Root (https://segfault42.github.io)

SegFault42 - Most recent posts

Ecriture d'un shell

Published 10-12-2018

(https://twitter.com/share)

Un shell est la couche le plus haut niveau du système Unix.

Pour faire simple, un shell est un programme qui prend en input une commande, la parse et l'exécute.

Nous allons diviser le travaillons en plusieurs parties :

- Récupérer en boucle l'entrée de l'utilisateur
- Parser l'entrée utilisateur
- Executer la commande
- Coder les builtins
- Gestion de l'environnement

Tout au long de ce tutorial, la compilation se fera comme suit :

```
clang -Weverything minishell.c -o minishell
```

1) Boucle principale

On commence par faire une boucle dans laquelle on lit STDIN (la commande de l'user)

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

int     main()
{
        char    *buffer = NULL;
        size_t  buf_size = 2048;

        // alloc buffer qui stockera la commande entree par l'user
        buffer = (char *)calloc(sizeof(char), buf_size);
        if (buffer == NULL) {
                perror("Malloc failure");
                return (EXIT_FAILURE);
        }

        // ecriture d'un prompt
        write(1, "$> ", 3);

        // lecture de STDIN en boucle
        while (getline(&buffer, &buf_size, stdin) > 0) {
                printf("cmd = %s\n", buffer);
                write(1, "$> ", 3);
        }

        printf("Bye \n");
        free(buffer);
}
```

Nous avons un programme qui affiche un prompt et qui stocke l’entrée de l’utilisateur en boucle. On envoie EOF (Ctrl+D au shell pour quitter)

Output :

```
$> ls -la
cmd = ls -la

$> cd ~/
cmd = cd ~/

$> pwd
cmd = pwd

$> Bye
```

Il faut maintenant faire une fonction qui parse la commande.

2) Parsing

Prenons pour exemple la commande :

```
ls -la /
```

Nous avons le nom du binaire (ls) et ses arguments.

la commande pourrait aussi être :

```
$>  ls       -la      /
```

Nous allons écrire une fonction qui va stocker notre commande (sans les espaces) dans un char ** Ce qui donnera :

```
[ls][-la][/]
```

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

static char     **split(char *raw_cmd, char *limit)
{
        char    *ptr = NULL;
        char    **cmd = NULL;
        size_t  idx = 0;

        // split sur les espaces
        ptr = strtok(raw_cmd, limit);

        while (ptr) {
                cmd = (char **)realloc(cmd, ((idx + 1) * sizeof(char *)));
                cmd[idx] = strdup(ptr);
                ptr = strtok(NULL, limit);
                ++idx;
        }
        // On alloue un element qu'on met a NULL a la fin du tableau
        cmd = (char **)realloc(cmd, ((idx + 1) * sizeof(char *)));
        cmd[idx] = NULL;
        return (cmd);
}

static void     free_array(char **array)
{
        for (int i = 0; array[i]; i++) {
                free(array[i]);
                array[i] = NULL;
        }
        free(array);
        array = NULL;
}
```

Comme on alloue dynamiquement notre char **, on fait une fonction (free_array)

qui va libérer notre allocation.

Nous sommes maintenant prêt à exécuter notre commande avec execve

3) Exécution

Pour exécuter notre commande nous allons utiliser le syscall execve.

Nous devons utiliser le syscall fork pour crée un nouveau processus et lancer notre commande dans ce dernier.

Ce qui donne ça :

```
static void     exec_cmd(char **cmd)
{
        pid_t   pid = 0;
        int             status = 0;

        // On fork
        pid = fork();
        if (pid == -1)
                perror("fork");
        // Si le fork a reussit, le processus pere attend l'enfant (process fo
        else if (pid > 0) {
                // On block le processus parent jusqu'a ce que l'enfant termine
                // on kill le processus enfant
                waitpid(pid, &status, 0);
                kill(pid, SIGTERM);
        } else {
                // Le processus enfant execute la commande ou exit si execve e
                if (execve(cmd[0], cmd, NULL) == -1)
                        perror("shell");
                exit(EXIT_FAILURE);
        }
}
```

Si on compile et on exécute le code, voilà ce qu'il se passe :

```
$> ls
shell: No such file or directory
$> /bin/ls
README.md  main.c  mainn.c  minishell
$> Bye
```

En envoyant une commande simple comme ls à notre shell, execve nous renvoie -1 et perror affiché No such file or directory.

Le premier argument d'execve doit être le chemin absolu du binaire à exécuter.

Pour lancer une commande sans donner le chemin absolu, nous devons chercher ou se trouve le binaire ls, concaténer le path + le nom du binaire et enfin le passer en premier argument à execve

Pour trouver ou se trouve un programme, nous devons utiliser la variable d’environnement PATH.

Si on exécute la commande :

```
$> echo $PATH
```

Nous allons avoir un output qui ressemble à celui-là :

```
/bin:/usr/bin:/usr/local/bin
```

Il s’agit des dossiers (séparer par ‘:‘) ou notre Shell va chercher notre binaire à exécuter.

Nous devons maintenant écrire la fonction qui va concaténer notre path et le binaire.

Il faut récupérer le contenu de la variable $PATH avec la fonction getenv. Elle prend un seul paramètre qui est la variable que l’on cherche et renvoie un pointeur sur le contenu de la variable passer en paramètre.

Si notre binaire n’est dans aucun dossier, on peut avertir l’utilisateur par un Command not found, sinon on peut exécuter notre execve : D.

La fonction qui récupère le contenue de la variable $PATH et qui renvoie le chemin absolu :

```
static void     get_absolute_path(char **cmd)
{
        char    *path = strdup(getenv("PATH"));
        char    *bin = NULL;
        char    **path_split = NULL;

        if (path == NULL) // si le path est null, on cree un path
                path = strdup("/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sb

        // si cmd n'est pas le chemin absolue, on cherche le chemin absolue du
        // binaire grace a la variable d'environment PATH
        if (cmd[0][0] != '/' && strncmp(cmd[0], "./", 2) != 0) {

                // On split le path pour verifier ou ce trouve le binaire
                path_split = split(path, ":");
                free(path);
                path = NULL;

                // On boucle sur chaque dossier du path pour trouver l'emplacer
                for (int i = 0; path_split[i]; i++) {
                        // alloc len du path + '/' + len du binaire + 1 pour le
                        bin = (char *)calloc(sizeof(char), (strlen(path_split[
                        if (bin == NULL)
                                break ;

                        // On concat le path , le '/' et le nom du binaire
                        strcat(bin, path_split[i]);
                        strcat(bin, "/");
                        strcat(bin, cmd[0]);

                        // On verfie l'existence du fichier et on quitte la bou
                        // renvoi 0
                        if (access(bin, F_OK) == 0)
                                break ;

                        // Nous sommes des gens propre :D
                        free(bin);
                        bin = NULL;
                }
                free_array(path_split);

                // On remplace le binaire par le path absolue ou NULL si le bi
                // n'existe pas
                free(cmd[0]);
                cmd[0] = bin;
        } else {
                free(path);
                path = NULL;
        }
}
```

Notre main devient :

```
int     main()
{
        char    *buffer = NULL;
        size_t  buf_size = 2048;

        // alloc buffer qui stockera la commande entree par l'user
        buffer = (char *)calloc(sizeof(char), buf_size);
        if (buffer == NULL) {
                perror("Malloc failure");
                return (EXIT_FAILURE);
        }

        // ecriture d'un prompt
        write(1, "$> ", 3);

        // lecture de STDIN en boucle
        while (getline(&buffer, &buf_size, stdin) > 0) {
                cmd = split(buffer, " \n\t");
                get_absolute_path(cmd);

                if (cmd[0] == NULL)
                        printf("Command not found\n");
                else
                        exec_cmd(cmd);

                write(1, "$> ", 3);
                free_array(cmd);

        }

        printf("Bye \n");
        free(buffer);
        }
```

À ce stade, notre shell exécute une commande mais n’a pas de builtin, ni d’environemnt.

On va maintenant ajouter quelques builtin à notre shell.

4) Built-in

Une builtin est une commande coder dans notre shell. c’est-à-dire une commande qui ne va pas s’exécuter avec execve.

Si on lance bash et que l’on supprime la variable d’environment PATH, notre shell doit quand même pouvoir exécuter des commandes rudimentaires.

les commandes cd, pwd, export, echo, exit ... doivent être exécutable.

Vous pouvez lister toutes les builtin sous bash avec la commande help

Nous allons voir les builtin cd, pwd . Il en existe beaucoup d'autres mais le but est juste que vous compreniez qu'est-ce qu'une builtin et comment les implémenter.

Pour la builtin cd, nous allons utiliser la fonction chdir

chdirprend en paramètre le path qui va devenir le dossier courant

Une exemple d'implementaion de la builtin cd :

```
void    built_in_cd(char *path)
{
        if (chdir(path) == -1) {
                perror("chdir()");
        }
}
```

Un exemple de la builtin pwd:

```
void    built_in_pwd(void)
{
        char cwd[PATH_MAX];

        if (getcwd(cwd, sizeof(cwd)) != NULL) {
               printf("%s\n", cwd);
        } else {
                perror("getcwd()");
        }
}
```

Une fois que nous passerons au code de l'environment, nous modifierons les built-in pour qu'elle utilise l'environment.

Le code à jour avec la gestion des built-in :

```
// Les deux includes a ajouter pour le type bool et le define PATH_MAX
#include <stdbool.h>
#include <linux/limits.h>

static bool     is_built_in(char *cmd)
{
        const char      *built_in[] = {"pwd", "cd", NULL};

        for (int i = 0; built_in[i]; i++) {
                if (!strcmp(built_in[i], cmd))
                        return (true);
        }
        return (false);
}

static void     exec_built_in(char **built_in)
{
        if (!strcmp(built_in[0], "pwd"))
                built_in_pwd();
        else if (!strcmp(built_in[0], "cd"))
                built_in_cd(built_in[1]);
}

int     main()
{
        char    *buffer = NULL;
        size_t  buf_size = 2048;
        char    **cmd = NULL;

        // alloc buffer qui stockera la commande entree par l'user
        buffer = (char *)calloc(sizeof(char), buf_size);
        if (buffer == NULL) {
                perror("Malloc failure");
                return (EXIT_FAILURE);
        }

        // ecriture d'un prompt
        write(1, "$> ", 3);

        // lecture de STDIN en boucle
        while (getline(&buffer, &buf_size, stdin) > 0) {
                cmd = split(buffer, " \n\t");

                if (cmd[0] == NULL)
                        printf("Command not found\n");
                else if (is_built_in(cmd[0]) == false) {
                        get_absolute_path(cmd);
                        exec_cmd(cmd);
                } else
                        exec_built_in(cmd);
```

```
                write(1, "$> ", 3);
                free_array(cmd);

        }

        printf("Bye \n");
        free(buffer);
}
```

Nous avons maintenant un shell qui peut exécuter une commande avec execve et des builtin :D.

Nous allons voir maintenant une chose importante dans un shell, l'environment.

5) l'environment

Dans n'importe quel shell Unix, la commande env affiche les variables d'environment.

Une variable d'environment sert à communiquer des informations entre plusieurs programmes.

En C, on peut récupérer l'ensemble des variables d'environment par le 3e argument de la fonction main, char **envp.

On peut soit crée un environment en dupliquant la variable envp, et/ou codant en dur un environment minimaliste.

Nous allons coder une fonction qui stocke les variables d'environment dans une liste chainée.

Si un utilisateur veut ajouter ou supprimer une variable, nous aurons juste à ajouter ou supprimer un maillon de notre liste : D.

Voilà la liste des variables qui seront ajoutées si elle n'existe pas dans envp :

- PATH : Pour avoir la liste des dossiers ou chercher les binaires a exécuter
- HOME : Pour connaitre ou est notre home :D
- OLDPWD : Pour connaitre le dossier dans lequel nous etions
- PWD : Pour connaitre le path actuelle
- SHLVL : Pour savoir combien de shell nous avons lancer

On commence par écrire notre fonction qui va dupliquer l'env. On parcourt envp et on stock chaque variable dans une liste chainée.

```
static void     add_env_var(char *var)
{
        struct passwd   *pw = getpwuid(getuid());
        char                    *alloc = NULL;

        if (!strcmp(var, "HOME")) {
                alloc = (char *)calloc(sizeof(char), strlen(pw->pw_dir) + strle
                if (alloc == NULL) {
                        fprintf(stderr, "Cannot add HOME\n");
                        return ;
                }
                strcat(alloc, "HOME=");
                strcat(alloc, pw->pw_dir);
        } else if (!strcmp(var, "PATH")) {
                alloc = strdup("PATH=/bin:/usr/bin");
                if (alloc == NULL) {
                        fprintf(stderr, "Cannot add PATH\n");
                        return ;
                }
        } else if (!strcmp(var, "OLDPWD")) {
                alloc = strdup("OLDPWD=");
                if (alloc == NULL) {
                        fprintf(stderr, "Cannot add OLDPWD\n");
                        return ;
                }
        } else if (!strcmp(var, "PWD")) {
                alloc = built_in_pwd();
                if (alloc == NULL) {
                        fprintf(stderr, "Cannot add PWD\n");
                        return ;
                }
        } else if (!strcmp(var, "SHLVL")) {
                alloc = strdup("SHLVL=1");
                if (alloc == NULL) {
                        fprintf(stderr, "Cannot add OLDPWD\n");
                        return ;
                }
        }

        add_tail(alloc);
}

static void     dup_env(char **envp)
{
        char    *var_lst[] = {"PATH", "HOME", "OLDPWD", "PWD", "SHLVL", NULL};
        ssize_t nb_elem = 5; // nombre d'element dasn var_lst

        // boucle sur l'env et stock les variables dans la liste
        for (int i = 0; envp[i]; i++) {
                add_tail(strdup(envp[i]));
```

```
                // On verifie que l'on a les variables d'environment minimal
                if (!strncmp(envp[i], "PATH", 4)) var_lst[0] = NULL;
                else if (!strncmp(envp[i], "HOME", 4)) var_lst[1] = NULL;
                else if (!strncmp(envp[i], "OLDPWD", 6)) var_lst[2] = NULL;
                else if (!strncmp(envp[i], "PWD", 3)) var_lst[3] = NULL;
                else if (!strncmp(envp[i], "SHLVL", 5)) var_lst[4] = NULL;
        }

        // On verifie qu l'on a les varaibles PATH, HOME, OLD_PWD et SHLVL
        // sinon on l'ajoute
        for (int i = 0; i < 5; i++) {
                if (var_lst[i] != NULL)
                        add_env_var(var_lst[i]);
        }
}
```

Je ne mets pas les fonctions pour manipuler ma liste chainée, le but de cet article est l'écriture d'un shell. Elles sont dans le code source.

J'ai modifié la builtin pwd. Maintenant qu'on a un env, ont parcour la liste jusqu'à trouver notre variable et on l'affiche.

Au lancement de shell on initialise la variable PWD si elle n'existe pas, et par la suite c'est la buitin cd qui modifiera PWD et OLD_PWD.

```
static char     *built_in_pwd(void)
{
        char    *cwd = NULL;

        // On alloue la longueur de PWD= + PATH_MAX + 1 pour le \0
        cwd = (char *)calloc(sizeof(char), PATH_MAX + strlen("PWD=") + 1);
        if (cwd == NULL)
                return (NULL);

        // On concatene le nom de la variable
        strcat(cwd, "PWD=");

        // et on stock le path actuelle apres le = de PATH=
        if (getcwd(&cwd[4], PATH_MAX) == NULL) {
                perror("getcwd()");
        }

        return (cwd);
}
```

J'ai aussi ajouté une builtin env qui affiche tout l'env :

```
static void     built_in_env(void)
{
        t_env   *tmp = first;

        while (tmp) {
                printf("%s\n", tmp->var);
                tmp = tmp->next;
        }
}
```

On peut aussi ajouter la builtin setenv pour ajouter une variable et unsetenv pour supprimer une variable. Je ne les ferais pas dans cet article, il suffit juste de supprimer un maillon ou d'en ajouter un à notre liste.

```
static void     built_in_cd(char *path)
{
        char    *oldpwd = NULL;
        char    *pwd = NULL;
        char    *pwd_ptr = NULL;

        if (path == NULL)
                return;
        if (chdir(path) == 0) {
                pwd = strrchr(get_env_var("PWD="), '=') + 1;
                oldpwd = strrchr(get_env_var("OLDPWD="), '=') + 1;

                if (oldpwd != NULL && pwd != NULL) {
                        strcpy(oldpwd, pwd);
                }
                if (pwd != NULL) {
                        pwd = &pwd[-strlen("PWD=")];
                        pwd_ptr = built_in_pwd();
                        strcpy(pwd, pwd_ptr);
                        free(pwd_ptr);
                        pwd_ptr = NULL;
                }
        } else {
                perror("chdir");
        }
}
```

Maintenant qu'on a un environment, on peut l'envoyer à execve à la place de NULL :D.

Voilà les bases d'un shell. Il y a énormément de chose qu'on peut ajouter:

- plein d'autre builtin
- un cd plus complet (gestion du ~ …)
- historique
- prompt dynamique avec le path …
- etc …

Le code complet en cliquant ici :