Linux : Appels Systèmes - BINV2181-B (fork et exec)

SOLUTION

2.A. Exercice sur l'appel système : `fork`

a) Ecrivez un programme qui définit une variable « a » de type `int` et l'initialise à 5, crée un processus fils, affiche la valeur de retour de l'appel système `fork` . Ensuite, il définit dans le fils une nouvelle variable « b » de type `int`, lui affecte la valeur de « a * 2 » puis affiche le contenu de a et de b ; le père définit une autre variable « b » de type int, lui affecte la valeur de « a * 5 » puis affiche le contenu de a et de b. Pour cet exercice, l'affichage peut se faire avec `printf`.

Cf. Ex2Aa.c

Qu'affiche ce programme lors de son exécution ?

```
la variable pidFils vaut : 9322
dans le pere : a = 5, b = 25
la variable pidFils vaut : 0
dans le fils : a = 5, b = 10
```

ou encore

```
la variable pidFils vaut : 9327
la variable pidFils vaut : 0
dans le fils : a = 5, b = 10
dans le pere : a = 5, b = 25
```

b) Ecrivez un programme qui affiche « trois .. deux .. un .. » avant de créer un fils qui affiche « partez ! ». Les écritures doivent se faire grâce à l'appel système `write`.

Cf. Ex2Ab.c

Qu'affiche ce programme lors de son exécution ?

```
trois .. deux .. un .. partez!
```

c) Ajoutez '\n' à la fin des deux chaînes de caractères.

Qu'affiche ce programme modifié lors de son exécution ?

Cf. Ex2Ac.c

```
trois .. deux .. un ..
partez!
```

d) Refaites les étapes b et c en remplaçant les appels système `write` par l'appel à la fonction `printf`.

Qu'affiche le programme sans '\n' lors de son exécution ?

Cf. Ex2Ad.c

```
trois .. deux .. un .. trois .. deux .. un .. partez!
```

Qu'affiche ce programme, si les chaînes de caractères se terminent par '\n'?

```
trois .. deux .. un ..
partez!
```

e) Y-a-t-il une différence entre les deux versions ? Si oui, justifiez-la puis neutralisez-la.

Cf. Ex2Ae.c

Oui. La sortie standard étant un « character device », le buffer du print n'est vidé que lorsqu'il est plein ou s'il contient un \n. Sans \n, il n'est pas vidé et le fils hérite d'un buffer qui contient déjà « `trois .. deux .. un ..` ».
A la fin du programme, les buffers sont vidés, celui du père qui contient « `trois .. deux .. un ..` » et celui du fils qui contient « `trois .. deux .. un .. partez!` »

RAPPEL : Les buffers sont vidés :
- quand le stream associé au buffer est fermé → p.ex. le processus se termine
- si le buffer est *line buffered*, quand il rencontre une newline → character devices (clavier/écran)
- si le buffer est *full buffered*, quand il est plein → character devices & block devices (fichier sur disque)
- s'il s'agit d'un *output buffer*, lorsqu'il est lu → p.ex. appel système read()

Pour résoudre le problème, seul **`fflush(stdout)`** est valable puisque l'affichage est opéré grâce à printf() ; le buffer à vider est donc celui qui est associé au stream stdout.

Note : **`tcflush(1,TCOFLUSH)`** ne convient pas car il viderait (effacer) le contenu du buffer système.

f) En conclusion, quelle est la différence majeure entre l'appel système `write` et la fonction `printf` ?

L'appel système write n'est pas bufferisé alors que la fonction printf l'est.

2.B. Exercice sur les appels système : fork et wait

a) Ecrivez un programme qui crée un fils qui affiche « 4 5 6\n » alors que le père affiche « 1 2 3\n ».

Cf. Ex2Ba.c

Testez ce programme et expliquez le résultat obtenu.

```
1 2 3
4 5 6
```

Le père et le fils s'exécutent « en parallèle » (ou disons plutôt comme un bon OS multi tâches).

Ici le papa affiche « 1 2 3\n » puis se termine et le fils affiche « 4 5 6\n » et se termine.

b) Modifiez votre programme afin que le fils affiche « 4 5 6\n » avant que le père n'affiche « 1 2 3\n ».

Cf. Ex2Bb.c

```
4 5 6
```

```
1 2 3
```

c) Modifiez votre programme afin que l'enfant se termine par un `exit`, puis affiche l'exit code de ce processus enfant. Consultez le man de `waitpid` pour trouver comment récupérer l'exit code du processus enfant.

d) Réécrivez le programme précédent en utilisant la librairie *utils* (présente sur Moovin). Celle-ci devrait vous faciliter la tâche et augmenter la clarté du code !

Cf. Ex2Bd.c

2.C. Exercice sur les appels système : `fork` et `exec`

Cf. Ex2C.c