

# A.P.I. du noyau Linux

Christophe BLAESS

christophe.blaess@logilin.fr

<https://www.blaess.fr/christophe/>



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

<b>Éléments de programmation noyau.....</b>	<b>3</b>
Chaînes de caractères et blocs.....	3
Fonctions numériques et conversions.....	5
<b>Mise au point et débogage.....</b>	<b>7</b>
Avertissements et paniques.....	7
<b>Éléments temporels.....</b>	<b>8</b>
Ticks.....	8
Mesure du temps.....	9
Travaux pratiques : mesure de la granularité de l'horodatage.....	10
Attentes.....	11
Actions différées.....	12
Travaux pratiques : appels-système de base avec /proc.....	14
<b>Gestion de la mémoire.....</b>	<b>15</b>
Allocations.....	16
Gestion des pages.....	19
Travaux pratiques : adresses virtuelles, réelles et pages.....	21

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* pour les dessins vectoriels
- *Gimp* pour les images bitmap

v. 9.12

<https://www.blaess.fr/christophe/>

<https://www.logilin.fr>

# Éléments de programmation noyau

## Chaînes de caractères et blocs

```
<linux/string.h>
```

### *Longueur de chaîne*

```
size_t strlen (const char *s);  
size_t strnlen (const char *s, size_t n);
```

### *Comparaison de chaînes*

```
int strcmp (const char *s1, const char *s2);  
int strncmp (const char *s1, const char *s2, size_t n);  
int strnicmp(const char *s1, const char *s2, size_t n);
```

### *Copie*

```
char *strncpy (char *s1, const char *s2, size_t n);  
char *strncpy (char *s1, const char *s2, size_t n);
```

La classique fonction `strcpy()` est marquée « *deprecated* » au profit de `strncpy()`.

Ces fonctions sont implémentées en espace noyau comme leurs contreparties de la bibliothèque C. Pour avoir le détail de leur utilisation, on peut consulter les pages de manuel habituelles.

### ***Concaténation***

```
char * strcat (char *s1, const char *s2);  
char * strncat (char *s1, const char *s2, size_t n);
```

### ***Recherche de caractères***

```
char * strchr (const char *s1, int c);  
char * strrchr (const char *s1, int c);  
char * strpbrk (const char *s1, const char *s2);  
size_t strspn (const char *s1, const char *s2);  
size_t strcspn (const char *s1, const char *s2);
```

### ***Recherche de sous-chaîne***

```
char * strstr (const char *s1, const char *s2);  
char * strsep (char **s1, const char *s2);
```

### ***Blocs mémoire***

```
void * memset (void *m, int c, size_t n);  
void * memcpy (void *m1, const void *m2, size_t n);  
void * memmove (void *m1, const void *m2, size_t n);  
int memcmp (const void *m1, const void *m2, size_t n);  
void * memchr (const void *m1, int c, size_t n);  
void * memscan (void *m, int c, size_t n);
```

## Fonctions numériques et conversions

```
int abs (int x);  
long labs (long x);
```

```
    min (x, y);        max (x, y);
```

```
int kstrtoint (const char *str, unsigned int base,  
              int *result);
```

```
int kstrtouint (const char *str, unsigned int base,  
               uint *result);
```

```
int kstrtoint_from_user (const char __user *str,  
                        size_t length,  
                        unsigned int base,  
                        int *result);
```

```
int kstrtouint_from_user (const char __user *ptr,  
                        size_t length,  
                        unsigned int base,  
                        unsigned int *result);
```

```
kstrtol(), kstrtoll(), kstrtos8(), kstrtos16(), kstrtos32(),  
kstrtos64(), kstrtobool()...
```

Ces fonctions sont préférables à `simple_strtol()` `simple_strtoul()` `simple_strtoll()`, etc qui ne permettent pas de détecter les erreurs de conversion

Pour le détail des fonction `kstrto...`(), voir le fichier `include/linux/kstrtox.h`

```
int sscanf (const char *s, const char *fmt, ...);  
int vsscanf (const char *s, const char *fmt, va_list args);
```

```
int sprintf (char *buf, const char *fmt, ...);  
int vsprintf (char *buf, const char *fmt, va_list args);  
int snprintf (char *buf, size_t size, const char *fmt...);  
int vsnprintf (char *buf, size_t size, const char *fmt,  
               va_list args);
```

# Mise au point et débogage

## Avertissements et paniques

### Cas extrêmes

(pas de synchronisation du système de fichiers)

```
panic("message");
```

### Avertissement dans les traces du noyau

```
WARN_ON(condition);  
WARN_ON_ONCE(condition);
```

```
WARN(condition, format...);  
WARN_ONCE(condition, format...);
```

On peut configurer dans `/proc/sys/kernel/panic` un délai en secondes au-delà duquel le système redémarre automatiquement en cas de *kernel panic*. Par défaut il contient `0` ce qui signifie « pas de redémarrage ».

# Éléments temporels

## Ticks

```
<linux/param.h>

#define HZ 1000                /* ticks par seconde */
```

```
<linux/jiffies.h>

unsigned long volatile jiffies; /* ticks depuis le boot */
```

```
int time_after      (unsigned long a, unsigned long b);
int time_after_eq   (unsigned long a, unsigned long b);
int time_before     (unsigned long a, unsigned long b);
int time_before_eq  (unsigned long a, unsigned long b);
```

time\_after(evt1, evt2) est vraie si l'heure evt1 est postérieure à evt2.

## À vous...

Observez et chargez le module de le module `example-III-01`. Comparez la valeur indiquée par la variable `jiffies` et celle fournie par la commande `uptime`.

La variable `jiffies` compte le nombre de *ticks* depuis le *boot*, mais sur les systèmes 32 bits elle part avec un décalage de cinq minutes pour garantir un rebouclage du compteur assez rapidement. On ne doit pas comparer des heures directement avec "<" ou ">" mais en utilisant les macros `time_after()`, etc.

## Mesure du temps

Il existe de nombreuses fonctions offrant des variations sur les types de données.  
Voir <linux/timekeeping.h> pour plus de détails.

Les plus classiques sont :

```
time64_t ktime_get_seconds(void);      // Depuis le boot
time64_t ktime_get_real_seconds(void); // Depuis 01/01/1970
```

Mesure en nanosecondes :

```
time64_t ktime_get_ns(void);           // Depuis le boot
time64_t ktime_get_real_ns(void);      // Depuis 01/01/1970
```

Le type **time64\_t** permet de stocker des nanosecondes, il existe des fonctions de conversions vers d'autres types comme struct timespec, struct timeval, jiffies, etc.

Le type **ktime\_t** a changé au cours du temps et il est préférable d'utiliser **time64\_t**.

### A vous...

Chargez le module `example-III-02`, et observez les valeurs fournies.

## Travaux pratiques : mesure de la granularité de l'horodatage

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

Exercice III-1 : mesure de la granularité de l'horodatage.

## Attentes

### *Attentes actives*

```
<linux/delay.h>

void ndelay (unsigned long nsec);
void udelay (unsigned long usec);
void mdelay (unsigned long msec);
```

### *Attentes préemptibles sans sommeil*

```
<linux/sched.h>

void schedule (void);
```

```
|| end = jiffies + duration;
|| while (time_before(jiffies, end))
||     schedule();
```

### *Sommeils*

```
void usleep_range (unsigned long min, unsigned long max);
void msleep      (unsigned long msecs);
void ssleep      (unsigned long seconds);
```

Les fonctions `ndelay()`, `udelay()`, `mdelay()` assurent des attentes avec une boucle de CPU actif. La précision est bonne s'il n'y a pas d'interruption qui survient. Il faut toujours utiliser ces fonctions avec des valeurs d'attente inférieures à 2000.

## Actions différées

```
<linux/timer.h>
```

```
struct timer_list {  
    void ( *function) (struct timer_list *);  
    unsigned long expires;  
    ...  
}  
  
timer_setup(struct timer_list *timer,  
            void (*callback)(struct timer_list *),  
            int flags);
```

```
void add_timer      (struct timer_list *timer);  
  
int  mod_timer      (struct timer_list *timer,  
                    unsigned long expiration);  
  
int  del_timer      (struct timer_list *timer);  
  
int  timer_pending  (struct timer_list *timer);
```

Le champ `expires` de la structure `timer_list` indique l'instant (valeur du compteur *jiffies*) auquel la fonction doit se déclencher. La structure `timer_list` contient d'autres champs que ceux mentionnés, il faut les initialiser correctement en appelant `init_timer()`.

La fonction `add_timer()` programme l'exécution différée. La fonction `mod_timer()` modifie l'instant de déclenchement, en reprogrammant une nouvelle exécution au besoin.

La fonction `timer_pending()` indique s'il y a encore une exécution ultérieure programmée pour la structure `timer_list` indiquée.

Les *flags* de `timer_setup()` permettent de configurer le comportement du *timer* vis-à-vis des coeurs de CPU ou des interruptions. On indique habituellement « 0 ».

### A vous...

Chargez le module `example - III - 03` et observez ses messages dans les traces du noyau.

## Timers de haute-résolution

Pour avoir une résolution inférieure à la milliseconde :

```
struct hrtimer {  
    enum hrtimer_restart (*function) (struct hrtimer *);  
    [...]  
}
```

```
void hrtimer_init (struct hrtimer *ht, clockid_t clock,  
                  enum hrtimer_mode mode);
```

On utilise généralement CLOCK\_REALTIME et HRTIMER\_MODE\_REL (relatif).

```
int hrtimer_start      (struct hrtimer *ht, ktime_t kt,  
                       const enum hrtimer_mode mode);  
int hrtimer_cancel     (struct hrtimer *ht);  
u64 hrtimer_forward    (struct hrtimer *ht, ktime_t now,  
                       ktime_t interval);  
u64 hrtimer_forward_now (struct hrtimer *ht, ktime_t kt);
```

```
ktime_t ktime_set (const long secs,  
                   const unsigned long nsecs)
```

La fonction appelée par le *hrtimer* doit renvoyer HRTIMER\_RESTART ou HRTIMER\_NORESTART.

### Voir aussi

- Fichiers `include/linux/hrtimer.h` et `include/linux/ktime.h`.

### À vous

Chargez le module `example-III-04` et affichant les fluctuations maximales d'un *hrtimer* à 10 kHz.

Quelle précision pouvez vous attendre d'un *hrtimer* sur votre système ?

## Travaux pratiques : appels-système de base avec /proc

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

Exercice III-2 : appels-système de base avec /proc.

## Gestion de la mémoire

Pour l'allocation occasionnelle de petits blocs (inférieurs à 32 pages environ), on conseille d'utiliser la famille `kmalloc()`. C'est l'utilisation classique pour les données internes d'un driver. Les pages allouées sont physiquement contiguës.

Pour l'allocation de gros blocs nécessitant une contiguïté en mémoire physique (pour des transferts DMA), on emploiera `__get_free_pages()`.

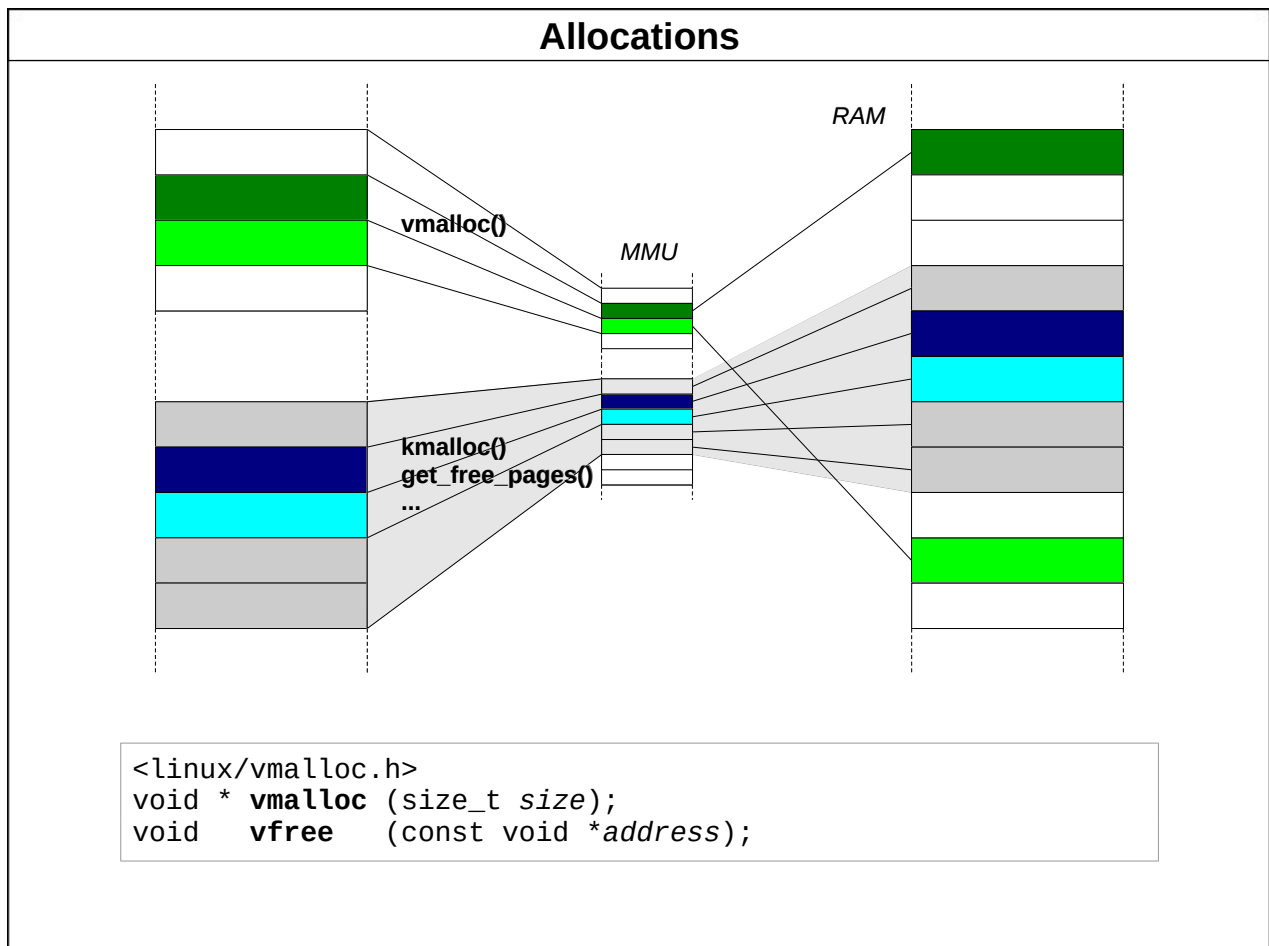
Enfin, pour allouer des gros blocs ne nécessitant pas de contiguïté physique, `vmalloc()` sera préférable (cas relativement rares pour les drivers).

La constante `PAGE_SIZE` correspond à la taille de page, et `PAGE_SHIFT` à la "largeur" en bits des pages.

Sur un PC, la constante `PAGE_SIZE` vaut 4096, et `PAGE_SHIFT` vaut 12 bits.

Dans le répertoire `drivers` du noyau 6.1, on rencontre :

- 3.262 appels à `kmalloc()`
- 3.411 appels à `kcalloc()`
- 15.652 appels à `kzalloc()`
- 443 appels à `vmalloc()`
- 108 appels à `__get_free_pages()`



`vmalloc()` renvoie un pointeur sur une zone de mémoire virtuelle. La mémoire allouée se trouve dans l'espace d'adressage du noyau.

L'utilisation de `vmalloc()` est généralement réservée aux blocs de tailles nettement supérieures à celle d'une page et ne nécessitant pas de contiguïté physique (pas de DMA).

### ***Famille de kcalloc()***

```
<linux/slab.h>

void *kalloc (size_t size, gfp_t flag);
void kfree (const void *address);

void *kccalloc (size_t size, size_t n, gfp_t flag);
void *kzalloc (size_t size, gfp_t flag);

void kfree_sensitive(const void *address); // depuis 5.9
void kzfree (const void *address);
```

*flag* doit contenir une seule des deux valeurs suivantes :

- GFP\_KERNEL : invocation depuis un appel-système (peut dormir),
- GFP\_ATOMIC : depuis un contexte d'interruption (pas de sommeil).

`kalloc()` renvoie un pointeur sur une zone de mémoire physiquement contiguë.

On emploie `kalloc()` pour les allocations de faible dimension. De manière portable, `kalloc()` est limité à des blocs de 128 Ko.

## ***Allocation de pages physiquement contiguës***

```
<linux/gfp.h>

unsigned long __get_free_pages (gfp_t flag, uint n);
unsigned long __get_free_page  (gfp_t flag);

void free_pages (unsigned long addr, uint n);
void free_page  (unsigned long addr);
```

`__get_free_pages()` fournit  $2^n$  pages.

La fonction `__get_free_pages()` permet d'obtenir des pages physiquement contiguës (comme `kmalloc()`) sans limite de taille. Les fonctions de la famille `free_pages()` utilisent une valeur de type `unsigned long` en guise d'adresse. On peut les forcer par un `cast` en `void*`. On utilise ces routines surtout pour des allocations de gros blocs utilisant du DMA.

## Gestion des pages

```
<linux/mm_types.h>
```

```
struct page {  
    [...]  
}
```

```
<asm/page.h>
```

```
struct page *virt_to_page (void *addr);  
void        page_address (struct page *pg);  
  
void        kmap          (struct page *pg);  
void        kunmap        (struct page *pg);  
  
struct page *pfn_to_page (int pfn);
```

```
<asm/io.h>
```

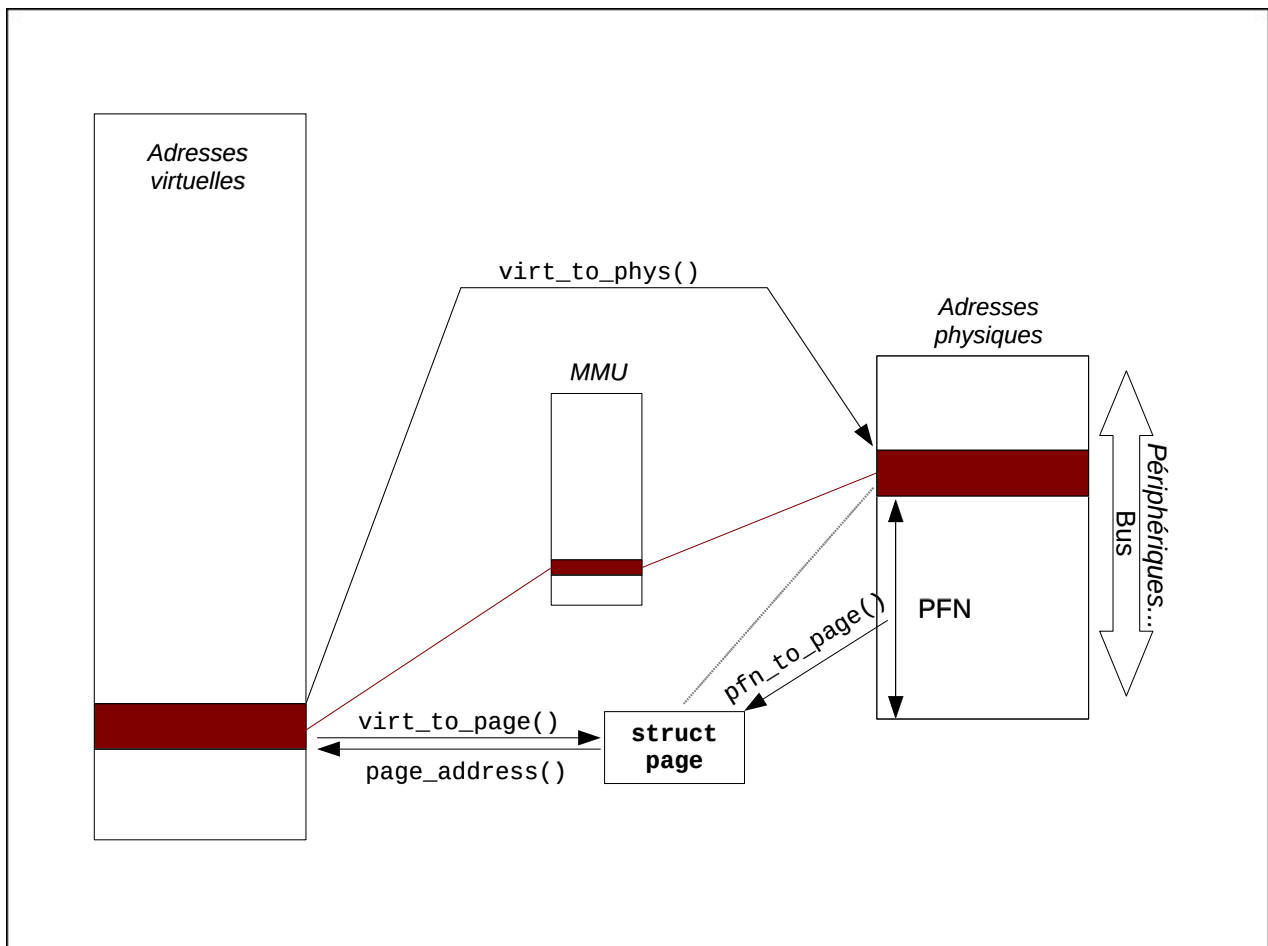
```
unsigned long virt_to_phys (void *addr);
```

La structure **page** est l'entité de base pour la gestion avancée de la mémoire. Elle contient divers champs que l'on n'utilise généralement pas dans les drivers, mais on la transmet directement entre diverses fonctions.

La fonction **virt\_to\_page()** renvoie un pointeur sur la structure **page** représentant une adresse.

La fonction **page\_address()** renvoie l'adresse où une page est visible dans l'espace noyau. Si la page n'est pas accessible, cette fonction renvoie NULL.

La fonction **kmap()** renvoie l'adresse où une page est accessible, en créant la projection si nécessaire. Il faudra libérer cette projection en appelant **kunmap()** après usage.



On obtient un pointeur sur la structure `page` correspondant à un numéro de page `pfn` (*Page Frame Number*) avec la fonction `pfn_to_page()`.

Pour obtenir un numéro de page, on décale l'adresse physique vers la droite de `PAGE_SHIFT` bits.

## Travaux pratiques : adresses virtuelles, réelles et pages

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

Exercice III-3 : adresses virtuelles, adresses réelles et pages mémoire.