Devoir 2 SR01

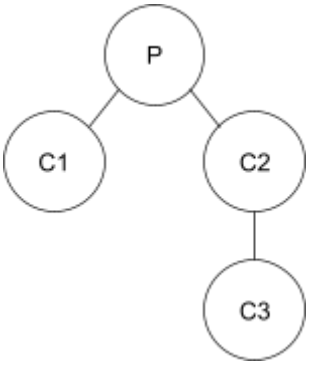
Table des matières :

Exercice 1……2
Question 1……2
Question 2……2
Question 3……2
Exercice 2 ……3
Question 1……3
Question 2……3
Question 3……3
Question 4……3
Question 5……3
Exercice 3……4
Question 1……4
Question 2……4
Question 3……4

Exercice 1 :

1/ Dans une instruction a && b, b n'est pas évaluée si l'évaluation de a donne 0, et dans dans une instruction a || b, b n'est pas évaluée si l'évaluation de a ne donne pas 0. Or, la fonction fork renvoi 0 pour le fils et 1 pour le père, donc dans fork() && b seulement le père exécute b, et dans fork() || b seulement le fils exécute b. Le processus courant va donc engendrer 3 autres processus.

2/ On obtient l'arbre généalogique suivant :



Avec P le processus courant, C1 et C2 les processus créé lors du premier appel de fork et C3 le processus créé lors du dernier appel de fork

3/ Il y a deux interprétations de cette consigne, on peut penser qu'il faut juste faire un fork simple et un fils qui s'endort, mais au vu du nom du programme j'ai supposé qu'il fallait faire un processus zombie. Je vais donner les deux codes dans le même fichier, le cas où le fils ne devient pas un zombie sera commenté.

Cas ou le fils ne devient pas un zombie :
Dans cette fonction on va faire un fork afin de générer le processus fils. On vérifie ensuite le succès de ce fork et en testant la valeur renvoyée par la fonction fork() on attribue les tâches au père et au fils.

Cas ou le fils devient un zombie :
Ici on fait la même chose que dans le programme précédent sauf que cette fois le père va dormir plus longtemps que son fils. Ainsi lorsque le fils se termine, ses informations ne sont pas récupérées correctement et il devient un zombie. On peut le vérifier en tapant la commande “ps aux | egrep "Z|defunct"“ dans un autre terminal 60 secondes après avoir lancé le programme (on a une fenêtre de 5s dès l'affichage de “Le processus fils devient un zombie” pour lancer la commande). Voici ce qui s'affiche :

```
┌──(kali㉿kali)-[~]
└─$ ps aux | egrep "Z|defunct"
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
kali      126037  0.0  0.0      0     0 pts/0    Z+   09:35   0:00 [s] <defunct>
kali      126084  0.0  0.1   6480  2220 pts/1    S+   09:36   0:00 grep -E --color=auto Z|defunct

┌──(kali㉿kali)-[~]
└─$
```

On voit bien que le processus est marqué comme Z, c'est donc bien un zombie.

Exercice 2 :

1/ Dans la fonction on ouvre le fichier avec fopen, on s'assure que l'ouverture c'est bien passée. Si non on renvoie une erreur, si oui on écrit le chiffre dans le fichier à l'aide de la fonction fprintf et on fini par fermer le fichier avec fclose.

2/ Dans la fonction on ouvre le fichier avec fopen, on s'assure que l'ouverture c'est bien passée. Si non on renvoie une erreur, si oui on lit le chiffre dans le fichier à l'aide de la fonction fscanf et on fini par fermer le fichier avec fclose.

3/ On ajoute le bout de code suivant qui va nous permettre de supprimer le fichier du système de fichier à l'aide de la fonction remove et de s'assurer du succès de cette opération.

```
if (remove(name) != 0) {
        perror("Erreur lors de la suppression du fichier");
        exit(EXIT_FAILURE);
}
```

4/ On crée le processus fils à l'aide de fork, on vérifie que l'opération s'est bien passée et on attribue les différentes tâches aux différents processus à l'aide de condition sur le pid.

5/ Le code de cette question se trouve dans le fichier exercice2q5.c pour plus de lisibilité et pour pouvoir l'exécuter plus facilement.
Ici le père va créer un premier fils qui va s'endormir/se mettre en pause instantanément, il va ensuite créer un second fils avant de se mettre lui-même en pause. Le second fils va demander à l'utilisateur d'entrer un nombre puis il va l'écrire dans le fichier "fichier2.txt" à l'aide de la fonction ecrire_fils. Il va ensuite envoyer un signal au premier fils afin de le réveiller. Ce dernier va à son tour demander à l'utilisateur d'entrer un nombre avant de l'écrire dans le fichier "fichier1.txt", à l'aide de la fonction ecrire_fils, avant d'envoyer un signal au père pour le réveiller. Le père va alors lire les valeurs dans les fichiers à l'aide de la fonction lire_pere et va ensuite les afficher à l'écran. On attend la fin des deux processus par mesure de sécurité.
Comme on utilise les handlers simplement pour réveiller les processus qui sont en attente on peut les laisser vide.

Exercice 3 :

1/

```
int maxi(int i, int j) {
    return (i > j) ? i : j;
}
```

2/

```
int max(int* tab, int debut, int fin) {
    int maximum = tab[debut];
    for (int i = debut + 1; i <= fin; i++) {
        if (tab[i] > maximum) {
            maximum = tab[i];
        }
    }
    return maximum;
}
```

3/ On crée la fonction max_parallele(int* tab, int debut, int fin) dans laquelle on exécutera la recherche. Cette dernière s'effectuera de manière récursive.
On commence par vérifier que la partie du tableau dans laquelle on effectue la recherche est inférieure ou égale au seuil. Si oui on fera une recherche séquentielle, si non on divisera la partie en deux et on attribuera chaque sous partie à un processus fils.
Chaque fils calculera le maximum de la partie qui lui est attribuée en appelant la fonction max_parallele et communiquerons le résultat de leur recherche à leur père en utilisant la fonction exit() pour retourner la valeur souhaitée lorsqu'ils se terminent.
Les valeurs passées dans l'appel système exit() sont ensuite récupérées par le processus père à l'aide des fonctions waitpid et WEXITSTATUS.
On renvoie ensuite le maximum des valeurs renvoyées par les fils.

On a ici privilégié la communication par appel système entre les différents processus afin de ne pas avoir de problème de synchronisation (qui pourraient apparaître si les processus communiquent à travers des fichiers temporaires). On aurait également pu choisir de communiquer par des pipes.