Compte rendu – Devoir 2

Exercice 1

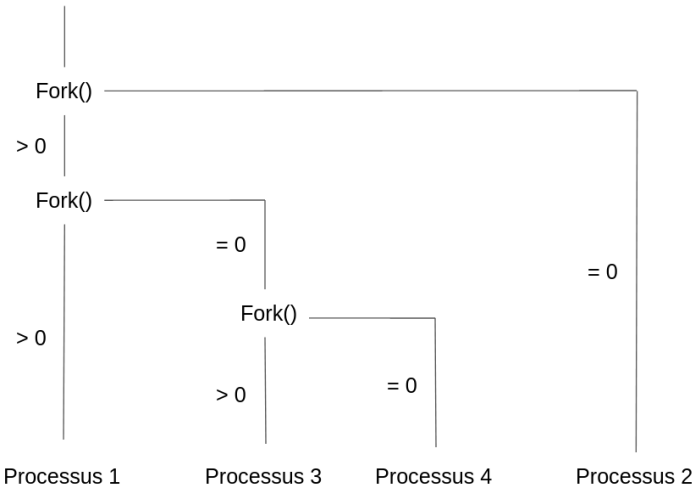
Objectif

Ce premier exercice sert à comprendre le fonctionnement de la fonction `fork` et la gestion de plusieurs processus.

Question 1

Lors de l'exécution, 3 processus fils sont créés (on obtient donc 4 processus au final). En effet, quand le premier `fork` est effectué, il en résulte deux cas : soit on se situe dans le processus fils, la valeur de retour est 0, le reste de l'expression n'est pas évaluée. Soit on se situe dans le processus père, la fonction renvoie alors une valeur positive, on rentre donc dans la deuxième partie de l'opérateur &&. Le raisonnement est le même dans la parenthèse : mais le dernier `fork` est seulement évalué si la première partie de l'opérateur || est nulle. Il en résulte la création de 3 processus fils qui s'ajoutent au processus père initial.

Question 2



Voici un schéma de l'arbre généalogique des différents processus :

Question 3

```c
#include <stdlib.h>      Martin Schneider, 2 weeks ago • zombie.cgit
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid = fork();

    if (pid == -1)
    {
        perror("fork");
        return EXIT_FAILURE;
    }
    if (pid > 0)
    {
        printf("Ici le père %d, au dodo !\n", getpid());
        sleep(60);
        wait(NULL);
        printf("Arret du processus fils\n");
    }
    if (pid == 0)
    {
        printf("Ici le processus fils %d, en vie pour 1 minute\n", getpid());
        exit(EXIT_SUCCESS);
    }
    return EXIT_SUCCESS;
}
```

Le programme commence par effectuer un `fork`, puis grâce au PID renvoyé par la fonction, on peut distinguer les actions à effectuer selon si on est dans le processus fils ou dans le père. Le processus fils affiche simplement un message et termine son exécution grâce à la fonction `exit`. Le processus père, quant à lui se met en veille pendant 60 secondes. Pendant ce temps, même si l'exécution du fils est terminée, ce dernier reste dans la table des processus avec le statut « Z » pour « zombie » (voir figure ci-dessous).

```
martin@martin-Ubuntu:~/Documents/Cours/GI01/SR01/Devoir_2$ ps aux | grep zombie
martin     15018  0.0  0.0   2496   576 pts/6    S+   14:59   0:00 ./zombie
martin     15019  0.0  0.0      0     0 pts/6    Z+   14:59   0:00 [zombie] <defunct>
martin     15039  0.0  0.0  11588   732 pts/0    S+   14:59   0:00 grep --color=auto zombie
```

Exercice 2

Objectif

Nous souhaitons faire communiquer un processus fils avec son processus père. Le premier doit écrire un entier dans un fichier, puis son processus père doit récupérer cet entier en lisant le ficher.

Les codes se situent dans les fichiers ex2_qt4.c et ex2_qt5.c

Sachant que les headers files, les fonctions `ecrire_fils` et `lire_pere` sont identiques dans ces deux fichiers.

Initialisation des headers files

```c
#include <stdlib.h>    // exit()...
#include <stdio.h>     // printf()...
#include <unistd.h>    // getpid()...
#include <sys/types.h> // pid_t
#include <sys/wait.h>  // wait()...
#include <errno.h>     // perror()
```

On définit deux constantes :

- NOMFICHIER qui sera le nom du fichier dans lequel on va écrire et lire le nombre ;
- NB qui sera le nombre qu'on écrira et lira dans le fichier.

Fonction ecrire_fils

```c
void ecrire_fils(int nb, char *name)
{
    FILE *file = fopen(name, "w"); // ouverture du fichier de nom name
    if (file == NULL)              // ouverture mal déroulée
    {
        perror("fopen"); // erreur d'ouverture
        exit(1);         // sortie fonction
    }
    if (fputc(nb, file) == EOF) // écriture de nb dans le fichier ouvert
    {                           // écriture mal déroulée
        perror("fputc");        // erreur d'écriture
        exit(1);                // sortie fonction
    }
    if (fclose(file) == EOF) // fermeture du fichier
    {                        // fichier mal fermé
        perror("fclose");    // erreur de fermeture
        exit(1);             // sortie fonction
    }
}
```

L'objectif de cette fonction est d'ouvrir un fichier de nom `name` donné en paramètres de la fonction, puis d'y écrire le nombre `nb` donné aussi en paramètres de la fonction. On veille à ce que chaque étape se déroule bien, si ce n'est pas le cas on affiche des erreurs et on quitte la fonction.

A savoir :

- La fonction `fopen` retourne un pointeur sur un fichier si l'ouverture s'est bien déroulée sinon elle retourne NULL ;
- La fonction `fputc` retourne le caractère qui vient d'être écrit si l'écriture s'est bien déroulée sinon elle retourne EOF (end of file) ;
- La fonction `fclose` retourne 0 si le flux a bien été fermé sinon elle retourne EOF (end of file).

Fonction lire_pere

```c
void lire_pere(int *nb, char *name)
{
    FILE *file = fopen(name, "r"); // ouverture du fichier de nom name

    if (file == NULL) // ouverture mal déroulée
    {
        perror("fopen"); // erreur d'ouverture
        exit(1);         // sortie fonction
    }
    int n = fgetc(file); // lecture du caractère écrit dans le fichier
    if (n != EOF)        // lecture effectuée sans erreur
    {
        *nb = n; // la valeur pointée par nb prend la valeur récupérée
    }
    if (fclose(file) == EOF) // fermeture du fichier
    {                        // fichier mal fermé
        perror("fclose");    // erreur de fermeture
        exit(1);             // sortie fonction
    }
    //Ajouté à la question 3
    if (remove(name) != 0)     // on détruit le fichier name
    {                          // fichier mal détruit
        perror("remove file"); // erreur de destruction
        exit(1);               // sortie fonction
    }
}
```

L'objectif de cette fonction est de récupérer dans un fichier `name` un caractère, si l'ouverture et la lecture du fichier s'est bien déroulée, ce caractère sera mis dans la variable nb passée en paramètres. On ferme ensuite le fichier, puis on détruit le fichier `name` qui ne nous servira plus (partie ajoutée à la suite de la question 3).

A savoir :

- La fonction `fgetc` retourne le caractère qu'il vient de récupérer si la récupération s'est bien déroulée sinon elle renvoie EOF (end of file) ;
- La fonction `remove` retourne 0 si le fichier a bien été détruit sinon elle retourne -1

Programme principal 1

```c
#define NAME "fichier"
#define NB 5
```

```c
int main(int argc, char const *argv[])
{
    pid_t pid = fork(); // on crée un processus fils à l'aide de fork
    int status;
    switch (pid)
    {
    case -1: // si pid == -1 alors le fork n'a pas fonctionné
        perror("fork");
        return (EXIT_FAILURE);
        break;
    case 0: // si pid == 0 alors on est dans le fils
        printf("Je suis le fils\n");
        printf("    Je vais écrire %d dans le fichier \"%s\"\n", NB, NAME);
        ecrire_fils(NB, NAME); // on écrit NB dans le fichier de nom NAME
        break;
    default:          // sinon pid > 0 alors on est dans le père
        wait(&status); // on attends que son fils retourne un statut
        printf("Je suis le père\n");
        if (status == 0) // si le statut retourné est 0 alors son fils a bien écrit dans le fichier
        {
            printf("    Je vais récupérer le nombre écrit par mon fils dans le fichier \"%s\"\n", NAME);
            int nb;
            lire_pere(&nb, NAME);                                  // on récupère le nombre écrit dans le fichier par son fils
            printf("    Le fils avait écrit %d dans le fichier\n", nb); // affichage du résultat
        }
        else
        {
            printf("    Le fils n'a pas pu écrire dans son fichier, on arrête le programme principal\n");
            return (EXIT_FAILURE);
        }
        break;
    }
    return 0;
}
```

Le premier programme principal demandé à la question 4 est codé dans le fichier **ex2_qt4.c**

Ce programme nous permet de créer un processus fils qui écrira dans le fichier NAME un nombre NB, définis au début comme constantes, ce nombre sera récupéré par son processus père.

On crée un processus fils à l'aide de la fonction `fork`, si on se situe dans le processus fils alors on écrira le nombre NB dans le fichier NAME à l'aide de la fonction `ecrire_fils` précédente. Si on se situe dans le processus père alors on attends la fin du processus fils pour récupérer la valeur qu'il a écrite dans le même fichier à l'aide de la fonction `lire_pere` précédente.

Preuve d'exécution

```
oceane@oceane-Lenovo-Ideapad-S340-15IWL:~/Documents/GI01/SR01/devoir2$ ./ex2_qt4
Je suis le fils
    Je vais écrire 5 dans le fichier "fichier"
Je suis le père
    Je vais récupérer le nombre écrit par mon fils dans le fichier "fichier"
    Le fils avait écrit 5 dans le fichier
```

Programme principal 2

```c
#define NAME1 "fichier1"
#define NB1 7
#define NAME2 "fichier2"
#define NB2 19
```

```c
    else
    {
        int st_fils1, st_fils2;
        if (waitpid(pid2, &st_fils2, 0) == -1) // on attend que le fils 2 renvoie un statut
        {
            perror("waitpid"); // erreur waitpid
        }
        if (st_fils2 == 0) // fils 2 a bien écrit dans son fichier
        {
            printf("On réactive le fils 1\n");
            if (kill(pid1, SIGCONT) == -1) // on réactive le fils 1
            {
                printf("Le fils 1 n'a pas pu être réactivé, on arrête le programme principal\n");
                perror("kill");
                return (EXIT_FAILURE);
            }
        }
        else //
        {
            printf("Le fils 2 n'a pas pu écrire dans son fichier, on arrête le programme principal\n");
            return (EXIT_FAILURE);
        }
        if (waitpid(pid1, &st_fils1, 0) == -1) // on attend que le fils 1 renvoie un statut
        {
            perror("waitpid"); // erreur waitpid
        }
        if (st_fils1 != 0) //le fils 1 a retourné une erreur
        {
            printf("Le fils 1 n'a pas pu écrire dans son fichier, on arrête le programme principal\n");
            return (EXIT_FAILURE);
        }
        printf("Je suis le père (PID=%d)\n", getppid());
        printf("    Je vais récupérer le chiffre écrit par mon fils 1 dans le fichier \"%s\"\n", NAME1);
        int nb1;
        lire_pere(&nb1, NAME1);
        printf("    Le fils 1 avait écrit %d dans le fichier\n", nb1);
        printf("    Je vais récupérer le chiffre écrit par mon fils 2 dans le fichier \"%s\"\n", NAME2);
        int nb2;
        lire_pere(&nb2, NAME2);
        printf("    Le fils 2 avait écrit %d dans le fichier\n", nb2);
    }
    return 0;
}
```

```c
int main(int argc, char const *argv[])
{
    pid_t pid1 = fork(); // on crée un processus fils 1
    int status;
    if (pid1 == -1)
    {
        perror("fork1");
        return (EXIT_FAILURE);
    }
    if (pid1 == 0) // on se situe dans le processus fils 1
    {
        printf("Je stop le fils 1 (PID=%d)\n", getpid());
        if (kill(getpid(), SIGSTOP) == -1) // on stop le fils 1 en attendant que le fils 2 écrive dans son fichier
        {
            printf("Le fils 1 n'a pas pu être stoppé, on arrête le programme principal\n");
            perror("kill");
            return (EXIT_FAILURE);
        }
        printf("Je suis le fils 1 (PID=%d)\n", getpid());
        printf("    Je vais écrire %d dans le fichier \"%s\"\n", NB1, NAME1);
        ecrire_fils(NB1, NAME1); // on écrit NB1 dans NAME1
        exit(0);
    }
    pid_t pid2 = fork(); // on crée un processus fils 2
    if (pid2 == -1)
    {
        perror("fork2");
        return (EXIT_FAILURE);
    }
    if (pid2 == 0) // on se situe dans le processus fils 2
    {
        printf("Je suis le fils 2 (PID=%d)\n", getpid());
        printf("    Je vais écrire %d dans le fichier \"%s\"\n", NB2, NAME2);
        ecrire_fils(NB2, NAME2); // on écrit NB2 dans NAME2
        exit(0);
    }
```

Le second programme principal demandé à la question 5 est codé dans le fichier **ex2_qt5.c**

Ce programme nous permet, à l'aide des signaux SIGSTOP et SIGCONT, de laisser le deuxième fils écrire NB2 dans le fichier NAME2, puis le premier fils écrit NB1 dans le fichier NAME1, le père enfin récupérera les deux nombres.

On crée un premier processus pid1 à l'aide de `fork`, quand on se situe pour la première fois dans le fils 1, on envoie le signal SIGSTOP pour le mettre en pause, quand il sera réactivé il écrira NB1 dans le fichier NAME1 à l'aide de la fonction `ecrire_fils` précédente. On crée ensuite un processus fils pid2 à l'aide de `fork`, il écrit NB2 dans le fichier NAME2 à l'aide de la fonction `ecrire_fils` précédente. Quand on rentre dans le processus père, il attend le signal du fils 2 pour réactiver le fils 1. Quand tout s'est bien déroulé, il ira récupérer les valeurs écrites par ses fils dans les fichiers à l'aide la fonction `lire_pere` précédente.

Preuve d'exécution

```
Je stop le fils 1 (PID=6748)
Je suis le fils 2 (PID=6749)
    Je vais écrire 19 dans le fichier "fichier2"
On réactive le fils 1
Je suis le fils 1 (PID=6748)
    Je vais écrire 7 dans le fichier "fichier1"
Je suis le père (PID=6119)
    Je vais récupérer le chiffre écrit par mon fils 1 dans le fichier "fichier1"
    Le fils 1 avait écrit 7 dans le fichier
    Je vais récupérer le chiffre écrit par mon fils 2 dans le fichier "fichier2"
    Le fils 2 avait écrit 19 dans le fichier
```

Exercice 3

Fonction maxi

Cette fonction très simple compare simplement deux entiers et renvoie le plus grand des deux.

```c
int maxi(int i, int j)
{
    if (i > j) // Si i est supérieur a j, on renvoie i
    {
        return i;
    }
    else // Sinon on renvoie j
    {
        return j;
    }
}
```

Fonction max

La fonction prend en argument le tableau initial, et les bornes sur lesquelles on doit rechercher l'entier maximum.

On crée une boucle qui parcourt le tableau compris entre les bornes. Pour chaque entier, ce dernier est comparé (grâce à la fonction précédente) avec le max temporaire. Si le nombre courant est plus grand alors le max temporaire prend la valeur du nombre courant. Ainsi, une fois le tableau parcouru, on est sûr de renvoyer l'entier le plus grand.

```c
int max(int *tab, int debut, int fin)
{
    int max = tab[debut];
    for (int i = debut; i <= fin; i++) // On parcours le tableau entre les index debut et fin
    {
        max = maxi(max, tab[i]); // max prend la veleur du plus grand entre l
    }
    return max;
}
```

Fonction find_max

```c
int find_max(int *tab, int debut, int fin)
{
    if ((fin - debut + 1) > SEUIL) // On vérifie si la taille du tableau dépasse SEUIL
    {
        int status1, status2;
        int max_debut = tab[debut];
        int max_fin = tab[fin];

        pid_t pid1 = fork(); // On crée le premier fils
        if (pid1 == 0)       // Instructions pour le fils 1
        {
            if (kill(getpid(), SIGSTOP) == -1) // Mise en pause du fils 1
            {
                printf("Un fils n'a pas pu être stoppé\n");
                perror("kill");
                return (EXIT_FAILURE);
            }
            int local_max = find_max(tab, debut, (debut + fin) / 2); // Recherche du max dans la 1ere partie du tableau
            ecrire_fils(local_max, "fichier1");                      // Ecriture du résultat dans le fichier 1
            exit(EXIT_SUCCESS);                                      // Fin du processus fils1
        }
        pid_t pid2 = fork(); // Création du second fils
        if (pid2 == 0)       // Instructions pour le fils 2
        {
            int local_max = find_max(tab, ((debut + fin) / 2) + 1, fin); // Recherche du max dans la 2eme partie du tableau
            ecrire_fils(local_max, "fichier2");                          // Ecriture du résultat dans le fichier 2
            exit(EXIT_SUCCESS);                                          // Fin du processus fils2
        }
        else // Instructions pour le procesus père
        {
            if (waitpid(pid2, &status2, 0) == -1) // Attend la fin de l'éxécution du fils 2
            {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }
            if (status2 != 0) // Vérifie que l'éxécution du fils 2 s'est déroulée correctement
            {
                printf("Un fils a retourné un statut d'erreur\n");
                exit(EXIT_FAILURE);
            }
            int max_fin;
            lire_pere(&max_fin, "fichier2"); // Lit la valeur transmise par le fils 2

            if (kill(pid1, SIGCONT) == -1) // Reactive l'éxécution du fils 1
            {
                printf("Un fils n'a pas pu être réactivé\n");
                perror("kill");
                return (EXIT_FAILURE);
            }
            if (waitpid(pid1, &status1, 0) == -1) // Attend la fin de l'éxécution du fils 1
            {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (status1 != 0) // Vérifie que l'éxécution du fils 1 s'est déroulée correctement
            {
                printf("Un fils a retourné un statut d'erreur\n");
                exit(EXIT_FAILURE);
            }

            int max_debut;
            lire_pere(&max_debut, "fichier1"); // Lit la valeur transmise par le fils 1

            return maxi(max_debut, max_fin); // Compare les deux valeurs obtenues et renvoie la plus grande
        }
    }
    else
    {
        return max(tab, debut, fin); // Renvoit le résultat de la recherche séquentielle dans le tableau de taille inférieure a SEUIL
    }
```

Pour rechercher le maximum dans un tableau de grande taille, nous avons choisi d'utiliser la récursivité pour décomposer progressivement le tableau en sous tableaux de plus petite taille.

La fonction `find_max` commence par déterminer si la taille du tableau tab dépasse la taille de SEUIL définie en constante. Si elle est inférieure à SEUIL, on fait simplement appel à la fonction `max` définie précédemment pour rechercher séquentiellement le maximum. Sinon, il faut décomposer la résolution du problème.

On reprend ici le principe de la question 5 de l'exercice 2.

Deux fils sont créés, le premier est mis en pause le temps de l'exécution du second. A chaque fils est confiée une moitié de tableau à traiter. Le premier fils reçoit la partie comprise entre l'index début et l'index (début + fin) / 2, le second reçoit la partie entre ((début + fin) / 2) + 1 et fin.

Ces deux fils créent une variable maximum local qui correspond à un nouvel appel de la fonction `find_max` sur leur partie de tableau. Si leur partie est toujours plus longue que SEUIL, une nouvelle division du tableau sera effectuée. Une fois que le fils 2 a récupéré son maximum local, il le transmet au processus père grâce à la fonction `ecrire_fils` et termine son exécution.

On rentre alors dans le processus père qui, après avoir récupéré le statut de fils 2 avec `waitpid`, lit la valeur écrite par le fils 2 avec la fonction `lire_pere` puis réactive le fils 1 à l'aide du signal SIGCONT. Le fils 1 effectue les mêmes actions que le fils 2 sur sa partie de tableau et stoppe son exécution. Le père peut ainsi comparer les deux valeurs obtenues et renvoyer la plus grande à l'aide de la fonction `maxi`.

Test des fonctions

Pour tester nos fonctions, on crée un tableau aléatoire d'une taille définie dans les constantes grâce à la fonction `random_int`. Ce tableau contient exclusivement des valeurs comprises entre 1 et 100.

```c
int random_int()
{
    return (rand() % MAX);
}

int main()
{
    srand(time(NULL));
    int tab[LEN];
    for (int i = 0; i < LEN; i++)
    {
        tab[i] = random_int();
        printf("%d\n", tab[i]);
    }
    printf("\nMaximum global du tableau : %d\n", find_max(tab, 0, LEN - 1));
    return 0;
}
```