# Programmation Système
## Travaux Pratiques (3), Licence 2 Informatique
## Gestion de processus (1/2)

Les exercices suivants ont pour but de vous familiariser avec les appels système de base relatifs à la gestion des processus.
Il vous est recommandé de consulter les pages man de ces primitives pour de plus amples informations sur leur syntaxe, leur sémantique et les éventuelles options qu'elles offrent.

Les instructions des exercices se repèrent par des icônes, qui sont les suivantes :

Information — Information concernant l'usage ou le rôle d'une commande, par exemple. Dans certains cas, il s'agit d'une information sur ce que vous êtes en train de faire ou sur ce qui se passe.

Exemple — Exemple d'utilisation.

Contrôle — Vérifier le résultat d'une (ou plusieurs) action(s).

Action — Effectuer la ou les action(s) décrite(s).

Question — Questions auxquelles vous devez répondre.

De plus, un `texte en police courier` correspond soit à une sortie écran soit à des noms spécifiques (menus, fenêtre, icône, processus, commandes…).
Un **texte en police times gras** correspond à ce que l'utilisateur doit introduire comme valeur de paramètre, ou encore, est utilisé pour attirer l'attention de l'utilisateur.

## Caractéristiques d'un processus

### Identité d'un processus et identité de son processus père

Les appels `getpid()` et `getppid()` permettent à un processus d'accéder à son identité ainsi qu'à celle de son père.

Écrire un programme qui affiche l'identité du processus correspondant ainsi que l'identité du père.

À quel processus correspond l'identité du père ?


Mohamed Taghelit , taghelit@univ-tours.fr

# Liens du processus avec les utilisateurs

## Propriétaires réel et effectif

Les primitives `getuid()` et `geteuid()` renvoient, respectivement, le propriétaire réel et le propriétaire effectif d'un processus.

Les exercices suivants doivent être accomplis en coordination avec un autre groupe.

Écrire un programme `prog_exemple.c` qui affiche les propriétaires réel et effectif du processus correspondant. Le fichier exécutable (`prog_exemple`) doit avoir des droits d'exécution pour tout le monde.

Modifier le programme précédent de façon qu'il ouvre un fichier texte existant `fichier_exemple` (passé en paramètre) dont vous êtes seul à y avoir des droits d'accès (aucun droit pour les autres).

Vérifier que l'exécution se déroule bien.

Lancer le programme `prog_exemple` de l'un des autres groupes avec comme paramètre leur propre fichier texte `fichier_exemple`.

Assurez-vous d'avoir les droits d'accès à leur répertoire de travail (répertoire contenant l'exécutable `prog_exemple` et le fichier `fichier_exemple`, ainsi que tous les répertoires intermédiaires).

Quel résultat obtenez-vous de la précédente exécution ?

Demander à l'autre groupe de positionner le `set-uid` de l'exécutable et réitérer l'étape précédente.

Quel résultat obtenez-vous ? Quelles sont vos conclusions ?

## Groupes réel et effectif

Les primitives `getgid()` et `getegid()` renvoient, respectivement, le groupe réel et le groupe effectif du processus.

Réitérer les exercices précédents en utilisant les primitives `getgid()` et `getegid()` en lieu et place des primitives `getuid()` et `geteuid()`.

L'attribution (ou la restriction) des droits, pour les fichiers exécutable et texte, doit porter sur le groupe. Considérer également, dans ce cas, le `setgid bit`.

Quel résultats obtenez-vous ? Est-ce ce à quoi vous vous attendiez ? Quelle est votre explication ?

### Modification des propriétaires réel et effectif

Les primitives `setuid()` et `seteuid()` permettent de modifier, respectivement, le propriétaire réel et le propriétaire effectif du processus.

Tester l'utilisation des deux primitives précédentes.

Quel résultat obtenez-vous ? Expliquer !

### Modification des groupes réel et effectif

Les primitives `setgid()` et `setegid()` permettent de modifier, respectivement, le groupe réel et le groupe effectif du processus.

Tester l'utilisation des deux primitives précédentes.

Quel résultat obtenez-vous ? Expliquer !

## Les temps CPU

La primitive `times()` renvoie les temps CPU consommés dans les deux modes (utilisateur et noyau) par le processus ainsi que par ses fils terminés.

Écrire un programme qui, après un certain calcul interne et avant de se terminer, affiche les temps CPU consommés.

Les temps retournés dans les différents champs de la structure `tms` sont exprimés en nombre de clics d'horloge. Pour les convertir en des durées exprimées en secondes, il suffit de les diviser par le nombre de clics d'horloge par seconde (`sysconf(_SC_CLK_TCK)`).

## Création de processus

### La primitive `fork()`

La primitive `fork()` crée un nouveau processus qui est un "clone" du processus père (appelant) au moment de l'appel (création).

Écrire un programme dont le processus correspondant crée un processus fils à l'aide de l'appel de la primitive `fork()`. Le processus père doit afficher son `pid` ainsi que celui de son fils. Le processus fils affiche, quant à lui, son `pid` ainsi que celui de son père (`ppid`).

Quelle est la valeur rendue par `getppid()` au niveau du processus fils ? Pourquoi ?

Le processus fils hérite de son père, entre autres, du même environnement ainsi que de la même table des fichiers ouverts.

Modifier le programme précédent de façon à vérifier les affirmations précédentes.

Le processus fils hérite d'une copie des données du père. Cette copie n'est réalisée que lors d'accès en écriture (*copy on write*) et est telle que :

- l'ensemble des adresses valides des deux processus est le même,
- les valeurs des données dans les deux processus sont les mêmes au retour de l'appel du `fork()`,
- les données physiques sont différentes.

Modifier le programme précédent de façon à vérifier les affirmations précédentes.

Pour l'exercice précédent, il suffit de montrer qu'une variable donnée a la même adresse virtuelle pour le père et le fils et que la modification de sa valeur par le père (respectivement, le fils) n'est pas répercutée chez le fils (respectivement, le père).

**La primitive `vfork()`**

La primitive `vfork()` permet de créer un processus fils sans recopier les données. Le processus fils travaille sur les données physiques du père. Afin de régler les problèmes d'accès aux données entre les deux processus, le processus père est bloqué jusqu'à ce que le processus fils soit se termine, soit se recouvre.

Vérifier les affirmations précédentes.

Procéder de la même façon que pour l'exercice précédent (utilisation d'une variable qui sera modifiée).

## La synchronisation père/fils

**Les primitives `wait()` et `waitpid()`**

Les primitives `wait()` et `waitpid()` permettent de synchroniser le processus sur la terminaison de ses descendants. Elles permettent également de récupérer des informations concernant cette terminaison.

Écrire un programme dont le processus correspondant se termine sans attendre la terminaison d'un processus fils qu'il aura au préalable créé.

Que devient le processus fils une fois terminé ?

Modifier le programme précédent de façon que le processus père attende cette fois-ci la terminaison de son fils.

Que devient le processus fils une fois terminé ?

Modifier le programme de façon à utiliser à chaque fois une valeur différente (`<-1`, `-1`, `0` et `>0`) de `pid` (paramètre de `waitpid()`). Tester également l'option non bloquante de l'appel.

Quelles sont vos déductions ?

Vérifier que le processus père récupère bien la valeur **`n`** lorsque son fils se termine par `exit(n)`, `_exit(n)` ou `return(n)`.

## La terminaison de processus

### Les primitives `exit()` et `_exit()`

Les primitives `exit()` et `_exit()` entraînent la terminaison d'un processus, à la demande de ce dernier.

`exit()` diffère de `_exit()`, entre autres, par le fait qu'elle vide les tampons associés aux fichiers ouverts.

Vérifier, par l'écriture d'un programme, l'affirmation précédente.

`exit()` diffère également de `_exit()` par le fait qu'elle appelle toutes les fonctions dont la demande d'exécution en cas de terminaison a été formulée au moyen de la fonction standard `int atexit(void (*fonction)(void));`.

Écrire un programme qui formule la demande d'exécution de trois fonctions (`func1()`, `func2()` et `func3()`) en cas de terminaison. Le processus doit se terminer sur un appel à `exit()`.

Vérifier, en remplaçant `exit()` par `_exit()` dans le programme précédent, que les fonctions (`func1()`, `func2()` et `func3()`) ne sont pas exécutées à la terminaison du processus.

## Session et groupe de processus

### Les primitives `getpgrp()` et `setpgid()`

La primitive `pid_t getpgrp(void);` renvoie le `PID` du leader du groupe de l'appelant.

Écrire un programme qui affiche le `PID` du processus leader du groupe auquel il appartient.

Un processus appartient au même groupe que celui auquel appartient son père et ceci jusqu'à ce qu'il demande à en changer.

Modifier le programme précédent de façon qu'il crée un processus fils qui affiche aussi le `PID` du processus leader du groupe auquel il appartient.

Vérifier si les deux processus précédents appartiennent au même groupe ?

Appartiennent-ils au même groupe auquel appartient le shell à partir duquel vous avez demandé l'exécution ?

Lorsqu'un processus leader d'un groupe de processus se termine, un signal `SIGHUP` est émis à chaque processus membre du groupe.

Modifier le programme précédent de façon que le processus fils ne se termine pas immédiatement (utiliser `pause()` à la fin du code du fils).

Que devient le processus fils une fois que le processus père (leader de son groupe) s'est terminé ?

La primitive `pid_t setpgid(pid_t pid, pid_t pgid);` rattache le processus `pid` au groupe `pgid`.

Modifier le programme précédent de façon que le processus fils change de groupe par rapport à son père.

Que devient le processus fils une fois que le processus père (leader de son groupe) s'est terminé ?

**Les primitives `getsid()` et `setsid()`**

La primitive `pid_t getsid(pid_t pid);` retourne le `PID` du leader de la session auquel appartient le processus `pid`.

La primitive `pid_t setsid(void);` permet à l'appelant de créer une nouvelle session dont il sera le leader. Cette session comportera un seul groupe dont le processus appelant est le leader et également l'unique membre.

Écrire un programme qui vérifie les affirmations précédentes.

**La primitive `tcgetpgrp()`**

La primitive `pid_t tcgetpgrp(int desc);` retourne le `PID` du leader du groupe de processus en premier plan du terminal de descripteur `desc`.

La nouvelle session créée à partir de l'appel à la primitive `pid_t setsid(void);` n'a pas de terminal de contrôle.

Écrire un programme qui vérifie l'affirmation précédente.