[[si:lab:lab5r]]

Module Kernel

Table of Contents

- Module Kernel
  - Drivere în sisteme embedded fără MMU
  - Drivere în sistem embedded cu MMU
  - Programarea modulelor de kernel
- Modul Hello World
- Makefile modul Hello World
- Utilitare pentru lucrul cu module
  - Exercițiu
- Kernel GPIO
  - Exercițiu
- Referințe

Un sistem embedded poate funcționa doar cu perifericele pe care le-am folosit deja (rețea, card SD, USB), va fi însă strict limitat la hardware pentru care exista deja suport. Ce se întâmplă însă atunci când dorim să folosim un hardware nou sau diferit de cel pentru care există suport?

Drivere în sisteme embedded fără MMU

În sistemele care nu au MMU scrierea unui driver (o bucată de cod care interfațează un anumit dispozitiv) este directă: Fie că driver-ul este foarte strâns cuplat cu întreg sistemul, fie că este sub forma unei biblioteci de funcții, totul se află în același spațiu de memorie cu programul care rulează (sau sistemul de operare, în cazurile mai complexe). Dezavantajele principale al acestui tip de sistem sunt lipsa de securitate și de stabilitate: Orice vulnerabilitate sau bug pot aduce întreg sistemul pe genunchi.

Mai mult, două programe (de exemplu într-un sistem de operare cu multi-tasking cooperativ) pot concura pentru aceeași resursă, chiar dacă nu rulează concomitent. Dacă luăm exemplul aplicațiilor de la PM, dacă o funcție configura seriala cu un anumit baud rate, apoi ceda controlul altei funcții care seta baud rate la altă valoarea, clar valoarea finală va fi a doua, codul asociat cu prima funcție nu va mai rula corect.

Deși pare greu de ajuns la o asemenea situație, în realitate este foarte ușor: unul din modurile de a implementa sisteme multitasking colaborative este cu corutine, funcții cu mai multe puncte de intrare care cedează controlul. În astfel de sisteme, o corutină joacă rolul unui proces. Astfel, având mai mult de un dezvoltator putem fi siguri că va apărea o situație ca în scenariul menționat mai devreme sau mai târziu.

Drivere în sistem embedded cu MMU

Sistemele cu MMU rezolvă această problemă având accesul la hardware doar dintr-un mod special, privilegiat. Astfel, codul care interfațează hardware-ul va fi pus într-o zonă specială de memorie, accesibilă decât din modul privilegiat. Un layer adițional va face comunicația între acest cod și codul neprivilegiat care îl folosește. Astfel, driver-ul va fi protejat de vulnerabilitățile programelor care îl utilizează, nu va avea probleme în urma crash-urilor lor și va putea face o arbitrare a accesului la resursa hardware pe care o gestionează.

Arhitectura kernelului Linux nu este una pur monolitică, ea permițând inserarea dinamică a unor bucăți de cod obiect numite module. Majoritatea driver-elor se găsesc sub formă de module încărcabile dinamic.

Programarea modulelor de kernel

Modulele sunt încărcate în memoria accesibilă doar în modul privilegiat, denumită și kernel-space. De aceea, programarea modulelor de kernel este guvernată de anumite reguli:

- Kernelul nu este link-at cu biblioteca standard C, nu se pot folosi niciunele dintre apelurile de bibliotecă cunoscute!
- Se pot folosi însă funcțiile, macrourile și variabilele exportate de kernel. Acestea se găsesc în headerele kernelului.
- Nu se accesează direct zona de memorie care poate fi accesată și din modul neprivilegiat (a.k.a. userspace). Tot ce provine din userspace trebuie privit cu suspiciune, există macro-uri speciale pentru transferul dintre cele două zone de memorie.
- Accesele invalide la memorie trebuiesc evitate deoarece sunt mult mai grave în kernel-mode, pe Windows se generează BSOD (Blue Screen of Death), pe Linux se dă mesajul kernel panic și se oprește sistemul.

Modul Hello World

Orice modul de kernel are nevoie de o funcție de inițializare și o funcție de cleanup. Prima se apelează atunci când modulul este încărcat, a doua este apelată când modulul este descărcat. De asemenea, modulele au nevoie de autor, licență și descriere, utilizarea acestor macro-uri este obligatorie.

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

MODULE_DESCRIPTION("Simple module");
MODULE_AUTHOR("SI");
MODULE_LICENSE("GPL");

static int __init init(void)
{
 printk(KERN_ALERT "Hello!\n");

 return 0;
}


static void exit(void)
{
 printk(KERN_ALERT "Goodbye!\n");
}


module_init(init);
module_exit(exit);
```

Modulul Hello World conține:

1. Headerele necesare
2. Definirea autorului, licenței și descrierii
3. Funcția de inițializare a modulului
   - Specificatorul __init este un shortcut pentru __attribute__ ((__section__(".init.text"))) care specifică gcc-ului în ce segment al executabilului să pună această funcție. În cazul modulelor care se încarcă odată cu kernelul funcția init se va apela o singură dată, după care va rămâne în memoria sistemului (kernel-space) până la închiderea sistemului. .init.text este un segment care este eliberat după inițializarea kernel-ului (se poate observa în timp ce pornește sistemul mesajul Freeing unused kernel memory: 108k freed)
   - Funcția printk este echivalentul în kernel al funcției printf. Ieșirea este însă direcționată către un fișier log, /var/log/messages. Diferența în folosire este dată și de specificarea priorității cu macro-ul KERN_ALERT, care de fapt se traduce în șirul de caractere <1>. Sistemul va putea fi configurat astfel să ignore anumite mesaje.
4. Funcția de cleanup a modulului
5. Înregistrarea celor două funcții ca init și exit pentru modulul respectiv.

Makefile modul Hello World

```
ARCH := avr32
CROSS_COMPILE := /home/student/buildroot/build_avr32/staging_dir/usr/bin/avr32-linux-
KDIR := /home/student/buildroot/project_build_avr32/atngw100/linux-2.6.27.6
PWD := $(shell pwd)

obj-m := hello_mod.o

kbuild:
 make -C $(KDIR) M=`pwd` ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) modules

clean:
 make -C $(KDIR) M=`pwd` clean
 -rm -f *~ Module.symvers Module.markers modules.order
```

- ARCH este arhitectura pentru care compilăm. Nu uitați că folosim un toolchain pentru cross-compiling, vom compila cod obiect pentru AVR32
- CROSS_COMPILE definește calea către toolchain. Calea se termină în avr32-linux- pentru că doar aceasta variază (e.g. sparc-linux-gcc, arm-linux-gcc), este folosită pentru a genera numele tuturor utilitarelor avr32-linux-gcc, avr32-linux-ldd, avr32-linux-ld.
- KDIR este directorul surselor nucleului, necesare atât pentru header-e cât și pentru Makefile-uri (observați că target-ul kbuild invocă alt makefile din directorul nucleului).
- obj-m este o variabilă ce reține fișierele obiect ce trebuiesc linkate într-un singur modul.

Utilitare pentru lucrul cu module

După cum s-a studiat în laboratoarele anterioare, insmod inserează module în kernel:

```
~#insmod hello_mod.ko
```

Modprobe face același lucru, dar cu module puse deja în sistemul de fișiere în locul corespunzător (/lib/modules/`uname -r`/... - uname -r este versiunea nucleului)

```
~#modprobe g_ether
```

Pentru descărcare, se folosește fie:

```
~#modprobe -r g_ether
```

fie:

```
~#rmmod g_ether
```

Afișarea modulelor încărcate se face cu lsmod

```
~#lsmod
```

Afișarea mesajelor date cu printk din modul se găsesc în /var/log/messages, se afișează cu:

```
~#dmesg | tail
```

sau cu

```
~#tail -f /var/log/messages
```

(nu a fost încercat pe ngw, urmărește în timp real fișierul de log)

Exercițiu

- Scrieți un modul hello_mod.c, compilați-l și rulați-l pe NGW.

Kernel GPIO

Până acum ați interacționat cu LED-urile prin interfața oferită de un driver în /sys/class/leds. Acum vom avea ocazia să vedem ce se află în spatele acelei interfețe.

AP7000 (procesorul de pe plăcuța de dezvoltare) are niște registre pentru "General Purpose I/O". Pentru fiecare dintre cele 5 porturi (A,B,C,D,E) avem un set de registre, dintre care enunțăm:

1. PER - pin enable register
2. OER - output enable register
3. CODR - clear output data register
4. SODR - set output data register

Registrele sunt pe 32 de biți, fiecare bit configurând un anumit pin, de la cel mai semnificativ (asociat pinului 0), la cel mai nesemnificativ (asociat pinului 31 de pe portul respectiv). Registrele suportă scriere cu măști, astfel încât doar pinii biților 1 vor fi afectați.

De exemplu, pentru a aprinde LED-ul de pe pinul 12 al portului E vom scrie:

```
per = 0x00080000
oer = 0x00080000
codr= 0x00080000
```

Toate registrele pentru GPIO ale unui port se găsesc în aceeași locație, cu offset-i diferiți față de aceeași adresă de bază. Atât adresa de bază a fiecărui port cât și offset-ii pentru toți regiștrii se găsesc atât în datasheet cât și ca definiții în header-ele nucleului.

Regiștrii se pot modifica direct cu __raw_writel sau cu biblioteca gpio. Biblioteca face în plus verificări de concurență, apoi apelează tot __raw_writel. Funcțiile puse la dispoziție sunt următoarele:

- int gpio_direction_input(unsigned gpio)
- int gpio_direction_output(unsigned gpio, int value), unde value este valoarea inițială.
- void gpio_set_value(unsigned gpio, int value)

gpio este un identificator dat fiecărui pin, format cu macro-ul GPIO_PIN_PA(nr) (pentru portul A). Header-ele necesare pentru funcțiile și macrourile enunțate sunt cele din exemplul de mai jos:

```
#include <asm/gpio.h>
#include <mach/at32ap700x.h>

...

void set_pin()
{
    gpio_direction_output(GPIO_PIN_PA(13),0); // configurează pinul 13 de pe portul A ca ieșire și valoarea inițială 0
}
...
```

Exercițiu

- Scrieți un modul care la inițializare sa aprindă cele 3 LED-uri și la ieșire să le inchidă.
- HINT: LED-urile sunt PA16, PA19, PE19

Referințe

Dacă vreți să citiți mai multe:

- Laborator PSO
- Linux Kernel Module Programming Guide
- Linux Device Drivers 3rd edition