STRUCTURI DE DEVICE DRIVERS IN LINUX

Nume: Ureche Mihai

Grupa: 433A

Cuprins:

1. Generalitati
2. Device Drivere in Linux
3. Identificator major si minor
4. Structuri de date importante pentru un dispozitiv de tip character
5. Inregistrarea si deinregistrarea dispozitivelor de tip character
6. Accesul la spatiul de adresa al procesului
7. Operatii implementate de device drivere de tip character
8. Sincronizare - cozi de asteptare
9. Bibliografie
10. Generalitati

Componentele hardware inglobate au nevoie de o component software care sa interactioneze direct si sa controleze componenta hardware, aceasta componenta software se numeste ***device driver.***

*Device driver-ele* sunt organizate in biblioteci care initializeaza hardware-ul si sunt componente intermediare intre hardware si nivelurile superioare de software.

Sistemele au diferite tipuri de component hardware ce au nevoie de ajutorul unor driver pentru a face legatura cu aplicatiile.

Exista diferite tipuri de drivere ce sunt folosite la:

* memorie (acces, alocare)
* initializare si controlul interfetelor de I/O
* initializare si transferuri pe magistrala

Driverele au incluse diferite functii cum ar fi: *dezactivare dispositive hardware, acces pentru citire, initializare, conigurare, activare, etc.*

O componenta hardware poate avea trei stari:

* *Inactiv* ( Hardware-ul este deconectat, fara alimentare sau dezactivat )
* *Ocupat* ( In aceasta stare hardware-ul poate prelucra date, de aceea este nevoie de un mecanism de eliberare )
* *Asteptare* ( Hardware-ul nu prelucreaza nici o data in momentul acesta, el poate permite o cerere de tip achizitie, citire sau scriere)

Uneori codul drivere-lor este integrat in alte niveluri sau este separat de alte niveluri software. Tipurile de software se pot executa in mai multe moduri, insa cele mai comune sunt:

* modul supervizor
* modul utilizator (user)

Diferenta dintre cele doua moduri este ca modul *supervizor* are mai multe drepturi de acces la componentele sistemului decat modul *utilizator*.

2.Device Drivere in Linux

Device driverele in UNIX se impart in doua categorii, ierarhie facuta dupa

viteza, volumul şi modul de organizare a datelor ce trebuie transferate de la

dispozitiv catre sistem si invers.

* **de tip caracter**: dispozitve lente, care gestioneaza un volum mic de date,iar accesul la date nu necesita operatii de cautare prea frecvente. Ex: tastatura, mouse-ul, placa de sunet, joystick-ul.
* **de tip bloc**: datele sunt organizate pe blocuri, iar volumul fiind mare operatiile de cautare sunt des folosite. Ex: hard disk-urile, cdrom-urile, ram discurile, unitatile de banda magnetica.

Cele doua tipuri de device drivere se apeleaza in mod diferit. Daca pentru dispozitivele de tip caracter apelurile de sistem ajung direct la device drivere, in cazul dispozitivelor de tip bloc device driverele nu lucreaza direct cu apelurile de sistem. Diferenta e data de interpunere a subsistemului de gestiune a fisierelor. Rolul acestuia este de a pregati device driverului resursele necesare (buffere), de a mentine in buffer cache datele recent citite si de a reordona operatiile de citire si scriere din motive de performanta.

Identificarea in UNIX a dispozitivelor se face cu ajutorul identificatorului.

Acestia se aloca atat static cat si dinamic, si este alcatuit din doua parti: **major** si **minor**. Astfel se identifica mai intai(prin major) tipul driverului prezent iar, prin minor, fiecare tip de dispozitiv deservit de acesta.

1. Identificator *major* si *minor*

Ca si in UNIX si in Linux dispozitivele au asociate cate un identificator unic, care poate fi alocat dinamic ( este folosit pentru compatibilitatea sistemelor) sau static ( este folosit inca de majoritatea driver-lor). Acest identificator are doua parti: ***major*** si ***minor***.

* *major* ( identifica tipul dispozitivului (disc SCSI, port serial, etc)
* *minor* ( identifica dispozitivul: primul disc, al doilea disc, etc)

*Majorul* in general identifica driver-ul pe cand minorul identifica dispozitivele fizice ce sunt deservite de catre driver. Un driver va avea asociat un major si va fi responsabil de toti minorii asociati cu acel major.

# ls -la /dev/hda? /dev/ttyS?

brw-rw---- 1 root disk 3, 1 2004-09-18 14:51 /dev/hda1

brw-rw---- 1 root disk 3, 2 2004-09-18 14:51 /dev/hda2

crw-rw---- 1 root dialout 4, 64 2004-09-18 14:52 /dev/ttyS0

crw-rw---- 1 root dialout 4, 65 2004-09-18 14:52 /dev/ttyS1

Folosind comanda *ls* se pot afla informatii despre fisierele de tip device asa cum se vede din exemplu de mai sus. Fisierele de tip character vor fi identificate cu ajutorul caracterului ***c*** iar cele de tip bloc cu ajutorul caracterului ***b***. *Majorul* si *minorul* pot fi observate in coloana 5 si 6 a rezultatului comenzii *ls*.

Pentru a alege identificatorul pentru un nou dispozitiv se pot folosi doua metode: static (se va alege un numar care nu mai este folosit de catre alt dispozitiv) sau dinamic. Dispozitivele incarcate, impreuna cu identificatorul major se gasesc in /proc/devices.

Comanda ***mknod*** este folosita pentru a crea un fisir de tip dispozitiv, aceasta primeste ca argument tipul ( bloc sau character ), majorul si minorul dispozitivului.

Ex: *name type major minor*

Mai jos avem un exemplu de creare a unui dispozitiv de tip character cu numele *mycdev* cu majorul 42 si minorul 0.

# mknod /dev/mycdev c 42 0

1. Structuri de date importante pentru un dispozitiv de tip character

Structura folosita de catre un dispozitiv de tip caracter folosita pentru inregistrarea acestuia in sistem este ***cdev***, acesta fiind reprezentat de catre aceasta structura si in kernel. Operatiile cu driver folosesc deobicei trei structure: ***struct file\_operations, struct file*** si ***struct inode***.

**Structura file\_operations**

Apelurile de sistem efectuate de utilizatori asupra fisierelor de tip dispozitiv sunt primite de catre device driverele de tip caracter nealterate. Pentru a putea implementa un device driver va trebui sa implementam si apelurile de sistem de lucru cu fisiere: *open, close, read, write, lseek, mmap,* *etc.* Structura ***file\_operations*** detine campuri care descriu operatiile enuntate mai sus.

#include <linux/fs.h>

struct file\_operations {

struct module \*owner;

loff\_t (\*llseek) (struct file \*, loff\_t, int);

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

[...]

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

[...]

int (\*open) (struct inode \*, struct file \*);

int (\*flush) (struct file \*, fl\_owner\_t id);

int (\*release) (struct inode \*, struct file \*);

[...]

};

Utilizatorul foloseste un apel de sistem diferit fata de semnatura functiei. Pentru a fi simplificata implementarea in device driver SO (Sistemul de operare) se interpune intre utilizator si device driver.

Functia ***open*** nu primeste ca parametru calea sau diversi parametrii care controleaza modul de deschidere a fisierului. De asemenea nici *read, write, release, ioctl, lseek* nu primesc ca parametru un descriptor de fisier. Ceea ce primesc ca parametrii aceste rutine sunt doua structure: *file* si *inode*.

Majoritatea parametrilor pentru operatiile prezentate au semnificatie directa:

* *file* si *inode* identifica fisierul de tip dispozitiv;
* *size* reprezinta numarul de octeti ce trebuie cititi sau scrisi;
* *offset* reprezinta offsetul de unde trebuie citit sau scris (trebuie actualizat corespunzator);
* *user\_buffer* reprezinta bufferul utilizatorului din care se citeste/in care se scrie;
* *whence* reprezinta modalitatea de seek;
* *cmd* si *arg* sunt parametrii trimisi de utilizatori la apelul *ioctl*.

**Structurile inode si file**

Din punctul de vedera al sistemului de fisere structura *inode* reprezinta un fisier. *Inode-ul* are urmatoarele attribute: dimensiunea, drepturile, timpii asociati fisierului. Un fisier intr-un sistem de fisiere este identificat de catre *inode* in mod unic.

Ca si *inode* si structura *file* reprezinta tot un fisier, dar este mai aproape de punctual de vedere al utilizatorului. Structuta *file* are urmatoarele atribute: *inode-ul, numele fisierului, atributele de deschidere ale fisierului, pozitia in fisier*. Structura *file* este asociata tuturor fisierelor deschise la un moment dat.

In cadrul ***device drivere-lor*** cele doua structuri ( *inode-ul* si *file-ul* ) au intotdeauna modalitati standard de folosire:

* *inode-ul* este folosit pentru a determina majorul si minorul device-ului asupra caruia se face operatia
* *file-ul* este folosit pentru a determina flag-urile cu care a fost deschis fisierul

Structura ***file*** contine, printre multe campuri, si:

* *f\_mode* care specifica permisiunile pentru citire (FMODE\_READ) sau scriere (FMODE\_WRITE);
* *f\_flags* care specifica flag-urile de deschidere a fisierului (O\_RDONLY, O\_NONBLOCK, O\_SYNC, O\_APPEND, O\_TRUNC etc.);
* *f\_op*, care specifica operatiile asociate fisierului (pointer catre structura *file\_operations*);
* *private\_data*, un pointer care poate fi folosit de programator pentru a pastra date specifice dispozitivului; pointerul va fi initializat la adresa unei zone de memorie alocate de progrmator.

Ca si *file* si *inode* contine printre multe informatii, un camp ***i\_cdev*** ( pointer catre structura care defineste dispozitivul de tip caracter ).

**Implementarea operatiilor**

Cand dorim sa implementam un device driver este recomandata creeare unei structuri care sa retina informatii despre dispozitivul dat, informatii utilizate in cadrul modulului. Pentru un dispozitiv de tip caracter, structura trebuie sa contina un camp de tipul ***struct\_cdev***, ce este folosit pentru a referi dispozitivul. In exemplu de mai jos este structura *struct my\_device\_data* este folosita in acest sens:

#include <linux/fs.h>

#include <linux/cdev.h>

struct my\_device\_data {

struct cdev cdev;

/\* my data starts here \*/

//...

};

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);

file->private\_data = my\_data;

//...

}

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;

//...

}

O structura precum *my\_device\_data* contine date associate unui dispozitiv. *Cdev* este un dispozitiv de tip caracter si este folosit pentru inregistrarea acestuia in sistem si identificarea dispozitivului. Campul *i\_cdev* din structura *inode* ne poate ajuta la aflarea pointer-ului catre membrul *cdev*. Informatiile de la *open* pot fi memorate in campul *private\_data* al structurii *file*, aceste informatii sunt disponibile si in rutinele *read, write, release, etc.*

1. Inregistrarea si deinregistrarea dispozitivelor de tip character

Pentru a inregistra sau deinregistra un dispozitiv va trebui sa specificam majorul si minorul acestuia. Pentru a optine majorul si minorul uni dispozitiv se foloseste macro-ul ***MKDEV*** pentru a le extrage din tipul ***dev\_t***. Functiile ***register\_chrdev\_region*** si ***unregister\_chrdev\_region*** sunt folosite pentru alocarea respectiv dezalocarea static a identificatorilor unui dispozitiv:

#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);

Se recomanda insa ca identificatorii de dispozitiv sa fie alocati dinamic cu ajutorul functiei ***alloc\_chrdev\_region***. Secventa de mai jos rezerva ***my\_minor\_count*** dispozitive, incepand de la dispozitivul cu majorul *my\_major* si minorul *my\_first\_minor* (daca se depaseste valoare maxima pentru minor, se trece la urmatorul major):

#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;

}

//...

Dupa ce identificatorii sunt atribuiti va trebui sa initializam dispozitivul de tip caracter (***cdev\_init***) de asemenea va trebui sa anuntam si nucleul de existenta lui (***cdev\_add***). Functia [***cdev\_add***](http://lxr.linux.no/linux+v2.6.28.7/fs/char_dev.c#L449)trebuie apelata doar dupa ce dispozitivul este pregatit sa primeasca apeluri. Eliminarea unui dispozitiv se realizeaza folosind functia [***cdev\_del***](http://lxr.linux.no/linux+v2.6.28.7/fs/char_dev.c#L471).

#include <linux/cdev.h>

void cdev\_init(struct cdev \*cdev, struct file\_operations \*fops);

int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count);

void cdev\_del(struct cdev \*dev);

Pentru a inregistra si initializa **MY\_MAX\_MINORS** dispozitive este folosita urmatoarea secventa:

#include <linux/fs.h>

#include <linux/cdev.h>

#define MY\_MAJOR 42

#define MY\_MAX\_MINORS 5

struct my\_device\_data {

struct cdev cdev;

/\* my data starts here \*/

//...

};

struct my\_device\_data devs[MY\_MAX\_MINORS];

struct file\_operations my\_fops = {

.owner = THIS\_MODULE,

.open = my\_open,

.read = my\_read,

.write = my\_write,

.release = my\_release,

.ioctl = my\_ioctl

};

int init\_module(void)

{

int i, err;

err = register\_chrdev\_region(MKDEV(MY\_MAJOR, 0), MY\_MAX\_MINORS,

"my\_device\_driver");

if (err != 0) {

/\* report error \*/

return err;

}

for(i = 0; i < MY\_MAX\_MINORS; i++) {

/\* initialize devs[i] fields \*/

cdev\_init(&devs[i].cdev, &my\_fops);

cdev\_add(&devs[i].cdev, MKDEV(MY\_MAJOR, i), 1);

}

return 0;

}

Urmatoarea secventa este folosita pentru a sterge si dezintegra **MY\_MAX\_MINORS** dispozitive::

void cleanup\_module(void)

{

int i;

for(i = 0; i < MY\_MAX\_MINORS; i++) {

/\* release devs[i] fields \*/

cdev\_del(&devs[i].cdev);

}

unregister\_chrdev\_region(MKDEV(MY\_MAJOR, 0), MY\_MAX\_MINORS);

}

1. Accesul la spatiul de adresa al procesului

Pentru un dispozitiv driver-ul reprezinta interfata cu care aplicatia poate comunica cu hardware-ul. Astfel uneori este nevoie ca in cadrul unui *device driver* sa accesam date in user-space. Accesarea directa a unui pointer din user-space poate duce la un comportament incorect (in functie de arhitectura, un pointer din user-space poate sa nu fie valid sau mapat in kernel-space), un kernel oops (pointerul din user-mode poate referi o zona de memorie care nu este rezidenta) sau probleme de securitate. Accesarea corecta a datelor din user-space se realizeaza prin apelarea macro-urilor/functiilor de mai jos:

#include <asm/uaccess.h>

put\_user(type val, type \*address);

get\_user(type val, type \*address);

unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n);

unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n)

Functiile sau macro-urile de mai sus intorc 0 in caz de suces si orice valoarea in caz de eroare:

* [***put\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/include/asm/uaccess.h#L223) pune in user-space la adresa address valoarea val; tipul poate fi unul pe 8, 16, 32, 64 de bisi (tipul maxim suportat depinde de platforma hardware);
* [***get\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/include/asm/uaccess.h#L155) analog cu functia precedenta, numai ca val va fi setata la o valoare identica cu valoarea de la adresa user-space data prin address;
* [***copy\_to\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/lib/usercopy_32.c#L838) copiaza din kernel-space de la adresa referita de from in user-space la adresa referita de to, size octeti;
* [***copy\_from\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/lib/usercopy_32.c#L860) copiaza din user-space de la adresa referita de from in kernel-space la adresa referita de to, size octeti.

Urmatorul exemplu ne ajuta sa intelegem cum se lucreaza cu aceste functii:

#include <asm/uaccess.h>

if (copy\_to\_user(user\_buffer, kernel\_buffer, size))

return -EFAULT;

else

return SUCCESS;

1. Operatii implementate de device drivere de tip caracter

**open si release**

Functia *open* realizeaza operatiile de initializare ale unui dispozitiv, de cele mai multe ori aceste operatii se refera la initializarea si completarea datelor specifice.

Functia *release* elibereaza resursele specifice dispozitivului: se dezaloca datele specifice si se inchide dispozitivul daca este ultimul apel *close*.

Functia *open* de cele mai multe ori are urmatoarea structura:

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;

}

Controlul accesului este o problema ce poate aparea la functia *open*. Uneori este necesar ca un dispozitiv sa fie deschis o singura data la un moment dat; mai exact, nu se permite al doilea *open* inainte de *release*.

Pentru a implementa aceasta restrictie se alege o modalitate de tratare a unui apel *open* pentru un dispozitiv deja deschis: se poate intoarce o eroare (-**EBUSY**), se pot bloca apelurile *open* pana la o operatie de *release* sau se poate inchide dispozitivul inainte de a realiza operația de *open*.

La apelul din user-space al functiilor *open* si *close* asupra dispozitivului, se vor apela operatiile *my\_open* si *my\_release* din driver. Un exemplu de apel din user-space:

int fd = open("/dev/my\_device", O\_RDONLY);

if (fd < 0) {

/\* handle error \*/

}

/\* do work \*/

//..

close(fd);

**read si write**

Functiile ***read*** si ***write*** transfera date intre dispozitiv si user-space:

* *read* citeste datele de la dispozitiv si le transfera in user-space;
* *write* citeste datele din user-space si le scrie pe dispozitiv.

Buffer-ul primit ca parametru reprezinta un pointer in user-space, motiv pentru care este necesara folosirea functiilor [***copy\_to\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/lib/usercopy_32.c#L838) sau [***copy\_from\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/lib/usercopy_32.c#L860).

Daca sa realizat un transfer partial inseamna ca valoarea intoarsa este mai mica decat parametrul *size* (numarul de octeti ceruti). Daca apare aceasta eroare atunci este apelata din nou functia corecpunzatoare apelului din sistem pana cand transferul se realizeaza corect.

Pentru a realiza un transfer de date format din mai multe transferuri partiale, vor trebui realizate urmatoarele operatii:

* se transfera numarul maxim de octeti posibil intre buffer-ul primit ca parametru si dispozitiv;
* se actualizeaza offset-ul primit ca parametru la pozitia de la care va incepe urmatoarea citire / scriere a datelor;
* se intoarce numarul de octeti transferati.

Exemplu de mai jos este un exemplu de simplu apel al functiei *read*, campul offset nu este actualizat astfel incat ca tot timpul va fi intors mesajul de la inceputul buffer-ului. Insa pentru ca functia sa fie implementata corect trebuie ca parametrul offset sa fie actualizat si sa se tina cont de el.

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;

}

Functia *write* are o structura asemanatoare cu cea a functiei *read*; citeste date din user-space folosind functia [***copy\_from\_user***](http://lxr.linux.no/linux+v2.6.28.7/arch/x86/lib/usercopy_32.c#L860) si le scrie pe dispozitiv.

La apelul functiilor *read* si *write* din user-space se vor apela operatiile *my\_read* si *my\_write* din driver. Mai jos avem un exemplu de cod pentru user-space:

if (read(fd, buffer, size) < 0) {

/\* handle error \*/

}

if (write(fd, buffer, size) < 0) {

/\* handle error \*/

}

**ioctl**

In afara de operatiile de *read* si *write*, driver-ul trebuie sa aibe posibilitatea de a realiza operatii de control asupra dispozitivului fizic, aceste operatii sunt realizate prin implementarea unei functii de tip ***ioctl***:

static int my\_ioctl (struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg);

Comanda transmisa din user-space este *cmd*. Daca la apelul din user-space se transmite un intreg, acesta poate fi accesat direct. Daca se trasmite un buffer, valoarea *arg* va fi un pointer catre acesta si trebuie accesat prin intermediul functiilor *copy\_to\_user* sau *copy\_from\_user*.

Trebuie sa alegem numerele corespondente cu comenzile inainte sa implementam functia *ioctl*. Se foloseste macrodefinitia ***\_IOC(dir, type, nr, size)*** pentru generarea codurilor *ioctl*.

Aceasta macrodefinitie are urmatorii parametrii:

* *dir* reprezinta directia de transfer a datelor *(\_IOC\_NONE, \_IOC\_READ, \_IOC\_WRITE)*;
* *type* reprezinta numarul magic;
* *nr* este numarul codului *ioctl* specific dispozitivului;
* *size* este dimensiunea datelor transferate.

Functia *ioctl* este implementata mai jos:

#include <asm/ioctl.h>

#define MY\_IOCTL\_IN \_IOC(\_IOC\_WRITE, 'k', 1, sizeof(my\_ioctl\_data))

static int my\_ioctl (struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg)

{

struct my\_device\_data \*my\_data =

(struct my\_device\_data\*) file->private\_data;

my\_ioctl\_data mid;

switch(cmd) {

case MY\_IOCTL\_IN:

if( copy\_from\_user(&mid, (my\_ioctl\_data \*) arg,

sizeof(my\_ioctl\_data)) )

return -EFAULT;

/\* process data and execute command \*/

break;

default:

return -ENOTTY;

}

return 0;

}

La apelul din user-space a functiei *ioctl*, se va apela functia *my\_ioctl* a driver-ului. Un exemplu de astfel de apel in user-space:

if (ioctl(fd, MY\_IOCTL\_IN, buffer) < 0) {

/\* handle error \*/

}

1. Sincronizare - cozi de asteptare

Atunci cand avem probleme de sincronizare putem folosi cozile de asteptare, de cele mai multe ori un thread este obligat sa astepte terminarea unei operatii, dar dorim ca aceasta asteptare sa nu fie *busy-waiting*.

Daca folosim cozi de asteptare sau functii care schimba starea thread-ului din planificabil in neplanificabil si invers se pot rezolva astfel de probleme.

In Linux, o coada de asteptare este o lista in care sunt trecute procesele care asteapta un anumit eveniment. O coada de asteptare este definita de tipul [*wait\_queue\_head\_t*](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L50) si poate fi folosita de functiile/macrourile:

#include <linux/wait.h>

DECLARE\_WAIT\_QUEUE\_HEAD(wait queue\_head\_t \*q);

void init\_waitqueue\_head(wait\_queue\_head\_t \*q);

int wait\_event(wait\_queue\_head\_t \*q, int condition);

int wait\_event\_interruptible(wait\_queue\_head\_t \*q, int condition);

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);

void wake\_up(wait\_queue\_head\_t \*q);

void wake\_up\_interruptible(wait\_queue\_head\_t \*q);

Rolurile macro-urilor/functiilor de mai sus sunt:

* [***init\_waitqueue\_head***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L80) initializeaza coada de asteptare; daca se doreste initializarea cozii la compilare, se poate folosi macroul [**DECLARE\_WAIT\_QUEUE\_HEAD**](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L74);
* [***wait\_event***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L187) si [***wait\_event\_interruptible***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L278)adauga thread-ul curent la coada de asteptare cat timp conditia este falsa, ii seteaza starea la **TASK\_UNINTERRUPTIBLE** sau **TASK\_INTERRUPTIBLE** si apeleaza scheduler-ul pentru planificarea unui nou thread; asteptarea va fi intrerupta atunci cand un alt thread va apela functia [***wake\_up***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L148);
* [***wait\_event\_timeout***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L221) si [***wait\_event\_interruptible\_timeout***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L306) au acelasi efect ca functiile de mai sus, doar ca asteptarea poate fi intrerupta la incheierea timeout-ului primit ca parametru;
* [***wake\_up***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L148) pune toate thread-urile oprite din starea **TASK\_INTERRUPTIBLE** si **TASK\_UNINTERRUPTIBLE** in starea **TASK\_RUNNING**; scoate aceste thread-uri din coada de asteptare;
* [***wake\_up\_interruptible***](http://lxr.linux.no/linux+v2.6.28.7/include/linux/wait.h#L153) aceeasi actiune, insa se folosesc doar thread-urile cu starea **TASK\_INTERRUPTIBLE**.

Cel mai simplu exemplu este cel al unui thread care asteapta ca valorii unui flag sa fie modificate. Initializarile se realizeaza prin secventa:

#include <linux/sched.h>

wait\_queue\_head\_t wq;

int flag = 0;

init\_waitqueue\_head(&wq);

Un thread va astepta ca flag-ul sa fie modificat la o valoare diferita de zero:

wait\_event\_interruptible(wq, flag !=0);

flag = 0;

in timp ce un alt thread va modifica valoarea flag-ului si va trezi thread-urile care asteapta:

flag = 1;

wake\_up\_interruptible(&wq);

3.Bibliografie

a) Linux Device Drivers, 3rd edition

b) Essential Linux Device Drivers - Chapter 5. Character Drivers

c) [http://cs.pub.ro/~pso/wiki/index.php/Laboratoare:Device\_drivere\_(Linux)#open\_.C8.99i\_release](http://cs.pub.ro/%7Epso/wiki/index.php/Laboratoare:Device_drivere_%28Linux%29#open_.C8.99i_release)

d) [http://vega.unitbv.ro/~romanca/psci/1-PSCI-Introducere-4spp.pdf](http://vega.unitbv.ro/%7Eromanca/psci/1-PSCI-Introducere-4spp.pdf)