esiea

Programmation Système

LAB2 - Fonctions générales

Table des matières

By Roxy

1 Objectifs

Ce LAB présente les notions fondamentales de programmation système. Il est basé sur la mise en parallèle de notions relatives aux commandes Shell (et Scripts Shell) avec leur équivalent en langage C.

- Documents : Ceux du LAB1 auxquels s'ajoutent « Support Linux – Le script Shell ».
- Notions abordées : l'environnement de développement, les variables d'environnement systèmes, le débogage.
- Commandes et fichiers exploités : GCC, commandes de gestion de fichiers, DDD (ou tout autre débogueur).

Travail à rendre : Vous devrez répondre directement à plusieurs questions au sein de ce document. Vous le copierez sur Moodle sous le nom «LAB2_*NOM*.pdf », vous devrez aussi rendre les différents fichiers bash (1 fichier) et C (2 fichiers).

Soit en tout 4 fichiers à déposer sur Moodle !

2 Rappels

Pour vous refamiliariser avec les scripts Shell, lisez et effectuez les commandes décrites dans le support « Linux – le script Shell.pdf ».

3 Retour sur le LAB1

Le LAB1 a été l'occasion de rappeler les bases du débogage. Un de vos camarades à rendu un programme bogué (il a donc eu 0, car ce dernier ne compile même pas !).

Déboguez ce programme et expliquer la ou les erreurs ci-dessous.
Rendez le code C corrigé sur Moodle dans un fichier appelé part-3_*NOM*.c

Toutes les modifications sont inclues dans les commentaires du fichier .c

4 Accès à l'environnement

4.1 Introductiona

Il est possible d'indiquer à un programme des éléments en paramètres qu'il pourra recevoir et utiliser à sa convenance. Ces éléments peuvent être de différents types (chaîne de caractère, valeurs numériques, fichiers, etc.).
Il est aussi possible de définir des variables au niveau de l'interpréteur de commandes (Shell). Un certain nombre de ces variables sont exposées et deviennent accessibles par l'ensemble des processus descendants du processus où elles ont été déclarées (en général le Shell de login).
Tous ces éléments constituent **l'environnement d'un programme**.

4.2 Les paramètres

En Shell :

En vous appuyant sur le document « Support Linux – Le script Shell », réalisez un script qui effectue le traitement suivant :

- testez le paramètre que vous passez en argument;
- si le paramètre est un répertoire : affiche "*XXX* : est un répertoire";
- si le paramètre est un fichier ordinaire : affiche "*XXX* : est un fichier";
- sinon affiche un message d'erreur (voir figure suivante).

```
[rexy@localhost ~]$ ./script2.sh tmp
        tmp est un répertoire.
[rexy@localhost ~]$ ./script2.sh script1.sh
        script1.sh est un fichier ordinaire
[rexy@localhost ~]$ ./script2.sh foo
        foo n'existe pas dans l'arborescence ou est d'un autre type.
[rexy@localhost ~]$ ./script2.sh
        Le paramètre est manquant
[rexy@localhost ~]$ ./script2.sh foo bar
        Trop de paramètres
[rexy@localhost ~]$
```

Rendez le script sur Moodle dans un fichier appelé part-4_*NOM*.sh

En C :

`main()` est une fonction comme une autre. Elle reçoit des paramètres et renvoie une valeur.

```
int main (int argc, char *argv[], char *envp[]);
```

Explication des paramètres :

- **`int argc`** (argument count) : nombre d'arguments passés au programme ; y compris le nom du programme lui-même. Ainsi, cette variable ne peut jamais être nulle ;
- **`char *argv[]`** (argument value) : pointeur vers un un tableau de pointeurs pointant vers un tableau de caractère de format chaîne (dernier caractère = 0). Chaque chaîne contient le nom d'un argument passé au programme. **`Argv[0]`** contient le nom du programme lui-même. Le tableau se termine (**`argv[argc]`**) par un pointeur de valeur nulle;
- **`char envp[]`** (environment program) : comme pour **`argv`**, mais le tableau de chaîne contient les variables d'environnement exportées par les processus ascendants. La norme SUSV4 préconise de ne plus utiliser ce pointeur au profit d'un autre dispositif (cf. § suivant);
- Valeur renvoyée (**`int`**) : entier utilisable par le processus appelant (récupérée par **`$?`** en Shell).

Exemple :

```
#include <stdio.h>
int main (int  argc, char *argv[])
{
      printf ("Je me nomme %s et on m'a envoyé %d paramètres\n", argv[0],argc-1);
}
```

Copiez ici une capture d'écran de son exécution avec 3 paramètres.

```
● → lab2 gcc -o code_c_arguments.o code_c_arguments.c
● → lab2 ./code_c_arguments.o Je Ne suis
  Je me nomme ./code_c_arguments.o et on m'a envoyé 3 paramètres
○ → lab2
```

4.3 Les variables d'environnement

En Shell :

Quelles commandes permettent de connaître les variables d'environnement définies sur votre système ?

Les deux commandes « env » et « printenv » affichent toutes les variables environnement. On a aussi la commande « set » qui affiche non seulement les variables d'environnement, mais aussi toutes les variables locales définies dans la session shell

- Créez une nouvelle variable « école » initialisée à « esiea » et affichez-la.
- Créez un Script-Shell qui affiche cette variable.
- Que devez-vous faire pour que cette variable soit connue de votre script ?

Détail des commandes et du script :

Pour que la variable soit connue de notre script, il faut utiliser la commande « export » qui va partager nos variables environnement avec tout sub-shell cree (soit lors de l'execution d'un script) :

Commandes :

➜ lab2 export ecole="esiea"

➜ lab2 echo $ecole

esiea

➜ lab2 ./afficher_ecole.sh

esiea

Code :

!/bin/bash

echo $ecole

En C :

La primitive **`getenv()`** permet d'obtenir la valeur courante d'une variable d'environnement particulière (**man 3 getenv**).

```
#include <stdlib.h>
char* getenv (char *string);
```

Explication des paramètres :

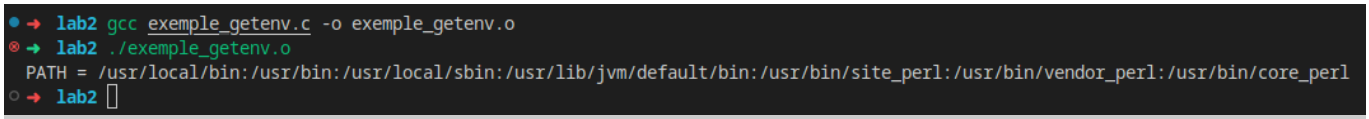
- paramètre **`char *string`** : chaîne contenant la variable dont on veut obtenir la valeur ;
- valeur renvoyée (**`char*`**) : pointeur sur une zone mémoire statique contenant la chaîne correspondant à la variable demandée si celle-ci existe.

Exemple :

```
#include <stdlib.h>
#include <stdio.h>
void main (void)
{
      char *ptr;  /* récupère le résultat de getenv() */

      ptr=getenv(''PATH'');
      printf(''PATH = %s\n'', ptr);
}
```

Copiez ici une capture d'écran de son exécution.



Toutes ces variables d'environnement héritées sont présentes dans un tableau de chaînes **`extern char **environ`** qu'il faut déclarer avant exploitation (**man 7 environ**).

Exemple :

```
#include <stdio.h>
extern char **environ;
int main (void)
{
      int nro_var = 0;
      for (nro_var=0; environ[nro_var]!= NULL; nro_var++)
            printf ("%d :\t%s\n", nro_var, environ[nro_var]);
      return 0;
}
```

5 Lancement d'une commande Shell

En Shell :

Rappel - Le **|** (pipe) permet de chaîner les commandes Shell (la sortie standard de la première commande est fournie en entrée de la deuxième commande et ainsi de suite).

Copiez ici la suite de commandes Shell permettant de connaître le nombre de variables d'environnement déclarées sur votre système ? Quel est ce nombre ?

```
$ env | wc -l
64
```

En C :

Il est possible, depuis un programme C, de lancer l'exécution d'une commande Shell via l'appel de la fonction **`system()`** (man 3 system). Cette fonction lance un Shell qui va exécuter la commande voulue. Le Shell lancé ne réalise aucune communication avec le programme appelant. Celui-ci reste en attente jusqu'à la fin de l'exécution de la commande.

Remarque : Il est parfois plus judicieux de reprogrammer en C la commande que l'on veut faire exécuter. En effet, la rapidité d'exécution est accrue et la communication est possible.

```
#include <stdlib.h>
int system (char *commande);
```

Explication des paramètres :

- paramètre **`char *commande`** : chaîne contenant la commande que l'on veut exécuter ;
- Valeur renvoyée (**`int`**) : un nombre contenant le statut de fin du Shell lancé.

Exemple :

```
#include <stdlib.h>
main ()
{
      system (''date'');
}
```

Réalisez un programme C qui appelle une commande Shell (ou une suite de commandes « pipées ») permettant d'afficher le nombre de variables d'environnement de votre système. Le programme doit

aussi afficher le statut de retour d'appel de la commande shell, collez une capture d'écran de son exécution ci-dessous **(Rendez le code C sur Moodle dans un fichier appelé part-5.c)**.

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

int main ()
{
int status_code = system ("env | wc -l");
printf("Code de retour : %d\n", status_code);
}
```

6 Gestion des erreurs

Le traitement des erreurs et leur analyse permettent d'améliorer la stabilité du programme tout en assurant un bon niveau de maintenabilité. Pour ces deux raisons, il est nécessaire de savoir gérer les erreurs et exploiter le débogueur.

En Shell :

Un script Shell s'exécute séquentiellement. Quand il rencontre une erreur : il s'arrête ; il identifie l'erreur ; il affiche un message sur le canal 2 des erreurs (ce canal est dirigé par défaut vers l'écran) et il continue la lecture du script. Il est possible de rediriger ce canal vers un fichier en mode concaténation (**`2>>error_file`**) ou vers un fichier vide (**`2>/dev/null`**).

Exemple :

```
#!/bin/bash
NB_USERS=`cat /etc/password | wc -l`
# le fichier ''/etc/password'' n'existe pas (c'est ''/etc/passwd'')
if [ $NB_USERS -gt 0 ]
then
      echo ''$NB_USERS sont déclarés sur votre système''
else
      echo ''une erreur a dû se produire''
fi
```

En termes de débogage, le développeur peut afficher l'état de sortie de chaque commande de son script afin de suivre précisément son exécution (cf. §2.3 du document « support Linux – le script Shell »).

Copiez ici le résultat d'exécution de ce script avec et sans redirection des erreurs. Testez l'option de suivi d'exécution des commandes.

AVEC UNE ERREUR :

➜ lab2 ./script_gestion_erreur.sh

cat: /etc/password: No such file or directory

une erreur a dû se produire

SANS ERREUR :

➜ lab2 ./script_gestion_erreur.sh

39 sont déclarés sur votre système

En C :

Par défaut, les appels système en échec renvoient « **-1** » ou parfois « **NULL** ». Pour fournir un complément d'information au programme appelant, la variable « **extern int errno** » contient un numéro d'erreur indiquant la cause réelle de l'échec (**man 3 errno**). À chaque numéro correspond une constante prédéfinie. Cette variable et les constantes associées sont déclarées dans le fichier header « **errno.h** ».

```
#define EPERM     1     /* Operation not permitted */
#define ENOENT    2     /* No such file or directory */
#define ESRCH     3     /* No such process */
#define EINTR     4     /* Interrupted system call */
#define EIO       5     /* I/O error */
#define ENXIO     6     /* No such device or address */
                        /* etc. */
```

Associée à cette variable, un pointeur sur un tableau de chaînes statiques « **extern char *strerror (int _ernum)** » peut être exploitée pour connaître le descriptif de l'erreur en fonction de son numéro (**man 3 strerror**). Ce pointeur est déclaré dans le fichier header « **string.h** ».

Exemple :

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
void main (void)
{
      FILE *file_desc = NULL;
      if ((file_desc = fopen(''/etc/password'', ''r'') == NULL)
      //le fichier ''/etc/password'' n'existe pas (c'est ''/etc/passwd'')
      {
            printf (''Erreur numéro : %d\n'', errno);
            printf (''Description : %s\n'', strerror(errno));
      }
      else fclose (file_desc);
}
```

Une solution plus directe consiste à exploiter la fonction **void perror (const char *s)** déclarée dans le fichier header **stdio.h**. Cette fonction affiche **sur la sortie des erreurs** un texte choisi par le programmeur immédiatement suivi du message d'erreur correspondant à **errno.**

Exemple

```
#include <stdio.h>
void main (void)
{
      FILE *file_desc = NULL;
      if ((file_desc = fopen(''/etc/password'', ''r'')) == NULL)
      // le fichier ''/etc/password'' n'existe pas (c'est ''/etc/passwd'')
      {
            perror (''Erreur d'ouverture de fichier '');
      }
      else fclose (file_desc);
}
```

- Testez le programme ci-dessus. **Conseil :** les bonnes pratiques de programmation exigent **de tester systématiquement** la valeur de retour liée au résultat d'exécution des fonctions**.**
- En utilisant la variable externe « **environ** » (cf. man), écrivez un programme C qui calcule et affiche le nombre de variables système. Exploitez le débogueur DDD pour lancer votre programme en mode « pas à pas » et afficher la variable « **environ[nro_variable]** » dans le cadre de suivi des variables.

Faites une copie d'écran de ce débogueur quand il est sur la 29e variable système. Le débogueur doit montrer le code source et le contenu de la variable environ[nro_variable].

```
(gdb) step
9               c++;
(gdb) print c
$8 = 28
(gdb) step
8           while (environ[c] != NULL) {
(gdb) step
9               c++;
(gdb) print c
$9 = 29
(gdb) print environ[c]
$10 = 0x7fffffffe341 "GJS_DEBUG_OUTPUT=stderr"
(gdb) list
4
5       int main() {
6           int c = 0;
7
8           while (environ[c] != NULL) {
9               c++;
10          }
11
12          printf("Nombre de variables système : %d\n", c);
13
```