Programmation Système

Introduction et Appels Systèmes

Emmanuel Bouthenot (emmanuel.bouthenot@u-bordeaux.fr)

Licence professionnelle ADSILLH - Université de Bordeaux

2017-2018

La Licence ADSILLH

- 18h de TD
- 18h de TP
- Contrôle Continu
- Un examen

La programmation Système

La Programmation Système c'est créer des programmes de bas niveau qui s'interfacent directement avec :

- le noyau du système d'exploitation
- les bibliothèques du cœur du système

Avoir de bonnes notions de programmation système permet d'être un meilleur développeur d'applications métiers :

- connaissances du fonctionnement du système et d'identification des problèmes de performances
- détection de problème de portabilité
- décider de travailler au niveau système quand cela devient nécessaire

Le système d'exploitation

Un système d'exploitation est un environnement logiciel qui fourni des services nécessaires à l'exécution de programmes pour un utilisateur final :

- Exécution de programmes
- Accès au matériel (stocker des données, écouter de la musique, ...)
- Accès au système de fichiers (lire, copier, supprimer un fichier)
- Accès à la mémoire (allocation, ...)
- Accès au réseau (se connecter à un serveur, ...)

Définition

Le noyau du système d'exploitation est la couche logicielle qui contrôle le matériel et fourni l'environnement dans lequel les programmes peuvent être exécutés.

Le(s) système(s) UNIX

Les systèmes UNIX sont une famille de systèmes d'exploitation multitâches et multi-utilisateurs basés sur le système **UNIX** originel crée en 1969 par Kenneth Thomson et Denis Ritchie au sein de Bell Labs Research.

Il en existe aujourd'hui de nombreuse variantes libres (GNU/Linux, FreeBSD, NetBSD, OpenBSD, Minix, ...) ou propriétaires comme MAC OS X.

Il existe un ensemble de standards réunis sous les normes **POSIX** et **SUS** (Single UNIX Specification) qui visent à unifier certains aspects de leur fonctionnement.

La Norme POSIX

The Portable Operating System Interface **POSIX** est une famille de normes techniques définie depuis 1988 par l'Institute of Electrical and Electronics Engineers (IEEE)

POSIX spécifie :

- les interfaces utilisateurs et les interfaces logicielles
- la ligne de commande standard et l'interface de script (Bourne shell)
- les services d'entrées/sorties de base (fichiers, terminaux, réseau)
- les attributs que doivent supporter les fichiers
- une interface de programmation standard

Comme l'IEEE vend très cher la documentation POSIX et ne permet pas sa publication sur Internet, certains se sont tournés vers le standard **Single UNIX Specification**.

Single Unix Specification

Single UNIX Specification (**SUS**) est un nom désignant un ensemble de spécifications permettant de certifier un système d'exploitation comme étant un Unix.

Le SUS est basé sur des travaux plus anciens de l'IEEE et de l'Open Group.

Plusieurs versions et standardisations: UNIX 95, UNIX 98 (SUSv2), UNIX 03 (SUSv4)

Les systèmes d'exploitation libres de type **Unix** ne sont pas certifiés SUS car le coût de la certification est très élevé. Cependant, la plupart des distributions Linux et FreeBSD une compatibilité, au moins partielle, avec le SUS et POSIX.

Linux Standard Base

La Linux Standard Base (**LSB**) est un projet commun de plusieurs distributions Linux sous la coupe du Free Standards Group (FSG) qui a pour but de concevoir et standardiser la structure interne des systèmes d'exploitation basés sur GNU/Linux.

La LSB spécifie :

- un ensemble de bibliothèques standards
- un nombre de commandes et d'utilitaires qui étendent le standard POSIX
- la structure de la hiérarchie du système de fichiers
- les différents run levels
- plusieurs extensions à X Window System

et entre autre la commande *lsb_release* qui quelque soit le système sur laquelle elle est utilisée retourne des informations propre au système de façon uniforme.

Architecture en couches

L'architecture des systèmes modernes se découpent en couches :

- n Machines / Architectures virtuelles ...
- 4 Couche applicative (applications métiers, frameworks, ...)
- 3 Couche système (appels système, ...)
- 2 Couche d'assemblage (language d'assemblage, ...)
- 1 Couche matérielle (firmware, microcode, ...)

Chaque couche dispose de facilités pour interagir avec les couches inférieures.

Quelle couche cibler ?

En général, plus la couche ciblée est proche du matériel meilleures sont les **performances**.

Il est assez courant de voir des routines écrites en assembleur pour avoir les meilleurs performances possibles (c'est le cas en particulier dans les pilotes, les jeux vidéos, dans les encodeurs/décodeurs de format vidéo, la cryptographie, . . .).

Plus la couche choisie est de bas niveau, plus la **portabilité** diminue.

Plus la couche ciblée est de haut niveau plus le code est **maintenable**.

Architecture d'un UNIX

- Architecture en couches avec un noyau UNIX en son cœur
- Toutes les couches ne sont pas strictement encapsulés
- L'implémentation des appels systèmes (**system calls** ou **syscalls**) est du ressort du noyau ('kernel-space code')

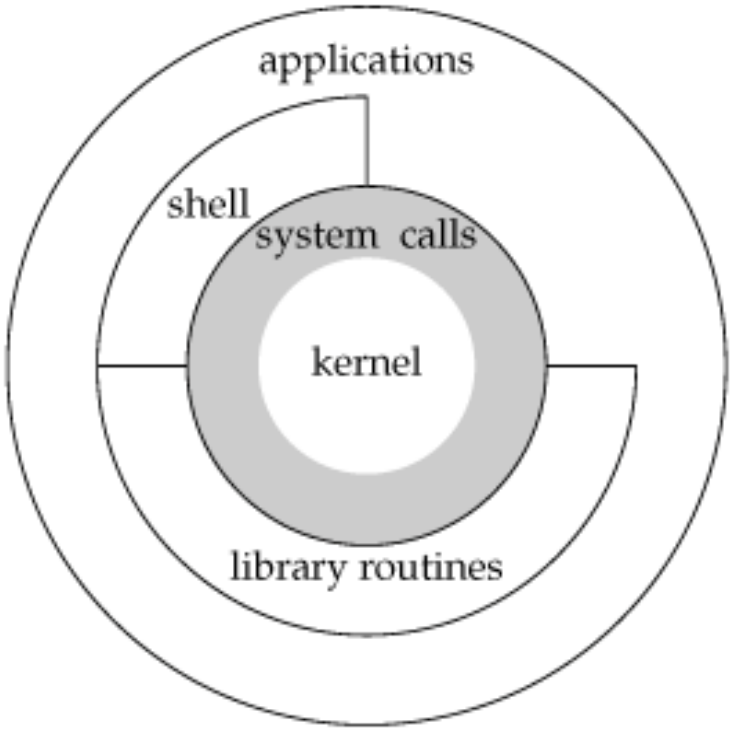

Figure: Couches du SE (**APUE** 1.1)

Le shell

Le (**shell**) est une application spécifique utilisée en mode interactif par les utilisateurs du système pour démarrer et controller des programmes.

Les (**applications**) sont les programmes utilisés par l'utilisateur final et qui le rende productif (Navigateur web, suite bureautique, lecteur multimédia, . . .).

La librairie standard C (libc)

Les appels systèmes fournissent l'interface (API) pour accéder aux services du noyau.

La librairie standard C (**library routines**) implémente les fonctionnalités de base nécessaires par la plupart des programmes :

- Entrées/Sorties tamponnées
- Allocation fine de la mémoire
- Gestion du temps
- Encapsulation des appels système du noyau (**stub**) avec une gestion des erreurs standardisée
- L'implémentation se fait en dehors du noyau ('user-space code')

Programmes et Processus

Définitions

- Un programme est un fichier exécutable qui réside dans le système de fichiers
- Un processus est une instance d'un programme en exécution

Plusieurs instances d'un même programme peuvent s'exécuter en même temps.

A chaque processus est associé :

- un identifiant de processus (**Process ID** ou **pid**)
- un espace d'adressage en mémoire vive pour stocker la pile, les données, ...

Descripteurs de fichiers

Définitions

- Un descripteur de fichier (**file descriptor** ou **fd**) est entier positif affecté par le noyau pour référencer un fichier utilisé par un programme en cours d'exécution

Un descripteur de fichier est unique au sein d'un même processus.

A chaque fois que le noyau ouvre ou créé un fichier pour un processus, il lui retourne le descripteur de fichier associé.

Les actions ultérieures sur le fichier requiert du processus qu'il passe ce descripteur de fichier au noyau.

Selon la philosophie d'UNIX ou 'tout est fichier', l'utilisation de descripteur de fichiers se retrouve pour les sockets, tubes, etc.

Un programme simple

Hello World en C:

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv) {
    printf("Hello World!\n");
    exit(EXIT_SUCCESS);
}
```

```
    # Installation d'un compilateur C,
    # de l'environnement de développement système
    apt-get install gcc libc6-dev
    # Compilation de note programme d'exemple
    gcc -Wall hello-world.c -o hello-world
    # Exécution du programme
    > ./hello-world
    Hello World!
```

strace

Pour comprendre les interactions entre le mode utilisateur, le mode noyau et le rôle joué par les appels système.

man 1 strace

strace - trace system calls and signals
strace [command [arg...]]
[...] strace runs the specified command until it exits. It intercepts and records the system calls which are called by a process [...]. The name of each system call, its arguments and its return value are printed on standard error [...]. strace is a useful diagnostic, instructional, and debugging tool. [...] Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. [...]

```
apt-get install strace
> strace ./hello-world
```

Exécution d'un programme

```
1   execve("./hello-world", ["./hello-world"], [/* 80 vars */]) = 0
2   brk(NULL)                                   = 0x1160000
3   access("/etc/ld.so.nohwcap", F_OK)          = -1 ENOENT (No such file or directory)
4   mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc9995c3000
5   access("/etc/ld.so.preload", R_OK)          = -1 ENOENT (No such file or directory)
6   open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7   fstat(3, {st_mode=S_IFREG|0644, st_size=119741, ...}) = 0
8   mmap(NULL, 119741, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc9995a5000
9   close(3)                                    = 0
10  access("/etc/ld.so.nohwcap", F_OK)          = -1 ENOENT (No such file or directory)
11  open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
12  read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0 \10\2\0\0\0\0\0"..., 832) = 832
13  fstat(3, {st_mode=S_IFREG|0755, st_size=1697504, ...}) = 0
14  mmap(NULL, 3803552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fc999001000
15  mprotect(0x7fc999198000, 2097152, PROT_NONE) = 0
16  mmap(0x7fc999398000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x197000) = 0x7f
17  mmap(0x7fc99939e000, 14752, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fc9993
18  close(3)                                    = 0
19  mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc9995a4000
20  mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc9995a3000
21  mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc9995a2000
22  arch_prctl(ARCH_SET_FS, 0x7fc9995a3700) = 0
23  mprotect(0x7fc999398000, 16384, PROT_READ) = 0
24  mprotect(0x600000, 4096, PROT_READ)         = 0
25  mprotect(0x7fc9995c5000, 4096, PROT_READ) = 0
26  munmap(0x7fc9995a5000, 119741)              = 0
27  fstat(1, {st_mode=S_IFREG|0644, st_size=1790, ...}) = 0
28  brk(NULL)                                   = 0x1160000
29  brk(0x1182000)                              = 0x1182000
30  write(1, "Hello World!\n", 13Hello World!
31  )           = 13
32  exit_group(0)                               = ?
33  +++ exited with 0 +++
```

Analyse de la trace

- exécution du programme par le shell qui invoque l'appel système **execve**
- le noyau lit le code du programme binaire **hello-world**
- ...
- le programme invoque la fonction **printf** de la libc
- la libc invoque l'appel système **write**
- le noyau affiche le texte sur la console
- ...
- le programme invoque **exit**
- la libc invoque l'appel système **exit_group**
- le noyau termine le programme

Les pages de manuel

```
# Installation des pages de manuel (section 2 et 3)
apt-get install manpages-dev manpages-fr-dev
```

Extrait de *man man*

Le tableau ci-dessous indique le numéro des sections de manuel ainsi que le type de pages qu'elles contiennent.

- **1** Programmes exécutables ou commandes de l'interpréteur de commandes (shell)
- **2** Appels système (fonctions fournies par le noyau)
- **3** Appels de bibliothèque (fonctions fournies par les bibliothèques des programmes)
- **4** Fichiers spéciaux (situés généralement dans /dev)
- **5** Formats des fichiers et conventions. Par exemple /etc/passwd
- **6** Jeux
- **7** Divers (y compris les macropaquets et les conventions)
- **8** Commandes de gestion du système (généralement réservées au superutilisateur)
- **9** Sous-programmes du noyau [hors standard]

Les pages de manuel (suite)

```
# La page de manuel des appels système (section 2)
man 2 syscalls
# La page de manuel de write (section 2)
man 2 write
# La page de manuel de printf (section 3)
man 3 printf
```

Les appels systèmes

Un appel système est un point d'entrée dans le noyau qui permet à un programme de demander au noyau de faire une action très précise.
Exemple d'invocation d'un appel système sur une architecture x86-32 :

1. le programme invoque l'appel système au travers d'une fonction surcouche de la libc
2. la fonction surcouche de la libc transcrit les arguments qui lui sont passés et les préparent en les mettant dans des registres spécifiques pour le noyau
3. la libc place le numéro de l'appel système dans un registre spécifique (%eax)
4. la libc invoque l'instruction **trap** (int 0x80) qui pour effet de basculer le processeur en mode noyau et exécuté le code

Les appels systèmes (suite)

5. en réponse le noyau invoque sa fonction **system_call()** (après validation des arguments, du numéro de l'appel système, ...)
6. le noyau exécute l'action demandée puis retour le résultat à l'appel **system_call()**
7. qui retourne le résultat à la fonction surcouche de la libc
8. en cas d'erreur, la libc positionne la variable globale **errno** avec le résultat
9. la libc retourne le résultat au programme appelant

Étapes de l'exécution d'un appel système

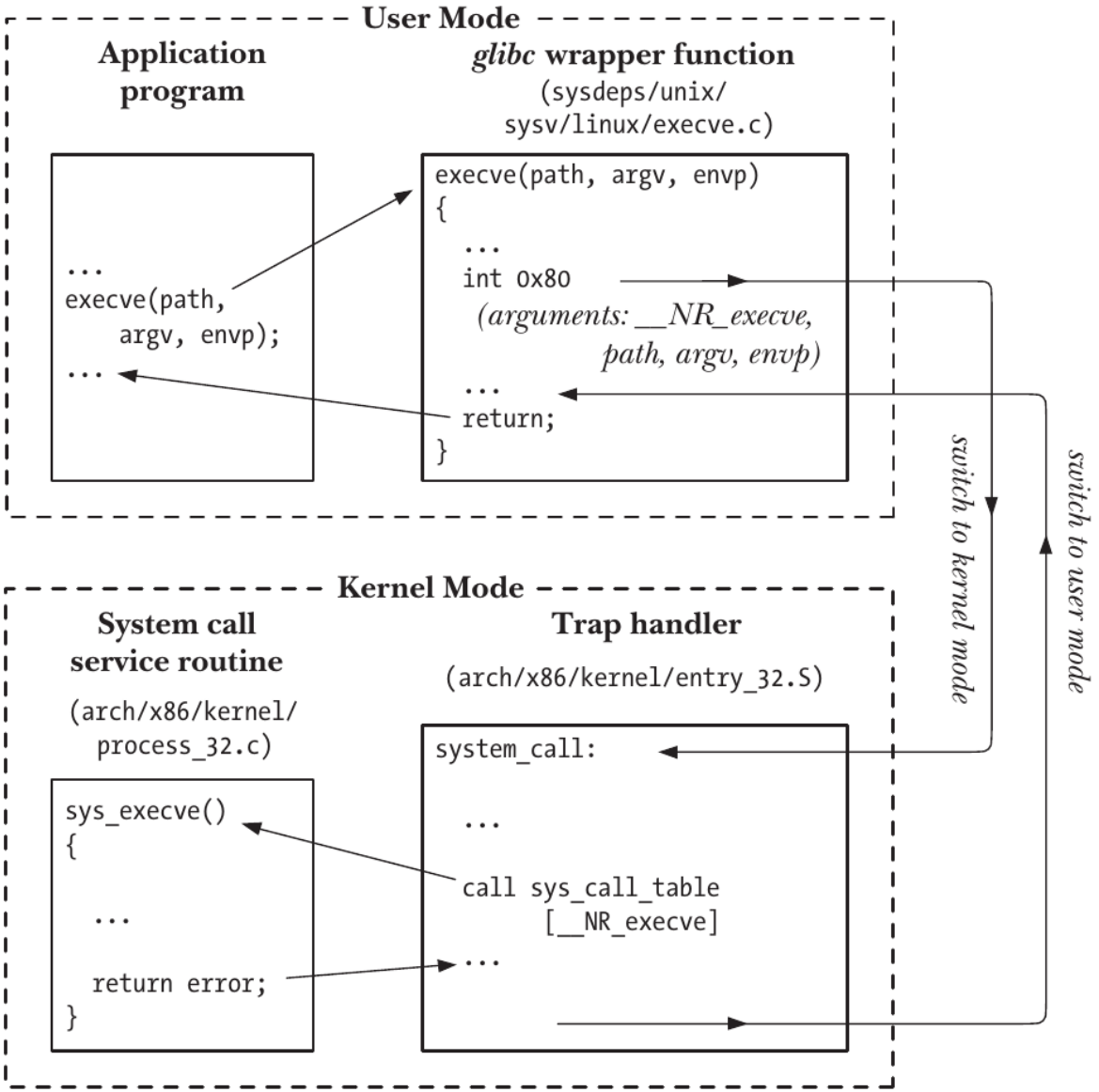

Figure: (**TLPI** 3-1)

errno

Provoquons une erreur :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

int main (int argc, char **argv) {
    int result = write(3, "It Works!\n", 10);
    if (result == -1) {
        perror("Error");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

```
    # Compilation
    gcc -Wall errno.c -o errno
    # Exécution
    > ./errno
    Error: Bad file descriptor
    > echo $?
    1
```

Bibliographie

[**APUE**] Advanced Programming in the UNIX Environment.
W. Richard Stevens and Stephen A. Rago.
Addison-Wesley Professional, 2005.

[**TLPI**] The Linux Programming Interface.
Michael Kerisk.
No Starch Press, 2010.