Exercice 1

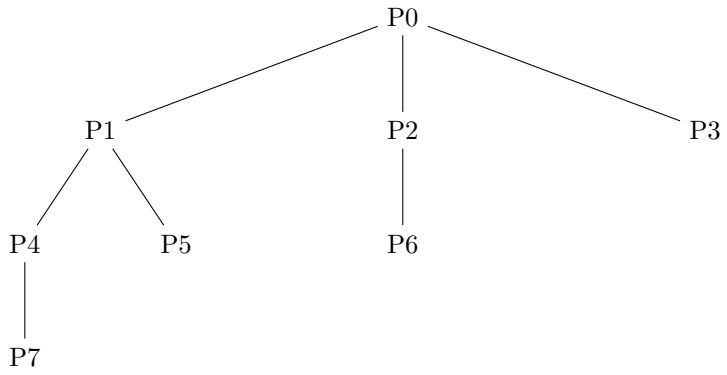
Le programme suivant crée 8 processus en tout. Cela s'explique par le fait que chaque appel à la fonction `fork()` duplique le processus qui l'exécute, ce qui double le nombre total de processus à chaque fois. Voici le programme :

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main() {
    fork();
    fork();
    fork();
    return EXIT_SUCCESS;
}
```

Déroulement de la création des processus :

- Après le premier appel à `fork()`, il y a 2 processus (le parent et un enfant).
- Après le deuxième appel à `fork()`, chaque processus existant (le parent et l'enfant) crée un autre processus, portant le total à 4 processus.
- Enfin, après le troisième appel à `fork()`, chaque processus existant crée un nouveau processus, portant le total à 8 processus.

Voici l'arbre des processus correspondant :



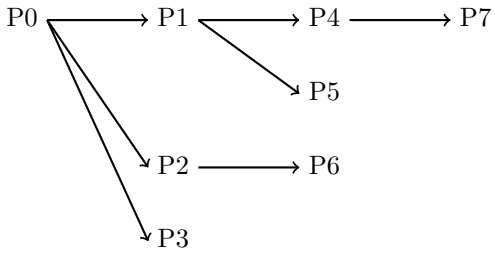
Au total, le programme crée 8 processus.

Exercice 2

Le programme suivant crée plusieurs processus en utilisant la fonction `fork()`. L'objectif est de dessiner l'arbre généalogique des processus créés :

```c
# include <unistd.h>
# include <stdio.h>
int main() {
    pid_t pid; int i;
    for (i=0; i<3;i++) {
        pid = fork();
        if (pid < 0 ){
            printf ("le fork ( ) a échoué \n") ;
        }
        else if (pid == 0){
            printf(" je suis le processus : %d, mon père est : %d\n", getpid(),
            ↪  getppid()) ;
        }
        else{
            printf("je suis le processus : %d, mon père est : %d\n", getpid(),
            ↪  getppid()) ;
        }
    }
    return 0 ;
}
```

À chaque itération du `for` (exécuté trois fois), un nouveau processus est créé. Voici l'arbre des processus résultant :



.

1 Exercice 03

Voici un programme en C qui crée 5 processus fils en utilisant la fonction `fork()` et attend leur terminaison :

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

#define NUM_PROCESSES 5

int main() {
    pid_t pids[NUM_PROCESSES]; // Tableau pour stocker les PID des processus
    ↪ fils
    int i;

    // Crée 5 processus fils
    for (i = 0; i < NUM_PROCESSES; i++) {
        pids[i] = fork(); // Crée un processus fils

        if (pids[i] < 0) {
            // Erreur lors de la création du processus
            perror("Erreur lors de fork");
            exit(EXIT_FAILURE);
        } else if (pids[i] == 0) {
            // Code du processus fils
            printf("Processus fils %d démarre\n", i);
            sleep(1); // Simule une activité
            printf("Processus fils %d se termine\n", i);
            exit(0); // Terminer le processus fils
        }
    }

    // Attendre la fin de chaque processus fils
    for (i = 0; i < NUM_PROCESSES; i++) {
        waitpid(pids[i], NULL, 0); // Attend la fin du processus fils
    }

    printf("Tous les processus fils sont terminés. Fin du programme
    ↪ principal.\n");

    return 0;
}
```

Exercice 5 :

Pour décomposer l'expression suivante :

$$y := \left(\frac{(a+b)}{(c-d)} + (e \cdot f) \right) + (a+b) \cdot (c-d)$$

nous identifions les opérations indépendantes que nous pourrions paralléliser. Les tâches sont les suivantes :

- **Tâche 1 (T1)** : Calculer $T1 = a + b$
- **Tâche 2 (T2)** : Calculer $T2 = c - d$

- **Tâche 3 (T3)** : Calculer $T3 = e \cdot f$
- **Tâche 4 (T4)** : Calculer $T4 = \frac{T1}{T2}$ (dépend de T1 et T2)
- **Tâche 5 (T5)** : Calculer $T5 = T1 \cdot T2$ (dépend de T1 et T2)
- **Tâche 6 (T6)** : Calculer $y = T4 + T3 + T5$ (dépend de T3, T4 et T5)

Condition de Bernstein

La condition de Bernstein stipule que deux tâches peuvent être parallélisées si elles n'ont pas de dépendances de données. Dans notre cas :

- Les Tâches 1, 2 et 3 (T1, T2, T3) peuvent être exécutées en parallèle car elles ne partagent pas de variables.
- Les Tâches 4 et 5 dépendent de T1 et T2.
- La Tâche 6 dépend des Tâches 3, 4 et 5.

Ainsi, il est possible de paralléliser cette expression en exécutant T1, T2 et T3 en parallèle, puis en calculant T4 et T5 (qui dépendent de T1 et T2), et enfin en exécutant T6.

Code C utilisant `fork()`

Voici un exemple de code C qui utilise `fork()` pour évaluer l'expression parallèlement :

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

int main() {
    int a = 10, b = 20, c = 30, d = 5, e = 2, f = 3;
    int T1, T2, T3, T4, T5;
    float y;

    // Tâche 1: T1 = a + b
    pid_t pid1 = fork();
    if (pid1 == 0) { // Processus fils
        T1 = a + b;
        exit(T1); // Retourne le résultat
    }

    // Tâche 2: T2 = c - d
    pid_t pid2 = fork();
    if (pid2 == 0) { // Processus fils
        T2 = c - d;
        exit(T2); // Retourne le résultat
```

```
    }

    // Tâche 3: T3 = e * f
    pid_t pid3 = fork();
    if (pid3 == 0) { // Processus fils
        T3 = e * f;
        exit(T3); // Retourne le résultat
    }

    // Attendre les résultats de T1, T2, T3
    int status;
    waitpid(pid1, &status, 0);
    T1 = WEXITSTATUS(status);

    waitpid(pid2, &status, 0);
    T2 = WEXITSTATUS(status);

    waitpid(pid3, &status, 0);
    T3 = WEXITSTATUS(status);

    // Tâche 4: T4 = T1 / T2
    T4 = T1 / T2;

    // Tâche 5: T5 = T1 * T2
    T5 = T1 * T2;

    // Tâche 6: y = T4 + T3 + T5
    y = T4 + T3 + T5;

    // Afficher le résultat
    printf("La valeur de y est: %f\n", y);

    return 0;
}
```

Explication du Code

- **Fork** : Trois processus fils sont créés pour exécuter les Tâches 1, 2 et 3.
- **Exit** : Chaque processus fils calcule sa tâche et retourne le résultat via `exit()`.
- **Wait** : Le processus père attend la fin des fils et récupère les résultats en utilisant `waitpid()`.
- **Calcul Final** : Le processus père calcule T4 et T5, puis la valeur finale de y.

Cette approche permet d'évaluer les parties indépendantes de l'expression en parallèle, optimisant ainsi le temps d'exécution.