Louka CHAPIRO - GI01
Soulaymane KEBLI - GI01

# Université de Technologie de Compiègne

## SR01 - Maîtrise des Systèmes Informatiques

Responsable
Mr. Lakhlef Hicham

---

# Devoir 2 - Processus et Programmation système

---

Responsables TD
Mr. Lakhlef Hicham
Mme. Ghada Jaber

Groupe n°7, Vendredi 10h15

# Sommaire

# Table des figures

# 1 Introduction

L'informatique système, de par sa complexité et sa diversité, requiert une compréhension approfondie des processus et de la gestion de fichiers. Ce rapport se penche sur un ensemble d'exercices clés, offrant une plongée au cœur des fondements de la programmation système. Ces exercices abordent des concepts cruciaux tels que la création et la gestion des processus, la communication entre processus et la manipulation de fichiers, constituant ainsi un pilier essentiel dans la compréhension de la dynamique des systèmes informatiques.

Le premier volet de ce rapport explore la création et la généalogie des processus, mettant en lumière les intrications subtiles entre les appels système fork() et les mécanismes de génération de processus. L'analyse approfondie de ces mécanismes permet de dévoiler la structure complexe des processus engendrés par des instructions telles que

```
1 fork() && (fork() || fork())
```

Ensuite, l'accent est mis sur la communication inter-processus à travers la manipulation de fichiers. Les exercices suivants requièrent une compréhension pointue des mécanismes d'écriture et de lecture de fichiers entre processus parents et enfants, exploitant les fonctionnalités offertes par le système d'exploitation pour échanger des données de manière sécurisée et efficace.

Enfin, l'exercice culminant présente une application pratique des notions précédentes. Il s'agit de la parallélisation de tâches, où plusieurs processus sont mobilisés pour la recherche de l'élément maximum dans un tableau de données volumineux. Cette approche met en exergue l'optimisation des performances en répartissant la charge de travail entre plusieurs processus, illustrant ainsi l'essence même de l'exécution parallèle dans le domaine de l'informatique système.

À travers ces exercices, ce rapport vise à explorer et à consolider les connaissances fondamentales en programmation système, offrant une plongée pratique et conceptuelle dans la gestion des processus et des fichiers, éléments essentiels à tout environnement informatique robuste et performant.

# 2 Exercice1 : Manipulation de fork() et de fichier zombie

L'exercice initial explore la création de processus via l'instruction 'fork()' dans un contexte logique complexe, mettant en jeu des opérateurs logiques. L'analyse de l'instruction fork() && (fork() || fork()) dévoile la génération de processus et leur hiérarchie, offrant ainsi un aperçu de la complexité des opérations concurrentes dans un système informatique.

## 2.1 Question 1 - Nombre de processus créés

Soit l'instruction C suivante :

```
1                    fork() && (fork() || fork())
```

Nous allon la décomposer afin de déterminer le nombre de procéssus créés.

Le premier fork() crée un processus. Donc, au début, il y a 2 processus (le processus parent initial et le processus enfant résultant du premier fork()). Le premier fork() renverra au processus parent son identifiant non nul, tandis qu'il renverra 0 au processus enfant.

Dans le processus parent initial, il y a une évaluation &&. Cela signifie que le premier fork() de l'expression logique sera évaluée à TRUE (1) dans le processus parent, tandis que dans le processus enfant, elle sera évaluée à FALSE (0) et n'appellera pas les deux fork() restants.

Maintenant, le 2ème fork() dans le processus parent va être exécuté, créant un nouveau processus. il renvoie une valeur non nulle au processus parent et une valeur nulle dans le processus enfant qui vient d'être crée. Ainsi, le processus parent ne continuera pas l'exécution jusqu'au dernier fork() (à cause du court-circuit) [1], alors que le processus enfant exécutera le dernier fork, puisque le premier opérande de || est 0.

Dans le processus enfant résultant du deuxième fork(), il y a donc une évaluation ||. Cela signifie qu'au moins un des deux fork() sera exécuté. Dans ce cas, les deux fork() sont exécutés, créant deux nouveaux processus. Donc, dans ce processus enfant, il y a 2 processus supplémentaires, ce qui donne **un total de 4 processus**.

Pour mieux comprendre cela, imaginons que nous donnons une tâche à tous ces processeurs, comme une simple demande d'affichage, afin de voir le nombre de fois que cette tâche est exécutée.
Prenons le code C suivant :

```
1                int main(){
2                    fork() && (fork() || fork());
3                    printf("Hello world !\n");
4                }
```

1. Ici, le court-circuit signifie essentiellement que si le premier opérande de && est zéro, le ou les autres opérandes ne sont pas évalués. Dans la même logique, si un opérande d'un || est 1, les autres opérandes n'ont pas besoin d'être évalués. En effet, les autres opérandes ne peuvent pas modifier le résultat de l'expression logique, ils n'ont donc pas besoin d'être exécutés, ce qui permet de gagner du temps.

En supposant que tout se passe bien dans la création de chaque processus , nous aurons alors le schéma suivant :

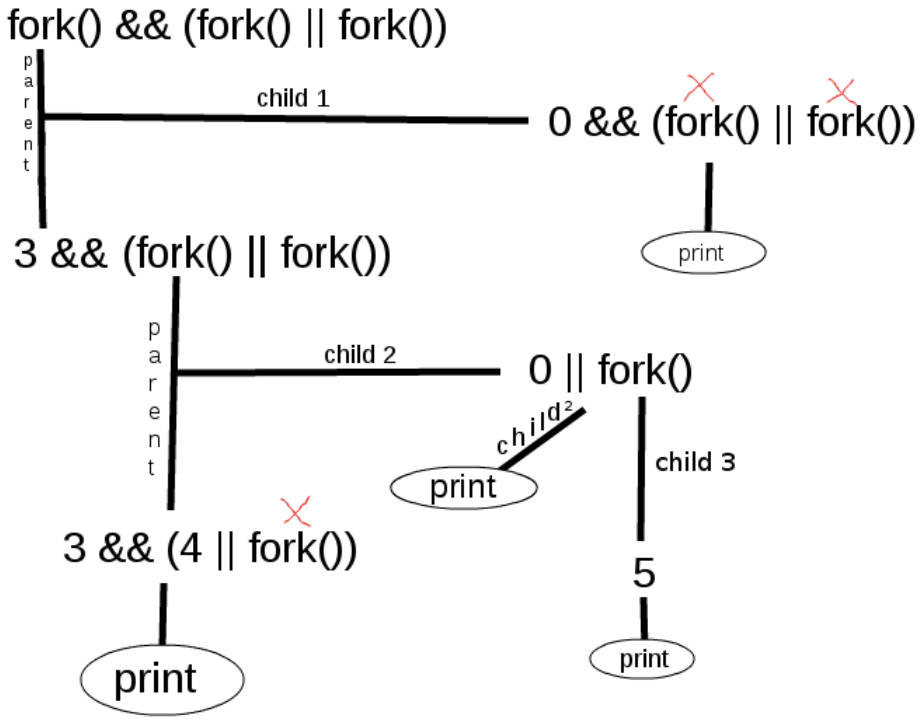


FIGURE 1 – Décomposition de l'instruction C

Dans ce schéma, nous pouvons observer les différents processus créés (parents et enfants), ainsi que les fork() non exécutés par le processus en cours (lorsqu'il y a une croix rouge cela signifie que le fork n'est pas exécuté). Nous avons également affecté des valeurs au processus (leur pid) pour bien comprendre l'instruction.
Enfin, on peut s'apercevoir qu'il n'y a seulement 4 print.

## 2.2 Question 2 - Arbre généalogique des processus

Après toutes les explications précédentes et le schéma ci-dessu, nous obtenons l'arbre généalogique des processus créés suivant :

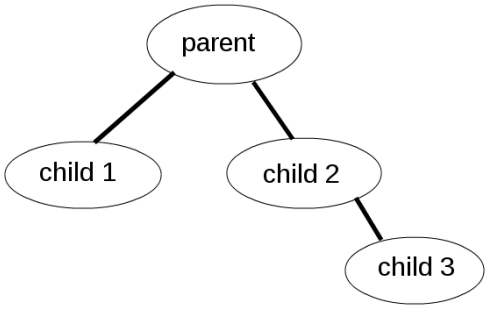


FIGURE 2 – Arbre généalogique des processus créés

## 2.3 Question 3 - Création d'un fichier zombie.c

Voici le code situé dans le fichier zombie.c du dossier :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void zombie() {
    pid_t pid;

    // Creation du processus enfant
    pid = fork();
    switch (pid) {
        case -1:
            // Erreur lors de la creation du processus enfant
            fprintf(stderr, "Erreur lors de la creation du processus enfant\n");
            exit(EXIT_FAILURE);
        case 0:
            printf("Processus enfant : Je suis ici et je vais dormir pendant 1 minute.\n");
            sleep(60); // Le processus enfant dort pendant 60 secondes
            printf("Processus enfant : Je me reveille maintenant.\n");
            exit(0);
        default:
            // Code execute par le processus parent
            printf("Processus parent : Le processus enfant a ete cree avec le PID : %d\n", pid);
            // Attente de la fin du processus enfant
            waitpid(pid,NULL,0);
            printf("Processus parent : Le processus enfant s'est termine.\n");
    }

}
```

Ce programme illustre la création d'un processus enfant à l'aide de la fonction fork() et l'utilisation de waitpid() pour attendre la terminaison du processus enfant.

Lors de son exécution, le programme commence par appeler fork(), une fonction qui crée un nouveau processus en dupliquant le processus appelant. En fonction de la valeur renvoyée par fork(), le code suit différentes branches. Si fork() renvoie -1, indiquant une erreur lors de la création du processus enfant, le programme affiche un message d'erreur approprié et termine de manière anormale. Si fork() renvoie 0, cela signifie que le code est exécuté dans le contexte du processus enfant nouvellement créé. Dans ce cas, le processus enfant affiche un message indiquant son statut, dort pendant 60 secondes à l'aide de sleep(60), puis se réveille pour afficher un autre message avant de se terminer avec exit(0).

Dans le cas où fork() renvoie une valeur différente de -1 et 0, cela signifie que le code est exécuté dans le contexte du processus parent. Le parent affiche le PID du processus enfant nouvellement créé, puis utilise waitpid() pour attendre la fin du processus enfant. Pendant cette attente, le processus parent se bloque jusqu'à ce que le processus enfant se termine. Une fois que le processus enfant est terminé, le parent affiche un message indiquant que le processus enfant s'est terminé.

En résumé, ce programme illustre la création d'un processus enfant, sa mise en sommeil, son réveil, et la manière dont le processus parent attend la fin de son exécution avant de se terminer à son tour. L'utilisation de waitpid() ici garantit que le processus parent ne se termine pas avant que son enfant n'ait terminé son exécution.

# 3 Exercice 2 - Communication entre père et fils

Dans cet exercice l'objectif est de faire communiquer un processus fils avec son processus père. Le processus fils doit écrire un entier dans un fichier, puis le processus père doit récupèrer cet entier en le lisant dans le fichier.

## 3.1 Question 1 - Écriture d'entier dans un fichier

Voici la fonction demandée :

```
1 void ecrire_fils(int nb, char* name){
2     FILE *file = fopen(name,"wb");
3     if (file==NULL){
4         perror("Erreur dans l'ouverture du fichier");
5         exit(0);
6     }
7     fprintf(file,"%d",nb);
8     fclose(file);
9 }
```

La fonction ecrire_fils permet d'écrire un nombre nb à l'intérieur d'un fichier de nom name. On utilise la fonction fopen() pour ouvrir le fichier en mode écriture, en spécifiant "wb", pour ouvrir en mode binaire[2]. On gère les problèmes liés à l'ouverture du fichier à l'aide d'un if. Si la variable file créé est NULL, on lève une erreur à l'aide de la fonction perror(). Si la création a fonctionné, on utilise la fonction fprintf() pour écrire le nombre nb dans le fichier. Enfin on ferme le fichier à l'aide de la fonction fclose().

2. Utile pour que les données ne soient pas modifiées lors de la lecture

## 3.2 Question 2 - Lecture d'entier dans un fichier

Voici la fonction demandée :

```
int lire_pere(int* nb, char* name) {
    FILE *file = fopen(name, "r");
    if (file == NULL) {
        perror("Erreur dans l'ouverture du fichier");
        exit(0);
    }
    if (fscanf(file, "%d", nb) != 1) {
        perror("Erreur dans la lecture du nombre par le
           pere");
        fclose(file);
        return 0;
    }
    fclose(file);
    return 1;
}
```

La fonction lire_pere a le fonctionnement inverse de la fonction précédente. Elle lit un nombre dans un fichier de nom name et place la valeur dans une variable nb. On utilise la fonction fopen() pour ouvrir le fichier en mode lecture, en spécifiant "r". On gère une erreur lors de l'ouverture de la même façon que dans la fonction précédente. On utilise la fonction fscanf() sur le fichier pour lire la valeur et la placer dans la variable nb. Cette fonction retourne le nombre de valeur affecté, s'il est différent de 1, cela signifie que la lecture à échoué et une erreur est levée. Ensuite le fichier name est fermé. La fonction retourne 1 en cas de réussite et 0 en cas d'échec.

## 3.3 Question 3 - Destruction après lecture

Voici la fonction précédente modifiée :

```
int lire_pere(int* nb, char* name) {
    FILE *file = fopen(name, "r");
    if (file == NULL) {
        perror("Erreur dans l'ouverture du fichier");
        exit(0);
    }
    if (fscanf(file, "%d", nb) != 1) {
        perror("Erreur dans la lecture du nombre par le
           pere");
        fclose(file);
        return 0;
    }
    fclose(file);
    if (remove(name) != 0) {
        return 0;
    }
    return 1;
}
```

On reprend la fonction lire_pere précédente. Cependant, il est désormais nécessaire de supprimer le fichier à la fin du programme. On utilise la fonction remove() après le fclose(). La fonction remove() permet de supprimer un fichier en fonction de son nom. Si le fichier est bien supprimé la fonction remove() retourne 0, si ce n'est pas le cas il y a eu une erreur, la fonction lire_pere retourne donc 0.

## 3.4 Question 4 - Résolution du problème sans signaux

Avant de répondre à cette question, nous devons introduire une fonction utilitaire qui nous servira pour tout le reste du rapport à partir d'ici. Voici la fonction :

```
1 //fonction qui genere un nom de fichier a partir d'un pid
2 char* nom_fichiers(pid_t pid, char cote){
3     char* nom_fichier = malloc(15*sizeof(char));//cree une
          chaine de 15 caracteres
4     sprintf(nom_fichier, "%d%c.txt", pid,cote);
5     //Le nom du fichier contient l'id processus pere plus g
          ou d si c'est le fils droit ou gauche
6     return nom_fichier;
7 }
```

Cette fonction permet de générer dynamiquement des noms de fichiers uniques basés sur l'ID d'un processus et un caractère indiquant le côté, facilitant ainsi la création et la gestion de fichiers spécifiques pour différents processus ou tâches, par exemple pour stocker des données ou des résultats spécifiques à chaque processus comme nous allons faire dans la suite du rapport.
Voici maintenant la fonction qui résout le problème de l'exercice :

```
1  int nb1, nb2;//variable globale
2  int ex2_1() {
3      printf("Entrez un nombre : ");
4      scanf("%d",&nb1);
5
6      int nb1_p;
7      pid_t pid;
8      pid = fork();
9      switch (pid) {
10         case -1:
11             printf("erreur dans la creation du processus");
12             exit(EXIT_FAILURE);
13         case 0:
14             ecrire_fils(nb1, nom_fichiers(getppid(),'g'));
15             exit(0);
16         default:
17             waitpid(pid,NULL,0);
18             lire_pere(&nb1_p,nom_fichiers(getpid(),'g'));
19             printf("Nombre recupere par le pere :
                   \033[1m%d\033[0m\n",nb1_p);
20     }
21     return 0;
22 }
```

L'objectif de cette fonction est qu'un processus fils écrit un nombre dans un fichier name et que ce nombre soit lu par le processus père.

On commence par demander à l'utilisateur un nombre à l'aide d'un printf() et d'un scanf(). Ensuite on crée un processus fils à l'aide de la fonction fork().
On réalise un switch sur pid :

- On gère l'erreur (égale à -1),
- Pour le processus fils (pid = 0), on utilise la fonction ecrire_fils() sur le fichier généré par la fonction nom_fichiers().
- Pour le processus père (pid supérieur à 0), on utilise la fonction waitpid() pour qu'il attende la fin d'exécution du processus fils. Ensuite, il utilise la fonction lire_pere() pour récupérer la valeur écrite par le processus fils. Enfin cette valeur est affichée à l'aide d'un printf().

## 3.5 Question 5 - Résolution du problème avec signaux

Voici les fonctions pour répondre aux problème avec l'utilisation des signaux :

```
1  int ex2_2() {
2      pid_t pid1, pid2;
3      int nb1_p, nb2_p;
4      printf("Entrez un premier nombre : ");
5      scanf("%d",&nb1);
6      printf("Entrez un second nombre : ");
7      scanf("%d",&nb2);
8
9      // Crzation du premier fils
10     pid1 = fork();
11
12     switch (pid1) {
13         case -1:
14             perror("Erreur dans la crzation du premier
                   processus fils");
15             exit(1);
16         case 0:// Code du premier fils
17             signal(SIGUSR1, handle_signal);
18             // Attache le signal SIGUSR1 au gestionnaire de
                   signaux
19             pause(); // Attends jusqu'a la reception du signal
20             exit(0);
21         default:// Code du processus parent
22             pid2 = fork();// Creation du deuxieme fils
23             switch (pid2) {
24                 case -1:
25                     perror("Erreur dans la creation du
                           deuxieme processus fils");
26                     exit(1);
27                 case 0:
28                     signal(SIGUSR2, handle_signal);
```

```
29                      // Attache le signal SIGUSR2 au
                           gestionnaire de signaux
30                      pause(); // Attends jusqu'a la reception
                           du signal
31                      exit(0);
32                  default:// Code du processus parent
33                      sleep(1);
34                      // Attente d'une seconde pour etre sur
                           que les fils sont prets
35
36                      // Envoi des signaux aux fils pour qu'ils
                           ecrivent dans les fichiers
37                      kill(pid1, SIGUSR1);
38                      kill(pid2, SIGUSR2);
39
40                      // Attente de la fin des fils
41                      waitpid(pid1,NULL,0);
42                      // Lecture des fichiers et recuperation
                           des valeurs
43                      lire_pere(&nb1_p,nom_fichiers(getpid(),'g'));
44                      waitpid(pid2,NULL,0);
45                      lire_pere(&nb2_p,nom_fichiers(getpid(),'d'));
46
47                      printf("Nombre recupere par le pere
                           depuis le fichier %s
                           :\033[1m%d\033[0m\n",
                           nom_fichiers(getppid(),'g'), nb1_p);
48                      printf("Nombre recupere par le pere
                           depuis le fichier %s
                           :\033[1m%d\033[0m\n",
                           nom_fichiers(getppid(),'d'), nb2_p);
49              }
50              return 0;
51      }
52  }
```

Avec comme fonction complémentaire au programme ci-dessus :

```
1  void handle_signal(int signum) {
2      if (signum == SIGUSR1) {
3          printf("Signal du premier fils recu.\n");
4          ecrire_fils(nb1,nom_fichiers(getppid(),'g'));
5      } else if (signum == SIGUSR2) {
6          printf("Signal du deuxieme fils recu.\n");
7          ecrire_fils(nb2,nom_fichiers(getppid(),'d'));
8      } else {
9          printf("Signal inattendu recu.\n");
10         return;
11     }
12 }
```

L'objectif de cette question est d'utiliser le mécanisme des signaux afin de gérer deux processus fils.

Cette fois-ci on demande donc deux valeurs à l'utilisateur à l'aide de printf() et scanf(). On crée un premier fils, noté pid1. On utilise l'instruction signal() pour que le premier fils exécute la fonction handle_signal() lorsqu'il reçoit le signal SIGUSR1. Avec la fonction pause(), le premier fils est mis en pause jusqu'à réception du signal SIGUSR1. Ensuite un second processus pid2 est créé par le processus père. Cette fois-ci il est mis en attente du signal SIGUSR2.

Enfin, le processus père envoie les deux signaux à ses fils à l'aide de la fonction kill(). La fonction handle_signal() utilise la fonction ecrire_fils pour écrire les deux valeurs dans deux fichiers textes générés par la fonction nom_fichiers(). Dans le cas du signal SIGUSR1, la valeur 1 est écrite dans le fichier 1 et dans le cas du signal SIGUSR2, la valeur 2 est écrite dans le fichier 2.

Le processus père attend la fin des deux processus fils grâce à la fonction waitpid() et récupère les valeurs grâce à la fonction lire_pere().

# 4 Exercice 3 - Exécution en parallèle

Dans cette partie nous allons réaliser un programme qui cherche l'élément maximum d'un très grand tableau en utilisant plusieurs processus.

## 4.1 Question 1 - Max entre deux entiers

Voici la fonction qui retourne le maximum entre deux entiers i et j :

```
1 //fonction qui retourne le max entre deux valeurs
2 int maxi(int i, int j){
3     return i>j ? i : j;
4 }
```

Cette fonction utilise une structure de condition ternaire, exprimée comme suit : i > j ? i : j. Cette structure vérifie si i est supérieur à j. Si c'est le cas, la fonction retourne i, sinon elle retourne j. En d'autres termes, cette fonction effectue une comparaison entre i et j, et retourne la valeur la plus grande des deux.

## 4.2 Question 2 - Recherche séquentielle d'un max

Voici la fonction qui retourne le maximum d'un tableau dans intervalle précisé trouvé séquentiellement :

```
1 //fonction qui retourne le max d'un tableau dans un
    intervalle precise
2 int max(int* tab, int debut, int fin){
3     if (debut>fin)
4         return -1;
5     if (debut==fin)
6         return tab[debut];
7
```

```
8       int max = 0;
9       for (int i = debut; i <= fin; ++i) {
10          max =  tab[i]>max ? tab[i] : max;
11      }
12      return max;
13  }
```

La fonction commence par vérifier deux conditions pour déterminer si l'intervalle donné est valide. Si debut est supérieur à fin, cela signifie que l'intervalle est incorrect et la fonction retourne alors -1 pour signaler une erreur. Si debut est égal à fin, cela indique un intervalle d'un seul élément dans le tableau, donc la fonction retourne la valeur de cet élément.

Ensuite, si l'intervalle contient plus d'un élément, la fonction initialise une variable max à zéro. Elle parcourt ensuite chaque élément du tableau entre les indices debut et fin inclus grâce à une boucle for. À chaque itération, elle utilise la fonction maxi (qui retourne le maximum entre deux valeurs) pour comparer l'élément du tableau à l'élément actuel maximum (max), mettant à jour max avec la valeur la plus grande trouvée jusqu'à présent dans l'intervalle.

Après avoir parcouru tous les éléments de l'intervalle, la fonction retourne la valeur maximale trouvée.

## 4.3 Question 3 - Division de tâches

Voici le code que nous avons écris pour résoudre ce problème :

```
1  #define SEUIL 3
2  //seuil pour savoir si on fait en division de processeur ou
       si on fait une recherche sequentiel
3  #define N_MAX 1000
4  //nombre max d'element pour tableau
5
6  int parallel_max(int arr[], int debut, int fin) {
7  // Si la taille de la partie du tableau est inferieure au
       seuil, recherche sequentielle
8       if (fin - debut <= SEUIL) {
9           return max(arr, debut, fin);
10      }
11
12      int mid = (debut + fin) / 2;
13
14      int left_max, right_max;
15      pid_t left_child, right_child;
16       // Creer un processus fils pour la premiere moitie du
           tableau
17      left_child = fork();
18      switch (left_child) {
19          case -1:
20              perror("erreur avec le fork");
21              exit(EXIT_FAILURE);
```

```
22          case 0:
23              // Code pour le fils gauche
24              left_max = parallel_max(arr, debut, mid);
25              ecrire_fils(left_max, nom_fichiers(getppid(),'g'
                    ));
26              // ecrire le maximum dans le fichier
27              exit(0);
28          default:
29          // Creer un processus fils pour la deuxieme moitie du
                tableau
30              right_child = fork();
31              switch (right_child) {
32                  case -1:
33                      perror("erreur avec le deuxieme fork");
34                      exit(EXIT_FAILURE);
35                  case 0:
36                      // Code pour le fils droit
37                      right_max = parallel_max(arr, mid + 1,
                            fin);
38                      ecrire_fils(right_max,
                            nom_fichiers(getppid(),'d'));
39                      // ecrire le maximum dans le fichier
40                      exit(0);
41                  default:
42                      // Code pour le parent
43                      waitpid(left_child, NULL, 0);
44                      // Lire le maximum ecrit par le fils
                            gauche
45                      lire_pere(&left_max,nom_fichiers(getpid(),'g'));
46                      waitpid(right_child, NULL, 0);
47                      // Lire le maximum ecrit par le fils droit
48                      lire_pere(&right_max,
                            nom_fichiers(getpid(),'d'));
49                      return maxi(left_max,right_max);
50              }
51      }
52  }
```

Expliquons maintenant ce programme :

Tout d'abord, le programme vérifie si la différence entre fin et debut est inférieure ou égale à une constante prédéfinie SEUIL, ici égale à 3. Si c'est le cas, la taille de la partie du tableau est considérée comme petite, et le programme effectue une recherche séquentielle du maximum dans cet intervalle en appelant la fonction max.

Si la taille de l'intervalle dépasse le seuil, le programme divise le travail en deux parties égales. Il calcule alors le milieu de l'intervalle avec mid = (debut + fin) / 2.

Ensuite, il crée deux processus fils à l'aide de la fonction fork(). Le premier fils est responsable de la première moitié du tableau, tandis que le deuxième fils s'occupe de la deuxième moitié.

Le fils gauche exécute récursivement la fonction parallel_max sur la première moitié du tableau, trouve le maximum, puis écrit cette valeur dans un fichier spécifique en utilisant la fonction ecrire_fils.

Le fils droit effectue une opération similaire sur la deuxième moitié du tableau, trouvant le maximum de cette partie et l'écrivant également dans un fichier distinct.

Pendant ce temps, le processus parent attend la fin de l'exécution des fils gauche et droit à l'aide de waitpid(). Une fois les deux fils terminés, le parent lit les valeurs maximales écrites par les fils dans les fichiers correspondants en utilisant lire_pere(). Enfin, il retourne le maximum entre les valeurs obtenues des fils gauche et droit en utilisant la fonction maxi().

Pour conclure, ce programme divise récursivement le travail sur le tableau en utilisant la programmation parallèle, séparant le travail entre les processus fils, collectant les résultats, et finalement, trouvant le maximum global à partir des maximums partiels obtenus par chaque processus fils.

Nous avons ensuite testé ce programme en initialisant un tableau de 1000 caractères par des entiers générés aléatoirement. Nous affichons ensuite les entiers à l'aide de la fonction maxi() (recherche séquentielle) et à l'aide de la fonction parallel_max() pour bien vérifier que nous obtenons la même chose.

```
1 int ex3(){
2     int tab[N_MAX];
3     for (int i = 0; i < N_MAX; ++i) {
4         tab[i] = (int) rand();
5         // Remplir le tableau avec des nombres aleatoires
              entre 0 et 99
6     }
7     printf("Max trouver sequentiellement avec la fonction
          maxi : \033[1m%d\033[0m\n", max(tab,0,N_MAX-1));
8     printf("Max trouver a l'aide de la division de tache :
          \033[1m%d\033[0m\n",parallel_max(tab,0,N_MAX-1));
9 }
```

# 5 Main du devoir

Nous avons rédigé une fonction main() qui permet de tester toutes les fonctions de tous les exercices de ce devoirs à l'aide d'un menu interactif. Cette fonction est disponible dans le fichier main.c. Nous ne l'afficherons pas dans ce rapport étant donnée sa densité mais il reste accessbile dans le dossier .zip du devoir.

# 6 Conclusion

La programmation système constitue un pilier fondamental dans la construction d'applications robustes et efficaces. À travers les différents exercices abordés, ce rapport a exploré en profondeur les concepts clés liés aux processus, à la gestion de fichiers et à la programmation parallèle.

L'étude de la création de processus, la manipulation de fichiers et la communication inter-processus a permis de saisir l'importance de ces éléments dans la construction d'applications complexes. La compréhension des mécanismes de création de processus avec fork(), la communication entre processus via des fichiers, et l'utilisation de mécanismes tels que wait()/waitpid() a été cruciale pour appréhender la coordination et la synchronisation entre différentes tâches.

Par ailleurs, l'exploration de la programmation parallèle a dévoilé les avantages et les défis liés à la répartition de tâches entre plusieurs processus. L'utilisation de la parallélisation pour optimiser la recherche de maximum dans un tableau volumineux a illustré l'efficacité de cette approche dans l'accélération de tâches complexes.

Ainsi, ce rapport a offert une plongée approfondie dans les rouages de la programmation système, soulignant l'importance cruciale des processus, de la gestion de fichiers et de la parallélisation dans la construction d'applications informatiques performantes et robustes. Ces fondements constituent un socle essentiel pour tout développeur cherchant à maîtriser les subtilités du fonctionnement des systèmes informatiques.