TP 2

1. Création de Processus: Utilisation de fork

Programme `fork1.c`

- **Question Prévoyez le résultat**: Le programme affiche d'abord "debut", puis crée un processus fils. Le processus fils incrémentera `x` et affichera son PID, celui de son parent, et la valeur de `x`. Le processus parent décrémentera `x` et affichera son PID, le PID de son fils, et la valeur de `x`. Finalement, "Fin" sera affiché deux fois: une fois par le processus parent et une fois par le fils.
- **Arbre des processus**: Le processus parent (P) crée un processus fils (F).
- **Compilation et test**:

```
┌──(jules@jules-MacBookPro)-[~/Documents/Polytech/SE/output]
└─$ ./"fork1"
debut
parent: je suis 12410, parent de 12411 et fils de 9558. x=0
Fin
child : je suis 12411, fils de 12410. x=2
Fin
```

- **Remarque**:
  Le résultat est bien le même que ce que j'avais décris à la question précédente.
  Les valeurs de la variable x confirment bien que la mémoire n’est pas partagée entre le processus père et fils.

1.1 Fork Imbriqué

- **Programme à écrire**:

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

int main() {
    pid_t pid;
    printf("Debut. \n");
    printf("P1 : mon pid est %d, mon ppid est %d. \n", (int) getpid(), (int) getppid())
    pid = fork();
    if (pid == 0) {
        printf("P2 : mon pid est %d, mon ppid est %d. \n", (int) getpid(), (int) getppi
    }
    else {
        printf ("Je suis %d et mon fils est %d. \n", getpid(), pid);
        pid = fork();
        if (pid == 0) {
            printf("P3 : mon pid est %d, mon ppid est %d. \n", (int) getpid(), (int) ge
            pid = fork();
            if(pid == 0) {
                printf("P5 : mon pid est %d, mon ppid est %d. \n", (int) getpid(), (int
            }
            else {
                printf ("Je suis %d et mon fils est %d. \n", getpid(), pid);
                pid = fork();
                if(pid == 0) {
                    printf("P6 : mon pid est %d, mon ppid est %d. \n", (int) getpid(),
                }
                else {
                    printf ("Je suis %d et mon fils est %d. \n", getpid(), pid);
                }
            }
        }
        else {
            printf ("Je suis %d et mon fils est %d. \n", getpid(), pid);
            pid = fork();
            if (pid == 0) {
                printf("P4 : mon pid est %d, mon ppid est %d. \n", (int) getpid(), (int
            }
            else {
                printf ("Je suis %d et mon fils est %d. \n", getpid(), pid);
            }
        }
```

```
    }
    return 0;
}
```

- **Résultat obtenue**:

```
┌──(jules@jules-MacBookPro)-[~/Documents/Polytech/SE/output]
└─$ ./"1.1"
Debut.
P1 : mon pid est 15800, mon ppid est 9558.
Je suis 15800 et mon fils est 15801.
P2 : mon pid est 15801, mon ppid est 15800.
Je suis 15800 et mon fils est 15802.
P3 : mon pid est 15802, mon ppid est 15800.
Je suis 15800 et mon fils est 15803.
P4 : mon pid est 15803, mon ppid est 15800.
Je suis 15802 et mon fils est 15804.
P5 : mon pid est 15804, mon ppid est 15802.
Je suis 15802 et mon fils est 15805.
P6 : mon pid est 15805, mon ppid est 15802.
```

- **Observation et commentaire sur l'ordre d'apparition des messages**:
  Je vois que l'ordre d'apparition des messages n'est pas l'ordre du code. Si je relance plusieurs fois le programme, j'observe que les PID et l'ordre d'apparition des messages changent. Je peux donc en déduire que les PID sont incrémentés et que l'ordre d'exécution des fils est aléatoire.

2. Terminaison de Processus

Programme `terminaison.c`

Question 1 : Donnez le nombre de processus créés, ainsi que les affichages effectués par chaque processus en justifiant.Vérifiez votre réponse en ajoutant des affichages dans le programme.

- **Nombre de processus créés**: 3 (le processus original `P1` , et deux processus fils `P2` et `P3`).
- **Affichages effectués**:
  - `P2` affichera `a : 20 \n` juste avant de terminer.

- `P1` attend que `P2` termine et affichera `a : 10 ; e : 1 \n` , où `e` est le code de sortie de `P2` (1 dans ce cas).

`P3` ne fait pas d'affichage directement mais influence le code de sortie que `P2` peut récupérer si `P2` utilisait `wait()` . Cependant, dans ce code, `P2` termine sans attendre `P3` , donc le code de sortie de `P3` n'est pas récupéré ni affiché par `P2` .

- **Vérification par Affichage Supplémentaire**

Voici le nouveau code avec des affichages supplémentaires :

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

int main(int argc, char *argv[]) {
    int a, e;
    a = 10;
    printf("Start: PID=%d, PPID=%d\n", getpid(), getppid()); // Affiche le PID au début
    if (fork() == 0) {
        a = a *2;
        printf("P2: PID=%d, PPID=%d, a=%d\n", getpid(), getppid(), a); // P2 affiche so
        if (fork() == 0) {
            a = a +1;
            printf("P3: PID=%d, PPID=%d, a=%d\n", getpid(), getppid(), a); // P3 affich
            exit(2);
        }
        wait(NULL); // P2 attends P3 pour synchroniser l'affichage
        exit(1);
    }
    wait(&e);
    printf("P1: PID=%d, a=%d ; e=%d \n", getpid(), a, WEXITSTATUS(e)); // P1 affiche so
    return(0);
}
```

Voici le résultat que j'obtiens :

```
┌──(jules@jules-MacBookPro)-[~/Documents/Polytech/SE/output]
└─$ ./"terminaison"
Start: PID=23654, PPID=9558
P2: PID=23655, PPID=23654, a=20
P3: PID=23656, PPID=23655, a=21
P1: PID=23654, a=10 ; e=1
```

Question 2 : On supprime l'instruction exit(2), reprenez la question précédente en conséquence.

1. **Premier processus fils** (`P2`):
   - Multiplie `a` par 2, donc `a = 20` .
   - Crée un processus fils (`P3`).
   - **Puisque `P3` n'effectue plus `exit(2)` , il continue l'exécution du reste du code dans `P2` .**
   - Affiche `a : 20 \n` .
   - Termine avec `exit(1)` .
2. **Second processus fils** (`P3`):
   - Incrémente `a` de 1, donc `a = 21` .
   - Poursuit l'exécution et effectue la même instruction d'affichage que `P2` (car il continue à exécuter le code après la condition `if (fork() == 0)` sans rencontrer un `exit()` précoce).
   - Affiche `a : 21 \n` .
   - Termine implicitement à la fin de `main()` , retournant 0 comme code de sortie par défaut (ceci n'est pas visible car `wait(&e)` dans le processus parent (`P1`) attend seulement le premier processus fils (`P2`)).
3. **Processus principal** (`P1`):
   - Ne modifie pas `a` après la création de `P2` , donc `a = 10` .
   - Attend que `P2` se termine et récupère son code de sortie via `e` .
   - Affiche `a : 10 ; e : 1 \n` .

Voici le résultat que j'obtiens :

```
┌──(jules@jules-MacBookPro)-[~/Documents/Polytech/SE/output]
└─$ ./"terminaison"
a : 20
a : 21
a : 10 ; e : 1
```

Question 3 : Modifiez le programme initial pour créer un processus zombie pendant 30 secondes.

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

int main(int argc, char *argv[]) {
    int a, e;
    a = 10;
    pid_t pid = fork();

    if (pid == 0) { // Processus enfant
        a = a * 2;
        pid_t pid_child = fork();
        if (pid_child == 0) { // Enfant de l'enfant (petit-enfant)
            a = a + 1;
            printf("Petit-enfant en pause, PID = %d\n", getpid());
            sleep(30); // Pause pour créer un processus zombie
            exit(2); // Le petit-enfant termine
        }
        // Le processus enfant se termine immédiatement
        // Ne pas attendre le petit-enfant exprès pour créer un zombie
        exit(1);
    }
    else {
        // Le processus parent attend immédiatement l'enfant pour éviter un zombie ici
        wait(&e);
        printf("a : %d ; e : %d\n", a, WEXITSTATUS(e));
        // Le parent se termine rapidement, laissant le petit-enfant devenir un zombie
        sleep(35); // Attendre un peu plus longtemps que le petit-enfant pour observer
    }
    return 0;
}
```

Dans ce programme, le "petit-enfant" s'endort pendant 30 secondes avant de se terminer. Cependant, son parent (l'enfant du processus initial) se termine presque immédiatement après sa création, sans attendre que le "petit-enfant" se termine. Cela crée une condition où le "petit-enfant" devient un processus zombie après sa terminaison, jusqu'à ce que le grand-parent (le processus initial) termine l'attente et nettoie le statut du zombie.

- **Vérification avec `ps -al`**

```
┌──(jules@jules-MacBookPro)-[~/Documents/Polytech/SE]
└─$ ps -al
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1781    1778  0  80   0 - 56593 do_sys tty2     00:00:00 gnome-session-b
0 S  1000   27958    9558  0  80   0 -   655 hrtime pts/0    00:00:00 terminaison
1 S  1000   27960    1729  0  80   0 -   655 hrtime pts/0    00:00:00 terminaison
4 R  1000   28028   28002  0  80   0 -  3858 -      pts/1    00:00:00 ps
```

Question 4 : Ajoutez à votre programme précédent l’instruction sleep(30) à la ligne 11 (avant exit(1)). Que se passe-t-il ? Expliquez.

Si nous ajoutons `sleep(30);` juste avant `exit(1);` dans le processus enfant, le comportement du programme change de manière significative. Cette modification introduit une pause dans le processus enfant avant qu'il ne termine, permettant potentiellement au "petit-enfant" de terminer et d'éviter de devenir un zombie pendant que l'enfant dort. Voyons cela plus en détail.

Programme Modifié avec `sleep(30)` Avant `exit(1)`

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

int main(int argc, char *argv[]) {
    int a, e;
    a = 10;
    pid_t pid = fork();

    if (pid == 0) { // Processus enfant
        a = a * 2;
        pid_t pid_child = fork();
        if (pid_child == 0) { // Enfant de l'enfant (petit-enfant)
            a = a + 1;
            printf("Petit-enfant en pause, PID = %d\n", getpid());
            sleep(30); // Pause pour permettre au parent de devenir temporairement un z
            exit(2); // Le petit-enfant termine
        }
        sleep(30); // Ajout du sleep ici
        printf("Enfant se termine, PID = %d\n", getpid());
        exit(1);
    }
    else {
        // Le processus parent attend l'enfant
        wait(&e);
        printf("a : %d ; e : %d\n", a, WEXITSTATUS(e));
        sleep(35); // Assurez-vous que le parent attend assez longtemps pour observer l
    }
    return 0;
}
```

1. **Processus parent (`P1`)**: Crée un processus enfant (`P2`) et attend sa terminaison avec `wait(&e)` avant d'entrer dans son propre `sleep(35)` pour assurer qu'il reste actif pour nettoyer tout processus zombie potentiel après le réveil.
2. **Processus enfant (`P2`)**: Multiplie `a` par 2 et crée un "petit-enfant" (`P3`). Au lieu de se terminer immédiatement, il entre maintenant en pause pour 30 secondes grâce au `sleep(30)` ajouté. Après cette pause, il imprime un message indiquant sa propre terminaison et se termine avec `exit(1)` .
3. **"Petit-enfant" (`P3`)**: Incrémente `a` de 1, dort également pour 30 secondes (comme avant), puis se termine avec `exit(2)` .

Effets de l'Ajout de `sleep(30)` Avant `exit(1)`

- **Synchronisation des Terminaisons**: L'ajout de `sleep(30)` avant `exit(1)` dans `P2` synchronise presque les terminaisons de `P2` et `P3`. Cela signifie que `P3` pourrait se terminer juste avant ou autour du même moment que `P2`. Si `P2` attend `P3` pour terminer grâce au `sleep()`, il y a moins de chances que `P3` devienne un zombie, car `P2` peut encore être en vie lorsque `P3` se termine. Néanmoins, si `P3` se termine avant que `P2` n'appelle `wait()`, il deviendra un zombie jusqu'à ce que `P2` se réveille et termine, permettant au processus parent de nettoyer.
- **Pas de Processus Zombie**: Avec les deux `sleep(30)` en place, il est très probable que le "petit-enfant" (`P3`) ne devienne pas un zombie du tout, car son processus parent (`P2`) ne se termine pas avant lui.

3. La Primitive exec

Programme `exec1.c`

Question : Résultat de l'exécution et explication de qui affiche quoi et pourquoi.

1. Après un délai de 2 secondes :

   ```
   je suis le fils !
   ```

   Cette sortie provient du processus fils après l'exécution de la commande `echo`.
2. Immédiatement après :

   ```
   Je suis ton père
   ```

   Cette sortie provient du processus parent après que le processus fils a terminé.

Le message `je suis le fils !` est affiché par le processus fils grâce à l'exécution du programme `/bin/echo` avec les arguments fournis. Le processus parent, après avoir attendu la fin de l'exécution du processus fils (`wait(NULL)`), affiche le message `Je suis ton père`.

L'utilisation de `wait(NULL)` dans le processus parent assure que le message du parent sera toujours affiché après celui du fils, car le parent attend la terminaison du fils avant de continuer son exécution.

Programme `exec2.c`

Question : Représentation des processus et ordre de terminaison

1. **Au point A**: Le programme affiche le nom de lui-même (c'est-à-dire `argv[0]`).
2. **Au point B**: Le processus principal (`P0`) appelle `fork()`. Cela crée un nouveau processus enfant (`P1`). À ce stade, nous avons deux processus: le processus parent (`P0`) et le processus enfant (`P1`).
3. **Au point C (dans P1)**: Le processus enfant `P1` appelle `fork()` une deuxième fois, créant un deuxième processus enfant (`P2`). Maintenant, `P1` devient un parent à `P2`.

À partir de ce code, le diagramme d'arbre des processus ressemble à ceci:

```
P0
 |
 P1
 |
 P2 ----> exec("/prog")
```

- **P0** est le processus initial.
- **P1** est créé par `P0` grâce à la première invocation de `fork()`.
- **P2** est créé par `P1` grâce à la deuxième invocation de `fork()` et exécute un nouveau programme appelé `prog` à l'aide de `exec()`.

Ordres de terminaison des processus

- **P2** doit terminer avant `P1` parce que `P1` appelle `exit(0)` immédiatement après la création de `P2` (et `P2` exécute un nouveau programme qui, une fois terminé, signifie la fin de `P2`).
- **P0** attend que `P1` termine grâce à l'appel `wait(&retour)` avant de continuer. Cela signifie que `P0` terminera après `P1` (et donc après `P2` également, puisque `P1` attend que `P2` termine).

Ainsi, les ordres possibles de terminaison sont:

- **P2** termine en premier, suivi de **P1**, et enfin **P0**.

4. Création de Processus en Chaîne

Question 1 : Ecrivez ce programme. Donnez le programme dans le compte rendu et représentez les processus créés pour N = 3.

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

int main() {
    int N = 3; // Nombre de processus à créer en plus du processus initial.
    pid_t pid;
    int i;

    for (i = 0; i < N; i++) {
        pid = fork();
        if (pid == 0) {
            // Processus enfant
            printf("Processus %d créé par processus %d\n", getpid(), getppid());
        } else {
            // Processus parent attend la fin du processus enfant
            wait(NULL);
            break; // Le processus parent ne crée pas plus de processus
        }
    }
    return 0;
}
```

```
┌──(jules@jules-MacBookPro)-[~/…/Polytech/SE/TP2/output]
└─$ ./"4"
Processus 36828 créé par processus 36827
Processus 36829 créé par processus 36828
Processus 36830 créé par processus 36829
```

Ce programme crée une chaîne de processus où chaque processus, à son tour, crée un processus enfant jusqu'à ce que le nombre total de processus créés atteigne N. Après avoir créé un processus enfant, le processus parent attend la fin de ce dernier avec `wait(NULL)` avant de terminer lui-même. Cela garantit que chaque processus enfant termine avant son parent, empêchant ainsi la création de processus zombies.

Pour N = 3, la chaîne de processus ressemblera à ceci :

```
Processus initial (main)
    |
    --> Processus 1 (créé par le processus initial)
         |
         --> Processus 2 (créé par le processus 1)
              |
              --> Processus 3 (créé par le processus 2)
```

- Le processus initial crée le Processus 1.
- Le Processus 1 crée le Processus 2.
- Le Processus 2 crée le Processus 3.

Question 2 : Modifiez le programme pour que le processus initial attende uniquement la fin de son fils. Donnez le programme modifié dans le compte rendu en justifiant les modifications effectuées.

Voici le programme modifié :

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

int main() {
    int N = 3; // Nombre de processus à créer en plus du processus initial.
    pid_t pid;
    int i;

    for (i = 0; i < N; i++) {
        pid = fork();
        if (pid == 0) {
            // Processus enfant
            printf("Processus %d créé par processus %d\n", getpid(), getppid());
            if (i < N-1) {
                continue; // Laisse le processus enfant créer son propre enfant
            } else {
                return 0; // Le dernier processus dans la chaîne ne crée pas d'enfant e
            }
        } else {
            // Processus parent
            if (i == 0) {
                wait(NULL); // Le processus initial attend la fin de son premier fils
            }
            break; // Les processus parents (sauf le processus initial) ne créent pas d
        }
    }
    return 0;
}
```

Justification des modifications :

- **Boucle `for`** : Parcourt toujours N fois, avec la possibilité de créer jusqu'à N processus en chaîne.
- **`if (pid == 0)`** : À l'intérieur de cette condition, chaque processus enfant peut potentiellement devenir parent s'il ne s'agit pas du dernier processus de la chaîne (`if (i < N-1)`). Le dernier processus dans la chaîne (quand `i == N-1`) se termine simplement sans créer de processus enfant.
- **`else`** : Pour le processus parent, une vérification supplémentaire (`if (i == 0)`) est effectuée pour s'assurer que seul le processus initial attend la fin de son premier fils avec `wait(NULL)` . Les processus intermédiaires (qui sont à la fois parents et enfants dans la chaîne, sauf le

processus initial) ne font pas d'attente et se terminent directement après la création de leur enfant, ce qui empêche la chaîne d'attendre à chaque niveau.

Question 3 : Modifiez le programme pour que le processus initial attende la fin de tous les processus créés. Donnez le programme modifié dans le compte rendu en justifiant les modifications effectuées.

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

void creerChaineDeProcessus(int N, int niveauActuel) {
    if (N == 0) return; // Cas de base : plus aucun processus à créer.

    pid_t pid = fork();

    if (pid == -1) {
        // Échec du fork.
        perror("fork");
        return;
    } else if (pid == 0) {
        // Processus enfant
        printf("Processus %d créé par processus %d\n", getpid(), getppid());
        creerChaineDeProcessus(N - 1, niveauActuel + 1); // Créer récursivement le proc
        exit(0); // Le processus enfant se termine après avoir créé son enfant.
    } else {
        // Le processus parent attend la terminaison de son enfant.
        wait(NULL);
        if (niveauActuel == 1) {
            printf("Processus initial (PID %d) termine après tous ses enfants.\n", getp
        }
    }
}

int main() {
    int N = 3; // Nombre de processus à créer dans la chaîne, sans compter le processus
    creerChaineDeProcessus(N, 1); // Commence la chaîne avec le niveau initial du proce
    return 0;
}
```

Dans ce programme :

- La fonction `creerChaineDeProcessus` est utilisée pour créer une chaîne de processus. Elle prend deux arguments : `N`, qui représente le nombre de processus restants à créer dans la chaîne, et `niveauActuel`, qui indique le niveau actuel du processus dans la chaîne pour contrôler l'affichage.
- Si `N` est égal à 0, cela signifie qu'il n'y a plus de processus à créer, et la fonction retourne sans faire de `fork()`.
- Lorsqu'un processus enfant est créé (`pid == 0`), il affiche son PID et le PID de son parent, puis appelle récursivement `creerChaineDeProcessus` pour créer son propre enfant, diminuant `N` de 1.
- Chaque processus enfant se termine après avoir créé son enfant (s'il en reste à créer), à l'exception du dernier processus de la chaîne qui se termine sans créer d'enfant.
- Le processus initial (au niveau 1) attend la terminaison de son premier enfant directement. Après l'attente, il affiche un message indiquant qu'il termine après tous ses enfants, ce qui est vérifié par `if (niveauActuel == 1)`.