Cuprins

Implementarea mecanismelor de intrare ieșire

Colecția de drivere dispozitiv la Linux si Windows – Diniță Mihai

Device drivere în Linux

În UNIX dispozitivele hardware sunt accesate de către utilizator prin intermediul fişierelor speciale, de tip device. Aceste fişiere sunt grupate în directorul /dev, iar apelurile de sistem open, read, write, close, seek, mmap, etc. sunt redirecţionate de către sistemul de operare către device driverul asociat cu dispozitivul fizic.

În lumea UNIX există două categorii de dispozitive şi implicit device drivere: de tip caracter şi de tip bloc. Această împărţire este făcută după viteza, volumul şi modul de organizare a datelor ce trebuie transferate de la dispozitiv către sistem şi invers. În prima categorie intră dispozitivele lente, care gestionează un volum mic de date, iar accesul la date nu necesită operaţii de căutare (seek) prea frecvente. Exemple sunt dispozitive cum ar fi tastatura, mouse-ul, porturile seriale, placa de sunet, joystick-ul. Cea de-a doua categorie cuprinde dispozitive la care volumul de date manipulate este mare, datele sunt organizate pe blocuri, operaţiile de căutare (seek) sunt frecvente. Exemple de dispozitive ce intră în această categorie sunt hard disk-urile, cdrom-urile, ram discurile, unităţile de bandă magnetică.

Pentru cele două tipuri de device drivere se folosesc API-uri diferite. Dacă pentru dispozitivele de tip caracter apelurile de sistem ajung direct la device drivere, în cazul dispozitivelor de tip bloc device driverele nu lucrează direct cu apelurile de sistem. Aceasta deoarece între user-space şi device driverul de tip bloc se interpune subsistemul de gestiune a fişierelor. Rolul acestuia este de a pregăti device driverului resursele necesare (buffere), de a menţine în buffer cache datele recent citite şi de a reordona operaţiile de citire şi scriere din motive de performanţă.

Identificator major şi minor

Tradiţional, în UNIX dispozitivele aveau asociate un identificator unic, fixat. Această tradiţie se păstrează şi în Linux, deşi este deja posibil ca identificatorii să se aloce dinamic (din motive de compatibilitate însă, majoritatea driverelor folosesc încă identificatori statici). Identificatorul este format din două părţi: **major** şi **minor**. Prima parte (major) identifică tipul dispozitivului (disc ide, disc scsi, port serial, etc.) iar cel de al doilea identifică un dispozitiv prezent în sistem (primul disc, al doilea port serial, etc.). De cele mai multe ori, majorul identifică driverul, în timp ce minorul identifică fiecare dispozitiv fizic deservit de driver.

După cum s-a precizat mai sus, anumiţi identificatori major sunt atribuiţi în mod static dispozitivelor. La alegerea identificatorului pentru un nou dispozitiv se pot folosi două metode: static (se alege un număr care pare să nu fie folosit deja) sau dinamic. În /proc/devices se găsesc dispozitivele încărcate, împreună cu identificatorul major.

Pentru a crea un fişier de tip device se poate folosi comanda mknod; aceasta ia ca argumente tipul (bloc sau caracter), majorul şi minorul dispozitivului (mknod name type major minor):

```
# mknod /dev/my_device c 42 0
```

Device drivere de tip caracter

Structuri de date importante

În kernel, un dispozitiv de tip caracter este reprezentat de structura cdev şi este folosit, după cum vom vedea, la înregistrarea acestuia în sistem.

Majoritatea operaţiilor cu drivere folosesc trei structuri importante: struct file_operations, struct file şi struct inode.

Operaţii

După cum am mai precizat, device driverele de tip caracter primesc nealterate apelurile de sistem efectuate de utilizatori asupra fişierelor speciale. Deci, pentru a implementa un device driver, vor trebui implementate apelurile de sistem de lucru cu fişiere: open, close, read, write, lseek, mmap, etc. Aceste operaţii se regăsesc în structura file_operations:

```
#include <linux/fs.h>

struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *file, loff_t offset, int whence);
   ssize_t (*read) (struct file *file, char __user * user_buffer,
                size_t size, loff_t *offset);
   ssize_t (*write) (struct file *file, const char __user * user_buffer,
                  size_t size, loff_t *offset);
   int (*ioctl) (struct inode *inode, struct file *file,
              unsigned int cmd, unsigned long arg);
   int (*open) (struct inode *inode, struct file *file);
   int (*release) (struct inode *inode, struct file *file);
};
```

Primul lucru care se observă atunci când ne uităm cu atenţie la operaţiile de mai sus, este faptul că signatura funcţiei diferă faţă de apelul de sistem pe care îl foloseşte utilizatorul. Aceasta pentru că sistemul de operare se interpune între utilizator şi device driver, pentru a simplifica implementarea în device driver.

Astfel, se observă că open nu primeşte ca parametru calea sau diverşii parametri care controlează modul de deschidere a fişierului. La fel se întâmplă şi cu read, write, close, ioctl, lseek care nu primesc ca parametru un descriptor de fişier. În schimb, aceste rutine primesc ca parametri două structuri: file şi inode. Ambele structuri reprezintă un fişier, dar din perspective diferite.

Structurile inode şi file

Un inode reprezintă un fişier din punctul de vedere al sistemului de fişiere. Atribute ale inode-urilor sunt dimensiunea, drepturile, timpii asociaţi fişierului. Un inode identifică în mod unic un fişier într-un sistem de fişiere .

Structura file reprezintă tot un fişier, dar mai aproape de punctul de vedere al utilizatorului. Dintre atributele structurii file enumerăm: inode-ul, numele fişierului, atributele de deschidere ale fişierului, poziţia în fişier. Toate fişierele deschise la un moment dat au asociate o structură de tip file.

Pentru a înţelege diferenţele dintre un inode şi un file, vom folosi o analogie din programarea orientată pe obiecte: dacă vom considera un inode o clasă, atunci file-urile sunt obiecte, adică instanţe ale clasei inode. Inode-ul reprezintă imaginea statică a fişierului (inode-ul nu are stare), pe când file reprezintă imaginea dinamică a fişierului (file-ul are stare).

Revenind la device drivere, cele două entităţi au aproape întotdeauna modalităţi standard de folosire: inode-ul se foloseşte pentru a determina majorul şi minorul device-ului asupra căruia se face operaţia, iar file-ul se foloseşte pentru a determina flag-urile cu care a fost deschis fişierul dar şi pentru a memora şi accesa mai târziu date private.

Structura file este reprezentată de struct file şi are următoarele câmpuri importante:

- `f_mode`, care specifică permisiunile pentru citire sau scriere
- `private_data`, un pointer care poate fi folosit de programator pentru a păstra date specifice driver-ului (este un pointer către o zonă de memorie nealocata)

Structura inode este reprezentată de struct inode şi conţine un câmp i_cdev, care este un pointer către dispozitivul de tip caracter (atunci când inode-ul referă file-ul unui dispozitiv de tip caracter).

Atunci când se creează un device driver, se recomandă crearea unei structuri care să conţină informaţii despre dispozitivul dat, informaţii utilizate în cadrul modulului. În cazul unui driver pentru un dispozitiv de tip caracter, structura va conţine un câmp de tipul struct cdev pentru a referi dispozitivul.

Minorul unui dispozitiv se folosea în versiunile anterioare de kernel pentru a indexa date private, alocate şi iniţializate la încărcarea driverului. Deşi acest lucru se mai practică, el este descurajat. Modalitatea recomandată de alocare şi folosire a datelor asociată cu un dispozitiv este prin definirea unei structuri de tipul my_device_data unde cdev reprezintă un dispozitiv de tip caracter şi este folosit după cum vom vedea la înregistrarea acestuia în sistem. Pointer-ul către membrul cdev se poate afla din inode, din câmpul i_cdev, iar apoi se poate afla pointerul spre structura asociată cu dispozitivul. De asemenea se observă că în câmpul private_data din file se pot memora informaţii la open care apoi sunt disponibile în rutinele read, write, close, etc.

Majoritatea parametrilor pentru operaţiile prezentate au semnificaţie directă:

- `file` şi `inode` au fost discutaţi deja;
- `size` reprezintă numărul de octeţi ce trebuie citiţi sau scrişi;
- `offset` reprezintă offsetul de unde trebuie citit sau scris (şi trebuie actualizat corespunzător);

- `user_buffer` reprezintă bufferul utilizatorului din care / în care se citeşte / scrie;
- `whence` reprezintă modalitatea de seek;
- `cmd` şi `arg` sunt parametrii trimişi de utilizatori la apelul `ioctl`.

Înregistrarea şi deînregistrarea dispozitivelor

Tipul dev_t este folosit pentru a păstra identificatorii unui dispozitiv (atât majorul, cât şi minorul) şi se poate obţine cu ajutorul macro-ului MKDEV:

MKDEV(int major, int minor);

Pentru a aloca şi dealoca (static) identificatorii unui dispozitiv, se folosesc funcţiile:

#include <linux/fs.h>

int register_chrdev_region(dev_t first, unsigned int count, char *name);
void unregister_chrdev_region(dev_t first, unsigned int count);

Este recomandat ca identificatorii de device drivere să fie alocaţi dinamic cu funcţia aloc_chrdev_region. Pentru o testare mai simplă, în teme va trebui să folosiţi rezervări.

Spre exemplu, următoarea secvenţă rezervă my_minor_count dispozitive, începând de la dispozitivul cu majorul my_major şi minorul my_first_minor:

#include <linux/fs.h>

```
...
   int err;
   err = register_chrdev_region(MKDEV(my_major, my_first_minor), my_minor_count,
                                "my_device_driver");
   if (err != 0) {
      /* report error */
      return err;
   }
...
```

Accesul la spaţiul de adresă al procesului

Un dispozitiv este interfaţa de comunicaţie între o aplicaţie şi hardware. Drept urmare, deseori va trebui să accesăm în cadrul unui device driver date din user-space. Accesarea spaţiului de adresă al proceselor nu se poate face, însă, direct (cum ar fi prin dereferenţierea unui pointer din user-space). Trebuie folosită una din funcţiile de mai jos:

- `put_user(type val, type* address);` pune în user-space la adresa `address` valoarea `val`; tipul poate fi unul pe 8, 16, 32, 64 de biţi (tipul maxim suportat depinde de platforma hardware); întoarce 0 în caz de succes, altă valoare în caz de insucces

- `get_user(type val, type* address);` analog cu funcţia precedentă, numai că `val` va fi setată la o valoare identică cu valoarea de la adresa user-space dată prin `address`

- `unsigned long copy_to_user(void *to, void *from, unsigned long size);` copiază din kernel-space de la adresa referită de `from` în user-space la adresa referită de `to`, `size` octeţi; întoarce 0 în caz de succes, altă valoare în caz de insucces

- `unsigned long copy_from_user(void *to, void *from, unsigned long size);` copiază din user-space de la adresa referită de `from` în kernel-space la adresa referită de `to`, size octeţi; întoarce 0 în caz de succes, altă valoare în caz de insucces

O secţiune uzuală de cod care lucrează cu aceste funcţii este prezentată mai jos:

```
#include <asm/uaccess.h>

if (copy_to_user(user_buffer, kernel_buffer, size))
   return -EFAULT;
else
   return SUCCESS;
```

Operaţii implementate de device drivere de tip caracter

Open şi close

În funcţia `open` se realizează operaţiile de iniţializare a unui dispozitiv. În majoritatea cazurilor, aceste operaţii se referă la iniţializarea dispozitivului şi completarea datelor specifice (în cazul în care este primul apel open). Funcţia `close` se ocupă de eliberarea resurselor specifice dispozitivului: se dealocă datele specifice şi se închide dispozitivul daca este ultimul apel close.

În cele mai multe cazuri, funcţia open va avea următoarea structură:

```
static int my_open(struct inode *inode, struct file *file)
{
   struct my_device_data *my_data =
         container_of(inode->i_cdev, struct my_device_data, cdev);

   /* validate access to device */
   file->private_data = my_data;

   /* initialize device */

   return 0;
```

```
}
```

O problemă care apare la implementarea funcţiei open este controlul accesului. Uneori este necesar ca un dispozitiv să fie deschis o singură dată la un moment dat; mai exact, nu se permite al doilea open înainte de close. Pentru a implementa această restricţie se alege o modalitate de tratare a unui apel open pentru un dispozitiv deja deschis: se poate întoarce o eroare (-EBUSY), se pot bloca apelurile open pâna la o operaţie de close sau se poate închide dispozitivul înainte de a realiza operaţia de open.

La apelul din user-space al funcţiilor open si close asupra dispozitivului, se vor apela operaţiile my_open şi my_close din driver. Un exemplu de apel din user-space:

```
int fd = open("/dev/my_device", O_RDONLY);
if (fd < 0) {
   /* handle error */
}

close(fd);
```

Read şi write

Funcţiile read şi write transferă date între dispozitiv şi user-space: funcţia read citeşte datele de la dispozitiv şi le transferă în user-space, în timp ce write citeşte datele din user-space şi le scrie pe dispozitiv. După cum s-a precizat, buffer-ul primit ca parametru reprezintă un pointer în user-space, motiv pentru care este necesară folosirea funcţiilor copy_to_user sau copy_from_user.

Valoarea întoarsă este numărul de octeţi scrişi sau citiţi. Dacă valoarea întoarsă este mai mică decât parametrul size (numărul de octeţi ceruţi), înseamnă ca s-a realizat un transfer parţial. De cele mai multe ori, aplicaţia apelează din nou funcţia corespunzătoare apelului de sistem (read sau write) până când se transferă numărul de date cerut.

Un exemplu de funcţie read:

```
static int my_read(struct file *file, char __user *user_buffer,
                           size_t size, loff_t *offset)
{
   struct my_device_data *my_data =
         (struct my_device_data*) file->private_data;

   /* read data from device in my_data->buffer */
   if(copy_to_user(user_buffer, my_data->buffer, my_data->size))
      return -EFAULT;

   return my_data->size;
}
```

Structura funcţiei write este similară: citeşte date de la dispozitiv folosind funcţia copy_from_user şi le scrie pe dispozitiv.

La apelul funcţiilor read şi write din user-space (folosind descriptorul de fişier obţinut în urma unui apel open), se vor apela operaţiile my_read şi my_write din driver. Un exemplu de cod pentru user-space:

```
read(fd, buffer, size);
write(fd, buffer, size);
```

Ioctl

Pe lângă operaţii de read şi write, un driver mai are nevoie de posibilitatea de a realiza anumite operaţii de control asupra dispozitivului fizic. Valoarea trimisă ca parametru (unsigned long arg) reprezintă valoarea transmisă din user-space. Dacă la apelul din user-space se transmite un întreg, acesta poate fi accesat direct. Dacă se trasmite un buffer, valoarea arg va fi un pointer către acesta şi trebuie accesat prin intermediul funcţiilor copy_to_user sau copy_from_user.

Înainte de a implementa funcţia `ioctl`, vor trebui alese numerele ce corespund comenzilor. O metodă este de a alege numere consecutive începând de la 0, dar se recomandă folosirea macrodefiniţiei `_IOC(dir, type, nr, size)` pentru generarea codurilor ioctl. Parametrii macrodefiniţiei sunt după cum urmează:

- `dir` reprezintă direcţia de transfer a datelor (`_IOC_NONE`, `_IOC_READ`, `_IOC_WRITE`);
- `type` reprezintă numărul ;
- `nr` este numărul codului `ioctl` specific dispozitivului;
- `size` dimensiunea datelor transferate.

Sincronizare - cozi de aşteptare

Cozile de aşteptare sunt extrem de utile în probleme de sincronizare. De multe ori este necesar ca un thread să aştepte terminarea unei operaţii, dar este de dorit ca această aşteptare să nu fie busy-wating. Folosind cozi de aşteptare şi funcţii care schimbă starea thread-ului din planificabil în neplanificabil şi invers putem rezolva aceste problem relativ simplu. În Linux, o coadă de aşteptare este o listă în care sunt trecute procesele care aşteaptă un anumit eveniment. O coadă de aşteptare este definita de tipul wait_queue_head_t şi poate fi manipulată cu următoarele funcţii:

- `void init_waitqueue_head(wait_queue_head_t * q);` iniţializează coada de aşteptare; dacă se doreşte iniţializarea cozii la compilare, se poate folosi macroul `DECLARE_WAIT_QUEUE_HEAD(queue)`

- `void wait_event(wait_queue_head_t q, int condition); int wait_event_interruptible(wait_queue_head_t q, int condition);` atâta timp cât condiţia este falsă, adaugă thread-ul curent la coada de aşteptare, îi setează starea la `TASK_UNINTERRUPTIBLE` sau `TASK_INTERRUTIBLE` şi execută scheduler-ul pentru planificarea unui nou thread; aşteptarea va fi întreruptă atunci când un alt thread va apela funcţia `wake_up()`

- `int wait_event_timeout(wait_queue_head_t q, int condition, int timeout); int wait_event_interruptible_timeout(wait_queue_head_t q, int condition, int timeout);` atâta timp cât condiţia este falsă, adaugă thread-ul curent la coada de aşteptare, îi setează starea la `TASK_UNINTERRUPTIBLE` sau `TASK_INTERRUPTIBLE` şi execută schedulerul pentru planificarea unui nou thread; aşteptarea va fi întreruptă atunci când expiră timeout-ul specificat, sau când un alt thread a apelat funcţia `wake_up()`

- `void wake_up(wait_queue_head_t * q);` pune toate thread-urile oprite din starea `TASK_INTERRUPTIBLE` şi `TASK_UNINTERRUPTIBLE` în starea `TASK_RUNNING`; scoate aceste thread-uri din coada de aşteptare

- `void wake_up_interruptible(wait_queue_head_t * q);` aceeaşi acţiune, însă se folosesc doar thread-urile cu starea `TASK_INTERRUPTIBLE`

Device drivere în Windows

Windows NT a preluat modelul I/O folosit de la sistemul de operare VMS. Acest model foloseşte pachete pentru a descrie operaţiile de I/O, pachete ce încapsulează informaţii cum ar fi operaţia (citire sau scriere), numărul de octeţi de citit, poziţia, etc. Aceste pachete poarta denumirea de **I/O Request Packets** (**IRP**). Device driverele (care sunt grupate în stive) vor primi aceste pachete prin intermediul rutinelor de dispatch. Device driverul poate să rezolve cererea imediat în rutina de dispatch, poate să marcheze operaţia ca fiind în aşteptare şi să o rezolve mai târziu, sau poate trimite pachetul următorului device driver din stivă.

I/O managerul este cel care se ocupă de translatarea apelurilor de sistem în IRP-uri, alocarea de buffere, trimiterea IRP-urilor în stiva de device drivere, trimiterea rezultatelor proceselor după rezolvarea unui IRP, şi toate operaţiile asociate cu IRP-urile.

Accesul la spaţiul de adresă al procesului

Relativ la transferul datelor între user-space şi kernel-space în general se folosesc două abordări:

- `BUFFERED I/O` managerul alocă un buffer în kernel space şi copiază datele din / în user space; device driverul poate accesa bufferul din kernel space

- `DIRECT I/O` managerul validează bufferul din user space şi adună informaţiile necesare accesării bufferului; device driverul trebuie să pregătească bufferul din user space pentru folosire prin maparea în kernel-space (şi eventual marcarea paginilor astfel încât acestea să nu fie evacuate).

I/O managerul este cel care realizează aceste operaţii şi pune la dispoziţia driver-ului un buffer în care se vor găsi datele transmise din spaţiul de adresă al procesului.

Structuri de date importante

Pentru fiecare driver încărcat în sistem există un obiect driver (DRIVER_OBJECT). Unui driver i se pot asocia mai multe dispozitive, descrise printr-un DEVICE_OBJECT (câte unul pentru fiecare device pe care îl controlează).

Structura DRIVER_OBJECT conţine câteva câmpuri importante:

- `DriverUnload`, pointer la funcţia de Unload a modulului
- `DeviceObject`, o listă înlănţuită a dispozitivelor asociate driver-ului

Structura DEVICE_OBJECT conţine următoarele câmpuri frecvent folosite:

- `DeviceExtension`, un bloc de memorie rezidentă alocată de I/O manager la înregistrarea dispozitivului; această structură poate fi folosită de către programatorul driverului pentru a păstra informaţii specifice dispozitivului.
- `Flags`, câmp ce specifica strategia de transfer a datelor din spaţiul de adresă al procesului: `DO_BUFFERED_IO` sau `DO_DIRECT_IO`

I/O Request Packet (IRP)

Un IRP este o structură alocată din memoria rezidentă şi este format dintr-un header şi informaţii pentru fiecare device driver din stivă.

Câmpurile din header accesibile device driverelor sunt:

- `IoStatus`, structură ce conţine câmpuri care vor trebui completate de device driver la terminarea operaţiei:
  - `Status`, starea operaţiei: `STATUS_SUCCESS` dacă operaţia s-a terminat cu succes, sau codul de eroare în caz contrar;
  - `Information`, numărul de octeţi transferaţi în caz de succes, 0 altfel;
- `AssociatedIrp.SystemBuffer`, pointer către bufferul alocat de sistem (dacă se foloseşte `BUFFERED I/O`;
- `MdlAddress`, pointer ce este folosit de device driver pentru maparea bufferului din user-space în kernel-space (dacă se foloseşte `DIRECT I/O`);
- `UserBuffer`, pointer către bufferul din user-space (dacă nu se foloseşte nici `BUFFERED I/O`, nici `DIRECT I/O`)

După header, în IRP sunt plasate informaţii adresate device driverelor din stivă. Pentru a afla informaţiile adresate device driverului curent, trebuie apelată funcţia IoGetCurrentIrpStackLocation. Aceasta va întoarce o structură de tipul IO_STACK_LOCATION, ce conţine următoarele câmpuri:

- `MajorFunction`, care identifică tipul operaţiei (`open`, `read`, `write`, `close`, etc.)
- `Parameters`, o uniune cu parametri pentru diversele tipuri de cereri (`open`, `read`, `write`, `close`, etc.); parametrii includ numărul de octeţi de citit, poziţia, etc.:
  - `Read`, structură ce conţine parametrii pentru o operaţie read:

        - `ByteOffset`, offset-ul de la care se realizează citirea
        - `Length`, numărul de octeţi citiţi
    - `Write`, structură ce conţine parametrii pentru o operaţie write:
        - `ByteOffset`, offset-ul la care se realizează scrierea
        - `Length`, numarul de octeţi scrişi
    - `DeviceIoControl`, structură ce conţine parametrii pentru o operaţie ioctl:
        - `IoControlCode`, codul de control pentru ioctl
        - `InputBufferLength`, dimensiunea buffer-ului de intrare
        - `OutputBufferLength`, dimensiunea buffer-ului de ieşire
- `DeviceObject`, obiectul ce identifică device-ul pe care se face operaţia
- `FileObject`, obiectul ce identifică fişierul pe care se face operaţia

Înregistrarea şi deînregistrarea dispozitivelor

În rutina DriverEntry, în afară de iniţializări, driverul ar trebui să detecteze dispozitivele fizice prezente în sistem şi să le anunţe acestuia cu ajutorul funcţiei IoCreateDevice, funcţie ce are următoarea signatură:

```
NTSTATUS IoCreateDevice(PDRIVER_OBJECT DriverObject,

                        ULONG DeviceExtensionSize,
                        PUNICODE_STRING DeviceName,
                        DEVICE_TYPE DeviceType,
                        LONG DeviceCharacteristics,
                        BOOLEAN Exclusive,
                        PDEVICE_OBJECT *DeviceObject);
```

Numele dispozitivului (DeviceName) trebuie să fie de forma "\Device\MyDevice". Tipul (DeviceType) poate fi unul din cele definite de Microsoft, sau poate fi definit de producător (pentru (cele mai multe) nume de constante consultaţi DDK). Caracteristicile dispozitivului (DeviceCharacteristics) sunt şi ele din cele mai diverse aşa că recomandăm, de asemenea, consultarea DDK. Flagul Exclusive indică modul de acces la dispozitiv: dacă este setat pe TRUE atunci dispozitivul nu va putea fi deschis simultan de mai multe thread-uri. În DeviceObject se întoarce un pointer către noul obiect de tip device alocat.

Odată cu alocarea obiectului se alocă şi spaţiu pentru datele private, de dimensiunea specificată în DeviceExtensionSize. Imediat după crearea obiectului de tip device, device driverul va iniţializa câmpul DeviceExtension cu informaţiile necesare. Ele pot fi apoi accesate din acelaşi câmp, mai târziu, în rutinele de dispatch.

Funcţii de dispatch

Pentru ca sistemul să ştie ce funcţii să apeleze pentru a trata o cerere din user space, device driverul trebuie să înregistreze una sau mai multe funcţii de dispatch. Funcţiile de dispatch au următoarea signatură:

```
NTSTATUS MyOpen(PDEVICE_OBJECT device, IRP *irp);
```

```
NTSTATUS MyRead(PDEVICE_OBJECT device, IRP *irp);
NTSTATUS MyWrite(PDEVICE_OBJECT device, IRP *irp);
NTSTATUS MyDeviceIoControl(PDEVICE_OBJECT device, IRP *irp);
NTSTATUS MyClose(PDEVICE_OBJECT device, IRP *irp);
NTSTATUS MyCleanup(PDEVICE_OBJECT device, IRP *irp);
```

Înregistrarea funcţiilor de dispatch se face în rutina DriverEntry prin setarea intrărilor IRP_MJ_CREATE, IRP_MJ_READ, etc. din vectorul MajorFunction, vector care se găseşte în obiectul ce identifică driverul:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT  driver, PUNICODE_STRING  registry) {
   ...
   driver->DriverUnload = DriverUnload;
   driver->MajorFunction[ IRP_MJ_CREATE ] = MyOpen;
   driver->MajorFunction[ IRP_MJ_READ ]= MyRead;
   driver->MajorFunction[ IRP_MJ_WRITE ] = MyWrite;
   driver->MajorFunction[ IRP_MJ_DEVICE_CONTROL ]= MyDeviceIoControl;
   driver->MajorFunction[ IRP_MJ_CLEANUP ] = MyCleanup;
   driver->MajorFunction[ IRP_MJ_CLOSE ] = MyClose;
   ...
}
```

Open şi Close

Funcţiile asociate cererilor de tip IRP_MJ_CREATE, IRP_MJ_CLOSE se vor apela la deschiderea, respectiv închiderea dispozitivului. Un dispozitiv, in terminologia unui device driver, este un obiect ce identifică un dispozitiv fizic (exemplu: portul serial COM1, portul serial COM2, portul paralel LPT1). Un driver poate gestiona mai multe dispozitive folosind, în general, aceleaşi funcţii de dispatch. Revenind la cele două rutine, acestea sunt, în general, folosite de device driver pentru a iniţializa hardware-ul, a aloca buffere şi a iniţia alte acţiuni administrative.

Read şi Write

Funcţiile asociate cererilor de tip IRP_MJ_READ şi IRP_MJ_WRITE trebuie să trateze cererile de citire, respectiv scriere. În aceste rutine, device driverele în general pregătesc hardware-ul şi pornesc operaţiile de citire sau scriere. Locaţia şi numărul de octeţi de scris sau citit se află din IO_STACK_LOCATION, din Parameters.Write.ByteOffset, Parameters.Write.Length, Parameters.Read.ByteOffset, Parameters.ReadLength.

Ioctl

Funcţia asociată cu IRP_MJ_DEVICE_CONTROL se foloseşte la apeluri de gen DeviceIoControl (echivalentul ioctl din Unix). Codul (sub)operaţiei, dimensiunea bufferului de intrare, dimensiunea bufferului de ieşire, se află din IO_STACK_LOCATION, din Parameters.DeviceIoControl.IoControlCode, Parameters.DeviceIoControl.InputBufferLength, Parameters.DeviceIoControl.OutputBufferLength. Bufferele de intrare sau ieşire se pot accesa din IRP (în funcţie de modul de transfer din user-space în kernel-space).

Codul operaţiei este dat de un număr pe 32 de biţi, în care sunt codificate informaţii despre apelul ioctl. Funcţia CTL_CODE oferă un mecanism simplu de generare a acestor coduri:

CTL_CODE(DeviceType, ControlCode, TransferType, RequiredAccess);

Tipul de transfer (TransferType) specifică tipul de acces la spaţiul de adresa al procesului (METHOD_BUFFERED, METHOD_IN_DIRECT, METHOD_OUT_DIRECT, METHOD_NEITHER). După cum s-a specificat mai sus, accesul este setat în momentul iniţializării dispozitivului, la setarea flag-urilor acestuia. În cazul funcţiei DeviceIoControl, accesul este codificat direct în codul ioctl şi depinde de parametrul TransferType (astfel tipul de acces la spaţiul de adresa al procesului depinde de codul ioctl, şi nu de flag-urile device-ului).

Cleanup

Funcţia asociată cu IRP_MJ_CLEANUP se apelează atunci când un proces renunţă la o cerere (fie că procesul renunţă, fie că, spre exemplu, procesul s-a terminat şi sistemul de operare renunţă la cerere). În această rutină device driverele trebuie să termine toate IRP-urile asociate cu FileObject-ul din IRP-ul primit ca parametru (al funcţiei de tratare IRP_MJ_CLEANUP).

Sincronizare - evenimente

Pentru realizarea sincronizării între thread-uri, există obiecte de sincronizare (KEVENT, KSEMAPHORE, KMUTEX, KTIMER, KTHREAD). În orice moment, aceste obiecte se pot afla în una din stările **signaled** sau **not-signaled**. Un thread poate aştepta ca un astfel de obiect sa ajungă în starea signaled printr-un apel al funcţiei KeWaitForSingleObject:

NTSTATUS KeWaitForSingleObject(PVOID Object, KWAIT_REASON WaitReason,
KPROCESSOR_MODE WaitMode, BOOLEAN Alertable,
PLARGE_INTEGER Timeout);

Un eveniment kernel este reprezentat de KEVENT şi poate fi de două tipuri: eveniment de notificare (NotificationEvent) sau de sincronizare (SynchronizationEvent). În cazul unui eveniment de notificare, când acesta trece în starea **signaled** rămâne în această stare până când este resetat în mod explicit. Mai mult, toate thread-urile care aşteaptă la acest eveniment sunt eliberate când evenimentul trece în starea signaled. Un eveniment de sincronizare trece automat în starea not-signaled în momentul în care un thread care aşteptă la eveniment este eliberat.

Bibliografie

http://cs.pub.ro/~pso/index.php

Interfața driver nucleu la Linux – Petrache Ion

Magistralele sistemului

Pentru ca un calculator să funcţioneze, trebuie implementate căi de comunicaţie între CPU,RAM, şi diversele dispozitivele de I/E . Aceste căi de comunicaţie portă numele de magistrală a sistemului.De fapt ceea ce poartă numele generic de magistrală, implică trei tipuri specializate de magistrale şi acestea sunt:

- magistrala de date: un grup de linii de comunicaţii care transferă datele paralel.

- magistrala de adrese: un grup de linii de comunicaţii care transferă o adresa paralel.

- magistrala de control: un grup de linii de comunicaţii care transmit informatii de control spre circuitele conectate. De exemplu magistrala de adrese se foloseste pentru a specifica ce tip de transfer de date poate să fie posibil: între processor şi RAM sau între procesor şi dispozitivele I/E.

-magistrala dispozitivelor de I/E: conectează procesorul la dispozitivele de I/E. Magistrala dispozitivelor de I/E este conectată la fiecare dispozitiv printr-o serie de componente hardware printre care porturile I/E, interfeţe, şi dispozitive de control.

Spațiul de adrese

Spațiul de adrese defineste o gamă de adrese discrete care corespund unei adrese fizice sau virtuale.O adresă de memorie defineste o locație fizică din memoria calculatorului unde poate fi informația gasită.La Linux spațiul de adrese se imparte în: spațiul de adrese virtuale al nucleului și spațiul de adrese virtuale al utilizatorului. Spațiul de adrese al nucleului este locația de memorie virtuală unde se gasesc firele de aplicație ale nucleului.Spațiul de adrese al utilizatorului este propiu unui proces ceea ce inseamnă că este mapat de către procesul însuși.În schimb spațiul de adrese al nucleul este comun tuturor proceselor.Spatiul de adrese al nucleului ocupă partea de sus a spațiului de adrese.În timplu execuției unui proces în modul utilizator doar spațiul de adrese al utilizatorului este accesibil.Încercarea de a scrie în spațiul de adese al nucleului va genera erori.În timpul execuției în modul nucleul atît spatiul de adrese al utilizatorului cât și cel al nucleului sunt accesibile.

Porturile de I/E

Fiecare dispozitiv care este conectat la magistrala I/E are setul său de adrese. Porturile de I/E pot fi mapate in spațiul de adrese fizice. Astfel un port de I/E poate fi tratat ca o locație de memorie. Porturile de I/E pot fi accesate cu funcții scrise in limbaj de asamblare şi implementate in nucleul sitemului de operare (inb(),outb(),etc.).

Nu așa de simplu precum accesarea porturilor este detectarea porturilor care au fost asignate unor dispozitive. Nucleul are un tabel cu resursele alocate unui dispozitiv. Toate aceste resurse sunt stocate intr-o structură de date de tip arbore.

Interfețele de I/E

Controlorul de interfață este un dispozitiv hardware inserat între grupul de porturi de I/E şi dispozitivul respectiv. Acesta se comportă ca un interpretor care transformă valorile de

la porturile I/E in comenzi şi date pentru dispozitiv. În sens invers, el detectează schimbările de stare ale dispozitivului şi reînoieste porturile de I/E care joacă rolul de registru de stare. Circuitul poate fi conectat la un controlor de intreruperi programabil cu ajutorul unei cereri de întrerupere (IRQ). Există doua tipuri de interfețe:

- interfete de I/E obișnuite: proiectate special pentru un anumit tip de dispozitiv hardware (interfață pentru tastatură, interfață grafică etc.).

- interfețe de I/E generale: folosite pentru a conecta diferite dispozitive hardware (porturile seriale, porturile paralele, USB).

Controlorul dispozitivelor

Un dispozitiv hardware complex este condus de un controlor pentru a implementa complexitatea operațiilor pe care le poate pune la dispoziție. Controlorul dispozitivelor are doua roluri importante:

- intrepretează comenzile de nivel înalt primite de la interfețele de I/E şi forțează dispozitivele să execute un set de acțiuni prin trimiterea de semnale electrice la el.

- Convertește şi interpretează semnalele electrice primite de la dispozitive şi modifică valoarea registrului de stare.

Un exemplu tipic de controlor este controlorul de disc care primeste comenzi de nivel înalt ca de exemplu "scrie un bloc de date" de la microprocesor şi le cconvertește în instrucțiuni ca poziționarea capului de scriere/citire al discului.

Accesul direct la memorie - DMA

Accesul direct la memorie este o funcție a calculatoarelor moderne care permite unui dispozitiv hardware din calculator să acceseze direct memoria sistemului independent de microprocesor.Exemple de dispozitive care folosesc DMA –ul sunt : placa grafica,placa de sunet, placa de rețea.Calculatoarele care folosesc această funcție pot transfera date de la memorie sau pot scrie în memorie fară a ocupa procesorul.Acesta fiind liber, poate efectua alte instrucțiuni.

Toate PC –urile includ un procesor auxiliar numit DMAC (Direct Memory Access Controlor), care poate fi pus sa transfere date intre RAM şi dispozitivele I/E. După ce este activat de către microporcesor acesta poate efectua transferuri de date de unul singur fara intervenția micropocesorului. Când datele de transferat au fost terminate DMAC –ul inițiază o cerere de întrerupere. DMAC –ul este folosit de dispozitive care transferă cantități mari de informație foarte încet.

Nivele de suport ale nucleului

Nucleul sistemului de operare Linux nu are suport pentru toate dispozitivele de I/E.De fapt sunt 3 nivele de suport pentru un dispozitiv hardware.Acestea sunt:

1. Fară suport – programele aplicatie interactionează direct cu portul de I/E al dispozitivului folosind intrucțiuni de intrate ieşire ale limbajului de asamblare.

2. Suport minim – nucleul nu recunoaste dispozitivul hardware, dar recunoaste interfata sa de I/E. Programele utilizatorilor sunt capabile să trateze interfata ca un dispozitiv secvential capabil de citire/scriere a secventelor de caractere.

3. Suport extins – nucleul recunoaste dispozitivul hardware şi operează cu interfața de I/E.

Drivere

Fiecare driver din modelul driverelor de dispozitiv este descris de un obiect device_driver.Acest obiect include patru metode pentru a controla sistemul plug and play și sistemul de gestiune al energiei.Metoda probe este invocată atunci când un dirver de dispozitiv descopera un dispozitiv care nu poate fi gestionat de către driver.Funcția corespunzătoare trebuie să controleze harware –ul și să efectueze noi verificări.Metoda remove este invocată atunci când un dispozitiv ”hot-pluggable” este sters.Metodele shutdown,suspend,resume sunt invocate atunci când nucleul își schimbă starea.Funcția driver_register() inserează un nou obiect device_driver in modelul de drivere dispozitiv și automat creează un nou director pentru el în sistemul de fișiere sysfs.Funcția driver_unregister() elimină un driver din modelul driverelor dispozitiv.Obiectul device_driver este static inclus intr-un descriptor. Obiectul device_driver este static inclus intr-un descriptor.De exemplu driverele pci sunt descrise in structura de date pci_driver.

Fisierele dispozitivelor

Sistemul de operare Linux este bazat pe noțiunea de fișier care este doar un conținut structurat al unei secvențe de biti.Din acest punct de vedere dispozitivele I/O sunt tratare ca fisiere speciale numite fisierele dispozitivelor.Astfel se folosesc aceleași apeluri de sistem ca la fisierele obișnuite. De exemplu acelaşi apel de sistem write ( ) poate fi folosit pentru a scrie intr-un fişier obisnuit cât şi pentru a trimite la imprimantă.

Având în vedere aceste caracteristici driverele dispozitivelor pot fi de două feluri: de tip bloc si de tip caracter.Diferența dintre ele nu este foarte evidentă dar se pot spune următoarele :

- Datele unui dispozitiv bloc pot fi accesate aleator şi timpul necesar pentru a transfera aceste date este mic din punctul de vedere al utilizatorului. Exemple tipice de astfel de dispozitive sunt: hard discul, CD-ROM-ul,DVD- ul.
- Datele unui dispozitiv de tip caracter nu pot fi accesare aleator.

Fişierele dispozitivelor au fost folostie încă de la la inceputurile sistemelor de operare Unix.Un fisier al unui dispozitiv este un fişier real stocat in sistemul de fişiere.I node ul sau nu trebuie totuşi să includă pointeri către date de pe disc.I node-ul trebuie să includă un identificator al dispozitivului hardware.

Uzual acest identificator constă în tipul de fisier dispozitiv (caracter sau bloc) și o pereche de numere. Primul numar se numeste numar major care identifică tipul dispozitivului.Toate fisierele dispozitivelor identificate de acelasi numar major si de acelasi tip impart acelaşi set de operații.Al doilea număr, numărul minor, identifică un dispozitiv specific dintr-un grup de dispozitive cu acelasi numar major.De exemplu un grup de discuri gestionate de către acelaşi controlor au acelaşi numar major dar au numere minore ddiferite.

Apelul de sistem mknod () este folosit pentru a creea fisiere dispozitiv.Primește numele fisierului, tipul său și numarul major si minor ca parametrii.
In mod normal un fisier dispozitiv este asociat unui dispozitiv hardware sau sau unei portiuni de dispozitiv (o partiție a uni disc). In unele cazuri un fisier de dispozitiv nu este asociat unui hardware real ci reprezintă un dispozitiv logic fictiv. De exemplu dispozitivul /dev/null corespunde unei găuri negre. Toate datele trimise la ea sunt aruncate și fisierul arată ca și cum ar fi gol.Din punctul de vedere al nucleului numele dispozitivului nu este relevant.

Bibliografie

Understanding the Linux Kernel, 3rd Edition - Daniel P. Bovet, Marco Cesati

Linux Kernel Internals - Tigran Aivazian

Gestionarul plug and paly la Windows – Ruse Sorin

Ce este plug and play?

Tehnologia plug and play se ocupă cu descoperirea si configurarea automată a dispozitivelor fizice, ea da posibilitatea unui utilizator sa introduca un dispozitiv in calculator si acesta sa fie recunoscut de catre computer. Utilizatorul nu trebuie sa-i comunice nimic calculatorului, spre deosebire de sistemele mai vechi de operare unde utilizatorul era nevoit sa dea detalii precise calculatorului despre dispozitivul adaugat.

Pe langa cei de la Microsoft au folosit un astfel de sistm si cei de la Machintosh, dar odata cu aparitia si extinderea mare a Plug and Play-ului de Windows, acesta a devenit standardizat.

Evolutia Plug and Play

Plug and paly a fost prima oara suportat de sisitemul de operare Widows 95, dar de atunci a suferit schimbari dramatice. Evolutia se datoreaza in mare masura initiativei de design “OnNow” care cauta sa definesca o abordare globala a problemelor legate de configurarea sisitemului si a dispozitivelor cu care acesta interactioneaza. Un prim rezultat al acestei initiative este interfata ACPI “Advanced Configuration and Power Interface” care defineste un nou sistem al placii de retea si al interfetei BIOS-ului . Prin aceste schimbari Plug and play-ul capata noi capabilitati cum ar fi: managementul energiei si noi posibilitati de configurare, totate sub controloul total al sistemului de operare. Incepand cu Windows 2000 sistemul care pana atunci purata numele de Plug and Play se va numi **W**indows **D**river **M**model (WDM).

Plug and Play în Linux

Sistemul de operare Linux nu a fost de la început un sistem plug and play, dar în prezent aceste probleme au fost rezolvate. La pornirea sistemului, BIOS-ul realizează plug and play şi configurează dispozitivele din sistem. Linux, la pornire poate reconfigura aceste setări sau le poate accepta.

În Linux fiecare driver realizează propria configurare a dispozitivelor. Aceasta era un lucru dificil, dar acum s-au pus la dispoziţia driverelor mecanisme din kernel pentru a realiza

mare parte din aceste opertaţii. Deci, într-un fel, si acum tot driverele realizează configurarea dispozitivelor, numai că realizează acest lucru prin instruirea kernel-ului asupra operaţiilor pe care trebuie să le execute.

Tehnologia plug and play în Linux se bazează în principal pe **Linux Device Model**, care include sistemul de fişiere sysfs şi hotplug, alături de mecanisme din user-mode, cum ar fi udev.

Linux Device Model

Înainte de versiunea 2.6, kernel-ul nu dispunea de un model unificat prin care să se obţină informaţii despre acesta. Din acest motiv s-a realizat un model pentru dispozitivele din Linux, **Linux Device Model**.

Scopul principal al acestui modelul este de a menţine structuri de date interne care să reflecte starea şi structura sistemului. Astfel de infomaţii includ ce dispozitive există în sistem, în ce stare se află din punct de vedere al managementului consumului (power management), la ce magistrală sunt ataşate, ce drivere au asociate, alături de structura magistralelor, dispozitivelor, driverelor din sistem.

Pentru a menţine aceste informaţii, kernel-ul foloseşte următoarele entităţi:

- **dispozitiv** - un dispozitiv fizic care este ataşat unei magistrale
- **driver** – o entitate software care poate fi asociată unui dispozitiv si execută operaţii cu acesta
- **magistrală** (**bus**) – un dispozitiv la care se pot ataşa alte dispozitive
- **clasă** – un tip de dispozitive care au o comportare similară; există o clasă pentru discuri, partiţii, porturi seriale, etc.
- **subsistem** – o vedere asupra structurii sistemului; subsistemele din kernel includ dispozitive (`devices` - o vedere ierarhică asupra tuturor dispozitivelor din sistem), magistrale (`bus` - o vedere a dispozitivelor în funcţie de cum sunt ataşate la magistrale), clase, etc.

Sysfs

Kernel-ul oferă o reprezentare a modelului său în userspace prin intermediul sistemului virtual de fişiere **sysfs**. Acesta este de obicei montat în directorul /sys şi conţine următoarele subdirectoare:

- `block` - toate dispozitivele de tip bloc disponibile în sistem (discuri, partiţii)
- `bus` - tipuri de magistrale la care se conectează dispozitivele fizice (pci, ide, usb)
- `class` - clase de drivere care sunt disponibile în sistem (net, sound, usb)
- `devices` - structura ierarhica a dispozitivelor conectate în sistem
- `firmware` - informaţii obţinute de la firmware-ul sistemului (ACPI)
- `module` - lista modulelor încarcate la momentul curent

După cum se poate observa, există o corespondenţă între structurile de date din kernel în cadrul modelului descris şi subdirectoarele din sistemul virtual de fişiere **sysfs**. Deşi această asemănare poate duce la confundarea celor două concepte, ele sunt diferite. Modelul pentru dispozitive în kernel poate funcţiona şi fara sistemul de fisiere **sysfs**, dar reciproca nu este adevărată.

Informaţia din **sysfs** se găseşte in fişiere ce conţin câte un atribut. Câteva atribute standard (reprezentate de fişiere sau directoare cu acelaşi nume) sunt următoarele:

- `dev` - identificatorul major şi minor al dispozitivului; acesta poate fi folosit pentru a crea automat intrările în directorul `/dev`
- `device` - o legătură simbolică spre directorul ce conţine dispozitive; acesta poate fi folosit pentru a descoperi dispozitivele hardware care oferă un anumit serviciu (spre exemplu dispozitivul PCI al plăcii de reţea `eth0`)
- `driver` - o legătură simbolică spre directorul ce conţine drivere (care se află în `/sys/bus/*/drivers`)

Sunt disponibile şi alte atribute, în funcţie de magistrala şi driverul folosit.

Diferente fata de Linux

Spre deosebire de Linux, în Windows (incepand cu Win 95) kernel-ul de ocupă de configurarea dispozitivelor şi în acest sens este cu adevărat un sistem plug and play. Dispozitivele sunt descoperite automat în timpul secvenţei de boot sau la inserare (hotplug), determinând încărcarea automată a driver-elor corespunzătoare.

Comparativ cu Linux, în Windows (versiunile de dupa Windows 95) se aplică un algoritm de rezolvare a conflictelor ce apar la alocarea de resurse (rebalansare) . În modelul anterior din Windows (legacy drivers), era necesară încarcarea explicită a driver-elor şi iniţializarea dispozitivelor asociate acestuia la încarcare . Folosind plug and play, acest lucru nu mai este necesar, întrucaât sistemul de operare se ocupă de aceste operaţii (la detectarea unui diuspozitiv se va apela o metodă specială a driver-ului care va adăuga dispozitivul).

În Windows (incepand cu Win2000), implementarea plug and play are mai multe componente software:

- managerul plug and play – are o parte în user-mode şi o parte în kernel-mode şi se ocupă cu detectarea şi configurarea dispozitivelor fizice
- managerul de consum (power manager) – se ocupă cu managementul consumului (pentru a reduce consumul de energie al sistemului, anumite dispozitive pot fi eliminate temporar din sistrem dacă nu sunt folosite o perioadă lungă de timp)
- regiştrii (registry) – conţin o bază de date a componente lor harware şi software instalate în sistem şi sunt folosiţi la identificarea şi localizarea resurselor de către dispozitive
- fişierele .inf (INF file)– descriu un dispozitiv, fiind necesar câte un astfel de fişier pentru fiecare dispozitiv la instalarea driver-ului; fiecare pereche dispozitiv/driver trebuie să aibă un astfel de fişier
- drivere plug and play – deşi există drivere care folosesc doar parţial arhitectura plug and play, se recomandă implementarea de drivere WDM (care respectă modelul Windows Driver Model) şi care suportă complet arhitectura plug and play

Componentele sistemului Plug and play (PnP)

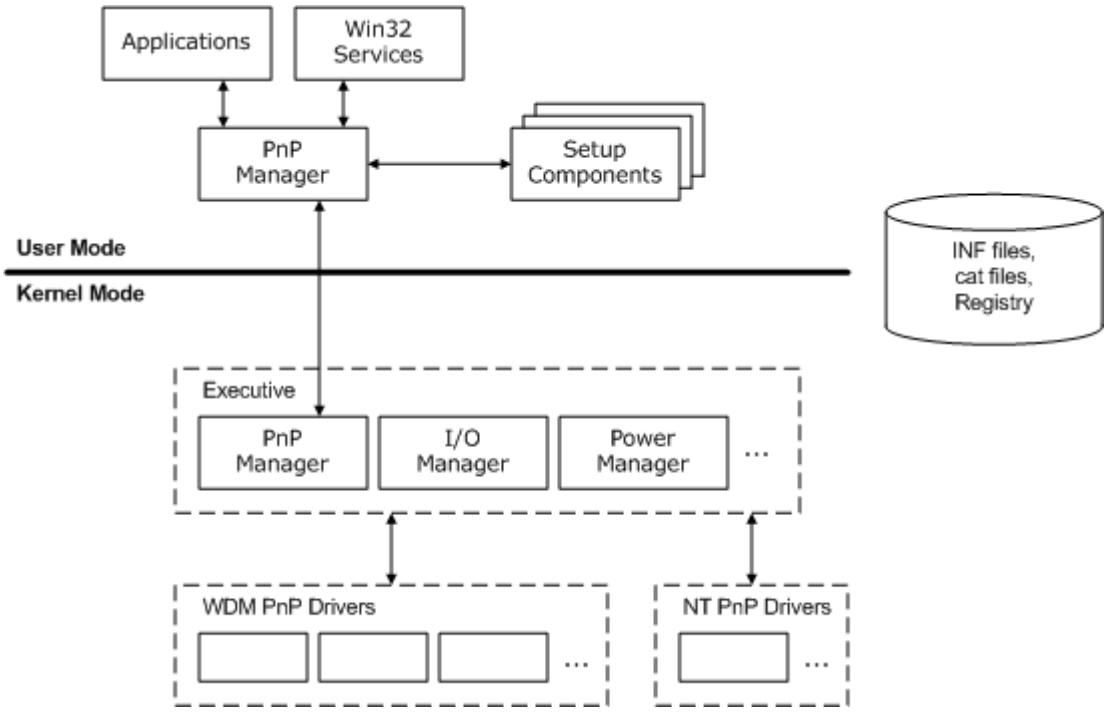

Figura 1

Windows Driver Model (WDM)

Windows Driver Model este un model unificat, ce permitee scrierea de drivere al căror cod sursă este compatibil pentru toate platformele Windiows. Un driver WDM (care respectă modelul Windows Driver Model) are următoarele caracteeristici:

- trebuie să aibă unul din tipurile de drivere WDM (bus driver, function driver, filter driver) şi să creeze dispozitive cu unul din tipurile WDM (Physical Device Object, Functional Device Object, Filter Device Object)
- trebuie să suporte plugg and play
- trebuie să suporte managementul consumului (power management)
- trebuie să suporte WMI (Windows Management Instrumentation); WMI este un mecanism prin care kernel-ul pune la dispozitia aplicaţiilor din user-mode informaţii (permite publicarea informaţiilor, configurarea dispozitivelor, un mecanism de notificări, logaarea evenimentelor, etc.)

Modelul WDM organizează driverele şi dispozitivele într-o stivă.

Astfel, driver-ele sunt împarţite în trei categorii:

- bus drivers – drivere asociate magistralelor din sistem; este obligatoriu să existe un astfel de driver pentru fiecare tip de magistrala din sistem; pot avea alte dispozitive conectate la magistrală; se află la cel mai jos nivel în stiva de drivere
- function drivers – drivere pentru un dispozitiv individual; se află deasupra driverelor pentru magistrală în stiva de drivere
- filter drivers – drivere care filtrează cererile pentru un dispozitiv, o clasă de dispozitive sau o magistrală; se pot afla deasupra unui driver de magistrală (modifică în acest caz comportamentul dispozitivului) sau deasupra unui driver funcţional (adaugă funcţionalităţi suplimentare)

În strânsă legătură cu tipurile de drivere, WDM defineşte şi tipul de obiecte ce descriu dispozitivele asociate fiecărui driver din stivă (Device_Object):

- Physical Device Object (PDO) - reprezintă un dispozitiv pe o magistrală pentru un driver dw magistrală; exista câte un astfel de obiect pentru fiecare tip de dispozitiv fizic şi este responsabil cu controlul la nivel low-level al dispozitivului
- Functional Device Object (FDO) - reprezintă un dispozitiv pentru un driver funcţional; există câte un astfel de obiect pentru fiecare funcţie logică sau abstractă care este o ferită nivelului superior
- Filter Device Object (filter DO) - reprezintă un dispozitiv pentru un driver de tip filtru; pot exista filtre atât pentru obiectele dispozitiv de tip fizic cât şi pentru cele de tip funcţional

Funcţionarea unui driver plug and play şi stările unui dispozitiv

Modelul WDM este o extensie a modelului anterior, NT. Astfel, DriverEntry rămâne funcţia de iniţializare a drverului, numai ca după cum s-a precizat, nu se vor mai iniţializa dispozitivele asociate aici. Pentru aceasta, va exista o altă funcţie *AddDevice*, care va fi apelată de Plug and Play Manager pentru fiecare dispozitiv asaociat. Operaţiile legate de dispozitiv sunt iniţiate de Plug and Play Manager prin transmiterea unui mesaj *IRP_MJ_PNP* (MajorFunction). Pentru a diferenţia operaţiile efectuate asupra dispozitivului se foloseşte codul minor (MinorFunction). Acest cod poate avea una din următoarele valori:

- `IRP_MN_START_DEVICE` pentru iniţializarea sau reiniţializarea dispozitivului cu resursele specificate
- `IRP_MN_QUERY_STOP_DEVICE` pentru a verifica dacă dispozitivul poate fi oprit in vederea rebalansării resurselor
- `IRP_MN_STOP_DEVICE` pentru a opri dispozitivul (pentru a fi repornit sau eliminat)
- `IRP_MN_CANCEL_STOP_DEVICE` pentru a informa că nu se va opri dispozitivul, dupa o operatie `IRP_MN_QUERY_STOP_DEVICE`
- `IRP_MN_QUERY_REMOVE_DEVICE` pentru a verifica dacă dispozitivul poate fi eliminat din sistem
- `IRP_MN_REMOVE_DEVICE` pentru a elimina dispozitivul din sistem (operaţiile care deiniţializează resursele iniţializate în funcţia `AddDevice`)

- *IRP_MN_CANCEL_REMOVE_DEVICE* pentru a informa că nu se va elimina dispozitivul din sistem, dupa o operatie *IRP_MN_QUERY_REMOVE_DEVICE*
- *IRP_MN_SURPRISE_REMOVAL* pentru a informa că dispozitivul a fost eliminat din sistem fără notificare in prealabil

Aceste coduri sunt valabile pentru toate driverele WDM. Pentru anumite tipuri de drivere (spre exemplu pentru driverele de tip magistrală sau pentru cele care au asociat un device de tip fizic şi se ocupă cu managementul controlului la nivel low-level) sunt definite coduri suplimentare (spre exemplu, pentru aflarea capabilităţilor unui dispozitiv, pentru aflarea interfeţei acestuia, aflarea resurselor conectate la o magistrală, etc.).

După cum se poate observa din operaţiile de mai sus, un dispozitiv trece prin diferite stări, în timp ce este configurat, pornit, eventual oprit pentru rebalansarea resurselor şi posibil eliminat. Aceste stări se pot împărţi în două categorii: stările prin care dispozitivul trece atunci când este adăugat în sistem şi stările prin care trece după ce este adăugat.

Iniţializarea driver-ului (`DriverEntry`)

La fel ca şi în cazul modelului NT, driverele WDM se iniţializează în rutina *DriverEntry*. Spre deosebire de aceasta, iniţializează dispozitivele fizice (nu se mai apelează *IoCreateDevice*), ci doar se iniţializează funcţiile driver-ului.

Funcţiile ce trebuiesc iniţializate reprezintă funcţiile de dispatch pentru operaţii de deschidere, scriere, citire, control, închidere dispozitiv. Pe lângă acestea, mai trebuie iniţializată funcţia pentru iniţializarea dispozitivelor (*AddDevice*) şi funcţia pentru mesajele plug and play (*IRP_MJ_PNP*).

O funcţie *DriverEntry* pentru un driver plug and play va arăta în modul următor:

```
NTSTATUS DriverEntry(PDRIVER_OBJECT driver, PUNICODE_STRING registry) {
    driver->DriverUnload = DriverUnload;
    driver->MajorFunction[ IRP_MJ_CREATE ] = Open;
    driver->MajorFunction[ IRP_MJ_READ ]= Read;
    driver->MajorFunction[ IRP_MJ_WRITE ] = Write;
    driver->MajorFunction[ IRP_MJ_CLEANUP ] = Cleanup;
    driver->MajorFunction[ IRP_MJ_CLOSE ] = Close;

    driver->DriverExtension->AddDevice = AddDevice;        /* PNP */
    driver->MajorFunction[ IRP_MJ_PNP ] = DispatchPnp;     /* PNP */
}
```

Iniţializarea dispozitivului (`AddDevice`)

După cum s-a observat mai sus, pentru iniţializarea dispozitivului există o funcţie AddDevice, care va fi apelată de Plug and Play Manager în momentul descoperirii dispozitivului. Această funcţie va prelua sarcina funcţiei DriverEntry din modelul NT şi va iniţializa dispozitivul.

Prototipul acestei funcţii este următorul:

```
NTSTATUS AddDevice(
   IN PDRIVER_OBJECT  DriverObject,
   IN PDEVICE_OBJECT  PhysicalDeviceObject
);
```

, unde DriverObject este un pointer către obiectul asociat driver-ului, iar PhysicalDeviceObject este un pointer către dispozitivul fizic (PDO), creat de un driver de nivel mai jos.

La iniţializarea dispozitivului se creează o legătură internă pentru dispozitiv printr-un apel al funcţiei IoCreateDevice, se creeaza o legătură simbolică pentru userspace printr-un apel al funcţiei IoCreateSymbolicLink şi se iniţializează datele private ale dispozitivului.

Pe lângă aceste operaţii, funcţia AddDevice mai trebuie să adauge obiectul asociat driverului în stiva de obiecte. Această operaţie se realizează printr-un apel al funcţiei:

```
PDEVICE_OBJECT IoAttachDeviceToDeviceStack (
   IN PDEVICE_OBJECT  SourceDevice,
   IN PDEVICE_OBJECT  TargetDevice
);
```

, unde SourceDevice este un pointer către obiectul asociat dispozitivului care apelează funcţia (noul vârf al stivei după apel), iar TargetDevice este un pointer către obiectul asociat altui dispozitiv (pointer către PDO-ul stivei). Funcţia întoarce un pointer către vechiul vârf al stivei, deci un pointer către dispozitivul situat sub dispozitivul apelant în stiva.

Întrucât obiectul a fost iniţializat în afara funcţiei DriverEntry, este necesară resetarea bit-ului pentru iniţializarea dispozitivului:

```
device->Flags &= ~DO_DEVICE_INITIALIZING;
```

Funcţie de dispatch pentru drivere WDM (`IRP_MJ_PNP`)

După cum s-a observat, funcţia AddDevice doar iniţializează dispozitivul şi datele sale private, fără a realiza operaţii legate de dispozitivul fizic. Astfel, mai trebuie rezervat, iniţializat si configurat dispozitivul fizic. În acest scop, există o nouă funcţie de dispatch, IRP_MJ_PNP. Un IRP corepunzător este creat la iniţializarea dispozitivului, eliminarea sa sau când se primesc cereri din partea acestuia. Plug and Play Manager-ul va apela funcţia de dispatch corespunzătoare (înregistrată în DeviceEntry), care trebuie să trateze aceste cazuri. Întrucât sunt mai multe operaţii, diferenţierea între acestea se realizează prin codul minor al IRP-ului. Aceste coduri au fost prezentate mai sus, la funcţionarea unui driver plug and plaay şi stările unui dispozitiv.

O astfel de funcţie de dispatch va diferenţia între aceste coduri şi va avea următoarea structură:

```
NTSTATUS DispatchPnp(IN PDEVICE_OBJECT device, IN PIRP irp) {
   PIO_STACK_LOCATION irpStack;
```

```
    irpStack = IoGetCurrentIrpStackLocation(irp);

    switch (irpStack->MinorFunction) {
    case IRP_MN_START_DEVICE:
      return HandleStartDevice(device,irp);
    case IRP_MN_STOP_DEVICE:
      return HandleStopDevice(device,irp);
    case IRP_MN_REMOVE_DEVICE:
      return HandleRemoveDevice(device,irp);
    default:
      return PassDownPnP(device,irp);
    }
  }
```

După cum se poate observa, pentru fiecare din operaţii se apelează o funcţie (ce va fi discutată în continuare), iar în cazul în care codul minor primit nu este suportat de driverul curent, este trimis următorului dispozitiv din stivă.

Transmiterea cererilor plug and play în stiva de dispozitive

Toate cererile plug and play sunt iniţiate de Plug and Play Manager şi sunt transmise driver-ului care se află în vârful stivei de dispozitive. Indiferent ce coduri minore ale IRP-urilor sunt tratate de către un driver, cele care nu sunt tratate de către acesta trebuie trimise mai departe în stiva de dispozitive, la driverele de pe niveluri mai joase, care ar putea trata acele coduri. Această operaţie este necesară, întrucât un driver se bazează pe driverele de pe nivele mai joase pentru realizarea anumitor operaţii (spre exemplu, un driver funcţional – FDO – se bazează pe driverul fizic - PDO). De asemenea, există cereri care interesează toate driverele din stivă (cum ar fi informarea asupra opririi dispozitivului).

Pentru cererile care sunt tratate de driver, trebuie completată informaţia IRP-ului legată de status (`IoStatus`) şi apelată funcţia `IoCompleteRequest`. Pentru a transmite un o cerere plug and play în jos pe stiva, fără a aştepta ca aceste pachete să fie tratate de un driver, se apelează funcţia `IoSkipCurrentIrpStackLocation`, care elimină intrarea pentru stiva driver-ului curent din IRP şi apoi `IoCallDriver` pentru a transmite IRP-ul driver-ului de la nivel inferior. Pointerul către driverul de sub cel curent în stivă a fost obţinut în urma apelului `IoAttachDeviceToDeviceStack` de la iniţializarea dispozitivului ([[#Iniţializarea dispozitivului (`AddDevice`)| Iniţializarea dispozitivului (`AddDevice`)]]).

Funcţia PassDownPnP, care realizează aceste operaţii şi este apelată în funcţia de dispatch de mai sus are următoarea implementare:

```
NTSTATUS PassDownPnP( IN PDEVICE_OBJECT device, IN PIRP irp ) {
  struct my_data * myData = (struct my_data *) device-> DeviceExtension;
  IoSkipCurrentIrpStackLocation(irp);
  return IoCallDriver(myData->pLowerDevice, irp);
}
```

Pornirea dispozitivului (`IRP_MN_START_DEVICE`)

Plug and Play Manager-ul, la boot sau în momentul conectării unui dispozitiv, le identifică şi transmite un IRP cu codul minor IRP_MN_START_DEVICE driver-ului corespunzător. Pe măsură ce identifică dispozitivele, Plug and Play Managerul le atribuie resursele cerute, cu evitarea pe cât posibil a conflictelor. Atunci când transmite driver-ului IRP-ul, îi transmite şi o listă cu resursele asociate dispozitivului fizic, în câmpurile Parameters.StartDevice.AllocatedResourcesTranslated şi Parameters.StartDevice.AllocatedResources ale acestuia. Aceste câmpuri conţin mai multe niveluri de vectori, şi în final structura CM_PARTIAL_RESOURCE_DESCRIPTOR ce descrie resursele, care pot fi de patru tipuri: porturi, întreruperi, memorie şi dma.

Resursele prezentate sunt de doua tipuri: raw (Parameters.StartDevice.AllocatedResources) şi translated (Parameters.StartDevice.AllocatedResourcesTranslated). Resursele raw sunt cele întâlnite în driverele din modelul NT şi pentru care trebuiau realizate operaţii de translatare. Din acest motiv, se vor folosi resursele translatate.

Oprirea dispozitivului (`IRP_MN_STOP_DEVICE`)

La primirea unui IRP cu codul minor IRP_MN_STOP_DEVICE, se vor executa operaţii pentru oprirea dispozitivului. Aceste operaţii sunt complementare celor executate în funcţia HandleStartDevice, la primirea unui IRP cu codul IRP_MN_START_DEVICE. Prin urmare, codul acestei funcţii este dependent de tipul de resurse deţinute de dispozitiv.

Funcţia HandleStopDevice, care realizează aceste operaţii şi este apelată în funcţia de dispatch de mai sus are următoarea implementare:

```
NTSTATUS HandleStopDevice(IN PDEVICE_OBJECT device, IN PIRP irp ) {
   struct my_data * myData = (struct my_data *) device-> DeviceExtension;
   /* IoDisconnectInterrupt(myData->pIntObj ); */
   return PassDownPnP(device, irp);
}
```

Eliminarea dispozitivului (`IRP_MN_REMOVE_DEVICE`)

La primirea unui IRP cu codul minor IRP_MN_REMOVE_DEVICE, se vor executa operaţii pentru eliminarea dispozitivului din sistem. Aceste operaţii sunt complementare celor executate în funcţia AddDevice şi sunt aceleaşi cu cele din DriverUnload, doar că pentru un singur dispozitiv (cel dar ca parametru). Funcţia va deiniţializa resursele, va şterge legăturile pentru dispozitiv şi va transmite IRP-ul în jos pe stivă.

În plus faţă de operaţiile cunoscute, apare deataşarea dispozitivului din stivă, printr-un apel al funcţiei IoDetachDevice.

Funcţia HandleRemoveDevice, care realizează aceste operaţii şi este apelată în funcţia de dispatch de mai sus are următoarea implementare:

```
NTSTATUS HandleRemoveDevice(IN PDEVICE_OBJECT device, IN PIRP irp ) {
   struct my_data * myData = (struct my_data *) device->DeviceExtension;
   /* IoDisconnectInterrupt(myData->pIntObj); */
   IoDeleteSymbolicLink(&myData->linkName);
```

```
    IoDetachDevice(bufferData->pLowerDevice);
    IoDeleteDevice(device);
    return PassDownPnP(device, irp);
}
```

Bibliografie:

1. http://tldp.org/HOWTO/Plug-and-Play-HOWTO.html
2. http://lwn.net/Articles/driver-porting/
3. http://msdn2.microsoft.com/en-us/library/ms798213.aspx
4. http://msdn2.microsoft.com/en-us/library/ms798213.aspx
5. The Windows 2000 Device Driver Book, Second Edition – Chapter 9. Hardware Initialization
6. Programming the Microsoft Windows Driver Model, Second Edition

Figura 1 → http://msdn2.microsoft.com/en-us/library/ms798233.aspx

Module incărcabile în Linux – Mihalea Ionuț

Cand programatorii de sistem vor sa adauge noi functionalitati kernel-ului Linux,sunt pusi in fata unei dileme interesante: sa scrie noul cod pentru a fi compilat ca un modul sau sa adauge in mod static noul cod la kernel?

Ca o regula generala,programatorii de system tind sa implementeze noul cod sub forma unui modul.Deoarece modulele pot fi apelate la cerere,cum vom vedea mai tarziu,kernelul nu trebuie umflat cu sute de programe des folosite.Aproape orice component de nivel inalt a kernelului Linux – sistem de fisiere,driverele componentelor,formatele executabile,si asa mai departe – pot fi compilate ca un modul.

Totusi,anumit cod Linux trebuie sa aiba legatura static,ceea ce inseamna ca acea componenta corespunzatoare trebuie inclusa in kernel,sau sa nu fie compilata deloc.Asta se intampla mai ales cand o componenta are nevoie de modificare a structurii de date.

Ca un exemplu,sa presupunem ca o componenta trebuie sa introduca campuri noi intr-un descriptor de proces.Facand link la un modul nu poate schimba o structura de date deja definita precum task_struct deoarece toate legaturile codului static continua sa vada vechea versiune: coruperi de date vor aparea foarte usor.O solutie partiala a acestei probleme ar fi sa se adauge static noile campuri descriptorului de proces,facandu-le disponibile componentei kernelului,necontand cum a fost linkata. Totusi,daca componenta kernelului nu este folosita,aceste campuri in plus,replicate in fiecare descriptor de proces folosesc memoria fara rost.Daca noua componenta a kernelului creste marimea descriptorului de proces foarte mult,utilizatorul ar avea o performanta a sistemului mai buna adaugand campurile necesare in structura de date doar daca componenta are link static catre kernel.

Ca un al doilea exemplu,consider o componenta a kernelului care trebuie sa inlocuiasca static un cod linkat.Este destul de clar ca nicio astfel de componenta nu poate fi compilata ca modul deoarece kernelul nu poate modifica codul masina existent deja in RAM cand se va linka modulul.De exemplu, nu e posibil sa se faca link la un modul care schimba

modul de alocare a paginilor,deoarece functiile de sistem sunt intotdeauna linkate static catre kernel.

Kernelul are doua task-uri cheie de facut pentru lucra cu modulele.Primul task este sa se asigure ca restul kernelului poate apela simbolurile globale ale modulului,cum ar fi punctul de intrare in functia principala.Un modul trebuie de asemenea sa stie adresele simbolurilor din kernel si din alte module. Asa ca referintele sunt rezolvate o data pentru totdeauna cand modulele sunt linkate.Al doilea task consta in a tine evidenta folosirii modulelor,asa incat niciun modul sa nu fie descarcat atata timp cat un alt modul sau o alta parte a kernelului il foloseste.O simpla numarare a referintelor tine evidenta folosirii fiecarui modul.

Implementarea modulelor

Modulele sunt salvate in sistemul de fisiere ca obiecte fisier de tip ELF.Kernelul considera doar module incarcate in RAM prin programul */sbin/insmod* si pentru fiecare dintre ele aloca o zona de memorie continand urmatoarele dat:

- Un modul obiect
- Un string null care reprezinta numele modulului (fiecare modul ar trebui sa aiba nume unice)
- Codul care implementeaza functiile modulului

Obiectul modul descrie modulul;campurile sale sunt in tabelul 1.O lista simplu linkata colecteaza toate obiectele modul,unde campul *next* a fiecarui modul indica urmatorul element din lista.Primul element din lista este apelat prin variabila *module_list* .Dar, de fapt, primul element din lista este mereu acelasi:este numit *kernel_module* si are referinta catre un modul fictiv care reprezinta un cod kernel linkata in mod static.

Tip	Nume	Descriere
unsigned long	size_of_struct	dimensiunea modululi obiect
struct module *	next	urmatorul element din lista
const char *	name	pointer catre numele modulului
unsigned long	size	dimensiunea modulului
atomic_t	uc.usecount	Numaratorul folosirii modulului
unsigned long	flags	fanioanele modulului
unsigned int	nsyms	numarul simbolurilor exportate
unsigned int	ndeps	numarul modulelor cu referinta
struct module_symbol *	syms	tabel cu simboluri exportate
struct module_ref *	deps	lista modulelor referite
struct module_ref *	refs	lista modulelor cu referinta
int (*)(void)	init	metoda de initializare
void (*)(void)	cleanup	metoda de curatare
struct exception_table_entry	ex_table_start	tabelul inceputului de exceptii

*		
struct exception_table_entry *	ex_table_end	tabelul sfarsitului de exceptii

Marimea totala a memoriei alocata unui modul este continuta in campul *size.*
Fiecare modul are propriul tabel de exceptii.Tabelul include adresele codului corector al modulului,daca exista.Tabelul este copiat in RAM cand modulul este linkat,si adresele sale de inceput si sfarsit sunt tinute in campurile *ex_table_start* si *ex_table_end* .

Numaratorul folosirii modulului

Fiecare modul are un numarator de folosire,tinut in campul *uc.usecount* a modulului obiect corespunzator.Numaratorul este incrementat cand o operatie ca implica functiile modulului este pornita si decrementat cand operatia se termina.Un modul poate fi delinkat cand cand folosirea counterului este nula.

Ca un exemplu,sa presupunem ca sistemul MS-DOS de fisiere a fost compilat ca un modul si modulul a fost linkat la pornirea programului.Initial,folosirea counterului modulului este nula.Daca userul monteaza un floppy-disk MS-DOS,numaratorul este incrementat cu 1.Invers,cand userul demonteaza floppy-diskul,numaratorul este decrementat cu 1.

Simboluri exportate

Cand se face link catre un modul,toate referintele catre simbolurile globale (variabile si functii)din codul obiect al modulului trebuiesc inlocuite cu adresele potrivite.Operatia,care este similara cu cea facuta de linker cand compileaza un program in User Mode,este delegata programului extern */sbin/insmod.*

Un tabel special este folosit de kernel pentru a tine simbolurile care pot fi accesate de module impreuna cu adresele corespunzatoare.Tabelul simbolurilor kernel este continut in sectiunea *_ksymtab* a segmentului de cod kernel,si adresele sale de inceput si sfarsit sunt identificate de doua simboluri produse de compilatorul C: *_start_ _ksymtab* si *_stop_ _ksymtab* .

Doar simbolurile kernelului folosite de anumite module sunt incluse in tabel.Daca un programator are nevoie,intr-un modul,sa acceseze un simbol de kernel care nu este deja exportat, poate adauga macroul *EXPORT_SYMBOL* in fisierul *kernel/ksyms.c* al codului sursa Linux.

Modulele linkate pot de asemenea sa exporte propriile simboluri,pentru ca alte module sa le poata accesa.Tabelul de simboluri de modul este continut in sectiunea *_ksymtab* a segmentului de cod modul.Daca codul sursa al modulului include macro-ul *EXPORT_NO_SYMBOLS* ,niciun simbol din acel modul nu este adaugat in tabel.Pentru a exporta un subset de simboluri din modul,programatorul trebuie sa defineasca *EXPORT_SYMTAB* inainte de a include headerul *include/linux/module.h.* Apoi poate folosi *EXPORT_SYMBOL* pentru a exporta un simbol anume.Daca nici *EXPORT_NO_SYMBOLS,*si nici *EXPORT_SYMTAB* nu apar in codul sursa al modulului,toate simbolurile globale sunt exportate.

Tabelul simbolurilor din sectiunea *_ksymtab* este copiat in zona de memorie cand modulul este linkat,si adresa zonei este tinuta in campul *syms* a modululi obiect.Simbolurile exportate de gernelul linkat static si de toate modulele linkate poate fi obinut citind fisierul */proc/ksyms* sau folosind procedura de apel de sistem *query_module()*.

Linkarea si delinkarea modulelor

Un utilizator poate linka un modul intr-un kernel care ruleaza executand programul extern */sbin/insmod*.Acest program face urmatoarele operatii:

1.Citeste din linia de comanda numele modulului ce va fi linkat.

2.Gaseste fisierul care contine codul obiect al modulului in sistemul de directoare. Fisierul este de obicei intr-un subdirector al directorului */lib/modules*.

3.Calculeaza marimea zonei de memorie necesara pentru a tine codul modulului, numele sau si modulul obiect.

4.Invoca procedura de sistem *create_module()*,dandu-i acesteia numele si marimea noului modul.Rutina de serviciu corespunzatoare *sys_create_module()* face urmatoarele operatii:

a. Verifica daca utilizatorului ii este permis sa faca linkarea modulului (procesul curent trebuie sa aiba capabilitatea CAP_SYS_MODULE).In orice situatie unde cineva adauga functionalitate kernelului,care are acces la toate datele si procesele din sistem,securitatea este vitala.

b. Invoca functia *find_module(),*pentru a scana lista *module_list* ale obiectelor modul si cauta un modul cu numele specificat.Daca este gasit,modulul a fost deja linkat,asa ca procedura de sistem se termina.

c. Invoca *vmalloc()* pentru a aloca zona de memorie pentru noul modul.

d.Initializeaza campurile obiectului modul la inceputul zonei de memorie si copiaza numele modulului sub obiect.

e. Insereaza obiectul modul in lista data de *module_list*.

f. Returneaza adresa de inceput a zonei de memorie alocata modulului.

5. Invoca procedura de apel de sistem *query_module()* cu subcomanda QM_MODULES pentru a lua numele modulelor deja linkate.

6. Invoca procedura de apel de sistem *query_module()* cu subcomanda QM_SYMBOL pentru a obtine tabelul simbolurilor de kernel si tabelele simbolurilor tuturor modulelor care sunt deja linkate.

7.Folosind tabelul de simboluri ale kernelului,tabelele de simboluri ale modulelor si adresa returnata de procedura de apel de sistem *create_module()* si realoca codul obiect inclus in fisierul modulului.

8.Aloca o zona de memorie in spatiul de adrese al modului utilizator si il incarca cu o copie a obiectului modul,numele modulului si codul modulului realocat pentru kernelul care ruleaza.Campurile adreselor obiectului indica codul relocat.Campul *init* este setat adresei relocate a functiei *init_module()* a modulului.

9.Invoca procedura de sistem *init_module()*,trecand-o adresei zonei de memorie a modului utilizator.Rutina de serviciu *sys_init_module()* face urmatoarele operatii:

a.Verifica daca utilizatorului ii este permis sa linkeze modulului.

b.Invoca *find_module()* pentru a gasi modulul obiect respectiv din lista indicata de *module_list.*

c.Suprascrie modulul obiect cu continutul obiectului corespunzator din zona de memorie a modului utilizator.

d. Face o serie de verificari ale adreselor din obiectul modul.

e. Copiaza partea ramasa din zona de memorie a modului utilizator in zona de memorie alocata modulului.

f.Scaneza lista modulelor si initializeaza campurile *ndeps* si *deps* ale obiectului
modul.

g.Seteaza numaratorul de folosire a modulului la 1.

h.Daca e definita,executa metoda *init* a modulului pentru a initializa structurile de date corect.Metoda este implementata de obicei de functia *init_module()* definita in interiorul modulului.

i.Seteaza numaratorul de folosire al modulului la 0 si revine.

10.Elibereaza zona de memorie a modului utilizator si termina.

Pentru a delinka un modul,utilizatorul trebuie sa invoce programul extern */sbin/rmmod* care face urmatoarele operatii:

1.Din linia de comanda citeste numele modulului ce va fi delinkat.

2.Invoca procedura de sistem *query_module()* cu subcomanda QM_MODULES pentru a obtine lista modulelor linkate.

3. Invoca procedura de sistem *query_module()* cu subcomanda QM_REFS de cateva ori,pentru a obtine informatii dependente de modulele linkate.Daca un modul este linkat peste un modul ce va fi scos,se termina.

4. Invoca procedura de sistem *delete_module()* trecandu-i numele modulului acesteia. Rutina de serviciu corespunzatoare *sys_delete_module()* face urmatoarele operatii:

a.Verifica daca utilizatorului ii este permis sa scoata modulul.

b. Invoca *find_module()* pentru a gasi obiectul modul corespunzator din lista indicata de *module_list.*

c.Verifica daca campurile *refs* si *uc.usecount* sunt null;altfel returneaza un cod de eroare.

d.Daca e definit,invoca metoda *cleanup* pentru a face operatii necesare inchiderii

curate a modulului.Metoda este implementata de obicei de functia *cleanup_module()* definita in interiorul modulului.

e.Scaneaza lista *deps* a modulului si sterge modulul din lista *refs*.

f.Sterge modulul din lista indicata de *module_list*.

g.Invoca *vfree()* pentru a elibera zona de memorie folosita de modul si revine.

Bibliografie:

Understanding The Linux Kernel -Bovet si Cesati