Folie 1:
Der Inhalt dieser Präsentation beschäftigt sich mit dem Thema FUSE im Rahmen des Proseminars ” Proseminar Speicher- und Dateisysteme“ an der Universität Hamburg. Die Teilnehmer des Seminars sollen lernen, wie Speicher und Dateisysteme in der heutigen Zeit funktionieren und sich mit diesen in Form einer Präsentation und einer Ausarbeitung auseinandersetzen. Es soll die damit vorhandene Möglichkeit genutzt werden, um eigene Dateisysteme zu schreiben. Dies ist besonders interessant, da Unix-Systeme nicht alle Dateisysteme nativ unterstützen und es sonst mit erheblich mehr Aufwand verbunden ist, sein eigenes Dateisystem zu schreiben. Es ist aber auch für Einsteiger geeignet, um sich einen einfachen Überblick zu schaffen..
Folie 2:
Die Gliederung wurde gewählt um einen guten Einstieg in das Thema Dateisysteme sowie in den grundlegenden Aufbau des Linuxkernel zu erhalten, um zu verstehen, inwiefern das Implementieren eines eigenen Dateisystems sinnvoll und zum Teil auch nötig ist. Schließlich wird FUSE noch einmal genauer betrachtet und so mit Hilfe der zuvor erklärten vier Grundbefehlen ein eigenes kleines C-basiertes Dateisystem zu schreiben. Am Ende werden noch gängige Fehler in FUSE erläutert und ein Fazit gezogen.

Folie 3:
Der Begriff FUSE bedeutet Filesystem in Userspace und beschreibt ein Kernel-Modul für Unixsysteme. Durch FUSE ist es möglich Dateisystem-Treiber in den Kernel-Mode in den User-Mode zu verlagern. So ist es für Benutzer möglich ihre eigenen Dateisysteme zu mounten(einzubinden), welche dann User-Mode laufen.[1]

Folie 4:
Um aber die Grundlage für ein tieferes Verständnis von FUSE zu legen, müssen zwei Fragen en vorher erläutert werden. Das wären: Was ist überhaupt ein Dateisystem und was ist der User-Space genau? Ein Dateisystem ist eine Ablageorganisation auf einem Datenträger eines Computers. Sie organisiert die geordnete Ablage von Dateien durch das Betriebssystem. So können durch das Dateisystem Dateien gespeichert, gelesen, verändert oder gelöscht werden. Die Dateien selber haben fast immer mindestens einen Dateiname und Attribute(Größe etc). So wird eine Art Inhaltstabelle aufgebaut, in der alle Dateien und ihre Speicheradressen verwaltet werden. Dateinamen sind üblicherweise in Verzeichnissen abgelegt, welche meist spezielle Dateien sind , die Verweise auf die jeweiligen Dateien enthalten. So verfügen alle Dateien in einem Dateisystem über ihre eigene eindeutige Adresse, über welche sie von dem Betriebssystem aufrufbar sind. [2]

Folie 5:
Um den Vorteil von FUSE zu verstehen, nämlich den Dateisystemtreiber aus dem Kernel-Mode in den User-Mode zu verlagern, muss der Aufbau eines Betriebssystems genauer betrachtet werden. Ein Betriebssystem enthält zwei Modi, nämlich den Kernel-Modus und den User-Modus. Um dieses Modell zu erklären, wird die Ringgraphik herangezogen. Damit ein Betriebssystem sicher und gut läuft, werden den Prozessen verschiedene Sicherheits-und Privilegierungsstufen zugewiesen. Diese Stufe schränkt den Handlungsraum des jeweiligen Prozesses entsprechend ein. So ist das Betriebssystem(bzw. dessen Kernel) im innersten Ring und hat alle Zugriffsrechte. Es darf direkte Hardwarezugriffe vornehmen, ohne die jeweiligen Dienste zu beauftragen. Anwendungsprogramme laufen normaler Weise im äußersten Ring. Die restlichen Ringe ,welche nicht mehr „alles dürfen“, werden User-Mode genannt und haben strikte Restriktionen.[1] So ist sichergestellt, dass Anwendungsprogramme keine Prozesse in

anderen Privilegierungsstufen beeinflussen können und sich auch nicht aus ihrer Stufe befreien. Dies schafft die sogenannte Speichervirtualisierung, bei der jeder Prozess seinen eigenen abgeschotteten Bereich im Speicher zugewiesen bekommt.[4]

Folie 6:
Nach dem die Grundlagen erklärt wurden kommen wir nun wieder auf FUSE zurück.User-Space-Dateisysteme haben einige deutliche Vorteile. Der erste, dass sie sich nicht im Kernel befinden, ermöglicht es ziemlich einfach, den Code zu verteilen. Das heißt, dass der Code nicht durch die extrem strengen und manchmal erschwerenden Prozesse der Prüfung, Bewertung, Rechtfertigung sowie der stilistischen Änderungen am Code, (weg ist sinnlos:Test, Bewertung, Begründung) muss. Hoffentlich war der Entwickler des User-Space-Dateisystem aufmerksam in punkto Sicherheit, Wartungsfreundlichkeit, Wechselwirkungen mit anderen Benutzeranwendungen usw., so dass der Code relativ sicher ist für Tests. Das im User-Space enthalten sein bedeutet auch, dass das Dateisystem-Code früher getestet werden muss, da ein Bug in der Regel nur zum Absturz der Anwendung und nicht zu einem Kernel Absturz führt. Entwickler können Vorabversionen veröffentlichen, bei denen noch nicht alles getestet ist und dann unmittelbar Rückmeldung über die Nützlichkeit des Dateisystems erhalten.[1]

Der zweite Vorteil ist, dass bei einer Erweiterung oder bei Vorhandensein eines Bug-Fixes , das Dateisystem (keine Wartezeiten für ein paar Kernel-Versionen) schnell aktualisiert werden kann.[6] Da das Dateisystem sich im User-Space befindet, kann es wie jede andere Anwendungssoftware über geeignete Distributionswerkzeuge aktualisiert werden. Dafür muss auf jeden Fall das Dateisystem einmal ausgehängt und wieder eingehängt werden,[6] was jedoch ein gewöhnlicher Prozess für einen administrativen Vorgang bei einer Wartung ist.

Ein dritter Vorteil ist, dass bei Laufen des Dateisystems im User-Space nicht das ganze Betriebssystem abstürzt , wenn es ein Fehler auftritt. Für den Absturz eines Kernel Dateisystem gibt es eine Reihe von möglichen Ursachen. Deswegen haben sowohl Kernel- als auch Dateisystem-Entwickler größte Sorgfalt in die Prävention eines Kernel Absturz gelegt. [6] Bei Laufen des Dateisystems im Kernel ist das Risiko der Erzeugung einer Reihe von Seiteneffekten größer als im User-Space, welches höchstens den Prozess zum Absturz bringen kann. So kann bei einem Fehler der Prozess beendet und das Dateisystem einfach neu eingehängt werden. Dabei könnten Beschädigungen an Dateien auftreten. Dieser Nachteil könnte jedoch durch entsprechende Mechanismen der Entwickler in Grenzen gehalten werden.[5]

Folie 7:
Es sind derzeit mehr als 20 verschiedene Sprachen Bindings für FUSE verfügbar, welche es Anwendern erlauben eine Dateisystem in einer anderen Sprache als C zu schreiben. Das ermöglicht Entwicklern verschiedene Programmierparadigmen , verschiedene Weise der Typsicherheit und Typzusicherung und so ist FUSE auch gedacht für verschiedene Anwendungsszenarien. Als Beispiel erlaubt das Benutzen von C++ oder C# das Entwickeln in einer hoch performanten, objektorientierten Sprache. Auf der anderen Seite stellen funktionale Sprachen wie Haskell und OCaml Werkzeuge wie Funktionen höhere Ordungen und lazy evaluation, welches die Modalität erleichtert und so wiederum mehr Produktivität ermöglicht.

Einige Programmiersprachen eignen sich besonders für bestimmte Anwendungen. Zum Beispiel wurde die Erlang Programmiersprache entwickelt, um fehlertolerante , in Echtzeit

laufende, dauerlaufende Anwendungen in verteilten Umgebungen zu unterstützen. [7] Als solches wurde es in einige verteilten Speichersystemen verwendet.
Halfs ist ein Dateisystem, das mit Hilfe von Haskell entwickelt wurde und sich gut eignet für die Entwicklung mit hoher Sicherheit sowie die Methodik für die Erstellung von Software-Systemen , welchen streng definierten Spezifikationen entsprechen.[8] Die Sprache Ruby , in der jeder Datentyp ein Object ist, wurde entwickelt, um gutes Interface-Design zu fördern und bietet viel Flexibilität für den Programmierer . [9]Ruby wird unter anderem verwendet um den Zugriff auf eine meteorologische Datenbank durch ein lokales Dateisystem zu gewährleisten.[9]

Folie 8:
Es stehen unteranderem FUSE-Dateisystemtreiber für die Hardware basierten Dateisysteme ext2,iso,ZFS bereit, für die Netzwerk basierten Dateisysteme NFS, smb,SSH

Folie 9:Die Hauptschwierigkeit beim Schreiben eines Dateisystems liegt im Interagieren mit dem Kernel. Im Einzelnen muss das Dateisystem mit dem Kernel VFS (Virtual File System)interagieren, um Zugriff auf die Hardware zu bekommen.
Bevor es FUSE gab war es lange Zeit einfach nicht möglich ein Dateisystem im User-Space zu schreiben. Jedes Dateisystem musste im Kernel-Space implementiert werden (in der Regel als Teil des Kernel oder als Modul). So stieß man bei der automatischen Ver- oder Entschlüsselung von Daten auf ein Problem. Das gleiche Problem entstand beim Komprimieren und Dekomprimieren, was zum Beispiel nötig ist , um sich die Daten in einer tar-Datei anzeigen zu lassen ohne die tar-Datei zu entpacken. [10] Diese Art von Operationen waren einer Integration im Kernel nicht möglich. Sie sollten also in den User-Space, aber trotzdem mit dem Kernel interagieren.
So wurde entschieden , dass ein eigenes VFS im User-Space implementiert wird welches jedoch mit Hilfe des Kernel-VFS handeln sollte [10]. So entstand FUSE(Filesystem in Userspace). Das Konzept war die Schaffung eines Kernel-Moduls, welches mit dem Kernel VFS im Auftrag nicht-privilegierter Nutzer interagiert und über eine einfache API zum Adressieren verfügt.

Um nun näher auf die Funktionsweise von FUSE einzugehen muss man wissen, dass FUSE aus drei Systemteilen besteht. [6] Der erste Teil ist das Kernelmodul , welches in den Code des VFS gehängt wird und wie ein Dateisystemmodul aussieht. Es beinhaltet allerdings eine spezielle Zugriffsmethode, welche von einem Userspace Prozess angesprochen werden kann. Wenn nun auf die Methode zugegriffen wird, hat sie die Aufgabe Dateisystemanfragen zu akzeptieren , in den das eigenen Protokoll zu übersetzen und die Anfragen über das Methoden Interface zu senden. Die Antworten der Anfragen kommen vom Userspace über die FUSE Methode zurück und werden dann wieder in die Form übersetzt, die der Kernel akzeptiert .

Im Userspace implementiert FUSE eine Bibliothek, welche die Kommunikation mit dem Kernelmodul übernimmt. Die Bibliothek akzeptiert Anfragen der FUSE Einheit und übersetzt sie in ein Set von Methodenaufrufen, welche den Methodenaufrufen des Kernel VFS ähneln. Diese Methoden sind unteranderem getAttr(),readdir,open(),read(),etc.

Als letztes gibt es die vom Entwickler eingebundene Komponente, welche erst das eigentliche Dateisystem implementiert. Diese Komponente füllt die struct fuse_operations

mit Pointern auf die Methoden für die Implentimierung der geforderten Operationen. Dieses Interface ist nicht allzu gut dokumentiert, wird aber in dem folgenden Beispiel später einfach dargestellt.

Das alte Dateisystemmodul AVFS benutzt FUSE um ein Dateisystem aus tar-und zip Dateien zu machen. Eine andere Möglichkeit wäre es ein Dateisystem zu erstellen, welches eine Website spiegelt mit Zugriff auf eine Object Database. FUSE ist die ideale Basis um die Idee hinter „alles ist eine Datei“ auszuführen.[6]

Folie 10:
Die Abbildung bildet das für Beispiele oft gewählte „Hello World“ als FUSE-Operation ab. Auf einer hohen Ebene kompiliert das „Hello World“ Dateisystem eine Binärdatei „hello“.Diese Binärdatei wird ausgeführt in der rechten oberen Ecke der Abbildung mit einem System Mount point /tmp/fuse. Daraufhin führt der Anwender ein ls -l Kommando gegen den Mount point ls -l /tmp/fuse. Dieses Kommando geht über die glibc zum VFS. Das VFS geht dann weiter zum FUSE Kernelmodul, da der Mount point zu einem FUSE basierten Dateisystem gehört. Das FUSE Kernelmodul kontaktiert dann über die glibc und libfuse(libfuse ist die FUSE Bibliothek ,welche sich im Userspace befindet) die eigentliche Binärdatei(„hello“).Die Binärdatei gibt das Ergebnis den Stack herunter zurück an das FUSE Kernelmodul , zurück durch das VFS und schließlich zurück zum ls -l Kommando.[11] Hierbei wurde auf einige Teile nicht genau eingegangen ,da die Abbildung nur den generellen Fluss der Operationen und Daten darstellen soll.

Folie 11:
Laut[12] ist ein nicht zu unterschätzender Punkt in der Entwicklung eines Dateisystems ist die Performance. Wenn ein natives Dateisystem wie Ext3 benutzt wird , so existieren nur zwei user kernel Mode Switches für eine Dateisystemoperation (zum und vom Kernel). Hier müssen keine Kontextwechsel gemacht werden, indem der Term Kontextwechsel genutzt wird, um dem Scheduler die Referenz zwischen Prozessen mit verschiedenen Adressräumen zu zeigen. User-Kernel Modus Switches sind nicht aufwendig und sind nur eingebunden, indem sie den Prozessor vom unprivilegierten Modus in den privilegierten Modus schalten oder anders herum. Das Benutzen von FUSE fordert nun für jeden Dateisystemaufruf zwei Kontextwechsel. Der erste Kontextwechsel ist von der User Anwendung, welche den Aufruf ausgelöst hat zur FUSE User Space Bibliothek und der andere in die andere entgegengesetze Richtung. So geschieht es, dass wenn FUSE weitere Operationen an das darunterlegende Dateisystem weitergibt, insgesamt zwei Kontext- und Modiwechsel durchgeführt werden. Ein Kontextwechsel hat gewisse Performancekosten. Auch wenn die Kosten von einer Menge Faktoren abhängen wie Prozessortyp, Workload und Speicherverwaltung. So müssen letztendlich bei einem Wechsel für den anstehenden Prozess die Prozessorregister gesichert und wiederhergestellt werden . Die Cache und translation lokalste buffer (TLB) Einträge müssen geflushed werden und dann erneut geladen werden, dann muss die Prozessorpipeline ebenfalls geflushed werden.

So trennt FUSE auch Lese und Schreibzugriffe in 128KB große Blöcke, um die Anzahl der sogenannten „Dirty Pages“ im Kernel zu minimieren. Wenn nun dem Kernel nicht ausreichend genug Speicher zur Verfügung steht aufgrund einer großen Anzahl von Dirty Pages, kann es zu Deadlocks kommen , wenn vom User-Space Prozess, welcher das dementsprechende Dateisystem implementiert, gewechselt wird. Da bis noch vor einiger Zeit FUSE Anfrage in Blöcke mit 4KB getrennt wurden , führte dies zu einer signifikanten Anzahl von Kontextwechseln für jede read-oder write-Operation , welche auf einer Buffergröße von mehr als 4KB ausgeführt wurde. Die Blockgröße von 128KB mildert

dieses Problem seitdem die meisten Anwendungen, beinhaltend auch gebräuchliche UNIX Werkzeuge wie cp,cat,tar und sftp, normalerweise Buffer benutzen die größer als 32KB sind. Ab dem Linuxkernel 2.6.27 aufwärts bietet sich mit der big_writes mount Option eingeschaltet eine Blockgröße von 128KB. Die Abbildung zeigt die benötigte Zeit im eine 16MB Datei zu schreiben mit einer variablen Blockgröße von 4KB zu 128KB.[12]

Folie 12:
Die FUSE Bibliothek stellt die Kommunikation mit dem Socket sicher und leitet die Anfrage an die jeweilige Implementierung weiter. Dies geschieht durch den sogenannten „Callback“ Mechanismus. Die Callbacks sind wie schon vorher beschrieben, eine Anzahl Funktionen, die implementiert werden müssen, um das Verhalten eines Dateisystems zu modellieren. Das struct fuse_operations enthält eben diese Callbacks. In unserem Beispiel „Hello World“ wird das Callback struct hellp_oper genannt. Von insgesamt über 34 Dateisystemoperationen werden für ein einfaches Dateisystem nur 4 Operationen benötigt, um die grundlegendsten Funktionen durchzuführen.

Folie 13:
Diese Funktionen innerhalb des Callbacks stellen -wie bereits geschrieben- die nötigen Dateisystem Operationen bereit . Diese Operationen lassen sich grob in folgende Kategorien einteilen: Verzeichnis-Operationen, Datei-Operationen und Metadateien-Operationen. Diese Operationen sind nicht nur in FUSE vorhanden, sondern in ähnlichem Aufbau in fast jedem Dateisystem. Mit den Verzeichnis-Operationen wird auf die Verzeichnisse zugegriffen und dann modifiziert. So können Verzeichnisse z.B. erstellt oder gelöscht werden. Die Datei-Operationen enthalten die Funktionen Dateien zu löschen ,zu verschieben oder umzubenennen. Besonders zu erwähnen sind die Funktionen read(),open() da diese einerseits später im Beispiel erläutert werden und sich mit insgesamt 4 solcher Operationen schon ein einfaches Dateisystem schreiben lässt. Mithilfe von Metadateioperationen lassen sich Attribute wie Größe und Pfad der Datei bestimmen und auslesen.

Folie 14:
Hier sind verschiedene Verzeichnisoperationen dargestellt.

Folie 15:
Hier sind einige grundlegende Dateioperation dargestellt.

Folie 16:
Hier sind einige grundlegende Dateioperation dargestellt.

Folie 17:
Hier sind einiger Metadateioperationen dargestellt.
Folie 18:
Da mithilfe von FUSE ein eigenes Dateisystem implementiert wird, kann es natürlich zu Fehlern kommen. Dafür wirft FUSE sogenannte Exceptions, die dementsprechend behandelt werden. Häufige Fehler sind unteranderem errno.ENOSYS, welcher auftritt wenn bestimmte Funktionen nicht implementiert sind.

Folie 19:
Hier folgt nun ein Beispiel, welches verdeutlichen soll,wie ein einfaches Dateisystem aufgebaut ,programmiert ist und funktioniert.

Folie 20:
Im Folgendem Beispiel sieht man nun wie eine einfaches Dateisystem in FUSE mit der Hilfe der Sprache C geschrieben werden kann. Für das Dateisystem brauchen wir in unserem Fall eine Datei und einen Pfad in dem wir die Datei auffinden können. In diesem sehr einfachen Beispiel sind Pfad als auch Datei als Stringliteral realisiert. Die Datei ist in diesem Fall „Hello World“ ,der Pfad ist /hello. Wir brauchen um dieses Dateisystem mount fähig zumachen nur vier Funktionen implementieren: getAttr(),readdir(),open() und read(). getAttr() liest die Metadaten der Datei aus ,in diesem Fall ist es hello_str und ist somit eine Metadateioperation. readdir() liefert die Verzeichniseinträge für jeder Datei im Verzeichnis und ist eine Verzeichnisoperation. open() öffnet die Datei und read liest die Datei aus. Diese beiden sind Dateioperationen.

Folie 21:
Mit der Methode hello_getattr() sollen nun die Attribute der „Datei“ hello_str ausgelesen werden. Dabei wird der Methode der Pfad der „Datei“ und ein Buffer übergeben. Der Buffer wird ,wenn die Operation erfolgreich durchlaufen wird mit dem Mode,nlink und der größer der Datei bestückt, welche in unserem Fall die Länge des Stringliteral „Hello World“ ist. Die erste if-Abfrage prüft ob es sich um eine Datei oder ein Verzeichnis handelt. Handelt es sich um eine Pfade wird der st_mode auf S_IFDIR gesetzt, welches nichts weiteres bedeutet als das es sich um eine einfaches Verzeichnis handelt. Der st_nlink -Wert ist die Anzahl der „Hard links“,also die Anzahl der Referenzen auf andere Dateien. Da es sich um eine leeres Verzeichnis handelt wird der nlink auf 2 gesetzt, einen für . und einen für den Dateieintrag in der Elterndatei. Wenn Pfad jedoch hello_path entspricht, wird der mode auf IFREG gesetzt, also das es sich um eine reguläre Datei handelt. Da es sich um eine Datei handelt wird nlink auf eins gesetzt und die Größe der Datei in st_size gespeichert.Die Größe ist wie bereits erwähnt die Länge des Strings. Wenn keine der beiden Bedingungen greift wird der Fehler ENOENT ausgeben, welcher bedeutet,dass es keine entsprechende Datei in dem jeweiligen Verzeichnis existiert.

Folie 22:
Mit der Methode hello_readdir soll das Verzeichnis des einfachen Dateisystems auslesen. Hierfür wird vorher überprüft ,ob es sich beim übergebenen Pfad tatsächlich um ein Verzeichnis handelt. Ist die nicht der Fall wird wieder der Fehler ENOENT ausgegeben. Dann wird dem filler (eine Funktion [Fuse_fill_dir_t] mit deren Hilfe wir Einträge in das Verzeichnis einfügen werden) die Einträge für das Verzeichnis übergeben.

Folie 23:
Mit der Methode hello_open soll die Datei als ReadOnly geöffnet werden, da in UNIX-Systemen bei einem open-Befehlt die Datei vorerst als ReadOnly geöffnet wird und erst bei einer Modifikation die Datei als ReadWrite neu geöffnet wird. Als erstes wird abgefragt,ob die richtige Datei angesprochen wurde, da dieses Dateisystem nur eine Datei hat. Wenn nicht wird der Fehler ENOENT ausgeben. Wenn bei der Datei nicht das ReadOnly Flag gesetzt ist wird der Fehler EACCES ausgeben,welche bedeutet,dass der Zugriff nicht erlaubt wird. Zu erwähne ist ,dass FUSE ein paar Übersetzungen durchführt. Der Systemeigene open() Aufruf gibt eine Dateibeschreibung wieder keine 0. So erkennt FUSE wenn eine 0 gesendet wurde und gibt eine Dateibeschreibung wieder(nicht unbedingt den Gleichen, der bei Aufruf des open()-Aufrufs erhalten wurde).

Folie 24:
Mit der Methode hello_read wollen wir den Inhalt der Datei lesen,welcher hier der Inhalt des Strings,also „Hello World“ ,ist. Hierbei wird als erstes das Vorhandensein der Datei geprüft inklusiver Fehlerausgabe bei Fehlen der Datei. Dann wird die Länge der Strings gespeichert ,da wir später den String auf einen Buffer schreiben wollen um in auslesen zu können und dafür den Startpunkt und Endpunkt des String brauchen. Der offset gibt uns die Position des letzten char in dem String „hello World“ also das Ende der Datei. Damit wir nun den String in den Buffer kopieren können muss sichergestellt sein,dass nicht über die Datei,den String hellp_str, hinaus gelesen und kopiert wird,weswegen die Indizes korrigiert werden.

Folie 25:
Nun initialisieren wir die Callbacks mit den im Beispiel vorgestellten Funktionen. Dieses Dateisystem kann nun gemountet werden und erfüllt die einfachsten Operationen eines Dateisystems.

Folie 26:Zusammenfassend lässt sich sagen ,dass FUSE ein gutes Werkzeug ist ,um einfacher ein eigene Dateisystem zu implementieren. Durch das Kernelmodul,welche das Übersetzen der Befehle in den Kernelkontex übernimmt, muss letztendlich nur die Implementation stimmen, damit das Dateisystem funktioniert. So ist das aufwendige Modellieren eines nativen, im Kernel existierenden Dateisystem,nicht mehr nötig. Auch im Hinblick auf die Performance wird FUSE immer angenehmer. Durch die mittlerweile relative große Chunksize von 128KB , müssen nun deutlich weniger Kontextwechsel gemacht werden, um Dateienoperationen performanter durchzuführen. Durch das breite Spektrum an unterstützten Programmiersprachen können auch hinsichtlich des Verwendungszweckes als auch der persönlichen Vorliebe geeignete Lösungen entwickelt werden. Es ist auch hinsichtlich des Sicherheitsaspektes eine gut geeignete Alternative ,da auch nicht privilegierte Benutzer Dateisysteme mounten können ,ohne dabei in Prozesse des Kernels oder andere Programme eingreifen zu können.

Die oben erwähnte Performanz hat aber auch durch die nötigen Kontextwechsel ihre Grenzen. So kann es zu Engpässen kommen , wenn viele einzelne Operationen in kurzer Zeit hintereinander ausgeführt werden. Aber auch FUSE stößt an seine Grenze wenn es darum geht Kernelfunktionalität zu überschreiben(z.B dentry cache).

Folie 27:
Quellenangaben:

[1] https://de.wikipedia.org/wiki/Filesystem_in_Userspace (aufgerufen am 13.08.2014)

[2]http://wiki.ubuntuusers.de/dateisystem (aufgerufen am 28.08.2014)

[3]https://de.wikipedia.org/wiki/Dateisystem(aufgerufen am 28.08.2014)

[4]https://de.wikipedia.org/wiki/Ring_%28CPU%29

[5]Network Distributed File System in User Space Ivan Voras, Mario Žagar Faculty of Electrical Engineering & Computing, University of Zagreb,
[6]FUSE – Filesystems in Userspace London Open Solaris Usergroup, Nov'08 Frank Hofmann, Open Solaris FUSE, presented by Frank Hofmann.
[7]https://de.wikipedia.org/wiki/Erlang_%28Programmiersprache%29(aufgerufen 03.09.2015)

[8]https://hackage.haskell.org/package/halfs (aufgerufen 03.09.2015)

[9]https://www.ruby-lang.org/de/ (aufgerufen 03.09.2015)

[10]http://www.netzmafia.de/skripten/unix/unix1.html (aufgerufen 03.09.2015)

[11]http://fuse.sourceforge.net/ (aufgerufen 11.09.2015)

[12] http://www.csl.sri.com/users/gehani/papers/SAC-2010.FUSE.pdf(aufgerufen 23.09.2015)