Operating Systems

October 31, 2019

Contents

1 Base des Systèmes d'exploitations

Contact :

- bernard.vankerm@henallux.be
- bernard.vankerm@belfius.be
- bernard.vankerm@skynet.be

2 Operating Systems : Advanced Course

2.1 1.Introduction

Au démarage de la machine, le **BIOS** initialise, verifie, ... puis passe la main au **Noyau** qui prendra le relais pour gérer les accès au matériel.

On sépare la mémoire vive physique en deux régions disjointes (l'une pour le noyau et l'autre pour les applications)

Pour répartir la charge équitablement pour tous les éléments du système, on fait appel à un **Scheduler**.

Les appels systèmes sont le principal moyen pour une application linux de comuniquer avec le noyau. (ex open, read et write pour la gestion de fichiers en unix)

2.1.1 Manuel

Le manuel est un guide d'utilisation des commandes.

man <nom_de_commande>

2.1.2 Variables d'envoronnement

Une variable d'environnement est une variable qui peut être vue comme visible par tous les programmes et qui influe sur le comportement normal d'une applicaion. (ex: langue du système, path actuel de l'invite de commande)

n.b. : Linux est Case Sensitive

n.b. : on peut utiliser le TAB pour auto compléter une ligne de commande.

Les jokers et Séparateurs on utilise des '/' comme séparateurs pour désigner un chemin d'accès.

Les '*'désigne plusieurs caractères. tandis que'?' n'en désigne qu'un seul.

Compilation de programmes en C On utilise GCC (Gnu Compiler collection)

```
gcc hello.c -o hello
```

on peut ensuite exécuter le programme avec un './'

```
./hello
```

2.2 2.Création de processus

2.2.1 Rappels sur les librairies

La pluspart des librairies sont des bibliothèques de fonctions (ex: stdlib, stdio, ...)

2.2.2 mode noyau et mode utilisateur

Pour communiquer avec le noyau, il existe deux modes de fonctionnement (le mode Noyau et le mode utilisateur)

> Un appel système consiste en une interruption logicielle (system trap) qui a pour rôle de changer de mode d'exécution pour passer du mode utilisateur au mode noyau, de récupérer les paramètres et de vérifier la validité de l'appel, de lancer l'exécution de la fonction demandée, de récupérer la (les) valeur(s) de retour et de retourner au programme appelant en passant à nouveau au mode utilisateur.

Les appels systèmes ne peuvent êtres effectuées que par le biais de procédures spécifiques. Certains appels ne peuvent être faits que par des programmes lancés par *root*.

2.2.3 Table des Processus

Une **table des Processus** est une liste de liens vers des **blocs de contrôles de processus**(bcp). On associe à chaque processus un bcp.

Un bcp contient l'**espace d'adressage** de l'application (segment text = code de l'app, var. globales, variables locales, pile et stack), **le contexte** (ensemble des registres processeur (ex : next instruction)) et **autres informations**.

2.2.4 Création de processus

Les appels systèmes permettent notamment la création et l'arrêt des processus. Un processus père peut créer des processus fils. Pour créer un processus on utilise **fork**.

nb Librairie : unistd.h

Crée un nouveau processus (fils) semblable au processus courant (père). Le contexte des deux processus est exactement le même mis à part leur identifiant (pid) et l'identifiant de leur père respectif (ppid). La valeur renvoyée n'est pas la même pour le fils et pour le père, ce qui permet de les différencier. Trois cas sont à envisager :

1. le processus père, celui qui fait le fork , renvoie le numéro du fils (pid).
2. le processus fils qui est créé commence son exécution dès la fin du fork et renvoie 0.
3. en cas d'erreur, le processus fils n'est pas créé et le père reçoit une valeur négative.

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

void fils1 (void) {
    printf("Debut du Fils 1\n");
```

```
    // Suite du Fils 1
    printf("Fin du Fils 1\n");
    exit(EXIT_SUCCESS);
}
int main (void) {
    int pid1;
    printf("Debut du Pere\n");
    pid1 = fork();
    if (pid1 <0) { //
        perror("Erreur lors de la creation (fork) du Fils 1");
        exit(EXIT_FAILURE);
    } // else inutile car exit
    if (pid1 == 0) {
        fils1();
    }
    //suite du pere
    printf("Fin du Pere\n");
    exit(EXIT_SUCCESS);
}
```

toute modification des variable avant le fork affecte le père et le fils. Tandis que toute moification postérieur au fork sera répercuté uniquement sur l'espace d'adressage propre à chaque processus.

2.2.5 Perror

librairie : unistd.h

Affiche un message sur la sortie d'erreur standard (stderr), décrivant la dernière erreur rencontrée durant un appel système ou une fonction de bibliothèque. Si str n'est pas NULL et si *str n'est pas un octet nul, la chaine de caractères str est imprimée, suivie de ': 'ou d'un blanc, puis du message correspondant à l'erreur, et enfin d'un saut de ligne.

```
perror ("fopen a genere l'erreur suivante");
```

2.2.6 Exit

livrairie : stdlib.h

Termine l'exécution du processus en renvoyant au père la valeur passée en paramètre(0 ok, autre = bad). Cet appel système a les effets suivants :

- tous les file descriptors appartenant au processus en question sont fermées.
- les processus fils du processus terminé deviennent zombies et sont adoptés par le processus init.
- le processus père reçoit un signal SIGCHLD.

2.2.7 getpid et getppid

librairie : stdlib.h

```
pid_t getpid (void);
```

Retourne l'identifiant du processus appelant. Cette Fonction réussit toujours.

```
pid_t getppid (void);
```

Retourne l'identifiant du processus père qui l'appelle (foncion réussit toujours)

2.2.8 wait

librairie : stdlib;

```
pid_t wait (int * status);
```

permets à un programme père d'attendre qu'un processus fils se termine. Status est utilisé pour réccupérer la valeur de fin du processus fils qui vient de se terminer.

> attention, un fils qui se termine mais qui n'a pas été attendu devient un zombie. lors de sa zombification, le noyau conserve des informations minimales pour permettre au pêre de l'attendre plus tard et obtenir les informations sur le fils. Tant que le zombie n'est

pas effacé du système par un wait, il prendra un emplacement dans la table des processus et si cette table est pleine, il sera impossible de créer de nouveaux processus.

2.2.9 sleep

librairie : unistd.h

```
void sleep(int time);
```

force un processus à s'endormir pour time secondes.

2.2.10 Effectuer une commande shell

librairie : stdlib.h

```
int system (const char * cmd);
```

exemple :

```
reponse = system("ls -l");
```

Permet de lancer une ligne de commande (sh) depuis un programme. En réalité, un processus fils est créé, exécutant sh avec comme entrée la commande cmd passée en argument. Cette fonction fait donc un fork suivi d'un exec dans la partie fils. Le père fait quant à lui un wait et renvoie la valeur de l'exit status du fils.

Stdout : utilise un buffer qui affiche quand : une ligne, flush, buffer de 1024 bytes plein. donc si deux processus se partagent la sortie std, il peut y avoir des collisions

3 chapitre 3 : Commandes de gestion de processus

3.1 Processus

- **ps** : Afficher l'état des processus en cours
- **pstree** : Afficher l'arbre des processus. La racine est init

3.2 Flèches

Une flèche est une forme limitée de communication entre processus utilisée par les systèmes de type Unix et ceux respectant les standards POSIX. Il s'agit d'une notification asynchrone envoyée à un processus pour lui signaler l'apparition d'un événement. Quand une flèche est envoyée à un processus, le système d'exploitation interrompt l'exécution normale de celui-ci. Si le processus possède une routine de traitement pour la flèche reçue, il lance son exécution. Dans le cas contraire, il exécute la routine de traitement des flèches par défaut.1

manuel sur les flèches :

```
man 7 signal
```

- **kill** : Envoyer une flèche au(x) processus dont on précise le(s) pid (SIGTERM si on ne précise pas quelle autre flèche utiliser)

  ctrl-c kill le programme, ctrl-z le stop. pour kill un programme mis sur pause par ctrl-z, on peut faire un kill %1 (1 etant le numero dans les parenthèses retourné par ctrl-z

- **killall** : Envoyer une flèche à un processus dont on précise le nom
- **top** : liste dynamique des processus actifs en arrière plan

3.3 Faire tourner un programme dans le background

4 Appels systèmes pour la gestion des processus

4.1 waitpid

Suspend l'exécution du processus appelant jusqu'à ce que le fils spécifié par son pid ait changé d'état.

4.1.1 librairies

- sys/wait.h
- sys/types.h

4.1.2 prototype

```
pid_t waitpid (pid_t pid, int *status, int options);
```

4.2 execv

Cette fonction active l'exécutable application à la place du processus courant. Les paramètres éventuels sont transmis sous forme d'un tableau de pointeurs sur des chaines de caractères, le dernier étant NULL.

4.2.1 prototype

```
int execv (const char *application, char *const argv[]);
```

4.2.2 Ouverture de fichier

voir labo 4 pour des exemples

5 Synchronisation des Processus