Proseminar
Konzepte von Betriebssystemkomponenten

Thema:

Module

von Michael Balda
am 02.02.2004

1 Allgemeines zum Modulkonzept

1.1 Was sind Module? Wozu dient das Modulkonzept?

Module sind Codeelemente, die nach Bedarf in den Kernel geladen werden können und somit die Funktionalität des Kernels erweitern. Das Modulkonzept wurde mit dem Linux-Kernel 1.2 eingeführt, vorher war es erforderlich, die gesamte Funktionalität des Kernels statisch in den Kernel-Code einzukompilieren. Dies hat zur Folge, dass der Kernel mit zunehmender Vielseitigkeit bezüglich Hardwareunterstützung und Funktionalität sehr groß und komplex wird. Mit dem Modulkonzept ist es hingegen möglich, generische Kernels zu erzeugen, die eine weite Vielfalt an Hardware unterstützen, indem je nach Bedarf die entsprechenden Gerätetreiber als Module eingebunden werden und der statisch gelinkte Kernel-Code auf die Grundfunktionalität beschränkt bleibt.

1.2 Vor- und Nachteile des Modulkonzepts

Das Modulkonzept ist mit einigen Vorteilen verbunden, die ein derartiges Konzept für ein modernes PC-Betriebssystem unverzichtbar machen. Durch das Einbinden von Modulen können Funktionen zum Kernel hinzugefügt werden, ohne dass ein Neustart erforderlich wäre. Würde man stattdessen Funktionen statisch in den Kernel integrieren, wäre das Neukompilieren des Kernels und ein Neustart des Systems unumgänglich. Fast alle Aufgaben des Kernels können von Modulen realisiert werden, somit stehen dem Anwender fast alle theoretischen Vorteile sogenannter Minimalkerne zur Verfügung, d.h. der Kernel selbst realisiert nur einige wenige Grundfunktionen, während die sonstige Funktionalität von Modulen bereitgestellt wird, aber vor allem aus Performancegründen wird mit Modulen das Konzept von Minimalkernen nur teilweise realisiert.
Die Möglichkeiten des Modulkonzepts sind jedoch mit einigen Einschränkungen verbunden, so ist es z.B. nicht möglich, in einem Modul Datenstrukturen zu verändern, die statisch in den Kernel integriert sind. Außerdem muss der Kernel solche Funktionen statisch integriert haben, die zum Einbinden und Entladen von Modulen erforderlich sind.

1.3 Unterschiede zwischen Modulen und normalen Anwendungen

Ein Unterschied zwischen Modulen und normalen Programmen liegt in der Art ihrer Anwendung: Ein Programm wird gestartet, seine *main*-Funktion aufgerufen, der Prozess läuft, vollendet eine oder mehrere Aufgaben und kann wieder beendet werden. Das Modul hingegen wird zunächst nur initialisiert, anschließend steht seine Funktionalität nach Bedarf zur

Verfügung, das Modul „läuft“ aber nicht. Werden seine Funktionen nicht mehr benötigt, kann das Modul wieder entfernt werden.
Ein Programm hat Zugriff auf Funktionen aus verschiedenen Standard-Bibliotheken wie etwa *printf* aus der Bibliothek *libc*. Da Module zum Kernel gehören und dieser auch ohne derartige Bibliotheken laufen sollte, stehen diese den Modulen nicht zur Verfügung. Ein Modul hat lediglich Zugriff auf die Symbole, die der Kernel oder andere Module exportieren, z.B. die Funktion *printk*, die mit printf vergleichbar ist und vom Kernel bereitgestellt wird.
Bei der Programmierung von Modulen ist auch noch ein weiterer Unterschied von entscheidender Bedeutung: Ist ein normales Programm fehlerhaft, kann es einen *Segmentation Fault* verursachen, der für den Rest des Systems keine Auswirkungen hat und verhältnismäßig leicht zu debuggen ist - ein durch ein Modul verursachter *Kernel Fault* hingegen kann schwerwiegende Auswirkungen auf die Systemstabilität haben.
Der entscheidende Unterschied ist, dass ein Modul im *Kernel Space* läuft und eine Anwendung im *User Space*. Im *User Space* gelten zahlreiche Einschränkungen wie etwa kein direkter Hardwarezugriff oder kein unautorisierter Speicherzugriff, somit ist gewährleistet, dass das Betriebssystem und andere Anwendungen geschützt sind. Diese Einschränkungen entfallen im *Kernel Mode*, also hat ein Modul Möglichkeiten, die einer normalen Anwendung nicht zur Verfügung stehen, Fehler können aber wesentlich kritischere Auswirkungen haben.

1.4 Aufgaben des Kernels bei der Modulverwaltung

Neben der Bereitstellung von Funktionen zum Laden und Entladen von Modulen muss der Kernel folgende zwei Hauptaufgaben bei der Modulverwaltung bewältigen:
Erstens muss gewährleistet sein, dass der Kernel Zugriff auf die globalen Symbole (Funktionen, Variablen etc.) der Module hat und umgekehrt ein Modul auf die globalen Symbole von Kernel und anderen Modulen zugreifen kann.
Zweitens sind die Abhängigkeiten von Modulen zu überwachen: Ein Modul darf nicht entladen werden, wenn ein anderes Modul dieses referenziert, d.h. von einem Modul abhängt, welches Funktionen bereitstellt, die das Modul zu seiner Arbeit braucht. Zum Beispiel könnte ein Modul *msdos* zur Unterstützung von MS-DOS-Partitionen ein Modul Namens *fat* referenzieren, das Funktionen für den Umgang mit auf einer FAT basierenden Dateisystemen bereitstellt.

2 Das Modulkonzept in Linux

2.1 Die Modutils

Für die Verwaltung der Module sind in Linux die Komponenten aus dem Paket *modutils* zuständig. Die wichtigsten Programme sind:

- ***insmod*** zum Einbinden von Modulen
- ***rmmod*** zum Entfernen von Modulen
- ***lsmod*** zum Anzeigen von Informationen über geladene Module (Inhalt der Datei /proc/modules)
- ***modprobe*** zum Einbinden von Modulen unter Beachtung der Abhängigkeiten

Modprobe installiert Module eigentlich wie *insmod*, überprüft aber vorher, ob für die Verwendung dieses Moduls noch andere Module von Nöten sind und lädt diese vorher. Die Abhängigkeitsinformationen finden sich in der vom Programm ***depmod*** erstellten Datei *modules.dep*.

2.2 Module mit kmod nach Bedarf laden

Es ist jedoch nicht immer erforderlich Module selbst zu laden. Unter Linux steht der sogenannte *kmod*-Prozess zur Verfügung, dessen Aufgabe es ist, Module selbstständig einzubinden, wenn der Kernel auf die Funktionalität eines bestimmten Moduls zurückgreifen muss oder es zu entfernen, wenn es nicht mehr gebraucht wird. Versucht also der Kernel auf eine Ressource zuzugreifen (z.B. auf eine MS-DOS-Partition), die z.Zt. nicht zur Verfügung steht, bringt er nicht etwa eine Fehlermeldung, sondern ruft eine Funktion des *kmod*-Subsystems auf und *kmod* versucht anschließend, diese Ressource durch Laden eines oder mehrerer Module (z.B. msdos und fat) verfügbar zu machen. Erst wenn *kmod* scheitern sollte, bringt der Kernel eine Fehlermeldung.

2.3 Das Modulobjekt

Die Informationen, die der Kernel von einem Modul haben muss, werden im sogenannten *Modulobjekt* verwaltet. Wird ein Modul geladen, reserviert der Kernel Speicher für den Modulnamen, das Modulobjekt und den Code des Moduls. Der Objektcode der Module liegt normalerweise als „.o"-Dateien im Verzeichnis /lib/modules/*Kernelversion*/ vor.
Die Modulobjekte werden in einer verketteten Liste verwaltet. Diese Liste beginnt mit einem Modulobjekt das auf ein fiktives Modul mit dem Namen *kernel module* verweist. Es repräsentiert den statisch gelinkten Kernel-Code.
Die wichtigsten Elemente des Modulobjekts:

Typ	Name	Kurzbeschreibung
struct module *	next	Zeiger auf nächstes Listenelement
const char *	name	Zeiger auf nullterminierten Namensstring
unsigned long	size	Größe des Moduls
atomic_t	uc.usecount	Referenzzähler
unsigned int	ndeps	Zahl referenzierter Module
unsigned int	nsyms	Zahl exportierter Symbole
struct module_symbol *	syms	Liste exportierter Symbole
struct module_ref	deps	Liste referenzierter Module
struct module_ref	refs	Liste abhängiger Module
int (*)(void)	can_unload	Gibt 1 zurück, falls Modul in Benutzung
int (*)(void)	init	Zeiger auf Initialisierungsmethode
void (*)(void)	cleanup	Zeiger auf Cleanup Methode

Tabelle 1: Die wichtigsten Felder des Modulobjekts

Da es sich um eine verkettete Liste handelt, wird ein Zeiger auf das nächste Element (*next*) benötigt. Der Name des Moduls wird als null-terminierter String (*name*) direkt hinter dem Modulobjekt abgelegt. *Size* gibt die gesamte Größe des Moduls incl. Modulobjekt und Namens-String an.
Bei *uc.usecount* handelt es sich um den Referenzzähler, mit dessen Hilfe überprüft werden kann, ob das Modul in Benutzung ist oder entfernt werden darf. Wird auf das Modul zugegriffen, wird der Zähler inkrementiert, nach Benutzung wieder dekrementiert.
In der struct *syms* wird eine Liste aller Symbole abgelegt, welche das Modul exportiert, also für andere Module bzw. den Kernel zugreifbar macht. In *deps* bzw. *refs* werden Listen mit abhängigen bzw. referenzierten Modulen abgelegt. Beim Laden eines Moduls, das von

anderen Modulen abhängt (bzw. andere Module *referenziert*), wird die *refs*-Liste in den zu diesen Modulen gehörigen Modulobjekten um einen Verweis auf das referenzierende Modul erweitert (siehe Abbildung).
Zudem enthält das Modulobjekt noch Zeiger auf Funktionen, die im Modul selbst implementiert werden: *Init* wird beim Einbinden des Moduls aufgerufen und dient zur Initialisierung des Moduls. *Can_unload* ist optional und wird beim Entfernen des Moduls aufgerufen, um festzustellen, ob das Modul entladen werden kann. Ist diese Funktion nicht implementiert, wird stattdessen der Referenzzähler verwendet. *Cleanup* wird ebenfalls vor dem Entfernen aufgerufen und soll z.B. Speicher freigeben oder abschließende Log-Einträge verfassen.

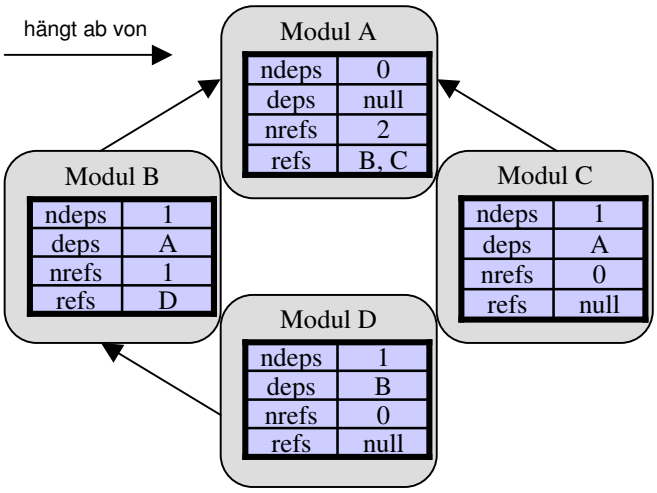

Abb. 1:Repräsentation Abhängigkeiten im Modulobjekt

2.4 Einbinden von Modulen

Wird ein Modul mittels „insmod *Modulname*“ installiert, sucht insmod zunächst die Datei, in der sich der Objektcode des Moduls befindet. Üblicherweise liegen diese Dateien in „/lib/modules/*Kernel-Version*/“ (kann aber auch in „/etc/module.conf“ oder einer Shell-Variablen festgelegt werden). Anschließend wird der Speicherbedarf ermittelt und die Berechtigung des Nutzers überprüft. Nach dem Reservieren des Speichers wird das Modulobjekt initialisiert, der Name des Moduls in den Speicher hinter das Modulobjekt kopiert und das Modulobjekt in die verkettete Liste eingehängt.
Nachdem die Adressen der exportierten Symbole von Kernel und anderen Modulen über die Liste der Modulobjekte ermittelt wurden, wird der Modulcode „*reloziert*“, d.h. alle Verweise auf globale und externe Symbole durch deren logische Adress-Offsets ersetzt.
Dann wird das Modul in einen eigenen Speicherbereich kopiert, die deps- bzw. ndeps-Felder im Modulobjekt gesetzt und die komplette Liste der Modulobjekte durchlaufen um die refs- bzw. nrefs-Felder der entsprechenden Modulobjekte zu setzen. Das Einbinden des Moduls wird durch die Ausführung der moduleigenen init-Funktion abgeschlossen.

2.5 Entfernen von Modulen

Ein bereits eingebundenes Modul kann mit „rmmod *Modulname*“ wieder entfernt werden. Mit der Option „-r“ können dabei auch die Module entfernt werden, die das entsprechende Modul referenzieren (ein Modul, das noch referenziert wird, kann nicht entfernt werden). Dazu werden zunächst diese Abhängigkeiten mit Hilfe der Einträge der *refs*-Felder in den Modulobjekten bestimmt und anschließend eine Liste der zu entfernenden Module in entsprechender Reihenfolge aufgebaut. Ist „–r“ nicht gesetzt, enthält die Liste nur das beim Aufruf angegebene Modul.
Nun werden die Listeneinträge folgendermaßen abgearbeitet: Zunächst wird die Berechtigung des Aufrufenden überprüft, anschließend das Modulobjekt in der verketteten Liste bestimmt und überprüft, ob das *refs*-Feld keine Einträge enthält (sonst Abbruch). Über die *can_unload*-Funktion oder den Referenzzähler *uc.usecount* wird sichergestellt, dass das Modul gerade nicht benutzt wird. Ist dies der Fall, kann die *cleanup*-Funktion aufgerufen werden und der Verweis auf das Modul aus der *refs*-Liste der referenzierten Module entfernt werden.

Mit dem Aushängen des Modulobjekts aus der verketteten Liste und der Freigabe des Modul-Speichers ist das Entfernen abgeschlossen.

3 Literaturliste

A. Rubini, J. Corbet. Linux Device Drivers (2nd Edition). O'Reilly & Associates, 2001.

D. P. Bovet, M. Cesati: Understanding the Linux Kernel (2nd Edition), O'Reily & Associates, Inc., 2003

The Linux Kernel Module Programming Guide Ver. 2.4.0
(http://www.faqs.org/docs/kernel/, Peter Jay Salzmann, 04.04.2003)

Vorlesungsskript Systemprogrammierung I
(http://www4.informatik.uni-erlangen.de/Lehre/WS02/V_SP1/Skript/, Franz J. Hauck, 2002)