

Projet 2

Mini Shell en C

Linux



Encadré Par : Mohamed BAKHOUYA
Abdelhak KHARBOUCH

Réaliser par : EL HANAFI Maha

Objectif :

Il s'agit de réaliser un interprète pour un langage de commande simplifié. L'objectif de ce projet :

- comprendre la structure d'un shell
- apprendre à utiliser quelques appels systèmes importants, typiquement ceux qui concernent la gestion des processus, les pipes et la redéfinition des fichiers standards d'entrée et de sortie, signaux.

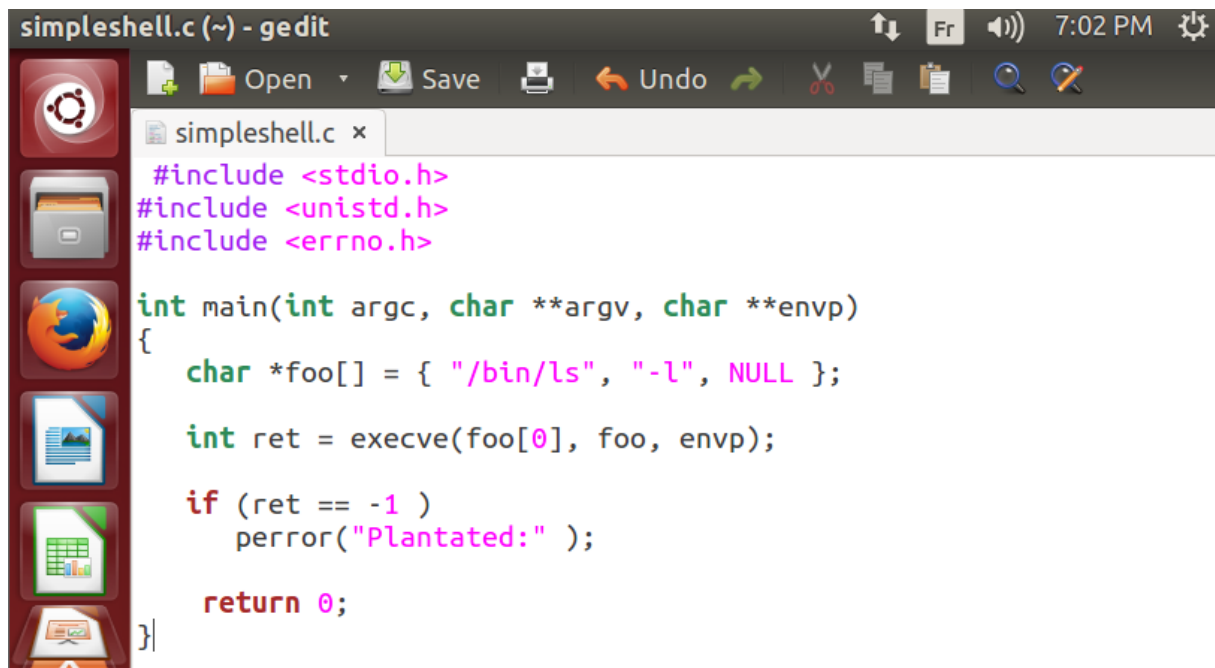
Introduction

Le projet qu'il s'agit d'écrire un programme **msh** pour mini shell. C'est une implémentation basique d'un interpréteur de commandes (ou Shell linux).

Implémentations :

- Un simple exemple sans fork() et avec la fonction exec.
- Exécuter les commandes basiques (exemple : ls, cd, ls -l, cat ...). On prend en considération les lignes de commandes et les arguments.
- Exécuter les fonction execl, execlp
- Quitter lorsque l'utilisateur tape la commande exit ou tape Ctrl+D et prendre en compte Ctrl+C
- On utilise les signaux pour installer une fonction gestionnaire (cf. trap du shell).

A-Simple minishell en C :



```
simpleshell.c (~) - gedit
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int main(int argc, char **argv, char **envp)
{
    char *foo[] = { "/bin/ls", "-l", NULL };

    int ret = execve(foo[0], foo, envp);

    if (ret == -1 )
        perror("Plantated:");

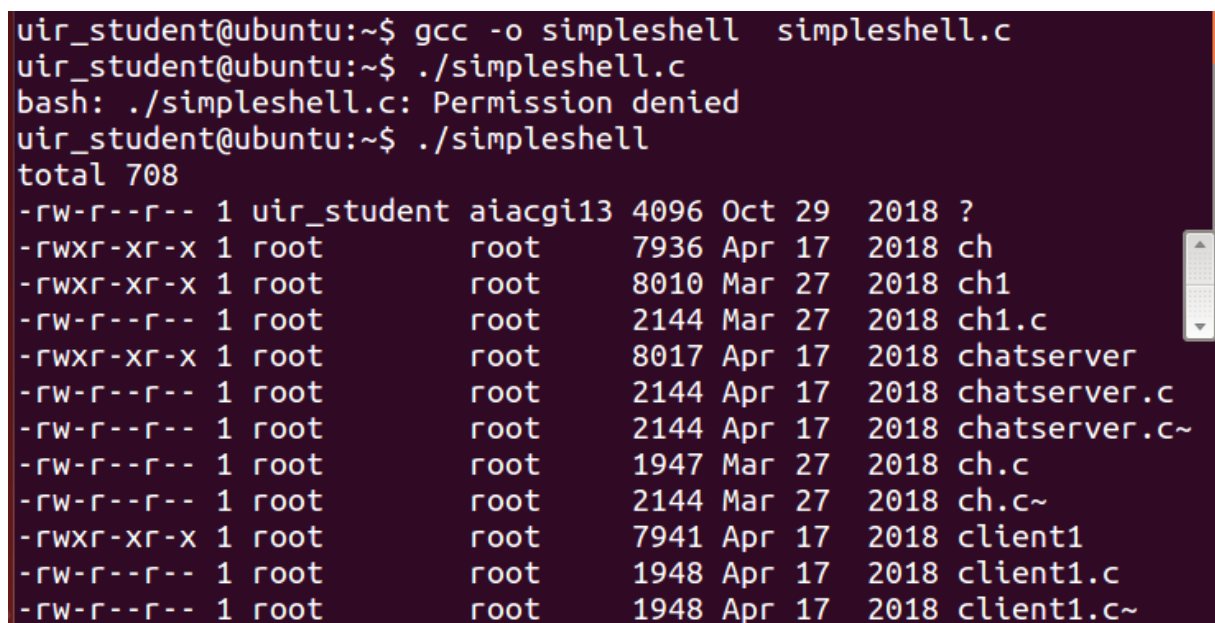
    return 0;
}
```

Dans le premier instant ce qui rend les choses plus explicites (le programme tente d'exécuter « /bin /»

En somme le premier argument c'est l'objet à exécuter.

Le second argument « argv » contient la liste des arguments à lui passer, avec argv[0] == le nom de l'objet exécuté (ou assimilé), comme ce que tu obtiens dans argv[0] en utilisant le prototype de main() : « int main (int argc, char **argv)

Et cela fonctionne :



```
uir_student@ubuntu:~$ gcc -o simpleshell simpleshell.c
uir_student@ubuntu:~$ ./simpleshell.c
bash: ./simpleshell.c: Permission denied
uir_student@ubuntu:~$ ./simpleshell
total 708
-rw-r--r-- 1 uir_student aiacgi13 4096 Oct 29 2018 ?
-rwxr-xr-x 1 root root 7936 Apr 17 2018 ch
-rwxr-xr-x 1 root root 8010 Mar 27 2018 ch1
-rw-r--r-- 1 root root 2144 Mar 27 2018 ch1.c
-rwxr-xr-x 1 root root 8017 Apr 17 2018 chatserver
-rw-r--r-- 1 root root 2144 Apr 17 2018 chatserver.c
-rw-r--r-- 1 root root 2144 Apr 17 2018 chatserver.c~
-rw-r--r-- 1 root root 1947 Mar 27 2018 ch.c
-rw-r--r-- 1 root root 2144 Mar 27 2018 ch.c~
-rwxr-xr-x 1 root root 7941 Apr 17 2018 client1
-rw-r--r-- 1 root root 1948 Apr 17 2018 client1.c
-rw-r--r-- 1 root root 1948 Apr 17 2018 client1.c~
```

B-Minishell en C

Partie 1 :

On lit une ligne, crée un fils, qui exécute la commande. Le seul problème est bien nettoyé la ligne des espaces et fin de ligne éventuellement présents

```
#define _GNU_SOURCE
```

```
// GNU_SOURCE pour avoir getline qui permet de borner la lecture de ligne
```

```
// pas comme scanf
```

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

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

```
#include <sys/wait.h>
```

```
#include <signal.h>
```

```
#include <string.h>
```

```
#include <fcntl.h>
```

```
#include <ctype.h>
```

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

```
inline void
```

```
erreur(const char *s) { perror(s); exit(1); }
```

```
// supprime les blancs qui traînent en queue et retourne la Nelle longueur
```

```
int nettoie(char * ligne) {
```

```
    int lg = strlen(ligne);
```

```
    while (isspace(ligne[lg - 1]))
```

```
        lg--;
```

```
    ligne[lg] = '\0';
```

```
    return(lg);
```

```
}
```

```
int main() {  
    char * ps1 = getenv("PS1"); // Le prompt du Bourne (Again) Shell  
    char* ligne = 0;  
    size_t MAXLIGNE = 0;  
    if (ps1 == NULL)  
        ps1 = "$ ";  
    while(1) {  
        int lgligne;  
        int pid;  
        printf("%s",ps1);  
        if (getline(&ligne, &MAXLIGNE, stdin) == -1)  
            break;          // C'est fini  
        if ((lgligne = nettoie(ligne)) == 0)  
            continue;       // juste une ligne vide, on continue  
        switch (pid = fork()) {  
            case -1:  
                erreur("fork");  
            case 0:  
                execl(ligne, ligne, NULL);  
                erreur(ligne);  
            default:  
                // on se bloque jusqu'à la fin de la commande  
                while (wait(0) != pid);  
        } } return 0;}
```

Et cela fonctionne :

```
uir_student@ubuntu:~$ gedit shellpartie1.c
uir_student@ubuntu:~$ gcc -o shellpartie1 shellpartie1.c
uir_student@ubuntu:~$ ./shellpartie1
$ ls
ls: No such file or directory
$ /bin/echo bonjour
/bin/echo bonjour: No such file or directory
$
$
$ ^C
```

Mais n'exploite pas Path ni les arguments sur la ligne de commande

Pour exploiter PATH, il suffit de mettre `execlp` au lieu de `execl`.

Partie2 : la deuxième version du code

Permettre de lancer des processus en arrière-plan. Il suffit de ne pas attendre le processus fils. Il faut aussi ne pas permettre au processus en arrière-plan de lire l'entrée standard ; on la détourne de puis `/dev/null` pour cela.

```
*partie2.c x
#define _GNU_SOURCE
// GNU_SOURCE pour avoir getline qui permet de borner la lecture
// de ligne
// pas comme scanf
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>
#include <stdlib.h>

typedef enum {
    false,
    true
} bool;

inline void
erreur(const char *s) { perror(s); exit(1); }
```

```
#define _GNU_SOURCE
```

```
// GNU_SOURCE pour avoir getline qui permet de borner la lecture de ligne
// pas comme scanf

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>
#include <stdlib.h>

typedef enum {
    false,
    true
} bool;

inline void
erreur(const char *s) { perror(s); exit(1); }

// supprime les blancs qui traînent en queue et retourne la nouvelle longueur
int nettoie(char * ligne) {
    int lg = strlen(ligne);
    while (isspace(ligne[lg - 1]))
        lg--;
    ligne[lg] = '\0';
    return(lg);
}

// Fait un tableau d'arguments avec les mots de la ligne
Void mkargs(char *args[], int MAXARGS, char * ligne) {
    static char IFS[] = " \t";
    int i = 0;
    args[i++] = strtok(ligne, IFS);
    while ((args[i++] = strtok(0, IFS)))
```

```

    if (i > MAXARGS) {
        fprintf(stderr, "plus de %d arguments, le reste n'est pas pris en compte\n",
MAXARGS);
        return;
    }
}

int
main() {
    char * ps1 = getenv("PS1"); // Le prompt du Bourne (Again) SHell
    char* ligne = 0;
    size_t MAXLIGNE = 0;
    char * args[256];
    if (ps1 == 0)
        ps1 = "$ ";
    while(1) {
        int lgligne;

        bool attendre = true; // commande synchrone en général */
        int pid;

        printf("%s",ps1);
        if (getline(&ligne, &MAXLIGNE, stdin) == -1)
            break;          // C'est fini
        if ((lgligne = nettoie(ligne)) == 0)
            continue;       // juste une ligne vide, on continue
        if (ligne[lgligne-1] == '&') {
            attendre = false; // une commande asynchrone
            ligne[lgligne - 1] = '\0';
        }

        switch (pid = fork()) {
        case -1:

```



```

    erreur("fork");
case 0:
    if ( ! attendre) { // la commande en arrière plan ne doit pas lire
        int fd = open("/dev/null", O_RDONLY); // l'entrée standard
        close(0);
        dup(fd);
    }

    mkargs(args, sizeof(args), ligne);
    execvp(args[0], args);
    erreur(args[0]);
default:
    if (attendre)    // on se bloque jusqu'à la fin de la commande
        while (wait(0) != pid);
    else
        printf("%d\n", pid); // on écrit juste le numéro du processus
    }
}return 0;
}

```

Et cela fonctionne : on tape les deux commandes : ls , ls -l

```

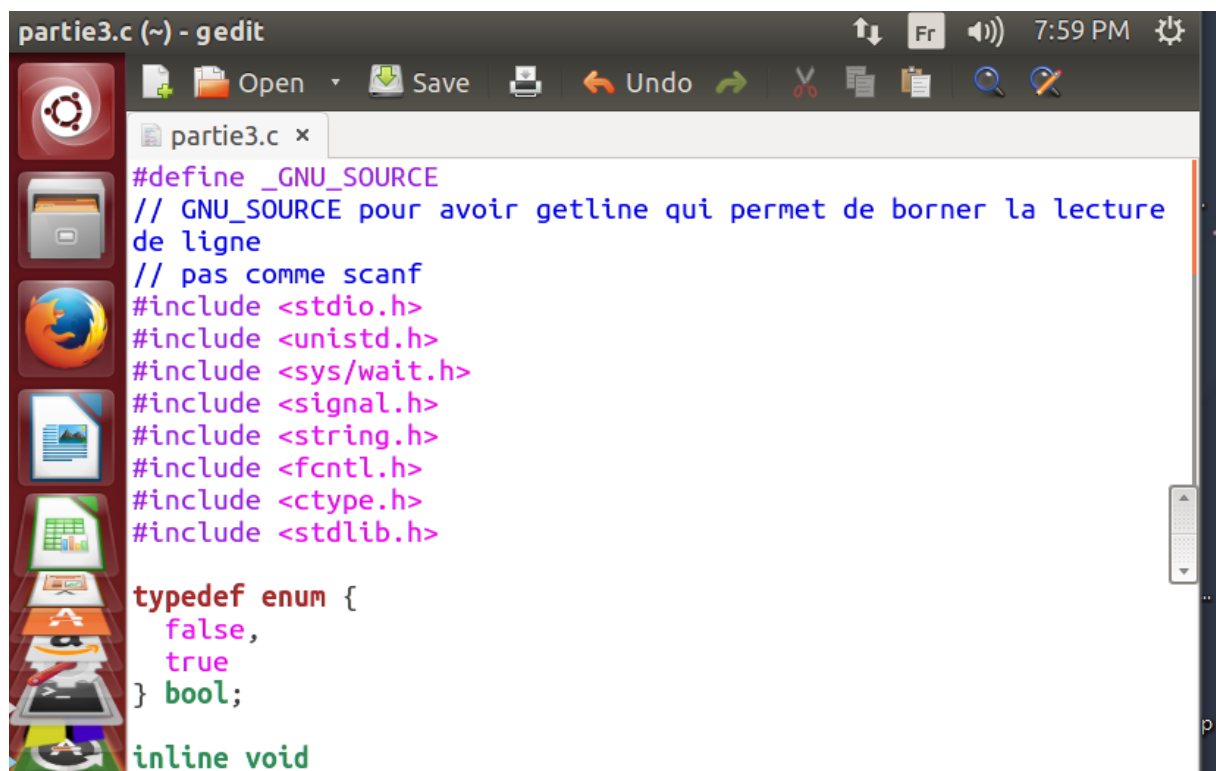
uir_student@ubuntu:~$ gcc -o partie2  partie2.c
uir_student@ubuntu:~$ ./partie2
$ ls
?          f2.sh      file4      s2.c~
ch         f2.sh~    file5      s4.c~
ch1        f3.c      file7      s5.c~
ch1.c      f3.c~    file8      sc2.sh
chatserver f3.sh     file9      sc2.sh~
chatserver.c f3.sh~  iman.c     scripts
chatserver.c~ f4.c     Music      she2~
ch.c       f4.c~    new        she4~
ch.c~      f4.sh    new1       she5~
client1    f4.sh~   new2       shell1
client1.c  f5.c     p1         shell1.c

```

```
$ ls -l
total 748
-rw-r--r-- 1 uir_student aiacgi13 4096 Oct 29 2018 ?
-rwxr-xr-x 1 root root 7936 Apr 17 2018 ch
-rwxr-xr-x 1 root root 8010 Mar 27 2018 ch1
-rw-r--r-- 1 root root 2144 Mar 27 2018 ch1.c
-rwxr-xr-x 1 root root 8017 Apr 17 2018 chatserver
-rw-r--r-- 1 root root 2144 Apr 17 2018 chatserver.c
-rw-r--r-- 1 root root 2144 Apr 17 2018 chatserver.c~
-rw-r--r-- 1 root root 1947 Mar 27 2018 ch.c
-rw-r--r-- 1 root root 2144 Mar 27 2018 ch.c~
```

Partie 3 : La dernière version du code

Cette dernière version va permettre de prendre en compte Ctrl-C . On utilise pour cela signal pour installer une fonction gestionnaire (cf. trap du shell) . La commande trap permet une gestion des signaux d'interruption et permet d'indiquer en argument la ou les commandes à exécuter lorsque le signal d'interruption spécifié se produit.



```
partie3.c (~) - gedit
#define _GNU_SOURCE
// GNU_SOURCE pour avoir getline qui permet de borner la lecture
// de ligne
// pas comme scanf
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>
#include <ctype.h>
#include <stdlib.h>

typedef enum {
    false,
    true
} bool;

inline void
```

```
#define _GNU_SOURCE
#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <signal.h>
```

```

#include <string.h>

#include <fcntl.h>

#include <ctype.h>

#include <stdlib.h>

typedef enum {
    false,
    true
} bool;

inline void
erreur(const char *s) { perror(s); exit(1); }

// supprime les blancs qui traînent en queue et retourne la nelle longueur
int nettoie(char * ligne) {
    int lg = strlen(ligne);
    while (isspace(ligne[lg - 1]))
        lg--;
    ligne[lg] = '\0';
    return(lg);
}

// Fait une tableau d'arguments avec les mots de la ligne
Void mkargs(char *args[], int MAXARGS, char * ligne) {
    static char IFS[] = " \t";
    int i = 0;
    args[i++] = strtok(ligne, IFS);
    while ((args[i++] = strtok(0, IFS)))
        if (i > MAXARGS) {
            fprintf(stderr, "plus de %d arguments, le reste n'est pas pris en compte\n", MAXARGS);
            return;
        }
}

#include <setjmp.h>

```

```

jmp_buf env;          // le "jump buffer" du début de lecture

// Fonction attachée à SIGINT et SIGQUIT

void nvligne(int sig) {
    printf("Interruption %d\n", sig);
    longjmp(env, 1);    // pour forcer la reprise de la lecture
}

Int main() {
    char * ps1 = getenv("PS1"); // Le prompt du Bourne (Again) SHell
    char* ligne = 0;
    size_t MAXLIGNE = 0;
    char * args[256];
    signal(SIGTERM, SIG_IGN);
    if (ps1 == 0)
        ps1 = "$ ";
    setjmp(env);        // C'est ici que l'on revient après une interruption
    signal(SIGINT, nvligne);
    signal(SIGQUIT, nvligne);
    while(1) {
        int lgligne;

        bool attendre = true; // commande synchrone en général */
        int pid;
        printf("%s",ps1);
        if (getline(&ligne, &MAXLIGNE, stdin) == -1)
            break;        // C'est fini
        if ((lgligne = nettoie(ligne)) == 0)
            continue;     // juste une ligne vide, on continue
        if (ligne[lgligne-1] == '&') {
            attendre = false; // une commande asynchrone
            ligne[lgligne - 1] = '\0';
        }
    }
}

```

```

}

switch (pid = fork()) {

case -1:

    erreur("fork");

case 0:

    // on doit pouvoir interrompre la commande

    signal(SIGQUIT, SIG_DFL);

    signal(SIGINT, SIG_DFL);

    signal(SIGTERM, SIG_DFL);

    if(!attendre) {

        // la commande en arrière plan ne doit pas lire l'entrée standard

        int fd = open("/dev/null", O_RDONLY);

        close(0);

        dup(fd);

    }

    mkargs(args, sizeof(args), ligne);

    execvp(args[0], args);

    erreur(args[0]);

default:

    if (attendre) { // on se bloque jusqu'à la fin de la commande

        int w;

        while((w = wait(0)) != pid && w != -1);

    }

    else // on écrit juste le numéro du processus

        printf("%d\n", pid);

}

}return 0;}

```

Et cela fonctionne : on tape quelques commandes et Ctrl+C :

```
uir_student@ubuntu:~$ gcc -o partie3 partie3.c
uir_student@ubuntu:~$ ./partie3
$ ls
?          f2.sh~      file7       s1.c~
ch         f3.c        file8       s2.c~
ch1        f3.c~       file9       s4.c~
ch1.c      f3.sh       iman.c      s5.c~
chatserver f3.sh~      Music       sc2.sh
chatserver.c f4.c       new         sc2.sh~
chatserver.c~ f4.c~      new1        scripts

$ ls -l
total 768
-rw-r--r-- 1 uir_student aiacgi13 4096 Oct 29 2018 ?
-rwxr-xr-x 1 root        root     7936 Apr 17 2018 ch
-rwxr-xr-x 1 root        root     8010 Mar 27 2018 ch1
-rw-r--r-- 1 root        root     2144 Mar 27 2018 ch1.c
-rwxr-xr-x 1 root        root     8017 Apr 17 2018 chatserver
-rw-r--r-- 1 root        root     2144 Apr 17 2018 chatserver.c
```

Si on tape Ctrl+C , il donne la main à l'utilisateur de tester une autre commande

```
istrib
$ ^CInterruption 2
$ ls
?          f2.sh~      file7       s1.c~
ch         f3.c        file8       s2.c~
ch1        f3.c~       file9       s4.c~
ch1.c      f3.sh       iman.c      s5.c~
chatserver f3.sh~      Music       sc2.sh
chatserver.c f4.c       new         sc2.sh~
chatserver.c~ f4.c~      new1        scripts
ch.c       f4.c~      new2        ch2

$ ^\Interruption 3
$ cd Desktop
cd: No such file or directory
$ cat
echo
echo
```

Sources :

<https://madelaine.users.greyc.fr/l3/systeme/doc2007/TPminish.html>

<https://github.com/lecotf/MiniShell/tree/master/srcs>

<https://github.com/racheliver/mini-shell--for-youtubechannel-users-/blob/master/ex1.c>