BELLAALLI Abderrachid
HAKIZIMANA Aymar Davy
NECHIFFOR Bogdan
POURBAIX Michaël

Rapport de Projet

Consigne :

Nous devions réaliser un petit programme en C qui devait répondre à cet énoncé :

"Écrire un programme qui prend des shell scripts (contenant des commandes) en paramètre dans la ligne de commande et qui les exécute. La réception des shell scripts et leur lecture se fait par un processus père qui ensuite envoie les commandes à un autre processus, le processus fils. Ce dernier exécute les commandes une par une et envoie les résultats au père afin qu'il les affiche."

2- Analyse du projet réalisé :

Tous d'abord les bibliothèques utilisées nous permettent d'avoir recours aux Appel système, fonctions et primitives.

Ensuite, nous définissons des *#define* pour pouvoir les modifier simplement.

Notre première fonction est la fonction main où l'on va introduire deux paramètres l'*argc* (*argument count* : qui va compter le nombre d'argument du tableau) et l'*argv* (*argument vector* : qui va contenir les arguments qui sont dans le tableau) dans notre cas, c'est ce que l'utilisateur va taper en ligne de commande après l'appel au programme (./shellExe2 monShell.sh monShell2.sh).

On dit au C, qu'on va utiliser une fonction qui s'appelle *my_popen()* car sinon il ne sait pas quoi faire s'il rencontre la fonction.

On déclare les variables nécessaires, ensuite on teste qu'il y a au moins un argument fourni par l'utilisateur. On fait une boucle (while) pour parcourir le tableau *argv []* (dans notre cas les *shellScripts* à exécuter) tant que le nombre des shellScripts est plus petit que le nombre d'arguments (argc). Et on ouvre (open) chaque fichier shellScript en lecture seule (O_RDONLY).

Ensuite on vérifie que la taille du fichier ouvert ne dépasse pas la taille maximale du fichier (CONTENT_SIZE 2048). Une fois cette vérification terminée on ferme le fichier.

On utilise le strtok () pour découper le contenu du shellScripts en plusieurs lignes, et on stock ces lignes dans une variable « ligne » (pointeur sur un char). A chaque ligne nous appelons la fonction my_popen () avec comme paramètre « ligne ».

Dans la fonction my_popen () :

Premièrement, on crée un tableau de deux valeurs (0 : pour lire et 1 : pour écrire) et on utilise pipe () pour le transforme en pipe.

Après cela on va utiliser l'appel système fork () pour créer un processus père et un processus fils.

On commence par le père (parce que le pid est supérieur à 0), dans celui-ci on va lire tant que l'on reçoit. Par défaut le fork affiche les résultats dans le STDOUT du père, ce n'est pas ce que l'on souhaite. Nous utilisons donc dup2() pour effectuer une redirection au STDOUT du fils afin que les résultats soient bien affichés sur le STDOUT du fils, sans oublier de bien fermer le pipe du père.

Ensuite on va dans le fils, on crée dynamiquement un tableau de pointeur qui contient les différents arguments à exécuter (les commandes Shell).

On découpe la ligne en mot.

Dans le tableau dynamique (qui a une taille variable), on utilise *malloc()* pour le premier argument (mot) du tableau et *realloc()* pour les autres arguments. À la fin du tableau on n'oublie pas de le fermer avec un NULL.

Pour finir, on va chercher la commande dans le tableau dynamique, on l'exécute et on envoi le résultat au père pour qu'il l'affiche.

3- Planning :

On a fait des petites réunions, chaque semaine, où l'on a mis en commun ce que l'on avait réfléchit et trouver sur le projet, et nous avons avancé petit à petit comme cela.

4- Conclusion :

Globalement l'entièreté du travail demandé est réalisée dans notre projet.

Nous avons rencontré beaucoup de difficultés avec les Appels Système car lorsqu'on appelle un AS, il ne faut pas oublier de le fermer et voir s'il n'y a pas de problème avec l'AS en question. Si on a un problème avec un AS on doit quitter le programme.

Nous avons également rencontré des erreurs de segmentation lors de la compilation (Cette erreur se produit lorsque que l'on souhaite accéder à une adresse qu'il ne trouve pas.) avec le compilateur nous avons utilisé le paramètre « *-g3* » et « *valgrind* » pour trouver nos erreurs et les débugger par la suite.

Nous avions d'abord créé un tableau statique car nous n'avions pas compris la différence entre statique et dynamique, nous avons alors modifié le code pour créer un tableau dynamiquement. Lors de cette étape nous avons rencontré des difficultés avec la fin du tableau, quand nous devions attribuer l'espace pour le NULL (pour finir le tableau).

Également des soucis se sont présenté à nous avec le strtock () et les pointeurs (qui sont toujours un peu flou pour nous).

Pour finir, ce projet nous a beaucoup aidé à mieux comprendre à la fois le fonctionnement de la mémoire en linux et le comportement du langage C. Ce projet permet aussi de mettre en pratique beaucoup de chose vue pendant le cours théorique.

Annexe : Code du Projet C 2021

BELLAALI Abderrachid, HAKIZIMANA Aymar Davy, NICHIFOR Bogdan et POURBAIX Michaël

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#define CONTENT_SIZE 2048
#define SIZE_READ_PILE 10

void my_popen(char ligne[]);



void main (int argc, char* argv[]) {
    int fd;
    char contenu[CONTENT_SIZE];
    ssize_t size_read;
    int qte;



        if (argc == 1) {
            printf("Pas assez d'arguments.\n");
            exit(0);
        }


        for (int j = 1; j < argc; ++j)
        {
            fd = open(argv[j], O_RDONLY);

            if (fd == -1) {
                perror("Ouverture du fichier");
                exit(1);
            }

            size_read = read(fd, contenu, CONTENT_SIZE);

            if (size_read == -1) {
                perror("Lecture du fichier");
                exit(1);
            }

            else if (size_read == CONTENT_SIZE) {
                printf("Le fichier est trop grand pour être executé.\n");
                exit(1);
            }
```

```
            if(close(fd) == -1) {
                perror("Fermeture du fichier");
                exit(1);
            };

            char * ligne = strtok ( contenu, "\n" );
            while ( ligne != NULL ) {

                my_popen(ligne);

                // On demande le token suivant.
                ligne = strtok ( NULL, "\n" );
            }
        }


    exit(0);
}



void my_popen(char ligne[])
{
    pid_t pid;
    int pipeFd[2];


    if (pipe(pipeFd) == -1) {
      perror("Problème création du Pipe().");
      exit(1);
    }

    pid = fork();

    if( pid < 0 ) {
      perror("Problème avec le Fork() !");
      exit(1);
    }

    else if ( pid > 0 )
    {
        if (close(pipeFd[1]) == -1) {
          perror("Problème avec fermeture Pipe() du parent !");
          exit(1);
        }

        char affichage[SIZE_READ_PILE];
        ssize_t quantite;


        while(quantite = read(pipeFd[0], affichage, SIZE_READ_PILE))
        {
```

```
        if (quantite == -1){
          perror("Probleme avec le Read() !");
          exit(1);
        }

        if (write(STDOUT_FILENO, affichage, quantite) == -1){
          perror("Probleme avec le Write() !");
          exit(1);
        }
    }


    if (close(pipeFd[0]) == -1) {
      perror("Problème fermeture Pipe() du parent !");
      exit(1);
    }

  return;
}


else
{
    char ** paramsList;

    int i = 0;

    if (close(pipeFd[0]) == -1) {
       perror("Problème fermeture Pipe() du fils !");
       exit(1);
    }


    char * mot = strtok ( ligne, " " );
    while ( mot != NULL )
    {

        if (i == 0) {

            paramsList = (char**) malloc(2 * sizeof(char*));
            if (paramsList == NULL)
            {
                perror("Problème avec malloc() :");
                exit(1);
            }
            paramsList[0] = mot;
        }

        else {
            paramsList = realloc(paramsList, (i + 2) * sizeof(char*));
            paramsList[i] = mot;
```

```
                if (paramsList == NULL)
                {
                    perror("Problème avec realloc() :");
                    exit(1);
                }
            }
            paramsList[i] = mot;

            mot = strtok ( NULL, " " );

            ++i;
        }
        paramsList[i] = NULL;

        if (dup2(pipeFd[1], STDOUT_FILENO) == -1) {
          perror("Probleme avec dup2() !");
          exit(1);
        }

        if (close(pipeFd[1]) == -1) {
          perror("Problème fermeture Pipe() du fils !");
          exit(1);
        }

        if (execvp(paramsList[0], paramsList) == -1) {
          perror("Probleme avec Execvp() !");
          exit(1);
        }

      exit(0);
    }

}
```