PNL - 4I402 : Programmer dans le noyau

Version 18.02

Julien Sopena[1]

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

Master SAR 1ère année - 2018/2019

Grandes lignes du cours

Rappels de C

Règles de style

Recommandation de programmation

Mise en garde

API noyau

- Afficher des informations
- Allocation mémoire
- Attendre un évènement
- Programmer une action
- Appel système depuis le noyau
- Retour de fonction

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

```
int f(unsigned 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(unsigned int a, unsigned int b) {
  return (a > b) ? a : b ;
}
```

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

```
int f(unsigned 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(unsigned int a, unsigned int b) {
  return (a > b) ? a : b ;
}
```

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

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

```
int f(unsigned 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 buf {
        int length;
        char *contents;
};

struct buf *thisBuf = (struct buf *) malloc (sizeof (struct buf));
thisBuf->length = (char *)  malloc (SIZE_XXX);
thisBuf->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 buf {
        int length;
        char *contents;
};

struct buf *thisBuf = (struct buf *) malloc (sizeof (struct buf));
thisBuf->length = (char *)  malloc (SIZE_XXX);
thisBuf->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 *thisBuf = (struct buf *) malloc (
                                sizeof (struct buf) + SIZE_XXX);
thisBuf->length = SIZE_XXX;
struct buf anotherTab = { 3, { 'a', 'b', 'c' } };
```

Structure avec tableau de taille variable

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

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

struct buf *thisBuf = (struct buf *) malloc (
                                sizeof (struct buf) + SIZE_XXX);
thisBuf->length = SIZE_XXX;
struct buf anotherTab = { 3, { 'a', 'b', 'c' } };
```

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 *thisBuf = (struct buf *) malloc (
                                sizeof (struct buf) + SIZE_XXX);
thisBuf->length = SIZE_XXX;
struct buf anotherTab = { 3, { 'a', 'b', 'c' } };
```

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

```c
#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 :

```c
     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

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
```

Outline

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);
```

Outline

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);
```

Outline

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*.

Outline

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.

Outline

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.

Outline

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

```
int _torture_create_kthread(int (*fn)(void *arg), void *arg,
           char *s, char *m, char *f, struct task_struct **tp)
{
        int ret = 0;

        VERBOSE_TOROUT_STRING(m);
        *tp = kthread_run(fn, arg, "%s", s);
        if (IS_ERR(*tp)) {
                ret = PTR_ERR(*tp);
                VERBOSE_TOROUT_ERRSTRING(f);
                *tp = NULL;
        }
        torture_shuffle_task_register(*tp);
        return ret;
}
EXPORT_SYMBOL_GPL(_torture_create_kthread);
```