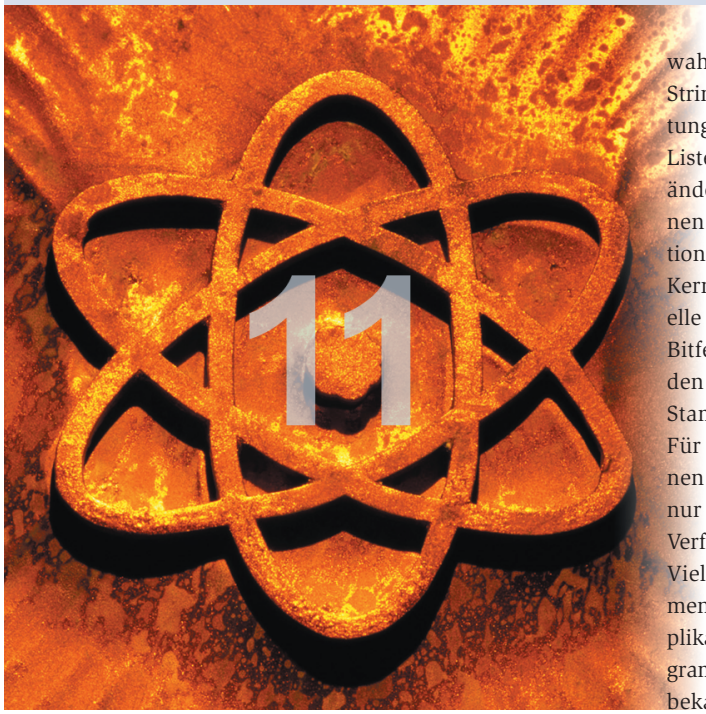
Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 11

Kern-Technik

Fertige Bibliotheken nehmen Programmierern Arbeit ab. Auch der Kernel bietet Hilfsfunktionen, obwohl er keinen Zugriff auf Libraries wie die Glibc hat. Von der Stringumwandlung bis zur Listenverwaltung findet sich allerlei Nützliches, das dieser Artikel erklärt und übersichtlich auflistet. Eva-Katharina Kunst, Jürgen Quade



Wer ein Anwendungsprogramm schreibt, muss nicht jede Funktion neu erfinden. Ein- und Ausgabe, Zeichen umwandeln, Strings verarbeiten – solche Aufgaben werden in Funktionen ausgelagert und in Bibliotheken gesammelt. Im Kernel fällt die Unterscheidung zwischen System- und Bibliotheksfunktion schwer (siehe Abbildung 1 und Kasten „Bibliotheks- und Systemfunktionen im Kernel“): Es gibt dort keine Bibliotheken, alles ist Systemcode. Dennoch lassen sich auch im Kernel Systemdienste von Hilfsfunktionen unterscheiden.

Hilfe ohne Bibliotheken

Obwohl der Kernel eine Reihe nützlicher Hilfsfunktionen mitbringt, gibt es keine Übersicht, die beschreibt, wie sie heißen und was sie tun. Reichhaltig ist die Auswahl zum Beispiel für die Stringformatierung, -verarbeitung und -konvertierung. Um Listen anzulegen und zu verändern, gibt es ebenfalls einen umfangreichen Funktionssatz. Daneben bietet der Kernel Funktionen für spezielle Aufgaben, etwa um große Bitfelder zu verarbeiten oder den Taktzyklenzähler (Time Stamp Counter) auszulesen. Für mathematische Operationen stellt der Kernel dagegen nur eine einzige Funktion zur Verfügung.

Viele Hilfsfunktionen stammen aus dem Bereich der Applikationen und sind den Programmierern meist prinzipiell bekannt. Manche Kernelvarianten unterscheiden sich von ihren Verwandten allerdings in Details: zum Beispiel hinsichtlich ihres Namens, aber auch bei den Parametern und ihrer Bedeutung.

Ausgabe über Syslog

Die Funktion »printf()« beispielsweise kommt im Kernel nur in abgewandelter Form vor und heißt dort »printk()«. Wie »printf()« ist sie für die Ausgabe in Ascii zuständig. Der Unterschied besteht aber nicht nur im Namen. So gibt es bei der Kernelversion keine Standardausgabe, wie sie »printf()« von Haus aus verwendet. Die Funktion »printk()« gibt die Daten mit dem Kernel-Log-Daemon »klogd« an den Syslog-Daemon »syslogd«, der sie in eine Logdatei schreibt.

Damit der Syslog-Daemon die Dringlichkeit der Nachricht einstufen kann, bietet »printk()« die Möglichkeit, so genannte Magics im Formatstring unterzubringen. Ein solches Magic besteht aus einer Ziffer von 1 (Emergence-Meldung) bis 7 (Debug-Meldung), die in spitzen Klammern steht. Es muss unbedingt als Erstes im Formatstring auftauchen. Eine Debugmeldung sieht damit beispielsweise so aus:

```
printk("<7>Irq=%d\n", irq);
```

Natürlich muss sich der Programmierer nicht die Nummern merken, denn die Headerdatei »linux/kernel.h« definiert für jede Dringlichkeitsstufe einen symbolischen Namen. Damit lässt sich die Debug-Ausgabe auch folgendermaßen kodieren:

```
printk(KERN_DEBUG "Irq=%d\n", irq);
```

Mit den Makros »pr_debug()« und »pr_info()« geht es noch einfacher (siehe Tabelle 1). Die Routine »pr_debug()« setzt nicht nur »KERN_DEBUG« vor den Formatstring, sondern schaltet die Debug-Meldungen ganz aus, wenn das Symbol »DEBUG« beim Kompilieren nicht definiert ist. Die obige Debug-Ausgabe lautet mit diesem Makro:

```
pr_debug("Irq=%d\n", irq);
```

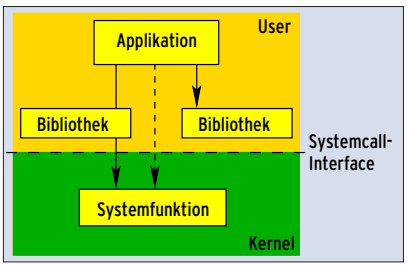


Abbildung 1: Applikationen nutzen Bibliotheksfunktionen und Systemdienste; die Letzteren stellt der Betriebssystemkern zur Verfügung.

Bibliotheks- und Systemfunktionen im Kernel

Aus Sicht einer Linux-Anwendung ist die Abgrenzung einfach: Systemfunktionen oder auch Systemcalls arbeitet der Kernel im Auftrag einer Applikation im privilegierten Modus ab. Routinen, die im Kontext der eigenen Applikation ablaufen (User-Kontext), sind Bibliotheks- oder Hilfsfunktionen.

Mathematische Funktionen, beispielsweise »sin()« für Sinus, zählen also zu den Bibliotheksfunktionen. Eine Ausgabe per »write()« ruft dagegen über die Systembibliothek einen Systemcall auf, der im Kernelmodus abläuft, weil er nur so auf die (sonst geschützte) Hardware zugreifen darf (siehe Abbildung 1). Dieser einfachen Definition zufolge kann es im Betriebssystemkern keine Bibliotheksfunktionen geben – schließlich läuft hier jeder Code im Kernel ab.

Dennoch ist eine Differenzierung hinsichtlich des Funktionsablaufs möglich: Routinen, die auf interne Datenstrukturen zurückgreifen und diese gar verändern, sind Systemfunktionen. Routinen, die zustandslos arbeiten, werden hier Hilfsfunktionen genannt. Im Quellcode des Kernels findet sich mit dem Ordner »lib« sogar ein eigenes Verzeichnis dafür.

Die Funktion »pr_info()« unterscheidet sich von »pr_debug()« in der Dringlichkeit: Sie verwendet die nächstdringliche Stufe 6 für »Info«. Außerdem lässt sich ihre Ausgabe nicht durch den Compiler ausschalten.

Sollen Ausgaben nur im Speicher und nicht direkt für den Syslog-Daemon aufbereitet werden, stehen dem Kernelprogrammierer die von der C-Bibliothek bekannten Funktionen »sprintf()«, »snprintf()«, »vsprintf()« und »vsnprintf()« zur Verfügung (siehe Tabelle 2). Deren Parameterliste und Wirkungsweise entsprechen genau den gleichnamigen Funktionen auf Anwendungsebene. Deshalb helfen bei Problemen die Manpages der Bibliotheksfunktionen weiter. Mitt-

Tabelle 1: Ausgabefunktionen in »linux/kernel.h«

Name und Parameter	Aufgabe
int printk (const char *fmt, ...)	Formatierte Ausgabe über Syslog
int pr_debug (const char *fmt, ...)	Formatierte Debugausgabe
int pr_info (const char *fmt, ...)	Formatierte Ausgabe von normalen Infos

Tabelle 2: Stringformatierung in »linux/kernel.h«

Name und Parameter	Aufgabe
int sprintf (char *buf, const char *fmt, ...)	Formatierte Ausgabe in »buf«
int snprintf (char *buf, size_t size, const char *fmt, ...)	Formatierte Ausgabe in »buf«, maximal »size« Zeichen
int vsprintf (char *buf, const char *fmt, va_list args)	Formatierte Ausgabe in »buf«
int vsnprintf (char *buf, size_t size, const char *fmt, va_list args)	Formatierte Ausgabe in »buf«, maximal »size« Zeichen

Tabelle 3: Stringdekodierung in »linux/kernel.h«

Name und Parameter	Aufgabe
int sscanf (const char *buf, const char *fmt, ...)	Konvertiert »buf« gemäß »fmt«
int vsscanf (const char *buf, const char *fmt, va_list args)	Konvertiert »buf« gemäß »fmt«
unsigned long simple_strtoul (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »unsigned long«
long simple_strtol (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »long«
unsigned long long simple_strtoull (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »unsigned long long«
long long simple_strtoll (const char *cp, char endp, unsigned int base)	Konvertiert »cp« in den Typ »long long«

lerweile dürfte es sich herumgesprochen haben, dass die Varianten mit einem zusätzlichen n wie »snprintf()« und »vsnprintf()« den ursprünglichen Versionen aus Sicherheitsgründen vorzuziehen sind, um das Risiko für Buffer Overflows zu verkleinern. Allerdings verleiten auch diese Funktionen den Programmierer zu Fehlern, beispielsweise dann, wenn er mit mehreren Aufrufen von »snprintf()« in ein und denselben Puffer schreibt.

Strings und Zeichen

Kernel 2.6.4 führt die neuen Funktionen »int scnprintf(char *buf, size_t size, const char *fmt, ...)« und »int vscnprintf(char *buf, size_t size, const char *fmt, va_list args)« ein. Sie unterscheiden sich von »snprintf()« respektive »vsnprintf()« dadurch, dass sie die Anzahl der wirklich geschriebenen Bytes zurückgeben – ohne das abschließende »0«-Zeichen mitzuzählen. Falls die Puffergröße (Parameter »size«) bereits beim Aufruf den Wert 0 besaß, liefern die Funktionen ebenfalls 0.

Die String-Funktionen eignen sich vor allem für die Ein- und Ausgabe Ascii-kodierter Daten über das Proc- und das Sys-Filesystem, siehe [1] und [2]. Ein weiteres Einsatzfeld ist die Dekodierung von Parametern beim Kernelstart oder beim Laden eines Moduls.

Während die vorgestellten Funktionen numerische Werte in Strings umwandeln können, sind diese Routinen für den umgekehrten Weg zuständig: »sscanf()«, »vsscanf()«, »simple_strtoul()«, »simple_strtol()«, »simple_strtoull()« und »simple_strtoll()« (Tabelle 3).

Die Funktion »sscanf(const char* buf, const char* fmt, ...)« dekodiert den String »buf« gemäß der Formatanweisung »fmt« und weist die darin enthaltenen Einzelwerte den übergebenen Variablen zu. Damit unterscheidet sie sich nicht von der gleichnamigen Funktion auf Anwendungsebene. Auch »simple_strtoul()«, »simple_strtoull()«, »simple_strtol()« und »simple_strtoll()« entsprechen den Anwendungsfunktionen ohne den Namensvorsatz »simple_...«.

Im Bereich der Stringverarbeitung finden sich weiterhin die so genannten Ctype-Funktionen »isalnum()«, »isalpha()«, »iscntrl()«, »isdigit()«, »isgraph()«, »islower()«, »isprint()«, »ispunct()«, »isspace()«, »isupper()«, »isxdigit()« und »isascii()« (Tabelle 4). Wie die gleichnamigen Applikationsfunktionen dienen sie dazu, einzelne Zeichen zu klassifizieren. Auch die häufig benötigten Funktionen »tolower()« und »toupper()« brauchen nicht selbst programmiert zu werden. Wer die Zeichen eines Strings einheitlich in Kleinbuchstaben oder in Großbuchstaben haben möchte, kann sich ihrer bedienen. Kernel 2.6 besitzt auch Funktionen, die Strings verglei-

Tabelle 4: Zeichenklassifizierung in »linux/ctype.h«

Name und Parameter	Aufgabe
unsigned char isalnum (unsigned char c)	Prüft, ob »c« ein alphanumerisches Zeichen (Buchstabe oder Ziffer) ist
unsigned char isalpha (unsigned char c)	Prüft, ob »c« ein alphabetisches Zeichen (Buchstabe) ist
unsigned char iscntrl (unsigned char c)	Prüft »c« auf Kontrollzeichen
unsigned char isdigit (unsigned char c)	Prüft, ob »c« zur Klasse der Ziffern gehört
unsigned char isgraph (unsigned char c)	Prüft, ob »c« zur Klasse der druckbaren Zeichen (ohne Space) gehört
unsigned char islower (unsigned char c)	Prüft, ob »c« ein Kleinbuchstabe ist
unsigned char isprint (unsigned char c)	Prüft, ob »c« druckbar ist
unsigned char ispunct (unsigned char c)	Prüft, ob »c« kein Buchstabe und keine Ziffer und dennoch druckbar ist
unsigned char isspace (unsigned char c)	Prüft, ob »c« ein Leerzeichen (Space, Tab, Formfeed) ist
unsigned char isupper (unsigned char c)	Prüft, ob »c« ein Großbuchstabe ist
unsigned char isxdigit (unsigned char c)	Prüft, ob »c« eine Hexadezimalziffer ist
unsigned char isascii (unsigned char c)	Prüft, ob »c« ein Ascii-Zeichen ist

Listing 1: Listenverarbeitung

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03
04 MODULE_LICENSE("GPL");
05
06 static spinlock_t sl;
07
08 struct my_list {
09    int counter;
10    struct list_head link;
11 };
12 struct list_head rootptr;
13
14 static int show_listfunction_usage( void )
15 {
16    int i;
17    struct my_list *akt_element;
18    struct list_head *loopvar, *tmp;
19
20    for(i=0; i<5; i++) { // Liste anlegen
21       // innerhalb der Critical Section darf
             nicht kmalloc()
22       // verwendet werden!
23       akt_element = kmalloc(sizeof(struct
            my_list), GFP_USER);
24       spin_lock(&sl);
25       akt_element->counter = i;
26       printk("adding: %p- %d\n", akt_element,
                i);
27       list_add(&akt_element->link, &rootptr);
28       spin_unlock(&sl);
29    }
30    spin_lock(&sl);
31    list_for_each_safe(loopvar, tmp, &rootptr)
      { // Liste freigeben
32       i = list_entry(loopvar, struct my_list,
             link)->counter;
33       printk("delete: %p - %d\n", loopvar, i);
34       list_del(loopvar);
35       kfree(list_entry(loopvar,struct my_list,
         link));
36    }
37    spin_unlock(&sl);
38    return 0;
39 }
40
41 static int __init modinit(void)
42 {
43    INIT_LIST_HEAD(&rootptr);
44    show_listfunction_usage();
45    return -EIO;
46 }
47
48 static void __exit modexit(void)
49 {
50    return;
51 }
52
53 module_init(modinit);
54 module_exit(modexit);
```

chen. Tabelle 5 listet von »strstr()« über »strlen()« bis »strncat()« alle auf. Mit aufgeführt sind Funktionen, die nicht auf Strings, sondern allgemein auf Speicherbereiche anwendbar sind wie etwa »memcpy()«, »memset()« & Co.

Listen im Kernel

Auch für komplexere Datenstrukturen bietet der Kernel Unterstützung. So gibt es für doppelt verkettete Listen eine Reihe von Hilfsfunktionen. Der Programmierer definiert in der zu verkettenden Datenstruktur ein Element vom Typ »struct list_head«. Fehlt noch der Listenkopf, den er entweder statisch durch »LIST_HEAD_INIT()« oder dynamisch durch Aufruf von »INIT_LIST_HEAD()« initialisiert. Beide Makros sind in der Headerdatei »linux/list.h« definiert. Die Funktion »list_add()« hängt Elemente in die Liste ein, während »list_del()« sie wieder entfernt.

Für den Zugriff auf die einzelnen Listenelemente bringt der Kernel mehrere Makros mit. Allen ist gemeinsam, dass sie eine Schleife darstellen, die über die Listenelemente iteriert. Die einfachste Variante ist »list_for_each()«.

Der Programmierer übergibt diesem Makro eine Variable vom Typ »struct list_head« (im Weiteren »akt_element«) und den Listenkopf. Innerhalb der Schleife zeigt »akt_element« auf das aktuelle Listenelement. Allerdings ist das ein Zeiger auf die Daten zur Listenverwaltung und nicht unbedingt auf die Anfangsadresse des tatsächlichen Datenobjekts – die liefert das Makro »list_entry()« (siehe Abbildung 2 und Listing 1, Zeile 32).

Für die etwas elegantere Iteration direkt über die Objekte stehen Entry-Varianten zur Verfügung, zum Beispiel »list_for_each_entry_safe()«. Abbildung 3 zeigt, wie man das Makro verwendet und was

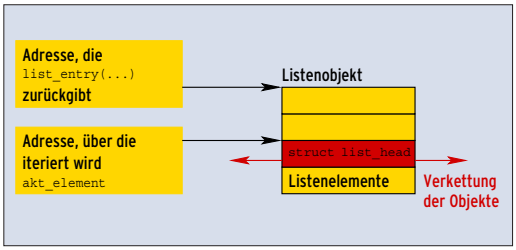


Abbildung 2: Bei einer Iteration über eine Liste zeigt das aktuelle Element »akt_element« häufig auf die Daten zur Listenverwaltung. Das Makro »list_entry()« liefert die Adresse des Anfangs der tatsächlichen Objektdaten.

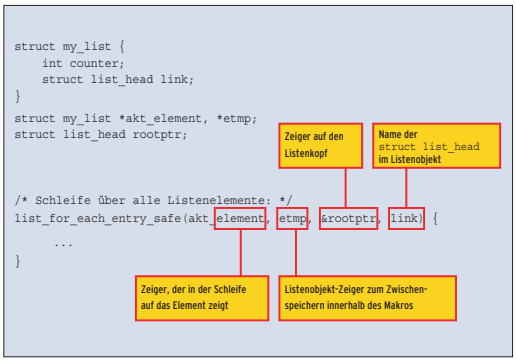


Abbildung 3: Das Makro »list_for_each_entry_safe()« realisiert eine Schleife direkt über die Listenobjekte, statt wie in Abbildung 2 nur über die Verwaltungszeiger.

Tabelle 5: String- und Speicherfunktionen in »linux/string.h«

Name und Parameter	Aufgabe
char *strstr (const char *string, const char *substring)	Sucht »substring« in »string«
int strnicmp (const char *s1, const char *s2, size_t len)	Längen-limitierter Stringvergleich (case insensitive)
char *strncpy (char *dest, const char *src, size_t len)	Kopiert »src« nach »dest«, maximal »len« Zeichen
char *strcpy (char *dest, const char *src)	Kopiert »src« nach »dest«
size_t strlcpy (char *dest, const char *src, size_t size)	Kopiert »src« nach »dest«, maximal »size« Zeichen, immer mit »0« terminiert
char *strncat (char *dest, const char *src, size_t count)	Hängt »src« an »dest« an, kopiert maximal »count« Zeichen, der Ergebnisstring ist Null-terminiert
size_t strlcat (char *dest, const char *src, size_t count)	Wie »strncat()«, gibt aber die neue Stringlänge zurück
size_t strspn (const char *s, const char *accept)	Durchsucht »s« nach den Zeichen aus »accept«
char *strsep (char s, const char *ct)	Zerlegt »s« in Token
char *strpbrk (const char *cs, const char *ct)	Findet das erste Auftreten von »ct« innerhalb von »cs«
size_t strlen (const char *s);	Gibt die Länge von »s« (ohne abschließende Null) zurück
size_t strnlen (const char *s, size_t count)	Berechnet die Länge des mit »count« Zeichen limitierten Strings »s«
int strcmp (const char *cs, const char *ct)	Stringvergleich
int strncmp (const char *cs, const char *ct, size_t count)	Stringvergleich mit maximal »count« Zeichen
char *strchr (const char *s, int c)	Findet das erste Auftreten des Zeichens »c« innerhalb von »s«
char *strrchr (const char *s, int c)	Findet das letzte Auftreten des Zeichens »c« innerhalb von »s«
void *memset (void *s, int c, size_t count)	Füllt den Speicher ab Adresse »s« mit den Zeichen »c«
void bcopy (const char *src, char *dest, int count);	Kopiert »src« nach »dest«
void *memcpy (void *dest, const void *src, size_t count)	Kopiert »src« nach »dest«
void *memmove (void *dest, const void *src, size_t count)	Wie »memcpy()«, allerdings mit überlappenden Speicherbereichen
int memcmp (const void *cs, const void *ct, size_t count)	Vergleicht die Speicherbereiche »cs« und »ct«
void *memscan (void *addr, int c, size_t size)	Sucht das Zeichen »c« innerhalb von »addr« (gibt die Adresse zurück, an der »c« erstmals auftaucht, sonst ein Byte nach dem Bereich)
void *memchr (const void *s, int c, size_t n)	Wie »memscan()«, aber mit anderem Rückgabewert, falls c nicht gefunden wird: »NULL«

die einzelnen Parameter bedeuten. Die Safe-Varianten stellen zudem sicher, dass der aktuelle Listeneintrag nicht aus der Liste entfernt wird, während der Kernel die Schleife durchläuft.

Es gibt noch eine ganze Reihe weiterer Funktionen, um Listen zu bearbeiten. Darunter solche, die zum Beispiel Elemente am Ende der Liste einhängen, zwei Listen zusammenfügen oder eine Liste teilen. Die Init-Funktionsvarianten sorgen zusätzlich noch dafür, dass der Listenkopf richtig, also als leere Liste, initialisiert wird.

Listing 1 demonstriert die Anwendung der Listenfunktionen. Die Funktion »show_listfunction_usage()« durchläuft eine Schleife und hängt mit »list_add()« fünf Elemente in die Liste ein (Zeilen 20 bis 28). Ist die Liste erzeugt, durchläuft das Modul sie mit dem beschriebenen Makro »list_for_each_safe()« (Zeile 31) und löscht die einzelnen Elemente mit »list_del()«. An die Daten kommt das Modul über das Makro »list_entry()« (siehe Abbildung 2).

Diesen Ablauf protokolliert der Syslog-Daemon, dem das Modul jeden Schritt über »printk()« mitteilt (Zeilen 26 und 33). Das dauerhafte Laden des Moduls wird verhindert, indem die Funktion »modinit()« den Fehlercode »-EIO« zurückgibt (Zeile 44). Das vereinfacht den Test, denn das Modul muss so nicht von Hand entladen werden.

Ohne Mathematik

Die Tabellen 1 bis 7 listen die wichtigsten Hilfsfunktionen respektive Makros im Kernel inklusive ihrer Parameter. Darüber hinaus gibt es beispielsweise die »min()«- und »max()«-Makros, die eine umfangreiche Typprüfung der übergebenen Argumente durchführen, oder die bereits erwähnten Funktionen für Bitfelder mit mehr als 32 Bit. Wer allerdings mathematische Funktionen sucht, geht leer aus. Da im Kernel Floating-Point-Operationen verboten sind, fehlen mathematische Funktionen mit Ausnahme der Integer-Wurzel »int_sqrt()«.

Erst kürzlich hat Kernelhacker Andrew Morton auf der Kernel-Mailingliste alle Programmierer dazu aufgefordert, die vorhandenen Hilfsfunktionen zu nutzen. Denn so müssen sie nicht nur selbst weniger Code schreiben, sondern vermeiden auch Fehler – schließlich sind die Hilfsfunktionen im Kernel bereits gründlich getestet.

Vorschau

Ein ladbares Modul produzieren ist die eine Sache, dabei guten Kernelcode zu schreiben eine andere. Selbst wer auf Applikationsseite schon viel programmiert hat, stolpert im Kernel leicht über versteckte Hindernisse. Wer sie kennt, muss nicht alle möglichen Fehler selber machen, die im Kernel oft zum Totalabsturz führen. Deshalb zeigt die Kern-Technik in der nächsten Folge, worauf der Kernelprogrammierer besonders achten sollte. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 6, Sys-Filesystem: Linux-Magazin 01/04, S. 94

[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 7, Proc-Filesystem: Linux-Magazin 02/04, S. 48

[3] Listings zu diesem Artikel: [http://www.linux-magazin.de/Service/Listings/2004/06/Kern-Technik]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.

Tabelle 6: Zeichenkonvertierung in »linux/ctype.h«

Name und Parameter	Aufgabe
unsigned char tolower (unsigned char c)	Gibt »c« als Kleinbuchstaben zurück
unsigned char toupper (unsigned char c)	Gibt »c« als Großbuchstaben zurück
unsigned char toascii (unsigned char c)	Konvertiert »c« in ein Ascii-Zeichen

Tabelle 7: Listenverarbeitung in »linux/list.h«

Name und Parameter	Aufgabe
void list_add (struct list_head *new, struct list_head *head)	Hängt das Listenelement »new« hinter »head« ein
void list_add_tail (struct list_head *new, struct list_head *head)	Hängt das Listenelement »new« vor »head« ein
void list_del (struct list_head *entry)	Löscht »entry« aus der Liste
void list_del_init (struct list_head *entry)	Löscht »entry« aus der Liste und initialisiert es wieder
void list_move (struct list_head *list, struct list_head *head)	Löscht das Element »list« und hängt es hinter »head« ein
void list_move_tail (struct list_head *list, struct list_head *head)	Löscht das Element »list« und hängt es vor »head« ein
int list_empty (const struct list_head *head)	Gibt »true« zurück, falls kein Element in der Liste »head« eingehängt ist
void list_splice (struct list_head *list, struct list_head *head)	Hängt alle Elemente von »list« in »head« ein
void list_splice_init (struct list_head *list, struct list_head *head)	Hängt alle Elemente von »list« in »head« ein und initialisiert »list«
list_entry (ptr, type, member)	Gibt das zum Listenelement »ptr« gehörige Objekt zurück
list_for_each (pos, head)	Iteriert über die Liste »head«
list_for_each_prev (pos, head)	Iteriert rückwärts über die Liste »head«
list_for_each_safe (pos, n, head)	Iteriert über die Liste »head«, wobei das Element vor dem Löschen geschützt ist
list_for_each_entry (pos, head, member)	Iteriert über die Objekte der Liste »head«
list_for_each_entry_reverse (pos, head, member)	Iteriert rückwärts über die Objekte der Liste »head«
list_for_each_entry_continue (pos, head, member)	Iteriert über die Objekte von »head« ab Listenelement »member«
list_for_each_entry_safe (pos, n, head, member)	Iteriert über »head« und schützt vor dem Löschen des Listeneintrags