PROGRAMMATION SYSTÈME: TP I52 - 1

FORK, EXEC, SIGNAL, KILL, PIPE, DUP2

Pour l'ensemble des exercices de ce TP, vous n'oublierez pas de lire soigneusement les pages du manuel pour les différentes fonctions à utiliser. Si une fonction porte le même nom qu'une commande du shell, vous obtiendrez par défaut le manuel de la commande. Tapez alors `man 2` *(ou* `man 3`*)* fonction *pour obtenir le bon manuel.*

1. PRÉLIMINAIRES

Afin d'éviter de perdre du temps sur certains problèmes, vous devez bien comprendre le mécanisme de gestion des affichages par un programme C. Pour cela, compilez et exécutez les 3 programmes suivants et déduisez-en comment fonctionne la fonction `printf()` et le rôle de la fonction `fflush()`.

```
#include <stdio.h>

int main(void)
{
  printf("Hello world");
  sleep(5);
  printf("\nBye Bye\n");
}
```

```
#include <stdio.h>

int main(void)
{
  printf("Hello world");
  fflush(stdout);
  sleep(5);
  printf("\nBye Bye\n");
}
```

```
#include <stdio.h>

int main(void)
{
  printf("Hello world\n");
  sleep(5);
  printf("\nBye Bye\n");
}
```

2. FORK

Dans un programme C, l'appel système `fork()` permet de dupliquer le processus en cours d'exécution. Le nouveau processus devient un fils du processus courant et les deux programmes s'exécutent en concurrence en l'absence de toute instruction précisant une attente de la part du père.

Exercice 1. Pour constater ceci, compilez et exécutez le code suivant:

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

int main(void)
{

  pid_t pid;
  int i;

  printf("Je suis seul\n");
  for (i = 10; i > 0; i--)
    {
       printf("%d ",i);
       fflush(stdout);
       sleep(1);
    }
  printf("\n Nous sommes deux maintenant\n");
  pid = fork();
  while(1);
}
```

Au moment où vous lancez votre programme, faites afficher dans un terminal la liste de vos processus en exécution. Quand la phrase *"Nous sommes deux maintenant"* apparaît, affichez de nouveau la liste des processus avec pour chacun d'eux le PID du père.

Remarque: le type `pid_t` correspond à un entier.

Exercice 2. Lisez le manuel de `fork()` pour déterminer la nature de la valeur renvoyée par cette fonction. Réalisez alors un programme qui se duplique et qui affiche selon que ce soit le père ou le fils qui est en exécution:

Date: Novembre 2007.

`Je suis le père, mon fils a pour PID:` *PID du fils*
`Je suis le fils, mon père a pour PID:` *PID du père*
Dans l'un des deux cas vous aurez besoin d'utiliser la fonction `getpid()`. Exécutez le programme plusieurs fois pour vous convaincre que l'exécution du père et du fils ne suit aucun ordre déterminé.

Exercice 3. Suite à l'exécution de la primitive `fork()`, le fils hérite de l'espace d'adressage de son père, il hérite donc de toutes les variables déclarées dans ce dernier. La question qui se pose est de savoir si par la suite ces données sont partagées ou non entre le père et le fils. Pour cela réalisez un programme qui initialise une variable n à 0 et qui se duplique. La variable n est ensuite incrémentée et affichée par le père et le fils. Qu'en déduisez-vous ?

3. Les Signaux

Il est possible d'envoyer à tout processus en cours d'exécution un signal via un appel système adéquat en utilisant par exemple à partir du shell la commande `kill`. Un signal est un moyen d'indiquer au processus une action à entreprendre à partir de conventions préétablies. Chaque signal a une signification particulière qui détermine le comportement du processus (sauf pour les signaux `SIGUSR1` et `SIGUSR2` qui sont destinés à être gérés par les utilisateurs). Les signaux sont identifiés par un numéro entier ou un nom symbolique décrit dans `/usr/include/asm/signal.h`. Les noms utilisables pour la commande `kill` sont obtenus en exécutant `kill -l`. Ainsi la commande `kill -KILL` *PID* (équivalente à `kill -9` *PID*) envoie un signal de terminaison qui ne peut être ignoré par le processus concerné. La fonction C `signal(..,..)` permet à un processus de capturer un signal et d'exécuter une fonction particulière à sa réception. Le manuel n'étant pas très clair sur la façon de l'utiliser, compilez et exécutez le programme suivant:

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

void traitersignal(int sig)
  {
     printf("Signal %d bien recu\n",sig);
  }

int main(void)

{
  signal(SIGUSR1, traitersignal);
  while(1);
}
```

Ouvrez alors un shell, récupérez le PID de votre processus et tapez `kill -USR1` PID. A chaque fois que votre programme recevra le signal `USR1` il exécutera la fonction `traitersignal` et reprendra normalement son exécution. La syntaxe de `signal` est donc: `signal(`*nom du signal à capturer, fonction de traitement*`)`. La fonction sera de type `void` et devra toujours admettre pour paramètre un entier destiné à recevoir le numéro du signal, même si vous n'en n'avez pas l'utilité.
Exercice 4. La fonction C `kill(..,..)` est l'équivalent de la commande shell `kill`. Réalisez un programme qui après s'être dupliqué réalise les actions suivantes:

- le père se met en attente de la réception du signal `SIGUSR1` et affiche le PID de son fils lorsqu'il l'a reçu.
- le fils envoie un signal `SIGUSR1` à son père.

Pour que cela fonctionne, il faut que le père s'exécute en premier. Utilisez `sleep()` pour endormir le fils pendant 5 secondes ce qui devrait permettre au père de commencer. Pour mettre en attente le père jusqu'à réception d'un signal, utilisez la fonction `pause()`.

Exercice 5. Réalisez le programme `veille` qui effectue les opérations suivantes :

. le programme se duplique;
. le père exécute une boucle infinie;
. si toutes les (au plus) 10 secondes, le fils ne reçoit pas le signal `SIGUSR1`, il stoppe l'exécution du père.

4. Exec

Exercice 6. Le C-shell possède une commande interne nommée `exec`. Ouvrez deux terminaux T1 et T2. Affichez dans T2 la liste des processus en exécution et repérez le PID de T1. Exécutez dans T1: `exec sleep 15`. Affichez de nouveau dans T2 la liste des processus en exécution, que constatez-vous ? Attendez la fin des 15 secondes. Que

se passe-t-il ? Déduisez en la fonctionnalité de `exec`.

En C, il existe une famille de fonctions qui correspond à la commande `exec`. Chaque fonction permet de remplacer le code exécutable du processus courant par celui qu'elle est chargée d'exécuter.

Conséquence 1: à moins que l'appel n'échoue (la commande spécifiée ne peut être exécutée), toutes les instructions qui suivent la fonction ne seront jamais exécutées.
Conséquence 2: la valeur de retour des fonction C de la famille `exec` n'est utilisable que si la fonction a échouée (auquel cas elle renvoie -1).
Conséquence 3: si un processus doit lancer une commande (via un appel à l'une des fonctions de la famille `exec`) puis poursuivre son exécution, il doit alors utiliser un fils qui se chargera de l'exécution de la commande.

Nous détaillerons ici uniquement les 4 fonctions `execl()`,`execlp()`,`execv()` et `execvp()`. Le premier argument de ces fonctions est le chemin absolu de l'exécutable. Pour les fonctions se terminant par `p`, si seul le nom de la commande est donné, elle est alors recherchée en utilisant la variable d'environnement `PATH`. Vous savez que dans tout programme C, on peut accéder à l'ensemble des paramètres présents sur la ligne de commande en utilisant le tableau `argv[]`.

. Le deuxième argument des fonctions `execl()` et `execlp()` doit être la valeur de `argv[0]`, c'est à dire le nom du programme lui même. Les arguments suivants sont des pointeurs sur des chaînes de caractères qui correspondent chacune à un paramètre que l'on passe à la commande à exécuter. Il faudra terminer par le pointeur `NULL` pour clore la liste des paramètres.
. Les fonctions `execv()` et `execvp()` n'acceptent que 2 arguments, le deuxième étant un tableau de chaînes de caractères, chaque élément correspondant aux paramètres de la ligne de commande.
Exemple: voici comment exécuter la commande `rm -r L2` avec ces différentes fonctions.

EXECL:
```
execl("/bin/rm","rm","-r","L2",NULL);
```

EXECLP:
```
execlp("rm","rm","-r","L2",NULL);
```

EXECV:
```
char *param[4];

param[0] = "rm";
param[1] = "-r";
param[2] = "L2";
param[3] = NULL;
execv("/bin/rm",param);
```

EXECVP:
```
char *param[4];

param[0] = "rm";
param[1] = "-r";
param[2] = "L2";
param[3] = NULL;
execvp("rm",param);
```

Exercice 7. Réalisez le programme `sequence` tel que `sequence com1 com2 ...comN` exécute les commandes `com1`, `com2`,..., `comN` de façon séquentielle. Le programme lancera un fils pour chaque commande à exécuter. Vous aurez besoin d'utiliser la primitive `wait()` qui permet à un père d'attendre la fin de son fils avant de poursuivre son exécution. Pour plus de simplicité, on supposera qu'aucune commande n'a besoin de paramètres pour s'exécuter et que chaque commande est accessible à partir de la variable d'environnement `PATH`. Testez votre programme en lançant `sequence who ls pwd`.

Exercice 8. Exécutez `sequence titi pwd`. Le manuel de `execlp()` indique que la fonction retourne dans l'appelant si une erreur s'est produite. Pouvez-vous expliquer ce qui s'est passé et le résultat obtenu ? Pour corriger ce problème il faut que le processus fils se termine si l'appel de `execlp` a généré une erreur. Pour mettre fin à l'exécution d'un processus on utilise la primitive `exit()`. Corrigez le programme `sequence` de façon à ce que `sequence titi pwd` ne renvoie qu'une fois le résultat de la commande `pwd`.

5. Communication par tube anonyme

Vous avez vu dans les exercices précédents qu'un processus fils hérite de toutes les variables de son père. Cependant, une fois l'appel `fork()` réalisé, vous ne connaissez pas pour l'instant de méthode permettant au père de communiquer de nouvelles données à son fils. Un tube est un moyen de transmettre des informations (sous forme de suite d'octets) d'un processus à un autre. L'appel système `pipe()` permet de créer pour un processus un tube de communication, ceci n'a bien sur d'intérêt que si le processus se duplique et désire pouvoir communiquer avec son fils. Le tube créé est un tube *anonyme*, ce qui signifie qu'il n'est connu que du processus créateur et de ses descendants. Toute autre processus ne peut accéder au tube (il existe sous Unix, un autre mécanisme permettant

de créer des tubes *nommés* qui peuvent être accédés par divers processus n'ayant aucun lien de parenté). La syntaxe d'appel de la primitive `pipe()` est `pipe(tube)` où ici `tube` est un tableau de deux entiers. La fonction renvoie dans `tube[0]` le numéro du descripteur associé au tube par lequel on peut y lire, et dans `tube[1]` le numéro du descripteur associé au tube par lequel on peut y écrire. Pour accéder au tube, on utilise les primitives `read()` et `write()` dont la syntaxe d'utilisation est:

. `read(`*num. de descripteur*, *adresse de stockage*, *nb. d'octets à lire*`)`,
. `write(`*num. de descripteur*, *adresse de l'objet*, *taille de l'objet en octets*`)`,

Il est possible à un processus qui possède les descripteurs de fermer pour lui le tube soit en lecture, soit en écriture, cependant il ne disposera plus d'aucun moyen pour le rouvrir ultérieurement. Enfin, le seul moyen de placer une fin de fichier dans un tube est de fermer ce tube en écriture dans **tous** les processus où il est ouvert dans ce mode. Tant qu'une fin de fichier n'est pas détectée dans le tube, l'appel à `read()` est bloquant pour le processus concerné si le tube est vide.
Important: Pensez toujours à fermer les extrémités du tube (avec la fonction `close()`) lorsque vous n'avez plus besoin de les utiliser.

Exercice 9. Soient x et y deux vecteurs de longueur n composés d'entiers, on appelle produit scalaire de x et y l'entier $\sum_{i=1}^{n} x_i y_i$. Il s'agit dans cet exercice d'écrire le programme `scal` qui calcule cette quantité en respectant les contraintes suivantes :

- la valeur n est un paramètre du programme,
- le processus `scal` engendre aléatoirement deux tableaux v_1 et v_2 de taille n composés d'entiers strictement inférieurs à 10 et les affiche (vous veillerez qu'à chaque nouvelle exécution du programme deux nouveaux tableaux soient engendrés),
- le processus engendre n fils qui doivent s'exécuter **en parallèle**, le fils numéro i calcule la valeur $v_1[i] \times v_2[i]$ et renvoie le résultat à son père,
- chaque fils affiche un message indiquant le calcul effectué (respectez la syntaxe donné dans l'exemple),
- le père calcule la somme de tous les résultats renvoyés,
- les instructions `else` et `switch` ne sont pas utilisées dans le programme, l'instruction `if` n'est utilisée qu'une seule fois.

Exemple:

```
# ./scal 5
    4     4     9     0     9
    3     9     0     9     1
Processus 0 a calcule     4 x     3 =    12
Processus 1 a calcule     4 x     9 =    36
Processus 3 a calcule     0 x     9 =     0
Processus 4 a calcule     9 x     1 =     9
Processus 2 a calcule     9 x     0 =     0
Resultat: 57
```

6. Duplication des descripteurs de fichiers

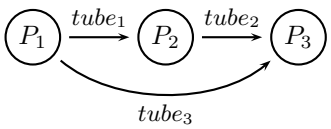
Considérons un processus ayant établi un tube de communication avec son fils, on souhaite que l'entrée du tube corresponde à la sortie standard du père et que la sortie du tube corresponde à l'entrée standard du fils. Pour pouvoir réaliser cela, nous allons utiliser la primitive `dup2(..,..)` dont la syntaxe est `dup2(`*descripteur1*, *descripteur2*`)`. Après exécution de cette fonction, *descripteur2* référence le fichier associé à *descripteur1*.
Exemple: Le descripteur de valeur `2` est associé à la sortie standard. Si `d` est un autre descripteur, après exécution de `dup2(d,2)` tout envoi d'informations utilisant le descripteur `2` ira dans le fichier associé au descripteur `d`.

Exercice 10. En utilisant les appels `pipe()` et `dup2()`, développez le programme `pipe` qui prend en paramètre deux commandes `com1` et `com2` et qui réalise le même travail que `com1 | com2`. Testez votre programme en exécutant `pipe ls wc`.

7. Recapitulatif

Exercice 11. Ecrire le programme `comptemot` qui accepte en paramètre une chaîne de caractères suivie d'un nombre quelconque de noms de fichiers (présents dans le répertoire courant). Ce programme renvoie sur la sortie standard le nombre total d'occurences de la chaîne dans les fichiers donnés en arguments. Pour réaliser ce travail, le programme exécute **en parallèle** autant de fils que de fichiers à traiter. Chaque fils traite un fichier en utilisant la commande `grep` pour compter le nombre d'occurences de la chaîne. Le résultat est renvoyé au père qui se charge de calculer la somme totale. Il n'est pas utilse d'utiliser la primitive `wait` pour ce programme.

Exercice 12. Ecrire un programme qui crée trois processus, P_1, P_2, P_3 reliés entre eux par des tubes de la manière suivante :



- P_1 : produit une suite de nombres aléatoires et écrit chacun des nombres sur les tubes 1 et 3; la longueur de cette suite sera passée en paramêtre du programme.
- P_2 : lit les nombres sur le tube 1, leur ajoute une valeur constante c, et les écrits sur le tube 2. Quand P_2 reçoit le signal `USR1` il incrémente c de 1 et quand il reçoit le signal `USR2` il décrémente c de 1.
- P_3 : lit les nombres de manière synchronisée sur les tubes 2 et 3 et affiche pour chaque couple, la valeur lue sur le tube 3 et sa différence avec la valeur lue sur le tube 2.