Masters SIAME / IGAI

Architecture des Couches Logicielles Basses

Jacques Jorda - Université Paul Sabatier

Plan

- Introduction
- Architecture du noyau Linux
- Introduction aux pilotes de périphériques
- L'environnement de travail
- Utilisation des modules
- Périphériques de type caractères
- Fonctionnalités avancées
- Périphériques de type bloc

Introduction

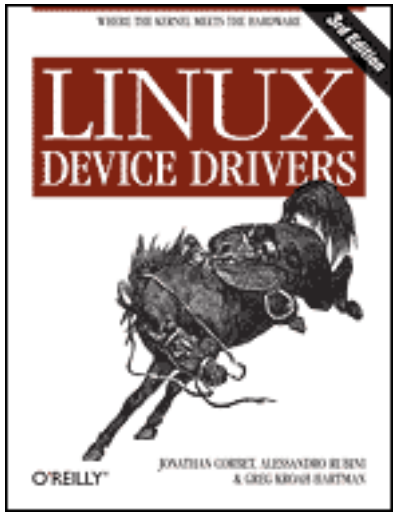
Introduction

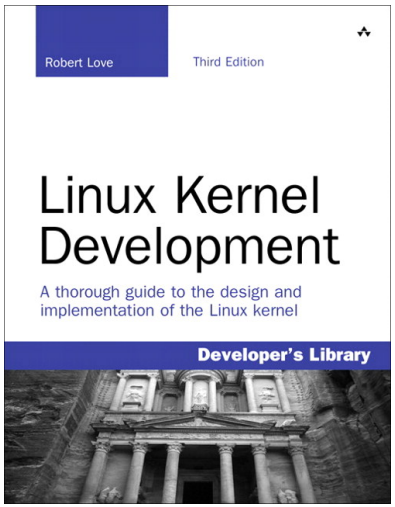
- **Présentation : Jacques Jorda**
  - Bureau 469 – IRIT 2
  - jorda@irit.fr
- **Format du cours**
  - Un peu de théorie
  - Beaucoup de pratique
    - Machines de la salle 14 Bât. 1R1
    - Votre machine personnelle (!!!)
- **Objectif :**
  - Présenter le développement noyau sous linux
  - Concevoir un module, un pilote de périphérique
- **Prérequis :**
  - Programmation C
  - Systèmes

Introduction

A consulter :

- Linux Device Drivers
  - Disponible sur https://lwn.net/Kernel/LDD3/
  - Versions couvertes anciennes mais les principes généraux restent d'actualité
- Linux Kernel Development
  - Pas de version électronique gratuite
  - Version 3, datant de 2010…



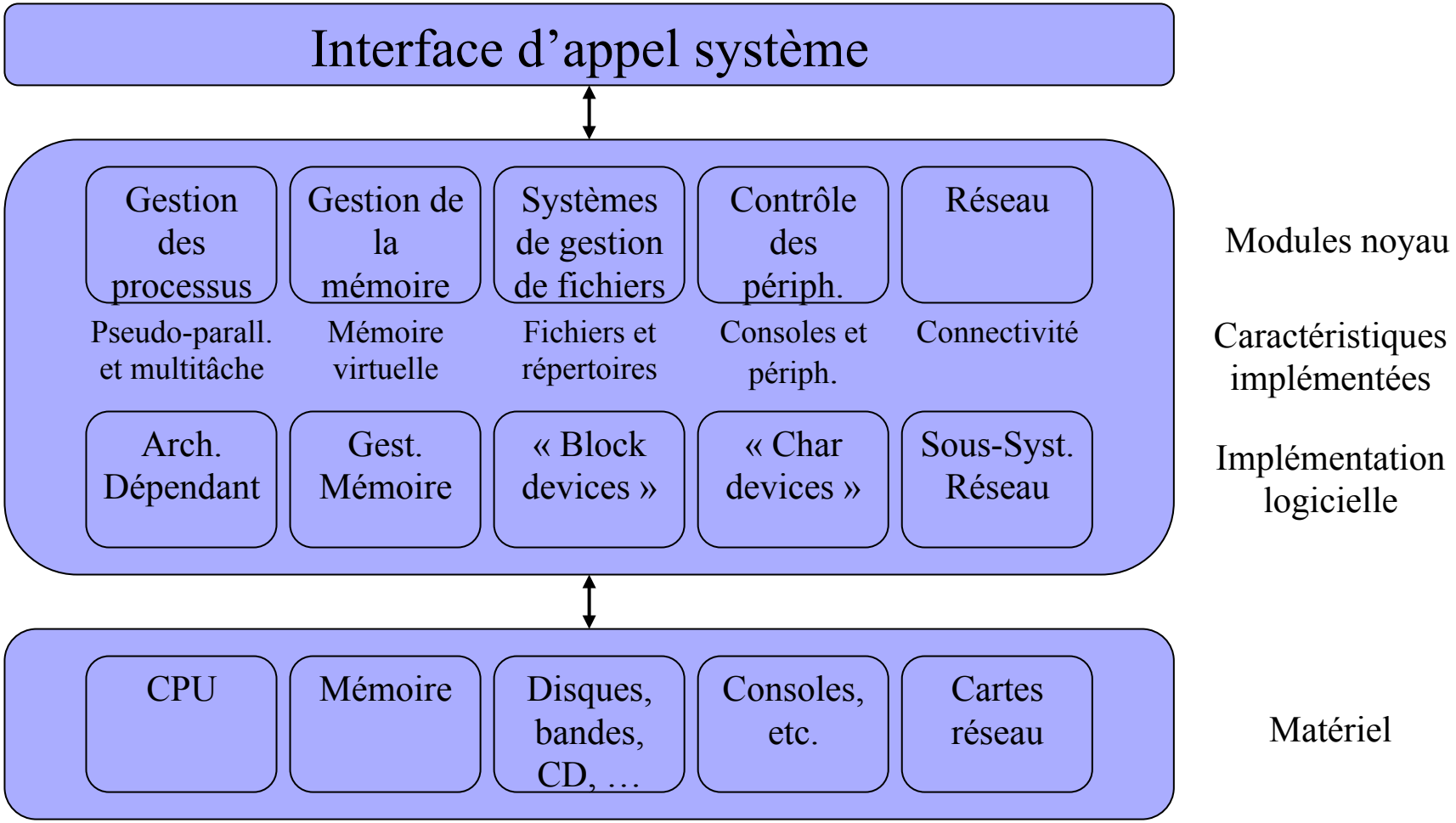


Architecture du noyau Linux

Architecture du noyau Linux

- **Le noyau – notions fondamentales**
  - Le Noyau
  - Mode utilisateur vs Mode Noyau
  - Gestion des processus
  - Gestion de la mémoire
  - Gestion des fichiers
  - Gestion des périphériques
- **Structure du noyau**
- **Le noyau Linux**

Le Noyau


Interface d'appel système
Gestion des processus
Gestion de la mémoire
Systèmes de gestion de fichiers
Contrôle des périph.
Réseau
Pseudo-parall. et multitâche
Mémoire virtuelle
Fichiers et répertoires
Consoles et périph.
Connectivité
Arch. Dépendant
Gest. Mémoire
« Block devices »
« Char devices »
Sous-Syst. Réseau
CPU
Mémoire
Disques, bandes, CD, …
Consoles, etc.
Cartes réseau
Modules noyau
Caractéristiques implémentées
Implémentation logicielle
Matériel

Mode Utilisateur VS Mode Noyau

- **Deux modes de fonctionnement (au moins) pour un processeur :**
  - Mode système : toutes les instructions sont exécutables
  - Mode utilisateur : limitation des ressources accessibles
- **Transition d'un mode à l'autre :**
  - Mode Système -> Mode Utilisateur : direct
  - Mode Utilisateur -> Mode Système :
    - Interruption physique (IRQ)
    - Interruption logicielle (SWI)
    - Exception
- **Intérêt :**
  - Sécurité
  - Stabilité

Gestion des processus

- **Processus :**
  - Création et destruction des processus
  - Dialogue entre eux et avec le monde extérieur
  - Deux types :
    - Processus classiques (process) :
      - Espace de travail privé
      - Latence importante sur création / changement de contexte
    - Processus légers (thread) :
      - Espace de travail partagé avec le père
      - Changement de contexte accélérés
- **Ordonnancement : « Scheduler » (gestionnaire CPU)**
  - Multitâche préemptif
  - Gestion des priorités
  - Inapte temps réel
    - Noyau non préemptif => Temps d'exécution non prédictible
    - Points de préemption dans le noyau depuis 2.6

Gestion de la mémoire

- **Mémoire physique :**
  - Correspond à la mémoire physiquement présente
  - Adressable et gérée par le noyau
  - Non accessible directement par les processus utilisateurs (mode utilisateur)
- **Mémoire virtuelle :**
  - Utilise la mémoire physique et de l'espace disque comme support sous-jacent
  - Correspond à des zones allouées aux processus utilisateurs (mode utilisateur)
  - Privée à chaque processus (mécanisme de protection implémenté par le noyau)

Gestion des fichiers

- **Mécanisme fondamental sous UNIX : tout objet est un fichier (ou presque…)**
- **Système de Gestion de Fichier (SGF) :**
  - Structure abstraite synthétisant l'accès à des ressources
    - Physiques : disques, bandes, CDROM, etc.
    - Virtuelles : support des API de SGF par des composants logiciels (par exemple, /proc)
  - Organisée en arborescence de répertoires et de fichiers
- **Répertoire : conteneur (de SGF, de répertoire, de fichier)**
- **Fichier :**
  - Physique : collection d'informations
  - Virtuel (« spécial ») : implémente un mécanisme d'accès à des ressources par l'API de fichier (par exemple, /dev/tty0)

Gestion des périphériques

- **Apparaissent comme des fichiers :**
  - Fichiers spéciaux : pas d'existence sur les disques sous-jacents
  - Point d'entrée pour l'accès au matériel correspondant
  - Sémantique d'accès type POSIX (open/read/write/close)
- **Quelques cas particuliers (périphériques réseau)**

Structure du noyau

Différents types de Noyaux:

- Noyaux monolithiques
  - Tout dans le noyau
  - Très volumineux => organisation chaotique
  - Ex. : Linux <=1.2, OS/360
- Micro-noyaux
  - Services fondamentaux uniquement. Le reste : micro-serveurs en espace utilisateur
  - Modulaire, mais très lent
  - Ex. : Mach
- Noyaux hybrides
  - Micro-noyaux enrichis ou monolithiques modulaires
  - Ex. : Linux, XNU (MacOS X)

Le noyau Linux

- **Linux est un noyau**
  - Interface avec le matériel
    - Masquage de la complexité du matériel
    - Masquage de la diversité du matériel
  - Il implémente
    - les mécanismes de sécurité
    - La gestion des processus
  - Il ne s'agit pas d'un système d'exploitation
- **Linux est un noyau « Unix-like »**
  - Stratégie tout-fichier
  - Compatibilité des bibliothèques
  - Ne peut pas recevoir la dénomination UNIX (propriétaire)

Le noyau Linux

- **Conçu Par Linus Torvalds en 1991**
  - Basé sur le système d'exploitation Minix
  - Utilisation des outils GNU et Licence GPL
  - Publicité des sources
  - Travail collaboratif
- **Taxonomie : version x.y.z**
  - x.y : numéro de version principal
    - y pair : version stable
    - y impair : version de développement
  - z : identification exacte de la version (2.6.3, 2.6.7, etc.)
- **Première version stable : 1.0 (~1993)**
- **Première version SMP : 2.0 (1996)**
- **Version actuelle : 4.9**

Le noyau Linux

- **Noyau**
  - Monolithique modulaire
  - Multitâche
  - Multi-Utilisateur
  - Multiprocesseur (depuis les versions SMP)
- **Multi plateforme**
  - Minimum requis : 32 bits (sans MMU !)
  - 32 bits : alpha, ARM, i386, m68k, mips, PPC, sparc, etc.
  - 64 bits : ia64, mips64, ppc64, x86_64, etc.
- **Adaptable à la cible (PDA -> Serveur)**

Introduction aux pilotes de périphériques

Introduction aux pilotes de périphériques

- Définitions
- Rôle
- Les modules
- Quelques aspects sur la sécurité

Définitions

- Portion de code autonome
- Exploite les fonctionnalités matérielles d'un périphérique
- Masque la complexité de ce périphérique et son implémentation
- Réalise l'interface entre l'API et le matériel

Rôle d'un driver

- **Apporte des moyens d'action sur un périphérique (les mécanismes)**
- **Ne contraint pas l'utilisateur quand à l'utilisation de ces moyens d'actions (la stratégie)**
- **Exemple**

Pilote de lecteur de disquettes = lecture d'un flot d'octets

vs

Accès aux données d'une disquette (qui, comment)

- **Objectif : développer un pilote non contraignant**

Rôle d'un driver

- **Caractéristiques :**
  - Support synchrone et asynchrone
  - Ré-entrant
    - SMP
    - Monoprocesseur : noyau non préemptif mais réentrant !
  - Exploite l'intégralité du matériel
  - Pas de sur-couche orienté contrainte
- **Le reste est livré séparément :**
  - Application de paramétrage et de configuration
  - Librairie utilisateur pour l'administration et/ou la programmation
- **Ce qui est intégré au noyau doit rester SIMPLE, RAPIDE, PETIT et PEU GOURMAND !**

Les modules

- **Portion de code qui peut être adjointe au noyau (par exemple, les pilotes de périphériques)**
  - Taille du noyau réduite
  - Augmente la modularité
- **Le code objet n'est pas un exécutable :**
  - modprobe : liaison et suppression du noyau
  - Obsolète, mais utiles pour des modules non situés dans /lib :
    - insmod : liaison dynamique au noyau
    - rmmod : suppression dynamique du noyau
- **Trois types de périphériques sous UNIX => trois types de modules :**
  - Périphériques de type « caractères »
  - Périphériques de type « blocs »
  - Périphériques de type « réseau »

Les modules

- **Périphériques caractères :**
  - Accès sous forme d'un flot d'octets
  - Point d'entrée UNIX : /dev
  - Proche des fichiers, bien que l'accès soit souvent séquentiel
  - Implémente typiquement les primitives « open », « close », « read » et « write »
  - Exemple :
    - Consoles textes
    - Ports séries
    - Etc.

Les modules

- **Périphériques blocs :**
  - Peuvent être vus comme un SGF
  - Accès par un nœud du SGF, comme pour les périphériques caractères
  - Ressemblent aux disques
  - Opérations souvent réalisées par blocs (en général de 1 Ko)
  - Implémente les opérations nécessaires au montage du SGF en plus des opérations classiques implémentées par les périphériques caractères

Les modules

- **Périphériques réseau (« network interface ») :**
  - Représente un périphérique matériel ou purement logiciel (ex : « loopback interface »)
  - Accès aux données sous forme de paquets
  - Pas de point d'entrées dans le SGF
  - Utilisation de noms uniques pour les nommer : eth_i

Les modules

- **D'autres modules existent :**
  - Pilotes SCSI
  - Pilotes USB
  - Pilotes FireWire
  - Etc.
- **Principe de fonctionnement :**
  - Le noyau dispose d'une couche abstraite implémentant la gestion de ces périphériques
  - Le pilote réalise l'interface entre cette couche et le matériel présent dans la machine

Les modules

- **D'autres fonctionnalités non matérielles sont implémentées sous forme de modules**
- **Exemple : le filesystem**
  - Aucun matériel associé
  - Fait correspondre des structures de haut niveau à des structures de plus bas niveau pour :
    - Contrôler la validité des noms de fichiers et de répertoires
    - Faire correspondre les fichiers et répertoires aux blocs de données communiqués aux périphériques blocs correspondants

La sécurité

- **Sécurité classique :**
  - Accidentelle
  - Délibérée
- **Pilote de périphérique = module de niveau noyau**
  - Peuvent constituer des trous de sécurité
  - Cas typique : vérification des tailles de buffer pour éviter l'exécution de code malveillant
- **Développement de pilote de périphérique :**
  - Globalement, la sécurité est du ressort de l'administrateur système
    - Cela ne doit pas figurer dans l'implémentation
  - Ponctuellement la sécurité doit être vérifiée au niveau du pilote de périphérique
    - C'est le cas lorsque les opérations du pilote peuvent affecter l'ensemble du système

L’environnement de travail

L’environnement de travail

- Pré-requis et limites
- L’environnement de TP de la salle 14

Pré-requis et limites

- **Développer un pilote de périphérique = développer un module**
- **Tester un module = charger le module dans le noyau**
  - Les commandes modprobe, insmod et rmmod nécessitent des privilèges étendus
    - Nécessité d'être Administrateur de la machine (root)
    - Ouvre des failles de sécurité !
  - Intègre le code développé au noyau lui-même
    - Le code intégré au noyau s'exécute avec les privilèges les plus élevés
    - Plantage fréquent !
- **Solution : travailler sur une machine virtuelle**

L’environnement de TP de la salle 14

- **Salle des machines :**
  - Ordinateurs PC sous Fedora
  - Machines virtuelles sous Fedora
  - Comptes utilisateurs locaux uniquement
  - Attributions fixes !!

Utilisation des modules

Utilisation des modules

- Modules et applications
- Compilation et chargement
- Table des symboles du noyau
- Initialisation et terminaison
- Utilisation des paramètres
- Utilisation des ressources
- Jouer dans l'espace utilisateur
- Problèmes de compatibilité

Utilisation des modules

Premier exemple

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

int init_module(void) {printk(KERN_ALERT "Hello, World\n") ; return 0;}
Void cleanup_module(void){printk(KERN_ALERT "Goodbye cruel world\n");}
```

- printk :
  - équivalent de printf
  - Primitive utilisable au niveau noyau
- KERN_ALERT : niveau de priorité du message

Exécution :

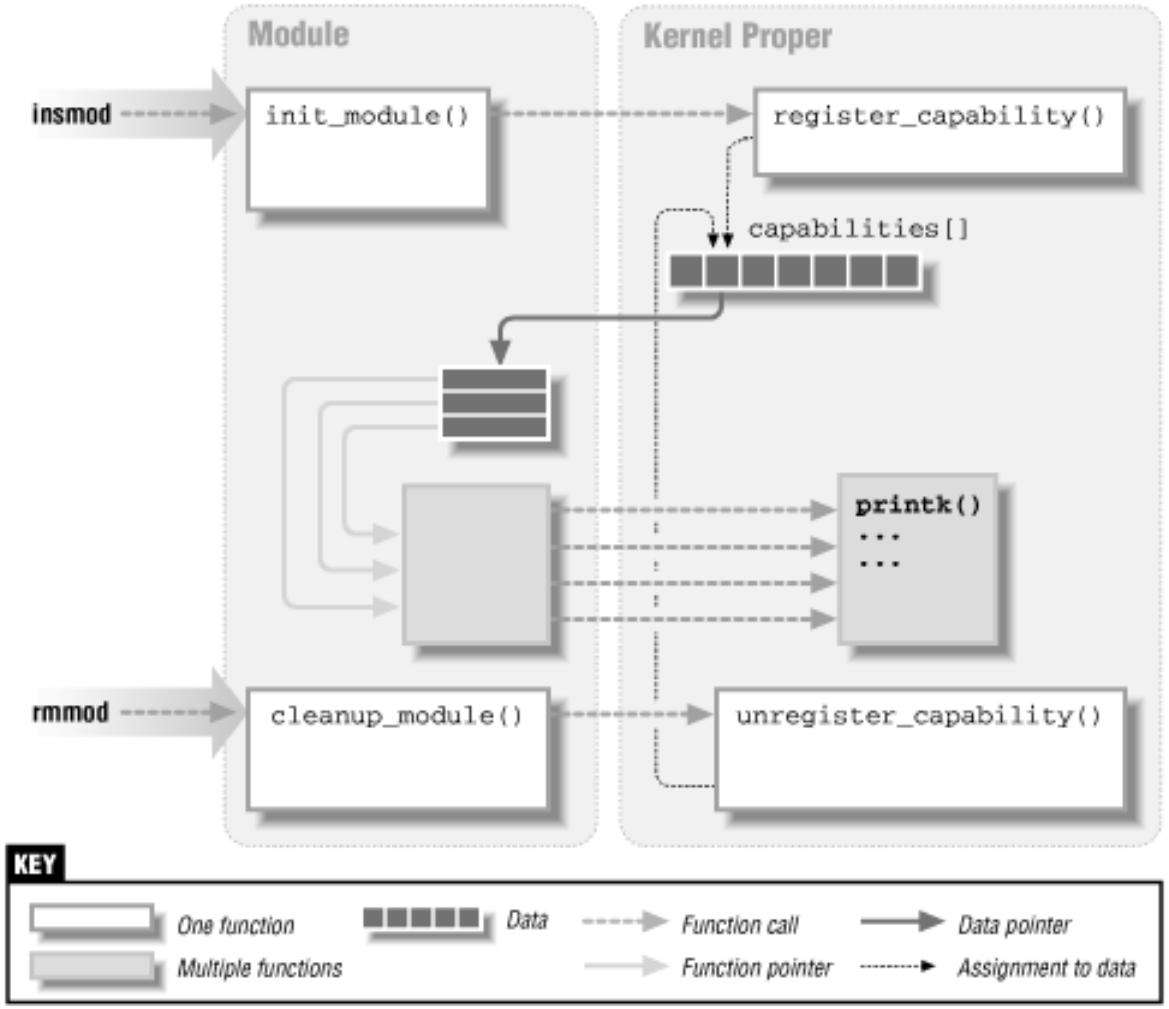
```
root# make
root# insmod hello.ko
Hello, World
root# rmmod hello
Goodbye cruel world
root#
```

- Il faut être administrateur
- Prise en charge des versions désactivée

Modules et applications

- **Application : réalise un travail du début à la fin**
- **Module :**
  - init_module : point d'entrée permettant l'enregistrement du module
  - cleanup_module : point d'entrée de libération des ressources et de désinscription
  - Pas de librairies : liaison avec le noyau uniquement
  - Headers relatifs au noyau : include/linux et include/asm dans /usr/src/linux
  - Visibilité des symboles et déclarations static

Modules et applications



Mode Utilisateur et Mode Noyau

- **Nécessaire pour préserver la cohérence et la sécurité du système**
- **S'appuie sur les modes d'exécutions du processeur**
  - Au moins deux modes :
    - Mode superviseur : toutes les instructions sont autorisées
    - Mode utilisateur : accès aux périphériques et à la mémoire limité et contrôlé
  - Si plus de deux modes existent : utilisation fréquente des deux extrêmes (par exemple, plateforme intel x86)

Espace utilisateur et espace noyau

- **Regroupe**
  - Le mode d'exécution
  - La notion d'espace mémoire séparé
- **Changement d'espace :**
  - Appel système – le contexte est celui du processus appelant
  - Interruption – contexte spécifique

(Pseudo-)Parallélisme

- **Risque de parallélisme dans le noyau**
  - Plusieurs processus utilisent le pilote
  - Interruption du périphérique pendant l'utilisation du pilote
  - Utilisation d'horloges noyau (timers)
  - Utilisation sur un système SMP
- **Même lorsque le noyau est non préemptif (2.4 et antérieurs), considérer qu'il l'est**
  - Permet l'exécution sur système SMP
  - Permet l'exécution sur noyau 2.6 et ultérieurs

Processus courant

- **Pointeur *current* défini dans *<asm/current.h>* inclus par *<linux/sched.h>***
- **Exemple d'utilisation :**
  - Nom de la commande du processus courant
  - PID du processus courant

```
printk("The process is \"%s\" (pid %i)\n",
          current->comm,
          current->pid);
```

- **Attention, renvoi un numéro différent pour tous les threads**
  - Lié à l'implémentation des threads au niveau kernel dans linux
  - Utiliser current->tgid pour obtenir le PID d'appartenance

Considérations générales

- **La pile noyau est limitée**
  - Pas de déclaration de variables automatiques de grande taille
  - Utilisation d'allocations mémoire dynamiques
- **Les fonctions bas niveau (celles commençant par __) doivent être manipulées avec précaution**
- **Pas de calcul en virgule flottante dans le noyau**
  - Généralement inutile
  - Nécessiterait de sauvegarder l'état du coprocesseur flottant aux frontières du noyau

Compilation et chargement

- **Depuis la version 2.6 :**
  - L'arborescence du noyau cible doit être présente sur le système
  - Le noyau correspondant doit être compilé
    - Le processus de compilation utilise les fichiers objets
    - Il est préférable d'exécuter le noyau ainsi construit
    - Attention aux noyaux des distributions qui sont souvent largement patchés
  - Les outils (compilateur, insmod, modprobe, etc.) doivent avoir les versions indiquées dans Documentation/Changes

Compilation et chargement

Création d'un fichier makefile

```
ifneq ($(KERNELRELEASE),)
   obj-m := Hello_World_4.o
   Hello_World_4-objs := Hello_World_4_Start.o Hello_World_4_Stop.o
else
   KERNEL_DIR ?= /lib/modules/$(shell uname -r)/build
   PWD := $(shell pwd)
default:
   $(MAKE) -C ${KERNEL_DIR} M=$(PWD) modules
endif
```

Compilation et chargement

- **insmod :**
  - Allocation de mémoire noyau pour le module
  - Copie du code objet depuis le fichier .ko dans cette zone
  - Résolution des références externes en utilisant la table des symboles du noyau
  - Appel de la fonction d'initialisation du module
- **rmmod :**
  - Supprime le module du noyau et libère les ressources
  - Nécessite que le module ne soit pas utilisé
- **modprobe :**
  - Identique à insmod, mais charge aussi les modules dépendant
  - Ne regarde que dans les répertoires standards
- **lsmod :**
  - Liste les modules chargés
  - Cf. /proc/modules

Gestion des versions

- **Compiler les modules pour chaque version de noyau**
  - Les modules sont liés avec (entre autre) vermagic.o :
    - Présent dans l'arborescence du noyau cible
    - Contient des informations sur les versions utilisées (noyau, compilateur, etc.)
  - Tentative de chargement sur une version incorrecte :
    - « Error inserting '…': -1 Invalid module format »
    - Il faut
      - créer une arborescence spécifique pour la cible
      - modifier KERNELDIR dans le Makefile
- **Création d'un module fonctionnant avec plusieurs versions :**
  - Utilisation de #ifdef
  - Utilisation des macros de versions

Gestion des versions

- **Macro définies dans <linux/version.h> (inclus par <linux/module.h>) :**
  - UTS_RELEASE : remplacé par la version du noyau (string) – ex : « 2.3.48 »
  - LINUX_VERSION_CODE : représentation binaire de la version du noyau – ex : 2.3.48 -> 131888 (i.e. 0x020330)
  - KERNEL_VERSION(major,minor,release) : retrouve la représentation binaire pour une version donnée du noyau

Gestion des plateformes

- **Linux est multi-plateforme**
  - rules.make inclus makefile.xxx pour obtenir des informations spécifiques
  - Permet de réaliser facilement des compilations croisées par le biais de $(CROSS_COMPILE)
  - L'architecture SPARC est à traiter avec soin (SPARC V8 identique à SPARC V9)

Table des symboles du noyau

- **Contient l'adresse des symboles publics (noyau et modules chargés)**
- **Consultable dans /proc/ksyms**
- **Permet de réaliser de l'empilage de modules**
  - Exemple : le driver de port parallèle se compose :
    - D'un module générique qui exporte des symboles standards
    - D'une implémentation spécifique au hardware de la machine

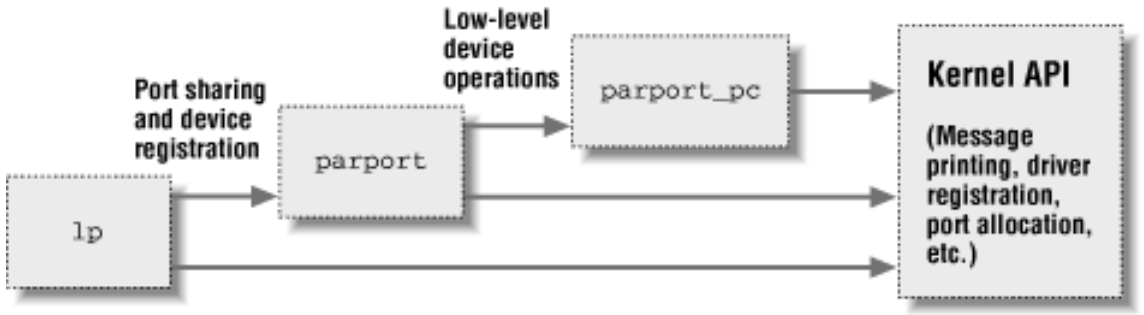


Table des symboles du noyau

- **Plusieurs stratégies :**
  - Kernel 2.6 et ultérieurs :
    - Tous les symboles sont privés par défaut
    - Exportation : macros à placer hors de toute fonction
      - EXPORT_SYMBOL(nom) : pour tous les modules
      - EXPORT_SYMBOL_GPL(nom) : uniquement pour les modules GPL
  - Kernel ultérieurs à 2.0 et antérieurs à 2.6 :
    - Aucun export :
      - Macro **EXPORT_NO_SYMBOL**
      - N'importe où (généralement, init_module)
    - Export de symboles :
      - Définir **EXPORT_SYMTAB** (avant include de module.h)
      - Utilisation de **EXPORT_SYMBOL** et **EXPORT_SYMBOL_NOVERS**
        - ✓ NOVERS pour ne pas inclure d'information de version
        - ✓ A l'extérieur de toute fonction (lié à l'expansion – Cf. module.h)
  - Kernel 2.0 : méthode spécifique (Cf. littérature)

Informations sur le module

- **Fortement conseillé :**
  - MODULE_LICENSE(*type_licence*)
    - *Type_licence* : « GPL », « GPL v2 », « GPL and additional rights », « Dual BSD/ GPL », « Dual MPL/GPL » et « Proprietary »
    - Le noyau est marqué *vicié* (tainted) si la licence d'un module chargé n'est pas libre
- **Autres informations :**
  - MODULE_AUTHOR
  - MODULE_DESCRIPTION
  - MODULE_VERSION
  - MODULE_ALIAS
  - MODULE_DEVICE_TABLE
- **Par convention, placées à la fin du fichier**

Informations sur le module

Exemple :

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

#define LICENCE "GPL"
#define AUTEUR "Barnabe Jorda barnaj@yahoo.fr"
#define DESCRIPTION "Exemple de module Master CAMSI"
#define DEVICE "Os à moelle"

int init_module(void){
  printk(KERN_ALERT "Hello world 2!\n");
  return 0;
}

Void cleanup_module(void){
  printk(KERN_ALERT "Goodbye Cruel world 2!\n");
}

/* Types de licences supportées :                                          */
/*      "GPL"                       GNU Public Licence V2 ou ultérieure    */
/*      "GPL v2"                    GNU Public Licence v2                  */
/*      "GPL and additional rights  GNU Public Licence v2 et droits complémentaires */
/*      "Dual BSD/GPL"              Licence GPL ou BSD au choix            */
/*      "Dual MPL/GPL"              Licence GPL ou Mozilla au choix        */
/*      "Propietary"                Produit à diffusion non libre (commercial) */
MODULE_LICENSE(LICENCE);
/* Classiquement : Nom Email                                               */
MODULE_AUTHOR(AUTEUR);
/* Ce que fait votre module                                                */
MODULE_DESCRIPTION(DESCRIPTION);
/* Périphériques supportés                                                   */
MODULE_SUPPORTED_DEVICE(DEVICE);
```

Initialisation et terminaison

- **init_module sert à l'enregistrement des fonctionnalités du module**
  - Une fonction spécialisée pour chaque type de fonctionnalité
  - Ces fonctions s'appuient sur une structure qui comprend généralement un pointeur vers les fonctions du module
  - Utilisation de __init et __initdata pour libérer la mémoire
- **Traitement des erreurs lors de l'initialisation**
  - Annuler l'enregistrement des fonctionnalités

```
int __init init_module(void)
{
   int err;
   /* Notre enregistrement necessite un pointeur et un nom */
   err = register_fn_1(ptr1, "skull");
   if (err) goto fail_ fn_1;
   err = register_ fn_2(ptr2, "skull");
   if (err) goto fail_ fn_2;
   …
   return 0; /* succes */

   fail_ fn_ 2: unregister_ fn_1(ptr1, "skull");
   fail_ fn_ 1: return err; /* propagation de l'erreur */
```

Initialisation et terminaison

- **cleanup_module : désinscrit les services enregistrés dans init_module**

```
void __exit cleanup_module(void)
{
   unregister_fn_2(ptr2, "skull");
   unregister_fn_1(ptr1, "skull");
   return;
}
```

  - __exit marque la fonction comme étant appelée uniquement lors du déchargement. La fonction est omise si
    - Le module est statiquement lié au noyau
    - Le noyau est paramétré pour interdire les déchargements de modules
- **Alternative : désinscription sélective et appel de cleanup_module dans init_module**
  - Plus lent
  - Plus élégant (pas de goto)
  - Attention à __init et __exit

Fonctions de chargement / déchargement explicites

- **init_module et cleanup_module peuvent être remplacées par des fonctions spécifiques**
  - Inclure <linux/init.h>
  - Utiliser
    - module_init(my_init_function)
    - module_exit(my_exit_function)
  - On peut marquer les fonctions statiques
    - Elles ne sont pas exportées
    - Ne pas marquer init_module et cleanup_module statiques, cela diffère de la définition dans module.h !

Fonctions de chargement / déchargement explicites

- **Exemple :**

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

static int hello_init(void)
{
  printk(KERN_ALERT "Hello world 1!\n");
  return 0;
}

static void hello_cleanup(void)
{
  printk(KERN_ALERT "Goodbye Cruel world 1!\n");
}
module_init(hello_init);
module_exit(hello_cleanup);
```

Assignation des paramètres

- **Fichier de configuration**
  - Typiquement : /etc/modules.conf
- **Directement par insmod**
  - Nécessite la déclaration du paramètre
  - Macro module_param de moduleparam.h :
    - module_param(nom, type, perm)
      - Nom : nom de la variable et du paramètre
      - Type : bool, charp, int, long, short et version non signées
      - Perm : autorisation (Cf. linux/stats.h) d'accès dans sysfs
  - Exemple :

```
static char *valeur_toto="Toto est content";
module_param(valeur_toto,charp,S_IRUGO)
…
root# insmod matos valeur_toto="titi"
```

- **Tester les valeurs des variables avec les valeurs par défaut afin de vérifier si l'utilisateur a modifié les paramètres**
  - Si oui : charger le driver avec les paramètres utilisateurs
  - Si non : procéder à l'auto-détection

Assignation des paramètres

- **Exemple :**

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

/* Définition d'un paramètre                                                    */
/*      Macro MODULE_PARM()                                                     */
/*           Nom du paramètre                                                   */
/*           Type du paramètre : b (byte), h (short integer), i (integer), l (long integer),     */
/*                               s (string - allocation par insmod)                              */
/*      Macro MODULE_PARM_DESC()                                                */
/*           Nom du paramètre                                                   */
/*           Description (chaîne de caractères)                                 */
char *NomUtilisateur="Jacques";
module_param(NomUtilisateur, charp, S_IRUGO);

static int hello_init(void){
  printk(KERN_ALERT "Hello %s!\n", NomUtilisateur);
  return 0;
}

static void hello_cleanup(void){
  printk(KERN_ALERT "Goodbye %s\n", NomUtilisateur);
}

module_init(hello_init);
module_exit(hello_cleanup);
```

Configuration automatique et manuelle

- **Configuration automatique :**
  - Plus confortable
  - Plus difficile à mettre en œuvre
- **Configuration manuelle :**
  - Plus simple à implémenter
  - Permet de modifier la configuration automatique

Assignation des paramètres

- **Exemple :**

```
/* port pouvant varier de 0x280 a 0x300 par pas de 0x010. */
#define SKULL_PORT_FLOOR 0x280
#define SKULL_PORT_CEIL 0x300
#define SKULL_PORT_RANGE 0x010
/* Autodetection, sauf si valeur "skull_port_base" definie au moment du insmod */
static int skull_port_base=0; /* 0 force l'autodetection */
Module_param (skull_port_base, int, S_IRUGO);

static int skull_find_hw(void) /* retourne le nombre de peripheriques */
{
    /* base contient soit la valeur manuelle, soit la premiere valeur a tester */
    int base = skull_port_base ? skull_port_base : SKULL_PORT_FLOOR;
    int result = 0;
    /* Un passage si valeur assignee, boucle sinon */
    do
    {
          if (skull_detect(base, SKULL_PORT_RANGE) == 0)
          {
                skull_init_board(base);
                result++;
          }
          base += SKULL_PORT_RANGE; /* prepare le passage suivant */
    }
     while (skull_port_base == 0 && base < SKULL_PORT_CEIL);
    return result;
}
```

Jouer dans l'espace utilisateur

- **Avantages :**
  - On a accès à l'ensemble de la librairie C (exemple : Serveur X, gpm, etc.)
  - On peut utiliser n'importe quel debugger
  - Tout plantage n'affecte que la tâche courante
  - Possibilité de swap (driver gros consommateur de ressources)
- **Inconvénients :**
  - Les interruptions ne sont pas visibles
  - Accès direct aux ressources :
    - Nécessite des droits étendus
    - Pénalisant en terme de performance
      - Swap nécessaire pour accéder au matériel
      - Swap disques éventuels

Jouer dans l'espace utilisateur

- **Habituellement :**
  - On s'appuie sur un driver de bas niveau pour l'accès aux ressources
  - Un driver client est développé au dessus du driver bas niveau
    - Facilité de mise au point
    - Accès aux ressources clients
- **Exemple :**
  - Scanner SCSI
  - Graveur de CD

TP N° 1

Configurer l'environnement de travail

- Se connecter sur le PC
- Copier `/nfs/images/FC22_UE2/FC22.*` dans `/media/storage/images/`
- Modifier FC22.xml :
  - Nom / chemin du disque dur virtuel : `<source file='…' />`
  - Adresse MAC : `<mac address='02:00:c0:00:00:xx' />` pour m1infoxx
- Lancer la VM :
  - `virsh create FC22.xml --console`
- Mot de passe root : camsi

Concevoir un module Hello World

- descriptif complet (auteur, description, etc.)
- paramètre (chaine de caractères apparaissant dans le message).
- Compiler et tester le module

Pilote de périphérique de type caractères

Un pilote de périphérique caractères

- Nombres majeurs et mineurs
- Les opérations fichiers
- La structure file
- Opérations open et release
- Gestion de la mémoire
- Concurrence et parallélisme
- Lectures et écritures

Nombres majeurs et mineurs

- **Périphériques**
  - Accessibles sous forme de fichiers spéciaux
  - Situés dans /dev
  - Types de fichiers
    - « c » pour les périphériques caractères
    - « b » pour les périphériques blocs
- **Exemple de « ls –l » dans « /dev »**

```
crw-rw-rw- 1 root    root     1, 3   Feb 23 1999  null
crw------- 1 root    root    10, 1   Feb 23 1999  psaux
crw------- 1 rubini  tty      4, 1   Aug 16 22:22 tty1
crw-rw-rw- 1 root    dialout  4, 64  Jun 30 11:19 ttyS0
crw-rw-rw- 1 root    dialout  4, 65  Aug 16 00:00 ttyS1
crw------- 1 root    sys      7, 1   Feb 23 1999  vcs1
crw------- 1 root    sys      7, 129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root    root     1, 5   Feb 23 1999  zero
```

Nombres majeurs et nombres mineurs

- **Nombres majeurs**
  - Première des deux colonnes
  - Identifie le driver correspondant au périphérique
  - Utilisé pour dispatcher l'exécution à l'ouverture
- **Nombres mineurs**
  - Seconde colonne
  - Ignoré par le système
  - Utilisé par le driver pour déterminer quel périphérique (parmi tous ceux connus) est accédé, ou comment un même périphérique est accédé

Nombres majeurs et nombres mineurs

- **Structure dev_t (linux/types.h) stocke le majeur et le mineur**
  - 32 bits pour la version 2.6.0
    - Majeur : 12 bits
    - Mineur : 20 bits
  - Macros dans <linux/kdev_t.h> :
    - MAJOR(dev_t dev)
    - MINOR(dev_t dev)
    - MKDEV(int major, int minor)

Nombres majeurs et nombres mineurs

Réservation de numéros de périphérique

- Revient à lui assigner un nombre majeur
  - 128 maxi pour noyau 2.0
  - 256 (en fait, 254) à partir de la version 2.2
  - 4096 depuis la version 2.6.0
- Fonction définie dans <linux/fs.h> :

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

  - Succès si retour >=0
  - Nécessite de trouver le nombre requis de périph. consécutifs
  - Peut passer au majeur suivant si *count* plus grand que le nombre de mineurs
  - Nécessite de connaître à l'avance le numéro de périphérique à utiliser
    - Assignation statique !

Nombres majeurs et nombres mineurs

- **Allocation dynamique**
  - Préféré à une allocation statique
  - Fonction définie dans <linux/fs.h> :

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

- **Libération des numéros**
  - Nécessaire lors du déchargement
  - Identique pour les réservations statiques ou dynamiques
  - Usuellement dans le module_cleanup
  - Fonction définie dans <linux/fs.h> :

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

  - OOPS
    - Pas de libération : /proc/devices ne peut plus être accédé
      - Une ligne pointe vers un module déchargé
      - Génération d'une faute lors de l'accès
    - Il vaut mieux redémarrer

Remarques sur l'allocation dynamique

- **Assignation statique :**
  - Périphériques les plus courants
  - Liste dans *Documentation/devices.txt*
- **Choix d'un majeur**
  - Fixe, au hasard
    - Gérable pour une diffusion limitée
    - Risque de collision sinon
  - Dynamique
    - Souplesse pour la diffusion
    - Problème de création du fichier spécial dans /dev
      - Utilisation de devfs (obsolète)
      - Utilisation d'un script pour lire /proc/devices
      - Utiliser la réutilisation des majeurs dynamiques
        - ✓ Création des nœuds la première fois
        - ✓ insmod / rmmod
        - ✓ Suppression des nœuds la dernière fois

Remarques sur l'allocation dynamique

```
#!/bin/sh
module="MyModule"
device="MyDevice"
mode="664"

# Appel de insmod avec les arguments passes
/sbin/insmod ./$module.ko $* || exit 1

# Supprime l'eventuelle occurrence existante du noeud
rm -f /dev/${device}

major=$(awk "\\$2==\"$module\" {print \\$1}" /proc/devices)

mknod /dev/${device} c $major 0

# Modifie les permissions et change le groupe ("staff" ou "wheel" selon les distrib.)
group="staff"
grep '^staff:' /etc/group > /dev/null || group="wheel"
chgrp $group /dev/${device}
chmod $mode /dev/${device}
```

Remarques sur l'allocation dynamique

```
#!/bin/sh
module="MyModule"
device="MyDevice"

# appelle rmmod avec les arguments
/sbin/rmmod $module $* || exit 1

# supprime l'occurrence du noeud
rm -f /dev/${device}
```

Les opérations fichiers

- **1 périphérique ⇔ 1 structure file**
- **1 driver ⇔ une structure file_operations**
- **Chaque file possède son file_operation**
- **La structure file_operation est croissante :**
  - Possibles problèmes de portabilité
  - Accroissement par la fin de la structure
  - Nécessité de recompiler
  - Les pointeurs de fonctions non assignés sont initialisés à NULL

Structure file_operations

```
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*unlocked_ioctl) (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 inode *, struct dentry *, 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
   *);
struct module *owner;
```

+ opérations asynchrones : `aio_read, aio_write, aio_fsync`

+ opérations diverses : `sendpage, sendfile , fasync, etc.`

Structure file_operations

- **Syntaxe d'initialisation nommée**

```
struct file_operations Myfops = {
  .owner          =    THIS_MODULE,
  .llseek         =    Myllseek,
  .read           =    Myread,
  .write          =    Mywrite,
  .unlocked_ioctl =    Myioctl,
  .open           =    Myopen,
  .release        =    Myrelease };
```

Structure file

- Ne pas confondre avec libc
- Représente un fichier ouvert au niveau du kernel
- Champs courants :

```
mode_t f_mode;
loff_t f_pos;
unsigned int f_flags;
struct file_operations *f_op;
void *private_data;
struct dentry *f_dentry;
```

Structure inode

- **Représente les fichiers au niveau du kernel**
  - Différent d'une structure file (descripteur de fichier ouvert)
  - Unique pour chaque fichier
- **Deux champs intéressants au niveau des drivers :**
  - `dev_t i_rdev` : numéro du périphérique
    - Susceptible de modifications (e.g. version 2.5)
    - Utilisation de macros spécifiques à la place :
      - `unsigned int iminor(struct inode *inode)`
      - `unsigned int imajor(struct inode *inode)`
  - `struct cdev *i_cdev`
    - Représentation interne d'un périphérique caractère
    - Nécessaire pour enregistrer les opérations liées au périphérique
    - Remplace `register_chrdev` et `unregister_chrdev` des versions précédentes

Enregistrement d'un périphérique

- **Inclure** `<linux/cdev.h>`
- **Deux solutions :**
  - ```
    struct cdev *my_cdev = cdev_alloc();
    my_cdev->ops = &my_fops;
    ```
  - ```
    void cdev_init(struct cdev *cdev,
                   struct file_operations *fops);
    ```
- **Dans tous les cas :** `cdev->owner = THIS_MODULE;`
- **Enfin :** `int cdev_add(struct cdev *dev, dev_t num, unsigned int count);`
  - Retour < 0 => échec !
  - Sinon, les opérations peuvent être invoquées par le noyau => il faut que le périphérique soit prêt !
- **Suppression :** `void cdev_del(struct cdev *dev);`

Initialisation

Exemple

```
int sample_init(void)
{
	/* allocation dynamique pour les paires (major,mineur) */
	if (alloc_chrdev_region(&dev,0,1,"sample") == -1)
	{
		printk(KERN_ALERT ">>> ERROR alloc_chrdev_region\n");
		return -EINVAL;
	}

	/* recuperation et affichage */
	printk(KERN_ALERT "Init allocated (major, minor)=(%d,%d)\n",MAJOR(dev),MINOR(dev));

	/* allocation des structures pour les operations */
	my_cdev = cdev_alloc();
	my_cdev->ops = &fops;
	my_cdev->owner = THIS_MODULE;

	/* lien entre operations et periph */
	cdev_add(my_cdev,dev,1);

	return(0);
}
```

Terminaison

Exemple

```
static void sample_cleanup(void)
{
	/* liberation */
	unregister_chrdev_region(dev,1);
	cdev_del(my_cdev);
}
```

L'opération open

- **Tâches à accomplir :**
  - Vérifier l'état du périphérique
  - Initialiser le périphérique (première ouverture)
  - Identifier le mineur (éventuellement mettre à jour f_op)
  - Allouer et mettre à jour les données de privées
- **Obtention des structures propriétaires englobantes**
  - Le champ `inode` contient une référence à `i_cdev`
  - Ce champ `i_cdev` pointe vers un champ d'une structure propre au driver
  - Utilisation de `container_off(pointer, container_type, container_field);`

L’opération release

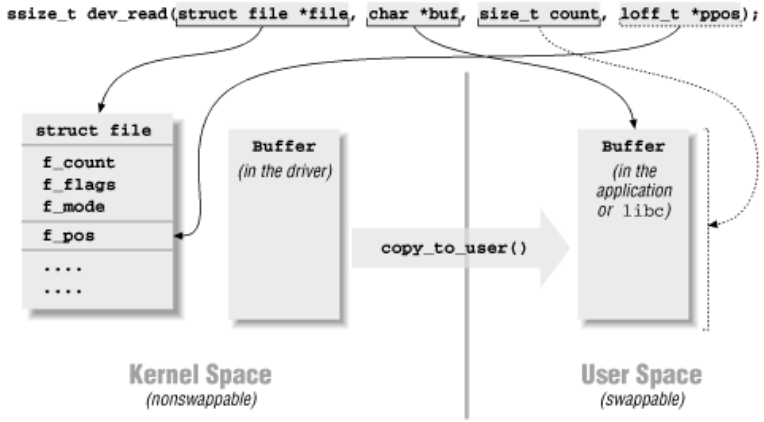
- **Tâches à accomplir**
  - Libérer les références à tout objet alloué lors de l’open
  - Désactiver le périphérique lors de la dernière fermeture

Lecture et écriture

- **Prototypes**

```
ssize_t read(struct file *filp, char *buff, size_t count,
             loff_t *offp);
ssize_t write(struct file *filp, const char *buff, size_t count,
              loff_t *offp);
```

- **Problème : copie entre les espaces d'adressage noyau et utilisateurs**
  - Opérations classiques directes impossibles
    - Mémoire virtuelle et pointeurs
    - Mappage mémoire différent (x86) => Pb références
  - Fonctions spéciales dans <asm/uaccess.h>



Lecture et écriture

- **Remarques :**
  - filp : structure file utilisée
  - count : taille requise pour l'opération
  - offp utilisé pour mettre à jour la position dans file
  - Le driver peut éventuellement transférer moins de données que demandé
  - Retour des fonctions :
    - <0 : code d'erreur en cas d'échec
    - >0 : la taille effectivement transférée
    - 0 : fin de fichier

Lecture et écriture

- **Prototypes**

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

- **Retour : le nombre d'octets restant à copier**
- **Accès à l'espace d'adressage utilisateur**
  - Les pages peuvent avoir été swappées
  - Le processus peut être endormi
    - Fonctions d'accès ré-entrantes
    - Sémaphores pour contrôler les accès concurrents

Lecture et écriture

Lecture

```
static ssize_t sample_read(struct file *f, char *buf, size_t size, loff_t *offset)
{
	int sizeToCopy = MIN(my_data.bufSize, size);
	printk(KERN_ALERT "Read called!\n");
	if(my_data.bufSize != 0)
	{
		if(copy_to_user(buf, my_data.buffer, sizeToCopy)==0)
		{
			my_data.bufSize = 0;
			kfree(my_data.buffer);
		}
		else
			return -EFAULT;
	}
	return sizeToCopy;
}
```

Ecriture

```
static ssize_t sample_write(struct file *f, const char *buf, size_t size, loff_t *offset)
{
	printk(KERN_ALERT "Write called!\n");
	if(my_data.bufSize != 0)
		kfree(my_data.buffer);
	my_data.buffer = (char *)kmalloc(size * sizeof(char), GFP_KERNEL);
	my_data.bufSize = size - copy_from_user(my_data.buffer, buf, size);
	return my_data.bufSize;
}
```

TP N° 2

Périphérique mémoire

- Lecture / écriture dans le même périphérique. La lecture est destructrice. L'écriture aussi.
- Lecture / écriture dans le même périphérique. La lecture est destructrice. L'écriture allonge le contenu.
- Lecture / écriture dans deux périphériques différents. La lecture est destructrice. L'écriture allonge le contenu.
- Lecture / écriture dans trois périphériques différents : un pour l'écriture, un pour la lecture destructrice, un pour la lecture non destructrice. L'écriture allonge le contenu.
- Lecture / écriture dans trois périphériques différents, un buffer par processus. Il faut créer un programme pour gérer les lectures / écritures…

Fonctionnalités avancées

Fonctionnalités avancées

- **La gestion de l'exclusion mutuelle**
  - Sémaphores *<obsolètes>*
  - Spinlocks
  - Mutex
- **Les IOCTL**
- **udev**

Exclusion mutuelle : les sémaphores

- **Include nécessaire : `<asm/semaphore.h>`**
- **Type : `struct semaphore`**
- **Initialisation :**
  - Cas général : `void sema_init (struct semaphore *sem, int count)`
  - Cas d'un mutex :
    - Initialisation statique
      - à 1 : `DECLARE_MUTEX(sem)`
      - à 0 : `DECLARE_MUTEX_LOCKED(sem)`
    - Initialisation dynamique
      - à 1 : `void init_MUTEX(struct semaphore *sem)`
      - à 0 : `void init_MUTEX_LOCKED(struct semaphore *sem)`

Exclusion mutuelle : les sémaphores

Opérations sur les sémaphores :

- P
  - Non interruptible : `void down(struct semaphore *sem)`
  - Interruptible : `void down_interruptibe(struct semaphore *sem)`
    - Retour égal à zéro : on a le sémaphore
    - Retour différent de zéro : opération interrompue
  - Sans attente : `void down_trylock(struct semaphore *sem)`
    - Retour égal à zéro : on a le sémaphore
    - Retour différent de zéro : sémaphore non accordé
- V
  - `void up(struct semaphore *sem)`

Exclusion mutuelle : les mutex

- Remplace l'interface `semaphore` devenue obsolète.
- Include nécessaire :

```
<linux/mutex.h>
```

- Déclaration statique :

```
static DEFINE_MUTEX(mymutex);
```

- Déclaration dynamique et initialisation :

```
struct mutex myMutex;
mutex_init(&myMutex);
```

- Acquisition :

```
mutex_lock(&myMutex);
```

- Libération :

```
mutex_unlock(&myMutex);
```

Exclusion mutuelle : les spinlocks

- **Identiques aux sémaphores, mais ne peuvent être endormis**
- **Petits, légers, efficaces !**
- **Attention dans le cas d'un système monoprocesseur mono-cœur non préemptif :**
  - Le spinlock va attendre sans pouvoir être endormis
  - On va boucler indéfiniment
- **Pour éviter ces problèmes les implémentations diffèrent suivant les cibles :**
  - Dans le cas d'un kernel non préemptible non SMP, pas disponibles !

Exclusion mutuelle : les spinlocks

- **Include nécessaire :**

```
<linux/spinlock.h>
```

- **Déclaration :**

```
spinlock_t   mySpinlock = SPIN_LOCK_UNLOCKED;
```

- **Acquisition du spinlock :**

```
spin_lock(&mySpinlock);
```

- **Libération du spinlock :**

```
spin_unlock(&mySpinlock);
```

Exclusion mutuelle : verrous lecteurs/écrivains

- **Include nécessaire :**

```
<linux/spinlock.h>
```

- **Déclaration :**

```
rwlock_t   myrwLock = RW_LOCK_UNLOCKED;
```

Ou

```
rwlock_t      myrwLock;
DEFINE_RWLOCK(myrwLock); // RW_LOCK_UNLOCKED obsolète
```

- **Acquisition du verrou en lecture :**

```
read_lock(&myrwLock);
```

- **Libération du verrou en lecture :**

```
read_unlock(&myrwLock);
```

- **Acquisition du verrou en écriture :**

```
write_lock(&myrwLock);
```

- **Libération du verrou en écriture :**

```
write_unlock(&myrwLock);
```

Les IOCTL

- **Exploités par un appel système depuis l'espace utilisateur :**

```
int ioctl(int fd, unsigned long cmd, char *argp)
```

  - fd : descripteur de fichier
  - cmd : commande à envoyer au périphérique
  - argp : arguments de la commande

- **Prototype au niveau du noyau :**

```
int (*ioctl)(struct inode *inode, struc file *filp,
        unsigned int cmd, unsigned long arg)
```

  - inode et filp : descripteurs du nœud
  - cmd : valeur passée par l'appel système
  - arg : l'argument passé (cast puis copy_from_user)
  - Include nécessaire : `<linux/ioctl.h>`

Obsolete (retiré en 2.6.35) : verrouille le Big Kernel Lock :-(

```
static struct file_operations query_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_close,
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,35))
    .ioctl = my_ioctl
#else
    .unlocked_ioctl = my_ioctl
#endif
};


#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,35))
static int my_ioctl(struct inode *i, struct file *f, unsigned int cmd, unsigned long arg)
#else
static long my_ioctl(struct file *f, unsigned int cmd, unsigned long arg) #endif
{
    switch(cmd){
    .....
    }
}
```

Les IOCTL

- **La commande est constituée de plusieurs champs de bits :**
  - type : magic number
    - Codé sur 8 bits
    - Choisir une valeur libre dans ioctl-number.txt
    - ‘G’, ‘O’, ‘Z’,‘g’, ‘k’, ‘x’ par exemple
  - Numéro :
    - Codé sur 8 bits
    - numéro de la commande dans la liste des commandes
  - Direction
    - Direction du transfert de données – le cas échéant
    - `IOC_NONE, IOC_READ, IOC_WRITE, IOC_READ|IOC_WRITE`
  - Taille
    - Dépendant de l’architecture
    - Taille des données échangées – le cas échéant

Les IOCTL

- **Obtention de la valeur d'une commande : on peut utiliser des macros prédéfinies dans `<linux/ioctl.h>` :**
  - `_IO(type, numéro)`
  - `_IOR(type, numéro, type_données)`
  - `_IOW(type, numéro, type_données)`
  - `_IOWR(type, numéro, type_données)`
- **Obtention des champs d'une commande :**
  - `_IOC_DIR(cmd)`
  - `_IOD_TYPE(cmd)`
  - `_IOC_NR(cmd)`
  - `_IOC_SIZE(cmd)`

Les IOCTL

- **Déclarations :**
  - Dans le driver :

```
#define SAMPLE_IOC_MAGIC 'k'
#define SAMPLE_IOCRESET _IO(SAMPLE_IOC_MAGIC, 0)
#define SAMPLE_IOC_MAXNR 0
```

  - Dans le programme user

```
#define SAMPLE_IOC_MAGIC 'k'
#define SAMPLE_IOCRESET _IO(SAMPLE_IOC_MAGIC, 0)
```

- **Implémentation driver**
  - Ne pas oublier de définir le .unlocked_ioctl dans la structure file_operations
  - Vérifier la commande passée avant tout traitement

Les IOCTL

Implémentation driver

- Exemple :

```
static int sample_ioctl (struct file *filp, unsigned int cmd, unsigned long arg)
{
    /* Vérification que la commande est valide
    * sinon : retourne ENOTTY (ioctl inconnu)  */
    if (_IOC_TYPE(cmd) != SAMPLE_IOC_MAGIC) return -ENOTTY;
    if (_IOC_NR(cmd) > SAMPLE_IOC_MAXNR) return -ENOTTY;
    switch(cmd)
    {
          case SAMPLE_IOCRESET:
                    /* On efface le buffer */
                    if(my_data.bufSize != 0)
                    {
                              my_data.bufSize = 0;
                              kfree(my_data.buffer);
                    }
                    break;
          default: /* Redondant, puisque déjà testé au dessus */
                    return -ENOTTY;
    }
    return 0;
}
```

Les IOCTL

Appel programme user

- Il faut inclure `<linux/ioctl.h>`
- Exemple :

```
int main(int argc, char* argv[])
{
            int file;

            if(argc ==2){
                        printf("Doing ioctl reset on %s\n",argv[1]);
                        file = open(argv[1], O_RDONLY);
                        if(file < 0){
                                    perror("open");
                                    printf("Error opening file %s!\n",argv[1]);
                                    return(errno);
                        }
                        ioctl(file,SAMPLE_IOCRESET,0);
                        close(file);
            }
            else
                        printf("usage: do_ioctl_reset <filename>\n");
            return 0;
}
```

udev

- **La création des nœuds dans /dev ne peut être manuelle**
  - Nécessite
    - des privilèges élevés
    - des compétences en administration
  - Ou la création statique de tous les nœuds possibles
- **On utilise udev (user-space device manager)**
  - Il s'exécute en mode user
  - Il dialogue avec hotplug (mode noyau)
  - Il crée les nœuds à la demande en fonction des périphériques présents
- **Il s'appui sur**
  - les services de sysfs
  - Des daemons (udevd) et des utilitaires (udevadm)
  - Des règles situées dans /etc/udev/rules.d/

udev

- **Le système de fichier virtuel sysfs**
  - Introduit dans le noyau 2.6, il s'appui sur ramfs
  - Il contient les informations relatives aux périphériques, aux pilotes et aux classes de périphériques :
    - /sys/block : un sous-répertoire pour chaque périphérique bloc présent dans le système. Chaque sous-répertoire contient
      - des fichiers, chaque fichier représentant un attribut (size, removable, etc.)
      - Un lien symbolique pointant vers le périphérique (dans /sys/devices)
      - Un sous-répertoire pour chaque partition, un sous-répertoire pour les statistiques, etc.

udev

Le système de fichier virtuel sysfs

- Les informations contenues :
  - /sys/bus : un sous-répertoire pour chaque type de bus supporté. Chaque bus a deux sous-répertoires :
    - devices : un lien symbolique vers chaque périphérique (dans /sys/devices) connecté sur ce bus
    - drivers : un sous-répertoire par pilote associé à ce bus. Ces sous-répertoires contiennent des attributs relatifs aux paramètres des pilotes, et des liens symboliques vers les périphériques auxquels ils sont liés.
  - /sys/class : un sous-répertoire par classe de périphérique (une classe = un type de périphérique : graphics, input, net, …)
    - Chaque sous-répertoire de classe contient un sous-répertoire par objet supporté (un périphérique peut contenir plusieurs objets : souris = « kernel mouse » + « input event »).
    - Chaque sous-répertoire objet contient des liens vers le périphérique et le pilote correspondant.

udev

Le système de fichier virtuel sysfs

- Les informations contenues :
  - /sys/devices : la hiérarchie complète des périphériques.
    - Correspondance de la relation de subordination physique vers la notion de répertoire / sous-répertoire.
    - Exceptions :
      - ✓ Platform devices : périphériques inhérents à une plateforme particulière. Par exemple : contrôleur série, floppy, etc.
      - ✓ System devices : composants du système ne correspondant pas à des périphériques physiques (pas de transfert de données) bien que pouvant s'appuyer sur du matériel. Exemple : CPU, timers, etc.
  - /sys/firmware : interface pour le microcode dépendant de la plateforme (BIOS, EFI, etc.)

udev

Le système de fichier virtuel sysfs

- Les informations contenues :
  - /sys/module : un sous-répertoire par module chargé dans le noyau. Il contient entre autre un attribut refcnt : valeur du compteur de références du module.
  - /sys/power : représente le sous-système de gestion de l'alimentation. Il contient deux attributs au moins :
    - disk : méthode de gestion de la veille sur disque
    - state : liste des états de veille (veille, suspension mémoire ou suspension disque)

udev

Les règles udev

- Elles permettent de gérer les périphériques qui apparaissent dans /dev :
  - Donner un nom consistent (indépendant de l'ordre de branchement / débranchement) à un périphérique
  - Modifier les permissions et les propriétés
  - Lancer des scripts au branchement / débranchement de périphérique
- Elles se situent dans /etc/udev/rules.d
- Elles sont traitées par ordre alphabétique
- Elles contiennent une partie « condition » (clefs de correspondances) et une partie « action » (clefs d'assignement).
- Exemple :

```
BUS=="usb", ATTR{idProduct}=="...", ATTR{idVendor}=="...",
  KERNEL=="...", NAME="%k", SYMLINK="usbMyDevice"
```

udev

- **Les règles udev**
  - Les clefs de correspondance utilisent principalement :
    - KERNEL : nom du périphérique donné par le noyau (ex. : sda)
    - SUBSYSTEM : nom du sous-système contenant le périphérique (ex : block)
    - DRIVER : nom du pilote de périphérique
    - Les attributs de sysfs, grâce à la clef ATTR
  - Les clefs d'assignation utilisent principalement :
    - NAME : nom du périphérique
    - SYMLINK : liens symboliques
    - MODE : les permissions sur le nœud (par exemple "0666").
    - PROGRAM : pour exécuter des programmes !!
    - Des caractères de substitution :
      - %k : nom donné au périphérique par le noyau
      - %n : numéro assigné au périphérique par le noyau (par exemple le numéro de partition d'un disque)
    - Des métacaractères : *, ?, [].

udev

Les utilitaires :

- udevadm : permet de gérer udev (info, test, control, etc.)
- Exemple :

```
udevadm info --query=property --name=/dev/sda
udevadm test /sys/class/block/sdb
```

  - Redémarrage de udev : `start_udev`
- Obsolète :
  - udevinfo : interroge udev sur les informations relatives à un périphérique

```
udevinfo -a -p /sys/block/sda
```

  - udevtest : teste la syntaxe d'une règle
  - Activer les messages de sortie de udevtest : udev_log = « yes » dans /etc/udev/udev.conf
  - udevstart : redémarrage de udev pour exécuter les nouvelles règles.

udev

- **La création d'un nœud dans /dev dans init_module**
  - Création de la structure de classe de périphérique :

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

  - Envoi d'un événement à udev pour la création des nœuds dans /dev :

```
struct device *device_create(struct class *class,
                struct device *parentDevice, dev_t devt,
                void *drvdata, const char *deviceName);
```

- **La suppression des nœuds dans cleanup_module**
  - Envoi d'un événement à udev pour la suppression du nœud :

```
void device_destroy(struct class * class, dev_t devt);
```

  - Suppression de la classe de périphérique :

```
void class_destroy(struct class *cls);
```

udev

```
int sample_init(void)
{
      /* allocation dynamique pour les paires (major,mineur) */
      if (alloc_chrdev_region(&dev,0,1,"sample") == -1)
      {
            printk(KERN_ALERT ">>> ERROR alloc_chrdev_region\n");
            return -EINVAL;
      }

      /* Création de la classe de périphérique */
      sample_class = class_create(THIS_MODULE, "SampleDevice");

      /* allocation des structures pour les operations */
      my_cdev = cdev_alloc();
      my_cdev->ops = &fops;
      my_cdev->owner = THIS_MODULE;
      /* lien entre operations et periph */
      cdev_add(my_cdev,dev,1);

      /* Envoi d'un événement à udev pour qu'il crée les noeuds dans /dev */
      device_create(sample_class, NULL, dev, NULL, "SampleDeviceNode");

      return(0);
}
```

```
static void sample_cleanup(void)
{
      /* liberation */
      unregister_chrdev_region(dev,1);
      /* Suppression du noeud dans /dev */
      device_destroy(sample_class, dev);
      /* Suppression du cdev */
      cdev_del(my_cdev);
      /* Suppression de la classe */
      class_destroy(sample_class);
}
```

TP N° 3

Périphérique mémoire

- Ajouter un verrou pour empêcher l'accès au périphérique tant qu'un processus y accède. Faire un programme C pour valider l'exclusion mutuelle (on fera un menu permettant de choisir open, read, write ou close).
- Utiliser un ioctl pour forcer l'effacement des buffers d'un processus. Appeler cet ioctl lorsque le programme C se termine.
- Ajouter la création / destruction automatique des nœuds dans /dev.
- Créer une règle qui nomme le nœud « myDevice » au lieu du nom défini ci-dessus, avec un synonyme « myDev » et des droits rw pour tout le monde.