# CONCEPTES AVANÇATS DE SISTEMES OPERATIUS (CASO)

Facultat d'Informàtica de Barcelona, Dept. d'Arquitectura de Computadors, curs 2014/2015 – 1Q

## Pràctiques de laboratori Mòduls i dispositius

### Material

La vostra instal·lació de Linux.

Una nova instal·lació que feu en una màquina virtual amb Qemu. Us anirà bé per evitar reboots lents i possibles pèrdues d'informació. A banda, podreu usar el GDB per depurar el codi del kernel (veieu la secció "Com depurar mòduls i el kernel de Linux", al final d'aquest document).

En el sistema de treball (nadiu o QEMU), els headers de linux instal·lats a /usr/src/linux-headers-<versio-del-kernel> i apuntats correctament des de </lib/modules/versio-del-kernel>/build.

### Mòduls del kernel de Linux

(Veure el tema de Mach/Eines/suport hardware, transp. 28-30)

Els mòduls de kernel són parts de la implementació del sistema operatiu que s'han extret, usen un conjunt limitat de rutines del kernel, se'ls ha definit una interfície, i es poden carregar i descarregar dinàmicament.

A més, se'ls afegeixen dues funcions de suport a la càrrega i a la descàrrega. Quan un mòdul es carrega, se li dóna l'oportunitat d'inicialitzar-se a través d'una crida a la seva funció d'inicialització. Aquesta funció s'ha de declarar d'una manera especial, per exemple:

```
static int inicialització (void)
{
     // realitzar les tasques necessàries per inicialitzar el mòdul
     // pot fer printk per mostrar missatges al kmsg
     printk( KERN_DEBUG "Hola món\n");
}
// cal indicar que és la funció d'inicialització del mòdul
module_init(inicialització);
```

Per donar suport a la descàrrega, hi ha el mateix tipus de funció, marcada com a `module_exit`:

```
static int finalització (void)
{
printk (KERN_DEBUG "Acabo!!!\n");
}
```

```
module_exit(finalització);
```

Veieu els fitxers de suport que acompanyen a l'enunciat d'aquesta pràctica en el fitxer moduls-suport.tar.bz2:

- Makefile
- interface.c
- implementation.c

Copieu aquests fitxers al vostre sistema i...

– Useu `$ make` per compilar l'exemple de mòdul.

– Comproveu que després de la compilació, teniu un fitxer `mymodule.ko`, com aquest:

```
-rw-rw-r--   1 xavim xavim 3871 Oct 16 10:44 mymodule.ko
```

– Inserteu el mòdul dins el kernel:

```
$ sudo insmod mymodule.ko
```

– Surt algun missatge al vostre terminal? Dóna algun error?

– Comproveu amb la comanda $ lsmod que el mòdul s'ha carregat:

```
$ lsmod
$ lsmod | grep mymodule
```

– Comproveu amb la comanda `$ dmesg` que el mòdul ha escrit el missatge de benvinguda entre els missatges del kernel.

– Descarregueu el mòdul:

```
$ sudo rmmod mymodule
```

– Comproveu també que el mòdul ha escrit el missatge de despedida.

## Implementació de dispositius de tipus caràcter

La funcionalitat més habitual per un mòdul és proporcionar el servei d'accés a algun dispositiu. En aquesta pràctica veurem d'un senzill dispositiu de tipus caràcter.

Per a fer-ho, obriu el segon fitxer de suport chardev.tar.bz2 i mireu el contingut dels fitxers per veure els següents punts:

1. Per donar d'alta un dispositiu cal triar primer els seus números identificadors, major i minor. Per exemple, agafem major=230 i minor=3. (interface.c)
2. Comproveu que al directori /dev no teniu cap dispositiu amb major 230. Si hi fos, canvieu el número per un que no existeixi.
3. Enregistrar i eliminar el dispositiu de caràcter. Per tal que el sistema operatiu sàpiga que el codi del mòdul implementa el vostre dispositiu 230, cal fer l'entregistrament amb la funció:

```
res = register_chrdev (MY_MAJOR, "nom-disp-CASO", &operacions);
```

on:

- MY_MAJOR pot ser un #define que pren per valor 230.
- <nom-disp-CASO> és la cadena amb el nom que voleu donar-li.
- operacions és la variable que conté l'estructura amb els punters a funcions de les operacions.

Aquest enregistrament es fa a la funció d'inicialització del mòdul. (interface.c: mychardrv_init)

També cal eliminar el dispositiu del sistema quan el mòdul es descarrega, i per fer això s'aprofita la funció de finalització del mòdul: (interface.c: mychardrv_exit)

```
unregister_chrdev(MY_MAJOR, "nom-disp-CASO");
```

Mireu la taula d'operacions i les crides a aquestes funcions (register_chrdev i unregister_chrdev) en el mòdul.

```
static const struct file_operations operacions = {
            .owner          = THIS_MODULE,
            .open           = mychardrv_open,
            .release        = mychardrv_close,
            .read           = mychardrv_read,
            .unlocked_ioctl   = mychardrv_ioctl,
            .write          = mychardrv_write,
};
```

4. Carregueu el mòdul i comproveu que s'ha carregat correctament:
   - No ha donat errors la càrrega
   - El mòdul apareix a la llista de l'`lsmod`.
   - El dispositiu apareix a `/proc/devices`

```
Character devices
  ...
  230 nom-disp-CASO
  ...
```

## Operacions per llegir el comptador d'obertures

A continuació veurem la implementació de tres de les operacions del dispositiu. Aquests operacions poden retornar els errors típics de UNIX/Linux, com a números negatius. Per exemple, un error `ENODEV`, per indicar que el dispositiu no està present, es retornaria com `-ENODEV`.

- open:

```
static int obrir(struct inode * inode, struct file * file)
```

veieu-la a interface.c: mychardrv_open.

La crida open incrementa un comptador de vegades que el dispositiu d'ha obert. Els paràmetres d'open són punters a l'i-node i fitxer que representen el descriptor de dispositiu obert. No els farem servir en la nostra implementació.
*Opcionalment: l'operació d'obrir pot comprovar si el fitxer que s'està obrint correspon al fitxer de dispositiu amb major=230 i minor=3. Pista: useu `imajor(inode)` i `iminor(inode)`.*

- release (que correspon a la funció de `close`):
  ```
  static int tancar(struct inode * inode, struct file * file)
  ```
  amb els mateixos paràmetres que l'open, decrementa el comptador de vegades que el dispositiu s'ha obert.

- read:
  ```
  static ssize_t llegir(struct file * file, char __user * buffer,
                                            size_t size, loff_t * offset)
  ```
  Fixeu-vos que els paràmetres són els de la crida a sistema `read`, afegint-hi l'offset que habitualment s'indica en la crida a sistema `lseek`. Si la mida que la lectura demana és major o igual al sizeof (comptador-d'obertures), se li retorna a l'usuari el comptador d'obertures, i la mida llegida correcta. Altrament retorna un error.

Proveu, llegint del dispositiu 10 cops i imprimint els números que torna. Pista: podeu usar la comanda `dd` per llegir un número limitat de vegades, i la comanda `od -v -X` per transformar la informació tornada pel dispositiu en una seqüència de números imprimibles.

## Còpia d'informació cap al / des del nivell usuari

Per treure dades cap a nivell usuari, Linux té la crida interna:

```
static inline long copy_to_user(void __user *to,
                                const void *from, unsigned long n);
```

Aquesta crida copia la informació apuntada per `from` a l'espai d'adreces del procés d'usuari actual, a l'adreça indicada per `to`. La quantitat d'informació copiada serà d'`n` bytes. La crida retorna el número de bytes que no s'han pogut copiar a l'espai de l'usuari. Si aquest valor de retorn és 0, llavors tot ha anat bé. En cas d'error, se suposa que no es pot escriure a l'espai de l'usuari per culpa d'una adreça invàlida.

La crida corresponent per copiar informació de l'usuari cap al sistema és semblant i també retorna el número de bytes que no s'han pogut copiar cap al sistema, zero (correcte) o el codi de l'error com a número negatiu:

```
static inline long copy_from_user(void *to,
                const void __user * from, unsigned long n);
```

Per tenir disponible la definició d'aquestes dues crides cal incloure el fitxer `<linux/uaccess.h>`.

## Operacions de lectura/escriptura d'strings

Ara completarem les operacions anteriors i introduïm les de `write` i `ioctl`:

- write

```
static ssize_t escriure(struct file * file, char __user * buffer,
                                            size_t size, loff_t * offset)
```

  Rep els mateixos paràmetres que `llegir`.

- unlocked_ioctl

```
static long control(struct file * file, unsigned int command,
                                           unsigned long argument)
```

  Permet canviar les característiques del fitxer o dispositiu. Farem servir aquesta operació per canviar el lloc del fitxer on es llegeix o s'escriu.

Una seqüència d'operacions com la següent (on hem omès la comprovació d'errors per claretat):

```
int fd = open ("mychardrv", O_RDWR, 0);
res = ioctl (fd, SET_STRING_ID, 1);                  // selecciona string #1
res = write(fd, "Hi, this is a first test\n", 26);    //    i l'escriu
res = ioctl (fd, SET_STRING_ID, 2);                  // selecciona string #2
res = write(fd, "Hi, this is a second test\n", 27);   //    i l'escriu
res = ioctl (fd, SET_STRING_ID, 1);                  // selecciona string #1
res = read(fd, cbuf, 10);                            //    i la llegeix
cbuf[res] = '\0';                                    // assegura que acaba bé
printf ("Read from ID 1: '%s' (%d bytes)\n", cbuf, res);   // la treu al tty
res = ioctl (fd, SET_STRING_ID, 2);                  // selecciona string #2
res = read(fd, cbuf, 100);                           //    i la llegeix
cbuf[res] = '\0';
printf ("Read from ID 2: '%s' (%d bytes)\n", cbuf, res);   // la treu al tty
close(fd);
```

Donaria la sortida següent:

```
Read from ID 1: 'Hi, this i' (10 bytes)
Read from ID 2: 'Hi, this is a second test
' (27 bytes)
```

Com es pot veure a la sortida, un fitxer del nostre dispositiu pot guardar un conjunt d'strings identificats per un enter (ID). L'operació ioctl es fa servir per canviar l'ID actiu, en el qual s'escriurà o es llegirà en les següents operacions de read/write. Dues escriptures seguides sobre el mateix identificador, fan que es guardi només la segona. Es pot llegir un mateix string més d'un cop i sempre es retornarà el mateix text, si no s'ha fet una operació d'escriptura entremig.

## Gestió de memòria

Per a implementar aquesta segona versió, haureu de fer servir les crides internes de Linux per demanar i alliberar memòria:

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

static inline void * kzalloc (size_t size, gfp_t flags);

static inline void   kfree   (const void * block);
```

Al demanar memòria podem usar alguns flags interessants:

– GFP_KERNEL, per demanar memòria en l'espai del kernel

– GFP_USER, per demanar memòria en l'espai de l'usuari

– GFP_ZERO, per demanar memòria que estigui netejada a zero

## Com depurar mòduls i el kernel de Linux

A part de córrer el vostre sistema (“*guest”*), si useu QEMU, aquest també permet que utilitzeu gdb des de la vostra màquina (“*host”*) per depurar el kernel que corre en el guest. Per a informació genèrica sobre l'ús de gdb, llegiu el document “Debugging Crash Course”, que podeu trobar al costat d'aquest enunciat.

### Arrencada del sistema

Per a poder debugar el kernel del *guest*, cal que arrenqeu el QEMU amb els següents arguments:

- *-s | -gdb tcp::1234*

  Permet que el gdb es connecti amb el QEMU a través del port TCP número 1234. Podeu canviar el port 1234 per un altre, si ho preferiu.

- -*snapshot* [opcional]

  No escriu cap canvi al disc de la màquina virtual (així eviteu comprovacions de disc el proper cop que arrenqueu, en cas que s'apagui malament la màquina). Permet que mateu QEMU (Ctrl+C) sense esperar a apagar la màquina *guest* correctament.

### Depuració de mòduls

Tant bon punt hagueu arrencat el gdb, li heu de dir que es connecti al QEMU (que ja hauria d'estar executant-se):

```
$ gdb
...
(gdb) set architecture i386:x86-64
The target architecture is assumed to be i386:x86-64
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xffffffff8100b02d in ?? ()
(gdb)
```

Un cop connectat, el gdb pararà l'execució del sistema guest, esperant una nova comanda.

Quan arrenqueu el gdb, aquest no coneix els continguts (variables, funcions), ni de Linux, ni del mòdul que voleu depurar. Si voleu que el gdb conegui la informació de Linux, podeu veure com s'obté a la secció "Depuració del kernel de Linux", més endavant.

Com que els mòduls del kernel es poden carregar dinàmicament i en qualsevol adreça (igual que les llibreries dinàmiques), cal descobrir quina és aquesta adreça. Un cop descoberta, se li pot dir al gdb que, a més del kernel, ha de conèixer sobre l'existència d'un mòdul en una adreça concreta.

Per a conèixer l'adreça a on un mòdul ha estat carregat, podeu mirar els continguts del fitxer */proc/modules*. La primera columna conté el nom del mòdul, i la última columna conté l'adreça a on el kernel ha carregat el mòdul.

**ATENCIÓ**: El fitxer l'heu d'accedir com a root, ja que com a usuari normal, el sistema amaga aquesta informació.

Per exemple, si teniu el mòdul `mymodule.ko` (*mymodule* a la primera columna del fitxer */proc/modules*) carregat a l'adreça *0xabcd*, el podreu depurar correctament amb la següent comanda del gdb:

```
(gdb) add-symbol-file mymodule.ko 0xabcd
(gdb)
```

Un cop fet això, ja podeu depurar el mòdul com si fós un programa normal, tot i que quan cridi a Linux no veureu correctament els símbols.

Podeu també veure el codi font del mòdul si poseu els fitxers font d'aquest en el directori corresponent del host. En el següent exemple, es tracta de */home/xavim/moduls-i-dispositius/charmod*, tal i com el gdb us mostra:

```
(gdb) continue
Continuing.
[New Thread 2]
[Switching to Thread 2]

Breakpoint 1, mychardrv_write (f=<optimized out>,
    address=<optimizedout>, size=<optimized out>, offset=<optimized out>)
    at /home/xavim/moduls-i-dispositius/charmod/interface.c:84
84          /home/xavim/moduls-i-dispositius/charmod/interface.c: No such
file or directory.
```

## Depuració del kernel de Linux

**ATENCIÓ**: Per simplicitat, és recomanable que utilitzeu el mateix sistema en el guest i en el host. Tingueu també en compte que els números de versió del kernel poden canviar en la vostra instal·lació.

Un cop la màquina *guest* està arrencada, arrenqueu el gdb en el *host* passant-li com a binari a depurar el fitxer binari del kernel.

En el cas d'un sistema Debian, si esteu corrent el kernel precompilat que ve amb el paquet *linux-image-3.10-3-amd64*, podeu instal·lar el paquet *linux-image-3.10-3-amd64-dbg*. Aquest últim paquet proporciona el fitxer */usr/lib/debug/vmlinux-3.10-3-amd64*, que és el que podeu utilitzar amb el gdb.

En el cas d'un sistema Ubuntu, heu d'afegir el repositori de debug, tal i com indica aquest document[1] i executar:

```
$ sudo apt-get update
$ sudo apt-get install linux-image-`uname -r`-dbgsym
```

Si compileu el vostre propi kernel, es tracta del fitxer *vmlinux* que es genera a l'arrel del vostre directori de compilació del kernel. En aquest cas, caldrà copiar el *vmlinux* del guest al host.

Com abans, tant bon punt hagueu arrencat el gdb amb un binari del kernel que contingui informació de

1 http://askubuntu.com/questions/197016/is-there-a-kernel-package-that-contains-symbols

depuració, li heu de dir al gdb que es connecti al QEMU (que ja hauria d'estar executant-se):

```
$ gdb /usr/lib/debug/vmlinux-3.10-3-amd64
...
Reading symbols from /usr/lib/debug/boot/vmlinux-3.10-3-amd64...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0xffffffff8100b02d in native_safe_halt () at /build/.../irqflags.h:49
49       in /build/linux-BpzSEt/linux-3.10.11/arch/x86/include/asm/
irqflags.h
(gdb)
```

Tot i que el binari conté informació de depuració, el gdb és incapaç de trobar el codi font del kernel. Per a això cal que instal·leu el paquet *linux-source-3.10* i que descomprimiu el codi font al directori on vulgueu:

```
$ cd ${HOME}/
$ tar xf /usr/src/linux-source-3.10.tar.xz
```

Un cop fet això, li heu de dir al gdb a on pot trobar el nou codi font (ja que no necessàriament volem posar-lo a on ell l'espera trobar, */build/linux-BpzSEt/linux-3.10.11*):

```
(gdb) set substitute-path /build/linux-BPzSEt/linux-3.10.11 ${HOME}/li
nux-source-3.10
```

En el cas de què també vulgueu depurar la funció d'inicialització del mòdul, el procés es complica un pèl més, ja que necessitareu conèixer l'adreça a la que el mòdul ha estat carregat abans de què el kernel acabi de realitzar-ne la càrrega (que és quan l'usuari pot executar una comanda per veure els continguts del fitxer */proc/modules*).

Per a això, cal que poseu un “breakpoint” a la funció `do_one_initcall` del kernel, que és just abans de cridar la funció d'inicialització del mòdul, però un cop ja s'ha carregat el mòdul en una adreça de memòria concreta (la que volem conèixer). Un cop s'activi el breakpoint (després d'haver executat la comanda `insmod` en el guest), executeu la comanda `lsmod` en el gdb. Aqyesta comanda proporciona un informació similar a la que hi ha en el fitxer */proc/modules*, però ho fa a través d'utilitzar les comandes del gdb per mirar els continguts de la memòria del guest.

Podeu trobar la comanda `lsmod` en el fitxer `.gdbinit` que es proporciona amb el codi de suport de la pràctica (que heu de copiar en el directori *home* del vostre usuari).