Chapitre 6

Programmation avec les appels système de Linux

Nous avons vu comment programmer des applications avec les instructions du microprocesseur. Nous n'avons pas vu cependant les instructions du microprocesseur utiles pour ce qu'on appelle la programmation système, en particulier pour programmer les entrées-sorties. Même avec ces instructions nouvelles, la programmation des entrées-sorties est longue et délicate. Mais, heureusement pour le programmeur d'applications, l'un des buts du système d'exploitation est de mettre en place un certain nombre de sous-programmes concernant ces points, qu'il suffit de réutiliser. Ces sous-programmes sont en général implémentés sous forme d'*appels système*.

Un **appel système** est un des procédés mis à la disposition du microprocesseur pour appeler un sous-programme. Nous étudierons comment concevoir un tel procédé dans la seconde partie, consacrée à la programmation système. Contentons-nous, dans cette partie, consacrée à la programmation des applications, de voir comment les utiliser pour concevoir un programme d'application.

Dans un système d'exploitation tel que *Linux*, les appels système permettent :

— de contrôler les périphériques ;
— de contrôler le système de fichiers ;
— de contrôler les processus.

6.1 Les appels système disponibles

Les appels système disponibles dépendent du système d'exploitation sur lequel on se trouve, et même de la version de celui-ci. La première chose à connaître est la liste des appels système disponibles sur le système d'exploitation pour lequel on programme l'application.

Liste des appels système.- Pour *Linux*, la liste des appels système disponibles se trouve dans le fichier :

`/usr/include/asm-x86_64/unistd.h`.

dont voici le début :

CONTREDIT LE NUMERO 0 UTILISE POUR SORTIR PROPREMENT

```
#ifndef _ASM_X86_64_UNISTD_H_
#define _ASM_X86_64_UNISTD_H_

#ifndef __SYSCALL
#define __SYSCALL(a,b)
#endif

/*
 * This file contains the system call numbers.
 *
 * Note: holes are not allowed.
 */

/* at least 8 syscall per cacheline */
#define __NR_read                               0
__SYSCALL(__NR_read, sys_read)
#define __NR_write                              1
__SYSCALL(__NR_write, sys_write)
#define __NR_open                               2
__SYSCALL(__NR_open, sys_open)
#define __NR_close                              3
__SYSCALL(__NR_close, sys_close)
```

Numéro d'un appel système.- Chaque appel système possède un **numéro**. Celui-ci est utilisé pour appeler la fonction correspondante. Ces numéros peuvent être différents d'un système d'exploitation à un autre, et même d'une version à une autre. En particulier, les numéros des appels système pour Linux 64 bits ne sont pas les mêmes que pour Linux 32 bits.

Arguments d'un appel système.- Un appel système possède en général des arguments. Ceux-ci sont documentés sur la page manuel correspondante.

6.2 Appel d'un appel système

Principe.- Avant d'effectuer un appel système, il faut placer son numéro et ses arguments là où il faut. Dans le cas d'un Linux 64 bits pour un microprocesseur x86-64, on place :

— son numéro dans le registre `%rax` ;

— les six premiers arguments dans les registres `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` et `%r9` ;

— les arguments suivants sur la pile.

L'appel système proprement dit s'effectue, soit grâce à l'instruction du microprocesseur :

```
syscall
```

si celle-ci est permise sur notre système (à vérifier, comme nous l'avons déjà vu), soit grâce à :

```
int $0x80
```

pour *Linux*, qui fonctionne toujours mais a l'inconvénient d'insister sur l'implémentation de *Linux*.

Le résultat d'un tel appel (s'il y en a un mais, en tous les cas au plus un) est placé dans `%rax`.

Exemple.- Nous avons déjà vu un exemple d'appel système dès notre premier programme

6.3 Préparation des arguments

Comment préparer les arguments ? Considérons par exemple `open()` et ce qui s'écrirait en langage C :

```
fd = open("foo.txt",O_RDWR|O_CREAT,0644)
```

On a vu que le numéro de cet appel est 2. Consultons la page manuel :

```
$ man 2 open
NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
          .................................
      int open(const char *pathname, int flags, mode_t mode);
          .................................
      open() returns a file descriptor, a small, non-negative integer
          .................................
      The parameter flags must include one of the following access  modes:
      O_RDONLY,  O_WRONLY,  or  O_RDWR.  These request opening the file
      read-only, write-only, or read/write, respectively.

      In  addition,  zero  or  more file creation flags and file status
      flags can be bitwise-or'd in flags.  The file creation flags are
      O_CREAT, O_EXCL, O_NOCTTY, and O_TRUNC.
          .................................
      The following symbolic constants are provided for mode:

      S_IRWXU  00700 user (file owner) has read, write and execute permission
      S_IRUSR  00400 user has read permission
      S_IWUSR  00200 user has write permission
      S_IRGRP  00040 group has read permission
      S_IWGRP  00020 group has write permission
      S_IXGRP  00010 group has execute permission
      S_IRWXO  00007 others have read, write and execute permission
      S_IROTH  00004 others have read permission
      S_IWOTH  00002 others have write permission
      S_IXOTH  00001 others have execute permission
        ...........................
```

On place donc le numéro de `open`, à savoir 2, dans le registre `%rax`.

Le premier argument est une chaîne de caractères terminée par le caractère `'\0'`, c'est-à-dire que nous devons placer dans `%rdi` son *adresse*, à savoir l'adresse de son premier caractère. Où la trouver ? Cela doit être precisé par celui qui appelle le programme contenant `open()`. Par exemple, la chaîne de caractères ainsi que son adresse peuvent se trouver sur la pile. Dans la suite nous supposons que le nom du fichier à ouvrir est tapé sur la ligne de commande.

Le deuxième argument est un indicateur. En langage C, on utilise la disjonction des macros `O_RDONLY` et `O_CREAT`. En langage d'assemblage nous devons connaître leurs valeurs *numériques*. Ces valeurs sont définies dans un certain fichier en-tête `*.h`. Trouvons-le :

```
$ grep -rl O_WRONLY /usr/include/
/usr/include/X11/Xw32defs.h
/usr/include/libsmbclient.h
/usr/include/bits/fcntl.h
/usr/include/asm-generic/fcntl.h
/usr/include/linux/fs.h
/usr/include/linux/smbno.h
/usr/include/alsa/hwdep.h
```

C'est apparemment le fichier `/usr/include/bits/fcntl.h` qui nous intéresse.

Question.- Pourquoi « apparemment » ?

Lisons ce fichier `/usr/include/bits/fcntl.h` :

```
.................
#ifndef _FCNTL_H
# error "Never use <bits/fcntl.h> directly; include <fcntl.h> instead."
#endif
#include <sys/types.h>
#ifdef __USE_GNU
# include <bits/uio.h>
#endif
/* open/fcntl - O_SYNC is only implemented on blocks devices and on files
   located on an ext2 file system */
#define O_ACCMODE          0003
#define O_RDONLY             00
#define O_WRONLY             01
#define O_RDWR               02
#define O_CREAT            0100 /* not fcntl */
#define O_EXCL             0200 /* not fcntl */
#define O_NOCTTY           0400 /* not fcntl */
#define O_TRUNC           01000 /* not fcntl */
.........
```

Notez que les valeurs des macros (y compris les « modes ») ci-dessus sont représentées en octal. Donc `O_RDWR|O_CREAT` devient en binaire :

```
00000000000000000000000000000010
00000000000000000000000001000000
--------------------------------
00000000000000000000000001000010
```

soit `0x42`. Un calcul analogue donne la valeur de « mode » en hexadécimal, soit `0x1a4`. **Vérifiez le calcul !**

Finalement nous pouvons écrire un programme simple, juste pour vérifier les raisonnements ci-dessus :

open-rdwr-0644.s

```
#
#       this one opens a file (name is given on the command line)
#       with O_RDWR|O_CREAT flags in 0644 mode
#
        .section .text
        .globl _start

_start:
        nop
        nop
        movq $2,%rax        # syscall number to %rax
        movq 16(%rsp),%rdi  # string address (argv[1]) to %rdi
        movq $0x42,%rsi     # the flags to %rsi
        movq $0x1a4,%rdx    # the mode to %tdx
        syscall
        movq %rax,%rdi      # the result (file descriptor) to %rdi
        movq $60,%rax
        syscall
        nop
        nop
```

```
$ as -o open-rdwr-0644.o open-rdwr-0644.s
$ ld open-rdwr-0644.o bitoa.o
$ ./a.out foo.txt
$ echo $?
3
$ ls -l
..............
-rw-r--r-- 1 vpg users    0 2012-04-11 00:53 foo.txt
..............
```

Utilisez la même technique pour les autres appels système.