Konzepte von Betriebssystem-Komponenten
VFS und ramfs als einfache Anwendung

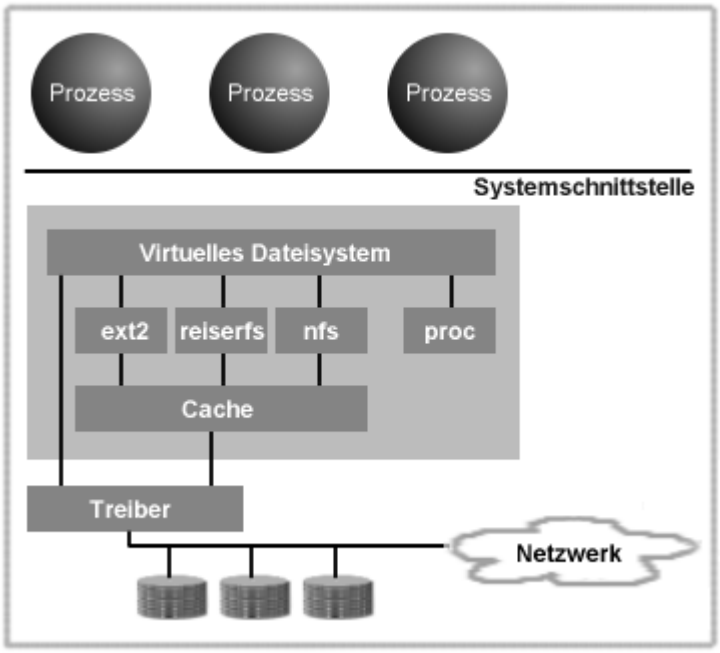
Max Lindner - 12.01.2004

Wozu ein Dateisystem und was macht es?

Ein Dateisystem ist notwendig, um Informationen sortiert auf der Festplatte abzulegen und sie geordnet wieder abrufen zu können. Dabei gibt es bei der Vielzahl der vorhandenen Dateisysteme teilweise Unterschiede, zum Beispiel bei der Behandlung von Groß-/Kleinschreibung oder der Rechteverwaltung. Aber im Grunde ist jedes Dateisystem dazu da, Informationen zu speichern und wiederzugeben. Dies geschieht mit Hilfe einer Verzeichnissstruktur. Die Operationen auf Dateisysteme sind dabei größtenteils gleich.

Wozu ein VFS (Virtual Filesystem Switch / Virtuelles Dateisystem)?

Auf Grund der Gemeinsamkeiten der Aufgaben und Operationen auf Dateisysteme lässt sich eine gemeinsame Schnittstelle schaffen, die diese Operationen abstrahiert und einen gemeinsamen Befehlssatz zur Verfügung stellt. Dies konnte jedoch nicht direkt in den Systemaufrufen geschehen. Dort wären die Funktionen fest einkodiert und es wäre schwieriger gewesen, neue Dateisysteme hinzuzufügen. Statt dessen wurde eine Softwareschicht im Kern geschaffen, die diese Aufgabe übernimmt. Diese Schicht nennt sich VFS und ist seit langem in den Linux Kernel implementiert.



Dateimodell des VFS

Das Dateimodell des VFS ist dem des ext2 nachempfunden, da dieses Dateisystem mit einem minimalen Overhead betrieben werden soll. Es bietet eine große Funktionalität. Jedes Dateisystem, dass in das VFS eingebunden werden soll, muss sich nach diesem Modell richten. Dabei muss aber nicht die volle Funktionalität implementiert sein. Es reicht beispielsweise auch, wenn man nur einen Lesezugriff implementiert. Dann kann von diesem Dateisystem nur gelesen werden.
Die wichtigstens Funktionen, die vom VFS bereitgestellt werden, sind unter anderem mount() und umount(), die Dateisysteme ein- und aushängen, stat() für Dateiinformationen, mkdir(), rmdir(), readdir() und rename() für Verzeichnisszugriffe und -manipulationen, chown(), chmod(), open() und creat() um Dateien zu modifizieren und read() und write() um Datei zu lesen und zu schreiben.
Dateien sind im VFS Inodes auf dem Datenträger, die durch einen Index einzigartig in diesem Dateisystem identifiziert werden können. Inodes enthalten Attribute einer Datei wie Benutzer- und Gruppenzugehörigkeiten, Zugriffsrechte und -zeiten, Dateigröße und

Anzahl der Hardlinks auf diese Inode, sowie Ortsinformationen ihres Inhalts.
Hardlinks sind Verzeichniseinträge, die auf eine Inode zeigen. Dateisysteme, die nicht mit Inodes arbeiten (zum Beispiel FAT), müssen diese simulieren, um dem Dateimodell des VFS zu entsprechen. Ebenso müssen Verzeichnisse als Dateien simuliert werden, wenn sie nicht von vornherein als solche behandelt werden. Der Dateiname wird nicht im Inode abgelegt, sondern im Verzeichnis. Dort finden sich die Verknüpfungen zwischen Dateinamen und Inode-Nummer.
Das Dateimodell des VFS ist objektorientiert aufgebaut. Jedes Objekt beschreibt Datenstrukturen und Methoden, die auf diese Strukturen zugreifen. Dabei ist zu beachten, dass Linux nicht in einer objektorientierten Sprache geschrieben ist und Objekte daher nur mit Datenstrukturen realisiert werden können.
Der Vorteil solcher “Objekte” ist eine größere Modularisierung der einzelnen Bestandteile, da sich hier Funktionen mit Hilfe von Pointern in Strukturen einbinden lassen. Außerdem wird so die Kommunikation innerhalb des VFS vereinfacht.
Das VFS arbeitet mit vier Objekten: Dem Superblock-, Inode-, Datei- und Dentry-Objekt. Das Superblock-Objekt enthält Informationen über das Dateisystem, verschiedene Methoden beispielsweise für Quotas sowie Variablen für maximale Dateigrößen, Anzahl der Blöcke, Inodes, wie viele davon frei sind und diverse Attribute des Dateisystems. Einige Methoden des Superblock-Objekts arbeiten mit Informationen aus dem Inode-Objekt. Dabei wird vom Datenträger ins Inode-Objekt geschrieben und umgekehrt. Das Inode-Objekt enthält alle Informationen, die vom Dateisystem benötigt werden, um eine Datei zu behandeln. Zu dieser Struktur gehören neben Feldern für die oben erwähnten Dateiattribute auch unter anderem ein Verweis auf den Superblock, ein Feld für Methoden auf Inodes wie Erstellen und Löschen von Dateien und Änderungen von Rechten sowie Felder für Standardmethoden auf Dateien und dateisystemspezifische Informationen.
Während sich das Superblock- und Inode-Objekt mit dem Inode an sich beschäftigen, steht das File-Objekt zwischen Prozess und den anderen Objekten. In den Feldern dieses Objekts befinden sich Angaben zu Dateisystem, Benutzer und Gruppe des Prozesses, verschieden Variablen für den Dateizeiger sowie Zeiger auf das entsprechende Dentry-Objekt und auf eine Struktur mit Dateioperationen (unter anderem Lesen und Schreiben einer Datei, Positionieren des Dateizeigers, Öffnen und Schließen einer Datei). Der Unterschied zu den bereits erwähnten Objekten besteht darin, dass dieses Objekt nur im Speicher existiert und dazu keine entsprechende Struktur im Dateisystem existiert. In einem Dentry-Objekten wird der Verzeichnisinhalt gespeichert. Dabei wird für jedes Verzeichnis ein eigenes Dentry-Objekt angelegt. Diese Dentry-Objekte werden auch im sogenannten Dentry-Cache gespeichert. Dies dient lediglich der Leistungssteigerung.

Auflösen des Pfadnamens

Wenn mit Dateien gearbeitet wird, wird dem VFS nur der Pfadname mitgeteilt. Das VFS muss überprüfen, ob der Pfad existiert. Dazu wird der Pfad aufgesplittet und jeder Teil einzeln überprüft. Trenner ist dabei sinnigerweise der “/”, der die Verzeichnisse trennt. Grundlage für eine erfolgreiche Überprüfung ist, dass nur der letzte Dateiname kein Vereichnis darstellt. Es gibt zwei Möglichkeiten, einen Pfad anzugeben. Zum einen relativ zu einem anderen Verzeichnis (außer dem Wurzelverzeichnis) oder absolut zum Wurzelverzeichnis. Beginnt der Pfad also mit einem “/”, so ist er absolut. Dies bestimmt den Ausgangspunkt der Pfadauflösung. Für diesen ist das Inode bekannt. Von diesem Ausgangspunkt wird jetzt für jeden Teil des Pfades das Inode bestimmt, indem das entsprechende Verzeichnis in ein Dentry-Objekt eingelesen wird. Natürlich wird bei jedem Teil überprüft, ob der Verzeichnissinhalt gelesen werden darf, das Verzeichnis eine symbolische Verknüpfung ist (in diesem Fall muss der Pfad dieser Verknüpfung

ebenfalls aufgelöst werden) und ob ein Dateiname nicht der Mountpoint eines eingehängten Dateisystems ist (dann muss die Auflösung im neuen Dateisystem fortgesetzt werden). Symbolische Verknüpfungen sind Dateien, in denen der Pfad einer anderen Datei gespeichert ist. Dieser Dateiname muss ebenfalls aufgelöst werden. Bei symbolischen Verknüpfungen ist jedoch auch eine gewisse Vorsicht geboten. Zu viele davon blähen den aufzulösenden Pfad sehr auf. So können symbolischen Verknüpfungen die Auflösung eines Pfadnames wesentlich beeinträchtigen. Daher sind hier kernelseitig ein paar Limitierungen gegeben: Es können maximal 40 symbolische Verknüpfungen insgesamt bearbeitet werden.
Bei der Pfadauflösung wird zwischen zwei Methoden unterschieden. Die normale Methode, die den Pfad bis zum Ende auflöst und eine Methode, die den Pfad bis zum vorletzten Dateinamen auflöst. Das ist zum Beispiel bei rmdir() nötig, weil die Inode (also das Verzeichnis) haben will, in dem die Operation ausgeführt wird.

Hinzufügen und Entfernen von Dateisystemen

Die Prozedur, bei der ein weiteres Dateisystem in die Verzeichnisstruktur eingehängt wird, nennt sich mounten. Das ist die übliche Verfahrensweise, ein Dateisystem zu benutzen. Bei diesem Vorgang ist einiges zu beachten. Das Verzeichnis, in das das Dateisystem eingehängt wird (der sogenannte Mountpoint), wird Wurzelverzeichnis des Dateisystems genannt. Ist das Wurzelverzeichnis des Dateisystems auch das Wurzelverzeichnis der Verzeichnisstruktur, dann ist dieses Dateisystem das Wurzeldateisystem.

Es können mehrere Dateisysteme in einen Mountpoint eingehängt werden. Dabei ist aber nur das zuletzt eingehängte Dateisystem sichtbar. Ein Dateisystem kann seit Kernel 2.4 mehrmals eingehängt werden.

Um ein Dateisystem zu benutzen, muss es im Kernel bekannt sein. Dafür muss es entweder fest in den Kernel kompiliert werden oder als Modul vorhanden sein. Das VFS registriert jedes Dateisystem, dass im Kernel fest einkompiliert beziehungsweise als Modul nachgeladen wird. Dabei wird für jedes Dateisystem ein neues Element in die Liste der Dateisystem-Typen eingefügt. Ein Element enthält Informationen über den Namen, Flags (unter anderem ob ein Gerät dafür vorhanden sein muss) sowie Methoden, den entsprechenden Superblock des Dateisystems auszulesen und in das Superblock-Objekt zu schreiben.

Das Einhängen des Wurzeldateisystems ist ein spezieller Fall. Ohne dieses Dateisystem kann kein weiteres eingehängt werden, weil keine Verzeichnisstruktur existiert. Es ist zum Systemstart also unbedingt erforderlich. Diese Prozedur geschieht in zwei Schritten. Als erstes wird das sogenannte rootfs eingehängt, das nur aus einem leeren Verzeichnis besteht, welches später als Mountpoint für das Wurzeldateisystem dient. Dies ist sinnvoll, weil so das Austauschen des Wurzeldateisystems vereinfacht wird. Oft wird nämlich zuerst ein minimales Dateisystem mittels einer Ram-Disk eingehängt, mit dessen Hilfe geeignete Treiber/Module für die Hardware geladen werden, bevor dann das echte Wurzeldateisystem eingehängt werden kann, welches sich auf einem Medium befindet, wofür im Kernel keine Treiber fest einkompiliert sind. Eine Realisierung des Ganzen nennt sich initrd (Initial Ramdisk). Damit kann man beim Booten auch schon auf Module zugreifen, was sonst erst nach dem Eingehängen des normalen Wurzeldateisystems funktioniert.
Das Einhängen funktioniert folgendermaßen:
Es beginnt mit der Überprüfung, ob der der aktuelle Prozess auch Root-Berechtigungen

hat. Als nächstes wird der Dateisystemtyp ermittelt und anschließend gespeichert. Nun folgt die Allozierung eines Dateisystem-Deskriptors, dessen Adresse ebenfalls gespeichert wird. Jetzt werden Name und Superblock initialisiert. Beim initialisieren des Superblocks werden allerdings unterschiedliche Methoden angewandt, je nachdem, welche Flags der Funktion übergeben wurde. Ist das einzuhängende Dateisystem ein spezielles Dateisystem, das kein Gerät braucht (wie das rootfs), so wird ein fiktives Blockgerät alloziert, mit dem dann weitergearbeitet wird. Wird ein Gerät benötigt, so wird als erstes der Pfad zum Mountpoint aufgelöst. Anschließend wird das Blockgerät geöffnet und in der Liste der Superblock-Objekte nach einem eventuell schon existierenden Superblock-Objekt gesucht. Ist dies erfolgreich, so ist dieses Dateisystem bereits eingehängt und wird ein weiteres mal mit dem gleichen Superblock-Objekt eingehängt werden. Anderenfalls wird ein neues Superblock-Objekt alloziert und mit Hilfe der Methoden dieses Objekts der Superblock des Geräts eingelesen. Jetzt wird bei beiden Methoden die Adresse des Superblocks zurückgegeben. Nun werden Superblock-, sowie Wurzelverzeichnis- und Moutpoint-Feld des Dateisystem-Deskriptors mit dem zurückgebenen Superblock beziehungsweise der Adresse aus dem Dentry-Objekt des Wurzelverzeichnises initialisiert. Zuletzt wird noch das Feld des “darunterliegenden” Dateisystems initialisiert und schließlich die Adresse des Dateisystem-Deskriptors zurückgegeben.
Das rootfs wird während der Systeminitialisation eingehängt. Später kann auf dieses leere Verzeichnis im 2.Schritt das Wurzeldateisystem eingehängt werden. Dies geschieht gegen Ende der Systeminitialisation.

In dieses Wurzeldateisystem können dann andere Dateisysteme eingehängt werden.

Beim Aushängen eines Dateisystems wird überprüft, ob der Pfad gültig ist, ob dort keine Dateien mehr geöffnet sind und ob dort überhaupt ein Dateisystem eingehängt ist. Dann wird überprüft, ob der Benutzer berechtigt ist, das Dateisystem auszuhängen. Ist eine Bedingung nicht der Fall, so wird der Vorgang abgebrochen. Ist das auszuhängende Dateisystem das Wurzeldateisystem, so wieder es nur-lesend erneut eingehängt. Befinden sich unterhalb des Mountpoints weitere eingehängte Dateisysteme, so werden diese mit ausgehängt.

Literaturverzeichnis:

-Goodheart, B., The magic garden explained, Prentice Hall, 1994
-Bovet D.P., Ceasti M., Understanding the Linux Kernel , Second Edition, O'Reilly 2002
-Beck M, u.a., Linux Kernelprogrammierung, 6.Auflage, Addison-Wesley 2001
-http://www.linuxfibel.de/filesys.htm