Université de Technologie de Compiègne

Devoir 2

Pascal Quach, Korantin Toczé

UV: *Maîtrise des systèmes informatiques (SR01)*
Date de rendu: *30 Décembre 2019*
Semestre: *A19*

Exercice 1 : Arbre généalogique de processus

Partie 1.

- Le `PID` du `shell` est 2000.
- Le `PID` du processus correspondant au parent est 2400.
- La numérotation des processus est séquentielle (incrémentation par 1).
- Un processus de `PID` *p* ne peut être exécuté qu'après la fin de l'exécution du processus de `PID` *p-1*.

Donner l'arbre généalogique des processus générés par chaque programme.

Programme 1.

```c
#include<unistd.h>
int main() {
    (fork() || fork()) && (fork() || fork());
}
```

Listing 1: Programme 1

Réponse. Le **ET** logique (`&&`) n'évalue pas la deuxième opérande si la première est fausse. Le **OU** logique (`||`) n'évalue pas la deuxième opérande si la première est vraie. La fonction `fork()` renvoie 0, donc faux, au processus fils et une valeur non nulle, donc vrai, au processus parent. La figure en annexe est utile pour observer le déroulement en détail du programme. L'arbre généalogique des processus est donc le suivant :

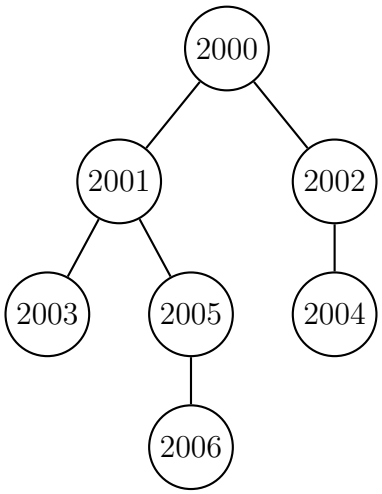


Figure 1: Arbre généalogique du programme 1

Programme 2.

```c
#include <unistd.h>
#include <stdlib.h>
int main() {
    int i = 0;
    fork();
    while (i < 4) {
        if (getpid() % 2 == 0) {
            fork();
        }
        i++;
    }
}
```

Listing 2: Programme 2

Réponse. La variable de la boucle `while`, `i` est présente dans le processus fils. L'arbre de déroulement du programme 2 est disponible en annexe. L'arbre généalogique des processus est donc le suivant :

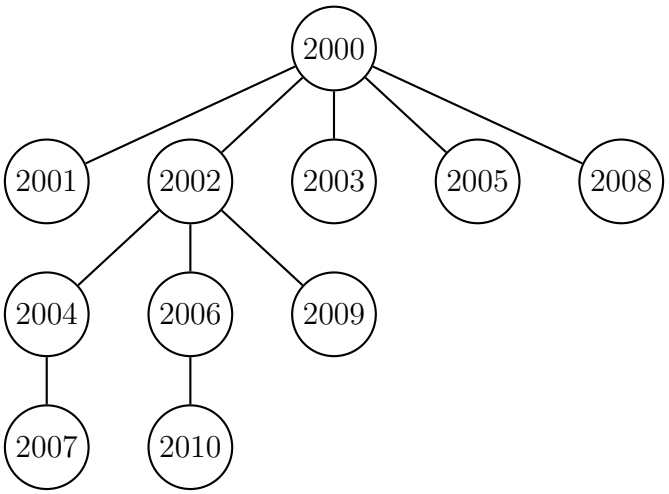
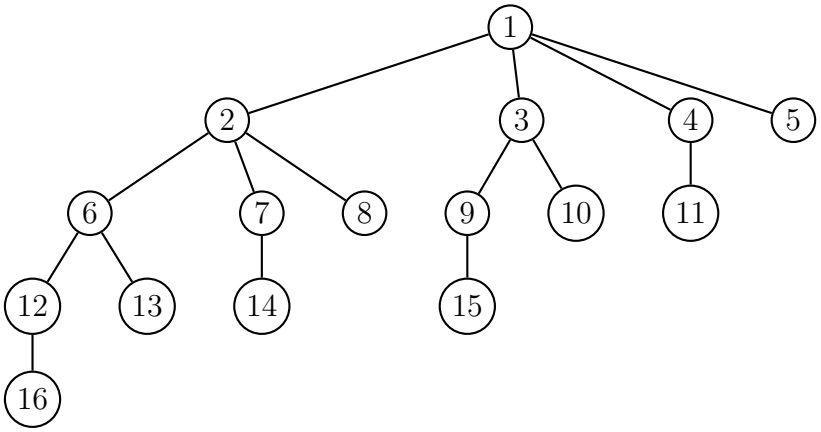


Figure 3: Arbre généalogique du programme 2

Partie 2.

En utilisant la fonction `fork()`, proposer un programme `C` permettant de générer cet arbre de processus. Pour chaque processus, afficher son `PID` et celui de son père.



Réponse. On réalise une boucle itérative pour exécuter `fork()` successivement.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    for (size_t i = 1; i < 5; i++) {
        int n = fork();
        if (n > 0) {
            printf("[parent][PID=%d] Le PID de mon fils est
               : %d.\n", getpid(), n);
            sleep(1);
        }
        else {
            printf("[fils][PID=%d] Le PID de mon père est :
               %d.\n", getpid(), getppid());
        }
    }
    return EXIT_SUCCESS;
}
```

Listing 3: Partie 2 - Programme en `C`

Exercice 2 : Gestionnaire d'applications

On va programmer un gestionnaire d'applications personnalisé (`Application-Manager`). La liste des applications à lancer est stockée dans le fichier `list_appli.txt`. Vous disposez de quelques exemples d'applications (`power_manager.c`, `network_manager.c`, `get_time.c`).

Question 1. Écrire un programme `ApplicationManager.c` qui doit:

- Créer un ensemble de processus fils chacun est responsable à l'exécution d'une application.
- Lors de l'arrêt d'une application, informer l'utilisateur en lui affichant le nom de l'application terminée.
- S'arrêter après avoir fermer toutes les applications en cours d'exécution lors de la réception d'un ordre de mise en veille de la part de `power_manager` (signal **`SIGUSR1`**).

NB : Lorsque `ApplicationManager` reçoit un signal **`SIGUSR1`** de la part d'un autre processus, il ne ferme pas les applications.

Réponse.

1. On vient d'abord lire les informations dans le fichier. On crée ensuite un semble de processus fils qui auront chacun un `id_processus` unique qui correspond à l'ordre dans lequel ils ont été lu. Les `PID`s des fils sont stockés dans une variable globale `pid_t *tab_pid_fils`. Après avoir traité `path` et `argv`, on exécute les commandes qui lancent les applications.
2. On utilise un *signal handler* de prototype `intercepter(int n, siginfo_t *signal_info)` défini par `sigaction`, i.e. `struct sigaction S.sa_handler = intercepter`. On fait correspondre le `PID` du processus qui a envoyé le signal avec son emplacement dans le tableau `tab_pid_fils` pour obtenir son `id_processus`. De là, le processus père a envoyé la liste de tous les noms par un *pipe* à `intercepter`. Il suffit de récupérer le bon nom de l'application pour ensuite l'afficher lors du traitement du signal. On vérifie le statut du fils à l'aide de la fonction `waitpid`, puis on détermine ce qui lui est arrivé à l'aide des *flags* : `WIFEXITED`, `WIFSIGNALED`, `WIFSTOPPED`, `WIFCONTINUED`.
3. La réception d'un signal **`SIGUSR1`** est vérifié dans `intercepter`. On vérifie que le signal envoyé est bien celui de **`SIGUSR1`**, `n == SIGUSR1`, que l'`id_processus` est bien 0 et que le nom du processus qui a activé le *signal handler* est bien `"Power Manager"`. On éteint ensuite toutes les applications, puis enfin on quitte le process en utilisant `exit`.

Question 2. Modifier le programme `power_manager.c` pour envoyer le signal **`SIGUSR1`** à l'`ApplicationManager` lorsque l'utilisateur tape 1 dans le fichier `mise_en_veille.txt`.

Réponse. On récupère le `PPID` à l'aide de la fonction `getppid()`. Ensuite, on envoie un signal **`SIGUSR1`** à l'aide de la fonction `kill`.

Exercice 3

L'objectif de cet exercice est de paralléliser le calcul de la somme ou le produit de deux matrices. La somme et le produit doivent se faire en parallèle, donc une approche dans laquelle le processus père attend la fin du traitement réalisé par un fils afin d'en créer un autre ne sera pas acceptée. L'écriture et la lecture des fichiers doivent se faire en utilisant `fread()` et `fwrite()`.

Question 1.

Créez un programme `Somme.c`. Lors de l'implémentation de ce programme, supposez que les matrices sont déjà saisies dans les fichiers.

- Il reçoit en paramètres dans la fonction `main` :
  - deux chemins vers deux fichiers binaires contenant chacun une matrice carrée de la même taille
  - le nombre de lignes N d'une des matrices.
- Dans le programme `Somme.c`, on commence par lire les deux matrices.
- Ensuite, on crée N processus fils. Chaque processus fils i :
  - calcule la somme des $i$ème lignes des deux matrices;
  - communique le résultat au processus père en utilisant les pipes.
- Une fois que le processus père récupère la somme de chaque ligne, il affiche la matrice résultante.

Réponse. On commence par une lecture des fichiers passés en arguments du main, puis on les stocke dans un tableau. On crée ensuite n tableaux pour n pipes, un par processus. Ensuite, on crée un processus par ligne qui calcule la somme des éléments, puis l'on écrit dans le pipe de ce processus. Enfin, on lit tous les résultats et on les affiche.

Question 2.

De même, créez un programme `Produit.c` qui reçoit les mêmes paramètres que le programme `Somme.c` mais qui effectue le produit de deux matrices.

- Dans le programme `Produit.c`, vous devez créer N processus fils, tout comme le programme `Somme.c`.
- Cependant, chaque processus fils va calculer le produit de la $i$ème ligne de la première matrice avec toutes les colonnes de la deuxième matrice.
- Ensuite, il communique le résultat au processus père.
- Une fois que le processus père récupère la somme de chaque ligne, il affiche la matrice résultante.

Réponse. Seul le calcul est différent du programme précédent, on ajoute une boucle pour parcourir toutes les colonnes de la première matrice et toutes les lignes de la seconde à chaque itération.

Question 3.

Créez un programme `ManipMatrice.c` qui permet de :

- saisir des matrices ou de les générer aléatoirement,
- de les stocker dans des fichiers binaires,
- et de faire appel en utilisant `execv()`, aux programmes `Somme.c` ou `Produit.c` selon le choix de l'utilisateur.

Réponse. Dans cette question, on va demander à l'utilisateur d'entrer deux matrices (stockées dans les fichiers "matrice1" et "matrice2"). On supposera les deux programmes déja compilés dans le même dossier sous le nom "Somme" et "Produit". On génere alors les matrices aléatoirement ou selon les entrées utilisateurs, puis on appelle les programmes précédents en utilisant execv.

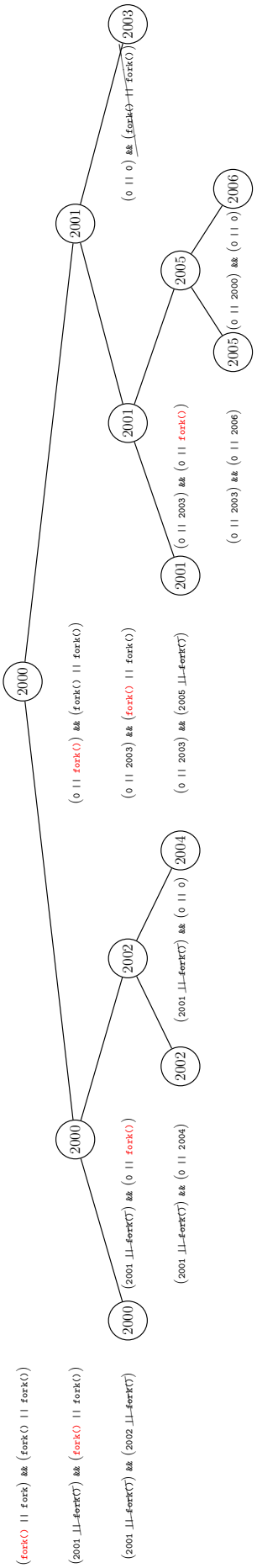


Figure 2: Arbre du déroulement des processus du programme 1

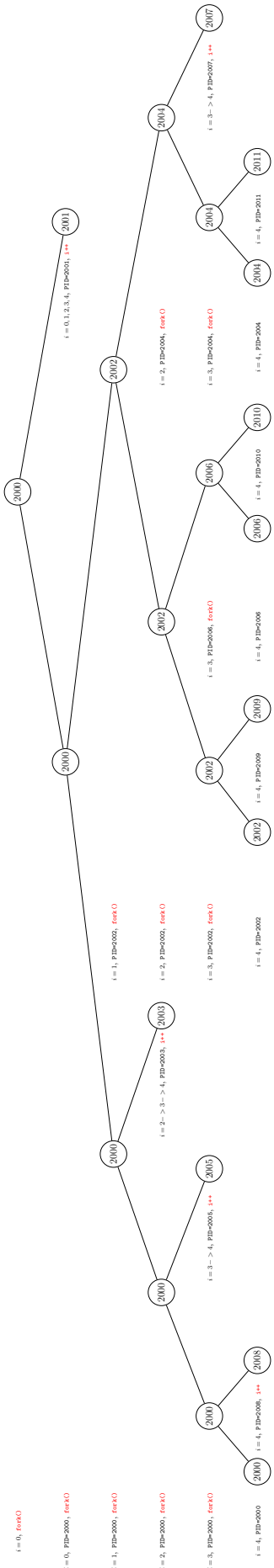


Figure 4: Arbre du déroulement des processus du programme 2