INTRODUCTION AUX SYSTEMES D'EXPLOITATION

TD1/TP1
Création et terminaison des processus

S O M M A I R E

1. La primitive fork ()

1.1. Description du fork ()

Un processus peut se dupliquer – et donc créer un nouveau processus – par la fonction :

```
int fork(void)
```

Cette fonction permet la création dynamique d'un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé.

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,
- 0 pour le processus fils,
- Le n° du processus fils (PID) pour le père.

Mis à part ce point, les deux processus sont exactement identiques et exécutent le même programme sur **deux copies distinctes** des données. Les espaces de données sont complètement séparés : la modification d'une variable dans l'un est invisible dans l'autre.

À l'issue d'un **fork()** les deux processus s'exécutent simultanément.

On souhaite généralement exécuter des tâches distinctes dans le processus père et le processus fils. La valeur retournée par **fork** est donc très importante pour différencier le processus père du processus fils. Un test permet d'exécuter des parties de code différentes dans le père et le fils.

```
int code_retour ;
code_retour  = fork ();
switch (code_retour ) {
   case –1       :
         printf ("Pbm lors de la creation du processus\n");
         break;
  case 0         :
         printf ("Je suis le processus fils \n");
         break;
   default        :
         printf ("Je suis le processus père\n");
         printf ("Je viens de créer le processus fils dont le pid est %d \n",code_retour);
}
```

1.2. Exercice N°1 (TD)

Le programme suivant permet d'illustrer le comportement du fork.

```
main ()  {
          int code_retour ;
          printf ("Debut du test fork ()\n");
          code_retour  = fork ();
          switch (code_retour ) {
             case –1        :
                    printf ("Pbm lors de la creation du processus\n");
                    break;
             case 0         :
                    printf ("Je suis le processus fils \n");
                    break;
             default        :
                    printf ("Je suis le processus père\n");
                    printf ("Je viens de créer le processus fils dont le pid est %d \n",code_retour);
          }

          printf ("code_retour %d\n", code_retour);
          printf ("Fin du test fork ()\n");
}
```

Exécuter ce programme puis expliquer le résultat obtenu.

2. La génétique des processus

2.1. Principes

Un processus fils hérite de la plupart des attributs de son père, à l'exception de :

- son PID, qui est unique et attribué par le système d'exploitation,
- le PID de son père,
- ses temps d'exécution initialisés à la valeur nulle,
- les signaux en attente de traitement,
- la priorité initialisée à une valeur standard,
- les verrous sur les fichiers.

Tous les autres attributs du père sont hérités, et notamment la table des descripteurs de fichiers.

2.2. Exercice N°2 (TD)

A l'aide de la procédure fork() et en utilisant les primitives getuid, getgid,… fournies en annexe (consultez le manuel sur machine), écrire un programme qui crée un nouveau processus et qui affiche pour les deux processus (père et fils) les caractéristiques générales d'un processus :

- Identifiant du processus.
- Identifiant du père du processus.
- Répertoire de travail du processus.
- Le propriétaire réel.
- Le propriétaire effectif.
- Le groupe propriétaire réel.
- Le groupe propriétaire effectif.

Avant d'afficher ces informations, on prendra bien soin de préciser si l'on est dans le processus père ou fils.

2.3. Exercice N°3 (TP)

Reprenons l'exercice précédent et positionnons une attente de quelques secondes (sleep (5)) dans le processus fils pour que le processus père se termine avant le fils.
Quel constat peut-on faire sur le résultat obtenu ?

2.4. Exercice N°4 (TP)

Cette exercice a pour objet d'illustrer le fait qu'un processus fils hérite d'une copie des descripteurs de fichiers ouverts par son père.
Reprenons la trame de l'exercice précédent et ajoutons-y **en préalable** à la création d'un nouveau processus l'ouverture d'un fichier « toto » en lecture/ecriture.
Attention, nous utiliserons les primitives systèmes pour l'accès au fichier : open, read, write et close (et non les fonctions de librairie : fopen, fwrite, fread, et fclose).

Soit le fichier « toto » contenant l'alphabet.

```
$ cat toto
abcefghijklmnoprstuvwxyz
```

Ecrire le programme qui réalise la séquence suivante :

1. Le processus fils commence par écrire « Fils » dans le fichier puis s'endort (sleep(2)).
2. Le processus père commence par s'endormir (sleep(1)) puis lit 4 caractères depuis le fichier et les affiche à l'écran.
3. Le processus père écrit ensuite « Père » dans le fichier et se termine.
4. Après s'être endormi (cf. étape1) le processus fils lit 4 caractères depuis le fichier et les affiche à l'écran puis se termine.

Expliquer le résultat obtenu à l'écran et dans le fichier en détaillant pas à pas le séquencement ainsi que l'effet des différentes instructions.

Remarque :
On prendra bien garde de ne pas omettre le fichier <fcntl.h> dans les fichiers d'inclusions. De manière générale, consulter le man afin de décider quoi mettre dans les inclusions.

Indice : une ouverture de fichier avant le fork aboutit à un fichier ouvert chez le fils et à un partage du pointeur de fichier entre père et fils. Le pointeur de fichier est un pointeur permettant de sauvegarder pour un processus l'emplacement de lecture/écriture dans un fichier.

3. Synchronisation père-fils

3.1. La primitive exit ()

void exit (int *status*)

La fonction **exit** met fin au processus qui l'a émis, avec un code de retour **status**.
Tous les descripteurs de fichiers ouverts sont fermés ainsi que tous les flots de la bibliothèque d'E/S standard.
Si le processus à des fils lorsque **exit** est appelé, ils ne sont pas modifiés mais comme le processus père prend fin, le nom de leur processus père est changé en 1, qui est l'identifiant du processus **init**.
Ce processus **init** est l'ancêtre de tous les processus du système excepté le processus 1 lui-même ainsi que le processus 0 chargé de l'ordonnancement des processus.
En d'autres termes, c'est l'ancêtre de tous les processus qui adopte tous les orphelins.

Par convention, un code de retour égal à zéro (exit(0)) signifie que le processus s'est terminé correctement, et un code non nul (généralement 1) signifie qu'une erreur s'est produite (exit(1)). Seul l'octet de droite de l'entier **status** est remonté au processus père. On est donc limité à 256 valeurs (8 bits) pour communiquer un code de retour à son processus père.

Le père du processus qui effectue un **exit** reçoit son code retour à travers un appel à **wait**.

3.2. La primitive wait ()

```
#include <sys/types.h>
#include <sys/wait.h>
int  wait (int * terminaison)
```

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 (il est mort, mais n'a pas délivré son dernier message).
L'élimination d'un processus terminé de la table ne peut se faire que par son père, grâce à la fonction **wait** (il faut que le père reconnaisse la mort du fils afin qu'il puisse reposer en paix).
Avec cette instruction :

- Le père se bloque en attente de la fin d'un fils (l'appel à wait est
- Elle rendra le n° (**PID**) du premier fils mort trouvé.
- La valeur du code de sortie est reliée au paramètre **d'`exit`** de ce fils.

On peut donc utiliser l'instruction **`wait`** pour connaître la valeur éventuelle de retour, fournie par **`exit()`**, d'un processus. Ce mode d'utilisation est analogue à celui d'une fonction.
`wait()` rend –1 en cas d'erreur.

Un processus exécutant l'appel système **wait** est endormi (à l'état bloqué) jusqu'à la terminaison d'un de ses fils. Lorsque cela se produit, le père est réveillé et **wait** renvoie le PID du fils qui vient de mourir.

3.3. Exercice N°5 (TD)

En repartant du programme de l'exercice 3, ajouter les instructions qui permettront au processus père d'attendre la terminaison de son fils et qui afficheront le code retour de celui-ci. On suppose dans un premier temps que le fils se termine correctement : exit(0).

3.4. La primitive waitpid ()

L'appel système **waitpid** permet de tester la terminaison d'un processus particulier, dont on connaît le PID.

```
#include <sys/types.h>
#include <sys/wait.h>
int waitpid (int pid, int *terminaison, int options)
```

La primitive permet de tester, en bloquant ou non le processus appelant, la terminaison d'un processus particulier ou appartenant à un groupe de processus donné et de récupérer les informations relatives à sa terminaison à l'adresse ***terminaison***.

Plus précisément le paramètre ***pid*** permet de sélectionner le processus attendu de la manière suivante :

- <-1 tout processus fils dans le groupe | pid |
- -1 tout processus fils (équivalent du wait())
- 0 tout processus fils du même groupe que l'appelant
- **> 0 processus fils d'identité pid (c'est le cas qui nous intéresse).**

Le paramètre options est une combinaison bit à bit des valeurs suivantes. La valeur WNOHANG permet au processus appelant de ne pas être bloqué si le processus demandé n'est pas terminé.

La fonction renvoie :

-1	en cas d'erreur,
0	en cas d'échec (processus demandé existant mais non terminé) en mode non bloquant,
> 0	le numéro du processus fils zombi pris en compte.

3.5. Interprétation de la valeur renvoyée

La valeur de l'entier ***terminaison***, au retour d'un appel réussi de l'une des primitives wait ou waitpid, permet d'obtenir des informations sur la terminaison ou l'état du processus.

De manière générale sous UNIX, pour un processus qui s'est terminé normalement c'est-à-dire par un appel **exit(n**), l'octet de poids faible de n est récupéré dans le second octet de *terminaison, ce que symbolise le schéma ci-dessous sur une machine où les entiers sont codés sur 32 bits.

Entier **n** du processus fils.

octet 3	octet 2	octet 1	octet 0

Entier ***terminaison*** récupéré par le processus père

? ? ? ? ? ? ? ?	? ? ? ? ? ? ? ?	octet 0	? ? ? ? ? ? ? ?

Pour un processus qui s'est terminé accidentellement à cause du signal de numéro sig, l'entier *terminaison est égal à sig, éventuellement augmenté de la valeur décimale 128 si une image mémoire (fichier core) du processus a été créée à sa terminaison.

L'interprétation de la valeur récupérée doit, pour des raisons de portabilité être réalisée par l'intermédiaire de macro-fonctions prédéfinies, appelées avec la valeur fournie au retour de l'appel à wait ou waitpid.

Fonction	Interprétation
WIFEXITED	Valeur non nulle si le processus s'est terminé normalement.
WEXITSTATUS	Fournit le code de retour du processus si celui-ci s'est terminé normalement.
WIFSIGNALED	Valeur non nulle si le processus s'est terminé à cause d'un signale.
WTERMSIG	Fournit le numéro du signal ayant provoqué la terminaison du processus.
…	

3.6. Exercice N°6 (TP)

a) En repartant du programme de l'exercice précédent, ajouter les instructions qui permettront au processus père d'interpréter les informations de terminaison de son processus fils :

- Code retour du processus fils
- Signal ayant provoqué la terminaison du processus fils.

On fera terminer le processus fils avec un code retour de 3.

b) Mettre ensuite le processus fils en sommeil (sleep (10)) et tuer ensuite celui-ci à l'aide de la commande shell : kill –15 pid_fils (sur un shell séparé).
On prendra soin de lancer le programme en arrière-plan.
Interpréter le résultat obtenu.

c) Reproduire le cas précédent avec « kill –9 pid_fils » (sur un shell séparé).

3.7. Exercice N°7 (TD)

En repartant du programme de l'exercice précédent, ajouter les instructions qui permettront au processus père d'interpréter les informations de terminaison de son processus fils :

- Le processus père doit créer 2 processus fils.
- Le premier processus fils
  - s'endormira pendant 5 secondes
  - affichera un message avant de s'endormir puis avant de se terminer
  - retournera la valeur 1 en code retour
- Le deuxième processus fils
  - s'endormira pendant 10 secondes
  - affichera un message avant de s'endormir puis avant de se terminer
  - retournera la valeur 2 en code retour
- Le processus père se mettra en attente de terminaison du deuxième processus fils et affichera les informations de terminaison de celui-ci.

4.Recouvrement du code du processus inital

4.1. Principe du recouvrement

La création d'un nouveau processus ne peut se faire que par le « recouvrement » du code d'un processus existant.
Cela se fait à l'aide d'une des fonctions de la famille **exec** qui permet de faire exécuter par un processus un autre programme que celui d'origine.

Lorsqu'un processus exécute un appel exec, il charge un autre programme exécutable en conservant le même environnement système :

- Du point de vue du système, il s'agit toujours du même processus : il a conservé le même pid.
- Du point de vue de l'utilisateur, le processus qui exécute exec disparaît, au profit d'un nouveau processus disposant du même environnement système et qui débute alors son exécution.

Les deux fonctions de base sont :

```
int execl (char *ref, char *arg0, …, char *argn, 0)
int execv (char *ref, char *argv [ ]);
```

Il existe d'autres fonctions analogues dont les arguments sont légèrement différents : execle (), execlp (), execv (), execcve (), execvp ().

Ces fonctions rendent –1 en cas d'échec.

4.2. Primitive execl

Dans le cas de **execl**, les arguments de la commande à lancer sont fournis sous la forme d'une liste terminée par un pointeur nul :

- **ref** est une chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
- **arg0, arg1, …, argn** sont les arguments du programme.
- Le premier argument, **arg0**, reprend en fait le nom du programme.

```
void main()
{
        execl("/bin/ls", "ls", "-l", NULL);
        printf("Erreur lors de l'appel à ls \n");
}
```

Remarque:
La partie de code qui suit l'appel d'une primitive de la famille exec ne s'exécute pas, car le processus où elle se trouve est remplacé par un autre. Ce code ne sert donc que dans le cas où la primitive exec n'a pas pu lancer le processus de remplacement (le nom du programme était incorrect par exemple).

4.3. Primitive execv

Dans le cas de **execv**, les arguments de la commande sont sous la forme d'un vecteur de pointeurs (de type argv [ ]) dont chaque élément pointe sur un argument, le vecteur étant terminé par le pointeur NULL :

- **ref** est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter.,
- **argv [ ]** contient la liste des arguments**.**

```
#include <stdio.h>
#define NMAX 5
void main () {
        char *argv [NMAX];
        argv [0] = "ls";
        argv [1] = "-l";
        argv [2] = NULL;
        execv ("/bin/ls", argv);
        printf("Erreur lors de l'appel à ls \n");
}
```

4.4. Exercice N°9 (TP)

Le programme suivant permet d'illustrer le comportement de exec.

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

main () {
        char buf [80];

        printf ("[%d]-- Debut du test exec () \n",getpid());
        sprintf (buf,"-- pid=[%d]", getpid());
        execl ("/bin/echo","echo","Execution","d'un test exec",buf, NULL);
        printf ("[%d]-- Echec de la fonction execl () \n",getpid());
        printf ("[%d]-- Fin du test exec () \n",getpid());
        exit (0);
}
```

a) Exécuter puis expliquer le résultat obtenu.

b) Remplacer la commande « /bin/echo » par « /bin/ech ». Expliquer le résultat obtenu.

4.5. Exercice N°10 (TP)

a) A l'aide des primitives fork et execl, écrire le programme qui permet de faire exécuter la commande ps avec l'option –l depuis un processus existant.
Ce dernier devra se mettre en attente du code retour de la commande ps.

b) Reprendre le programme précédent mais en utilisant la commande execv.

4.6. Exercice N°11 (TP)

Grâce aux 3 instructions, **`fork()`**, **`execv(),`** et **`wait(),`** écrire un interpréteur de commandes simplifié dont le fonctionnement sera similaire à un shell.

5. Annexe 1 : Primitives d'accès aux caractéristiques générales d'un processus

Nous rappelons ci-dessous la liste des principales caractéristiques d'un processus que l'on trouve dans le bloc de contrôle UNIX et un certain nombre de primitives permettant leur consultation ou leur modification.
A la création d'un processus un très grand nombre de ses attributs sont hérités du processus père, c'est-à-dire ont comme valeur initiale la valeur actuelle des attributs correspondants dans le processus père.

5.1. Identité du processus et celle de son père

Un processus a accès respectivement à son PID et à celui de son père par l'intermédiaire respectivement de la fonction **getpid** et **getppid**.

```
#include <unistd.h>
int getpid(void);
int getppid(void);
```

5.2. Répertoire de travail du processus

Deux fonctions sont disponibles pour obtenir et changer le répertoire de travail d'un processus. La connaissance de ce répertoire est importante, car toutes les chemins relatifs utilisés sont exprimés par rapport à celui-ci.
La fonction **chdir** permet de changer le répertoire de travail, la fonction **getcwd** de récupérer le chemin absolu du répertoire de travail courant.

```
#include <unistd.h>
int chdir(const char *chemin);
char * getcwd(char * buf, unsigned long taille);
```

Le paramètre **taille** correspond à la longueur maximale du chemin à récupérer.
Si cette longueur n'est pas suffisante, la fonction va echouer.

5.3. Masque de création des fichiers

Le masque de création de fichiers umask permet d'interdire le positionnement de certaines permissions. Les permissions demandées lors de création de fichiers par les commandes creat ou open, sont « masquées » par le umask.
La fonction **umask** permet de modifier la valeur du « umask » tout en retournant la valeur du masque précédent.

```
#include <unistd.h>
unsigned int  umask(unsigned int masque);
```

5.4. La table des desripteurs de fichier

Chaque processus peut posséder à un instant donné un maximum de **OPEN_MAX** (constante définie dans **<limits.h>)** descripteurs de fichiers**.**
Un processus hérite à sa création de tous les descripteurs de son père. Il a donc accès aux mêmes entrées/sorties.
Il peut acquérir de nouveaux descripteurs par l'intermédiaire des primitives standards (**open, creat, dup, ..)** ou en libérer par la primitive **close**.

5.5. Liens du processus avec les utilisateurs

Ces liens unissent le processus :
- d'une part à des utilisateurs individuels (identifiés par des constantes de type uid_t)
- et d'autre part à des groupes d'utilisateurs (identifiés par des constantes de type gid_t)

Il existe 4 liens différents étant tous hérités du processus père à la création du processus :
- Le propriétaire réel.
- Le propriétaire effectif.
- Le groupe propriétaire réel.
- Le groupe propriétaire effectif.

Ces valeurs peuvent éventuellement être modifiés au cours de l'exécution du processus

Différence entre propriétaire réel et effectif

Le propriétaire réel est l'utilisateur ayant lancé le processus.
Le propriétaire effectif est utilisé pour vérifier certains droits d'accès du processus vis-à-vis des différents objets du système (fichiers et envoi de signaux à destination d'autres processus).
Normalement le propriétaire effectif correspond au numéro du propriétaire réel.
On peut cependant positionner l'utilisateur effectif d'un processus au propriétaire du fichier exécutable à l'aide du **set-uid** bit. Si celui-ci a été positionné sur le fichier exécutable, alors le fichier s'exécutera avec les droits du propriétaire et non avec les droits de celui qui l'a lancé.
Cette technique permet notamment d'autoriser exceptionnellement un utilisateur quelconque à lancer un programme système (nécessitant les droits d'exécution root) sans avoir à se connecter en tant que root.

Consultation des valeurs

La fonction getuid (respectivement getgid) retourne le numéro d'identification de l'utilisateur (respectivement le numéro d'identification du groupe) à qui appartient le processus. Ce numéro peut être différent de celui du propriétaire du fichier exécutable.

```
#include <unistd.h>
#include <sys/types.h>
int getuid(void);
int getgid(void);
```

Il en est de même pour le propriétaire effectif au travers des fonctions geteuid et getegid.

```
#include <unistd.h>
#include <sys/types.h>
int geteuid(void);
int getegid(void);
```

Changement des valeurs

La fonction setuid (respectivement setgid) permet de modifier à la fois le propriétaire réel et le propriétaire effectif d'un processus. Cette possibilité est offerte en générale au super-utilisateur du système (root dont le numéro d'identification est 0).
C'est de cette manière que sont initialiées les valeurs des propriétaires des processus SHELL créés au cours de la procédure login de connexion des utilisateurs.

```
#include <unistd.h>
#include <sys/types.h>
int setuid (int uid)
int setgid (int gid)
```