

# Projecte de Sistemes Operatius: Manegadors de Dispositius

Yolanda Becerra, Julita Corbalán, Juan José Costa, Jordi Garcia,  
Marisa Gil, Jordi Guitart, Amador Millan, Gemma Reig<sup>1</sup>

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Setembre 2011

## 1 INTRODUCCIÓ

---

L'objectiu d'aquesta pràctica és aprofundir en el funcionament intern d'un sistema operatiu. En particular, s'ha de ser capaç de modificar les estructures de dades bàsiques d'un sistema operatiu i augmentar les seves funcionalitats.

En aquesta segona pràctica treballarem amb una distribució genèrica de Linux (en concret la 2.6) i implementarem diversos mòduls de kernel per a afegir noves funcionalitats.

Al laboratori, cal arrancar l'Ubuntu i el kernel etiquetat amb Proso com sempre.

A [1] podeu trobar tota la documentació pertinent als mòduls de kernel de Linux. Bàsicament, els mòduls permeten modificar/afegir parts al kernel dinàmicament, mentre està en execució, sense haver de recompilar o relinkar, com heu fet a l'anterior projecte: així, podem veure una altra manera de modificar codi de sistema.

Lògicament, no tot està permès, i no totes les funcions es poden inserir d'aquesta manera al kernel. Les més habituals i apropiades són els manegadors de dispositius. A més, cal ser usuari privilegiat, no tothom està autoritzat a fer canvis al sistema. Un exemple típic de mòdul és el dels manegadors (drivers) d'impressora. Imagineu un ordinador portàtil que el feu servir tan a casa com a la feina, de manera que tindreu diversos manegadors d'impressora instal·lats en ell. Però no heu de tenir la mateixa impressora a casa que a la feina, de manera que, tot i tenir els manegadors instal·lats, no sempre tindreu disponible el dispositiu físic (impressora).

---

<sup>1</sup> Aquest document ha estat preparat amb la participació dels següents professors que han impartit PROSO en altres cursos: Ruben Gonzalez, Silvia Llorente i Pablo Chacín.

En concret, en aquesta pràctica, s'afegirà un mecanisme de monitorització per algunes crides a sistema de Linux. Aquesta monitorització s'afegirà dinàmicament mitjançant la utilització d'un mòdul, sense la necessitat de recompilar el kernel de Linux. Un cop monitoritzat, permetrem que l'usuari pugui accedir a aquestes dades mitjançant un dispositiu que ens permetrà triar les estadístiques a consultar. Per tant, caldrà crear un manegador per aquest dispositiu i, per tal de no haver de recompilar el kernel, farem servir un altre mòdul.

A continuació, veurem un petit resum dels conceptes que s'han de tenir clars així com el codi bàsic per a crear mòduls, dispositius i manegadors.

## 2 CONCEPTES PREVIS

### 2.1 Mòduls (*Linux Kernel Modules*)

És un mecanisme per afegir un conjunt de rutines i dades al sistema de forma dinàmica. Cada mòdul es compon d'un fitxer objecte (no muntat fins a executable) que pot ser muntat (insertat) dinàmicament al sistema en execució amb el programa `insmod` i desmuntat amb el programa `rmmod`.

#### 2.1.1 Definint el mòdul

En essència un mòdul només necessita definir una funció d'inicialització i una de finalització, tal com es pot veure a la Figura 1, en la que es defineixen les funcions `Mymodule_init` i `Mymodule_exit`.

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

MODULE_LICENSE("GPL");

/*
 * Inicialització del mòdul.
 */
static int __init Mymodule_init(void)
{
    /* Codi d'inicialització */
    printk(KERN_DEBUG "Mymodule carregat amb exit\n");
    return 0;
    // Aquesta funció retorna 0 si tot ha anat correctament i < 0 en cas d'error
}
/*
 * Descarrega el mòdul.
 */
static void __exit Mymodule_exit(void)
{
    /* Codi de finalització */
}
module_init(Mymodule_init);
module_exit(Mymodule_exit);
```

**Figura 1: Codi bàsic d'un mòdul (`mymodule.c`)**

Els *tokens* opcionals `__init` i `__exit` s'utilitzen per indicar al kernel que aquestes funcions només poden ser usades en el moment d'inicialitzar/finalitzar el mòdul.

Les rutines definides amb les macros `module_init` i `module_exit` s'executen automàticament, al carregar i descarregar el mòdul respectivament. L'ús d'aquestes macros és obligatori.

### 2.1.2 Definició de paràmetres del mòdul en temps de càrrega

En la versió 2.4 i posteriors apareix la possibilitat de definir paràmetres en temps de càrrega. La interfície és molt senzilla:

**module\_param** Per definir el paràmetre, el seu tipus, i els permisos del fitxer que es crearà a `sysfs`<sup>2</sup> i que permetrà a l'usuari accedir al paràmetre (en el nostre cas 0, que vol dir que no es crearà cap fitxer).

**MODULE\_PARM\_DESC** Permet afegir una petita descripció al paràmetre (que es pot consultar més tard mitjançant la comanda `modinfo`).

**MODULE\_AUTHOR** Inclou el nom de l'autor dins el mòdul.

**MODULE\_DESCRIPTION** Inclou una descripció del mòdul

**MODULE\_LICENSE** Indica quin tipus de llicència té el mòdul: GPL, BSD, ...

Aquí hi ha un petit exemple, al codi font del mòdul definirem un paràmetre (`pid`) de tipus enter que podrem modificar en temps de càrrega:

```
#include <linux/moduleparam.h>
...
int pid = 1;
module_param(pid, int, 0);
MODULE_PARM_DESC(pid, "Process ID to monitor (default 1)");
...
MODULE_AUTHOR("Joe Bloggs <joe.bloggs@somewhere>");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("ProSO driver");
...
```

### 2.1.3 Compilació d'un mòdul

Per a compilar el codi de la Figura 1 que guardem amb el nom `mymodule.c`, s'ha de crear un Makefile com aquest:

```
obj-m += mymodule.o
```

---

<sup>2</sup> Sysfs és un sistema de fitxers que normalment està a `/sys`. Teniu més detalls del funcionament d'aquest sistema al capítol 2 del llibre "Linux Device Drivers" citat a la bibliografia d'aquest document.

```
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Aquesta comanda donarà com a resultat un fitxer ELF amb el nom **mymodule.ko** (ko=kernel object)

### 2.1.4 Quines utilitats ofereix Linux per a gestionar mòduls

Linux ofereix els programes `insmod`, `lsmod` i `rmmod`, que instal·len, llisten i eliminen mòduls del sistema respectivament. A més, també ofereix: `modprobe`, una comanda per determinar si existeix un mòdul amb el nom donat i instal·lar-lo automàticament; i `modinfo`, per a consultar informació sobre el mòdul. Per carregar el mòdul farem:

```
#insmod mymodule.ko
```

Quan es carrega el mòdul li podem indicar algun paràmetre dels que tingui definits:

```
#insmod mymodule.ko pid=1
```

Proveu també a executar la comanda:

```
#modinfo mymodule.ko
```

### 2.1.5 Què fa internament el programa `insmod`?

Carrega el fitxer donat a l'espai d'adreces del sistema operatiu i enllaça qualsevol símbol que encara estigui sense resoldre en el fitxer objecte amb la taula de símbols del sistema en execució.

També permet canviar certs valors de variables enteres o cadenes de l'objecte, de forma que el mòdul (el manegador que contingui) pot ser configurat en temps de càrrega.

### 2.1.6 Quan es pot eliminar un mòdul?

Un mòdul només es pot eliminar quan ja no s'està accedint a ell. Per saber si està en ús o no, el kernel utilitza un comptador de referències que és necessari mantenir actualitzat. Per exemple, totes les funcions que poden ser cridades des de fora del mòdul han d'incrementar el comptador de referències al principi de la seva execució i decrementar-lo al final. Per a fer aquesta actualització el programador del mòdul pot utilitzar les funcions:

- `try_module_get(THIS_MODULE)`: Incrementa el comptador
- `module_put(THIS_MODULE)`: Decrementa el comptador

Aquests comptadors es poden consultar al dispositiu especial `/proc/modules`. Si aquest comptador no és zero, no es pot descarregar el mòdul, per tant, és

important mantenir consistent el nombre de get's i put's que fem del comptador.

### 2.1.7 Expressant dependències entre mòduls

Hi ha vegades en que un mòdul necessita de la funcionalitat d'un altre mòdul, de forma, que no es pot inserir el primer mòdul fins que el segon no ha estat inserit. Per tal de carregar tots els mòduls necessaris automàticament, Linux permet expressar aquestes dependències mitjançant el fitxer `/lib/modules/version/modules.dep` (per exemple `/lib/modules/2.6.27-proso/modules.dep`). Per exemple, si un mòdul `modulA` necessita el mòdul `modulB` es pot expressar així:

```
/path_complet/modulA.ko: /path_complet/modulB.ko
/path_complet/modulB.ko:
```

Cal notar, que s'ha d'explicitar el *path* complet al mòdul. D'aquesta manera, la comanda `modprobe` ens facilita la vida i al fer un

```
#modprobe modulA.ko
```

carregarà automàticament els dos mòduls amb l'ordre correcte.

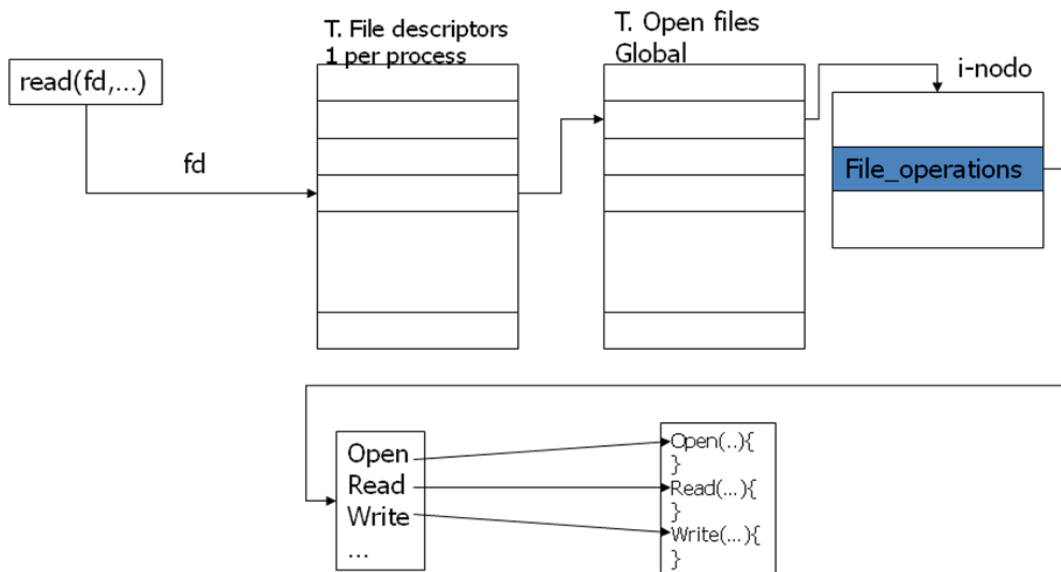
## 2.2 Dispositius

Un dispositiu (device) és un perifèric, real o virtual, que els usuaris poden fer servir per fer entrada/sortida o per interaccionar amb el nucli del sistema operatiu.

### 2.2.1 Què és un manegador de dispositiu?

Com el seu nom indica: manega el dispositiu. És el conjunt de rutines i variables dependents que controlen el funcionament d'un dispositiu (`open`, `read`, `write`, ...), com es pot veure a la Figura 2.

Per norma general, aquestes rutines que controlen el funcionament d'un dispositiu necessitaran accedir a certes instruccions (in/out) o a determinades adreces que un usuari normal no ha de poder accedir. Per poder accedir a aquestes operacions i/o dades, el codi, s'ha d'executar en mode sistema i per aquest motiu, el manegador s'incorpora al codi del sistema operatiu.



**Figura 2 Estructures de dades per a la gestió de dispositius a Unix.**

## 2.2.2 Com s'instal·la un manegador de dispositiu al sistema?

Hi ha dos mecanismes possibles:

- Estàticament. Recompilació de tot el sistema incloent les rutines pròpies del manegador.
- Dinàmicament. Utilitzant crides a sistema o programes que ens permetin incloure dinàmicament fitxers objecte a dins del nucli del sistema operatiu (**per exemple, un mòdul**). Podeu veure com es compila i s'instal·la un mòdul a les seccions 2.1.3 i 2.1.4.

## 2.2.3 Definir les operacions del dispositiu

Per a definir el manegador, només hem de definir el conjunt d'operacions vàlides per al dispositiu utilitzant l'estructura `file_operations`, definida al fitxer de capçaleres `<linux/fs.h>`. El seu format és:

```
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(*aio_read) (struct kiocb *, char __user *, size_t, loff_t *);
    ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t(*aio_write) (struct kiocb *, const char __user *, size_t,
        loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
        unsigned long);
};
```

```

int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*aio_fsync) (struct kiocb *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t(*readv) (struct file *, const struct iovec *, unsigned long,
                loff_t *);
ssize_t(*writev) (struct file *, const struct iovec *, unsigned long,
                loff_t *);
ssize_t(*sendfile) (struct file *, loff_t *, size_t, read_actor_t,
                  void __user *);
ssize_t(*sendpage) (struct file *, struct page *, int, size_t,
                  loff_t *, int);
unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                   unsigned long, unsigned long,
                                   unsigned long);
};

```

Les que utilitzarem nosaltres són `open`, `release` (que correspon al `close`), `read` i `ioctl`. La funció `open` ha de retornar 0 si tot va be, < 0 en cas d'error. La funció `release` ha de retornar 0 si tot va be, < 0 en cas d'error. La funció `read` té com a arguments el *buffer* d'usuari on deixar els caràcters llegits, el nombre de caràcters que s'han de llegir *size* i un paràmetre *offset* d'entrada/sortida que indica la situació del punter de lectura/escriptura abans de fer la lectura, i on s'ha de deixar la situació d'aquest punter després de fer la lectura. La crida ha de retornar el nombre de bytes llegits, 0 si s'ha arribat a final de *fitxer*, < 0 en cas d'error. La funció `ioctl` ha de retornar 0 si tot va be, < 0 en cas d'error.

El primer camp de l'estructura `file_operations`, anomenat `owner`, s'utilitza si el manegador és instal·lat amb un mòdul. Aquest camp permet estalviar al programador el manteniment explícit del comptador de referències del mòdul (explicat a la secció 2.1.6). Si l'inicialitzem amb la macro `THIS_MODULE`, el kernel s'encarrega d'incrementar i decrementar el comptador de referències del mòdul de la manera adequada i no cal utilitzar les funcions `try_module_get` i `module_put`<sup>3</sup>.

Dins d'aquest fitxer de capçaleres també es troben les definicions de les estructures `struct inode` i `struct file`.

Per tal de facilitar la lectura i comprensió del codi es poden definir els camps d'aquesta estructura amb noms (*tags*), tal com:

```

struct file_operations mymod_fops = {
    owner: THIS_MODULE,

```

<sup>3</sup> Tingueu en compte, que es considera que un manegador està en us des del moment en que s'obri el dispositiu fins que es tanca.

```
read: mymod_read,  
ioctl: mymod_ioctl,  
open: mymod_open,  
release: mymod_release,  
};
```

De totes maneres, cal tenir present que aquesta sintaxi no és estàndard de C, és una extensió del compilador GNU. Cas que no disposem d'aquest compilador, haurem de fer ús del nostre vell amic NULL a tots aquells camps que no vulguem inicialitzar.

Per últim, cal dir que només cal oferir aquelles operacions que tingui el manegador.

## 2.2.4 Identificació de dispositius

Com sap el sistema que una determinada crida al sistema fa referència a un manegador?

Per defecte. . . no ho sap.

Al moment d'instal·lar el manegador, cal indicar explícitament el seu identificador, que ha de ser únic i que es genera a partir de dos números enters anomenats *major* i *minor* (tradicionalment el *major* s'utilitza per a identificar un tipus de dispositiu i el *minor* per a distingir entre diferent subtipus). Aquesta operació s'anomena *registrar* el manegador i es fa en dues fases: primer es reserva l'identificador i després s'associen les operacions amb l'identificador seleccionat.

Per a generar un identificador de manegador s'ha d'utilitzar la macro `MKDEV` que, donats un *major* i un *minor*, omple una estructura de tipus `dev_t` amb l'identificador corresponent.

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

La definició de l'estructura `dev_t` es troba al fitxer de capçaleres `<linux/types.h>`.

D'altra banda, quan s'afegeix un nou dispositiu al sistema és necessari indicar també el *major* i el *minor* que componen l'identificador del manegador encarregat de gestionar-lo. Així, cada cop que es faci una operació sobre un dispositiu, el sistema utilitzarà l'identificador per a localitzar les operacions pròpies del seu manegador.

## 2.2.5 Com es registra un manegador de dispositiu?

El primer pas per registrar un manegador és reservar el seu identificador. Amb la funció `register_chrdev_region` definida al fitxer de capçaleres `<linux/fs.h>` podem reservar un rang d'identificadors de manegadors:

```
int register_chrdev_region (dev_t first, unsigned int count, const
char *name);
```

Els arguments són el primer identificador de la regió que volem reservar (*first*), que prèviament s'ha de generar a partir d'un *major* i un *minor* amb la macro `MKDEV`; el nombre d'identificadors que volem reservar (*count*); i el nom del dispositiu (*name*), que sortirà a `/proc/devices`. Un retorn negatiu indica error.

Aquesta funció reserva *count* identificadors, tots ells amb el mateix *major* (el del paràmetre *first*) y amb *minors* consecutius (començant pel *minor* que forma part del paràmetre *first*)<sup>4</sup>.

Per alliberar els identificadors dels manegadors i permetre que s'utilitzin per a nous registres tenim la crida `unregister_chrdev_region`.

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Els arguments són el primer identificador de la regió (*first*) i el nombre de identificadors que formen part de la regió (*count*).

Un cop reservats els identificadors és necessari associar les operacions específiques del manegador. Per fer-ho s'utilitza una estructura de tipus `cdev` que trobareu definida al fitxer de capçaleres `<linux/cdev.h>`. El primer que cal fer es declarar una nova estructura de tipus `cdev`:

```
struct cdev *my_cdev;
```

i reservar espai per aquesta estructura amb la funció<sup>5</sup>:

```
struct cdev *cdev_alloc();
```

---

<sup>4</sup> També és possible deixar que sigui el sistema el que ens assigni tots els identificadors dels manegadors, sense passar com a paràmetre el primer identificador del rang, però per això s'ha d'utilitzar la funció `alloc_chrdev_region` en lloc de `register_chrdev_region`. Per a més detalls sobre aquesta funció alternativa podeu consultar el capítol 3 del llibre "Linux Device Drivers" citat a la bibliografia d'aquest document.

<sup>5</sup> Si la variable de tipus `cdev` es defineix de forma estàtica, no com a un punter, llavors en comptes d'utilitzar la funció `cdev_alloc` s'ha d'utilitzar la funció `cdev_init`. Al llibre "Linux Device Drivers", citat a la bibliografia, en podeu trobar la seva interfície.

A continuació s'han d'inicialitzar els seus camps. Aquesta estructura té dos camps que hem d'inicialitzar: un camp `owner`, per a que el sistema mantingui el comptador de referències de l'estructura i que s'ha d'inicialitzar amb la macro `THIS_MODULE`, i un camp `ops` que s'ha d'inicialitzar amb la estructura `file_operations` que contingui les funcions específiques del manegador.

I per últim s'ha d'afegir aquesta estructura a les estructures de manegadors registrades al sistema. Per fer-ho tenim la funció:

```
int cdev_add(struct cdev *dev, dev_t first, unsigned int count);
```

Els paràmetres d'aquesta funció són l'estructura que conté les operacions del manegador (`dev`), el primer identificador de la regió (`first`) i el nombre de manegadors de la regió que volem associar a aquestes operacions (`count`). Aquesta funció retorna un número negatiu si hi ha algun error. Fins que no es completi amb èxit l'execució d'aquesta funció el manegador no serà visible per al sistema i, per tant, no es podran utilitzar les seves funcions.

A continuació tenim un petit exemple, on es declara i s'inicialitza un nou `cdev`:

```
dev_t dev = MKDEV(mymajor, myminor);
struct cdev *my_cdev;
my_cdev = cdev_alloc();
my_cdev->owner = THIS_MODULE;
my_cdev->ops = &my_fops; /* my_fops és una estructura estàtica de
tipus file_operations prèviament inicialitzada amb les operacions
del manegador */
cdev_add( my_cdev, dev, ndev ); /* Cal fer comprovació d'errors */
```

Quan el manegador ja no està en us és necessari eliminar la seva estructura `cdev` del sistema. Per a fer-ho s'utilitza la funció:

```
void cdev_del(struct cdev *dev);
```

## 2.2.6 Com es decideixen el major i el minor ?

A partir del *major* i del *minor* es construeix l'identificador del manegador. Per tant hem de seleccionar una combinació que cap altre dispositiu tingui assignada. Al fitxer `/proc/devices` tenim la llista tots els manegadors instal·lats amb el seu

*major*<sup>6</sup>. Tot i que la versió 2.6 del kernel permet que diferents manegadors tinguin el mateix *major*, per a assegurar-nos que la combinació *major-minor* està lliure podem seleccionar un *major* no assignat. D'aquesta manera podem seleccionar qualsevol *minor* amb la seguretat que la combinació està lliure.

Existeix una alternativa que allibera al programador de la tasca de seleccionar aquests números, que és indicar al sistema que ens reservi ell dinàmicament el rang d'identificadors del manegador (lo qual fa que implícitament decideixi el *major* i els *minors* de la regió)<sup>7</sup>. Cal remarcar que amb aquesta opció pot ser que cada vegada que instal·lem el manegador el seu identificador sigui diferent, la qual cosa s'ha de tenir en compte a l'afegir els dispositius manegats.

## 2.2.7 A dintre el manegador de dispositiu, com es sap el major i el minor?

Amb les macros:

```
int MAJOR(dev_t dev);
int MINOR(dev_t dev);
```

El valor de `dev` s'extreu de l'inode (que és un dels paràmetres que reben totes les funcions de dispositiu que es veuen a la secció 2.2.3)

## 2.2.8 Com associar un fitxer a un dispositiu?

Els dispositius són visibles des del sistema de fitxers (per defecte els fitxers que pengen de `/dev/*` són dispositius). I es poden crear amb la crida a sistema `mknod`.

```
#mknod fitxer tipus major minor
```

Els arguments són: el *fitxer*, que identifica el fitxer que usarem com a dispositiu; el *tipus* (una 'c' per crear un dispositiu de caràcters); el *major* i el *minor*, que són enters que permeten identificar al dispositiu dins del sistema (a la secció 2.2.4 podeu trobar una mica més de detall, inicialment, podeu usar un *minor* qualsevol).

Per a consultar els diferents dispositius creats al sistema podeu consultar el fitxer

---

<sup>6</sup> A les versions anteriors de Linux el *major* servia per a identificar el manegador del dispositiu, i el *minor* només l'utilitzava internament el manegador per a distingir entre els diferent tipus de dispositius que podia gestionar. A partir de la versió 2.6 es considera que tots dos nombres són necessaris per identificar les operacions associades al dispositiu. Malgrat això el format del `/proc/devices` encara mostra només el *major* del manegador.

<sup>7</sup> Per a més detalls consulteu el capítol 3 del llibre "Linux Device Drivers" citat a la bibliografia d'aquest document.

`/proc/devices`, on apareixen els dispositius (major i noms registrats) disponibles agrupats pel tipus del dispositiu.

Un cop creat el fitxer de dispositiu és necessari definir les funcionalitats d'aquest "fitxer", és a dir, quines operacions permetem sobre ell. Això s'aconsegueix amb el manegador de dispositiu (device driver).

### 2.2.9 Com pot accedir l'usuari al dispositiu?

L'usuari pot accedir al dispositiu amb les crides a sistema habituals d'accés a fitxers: `open`, `close`, `read`, `write`, `ioctl`, etc.

L'única d'aquestes crides al sistema que és realment dependent del perifèric és la crida `ioctl`, que permet fer operacions específiques al perifèric amb la combinació dels seus dos darrers paràmetres.

## 2.3 Linux

### 2.3.1 On és el codi del linux?

Al directori `/usr/src/linux` hi ha els fonts del sistema. A `/usr/src/linux/include` les capçaleres corresponents a la versió del sistema. Al capítol 3 del llibre [2] podeu trobar les diferents rutines i estructures que usa Linux per a gestionar processos: `for_each_process`, `find_task_by_pid`, .... (busqueu-les a <http://lxr.linux.no>)

### 2.3.2 Símbols

Quan parlem de *símbols* ens estem referint a noms de **variables** i noms de **rutines**. Els símbols d'un fitxer objecte es poden consultar amb la comanda `nm`.

Un altre tipus de símbols que no es poden visualitzar amb la comanda anterior són les macros definides amb la directiva **#define**, per exemple, la macro `current`, que retorna un punter de tipus `struct task_struct`, que fa referència a les dades de control del procés en execució, altrament anomenades PCB (*Process Control Block*) (consulteu <http://lxr.linux.no>) .

### 2.3.3 La taula de símbols de Linux

Linux exporta tot un conjunt de símbols i, per tant, els podem fer servir des dels mòduls. Per a cada símbol exportat (nom de variable, rutina, ...) el sistema guarda el **nom** i l'**adreça** de memòria on es troba localitzat (adreça lògica de sistema).

Aquesta taula de símbols es crea en compilar el sistema, ja que cal tenir el nom del símbol i la seva adreça. Per exportar una variable o funció cal usar la macro `EXPORT_SYMBOL(nom_símbol)` i recompilar el kernel (veieu exemples a <http://lxr.linux.no>). **Els mòduls, donat que formen part del kernel, també poden exportar símbols fent servir aquesta mateixa macro.**

Al fitxer `kernel/ksyms.c` trobareu els símbols exportats independents de

l'arquitectura i al fitxer `arch/i386/kernel/i386_ksyms.c` trobareu els símbols exportats que **depenen** de l'arquitectura Intel.

### 2.3.4 Com saber què conté la taula de símbols del sistema

Hi ha dues maneres:

- Llegint el dispositiu `/proc/ksyms`
- Emprant la comanda `ksyms -a`

### 2.3.5 Què cal retornar a les rutines internes del sistema? Qui rep aquesta informació?

Per conveni, a tot Linux es retorna  $\geq 0$ , per indicar que tot funciona correctament. Un valor negatiu,  $< 0$ , indica un error. El tipus d'error s'indica amb el valor absolut del codi retornat. Cal buscar el seu significat al fitxer de capçaleres `<sys/errno.h>`.

### 2.3.6 Què cal fer si en cridar una rutina interna del sistema, ens retorna error? Quin tipus d'error retornem?

Si no tenim un tractament específic, cal retornar el mateix error que ja ens ha retornat la rutina del sistema.

### 2.3.7 Espai d'usuari i espai de sistema

En cada instant d'execució, el sistema contempla dues funcions de traducció d'adreces. Depenent del mode d'execució del processador, s'utilitza una funció o l'altre.

El més habitual és tenir una única funció de traducció pel codi de sistema, i una funció de traducció per cada procés que hi ha en execució. Aquest mecanisme ens garanteix seguretat al sistema, ja que l'usuari no pot canviar dades del sistema des de les seves aplicacions, o entre aplicacions, en no poder accedir als espais d'adreces d'altres processos.

Així, com que el mecanisme d'accés a memòria depèn del mode d'execució i en mode sistema és necessari accedir a l'espai d'adreces de l'usuari (per exemple, pas de paràmetres a determinades crides al sistema), es necessitaran unes instruccions especials per indicar al processador que tot estar actiu el mode sistema, cal utilitzar l'espai d'adreces d'usuari.

Recordeu que, com ja heu fet al projecte 1, haureu de comprovar en tot cas si és possible fer aquesta còpia d'informació entre l'espai d'usuari i el de sistema.

### 2.3.8 Quines operacions existeixen per copiar dades entre espais d'adreces?

Bàsicament n'hi han dues i estan declarades a `<asm-i386/uaccess.h>`:

```
unsigned long copy_from_user(void *to, const void *from, unsigned long count);
```

Per copiar de mode usuari a mode sistema.

```
unsigned long copy_to_user(void *to, const void *from, unsigned long count);
```

Per copiar de mode sistema a mode usuari.

Comproveu els paràmetres i valor de retorn a <http://lxr.linux.no>.

### 2.3.9 Com funciona el printk?

És la rutina que es pot utilitzar a dins del *kernel* per escriure informació a la consola de l'ordinador. No obstant, el seu ús (excessiu) està desencoratjat.

El format i paràmetres que utilitza són una versió limitada dels que reconeix la rutina `printf` de la llibreria de C. És una funció d'escriptura *line-buffered*, que vol dir que **fins no trobar el salt de línia no escriu les dades**.

Com a característica especial, cal comentar que els primers caràcters de la cadena s'interpreten com la prioritat del missatge que es vol escriure. El format d'aquesta informació és:

```
printk ("<N>Goodbye cruel world\n");  
printk (KERN_EMERG "Goodbye cruel world\n");
```

on `N` és un número, de 0 a 7. Depenent de la prioritat que es posi, el missatge surt a llocs diversos: la consola de l'ordinador, algun fitxer de log (com per exemple `/var/log/messages` o `/var/log/kern.log`), etc. El nom dels fitxers de log dependrà de la configuració del sistema (`/etc/syslog.conf`).

En el fitxer `<linux/kernel.h>` trobareu la definició d'algunes macros, com `KERN_EMERG`, per a definir diferents prioritats. Si la prioritat és inferior a `console_loglevel` (primer valor del fitxer `/proc/sys/kernel/printk`), el missatge s'escriu a la consola. Si `syslogd` i `klogd` estan executant-se, el missatge també s'escriurà al fitxer de log, tant si surt per la consola com si no.

Tots aquests missatges emesos pel kernel s'emmagatzemen a una estructura anomenada "kernel ring buffer" que pot examinar-se amb la comanda `dmesg`. Aquest buffer no és il·limitat i a mida que es van generant nous missatges a kernel es van eliminant el més antics (una forma de cua circular).

Per a més informació consultar l'apèndix o fer:

```
man dmesg
man syslogd
man syslog.conf
```

### 3 DESCRIPCIÓ DEL TREBALL A REALITZAR

---

En aquesta pràctica es vol modificar el sistema per a que prengui estadístiques d'ús del sistema. Tots els sistemes operatius d'avui en dia, disposen de diferents sistemes per emmagatzemar estadístiques sobre ells mateixos, de manera que es fàcil identificar possibles problemes i actuar en conseqüència. En el nostre cas particular, es vol conèixer el temps mig de resposta del sistema, i ens centrarem en les crides a sistema.

Per fer això, caldria modificar el punt d'entrada de cadascuna de les crides a sistema a comptabilitzar. En el nostre cas, voldrem guardar la informació de les crides **open**, **write**, **clone**, **close** i **lseek**. La informació que és vol extreure per cada tipus de crida es pot resumir en:

- Quantes vegades s'inicia la crida.
- Quantes vegades finalitza correctament.
- Quantes vegades finalitza incorrectament.
- El temps total executant la crida.

El fet d'afegir aquesta instrumentació a cada crida a sistema pot alentir el correcte funcionament del sistema. És per això que es demana que aquesta instrumentació sigui dinàmica i que es pugui activar i desactivar.

Per aconseguir aquest dinamisme, per activar la instrumentació interceptarem la taula de *crides a sistema* de Linux i modificarem cadascuna de les funcions a mesurar per una funció local que mesurarà el temps que triga a executar la funció antiga. Aquesta funció local ha de tenir la mateixa interfície que la crida a sistema corresponent (podeu consultar aquesta interfície al codi font de Linux).

En resum, caldrà fer dos mòduls:

- **Mòdul 1** Per interceptar les crides i mesurar el temps.
- **Mòdul 2** Per accedir a les estadístiques del sistema.

#### 3.1 Mòdul 1: Intercepció i mesures

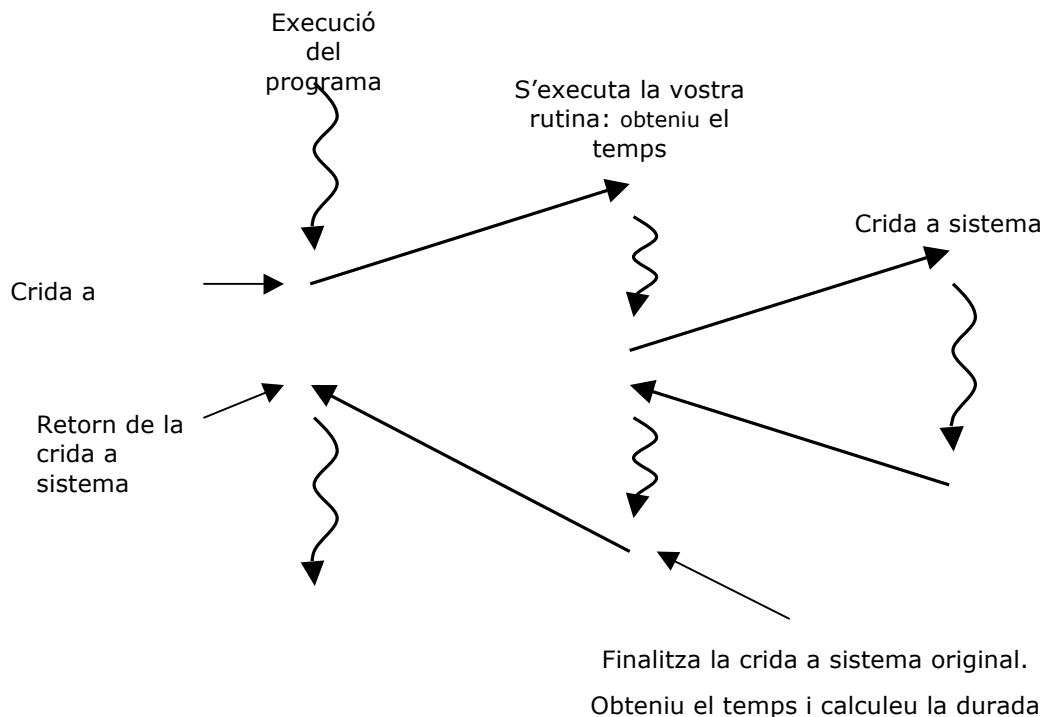
Aquest mòdul serà l'encarregat d'interceptar la taula de símbols (modificar la taula de crides a sistema), inserir les funcions d'instrumentació a l'inserir el mòdul al sistema (activant la instrumentació), i eliminar-les al treure'l.

Les funcions d'instrumentació estaran situades al mòdul i s'encarregaran

d'actualitzar els comptadors de les crides a sistema pel procés actual. Per realitzar el comptatge del temps, podeu fer servir el mecanisme explicat a la secció 3.1.6 "Com es pot comptar el temps?".

A partir de la taula de crides a sistema, substituïrem les funcions originals a monitoritzar per les nostres rutines. Aquestes rutines de monitorització hauran de (veure Figura 3):

1. Marcar inici de crida a sistema
2. Executar crida a sistema original
3. Calcular el temps total d'execució i capturar el resultat de la crida



**Figura 3. Esquema d'intercepció de crides a sistema**

### 3.1.1 Interceptant la taula de crides a sistema

La taula de crides a sistema s'anomena *sys\_call\_table* i és una taula de punters a funció. Per tant, si al vostre mòdul definiu:

```
extern void * sys_call_table[];
```

tindreu un símbol que referència la taula de crides a sistema. (Busqueu-la a <http://lxr.linux.no>)

### 3.1.2 Interceptant les crides a sistema

Un cop tenim la taula de crides a sistema, només ens cal guardar les funcions a monitoritzar. Un exemple on es guarda a una variable:

```
sys_open_old = sys_call_table[POS_SYSCALL_OPEN];
```

a partir d'aquest moment, dins la variable *sys\_open\_old* tenim un punter a la crida a sistema *open* original.

### 3.1.3 Modificant la taula de crides a sistema

Com és fàcil imaginar, la taula es modifica d'igual manera, per tant, si hem definit una funció per a monitoritzar l'*open* anomenada *sys\_open\_local*, només caldrà fer:

```
sys_call_table[POS_SYSCALL_OPEN] = sys_open_local;
```

### 3.1.4 Capceleres de les crides a tracejar

Es posen a mode d'exemple les capceleres de les crides que es tracejaren al modul 2, fixeuvos que les capçaleres de kernel no son iguals que les d'usuari (les crides a sistema):

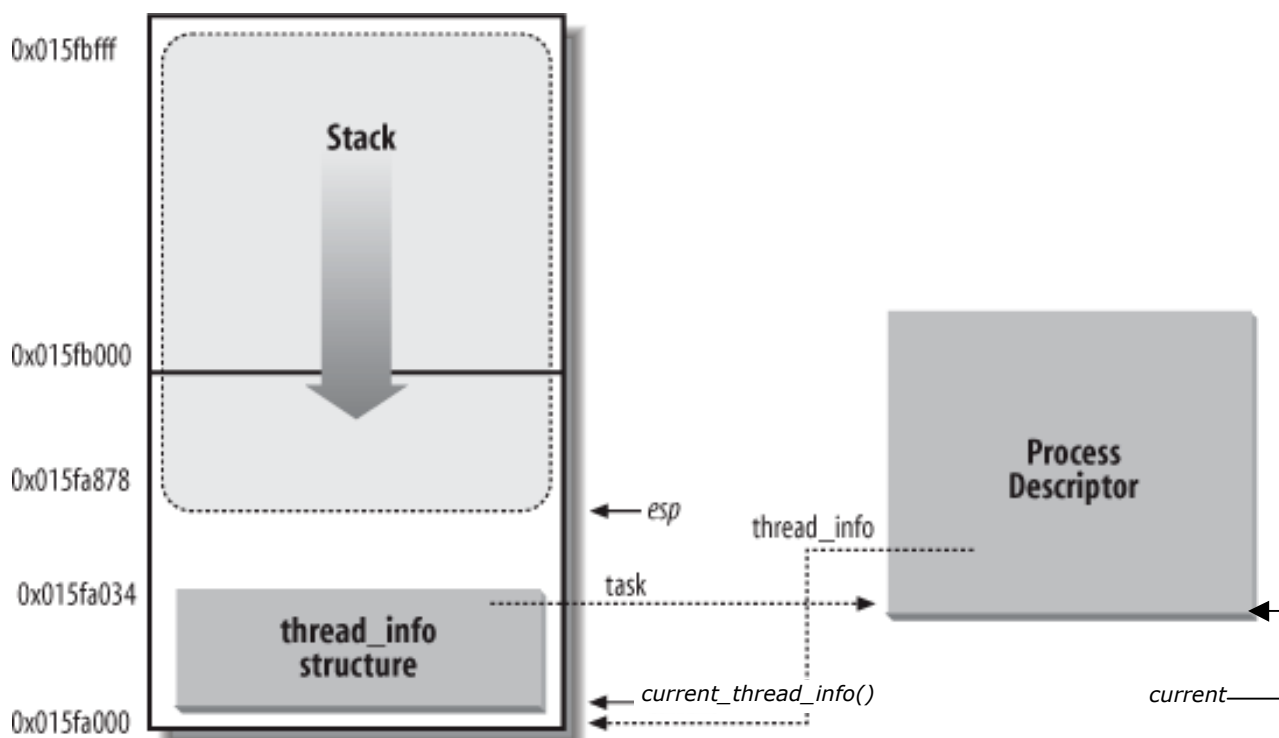
```
long sys_open(const char __user * filename, int flags, int mode);  
long sys_close(unsigned int fd);  
ssize_t sys_write(unsigned int fd, const char __user * buf, size_t  
count) ;  
off_t sys_lseek(unsigned int fd, off_t offset, unsigned int origin) ;  
int sys_clone(struct pt_regs regs) ;
```

Per a obtenir la resta de capceleres de crides del System Call Table podeu trobar-les a:

[http://www.jollen.org/blog/2006/10/linux\\_2611\\_system\\_calls\\_table.html](http://www.jollen.org/blog/2006/10/linux_2611_system_calls_table.html)

### 3.1.5 On guardem les estadístiques?

Al kernel 2.6 la gestió del PCB és diferent que al kernel 2.4 i per tant diferent que a Zeos. A partir del 2.6 s'ha decidit separar el PCB en dos components: `task_struct` i `thread_info`.



**Figura 4** Compartició de 2 pàgines de memòria per la pila de kernel i el `thread_union`<sup>8</sup>

La `task_struct` conté la informació del procés i un punter al `thread_info`.

El `thread_info` és la estructura que comparteix espai amb la pila i conté informació bàsica del thread associada al seu context d'execució i un punter al `task_struct`. La definició del `thread_info` i la definició de la `thread_union` (que és la union del `thread_info` i la pila) a <http://lxr.linux.no>, veieu també la Figura 4.

Per obtenir l'adreça base del `task_struct` s'ha de fer servir la macro `current`. Igualment per obtenir l'adreça base del `thread_info` s'ha de fer servir la rutina `current_thread_info()`. De igual forma si partim de la estructura `thread_info` i volem anar a la seva corresponent `task_struct` haurem de fer servir el camp `task` (de

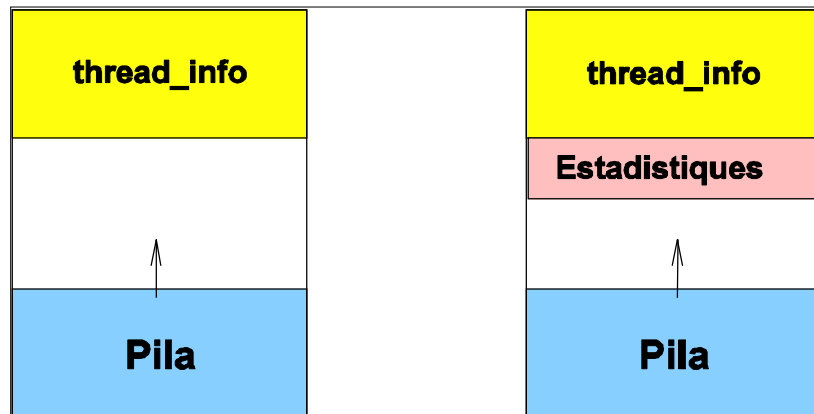
<sup>8</sup> Imatge obtinguda del Understanding the Linux Kernel, Third Edition. Aquesta imatge ha estat modificada per a corregir un error.

l'estructura `thread_info`, Figura 4)

Donat que la informació estadística ocupa poc espai i per associar-la fàcilment amb el procés, guardarem aquestes dades a continuació de la estructura `thread_info`. Per aconseguir-ho, generarem una "nova" estructura `my_thread_info` que contingui la antiga estructura `thread_info` i la estructura de les estadístiques del procés (veure Figura 5).

Per a cada procés és necessari saber si s'han inicialitzat les seves dades estadístiques o no. Penseu que per a la creació d'un nou procés Linux pot aprofitar estructures de dades que estaven associades a processos que ja han mort. Això vol dir que aquestes estructures poden conservar dades relatives a l'antic procés.

Per a que el mòdul pugui detectar si les dades estadístiques estan inicialitzades, podeu afegir un nou camp a l'estructura de les estadístiques que contingui el pid del procés associat a aquelles dades. Si el pid no coincideix amb el procés actual vol dir que aquest procés no ha inicialitzat les dades i, per tant, cal fer-ho i actualitzar el camp del pid.



**Figura 5 On es guarden les estadístiques?**

### 3.1.6 Com es pot comptar el temps?

Per realitzar el comptatge del temps, podeu fer servir la funció `proso_get_cycles` que s'implementa d'aquesta manera:

```
#define proso_rdtsc(low,high) \
__asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))

static inline unsigned long long proso_get_cycles (void) {
    unsigned long eax, edx;

    proso_rdtsc(eax, edx);
    return ((unsigned long long) edx << 32) + eax;
}
```

### 3.1.7 Altres restriccions

Volem que cada procés tingui els seus propis comptadors per les crides a sistema, per tant, s'ha d'inicialitzar a 0 per cada nou procés.

Com a última restricció i per comprovar que tot funciona correctament, es vol que al desinstal·lar el mòdul del sistema, **s'imprimeixin per pantalla les estadístiques del pid que s'haurà entrat com a paràmetre a l'inserir el mòdul.**

Cal evitar la descàrrega del mòdul si hi ha algun procés amb alguna crida interceptada (podeu usar `try_module_get` i `module_put` explicats a la secció 2.1.6).

### 3.1.8 Joc de proves

Per a comprovar el correcte funcionament del vostre mòdul haureu de presentar un joc de proves que ho demostrï. L'esquema d'aquest joc de proves és:

- Inici del joc de proves. Imprimir missatge d'inici.
- Imprimir PID procés actual i bloquejar el procés fins que es premi una tecla.
- (Carregar el mòdul amb el PID del joc de proves)
- Prémer una tecla i continuar el joc de proves.
- Provar totes les crides a sistema monitoritzades.
- Imprimir missatge de finalització del joc de proves. Penjar el procés en un bucle infinit..
- (Descarregar el mòdul, imprimint les estadístiques del procés) Comprovar que els resultats són coherents amb el joc de proves.
- Finalitzar joc de proves.

## 3.2 Mòdul 2: *Accedint a la informació del sistema*

L'objectiu d'aquesta pràctica és construir un mòdul que permeti consultar la informació guardada a la pràctica anterior. Amb el mòdul anterior es monitoritzaran tots els processos creats. Amb aquest mòdul crearem un dispositiu per extreure aquesta informació. Per a crear aquest dispositiu haureu de seleccionar un major i un minor que permetin identificar al manegador i que utilitzareu durant el procés de registre del manegador i per a la creació del fitxer que el representi al sistema de fitxers.

Podrem triar el procés i la crida a sistema de la qual volem accedir a la informació (encara que es continuaran monitoritzant tots els processos i les cinc crides a sistema). Cal crear un nou dispositiu que ens permeti realitzar les següents operacions:

```
ssize_t read (struct file *f, char __user *buffer, size_t s, loff_t
*off);
int ioctl (struct inode *i, struct file *f, unsigned int arg1,
unsigned long arg2);
int open (struct inode *i, struct file *f);
int release (struct inode *i, struct file *f);
```

- **open**. Només es pot obrir el dispositiu un cop i només ho pot fer l'usuari 'root' (euid==0).
- **read**. Al llegir aquest dispositiu retornarà a l'espai d'usuari (buffer) una estructura amb informació sobre la crida a sistema monitoritzada actualment. L'estructura ha d'haver estat creada prèviament. El nombre de bytes a llegir serà el mínim entre s i sizeof(struct t\_info).
- **ioctl**. Amb aquesta crida modificarem el comportament del dispositiu (procés seleccionat, etc).
- **release**. Allibera el dispositiu.

Per defecte, es retorna la informació del procés que ha fet l'open i les estadístiques de la crida open. L'estructura que es retorna a l'usuari és del tipus:

```
struct t_info {
    int num_entrades;
    int num_sortides_ok;
    int num_sortides_error;
    unsigned long long durada_total;
}
```

Per tal de controlar el comportament d'aquest nou dispositiu mitjançant la crida `ioctl` li definirem els següents paràmetres (els valors entre parèntesi, indiquen un valor constant):

- `CANVI PROCES` (`arg1=0`) El paràmetre `arg2` indica, **per referència**, el nou identificador de procés que cal analitzar. Si el punter és `NULL`, vol dir que cal tornar al procés que ha realitzat l'open. Si el procés no existeix cal retornar error.
- `CANVI SYSCALL` (`arg1=1`) Permet canviar la crida de la que consultem les estadístiques. El paràmetre `arg2` indica:
  - `OPEN` (0)
  - `WRITE` (1)
  - `LSEEK` (2)

- CLOSE (3)
- CLONE (4)
- RESET\_VALORS (arg1=2) Posa a zero els valors del procés que s'està analitzant en aquest moment.
- RESET\_VALORS\_TOTS\_PROCESSOS (arg1=3) Posa a zero els valors de tots els processos que s'estan analitzant.

La crida retornarà un zero en cas de que tot vagi correctament, i < 0 en cas d'error (indicant l'error corresponent).

## 4 MONITORITZACIÓ DINÀMICA

---

Es vol afegir més dinamisme als mecanismes d'instrumentació explicats fins ara, per això caldrà modificar els mòduls creats.

### 4.1 Canvis al Mòdul 1

Volem que la monitorització de les crides a sistema es pugui activar/desactivar dinàmicament. Per això farem:

- Per defecte, les cinc crides a sistema estaran instrumentades (igual que al mòdul 1).
- S'afegiran dues funcions noves per activar i desactivar la monitorització de les crides. Aquestes funcions hauran de ser accessibles des del mòdul 2.
- Es penalitzarà no fer ús d'una taula per guardar les adreces de les crides a sistema.

### 4.2 Canvis al Mòdul 2

Activarem/desactivarem les crides des del codi d'usuari ampliant la funcionalitat de la crida `ioctl`, que utilitzarà les funcions del mòdul 1.

- Activar i desactivar les crides a sistema a instrumentar de forma selectiva. Per tant, cal modificar la crida `ioctl` per a rebre dos noves operacions:
  - ACTIVAR\_SYS\_CALL (arg1=4) num : Activa la instrumentació de la crida a sistema *num* (o totes si és un número negatiu)
  - DESACTIVAR\_SYS\_CALL (arg1=5) num : Desactiva la instrumentació de la crida a sistema *num* (o totes si és un número negatiu)
- L'usuari ha de poder introduir de manera fàcil el tipus de crida a instrumentar, per exemple, usant constants.

## 5 ¿QUÈ CAL PRESENTAR ?

---

TOTS els fitxers fonts (incloent Makefiles) que hagueu creat, i els jocs de proves que hagueu usat per provar la vostra pràctica. A més, heu d'entregar un fitxer README describint el vostre joc de proves i les instruccions per a executar-lo.

## 6 REFERÈNCIES

---

[1] By [Jonathan Corbet](#), [Alessandro Rubini](#), [Greg Kroah-Hartman](#) : *Linux Device Drivers*, Third Edition February 2005 .. (<http://lwn.net/Kernel/LDD3/>)

[2] Daniel P. Bovet, Marco Cesati: *Understanding the Linux Kernel*. O'Reilly. November 2005.

[3] <http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN245>

## 7 APÈNDIX: COM EXPORTAR EL SÍMBOL SYS\_CALL\_TABLE

---

Al laboratori el símbol `sys_call_table` ja està exportat però si voleu fer-ho a casa, hauríeu de fer la següent seqüència de passos. Amb això, generareu una nova imatge del kernel amb nom `linux.2.6.xx-proso` on `xx` és la subversió que tingueu.

1. Aneu al director amb el codi font del kernel (suposem `/usr/src/linux`)  
`# cd /usr/src/linux`
2. Modificar el fitxer `arch/i386/kernel/i386_ksyms.c` i afegiu:

```
extern void * sys_call_table[];  
EXPORT_SYMBOL(sys_call_table);
```

3. Editar la variable `EXTRAVERSION` del Makefile per tal que el nom de la imatge de kernel sigui conegut  
`# vi Makefile`  
`...`  
`EXTRAVERSION=-proso`  
`...`
4. Utilitzar el fitxer de configuració actual (el de la imatge actual per exemple) de `/boot/`

- ```
# cp /boot/config-2.6.XXX .config
```
5. Preparar l'entorn:  

```
# make oldconfig
```
  6. Recompilar  

```
# make
```
  7. Recompilar els mòduls del sistema:  

```
# make modules_install
```
  8. Instal·lem la imatge (vmlinuz-2.6.XXX-proso) i els símbols visibles (System.map-2.6.XXX-proso) al directori */boot/* :  

```
# make install
```
  9. Generem un fitxer d'arrencada amb els mòduls que siguin necessaris (ja que sinó no arranca):  

```
# mkinitramfs -o /boot/initrd.img-2.6.XXX-proso 2.6.XXX-proso
```
  10. Modifiquem el fitxer d'arrencada del grub perquè agafi aquesta nova imatge */boot/grub/menu.lst*.  

```
# vi /boot/grub/menu.lst
```
  11. Cal modificar els camps següents per tal que cridin a la nova imatge i el nou fitxer *initrd*:  

```
title  
kernel  
initrd
```

## 8 CRIDA DMESG

---

La comanda *dmesg* serveix per a imprimir i controlar "the kernel ring buffer".

El format es el següent: `dmesg [ -c ] [ -n level ] [ -s bufsize ]`

-c Esborrar el "kernel ring buffer després de llegir-ho

-s bufsize Estableix el tamany del "kernel ring buffer"

-n Estableix el nou nivell de "log" del missatges de sistema . No tira els antics.