Systèmes d'exploitation pour l'embarqué

UV 5.2 - Exécution et Concurrence

Paul Blottière

ENSTA Bretagne

8 Décembre 2015

https://github.com/pblottiere

Amélioration continue

Contributions

- Dépôt du cours : https://github.com/pblottiere/embsys
- Souhaits d'amélioration, erreurs, idées de TP, ... : ouverture d'Issues (avec le bon label!)
- Apports de corrections : Pull Request

Kernel Linux et modules - version 2.6.x

Plan

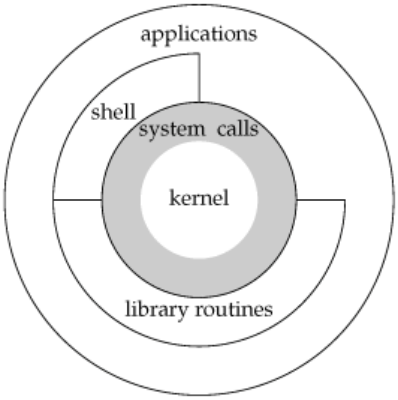
1. Noyau : généralités
2. user-space et kernel-space
3. Architectures noyaux
4. Phases de boot
5. Périphérique bloc ou caractère
6. Administrer les modules sous Linux
7. Développement de module kernel

Noyau : généralités (1)

Les fonctions d'un kernel

Le noyau d'un système d'exploitation :

- gère la partie matérielle (mémoire, processeurs, périphériques, ...)
- offre la gestion des ressources matérielles à travers des appels système
- gère les tâches à exéctuer (ordonnancement)



[Couches d'un OS]

Noyau : généralités (2)

Les enjeux

La programmation kernel nécessite une attention particulière sur :

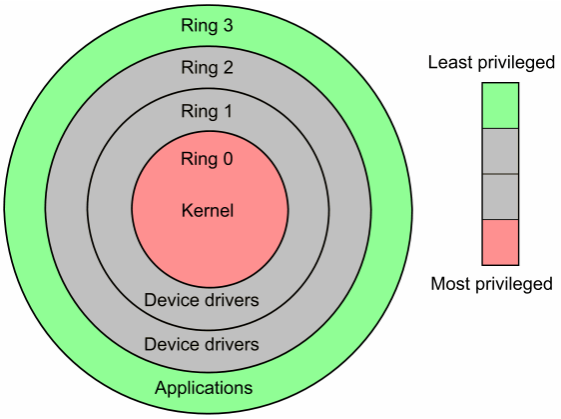
- les problèmes de performance
- les failles de sécurité
- le système de fichiers (corruption)
- la gestion mémoire

user-space et kernel-space (1)

Ring

L'architecture même d'un processeur impose des niveaux de privilèges pour accéder et sécuriser l'accès aux instructions, à l'espace d'adressage ou aux entrées-sorties.

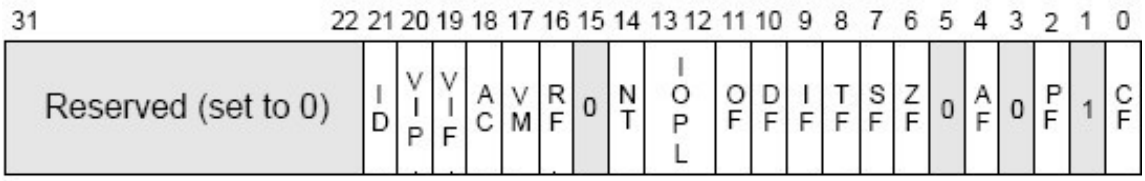
Ces niveaux sont appelés **Anneaux de protection** ou **Rings**.



user-space et kernel-space (2)

Ring

Dans un processeur x86, la notion d'anneau est représentée via le champ **Input/Output Privilege Level** du registre EFLAG :



=> le champ IOPL ne peut être modifié qu'avec le niveau de privilège le plus élevé!

user-space et kernel-space (3)

Besoin d'espace

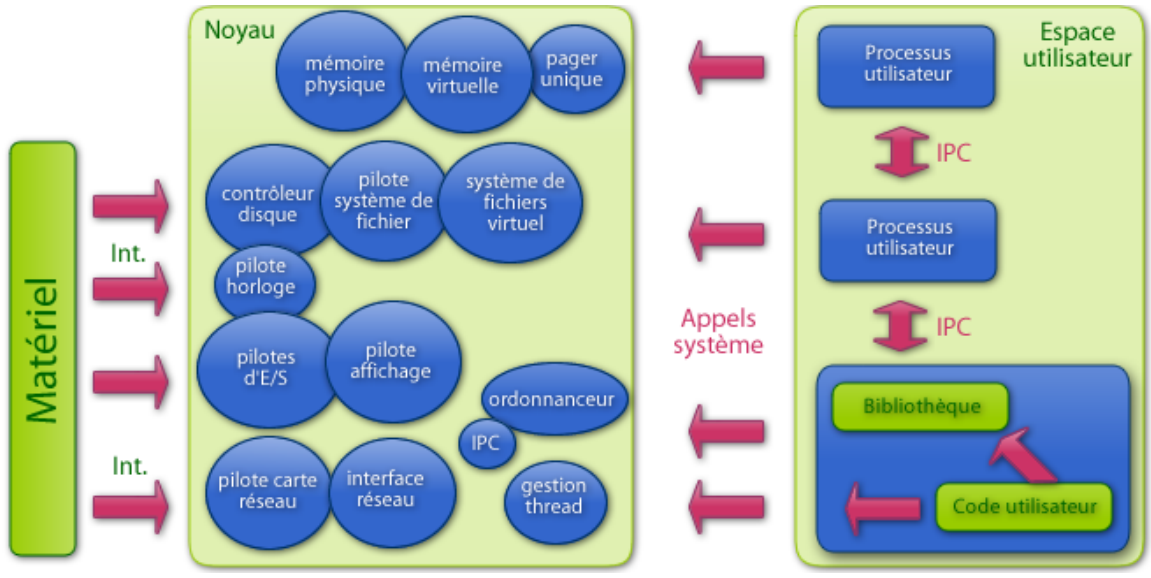
Un OS moderne possède deux espaces mémoire distincts :

- Espace utilisateur : programmes qui discutent avec le matériel à travers le kernel via les appels système. Chaque processus exécuté dans l'espace utilisateur possède sa propre plage de mémoire virtuelle.
- Espace kernel : réservé pour le kernel et ses drivers. Possède le niveau de privilège le plus élevé.

Architectures noyaux (2)

Noyaux monolithiques non modulaires

Dans un noyau monolithique pur (comprendre non modulaire), tous les pilotes de périphériques sont compilés avec le noyau pour former un seul binaire.



[Kernel monolithique]

Architectures noyaux (3)

Noyaux monolithiques non modulaires

Les avantages :

- architecture simple
- vitesse d'exécution

Les inconvénients :

- empreinte mémoire dépendante du nombre de drivers
- difficile à maintenir
- une erreur dans un driver met en danger tout le système (kernel space)

Architectures noyaux (4)

Noyaux monolithiques modulaires

Dans un noyau monolithique modulaire, on choisit au moment de la compilation du kernel si un driver doit être inclus dans le "binaire kernel" ou bien compilé en tant que **module**.

Un module peut être chargé à la volée pour rajouter une fonctionnalité (comme gérer un nouveau type de matériel).

Mais, une erreur dans un module, même chargé à chaud, met toujours la stabilité du système en péril.

Architectures noyaux (5)

Noyaux monolithiques modulaires

Depuis la version 1.2 de Linux, l'architecture est modulaire.

Les modules kernel sont indiqués via un **m** dans la configuration du noyau :

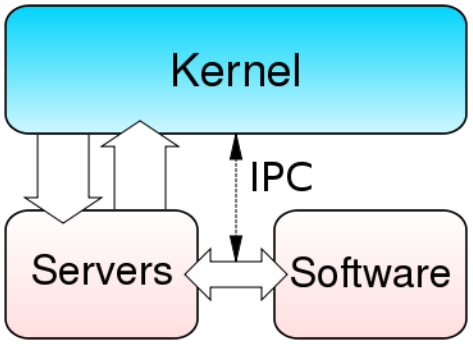
```
> uname -r
4.1.0-2-686-pae
> tail /boot/config-4.1.0-2-686-pae
CONFIG_AVERAGE=y
CONFIG_CORDIC=m
# CONFIG_DDR is not set
CONFIG_OID_REGISTRY=m
CONFIG_UCS2_STRING=y
CONFIG_FONT_SUPPORT=y
# CONFIG_FONTS is not set
```


Architectures noyaux (6)

Micro-noyaux

Un micro-noyau est beaucoup plus léger/petit qu'un noyau monolithique : 6 millions de LOC pour Linux contre moins de 50 000 LOC pour un micro-noyau.

La plupart des services d'un noyau monolithique sont ici exécutés dans l'espace utilisateur via des serveurs de fonctionnalités, réduisant ainsi le volume du kernel.



[Micro kernel]

Architectures noyaux (7)

Micro-noyaux

Les avantages :

- kernel facile à maintenir de par le faible volume de code
- une erreur dans un driver ne remet pas en cause la stabilité du système (exécuté dans l'espace utilisateur : mode protégé)

Les inconvénients :

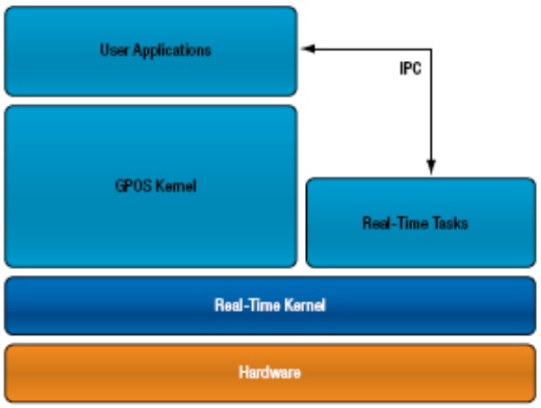
- lent car beaucoup d'appels système pour assurer la communication entre serveurs et kernel.
- communication entre services très lourde (IPC).

Architectures noyaux (8)

Noyaux temps réel

L'architecture classique est un noyau hybride combinant un noyau monolithique généraliste avec un micro-noyau spécialisé temps-réel.

Le noyau généraliste est considéré par le micro-noyau comme un service.



[Noyau temps réel]

Phases de boot (1)

BIOS, UEFI et bootloader

BIOS (Basic Input/Output System) : firmware présent sur la mémoire flash de la carte mère. Il lit le MBR (Master Boot Record : premier secteur de 512 o d'un disque dur) pour déterminer la position du bootloader et le lancer.

=> taille des partitions limitée à 2.2 To

UEFI (Unified Extensible Firmware Interface) : lit la GPT du disque pour déterminer la position du bootloader et le lancer.

=> taille des partitions limitée à 9.4 Zo

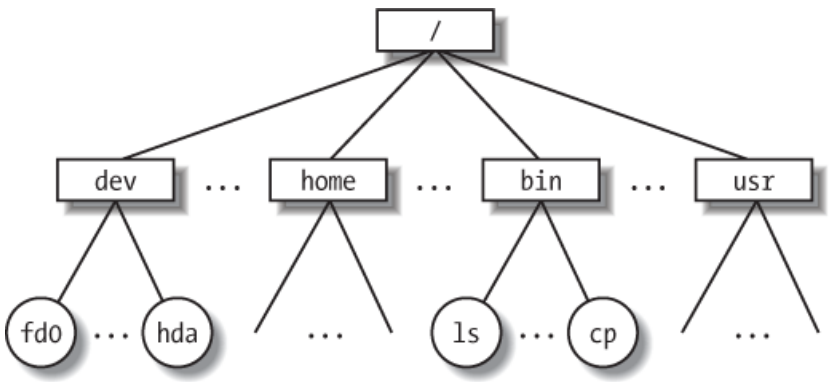
Bootloader (chargeur d'amorçage) : permet à l'utilisateur de choisir le système d'exploitation à lancer (multi-boot). Il est notamment chargé d'exécuter le kernel.

Phases de boot (2)

filesystem

Le bootloader, en plus de lancer le kernel, charge en mémoire une image compressée du système de fichiers (initrd, initramfs, ...).

Le kernel décompresse ensuite l'image et monte le système de fichiers. On a alors le RFS (Root FileSystem).



Phases de boot (3)

init

Par défaut, le kernel lance la phase d'initialisation en utilisant le binaire **/sbin/init** du RFS. On peut indiquer un autre programme d'initialisation grâce à l'option **init=**.

```
> cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.1.0-2-686-pae \
    root=UUID=4dbf9632-e453-43e9-af6f-4d8bbaa4186f \
    ro quiet
```

Le processus **init** est le père de tous les processus et possède un PID=1!

Le binaire d'initialisation consulte le fichier **inittab** pour connaître l'ordre dans lequel il doit démarrer les processus du système.

Phases de boot (4)

run-levels

Le niveau d'exécution, ou runlevel, contrôle le choix des processus démarrés automatiquement par le système :

- 0 : arrêt du système
- 1 ou S : mode maintenance
- 2, 3, 4 ou 5 : mode multi-utilisateurs
- 6 : redémarrage du système

Un RC (Run Command) est un processus qui se lance au démarrage du système à un runlevel particulier et se termine à un autre runlevel.

Phases de boot (5)

run-levels

Run-levels du service réseau :

```
> head /etc/init.d/networking
#!/bin/sh -e
### BEGIN INIT INFO
# Provides:          networking ifupdown
# Required-Start:    mountkernfs local_fs random
# Required-Stop:     local_fs
# Default-Start:     S
# Default-Stop:      0 6
# Short-Description: Raise network interfaces.
# Description:       Prepare /run/network ...
### END INIT INFO
```


Périphérique bloc ou caractère (1)

/dev

Le répertoire **/dev** du RFS offre un moyen de communiquer directement avec le materiel.

Ce répertoire peut être soit peuplé de manière fixe, soit peuplé au moment du boot par scan du materiel présent (par l'utilitaire **udev**).

Dans le cas d'un fichier spécial de périphérique matériel, il existe deux types : bloc ou caractère.

Périphérique bloc ou caractère (2)

mknod

L'utilitaire **mknod** peut être utilisé pour créer des noeuds au sein du système de fichiers :

```
# mknod rights name type major-number minor-number
> mknod -m 622 RFS_PATH/dev/console c 5 1
> mknod -m 666 RFS_PATH/dev/null c 1 3
> mknod -m 444 RFS_PATH/dev/urandom c 1 9
```

Signification des numéros :

- majeur : permet au kernel de déterminer quel pilote utiliser lors d'une écriture/lecture/ouverture sur le noeud
- mineur : réservé au pilote pour différencier des sous catégories d'un même type de matériel.

Périphérique bloc ou caractère (3)

block device

L'accès à une ressource matérielle à travers un fichier spécial de type bloc est bufferisé. Les commandes demandées peuvent même être réorganisées pour optimiser les accès en écriture/lecture.

Les périphériques de stockage sont donc de type bloc :

```
> ls -l /dev/sda[1-3]
brw-rw---- 1 root disk 8, 1 déc.   3 21:50 /dev/sda1
brw-rw---- 1 root disk 8, 2 déc.   3 21:50 /dev/sda2
brw-rw---- 1 root disk 8, 3 déc.   3 21:50 /dev/sda3
```

Périphérique bloc ou caractère (4)

character device

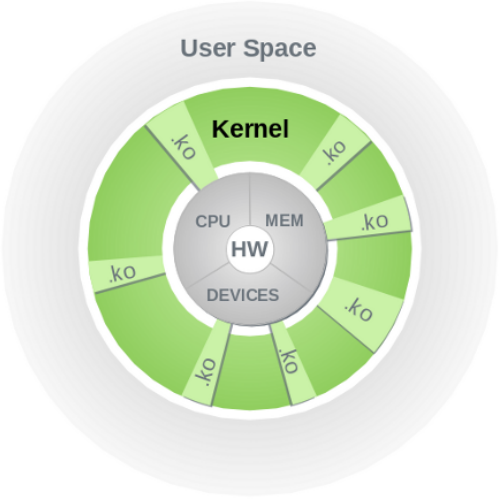
L'accès à une ressource matérielle via un fichier de type caractère est direct et non bufferisé. La plupart des périphériques sont de type caractère.

```
> ls -l /dev/ttyS[0-3]
crw-rw---- 1 root dial 4, 64 déc. 3 21:50 /dev/ttyS0
crw-rw---- 1 root dial 4, 65 déc. 3 21:50 /dev/ttyS1
crw-rw---- 1 root dial 4, 66 déc. 3 21:50 /dev/ttyS2
crw-rw---- 1 root dial 4, 67 déc. 3 21:50 /dev/ttyS3
```


Administrer les modules sous Linux (1)

Où sont-ils?

Sous Linux, un module kernel étend sans reboot les fonctionnalités du noyau et est représenté par des fichiers ayant l'extension **.ko** (comme Kernel Object).



[Modules]

Les modules sont présents dans le répertoire */lib/modules/version*.

Administrer les modules sous Linux (2)

Phases de chargement

Lorsque le kernel a besoin d'une fonctionnalité qu'il ne trouve pas dans son espace, le démon kernel **kmod** lance l'utilitaire **modprobe** pour charger le module :

```
load_module() {
  local module args
  module="$1"
  args="$2"

  if [ "$VERBOSE" != no ]; then
    log_action_msg "Loading kernel module $module"
    modprobe $module $args || true
  else
    modprobe $module $args > /dev/null 2>&1 || true
  fi
}
```


Administrer les modules sous Linux (3)

Phases de chargement

Des modules peuvent avoir des dépendances vers d'autres modules. L'utilitaire **modprobe** va chercher ces dépendances dans le fichier */lib/modules/version/modules.dep*.

Suite à cela, la commande **insmod** est utilisée pour charger les modules prérequis puis le module en question.

Par exemple :

```
> insmod /lib/modules/ver/kernel/fs/fat/fat.ko
> insmod /lib/modules/ver/kernel/fs/msdos/msdos.ko
```

équivaut à :

```
> modprobe msdos
```

Administrer les modules sous Linux (5)

Déchargement

Un module ne peut pas être déchargé si il est actuellement utilisé ou nécessaire pour d'autres modules chargés.

On peut voir le nombre de processus utilisant un module via la commande **lsmod** ou bien en étudiant le troisième champ du fichier virtuel **/proc/modules**.

Le déchargement d'un module se fait via la commande **rmmod**.

Développement de module kernel (1)

La base

Deux fonctions sont nécessaires dans un module kernel :

- une fonction d'initialisation appelée au moment de l'appel de **insmod** et enregistrée via **module_init**.
- une fonction de nettoyage appelée juste avant le déchargement du module et enregistrée via **module_exit**.

Un module peut être documenté via des macros comme **MODULE_AUTHOR** ou **MODULE_DESCRIPTION**.

=> ces informations sont ensuite consultables grâce à la commande **modinfo**.

Développement de module kernel (2)

Les appels système

Un module étant exécuté dans l'espace kernel, il n'a pas accès aux librairies fournies par le système d'exploitation comme la glibc.

Seules les fonctions fournies par le kernel lui même sont accessibles. Elles sont définies dans le fichier virtuel /proc/kallsyms :

```
> head -5 /proc/kallsyms
c1000358 T _stext
c1001000 T hypercall_page
c1001000 t xen_hypercall_set_trap_table
c1001020 t xen_hypercall_mmu_update
c1001040 t xen_hypercall_set_gdt
```


Développement de module kernel (3)

printk : mécanisme de log kernel avec niveaux de priorité

```
// example coming from the LKMPG
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#define DRIVER_AUTHOR "Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC   "A sample driver"

static int __init hello_2_init(void)
{
        printk(KERN_INFO "Hello, world 2\n");
        return 0;
}
static void __exit hello_2_exit(void)
{
        printk(KERN_INFO "Goodbye, world 2\n");
}
module_init(hello_2_init);
module_exit(hello_2_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
```

Développement de module kernel (4)

Paramètres

Un module peut prendre des paramètres au moment de son chargement :

```
> insmod custommodule.ko param1=3
```

Pour cela, les paramètres doivent être enregistrés via la fonction **module_param** et documentés via **MODULE_PARM_DESC** :

```
static type varname = initial_value;
module_param(varname, type, rights);
MODULE_PARAM_DESC(varname, "VARNAME DOCUMENTATION")
```


Développement de module kernel (5)

Configuration et /proc

Un module kernel peut lire et écrire des fichiers dans le répertoire **/proc**. Un utilisateur peut alors configurer le comportement du module via l'édition de ces fichiers.

Pour cela, les fonctions **create_proc_entry**, **copy_to_user** et **copy_from_user** sont utilisées.

Remarque : dans le monde kernel, le référentiel est l'utilisateur. Ainsi, une fonction de lecture définie dans le module va écrire des données alors qu'une fonction d'écriture va en lire.

Développement de module kernel (6)

Character device

Plusieurs opérations sont possibles sur le matériel : écriture, lecture, configuration, flush, ... Ces actions sont représentées par des pointeurs de fonctions dans la structure **file_operations**.

Par défaut, les actions pointent vers NULL. Si le développeur du module veut customiser l'action d'ouverture et de lecture du device :

```
struct file_operations fops = {
    .read = custom_device_read,
    .open = custom_device_open
};
```


Développement de module kernel (7)

Les états

Un module peut gérer l'état des tâches qui communiquent avec lui et même les ajouter dans des WaitQ que le scheduler réveillera au moment opportun!

=> c'est notamment le cas lors d'appels bloquants.

Les fonctions/macros suivantes sont utilisées :

- DECLARE_WAIT_QUEUE_HEAD
- wait_event_interruptible
- wake_up
- ...

Conclusion

La programmation kernel est souvent vue comme de la magie noire... Va savoir pourquoi...

Références

- The Linux Kernel Module Programming Guide - Peter Jay Salzman
- Linux Embarqué - Pierre Ficheux
- Développement système sous Linux - Christophe Blaess