# NMV : Programmer dans le noyau

**Version 15.02**

Julien Sopena[1]

[1]julien.sopena@lip6.fr
Équipe RÉGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Master SAR 2ème année - NMV - 2016/2017



## Grandes lignes du cours





## Outline





## Les fonctions "inline"



## Les fonctions "inline"

Le mot-clé **inline** permet, au compilateur, de remplacer un appel de fonction par le code de cette fonction.



## Les fonctions "inline"

Le mot-clé **inline** permet, au compilateur, de remplacer un appel de fonction par le code de cette fonction.

**Avantage** $\Longrightarrow$
- permet d'économiser le coût d'un appel de fonction
- permet des optimisations impossibles avec un appel

**Inconvénient** $\Longrightarrow$
- augmente la taille du code, donc les *cache miss*
- utilisation accrue des registres pour les paramètres



## Les fonctions "inline"

Le mot-clé **inline** permet, au compilateur, de remplacer un appel de fonction par le code de cette fonction.

**Avantage** $\Longrightarrow$
- permet d'économiser le coût d'un appel de fonction
- permet des optimisations impossibles avec un appel

**Inconvénient** $\Longrightarrow$
- augmente la taille du code, donc les *cache miss*
- utilisation accrue des registres pour les paramètres

```
inline int max(int a, int b) {
  return (a > b) ? a : b ;
}
```

```
int f(int y) {
  return max(y,2*y) ;
}
```



## Les fonctions "inline"

Le mot-clé **inline** permet, au compilateur, de remplacer un appel de fonction par le code de cette fonction.

**Avantage** $\Longrightarrow$
- permet d'économiser le coût d'un appel de fonction
- permet des optimisations impossibles avec un appel

**Inconvénient** $\Longrightarrow$
- augmente la taille du code, donc les *cache miss*
- utilisation accrue des registres pour les paramètres

```
inline int max(int a, int b) {
  return (a > b) ? a : b ;
}
```

```
int f(int y) {
  return max(y,2*y) ;
}
```

```
int f(int y) {
  return (y>2*y)?y:2*y;
}
```

## Les fonctions "inline"

Le mot-clé **inline** permet, au compilateur, de remplacer un appel de fonction par le code de cette fonction.

**Avantage** $\Longrightarrow$ { permet d'économiser le coût d'un appel de fonction ; permet des optimisations impossibles avec un appel }

**Inconvénient** $\Longrightarrow$ { augmente la taille du code, donc les *cache miss* ; utilisation accrue des registres pour les paramètres }

```
inline int max(int a, int b) {
  return (a > b) ? a : b ;
}
```

```
int f(int y) {
  return max(y,2*y) ;
}
```

```
int f(int y) {
  return (y>2*y)?y:2*y;
}
```

```
int f(int y) {
  return 2*y ;
}
```



## Annotations de prédiction de branche

Les annotations **likely()** et **unlikely()** permettent à *gcc* d'optimiser les branchements, en lui indiquant une prédiction de branche.

Ces annotations ne sont pas *POSIX* mais propre à *gcc*.

```
static void next_reap_node(void)
{
   int node = __this_cpu_read(slab_reap_node);

   node = next_node(node, node_online_map);

   if (unlikely(node >= MAX_NUMNODES))
       node = first_node(node_online_map);

   __this_cpu_write(slab_reap_node, node);
}
```



## Annotations pour le passage des paramètres

L'annotation **asmlinkage** sur le prototype d'une fonction indique à *gcc* de toujours placer/chercher les paramètres dans la pile.

Sans cette annotation *gcc* cherche souvent à optimiser les appels à la fonction laissant les paramètres dans des registres.

Cette annotation bloque cette optimisation mais simplifie l'appel à la fonction depuis du code assembleur.

On la retrouve par exemple dans tous les appels systèmes :

```
asmlinkage long sys_close(unsigned int fd);
```

**asmlinkage** est en fait une macro défini dans *asm/linkage.h* :

```
#define asmlinkage CPP_ASMLINKAGE __attribute__((syscall_linkage))
```



## Les unions en mémoire

Une **union** est une type spécial permettant de stocker différents types de données dans un même espaces mémoire :

*chaque champ d'une* **union** *est un alias typé du même espace*

```
union {
        short x;
        long y;
        float z;
} monUnion;
```

Un exemple d'utilisation dans le noyau :

```
union thread_union {
        struct thead_info thed_info;
        usigned long stack[2048]; // pile de 8Ko
}
```



## Structure avec tableau de taille variable

En C il faut associer le tableau et sa taille :
$\Longrightarrow$ utilisation d'une **struct** (la taille devient un attribut)



## Structure avec tableau de taille variable

En C il faut associer le tableau et sa taille :
$\Longrightarrow$ utilisation d'une **struct** (la taille devient un attribut)

```
struct tab {
        int length;
        char *contents;
};

struct buf *thisTab = (struct buf *) malloc (sizeof (struct buf));
thisTab->length = (char *)  malloc (SIZE_XXX);
thisTab->length = SIZE_XXX;
```



## Structure avec tableau de taille variable

En C il faut associer le tableau et sa taille :
$\Longrightarrow$ utilisation d'une **struct** (la taille devient un attribut)

```
struct tab {
        int length;
        char *contents;
};

struct buf *thisTab = (struct buf *) malloc (sizeof (struct buf));
thisTab->length = (char *)  malloc (SIZE_XXX);
thisTab->length = SIZE_XXX;
```

Cette implémentation classique rend complexe :

- **la libération :** il faut faire deux `free`
- **la copie :** dupliquer la structure ne suffit pas



## Structure avec tableau de taille variable

**Solution :** le **struct hack** ou **tail-padded structures**

## Structure avec tableau de taille variable

**Solution :** le **struct hack** ou **tail-padded structures**

```
struct buf {
        int length;
        char contents[0];
};

struct buf *thisTab = (struct buf *) malloc (
                                sizeof (struct buf) + SIZE_XXX);
thisTab->length = SIZE_XXX;
struct tab anotherTab = { 3, { 1, 2, 3 } };
```



## Structure avec tableau de taille variable

**Solution :** le **struct hack** ou **tail-padded structures**

```
struct buf {
        int length;
        char contents[0];
};

struct buf *thisTab = (struct buf *) malloc (
                                sizeof (struct buf) + SIZE_XXX);
thisTab->length = SIZE_XXX;
struct tab anotherTab = { 3, { 1, 2, 3 } };
```

Cette implémentation simplifie :

- **la libération :** un seul `free` libère la mémoire
- **la copie :** dupliquer la structure suffit pour copier le tableau



## Structure avec tableau de taille variable

**Solution :** le **struct hack** ou **tail-padded structures**

```
struct buf {
        int length;
        char contents[0];
};

struct buf *thisTab = (struct buf *) malloc (
                                sizeof (struct buf) + SIZE_XXX);
thisTab->length = SIZE_XXX;
struct tab anotherTab = { 3, { 1, 2, 3 } };
```

Cette implémentation simplifie :

- **la libération :** un seul `free` libère la mémoire
- **la copie :** dupliquer la structure suffit pour copier le tableau

Plusieurs implémentation sont possibles :

- `int content[]` : dans la norme *C99* (**flexible array member**)
- `int content[1]` : portable mais ambigu
- `int content[0]` : moins ambigu mais non *pedantic*



## Vecteurs VS Pointeurs

```
void main (void)
{
     char *yes = "da";
     char oui[4];

     yes = oui ;
     oui = yes ;
}
```

```
gcc main.c && ./a.out
```



## Vecteurs VS Pointeurs

```
void main (void)
{
     char *yes = "da";
     char oui[4];

     yes = oui ;
     oui = yes ;
}
```

```
gcc main.c && ./a.out
   main.c:7:10: erreur: assignment to expression with array type
      oui = yes
            ^
```



## Vecteurs VS Pointeurs

```
void main (void)
{
     char *yes = "da";
     char oui[4];

     yes = oui ;
     oui = yes ;
}
```

```
gcc main.c && ./a.out
   main.c:7:10: erreur: assignment to expression with array type
      oui = yes
            ^
```

**yes :** est un pointeur de caractère contenant l'adresse du premier caractère de la chaîne `"da"`

**oui :** est un identificateur de vecteur. C'est une **constante symbolique**.



## Ambigüité des identificateurs de vecteur.

```
void main (void)
{
     char *yes = "da";
     char oui[4];

     printf("%p - %p", yes, &yes);
     printf("%p - %p", oui, &oui);
}
```

```
gcc main.c && ./a.out
```



## Ambigüité des identificateurs de vecteur.

```
void main (void)
{
     char *yes = "da";
     char oui[4];

     printf("%p - %p", yes, &yes);
     printf("%p - %p", oui, &oui);
}
```

```
gcc main.c && ./a.out
        0x4005e4 - 0x7fff629b2b28
   0x7fff629b2b20 - 0x7fff629b2b20
```

## Ambigüité des identificateurs de vecteur.

```
void main (void)
{
     char *yes = "da";
     char oui[4];

     printf("%p - %p", yes, &yes);
     printf("%p - %p", oui, &oui);
}
```

```
gcc main.c && ./a.out
         0x4005e4 - 0x7fff629b2b28
   0x7fff629b2b20 - 0x7fff629b2b20
```

Un **identificateur de vecteur** est une **constante symbolique**. Son adresse n'a donc pas vraiment de sens, mais elle est confondue par le compilateur avec la valeur de cette constante.



## Les pointeurs de fonction : déclaration

Un pointeur de fonction se déclare suivant la syntaxe :

```
type_de_retour(* nom_du_pointeur) (liste_des_arguments);
```

- Déclaration d'un pointeur de fonction sans retour et sans argument :

```
void (*monPointeur)(void);
```

- Déclaration d'un pointeur de fonction avec retour et arguments :

```
int (*monPointeur)(int, char);
```



## Les pointeurs de fonction : obtenir une adresse

Pour obtenir l'adresse d'une fonction on utilise l'opérateur **&** :

```
void maFonction (int toto)
{
  ....
}

void (* monPointeur) (int);     /* Déclaration    */
monPointeur = &maFonction;      /* Initialisation */
```

Cependant, l'identifiant d'une fonction est aussi son adresse. En jouant sur cette ambiguïté on peut aussi écrire :

```
monPointeur = maFonction;       /* Initialisation */
```



## Les pointeurs de fonction : appeler la fonction

```
#include <stdio.h>

void afficherBonjour(char * nom)
{
     printf("Bonjour %s\n",nom);
}

int main (void)
{
     void (*pointeurSurFonction)(char *);    /* Déclaration    */
     pointeurSurFonction = afficherBonjour;  /* Initialisation */
     (*pointeurSurFonction)("zero");         /* Appel          */
     return 0;
}
```

Comme pour l'adresse, on peut jouer sur l'ambiguïté pour écrire :

```
     pointeurSurFonction("zero");            /* Appel simplifié*/
```



## Les pointeurs de fonction : fonction en paramètre

L'utilisation de **callback**, très fréquente dans le noyau, repose sur le passage en paramètre de pointeurs de fonction.

```
void myRelease (struct elem *elem)
{
    ...
}

void elem_put(struct elem *elem, void (* release)(struct elem *))
{
    elem->refcount--;
    if (!elem->refcount)
        release(elem);
}

void main (void)
{
    struct elem myElem;
    elem_put(myElem, myRelease)
}
```



## Les pointeurs de fonction : retourner une fonction

Beaucoup plus rare dans le noyau que le *callback*, une fonction peut retourner un pointeur vers une autre fonction :

```
type_de_retour_de_la_fonction_retournee (* ma_fonction (liste_arg)) (liste_arg_fonction_retournee)
```

Exemple d'une fonction qui retourne un pointeur vers `atoi` :

```
int atoi(const char *nptr){
        ...
}

int (* maFonction (void)) (const char *) {
        return atoi;
}
```



## Outline





## Règles de style : indentation

Le noyau tente de maintenir une certaine homogénéité. Il faut donc se tenir strictement aux règles de style et d'indentation définies dans :

**doc/Documentation/CodingStyle**

**longueur des lignes :** il faut veiller à ne pas dépasser les 80 caractères.

**identation :**
- l'indentation utilise le caractère **tabulation**
- l'indentation doit correspondre à 8 caractères

```
Now, some people will claim that having 8-character
indentations makes the code move too far to the right,
and makes it hard to read on a 80-character terminal
screen. The answer to that is that if you need more
than 3 levels of indentation, you're screwed anyway,
and should fix your program.
```

## Règles de style : accolades

- Les accolades ouvrantes se mettent en fin de ligne sauf pour les fonctions où elles sont placées sur une ligne dédiée.

```
for (i = 0 ; 6 > i ; i++) {
  x++;
}
```

```
int inc (int x)
{
  return x++ ;
}
```

- Limiter l'utilisation des accolades dans les branchements conditionnels au cas où elles sont nécessaires sur au moins une branche.

```
if (y > x) {
  x = y;
  y++;
}
```

```
if (x < 0)
  x = -x;
```

```
if (y > x) {
  x = y;
  y++;
} else {
  x++;
}
```



## Règles de style : espaces

L'utilisation des espaces suit une logique *function-versus-keyword usage* :

**un espace après :** `if`, `switch`, `case`, `for`, `do` et `while`

**pas d'espace après :** – les identificateurs de fonction
– `sizeof`, `typeof`, `alignof` et `__attribute__`

Pour les opérateurs tout dépend de l'arité :

**espaces autour :** des opérateurs binaires
`=` `+` `-` `<` `>` `*` `/` `%` `|` `&` `^` `<=`

**pas d'espace après :** des opérateurs unaires `&` `*` `+` `-` `~` `!`

**pas d'espace autour :** des opérateurs d'accès aux champs `.` `->`

Pour les parenthèses :

**les parenthèses ouvrantes :** ne doivent pas être suivies d'un espace

**les parenthèses fermantes :** ne doivent pas être précédées d'un espace

Il ne doit pas y avoir d'espace en fin de ligne même si celle-ci est vide.



## Outline





## Ne pas abuser des appels de fonction

**Contrairement à la pile utilisateur, la pile noyau est très petite**

Sa taille maximum est définie au moment de la compilation du noyau et on ne peut pas la faire grandir dynamiquement.

Usuellement, elle tient sur 2 pages, soit :

- **8Ko** pour une architecture 32-bit
- **16Ko** pour un architecture 64-bit

Il faut donc éviter :

- les grosses allocations en pile
- les fonctions récursives trop profondes.



## Ne pas utiliser de nombres flotants

**Les opérations sur les flottants sont complexes et couteuses**

C'est le noyau qui s'en charge à l'aide d'une **trap** spéciale.

Le mécanisme pour passer des entiers aux flottants dépend alors des architectures et utilise des registres dédiés.

Il faut absolument **éviter l'utilisation de flottant** dans le code du noyau. Car la gestion du registre doit alors se faire à la main.



## Utilisation de types génériques

Pour assurer la portabilité d'une architecture à une autre, il faut utiliser des **types génériques** propres au noyau définis dans *linux/types.h* :

```
u8 : unsigned byte (8 bits)
u16 : unsigned word (16 bits)
u32 :unsigned 32-bit value
u64 : unsigned 64-bit value
```

```
s8 : signed byte (8 bits)
s16 : signed word (16 bits)
s32 : signed 32-bit value
s64 : signed 64-bit value
```

Exemple de fonction issue du bus i2c :

```
s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
```



## API noyau : Types génériques

Pour des variables pouvant être visibles depuis l'espace utilisateur (ex : `ioctl`), il est demandé d'utiliser les types préfixés par `__` :

```
__u8 unsigned byte (8 bits)
__u16 unsigned word (16 bits)
__u32 unsigned 32-bit value
__u64 unsigned 64-bit value
```

```
__s8 signed byte (8 bits)
__s16 signed word (16 bits)
__s32 signed 32-bit value
__s64 signed 64-bit value
```



## API noyau : Types génériques

Exemple d'envoi d'un message de contrôle à un device USB :

```
struct usbdevfs_ctrltransfer {
  __u8 requesttype; __u8 request;
  __u16 value; __u16 index; __u16 length;
  __u32 timeout; /* in milliseconds */
  void *data;
};

#define USBDEVFS_CONTROL_IOWR('U', 0, struct usbdevfs_ctrltransfer)
```

## Outline





## Les dangers de la programmation noyau

Travailler directement au cœur du noyau peut le rendre instable et engendrer un `KERNEL PANIC`, le rendant donc inutilisable !

Avant toute installation d'un nouveau noyau, il est conseillé d'en enregistrer un autre dans le *bootloader* qui pourra servir de démarrage de secours.

Il est conseiller, si possible (voir plus loin), de travailler son code sous forme de modules qui pourront être chargé dynamiquement dans un système stable.

### Mise en garde

Dans tous les cas, gardez à l'esprit qu'un bogue peut corrompre votre système de fichier ou un driver de périphériques.
Vous pouvez donc **perdre définitivement toutes les données** stockées dans un périphérique connecté à votre système.



## Outline





## Méthode 1 : travail en local.

La méthode la plus simple pour développer du code noyau, reste de **travailler en local** sur sa machine et de (dé)charger manuellement sur le noyau en courant.

Plusieurs méthodes existent pour récupérer des informations de debugage :

```
sudo tail -f /proc/kmsg

sudo tail -f /var/log/messages

dmesg [ -c ] [ -n niveau ] [ -s taille ]

syslogd
```



## Méthode 1 : travail en local.

**Avantage :** facile à mettre en place et à utiliser.
**Inconvénient :** si le noyau devient instable, il peut être nécessaire

de rebooter. A préconiser pour de petits drivers mais à déconseiller vivement pour des drivers complexes (réseau ou *vfs* par exemple). C'est envisageable pour des modules, mais s'il est nécessaire de modifier le cœur même du noyau, alors cette technique est à éviter.



## Méthode 2 : User Mode Linux

**User Mode Linux** ou **UML** est un noyau Linux compilé qui peut être exécuté dans l'espace utilisateur comme un simple programme. Il permet donc d'avoir plusieurs systèmes d'exploitation virtuels (principe de virtualisation) sur une seule machine physique hôte exécutant Linux.



## Méthode 2 : User Mode Linux.

**Avantage :**

- Lancer des noyaux sans avoir besoin de redémarrer la machine.
- Si un *UML* plante, le système hôte n'est pas affecté.
- Un utilisateur sera root sur un *UML*, mais pas sur l'hôte.
- *gdb* peut servir à débuguer le noyau en développement puisqu'il est considéré comme un processus normal.
- Il permet de mettre en place un réseau complètement virtuel de machines Linux, pouvant communiquer entre elles. Il est alors possible des fonctionnalités réseaux.

**Inconvénient :**

- Très lent, plutôt conçu pour des tests fonctionnels que pour la performance.
- Nécessite de patcher le noyau



## Méthode 2 BIS : Kernel Mode Linux (KML).

Le **Kernel Mode Linux** (**KML**) est la technique réciproque de UML, permet d'exécuter dans le noyau un processus habituellement prévu pour l'espace user.
Tout comme pour UML, cela nécessite de patcher le noyau et

d'activer la fonctionnalité lors de la Compilation du noyau.
Les architectures supportées sont : *IA-32* et *AMD64*.

Actuellement, les binaires ne peuvent pas modifier les registres

suivants : *CS*, *DS*, *SS* ou *FS*.

## Méthode 3 : machine virtuelle.

Le développement de code noyau peut aussi se faire dans une **machine virtuelle**, s'exécutant au dessus d'un OS "classique". Ces machines virtuelles sont dites de type 3 : elles permettent d'émuler une machine nue de façon à avoir un système d'exploitation à l'intérieur d'un autre. Les deux systèmes peuvent alors être différents.
Parmi de telles architectures on peut citer :

- *QEMU*,
- *VMWARE*,
- *BOCHS*,
- *VirtualBox*.



## Méthode 3 : machine virtuelle.

Voici un code permetant d'utiliser **QEMU** :

```
# Création du rootfs
mkdir iso
# Création de l'image ISO
mkisofs -o rootfs-dev.iso -J -R ./iso
# Cela peut être une recopie d'un média
dd if=/dev/dvd of=dvd.iso # pour un dvd
dd if=/dev/cdrom of=cd.iso # pour un cdrom
dd if=/dev/scd0 of=cd.iso # pour cdrom scsi
# Simulation
qemu -boot d -cdrom ./rootfs-dev.iso
# Montage
sudo modprobe loop
sudo mount -o loop rootfs-dev.iso /mnt/disk
# Démontage
sudo umount mnt/disk
```



## Méthode 3 : machine virtuelle.

**Avantages :**

- Très pratique pour développer à l'intérieur du noyau.
- Pas besoin de patcher le noyau comme avec *UML*.

**Inconvénient :**

- Dépend de la puissance de la machine hôte.
- Performance lié à la présence d'une virtualisation matérielle (type *Intel-VT*).
- Peut nécessiter de régénérer l'image à chaque fois que l'on souhaite la tester.



## Méthode 4 : via un seconde machine.

La dernière méthode consiste à travailler sur une **seconde machine de développement**. Cette machine est reliée à la machine principale qui peut alors la monitorer.
De loin la technique la plus adaptée car permet de développer au

coeur du noyau ou bien des modules complexes.
Cette technique est de plus adaptée pour un usage embarqué.



## Méthode 4 : via un seconde machine.

**Avantages :**

- Très pratique pour des développement sur le noyau même.
- Permet de débuguer (via le patch kdb et l'utilitaire kgdb) via la liaison série ou le réseau le noyau courant du second système en pouvant poser un point d'arrêt.

**Inconvénient :**

- Nécessite de disposer d'une seconde machine.



## Remplacement le port série pour le debugage

**Sur la plate-forme de développement**

- Pas de problème. Vous pouvez utiliser un convertisseur USB<-> série. Bien supporté par Linux. Ce périphérique apparaît en tant que /dev/ttyUSB0.

**Sur la cible :**

- Vérifiez si vous avez un port IrDA. C'est aussi un port série.
- Si vous avez une interface Ethernet, essayez de l'utiliser.
- Vous pouvez aussi connecter en JTAG directement les broches série du processeur (vérifiez d'abord les spécifications électriques !).



## Outline





## API noyau :

**ATTENTION**

Lorsque vous allez programmer dans le noyau, oubliez toutes les bibliothèques que vous aviez l'habitude d'utiliser et en premier lieu la *libc*.

Heureusement, vous ne serez pas totalement démuni. Le noyau est totalement autonome et implémente lui même une série de fonctionnalités de base.
L'ensemble de ces fonctions disponibles est parfaitement

documenté dans la documentation **kernel-api** :

```
linux/Documentation/DocBook/kernel-api.tmpl
```

## API noyau : Affichage.

Une des fonctionnalités nécessaire à tout développement est celle de l'affichage. Avec **printk()**, le noyau offre une fonction au fonctionnement quasi identique au classic *printf()*.
Il existe cependant quelques différences :

- Toute chaine de caractères est censée être préfixée par une valeur de priorité. Le fichier *kernel.h* définis 8 niveaux qui vont de **KERN_EMREG** à **KERN_DEBUG**.
- Le flux est récupéré par **klogd**, peut passer dans un *syslogd* et finit généralement dans */var/log/kern.log*.

**Exemple**

```
printk(KERN_DEBUG "Au retour de f() : i=%i", i);
```



## API noyau : Manipulation de la mémoire.

L'allocation de mémoire dans le noyau se fait grâce à la fonction **kmalloc()**. Pendant noyau de la fonction *malloc()*, cette fonction présente des caractéristiques propres :

- Rapide (à moins qu'il ne soit bloqué en attente de pages)
- N'initialise pas la zone allouée
- La zone allouée est contiguë en RAM physique
- Allocation par taille de $2^n - k$ ($k$ : quelques octets de gestion) : Ne demandez pas 1024 quand vous avez besoin de 1000 : vous recevriez 2048 !

**Exemple**

```
data = kmalloc(sizeof(*data), GFP_KERNEL);
```



## Organisation de la mémoire.

**Remarque**

Il est possible d'étendre l'utilisation de la mémoire au delà des 4 Go Via l'utilisation de la **mémoire haute** (**ZONE_HIGHMEM**).



## API noyau : Options du kmalloc.

Les options du *kmalloc* sont définies dans *include/linux/gfp.h* (**GFP** pour : Get Free Pages) :

- **GFP_KERNEL :** Allocation mémoire standard du noyau. Peut être bloquante. Bien pour la plupart des cas.
- **GFP_ATOMIC :** Allocation de RAM depuis les gestionnaires d'interruption ou le code non lié aux processus utilisateurs. Jamais bloquante.
- **GFP_USER :** Alloue de la mémoire pour les processus utilisateurs. Peut être bloquante. Priorité la plus basse.
- **GFP_NOIO :** Peut être bloquante, mais aucune action sur les E/S ne sera exécutée.
- **GFP_NOFS :** Peut être bloquante, mais aucune opération sur les systèmes de fichier ne sera lancée.
- **GFS_HIGHUSER :** Allocation de pages en mémoire haute en espace utilisateur. Peut être bloquante. Priorité basse.



## API noyau : Autres options du kmalloc.

Autres macros définissant des options supplémentaires et pouvant être ajoutées avec l'opérateur | :

- **__GFP_DMA :** Allocation dans la zone DMA
- **__GFP_HIGHMEM :** Allocation en mémoire étendue (x86 et sparc)
- **__GFP_REPEAT :** Demande d'essayer plusieurs fois. Peut se bloquer, mais moins probable.
- **__GFP_NOFAIL :** Ne doit pas échouer. N'abandonne jamais. Attention : à n'utiliser qu'en cas de nécessité !
- **__GFP_NORETRY :** Si l'allocation échoue, n'essaye pas d'obtenir de page libre.



## API noyau : Manipulation de la mémoire par page.

Pour l'allocation de grosses tranches mémoire, il existe une série de fonctions plus appropriées que *kmalloc*, puisqu'elles fonctionnent par **page mémoire** :

- **unsigned long get_zeroed_page(int flags) :** Retourne un pointeur vers une page libre et la remplit avec des zéros
- **unsigned long __get_free_page(int flags) :** Identique, mais le contenu n'est pas initialisé
- **unsigned long __get_free_pages(int flags, unsigned long order) :** Retourne un pointeur sur une zone mémoire de plusieurs pages continues en mémoire physique avec *order*$= log_2$(nombre de pages).



## API noyau : Mapper des adresses physiques

La fonction **vmalloc()** peut être utilisé pour obtenir des zones mémoire continues dans l'espace d'adresse virtuel, même si les pages peuvent ne pas être continues en mémoire physique :

```
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

La fonction **ioremap()** ne fait pas d'allocation, mais fait correspondre le segment donné en mémoire physique dans l'espace d'adressage virtuel.

```
void *ioremap(unsigned long phys_addr, unsigned long size);
void iounmap(void *address);
```



## API noyau : Les *wait queues*.

Les **wait queues** sont une liste de tâches endormies, en attente d'une ressource : lecture d'un *pipe*, attente d'un paquet sur une interface réesaux, etc.
Pour s'enregistrer dans l'une de ces queues, un processus on peut

utiliser une des deux fonctions suivantes :

- **sleep_on(queue) :** Le processus s'endort et ne sera réveiller que la ressource sera disponible.
- **interruptible_sleep_on(queue) :** Le processus peut aussi être réveillé par un signal.

Lorsque la ressource est prête, son **handler** reveille tous les processus de la liste avec un appel à la fonction **wake_upqueue**.

## API noyau : Les *wait queues*.

**Attention**

Comme avec les *pthread_cond_wait()*, il est possible que l'un des processus réveillés ne soit activé par le *scheduler* qu'une fois la ressource utilisée par d'autre.

Il faut donc toujours tester la disponibilité de la ressource après chaque sortie d'une *wait queue*.



## API noyau : Les *task queues*.

Les **task queues** permettent à un processus de procrastiner une ou plusieurs tâches.
Chaque *task queues* contient une liste chainée de structures

contenant un pointeur de fonction (la tâche) et un pointeur de donnée (l'objet de la tache).
A chaque fois qu'une *task queues* est exécutée, toutes les fonctions

enregistrées sont exécutées, une à une, avec leurs données en paramètre.



## API noyau : Appels systèmes.

Le noyau offre aux applications un ensemble d'appels système (plus de 200) pour réaliser des tâches simple vue de l'application mais complexe vu du noyau.
Si l'on peut utiliser les librairies standard, on peut très bien utiliser

ces appels systèmes au sein même d'un code noyau.
Un bon programmeur système n'aura donc pas de mal à coder

dans le noyau.



## API noyau : Convention de retour

Les fonctions du noyau suivent la même convention de retour que les appels système :

- **positif ou nul** en cas de succès
- **négatif** en cas d'erreur (opposé de la valeur *errno*).

Si la fonction retourne un pointeur, on utilise une autre convention :

- le codes d'erreur est re-encodé par la macro **ERR_PTR()** et est retourné comme un pointeur.
- la fonction appelante utilise la macro **IS_ERR()** pour déterminer s'il s'agit d'un code d'erreur, au quel cas la macro **PTR_ERR()** permet de l'extraire.



## API noyau : Convention de retour

**Exemple**

```
asmlinkage long sys_open(const char* filename) {
  char* tmp;
  int fd, error;

  tmp = getname(filename);
  fd = PTR_ERR(tmp);
  if (! IS_ERR(tmp)) {
    fd = get_unused_fd();
    if(fd >= 0) {
      \* On peut ouvrir le fichier *\
      \* Et retourner le file descriptor *\
    }
  }
  return fd;
}
```



## API noyau : Table des symboles.

Le noyau maintient une table des symboles destinée à l'édition de liens dynamiques lors de l'insertion des modules. Ces symboles sont visibles dans **/proc/ksyms**.

Tout symbole qui peut être utilisé dans un module doit être explicitement exporté avec la macro **EXPORT_SYMBOL()**.

Si un module utilise des symboles d'un autre module, il est dit dépendant de ce dernier : commande *depmod*.



## Outline





## Concurrence et synchronisation : problèmes.

Le problème des accès concurrents à une ressource critique du noyau peut avoir une cause matérielle et/ou logicielle :

- **Préemption :** Depuis sa version 2.6, Linux est devenu un noyau préemptif. Un processus peut être interrompu au milieu d'un code noyau et laissé sa place à un autre processus.
- **Multi-processeurs :** Avec l'arrivée des machines multi-processeurs, se pose le problème de l'exécution parallèle de code noyau.

## Concurrence et synchronisation : Les Solutions.

Plusieurs solutions sont envisageables pour résoudre le problème des accès concurrents :

1. Bloquer les interruptions
2. Opérations atomiques
3. Big Kernel Lock
4. Sémaphores
5. Spinlocks



## Concurrence et synchronisation : Ininterruptible.

Dans une architecture mono-processeur, le problème résulte uniquement de la préemption. On peut donc le résoudre en désactivant les interruptions : le code devient alors **non-préemptible**.
Pour ce faire, on peut utiliser les macro *local_irq_disable()* et

*local_irq_enable()* qui utilisent l'instruction assembleur *cli* (resp. *sti*) pour désactiver (resp. activer) les interruptions **sur le processeur local**.

**Exemple**

```
local_irq_disable();
        /* code non préemptible ... */
local_irq_enable();
```



## Concurrence et synchronisation : Opérations atomiques

Il possible d'éviter le problème d'accès concurrent en utilisant des fonctions garantissant l'atomicité d'une opération.

Ces fonctions sont dépendantes de l'architecture matérielle et sont définies dans **linux/include/asm/atomic.h**.

**Exemple**

Si *p* est un pointeur d'entier on peut utiliser : *atomic_inc(p)*, *atomic_set(p,i)* et *atomic_add(p)*.



## Concurrence et synchronisation : BKL

Dans un système multiprocesseur on ne peut résoudre le problème en bloquant les interruptions. Une solution consiste alors à verrouiller l'ensemble du noyau avec un **Big Kernel Lock (BKL)**.
**Avantage :** C'est la solution la plus simple.

**Inconvénient :** C'est extrêmement couteux car on bloque

l'ensemble des processeurs.

**Exemple**

```
lock_kernel();
/* critical region ... */
unlock_kernel();
```



## Concurrence et synchronisation : BKL

**Fin annoncée du BKL**

Suite à une longue discussion sur la liste diffusion du noyau, il a été décidé par Linus Torvalds de supprimer progressivement le Big Kernel Lock.

Le travail va se dérouler dans la branche "kill-the-BKL" mais il sera sans doute possible aux utilisateurs du noyau principal d'activer une nouvelle option (CONFIG_DEBUG_BKL) afin de participer eux aussi à la chasse aux bugs.



## Concurrence et synchronisation : Sémaphores

Les **sémaphores** permettent de réaliser une synchronisation entre les processus. Cette méthode pause tout de même le problème de l'attente passive qui peut conduire à un changement de contexte.

**Exemple**

```
struct semaphore mr_sem;
sema_init(&mr_sem, 1);
/* usage count is 1 */
if (down_interruptible(&mr_sem))
/* semaphore not acquired; received a signal ... */
/* critical region (semaphore acquired) ... */
up(&mr_sem);
```

Comme le processus s'endort pour attendre le sémaphore, cette solution est réservée au code noyau s'exécutant en *contexte utilisateur*.



## Concurrence et synchronisation : Spinlocks

Les **spinlocks** sont une implémentation de sémaphore avec attente active. Leur manipulation est plus complexe que celle de sémaphores "passifs".

```
spinlock_t mon_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mon_lock, flags);
/* critical section ... */
spin_unlock_irqrestore(&mon_lock, flags);
```

Les spinlocks ne produisent aucun code si le noyau est compilé mode non-préemptible et sans support SMP.



## Outline

## Module or not module : Opter pour le module.

Lorsque l'on développe une fonctionnalité pour le noyau, on doit choisir entre :

- intégrer son **code dans le noyau** au travers d'un patch. Elle sera alors intégrer statiquement au noyau.
- créer un nouveau **module** que l'on pourra charger dynamiquement dans le noyau.

**Règle de choix**

Toujours choisir la solution du module si elle est techniquement possible.



## Avantages et limites des modules.

**Avantages :**

- Plus simple à développer.
- Simplifie la diffusion.
- Évite la surcharge du noyau.
- Permet de résoudre les conflits.
- Pas de perte en performance.

**Limites :**

- On ne peut pas modifier les structures internes du noyau. Par exemple, ajouter un champs dans le descripteur des processus.
- Remplacer une fonction lié statiquement au noyau. Par exemple, modifier la manière dont les cadres de page sont alloués.



## Un module Linux c'est quoi ?

**Définition**

*Un module est une bibliothèque chargée dynamiquement dans le noyau et pouvant générer un appel de fonction au moment de son chargement et de son déchargement.*

Un module minimal comme le nôtre contient au moins les fichiers d'en-tête suivants :

```
#include <linux/module.h>  /* API des modules */
#include <linux/kernel.h>  /* Si besoin : KERN_INFO dans printk() */
#include <linux/init.h>    /* Si besoin : fonction d'init et d'exit */
```

Un module peut enregistrer des fonctions à exécuter lors de son chargement et de son déchargement :

```
module_init(pointeur_fonction_init);
module_exit(pointeur_fonction_exit);
```



## Identification du module

Il est possible d'identifier le module en utilisant des macros spécifiques, le plus souvent placées au début du code source :

```
MODULE_DESCRIPTION("Hello World module");
MODULE_AUTHOR("Julien Sopena, LIP6");
MODULE_LICENSE("GPL");
```

```
modinfo helloworld.ko
  filename:       helloworld.ko
  description:    Hello World module
  author:         Julien Sopena, LIP6
  license:        GPL
  vermagic:       2.6.30-ARCH 686 gcc-4.4.1
  depends:
```



## Licence de distribution du module

Depuis le noyau 2.6, la définition de la licence est nécessaire.

Dans le cas contraire, on obtient un message d'erreur dans les traces du noyau :

```
module license 'unspecified' taints kernel.
```



## Exemple de module : helloworld.c

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

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

static void hello_exit(void) {
  printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);

module_exit(hello_exit);

MODULE_LICENSE("GPL");
```



## Compiler un module

Le Makefile ci-dessous est réutilisable pour tout module Linux 2.6. Lancez juste make pour construire le fichier hello.ko

```
[language=make]
# Makefile pour le module hello
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
  $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Il est à noter que les modules sont seulement compilés et pas linkés. Le linkage s'effectuant lors du chargement du driver dans le noyau Linux.

Dans les linux 2.4, l'extension des modules était **.o**. Désormais, avec la famille des 2.6, c'est **.ko** pour *kernel object*.



## Chargement et déchargement d'un module

Pour charger un module du noyau on utlise **insmod** :

```
insmod helloworld
```

Résultat au chargement :

```
Hello, world
```

Pour décharger un module du noyau on utlise **rmmod** :

```
rmmod helloworld
```

Résultat au déchargement :

```
Goodbye, cruel world
```

## Exemple de module avec paramètre : hello_parma.c

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
MODULE_LICENSE("GPL");
static char *whom = "world";
module_param(whom, charp, 0);
static int howmany = 1;
module_param(howmany, int, 0);
static int hello_init(void) {
  int i;
  for (i = 0; i < howmany; i++)
   printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
  return 0;
}
static void hello_exit(void) {
  printk(KERN_ALERT "Goodbye, cruel %s\n", whom);
}
module_init(hello_init);
module_exit(hello_exit);
```



## Passer des paramètres aux modules

Il existe 3 façons de passer des paramètres à un module :

- Avec insmod ou modprobe :
  `insmod ./hello_param.ko howmany=2 whom=universe`
- Avec modprobe en changeant le fichier /etc/modprobe.conf :
  `options hello_param howmany=2 whom=universe`
- Avec la ligne de commande du noyau, lorsque le module est lié statiquement au noyau :
  `options hello_param.howmany=2 hello_param.whom=universe`



## Dépendances de modules

Les dépendances des modules n'ont pas à être spécifiées explicitement par le créateur du module.

Elles sont déduites automatiquement lors de la compilation du noyau, grâce aux symboles exportés par le module :
A **dépend de** B si A utilise un symbole exporté par B.

Les dépendances des modules sont stockées dans :
*/lib/modules/<version>/modules.dep*

Ce fichier est mis à jour (en tant que root) avec **depmod** :

```
depmod -a [<version>]
```

La commande **modprobe** permet de charger un module avec toutes ses dépendances.



## Exemple de module avec dépendance

Dans l'exemple helloword.c, on remplace *printk()* par *my_printk()*
Puis implémente cette fonction dans un autre module *my_printk*.

Notons que ce module n'a pas de fonction *init* :

```c
/* my_printk.c */
#include <linux/init.h>
#include <linux/module.h>

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



## Exemple de module avec dépendance

Lorsque les deux modules sont compilés, on doit tout d'abord insérer le module définissant la fonction afin d'éviter une erreur de dépendance.

```
insmod helloworld2.ko
    insmod: error inserting 'helloworld2.ko': -1 Unknown symbol
inmodule
insmod ../my_printk/my_printk.ko
insmod helloworld2.ko
```

Autre solution, utiliser modprobe après instalation des modules :

```
modprobe -v helloworld2
    insmod /lib/modules/version_noyau/extra/my_printk.ko
    insmod /lib/modules/version_noyau/extra/helloworld2.ko
```