1-SN

Systèmes d'exploitation centralisés

« Minishell »

Saïd AIT FASKA

2020-2021

Table des matières

1 Introduction

L'objet de ce projet est de construire un minishell similaire a celui utilisé de shell bash qui est étudié dans la matière systèmes d'exploitation centralisées. Grâce aux notions vues pendant le cours, TDs et TPs .

2 Architecture du projet

Dans cette partie de l'architecture du projet minishell je me suis focalise sur une architecture le plus simple possible pour avoir une meilleure simplicité : j'ai mis en place des fonctions avant le début du " main " du programme , dans le but de les utiliser en les appelant dans le coeur du programme , ci dessus une figure de quelques fonctions implantées(utiles pour la Question 6) .

```
/*get indice pid */
int get_indice(pid_t pid){
        for(int i=0 ; i<= list_jobs.Nombre_fils ; i++){
                if(list_jobs.list[i].job_pid == pid){
                        return i;
                }
        }
        return 0;       // pid pas disponible -> erreur
}

/*get pid from jobs_list */
int get_pid(int identifiant){
        for(int i=0 ; i< list_jobs.Nombre_fils ; i++){
                if(list_jobs.list[i].job_ident == identifiant){
                        return list_jobs.list[i].job_pid;
                }

        }
        return 0;
}

/* Etat du pid dans la liste */
void Etat_pid(pid_t pid){
        for(int i=0 ; i<= list_jobs.Nombre_fils ; i++){
                if(list_jobs.list[i].job_pid == pid){

                        return list_jobs.list[i].job_etat;
                }
                //job_curseur = job_curseur[i].suivant;
        }
}
```

Figure 1 – Quelques fonctions implantées

Ces fonctions permettent d'avoir l'indice du pid , le pid lui même ainsi que son etat dans la liste des processus

⋆Remarque : J'ai utilisé une liste pour enregistrer les processus , en systèmes d'exploitation centralisées on utilise souvent (préférable) des tableaux qui permet une bonne optimisation de temps de parcourt, ainsi de recherche mais cette stratégie n'a pas été applique ici

Pour les traitants j'ai utilisé trois handlers (CHLD,SIGINT, SIGSTOP) ci dessus une figure de leur implantation :

```c
/* handler_chld */
void handler_chld(int signal_num) {
     int fils_termine, wstatus ;
     //printf("\nJ'ai reçu le signal %d\n",  signal_num) ;
     do{
        fils_termine = (int) waitpid(-1, &wstatus, WNOHANG | WUNTRACED | WCONTINUED);
        if ((fils_termine == -1) && (errno != ECHILD)){
              perror("\n waitpid" );
         }
           else if (fils_termine > 0) {
                int k = get_indice(fils_termine);
                if(k != -1){
                        if WIFEXITED(wstatus){
                                supprimer_pid(fils_termine);
                        }
                }
        }
        }while(fils_termine > 0);
                return;
  }

/*  handler stop */
void handler_SIGTSTP(int sig){
        kill(pid_foreground,SIGSTOP);
        supprimer_pid(pid_foreground);
}

/*Traitant handler Sigint */     //on tue le pid qui est en avant plan //
void handler_SIGINT(int sig) {
        kill(SIGKILL,pid_foreground);
        supprimer_pid(pid_foreground);
}
```

Figure 2 – Handler signaux

3 Programme principale

3.1 Une première vue :

Le programme principale qui commence dès la ligne 228 avec le *main* j'ai défini les variables qui sont utilisés dans ce projet , ainsi que l'association des signaux aux traitants (handler), suit de la boucle infinie *while* Ci dessus un figure de cette partie

```c
int main() {
        int codeTerm,pid,retour1,retour2,retour3,retour4,retour5,retour6;
        int wstatus,wstatus1;
        list_jobs.Nombre_fils = 0;
        int p1[2],p2[2];
        int fils;
        /* Signaux */

        signal(SIGCHLD, handler_chld);
        signal(SIGSTOP,&handler_SIGTSTP);
        signal(SIGINT,&handler_SIGINT);

        /* signal mask */
        sigprocmask(SIG_BLOCK, &handler_SIGTSTP, NULL);
        sigprocmask(SIG_BLOCK, &handler_SIGINT, NULL);
        while(1){
```

Figure 3 – programme principale

Voir partie **Réponse aux questions** pour Le masquage des signaux SIGTSTP et SIGINT.

En executant le code du programme on voit apparaitre dans le terminale l'affichage implanté pour le minishell : ci dessus une figure de cet affichage en modifiant le couleur de prompt (couleur vert) :

Figure 4 – Vue miniShell

Pour verifier le bon functionement du code on peut effectuer des petits tests , pour cela voici ci dessous les differents test effectues pour cette premiere phase du projet :

→ **Commande ls :**

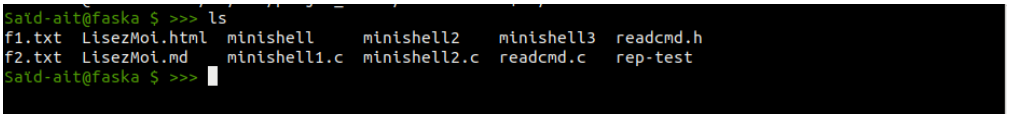

Figure 5 – test 1

→ **Commande $ls - l$:**

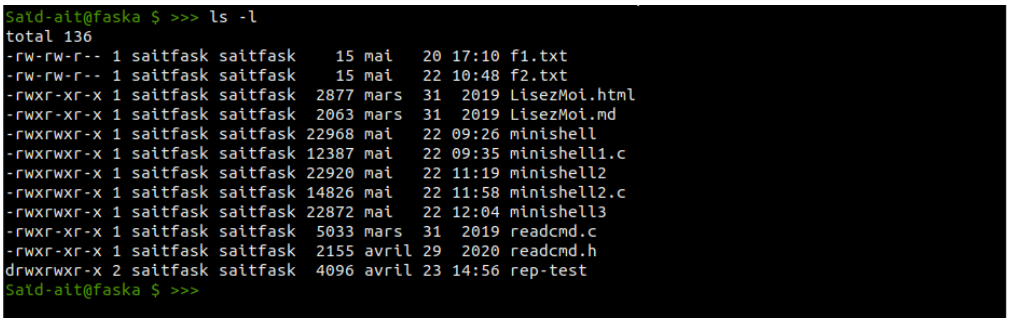

Figure 6 – test 2

Des tests supplémentaires seront dans la dernière de ce rapport , partie **Tests supplémentaires**.

4 Reponse aux questions

⋆ **Question 2 :**

Pour cette question , j'ai mis en ouvre une boucle "while" infinie , et pour pouvoir simuler une session , j'ai trité les differents cas selon la ligne de commande (ligne commande vide, un appui sur

la touche entree ... etc) Voir code fichier minishell.c

⋆ **Question 6 :**

Pour cette question j'ai implanté les fonctions :

– **jobs** : cette commande permet d'avoir une liste des processus ansi que leur etat (suspendu, actif,FG ou BG). Ci dessous une implantation de cette commande :

```
/* afficher le pid , son etat */
void afficher_pid(job_t job_1){
     printf("\n");
     printf(" ------------------------------------------------------ \n");
    if (job_1.job_etat == FG){
        printf(" -- %d   de  pid: %d    est FG    \n" ,job_1.job_ident ,job_1.job_pid);
        fflush(stdout);
    } else if (job_1.job_etat == BG){
        printf(" -- %d   de  pid: %d   est BG      \n" ,job_1.job_ident ,job_1.job_pid);
        fflush(stdout);
    } else {
        printf(" -- %d   de  pid: %d   est  Suspendu     \n" ,job_1.job_ident ,job_1.job_pid);
        fflush(stdout);
    }
        printf(" ------------------------------------------------------ \n");
}

/* Afficher la liste de jobs */
void afficher_jobs(){
        //printf("JE SUIS DANS AFFICHER JOBS \n");
        for(int i=0 ; i<= list_jobs.Nombre_fils ; i++){    /* all list jobs */
                afficher_pid(list_jobs.list[i]);
        }
}

```

Figure 7 – implantation commande jobs

On affiche donc une liste en lignes des pid ansi que leur état , le test de cette commande est dans la figure suivante :

```
Saïd-ait@faska $ >>> jobs

 ------------------------------------------------------
 -- 1   de  pid: 15064   est BG
 ------------------------------------------------------

 ------------------------------------------------------
 -- 3   de  pid: 15073   est  Suspendu
 ------------------------------------------------------

 ------------------------------------------------------
 -- 5   de  pid: 16648   est  Suspendu
 ------------------------------------------------------

 ------------------------------------------------------
 -- 6   de  pid: 16648    est FG
 ------------------------------------------------------
Saïd-ait@faska $ >>>
```

Figure 8 – test jobs

– **stop** : commande qui permet de suspendre un processus voici une figure de son implentation :

```
177 /* Stop */
178 void stop(char ** cmd){
179         if (cmd[1] == 0 ) {
180                 printf("Stoped succesfuly 1/1 \n");
181         }else{
182                         // la fct atoi permet de convertir chaine en entier codage ASCII //
183              int pid = atoi(cmd[1]);
184              kill(pid,SIGSTOP);
185              list_jobs.list[get_indice(pid)].job_etat = suspendu;
186              printf(" process stoped  2/2 \n");
187         }
188 }
```

Figure 9 – implantation fonction stop.

Pour vérifier le bon fonction de cette commande on l'appelle dans le minishell avec un identifiant tiré de la liste jobs Voici le test correspondant :

```
Saïd-ait@faska $ >>> jobs

 ----------------------------------------------------
 -- 1   de  pid: 19339   est BG
 ----------------------------------------------------

 ----------------------------------------------------
 -- 2   de  pid: 19339    est FG
 ----------------------------------------------------
Saïd-ait@faska $ >>> stop 1
 process stoped  2/2
Saïd-ait@faska $ >>> jobs

 ----------------------------------------------------
 -- 1   de  pid: 19339   est  Suspendu
 ----------------------------------------------------

 ----------------------------------------------------
 -- 2   de  pid: 19339    est FG
 ----------------------------------------------------
Saïd-ait@faska $ >>>
```

Figure 10 – test fonction stop

Pour ce test j'ai mis un l'exécution d'un processus en arrière-plan (BG) ensuite j'ai applique la commande "stop" pour suspendre son exécution , comme vu dans la figure ci-dessus
– **bg** : cette commande permet de reprendre en arrier plan (background) un processus . Pour l'implantation de cette fonction :

```
160 void bg_pid(char**cmd){
161         if (cmd[1] == 0) {
162
163                 printf("il faut identifiant du pid execute en BG .\n");
164
165          } else if (get_pid((cmd[1])) < 0){
166
167                 printf("id error \n");
168
169         } else {
170                 printf("je suis 2 ");
171                 int pid = get_pid(cmd[1]);
172                 kill(pid,SIGCONT);
173                 list_jobs.list[get_indice(pid)].job_etat = BG;
174         }
175 }
176
```

Figure 11 – Implantation BG

Pour le test de la commande *bg* est donné en figure suivante :

```
 ----------------------------------------------------
 -- 5   de  pid: 20572   est  Suspendu
 ----------------------------------------------------

 ----------------------------------------------------
 -- 6   de  pid: 20572    est FG
 ----------------------------------------------------
Saïd-ait@faska $ >>> BG 5
Saïd-ait@faska $ >>> jobs

 ----------------------------------------------------
 -- 1   de  pid: 19339   est  Suspendu
 ----------------------------------------------------

 ----------------------------------------------------
 -- 3   de  pid: 20538   est  Suspendu
 ----------------------------------------------------

 ----------------------------------------------------
 -- 5   de  pid: 20572   est BG
 ----------------------------------------------------

 ----------------------------------------------------
 -- 6   de  pid: 20572    est FG
 ----------------------------------------------------
Saïd-ait@faska $ >>>
```

Figure 12 – test BG

– **fg** : Cette commande permet de ramener un processus en avant plant pour son exécution , j'ai mis une condition sur l'identifiant , on vérifie qu'il est bien saisie dans la ligne de commande , sinon on a un erreur . ci dessous l'implantation de cette commande :

```
138 void fg_pid(char** cmd){
139           if (cmd[1] == 0) {
140
141                printf("il faut identifiant du pid execute en FG .\n");
142
143           } else if (get_pid(cmd[1]) < 0){
144
145                printf("id error\n");
146
147          } else {
148                int pid = get_pid(atoi(cmd[1]));
149                int indice = get_indice(pid);
150                list_jobs.list[indice].job_etat = FG;
151                printf("Pid en FG\n");
152                kill(pid, SIGCONT);
153                list_jobs.list[get_indice(pid)].job_etat = FG;
154                //wait(&curseur);
155                //supprimer_pid(pid);
156        }
157   }
158
```

Figure 13 – Implatation FG.

Pour le teste de cette commande on obtient les résultats, suivants, en créant un liste de jobs qui contient des processus , on reprend donc un processus qui était suspendu au début en avant-plan

```
Saïd-ait@faska $ >>> jobs

 ---------------------------------------------------
 -- 1   de  pid: 14403   est  Suspendu
 ---------------------------------------------------

 ---------------------------------------------------
 -- 2   de  pid: 14403    est FG
 ---------------------------------------------------
Saïd-ait@faska $ >>> FG 1
Pid en FG
Saïd-ait@faska $ >>> jobs

 ---------------------------------------------------
 -- 1   de  pid: 14403    est FG
 ---------------------------------------------------

 ---------------------------------------------------
 -- 2   de  pid: 14403    est FG
 ---------------------------------------------------
Saïd-ait@faska $ >>>
```

Figure 14 – Test FG.

⋆ **Question 7 :**

– **ctrl-z et ctrl-c** : Le traitants de ces signaux sont donne en figure en tout début de rapport (handler signaux) ,Comme dit dans l'énonce il faut pouvoir suspendre un processus en fg (CTRL-Z) ainsi que l'envoie d'un signal SIGINT (CTRL-C) qui permet la terminaison du processus en avant-plant sans sortir de la boucle :

Le masquage des signaux permet d'eviter de sortir de la boucle ,ci dessous une figure de l'association des signaux aux traitants (handlers) :

```
234         /* Signaux */
235
236         signal(SIGCHLD, handler_chld);
237         signal(SIGSTOP,&handler_SIGTSTP);
238         signal(SIGINT,&handler_SIGINT);
239
240         /* signal mask */
241         sigprocmask(SIG_BLOCK, &handler_SIGTSTP, NULL);
242         sigprocmask(SIG_BLOCK, &handler_SIGINT, NULL);
```

Figure 15 – CTRL-Z , CTRL-C.

Pour pouvoir tester cette implantation on lance dans le terminal du minishell une attente de 10s :" **sleep 10**", et on frappe de suite (avant la terminaison des 10s CTRL-C pour pouvoir l'effet de l'implantation : Ci dessus une figure de ce test (vous pouvez exécuter vous même et voir si vous le souhaitez) :

```
Saïd-ait@faska $ >>> sleep 10
^CSaïd-ait@faska $ >>>
```

Figure 16 – test signaux

→ **Remarque :**

Dans cette partie des signaux (Q7) j'ai utilisé l'indication de "waitpid" de l'énonce , ainsi que le handler chld fourni sur Moodle en l'adaptant a mon code

5 Tests Supplémentaires

Cette partie permet de verifier la robustesse du projet MiniShell ainsi que les tests pour les questions :

Q8 (Redirections) et Q9 , Q10 concernant les Tubes :

▷ **Redirections :**

Pour traiter ces questions , j'ai utilisé le fichier "readcmd" pour utiliser *in* et *out* (voir implantation sur le fichier source minishell) pour le test , j'ai crée un fichier f1.txt qui contient la phrase suivante : « Bonjour ça va » et en exécutant la commande " cat < f1.txt > f2.txt " sur le minishell on voit

l'affichage suivant :

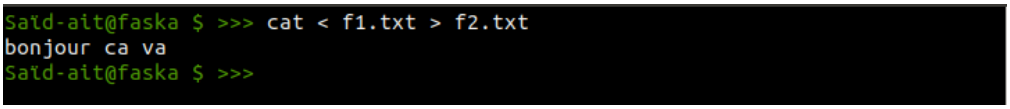

Figure 17 – test redirections

On peut voir dans le fichier f2.txt que la phrase qui était au début dans f1.txt a été écrite dans le fichier f2.txt qui est vide au départ : donc la commande marche bien

▷ **Tubes :**

Pour les tests de cette partie j'ai lance la commande " ls | wc -l" dans le terminal de la machine (pas de minishell) et j'obtiens de resultat :

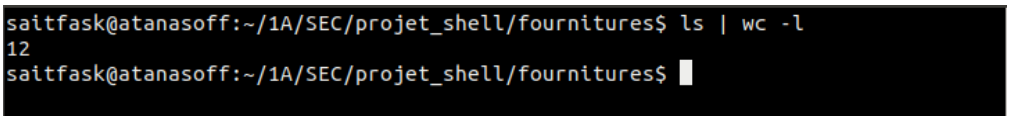

Figure 18 – test tubes

et celui obtenu d'apres le MiniShell est le suivnat :

Figure 19 – test redirections

On obtient donc le même résultat , en effet cette commande permet de compter le nombre de fichiers dans un répertoire , dans mon fichier il y a bien 12 fichiers
Pour les pipelines on peut exécuter la commande donnée en énonce en l'adaptant aux fichiers disponibles dans le répertoires , ci dessous le résultat de l'exécution

6 Conclusion

Ce projet du MiniShell m'a permis d'approfondir les différents aspects vus en cours ainsi qu'en TDs et TPs. Les principales difficultés viennent lors de la Question 6 quand il faut implanter les signaux et faire en sorte que respectent les consignes ; cette question m'a personnellement pose beaucoup des soucis car il fallait faire attention a que le signaux envoyés n'interrompent le programme es sortir de la boucle ce qui engendre la sortie du minishell.