

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 supposons que tous les appels réussissent sans problème, mais une application réelle devrait bien entendu traiter les cas d'erreurs.

▷ **Question 1:** Qu'affiche l'exécution du programme suivant :

```

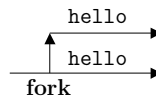
1 int main() {
2     pid_t pid;
3     int x = 1;
4
5     pid = fork();
6     if (pid == 0) {
7         printf("Dans fils : x=%d\n", ++x);
8         exit(0);
9     }
10
11     printf("Dans père : x=%d\n", --x);
12     exit(0);
13 }
```

▷ **Question 2:** On considère les deux programmes suivants et leur schéma d'exécution.

- *Exemple 1* : un clonage

```

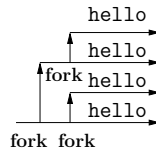
1 int main() {
2     fork();
3     printf("hello!\n");
4     exit(0);
5 }
```



- *Exemple 2* : deux clonages

```

1 int main() {
2     fork();
3     fork();
4     printf("hello!\n");
5     exit(0);
6 }
```



- *Question* : Illustrer l'exécution de ce programme :

```

1 int main() {
2     fork();
3     fork();
4     fork();
5     printf("hello!\n");
6     exit(0);
7 }
```

▷ **Question 3:** Combien de lignes «hello!» imprime chacun des programmes suivants ?

Programme 1 :

```

1 int main() {
2     int i;
3
4     for (i=0; i<2; i++)
5         fork();
6     printf("hello!\n");
7     exit(0);
8 }
```

Programme 2 :

```

1 void doit() {
2     fork();
3     fork();
4     printf("hello!\n");
5 }
6 int main() {
7     doit();
8     printf("hello!\n");
9     exit(0);
10 }
```

Programme 3 :

```

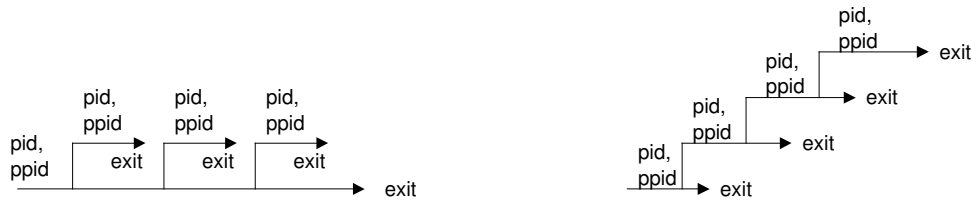
1 int main() {
2     if (fork())
3         fork();
4     printf("hello!\n");
5     exit(0);
6 }
```

Programme 4 :

```

1 int main() {
2     if (fork()==0) {
3         if (fork()) {
4             printf("hello!\n");
5         }
6     }
7 }
```

▷ **Question 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

▷ **Question 1:** Le programme ci-après lance un processus qui attend la fin de ses fils et imprime leur code de retour.

```

1 #define N 2
2
3 int main() {
4     int status, i;
5     pid_t pid;
6
7     for (i = 0; i < N; i++)
8         if ((pid = fork()) == 0) /* fils */
9             exit(100+i);
10
11     /* le père attend la fin de ses fils */
12     while ((pid = waitpid(-1, &status, 0)) > 0) {
13         if (WIFEXITED(status))
14             if (WEXITSTATUS(status) == 0) {
15                 printf("Le fils %d s'est terminé normalement avec le code 0.\n",pid);
16                 printf("Il n'a donc pas rencontré de problème.\n");
17             } else {
18                 printf("Le fils %d s'est terminé avec le code %d.\n",pid,WEXITSTATUS(status));
19                 printf("Il s'agit en général d'un code d'erreur (cf. la page man du processus déclanché)\n");
20             }
21         else
22             printf("Le fils %d s'est terminé à cause d'un signal\n", pid);
23     }
24     if (errno != ECHILD)
25         perror("erreur dans waitpid");
26
27     exit(0);
28 }

```

Modifier ce programme pour qu'il traite les fils dans l'ordre dans lequel ils ont été créés (en affichant leur code de retour).

▷ **Question 2:** On considère le programme suivant :

```

1 int main() {
2     int status;
3     pid_t pid;
4
5     fprintf(stderr,"Hello\n");
6     pid = fork();
7     fprintf(stderr,"%d\n", !pid);
8     if (pid != 0) {
9         if (waitpid(-1, &status, 0) > 0) {
10            if (WIFEXITED(status) != 0)
11                fprintf(stderr,"%d\n", WEXITSTATUS(status));
12        }
13    }
14    fprintf(stderr,"Bye\n");
15    exit(2);
16 }

```

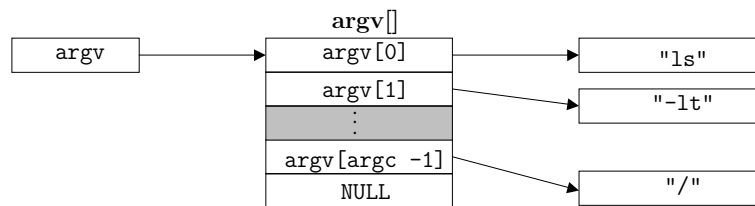
Combien de lignes ce programme imprime-t-il? Discuter les ordres possibles dans lesquels ces lignes sont imprimées.

★ **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.

```
1 #include <unistd.h>
2 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`).



▷ **Question 1:** Que fait le programme suivant ?

```
1 #include<stdio.h>
2 #include<unistd.h>
3 #define MAX 5
4 int main() {
5     char *argv[MAX];
6     argv[0] = "ls"; argv[1] = "-lR"; argv[2] = "/"; argv[3] = NULL;
7     execvp("ls", argv);
8 }
```

▷ **Question 2:** Écrire un programme `doit` qui exécute une commande Unix qu'on lui passe en paramètre.
Exemple :

```
1 doit ls -lt /
```