

Nowe jądro 2.6 – programowanie jądra i sterowników – odcinek 6

TECHNIKA JĄDROWA

Do nowości w jądrze 2.6 należy model urządzeń i system plików sys.

Pokażemy, jak wykorzystać je we własnych modułach jądra.

EVA-KATHARINA KUNST I JÜRGEN QUADE

Sterowniki sprzętowe stanowią sporą część jądra. Linux 2.6 udostępnia jednolite interfejsy do ich programowania i wykorzystywania: model urządzeń i system plików sys. Informacje na ich temat są lakoniczne. Nic dziwnego, te składniki są zupełnie nowe i trwa ich rozwój. Każdy, kto chce we własnych sterownikach urządzeń obsługiwać nowy model, musi sam wyszukiwać zmienne, interfejsy i semantykę wywołań w kodzie źródłowym Linuksa. W drzewie jądra znajdują się pewne pliki tekstowe, które opisują nowy model sterowników (zobacz *Documentation/driver-model*), lecz w pewnych szczegółach odbiegają one od faktycznej implementacji.

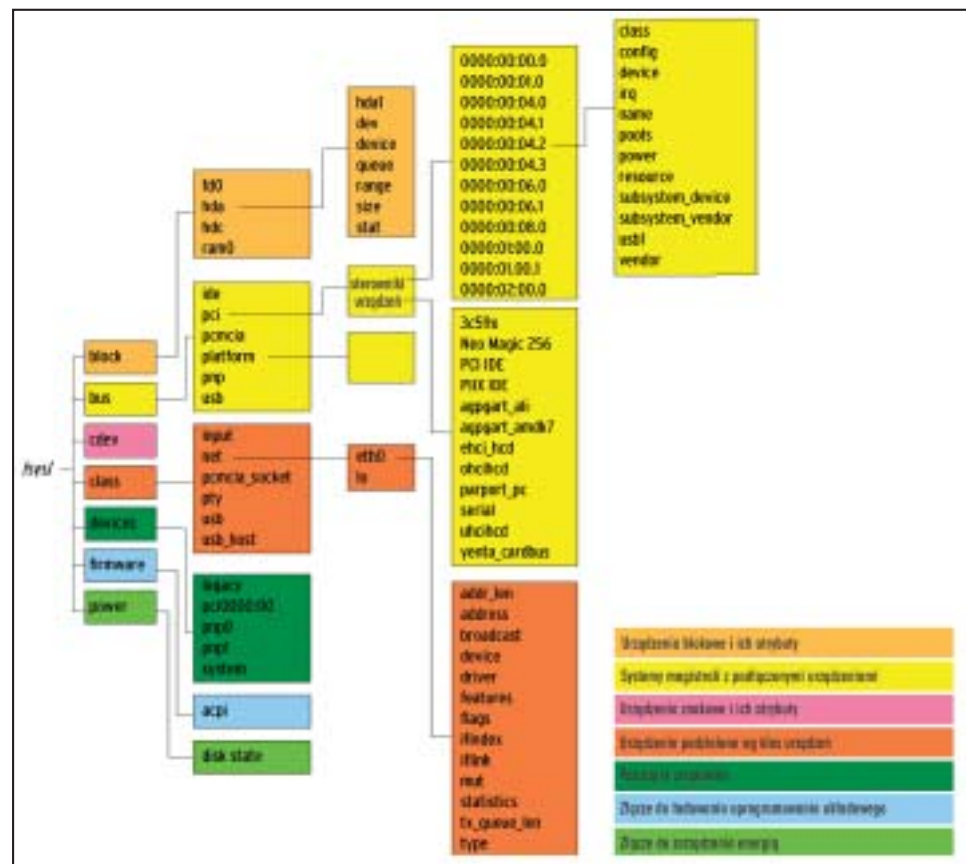
Także nazewnictwo jest bałaganiarskie. I tak nowy model urządzeń pojawia się w dyskusji programistów raz jako „device model” (model urządzeń), a raz jako „driver model” (model sterownika). Model urządzeń stanowi zarazem zestaw funkcji API jądra, a także system wykonawczy, który zarządza strukturami sterowników w jądrze.

Porządkowanie za pomocą sysfs

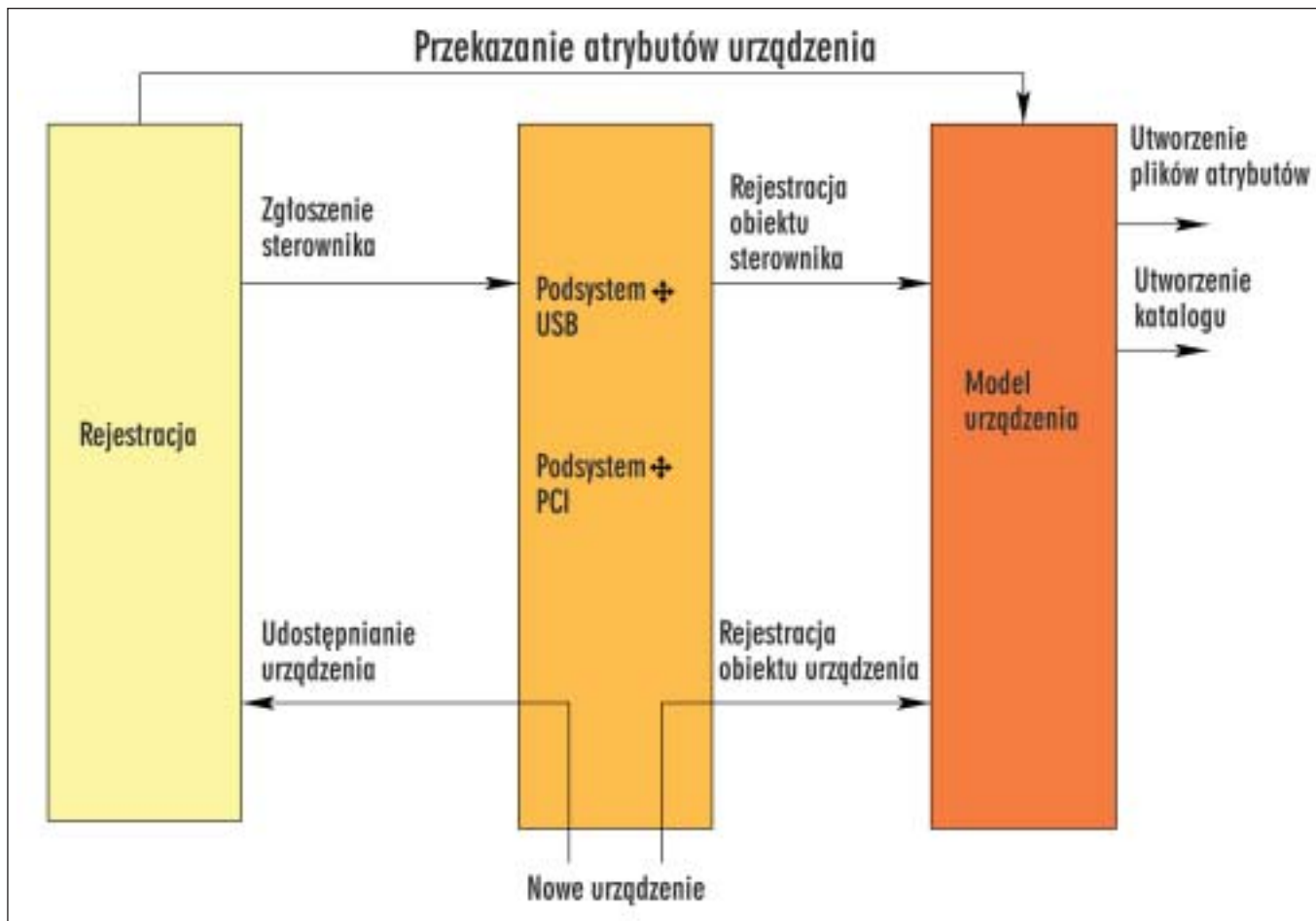
Linus Torvalds przyjął model urządzeń w nowym jądrze, choć nie jest on jeszcze zupełnie dojrzały, gdyż wielu programistów wiąże duże nadzieje z nowymi składnikami. Model wprowadza pewien porzą-

dek do krainy sterowników, ułatwia zarządzanie energią, a także – zamiast systemu

plików urządzeń (devfs) – zarządza plikami urządzeń [1].



Rysunek 1: System plików sys kategoryzuje informacje modelu urządzeń i przedstawia je w postaci drzewa katalogów. Dla każdego obiektu, na przykład sterownika, istnieje osobny katalog. Atrybuty obiektów, chociażby dane na temat stanu sterownika, są udostępniane w postaci plików.



Rysunek 2: Jeśli sterownik ma obsługiwać własne atrybuty lub jeśli nie zgłasza się on w podsystemie sterowników (takim jak PCI, USB), programista musi jawnie użyć modelu urządzenia.

Model urządzeń odzwierciedla relacje procesorów systemu z układami sterującymi, a tych z kolei z kartami rozszerzeń i pozostałym sprzętem. Obok struktury sprzętu model odzwierciedla także należące do niego składniki programowe, na przykład sterowniki urządzeń. Na podstawie informacji zgromadzonych w modelu urządzenia jądro jest w stanie obsługiwać strukturalne zarządzanie energią. Istnieje więc kolejność, w jakiej system operacyjny wyłącza sprzęt: najpierw muszą zostać wyłączone urządzenia przyłączone do magistrali, zanim w stan oszczędzania energii przejdzie sama magistrala, a następnie procesor.

System plików `sys` jest wirtualny tak jak system plików `proc`: jego katalogi i pliki są tworzone dynamicznie i nie znajdują się faktycznie na dysku twardym. Drzewo katalogów generowane przez system plików `sys` odbija strukturę sprzętów i jego oprogramowania wewnątrz samego systemu. Użytkownik styka się z modelem urządzeń

za pośrednictwem systemu plików `sys` (`sysfs`). Dlatego też musi dołączyć system plików `sys` do drzewa katalogów.

```
mount -t sysfs sysfs /sys
```

Aby otrzymać informacje z systemu plików `sys`, można posłużyć się jego własnym zestawem funkcji API, biblioteką `libsysfs` [4]. Przy stosowaniu przestarzałego już systemu `devfs` poprzez `udev` (zobacz [1]) system plików `sys` odgrywa decydującą rolę.

Rysunek 1 pokazuje, na jakie kategorie dzielą się sterowniki i urządzenia oraz jakie dodatkowe interfejsy są dostępne, na przykład do ładowania oprogramowania układowego [2].

Obiekty pod wieloma postaciami

Zgodnie z różnymi kategoriami te same urządzenia i sterowniki pojawiają się w systemie plików `sys` w wielu miejscach. Jedna

karta sieciowa PCI jest szufladkowana przez model urządzenia jako magistrala (PCI), ale też jako urządzenie sieciowe. Występowanie plików w wielu miejscach jest realizowane za pomocą dowiązań symbolicznych.

System plików `sys` pokazuje nie tylko strukturę urządzenia. Liście drzewa są również pseudoplikami, które można odczytać i zapisać. Tak jak w systemie plików `proc`, można tu czytać i ustawiać atrybuty sprzętu i oprogramowania. Możliwości wykorzystania są niemalże nieograniczone – np. nazwa producenta, numer wersji, stan urządzenia, statystyka transferu karty sieciowej...

Automatyczna rejestracja

Włączanie i wyłączanie urządzeń oraz sterowników przy zastosowaniu modelu urządzeń następuje często niejawnie. To zadanie przejmują podsystemy PCI, USB i podsys-

Listing 1: Sterownik urządzenia wirtualnego

```

01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/device.h>
06 #include <linux/completion.h>
07
08 #define DRIVER_MAJOR 240
09
10 MODULE_LICENSE ("GPL");
11
12 static struct file_operations
Fops;
13 static DECLARE_COMPLETION
( DevObjectIsFree );
14 static int Frequenz; //zmienna
stanu urządzenia
15
16 static void mydevice_release
( struct device *dev )
17 {
18     complete( &DevObjectIsFree );
19 }
20
21 struct platform_device mydevice =
{
22     .name = "MyDevice",
23     .id = 0,
24     .dev = {
25         .release = mydevice_release,
26     }
27 };
28
29 static struct device_driver
mydriver = {
30     .name = "MyDevDrv",
31     .bus = &platform_bus_type,
32 };
33
34 static ssize_t ReadFreq
( struct device *dev, char *buf )
35 {
36     sprintf(buf, "Częstość: %d",
Frequenz );
37     return strlen(buf)+1;
38 }
39
40 static ssize_t WriteFreq( struct
device *dev, const char *buf,
size_t count )
41 {
42     Frequenz = simple_strtoul
( buf, NULL, 0 );
43     return strlen(buf)+1;
44 }
45
46 static DEVICE_ATTR( freq ,
S_IRUGO|S_IWUGO, ReadFreq, WriteFreq );
47
48
49 static int __init DrvInit(void)
50 {
51     if(register_chrdev(DRIVER
_MAJOR, "MyDevice", &Fops) == 0) {
52         driver_register(&mydriver);
// rejestrowanie sterownika
53     platform_device_register
( &mydevice );// rejestrowanie
urządzenia
54     mydevice.dev.driver = &mydriver;
// połączenie ich ze sobą
55     device_bind_driver
( &mydevice.dev ); // połączenie
sterownika z urządzeniem
56     device_create_file
( &mydevice.dev, &dev_attr_freq );
// atrybuty
57     return 0;
58 }
59     return -EIO;
60 }
61
62 static void __exit DrvExit(void)
63 {
64     device_remove_file( &mydevi-
ce.dev, &dev_attr_freq );
65     device_release_driver
( &mydevice.dev );
66     platform_device_unregister
( &mydevice );
67     driver_unregister(&mydriver);
68     unregister_chrdev
(DRIVER_MAJOR,"MyDevice");
69     wait_for_completion
( &DevObjectIsFree );
70 }
71
72 module_init( DrvInit );
73 module_exit( DrvExit );

```

tem sieciowy (zobacz Rysunek 2). Kiedy tylko sterownik wywoła funkcję *register_chrdev()*, przedstawioną w [3], jądro tworzy wpis. Jednak programista jądra powinien jawnie obsługiwać model urządzenia.

Jeśli zaś programista chce utworzyć atrybuty do odczytu i zapisu, powinien użyć do tego celu modelu urządzenia. W innych wypadkach model urządzenia jest obligatoryjny, mianowicie wtedy, gdy:

- urządzenia nie są podłączane przez USB, PCI lub swoisty, standardowy system magistrali;

- zostaje dołączony nowy system magistrali lub

- są definiowane nowe klasy urządzeń.

Programista we własnym kodzie musi także zgłosić sterownik i urządzenie w modelu urządzenia, a także udostępnić sensowne atrybuty.

Zgłaszanie sterowników w modelu urządzenia

Kiedy sterownik zgłasza się w podsystemie PCI, USB lub IDE, jest on automatycznie umieszczany w systemie plików sys, przyporządkowany odpowiedniemu obiektowi magistrali. To samo dotyczy magistrali I2C (między układami sprzętowymi), EISA i Microchannel. We wszystkich innych wypadkach sterownik musi albo zdefiniować nowy system magistrali, albo podłączyć się pod tak zwaną magistralę platformy. Ponieważ wewnątrz modelu urządzeń każdy sterownik musi mieć przyporządkowany system magistrali, magistrala platformy stanowi pojemnik dla wszystkich sterowników, które nie należą do innych magistrali.

Przyłączenie sterownika do magistrali platformy przebiega standardowo: najpierw

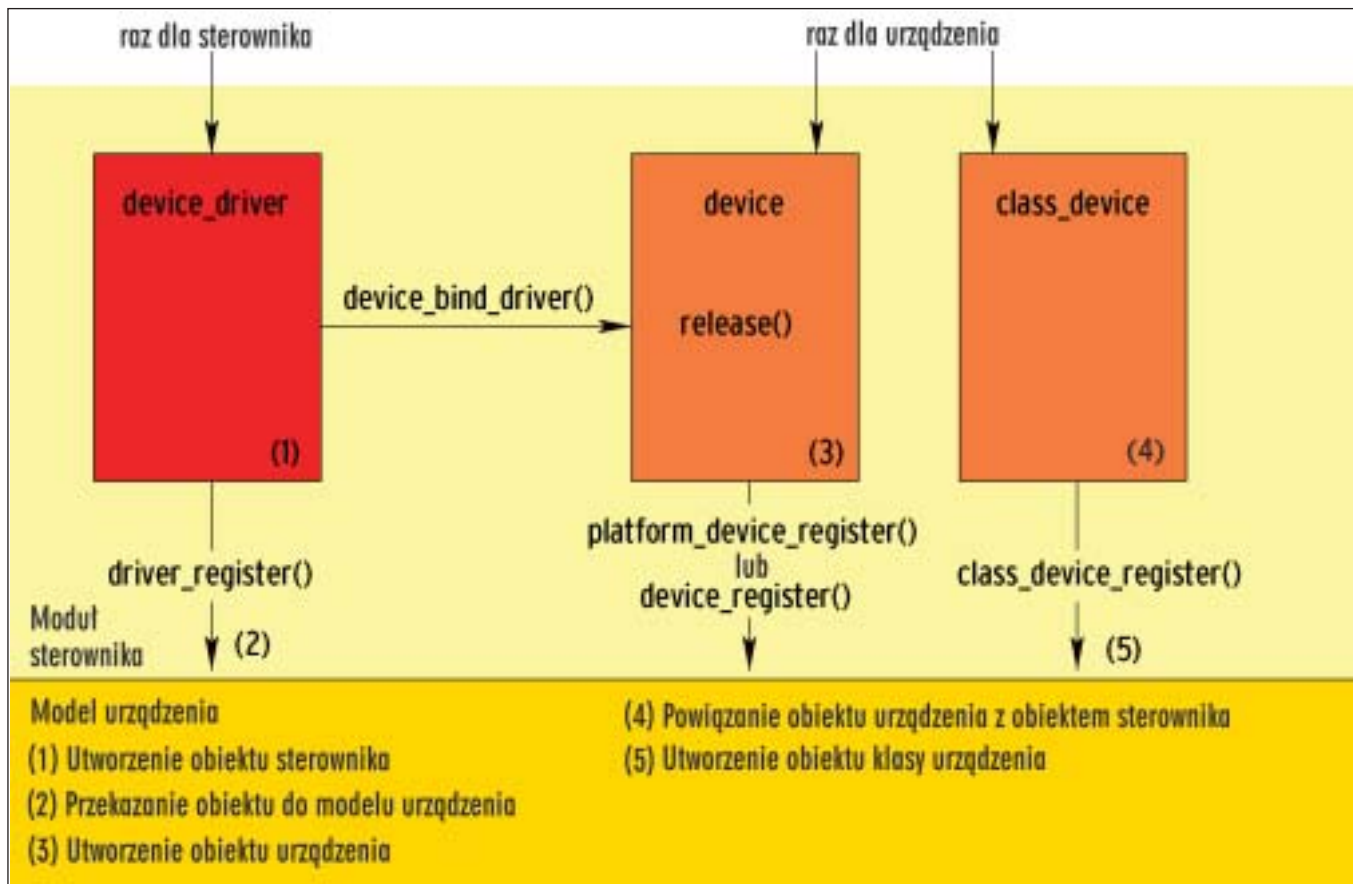
programista definiuje i inicjuje obiekt, a więc strukturę danych w języku C. Następnie przekazuje obiekt do jądra, które załatwia resztę. Kiedy sterownik nie jest już używany, musi się wyrejestrować z jądra.

Na Listingu 1 zdefiniowana jest struktura danych *struct device_driver* i inicjuje ona elementy *name* i *bus* (wiersz 29-32). Typ danych *struct device_driver* i symbol *platform_bus_type* zostały zdefiniowane w pliku nagłówkowym *linux/device.h*.

Przy inicjacji sterownik przekazuje obiekt do jądra przez wywołanie funkcji *driver_register()* (wiersz 52).

Po przetworzeniu funkcji *driver_register()* w zamontowanym systemie plików sys pojawia się nowy katalog */sys/bus/platform/drivers/MyDevDrv/*.

Funkcja *driver_unregister()* usuwa później ten wpis (wiersz 67).



Rysunek 3.: Moduł sterownika musi zdefiniować obiekty dla sterownika i dla każdego urządzenia, a następnie przekazać modelowi urządzeń.

Zgłaszanie urządzeń

Moduły muszą nie tylko zgłaszać sterownik w czasie działania systemu modelu urządzenia, lecz także samo urządzenie, jeśli nie zostanie automatycznie rozpoznane przez sterownik i podsystem PCI, USB lub inny. Takie urządzenia są określone w modelu jako urządzenia platformy lub systemowe. Urządzenia systemowe są w istocie cegiełkami samego komputera, a więc procesor, kontroler przerwania czy też czasomierz. Te wpisy znajdują się w katalogu `/sys/devices/system/`. Wszystkie pozostałe urządzenia są urządzeniami platformy. Dla nich tworzone są wpisy w katalogu `/sys/devices/legacy/`.

Moduł definiuje urządzenie jako obiekt i inicjuje je przy zgłaszaniu, tak samo jak w wypadku sterowników (zobacz Listing 1, wiersz 21-27). Kod musi zawierać jednak poza tym funkcję `release()` (wiersz 16). Jest ona wywoływana przez jądro, kiedy model urządzenia nie potrzebuje już obiektu urządzenia. Moduł musi znowu sprawdzić, czy obiekt zwolnił pamięć należącą do obiektu po wywołaniu funkcji `release()`.

Najłatwiej zrobić to za pomocą obiektu Completion [1] (wiersz 13, 18 i 69), co jest oczekiwane przy definicji sterownika.

Funkcja `platform_device_register()` przekazuje obiekt modelowi urządzenia (wiersz 53). Wówczas jądro tworzy katalog `/sys/devices/legacy/MyDevice/`. Podprogram `platform_device_unregister()` umożliwia później usunięcie wpisu katalogu (wiersz 66).

Punkty dowiązania

Łatwo zarejestrować obiekty sterowników i urządzeń w modelu urządzenia – lecz oba obiekty nie pozostają ze sobą w żadnym związku. Tak więc model urządzeń nie wie, że urządzenie jest używane przez przykładowy sterownik i na odwrót, iż ten sterownik jest odpowiedzialny za właśnie zdefiniowane urządzenie. To powiązanie tworzy funkcja `device_bind_driver()`. Parametrem tej funkcji jest adres obiektu urządzenia. Przed jej wywołaniem adres obiektu sterownika `mydriver` musi jednak zostać wprowadzony do obiektu urządzenia `mydevice`.

```
mydevice.dev.driver = &mydriver;
```

```
device_bind_driver(&mydevice.dev);
```

Model urządzenia tworzy dowiązanie z katalogu `/sys/bus/platform/drivers/MyDriver` do `/sys/devices/legacy/MyDevice`.

Definiowanie własnych atrybutów

Wcześniej model urządzeń zmieniał wpisy katalogów w systemie plików `sys`. Naprawdę atrakcyjną możliwością jest prosty odczyt i zapis parametrów oraz stanów urządzenia i sterownika poprzez pliki atrybutów. Do tego konieczne są dwie rzeczy. Po pierwsze, należy zdefiniować funkcje wywoływane przez jądro, kiedy użytkownik odczytuje lub zapisuje atrybut. Po drugie, pliki atrybutów muszą zostać utworzone w systemie plików `sys` (zobacz ramka).

Funkcje odczytu plików atrybutów zawsze mają dwa parametry. Pierwszy określa przynależny obiekt, na przykład obiekt urządzenia lub sterownika. Drugi parametr to adres obszaru pamięci, w którym powin-

ny być przechowywane dane atrybutów. Ponieważ przez system plików sys nie mogą być wymieniane duże ilości danych, ten bufor ma wielkość ograniczoną do jednej strony pamięci. Ta stała granica sprawia, że jego użycie jest łatwiejsze niż w wypadku systemu plików proc. Dane, które są wymieniane między sterownikiem a obszarem użytkownika, powinny być zakodowane w ASCII. Dzięki temu za pomocą programów systemowych można w łatwy sposób uzyskać dostęp do atrybutów. Tak więc polecenie `echo „1” > /sys/legacy/MyDevice/attr` zapisuje wartość 1 w atrybucie `attr`, podczas gdy polecenie `cat /sys/legacy/MyDevice/attr` odczytuje i podaje odpowiednią wartość.

Funkcje zapisu, oprócz wspomnianych dwóch parametrów, mają też trzeci, wskazujący liczbę zapisywanych bajtów. Poza tym funkcje odczytu i zapisu przekazują dane wewnątrz obszaru jądra. Za przesłanie do obszaru użytkownika odpowiada sam model urządzenia.

Gdy funkcje dostępu do plików atrybutów są zdefiniowane, w systemie plików sys muszą zostać utworzone pliki. Makra `DRIVER_ATTR()` i `DEVICE_ATTR()` definiują i inicjują obiekty, które następnie funkcje `driver_create_file()` i `device_create_file()` przekazują do modelu urządzenia.

Nazwy obiektów utworzonych za pomocą makr `DRIVER_ATTR()` i `DEVICE_ATTR()` pojawiają się w pierwszym parametrze z przedrostkiem `driver_attr_` lub `device_attr_`. Tak więc w następującym wierszu zostaje zdefiniowany obiekt `driver_attr_version`:

```
static DRIVER_ATTR( version,
S_IRUGO,
ShowVersion, NULL );
```

Drugi parametr określa prawa dostępu do pliku atrybutów. Plik nagłówkowy `linux/stat.h` deklaruje możliwe wartości (zobacz Tabela 1). Ponieważ ten przykład korzysta tylko z funkcji odczytu `ShowVersion()`, makro ustawia także tylko prawa odczytu (tutaj dla użytkownika, grupy i pozostałych). Ostatni parametr ma wartość `NULL`; nie wskazuje więc na funkcję zapisu, która w przeciwnym razie w tym miejscu by się pojawiała. Znaczenie parametru dotyczy odpowiednio makra `DEVICE_ATTR()` (zobacz Ramka „Programowanie atrybutów” i Listing 1).

Funkcje `driver_remove_file()` i `device_remove_file()` usuwają utworzone pliki, kiedy sterownik na przykład już nie będzie korzystał z urządzenia.

Pliki atrybutów także dla PCI

W wypadku urządzeń PCI struktury danych `struct pci_driver` i `struct pci_dev` zawierają zawsze `struct device_driver` i `struct device`. Nic łatwiejszego więc utworzyć także dla nich pliki atrybutów. Podczas inicjacji sterownika funkcja `driver_create_file()` tworzy pliki atrybutów, zaś przy inicjacji urządzenia funkcja `device_create_file()` – odpowiednie pliki urządzenia.

Listing 2: Programowanie atrybutów

```
01 static ssize_t ReadFreq(
02     ( struct device *dev, char *buf )
03     ...
04 )
05
06 static ssize_t WriteFreq(
07     ( struct device *dev, const char *buf, size_t count )
08     ...
09 )
10
11 static DEVICE_ATTR( freq,
12     S_IRUGO|S_IWUGO, ReadFreq,
13     WriteFreq );
14
15 static int DeviceProbe( ... )
16 {
17     ...
18 }
19
20 static int DeviceRemove( ... )
21 {
22     ...
23     device_remove_file(&MyDevice,
24         dev, &dev_attr_freq );
25     ...
26 }
```

Programowanie atrybutów

Makro `DEVICE_ATTR()` (Listing 2, wiersz 11) z użyciem pliku `freq` definiuje strukturę danych `dev_attr_freq` (przedrostek `dev_attr_` zostaje dołączony do nazwy zmiennej). Zapewnia ono dostęp z prawami do odczytu i zapisu oraz tworzy funkcje dostępu. Podczas inicjacji urządzenia w funkcji `DeviceProbe()` funkcja `device_create_file()` (wiersz 16) tworzy plik atrybutu.

Jądro wywołuje funkcję `WriteFreq()`, kiedy aplikacja zapisuje dane w pliku atrybutu `freq`. Dane do zapisania znajdują się już w obszarze jądra. Kiedy aplikacja odczytuje plik atrybutów `freq`, jądro wywołuje funkcję `ReadFreq()`. Żądane dane mogą być bezpośrednio zapisane w buforze `buf`. Funkcja zwraca liczbę odczytanych bajtów.

Podczas dezinicjacji urządzenia w funkcji `DeviceRemove()` funkcja `device_remove_file()` (wiersz 23) usuwa plik atrybutu z systemu plików `sys`.

Tabela 1: Prawa dostępu

Kopf: Symbol	Znaczenie
<code>S_IRWXU</code>	Odczyt, zapis i wykonywanie dla użytkownika
<code>S_IRUSR</code>	Odczyt dla użytkownika
<code>S_IWUSR</code>	Zapis dla użytkownika
<code>S_IXUSR</code>	Wykonywanie dla użytkownika
<code>S_IRWXG</code>	Odczyt, zapis i wykonywanie dla grupy
<code>S_IRGRP</code>	Odczyt dla grupy
<code>S_IWGRP</code>	Zapis dla grupy
<code>S_IXGRP</code>	Wykonywanie dla grupy
<code>S_IRWXO</code>	Odczyt, zapis i wykonywanie dla pozostałych
<code>S_IROTH</code>	Odczyt dla pozostałych
<code>S_IWOTH</code>	Zapis dla pozostałych
<code>S_IXOTH</code>	Wykonywanie dla pozostałych
<code>S_IRWXUGO</code>	Kombinacja <I> <code>S_IRWXU S_IRWXG S_IRWXO</code> <I>
<code>S_IALLUGO</code>	Kombinacja <I> <code>S_ISUID S_ISGID S_ISVTX S_IRWXUGO</code> <I>
<code>S_IRUGO</code>	Kombinacja <I> <code>S_IRUSR S_IRGRP S_IROTH</code> <I>
<code>S_IWUGO</code>	Kombinacja <I> <code>S_IWUSR S_IWGRP S_IWOTH</code> <I>
<code>S_IXUGO</code>	Kombinacja <I> <code>S_IXUSR S_IXGRP S_IXOTH</code> <I>

Listing 3: Nowe klasy urządzeń

```

01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/device.h>
06
07 MODULE_LICENSE("GPL");
08
09 static void GameDeviceRelease(
10 struct device *dev );
11 int GameClassHotplug( struct
12 class_device *dev, char **envp,
13 int num_envp, char *buffer,
14 int buffer_size );
15 void GameClassRelease( struct
16 class_device *dev );
17
18 static struct device_driver
19 GameDriver = {
20 .name = "GameDriver",
21 .bus = &platform_bus_type,
22 };
23 struct platform_device GameDevice = {
24 .name = "GameDevice",
25 .id = 0,
26 .dev = {
27 .release = GameDeviceRelease,
28 };
29 };
30 static struct class GameClass = {
31 .name = "GameClass",
32 .hotplug = GameClassHotplug,
33 .release = GameClassRelease,
34 };
35 static struct class_device
36 GameClassDevice = {
37 .class = &GameClass,
38 };
39 static struct file_operations
40 Fops;
41 static DECLARE_COMPLETION
42 ( DevObjectIsFree );
43 static void GameDeviceRelease(
44 struct device *dev )
45 {
46 complete( &DevObjectIsFree );
47 }
48 int GameClassHotplug( struct
49 class_device *dev, char **envp,
50 int num_envp, char *buffer,
51 int buffer_size)
52 {
53 printk("GameClassHotplug( %p
54 )\n", dev );
55 return 0;
56 }
57 void GameClassRelease( struct
58 class_device *dev )
59 {
60 printk("GameClassRelease( %p
61 )\n", dev );
62 }
63 static int __init DrvInit(void)
64 {
65 if(register_chrdev(240,
66 "gamedevice", &Fops) == 0) {
67 driver_register(&GameDriver);
68 platform_device_register
69 ( &GameDevice );
70 GameDevice.dev.driver =
71 &GameDriver;
72 device_bind_driver
73 ( &GameDevice.dev );
74 class_register( &GameClass );
75 GameClassDevice.dev =
76 &GameDevice.dev;
77 strcpy( (void *)&GameClass
78 Device.class_id,
79 "GameClassDevice", 16 );
80 class_device_register
81 ( &GameClassDevice );
82 return 0;
83 }
84 return -EIO;
85 }
86 static void __exit DrvExit(void)
87 {
88 class_device_unregister
89 ( &GameClassDevice );
90 class_unregister( &GameClass );
91 device_release_driver
92 ( &GameDevice.dev );
93 platform_device_unregister
94 ( &GameDevice );
95 driver_unregister(&GameDriver);
96 unregister_chrdev(240,
97 "gamedevice");
98 wait_for_completion
99 ( &DevObjectIsFree );
100 }
101 int module_init( DrvInit );
102 module_exit( DrvExit );

```

```

driver_create_file( &pcidrv.driver,
&driver_attr_mytext );
device_create_file( &pcidev->dev,
&dev_attr_mytext );

```

Klasy urządzeń

Procedura zgłaszania urządzenia w klasie urządzeń jest znana: kiedy sterownik zgłosi się w podsystemie – na przykład podsystemie sieciowym – ten realizuje rejestrację w modelu urządzeń. Jeśli zaś nie ma żadnego odpowiedniego podsystemu, programista sterownika sam musi się o to zatroszczyć. Tak jest w wypadku urządzeń należących do klas „input” lub „pcmcia_socket”. W tym celu programista jądra definiuje obiekt, inicjuje go i przekazuje modelowi urządzeń.

Przekazanie następuje podczas inicjacji urządzenia, na przykład w wypadku sprzętu podłączonego do magistrali PCI w funkcji detekcji. Wówczas należy postępować w następujący sposób:

- Nazwy katalogów, w których później pojawiają się wpisy, kopiujemy do składowej *class_id*.

- Adres obiektu sterownika *pci_drv.driver* (typ *struct device_driver*) wpisujemy do struktury *dev* (typ *struct device*).

- Adres obiektu urządzenia *dev->dev* (typ *struct device*) wstawiamy do obiektu klasy urządzenia *myclass*, a mianowicie do składowej *myclass.dev*.

- Adres obiektu klasy *input_class* (zdefiniowanej wstępnie w pliku *<linux/input.h>*), w której katalogu zostanie utworzony podkatalog *class_id*, zapisujemy w obiekcie klasy urządzenia *myclass*, a mianowicie w składowej *myclass.class*.

- Kiedy obiekt jest tak zainicjowany, funkcja *class_device_register()* przekazuje go do jądra.

```

static struct class_device
myclass;

...

strcpy((void *)&myclass.class_id,
"MyDev", 6 );
dev->dev.driver = &pci_drv.driver;
myclass.dev = &dev->dev;
myclass.class = &input_class;
class_device_register( &myclass );

```

Model urządzenia tworzy następnie katalog *MyDev* i ustawia dowiązania do odpowiednich obiektów urządzenia i sterownika.

```
/sys/class/input
^-- MyDev
|-- device -> ->> §§
../../../../devices/
pci0000:00/0000:00:08.0
^-- driver -> ->> §§
../../../../bus/pci/drivers/pci_drv
```

Także urządzenie klasy może mieć atrybuty. W stosunku do zaprezentowanej już metody programowanie różni się jedynie przedrostkiem nazwy makr i funkcji (*CLASS* zamiast *DRIVER* lub *DEVICE*).

Ciekawsze jest definiowanie własnej klasy. W tym celu należy zaimplementować funkcje *hotplug* i *release*, a następnie przekazać je za pomocą obiektu klasy do modelu urządzenia Linuksa. Funkcja *hotplug* jest wywoływana, kiedy tylko sterownik zgłosi się w klasie, zaś funkcja *release* wtedy, gdy się z niej wyrejestruje. Na Listingu 3 przedstawiono kompletny przykład sterownika, który tworzy klasę urządzenia o nazwie *GameClass*, urządzenie klasy *GameDevice* i urządzenie wirtualne *GameClassDevice*.

Definiowanie innych magistrali zewnętrznych

Definicja własnej magistrali jest jeszcze mniej skomplikowana niż utworzenie nowej

klasy. W tym celu programista definiuje obiekt magistrali, inicjuje go przy użyciu nazwy i rejestruje funkcją *bus_register()*:

```
struct bus_type can = {
.name = "CAN",
};
...
static int __init MyModulInit
(void)
{
...
bus_register( &can );
```

Koncepcja modelu urządzenia ma jasną strukturę, lecz ponieważ należący do niej kod jądra nie jest jeszcze całkowicie dojrzały, niewielkie błędy programistów prowadzą szybko do zawieszenia systemu – wówczas pozostaje tylko zresetowanie komputera.

W następnym odcinku

Praca się jednak opłaci, bo model urządzenia ma przyszłość. Nie tylko z powodu coraz ważniejszego zarządzania energią, lecz także dlatego, że system plików przejmie wiele dotychczasowych zadań systemu plików *proc*. Ten jednak nie zostanie całkowicie zastąpio-

ny. Dlatego też w następnym odcinku Techniki jądrowej pokażemy, jak działa system plików *proc* i w jaki sposób programuje się dla niego własne moduły. (ofr) ■

INFO

- [1] Greg Kroah-Hartmann, „udev – A Userspace Implementation of devfs”: <http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Kroah-Hartman-OLS2003.pdf>
- [2] Podręcznik Estrada Sainz: Firmware Class. Dokumentacja w źródłach jądra w pliku *Documentation/firmware_class*.
- [3] Eva-Katharina Kunst i Jürgen Quade „Technika jądrowa”, odcinek 4, Linux Magazine nr 17 (czerwiec 2005).
- [4] Libsysfs, składnik zestawu „Linux Diagnostic Tool”: <http://linux-diag.sourceforge.net>.

AUTORZY

Eva-Katharina Kunst, dziennikarka i Jürgen Quade, profesor w Hochschule Niederrhein, należą od czasu powstania Linuksa do fanów oprogramowania Open Source.

LinuxWorld Conference & Expo – Worldwide Series



LinuxWorld

- San Francisco: August 8 – 11, 2005 www.linuxworldexpo.com
- Beijing: August 17 – 19, 2005 www.linuxworldchina.com
- Moscow: September 7 – 9, 2005 www.linuxworldexpo.ru
- Cape Town: September 14 – 16, 2005 www.linuxworldexpo.co.za
- London: October 5 – 6, 2005 www.linuxworldexpo.co.uk
- Utrecht: November 9 – 10, 2005 www.linuxworldexpo.nl
- Frankfurt: November 15 – 17, 2005 www.linuxworldexpo.de
- Mexico City: February 14 – 17, 2006 www.linuxworldexpo.com
- Sydney: March 28 – 30, 2006 www.linuxworldexpo.com
- Boston: April 3 – 6, 2006 www.linuxworldexpo.com
- Milan: April 12 – 14, 2006 www.linuxworldexpo.it
- Toronto: April 24 – 26, 2006 www.linuxworldexpo.co.za
- Johannesburg: May 16 – 19, 2006 www.linuxworldcanada.com



World's leading Trade Show for Linux and Open Source in business

Where open minds meet!