

Les appels-système Linux

Christophe BLAESS

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



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

Appels-système.....	3
Principe.....	3
Suivi d'un appel-système.....	5
Travaux pratiques : appels-système invoqués par une application.....	8
Environnement du noyau.....	9
Contexte de tâche.....	9
Espaces d'adressage.....	10
Système de fichiers /proc.....	12
Création d'entrées.....	12
Callback de lecture.....	13
Lecture et écriture depuis un buffer.....	14
Callback d'écriture.....	15
Travaux pratiques : échange entre kernel et espace utilisateur.....	16

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

v. 9.12

Appels-système

Principe

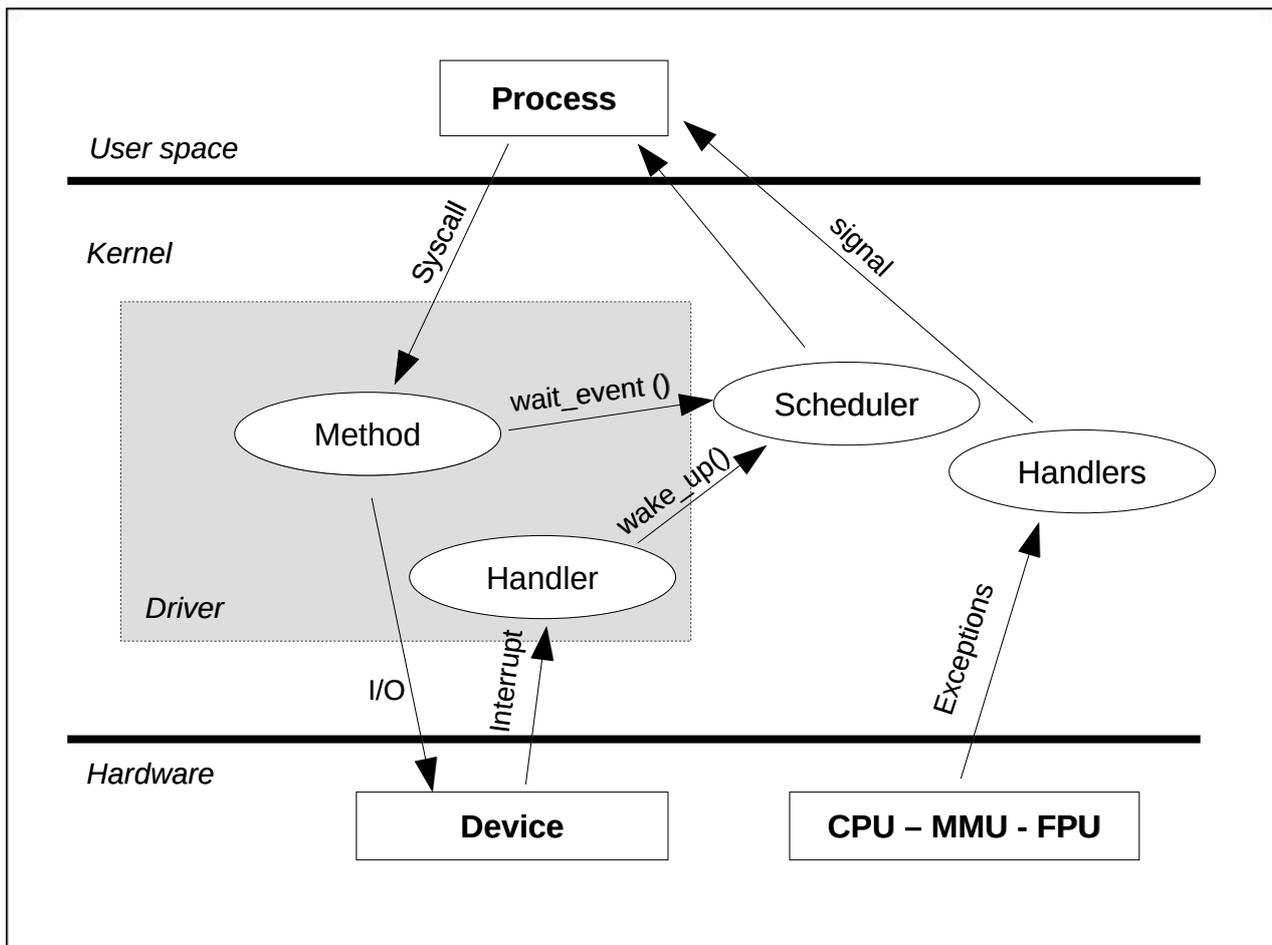
Linux est conçu pour s'exécuter sur des processeurs disposant de deux niveaux d'exécution (au moins) :

- un niveau **utilisateur** (*protégé*) pour l'exécution des applications ;
- un niveau **superviseur** (*réel*) pour l'exécution des opérations privilégiées – accès au matériel.

Les processus applicatifs s'exécutent en mode utilisateur (sécurité et performance), le noyau en mode superviseur (accès total à la mémoire et au matériel).

Le passage de l'espace utilisateur à l'espace noyau se fait par l'intermédiaire d'appels-système ou d'interruptions.

L'utilitaire `strace(1)` permet un suivi des appels-système d'un processus.



Lorsqu'un processus s'exécute (en mode utilisateur), la commutation en mode noyau peut se faire :

- à la demande du processus, lors d'un appel-système (interruption 0x80) ;
- à l'arrivée d'une interruption matérielle (par exemple contrôleur disque, ou *timer* système) ;
- à la levée d'une exception par le processeur (par exemple accès mémoire invalide) ;

En réaction à certaines conditions le noyau peut ajouter un signal dans la liste de ceux en attente pour un processus.

Avant tout retour d'interruption (matérielle ou logicielle), l'ordonnanceur vérifie si un autre processus doit être sélectionné pour exécution.

Les communications avec les périphériques sont possibles depuis les espaces noyau et utilisateur grâce aux instructions d'entrées-sorties.

Suivi d'un appel-système

```
#include <unistd.h>

int main(void)
{
    write(STDOUT_FILENO, "Hello !\n", 8);
    return 0;
}

$ gcc write_stdout.c -Wall -o write_stdout -g -static
$ objdump -d write_stdout | less
...
08048208 <main>:
...
8048229: c7 04 24 01 00 00 00    movl    $0x1, (%esp)
8048230: e8 fb 77 00 00         call   804fa30 <__libc_write>
...
0804fa30 <__libc_write>:
804fa3b: 8b 54 24 10           mov     0x10(%esp), %edx
804fa3f: 8b 4c 24 0c           mov     0xc(%esp), %ecx
804fa43: 8b 5c 24 08           mov     0x8(%esp), %ebx
804fa47: b8 04 00 00 00       mov     $0x4, %eax
804fa4c: cd 80                int     $0x80
804fa4e: 5b                   pop     %ebx
804fa5a: c3                   ret
...
```

Lorsqu'un processus effectue un appel-système, la routine correspondant dans la bibliothèque C invoque une interruption logicielle (0x80 sur un PC) après avoir rempli un registre (EAX en l'occurrence) avec une valeur représentant l'appel voulu (4 pour `write()`).

Voir aussi :

[BLAESS 2016] « *Interactions entre espace utilisateur, noyau et matériel* » Gnu/Linux Magazine Hors Série 87 « *Spécial Kernel* » - Diamond éditions, octobre 2016.
Disponible sur <https://www.blaess.fr/christophe/articles/>.

```
linux/include/asm-i386/mach-default/irq_vectors.h :
```

```
    [...]
#define SYSCALL_VECTOR  0x80
    [...]
```

```
linux/arch/i386/kernel/traps.c :
```

```
    [...]
void __init trap_init(void)
{
    [...]
    set_trap_gate(0, &divide_error);
    set_intr_gate(1, &debug);
    set_intr_gate(2, &nmi);
    [...]
    set_trap_gate(13, &general_protection);
    set_intr_gate(14, &page_fault);
    set_trap_gate(15, &spurious_interrupt_bug);
    set_trap_gate(16, &coprocessor_error);
    [...]
    set_system_gate(SYSCALL_VECTOR, &system_call);
    [...]
}
```

Au boot du kernel, une routine nommée `system_call()` est installée en tant que gestionnaire pour l'interruption (logicielle) nommée `SYSCALL_VECTOR`.

Ce numéro correspond à l'interruption `0x80`.

```
linux/arch/i386/kernel/entry.S:
```

```
[...]  
ENTRY(system_call)  
[...]  
    cml $(nr_syscalls), %eax  
    jae syscall_badsys  
syscall_call:  
    call *sys_call_table(,%eax,4)  
    movl %eax,PT_EAX(%esp)  
syscall_exit:  
[...]
```

```
linux/arch/i386/kernel/syscall_table.S:
```

```
ENTRY(sys_call_table)  
    .long sys_restart_syscall /* 0 */  
    .long sys_exit  
    .long sys_fork  
    .long sys_read  
    .long sys_write  
    .long sys_open           /* 5 */  
[...]
```

```
linux/fs/read_write.c:
```

```
asmlinkage ssize_t sys_write(unsigned int fd,  
                             const char __user * buf, size_t count)  
{  
    [...]  
}
```

La routine `system_call()` est écrite en assembleur, elle recherche la `eax`^{ième} entrée d'une table nommée `sys_call_table`. Il s'agit d'une table d'adresses sur 32 bits (4 octets).

La cinquième entrée de cette table (celle d'indice 4) est un pointeur sur une routine nommée `sys_write()`. On peut voir dans cette table tous les appels-système implémentés par le noyau.

La routine `sys_write()` ne dépend plus de l'architecture, elle est écrite en C.

Sur les noyaux récents, une macro permet de construire la ligne de prototype du noyau, et d'extraire facilement le type de ses paramètres (pour la documentation par exemple). On trouve donc les lignes suivantes :

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,  
                size_t, count)  
{  
    [...]  
}
```

Travaux pratiques : appels-système invoqués par une application

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

Exercice II-1 : appels-système invoqués par une application

Environnement du noyau

Contexte de tâche

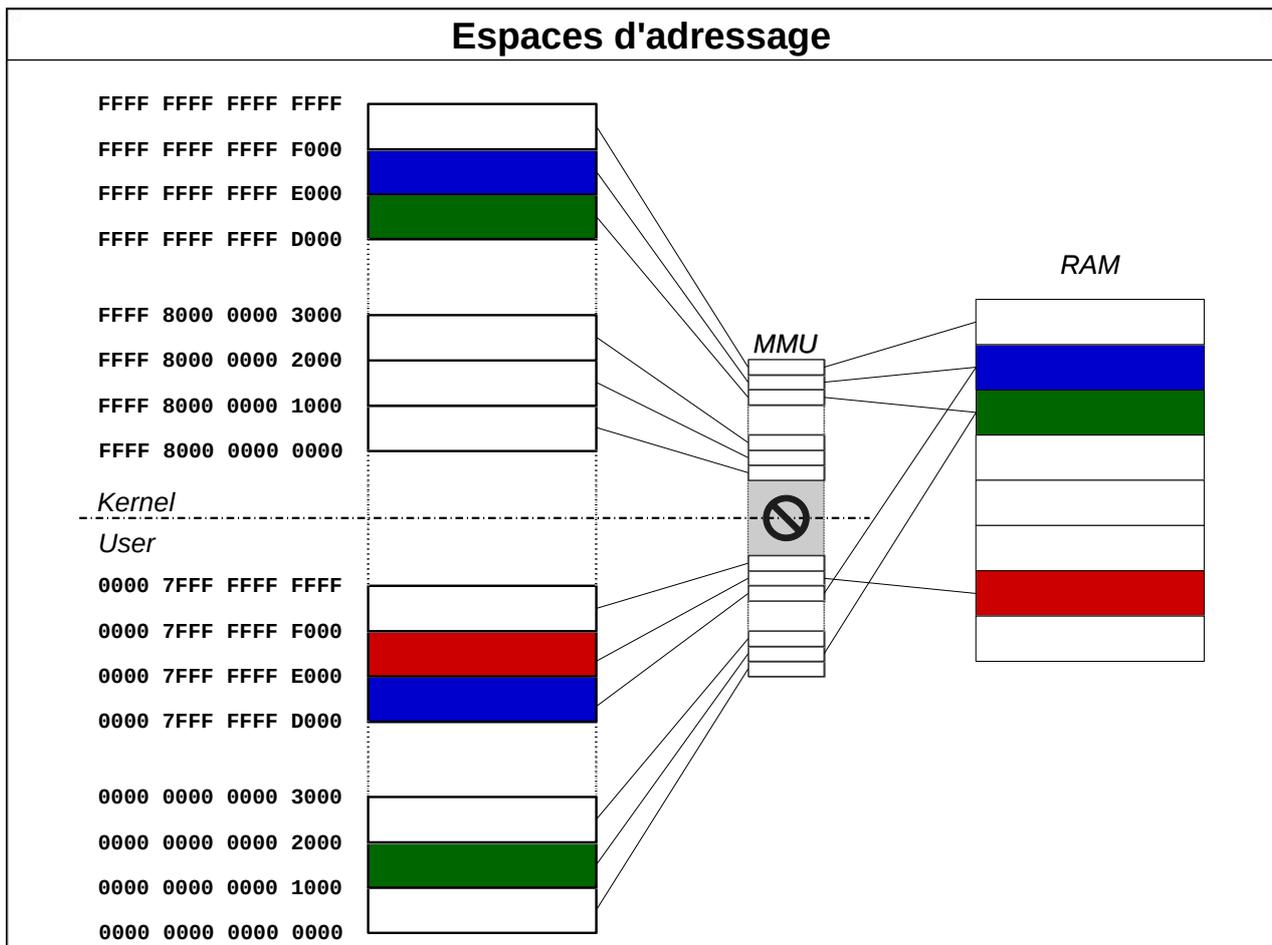
```
<linux/sched.h>
struct task_struct *current;
struct task_struct {
    ...
    void *stack;
    ...
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    ...
    cpumask_t cpus_mask;
    ...
    struct mm_struct *mm, *active_mm;
    ...
    int exit_state, exit_code, exit_signal;
    ...
    pid_t pid;
    ...
    struct task_struct *real_parent, *parent;
}
```

Durant un appel-système, le pointeur `current` représente toujours la tâche appelant.

Ce pointeur est défini sur la plupart des architectures comme une macro renvoyant le résultat d'une fonction. Ceci évite une tentative involontaire d'affectation de `current`.

A vous...

Chargez le module `example-II-01`, et déterminez qui est le processus `current`, ainsi que son père.



Un processus a une vision abstraite de la mémoire. Il voit un espace d'adressage continu dans lequel le système projette des fenêtres de RAM par le biais de la *MMU*. Le noyau a parfois une vision de l'ensemble de la RAM.

Sur les systèmes 32 bits, le noyau a généralement un espace d'adressage situé dans le giga-octet supérieur (0xC0000000 à 0xFFFFFFFF).

Sur les systèmes 64 bits, le noyau occupe les 128 To d'adressage à partir de 0xFFFF 8000 0000 0000.

Un pointeur en espace utilisateur n'est pas nécessairement valide en mode noyau.

```
<asm/uaccess.h>
<linux/uaccess.h>
```

Copies de blocs :

```
unsigned long copy_from_user (void *k_dest,
                               const void *u_src,
                               unsigned long n);

unsigned long copy_to_user   (void *u_dest,
                               const void *k_src,
                               unsigned long n);
```

Si `copy_from_user()` ou `copy_to_user()` renvoie une valeur non-nulle, l'appel-système doit échouer en retournant `-EFAULT`.

```
long strncpy_from_user(char *k_dst,
                       const char __user *u_src,
                       long max);
```

Système de fichiers /proc

Création d'entrées

```
<linux/proc_fs.h>
```

```
struct proc_dir_entry {  
    ...  
}
```

```
struct proc_dir_entry *proc_create (const char *name,  
                                     mode_t mode,  
                                     struct proc_dir_entry *parent,  
                                     struct proc_ops *props);  
  
struct proc_dir_entry *proc_mkdir (const char *name,  
                                     struct proc_dir_entry *parent);  
  
void proc_remove (struct proc_dir_entry *entry);
```

```
struct proc_ops {  
    ssize_t (*proc_read)(struct file *, char *, size_t, loff_t *);  
    ssize_t (*proc_write)(struct file*, const char *, size_t, loff_t*);  
    ...  
}
```

Un pointeur NULL en guise de parent dans `proc_create()` rattache directement à /proc.

A vous...

Chargez le module `example-II-02`, et observez les modifications dans /proc.

Callback de lecture

Routine invoquée lorsqu'un processus exécute une lecture depuis le fichier dans /proc :

```
cat /proc/nom_entree
```

```
ssize_t  read_callback (struct file *filp,  
                        char * __user buffer,  
                        size_t      size_max,  
                        l_off_t     *offset);
```

Attention, le *buffer* est dans l'espace utilisateur.

Il faut utiliser `copy_to_user ()` pour le remplir, sans dépasser `taille max_size`.

La fonction doit renvoyer le nombre d'octets inscrits dans la page.

A vous...

Chargez le module `exemple-II-03`, et vérifiez le contenu de la nouvelle entrée de /proc.

Le comportement est-il celui que vous attendiez ?

Sinon comment pourrait-on remédier au problème ?

Essayez le module `exemple-II-04`, fonctionne-t-il mieux ?

Lecture et écriture depuis un buffer

Lorsque les lectures et / ou écritures se font depuis un buffer, on peut implémenter facilement les appels-système avec :

```
ssize_t simple_read_from_buffer(void __user *u_buffer,  
                                size_t      max_buffer,  
                                loff_t      *offset,  
                                const void *src_buffer,  
                                size_t      buffer_len);  
  
ssize_t simple_write_to_buffer(void          *dst_buffer,  
                               size_t      buffer_max,  
                               loff_t      *offset,  
                               void __user *from_buffer,  
                               size_t      from_length);
```

Callback d'écriture

Routine invoquée lorsqu'un processus écrit dans le fichier de /proc :

```
echo 5 > /proc/nom_entree
```

```
ssize_t  write_callback (struct file *filp,  
                        const char * __user buffer,  
                        size_t      size,  
                        l_off_t     *offset);
```

Il faut copier le *buffer* depuis l'espace utilisateur, avec `copy_from_user()` avant de lire son contenu.

La fonction doit renvoyer le nombre d'octets lus dans le buffer (normalement `size`).

A vous...

Chargez le module `example-II-05`, et essayez d'écrire et de lire dans l'entrée de /proc correspondante.

Travaux pratiques : échange entre kernel et espace utilisateur.

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

Exercice II-2 : écriture depuis le kernel vers l'espace utilisateur

Exercice II-3 : échanges entre le kernel et l'espace utilisateur.