open wide
INGENIERIE

JDEV2013

Les pilotes en mode caractère

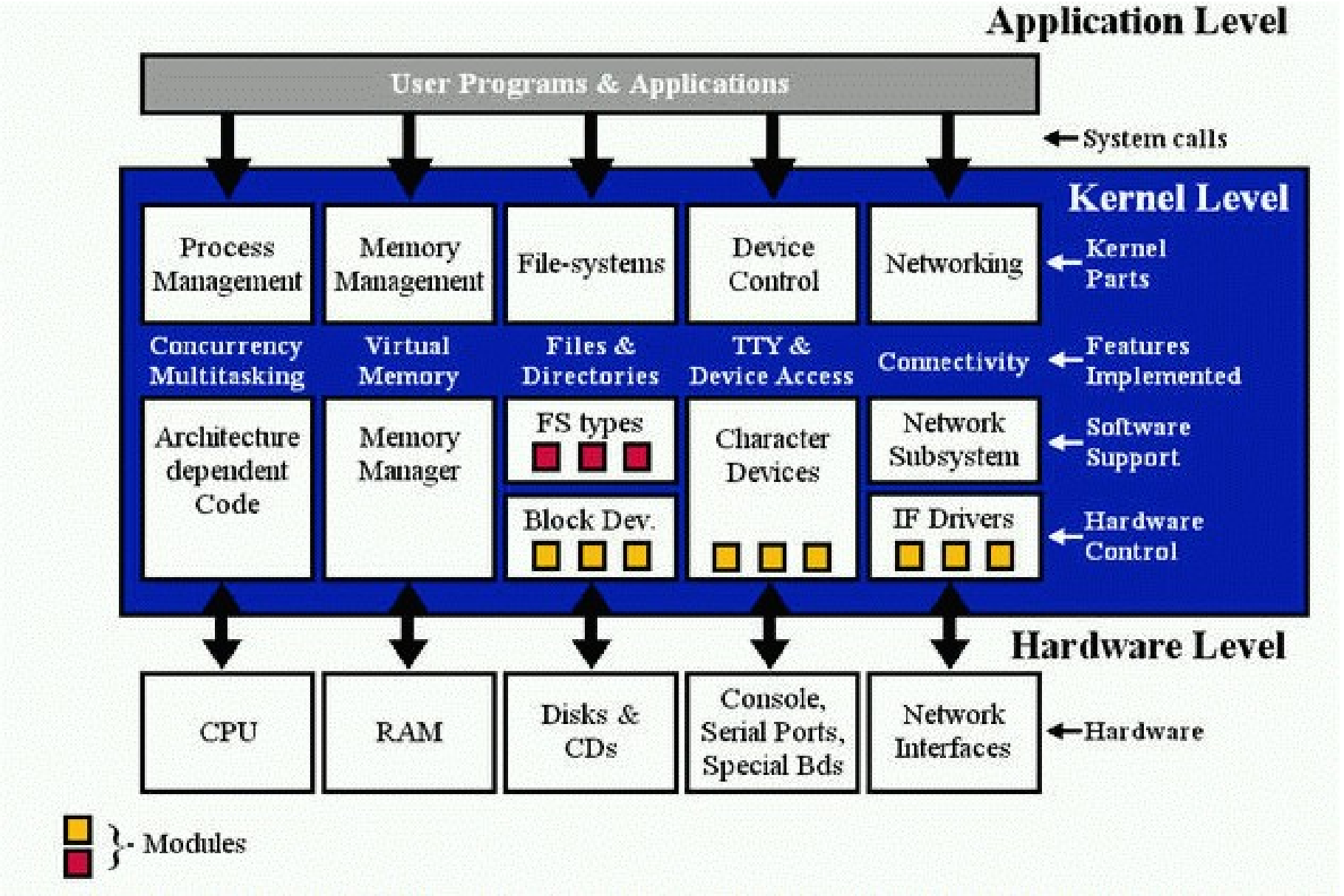
Pierre Ficheux (pierre.ficheux@openwide.fr)

Septembre 2013

Les pilotes Linux

- Interface entre un programme utilisateur et le matériel
- Accès direct aux ports d'I/O possible en mode utilisateur sur certaines architectures `ioperm()`, `inb()`, `outb()`
- Très proche de l'architecture UNIX développée en 1970 !
- API relativement simple par rapport à d'autres OS (!)
- Conforme à l'API du noyau Linux → pas de *standard*
- Un pilote est soit :
  - un module *dynamique* `.ko`
  - compilé « en statique » avec les sources du noyau Linux (dans `bzImage`)

Structure du noyau et des pilotes



Différents types de pilotes

- Pilote caractère (*character driver*)
  - 90% des cas
  - Les échanges avec le périphérique ne sont pas « bufferisés »
- Pilote bloc (*block driver*)
  - Pour les périphériques de stockage (disque, mémoire, etc.)
  - On traite des blocs de données: 512, 1024, 2048 octets
- Pilote réseau (*network driver*): API spécifique utilisant la structure `sk_buff`

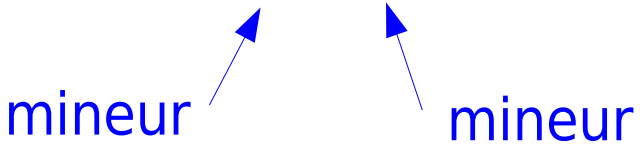
Structure d'un pilote

- Approche « objet »
- Utilise des *méthodes* pour contrôler le périphérique, correspondant aux *appels systèmes*
  - `open()` : ouverture
  - `release()` : fermeture
  - `read()` : lecture
  - `write()` : écriture
  - `ioctl()` : contrôle d'entrée/sortie
  - ...
- Les méthodes sont optionnelles sauf `open()` et `release()`

Interface avec l'espace utilisateur

- Le pilote est chargé dans l'espace mémoire du noyau (*kernel space*)
- Identifié par 2 entiers: *majeur* et *mineur*
  - majeur: type de périphérique (exemple: 4 pour *tty*)
  - mineur: instance (exemple: premier UART = 64)
- Dans l'espace utilisateur on utilise un répertoire de fichiers spéciaux: `/dev`
- Chaque fichier a un *majeur* et un *mineur*

```
crw-rw---- 1 root uucp 4, 64 sep  5  2007 /dev/ttyS0
```



Interface avec l'espace utilisateur, suite

- Les pilotes réseau n'utilisent pas ce principe (pas d'entrée dans `/dev`)
- Dans un programme, on ouvre le fichier spécial et on obtient un «descripteur» (fd = *file descriptor*)

  `fd = open ("/dev/ttyS0", O_RDWR)` → appel `open()` dans le pilote
- Ensuite on peut effectuer des *appels système* sur `fd`

  `read (fd, buf, sizeof(buf))` → appel `read()`

  `close (fd)` → appel `release()`

Limites du principe majeur/mineur

- Choix du majeur/mineur imposé, voir fichier `Documentation/devices.txt`
- Pour les pilotes ajoutés, allocation *dynamique* de majeur ou mineur si le majeur est défini (*misc* driver, voir `/proc/misc`)
- Couches pour faciliter la manipulation de `/dev`
  - devfs (version 2.4): identification par nom, obsolète
  - udev: (version 2.6 et supérieur): création dynamique des entrées de `/dev` (démon `udevd`), basé sur le pseudo système de fichiers `/sys`

API de programmation

- Basée sur l'API des modules
- Ajout de la déclaration des méthodes dans la structure `file_operations` (pilote caractère) définie dans `linux/fs.h`

```
static struct file_operations mydriver_fops = {
    .owner    =  THIS_MODULE,
    .read     =  mydriver_read,
    .write    =  mydriver_write,
    .open     =  mydriver_open,
    .release =  mydriver_release,
    .llseek   =  mydriver_llseek,
};
```

Allocation du majeur

- API « classique » (2.4) basée sur :
  - `register_chrdev()`
  - `unregister_chrdev()`
- Gestion des « classes » répertoriées dans `/sys/class` (et utilisées par UDEV) :
  - `class_create()`
  - `device_create()`
- Nouvelle API (2.6 et plus) basée sur les fonctions *CDEV*
  - `cdev_init()`
  - `cdev_add()`

Fonction d'initialisation du pilote

- Dans la fonction d'initialisation, on affecte dynamiquement le majeur
- On peut obtenir la valeur dans `/proc/devices` → utiliser `mknod`
- Déclaration des paramètres

```
static int major = 0; /* Major number */
module_param(major, int, 0644);
MODULE_PARM_DESC(major, "Static major number
(none = dynamic)");
```

Fonction d'initialisation du pilote, suite

```
static int __init mydriver1_init(void) {
  int ret;
  ret = register_chrdev(major, "mydriver1",
&mydriver1_fops);
  if (ret < 0) {
     printk(KERN_WARNING "mydriver1: unable to get a
major\n");
     return ret;
  }
  if (major == 0)
    major = ret; /* dynamic value */


  printk(KERN_INFO "mydriver1: successfully loaded with
major %d\n", major);
  return 0;
}
```

open wide INGENIERIE

Fonction de libération du pilote

```
static void __exit mydriver1_exit(void)
{
  unregister_chrdev(major, "mydriver1") ;
  printk(KERN_INFO "mydriver1: successfully
unloaded\n");
}

module_init(mydriver1_init);
module_exit(mydriver1_exit);
```

Utilisation du pilote *misc*

- Principe: le majeur est défini (valeur 10) → utilise la classe *misc*
- Le mineur est alloué dynamiquement
- La liste des pilotes chargés est disponible dans `/proc/misc`
- Avantage: directement géré par *udev* (création automatique de l'entrée dans `/dev`) → possible également en utilisant `/sys/class`

Les méthodes `open()` et `release()`

- Associées aux appels système `open()` et `close()` dans l'espace utilisateur

```
static int mydriver1_open (struct inode *inode, struct file *file) {
   printk(KERN_INFO "mydriver1: open()\n");
   return 0;
}

static int mydriver1_release (struct inode *inode, struct file *file) {
   printk(KERN_INFO "mydriver1: release()\n");
   return 0;
}
```

```
fd = open ("/dev/mydriver1", O_RDWR);
```

```
close (fd);
```

Les méthodes `read()` et `write()`

- Associées aux appels système `read()` et `write()` dans l'espace utilisateur

```
static ssize_t mydriver1_read(struct file *file, char
*buf, size_t count, loff_t *ppos) {
  printk(KERN_INFO "mydriver1: read()\n");
  return count;
}
```

```
n = read (fd, buf, n);
```

```
static ssize_t mydriver1_write(struct file *file, const
char *buf, size_t count, loff_t *ppos) {
  printk(KERN_INFO "mydriver1: write()\n");
  return count;
}
```

```
n = write (fd, buf, n);
```

Test du pilote minimal

- Fichier `Makefile` identique à celui d'un module
- Insertion par `insmod`
- Lecture du majeur dynamique dans `/proc/devices`

```
# mknod /dev/mydriver1 c 253 0
```

- Test d'ouverture puis fermeture

```
# < /dev/mydriver1
```

- Test d'écriture puis lecture

```
# echo x > /dev/mydriver1
# cat /dev/mydriver1
```

- Si on utilise *misc* ou une classe → création automatique de l'entrée

Utilisation des classes, 1/2

- Ajout d'une entrée dans `/sys/class` pour créer le fichier spécial par UDEV
- Procédure à suivre :
  - Création d'une classe par `class_create()`
  - Ajout du device à la classe par `device_create()`
  - Suppression par `device_destroy()` et `class_destroy()`

- On peut utiliser une classe existante créée par un module d'initialisation (exemple : `tty.h`)

```
extern struct class *tty_class ;
```

Utilisation des classes, 2/2

```
static struct class *mydriver1_class;

static int __init mydriver1_init(void) {
   mydriver1_class = class_create (THIS_MODULE, "myclass");
   device_create (mydriver1_class, NULL, MKDEV(major, 0),
NULL, "mydriver1");
   ...
}

static void __exit mydriver1_exit(void) {
   device_destroy (mydriver1_class, MKDEV(major, 0));
   class_destroy (mydriver1_class);
   ...
}
```

Prototypes

```
struct class * class_create (
  struct module *owner,
  const char *name
);

struct device * device_create (
  struct class *class,
  struct device *parent,
  dev_t   devt,
  const char *fmt
  const char *name;
);
```

Utilisation de CDEV

- Méthode « standard » pour les « vrais » pilotes 2.6 et supérieur
- Allocation de NB mineurs par :

```
alloc_chrdev_region (&maj, 0, NB, "mydriver") ;
```

nombre de mineurs

1er mineur

- Pour chaque mineur, on utilise :

```
cdev_init()
cdev_add()
```

- Plus la gestion des classes :

```
class_create()
device_create() → /dev/mydriver0, 1, ...
```

- Voir l'exemple complet dans le répertoire `cfake`

Prototypes

```
int alloc_chrdev_region (dev_t *dev, unsigned
baseminor, unsigned count, const char *name);


void unregister_chrdev_region (dev_t from,
unsigned count);


void cdev_init (struct cdev *cdev, const struct
file_operations *fops);


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


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

Échange de données espaces utilisateur/noyau

- Le pilote et le programme utilisateur n'utilisent pas le même espace de mémoire → `access_ok()`
- Utilisation de macros dédiées aux échanges :

```
copy_from_user (void *to, void *from,
unsigned long size)
```
← utilisé dans `write()`

```
copy_to_user (void *to, void *from,
unsigned long size)
```
← utilisé dans `read()`

- On utilise également `put_user(variable, addr)` et `get_user(variable, addr)`

```
int type;

put_user (type, (char __user*)arg);

get_user (type, (char __user*)arg);
```

- Prototypes définis dans `asm/uaccess.h`

Exemple avec `write()`

```
static ssize_t mydriver3_write(struct file *file,
const char *buf, size_t count, loff_t *ppos)
{
  size_t real;
  real = min((size_t)BUF_SIZE, count);
  if (real)
    if (copy_from_user(buffer, buf, real))
      return -EFAULT;
  num = real; /* Destructive write (overwrite
previous data if any) */
  printk(KERN_DEBUG "mydriver3: wrote %d/%d chars
%s\n", real, count, buffer);
  return real;
}
```

position courante (0)

open wide INGENIERIE

Exemple avec read()

```
static ssize_t mydriver3_read(struct file *file, char
*buf, size_t count, loff_t *ppos)
{
  size_t real;
  real = min(num, count);
  if (real)
    if (copy_to_user(buf, buffer, real))
      return -EFAULT;
  num = 0; /* Destructive read (no more data after a
read) */
  printk(KERN_DEBUG "mydriver3: read %d/%d chars
%s\n", real, count, buffer);
  return real;
}
```

Prise en compte de la position courante (ppos)

- Le paramètre `ppos` n'est PAS traité dans l'exemple car il vaut 0 par défaut → le tampon doit être lu par un seul appel `read()`
- `ppos` devrait être modifié par le pilote lors de la lecture/écriture du tampon → `read()` ou `write()`
- Il peut être modifié dans la méthode `llseek()` du pilote

  → `lseek()` coté espace utilisateur :

```
lseek (fd, offset, SEEK_SET);
```

- Voir l'exemple du répertoire `mydriver3_ppos` ou le pilote PCI

Test de l'échange

- Insertion du module et écriture

```
# insmod mydriver3.ko
# echo -n salut > /dev/mydriver3
```

- Cela provoque l'affichage suivant dans les traces du noyau :

```
mydriver3: open()
mydriver3: wrote 5/5 chars salut
mydriver3: release()
```

- Lecture du tampon:

```
$ cat /dev/mydriver3
```

Affichage → `"salut"`

La méthode `ioctl()`

- Tout ce que l'on ne fait pas avec `read()` ou `write()`
- Configuration du périphérique (fixer le début du port série: B115200)
- Lecture de l'état du périphérique
- Utilise également `copy_to_user()` et `copy_from_user()`
- Entrée ajoutée à la structure `file_operations`

```
.ioctl   =              mydriver5_ioctl,
.unlocked_ioctl   =   mydriver5_ioctl, // 2.6.36 +
```

- Prototypes

```
int mydriver5_ioctl (struct inode *inode, struct file
*file, unsigned int cmd, unsigned long arg)
```

```
int mydriver5_unlocked_ioctl (struct file *file, unsigned
int cmd, unsigned long arg)
```

2.6.36 et +

La méthode `ioctl()`, écriture

```
switch (cmd)
{
  case SET_MODE :
    switch (arg) {
    case MODE1 :
      my_mode = MODE1;
      break;
    case MODE2 :
      my_mode = MODE2;
      break;
    default :
      printk(KERN_WARNING "mydriver5: arg %x
unsupported in the SET_MODE ioctl command\n",
(int)arg);
      return -EINVAL;
    } /* SET_MODE */
```

La méthode `ioctl()`, lecture

```
case GET_MODE:  /* Send my_mode value to user space
*/
    if  (put_user (my_mode, (long __user *)arg))) {
      printk(KERN_WARNING "mydriver5: put_user
error\n");
      return -EFAULT;
     }
     break;


  default :
    printk(KERN_WARNING "mydriver5: 0x%x unsupported
ioctl command\n", cmd);


    return -EINVAL;
} /* End of switch */
```

Interruptions et événements

- Ce point n'est pas spécifique aux pilotes en mode caractère !
- Les événements d'E/S sont souvent asynchrones (ex: clavier, arrivée de paquets réseau, etc.)
- Il existe deux méthodes pour superviser les E/S:
  - l'attente active/scrutation (*polling*)
  - le traitement par interruptions
- Sous Linux, le traitement d'une interruption s'effectue *uniquement* en espace noyau

Interruptions et événements, suite.

- Dans le mode *polling*, on relâche le CPU avec la fonction `schedule()` après chaque test infructueux:

```
for ( ; ; ) {
    if (evenement) break;
    schedule() ;
}
```

- Dans le mode interruptible, le processus appelant est endormi et placé dans une file d'attente
- C'est le gestionnaire de l'interruption attendue qui sera chargé de le réveiller (ainsi que tous les autres processus en attente dans la file)

Traitement de l'interruption

- Les gestionnaires d'interruptions (*interrupt handler*):
  - doivent être rapides ;
  - ne doivent pas appeler des routines qui peuvent endormir le processus (ex: `kmalloc( )` non atomique)
- Pour ces raisons, la gestion des interruptions est découpée en deux parties:
  - une partie rapide et non interruptible (gestionnaire d'interruptions *top-half*)
  - une partie lente placée dans une file de tâches (*bottom-half*)
- Les *bottom-halves* ne sont pas obligatoires.

Traitement de l'interruption, suite.

- Un gestionnaire d'interruptions est déclaré grâce à la fonction `request_irq()`
- Il est libéré grâce à la fonction `free_irq()`
- Prototypes dans `linux/sched.h` et `linux/interrupt.h`

```
int request_irq(unsigned int irq, irqreturn_t
(*handler)(int, void *), unsigned long flags /*
IRQF_SHARED */, const char *device, void
*dev_id);


void free_irq(unsigned int irq, void *dev_id);
```

Traitement de l'interruption, suite.

- Quand l'interruption `irq` survient, la fonction `handler()` est appelée
- Le champ `dev_id` sert à identifier les périphériques en cas de partage d'interruption (`IRQF_SHARED`, ex: PCI)
- Il est en général employé pour transmettre au gestionnaire d'interruptions une structure spécifique au périphérique.
- Si IRQF_SHARED, le champ ne DOIT pas être à `NULL` !
- La liste des IRQ déjà déclarées est disponible dans `/proc/interrupts`

Top-half et bottom-half

- Le code de retour du handler doit être `IRQ_NONE` ou `IRQ_HANDLED` suivant l'état du matériel (si une irq est vraiment disponible)
  - `IRQ_NONE`: irq non traitée => traitée par un autre handler
  - `IRQ_HANDLED`: irq traitée
- Les bottom-halves sont des fonctions du noyau utilisées pour la gestion de tâches asynchrones
- Ordonnancés à chaque retour d'appel système, d'exception ou de gestionnaire d'interruption
- Ils peuvent être préemptés par des interruptions
- Utilisation de *tasklets* ou *workqueues*

Tasklet et workqueue

- Tasklet
  - Fonctionne uniquement dans un contexte *d'interruption* → ne peut pas s' « endormir »
  - Rapide
- Workqueue
  - S'exécute dans un contexte de *processus*
  - Plus lente
- Le plus souvent on utilise la *tasklet* pour le bottom-half
- Dans les 2 cas on peut utiliser une déclaration statique ou dynamique (voir les exemples)

Exemple avec tasklet

- Exemple:

```
void ma_routine_bh(unsigned long) { /* ...
  code du bottom-half ... */ }

DECLARE_TASKLET(ma_tasklet, ma_routine_bh,
  0);


/* Top-half */

void mon_handler_irq(int irq, void *dev_id)

{

    tasklet_schedule(&ma_tasklet);

}
```

Prise en compte d'une IRQ par l'espace utilisateur

- Dans la méthode `read()` du pilote, on bloque le processus en attente d'interruption → attente dans l'appel système `read()`

```
static DECLARE_WAIT_QUEUE_HEAD(read_wait_queue);
...
ret = wait_event_interruptible(read_wait_queue, condition);
```

- La réception d'une IRQ débloque le processus dans le bottom-half

```
wake_up_interruptible (&read_wait_queue);
```

- Voir l'exemple du clavier PC: la commande `cat` est débloquée par l'action sur une touche

```
# insmod testintr.ko
# cat /dev/testintr
```