Introduction Système Exploitation

Processus

Abdelkader Gouaïch

BUT 2, IUT Montpellier

2022

Section 1

Introduction

Sous-section 1

Définitions : Processus, Programme et Thread

Programme et Processus

Le concept de processus est une abstraction fondamentale dans un système d'exploitation et en particulier un système d'exploitation de la famille Unix.

Avant de nous lancer dans les détails techniques, il est important pour nous de donner quelques définitions et préciser certaines notions.

Il nous faut distinguer entre le notion de *programme* et celle de *processus*.

En système d'exploitation, le terme programme désigne un fichier exécutable qui résulte de la phase de compilation. Ce fichier est au format binaire, il s'agit donc d'un tableau d'octets sauvegardé sur le disque.

Comme pour n'importe quel autre fichier, nous allons associer à ce fichier des droits et un couple propriétaire/groupe.

Processus et thread

Si nous avons défini le processus comme une activité, alors le *thread* est le moteur de cette activité. C'est donc l'unité d'exécution dans un processus qui permet d'interpréter les instructions du programme sur une CPU et de changer l'état de la mémoire.

Chaque thread dispose :

- d'un espace mémoire pour les variables des fonctions (appelé la *pile*)
- d'un état du processeur ce qui correspond concrètement aux valeurs des *registres* du processeur. Un registre est simplement un mot mémoire (un tableau de n octets)
- d'un pointeur (adresse) vers la prochaine instruction à exécuter

La programmation *monothread* par définition nous propose un seul thread d'exécution. La correspondance entre un processus et un thread est donc 1 à 1. Il nous arrivera alors de confondre le processus et son thread principal.

Le process ID

Le système d'exploitation associe à chaque nouveau processus un identifiant unique.

Cet identifiant, appelé *Process ID ou PID*, est simplement un entier qui est unique à un moment donné.

Cela veut dire, que le PID d'un processus terminé peut être réutilisé mais que deux processus actifs n'auront jamais le même identifiant durant leur exécution.

Nous pouvons identifier quelques processus particuliers :

- le process *idle*, de PID 0, est un processus qui le processeur exécute quand qu'aucune activité n'est en cours d'exécution
- le processus *init*, de PID 1, est le premier processus créé par la séquence de *boot*. le processus init est responsable de la création de tous les autres processus.

Vous pouvez consulter le programme de init sous certains répertoires

Le PID

Le PID est un entier de type `pid_t`. Ce type est défini dans le fichier header `<sys/types.h>`. Il s'agit d'un renommage du type `int`. Avec cette technique, nous gardons les valeurs entières mais nous informons le compilateur que ce sont des identifiants de processus.

Voici la signature de la fonction `C` qui permet d'avoir le PID du processus en cours d'exécution :

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

Pour récupérer le PID du processus parent, nous pouvons utiliser la fonction :

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```

Sous-section 2

La hiérarchie de processus

Relation Père Fils

Un processus peut engendrer un nouveau processus. Le processus à l'origine de la création est appelé le père et le processus engendré est appelé le fils

Le process Init

Chaque processus dans le système est nécessairement généré par un autre. La seule exception concerne le processus init (PID=1) qui est créé ex nihilo par la séquence de bootstrap.

Représentation arborescente

La relation père/fils peut se représenter comme un arbre hiérarchique :

- la racine de l'arbre est le processus init
- chaque processus ne possède qu'un seul père connu par son PPID

Gestion des droits

Un processus est associé à un utilisateur et à un groupe d'utilisateurs. L'utilisateur et le groupe sont représentés par des entiers notés UID et GID Un processus fils hérite de l'UID et du GID de son père à la création Le UID et GID sont utilisés pour les gérés les droits d'accès aux ressources (accès aux fichiers par exemple)

Sous-section 3

Création de nouveaux processus

Introduction

Unix distingue entre le chargement d'un programme en mémoire pour son exécution et la création d'un nouveau processus par duplication.

Dans le premier cas, l'image du nouveau programme est chargée et *remplace* l'ancienne image. L'exécution du processus continue ainsi avec la nouvelle image.

Nous remarquons que le PID du processus n'est pas modifié. Il s'agit du même processus qui continue son exécution avec un nouveau programme.

Pour traiter les cas de chargement d'image nous utilisons la famille des fonctions `exec`.

Dans le deuxième cas, nous gardons la même image du programme exécutable mais un *nouveau* processus fils est créé avec un PID différent.

Appel système exec

La famille exec regroupe plusieurs fonctions qui jouent le même rôle : remplacer l'image du programme avec une nouvelle.

Le processus va continuer son exécution avec la nouvelle image du programme et prenant en compte de nouveaux arguments.

execl

```
#include <unistd.h>
int execl (const char *path, const char *arg, ...);
```

`execl()` va remplacer l'image du programme utilisée par le processus courant avec une nouvelle qui se chargée à partir du chemin indiqué par `path`.
Les nouveaux arguments du processus sont indiqués à partir de `arg`.
Le paramètre `arg` est donc le premier paramètre qui représente le nom que nous souhaitons donner au processus.
Les autres arguments sont indiqués à sa suite comme des valeurs séparées par des virgules.
Nous devons utiliser la constante NULL pour terminer la liste de ces

Appel système fork()

`fork` littéralement veut dire une bifurcation nous permet de créer deux chemins d'exécution à partir du même programme.

Voici sa signature :

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

fork() va créer un nouveau processus avec les caractéristiques suivantes :

- Il possède exactement la même image du programme
- le ppid est identique au pid du père
- les signaux en attente sont réinitialisés
- les fichiers ouverts sont hérités

Exemple

```
pid_t pid;
```

Combinaison fork/exec

Nous pouvons maintenant utiliser les deux primitives pour créer de nouveaux processus qui vont exécuter de nouveaux programmes.

`fork()` va prendre en charge la création du nouveau processus. Ensuite, un des processus, le fils par exemple, va utiliser exec pour charger une nouvelle image du programme.

Exemple

```
pid_t pid;
pid = fork ();
if (pid == -1) perror ("fork"); /* fils */
if (!pid)
{
    const char *args[] = { "ls", NULL };
    int ret;
    ret = execv ("/bin/ls", args);
    if (ret == -1) {
       perror ("execv");
       exit (EXIT_FAILURE);
    }
}
```

Sous-section 4

Terminer un processus

exit()

Pour mettre fin correctement à un processus en cours d'exécution, nous pouvons utiliser la fonction `exit()` qui a pour signature :

```
#include <stdlib.h>
void exit (int status);
```

L'appel à exit va permettre au kernel de réaliser les opérations suivantes :

- appel des fonctions, triggers, qui sont en attente de l'événement de terminaison du processus
- écriture des données I/O mises dans la mémoire tampons, on parle alors de *flush* des données I/O
- destruction de tous les fichiers temporaires
- libération des ressources allouées par le noyau : mémoire, fichiers ouverts, sémaphores.
- notification au parent de la fin de son processus fils

Enregistrement d'un trigger sur terminaison

Nous pouvons enregistrer une fonction qui sera appelée lors de la fin du processus.

Cette fonction est un *trigger* : c'est à dire que son appel sera déclenchée par un événement particulier, en l'occurrence le 'exit' du processus.

Voici la signature de la fonction d'enregistrement du trigger :

```
#include <stdlib.h>
int atexit (void (*function)(void));
```

Le type `void (*function)(void)` est un type de fonction. Une fonction de ce type ne prend pas de paramètres (void) et ne retourne pas de résultat (void).

Toute fonction qui correspond à cette signature est éligible pour l'enregistrement par `atexit`.

Le trigger est appelé si `exit()` est explicitement appelé ou bien si

Attendre la fin des processus fils

Un processus parent peut être intéressé par l'état de terminaison de son processus fils.

Nous nous trouvons alors dans une situation où le fils va terminer son traitement mais nous devons garder certaines informations pour le père.

Cet état post terminaison du processus fils est appelé l'état *zombie*.

Une fois que le parent a lu les informations, le processus zombie est retiré de la liste des processus.

Pour lire consulter l'état de terminaison du processus fils, nous allons utiliser la fonction wait() :

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

L'appel à cette fonction est bloquant. Le processus appelant, qui est

Sous-section 5

La gestion des droits

Utilisateurs et Groupes

Unix est un système d'exploitation multi utilisateurs. Chaque utilisateur est identifié par un entier unique *UID*.

Il existe également des groupes et chaque groupe est identifié par un entier unique le *GID*.

La correspondance entre les identifiants, les entiers, et les noms symboliques, chaine de caractères, se trouve dans les fichiers `/etc/passwd` et `/etc/group`

Les identifiants d'utilisateur et groupe associés au processus sont très importants dans la mesure où ils vont permettre à ce processus d'accéder à certaines ressources ou de réaliser certaines opérations sensibles.

Nous allons voir que nous pouvons, temporairement, demander des droits supplémentaires pour certaines opérations, mais ensuite nous allons essayer de toujours revenir au principe des droits minimaux : donner un le minimum des droits pour les processus.

Les différents User IDs

Il existe 4 users ids :

- real user id
- effective user id
- saved user id

real user id :

il correspond au UID de l'utilisateur qui a *lancé* le processus. Cette valeur est copiée lors d'un `fork` et ne change pas avec `exec`.
En général, le processus de `login`, qui est le shell, va mettre le real user id à la valeur de l'utilisateur qui vient de se connecter. Ensuite, tous les processus issus du `fork/exec` de ce shell vont hériter ce cette valeur. Ceci explique, que tous les processus lancés depuis votre shell vous appartiennent.
Seul le root peut modifier cette valeur et les processus utilisateurs ne peuvent pas la modifier.

effective user id

Sous-section 6

Les sessions et les groupes de processus

Introduction

Chaque processus est membre d'un groupe appelé 'process group'.

Comme son nom l'indique, il s'agit simplement d'un agrégat de processus qui vont accomplir une tâche complexe nécessitant l'usage de plusieurs processus indépendants. Nous appelons cette tâche un 'job'.

Un avantage de regrouper les processus dans un groupe est de pouvoir communiquer plus simplement avec l'ensemble des processus impliqués dans le job.

Par exemple, nous pouvons envoyer un seul signal au groupe et ce signal sera par la suite transmis à l'ensemble des processus du groupe. Nous pourrions par exemple terminer l'ensemble du job avec un seul signal `Ctrl^C`.

Un groupe de processus possède un identifiant (pgid) et un processus leader. Le gpid va correspondre au PID du leader.

Les sessions

Un *terminal* est l'interface par laquelle l'utilisateur interagit avec le système d'exploitation en saisissant des commandes et en consultant les résultats des commandes en texte.

Une *session* est une collection de groupes de processus.

Le but d'une session est de faciliter l'interaction entre le terminal et les groupes de processus de l'utilisateur.

Quand un utilisateur se connecte, le processus de login va créer une nouvelle *session*. Cette session va contenir uniquement le processus de login qui sera son leader. Par conséquent, l'identifiant de la session sera le pid du processus de login.

Le but des sessions est d'organiser les activités d'un utilisateur connecté au système.

Une session va associer pour l'utilisateur un terminal de contrôle qui lui permet de saisir les commandes et de lire les résultats.