Fiche de TP-TD no. 5

En TP

Création

Exercice 1. Écrire un programme C qui affiche son PID, puis crée deux fils. Ils affichent leur PID et leur filiation (c'est-à-dire, le PID du père) et se terminent.

Exercice 2. Écrire un programme C qui affiche son PID, puis crée un fils et un petit-fils. Ils affichent leur PID et leur filiation et se terminent.

Exercice 3. Écrire un programme C qui change de PID toutes les 2 secondes, et affiche à chaque fois celui-ci.

Terminaison

Exercice 4. Ecrire un programme C qui affiche son PID, puis crée un fils. Le père affiche son PID, puis affiche un message signalant que son fils est mort lorsque c'est bien le cas. Le fils affiche son PID, s'endort 5 secondes puis se termine.

Exercice 5. Ecrire un programme C qui affiche son PID, puis crée un fils. Le père affiche son PID, attend la terminaison du fils puis affiche son code de sortie.

Recouvrement

Exercice 6. Ecrire un programme `verifier.c` dont l'usage sera `verifier com arg1 .. argn`, et qui effectue les opérations suivantes :

1. Il lance la commande `com arg1 .. argn` et signale une éventuelle erreur lors du lancement.
2. Il attend la fin de l'exécution et précise par un message le résultat (succés ou échec).

Rappels

– Utilisez `fork()` pour créer un fils. Cette 'fonction' retourne 0 au fils, et le PID du fils au père. Consultez `man fork`.
– La 'fonction' `wait(int *exit_status);` endort le père jusqu'à la terminaison d'un fils. Consultez `man 2 wait` ainsi que `man 3 exit`.
– La 'fonction' `execvp(const char *com, char *const argv[]);` transforme (mieux, recouvre) un processus en la commande `com`, appelée avec les paramètres tirés du tableau `argv`. Consultez `man execvp`.
– La 'fonction' `sleep(n);` endort un processus pendant `n` secondes.

En TD

Création de fils

Exercice 7. Considérez le programme suivant :

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 int main (int argc, char *argv[]) {
 5   int n, i, p;
 6
 7   if (argc != 2) {
 8     printf ("Erreur: 1 argument attendu\n");
 9     exit(1);
10   }
11
12   n = atoi(argv[1]);
13
14   printf("Pere pid=%d\n", getpid());
15
16   for (i = 1; i <= n; i++) {
17     p = fork();
18     if (p < 0) { perror ("fork"); exit(EXIT_FAILURE); }
19
20     if(!p){
21       printf("Fils pid=%d\n", getpid());
22       exit(0);
23     }
24
25   }
26
27   exit(EXIT_SUCCESS);
28
29 }
```

- Expliquez ce qui est accompli par ce programme, ligne par ligne.
- Expliquez ce qui est accompli par ce programme, d'un point de vue global.
- Pour quelle raison inclure dans ce programme `stdio.h`, `stdlib.h`, et `unistd.h` ? Soyez précis.
- Que se passe si on supprime du programme la ligne 22 ?
- Comment améliorer la gestion des erreurs du programme ?

Création de petits-fils

Exercice 8. Ecrire un programme C qui crée 3 fils, chacun des fils créant 2 petits-fils. Le programme et chacun des fils ou petits-fils affiche son rang dans la fratrie, son PID et se termine.

Exercice 9. Modifier le programme pour que le nombre de fils et de petits-fils qu'il crée soient recus en argument de la ligne de commande.

Recouvrement séquentiel

Exercice 10. Écrivez un programme C qui, par le biais de créations de processus et de recouvrements, exécute la suite de commandes `who ; pwd ; ls -l` (rappel : le point-virgule signifie qu'une commande est exécutée lorsque la commande précédente est terminée).

La commande `myif`

Exercice 11. Écrire un programme C `myif.c` qui admet la ligne de commande suivante : `myif commande1 args . . . -then commande2 args . . . [ -else commande3 args . . . ] -fi` Le programme exécute la première commande puis, selon qu'elle aie réussi ou échoué, exécute la deuxième ou la troisième commande.

Rappels

– La fonction `int atoi(char *s);` de `stdlib.h` convertit le début de la chaîne pointée par `s` en entier de type `int`.