Chapitre 3

Lancement d'un programme : `exec`

3.1 Rappels : Arguments en ligne de commande

La fonction `main` d'un programme peut prendre des arguments en ligne de commande. Par exemple, si un fichier `monprog.c` a permis de générer un exécutable `monprog` à la compilation,

```
$ gcc monprog.c -o monprog
```

on peut invoquer le programme `monprog` avec des arguments

```
$ ./monprog argument1 argment2 argument3
```

Exemple. La commande `cp` du bash prend deux arguments :

```
$ cp nomfichier1 nomfichier2
```

Pour récupérer les arguments dans le programme C, on utilise les paramètres `argc` et `argv` du `main`. L'entier `argc` donne le nombre d'arguments rentrés dans la ligne de commande **plus 1**, et le paramètre `argv` est un tableau de chaînes de caractères qui contient comme éléments :

- Le premier élément `argv[0]` est une chaîne qui contient le nom du fichier executable du programme ;
- Les éléments suivants `argv[1]`, `argv[2]`,etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i;
   if (argc == 1)
      puts("Le programme n'a reçu aucun argument");
   if (argc >= 2)
      {
          puts("Le programme a reçu les arguments suivants :");
          for (i=1 ; i<argc ; i++)
             printf("Argument %d = %s\n", i, argv[i]);
      }
   return 0;
}
```

3.2 L'appel système exec

3.2.1 Arguments en liste

L'appel système exec permet de remplacer le programme en cours par un autre programme sans changer de numéro de processus (PID). Autrement dit, un programme peut se faire remplacer par un autre code source ou un script shell en faisant appel à exec. Il y a en fait plusieurs fonctions de la famille exec qui sont légèrement différentes.

La fonction execl prend en paramètre une *liste* des arguments à passer au programme (liste terminée par NULL).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
   /* dernier élément NULL, OBLIGATOIRE */
   execl("/usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);

   perror("Problème : cette partie du code ne doit jamais être exécutée");
   return 0;
}
```

Le premier paramètre est une chaîne qui doit contenir le chemin d'accès complet (dans le système de fichiers) au fichier exécutable ou au script shell à exécuter. Les paramètres suivants sont des chaînes de caractère qui représentent les arguments passés en ligne de commande au main de ce programme. La chaîne argv[0] doit donner le nom du programme (sans chemin d'accès), et les chaînes suivantes argv[1], argv[2],etc... donnent les arguments.

Concernant le chemin d'accès, il est donné à partir du répertoire de travail (PWD), ou à partir du répertoire racine / s'il commence par le caractère /
(exemple : /home/remy/enseignement/systeme/script1).

La fonction execlp permet de rechercher les exécutables dans les répertoires apparaîssant dans le PATH, ce qui évite souvent d'avoir à spécifier le chemin complet.

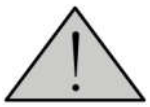
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
   /* dernier élément NULL, OBLIGATOIRE */
   execlp("emacs", "emacs", "fichier.c", "fichier.h", NULL);

   perror("Problème : cette partie du code ne doit jamais être exécutée");
   return 0;
}
```

3.2.2 Arguments en vecteur

Nous allons étudier l'une d'entre elles (la fonction `execv`). La différence avec `execl` est que l'on n'a pas besoin de conaître la liste des arguments à l'avance (ni même leur nombre). Cette fonction a pour prototype :

```
int execv(const char* application, const char* argv[]);
```

Le mot `const` signifie seulement que la fonction `execv` ne modifie pas ses paramètres. Le premier paramètre est une chaîne qui doit contenir le chemin d'accès (dans le système de fichiers) au fichier exécutable ou au script shell à exécuter. Le deuxième paramètre est un tableau de chaînes de caractères donnant les arguments passés au programme à lancer dans un format similaire au paramètre `argv` du `main` de ce programme. La chaîne `argv[0]` doit donner le nom du programme (sans chemin d'accès), et les chaînes suivants `argv[1], argv[2],etc...` donnent les arguments.



Le dernier élément du tableau de pointeurs argv doit être NULL pour marquer la fin du tableau. Ceci est dû au fait que l'on ne passe pas de paramètre `argc` donnant le nombre d'argument

Concernant le chemin d'accès, il est donné à partir du répertoire de travail ($PWD), ou à partir du répertoire racine / s'il commence par le caractère /
(exemple : `/home/remy/enseignement/systeme/script1`).

Exemple. Le programme suivant édite les fichiers .c et .h du répertoire de travail avec `emacs`.

Dans le programme, le chemin d'accès à la commande `emacs` est donné à partir de la racine `/usr/bin/emacs`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
   char * argv[] = {"emacs", "fichier.c", "fichier.h", NULL}
   /* dernier élément NULL, obligatoire */
   execv("/usr/bin/emacs", argv);

   puts("Problème : cette partie du code ne doit jamais être exécutée");
   return 0;
}
```

Remarque 3.2.1 *Pour exécuter un script shell avec `execv`, il faut que la première ligne de ce script soit*

```
#! /bin/sh
```

ou quelque chose d'analogue.

En utilisant `fork`, puis en faisant appel à `exec` dans le processus fils, un programme peut lancer un autre programme et continuer à tourner dans le processus père.

Il existe une fonction `execvp` qui lance un programme en le recherchant dans la variable d'environnement PATH. L'utilisation de cette fonction dans un programme *Set-UID* pose des problèmes de sécurité (voir explications plus loin pour la fonction `system`

3.3 La fonction `system`

3.3.1 La variable PATH dans unix

La variable d'environnement PATH sous unix et linux donne un certain nombre de chemins vers des répertoires où se trouve les exécutables et scripts des commandes. Les chmins dans le PATH sont séparés par des ' :'.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/remy/bin:.
```

Lorsqu'on lance une commande dans une console, le système va cherche l'exécutable ou le script de cette commande dans les répertoires donnés dans le PATH. Chaque utilisateur peut rajouter des chemins dans son PATH (en modifiant son fichier .bashrc sous linux). En particulier, l'utilisateur peut rajouter le répertoire '.' (point) dans le PATH, ce qui signifie que le système va chercher les commandes dans le répertoire de travail donné dans la variable d'environnement PWD. La recherche des commandes dans les répertoires a lieu dans l'ordre dans lequel les répertoires apparaîssent dans le PATH. Par exemple, pour le PATH donné ci-dessus, la commande sera recherchée d'abord dans le répertoire `/usr/local/bin`, puis dans le répertoire `/usr/bin`. Si deux commandes de même nom se trouve dans deux répertoires du PATH, c'est la première commande trouvée qui sera exécutée.

3.3.2 La fonction `system`

La fonction `system` de la bibliothèque stdlib.h permet directement de lancer un programme dans un programme C sans utiliser `fork` et `exec`. Pour celà, on utilise l'instruction :

```
#include <stdlib.h>
...
system("commande");
```

Exemple. La commande unix `clear` permet d'effacer la console. Pour effacer la console dans un programme C avec des entrées-sorties dans la console, on peut ustilser :

```
system("clear");
```

Lorsqu'on utilise la fonction `system`, la commande qu'on exécute est recherchée dans les répertoires du PATH comme si l'on exécutait la commande dans la console.

3.4 Applications suid et problèmes des sécurité liés system, execlp ou exevp

Dans le système unix, les utilisateurs et l'administrateur (utilisateur) on des droits (que l'on appelle privilèges), et l'accès à certaines commandes leur sont interdites. C'est ainsi que, par exemple, si le système est bien administré, un utilisateur ordinaire ne peut pas facilement endomager le système.

Exemple. Imaginons que les utilisateurs aient tous les droits et qu'un utilisateur malintentioné ou distrait tape la commande

```
$ rm -r /
```

Cela supprimerait tous les fichiers du système et des autres utilisateurs et porterait un préjudice important pour tous les utilisateurs du système. En fait, beaucoup de fichiers sont interdits à l'utilisateur en écriture, ce qui fait que la commande `rm` sera ineffective sur ces fichiers.

Pour celà, lorsque l'utilisateur lance une commande ou un script (comme la commande `rm`), les privilèges de cet utilsateur sont pris en compte lors de l'exécution de la commande.

Sous unix, un utilisateur A (par exemple root) peut modifier les permissions sur un fichier exécutable pour que tout autre utilisateur B puisse exécuter ce fichier avec ses propres privilèges (les privilèges de A). Cela s'appelle les permissions suid.

Exemple. Supposons que l'utilisateur `root` tape les commandes suivantes :

```
$ gcc monprog.c -o monprog
$ ls -l
-rwxr-xr-x  1 root root 18687 Sep  7 08:28 monprog
-rw-r--r--  1 root root  3143 Sep  4 15:07 monprog.c
$ chmod +s monprog
$ ls -l
-rwsr-sr-s  1 root root 18687 Sep  7 08:28 monprog
-rw-r--r--  1 root root  3143 Sep  4 15:07 monprog.c
```

Le programme `moprog` est alors suid et n'importe quel utilisateur peut l'exécuter avec les privilèges du propriétaire de monprog, c'est à dire `root`.

Supposons maintenant que dans le fichier `monprog.c` il y ait l'instruction

```
system("clear");
```

Considérons un utilisateur malintentionné `remy`. Cet utilisateur modifie son PATH pour rajouter le répertoire '.', (point) mais met le répertoire '.' au tout début de PATH

```
$ PATH=.:\PATH
$ export PATH
$ echo \PATH
.:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/remy/bin:.
```

Dans la recherche des commandes dans les répertoires du PATH, le système cherchera d'abord les commandes dans le répertoire de travail '.'. Supposons maintenant que l'utilisateur `remy` crée un script appelé `clear` dans son répertoire de travail. qui contienne la ligne
`rm -r /`

```
$ echo "rm -r /" > clear
$ cat clear
rm -r /
$ chmod +x clear
$ monprog
```

Lorsque l'utilisateur `remy` va lancer l'exécutable `monprog` avec les privilèges de `root`, le programme va exécuter le script `clear` de l'utilisateur (au lieu de la commande `/usr/bin/clear`) avec les privilèges de `root`, et va supprimer tous les fichiers du système.

Il ne faut jamais utiliser la fonction `system` ou la fonction `execvp` dans une application suid, car un utilisateur malintentioné pourrait exécuter n'importe quel script avec vos privilèges.

3.5 Exercices

Exercice 3.1 $(*)$ Écrire un programme qui prend deux arguments en ligne de commande en supposant qu ce sont des nombres entiers, et qui affiche l'addition de ces deux nombres.

Exercice 3.2 $(*)$ Ecrire un programme qui prend en argument un chemin vers un répertoire R, et copie le répertoire courant dans ce répertoire R.

Exercice 3.3 $(*)$ Ecrire un programme qui saisit un nom de fichier texte au clavier et ouvre ce fichier dans l'éditeur `emacs`, dont le fichier exécutable se trouve à l'emplacement `/usr/bin/emacs`.

Exercice 3.4 $(**)$ Ecrire un programme qui saisit des noms de répertoires au clavier et copie le répertoire courant dans tous ces répertoires. Le programme doit se poursuivre jusqu'à ce que l'utilisateur demande de quitter le programme.

Exercice 3.5 $(**)$ Ecrire un programme qui saisit des nom de fichiers texte au clavier et ouvre tous ces fichiers dans l'éditeur `emacs`. Le programme doit se poursuivre jusqu'à ce que l'utilisateur demande de quitter.

Exercice 3.6 $(***)$ Considérons les coefficients binômiaux C_n^k tels que

$$C_i^0 = 1 \text{ et } C_i^i = 1 \text{ pour tout } i$$

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$$

Écrire un programme pour calculerC_n^k qui n'utilise aucune boucle (ni `while` ni `for`), et qui n'ait comme seule fonction que la fonction `main`. La fonction `main` ne doit contenir aucun appel à elle-même.On pourra utiliser des fichiers textes temporaires dans le répertoire `/tmp`.