RAPPORT DU DEVOIR n°2

Exercice 1:

1) On compte le processus père 1, un premier fils 1.1 et aussi un deuxième fils 1.2 et un petit fils 1.2.1.

Le processus père exécute le premier fork (avant le &&).

- Si ce fork() retourne une valeur différente de 0 (où une valeur de -1 indique que l'exécution du fork() a échoué et une valeur de 1 indique qu'on est dans le processus père), alors le processus père continue l'exécution de l'expression.
- Sinon, si ce fork() retourne 0, alors cela signifie qu'on est dans un processus fils (ici, c'est le premier fils 1.1 créé par l'éxécution de ce fork() par le père), et dans ce cas, ce processus fils 1.1 n'évaluera pas le reste de l'expression situé après &&, car l'opérateur logique && veut que les deux opérandes soient évaluées à true et le premier opérande est ici évalué à 0, ce qui représente false en C.

A	B	A ou B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A et B
0	0	0
0	1	0
1	0	0
1	1	1

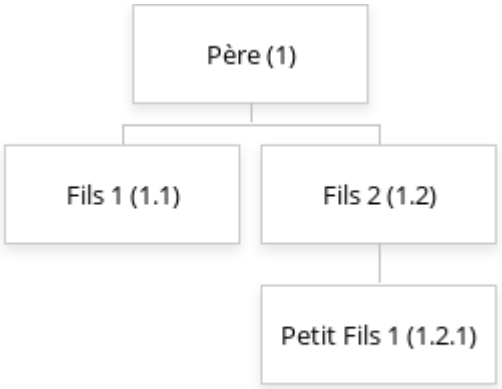
Tables de vérité du OU logique et du ET logique

À ce stade, le processus fils 1.1 est créé. Le processus père continue l'exécution après le &&. En effet, d'après le comportement du && logique, l'expression (fork() || fork()) peut être exécutée uniquement par le processus père (si la valeur retournée par le premier fork() est différente de 0).

- Si le premier fork() au sein de cette expression retourne 0, ce qui implique qu'on est dans le deuxième processus fils du père noté 1.2, ce processus continue l'exécution du fork() situé après le || (comportement de l'opérateur logique ||). Ce processus fils 1.2 (créé par le premier fork() de (fork() || fork())) continue donc l'évaluation et exécute le deuxième fork() dans (fork() || fork()), créant ainsi un fils, noté 1.2.1, qui est donc le petit-fils du processus père initial.

- Sinon, si ce fork() situé avant le || retourne une valeur différente de 0, interprétée comme true en C, indiquant qu'on est toujours dans le processus père initial, alors l'opérateur || court-circuite l'évaluation, et le deuxième fork() de l'expression (fork() || fork()) n'est pas exécuté.

2) L'arbre généalogique des processus créés par cette commande C est :



3) Pour le code zombie.c, on effectue un fork() pour doubler le processus père, on utilise la fonction system("") pour effectuer une commande dans le terminal. Cette commande crée un sous-processus du processus fils pour effectuer la ligne de commande passée en paramètre. On aurait pu utiliser la fonction execl, qui aurait remplacé le processus fils par un processus exécutant la commande composée des paramètres passés à la fonction.

Exercice 2:

Le but de l'exercice est de créer deux processus fils d'un premier processus père en c. Chaque processus doit écrire tour à tour un nombre entier entré par l'utilisateur dans un fichier, ce nombre sera ensuite lu par le processus père. C'est le deuxième processus fils qui doit écrire le deuxième nombre dans un fichier "name2.txt". Ensuite, le premier fils fait de même avec le fichier "name1.txt" et le premier entier tapé par l'utilisateur. Une fois cela effectué, le père lit successivement les deux entiers et détruit les fichiers textes.

Pour commencer, on écrit la fonction void ecrire_fils(int nb, char* name). On déclare un pointeur vers un objet FILE qui désigne le fichier de nom "name". On utilise la fonction fopen(fichier, "w") en mode lecture pour commencer à écrire dans le fichier. On contrôle que l'ouverture se soit bien effectuée. On écrit l'entier "nb" avec la fonction fprintf() puis on ferme le fichier avec fclose().

Ensuite, on écrit la fonction void lire_pere(int*nb, char*name) qui va lire un entier dans un fichier. On déclare aussi un pointeur vers un objet de type fichier avant de l'ouvrir en mode lecture. On vérifie que l'ouverture s'est faite sans problèmes. On conserve la valeur de fscanf() dans un entier pour pouvoir contrôler que la lecture s'est bien faite. Le dernier paramètre de fscanf() est directement "nb" et non son adresse "&nb", car nb est déjà un pointeur d'entier. Si la valeur de renvoi de fscanf() est 1, une erreur s'affiche sinon on affiche l'entier. Cet affichage est utile pour pouvoir suivre le déroulement du programme. La fonction se termine par la fermeture du fichier.

Il s'en suit la fonction entrer_nb qui demande un entier valide (dans le bon format) à l'utilisateur jusqu'à ce que son entrée soit valide. On vide la mémoire tampon dans la boucle while pour que la valeur testée dans la condition while soit actualisée avec la dernière entrée de l'utilisateur.

La fonction void signal_handler(int sig) sert de prétexte pour sortir les différents processus mis en pause dans le main(). En effet, lorsqu'elle est appelée dans la fonction signal(SIGNAL, signal_handler), elle permet de terminer la fonction pause(). Un affichage est tout de même effectué en fonction du type de signal reçu pour pouvoir mieux suivre l'exécution.

Le main() contient dans ses première lignes les entrées des entiers via la fonction enter_nb(), qui sont garantis d'être différents avec une boucle while. Ensuite, on déclare et initialise un entier de type pid_t avec la valeur d'un fork(), pour créer un processus fils du processus initial (status1). S'en suit une condition switch-case qui teste sa valeur: si status1 vaut 0, alors il s'agit du processus fils, sinon (cas default) du père. Les instructions du processus fils sont d'abord un printf pour mieux suivre l'exécution, puis un lien entre SIGUSR1 et signal_handler(), via la fonction signal(). Comme c'est d'abord l'entier 2 qui doit être écrit par le fils 2, ce premier fils doit être mis en attente du deuxième fils. On utilise pour cela la fonction pause(). Une fois qui le processus sera sorti de sa latence, il appelle ecrire_fils.

En ce qui concerne le père, on lui fait exécuter un deuxième fork(), dont la valeur est stockée dans l'entier status2. Le père ne doit pas encore lire quoi que ce soit. Si il appelait maintenant lire_pere(), il n'y aurait aucun fichier dans lequel lire. On écrit un autre switch-case avec les mêmes conditions que le premier. Le fils 2 écrit avec ecrire_fils() puis envoie un signal au premier fils avec kill(status, SIGUSR1). Status est le pid du premier fils. Ce processus s'arrête à la fin de ces instructions. Le signal envoyé est reçu par le premier fils ce qui le sort de sa pause, effectue l'affichage de signal_handler puis appelle ecrire_fils() à son tour, avec le premier entier tapé au clavier. De cette façon l'ordre d'écriture est respecté.

Le père a quant à lui reçu un lien avec le SIGUSR2 pour signal_handler avec la fonction signal. Il est mis en pause, pour attendre que les deux processus fils aient terminé d'écrire. Le premier fils envoie le signal SIGUSR2 via la fonction kill, au pid du père(récupéré donc par getppid() dans les instructions du premier fils).

Lorsque le père reçoit ce signal, il appelle signal_handler et affiche le message correspondant avant de lire les deux entiers dans les deux fichiers distincts et de supprimer les fichiers à chaque lecture.

Exercice 3:

Objectif

Cet exercice illustre l'intérêt d'une exécution parallèle. L'objectif est de diviser une tâche très lourdre (recherche d'un maximum dans un très grand tableau) en sous-tâches indépendantes, traitées simultanément par plusieurs processus, afin de réduire le temps de calcul. Plus précisément, on veut trouver le maximum dans un tableau de grande taille en le divisant en sous-tableaux plus petits jusqu'à atteindre une taille seuil définie comme une constante symbolique **SEUIL** (ici fixée à 10).
Lorsque la taille du sous-tableau est inférieure ou égale à ce seuil, une recherche séquentielle est effectuée. Ensuite, chaque processus fils communique son résultat au processus parent, qui sélectionne enfin le maximum global parmi tous les maxima locaux des subdivisions successives.

Principe de fonctionnement

Le programme repose sur une approche récursive où chaque processus travaille sur une partie du tableau et communique ses résultats au processus parent. Voici les étapes principales :

1. **Division initiale** :
    - Le processus principal divise le tableau en deux moitiés.
    - Il crée deux processus fils : l'un analyse la première moitié, l'autre la seconde.
2. **Récursivité** :
    - Chaque processus fils divise sa portion en deux et crée deux nouveaux processus pour traiter ces moitiés.
    - Ce processus se répète jusqu'à atteindre des sous-tableaux de taille inférieure ou égale à **SEUIL**.
3. **Recherche séquentielle** :
    - Lorsque la taille d'un sous-tableau est inférieure ou égale à **SEUIL**, une recherche séquentielle est effectuée pour trouver le maximum local.
4. **Communication des résultats** :

- Les processus fils communiquent leurs résultats au processus parent en écrivant leurs maxima locaux dans des fichiers temporaires (via la fonction `ecrire_fils`).
- Le processus parent récupère ces valeurs grâce à la fonction `lire_pere` et détermine le maximum global.

Description détaillée des fonctions

1) `maxi(int i, int j)`

La fonction de service `maxi` prend deux entiers en entrée, les compare et retourne le plus grand. Elle est utilisée comme une brique de base dans plusieurs autres fonctions pour comparer des valeurs.

2) `max(int* tab, int debut, int fin)`

La fonction de service `max` effectue une recherche séquentielle pour trouver le maximum dans une partie d'un grand tableau comprise entre les indices `debut` et `fin`. Elle initialise le maximum au premier élément du sous-tableau, puis parcourt les éléments restants en les comparant au maximum courant en appelant la fonction **`maxi`**. Si un élément est plus grand, il devient le nouveau maximum.

3) `ecrire_fils(int nb, char* name)`

Cette fonction permet à un processus de sauvegarder un entier dans un fichier. Elle est utilisée pour transmettre un résultat d'un processus fils à son parent. Le fichier temporaire est nommé par le processus appelant.

4) `lire_pere(int* nb, char* name)`

Cette fonction lit un entier stocké dans un fichier temporaire par un processus fils et supprime le fichier une fois la lecture terminée.

5) `max_parallele(int* tab, int debut, int fin)`

C'est la fonction qui implémente la recherche parallèle. Elle divise le tableau en deux parties et crée deux processus fils pour traiter chaque moitié, et qui eux-même créent des petits fils qui traitent chaque quart, etc. Si le sous-tableau est suffisamment petit (moins de **SEUIL** éléments), elle appelle la fonction `max` pour effectuer une recherche séquentielle. Sinon, elle continue la division récursive en créant de nouveaux processus.

- Le principe est le suivant :
  - Si la taille du sous-tableau dépasse **SEUIL**, le tableau est divisé en deux.
  - Deux processus fils sont créés via des appels à `fork()`, et chacun traite une moitié du sous-tableau.

- Une fois les maxima locaux obtenus par les processus fils, la fonction compare tous ces résultats pour trouver le maximum global.

Les fonctions **`ecrire_fils(int nb, char* name)`** et **`lire_pere(int* nb, char* name)`** assurent la communication inter-processus.

- `ecrire_fils` permet à un processus fils d'écrire son résultat dans un fichier.
- `lire_pere` permet au processus parent de lire ce résultat, tout en supprimant le fichier temporaire une fois la lecture terminée.

→ **`main()`**

La fonction principale crée un tableau de 100 éléments remplis de valeurs aléatoires, puis utilise `max_parallele` pour rechercher le maximum. Enfin, elle affiche le résultat.