TD1 : Processus UNIX

Système I

Deuxième année

L'objectif de ce TD est d'approfondir les notions relatives aux processus et de les appliquer au cadre spécifique d'Unix.

★ Exercice 1: Création de processus avec l'appel système `fork`

La commande `fork()` crée un processus «fils» du processus appelant (le «père»), avec le même programme que ce dernier. La valeur renvoyée par `fork()` est :

- Au père : le numéro (PID) du processus fils.
- Au fils : 0 (il peut retrouver le PID de son père avec `getppid()`)

En cas d'échec (table des processus pleine), aucun processus n'est créé, et la valeur -1 est renvoyée. Dans la suite, nous supposerons que tous les appels réussissent sans problème, mais une application réelle devrait bien entendu traiter les cas d'erreurs.

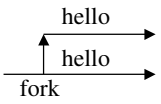
```c
int main() {
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {
        printf("Dans fils : x=%d\n", ++x);
        exit(0);
    }

    printf("Dans père : x=%d\n", --x);
    exit(0);
}
```

Q 1: Qu'affiche l'exécution du programme ci-contre.

Q 2: On considère les deux programmes suivants et leur schéma d'exécution.

- *Exemple :* un clonage

```c
int main() {
   fork();
   printf("hello!\n");
   exit(0);
}
```



- *Question 1 :* Illustrer l'exécution du programme suivant.

```c
int main() {
   fork();
   fork();
   printf("hello!\n");
   exit(0);
}
```

- *Question 2 :* Même question pour le programme suivant.

```c
int main() {
   fork();
   fork();
   fork();
   printf("hello!\n");
   exit(0);
}
```

Q 3: Combien de lignes «hello !» imprime chacun des programmes suivants ?

Programme 1 :

```c
int main() {
  int i;

  for (i=0; i<2; i++)
    fork();
  printf("hello!\n");
  exit(0);
}
```

Programme 2 :

```c
void doit() {
  fork();
  fork();
  printf("hello!\n");
}
int main() {
  doit();
  printf("hello!\n");
  exit(0);
}
```

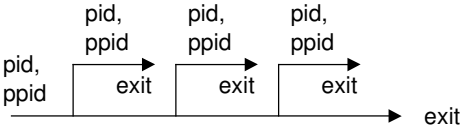
Programme 3 :

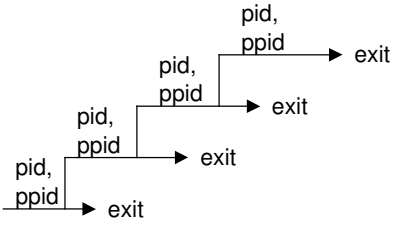
```c
int main() {
  if (fork())
    fork();
  printf("hello!\n");
  exit(0);
}
```

Programme 4 :

```c
int main() {
  if (fork()==0) {
    if (fork()) {
      printf("hello!\n");
    }
  }
}
```

Q 4: On considère les deux structures de filiation (chaîne et arbre) représentées ci-après.





Écrire un programme qui réalise une chaîne de `n` processus, où `n` est passé en paramètre de l'exécution de la commande (par exemple, n = 3 sur la figure ci-dessus). Faire imprimer le numéro de chaque processus et celui de son père. Même question avec la structure en arbre.

★ Exercice 2: Relations entre processus père et fils

Q 5: Le programme ci-après lance un processus qui attend la fin de ses fils et imprime leur code de retour. Modifier ce programme pour qu'il traite les fils dans l'ordre danslequel ils ont été créés (en affichant leur code de retour).

```c
#define N 2

int main() {
    int status, i;
    pid_t pid;

    for (i = 0; i < N; i++)
        if ((pid = fork()) == 0)  /* fils */
            exit(100+i);

    /* le père attend la fin de ses fils */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
           if (WEXITSTATUS(status) == 0) {
              printf("Le fils %d s'est terminé normalement avec le code 0.\n",pid);
              printf("Il n'a donc pas rencontré de problème.\n");
           } else {
              printf("Le fils %d s'est terminé avec le code %d.\n",pid,WEXITSTATUS(status));
              printf("Il s'agit en général d'un code d'erreur (cf. la page man du processus déclenché)\n");
           }
        else
            printf("Le fils %d s'est terminé à cause d'un signal\n", pid);
    }
    if (errno != ECHILD)
        perror("erreur dans waitpid");

    exit(0);
}
```

Q 6: On considère le programme suivant :

```c
int main() {
    int status;
    pid_t pid;

    fprintf(stderr,"Hello\n");
    pid = fork();
    fprintf(stderr,"%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1, &status, 0) > 0) {
            if (WIFEXITED(status) != 0)
                fprintf(stderr,"%d\n", WEXITSTATUS(status));
        }
    }
    fprintf(stderr,"Bye\n");
    exit(2);
}
```

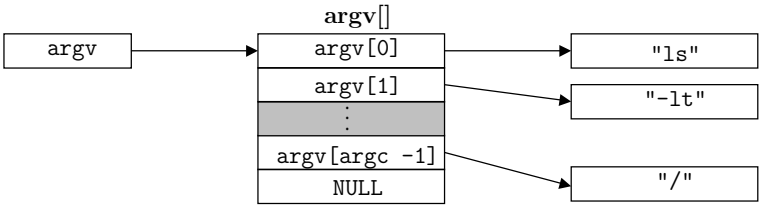
Combien de lignes ce programme imprime-t-il ? Discuter les ordres possibles dans lesquels ces lignes sont imprimées.

★ **Exercice 3:** Exécution d'un programme

La famille de primitives `exec` permet de créer un processus pour exécuter un programme déterminé (qui a auparavant été placé dans un fichier, sous forme binaire exécutable). On utilise en particulier `execvp` pour exécuter un programme en lui passant un tableau d'arguments. Le paramètre `filename` pointe vers le nom (absolu ou relatif) du fichier exécutable, `argv` vers le tableau contenant les arguments (terminé par `NULL`) qui sera passé à la fonction `main` du programme lancé. Par convention, le paramètre `argv[0]` contient le nom du fichier exécutable, les arguments suivants étant les paramètres successifs de la commande.

```c
#include <unistd.h>
int execvp(char *filename, char *argv[]);
```

Noter que les primitives `exec` provoquent le ≪recouvrement≫ de la mémoire du processus appelant par le nouveau fichier exécutable. Il n'y a donc pas normalement de retour (sauf en cas d'erreur – fichier inconnu ou permission – auquel cas la primitive renvoie -1).



Q 7: Que fait le programme suivant ?

```c
#include<stdio.h>
#include<unistd.h>
#define MAX 5
int main() {
   char *argv[MAX];
   argv[0] = "ls"; argv[1] = "-lR"; argv[2] = "/"; argv[3] = NULL;
   execvp("ls", argv);
}
```

Q 8: Écrire un programme `doit` qui exécute une commande Unix qu'on lui passe en paramètre. *Exemple :*

```
    doit ls -lt /
```