the IEEE Standard for Information Technology *Portable Operating System Interface* (POSIX) [pos99].

<table>
<thead>
<tr>
<th>System call name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mount, umount</td>
<td>Mount and unmount the file system</td>
</tr>
<tr>
<td>mkdir, rmdir</td>
<td>Make and remove directories</td>
</tr>
<tr>
<td>readdir, link, rename</td>
<td>Manipulate directory entries</td>
</tr>
<tr>
<td>creat, open, close</td>
<td>Create, open and close files</td>
</tr>
<tr>
<td>read, readv, write, writev, sendfile</td>
<td>file input/output operations</td>
</tr>
<tr>
<td>chdir</td>
<td>change directory</td>
</tr>
<tr>
<td>stat, fstat, lstat, access</td>
<td>file status</td>
</tr>
</tbody>
</table>

Table 3.1.: System calls related to file systems - Based on [Pat03] and [BC05, p.460]

### 3.1. Virtual File System (VFS)

The following section is based on chapter 12 of Bovet’s *Understanding the Linux kernel* [BC05] if not referenced otherwise. The VFS is a kernel software layer abstracting from the actual file system to support various file systems. It is located between the standard library in the user space and the file system implementation in the kernel space as shown in 3.1.

Therefore, it provides applications with a uniform API defined by POSIX to access different file systems and serves as an interface to cover local file systems as well as accesses to remote directories.

Due to its ability of providing an interface to several kinds of file systems it is a very successful concept handling the system calls related to a file system [BC05, p.456]. This is a difficult task as the file system differ not only in details but in their whole design. As Maurer states [Mau08, p.519] the kernel supports more than 40 file systems from FAT (File Allocation Table) to NFS (Network File System) or pseudo file systems as /proc-fs. VFS basically consists of the idea of a common file model to represent all supported file systems. The common file model particularly construed for UNIX based file systems. Nearly no overhead is produced for ext2,3 and other native file system to run whereas non native systems such as FAT need to translate their implementation. The FAT file systems does not regard directories as files and therefore the implementation must be able to create the corresponding file for each directory at runtime.

VFS basically consists of the idea of a common file model to represent all supported file systems. The common file model particularly construed for UNIX based file systems. Nearly no overhead is produced for ext2,3 and other native file system to run whereas non native systems such as FAT need to translate their implementation. The FAT file systems does not regard directories as files and therefore the implementation must be able to create the corresponding file for each directory at runtime.

The common file model is an object-oriented attempt. Those objects are implemented as plain data structures with field holding the pointers to the necessary functions.[BC05, p.459ff]

To enable this wide functionality four object types are defined [SGG13, p.746]:

24
• **superblock object**: It is representing the whole file system. It stores information related to the mounted file system.

• **inode object**: short for index-node [Tan09, p.907] . Every file has just one inode. The inode number is used to uniquely identify the file within the file system. As directories and devices are regarded as file they also have related inodes.

• **file object**: This describes the interaction of the process and opened file. This information only exists in kernel memory when the file is open.

• **dentry object**: The directory entry object represents a directory entry to make the path lookup and similar operations more easy. Dentry objects are buffered in the dentry cache.

The functionality of a file system is split up between those four objects, so the system calls listed in 3.1 are also divided. To show the responsibility of each object the following table 3.2 was included. These methods are available to all possible file system types but only a part applies to a specific file system. The fields of unimplemented methods are set to NULL.

<table>
<thead>
<tr>
<th>Object</th>
<th>Related operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superblock</td>
<td>alloc_inode, write_inode, sync_fs, remount_fs, show_options</td>
</tr>
<tr>
<td>Inode</td>
<td>create, lookup, link, mkdir, rename, truncate, permissions</td>
</tr>
<tr>
<td>File</td>
<td>llseek, read, write, readdir, mmap, open, fsync, lock, sendfile</td>
</tr>
<tr>
<td>Dentry</td>
<td>d_revalidate, d_hash, d_compare, d_delete, d_release</td>
</tr>
</tbody>
</table>

Table 3.2.: Objects of the VFS and a selection of their operations - Based on [Tan09, p.908] and [BC05, ch.12]

### 3.1.1. VFS internals

The internals of the virtual file system are explained in detail to give insight in this important software layer where every file system related system call is passed through.

#### Superblock object

When the file system is mounted the superblock structure is created by `alloc_super`. The superblock object is essential as its destruction would result in an unreadable file system. It holds the information about each mounted file system in `s_fs_info` and also a pointer to the `file_system_type` structure to determine the type. As those pointer duplicate data the VFS needs to care about consistency problems. Otherwise the VFS superblock might not be synchronized with the the related superblock on disk. This is handled by a number of flags regarding the status of the memory. When a inode is modified it is not only stored in the `s_inodes` struct but also in the `s_dirty` struct. To minimize the problem of a corrupted file system when the system suddenly shuts down,
this list of dirty inodes is periodically copied to disk.
The operations pointed to by $s\_op$ are high-level methods like allocating inodes and
mounting disks as listed in table 3.2.
The following table gives an overview over the structure of a superblock. There are only
the most relevant fields included as the whole struct consists of 37 fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_dev$</td>
<td>Device identifier for the device the file system is held in</td>
</tr>
<tr>
<td>$s_blocksize$</td>
<td>Blocksize in bytes</td>
</tr>
<tr>
<td>$s_type$</td>
<td>Pointer to the $file_system_type$ structure of the mounted file system</td>
</tr>
<tr>
<td>$s_op$</td>
<td>Pointer to the $super_operations$ structure</td>
</tr>
<tr>
<td>$s_inodes$</td>
<td>List of all inodes</td>
</tr>
<tr>
<td>$s_dirty$</td>
<td>List of modified inodes</td>
</tr>
<tr>
<td>$s_files$</td>
<td>List of file objects</td>
</tr>
<tr>
<td>$s_fs_info$</td>
<td>Pointer to superblock information of the actual file system such as Ext2</td>
</tr>
</tbody>
</table>

Table 3.3.: Fields of the $super\_block$ structure - Based on [BC05, p.462 ff]

**Inode object and file object**

To access files the inode and the file objects are needed. The inode object contains the
pointer to the disk block with the actual file data and meta data such as access right and
date of last change whereas the file object offers the access to the opened file [SGG13, p.747].

The inode number $i\_ino$ is used to uniquely identify the inode throughout the whole
file system. Each file has just one inode holding meta data such as $i\_mode$, $i\_size$ or
$i\_state$. The $address\_object$ is the most important structure in the page cache which
is discussed in section 2.8. The only relevant point is, the actual data is pointed to by
the $i\_data$ field [BC05, p.601].

Although, the inode contains the information to handle the file, the file name is not
included in the inode but in the dentry object.

The hash table $inode\_hashtable$ is used to speed up the search for an inode when both
the inode number and the superblock object of the file system are known. To handle
possible collision the $i\_hash$ holds backward and forward pointer to those other inode
which have the same hash.

The methods of an inode listed in the $i\_op$ struct covering the management of inodes
and hard links are listed exemplarily in table 3.2. The following table contains only the
most interesting fields of the inode object as it includes an overall number of 46 fields.
The file object is created when the file is opened. It describes the process interaction with an opened file. Therefore, the most important information stored is the file pointer held in \texttt{f_pos} which represents the current position inside a file. The file pointer is kept in file object to allow concurrent access to the same file by different processes.

The operations of the file object \texttt{f_op} shown in 3.2 alter the data contents of a file such as read and write but also memory mapping or modifying the file pointer. The file object structure includes 20 fields of which a selection is shown in table 3.5

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{f_list}</td>
<td>Pointers for the file object list</td>
</tr>
<tr>
<td>\texttt{f_dentry}</td>
<td>Dentry object related to this file</td>
</tr>
<tr>
<td>\texttt{f_vfsmnt}</td>
<td>Mounted file system which contains the opened file</td>
</tr>
<tr>
<td>\texttt{f_op}</td>
<td>Pointer to the file methods table</td>
</tr>
<tr>
<td>\texttt{f_count}</td>
<td>Reference counter for the file object</td>
</tr>
<tr>
<td>\texttt{f_mode}</td>
<td>Process access mode</td>
</tr>
<tr>
<td>\texttt{f_pos}</td>
<td>Offset of the file pointer</td>
</tr>
<tr>
<td>\texttt{f_mapping}</td>
<td>Pointer to the \texttt{address_space} object related to the opened file</td>
</tr>
</tbody>
</table>

Table 3.5.: Fields of the file object - Based on [BC05, p.471]

The file descriptor of the opened file is held in the \texttt{FD} array field of the \texttt{files_struct} field in the process descriptor which is explained in section 2.9.

The file descriptor identifies the opened file and is the procedure a user application handles the file. However communication via the file descriptor as a shared memory is not possible as two different processes handle the same file in two different descriptors with different numbers.
Dentry object

When a directory is loaded, it is transformed by the VFS to a dentry object. For every part of a looked up pathname a dentry object is created. The dentry object links the path component to the related inode object. The pathname lookup is described in detail in section 3.1.1.

The fields \texttt{d_inode} and \texttt{filename} establish the link between the file name and corresponding inode object. The \texttt{d_mounted} field holds the number of file systems which are mounted in this directory as it is possible to mount more than one file system in a mount point. The dentry object can be in one of the following four states \textit{free}, \textit{unused}, \textit{in use} or \textit{negative}. An object which is free is neither used by the VFS nor does it contain any valid information. Whereas an unused dentry object is one which is not currently used but the \texttt{d_inode} fields still points to the related inode. An in use object is the one having a positive \texttt{d_count} usage counter and holding the related inode. Negative is the misleading term to describe a dentry object where the corresponding inode does not exist either because the inode was deleted or as a result of the pathname lookup of a non existent file.

The operations of a dentry object, listed in table 3.2, basically reading and modifying directories, are pointed to by the \texttt{d_op} fields.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Field & Description \\
\hline
\texttt{d_count} & Usage counter for the dentry object \\
\hline
\texttt{d_inode} & Holds the inode related with the filename \\
\hline
\texttt{d_name} & Filename \\
\hline
\texttt{d_child} & Pointer to the list of sibling dentries \\
\hline
\texttt{d_subdirs} & Pointer to the list of subdirectory dentries \\
\hline
\texttt{d_op} & Pointer the table of dentry methods \\
\hline
\texttt{d_sb} & Superblock object related to the file \\
\hline
\texttt{d_mounted} & Counter for the number of mounted file systems in this dentry \\
\hline
\end{tabular}
\caption{Fields of the dentry object - Based on [BC05, p.475]}
\end{table}

File system type registration

To keep track of whose code is currently included in the kernel the VFS performs the file system type registration [BC05, p.482]. The \texttt{file_system_type} object represents the registered file system. The fields contain file system type flags, the name, the methods for reading the superblock and a list of superblock objects having the same file system type.

The flags such as \texttt{FS_REQUIRES_DEV} and \texttt{FS_BINARY_MOUNTDATA} indicate whether every file system of this type has to be located on a physical disk and whether file system uses binary mount data.

The \texttt{register_filesystem()} is invoked for every file system specified at the compile
time when the system is initialised. It is also invoked when a file system is loaded via a kernel module.

Path name lookup

The path name lookup is the procedure of deriving the inode from the corresponding path name explained based on Bovet [BC05, p. 495ff]. Therefore the path name is analysed and split into parts each being a file name whereby every file name except the last needs to identify a directory.

A path name can be absolute or relative. Beginning with / indicates an absolute path and the search starts at the file system’s root directory. A relative path starts at the current working directory.

By this, the dentry object is determined. Now the entry matching the first part of the path name is examined and corresponding inode is derived. Then the directory including this inode is read from disk and the entry matching second part is examined to derive the next inode. This method is repeated until the last part of the path name.

Keeping the most recently used dentry objects in memory, the dentry cache speeds up this whole procedure.

The following checks have to be performed which complicates this method:

- The access rights for each directory have to be verified.
- The file name can be a symbolic link to any random path name. In this case the path name can be much longer than expected and the analysis needs to be continued for the rest of the path components.
- Symbolic links can also lead to circular references. Therefore the kernel needs to recognize endless loops.
- As a file name can be the mount point of another file system, the lookup must be able to continue in the new file system.
- As a path name may refer to different files in different name spaces, the lookup has to take place inside the appropriate processor’s name space.
The calling sequence of a system call via the VFS is depicted in Figure 3.1. To read data the related file must be opened first. Therefore the application invokes the open system call which calls the wrapper routine `sys_open` of the standard library. This wrapper routine passes the system call number and the file name as a parameter and issues either the `int $ 0x80` assembly instruction or the `sysenter` instruction which will perform the necessary storage operations and then change into kernel mode. After checking flags the work is passed to `do_sys_open`. An opened file in the kernel is represented by a file descriptor which is an integer number acting as an index for the process array (`task_struct -> files -> fd_array`). Since the file name is passed, the `do_filp_open` function needs to find the file inode. Therefore the `path_lookup` function is invoked and several checks on access rights are performed. Thereafter the newly generated file object is placed on the `s_files` list of the superblock and the `open` function of the specific file system via the related `file_operations` struct. The file object is then installed in `task_structure -> files -> FD` of the process and the control is passed back to the user process which returns the file descriptor to the application [Mau08, p.520-570].

3.2. Special file systems

3.2.1. `/proc-system`

The `proc` file system or `/proc` file system on Linux is considered a special file system. It does not need any storage space on any kind of hardware as the kernel generates the file contents when they are needed. Therefore the `/proc/version`, for example, has a
The output is created on the fly which is a very unusual behaviour for a file system [Mau08, p. 520].
The /proc file system provides information about the processes and the system such as the working directory, files to represent the virtual memory of a process or even information about the CPU. It also lists all supported file systems in the /proc/filesystems file list [Kou09]. The mount point is the directory /proc which explains the name.

### 3.2.2. Temporary file system (tmpfs)

This subsection is based on the work of Koutoupis [Kou09]. Both the RAM-disk and the temporary file system (tmpfs) are file systems only living in the RAM. This grants the advantage of a much higher speeds and the system can handle great workloads with very small latencies. At the same time, this is also the great disadvantage as all data stored will be lost when the system shuts down.

The difference between the RAM-disk and the tmpfs is their behaviour towards dynamically allocating memory.

Although, the tmpfs is mounted with either a fixed percentage of memory or an absolute value of bytes, it may be resized dynamically while being online. To reduce the memory used, tmpfs is able to swap unneeded pages.

In contrast the RAM-disk does not have set boundaries and is not able to swap. It eventually grows in the memory until no place is left and therefore crashes the system.

The tmpfs is a advancement of the RAM-disk providing memory monitoring and swapping. It is supported in Linux from version 2.4 on.
4. User space file systems

This chapter is about the FUSE project and its basic functionality. First, an overview of the components of the FUSE framework is given. Afterwards, the system call mechanism with FUSE and its additional overhead is explained. At last, the mount options are analysed and it is shown how to implement a file system with the FUSE API.

4.1. Filesystem in Userspace (FUSE)

The following section is based on the API of the FUSE project if not marked otherwise [Ose14]. The most important file systems such as ext4 and tmpfs are running in the kernel as this has been the only way for a long time. In the early 1980s, in the debate about micro-kernel developers came up with an idea which would allow mounting a file system in the user space. The FUSE project was started in 2004 and implemented in the Linux kernel by 2005.

FUSE is a file system framework consisting of the following parts:

- kernel module (fuse.ko): Providing the opportunity for a user to write an own file system without editing the kernel code and also avoid license problems. Additionally no recompilation or rebooting of the kernel is necessary.

- user space library (libfuse): Due to the offered stable library interface writing a file system becomes remarkably easier. This enables any user with basic programming skills to test their approach to a new file system without the need to study and change the kernel’s source code.

- mounting (fusermount): With this component the file system can be mounted not only with superuser privileges but also by nonprivileged user. There are several parameters which can be passed as a mount option. They are explained in the section 4.1.2

In the FUSE kernel module there are two separate parts namely a proc file system component and the file system calls. The first one, in kernel/dev.c, is responsible for handling the input and output requests to the file /dev/fuse. The second one holds the file system calls in kernel/file.c, kernel/inode.c and kernel/dir.c. All of those system calls invoke a request_send() function which adds those request to a structure holding a list of all requests. Depending on the variant, the request function either waits for a reply or continues without any response.
The mounting of the user space file system occurs when the user mode program calls the `fuse_main()`. This calls `fuse_mount` which creates a UNIX domain socket pair and executes `fusermount`, in `util/fusermount.c`. The UNIX domain socket is a way for processes to communicate which is defined in POSIX [pos99]. After ensuring the FUSE module is loaded, `fusermount` opens `/dev/fuse`. The related file descriptor is sent back to `fuse_mount()` via the UNIX domain socket and finally returned to `fuse_main()`. Then `fuse_new()` is invoked and thereby the structure to hold an image of the file system data is allocated. Afterwards, the file system system calls in `/dev/fuse` are read by either `fuse_loop()` or `fuse_loop_mt()` and the user mode functions, stored in the `fuse_operations` struct, are called. The related results are written to `/dev/fuse` and thereby passed to the system calls.

So, `/dev/fuse` enables the communication between the kernel module and the user space library.

FUSE offers two different APIs to implement a user space file system. The first one is a low-level interface, defined by the `fuse_lowlevel_ops` data structure, which is very similar to the VFS. For example the user space file system needs to manage the related inodes. ZFS-FUSE, discussed in section 5.4, is implemented using the `fuse_lowlevel_ops` data structure [RG10].

The other one, `fuse_operations`, is utilised by most of the FUSE-based file systems as the high-level API offers more abstraction. The developer do not need to concern themself with inodes but just with pathnames. The translation from inodes to pathnames is performed by `libfuse`.

### 4.1.1. System call with FUSE

The following subsection explains how a system call is performed with the additional overhead of FUSE, based on the work of Rajgarhia [RG10]. The FUSE kernel module registers the FUSE file system with the VFS as shown in figure 4.1. In the following, the user space file system is assumed to be mounted in `/mountpoint`. Now, if an application issues, for example, a `read()` system call for the file `/mountpoint/file`, the VFS passes the request to the related handler. Thereafter, the page cache is checked whether the data is already cached. In this case the data is returned immediately. Otherwise, the system call has to be forwarded via `/dev/fuse` to the FUSE library `libfuse`, which issues the related function in the user space file system `fuse_fs`. After performing the action, the read data is returned in the corresponding buffer from `libfuse` through the kernel to the calling application.

If an application calls a `read()` system call in a native file system such as ext4, there are no context switches between processes with different address spaces, as the system call is directly invoked and passed to the VFS.

This needs only a mode transition from user mode to kernel mode and vice versa after finishing.

Using a user space file system introduces at least two additional context switches: from
the issuing application to the user space library \texttt{libfuse} and back again. In section 5.5 the costs of a context switch are analysed. They depend on a number of factors such as the used hardware, the workload and also the memory access pattern. As they require the caches such as the TLB to be flushed, context switches can be very expensive.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fuse_diagram.png}
\caption{Path of the read() system call through FUSE}
\end{figure}

In table 4.1, some of the mostly used operations of the high-level API of the FUSE framework are shown. They look very similar to the system calls, making their usage intuitive which is necessary as the documentation is still "experimental" [Ose14].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read()</td>
<td>Read data from an open file</td>
</tr>
<tr>
<td>write()</td>
<td>Write data to an open file</td>
</tr>
<tr>
<td>getattr()</td>
<td>Get the file attributes</td>
</tr>
<tr>
<td>truncate()</td>
<td>Change the size of a file</td>
</tr>
<tr>
<td>open()</td>
<td>File open operation</td>
</tr>
<tr>
<td>mkdir()</td>
<td>Create a directory</td>
</tr>
</tbody>
</table>

Table 4.1.: Operations of the \texttt{fuse_operations} struct

\textbf{4.1.2. Mount options}

FUSE provides an extensive list of mount options, of which the chosen ones are of special interest. They can be specified within the mount command:
Mount parameter | Description                                                                                                                                 |
------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
large_read        | enables large_reads, only useful with kernel version 2.4 [man]                                                                            |
big_writes        | enables larger writes than 4 KB                                                                                                           |
max_write=N       | sets the maximum size of write requests, by default 128 KB [man]                                                                          |
direct_io         | use direct I/O, bypassing the page cache                                                                                                  |
write-back_cache  | asynchronous, buffered writes (fuse version 30, kernel version 3.15)                                                                         |

Table 4.2.: Mount options for FUSE-based file systems  
Based on [man]

The mount option large_read was only useful with the kernel version 2.4 as the kernel versions 2.6 and later manage the request size internally for greater performance. The measurements confirmed this since there was no difference between mounting without and mounting with large_reads.

The mount options listed in table 4.2 are discussed in chapter 5.

4.2. Implementation of a FUSE-based file system

In the following an overview is given, how to implement a file system with the help of FUSE.

For working focused, it is necessary to first decide what kind of file system one is aiming for. There are two different APIs, each providing a data structure holding the operation: fuse_operations struct and the fuse_lowlevel_ops struct.

Most file systems are based on the high-level API and fuse_operations as it offers a more abstract interface covering the more complex internals.

File systems such as ZFS-FUSE base on the low-level API and the related fuse_lowlevel_ops struct.

The advantage is that one is able to implement how the inodes of the VFS are handled and therefore has more possibilities for low-level optimisation.

Afterwards, the decision has to be taken which FUSE version should be used. At the moment, the last official version is 29. The write-back cache is first included in FUSE version 30, which can be cloned from the Git repository. However, it does need to be compiled by the user. Additionally, a newer kernel (3.15. or higher) is necessary to enable the desired support.

Listing 4.1 shows how the fuse_operations struct of the high-level API is used and how the fuse_main function is invoked. The choice of operations relates to the operations used in the memfs.
# define FUSE_USE_VERSION 30

#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

#include "emptyfs.h"

struct fuse_operations emptyfs_oper = {
    .chmod = emptyfs_chmod,
    .chown = emptyfs_chown,
    .create = emptyfs_create,
    .destroy = emptyfs_destroy,
    .getattr = emptyfs_getattr,
    .init = emptyfs_init,
    .link = emptyfs_link,
    .mkdir = emptyfs_mkdir,
    .open = emptyfs_open,
    .read = emptyfs_read,
    .readdir = emptyfs_readdir,
    .rmdir = emptyfs_rmdir,
    .statfs = emptyfs_statfs,
    .truncate = emptyfs_truncate,
    .unlink = emptyfs_unlink,
    .utimens = emptyfs_utimens,
    .write = emptyfs_write
};

int main (int argc, char* argv[])
{
    return fuse_main(argc, argv, &emptyfs_oper, NULL);
}

Listing 4.1: Emptyfs.c - example for implementing a FUSE-based file system
The operations specified in listing 4.1 are the subset which is also used in the memfs, which will be analysed in chapter 5.
The user is able to choose the methods needed for their file system and ignore those not important to the desired functionality.
If the file system is to be benchmarked later on, it is vital to either write an own benchmark or examine the benchmark internals to see what operations are tested and how far they need to actually work. Otherwise, the benchmark will produce errors.
A possible implementation for chown, which does nothing at all, is shown in listing 4.2.

```c
#include "emptyfs.h"

int emptyfs_chown (const char * path, uid_t uid, gid_t gid)
{
    return 0;
}
```

Listing 4.2: chown.c - example for implementing a FUSE-based file system
5. Comparison

In this chapter the comparison of tmpfs vs. memfs, a FUSE-based file system, as well as of ZFS-FUSE vs. ZFS on Linux takes place. Also the new FUSE version, 30, is analysed in contrast to 29. Therefore the Linux kernel 3.13 and 3.16 are used.

5.1. Tools

If not referred to otherwise, this section is based on the 'Linux Performance and Tuning Guidelines' by Ciliendo and Kunimasa [CK07].

There are several tracing and monitoring tools available to track what is happening inside the file system and the kernel. They are also able to evaluate the system’s workload and the memory usage.

A short overview is given on the advantages of the tools related to file system analysis.

5.1.1. Monitoring

- **top**: This command is used to display the process activity. There are different possibilities to sort the list such as the process ID (PID), age or the cumulative time. By default, the task are sorted by CPU usage.
  This tool is useful not only for file system monitoring but for every case the user wants to see the CPU workload and which process plays the biggest role.

- **vmstat/mpstat**: It offers information about the memory management. Therefore, the number of processes (waiting and sleeping), the memory (free, buffered, cached) as well as I/O traps and the CPU activity are displayed.
  The last is the most interesting one, if analysing file systems, as the time spent in kernel mode is distinguished from the one spent in the user mode. This is convienient for a basic overview for a FUSE-based file system to see how much time is spent in which mode. **mpstat** is similar to **vmstat** but provides the mentioned information for each of the available CPUs on the system.

- **free**: This command provides detailed information about the amount of memory used as well as the swap and also the kernel caches.
  **free** is useful to track for example whether flushing a cache was successful or how much swap is used if you create a high file system workload.
strace:
The system calls invoked by a process and the related signals are recorded with strace. It enables the possibility to see what system calls are called internally for a specific system call. As it is very low-level and detailed, the results become confusing quickly.

DTrace:
Based on the strace command, DTrace allows a more high-level study of the system calls. DTrace provides all in all more than 360 000 different probes (instrumentation points) so basically measuring everything is possible. A guide how to use DTrace is offered by Gregg [GM11], explaining also the used script language D. Unfortunately some of the most interesting probe packages are not available for Linux yet. Additionally, DTrace does not run with newer kernel version 3.15 and 3.16 which are necessary to examine the FUSE write-back cache.

5.1.2. Benchmarking
To measure a file system’s performance the IOzone file system benchmark is often used. It provides the measurement of number of file system operation and enables export in an Excel compatible format allowing simple evaluation of the data.

- iozone -a: runs the automatic mode
- -R: provides Excel compatible output
- -I: sets O_DIRECT and therefore enables direct I/O, bypassing the kernel page cache
- -O: output in operation/second
- fileop -f -s: measures the performance for file system operation, as read, write, open and mkdir
  -f: sets the force factor, 2^f, which determines how many files are analysed
  -s: sets the file size

5.2. Test environment
- Operating system: Ubuntu 14.4
  Linux kernel version 'old': 3.13.0-33-generic
  Linux kernel version 'new': 3.16.0-031600-generic
- Processor: Intel i5-480M
  Number of cores: 2
  Number of threads: 4
  Clock rate: 2.66 GHz
5.3. Performance of volatile memory file systems

In this section the meta data overhead as well as the performance of read and write operations are benchmarked for the temporary file system (tmpfs) and the memory file system (memfs) [Kuh08]. They both are file systems working only on volatile memory.

5.3.1. Meta data overhead

In the following, the Linux kernel version 3.13.0-33-generic and 3.16.0-031600-generic are compared with regard to the induced meta data overhead. Therefore, the benchmarks are run with a block size of 0.

Figure 5.1 and 5.2 show a comparison between the meta data operations create, delete, open and stat. Especially, for the tmpfs, a significant improvement of the performance is offered by the newer kernel. For the stat operation there is a speed up from 530 000 operations per second to 623 000 per second. Comparing both figures, the performance lack of memfs in contrast to tmpfs becomes obvious. The best throughput of memfs is about 35 000 operations per second whereas tmpfs is able to perform 600 000 operations per second. The second figure also implies the overhead between the different mount options of FUSE is nearly the same. The user can use whatever mount option is most fitting for his file system without adding additional overhead with a specific mount option.

As the performance of read and write is a about 10 times larger, those operations are depicted separately in figure 5.3. The difference for the writing operation of both file systems is significant as the performance is nearly tripled.

As seen before, the mount options behave as expected. There is an increase with the new cache for writing with every other option adding no boost or penalty. However, those results are so good, the question rose whether the benchmark was checking for a reading and writing size of 0 bytes.

Therefore, another measurement was run with a block size of 1 which still should enable an estimation of the overhead.

As shown in figure 5.4, there is a drastic performance drop to 42 000 operations per second for the writing operations of memfs without any mount options. The writing performance of tmpfs is halved, the one of memfs decreased by a factor of
126 whereas memfs with the write-back cache turned on decreased only by factor of 3.5. This is an improvement in relation to no mount options by a factor of 33! This demonstrates the enormous boost the write-back cache has on a FUSE-based file system when writing even small data.

The first run of writing operations with the write-back cache after restarting the whole system is slower than the following. Even so, mounting and unmounting the file system and even flushing the page cache shows no performance penalty with regards to the cache.

Figure 5.4 illustrates only the results of the runs with a filled write-back cache.

For this thesis there were also measurements run with the fileop script of IOzone. Unfortunately, those results were implying a dramatic performance drop for reading with the new kernel which could not be supported by any other measurement.

Therefore, those results are not discussed further.

It indicates, however, that the evaluation of file systems is not as simple as it seems.
Figure 5.2.: Meta data overhead: memfs - kernel version 3.13. vs 3.16.

Figure 5.3.: Meta data overhead: memfs - tmpfs - kernel version 3.13. vs 3.16.
Figure 5.4.: Meta data overhead: memfs - tmpfs - Block size 1

5.3.2. Iozone

To see the performance of a file system while operating on actual data, the following measurements with IOzone in automatic mode are performed. As the meta data analysis proved the newer kernel version to be faster, the other measurements will be only performed on the kernel version 3.16.

Reading

In figure 5.5 and 5.6 the read operations are depicted with file size at the abscissa and the record size at the ordinate.

The highest value for tmpfs is about 8 GB/s whereas the memfs is only about 6 GB/s when the file size equals the record size.

memfs is not able to keep up this impressive performance for file sizes bigger than 1024 kB. Figure 5.7 shows the impact of the write-back cache on reading operation which widens the cache peeks and also increases their height.

tmpfs obviously benefits from caching as performance on small file sizes is significantly higher than on large ones. Still, after dropping to the lowest performance of 1.9 GB/s, the performance grows again with the increasing file size as the overhead for opening the file has a greater impact than the record size.

The best performance for large files is about 4.5 GB/s whereas memfs is only capable of 1.6 GB/s for large files sizes.
Figure 5.5.: Performance of tmpfs: read - iozone

Figure 5.6.: Performance of memfs: read-iozone
Figure 5.7.: Performance of memfs with write-back cache: read-iozone

Writing

Figure 5.8 and 5.9 show that tmpfs is capable of 10 times the performance of memfs without additional mount options. While tmpfs increases its throughput with the growing file size, memfs struggles with large file sizes as well as the growing record size. Since memfs dynamically allocates the required memory on demand, there are performance drops due to the reallocation which can be seen as 'holes' in the surface chart.

Those effects are not visible in figure 5.10 and 5.11, since both the big writes and the write-back cache smooths the surface.

The mount option `big_writes` improves the performance by a factor of 4 from a maximum of 220 MB/S to 889 MB/s because the internal record size of FUSE for write operations is 4 kB. With `big_writes` a default block size of 128 kB is used reducing the overhead for the write operation. The user is able to set this size by adding `-o max_write=N` within the mount command, where N is the size in bytes.

Furthermore, the write-back caches increases the writing performance dramatically by a factor of 12 from 220 MB/s to 2 600 MB/s and even when the cache size is exceeded the performance is still capable of a maximum of 191 MB/s for a 65 000 kB file size.
Figure 5.8.: Performance of tmpfs: write - iozone

Figure 5.9.: Performance of memfs: write - iozone
Figure 5.10.: Performance of memfs with big writes: write - iozone

Figure 5.11.: Performance of memfs with write-back cache: write - iozone
5.4. Performance of disk-based file systems

To see how FUSE performs with disk-based file systems, additional measurements were included.
Since the kernel increases the number of the supported file systems steadily most FUSE projects are not maintained any more.
As the ZFS-FUSE project used the low-level API the developer put more thoughts into the development resulting in a better performance than most of the other FUSE-based file systems except the cluster file systems. Due to the licensing problems with ZFS and the Linux kernel, ZFS on Linux provides only a kernel module to enable the usage of ZFS in the kernel.
This allows the comparison of very similar file systems differing mostly in the usage of FUSE. Hence, having a native and a FUSE-based implementation of ZFS is suitable for measuring the overhead induced by FUSE.

ZFS originally stands for zettabyte file system as it supports 128-bit addresses [BAH+03]. It is not a simple file system as it also provides error detection, copy on write (COW) and RAID-Z [Top12].

5.4.1. Reading

For ZFS and ZFS-FUSE the performance is only analysed on the newer kernel version as all previous measurements indicate an improvement.
Figure 5.12 and 5.13 show the reading operation of ZFS and ZFS-FUSE which display the same basic behaviour although ZFS obviously benefits from the internal caches. Those caches lead to a boost of over 8 GB/s. In regard to large files, both file systems are capable of a similar capacity around 3 GB/s.
As this is an impressive performance, there the question rose what impact the caches have to this result.
Unfortunately, there was no possibility to measure the operation without the caches effect as ZFS does not support the kernel flag `O_DIRECT` which is used by IOzone to bypass the kernel page cache. ZFS-FUSE, in contrast, does not fail, but has exactly the same results as before, which implies the flag is simply ignored.
Figure 5.12.: Performance of zfs: read - iozone

Figure 5.13.: Performance of zfs-fuse: read - iozone

5.4.2. Writing

In figure 5.14 and 5.15, the performances of writing to an actual disk are depicted. In contrast to the reading operation, the writing operation of both file systems are not as capable. The performance for the native ZFS more than halved, 3.5 GB/s as maximum, whereas ZFS-FUSE drops to about 580 MB/s which is a decreasing factor of about 10. Regardless of the good performance for writing small files, the performance decreases significantly for large files with a size of 65,536 kB or larger. The write-back cache is not yet available for ZFS-FUSE as it does not support the FUSE
version 30 which is needed for the new mount option. Those results are clearly influenced by the caches as the disk is not capable of such throughput. As already mentioned before, there was no simple possibility to bypass the kernel to measure the actual effect of the cache.

![zfs write performance graph]

Figure 5.14.: Performance of zfs: write - iozone

![zfs-fuse write performance graph]

Figure 5.15.: Performance of zfs-fuse: write - iozone

### 5.5. Time of a context switch

As context switches have an impressive impact on the performance of a file system, there is the need to know how long such an operation takes.
However, it is not possible to present exact results for a general situation. Since the context switch may be a switching of tasks either having the same address space (threads) or not (processes). Therefore, very different contexts are needed which may need TLB flushing or not. The flushing and restoring of those caches has the greatest influence on the time needed for a context switch. David et al. found the direct cost of a context switch to be 48 microseconds with an additional indirect overhead of 17% to 25% of the task’s direct overhead [DCC07]. They performed those measurements for various pairs of tasks, too. Those indirect costs are also analysed by Li et al. who found them to vary between several microseconds and more than 1 000 microseconds [LDS07]. Refilling the L2 cache has an enormous impact on the performance. Additionally, the test environment may affect the accuracy as background handling of file system interrupts disturb the measurements. Those difficulties lead to vague values for the time of a context switch around 1 000 - 30 000 cycles of a CPU [Hü13].
6. Conclusion

The aim of this thesis is to enable a comparison between kernel and user space file systems.
Therefore, the preceding chapters provide an insight to the system internals of the operating system as well as different file systems.
The comparison in chapter 5 implies several points.
First, the kernel version 3.16 has an impressive performance gain even for a native RAM file system such as tmpfs which is normally seen as the top of the file systems’ performance.
Also, the performance of the meta data operation of a FUSE-based file system does not depend on the used mount options.
While, in general, the FUSE-based file systems show a lower performance than the native file systems, seeing the FUSE project only as inferior to the kernel file systems is doing FUSE wrong. Especially, the write-back cache introduced with FUSE version 30 and the kernel 3.15, shows an enormous impact on the writing performance of the memfs.
At least, for file sizes from 64 kB to 4 MB the operations per second reach the same range as the tmpfs, which is an impressive result for every file system but particularly for a user space file system. And although the performance for file size between 4 MB and 500 MB of memfs halves, it is still capable of 500 000 kB per second which is five times the amount the file system is able to perform without the write-back cache.
In contrast to the kernel version 3.13, the performance increased even by a factor of seven.

For ZFS-FUSE, the reading performance is a brilliant result which shows the upper end for the writing performance is yet not reached.
The drastic improvement with the newer kernel and the write-back cache shows the potential the FUSE project still has, if someone is just interested in working on it.
To encourage the users and the community to do so, the developers need to work on the FUSE documentation as the less frequently used operations do not have any description at all.
Thus, the user has to study the examples to get a sense what the function is supposed to do which is not functional.
The FUSE project is a great possibility to develop file systems for a specific use without changing the kernel code also enabling mounting for non privileged users.
But as the kernel includes a list of over 40 file systems which do not need the additional context switches, FUSE has to improve its presentation before it is too late.
Bibliography


List of Acronyms

CPL  Current Privilege Level.  10, 12, 16

cs  code segment.  10

DPL  Descriptor Privileged Level.  12, 16

fd  File Descriptor.  22, 27, 30

FUSE  Filesystem in Userspace.  4, 9, 17, 32–35, 38–41, 45, 48, 50, 52

GDT  Global Descriptor Table.  12, 16

LDT  Local Descriptor Table.  12

TLB  Transition Lookaside Buffer.  13, 34, 51

TSS  Task State Segment.  16

VFS  Virtual File System.  23–25, 28, 30, 33, 35
List of Figures

2.1 Protection ring ................................................. 11
2.2 Logical address translation ................................. 12
2.3 The Linux paging model .................................... 13
2.4 System call ..................................................... 15
2.5 Write-back cache with write allocation ................. 20
2.6 Write-through cache with no-write allocation .......... 21
3.1 virtual file system ........................................... 30
4.1 Path of the read() system call through FUSE .......... 34
5.1 Meta data overhead: tmpfs - kernel version 3.13. vs 3.16. 41
5.2 Meta data overhead: memfs - kernel version 3.13. vs 3.16. 42
5.3 Meta data overhead: memfs - tmpfs - kernel version 3.13. vs 3.16. 42
5.4 Meta data overhead: memfs - tmpfs - Block size 1 ...... 43
5.5 Performance of tmpfs: read - iozone ................... 44
5.6 Performance of memfs: read-iozone ...................... 44
5.7 Performance of memfs with write-back cache: read-iozone 45
5.8 Performance of tmpfs: write - iozone .................. 46
5.9 Performance of memfs: write - iozone .................. 46
5.10 Performance of memfs with big writes: write - iozone 47
5.11 Performance of memfs with write-back cache: write - iozone 47
5.12 Performance of zfs: read - iozone .................... 49
5.13 Performance of zfs-fuse: read - iozone ................ 49
5.14 Performance of zfs: write - iozone ................... 50
5.15 Performance of zfs-fuse: write - iozone ............... 50
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Access times for the different memory levels (rough estimate)</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Fields of the <code>task_struct</code></td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>System calls related to file systems - Based on [Pat03] and [BC05, p.460]</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Objects of the VFS and a selection of their operations - Based on [Tan09, p.908] and [BC05, ch.12]</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Fields of the <code>super_block</code> structure - Based on [BC05, p.462 ff]</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Fields of the inode object - Based on [BC05, p.467f]</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Fields of the file object - Based on [BC05, p.471]</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Fields of the dentry object - Based on [BC05, p.475]</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Operations of the <code>fuse_operations</code> struct</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Mount options for FUSE-based file systems</td>
<td>35</td>
</tr>
</tbody>
</table>
Appendices
A. Usage Instructions

A.1. FUSE

A.1.1. FUSE version 29
   • sudo apt-get install libfuse-dev

A.1.2. FUSE version 30
   • git clone http://git.code.sf.net/p/fuse/fuse fuse-fuse
   • sudo apt-get install autoconf libtool
   • ./makeconf.sh
   • ./configure
   • make
   • sudo make install

A.2. File systems

A.2.1. tmpfs
   • Basic (max. 50% of the RAM):
     sudo mount -t tmpfs /PathToMountpoint
   • Relative (20% of RAM):
     sudo mount -t tmpfs -o size=20% none /PathToMountpoint
   • Absolute (200 MiB):
     sudo mount -t tmpfs -o size=200M none /PathToMountpoint

A.2.2. memfs
   • make
   • ./source/memfs [-o options] /PathToMountPoint
• for FUSE version 30: set flags in makefile to \texttt{pkg-config fuse3}

• signature of \texttt{filler()} changed, it has got an additional parameter 'kernel flag'

\textbf{A.2.3. ZFS-FUSE}

• \texttt{sudo apt-get install zfs-fuse}

• Create a partition 'name' on /device to use for zfs.

• \texttt{sudo zpool create name [type] /device}

• check \texttt{sudo zpool list} and \texttt{sudo zpool status}, to see whether it was successful

• \texttt{df}, to see all the mounted file system and their corresponding mount points. Normally 'name' of type zfs-fuse should be already mounted in '/device'. If you want to use the FUSE mount options you need to unmount 'name'.

• unmounting: \texttt{sudo umount name}

• mounting: \texttt{sudo zfs mount [-o options] name}, where -o takes the FUSE parameters for mounting

\textbf{A.2.4. ZFS}

• \texttt{sudo apt-get install ubuntu-fuse}

• Create a partition 'name' on /device to use for zfs.

• \texttt{sudo zpool create name [type] /device}

• check \texttt{sudo zpool list} and \texttt{sudo zpool status}, to see whether it was successful

• \texttt{df}, to see all the mounted file system and their corresponding mount points. Normally 'name' of type zfs should be already mounted in '/device'

• mounting: \texttt{sudo zfs mount name}

• unmounting: \texttt{sudo umount name}

\textbf{A.3. Benchmarking}

\textbf{A.3.1. IOzone}

• \texttt{sudo apt-get install iozone3}

• \texttt{fileop -f filenumber -s filesize}, benchmarking the most common file related operations
iozone -a -R -b excel_outputfile.xls, benchmarking the read and write operation in automatic mode with results in excel format

A.4. Tracing

A.4.1. Git and DTrace

- sudo apt-get install git
- git clone https://github.com/dtrace4linux/linux
- depending on your operating system run the related skript
  - tools/get-deps-arch.pl, if using ArchLinux
  - tools/get-deps.pl, if using Ubuntu
  - tools/get-deps-fedora.sh, if using RedHat/Fedora
- cd ..
- sudo make all
- sudo make install
- sudo make load

A.4.2. DTraceToolkit

providing a collection of documented DTrace scripts:

- download latest DTraceToolkit on http://www.brendangregg.com/dtraceToolkit.html
- gunzip and 'tar xvf' the file
- sudo apt-get install ksh, for installing the ksh
- open install.sh and uncomment the line 'PATH=/usr /bin ...
- sudo ln -s /bin/sh /usr/bin/sh, creates a symbolic link, as the directory which contains the shell is not what the developers assumed it would be
- change to the toolkit directory
- ./install

A.4.3. mpstat/ sysstat

- apt-get install sysstat
A.4.4. perf

- `sudo apt-get install linux-tools` will install the tools for the latest Linux release, which will cause an error if you are booting a previous Linux version.

- therefore find your Kernel release by typing: `uname -r`

- `sudo apt-get install linux-tools-3.2.0.65`
Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 28. August 2014 ................................................