Nom : Khallouk Achraf

SCRIPTING SHELL ET PROGRAMMATION SYSTÈME

Projet de Fin de Module

Introduction:

À l'amorçage du système, on peut voir l'écran de connexion. Nous ouvrons une session dans le système à l'aide de notre nom d'utilisateur, suivi de notre mot de passe.

Dès lors, nous pouvons voir l'interpréteur de commandes où nous saisissons habituellement les commandes à exécuter. L'interpréteur de commandes, comme le décrit Richard Stevens dans son ouvrage Advanced Programming in the Unix Environment, est un interpréteur en ligne de commande qui lit des entrées utilisateur et exécute des commandes.

C'était le point de départ pour moi. Un programme (notre *shell*) exécutant un autre programme (ce que l'utilisateur saisit à l'invite de commande). Je savais que execve et sa famille de fonctions pouvaient le faire, mais je n'avais jamais pensé à son utilisation pratique.

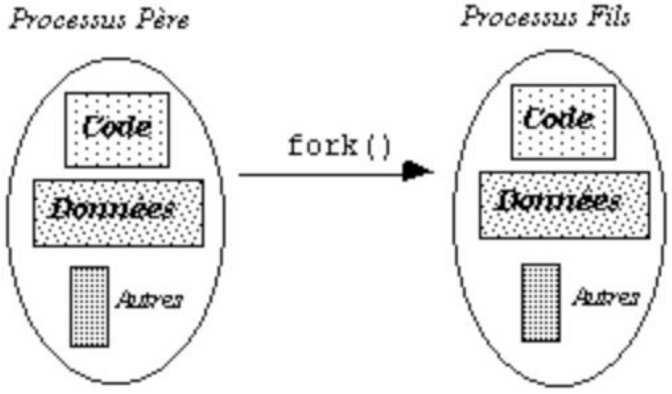
Problématique:

Il s’agit de développer un SHELL personnalisé, en langage C. Le programme de ce SHELL devrait comprendre les commandes et les exécuter.
Il est demandé d’utiliser la majorité des appels systèmes et techniques de communication et de synchronisation entre processus vus au cours.
Une fois sur le terminal, on devrait exécuter le programme SHELL développé. Cela donnera la main à une invite de commande personnalisée. Une fois on tape une commande, avec des signes spéciaux au SHELL développé, le système devrait répondre à la commande. Par exemple pour lister les fichiers et répertoires du répertoire courant, on pourrait taper par exemple une commande : Lister (qui remplace la commande reconnu par le SHELL BASH : ls), et pour supprimer un fichier on pourrait utiliser Supprimer (à la place de la commande du BASH : rm).

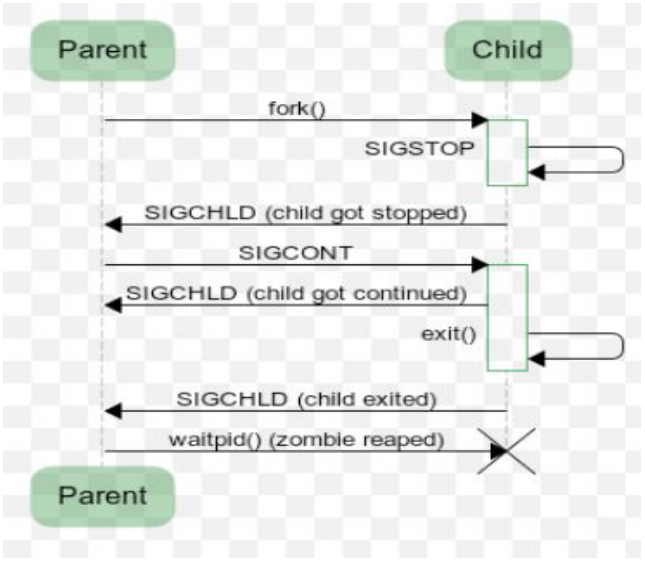
Et ainsi de suite.
Votre SHELL sevrait exécuter la plupart des commandes Linux les plus utilisés, doit comprendre les tubes anonymes, et doit utiliser les techniques de communication et de synchronisation entre processus vues au cours (tubes, signaux, moyens de communication IPC).

Notions utilisée dans le projet:

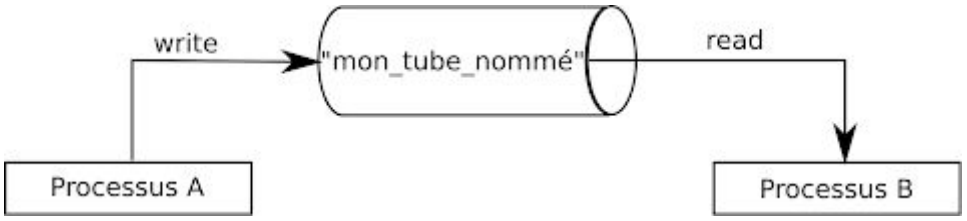
***fork*():** Comme on l'a déjà dit, la primitive fork() permet la création dynamique d'un nouveau processus, il duplique le processus père, donne au processus fils un nouveau numéro pid et le fait hériter des principales informations du père.



les tubes anonyme: est un moyen de transmission d'information entre des processus ayant un ancêtre commun. Les descendants ayant été crées par fork( )
Signal: Un signal est un moyen de communication indiquant une action à entreprendre. Il n'y pas de communication de données.
Example:



tubes nommées: C'est des tubes qui ont un nom dans le système de fichiers. Tout processus peut l'ouvrir, à condition d'en connaître le nom et d'en avoir les droits d'accès.



Semaphore: Un sémaphore est une variable accessible seulement à l'aide des opérations P (Proberen :tester) et V (Verhogen : incrémenter)

Comprendre le shell:

Commençant par la fonction main de la fichier main: qui lance init() , printPath() et start_shell()

```
int main(int argc, char **argv)
 {


    // Chargez le fichier de configuration.
     init();
     printPath(); // print répertoire actuel
     printf(" \n ");
     start_shell(); // lancer le shell



     return 0;}
```

init() : est une simple fonction qui imprime un message de 'welcoming', et attend 2 secondes pour afficher le message, et puis continuer.

printPath(): est aussi est une simple fonction qui imprime le répertoire courant

```
void printPath()
{
   char cwd[1024];
   getcwd(cwd, sizeof(cwd));
  printf("\nrépertoire courant: %s", cwd);
}
```

start_shell(): est la fonction coeur de cette shell, il attend deux signal : SIGINT, SIGSTOP. dans le cas de SIGINT le shell affiche un seule message, ms dans le cas de SIGSTOP , si vous cliquez sur CTRL-Z vous pouvez allez vers github repository, qui contient le code source.

```
void start_shell(void)
{
 char *line;
 char **args;
    Initsem(S1, 1);
   signal(SIGINT, GetSignal);
   signal(SIGSTOP,GetSignal);
   while (1)
   {
       printf("%s#",getenv("USER"));
       line = read_line();
       args = split_line(line);
       cmd_execute(args);
         if(strcmp(args[0],"cd")==0 || strcmp(args[0],"enter")==0){
         chdir(args[1]);
       }
       free(line);
       free(args);
   }
}
```

read_line et split_line sont des fonctions définies dans shell_function.h, le premier est une fonction qui lit de l'interpréteur de commandes et les stocker dans un variable ‘line’,

```
char *read_line(void) //Lire une ligne d'entrée de stdin.
{ char *line = NULL;
 ssize_t bufsize = 0;  //  allouer un buffer
 if (getline(&line, &bufsize, stdin) == -1){
   if (feof(stdin)) {
     exit(0);
   } else  {
     perror("readline error");
     exit(1); }
 }
 return line;}
```

La deuxième est split_line() qui prend comme attribut le variable que read_line a retourné, et les devise en des tokens, puis le retourner dans un tableau.

```
char **split_line(char *line) //Divisez une ligne en tokens
{
//allouer un buffer
// creer les tokens
char **tokens = malloc(bufsize * sizeof(char*));
 char *token, **tokens_backup;
//verifier que les tokens ne sont pas null
if (!tokens) {...}
//Divisé la line
 token = strtok(line, SPACE);
 while (token != NULL) {....}
returner le tableau des tokens
 return tokens;
}
```

cmd_execute(var): est une fonction qui exécute la ligne qu'on a entré, il prend les tokens comme attribut, les diviser on args[i] et les exécuter dans un processus fils et utilisant les sémaphores qui garantit que ceux-ci ne peuvent y accéder que de façon séquentielle. et finalement exécuté la commande utilisant cmd qui est une compulation de cmd.c

```
void cmd_execute(char **args)
{ if (fork() == 0){
     P(S1);
execlp("/Users/devmqk/Desktop/00000/MT/linux\(appel_systeme\)/appelSysteme/projet/c
md",
"/Users/devmqk/Desktop/00000/MT/linux\(appel_systeme\)/appelSysteme/projet/cmd",
args[0], args[1], args[2], NULL);
       .....}
```

l'exécution de la commande:

les commande seront de la format : cmd arg[1] arg[2] arg[3]

Le fichier cmd, tout d'abord stock les noms de commandes avec des int spécifiée dans un tableau des structures,

```
typedef struct      //La structure
{   char *key;   int val;} CMD;
CMD table_commandes[] = {......}  //Tableau des structures
```

Puis il lance un switch qui a comme attribut la commande entré par l'utilisateur, et comme switch en langage c ne fonctionne pas avec les string, j'ai utilisé une fonction

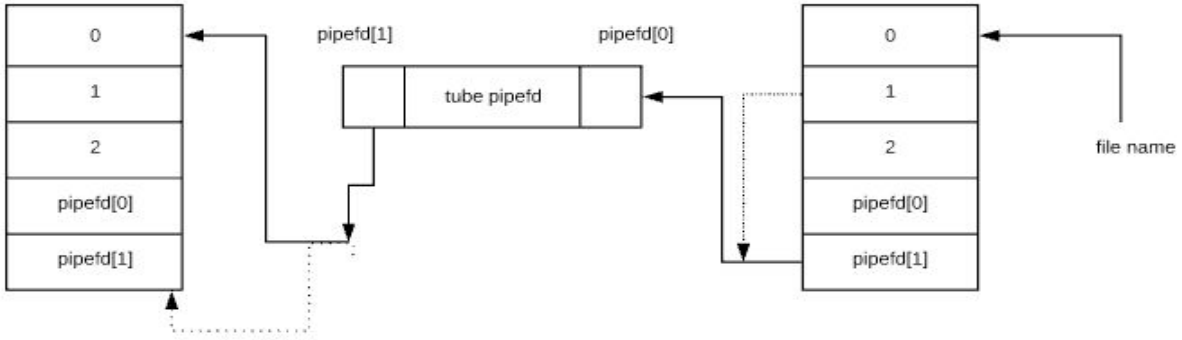
cmnd_keys(char *) qui prend la commande et les comparer avec les keys de notre tableau table_commande[], puis retourner la valeur de ce clé , par exemple si on a entrer la commande : cd dossier1 , cmnd_keys va le comparer et retourner la valeur CD qui est défini en `shell_function.h` comme 11, puis le switch prendra la valeur int, et exécuter la commande.

```
switch (cmnd_keys(cmd))
  {case …..}
```

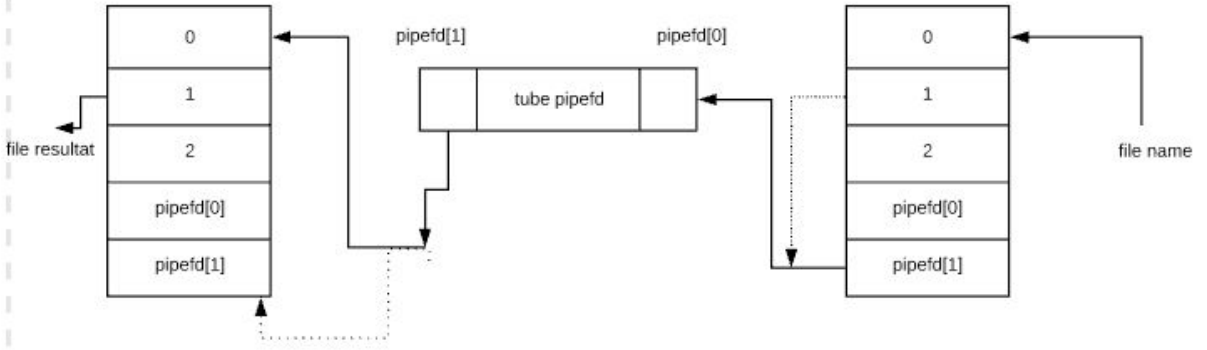
L'utilisation des tubes nommés:

Dans ce projet j'ai utilisé les tubes nommées, pour utileser '<' et '>' pour importer et pour extraire
Exécution d'executeFileOutCommand pour utiliser ">" et executeFileInCommand pour utiliser "<"

```
int executeFileInCommand(char * commandName, char * argv[], char * filename) {}
```

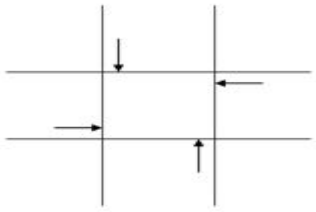



```
int executeFileOutCommand(char * commandName, char * argv[], char * filename) {}
```



Les semaphores:

tout d’abord on a créer une mini bibliothèque tubesem.h qui contient les déclaration des fonction necessire utiliser ,Initsem, P, V, attente et message



Exclusion mutuelle dans un carrefour

Exécution et captures d'écrans:

Comment Executer :

Si vous voulez exécuter le shell pour la 1ere fois il suffit de tapez dans votre terminal :

#make main

Si non vous tapez dans le terminal ./shell pour executer le shell

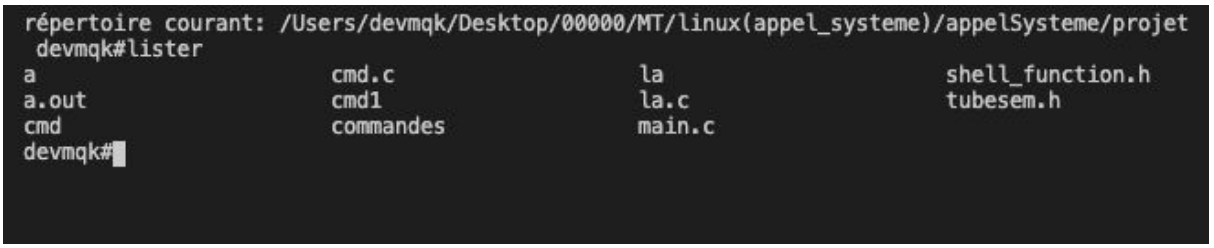
Pour vérifier les commandes valables : vous pouvez tapez menu pour afficher tous les commandes de notre shell

```
void menu(){
   printf("------------------------------Commandes valable-----------------------------------\n\n");
              // les options valables:
}
```

Menu:

```
------------------------Commandes valable-----------------------------

                            Shell Creer par Achraf Khallouk

lister :   une commande qui liste les fichier et les dossiers (toutes les options de ls valabe)
fsupprimer:   une commande pour supprimer les fichiers (tout les option de rm sont valable)
copier :   une commande pour copier les fichiers utilasant le fichier comme 1er argument et le dossier cible comme 2eme (tous les options de cp sont valable)
deplacer : une commande pour couper les fichiers utilasant le fichier comme 1er argument et le dossier cible comme 2eme (tous les options de mv sont valable)
courant:   une commande pour afficher le dossier courant
dcreer:   une commande pour creer les dossier (tous les option de mkdir sont valabe)
dcreer:   une commande pour creer les fichiers (tous les option de touch sont valabe)
edit:   une commande pour edit les fichiers(tous les option de vi sont valable)
dsupprimer:   une commande pour supprimer les dossiers (tout les option de rmdir sont valable)
afficher:   une commande pour afficher le contenu d'un fichier
cd:  pour changer les directories
clear: pour effacer l'ecran

**pour acceder au github repository click CTRL-Z**

**********pour plus d'info tapez : help NomCommande**********
```

lister: est une commande qui permet de lister le contenu d'un répertoire, avec tous les options de la commandes ls valables comme -l -a …



fsupprimer: est une commande permettant de supprimer des fichiers . avec tous les options de la commandes rm valables

copier: la commande permet de faire la copie d'un ou plusieurs fichier mais aussi d'un ou plusieurs répertoire **à** la fois et en ligne de commande

deplacer: permettant de déplacer des fichiers et des répertoires. Il permet également de renommer un fichier ou un répertoire

courant: Elle permet d'afficher le chemin d'accès vers le répertoire où se situe l'utilisateur qui a entré la commande.

dcreer: est une commande permettant de créer des répertoires

fcreer: est une commande permettant de modifier le timestamp de dernier accès et de dernière modification d'un fichier. Cette commande permet également de créer un fichier vide.

```
devmqk#fcreer fichier
devmqk#lister
\a                  cmd.c               fichier             newLa
a.out               cmd1                la.c                shell_function.h
cmd                 dossier             main.c              tubesem.h
devmqk#
```

dsupprimer: est une commande permettant de supprimer des répertoires.

```
devmqk#dsupprimer dossier
devmqk#lister
a                   cmd.c               la.c                shell_function.h
a.out               cmd1                main.c              tubesem.h
cmd                 fichier             newLa
devmqk#
```

afficher :est une commande permettant de concaténer des fichiers ainsi **que** d'afficher leur contenu sur la sortie standard

edit: ouvrir l'éditeur de text vi qui est un des éditeurs de texte les plus populaires sous **Linux** (avec Emacs et pico) malgré son ergonomie très limitée. En effet, **Vi** (prononcez Vihaille) est un éditeur entièrement en mode texte, **ce** qui signifie **que** chacune des actions se fait **à** l'aide de commandes texte
Commande :

```
répertoire courant: /Users/devmqk/Desktop/00000/MT/linux(appel_systeme)/appelSysteme/projet
 devmqk#edit main.c
```

Résultat:

```
#include "shell_function.h"
#include "tubesem.h"



Semaphore S1;

void getSignal(int sig)
{

    printf("\nMerci pour utiliser mon shell\n");
    sleep(2);
    exit(0);
}


void init()
{
  // print welcome message

        printf("\nMini Projet Creation d'une shell\n");
    sleep(2);
    clear();
}

void cmd_execute(char **args)
{
  if (fork() == 0)
    {
        P(S1);
```

cd: a navigation d'un répertoire à un autre s'effectue avec la commande *cd* succédée du nom du répertoire. et vous affiche le dossier courant

```
devmqk#cd a
/Users/devmqk/Desktop/00000/MT/linux(appel_systeme)/appelSysteme/projet/a
devmqk#
```

help: travail comme man , il suffit de taper: help nomcommand pour afficher l'aide

```
devmqk#help edit
```

Quitter: envoi un signal SIGINT, pour quitter le shell ,:

```
devmqk#^C
Merci pour utiliser mon shell
(base) MBP-de-DevMQK:projet devmqk$
```

Vous pouvez aussi comme le menu affiche quitter avec SIGSTOP (CTRL-Z) pour aller vers github repository :

```
 devmqk#^Z
ouvrir dans 3 secondes
Merci pour utiliser mon shell
**************Khallouk Achraf***************
(base) MBP-de-DevMQK:projet devmqk$
```

Conclusion:

Nous arrivons au terme de cette partie de la création d'un mini système,mini shell . À ce stade nous avons un shell opérationnel, bien que incomplet par rapport à bash.

Des points sont à revoir dans ce code, notamment la gestion des pipes limités à deux commandes. Aussi que les arguments acceptés par le ligne ne dépasse pas 3 arguments.