Programmation système : introduction

Juliusz Chroboczek

7 Octobre 2009

1 Accès au matériel

Dans un ordinateur, le processeur communique non seulement avec la mémoire principale, qui contient le code à exécuter et les données sur lesquelles celui-ci opère, mais aussi avec des *périphériques de stockage de masse*, ou mémoire secondaire, et des *périphériques d'entrées sorties*. Le programmeur doit avoir accès à des facilités qui lui permettent d'accéder à ceux-ci.

1.1 Programmation sur matériel nu

Il est bien-sûr possible d'accéder aux périphériques directement à partir du code utilisateur ; on parle alors de *programmation sur matériel nu*. Cette approche présente plusieurs désavantages :

- le code dépend d'une connaissance intime des détails du matériel, et le changement d'un périphérique force la reécriture d'une quantité potentiellement grande de code ;
- le code n'est pas modulaire : l'accès au matériel est mélangé à la logique du programme elle-même.

De nos jours, la programmation sur matériel nu n'est plus praticable sur les ordinateurs classiques, du fait de l'existence de matériel complexe, hétérogène, et partagé entre différentes applications (c'est notamment le cas des disques). Elle se pratique encore dans les systèmes dits *embarqués*, par exemple votre four à micro-ondes. Cependant, elle est de plus en plus rare : même votre téléphone portable a un système d'exploitation.

1.2 HAL et moniteur

Afin d'éviter de rendre nos programmes dépendants des détails du matériel et afin de séparer la logique du programme de l'accès au matériel, il est naturel de mettre le code dit *de bas niveau* (proche du matériel) dans une bibliothèque séparée. Une telle bibliothèque est communément appelée *HAL*, acronyme développé, selon les auteurs, en *Hardware Abstraction Layer* ou en *Hardware Access Library*. (De nos jours, la HAL est elle-même modulaire — elle est constituée de l'ensemble des *pilotes de périphérique (drivers)*.)

Moniteur Si la même HAL est utilisée par tous les programmes, elle est souvent *résidente*, c'est à dire chargée une seule fois lors du lancement *(boot))* du système et présente en mémoire à tout moment.

Un *moniteur* est composé d'une HAL résidente et d'une interface utilisateur permettant notamment de charger des programmes en mémoire et de les exécuter, et parfois aussi de contrôler manuellement les périphériques (par exemple de renommer des fichiers sur disque, ou de manipuler l'imprimante). Un exemple bien connu de moniteur est le système *MS-DOS*.

1.3 Système d'exploitation

Un moniteur permet d'isoler le code utilisateur des dépendances matérielles ; cependant, l'utilisation du moniteur n'est pas enforcée, et il est possible de le contourner et d'accéder directement au matériel. Ceci a pour conséquences qu'un moniteur n'est pas suffisant pour s'assurer qu'aucune contrainte de sécurité ou de modularité n'est violée par le code utilisateur, ce qui est important notamment sur une machine partagée entre plusieurs utilisateurs.

Un *système d'exploitation*, ou *noyau du système d'exploitation (operating system kernel)*, est une couche située entre le logiciel utilisateur et le matériel dont l'utilisation est obligatoire pour tout processus voulant accéder au matériel.

Protection matérielle Ce qui précède implique que le noyau peut empêcher le code utilisateur d'accéder directement au matériel. Ceci est généralement implémenté en distinguant au niveau matériel entre deux types de code : le code dit *privilégié* ou *code noyau*, qui a le droit d'accéder au matériel, et le code dit *non-privilégié* ou *code utilisateur*, qui n'en a pas le droit.

Lorsqu'un programme utilisateur essaie d'accéder (délibérément, ou, comme c'est le plus souvent le cas, par erreur) au matériel ou à des parties de la mémoire qui lui sont interdites, le matériel de *protection* passe la main au noyau qui, typiquement, tue le processus coupable[1].

Appel système L'invocation de code privilégié par du code non-privilégié se fait au moyen d'un *appel système*. Un appel système est semblable à un appel de fonction, mais effectue en plus deux changements de niveau de privilège : une transition vers le mode privilégié lors de l'appel, et une transition vers le mode non-privilégié lors du retour de l'appel système.

De ce fait, un appel système est lent, typiquement entre 1 000 et 10 000 fois plus lent qu'un appel de fonction sur du matériel moderne.

1.4 Virtualisation et abstraction du matériel

La vision donnée par le système des périphériques n'est pas complètement uniforme. Dans certains cas, le système se contente de vérifier les privilèges du processus avant de lui donner accès au « vrai » matériel ; on parle alors de simple *médiation*.

Dans d'autres cas, le système présente au processus utilisateur des périphériques *virtuels*, qui se comportent comme de vrais périphériques, mais ne correspondent pas directement à la réalité ; par exemple, il peuvent être plus nombreux que les périphériques réels. C'est par exemple souvent le cas de la mémoire et des consoles. On parle alors de *virtualisation* du matériel.

1 Unix, par exemple, envoie un signal `SIGSEGV` ou `SIGBUS` au processus.

Enfin, il arrive que le système présente des périphériques *abstraits*, de plus haut niveau que les périphériques réels. C'est généralement le cas des disques (qui sont présentés comme une arborescence de fichiers plutôt qu'une suite de secteurs) et des interfaces réseau. On parle alors d'*abstraction*.

2 Le système Unix

Unix, une famille de systèmes d'exploitation basée sur le système Unix des Bell Labs d'AT&T, développé dans les années 70, servira de base à ce cours. La famille Unix a aujourd'hui de nombreux représentants, aussi bien libres (Linux, FreeBSD, NetBSD, Minix etc.) que moins libres (Solaris) et propriétaires (HP/UX, Mac OS X etc.).

2.1 Fonctions *stub*

Le mécanisme exact d'appel système dépend du matériel. Pour cette raison, chaque système Unix inclut dans la librairie C standard (`libc`) un certain nombre de fonctions dites *stubs* ou *wrappers*, dont chacune a le rôle d'invoquer un appel système. Par exemple, l'appel système `time`(2) est invoqué par la fonction

```
int time(int *);
```

déclarée dans le fichier d'entête `<time.h>`.

Le manuel Unix (accessible à l'aide de la commande `man`) distingue entre les *stubs* d'appels système, qu'il documente dans la section 2, et les « vraies » fonctions, qu'il documente dans la section 3.

Convention d'appel et variable `errno` La librairie standard définit une variable globale appelée `errno` qui sert à communiquer les résultats d'erreur entre les fonctions *stub* et le code utilisateur. Lorsqu'un appel système réussit, une fonction *stub* retourne un entier positif ou nul[2] ; lorsqu'il échoue, elle stocke le numéro de l'erreur dans la variable `errno`, et retourne une valeur strictement négative[3].

La variabe `errno` ainsi que des constantes symboliques pour les codes d'erreurs sont déclarées dans le fichier d'entête `<errno.h>`. Pour afficher les messages d'erreurs, on peut utiliser les fonctions `strerror`(3) et `perror`(3) (déclarées dans `<string.h>` et `<stdio.h>` respectivement).

Exécution d'une fonction *stub* Une fonction *stub* effectue les actions suivantes :

- stocker les arguments de l'appel dans les bons registres ;
- invoquer l'appel système ;
- interpréter la valeur de retour et, si besoin, positionner la variable `errno`.

2 Il y a quelques exceptions à cette convention.

3 Ce mécanisme, qui sert à compenser l'absence d'exceptions en C, pose certains problèmes pour l'écriture de code fiable en présence de *threads* multiples.

2.2 Le standard POSIX

Au début des années 1990, Unix s'était morcelé en un certain nombre de systèmes semblables mais incompatibles. L'écriture d'un programme portable entre les différents dialectes d'Unix était devenue un exercice inutilement excitant.

Le standard *IEEE 1003*, communément appelé *POSIX (Portable Operating System Interface)* vise à créer un ensemble commun de fonctions disponibles sur tous les systèmes Unix.

Les interfaces POSIX restent en grande partie compatibles avec les interfaces Unix traditionnelles, mais remplacent les types concrets (tels que `int`) par des types dits « opaques » dont la définition précise n'est pas spécifiée par POSIX, et est donc laissée à la discrétion de l'implémentation. Par exemple, POSIX définit l'appel `time` comme

```
time_t time(time_t *);
```

où le choix du type concret pour lequel `time_t` est un alias est laissé à l'implémenteur. Typiquement, `time_t` est un alias pour `int`, défini par

```
typedef int time_t;
```

mais rien n'empêche une implémentation de faire, par exemple,

```
typedef long int time_t;
```

POSIX considère la différence entre une fonction et un appel système comme étant un détail de l'implémentation, et ne différencie donc pas entre les sections 2 et 3 du manuel.

Tous les Unix modernes, libres ou propriétaires, implémentent le standard POSIX.

La version 2003 du standard POSIX est identique à la *« Single Unix Speci cation »* version 3, qui est disponible en ligne à l'adresse `http://www.unix.org/`.