

Travaux Pratiques 1 : Gestion des processus

Objectifs : création de processus, héritage, processus zombies, synchronisation des processus.

Exercice 1 : Création de processus

Un appel à *fork()* par un processus, appelé processus-père, demande à UNIX de mettre en activité un nouveau processus (appelé processus-fils) qui est une copie conforme du processus courant, pour la plupart de ses attributs.

Cette fonction rend :

- -1 en cas d'échec,
- Dans le processus-fils, *fork()* renvoie 0
- Alors que dans le processus-père, elle renvoie le PID de la copie.

La syntaxe est la suivante :

```
#include <unistd.h>
pid_t p = fork();
```

Quelques primitives utiles :

- *getpid()* et *getppid()* : Un processus a accès respectivement à son PID et à celui de son père par l'intermédiaire respectivement des primitives *getpid()* et *getppid()*.
- *exit()* : La primitive *exit(int status)* met fin au processus qui l'a émis, avec un code de retour *status*.
 - Si le processus a des fils lorsque *exit()* est appelé, ils ne sont pas modifiés, mais quand le processus-père prend fin, le nom de leur processus père est changé en 1.
 - Par convention, un code de retour égal à 0 signifie que le processus s'est terminé correctement, et un code non nul (généralement 1) signifie qu'une erreur s'est produite.
 - Le père du processus qui effectue un *exit()* reçoit son code retour à travers un appel à *wait()*.

- 1) Écrire un programme C qui crée deux fils, l'un affiche les entiers de 1 à 50, l'autre de 51 à 100.

Exercice 2 : Héritage

Quelques primitives utiles :

- *getuid()* : (propriétaire réel) retourne le numéro d'identification de l'utilisateur à qui appartient le processus. Il en est de même pour le propriétaire effectif via la primitive *geteuid()*.
- *getgid()* : (propriétaire réel) retourne le numéro d'identification du groupe à qui appartient le processus. Il en est de même pour le propriétaire effectif via la primitive *getegid()*.
- *getcwd(char * buf, unsigned long taille)* : retourne le chemin absolu du répertoire de travail courant.
- *sleep(int sec)* : permet de suspendre l'exécution d'un processus pendant un intervalle de temps passé en paramètre.
- *nice(int incr)* : À chaque processus est associée une valeur du paramètre système *nice*, entre -20 et 19, et qui indique son niveau de priorité (l'importance relative des processus).
 - Par défaut, cette valeur est égale à 0.
 - Plus cette valeur est faible, plus le processus est prioritaire pour accéder à la ressource CPU de la machine.
- *times(struct tms *buffer)* : retourne les temps CPU de traitement actuels dans *struct tms* pointée par *buf*. Il existe 4 champs dans la structure *struct tms* :

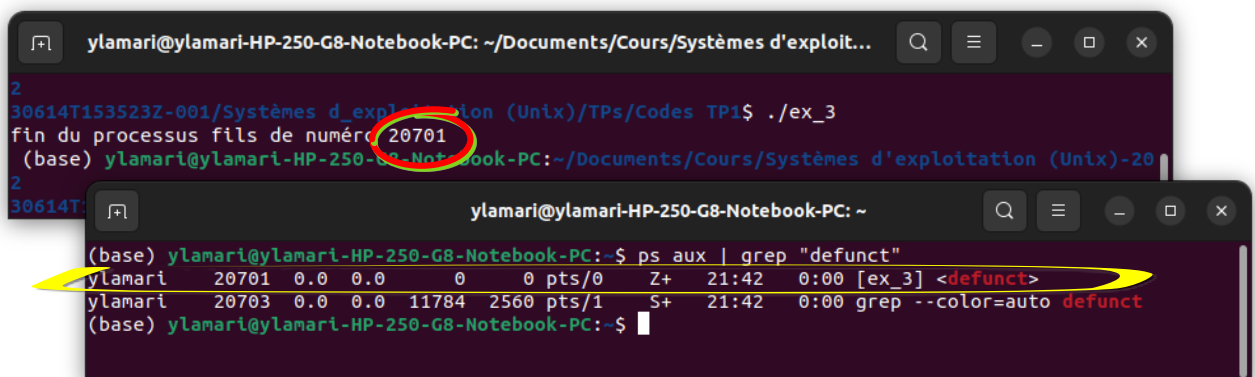
```
struct tms {  
    clock_t tms_utime; /* temps utilisateur */  
    clock_t tms_stime; /* temps système */  
    clock_t tms_cutime; /* temps utilisateur des fils */  
    clock_t tms_cstime; /* temps système des fils */  
};
```

- Le champ *tms_utime* contient le temps CPU passé à exécuter les instructions du processus appelant.
 - Le champ *tms_stime* contient le temps CPU passé dans le système lors de l'exécution des tâches pour le compte du processus appelant.
 - Le champ *tms_cutime* contient la somme des valeurs *tms_utime* et *tms_cutime* pour tous les processus-fils terminés en attente.
 - Le champ *tms_cstime* contient la somme des valeurs *tms_stime* et *tms_cstime* pour tous les processus-fils terminés attendus.
- 1) Écrire un programme C qui crée un fils, en suite affiche les caractéristiques du père et du fils en appelant les fonctions : *getuid()*, *geteuid()*, *getegid()*, *getcwd()*, *nice()*, et la structure *tms*. Que remarquez-vous ?
 - 2) Écrire un programme C qui crée deux variables *m* (locale) et *n* (globale) ayant comme valeurs initiales 1000 chacune, ensuite qui crée un fils qui héritera au début ces valeurs. Il faut faire en sorte que les processus fils et père modifient les valeurs de *m* et *n*. Afficher les adresses et les valeurs des variables *m* et *n* avant et après la modification dans les deux processus. Que remarquez-vous ?

Exercice 3 : Processus Zombies

Lorsque le fils se termine, si son père ne l'attend pas, le fils passe à l'état *defunct* (ou zombi) dans la table des processus.

- 1) Écrire un programme C qui crée un processus fils Zombie durant 30 secondes. Vérifier l'état du processus fils à l'aide de la commande *ps* dans un deuxième terminal.



```
(base) ylamari@ylamari-HP-250-G8-Notebook-PC: ~/Documents/Cours/Systèmes d'exploit...  
2  
30614T153523Z-001/Systèmes d'exploitation (Unix)/TPs/Codes TP1$ ./ex_3  
fin du processus fils de numéro 20701  
(base) ylamari@ylamari-HP-250-G8-Notebook-PC: ~/Documents/Cours/Systèmes d'exploitation (Unix)-20  
2  
30614T  
(base) ylamari@ylamari-HP-250-G8-Notebook-PC: ~  
(base) ylamari@ylamari-HP-250-G8-Notebook-PC:~$ ps aux | grep "defunct"  
ylamari 20701 0.0 0.0 0 0 pts/0 Z+ 21:42 0:00 [ex_3] <defunct>  
ylamari 20703 0.0 0.0 11784 2560 pts/1 S+ 21:42 0:00 grep --color=auto defunct  
(base) ylamari@ylamari-HP-250-G8-Notebook-PC:~$
```

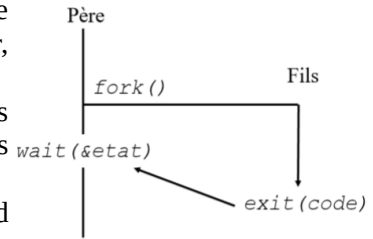
Exercice 4 : Synchronisation père-fils

Quelques primitives utiles :

- *wait(int *status)* : permet à un processus appelant de suspendre son exécution en attente de recevoir un signal de fin de l'un de ses fils.

Les principales actions de *wait()* sont :

- Recherche de processus fils dans la table des processus : si le processus n'a pas de fils, la primitive `wait()` renvoie une erreur, sinon elle incrémente un compteur.
 - S'il y a un zombie, il récupère les paramètres nécessaires (accounting) et libère l'entrée correspondante de la table des processus.
 - S'il y a un fils, mais pas zombie, alors le processus se suspend (état endormi) en attente d'un signal.
 - Lorsque le signal est reçu, la cause du décès est stockée dans la variable 'status'. Trois cas à distinguer :
 - processus stoppé,
 - processus terminé volontairement par `exit()`,
 - processus terminé à la suite d'un signal.
 - Enfin la primitive `wait()` permet de récupérer le PID du processus fils : `pid = wait(&status)`.
- `waitpid(pid t pid, int *status, int options)` : permet de tester la terminaison d'un processus particulier, dont on connaît le PID.
- 1) Écrire un programme C qui permet de simuler le cas normal, c'est-à-dire, lorsque le processus-père attend la terminaison du processus-fils. Le programme consiste à créer un processus-fils à l'aide de la primitive `fork()`. Ensuite, le fils affichera son identité à l'aide de la primitive `getpid()` et se terminera en utilisant la primitive `exit(int statut)` avec un code de retour status. Côté processus-père, il attend la terminaison du fils à l'aide de la primitive `wait()`. Une fois le fils est terminé, le père se réveille et affiche la valeur de retour du `wait()` (PID) et le status.
 - 2) Écrire un programme C qui permet de simuler le cas d'un Zombie, c'est-à-dire, lorsque le processus-père n'attend pas son fils et qui reste toujours en cours d'exécution après la mort de son fils. Vérifier l'état du processus fils à l'aide de la commande `ps` dans un deuxième terminal.
 - 3) Écrire un programme C qui permet de simuler le cas d'un Zombie momentanément, c'est-à-dire, lorsque le processus-père reçoit le signal de terminaison de son fils et n'exécute la primitive `wait()` qu'après la primitive `sleep()`. Le processus fils reste zombie momentanément, le temps pour le père de recevoir les différentes informations. Vérifier l'état du processus fils à l'aide de la commande `ps` dans un deuxième terminal.



A noter : Codes d'état du processus (colonne STAT de la commande `ps`)

- D Sommeil ininterrompu (généralement E/S) un état bloqué. Le processus attend une condition matérielle et ne peut gérer aucun signal.
- R Le processus en cours d'exécution est en cours d'exécution ou prêt à être exécuté
- S Veille interrompue (attente de la fin d'un événement) a État bloqué d'un processus et attente d'un événement ou d'un signal provenant d'un autre processus.
- T Arrêté, soit par un signal de contrôle de tâche, soit parce qu'il est en cours de traçage. Le processus est arrêté et peut être redémarré par un autre processus.
- Z Processus défunt (« zombie »), terminé mais non récupéré par son processus parent terminé, mais les informations sont toujours là dans la table des processus.
- < haute priorité
- N N faible priorité
- L a des pages verrouillées en mémoire
- s est un animateur de session
- l est multi-thread
- + est dans le groupe de processus de premier plan.
- I Inactif. Il est utilisé pour les threads du noyau qui utilisent l'état `TASK_IDLE` lorsqu'ils sont inactifs.