

# Le noyau Linux et ses modules

Christophe BLAESS

christophe.blaess@logilin.fr  
<https://www.blaess.fr/christophe/>  
<https://www.linkedin.com/in/christophe-blaess/>

*« Sous Linux on a le noyau  
Sous Windows on a les pépins... »*  
Michael Opdenacker



Ingénierie et formations sur Linux et les logiciels libres  
<https://www.logilin.fr>

<b>Le noyau Linux et ses modules.....</b>	<b>3</b>
Introduction.....	3
Évolutions du noyau Linux.....	4
Principes des modules.....	5
Travaux pratiques : manipulation des modules précompilés.....	6
<b>Développement en mode noyau.....</b>	<b>7</b>
Outils.....	7
Compilation du noyau.....	8
Programmation en mode noyau.....	9
<b>Écriture d'un module.....</b>	<b>10</b>
Structure générale.....	10
Fichiers d'en-tête et macros.....	12
Licence GPL.....	13
Messages du noyau.....	14
Extensions de printk().....	15
Travaux pratiques : écriture, compilation et utilisation d'un module.....	16
Dépendances entre modules.....	17
Makefile typique pour la compilation de modules.....	18

Ce support de formation est distribué sous licence **Creative Commons 4.0**



*(Attribution - Partage dans les mêmes conditions).*

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* et *Excalidraw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

Écriture de drivers et programmation noyau Linux

v. 9.12

# Le noyau Linux et ses modules

## Introduction

GNU/Linux est un système d'exploitation compatible Unix, libre et accompagné de logiciels libres, développé essentiellement par des bénévoles.



Influences :

- Unix AT&T, BSD, etc.
- Projet GNU (*Gnu's Not Unix*) – Richard Stallman, FSF (*Free Software Foundation*), 1984
- Noyau Linux – Linus Torvalds, 1991.

Le noyau Linux permet de disposer d'un système :

- multi-tâches : ordonnancement temps-partagé préemptif, *soft realtime* ;
- multi-utilisateurs : modèle Unix - utilisateur normal ou administrateur *root* ;
- multi-processeurs, multi-cœurs, *hyper-threading*.
- multi-architectures : Intel x86, ARM, RiscV, PPC, M68k, Mips, etc.

Source logo « Gnu/Linux » : <https://commons.wikimedia.org/wiki/File:Gnulinux.png>

Manchot « Tux » créé par Larry Ewing, mascotte « GNU Head » créée par Aurelio A. Eckert.

## Voir aussi

Articles :

- [BLAESS 2003] : Christophe Blaess – *Linux, histoire d'un noyau* - Linux Magazine France, Hors-Série 16 (Septembre-Octobre 2003). <http://www.blaess.fr/christophe/articles/>

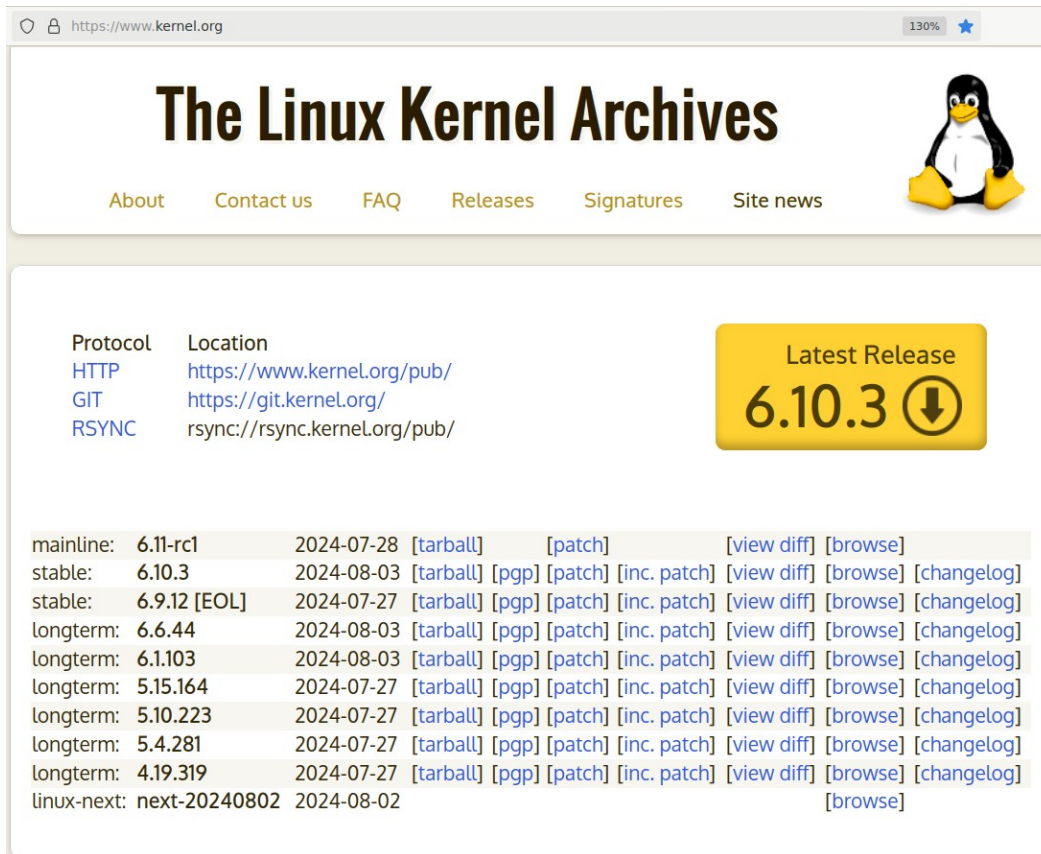
Web :

- <http://www.gnu.org/> : Le Projet GNU

Livres :

- [TANENBAUM 2006] : Andrew Tanenbaum & Albert Woodhull – *Operating Systems, Design and Implementation* – Third edition – Pearson, 2006.
- [RAYMOND 2003] : Eric S. Raymond – *The Art of Unix Programming* – Addison-Wesley, 2003.
- [TORVALDS 2002] : Linus Torvalds & David Diamond – *Just for Fun : The Story of an Accidental Revolutionary* – Harper Business, 2002

## Évolutions du noyau Linux



The screenshot shows the 'The Linux Kernel Archives' website. At the top, there's a navigation bar with links: About, Contact us, FAQ, Releases, Signatures, and Site news. A penguin logo is on the right. Below the navigation bar, there's a section for 'Latest Release' with a large yellow button showing '6.10.3' and a download icon. To the left of this, there's a table with columns 'Protocol' and 'Location'. Below that, there's a table listing various kernel versions and their corresponding links for downloading, patching, and viewing diffs.

Protocol	Location
HTTP	<a href="https://www.kernel.org/pub/">https://www.kernel.org/pub/</a>
GIT	<a href="https://git.kernel.org/">https://git.kernel.org/</a>
RSYNC	<a href="rsync://rsync.kernel.org/pub/">rsync://rsync.kernel.org/pub/</a>

Version	Date	[tarball]	[patch]	[view diff]	[browse]	[changelog]
mainline: 6.11-rc1	2024-07-28					
stable: 6.10.3	2024-08-03					
stable: 6.9.12 [EOL]	2024-07-27					
longterm: 6.6.44	2024-08-03					
longterm: 6.1.103	2024-08-03					
longterm: 5.15.164	2024-07-27					
longterm: 5.10.223	2024-07-27					
longterm: 5.4.281	2024-07-27					
longterm: 4.19.319	2024-07-27					
linux-next: next-20240802	2024-08-02					

La liste de discussion pour les développeurs du noyau est gérée par un robot Majordomo sur (<http://vger.kernel.org/>). Les sources du noyau sont archivées sur *Kernel.Org* (<http://www.kernel.org/>). Il existe plusieurs versions simultanées du noyau :

- des versions *stables*, par exemple 6.9.12, 6.10.3
- une version *release candidate*, par exemple 6.11-rc1

Les versions de développement sont accessibles en utilisant le système git. Le dépôt `git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git` contient toutes les évolutions depuis la 2.6.12

L'interface de programmation **externe** du noyau (API des appels-système) reste toujours stable. Une application n'a pas besoin d'être modifiée, ni même recompilée entre deux versions du noyau.

L'interface de programmation **interne** du noyau (utilisée par les drivers) peut changer entre deux versions successives du noyau.

### Voir aussi

- *Linux Weekly News* (<http://lwn.net/kernel/>)
- *The Linux Documentation Project* (<http://www.tldp.org/>)

## Principes des modules

Le noyau Linux est monolithique avec possibilité de charger dynamiquement des modules supplémentaires.

**Module** : fichier objet (extension **.ko** – *kernel object*) que l'on peut insérer dans le noyau en cours de fonctionnement.

Commandes d'administration essentielles :

`/sbin/lsmod` : affiche la liste des modules chargés, leurs dépendances et leurs tailles ;

`/sbin/modinfo module.ko` : affiche des informations sur le module.

`/sbin/insmod module.ko [param=valeur...]` : charge le module indiqué.

`/sbin/rmmod module` : décharge le module.

`/sbin/depmod` : examine les dépendances entre les modules du système.

`/sbin/modprobe module` : charge un module et ceux dont il dépend.

Le support des modules est une option que l'on active ou non à la compilation du noyau. La possibilité de décharger un module est une option dépendant de la compilation du noyau et du module lui-même.

### Voir aussi

**Pages de manuel :**

- `lsmod(8)`
- `modinfo(8)`
- `insmod(8)`
- `rmmod(8)`
- `depmod(8)`
- `modprobe(8)`

## Travaux pratiques : manipulation des modules précompilés.

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice I-1 : liste des modules chargés par le noyau.

Exercice I-2 : modules précompilés par la distribution.

Exercice I-3 : chargement d'un module précompilé.

Exercice I-4 : déchargement d'un module précompilé.

# Développement en mode noyau

## Outils

Le développement du noyau Linux et de ses modules se fait avec le compilateur *Gnu GCC*.

Les lignes de commande de GCC pour le noyau étant complexes, on fait appel à un *Makefile*.

Pour choisir proposer les nombreuses options de compilation, le noyau s'appuie sur un système nommé *Kconfig*.

Il existe peu d'outils de débogage du noyau. Les développeurs Linux utilisent surtout le système des traces du noyau.

Il existe une option de compilation nommée *KGDB* venant insérer des points de débogage dans un noyau. Un deuxième poste connecté par liaison série ou réseau permet d'utiliser le débogueur *GDB* classique pour basculer le noyau en pas-à-pas.

A titre indicatif, le code source du noyau Linux 5.12.9 contient :

- 1,2 Go de code source (dont 709 Mo pour le sous-système `drivers/`)
- 72 945 fichiers (dont 29 155 dans `drivers/`)
- 29 751 fichiers de code source C (dont 17 668 dans `drivers/`)
- 23 090 fichiers d'en-tête C ( `.h`)
- 1 266 fichiers assembleurs ( `.S`)
- 2 607 fichiers de compilation *Makefile*
- 7 630 fichiers de documentation
- 27.603.676 lignes de code source
- 18.781 options de compilation

## Compilation du noyau

La compilation du noyau Linux **n'est pas une opération indispensable** pour l'utilisation courante. C'est surtout une étape nécessaire pour l'ajuster exactement à un matériel précis et un environnement restreint.

- Charger une archive "officielle" du noyau Linux :  
`wget https://www.kernel.org/pub/linux/kernel/v5.x/linux-5.18.1.tar.xz`
- Décompresser l'archive :  
`tar xJf linux-5.18.1.tar.xz`  
`cd linux-5.18.1`
- Configurer le noyau selon les options voulues :  
`make x86_64_defconfig`  
`make menuconfig`
- Lancer la compilation du noyau et des modules :  
`make -j 8`
- Installer le noyau et redémarrer le système :  
`make modules_install`  
`make install`

La compilation proprement dite peut se faire sous n'importe quelle identité, (et dans un répertoire personnel). Seule l'installation finale du noyau nécessite les droits *root*.

Pour compiler un noyau pour un système embarqué on procède par *cross-compilation* : on compile le noyau sur un PC pour le charger ensuite sur le système cible.

Voir par exemple l'article « *Création d'un système complet avec Buildroot* » sur <https://www.blaess.fr/christophe/articles/creer-un-systeme-complet-avec-buildroot/>

ou

mon cours en ligne « Linux embarqué avec Yocto Project »  
(<https://www.blaess.fr/christophe/yocto-lab>)

## Programmation en mode noyau

La programmation en mode noyau impose des règles assez strictes.

Ne pas inclure les fichiers d'en-tête usuels du système, mais ceux du noyau. Accessible par `<linux/...>` ou `<asm/...>`.

Le noyau n'a pas accès aux fonctions de la bibliothèque C. Une bibliothèque minimale implémente les fonctionnalités de base (`strcpy()`, `strlen()`, `sscanf()`, `sprintf()`...)

Pas de nombres réels. Utiliser seulement les types entiers.

Protéger les accès aux données partagées manipulées dans les gestionnaires d'interruption ou sur les systèmes multicœurs.

Être paranoïaque !

# Écriture d'un module

## Structure générale

À l'insertion d'un module, exécution automatique de la routine.

```
int init_module(void);
```

Cette routine doit renvoyer zéro si le chargement se passe bien, et un code d'erreur négatif sinon (voir <asm/errno.h>).

```
int __init init_module (void)
{
    buffer = vmalloc(BUFFER_LENGTH);
    if (buffer == NULL)
        return -ENOMEM;
    return 0;
}
```

À la suppression du module, exécution de :

```
void cleanup_module(void);
```

Si `cleanup_module()` n'est pas défini, on ne pourra pas décharger le module.

Pour pouvoir utiliser le même code en module et intégré directement dans le noyau, on emploie plutôt :

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

...

static int __init my_driver_init (void)
{
    ...
    return 0;
}

static void __exit my_driver_exit (void)
{
    ...
}

...

module_init(my_driver_init);
module_exit(my_driver_exit);
```

Les directives `__init` et `__exit` sont destinées au compilateur pour savoir à quel moment le code est utilisé.

## Fichiers d'en-tête et macros

Pour compiler correctement un module, il faut inclure de préférence :

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

Il est possible de fournir les informations suivantes :

```
MODULE_AUTHOR("nom de l'auteur <adresse@mail.org>");
MODULE_DESCRIPTION("courte description pour modinfo");
MODULE_VERSION("1.4-test8");
MODULE_LICENSE("licence");
```

Type de licences :

- "GPL "
- "GPL v2"
- "GPL and additional rights"
- "Dual BSD/GPL "
- "Dual MPL/GPL "
- "Proprietary"

Un module peut indiquer un ou plusieurs alias qui permettront également de le charger avec :

```
MODULE_ALIAS("alias");
```

On peut également indiquer des alias dans `/etc/modprobe.conf`.

## Licence GPL

Le noyau Linux est placé sous licence GPL v.2 (*General Public License*).

Les implications sont les suivantes :

- le logiciel est **libre** suivant les critères de la FSF ;
- la clause du *copyleft* couvre la **redistribution** du logiciel.

Un logiciel est **libre** si son utilisateur dispose des quatre droits :

- exécution et utilisation du logiciel ;
- analyse et adaptation du logiciel à ses propres besoins ;
- redistribution de copies non modifiées du logiciel ;
- redistribution de versions modifiées du logiciel.

Le *copyleft* impose que la redistribution du logiciel s'accompagne des mêmes droits que ceux concédés initialement

D'autres licences libres : LGPL (*Lesser General Public License*), MPL (*Mozilla Public License*), BSD (*Berkeley Software Distribution*), etc.

Les développeurs du noyau Linux insistent sur le fait que **la GPL ne doit pas être perçue comme une contrainte** (ce qui est trop souvent le cas dans le monde industriel) mais comme une garantie de pérennité et de maintenance du code et de ses dérivés.

L'intégration *sans modification* de code sous licence MPL dans un produit propriétaire est possible.

La licence BSD ne contient pas la clause du *copyleft*. Le code sous licence BSD peut être intégré et modifié dans du code propriétaire.

### Voir aussi

**Web** : les licences libres

- <http://opensource.org/licenses/> : Point de vue de l'OSI (*Open Source Initiative*)
- <http://www.gnu.org/licenses/> : Point de vue de la FSF

## Messages du noyau

```
<linux/kernel.h>
```

```
int printk(const char * format ...)
```

`printk()` renvoie le nombre de caractères écrits dans le journal système. Le message est envoyé à `syslogd`, on le voit dans `/var/log/messages` ou avec la commande `dmesg`.

Le format commence par un indicateur de gravité du message :

- `KERN_DEBUG` : message de débogage,
- `KERN_INFO` : information en fonctionnement normal,
- `KERN_NOTICE` : information normale mais significative,
- `KERN_WARNING` : avertissement,
- `KERN_ERR` : condition d'erreur système,
- `KERN_CRIT` : erreur critique pour le système,
- `KERN_ALERT` : réponse immédiate nécessaire,
- `KERN_EMERG` : système inutilisable.

```
|| printk(KERN_INFO "Counter = %d\n", cnt);
```

La commande « `dmesg -L always` » affiche les messages du noyau en les colorant en fonction du niveau de gravité.

La commande « `dmesg -c` » affiche les messages du buffer du noyau puis les efface définitivement.

## Extensions de printk()

On préfère généralement les fonctions suivantes au simple `printk()` :

```
int pr_debug(const char *format, ...);  
int pr_info(const char *format, ...);  
int pr_notice(const char *format, ...);  
int pr_warn(const char *format, ...);  
int pr_err(const char *format, ...);  
int pr_crit(const char *format, ...);  
int pr_alert(const char *format, ...);  
int pr_emerg(const char *format, ...);
```

`pr_debug()` fonctionne si la constante symbolique `DEBUG` est définie en début de fichier.

Il existe des variantes pour un seul affichage même si le message se répète :

```
int pr_info_once(const char *format, ...);  
int pr_notice_once(const char *format, ...);
```

Pour continuer un message (non terminé par un « `\n` ») on peut utiliser :

```
int pr_cont(const char *format, ...);
```

## Travaux pratiques : écriture, compilation et utilisation d'un module

Énoncés et solutions sur <https://www.logilin.fr/tp/ild>.

Exercice I-5 : écriture, compilation et test d'un module complet.

## Dépendances entre modules

Seuls les symboles exportés explicitement par un module seront accessibles dans un autre module. Les macros suivantes permettent d'exporter un symbole :

- `EXPORT_SYMBOL(my_symbol)` : le symbole est utilisable dans tout module ;
- `EXPORT_SYMBOL_GPL(my_symbol)` : le symbole est utilisable dans les modules sous licence GPL ou compatible.

On peut rencontrer également - mais c'est plus rare - `EXPORT_PER_CPU_SYMBOL()`, et `EXPORT_PER_CPU_SYMBOL_GPL()`.

À titre indicatif, dans la branche Linux 5.12, les exportations de symboles se répartissent en :

- 16 257 `EXPORT_SYMBOL()`
- 16 977 `EXPORT_SYMBOL_GPL()`

Les dépendances entre modules sont gérées automatiquement par le noyau. Il est impossible de décharger un module si un autre utilise ses services.

### À vous...

Observez et testez le chargement de deux modules (`exemple-I-02` et `exemple-I-03`) qui contiennent des dépendances.

Copiez les deux exemples dans un sous répertoire (créé pour l'occasion) de

```
/lib/modules/$(uname -r)/
```

puis lancez la commande `depmod`.

Essayez de charger `exemple-I-03.ko` avec la commande « `modprobe exemple-I-03` ».

Le fichier `exemple-I-02.ko` est-il chargé automatiquement ? Grâce à quel mécanisme ?

Retirez `exemple-I-03` du noyau avec « `modprobe -r exemple-I-03` ». Le fichier `exemple-I-02` est-il également retiré ?

Pensez à effacer le sous-répertoire et à relancer « `depmod` » avant de passer à la suite.

## Makefile typique pour la compilation de modules

```
Makefile:

ifneq ($(KERNELRELEASE),)

obj-m := my_driver.o

else

KERNEL_DIR ?= /lib/modules/$(shell uname -r)/build
MODULE_DIR := $(shell pwd)

modules:
    $(MAKE) -C $(KERNEL_DIR) M=$(MODULE_DIR) modules

clean:
    rm -f *.o *.ko *.mod.c

endif
```

On compile ainsi le module pour le noyau actuellement en cours d'exécution. Sinon il faut indiquer l'emplacement des sources du noyau cible dans `KERNEL_DIR`.

Pour une cross-compilation, le fichier Makefile devra également configurer `ARCH` et `CROSS_COMPILE`.

Il faut avoir déjà exécuté `make menuconfig` pour le noyau cible.

### Voir aussi

- Fichier `Documentation/kbuild/makefiles.rst` des sources du noyau

La chaîne de compilation du noyau sait qu'elle doit compiler sous forme de module tous les fichiers objets mentionnés dans la variables `obj-m`.

Si un module est construit à partir de plusieurs fichiers sources, on écrit :

```
obj-m      += my_driver.o
my_driver-y := my_file_1.o my_file_2.o my_file_3.o
```

Le « `m` » dans « `obj-m` » indique que le code doit être compilé comme module externe. On pourrait utiliser « `obj-y` » pour inclure le code statiquement dans un noyau que l'on recompilait.

Si un module utilise des fonctionnalités du noyau qui dépendent de sa version, on utilisera une compilation conditionnelle.

```
06-driver-reseau/example-VI-01.c:
```

```
[...]
#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 15, 0)
    eth_hw_addr_set(net_dev, hw_address);
#elif LINUX_VERSION_CODE >= KERNEL_VERSION(3, 14, 0)
    ether_addr_copy(net_dev->dev_addr, hw_address);
#else
    memcpy(net_dev->dev_addr, hw_address, 6);
#endif
[...]
```

La macro `KERNEL_VERSION` fournit un entier long par décalage binaire de ses arguments :

`KERNEL_VERSION(5, 15, 0)` vaut `0x050F00`.

Ceci concerne les drivers développés hors du noyau. Ceux distribués avec le noyau sont mis à jour directement s'il y a une modification de l'API interne.