Geheimzinnige eenvoud van een kerneldriver

Voor de meeste Linux gebruikers, is de kernel een groot geheim dat als een noodzakelijk kwaad wordt gezien. In het verleden was het echter noodzakelijk om zelf in de kernel te knutselen om bijvoorbeeld sommige soundkaarten en tapedrivers aan de praat te krijgen. Tegenwoordig is dit echter niet altijd meer nodig.

Auteur: Pascal Schiks

Een van de taken van de kernel is het besturen van de hardware in onze computer. Om de kernel minder groot en vooral veel flexibeler te maken is men overgestapt op het modulariseren van de kernel. Vooral zaken die niet tijdens het opstarten nodig zijn, kunnen in een module gecompileerd worden en later of tijdens het bootproces geladen worden. Op deze manier is het mogelijk drivers te starten, stoppen en wijzingen terwijl het systeem gewoon blijft draaien, of is het mogelijk gedeelten van de kernel alleen maar te laden wanneer daar behoefte aan is.
In dit artikel probeer ik een idee te geven wat zo'n driver nu eigenlijk doet. Het doel is het aansturen van een aantal LED's die via de printerpoort zijn aangesloten.

Nodes

Alle devices kunnen worden beschouwd als speciale files. Deze files staan in de directory `/dev`. Wanneer je in deze directory kijkt, zul je zien dat deze files geen lengte hebben, maar wel een zogenaamd Major en Minor nummer. Deze terminologie stamt nog uit de tijd dat bij grote computersystemen meerdere interfaces op een kaart waren ondergebracht. Zo hebben we op mijn werk op een bepaalde machine een interfacekaart met 8 RS232 poorten erop (8 LAS kaart). Zo'n kaart heeft dan een Major nummer en elke RS232 aansluiting heeft een ander Minor nummer. Voor elk ervan wordt dus een nieuwe node aangemaakt.
Wanneer je `ls -l /dev/lp*` intikt dan krijg je alle nodes te zien die naar de LPT driver verwijzen. Zij hebben allemaal Major nummer 6 en een verschillend Minor nummer. Ook kun je voor de permissiebits zien dat het om een character device gaat. Voor onze driver heb ik het major nummer 63 gekozen (die is kennelijk nog vrij). Door het commando `mknod -m 766 /dev/lptdrv u 63 0` met root rechten uit te voeren wordt de node aangemaakt.

Modules

Kernel drivers kunnen direct in de kernel worden meegecompileerd of als module worden geladen. Het heeft echter alleen zin om een driver direct in de kernel te laden wanneer deze tijdens boottime al nodig is, zoals bijvoorbeeld de driver voor de harde schijf en console. Je soundkaart of netwerk interface heb je pas later nodig (ehhh... tenzij je via een netwerk boot of tijdens het booten MP3 bestandjes wil afspelen). In ons geval is het dus eigenlijk alleen interessant om de driver als module te laten laden. Een driver vast meelinken in de kernel is overigens vrij eenvoudig te realiseren
Een kernel module is een gewoon object file die minimaal aan de volgende voorwaarden moet voldoen. Vooraan in de source code dienen de volgende declaraties te staan

```
#define __KERNEL__
#define MODULE

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

Verder moeten er minimaal de volgende twee functies bestaan

```
int init_module(void);
void cleanup_module(void);
```

De twee defines informeren de compiler dat het gaat om een kernel module De twee includes zijn nodig om een aantal structures te definiëren die binnen de module nodig zijn.
De functie init_module wordt aangeroepen bij het laden van de module en de ten slotte wordt de functie cleanup_module aangeroepen wanneer de module uit het geheugen wordt verwijderd.
De allereenvoudigste module zal er dus als volgt uit kunnen zien.

```
#define __KERNEL__
#define MODULE
#include <linux/kernel.h>
#include <linux/module.h>
```

```
int init_module(void)
{
  printk("Mijn module wordt geladen\n");
}

void cleanup_modules(void)
{
  printk("Mijn module wordt verwijderd\n");
}
```

De functie printk is gedefinieerd in `linux/kernel.h` en werkt vrijwel hetzelfde als de c functie printf.

Compileren van een module

Om onze module te compileren, gebruiken we de GNU C compiler Het commando om de module te compileren wordt dan

```
gcc mijnmodule.c -O2 -c -o mijnmodule.o
```

De parameter -O2 is een optimalisatie die nodig is om eventuele macro's correct te behandelen (omdat er later niet meer echt gelinkt wordt, moet dat nu gebeuren) De parameter `-c` vertelt de compiler dat de output objectcode moet worden, die normaliter later nog gelinkt kan worden. Maar omdat het een module betreft gebeurt dat niet. Ten slotte wordt met de instructie `-o mijn module.o` de naam van het resultaat bepaald.
Wellicht is het handig om de compileer-opdracht in een Makefile te zetten zodat enkel de opdracht `make` voldoende is om het programma te compileren.

Starten en stoppen

Het laden en verwijderen van een kernel module is enkel aan root voorbehouden. Het gaat tenslotte om een handeling die het systeem behoorlijk kan aantasten. Om de module te laden moet dus eerst ingelogd worden als root of deze rechten door middel van het su commando worden verkregen.
Daarna kan de module gestart worden met het volgende commando:

```
insmod mijnmodule.o
```

De welkomstboodschap wordt nu getoond, en het commando `lsmod` laat zien dat de module geladen is. Ook is het berichtje terug te vinden in het bestand `/var/syslog`.
Met het commando `rmmod mijnmodule` wordt de module weer uit het geheugen verwijderd.

File operations

Nu wordt het tijd dat we de module enige functionaliteit geven. Een driver kan worden behandeld als een file en er kunnen dus ook file operaties op los worden gelaten. De operaties die we nu zullen implementeren zijn open, close en write. Voor elk van deze operaties moeten we een functie maken en deze vervolgens bij het besturingssysteem (Linux) registreren. Om dit voor elkaar te krijgen is er in `linux/fs.h` de structure file_operations gedefinieerd. In deze structure kunnen we voor alle gewenste operaties een functienaam invullen. Voor de overige functies vullen we gewoon NULL in.
Onze declaratie voor de file operations ziet er dan als volgt uit:

```
static struct file_operations lptdrv_
operations =
{
  NULL,             /* lseek */
  NULL,             /* read  */
  lptread_write,    /* write */
  NULL,             /* read dir */
  NULL,             /* poll */
  NULL,             /* IOctl */
  NULL,             /* mmap */
  lptread_open,     /* open */
  NULL,             /* flush */
  lptread_close,    /* release */
  NULL,             /* fsync */
  NULL,             /* fasync */
  NULL,             /* check media change */
  NULL,             /* revalidate */
  NULL,             /* lock */
};
```

In de init_module functie kunnen we nu deze functies registreren zodat ze bij de kernel bekend zijn. Dit gaat op de volgende manier:

```
register_chrdev(MAJOR_NR,NODENAME,&lptdrv_
operations);
```

Let op: *met verschillende kernelversies veranderen de structuren en functie declaraties zoals gedefinieerd in de include files nogal eens. Dit heeft o.a. tot gevolg dat de voorbeeldjes uit het verder overigens zeer interessante boek van Alessandro Rubini niet op de latere kernels te compileren zijn. Mijn programma heb ik getest op kernel 2.2.x series.*

Output naar de printerpoort

De geregistreerde functies moeten nog wel eerst gemaakt worden. Dit is dus waar het eigenlijk om gaat. We beginnen met de meest interessante, n.l. de functie lptdrv_write
De declaratie is als volgt:

```
ssize_t lptread_write(struct file *filehand↵
le, const char *data_buffer,size_t ↵
buff_size,loff_t *buff_ptr)
```

Als eerste parameter wordt de filehandle meegegeven. Dit is dezelfde filehandle als die van het programma dat het device probeert aan te spreken. Het is tenslotte zo dat onze driver de andere kant van het operatingsystem vormt en dus uiteindelijk datgene moet doen wat door de user functies open, write en close gevraagd wordt. De tweede parameter is het adres in userspace waar de te versturen informatie staat. Hier stuiten we ineens op een vervelend probleem: zoals een gewone gebruiker niet aan kernel memory mag en kan komen, zo kan de kernel niet aan userspace komen. Om toch aan de aldaar opgeslagen informatie te komen is er een workaround in de vorm van de macro __get_user gemaakt. Met deze functie is het mogelijk om data vanuit userspace naar kernelspace te halen. De instructie:

```
__get_user(data,data_buffer);
```

kopieert een byte vanuit userspace naar data zodat we de verkregen informatie vervolgens kunnen gebruiken om deze op de volgende manier naar de LPT poort te sturen:

```
outb(data,LPTADDR);
```

Bij het verlaten van de functie moet het aantal succesvol verstuurde bytes teruggemeld worden met behulp van de return instructie.
De functies `lptdrv_open` en `lptdrv_close` moeten het device (de LED's) initialiseren. Nu ja... daar valt weinig anders aan te initialiseren dan het netjes uitzetten van de LED's. Dus deze functies zijn vrijwel identiek en doen eigenlijk alleen maar een paar `outb` instructies die welhaast voor zich spreken.

```
ssize_t lptread_open(struct inode *inodeptr,↵
 struct file *filehandle)
{
  outb(0x0C,LPTCTL); /* Zet de groene LED ↵
aan */
  return 0;
}

ssize_t lptdrv_close(struct inode *node, ↵
struct file *filehandle)
{
  outb(0,LPT);        /* zet alle rode LED's ↵
 uit */
  outb(0x0D,LPTCTL); /* zet de groene LED ↵
uit  */
  return 0;
}
```

Hiermee is onze driver eigenlijk compleet. De complete code hoef je niet in te tikken maar kun je gewoon downloaden vanaf: **WWW.LINUXMAG.NL** of **WWW.NEDLINUX.NL**
Bij de tar.gz file zit ook een eenvoudig programmaatje om de driver te testen door de lampjes om beurten te laten branden. Ook kun je de driver snel even testen door er een bestandje naar toe te kopiëren met `cp mijnbestand /dev/lptdrv` maar omdat na het sluiten van de `file` de LED's weer uitgaan, zul je daar niet veel van zien. Natuurlijk is deze beschrijving heel erg basic en gaat niet al te diep op problemen in. Zo is mijn driver nog niet in staat om meerdere lpt kaarten van hetzelfde type te herkennen en te benutten. En ook vertel ik het operating system niet correct hoeveel data ik nu eigenlijk verstuurd heb. Maar het geeft wellicht toch een inzicht hoe zo'n driver werkt.
Wanneer je er meer over wilt weten kan ik het boek 'Linux device drivers' van Allessandro Rubini (O'Reily uitgeverij) aanraden.
Dit boek is weliswaar voornamelijk op de 2.0 kernels geënt, maar wanneer je daar rekening mee houdt, maakt dat verder weinig uit.

Testen

Om de LEDdriver te testen heb ik het programmaatje drvtest.c geschreven. Het opent de special file /dev/lptdrv die de LEDdriver verbindt en laat de LEDs om beurten oplichten. Vervolgens wilde ik er ook een praktische toepassing voor bedenken. Dus heb ik mdmlts.c bedacht. Dit programma laat de LEDs werken als modemledjes. Voor het compileren moet je even het poortnummer (line 13) aanpassen met het poortnummer van je modem. Het programma betrekt zijn informatie uit de /proc directory. Wanneer het modem twee minuten lang geen activiteit vertoont, dan gaat het programma in demo mode. Gebruik makend van de tabel op regel 29 worden er dan diverse patroontjes op de LEDs getoverd. Zelf heb ik dit programma op mijn Slackware systeem in het bestand /etc/rc.d/rc.M gezet door er de volgende regel in toe te voegen /usr/bin/mdmlts &. De & erachter zorgt ervoor dat het als achtergrond taak wordt opgestart.

De source code van het hele spul kun je via internet downloaden op www.linuxmag.nl en op www.nedlinux.nl. Het is in een gezipt tar bestandje gezet na het downloaden tik je in tar -zxvf lptdrv.tgz
Er wordt dan een directory met de bestandjes aangemaakt een make opdracht in deze directory is voldoende om het spul te compileren.