Der Linux Kernel

Was ist der Kernel?

Der Kernel eines Betriebsystems ist die **Abstraktionsschicht** für die Hardware.

Anwender sollen nicht direkt auf die Hardware zugreifen, sondern machen dies mit Hilfe des Kernels.

Beispiele:

- Lesen und Schreiben von Daten (I/O)
- Netzwerkzugriffe
- Memory-Management

Verschiedene Modi

Damit Anwender auf die Hardware zugreifen können, stellt der Kernel Software zur Verfügung. Mit Hilfe dieser so genannten Gerätetreiber und einer API kann das Gerät genutzt werden. Zu den Treibern gehören Einträge (Gerätedateien) im Verzeichnis `/dev`

Moderne Prozessoren können in zwei verschiedenen Modi betrieben werden:

- **Kernel-Mode**
- **User-Mode**

Verschiedene Modi

Im **Kernel Mode** kann direkt und unbeschränkt auf die Hardware zugegriffen werden. Im User-Mode ist dies nicht möglich.

Prozesse, die Benutzern (auch `root`) gehören, werden im User-Mode betrieben. Zugriffe auf die Hardware werden durch eine **Schnittstelle** zum Kernel durchgeführt. Der Anwender arbeitet dann auf einer höheren **Abstraktionsebne**.

Schnittstellen

Schnittstellen zum Kernel sind etwa

- System Calls,
- Signale oder
- Geräte (devices)
- Virtuelle Dateisysteme wie /sys oder /proc

Kein System Call

Insbesondere System Calls werden von praktisch jedem Programm genutzt. Das folgende Programm nutzt `printf`. Achtung: `printf` ist kein System Call!

```
int main()
{
    printf("Hello World\n");
    return 0;
}
```

Da es aber eine Ausgabe - also ein Zugriff auf die Hardware - stattfindet, muss es irgendwo System Calls geben.

Der System Call `write`

Tatsächlich nutzt die Funktion `printf` den System Call `write`. Hello World kann daher auch wie folgt formuliert werden:

```
#include <unistd.h>
int main()
{
  const char msg[] = "Hello, World!\n";
  write(STDOUT_FILENO, msg, sizeof(msg)-1);
  return 0;
}
```

`STDOUT_FILENO` ist dabei eine Konstante, die in `unistd.h` definiert ist und ein Handle für das Gerät `/dev/stdout` ist.

Der System Call `read`

Das Verzeichnis `/dev` enthält viele Gerätedateien. Dazu gehört etwa auch `/dev/random`, ein virtuelles Gerät, das genutzt werden kann, um Zufallsdaten zu generieren:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(){
  int dev_random_fd  =  open("/dev/random", O_RDONLY);
  char random_byte[1];
  read(dev_random_fd, random_byte, 1);
  int random_value=*random_byte;
  printf("Random Value: %i\n",random_value);
  return 0;
}
```

Kernel Module

Bevor wir uns mit dem Bau eines eigenen Kernels beschäftigen, sehen wir wie man einen bestehenden Kernel zur Laufzeit *erweitern* kann ohne den Kernel zu bauen.

Mit

```
lsmod
```

oder

```
cat /proc/modules
```

kann man sich diese so genannten Kernel-Module auflisten lassen.

Ein einfaches Modul

Es ist grundsätzlich nicht schwer selbst ein Modul zu entwickeln. Ein einfacher Fall kann wie folgt aussehen:

```
#include <linux/module.h>
#include <linux/init.h>

static int  hello_init(void) {
  printk("module hello: Hello world!\n");
  return 0;
}
static void  hello_exit(void) {
  printk("module hello: Goodbye world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Ein einfaches Modul

Über die Funktionen **`module_init`** und **`module_exit`** werden die Funktionen eingestellt, die beim initialisieren und beenden des Moduls aufgerufen werden. Mit **`printk`** werden Kernel-Messages geschrieben.

Das zugehörige Makefile sieht wie folgt aus:

```
obj-m := hello.o
```

Das Makefile ist alleine nicht ‚überlebensfähig', sondern wird während des Build-Prozesses in ein anderes Makefile eingebunden.

Der Build-Prozess

Der Build-Prozess wird mit dem Target **modules** wie folgt gestartet:

```
make -C <kernel_source>  M=$PWD modules
```

Mit der Option **-C** wird in eine anderes Verzeichnis gewechselt und das dortige Makefile ausgeführt. Der Option wird der Teil der Kernel-Sourcen übergeben, der für die Kernel-Module zuständig ist. In Ubuntu-Systemen findet man diesen in einem der **build**-Verzeichnisse unter **/lib/modules**.

Das eigentliche Makefile wird mit Hilfe der Variablen **M** eingebunden.

Nach dem Build-Prozess

Das übersetzte Modul wird wie folgt zum Kernel geladen:

```
insmod hello.ko
```

Die mit `printk` erzeugte Nachricht sieht man mit

```
dmesg
```

Entfernt wird das Modul mit:

```
rmmod hello.ko
```

Module und Devices

Das Beispiel-Modul ist wenig nützlich. Typischerweise werden Module mit Signalen oder insbesondere Geräten verbunden.

Unter Unix unterscheidet man Character- und Block-Devices. Character-Devices lesen und schreiben Daten zeichenweise. Beispiele sind Keyboards, Mäuse oder Terminals.

Im folgenden wird die Entwicklung eines einfachen Character-Devices *skizziert*.

Der Weg in das Gerät

Was passiert bei

```
echo -n xyz > /dev/adevice
```

Der Kernel ermittelt die zu `/dev/adevice` gehörende id (‚major‘), die wir mit `ls /dev/adevice` sehen.

```
crw-rw-rw- 1 root root 42, 0 Apr  7 08:10 /dev/adevice
```

Unter der major-ID hat der Kernel ein Modul registriert. Er führt dessen `write`-Funktion aus und liest `xyz` ein.

Analog wird etwa bei `cat /dev/adevice` die `read`-Funktion des Moduls ausgeführt.

Datei-Operationen

Die `read` und `write` Funktionen sind Teil der `file_operations` des Kernel-Moduls:

```
struct file_operations simple_fops = {
   .read = simple_read,
   .write = simple_write,
   .open = simple_open,
   .release = simple_release
};
```

Die grundlegende Funktionalität der Operationen ergibt sich aus dem Namen.

Die Initialisierung

Die `file_operations` des Moduls werden typischerweise bei der Initialisierung gesetzt:

```
int simple_init(void) {
  register_chrdev(42,"simple",&simple_fops);
  ..
```

Mit dem Aufruf von `register_chrdev` wird auch die major-ID (hier 42) gesetzt. Zusätzlich wird dem Modul noch ein Label (hier simple) gegeben.

Verbinden mit einem Geräteknoten

Nachdem das Modul zum Kernel hinzugefügt wurde, kann man ihm einen Geräteknoten zuordnen:

```
mknod /dev/adevice c 42 0
```

Das `c` zeigt an, dass ein Charakter-Device erzeugt wird.

Das Gerät kann anschließend für Ein-und Ausgaben genutzt werden.