Système d’exploitation LINUX TP N 2 - Processus

Nom et Prénom : Souha Bousselmi

Sommaire :

1. Utilisation des commandes du shell
2. Introduction aux exercices en C
3. Création de processus (fonction fork)
   1. Fonctions utilisées
   2. Exercice 1
   3. Exercice 2
   4. Synchronisation de processus père et fils (mécanisme wait/exit)
      1. Fonctions utilisées
      2. Mécanisme wait/exit
   5. Père et fils exécutent des programmes différents
      1. Fonction utilisée
      2. Fonctionnement de exec
   6. Exercices

# Utilisation des commandes du shell :

Les exercices suivants s'attachent à montrer les différences entre processus et utilisateur et à mettre en évidence quelques points importants du système Unix.

1. Pour constater la différence utilisateur/processus, on essaiera la séquence de commandes suivantes :

*who (liste des utilisateurs connectés)*



*who am i (qui suis-je ...)*



*ps -l (liste des processus attachés au terminal)*



*ps -axl (liste des processus présents sur le système)*



*ou*

*ps -ael suivant le système utilisé*

On remarquera les différents processus :

* init de pid 1 et le swapper-scheduler de pid 0.
* les démons (notés ?),
* ceux qui déroulent un shell (sh, csh ou ksh),

1. Envoi d'une commande en arrière plan (option &) : *sleep n* met le processus qui l'exécute en pause pendant n secondes.

Envoyer la séquence suivante et observer la filiation des processus :

*ps -l* (liste des processus attachés au terminal courant) *sleep 20&* (processus qui dort pendant 20s en arrière plan) *sleep 30&* (processus qui dort pendant 30s en arrière plan) *ps -l* (pour voir la nouvelle configuration de processus)

1. Effet de la commande exec.
   1. Faire : *exec ps*

**que se passe-t-il et pourquoi ?**

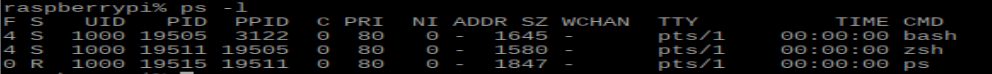
**Indication** : pour exec, comme pour quelques autres commandes (built-in commands) le shell ne créée pas de nouveau processus, c'est à qu'il ne fait pas appel à *fork* (cf. man exec).

* 1. Travailler dans une nouvelle fenêtre,
  2. Entrer maintenant la commande : zsh,





* 1. Relever le numéro du processus qui fait tourner ce nouveau *shell* courant (en faisant *ps -l*),



* 1. Faire maintenant : *exec ps -l.*



Relever le pid du processus qui exécute ce programme *ps -l*.

Pid ps –l =19511

**Quelle est sa valeur et pourquoi ?**

**Pid ps –l = pid zsh**

# Introduction aux exercices en C :

Créer un répertoire appelé TPProc dans lequel vous travaillerez :

cd Pour aller dans le home directory mkdir TPProc

cd TPProc

Par la suite, pour compiler les fichiers et créer les exécutables, utiliser la commande gcc.

Par exemple (premier exercice) :

gcc exo1.c -o exo1

# Création de processus (fonction fork)

## Fonctions utilisées :

On va tout d'abord utiliser les fonctions Unix suivantes :

* fork()

Cette fonction va créer un processus. La valeur de retour n de cette fonction indique :

* + n > 0

On est dans le processus père

* + n = 0

On est dans le processus fils

* + n = -1

fork a échoué, on n'a pas pu créer de processus

* getpid() :

Cette fonction retourne le numéro du processus courant.

* getppid() :

Cette fonction retourne le numéro du processus père.

## Exercice 1 :

1. Quel est le processus de pid 1 ?

2. Lancer le programme ci-dessous avec les arguments 10 20. Tapez ps -la dans un autre

terminal avant la fin du père, avant la fin du fils. Quels sont les ppid du pere et du fils ?

Donnez une explication.

3. Lancer le programme ci-dessous avec les arguments 10 0. Tapez ps -la dans un autre terminal avant la fin du père. Que constatez-vous ?

#include <unistd.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main(int argc, char \* argv[]) {

pid\_t pid;

int attente\_fils,attente\_pere;

if(argc != 3)

perror("usage: ex1 n m\n");

attente\_pere = atoi(argv[1]);

attente\_fils = atoi(argv[2]);

switch(pid=fork()) {

case -1:

perror("fork error");

break;

case 0:

sleep(attente\_fils);

printf("fils attente finie\n");

break;

default:

sleep(attente\_pere);

printf("pere attente finie\n");

break;

}

}

## III.2 Exercice 2 :

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*fork-mul.c\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ #include <stdio.h>

#include <sys/types.h> #include <unistd.h> int main (void)

{

int valeur, valeur1 ;

printf (" print 1 - Je suis le processus pere num=%d \n", (int)getpid() );

valeur = fork();

printf (" print 2 - retour fork: %d - processus num= %d -num pere=%d \n", valeur, (int)getpid(), (int)getppid() );

valeur1 = fork();

printf (" print 3 - retour fork: %d - processus num= %d -num pere=%d \n", valeur1, (int)getpid(), (int)getppid() );

return 0;

}

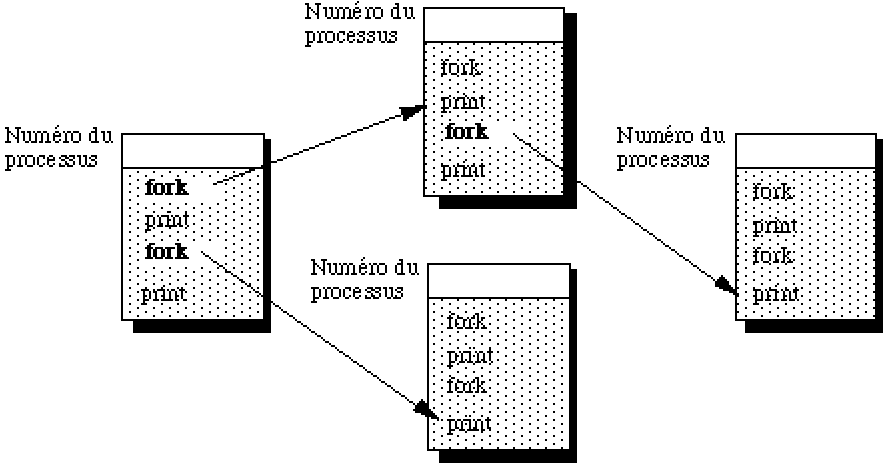
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Récupérez le programme : fork-mul.c Compiler fork-mul.c, puis l'exécuter.

**Après exécution et à l'aide du schéma suivant, relever les numéros des processus et numéroter l'ordre d'exécution des instructions printf de façon à retrouver l'ordre d'exécution des processus.**

Remarque :

si on relève un numéro de processus égal à 1, il s'agit du processus init, père de tous les processus. Init adopte les processus orphelins, c'est à dire qu'un processus dont le père s'est terminé devient fils de 1, et getppid() renvoie 1.



# Synchronisation de processus père et fils (mécanisme wait/exit)

## Fonctions utilisées :

* + exit( i )

termine un processus, i est un octet (donc valeurs possibles : 0 à 255) renvoyé dans une variable du type int au processus père.

* + wait( &Etat )

met le processus en attente de la fin de l'un de ses processus fils.

Quand un processus se termine, le signal SIGCHILD est envoyé à son père. La recéption de ce signal fait passer le processus père de l'état bloqué à l'état prêt. Le processus père sort donc de la fonction wait.

La valeur de retour de wait est le numéro du processus fils venant de se terminer. Lorsqu'il n'y a plus (ou pas) de processus fils dont il faut attendre la fin, la fonction wait renvoie -1.

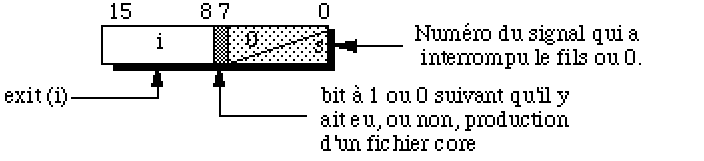
Chaque fois qu'un fils se termine le processus père sort de wait, et il peut consulter Etat pour obtenir des informations sur le fils qui vient de se terminer.

Etat est un pointeur sur un mot de deux octets :

* L'octet de poids fort contient la valeur renvoyée par le fils ( i de la fonction exit( i )),
* L'octet de poids faible :
  + contient 0 dans le cas général,
  + En cas de terminaison anormale du processus fils, cet octet de poids faible contient la valeur du signal reçu par le fils.

Cette valeur est augmentée de 80 en hexadécimal (128 décimal), si ce signal a entrainé la sauvegarde de l'image mémoire du processus dans un fichier core.

Schéma résumant le contenu du mot pointé par Etat à la sortie de wait :



## Mécanisme wait/exit

Récupérez le fichier fork-sync.c , compilez et exécutez le fork-sync.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fork-sync.c \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h> #include <sys/types.h> #include <unistd.h> #include <sys/wait.h> #include <stdlib.h>

int main (void)

{

int valeur, ret\_fils,etat ;

printf ("Je suis le processus pere de pid %d \n", (int)getpid()); valeur=fork();

switch (valeur)

{

case 0 :

printf("\t\t\t\t\*\*\*\*\*\*\*\*\n\t\t\t\t\* FILS \* \n\t\t\t\t\*\*\*\*\*\*\*\*\n"); printf ("\t\t\t\tProc fils de pid %d \n\t\t\t\tPere de pid %d \n",

(int) getpid(),(int) getppid() ); printf("\t\t\t\tJe vais dormir 30 secondes ...\n"); sleep (30);

printf("\t\t\t\tJe me reveille ,\n\t\t\t\tJe termine mon execution par un EXIT(7)\n");

exit (7);

case -1:

printf ("Le fork a echoue"); exit(2);

default:

printf("\*\*\*\*\*\*\*\*\n\* PERE \*\n\*\*\*\*\*\*\*\*\n");

printf ("Proc pere de pid %d \nFils de pid %d \n", (int) getpid(),valeur );

printf ("J'attends la fin de mon fils...\n"); ret\_fils = wait (&etat);

printf("Mon fils de pid %d est termine,\nSon etat etait : %0x\n", ret\_fils,etat);

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# Père et fils exécutent des programmes différents

## Fonctions utilisées :

La fonction exec charge un fichier dans la zone de code du processus qui l'appelle,

**remplaçant** ainsi le code courant par ce fichier. Une des formes de cette fonction est execl :

execl (fic, arg0, [arg1, ... argn,] (char \*)0) char \* fic, arg0, arg1, ... argn;

Commentaires :

* fic est le nom du fichier exécutable qui sera chargé dans la zone de code du processus qui appelle execl.

Si ce fichier n'est pas dans le répertoire courant, il faut donner son nom complet (chemin absolu).

* Les paramètres suivants sont des pointeurs sur des chaines de caractères contenant les arguments passés à ce programme (cf. argv en C).

La convention UNIX impose que la première soit le nom du fichier lui-même et que le

dernier soit un pointeur nul.

Par exemple, si on veut charger le fichier appelé prog qui se trouve dans le répertoire courant et qui n'utilise auncun argument passé sur la ligne de commande :

execl ("prog", "prog", (char \*)0)

## V.1 Fonctionnement de exec :

Compiler puis exécuter le fichier fexec.c.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fexec.c \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h> #include <sys/types.h> #include <unistd.h> #include <sys/wait.h>

int main (int argc, char \*argv[])

{

int Pid;

int Fils,Etat;

/\*

On peut executer ce programme en lui passant diverses

commandes en argument, par exemple, si l'executable est fexec : fexec /usr/bin/ps

\*/

if (argc != 2)

{printf(" Utilisation : %s fic. a executer ! \n", argv[0]); exit(1);

}

printf (" Je suis le processus %d je vais faire fork\n",(int) getpid()); Pid=fork();

switch (Pid)

{

case 0 :

printf (" Coucou ! je suis le fils %d\n",(int) getpid());

printf (" %d : Code remplace par %s\n",(int) getpid(), argv[1]); execl(argv[1], argv[1], NULL);

printf (" %d : Erreur lors du exec \n", (int) getpid()); exit (2);6+56

case -1 :

printf (" Le fork n'a pas reussi "); exit (3) ;

default :

/\* le pere attend la fin du fils \*/

printf (" Pere numero %d attend\n ",(int) getpid()); Fils=wait (&Etat);

printf ( " Le fils etait : %d ", Fils);

printf (" ... son etat etait :%0x (hexa) \n",Etat); exit(0);

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

On pourra y faire exécuter par execl un fichier très simple, par exemple celui dont le source est ci-dessous :

int main (void)

{

printf("Coucou, ici %d !\n", getpid() ); sleep (4);

return 6;

}

# Exercices :

# Exercice 1 :

D'après les exemples précédents, écrire un programme dont le fonctionnement est le suivant :

* il lit sur la ligne de commande (utiliser argc et argv) le nombre N de processus à créer.
* il crée ces N processus en faisant N appels à fork (cf. plus loin la tâche assignée à ces processus).

Récupérez un canevas du programme à écrire ainsi que des indications sur l'utilisation de argc, argv et sur la création des N fils.

* il se met en attente (appel à Pid\_Fils = wait(&Etat)) de ces N processus fils et visualise leur identité (Pid\_Fils et valeur de Etat) au fur et à mesure de leurs terminaisons.

Pour attendre la fin de tous les fils, utiliser le fait que wait renvoie la valeur -1 quand il n'y a plus de processus fils à attendre.

Ce que fait chacun des processus fils Pi:

* + il visualise son pid (getpid) et celui de son père (getppid),
  + il se met en attente pendant 2\*i secondes (sleep (2\*i)), visualise la fin de l'attente,
  + il se termine par exit (i).

Compiler en utilisant l'option -Wall de gcc, il ne doit pas y avoir de warnings après la compilation.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* canevas du programme \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

* Dans main, on indique comment utiliser
* les parametres passes sur
* la ligne de commande

\*

\*/

int main (int argc, char \*argv[])

{

int Nbre\_de\_Proc, i\_fils, pid\_fils;

if (argc != 2) /\* On utilise un seul parametre \*/

{

printf(" Utilisation : %s nbre-de-processus ! \n", argv[0]); exit(2);

}

Nbre\_de\_Proc = atoi (argv [1]); /\* conversion ascii -> entier \*/

/\*

\* creation des processus fils

\*

\*/

for ( i\_fils =1; i\_fils <= Nbre\_de\_Proc ; i\_fils++ )

{

pid\_fils = fork(); switch (pid\_fils)

{

case 0 :

fils(i\_fils); /\* il faut ecrire la fonction fils ... \*/ break;

case -1 :

perror("Le fork n'a pas reussi "); exit(33); /\* si erreur -> fin du pere ! \*/

}

}

/\*

* Dans la fonction pere, on utilisera le
* fait que wait renvoie la valeur -1 quand
* il n'y a plus de processus fils a attendre.

\*

\*/

pere(); /\* il faut aussi ecrire la fonction pere ... \*/ return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Exercice2**

Ecrire un programme qui crée 2 processus l’un faisant la commande ls -l, l’autre ps -l. Le

père devra attendre la fin de ses deux fils et afficher quel a été le premier processus à terminer.