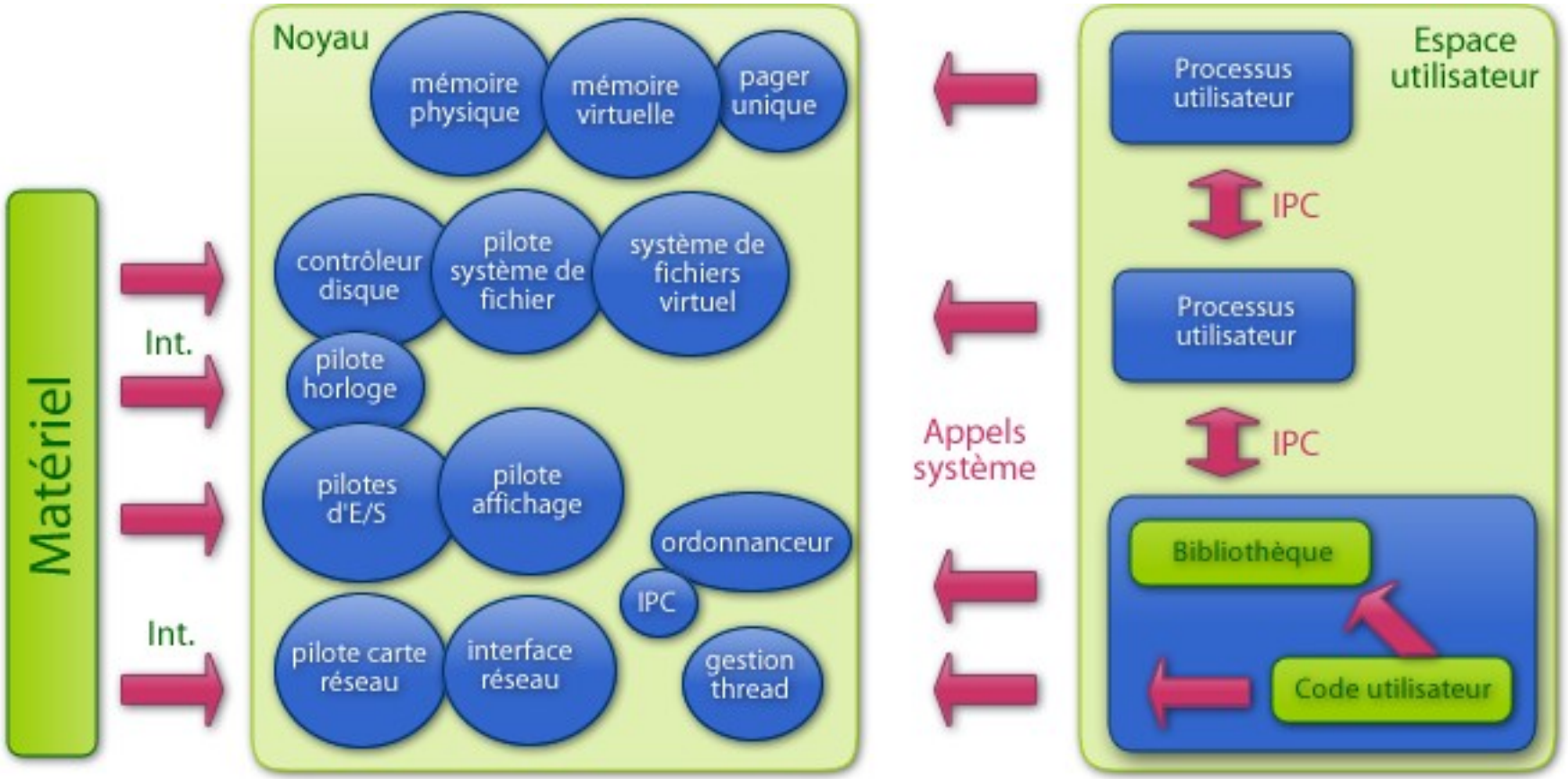
Les modules Linux

Pierre Ficheux (pierre.ficheux@openwide.fr)

Septembre 2013

Rappel de l'architecture de Linux



Here we are !

open wide INGENIERIE

Développement en espace noyau

- Utilisation de l'API des modules Linux (spécifique, non POSIX)
- Programmation plus complexe qu'en espace utilisateur
- Risque de « crash » en cas de bogue, mise au point délicate (accès en mode protégé)
- Programmation (uniquement) en C
- Différence entre *module* et *pilote* (ou driver)
  - Un pilote est toujours un module (source) mais peut être *statique* ou *dynamique*
  - Un module n'est pas forcément un pilote (pas d'interface avec l'espace utilisateur)

API des modules Linux

- Les modules existent depuis la version 2.0
- Depuis la version 2.6, le suffixe est `.ko` (Kernel Object)
- Un module est un objet chargé dynamiquement dans la mémoire du noyau (analogie avec une bibliothèque partagée .so)
- Pour manipuler les modules 2.6 on utilise les *module-init-tools*
  - `insmod/modprobe`: charger en mémoire + exécuter
  - `rmmod`: retirer de la mémoire (« déchargement »)
  - `lsmod`: lister

API des modules Linux, suite

  - `depmod`: calculer les dépendances et créer `modules.dep`
  - `modinfo`: afficher les informations sur le module
- Exemples d'utilisation :

```
# insmod helloworld.ko
$ lsmod | grep helloworld
# rmmod helloworld
# depmod -a
$ modinfo helloworld.ko
```

open wide INGENIERIE

Un module *Hello World*

- Le module effectue un simple `printk()` au chargement et déchargement
- Les en-têtes à utiliser sont :

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

- Au minimum
  - Points d'entrée `module_init()` et `module_exit()`
  - `module_init()` appelé lors du chargement (`insmod`)
  - `module_exit()` appelé lors du déchargement (`rmmod`)

La fonction `module_init()`

- On définit la fonction `<nom_module>_init` puis on appelle la macro `module_init()`

```
static int __init hello_init(void)
{
  printk(KERN_INFO "Hello World\n");
  return 0;
}

module_init(hello_init);
```

- Noter l'absence de virgule après `KERN_INFO`

La fonction `module_exit()`

- On définit la fonction `<nom_module>_exit` puis on appelle la macro `module_exit()`

```
static void __init hello_exit(void)
{
  printk(KERN_INFO "Goodbye World\n");
}

module_exit(hello_exit);
```

Le fichier `Makefile`

- Pour la version 2.4, fichier `Makefile` était créé « à la main »
- Nette amélioration en 2.6 car on utilise le système de compilation du noyau
- Buts identiques à la compilation du noyau en modifiant le répertoire de travail

```
all: make -C $(KDIR) SUBDIRS=$(PWD) modules
install: make -C $(KDIR) SUBDIRS=$(PWD) modules_install
clean: make -C $(KDIR) SUBDIRS=$(PWD) clean
```

open wide INGENIERIE

Le fichier `Makefile`, suite

- Définition du répertoire du noyau à utiliser:
  - Par défaut (compilation native)

    ```
    KDIR= /lib/modules/$(shell uname -r)/build
    ```

  - Sinon, modifier `KDIR` (compilation croisée)
- Fichier objet à utiliser :

  ```
  obj-m := helloworld.o
  ```

- Si plusieurs fichiers objets :

  ```
  MODULE_NAME = my_mod
  $(MODULE_NAME)-objs = obj1.o obj2.o
  obj-m := $(MODULE_NAME).o
  ```

Compilation et test du module

- On utilise la commande `make` :
  - `$ make` → compilation par défaut
  - `$ make V=1` → mode « verbeux »
- Test de chargement du module (en *root*)

```
# insmod helloworld.ko
```

- Le message doit apparaître dans les traces du noyau

```
$ dmesg
# tail /var/log/messages
```

- Dé-chargement

```
# rmmod helloworld
```

Installation du module

- La commande `insmod` sert à tester un fichier .ko unique (pas de gestion de dépendances)
- La procédure normale est la suivante:
  - Installer le module

    ```
    # make install
    ```

    qui effectue un `make modules_install`
  - Calculer les dépendances
  - `# depmod -a`
- Les fichiers sont installés dans

  `/lib/modules/<version-noyau>/extra`
- Le fichier `modules.dep` est mis à jour par la commande `depmod -a`

Installation du module, suite

- Le répertoire `extra` est destiné aux fichiers externes à l'arborescence des sources du noyau
- Les modules des sources sont installés dans `kernel`
- Pour charger, on utilise la commande `modprobe`:

```
# modprobe -v helloworld
```

- La commande `modprobe` charge les modules liés *avant* de charger le module (voir `modules.dep`)

Identification du module

- Macros permettant d'identifier la licence, l'auteur, etc.

```
MODULE_DESCRIPTION("Hello World module");
MODULE_AUTHOR("Pierre Ficheux, Open Wide");
MODULE_LICENSE("GPL");
```

- Les informations sont exploitables par la commande `modinfo` :

```
$ modinfo helloworld.ko
filename:       helloworld.ko
description:    Hello World module
author:         Pierre Ficheux, Open Wide
license:        GPL
```

Vérification de la licence

- Depuis la version 2.6, la licence est vérifiée au chargement du module
- L'absence de licence ou une licence autre que GPL provoque l'affichage d'un des messages suivants :
  - `Module licence 'unspecified' taints kernel.`
  - `Module licence 'MY_LICENCE' taints kernel`
- Le module est tout de même chargé, *sauf* si l'auteur de la fonction exportée impose une licence GPL (exemple : USB)

open wide INGENIERIE

Dépendance des modules

- Une fonction est définie dans un module et utilisée dans d'autres (API noyau)
- On déclare la fonction dans le module puis on utilise la macro `EXPORT_SYMBOL_GPL`

```
void my_printk (char *s) {
  printk (KERN_INFO "my_printk: %s\n", s);
}
EXPORT_SYMBOL_GPL(my_printk);
```

- Seuls les modules sous GPL peuvent l'utiliser → sinon on utilise `EXPORT_SYMBOL`

Dépendance des modules, suite

- On charge le module contenant la fonction puis les modules l'utilisant

```
# insmod ../my_printk/my_printk.ko
# insmod helloworld2.ko
```

- Après l'installation le fichier `modules.dep` contient:

```
extra/helloworld2.ko: extra/my_printk.ko
extra/my_printk.ko:
```

- On peut alors utiliser `modprobe` qui appelle `insmod`

```
# modprobe -v helloworld2
insmod /lib/modules/<version>/extra/my_printk.ko
insmod /lib/modules/<version>>/extra/helloworld2.ko
```

Passage de paramètres

- Le but est de passer des paramètres au chargement du module
- On utilise les macros du fichier `linux/moduleparam.h`
- Types: entier, tableau d'entiers, tableau de caractères, ...

```
module_param (entier, int, 0644);
module_param_array (tableau, int, NULL, 0644);
module_param_string (chaine, chaine, sizeof(chaine), 0644);
```

- Déclaration des variables:

```
static int entier;
static char chaine[MAX_STRING];
static int tableau[MAX_TAB];
```

open wide INGENIERIE

Passage de paramètres, suite

- Description des paramètres, accessible par la commande `modinfo` :

```
MODULE_PARM_DESC(entier, "Un entier");
MODULE_PARM_DESC(chaine, "Une chaîne de caractères");
MODULE_PARM_DESC(tableau, "Un tableau d'entiers");
```

- Utilisation avec `insmod`

```
# insmod hello.ko entier=10 chaine="test" tableau=1,2,3
```

- Automatisation lors du chargement par un fichier de configuration (dépend de la distribution)

```
options hello entier=10 chaine="test" tableau=1,2,3
```