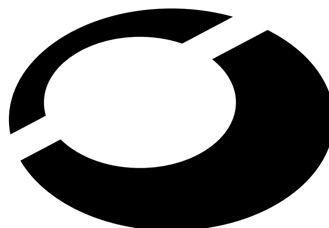
www.alcove.fr

Noyau Linux et pilotes de périphériques

Julien Gaulmin
julien.gaulmin@fr.alcove.com

version 1.60



l'informatique est libre

Alcôve

Copyright © 2000 Julien Gaulmin `julien.gaulmin@fr.alcove.com`, Alcôve

Ce document peut être reproduit, distribué et/ou modifié selon les termes de la Licence GNU de Documentation Libre (*GNU Free Documentation Licence*) dans sa version 1.1 ou ultérieure telle que publiée, en anglais, par la *Free Software Foundation* ; sans partie invariante, avec comme première de couverture (*front cover texts*) les deux premières pages, et sans partie considérée comme quatrième de couverture (*back cover texts*)

Une copie de la licence est fournie en annexe et peut être consultée à l'url :
http ://www.gnu.org/copyleft/fdl.html

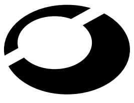
Alcôve
Centre Paris Pleyel
153 bd Anatole France
93200 Saint-Denis, France

Tél. : +33 1 49 22 68 00
Fax : +33 1 49 22 68 01
E-mail : alcove@alcove.fr, Toile : www.alcove.fr

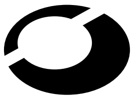
Table des matières

Chapitre 1 Architecture générale du noyau Linux 3
1.1 Introduction au noyau Linux 5
1.2 Concepts de base 10
1.3 Composants et mécanismes 17
Chapitre 2 Environnement de développement 32
2.1 Organisation des sources 34
2.2 Outils de développement 39
2.3 Méthodes de débogage noyau 44
2.4 Interface avec la communauté 50
2.5 Licences 54
Chapitre 3 Modules noyau 58
3.1 Manipulation 60
3.2 Implémentation, routines de base 63
3.3 Travaux pratiques 73

Chapitre 4 Services de base du noyau 75
4.1 Interface utilisateur 77
4.2 Gestion de la mémoire 86
4.3 Accès au matériel 91
4.4 Bus PCI 102
4.5 Transferts DMA 113
4.6 Interruptions et événements 120
4.7 Files d’attente 132



Architecture générale du noyau Linux

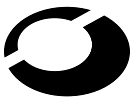


Objectifs du chapitre

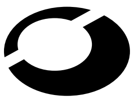
Introduction au noyau Linux ;

Principes de programmation ;

Composants et mécanismes.



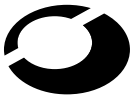
Introduction au noyau Linux



Histoire

Créé par **Linus Torvalds** en 1991 pour ses besoins personnels :
- découvrir les entrailles des systèmes d'exploitation,
- pas de vrai *OS* financièrement accessible à l'époque,
- insuffisances et inertie de Minix.

Des choix primordiaux pris par Linus dès le début :
- utilisation des outils GNU (compilateur et utilitaires),
- publication des sources sur Internet,
- choix de la licence GNU GPL,
- travail collaboratif => émergence rapide d'une communauté de développeurs.



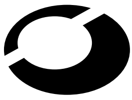
Première version stable (1.0) en 1992 ;

Ajout de la compatibilité POSIX dans la version 1.2 (1995) ;

Support des machines multiprocesseurs (SMP) et amélioration de la portabilité dans les versions 2.0 (1996) ;

Amélioration des performances générales, accroissement du nombre de plates-formes et de périphériques supportés dans les versions 2.2 (1999) ;

Amélioration des performances SMP et de la couche réseau dans les versions 2.4 (2001).



Principales caractéristiques

Noyau monolithique ;

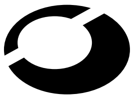
Architecture de type Unix (concept du "tout fichier") ;

Multitâche ;

Multiutilisateur ;

Multiprocesseur ;

Multiplate-forme ;



Support du chargement dynamique de modules noyau ;

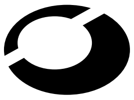
Protection de la mémoire entre processus ;

Chargement des exécutables à la demande ;

Partage des pages entre exécutables ;

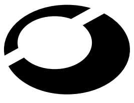
Taille des caches dynamique ;

Support des bibliothèques partagées...



Section 2

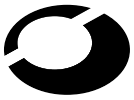
Concepts de base

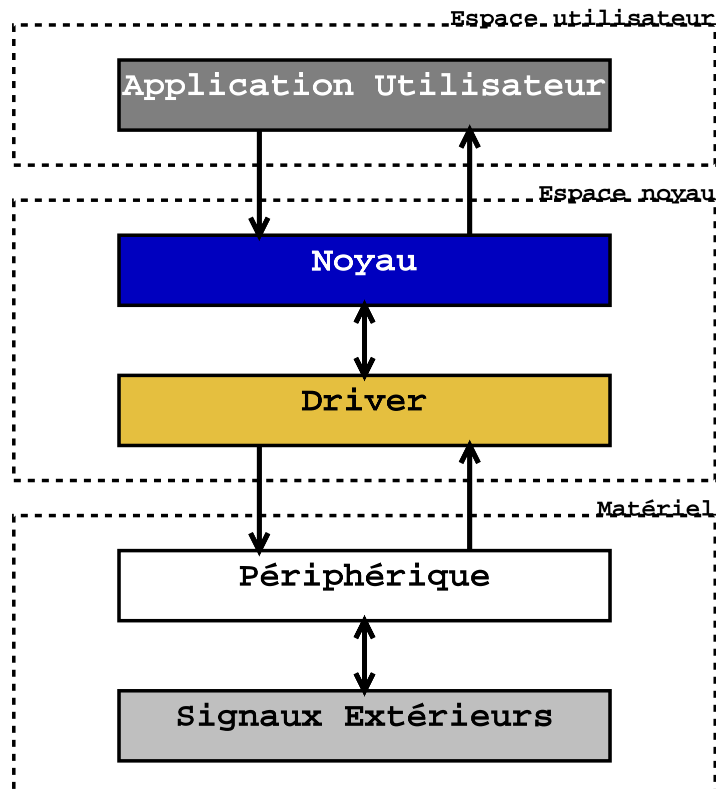


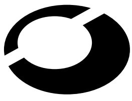
Monde utilisateur / monde noyau

Utilisation de la protection matérielle du microprocesseur ;

Distinction entre deux mondes :
- espace noyau où tout est permis (même le pire),
- espace utilisateur où les possibilités sont plus restreintes (ex : accès mémoire).



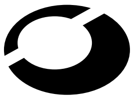




Trois types de transitions utilisateur/noyau :
– appels système,
– interruptions => gestionnaires d'interruptions du noyau,
– exceptions (ex : accès mémoire illégal, instruction illégale...).

Points d'entrées dans le noyau limités (˜190) pour les utilisateurs => appels système ;

Chaque processus utilisateur se décompose donc en deux parties :
– une partie noyau => les appels système (ex : `open()`, `read()`...),
– une partie utilisateur => le reste (ex : gestion, algorithme...).



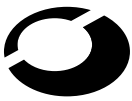
Concepts de base

Alcôve - Noyau Linux et pilotes de périphériques

Règle essentielle

"Toujours essayer de déporter la complexité en dehors du noyau"

Conséquences :
– le noyau doit rester le plus petit possible,
– les appels système doivent rester en nombre limité,
– la complexité réside dans la bibliothèque C (libC), ou
– dans les applications utilisateurs.

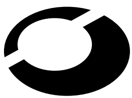


Pilote de périphérique noyau ou application en mode utilisateur ?

Pilote de périphérique noyau :
– accès direct à la mémoire et aux périphériques,
– rapidité (moins de couches logicielles).

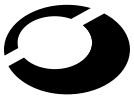
Application en mode utilisateur :
– déporte la complexité en dehors du noyau,
– exécution en mode protégé,
– indépendance vis-à-vis des interfaces, ports...

Ex : Ghostscript (imprimantes), Sane (scanners)...

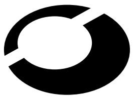


Structures du noyau

Plusieurs types de structures sont communément utilisés dans le noyau :
– structures complexes (ex : `task_struct` définie dans `<linux/sched.h>`),
– macro-commandes (ex : `current` définie dans `<asm/current.h>`),
– tableaux (ex : `task[]` déclaré dans `kernel/sched.c`),
– listes chaînées complexes (ex : liste de structures `vm_area_struct` organisées en arbre AVL - `<linux/mm.h>`),
– pseudo-objets composés de structures avec pointeurs sur fonctions en guise de méthodes (ex : structure `module` défini dans `<linux/module.h>`).



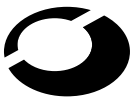
Composants et mécanismes



Gestion de la mémoire

Espace d'adressage noyau :
- vision linéaire de la RAM (pas de segmentation),
- pas de protection lors d'accès illégaux.

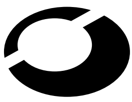
Espace d'adressage utilisateur :
- virtuel (ne correspond pas à de la mémoire réelle),
- privé pour chaque processus,
- allocation "à l'avance" (allocation réelle à l'utilisation),
- de taille maximum supportée par l'architecture.



Processus

Unité d'exécution élémentaire du système d'exploitation ;

Deux types :
- processus classique *(process)* : possède une réplique des ressources du processus père (amélioré par des techniques de *copy on write* ou de `vfork()`),
- processus léger *(thread)* : partage un maximum de ressources avec le processus père => changements de contextes accélérés.



Ordonnancement *(scheduling)*

Multitâche préemptif ;

Politique Unix de partage du temps machine entre processus ;

API POSIX.1b pour le temps réel ;

Linux n'est pas pour autant un noyau temps réel :
- **pas de préemption au niveau du noyau** ,
- temps d'exécution non prédictif,
- système non déterministe.

Gestion des priorités (41 niveaux standards + 99 niveaux temps réel).

Gestion des interruptions

Liées à un périphérique (ex : port, bus, interface, *timer...*) ;

Une interruption matérielle peut préempter un processus même s'il est en mode noyau ;

Traitement en deux phases :
– *top-half* : partie rapide et ininterruptible (ex : acquittement de l'interruption auprès du périphérique), et
– *bottom-half* : partie lente placée dans une file de tâches qui sera vidée par le *scheduler* (ex : traitement relatif à l'interruption).

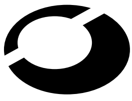
Possibilité de partage d'IRQ.

Exclusion mutuelle

Le noyau Linux n'est pas préemptif mais il est ré-entrant ;

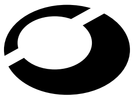
Sur une machine SMP, plusieurs appels système peuvent s'exécuter en concurrence ;

Sur une machine monoprocesseur, un appel système peut survenir alors qu'un autre appel s'est trouvé bloqué en attente d'une ressource ;



Un gestionnaire d'interruptions peut prendre la main alors que le noyau était en train de traiter un appel système ;

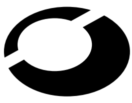
=> mécanismes de blocage *(locking)* pour protéger :
– les variables globales,
– les fonctions non ré-entrantes,
– les ressources *hardware* critiques.



Système de fichiers

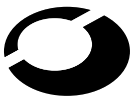
Pilier de base de tous les systèmes Unix ;

Deux types fondamentaux :
– réel : repose sur un périphérique de stockage matériel (ex : disques dur, CDROM...),
– virtuel : émulation d'un périphérique de stockage pour profiter de l'interface des systèmes de fichiers (ex : *drivers,* `/proc`...).



Couche d'abstraction pour l'écriture de systèmes de fichiers (VFS) :

– peut reposer sur tout périphérique bloc,
– implémente les mécanismes de base :
  - `open()`/`close()`, `read()`/`write()`, `lseek()`...,
  - héritage des mécanismes génériques si non implémentés.
– nécessite des utilitaires spécifiques (ex : formatage, vérification, *tuning...*).



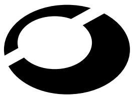
Réseau

Un des points forts de Linux ;

Supporte de nombreux périphériques et protocoles ;

Découpé en deux groupes de sous-systèmes :

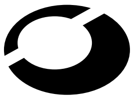
– les **pilotes d'interfaces** qui correspondent aux couches Physiques (1) et Liaison de données (2) du modèle OSI (ex : Ethernet, Jetons, PPP...), et
– les **protocoles** qui correspondent aux couches Réseau (3) et Transport (4) (ex : TCP, IP, IPX, Appletalk...).



Les pilotes d'interfaces utilisent des pilotes de périphériques pour accéder physiquement aux cartes ;

La communication entre pilotes de périphériques, pilotes d'interfaces et protocoles se fait par paquets (ex : structure `sk_buff` définie dans `<linux/skbuff.h>`) ;

La couche IP gère les tables de routage et la résolution des noms.



Modules dynamiques du noyau

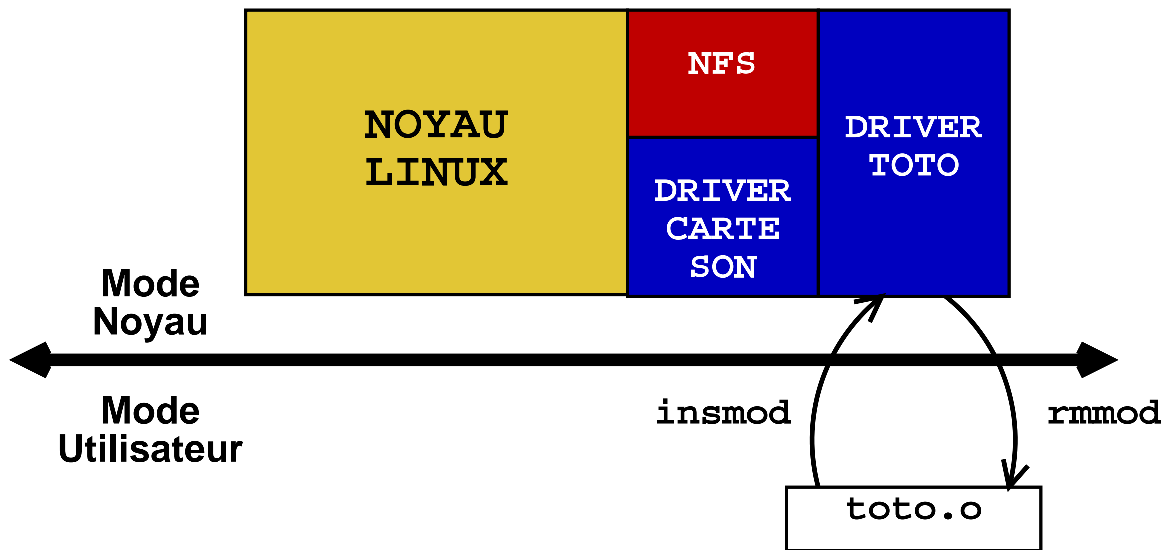
Fichiers objets (`.o`) implémentant une fonctionnalité du noyau ;

Utilisés pour réduire la taille du noyau et le rendre plus modulaire ;

Possibilité de charger/décharger dynamiquement *(run-time)* une fonctionnalité du noyau (ex : *driver,* protocole, système de fichiers...) ;

Gestion des dépendances entre modules par `modprobe` ;

Gestion de l'édition de liens dynamique par `insmod`.



Pilotes de périphériques *(drivers)*

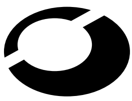
Programmes greffés au noyau pour faire l'interface entre le noyau et des périphériques matériels ;

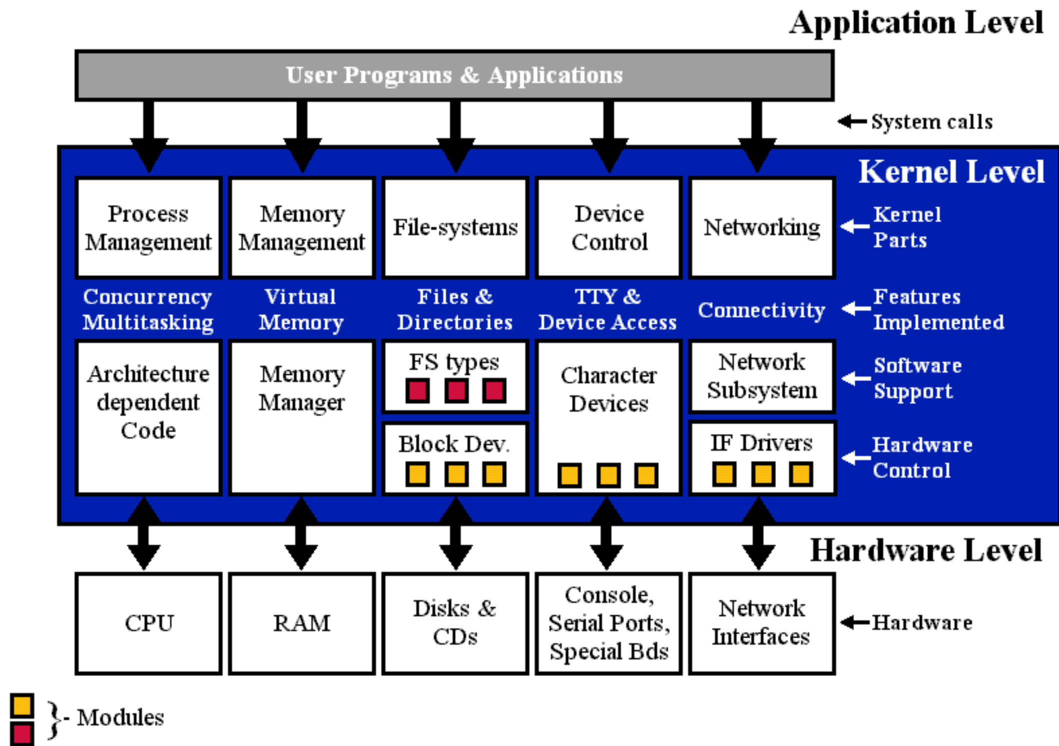
Il doivent donc présenter deux interfaces :
– une interface générique et *immuable* pour les utilisateurs, et
– une interface pour communiquer avec le noyau.

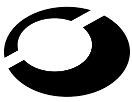
Sous Linux, l'interface utilisateurs prend la forme de fichiers spéciaux ;

On utilise donc les appels système standards associés aux fichiers (ex : `open()`, `read()`, `write()`...) ;

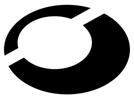
Peuvent être implémentés sous forme d'un ou plusieurs modules dynamiques ou compilés dans le noyau.







Environnement de développement



Objectifs du chapitre

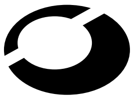
Organisation des sources ;

Outils de développement ;

Méthodes de débogage noyau ;

Interface avec la communauté ;

Licences.



Section 1

Organisation des sources

Distribution

Archives des sources dans des fichiers `linux-X.Y.Z.tar.bz2`
- `X` = *VERSION* (˜ 4 à 5 ans) : numéro de version,
- `Y` = *PATCHLEVEL* (˜ 1 à 2 ans) : numéro de sous-version (pair => stable, impair => instable),
- `Z` = *SUBLEVEL* (˜ 1 à 6 mois) : numéro de sous-sous-version.

Patches de mise à jour sous forme de fichiers `patch-X.Y.Z+1.bz2` ;

Fichiers `.sign` pour vérifier la signature numérique des archives et des *patches* (ex : `gpg -verify linux-X.Y.Z.tar.bz2.sign linux-X.Y.Z.tar.bz2`).

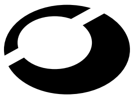
Arborescence

`Documentation/` : documentation relative au noyau et à ses sous-systèmes (ex : `sound/`, `DocBook/`...) ;

`arch/` : parties spécifiques aux architectures (ex : `alpha/`, `arm/`, `i386/`...) ;

`drivers/` : pilotes de périphériques (ex : `usb/`, `net/`...) ;

`fs/` : systèmes de fichiers (ex : `ext2/`, `fat/`...) et VFS ;

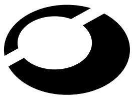


`include/` : fichiers d'en-tête (ex : `asm-i386/`, `linux/`, `net/`...) ;

`init/` : procédure d'initialisation *(boot)* du noyau ;

`ipc/` : mécanismes de communication inter-processus (ex : mémoire partagée, sémaphores...) ;

`kernel/` : coeur du noyau (ex : *scheduler,* signaux...) ;

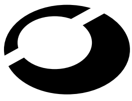


`lib/` : mini bibliothèque C pour utiliser dans le noyau (ex : `strcmp()`, `sprintf()`...) ;

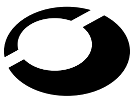
`mm/` : gestion de la mémoire ;

`net/` : interfaces et protocoles réseaux (ex : `ethernet/`, `ipv4/`...) ;

`scripts/` : scripts utilitaires pour la configuration et la compilation du noyau.



Outils de développement

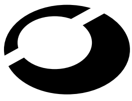


Compilation

Seuls outils de compilation supportés :
– GCC (utilisation d'optimisations spécifiques...), et
– GNU Make (syntaxe...).

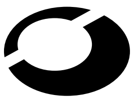
Version minimum des outils => `Documentation/Changes` ;

Utiliser l'optimisation `-O` de GCC pour que les fonctions *inline* des en-têtes soient interprétées ;



Ex : *Makefile*

```
CC = gcc

CCOPTS = -Wall -Wstrict-prototypes -O2

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

MODVERSIONS = -DMODVERSIONS -include
$(KERNELSRC)/include/linux/modversions.h

CFLAGS = -I$(KERNELSRC)/include $(CCOPTS) -DMODULE
$(MODVERSIONS) -D__KERNEL__

%.o : %.c

$(CC) $(CFLAGS) -c $< -o $@
```

Langage C et bibliothèque C

Pas de bibliothèque C (libC) pour la programmation noyau ;

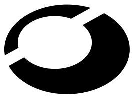
Fonctions utiles dans le répertoire `lib/` des sources du noyau ;

Normes de codage décrites dans `Documentation/CodingStyle` ;

Il est conseillé de découper les gros fichiers C en plusieurs fichiers aux fonctions bien définies (ex : couches logicielles) ;

Les fichiers objet (`.o`) peuvent ensuite être réunis en un seul module (un seul `module_init()`...) à l'aide de l'éditeur de liens *(linker)* ;

Ex : `ld -r part1.o part2.o -o module.o`



Patches

Collecte les modifications d'un ou plusieurs fichiers ;

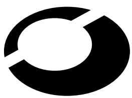
Nécessité de posséder le fichier ou l'arborescence d'origine ;

Ex : création d'un *patch*

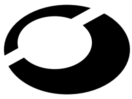
```
$ diff -urN <origine> <destination> >
patch_<description>
```

Ex : utilisation du *patch*

```
cd <origine>

patch -p1 < patch_<description>
```

Méthodes de débogage noyau



Console

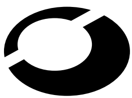
Fonction du noyau `printk()` pour envoyer des messages sur la console ;

Plusieurs niveaux de débogage définis dans `<linux/kernel.h>` (ex : `KERN_EMERG`, `KERN_INFO`...) ;

Les messages sont logués par le *daemon* `syslogd` (entrées `kern.*`) ;

L'application `dmesg` affiche et contrôle le tampon circulaire utilisé par `printk()` ;

Prototype :
– `int` **printk** `(const char *fmt, ...);`



KGDB

Patch `kgdb` pour le noyau (`http ://kgdb.sourceforge.net`) ;

Nécessité d'une deuxième machine :
– reliée à la cible par câble série, et
– utilise GDB-client pour déboguer la cible.

Facile à mettre en oeuvre et efficace (débogage en mode source).

KDB

Débogueur noyau embarqué ;

Développé par SGI (`http ://oss.sgi.com/projects/kdb/`) ;

Il comprend :
– un ensemble de commandes utilisateurs permettant de déboguer le noyau et les pilotes de périphériques en temps réel, et
– un débogueur noyau en mode assembleur (pas de mode source).

Débogueur difficile à exploiter de par sa complexité.

Profiling

Possibilité d'analyser le temps passé dans chaque fonction du noyau :
– paramètre de boot : `profile=n`,
– informations binaires dans `/proc/profile`,
– lecture grâce à l'utilitaire `readprofile`.

Permet de déceler :
– des dysfonctionnements structurels, et/ou
– des parties d'un *driver* à optimiser.

Autres possibilités

Débogage par requêtes :
- fichier virtuel dans `/proc`, ou
- appel système `ioctl()`.

Débogage avec GDB en lecture seule sur `/proc/kcore` ;

Utilisation de *User Mode Linux* pour les parties indépendantes du matériel ;

Analyse des sorties de l'application `strace` ;

En cas de *oops,* analyse et interprétation avec l'application `ksymoops` (utilise le fichier `System.map`).

Interface avec la communauté

Documentation

Le répertoire `Documentation/` des sources du noyau ;

Le projet de documentation Linux - LDP (`http ://www.linuxdoc.org`) : guides, *HOWTOs,* FAQs, pages de `man`... ;

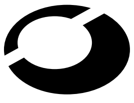
Les pages personnelles des mainteneurs (voir *newsgroups*) ;

Les autres documents du web (`http ://www.google.com/linux/`).

Informations

Listes de diffusion spécialisées (*newsgroups* - `http ://www.uwsg.indiana.edu/search/`) : `linux-kernel`, `kernel-newbies`, `linux-net`... ;

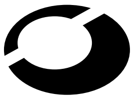
Les résumés hebdomadaires des discussions de la liste `linux-kernel` :
– *Kernel Traffic* (`http ://kt.zork.net/kernel-traffic/`), et
– *Kernel Notes* (`http ://www.kernelnotes.org`).



FAQ du noyau Linux (`http ://www.tux.org/lkml/`) ;

Forums de discussions : `comp.os.development.kernel`... ;

Courriers électroniques aux mainteneurs ou aux auteurs d'une partie de code source.



Section 5

Licences

La *GNU General Public License* (GPL)

Maintenue et défendue par la *Free Software Fundation* (FSF) ;

Donne le droit de copier, modifier et redistribuer le logiciel si les sources (+ modifications) sont aussi redistribuées (ou au moins librement disponibles) ;

Pas de notion de gratuité, seulement de liberté ;

Possibilité de vendre un logiciel sous licence GPL mais pas d'empêcher les utilisateurs de le modifier et de le redistribuer (gratuitement ou pas).

Alcôve - Noyau Linux et pilotes de périphériques

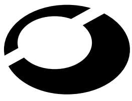
La *GNU Lesser General Public License* (LGPL)

Maintenue et défendue par la *Free Software Fundation* (FSF) ;

Plus permissive que la GPL ;

Possibilité d'utiliser du code LGPL dans un programme dont le code n'est pas libre (ex : application sous licence incompatible avec la GPL) ;

Utilisée principalement pour les bibliothèques (ex : GNU libC).



Les modules binaires de Linux

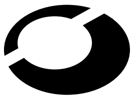
Close supplémentaire dans la licence GPL de Linux ;

Permet l'utilisation et la distribution de modules binaires ;

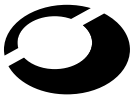
Doit utiliser uniquement l'interface standard des modules ;

Impossibilité de mettre à disposition des clients des modules binaires compatibles avec toutes les configurations possibles du noyau (ex : versions, *patches* externes...) ;

Légal mais **_très_ déconseillé** (pas d'aide ni de support de la communauté).



Modules noyau

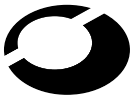


Objectifs du chapitre

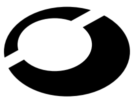
Manipulation ;

Implémentation, routines de base ;

Travaux pratiques.

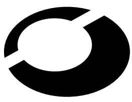


Manipulation

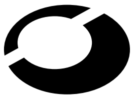


Ensemble d'outils pour manipuler les modules => `modutils` :

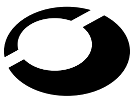
– `lsmod` : listage des modules chargés dans le noyau,
– `{ins,rm}mod` : insertion/extraction manuelle d'un module dans le noyau,
– `modprobe` : insertion et extraction automatique d'un module dans le noyau avec résolution des dépendances (chargement des autres modules nécessaires),



– `modinfo` : informations générales sur un module (ex : auteur, description, paramètres...),
– `depmod` : génère la liste des dépendances d'un ensemble de modules,
– `kmod` : *thread* du noyau pouvant demander le chargement d'un module quand il en a besoin (ex : montage d'une partition dont le système de fichiers n'est pas encore chargé) ; il remplace l'ancien *daemon* `kerneld`.



Implémentation, routines de base



Descriptions et arguments

Gérés par des macros dédiées (`<linux/module.h>`) :

– `MODULE_AUTHOR()`,
– `MODULE_DESCRIPTION()`,
– `MODULE_LICENSE()`,
– `MODULE_SUPPORTED_DEVICE()`,
– `MODULE_PARM()`,
– `MODULE_PARM_DESC()`.

Ex :

```
MODULE_PARM(mon_entier, "i");
MODULE_PARM_DESC(mon_entier, "Mon paramètre
entier");
```

Initialisation et libération

Macros définies dans le fichier `<linux/init.h>` ;

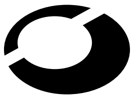
- `module_init()` : déclaration de la fonction d'initialisation (anciennement `int init_module(void);`),
- `module_exit()` : déclaration de la fonction de libération (anciennement `void cleanup_module(void);`).

Prototypes inchangés par rapport à l'ancienne méthode ;

Plus besoin d'utiliser des `#ifdef MODULE` ;

La fonction d'initialisation doit contenir tout le code nécessaire à l'initialisation du module (ex : allocations, initialisations matérielles, réservation des ressources...) ;

Au contraire, la fonction de libération doit défaire tout ce qui a été fait à l'initialisation (ex : libération de mémoire, désactivation du matériel, libération des ressources...) ;

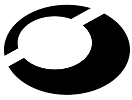


Linux utilise les optimisations de GCC pour libérer certaines parties de code inutiles en mémoire après leur instanciation (`<linux/init.h>`) :

– `__init` : fonctions d'initialisation (libérées après initialisation),
– `__exit` : fonctions de libération (ignorées si liées statiquement au noyau - 2.4 uniquement),
– `__initdata` : données utilisées dans une fonction d'initialisation (libérées après initialisation),
– `__exitdata` : données utilisées dans une fonction de libération (ignorées si liées statiquement au noyau - 2.4 uniquement).

Ex :

```
int __init ma_fonction_init(void) { /* ...code... */ }

module_init(ma_fonction_init);
```

Compteur d'usage

Chaque module possède un compteur d'usage ;

Permet d'éviter qu'un module soit déchargé quand il est utilisé ;

On utilise les macros suivantes (`<linux/module.h>`) :
– `MOD_INC_USE_COUNT` : incrémente le compteur d'utilisation,
– `MOD_DEC_USE_COUNT` : décrémente le compteur d'utilisation,
– `MOD_IN_USE` : indique la valeur du compteur d'utilisation.

À utiliser dès qu'on entre puis sort d'un point d'entrée du module.

Symboles

Les symboles exportés par le noyau sont disponibles dans `/proc/ksyms` ;

Utilisés par `insmod` pour l'édition de liens dynamique ;

La plupart des symboles sont exportés dans `kernel/ksyms.c` ;

Les modules peuvent exporter des symboles supplémentaires grâce à la macro `EXPORT_SYMBOL()` (`<linux/module.h>`) ;

La macro `EXPORT_NO_SYMBOLS()` permet de n'exporter aucun symbole ;

L'utilitaire `ksyms` permet d'afficher plus finement le contenu de `/proc/ksyms`.

Dépendances dynamiques

Possibilité de charger dynamiquement des modules depuis le noyau ;

La fonction `request_module()` demande à `kmod` de charger un module ;

Ex :

```
if (!mon_service) request_module();

if (!mon_service) return -ENODEV;
```

Intégration aux sources du noyau

Choisir le répertoire adéquat dans l'arborescence du noyau ;

Créer une sous-arborescence si nécessaire (ex : plusieurs éléments) ;

Éditer les fichiers `Config.in` et `Makefile` du répertoire hôte ;

Ajouter une entrée en s'inspirant des autres ;

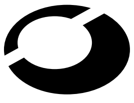
Ajouter une entrée d'aide dans `Documentation/Configure.help` ;

Modifications simplifiées pour les versions 2.4 du noyau (`Documentation/kbuild/makefile.txt`) ;

Les paramètres de boot sont ajoutés grâce à la macro `__setup()` ;

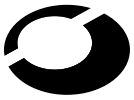
Prototypes (`<linux/init.h>`) :

– `void` **__setup** `(char *append, int (*extract) (char *));`



Section 3

Travaux pratiques

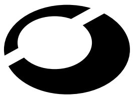


Travaux pratiques

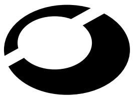
Création d'un module minimaliste *(HelloWorld)* ;

Gestion d'arguments au chargement ;

Intégration au noyau et édition statique de liens.



Services de base du noyau



Objectifs du chapitre

Interface utilisateur ;

Gestion de la mémoire ;

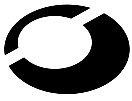
Accès au matériel ;

Bus PCI ;

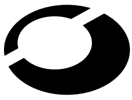
Transferts DMA ;

Interruptions et événements ;

Files d'attente.



Interface utilisateur



Système de fichiers

Accès au pilote de périphériques via un ou plusieurs fichiers spéciaux ;

Le pilote définit ses points d’entrée grâce à une structure `file_operations` (`<linux/fs.h>`) ;

Les routines standards d’accès au système de fichiers sont ainsi surchargées par les routines du pilote ;

L'ancienne pratique consistait à remplacer les méthodes non implémentées par des pointeurs `NULL` ;

Les derniers noyaux 2.2 et 2.4 utilisent une syntaxe de déclaration spécifique à GCC ;

Certaines méthodes non implémentées sont remplacées par des méthodes par défaut (ex : `open()`, `close()`...) ;

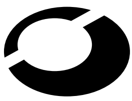
Les autres méthodes non implémentées retournent `-EINVAL` ;

Interface utilisateur

Alcôve - Noyau Linux et pilotes de périphériques

Ex :

```
static struct file_operations mondriver_fops = {
owner : THIS_MODULE, /* 2.4 uniquement */
read : mondriver_read,
write : mondriver_write,
open : mondriver_open,
release : mondriver_release,
};
```

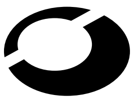


Majeurs et mineurs

Les fichiers spéciaux sont créés grâce à la commande `mknod` ;

Les fichiers spéciaux comportent un identifiant unique (`i_rdev` sur 16 bits) séparé en deux nombres (8 bits) :
– *major* : identifiant de type de pilote (ex : IDE, `/dev/hd*` => Major=3),
– *minor* : identifiant d'un pilote en particulier parmi un ensemble (ex : IDE, `/dev/hda1` => minor=1).

Les majeurs `0` et `255` sont réservés ;

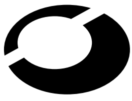


La liste des majeurs et mineurs déjà utilisés est tenue à jour dans `Documentation/devices.txt` et `<linux/major.h>` ;

Macros `MAJOR()` et `MINOR()` pour connaître le majeur ou le mineur d'un fichier spécial ;

Ex :

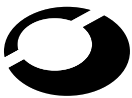
```
static int mondriver_open(struct inode *inode,
struct file *file) {
unsigned int minor = MINOR(inode->i_rdev);
/* ou alors file->f_dentry->d_inode->i_rdev */
/* ...reste du code... */
}
```



En pratique, un majeur est souvent supporté par un seul pilote de périphériques ;

Les mineurs servent alors à identifier les différentes cartes ou sous-ensembles pilotés ;

Gestion de plusieurs cartes identiques dans un *driver* grâce à ce procédé :
- structure décrivant chaque carte (ex : *buffers, mutex...*),
- tableau de ces structures indexé par le mineur,
- n fichiers dans `/dev` avec majeurs identiques et mineurs différents.



Valeurs de retour

Valeurs négatives d'erreurs définies dans `<linux/errno.h>` et `<asm/errno.h>` ;

Valeurs positives ou nulles en cas de fonctionnement correct ;

Toutes les fonctions du noyau utilisent cette normalisation ;

Transmission des valeurs d'erreur entre fonctions ;

Ex : argument invalide

```
if (arg > 3) return -EINVAL;
```

Ex : transmission d’erreur

```
int ret;

ret = fonction_noyau();

if (ret < 0) return ret;
```


Gestion de la mémoire

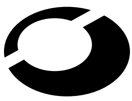
Mémoire dans le noyau

Plusieurs routines d'allocation mémoire cohabitent dans le noyau :

- `kmalloc()/kfree()` : alloue/désalloue un bloc de mémoire réelle contiguë dans le noyau (`GFP_KERNEL` => peut dormir, `GFP_ATOMIC` => atomique),
- `__get_free_pages()/free_pages()` : alloue/désalloue un nombre entier (2ˆn) de pages contigües de mémoire réelle (idem précédentes mais pour des besoins plus conséquents - limité à 128 ko),
- `vmalloc()/vfree()` : alloue/désalloue un bloc de mémoire virtuelle composé de plusieurs blocs discontiguës de mémoire réelle.

Prototypes (`<linux/{v}malloc.h>`) :

- `void *` **kmalloc** `(size_t size, int flags);`
- `void` **kfree** `(const void *addr);`
- `unsigned long` **__get_free_pages** `(int gfp_mask, unsigned long order);`
- `void` **free_pages** `(unsigned long addr, unsigned long order);`
- `void *` **vmalloc** `(unsigned long size);`
- `void` **vfree** `(void *addr);`



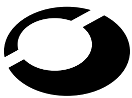
Transferts noyau/utilisateur

Le noyau utilise de la mémoire réelle ou virtuelle ;

Les processus utilisent chacun leur propre espace mémoire virtuel ;

Besoin de transférer les données de l'espace mémoire noyau vers l'espace mémoire du processus appelant et inversement ;

Les principales fonctions sont :
- `{get,put}_user()` : transfert d'une variable depuis/vers l'espace mémoire utilisateur (utilisation en *lvalue*), et
- `copy_{from,to}_user()` : transfert d'un *buffer* depuis/vers l'espace mémoire utilisateur.



Ces fonctions font toutes appel à la fonction `access_ok()` qui vérifie la validité des *buffers* utilisateurs ;

On peut s'affranchir de cet appel en utilisant les mêmes fonctions préfixées par __ (déconseillé).

Prototypes (`<asm/uaccess.h>`) :
- `int` **get_user** `(lvalue, addr);`
- `int` **put_user** `(expression, addr);`
- `void` **copy_{to,from}_user** `(unsigned long dest, unsigned long src, unsigned long len);`
- `int` **access_ok** `(int type, unsigned long addr, unsigned long size);`

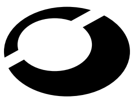
Accès au matériel

Ports d'entrées/sorties

Architecture Intel => ports d'entrées/sorties (E/S) ;

Plages d'adresses utilisées par les périphériques présents sur le bus d'E/S (ex : registres de contrôle) ;

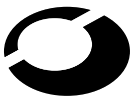
On accède à ces ports depuis le noyau grâce aux fonctions :
– `in{b,w,l}()`/`out{b,w,l}()` : lit/écrit 1, 2 ou 4 octets consécutifs sur un port d'E/S,
– `in{b,w,l}_p()`/`out{b,w,l}_p()` : lit/écrit 1, 2 ou 4 octets consécutifs sur un port d'E/S et fait une pause (une instruction),
– `ins{b,w,l}()`/`outs{b,w,l}()` : lit/écrit des séquences de 1, 2 ou 4 octets consécutifs sur un port d'E/S.



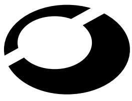
Les pilotes peuvent accéder à des ports d'E/S qui ne leur appartiennent pas (ex : *probing*) ;

Pour éviter les conflits, le noyau peut réserver des ports pour des pilotes de périphériques :
– `request_region()` : réserve un port d'E/S,
– `check_region()` : vérifie si le port n'est pas déjà réservé,
– `release_region()` : libère un port d'E/S.

La liste des ports déjà réservés apparaît dans `/proc/ioports`.



Prototypes x86 (`<asm/io.h>`) :
– `unsigned char` **inb** `(unsigned short port);`
– `void` **outb** `(unsigned char byte, unsigned short port);`
– `unsigned char` **insb** `(unsigned short port, void *addr, unsigned long count);`
– `void` **outsb** `(unsigned short port, void *addr, unsigned long count);`

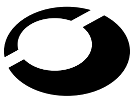


Mémoire d’entrées/sorties

Les périphériques d’E/S peuvent aussi posséder de la mémoire partagée (ex : *frame buffer* des cartes graphiques) ;

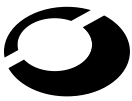
Cette mémoire est accessible comme de la mémoire centrale (contrairement aux ports d’E/S) ;

Cependant, le noyau manipule des adresses linéaires virtuelles (la mémoire centrale est elle-même *mappée* à l’adresse `PAGE_OFFSET`) ;



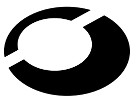
On doit donc dans certains cas (ex : PCI => adresses > `PAGE_OFFSET`) *mapper* les plages d’entrées/sorties dans l’espace linéaire du noyau :

- `ioremap()` : *mappe* une plage d’adresses physiques sur une plage d’adresses linéaires (semblable à `vmalloc()`),
- `iounmap()` : libère une plage préalablement *mappée* .

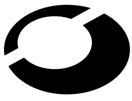


Pour éviter les conflits, le noyau peut réserver des plages mémoire d'E/S pour des pilotes de périphériques (2.4 uniquement) :
- `request_mem_region()` : réserve une plage d'E/S,
- `check_mem_region()` : vérifie si la plage n'est pas déjà réservée,
- `release_mem_region()` : libère une plage d'E/S.

La liste des plages mémoire d'E/S partagées qui sont déjà reservées apparaît dans `/proc/iomem`;

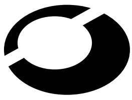


Prototypes (`<asm/io.h>` et `<linux/ioport.h>`) :
- `void *` **ioremap** `(unsigned long offset, unsigned long size);`
- `void` **iounmap** `(void *addr);`
- `struct resource *` **request_{mem_}region** `(unsigned long start, unsigned long n, const char *name);`
- `int` **check_{mem_}region** `(unsigned long start, unsigned long n);`
- `void` **release_{mem_}region** `(unsigned long start, unsigned long n);`



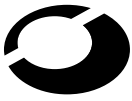
On accède ensuite à la mémoire partagée des E/S grâce aux fonctions suivantes :

– `read{b,w,l}()`/`write{b,w,l}()` : lit/écrit respectivement 1, 2 ou 4 octets consécutifs dans de la mémoire d'E/S,
– `memcpy_{from,to}io()` : lit/écrit un bloc d'octets consécutifs dans de la mémoire d'E/S,
– `memset_io()` : remplit une zone de mémoire d'E/S avec une valeur fixe,
– `virt_to_bus()`/`bus_to_virt()` : traduction entre adresses virtuelles linéaires et adresses réelles sur le bus.



Prototypes x86 (`<asm/io.h>`) :

– `char` **readb** `(void *addr);`
– `void` **writeb** `(char byte, void *addr);`
– `void *` **memcpy_{from,to}io** `(void *dest, const void *src, size_t count);`
– `void *` **memset_io** `(void *addr, int pattern, size_t count);`
– `unsigned long` **virt_to_bus** `(volatile void *addr);`
– `void *` **phys_to_virt** `(unsigned long addr);`



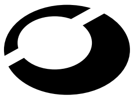
Accès aux bus

Les bus les plus courants (ex : PCI, EISA...) bénéficient généralement d'une interface d'encapsulation du noyau ;

On peut trouver la définition des fonctions relatives à ces bus dans les fichiers `<linux/<nom_du_bus>.h>` ;

Les méthodes d'accès aux bus sont généralement documentées dans `Documentation` (ex : `Documentation/pci.txt`) ;

Prototype de driver réseau utilisant le PCI dans `drivers/net/pci-skeleton.c` (2.4 uniquement).



Bus PCI

Concepts

Bus le plus répandu sur les architectures supportées par Linux ;

Configuration dynamique des ressources à l'initialisation du noyau ;

Interface simple d'encapsulation du bus dans le noyau ;

Gère les différentes versions et variantes du bus PCI (*Hot-plug,* CardBus...) ;

Utilisation de l'outil `lspci` (lecture de `/proc/bus/pci`) et `/proc/pci` pour débogage et informations ;

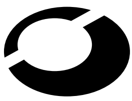
Le sous-système PCI repose sur un mécanisme de gestion dynamique des périphériques et de leurs états (2.4 uniquement).

Implémentation

On définit une structure `pci_driver` qui pointe sur les routines de contrôle du pilote ;

Ex :

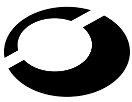
```
static struct pci_driver mon_pci_driver = {
name : "mondriver",
id_table : mondriver_id_table,
probe : mondriver_probe,
remove : mondriver_remove
};
```



Les fonctions `pci_{register,unregister}_driver()` servent alors à ajouter/supprimer le pilote de la liste des pilotes PCI ;

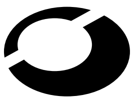
Le tableau `mondriver_id_table` décrit les périphériques PCI (identifiants standardisés) que supporte le pilote ;

À chaque fois qu'un périphérique définit dans le tableau `mondriver_id_table` change d'état (détection, suppression...), la fonction correspondante de la structure `pci_driver` est appelée (`mondriver_probe()`, `mondriver_remove()`...) ;



Ex :

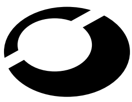
```
static struct pci_device_id mondriver_id_table[]
__devinitdata = {
{0x10B7, 0x9200, PCI_ANY_ID, PCI_ANY_ID, 0, 0,
madata_1},
{0x1105, 0x8400, PCI_ANY_ID, PCI_ANY_ID, 0, 0,
madata_2},
{0,}
};
MODULE_DEVICE_TABLE(pci, mondriver_id_table);
```



La fonction `mondriver_probe()` doit contenir tout le code nécessaire à l'initialisation d'un périphérique (ex : allocations privées, initialisations matérielles, réservation des ressources...) ;

Au contraire, la fonction `mondriver_remove()` doit défaire tout ce qui a été fait à l'initialisation (ex : libération des données allouées et des ressources, désactivations matérielles...) ;

Les fonctions d'initialisation et de libération du pilote ne s'occupent alors que des paramètres globaux du pilote (pilote PCI, nombre majeur, ressources et données globales...) ;

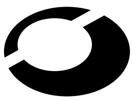


Bus PCI

Alcôve - Noyau Linux et pilotes de périphériques

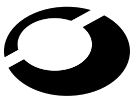
Les fonctions d'initialisation du pilote sont les suivantes :
- `pci_{register,unregister}_driver()` : enregistrement/désenregistrement du pilote PCI,
- `pci_module_init()` : encapsulation de `pci_register_driver()` pour les modules.

Prototypes (<linux/pci.h>) :
- `int` **pci_register_driver** `(struct pci_driver *drv);`
- `void` **pci_unregister_driver** `(struct pci_driver *drv);`
- `int` **pci_module_init** `(struct pci_driver *drv);`
- `int` **probe** `(struct pci_dev *dev, const struct pci_device_id *id);`
- `void` **remove** `(struct pci_dev *dev);`

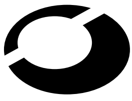


Un ensemble de fonctions utilitaires permet de manipuler les ressources et les en-têtes PCI :

– `pci_{set,get}_drvdata()` : attache/récupère une donnée privée (ex : pointeur sur structure privée) sur le descripteur du périphérique,
– `pci_{enable,disable}_device()` : active/désactive le périphérique au niveau matériel,
– `pci_{read,write}_config_{byte,word,dword}()` : lit/écrit 8, 16 ou 32 bits dans les en-têtes PCI,

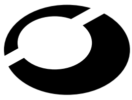


– `pci_find_capability()` : détermine si le périphérique supporte une fonctionnalité PCI donnée (ex : `PCI_CAP_ID_PM`),
– `pci_{request,release}_regions()` : réserve/libère automatiquement tous les ports et plages mémoire d'E/S du périphérique,
– `pci_resource_{start,end,len,flags}()` : retourne l'adresse de début/fin, la longueur ou les caractéristiques d'une ressource d'E/S (port ou mémoire).



Prototypes (<linux/pci.h>) :

- void * **pci_get_drvdata** (struct pci_dev *dev);
- void **pci_set_drvdata** (struct pci_dev *dev, void *data);
- int **pci_enable_device** (struct pci_dev *dev);
- void **pci_disable_device** (struct pci_dev *dev);
- int **pci_read_config_{byte,word,dword}** (struct pci_dev *dev, int addr, u{8,16,32} *val);



- int **pci_write_config_{byte,word,dword}** (struct pci_dev *dev, int addr, u{8,16,32} val);
- int **pci_find_capability** (struct pci_dev *dev, int capability);
- int **pci_request_regions** (struct pci_dev *dev, char *name);
- int **pci_release_regions** (struct pci_dev *dev);
- unsigned long **pci_resource_{start,end,len,flags}** (struct pci_dev *dev, int bar);

Transferts DMA

Concepts

Direct Memory Access ;

Permet au processeur de se décharger des transferts entre mémoire centrale et périphériques au détriment d'un contrôleur spécialisé (DMAC) ;

Le principe consiste à configurer le contrôleur DMA du périphérique en lui indiquant :
– l'adresse bus (fonction `virt_to_bus()`) du *buffer* DMA,
– la direction de transfert,
– la quantité de données.

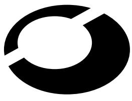
Le noyau fournit des services de haut niveau pour la configuration de *buffers* dédiés au transfert DMA vers/depuis des périphériques PCI.

Configuration

Interface documentée dans `Documentation/DMA-mapping.txt` (2.4 uniquement) ;

Les fonctions permettant de configurer les périphériques PCI pour supporter les transferts DMA sont les suivantes :
- `pci_set_master()` : active le contrôle du bus *(bus-mastering)* du périphérique,
- `pci_dma_supported()` : détermine les limitations d'adressage DMA du périphérique,
- `pci_set_dma_mask()` : détermine et configure le masque d'adressage DMA pour le périphérique.

Prototypes (`<{asm,linux}/pci.h>`) :
- `void` **pci_set_master** `(struct pci_dev *dev);`
- `int` **pci_dma_supported** `(struct pci_dev *dev, dma_addr_t mask);`
- `int` **pci_set_dma_mask** `(struct pci_dev *dev, dma_addr_t mask);`

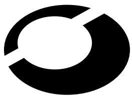


Transferts

Il existe deux types de *mapping* mémoire permettant de réaliser des transferts DMA entre un *buffer* noyau et un périphérique :

– *mapping* synchrone *(consistent)* : le processeur peut accèder aux données du *buffer* sans avoir à instancier explicitement un transfert DMA pour se synchroniser avec le périphérique,

– *mapping* asynchrone *(streaming)* : les données du *buffer* ne sont consistantes que pour un seul transfert et doivent être resynchronisées explicitement à chaque nouveau transfert.



Les fonctions de manipulation des *buffers* DMA sont les suivantes :

– `pci_{alloc,free}_consistent()` : alloue/libère et prépare/annule un *buffer* pour réaliser des transferts DMA synchrones,

– `pci_{map,unmap}_single()` : prépare/annule un *buffer* pour réaliser des transferts DMA asynchrones dans une direction donnée,

– `pci_dma_sync_single()` : réinitialise un *buffer* DMA asynchrone pour un nouveau transfert.

Prototypes (`<asm/pci.h>`) :

– `void *` **pci_alloc_consistent** `(struct pci_dev *dev, size_t size, dma_addr_t *dma_handle);`

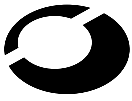
– `void` **pci_free_consistent** `(struct pci_dev *dev, size_t size, void *buffer, dma_addr_t dma_handle);`

– `dma_addr_t` **pci_map_single** `(struct pci_dev *dev, void *buffer, size_t size, int direction);`

– `void` **pci_unmap_single** `(struct pci_dev *dev, dma_addr_t dma_addr, size_t size, int direction);`

– `void` **pci_dma_sync_single** `(struct pci_dev *dev, dma_addr_t dma_handle, size_t size, int direction);`

Interruptions et événements

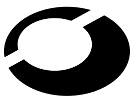


Supervision d'événements d'entrées/sorties

Événements d'E/S souvent asynchrones et imprédictibles (ex : clavier) ;

Deux méthodes pour superviser les E/S :
– attente active/scrutation *(polling),* ou
– interruptions.

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

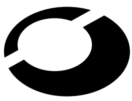


Ex : *polling*

```
for (;;) {
if (lit_etat(carte) & ETAT_FIN) 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).

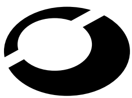


Gestionnaires d'interruptions *(interrupt handler)*

Propriétés :
- doivent êtres rapides, et
- ne doivent pas appeler des routines qui peuvent dormir (ex : `kmalloc()` non atomique).

Pour ces raisons, la gestion des interruptions est découpée en deux parties :
- une partie rapide et ininterruptible (gestionnaire d'interruptions), et
- une partie lente placée dans une file de tâches *(bottom-half).*

La première peut exister sans la seconde mais pas l'inverse ;

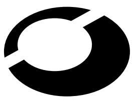


Un gestionnaire d'interruptions est déclaré grâce à la fonction `request_irq()` ;

Il est libéré grâce à la fonction `free_irq()` ;

Prototypes (`<linux/sched.h>`) :

- `int` **request_irq** `(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long flags` /* *SA_{INTERRUPT,SHIRQ}* */ `, const char *device, void *dev_id);`
- `void` **free_irq** `(unsigned int irq, void *dev_id);`

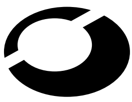


Quand l'interruption `irq` survient, la fonction `handler()` est appelée ;

Le champ `dev_id` sert à identifier les périphériques en cas de partage d'IRQ (`SA_SHIRQ`) ;

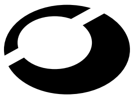
Il peut être employé pour transmettre au gestionnaire d'interruptions une structure spécifique au périphérique ;

La liste des IRQ déjà déclarées est disponible dans `/proc/interrupts` ;



Deux moments opportuns pour déclarer un gestionnaire d'interruptions :

– à l'installation du pilote (`module_init()`) => désinstallation au déchargement du module (`module_exit()`), ou
– à la première ouverture du pilote par un programme utilisateur (point d'entrée `open()`) => il faut alors :
  - installer le gestionnaire au premier `open()` (utilisation d'un compteur d'utilisation ou de `MOD_IN_USE` si module), et
  - désinstaller le gestionnaire au dernier `close()`.

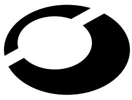


bottom-halves

Fonctions du noyau utilisées pour la gestion de tâches asynchrones ;

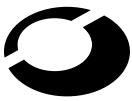
Ordonnancées à chaque retour d'appel système, d'exception ou de gestionnaire d'interruption ;

Parmi celles-ci, trois sont particulièrement importantes pour les pilotes de périphériques (`<linux/interrupt.h>`) :
– `IMMEDIATE_BH` : consomme une queue de tâches (`tq_immediate`) qui est alimentée par les *drivers,*



– `TQUEUE_BH` : appelée à chaque *tick* d'horloge si la queue de tâches `tq_timer` n'est pas vide,
– `NET_BH` : permet de signaler un événement aux couches réseau supérieures.

Les parties lentes des gestionnaires d'interruptions (tâches) sont mises en queue de `tq_immediate` (ou autre) par le gestionnaire d'interruptions :
– déclaration d'une tâche (structure `tq_struct`, voir section "Queues de tâches"),
– mise en queue de la tâche (fonction `queue_task()`).

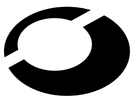


On active la *bottom-half* avec la fonction `mark_bh()` ;

Prototypes (`<linux/tqueue.h>`) :

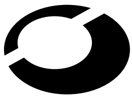
– `int` **queue_task** `(struct tq_struct *bh_pointer, task_queue *bh_list);`

– `void` **mark_bh** `(int nr);`



Ex :

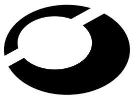
```
void ma_routine_bh(void *) { /* ... code bottom-half ... */ }

static struct tq_struct ma_tache;

void mon_handler_irq(int irq, void *dev_id, struct
pt_regs *regs)

{ ...

PREPARE_TQUEUE(& ma_tache, ma_routine_bh, &
tq_immediate);

queue_task(& ma_tache, & tq_immediate);

mark_bh(IMMEDIATE_BH);

... }
```



Mécanismes de *bottom-halves* améliorés pour les versions 2.4 du noyau ;

Permettent d'accroître les performances sur des machines SMP ;

Deux nouveaux concepts dérivés des *bottom-halves* :
- *tasklets* : peuvent s'exécuter sur différents CPU (mais une seule instance à la fois),
- *softirqs* : peuvent s'exécuter sur différents CPU (plusieurs instances simultanément).



Section 7

Files d'attente

Concepts

Un processus peut attendre un événement dans le noyau (ex : données) ;

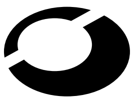
Le processus se met dans une file d’attente et relâche le CPU ;

Lorsque l’événement intervient, il suffit de réveiller les processus de la file correspondante.

Implémentation

Les routines de manipulation des files d’attente sont les suivantes :

- `{interruptible_}sleep_on{_timeout}()` : endort le processus courant (état inéligible), de façon interruptible ou pas et avec ou sans délais d’expiration, et le place dans une file d’attente,
- `wake_up{_interruptible}()` : réveille les processus d’une file d’attente et les rend éligibles,
- `wait_event{_interruptible}()` : endort le processus courant, de façon interruptible ou pas, en attente d’un événement.

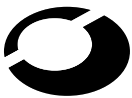


Lors du réveil après une attente interruptible, il faut s'assurer que le processus n'a pas été réveillé par un signal (dans quel cas l'appel système doit retourner `-ERESTARTSYS`) ;

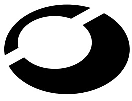
La fonction `signal_pending()` permet de s'en assurer ;

Initialisation d'une file d'attente (type `wait_queue_head_t`) :
– à la déclaration => `DECLARE_WAIT_QUEUE_HEAD()`, ou
– durant l'exécution (*runtime*) => `init_waitqueue_head()`.

Prototypes (`<linux/wait.h>`) :
– **DECLARE_WAIT_QUEUE_HEAD** `(name);`
– `void` **init_waitqueue_head** `(wait_queue_head_t *wq);`

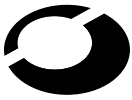


Prototypes (`<linux/sched.h>`) :
– `void` **{interruptible_}sleep_on** `(wait_queue_head_t *wq);`
– `long` **{interruptible_}sleep_on_timeout** `(wait_queue_head_t *wq, long timeout);`
– `void` **wake_up{_interruptible}** `(wait_queue_head_t *wq);`
– `void` **wait_event{_interruptible}** `(wait_queue_head_t *wq, condition);`
– `int` **signal_pending** `(struct task_struct *task);`

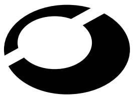


Ex :

```
static DECLARE_WAIT_QUEUE_HEAD(ma_file);

void *lecture_hard (void *adresse) { ...

interruptible_sleep_on(& ma_file);

if (signal_pending(current)) return -ERESTARTSYS;

... }

void mon_handler(int irq, void *priv, struct
pt_regs *regs) { ...

wake_up_interruptible(& ma_file);

...}
```

Périphériques mode caractère



Objectifs du chapitre

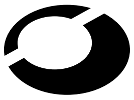
Concepts généraux ;

Particularités des *char devices* ;

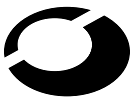
Mécanismes de base ;

Méthodes usuelles ;

Travaux pratiques.



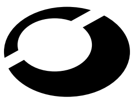
Concepts généraux



Types de périphériques

Plusieurs types de périphériques :

- caractère *(char)* : lectures et écritures séquentielles (ex : carte graphique) => non *bufferisés,*
- bloc *(block)* : accès aléatoires et par blocs de données (ex : disque dur) => *bufferisés,*
- réseau *(net)* : matériel de communication utilisé par les différentes couches réseau (ex : carte Ethernet).

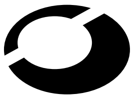


Les périphériques *char* et *block* possèdent une ou plusieurs entrées dans `/dev` ;

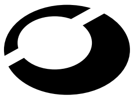
Distinction à la création du fichier spécial avec `mknod` ;

Ex : `# mknod /dev/mon_driver c 63 2`

Les nombres majeurs ne sont pas partagés entre les deux types de périphériques => 256 chacun.



Particularités des *char devices*



Particularités des *char devices*

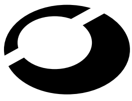
S’apparentent à des fichiers classiques ;

Lectures et écritures séquentielles ;

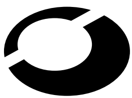
Temps d’accès pouvant varier selon la position de la donnée (ex : bandes) ;

Transferts de données de tailles arbitraires ;

Ex : cartes graphiques, bandes, imprimantes...



Mécanismes de base



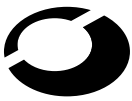
Déclaration

Déclaration du pilote à son initialisation ;

Réservation du nombre majeur ;

Fonctions `{register,unregister}_chrdev()` pour enregistrer/désenregistrer le pilote caractère auprès du noyau (tableau `chrdevs[M]`) ;

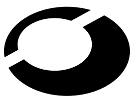
Système de fichiers virtuel `devfs` dans les noyaux 2.4 => utilisation de `devfs_{register,unregister}_chrdev()` ;



Prototypes (`<linux/fs.h>`) :

– `int` **register_chrdev** `(unsigned int major, const char *name, struct file_operations *fops);`

– `int` **unregister_chrdev** `(unsigned int major, const char *name);`

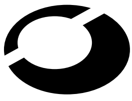


Possibilité d'allocation dynamique du majeur en mettant `major` à `0` ;

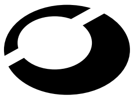
Les périphériques et leurs nombres majeurs sont listés dans `/proc/devices` ;

Enregistrement du pilote dans la fonction d'initialisation de celui-ci (déclarée avec `module_init()`) ;

Désenregistrement du pilote dans la fonction de destruction de celui-ci (déclarée avec `module_exit()`).



Méthodes usuelles



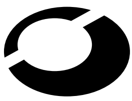
`open()`**/**`release()`

Appelées à l'ouverture/fermeture du fichier spécial ;

Ces méthodes peuvent ne pas être déclarées mais le *driver* ne sera pas averti lors de ces événements (déconseillé) ;

Prototypes (`<linux/fs.h>`) :

- `int` **open** `(struct inode *inode, struct file *file);`
- `int` **release** `(struct inode *inode, struct file *file);`



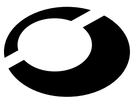
`read()`**/**`write()`

Généralement utilisées pour lire/écrire des données sur le périphérique ;

Méthodes les plus couramment implémentées ;

Prototypes (`<linux/fs.h>`) :

- `ssize_t` **read** `(struct file *file, char *buffer, size_t size, loff_t *offset);`
- `ssize_t` **write** `(struct file *file, const char *buffer, size_t size, loff_t *offset);`



`ioctl()`

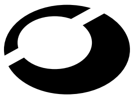
Généralement utilisée pour contrôler le périphérique ;

La plus complète de toutes les méthodes ;

On peut implémenter toutes les fonctionnalités d'un *driver* avec des commandes `ioctl()` ;

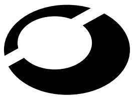
Prototype (`<linux/fs.h>`) :

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



Autres méthodes utiles

– `llseek()` : modifie la position courante de lecture/écriture,
– `poll()` : base pour l'implémentation des appels système `select()` et `poll()`,
– `mmap()` : *mappe* une partie des données du périphérique dans l'espace mémoire du processus utilisateur,
– `owner` : pointe sur le module qui contrôle le pilote (2.4 uniquement).



Prototypes (`<linux/fs.h>`) :
– `loff_t` **llseek** `(struct file *file, loff_t offset, int orig);`
– `unsigned int *` **poll** `(struct file *file, struct poll_table_struct *tab);`
– `int *` **mmap** `(struct file *file, struct vm_area_struct *vma);`

Utilisation

On déclare les méthodes implémentées dans une structure `file_operations`;

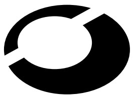
Utilisation de la syntaxe améliorée de GCC ;

Les déclarations peuvent être faites dans le désordre ;

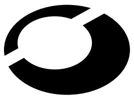
Plus besoin de remplacer les méthodes non implémentées par un pointeur NULL ;

Ex :

```
static struct file_operations mondriver_fops = {
owner : THIS_MODULE, /* 2.4 uniquement */
read : mondriver_read,
write : mondriver_write,
ioctl : mondriver_ioctl,
open : mondriver_open,
release : mondriver_release,
};
```



Travaux pratiques



Pilote simple avec méthodes `read()` et `write()` qui affichent un message ;

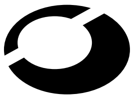
Utilisation d'un *buffer* linéaire pour que `read()` lise (de façon destructive) les données écrites par `write()` ;

Endormissement si pas de données à lire (mode bloquant) ;

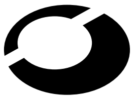
Appel `ioctl()` pour gérer le mode bloquant/non-bloquant ;

Implémentation multipériphérique (utilisation du mineur) ;

Gestion des interruptions clavier et *bottom-halves* .



Périphériques mode bloc

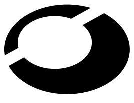


Objectifs du chapitre

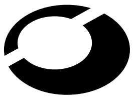
Particularités des *block devices* ;

Mécanismes de base ;

Méthodes supplémentaires.



Particularités des *block devices*

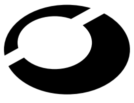


Périphériques à accès aléatoires ;

Temps d'accès considérés indépendants de la position de la donnée et de l'état du périphérique ;

Périphériques pouvant accueillir un système de fichiers ;

Possibilité d'accès direct ou par VFS ;

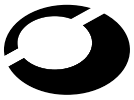


Particularités des *block devices*

Classiquement utilisés pour les disques ou assimilés ;

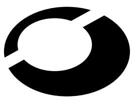
Utilisent les différents caches et tampons du noyau ;

Ex : disques, disquettes, disques virtuels, fichiers images...



Section 2

Mécanismes de base



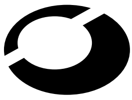
Déclaration

Déclaration du pilote à son initialisation ;

Réservation du nombre majeur ;

Fonctions `{register,unregister}_blkdev()` pour enregistrer/désenregistrer le pilote bloc auprès du noyau (tableau `blkdevs[M]`) ;

Utilisation de `devfs_{register,unregister}_blkdev()` pour les noyaux 2.4 ;



Prototypes (`<linux/fs.h>`) :

– `int` **register_blkdev** `(unsigned int major, const char *name, struct file_operations *fops);`
– `int` **unregister_blkdev** `(unsigned int major, const char *name);`

Paramétrage

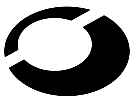
Pour pouvoir utiliser les services de *bufferisation* des entrées/sorties du noyau, le pilote doit renseigner les informations suivantes :

– `hardsect_size[M][m]` : taille des secteurs matériels du périphérique,
– `blk_size[M][m]` : capacité du périphérique (en kilooctets),
– `blksize_size[M][m]` : taille des blocs de données utilisés pour les transferts avec le périphérique (en octets),
– `read_ahead[M]` : nombre de secteurs supplémentaires à lire à chaque accès.

Couches logicielles

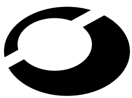
Deux niveaux de programmation d'un pilote de périphérique bloc :

– haut niveau : utilisation des interfaces d'abstraction du noyau (ex : *buffer cache*) pour orchestrer les demandes de lecture/écriture en provenance du VFS,
– bas niveau : pilotage physique du périphérique et prise en compte de ses spécificités (ex : périphériques amovibles).



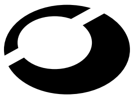
Principales fonctions du sous-système bloc :

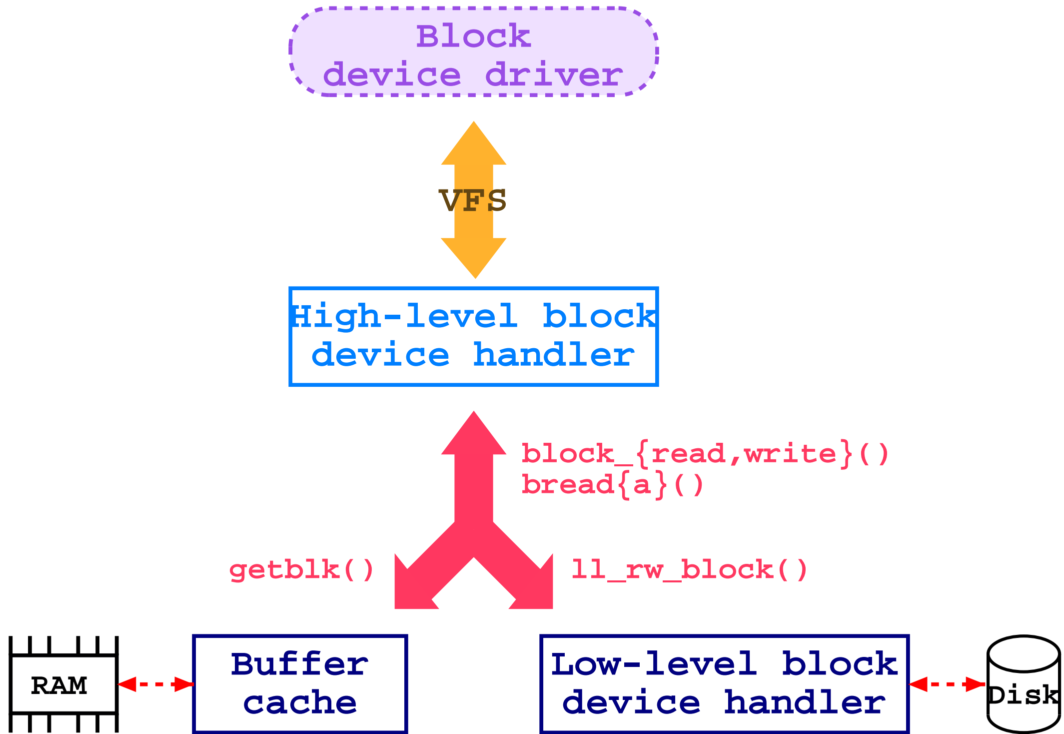
- `block_{read,write}()` : demande la lecture/écriture de blocs (appelés directement par les méthodes `read()`/`write()` des *drivers*),
- `getblk()` : regarde dans le cache si les données ne sont pas déjà disponibles (appelé par `block_{read,write}()`),
- `ll_rw_block()` : lance la lecture/écriture effective des données grâce au *driver* bas niveau (appelé par `getblk()` si les données ne sont pas dans le cache).

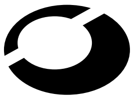


Prototypes (`<linux/fs.h>`) :

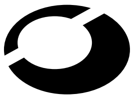
- `ssize_t` **block_read** `(struct file *filp, char *buf, size_t count, loff_t *ppos);`
- `ssize_t` **block_write** `(struct file *filp, const char *buf, size_t count, loff_t *ppos);`
- `struct buffer_head *` **getblk** `(kdev_t dev, int block, int size);`
- `void` **ll_rw_block** `(int rw, int nr, struct buffer_head *bh[]);`







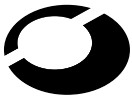
Méthodes supplémentaires



Les pilotes de périphériques bloc utilisent, en plus des méthodes déjà décrites pour les pilotes caractères, des méthodes supplémentaires :

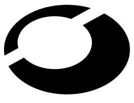
– `f{a}sync()` : synchronise de façon synchrone/asynchrone les données virtuelles (ex : *buffer cache*) avec les données physiques sur le périphérique,
– `check_media_change()` : vérifie si un périphérique amovible a été changé.

Même si ces méthodes existent aussi pour les périphériques caractères, elles ne sont pas fréquemment utilisées pour ceux-ci.

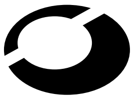


Prototypes (`<linux/fs.h>`) :

– `int *` **fsync** `(struct file *file, struct dentry *dentry);`
– `int` **fasync** `(int fd, struct file *file, int on);`
– `int *` **check_media_change** `(kdev_t dev);`



Périphériques réseau

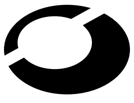


Objectifs du chapitre

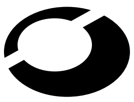
Concepts ;

Mécanismes de base ;

Travaux pratiques.



Concepts



Particularités des *net devices*

Périphériques de communication ;

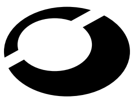
Interface avec le matériel pour fournir une abstraction de celui-ci aux divers interfaces et protocoles réseau (modèle OSI) ;

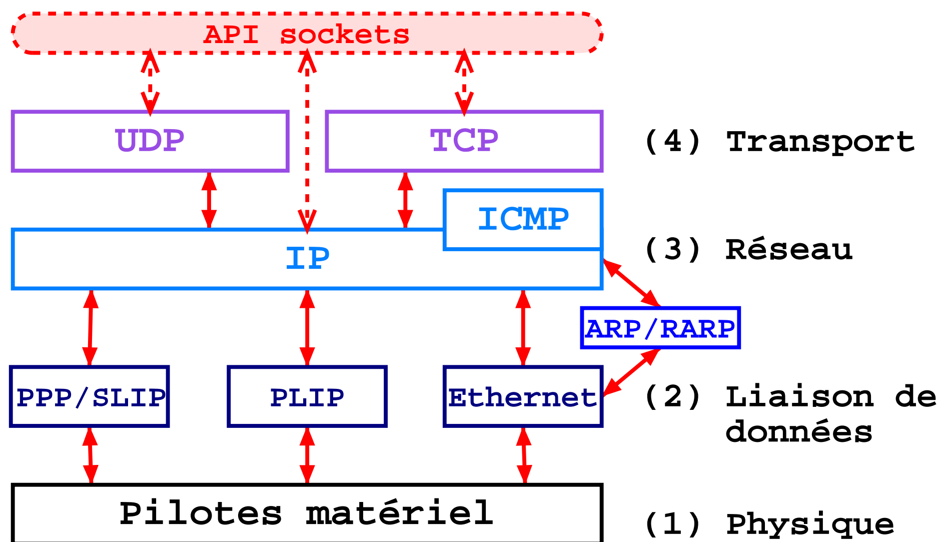
Reposent sur un modèle en couches ;

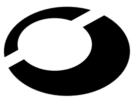
Pas de fichier spécial dans `/dev` ;

Utilisation de l'outil `tcpdump` pour débogage ;

Ex : cartes Ethernet, ISDN...







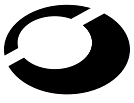
Paramétrage

Nommage par type d'interface (ex : ethX, pppX...) ;

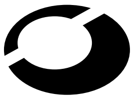
Type point à point (ex : PPP, PLIP...) ou non (ex : Ethernet, jetons...) ;

Support de la diffusion *(broadcast, multicast)* ;

Activation ou pas du filtrage matériel (ex : `/proc/sys/net/ipv4/conf/<device>/rp_filter`).



Mécanismes de base

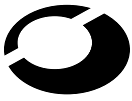


Déclaration

Déclaration du pilote à son initialisation ;

Fonctions `{register,unregister}_netdev()` pour enregistrer/désenregistrer le pilote réseau auprès du noyau ;

Prototypes (`<linux/netdevice.h>`) :
- `int` **register_netdev** `(struct net_device *dev);`
- `void` **unregister_netdev** `(struct net_device *dev);`



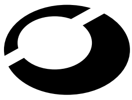
Interface

Pas de fichier spécial dans `/dev` ;

Pas de transfert de données direct au pilote de périphériques ;

Contrôle du pilote avec `ifconfig` ;

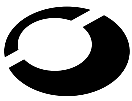
Le pilote est utilisé par la couche réseau supérieure (NET3) ;



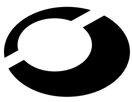
Trois interfaces pour le pilote :
– matérielle,
– couche supérieure NET3,
– contrôle du pilote (`ifconfig`).

Le pilote définit ses points d'entrée grâce à une structure `net_device` (`<linux/netdevice.h>`) ;

Initialisation de la structure `net_device` (`device` pour les noyaux 2.2) grâce à l'une des fonctions `<interface>_setup()` (ex : `ether_setup()`) ;



Surcharge des attributs et des méthodes définis dans `net_device` :
- `name[]` : nom du périphérique,
- `init()` : initialisation du pilote,
- `open()`/`stop()` : mise en/hors service de l'interface (appelées par `ifconfig up/down`),
- `hard_start_xmit()` : transmission de données au matériel,
- `get_stats()` : récupération des statistiques (utilisée par `ifconfig`),
- `do_ioctl()` : configuration spécifique du pilote (15 commandes à partir de `SIOCDEVPRIVATE`).



Prototypes (`<linux/netdevice.h>`) :
- `int` **open** `(struct net_device *dev);`
- `int` **stop** `(struct net_device *dev);`
- `int` **hard_start_xmit** `(struct sk_buff *skb, struct net_device *dev);`
- `struct net_device_stats *` **get_stats** `(struct net_device *dev);`
- `int` **do_ioctl** `(struct net_device *dev, struct ifreq *ifr, int cmd);`

Tampons `sk_buff`

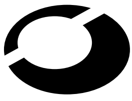
Socket buffers ;

Conteneurs pour les paquets de données réseau ;

Contiennent des informations d'adresses et des données ;

Méthodes de manipulation des `sk_buff` :
– `{alloc,kfree}_skb()` : alloue/désalloue un `sk_buff`,
– `skb_{put,push}()` : empile/dépile des données dans un `sk_buff`,
– `skb_pull()` : extrait les données d'un `sk_buff` (suppression des en-têtes).

Prototypes (`<linux/skbuff.h>`) :
– `struct sk_buff *` **alloc_skb** `(unsigned int size, int gfp_mask);`
– `void` **kfree_skb** `(struct sk_buff *skb);`
– `unsigned char *` **skb_{put,push,pull}** `(struct sk_buff *skb, unsigned int len);`



Transmission de données

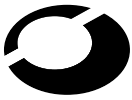
Méthode dédiée à la transmission => `hard_start_xmit()` ;

Appelée par les couches supérieures (NET3) ;

Fin de transmission signalée par une interruption matérielle ;

Les données à transmettre sont stockées dans des tampons `sk_buff` ;

Elles sont extraites pour être transmises à la carte selon la procédure matérielle de celle-ci.



Réception

Également signalée par une interruption ;

Même IRQ que pour la transmission => distinction par lecture de l’état sur la carte ;

Le gestionnaire d’interruptions crée un tampon `sk_buff` ;

Les données récupérées sur la carte sont stockées dans le tampon ;

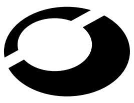
Le traitement du tampon est déporté aux couches supérieures (NET3) par la *bottom-half* `NET_BH` ;

Les noyaux 2.4 remplacent ce mécanisme par un appel à la fonction `netif_rx()` qui transmet le tampon à une *softirq* dédiée.

Prototype (`<linux/netdevice.h>`) :
– `void` **netif_rx** `(struct sk_buff *skb);`

Interface couche NET3/pilote

Méthodes permettant la synchronisation entre le pilote de périphériques et la couche NET3 :
– `netif_start_queue()` : initialisation de la queue de tampons `sk_buff` qui permet à la couche NET3 d'accumuler les paquets à transmettre au pilote,
– `netif_{stop,wake}_queue()` : suspend/reprend le remplissage de la queue de `sk_buff` par NET3.

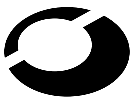


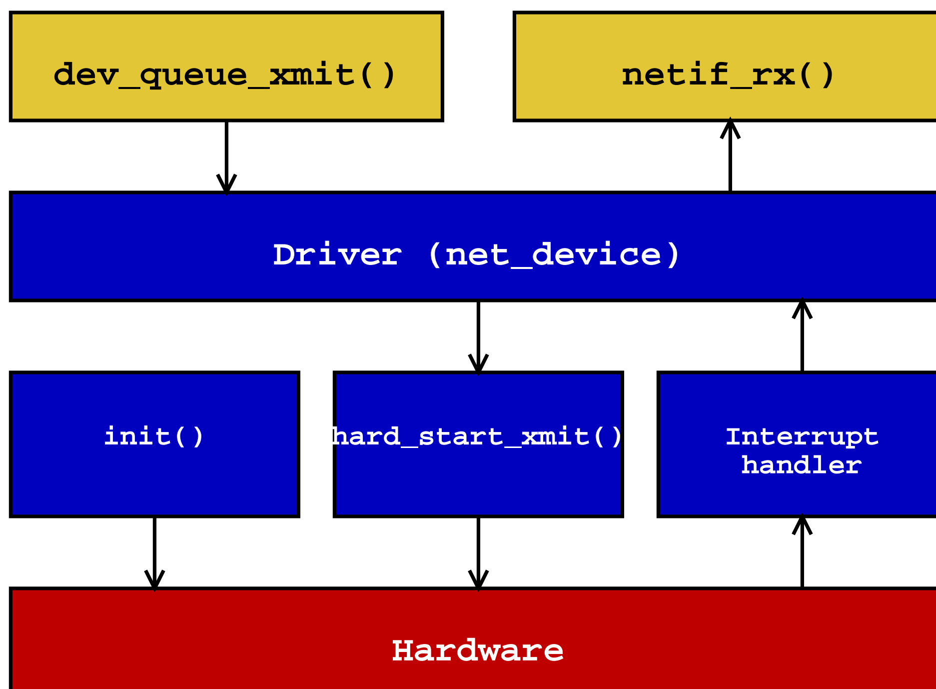
Les transferts de données entre la couche NET3 et le pilote sont effectués par les macros :

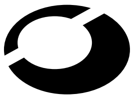
– `dev_queue_xmit()` : envoie des paquets via un appel à la méthode `hard_start_xmit()` du pilote,
– `netif_rx()` : commande le traitement d'un paquet par la couche NET3 via une *softirq* (2.4 uniquement).

Prototypes (`<linux/netdevice.h>`) :

– `void` **netif_{start,stop,wake}_queue** `(struct net_device *dev);`





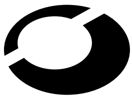


Résolution d'adresses

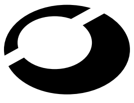
Utilisation du protocole ARP ;

Méthode générique pour les pilotes de cartes Ethernet ;

Peut être surchargée dans le cas d'autres interfaces :
– quasi-Ethernet (ex : PLIP, SLIP...), ou
– autres (possibilité de gérer ses propres en-têtes matériels).



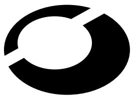
Travaux pratiques



Analyse du pilote *dummy* ;

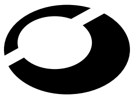
Analyse et mise en place du pilote de périphériques SNULL *(Simple Network Utility for Loading Localities)* ;

Analyse et mise en place du pilote d'interface INSANE *(INterface SAmple for Network Errors)* .



Chapitre 8

Concepts avancés



Objectifs du chapitre

Synchronisation ;

Reports d'exécution ;

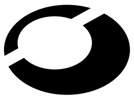
Interface `/proc` ;

Système de fichiers `devfs` ;

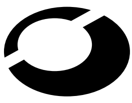
Threads noyau ;

Divers (types, conversions, listes...) ;

Travaux pratiques.



Synchronisation



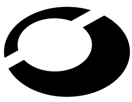
Problématique

Noyau non préemptif mais :

– un processus en mode noyau peut relâcher le CPU :
  - volontairement (ex : `schedule()`, `sleep_on()`...), ou
  - involontairement (ex : `kmalloc()`, `{get,put}_user()`...).

– un processus en mode noyau peut être interrompu par un gestionnaire d'interruptions,

– un autre processus peut se trouver en mode noyau au même moment mais sur un autre processeur.



Pour toutes ces raisons, on doit protéger les données ou ressources partagées afin d'éviter les accès concurrents ;

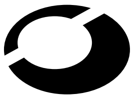
Plusieurs mécanismes, plus ou moins coûteux, existent dans le noyau pour résoudre ces problèmes :

– opérations atomiques,

– désactivation des interruptions,

– sémaphores, et

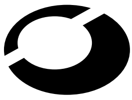
– verrous.



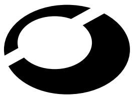
Opérations atomiques

Opérations atomiques sur les bits :
- `{set,clear,change,test}_bit()`,
- `test_and_{set,clear,change}_bit()`.

Prototypes (`<asm/bitops.h>`) :
- `void` **{set,clear,change}_bit** `(int nr, volatile void *addr);`
- `int` **test_*** `(int nr, volatile void *addr);`

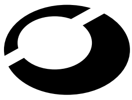


Opérations atomiques sur des entiers (type `atomic_t`) :
- `atomic_{read,set,add,sub,inc,dec}()`,
- `atomic_dec_and_test()`,
- `atomic_inc_and_test_greater_zero()`,
- `atomic_{clear,set}_mask()` (x86 uniquement).



Prototypes (`<asm/atomic.h>`) :

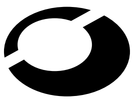
- `int` **atomic_read** `(atomic_t *v);`
- `void` **atomic_set** `(atomic_t *v, int i);`
- `void` **atomic_{add,sub}** `(int i, volatile atomic_t *v);`
- `void` **atomic_{inc,dec}** `(volatile atomic_t *v);`
- `int` **atomic_{inc,dec}_and_test*** `(volatile atomic_t *v);`
- `void` **atomic_{clear,set}_mask** `(int mask, volatile atomic_t *addr);`



Désactivation des interruptions

Protection des ressources en concurrence avec des gestionnaires d'interruptions uniquement :

- `cli()`/`sti()` : désactive/active les interruptions sur tous les processeurs,
- `{save,restore}_flags()` : sauvegarde/restore les drapeaux d'interruptions *(IF)* sur tous les processeurs,
- `{disable,enable}_irq()` : désactive/active une seule ligne d'IRQ sur tous les processeurs.



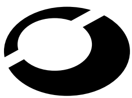
On peut appliquer certaines de ces macros uniquement sur le processeur courant en les préfixant par __ ;

Prototypes (`<asm/system.h>` et `<asm/irq.h>`) :

– `void` **{cli,sti}** `(void);`
– `void` **{save,restore}_flags** `(unsigned long old);`
– `void` **{enable,disable}_irq** `(unsigned int irq);`

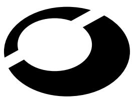
Attention, ne pas utiliser de fonctions qui relâchent le CPU (appel à `schedule()`) dans une section ininterruptible ;

La macro `restore_flags()` fait un appel automatique à `sti()` afin de réactiver les interruptions ;



Ex :

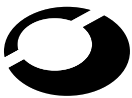
```
save_flags(old);

cli();

/* ...accès exclusif à la ressource partagée... */

restore_flags(old);
```



Sémaphores

Mécanisme universel de synchronisation et de protection des ressources partagées ;

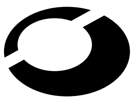
Basé sur un compteur et deux primitives :
- `P()` : "Puis-je accéder à la ressource ?" - décrémente le compteur et endort le processus si le compteur est inférieur à 0, et
- `V()` : "Vas-y !" - incrémente le compteur et débloque le premier processus bloqué sur la ressource.



Il existe des sémaphores pour différentes problématiques :
- exclusion mutuelle *(mutex),*
- cohabitation producteurs-consommateurs *(semaphore),*
- cohabitation lecteurs-écrivains *(rwsem).*

Pour la cohabitation producteurs-consommateurs, les consommateurs ne peuvent consommer que ce qui a été produit (sémaphores à compte) ;

Pour la cohabitation lecteurs-écrivains :
- plusieurs lecteurs peuvent lire la ressource simultanément,
- un seul écrivain peut la modifier.



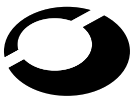
Efficaces mais coûteux en ressources et pas très rapides ;

Principalement utilisés pour gérer la concurrence dans le code noyau exécuté à la demande de processus utilisateurs (appels système) ;

Les fonctions `down()`/`up()` implémentent les concepts `P()` et `V()` ;

La fonction `down()` endort le processus appelant si la ressource n'est pas disponible ;

Les sémaphores sont donc à éviter dans un cas de concurrence avec un gestionnaire d'interruptions ;



Les variantes suivantes sont également utiles :

– `down_trylock()` : permet de ne pas endormir automatiquement le processus si la ressource n'est pas accessible (utilisable dans les gestionnaires d'interruptions),
– `down_interruptible()` : endort le processus si la ressource n'est pas accessible mais peut être réveillé par un signal (ne pas oublier de faire alors un `up()`).

Initialisation d'un *mutex* ou d'un *semaphore* (type struct semaphore) :
- à la déclaration => DECLARE_MUTEX{_LOCKED}() ou __DECLARE_SEMAPHORE_GENERIC(),
- durant l'exécution (*runtime*) => init_MUTEX{_LOCKED}() ou sema_init().

Prototypes (<{linux,asm}/semaphore.h>) :
- **DECLARE_MUTEX{_LOCKED}** (name);
- **__DECLARE_SEMAPHORE_GENERIC** (name, val);
- void **init_MUTEX{_LOCKED}** (struct semaphore *sem);
- void **sema_init** (struct semaphore *sem, int val);
- void **{down,up}** (struct semaphore *sem);
- int **down_*** (struct semaphore *sem);

Initialisation d'un *rwsem* (type struct rw_semaphore) :
- à la déclaration => DECLARE_RWSEM(),
- durant l'exécution (*runtime*) => init_rwsem().

Prototypes (<{linux,asm}/rwsem.h>) :
- **DECLARE_RWSEM** (name);
- void **init_rwsem** (struct rw_semaphore *sem);
- void **{down,up}_{read,write}** (struct rw_semaphore *sem);

Ex : exclusion mutuelle

```
static DECLARE_MUTEX(toto_sem);
down(& toto_sem);
```

/* ...accès exclusif à la ressource toto en lecture/écriture... */

```
up(& toto_sem);
```

Ex : lecteurs

```
static DECLARE_RW_SEM(titi_rwsem);
down_read(& titi_rwsem);
```

/* ...accès exclusif à la ressource titi en lecture uniquement... */

```
up_read(& titi_rwsem);
```

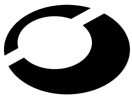

Verrous

Les sémaphores sont coûteux en matière de ressources ;

Les verrous *(locks)* présentent une bonne alternative pour les sections critiques intra-noyau ;

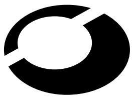
Basés sur des mécanismes d'attente active de la ressource :
- boucle `while()` sur architectures SMP, et
- non-préemption noyau sur architectures monoprocesseur.

Plus désactivation éventuelle des interruptions dans le cas de concurrence avec un gestionnaire d'interruptions ;

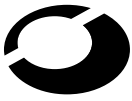


Il existe des verrous pour différentes problématiques :
- exclusion mutuelle *(spinlocks),*
- cohabitation lecteurs-écrivains *(rwlocks).*

Pour la cohabitation lecteurs-écrivains :
- plusieurs lecteurs peuvent lire la ressource simultanément,
- un seul écrivain peut la modifier.

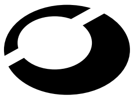


Les fonctions de manipulation des *locks* (types `spinlock_t` ou `rwlock_t`) sont les suivantes :
- `{read,write}_lock()` : fermeture d'un verrou par un lecteur/écrivain,
- `{read,write}_unlock()` : ouverture d'un verrou par un lecteur/écrivain,
- `spin_{lock,unlock}()` : ouverture/fermeture d'un verrou d'exclusion mutuelle,
- `spin_trylock()` : test de fermeture d'un verrou d'exclusion mutuelle.



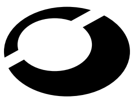
Peuvent être fatales si utilisées pour protéger une ressource accédée par un gestionnaire d'interruptions ;

=> utilisation des suffixes suivants :
– `_irq` : blocage/déblocage des interruptions avec les fonctions `*_lock()`/`*_unlock()`,
– `_irqsave` : blocage des interruptions et sauvegarde du contexte avec les fonctions `*_lock()`,
– `_irqrestore` : déblocage des interruptions et restauration du contexte avec les fonctions `*_unlock()`.



Initialisation d'un *spinlock* (type `spinlock_t`) :
– à la déclaration => constante `SPIN_LOCK_UNLOCKED`, ou
– durant l'exécution (*runtime*) => `spin_lock_init()`.

Initialisation d'un *rwlock* (type `rwlock_t`) :
– à la déclaration => constante `RW_LOCK_UNLOCKED`, ou
– durant l'exécution (*runtime*) => `rw_lock_init()` (2.4 uniquement).



Prototypes (`<asm/spinlock.h>`) :

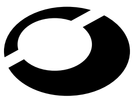
– `void` **{spin,rw}_lock_init** `({spin,rw}lock_t lock);`

– `void` **{spin,read,write}_{lock,unlock}{_irq}** `({spin,rw}lock_t lock);`

– `void` **{spin,read,write}_lock_irqsave** `({spin,rw}lock_t lock, unsigned long flags);`

– `void` **{spin,read,write}_unlock_irqrestore** `({spin,rw}lock_t lock, unsigned long flags);`

– `int` **spin_trylock** `(spinlock_t lock);`



Ex : lecteurs-écrivains

```
rwlock_t toto_rw_lock = RW_LOCK_UNLOCKED;

read_lock(& toto_rw_lock);
```

/* ...accès exclusif à la ressource toto en lecture uniquement... */

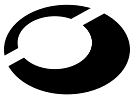
```
read_unlock(& toto_rw_lock);
```

[...]

```
write_lock(& toto_rw_lock);
```

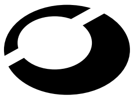
/* ...accès exclusif à la ressource toto en lecture/écriture... */

```
write_unlock(& toto_rw_lock);
```



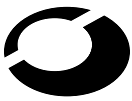
Ex : exclusion mutuelle

```
spinlock_t mon_spin_lock = SPIN_LOCK_UNLOCKED;
spin_lock(& mon_spin_lock);
/* ...accès exclusif à la ressource toto en lecture/écriture... */
spin_unlock(& mon_spin_lock);
```

Section 2

Reports d’exécution



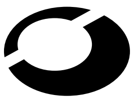
Délais

La variable globale `jiffies` représente le nombre de `ticks` d’horloge écoulés depuis le démarrage de la machine (`<linux/param.h>`) ;

`HZ` est le nombre de `ticks` d’horloge par seconde (100 sur plate-forme Intel) ;

Ex : attente coopérative 2 secondes

```
unsigned long jiffies_fin = jiffies + 2 * HZ;

while (jiffies < jiffies_fin) schedule();
```

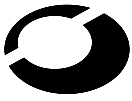



La fonction `schedule_timeout()` permet de réaliser une attente coopérative plus efficace (en ayant pris soin de changer le status du processus courant avec la macro `set_current_state()`) ;

Ex : attente coopérative 2 secondes

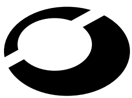
```
set_current_state(TASK_INTERRUPTIBLE);

schedule_timeout(2 * HZ);
```

Pour des délais plus courts et non coopératifs (ex : synchronisation avec le matériel) on préférera `udelay()` ou `mdelay()` ;



Prototypes (`<linux/sched.h>` et `<linux/delay.h>`) :

– `long` **schedule_timeout** `(long timeout);`
– `void` **set_current_state** `(int state);`
– `void` **udelay** `(unsigned long usecs);`
– `void` **mdelay** `(unsigned long msecs);`



Queues de tâches

Autre façon de reporter certaines exécutions ;

Permettent d’accumuler des tâches dans une queue puis de les lancer de manière séquentielle à un moment choisi ;

Initialisation d’une queue de tâches (type `task_queue`) à l’initialisation => `DECLARE_TASK_QUEUE()` ;

Initialisation d’une tâches (type `struct tq_struct`) au *runtime* (2.4 uniquement) :
– `INIT_TQUEUE()` : initialise et prépare une queue de tâches, ou
– `PREPARE_TQUEUE()` : prépare une queue de tâches.

On utilise ensuite les fonctions suivantes :
– `queue_task()` : ajoute une tâche à une queue, et
– `run_task_queue()` : consomme toutes les tâches d’une queue.

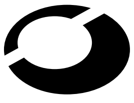
Les *bottom-halves* sont implémentées grâce à des queues de tâches ;

Prototypes (`<linux/tqueue.h>`) :
– **DECLARE_TASK_QUEUE** `(name);`
– `void` **{INIT,PREPARE}_TQUEUE** `(struct tq_struct *task, void (*routine)(void *), void *data);`
– `int` **queue_task** `(struct tq_struct *task, task_queue *queue);`
– `void` **run_task_queue** `(task_queue *queue);`

Timers noyau

Permettent d’exécuter une fonction donnée à une date (en `jiffies`) précise dans le futur ;

Routines de contrôle :
– `init_timer()` : créer un nouveau *timer,*
– `mod_timer()` : initialise (champ `expires`) et lance un *timer,*
– `add_timer()` : lance un *timer* (peu utilisé),
– `del_timer()` : destruction du *timer* avant son expiration.



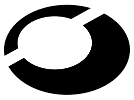
Prototypes (`<linux/timer.h>`) :

– `void` **init_timer** `(struct timer_list *timer);`

– `void` **mod_timer** `(struct timer_list *timer, unsigned long expires);`

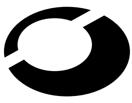
– `void` **add_timer** `(struct timer_list *timer);`

– `int` **del_timer** `(struct timer_list *timer);`

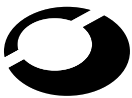


Ex : *timer* noyau à 5 secondes

```
static struct timer_list mon_timer;

init_timer(& mon_timer);

mon_timer.function = ma_fonction;

mon_timer.data = mon_argument;

mod_timer(& mon_timer, jiffies + 5 * HZ);
```



Interface `/proc`



Concepts

Système de fichiers virtuel ;

N'est pas associé à un périphérique ;

N'existe que le temps de vie du noyau ;

Possibilité de créer une sous-arborescence ;

Historiquement utilisé pour présenter des informations sur les processus à l'espace utilisateur ;

Concept étendu pour fournir/modifier toutes sortes d'informations/paramètres du noyau.

Implémentation

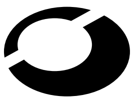
On définit une structure `proc_dir_entry` qui décrit les caractéristiques du fichier (ex : droits, nom, fonctions de lecture/écriture...) ;

On utilise alors les fonctions `proc_{register,unregister}()` pour créer/détruire une entrée dans `/proc` ;

Valable si la taille du fichier n'excède pas `PAGE_SIZE` ;

Sinon, déclaration de structures `file_operations` et `inode_operations` ;

Des fonctions automatisent certaines phases de la procédure :
- `{create,remove}_proc_entry()` : définit et crée/supprime une entrée dans `/proc`,
- `create_proc_{read,info}_entry()` : définit et crée une entrée en lecture seule dans `/proc` (2.4 uniquement),
- `proc_mkdir()` : crée un sous-répertoire dans `/proc`,
- `proc_symlink()` : crée un lien logique dans `/proc` (2.4 uniquement),
- `proc_mknod()` : crée un fichier spécial d'accès à un périphérique dans `/proc` (2.4 uniquement).

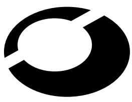


Prototypes (`<linux/proc_fs.h>`) :

– `struct proc_dir_entry *` **create_proc_entry** `(const char *name, mode_t mode, struct proc_dir_entry *parent);`

– `struct proc_dir_entry *` **create_proc_read_entry** `(const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void * data);`

– `struct proc_dir_entry *` **create_proc_info_entry** `(const char *name, mode_t mode, struct proc_dir_entry *base, get_info_t *get_info);`

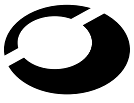


– `void` **remove_proc_entry** `(const char *name, struct proc_dir_entry *parent);`

– `struct proc_dir_entry *` **proc_mkdir** `(const char *name, struct proc_dir_entry *parent);`

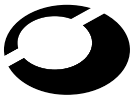
– `struct proc_dir_entry *` **proc_symlink** `(const char *name, struct proc_dir_entry *parent, const char *dest);`

– `struct proc_dir_entry *` **proc_mknod** `(const char *name,r mode_t mode, struct proc_dir_entry *parent, kdev_t rdev);`

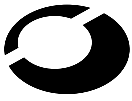


Ex : fichier /proc/helloworld

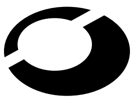
```
int hello_init(void) {...
struct proc_dir_entry *proc_entry;
proc_entry = create_proc_entry("helloworld", 0,
NULL);
if (proc_entry) proc_entry->get_info =
hello_get_info;
...}
```


```
int hello_get_info(char *page, char **start, off_t
off, int count, int eof) {
return sprintf(page, "helloworld\n");
}
void hello_exit(void) {...
remove_proc_entry("helloworld", NULL);
...}
```



Système de fichiers `devfs`



Concepts

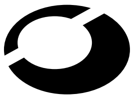
Système de fichiers virtuel (pas associé à un périphérique, n’existe que le temps de vie du noyau...) ;

Les fichiers spéciaux associés à un pilote de périphériques sont créés dynamiquement par celui-ci lors de son enregistrement ;

Représentation hiérarchique des périphériques => simplification ;

Compatibilité ascendante assurée par un *daemon* utilisateur (`devfsd`) ;

Noyaux 2.4 uniquement (`Documentation/filesystems/devfs/`).

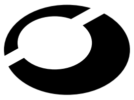


Implémentation

On préfixe les fonctions `{register,unregister}_{chr,blk}dev()` par `devfs_` pour enregister/désenregister le pilote de périphériques ;

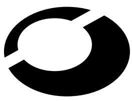
Les fonctions suivantes permettent de manipuler le système de fichiers virtuel :

– `devfs_{register,unregister}_{chr,blk}dev()` : enregistre/désenregistre un pilote de périphériques de type caractère/bloc auprès du noyau,
– `devfs_{register,unregister}()` : crée/supprime une nouvelle entrée dans `devfs`,
– `devfs_mk_symlink()` : crée un lien logique dans `devfs`,
– `devfs_mk_dir()` : crée un répertoire dans `devfs`.

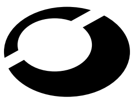


Prototypes (`<linux/devfs_fs_kernel.h>`) :

– `int` **devfs_register_{chr,blk}dev** `(unsigned int major, const char *name, struct file_operations *fops);`
– `int` **devfs_unregister_{chr,blk}dev** `(unsigned int major, const char *name);`
– `devfs_handle_t` **devfs_register** `(devfs_handle_t dir, const char *name, unsigned int flags, unsigned int major, unsigned int minor, umode_t mode, void *ops, void *info);`



- void **devfs_unregister** (devfs_handle_t de);
- devfs_handle_t **devfs_mk_dir** (devfs_handle_t dir, const char *name, void *info);
- int **devfs_mk_symlink** (devfs_handle_t dir, const char *name, unsigned int flags, const char *link, devfs_handle_t *handle, void *info);



Ex : enregistrement d'un pilote caractère

```
static devfs_handle_t devfs_handle;

int toto_init(void) {

devfs_register_chrdev(MAJOR_NR, "toto",

& toto_fops);

devfs_handle = devfs_register(NULL, "toto_7",
DEVFS_FL_DEFAULT, MAJOR_NR, 7, S_IFCHR | S_IRUGO |
S_IWUSR, & toto_fops, NULL);

}
```

Threads noyau

Description

Processus légers du noyau ;

Utilisés pour implémenter les *daemons* noyau :
- chargés de réaliser des tâches récurrentes mais néanmoins importantes au sein du noyau (ex : *swap* , vidange des caches...),
- lancés au démarrage par le processus 0.

N'interagissent pas avec l'espace utilisateur ;

Utilisés par certains pilotes de périphériques (ex : gestion du journal de ReiserFS).

Implémentation

Un *thread* noyau hérite d'une copie des ressources de son père à sa création ;

Ressources inutiles dans le cas d'un *daemon* noyau => libération des ressources et "émancipation" ;

Le processus qui tue le *thread* doit attendre la fin effective de celui-ci avant de se terminer lui-même => synchronisation (semaphore ou variable de complétion pour les noyaux récents) ;

Fonctions de gestion des *threads* noyau :
– `kernel_thread()` : crée un *thread* noyau,
– `daemonize()` : détache le *thread* courant de son père (rattaché à *init*) et libère les ressources inutiles,
– `kill_proc()` : envoie un signal à un processus depuis le noyau (utilisé pour tuer un *thread* noyau).

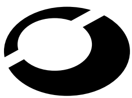
Prototypes (`<linux/sched.h>`)
– `int` **kernel_thread** `(int (*kth)(void *), void *arg, unsigned long flags);`
– `void` **daemonize** `(void);`
– `int` **kill_proc** `(pid_t pid, int sig, int priv);`

Initialisation d’une variable de complétion (type `struct completion`) :

- à la déclaration => `DECLARE_COMPLETION()`, ou
- durant l’exécution (*runtime*) => `init_completion()`.

Fonctions de complétion des *threads* noyau :

- `wait_for_completion()` : attend la completion effective du *thread* ,
- `complete()` : complète le *thread* ,
- `complete_and_exit()` : complète et termine le *thread* .

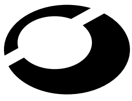


Threads **noyau**

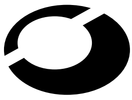
Alcôve - Noyau Linux et pilotes de périphériques

Prototypes (`<linux/completion.h>`)

- **DECLARE_COMPLETION** `(name);`
- `void` **init_completion** `(struct completion *comp);`
- `void` **wait_for_completion** `(struct completion *comp);`
- `void` **complete** `(struct completion *comp);`
- `void` **complete_and_exit** `(struct completion *comp, long code);`



Divers



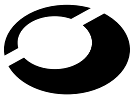
Types de données

Types de données standards (ex : `char`, `int`, `long`...) ;

Peuvent être de tailles différentes selon les architectures ;

Cependant, on peut avoir besoin de variables de tailles fixes (ex : *mappage* d'une structure correspondant à l'ensemble des registres d'un périphérique) ;

Utilisation de types spéciaux (`<asm/types.h>`) :
– `u{8,16,32,64}` : types 8, 16, 32 ou 64 bits non signés,
– `s{8,16,32,64}` : types 8, 16, 32 ou 64 bits signés.

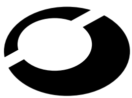


Conversions

Deux formats de disposition des octets dans les mots :

– "Gros-boutiens" *(Big-endian)* : les octets de poids fort sont placés avant les octets de poids faible (ex : Motorola 68k),
– "Petit-boutiens" *(Little-endian)* : les octets de poids faible sont placés avant les octets de poids fort (ex : Intel x86).

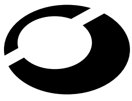
Certains périphériques peuvent utiliser un format différent de celui de la machine pour la représentation de leurs données internes ;



Des macros servent à optimiser les conversions entre différents formats (`<asm/byteorder.h>`) :

– `cpu_to_{le,be}{16,32,64}()` : conversion d'une donnée au format local utilisé par le CPU vers un format déterminé,
– `{le,be}{16,32,64}_to_cpu()` : conversion d'une donnée dans un format déterminé vers le format local utilisé par le CPU.

Ces mêmes fonctions convertissent la donnée *in situ* quand elles sont suffixées pas un `s` et la donnée pointée quand elles sont suffixées par un `p`.



Listes doublement chaînées

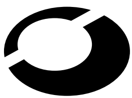
Très utilisées dans le noyau ;

Mécanisme d'encapsulation pour créer ses propres listes ;

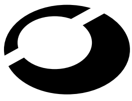
Simplifie les manipulations (parcours, ajout/retrait d'un élément...) ;

Ajout d'un champ de type `struct list_head` dans la structure des éléments à chaîner ;

Initialisation d'une tête de liste (type `struct list_head`) :
– à la déclaration => `LIST_HEAD()` et `LIST_HEAD_INIT()`, ou
– durant l'exécution (*runtime*) => `INIT_LIST_HEAD()`.

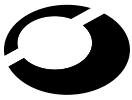


Fonctions de manipulation de la liste et des éléments :
– `list_add{_tail}()` : ajoute un élément en tête/queue d'une liste,
– `list_del()` : supprime un élément d'une liste,
– `list_entry()` : donne accès à un élément d'une liste,
– `list_for_each()` : parcours une liste élément par élément,
– `list_splice()` : joint deux listes en une seule.

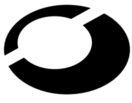


Prototypes (`<linux/list.h>`) :

- **LIST_HEAD{_INIT}** `(name);`
- `void` **INIT_LIST_HEAD** `(struct list_head *head);`
- `void` **list_del** `(struct list_head *entry);`
- `void` **list_add{_tail}** `(struct list_head *new, struct list_head *head);`
- `void` **list_del** `(struct list_head *entry);`
- `type *` **list_entry** `(struct list_head *entry, type, member);`
- `void` **list_for_each** `(struct list_head *current, struct list_head *head);`
- `void` **list_splice** `(struct list_head *list, struct list_head *head);`



Travaux pratiques



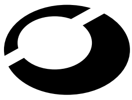
Interface `/proc` minimaliste dans le module de test ;

Liste des processus en cours dans `/proc` ;

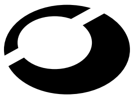
Utilisation d'un *thread* noyau pour mettre à jour la liste des processus en cours ;

Mécanisme de synchronisation lecture/écriture (sémaphore) dans le pilote du périphérique *buffer* ;

Mécanisme d'expiration *(timer)* pour le mode bloquant.



Références



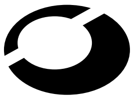
Objectifs du chapitre

Livres ;

Livres électroniques ;

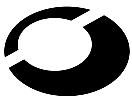
Articles et documents ;

Glossaire.



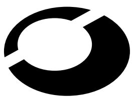
Livres

– Linux Device Drivers (O'Reilly),
– Understanding the Linux Kernel (O'Reilly),
– Programmation Linux 2.0 (Eyrolles).



Livres électroniques

– Linux Kernel Internals (noyaux 2.4)
  [http ://www.linuxdoc.org/LDP/lki/],
– Linux Kernel Module Programming Guide (noyaux 2.0 et 2.2)
  [http ://www.linuxdoc.org/LDP/lkmpg/],
– Linux Kernel Hackers' Guide (noyaux 2.0 et 2.2)
  [http ://www.linuxdoc.org/LDP/khg/],
– The Linux Kernel (noyaux 2.0)
  [http ://www.linuxdoc.org/LDP/tlk/],
– The Linux Programmer's Guide (API espace utilisateur)
  [http ://www.linuxdoc.org/LDP/lpg/].



Articles et documents

– Le projet de documentation du noyau
  (Documentation/DocBook/)
  [http ://kernelbook.sourceforge.net],
– Les liens de *kernelnewbies*
  [http ://www.kernelnewbies.org/links/],
– Recueil de liens sur le noyau
  [http ://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html],
– Architecture conceptuelle du noyau Linux
  [http ://plg.uwaterloo.ca/~itbowman/CS746G/a1/],
– Architecture concrète du noyau Linux
  [http ://plg.uwaterloo.ca/~itbowman/CS746G/a2/].

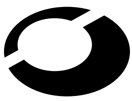
Glossaire

- **API** , *Application Programming Interface* : interface de programmation,
- **AVL** , *Adelson-Velskii and Landis* : organisation en arbre binaire toujours équilibré permettant des recherches en O(log n) où une recherche traditionnelle serait en O(n),
- **BE** , *Big-Endian* : format de disposition de mots binaires dans lequel les octets de poids fort sont placés avant les octets de poids faible,
- **DMA** , *Direct Memory Access* : mécanisme évitant d'avoir à utiliser le CPU pour gérer des transferts de données entre mémoire centrale et périphériques,

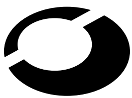
Références

Alcôve - Noyau Linux et pilotes de périphériques

- **FAQ** , *Frequently Asked Questions* : Foire Aux Questions en français, c'est le document à lire avant de poser une question sur une liste de diffusion,
- **FSF** , *Free Software Fundation* : organisation d'intérêt public dont le principale objectif est de promouvoir et défendre les logiciels libres (licences GPL et LGPL),
- **GPL** , *General Public Licence* : licence libre de la *Free Software Fundation,*
- **IP** , *Internet Protocol* : protocole de réseau (˜ couche 3 du modèle OSI) basé sur le principe du *best effort,*
- **IRQ** , *Interrupt ReQuest* : demande d'interruption matérielle,



– **ISDN** , *Integrated Services Digital Network* : réseau numérique permettant le transport de la voix et des données,
– **LE** , *Little-Endian* : format de disposition de mots binaires dans lequel les octets de poids faible sont placés avant les octets de poids fort,
– **LGPL** , *Lesser General Public Licence* : licence libre de la *Free Software Fundation* surtout utilisée pour les librairies,
– **OSI** , *Open System Interconnect* : modèle en couches fournissant un cadre conceptuel et normatif aux échanges entre systèmes hétérogènes,



– **POSIX** , *Portable Operating System for Computer Environment* : Norme Unix de l'IEEE (numérotée 1003.1) spécifiant le noyau du système ; elle comporte des extensions pour les noyaux temps réel (1003.1-b) et les processus légers (1003.1-c),
– **PLIP** , *Parallel Line Internet Protocol* : protocole de communication sur port parallèle utilisé pour encapsuler IP dans des communications par ligne parallèle,
– **PPP** , *Point to Point Protocol* : protocole de communication point à point utilisé pour encapsuler IP dans des communications par ligne série (ex : modem),
– **SLIP** , *Serial Line Internet Protocol* : protocole de communication point à point utilisé (comme PPP) pour encapsuler IP dans des communications par ligne série (ex : modem),

– **SMP** , *Symmetric MultiProcessing* : utilisation de plusieurs processeurs identiques partageant les mêmes ressources matérielles (mémoire...) et pilotés par un seul système d'exploitation => transparent pour l'utilisateur,
– **TCP** , *Transmission Control Protocol* : protocole de transport (˜ couche 4 du modèle OSI) fonctionnant en mode connecté,
– **VFS** , *Virtual File System* : couche d'abstraction haut niveau du noyau permettant l'écriture rapide de systèmes de fichiers.