Le code des exercices a été rédigé par Elena BERNARD sauf celui du fichier iutBM.c réalisé en partie avec une IA générative.

TP 4

Processus père et fils

Nom du fichier : processus.c

Ce fichier contient la fonction `affiche_car`, ainsi que le programme demandé dans le TP4 dans la partie "Creation de processus".

Q1 : irectement. Cependant, leurs créations s'effectuent "dans l'ordre", c'est a dire que si le dernier processus créer est le 9, alors le suivant sera le 10. Si on connait le numéro du dernier processus créer, on peut alors prévoir les suivants. On peut vérifier cela en créant 100 processus a la chaine (cf. 100 processus fils) et observer que leurs PID se suivent.

Q2 : Ajouter `sleep(5);` dans le programme force le père a attendre que ses fils finissent d'attendre avant de poursuivre son execution.

100 processus fils

Nom du fichier : 100_fils.c

Ce programme créer 100 processus fils à partir du même programme père.

On effectue une boucle de 100 ittérations avec `for (int i = 0; i < 100; i)` A l'intérieur, on utilise `fork()` afin de "diviser" le processus en un père et son fils. On met la valeur de retour de `fork()` dans la variable `pid_t pid`. Celle-ci vaut 0 pour le processus fils et la valeur du pid pour le processus père.

Grace à `switch(pid)` on cherche a savoir si on se trouve dans le pere ou le fils. `case -1:` est un cas d'erreur `case 0` est le cas ou nous sommes dans le processus fils. Le programme affiche le PID et PPID du fils, et retourne 0. `default` est le cas ou nous sommes dans le processus père, et continue à parcourir la boucle.

Avant de mettre fin au programme, on lance 100 fois `wait(NULL)` afin d'attendre que les 100 fils de teminent, et on retourne ensuite 0.

100 processus petits-fils

Nom du fichier : 100_petits_fils.c

Ce programme créer 100 processus fils à la suite les uns des autres.

On effectue une boucle de 100 ittérations avec `while (n <= 100)`. (Ici on utilise un while afin que chaque pere attende la fin de son fils)

A l'intérieur, on utilise `fork()` afin de "diviser" le processus en un père et son fils. On met la valeur de retour de `fork()` dans la variable `pid_t pid`. Celle-ci vaut 0 pour le processus fils et la valeur du pid pour le processus père.

Grace à `switch(pid)` on cherche a savoir si on se trouve dans le pere ou le fils. `case -1:` est un cas d'erreur `case 0` est le cas ou nous sommes dans le processus fils. Le programme affiche le PID et PPID du fils, et continue son execution en incrementant n. `default` est le cas ou nous sommes dans le processus père. Grâce à `wait(NULL)`, le pere attend que son fils se termine, puis retourne 0.

On met fin au programme en retournant 0.

TP 5

Le code étant plus commenté que pour le TP4, les explications dans ce fichier seront plus succinte.

Recouvrement

Nom du fichier : creerterm.c

Ce programme permet de creer un terminal avec xterm sans utiliser `system()` ou `popen()`.

Créer une commande

Nom du fichier : creercommande.c

Ce programme reprend le principe de creerterm, mais utilise le terminal xterm pour lancer une commande passée en paramètre du programme.

Exemple d'execution : `./creercommande ls`

Ouvre un terminal xterm et lance la commande `ls`

Mon systeme

Nom du fichier : mockSystem.c

Ce programme recréer la fonction `system()` présente en C, en faisant appel à d'autre méthodes. `mon_systeme()` est testée ensuite afin de vérifier si elle répond au critères donnés sur cours-info.

TP 6

Interceptions des signaux avec signal

- **Sans réarmement automatique**

Nom du fichier : interception_signal_r-manuel.c

Ce programme utilise la fonction signal afin d'intercepter les signaux produits par `CTRL+C` par exemple. Par défaut la fonction signal se relance d'elle même pour "attraper" d'autre signaux, ici on la relance donc de force.

- **Avec réarmement automatique**

Nom du fichier : interception_signal_r-auto.c

Ce programme utilise la fonction signal afin d'intercepter les signaux produits par `CTRL+C` par exemple. Par défaut la fonction signal se relance d'elle même pour "attraper" d'autre signaux, ici on laisse alors le

programme continuer son cours sans relancer de force.

Ces deux programmes ont une boucle `while` infinie. Une version attendant 10 secondes avant de terminer est disponnible dans les commentaires de chacun de ces programmes, cependant, ceux-ci s'arrêtent apres avoir attrapé un signal.

Interceptions des signaux avec sigaction

- **Sans réarmement automatique**

Nom du fichier : interception_sigaction_r-manuel.c

Ce programme utilise la structure sigaction afin d'intercepter les signaux produits par `CTRL+C` par exemple. Par défaut (sans `flags`) elle se relance d'elle même pour "attraper" d'autre signaux, ici on la relance donc de force en ajoutant `sa.sa_flags = SA_RESETHAND;`.

- **Avec réarmement automatique**

Nom du fichier : interception_sigaction_r-auto.c

Ce programme utilise la structure sigaction afin d'intercepter les signaux produits par `CTRL+C` par exemple. Par défaut (sans `flags`) elle se relance d'elle même pour "attraper" d'autre signaux.

Ces deux programmes ont une boucle `while` infinie. Une version attendant 10 secondes avant de terminer est disponnible dans les commentaires de chacun de ces programmes, cependant, ceux-ci s'arrêtent apres avoir attrapé un signal.

Synchronisation des processus

Nom du fichier : iutBM.c

Le code de ce programme a été réalisé en partie avec l'aide d'une IA générative. Une version "fonctionnelle" avait été initialement rédigée à l'aide de `signal()`, mais celle-ci ne semblait fonctionner que 10% du temps. Après quelques tentative infructueuse de faire fonctionner ce programme de manière fiable, une IA générative a été utilisé afin de "debugger".

L'IA a donc servi à convertir l'utilisation de `signal()` en l'utilisation de `sigaction`. De plus, des modifications ont été apporté au code tels que :

- l'ajout de la fonction `set_signal_handler()` permettant d'utiliser `sigaction`
- l'ajout des lignes `fflush(stdout);` permettant de vider la "file d'attente" des `printf()`
- l'ajout de la gestion d'erreur lors de la création de chaques fils
- l'ajout des commentaires (en anglais) dans le code

Le programme ainsi modifié semble fonctionné de manière plus fiable, bien que toujours incertaine. En effet, il arrive encore que celui-ci ne s'execute pas comme prévu et n'affiche pas les valeurs attendues. Il semble donc clair que la structure même du code est à revoir que celle-ci ne permet visiblement pas la bonne execution de la tâche demandée.