

Création de Processus

Exercice 2.1 Écrire un programme `pere-et-fils` qui lance un processus un fils, père et fils déclineront ensuite leur identité.

```
je m'appelle 5585 et je suis le père de 5586
je m'appelle 5586 et je suis le fils de 5585
```

Comment faire en sorte que le processus père affiche son message toujours après son fils ? Toujours avant ? Faire en sorte que le fils devienne un zombie. Faire en sorte qu'il devienne orphelin.

Exercice 2.2 Écrire un programme qui crée N processus fils (N étant passé en paramètre sur la ligne de commande) et qui attend la fin de tous les fils avant de se terminer à son tour. Les N processus fils se contenteront simplement d'afficher leur numéro de rang (de 0 à $N - 1$) sur la sortie standard.

Exercice 2.3 Écrire une fonction `System(char *commande)` qui lance un processus exécutant un shell pour interpréter la commande passée en paramètre puis attends la fin de cette interprétation pour retourner. Voir la fonction `system`.

Exercice 2.4 Il s'agit de programmer un lanceur de commandes.

1. En utilisant une des fonctions de la famille `exec`, écrire la commande `execute commande liste-de-parametres`.

```
> execute ls -l
*** execution
total 4
-rw-r--r--    1 paw      paw           240 Apr  2 10:37 #shell.c#
*** code de retour : 0
```

2. Comparer la valeur de retour retournée par votre shell et celle produite par `bash` lorsque le programme est interrompu intempestivement.

Exercice 2.5 Un embryon de shell vous est fourni dans le répertoire `Shell`. Le programme `Shell.c` est, en l'état, capable d'analyser les lignes de commande qui lui sont soumises. Pour réaliser cette analyse ce programme utilise une fonction d'*analyse syntaxique* qui renvoie l'arbre syntaxique associé à la ligne de commande¹. C'est un arbre binaire qui décrit comment sont combinées les commandes contenues dans la ligne de commande donnée. Voici la définition de la structure de données utilisée :

```
typedef enum expr_t {
    VIDE,           // Commande vide
    SIMPLE,         // Commande simple
```

1. NB. cette fonction retourne `NULL` en cas de ligne syntaxiquement incorrecte.

```

SEQUENCE,          // Séquence (;)
SEQUENCE_ET,       // Séquence conditionnelle (&&)
SEQUENCE_OU,       // Séquence conditionnelle (||)
BG,                // Tache en arriere plan
PIPE,              // Pipe
REDIRECTION_I,     // Redirection entree
REDIRECTION_O,     // Redirection sortie standard
REDIRECTION_A,     // Redirection sortie standard, mode append
REDIRECTION_E,     // Redirection sortie erreur
REDIRECTION_EO,    // Redirection sorties erreur et standard
} expr_t;

typedef struct Expression {
    expr_t type;
    struct Expression *gauche;
    struct Expression *droite;
    char    **arguments;
} Expression;

```

Par exemple, une ligne de commande comme `cat -n lol > fic &` sera décrite par un arbre ayant pour racine un nœud de type `BG` qui a pour fils gauche un nœud de type `REDIRECTION_O` dont le champ `arguments[0]` désigne la chaîne `"fic"`. Le nœud `REDIRECTION_O` a lui même pour fils un nœud de type `SIMPLE` dont le champ `arguments[]` désigne un tableau tel que `{"cat", "-n", "lol", NULL}`.

En vous appuyant sur l'exemple ci dessus, modifier la fonction `evaluer_expr()` afin de mettre au point les traitements suivants :

1. exécution d'une commande simple ;
2. mise en place d'une éventuelle redirection de la sortie standard ;
3. mise éventuelle en arrière plan d'une commande et élimination des zombies ;

Ensuite on cherchera à évaluer les séquences(;) de commandes. Par exemple la commande `ls -al ; cat -n lol` sera décrite par un arbre dont la racine sera de type `SEQUENCE`, cette racine ayant pour fils gauche une feuille de type `SIMPLE` contenant la commande simple `ls -al` et pour fils droit une autre feuille de type `SIMPLE` décrivant `cat -n lol`.

On pourra consulter la fonction `afficher_expr()` pour voir comment on peut explorer récursivement la structure de donnée associée à une expression.