Réalisation d'un shell en C

Michel Billaud (michel.billaud@laposte.net)

3 février 2023

Table des matières

1 Objectif **2**

2 Première étape : faire une boucle de dialogue **2**
2.1 Utilisation de `getline()` . . . 2
2.2 Code de démonstration . . . 2
2.3 Exemple d'exécution . . . 3
2.4 Compilation et exécution . . . 4
2.5 Vérification des fuites mémoires . . . 4

3 Seconde étape : représenter un tableau de chaînes **5**
3.1 Code des tests, fonctionnalités (`tests-ptr-array.c`) . . . 5
3.2 Fichier d'en-tête `ptr-array.h` . . . 6
3.3 Implémentation des fonctions `ptr-array.c` . . . 7
3.4 Compilation et exécution des tests . . . 9

4 Troisième étape : découpage de la ligne **9**
4.1 Tests du découpage (`test-split-line.c`) . . . 10
4.2 Le fichier d'entête (`split-line.h`) . . . 11
4.3 Le code du découpage (`split-line.c`) . . . 11

5 L'exécution des commandes **13**
5.1 La structure `command_result` . . . 13
5.2 La fonction `main()` . . . 13
5.3 Les commandes internes . . . 14
5.4 La fonction `execute_command()` . . . 15
5.5 Lancement d'un programme par `execute_external_command()` . . . 17

6 Exemple d'exécution **18**

7 Conclusion **19**

8 Annexes **19**
8.1 Code entier `mini-shell.c` . . . 19
8.2 `Makefile` utilisé . . . 23
8.3 Exemple de recompilation complète . . . 24

1 Objectif

La réalisation d'un mini-*shell* (interprète de commandes) est un projet classique de programmation système en C.

Dans sa version la plus simpliste, un shell est une boucle qui

- affiche une chaîne d'invite (*prompt*),
- lit une commande,
- lance son exécution,
- et recommence.

Les commandes sont des suites de mots : en général le chemin d'accès d'un programme (exécutable), suivi par des options, des arguments…

Il y a aussi des commandes dites *internes*, par exemple `exit`, qui fait arrêter la boucle du shell. Un autre exemple est la commande `cd` qui change le répertoire courant.

Ce document montre un point de départ pour la réalisation d'un tel shell, avec une version qui exécute des commandes simples.

2 Première étape : faire une boucle de dialogue

Le dialogue se fait sur la base d'une boucle qui

- demande à l'utilisateur de taper quelque chose,
- lit la ligne qui a été tapée,
- fait quelque chose avec,
- et recommence ;

et ceci, tant qu'on n'a pas tapé quelque chose de particulier qui dit d'arrêter.

2.1 Utilisation de `getline()`

Pour lire une ligne utilisateur, on peut recommander d'utiliser la fonction `getline`, qui lit une ligne complète sans limitation de taille. C'est une fonction standard POSIX.

Elle travaille avec un tampon, tableau de caractères qui est alloué et rallongé automatiquement au besoin. Ce tampon est décrit par deux variables

- un pointeur qui indique son début (initialement on y met `NULL`),
- sa taille dans un entier (initialement 0),

dont on passe l'adresse à la fonction `getline()`.

La fonction `getline()` repose sur la gestion du tampon par `alloc()` et `realloc`().

2.2 Code de démonstration

Le code ci-dessous montre la réalisation d'une boucle avec lecture par `getline()`

```c
// demo-boucle.c

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

int main()
{
    char *line = NULL;
    size_t line_size = 0;
    printf("--- Démo Boucle\n");
    printf("tapez exit pour arrêter\n");

    int numero = 1;
    for (bool run_loop = true; run_loop; ) {
        printf("%d> ", numero);
        int length = getline(& line, &line_size, stdin);
        if (length < 0) {
            break;
        }
        printf("=> %s\n", line);
        if (strncmp(line, "exit", 4) == 0) {
            run_loop = false;
        }
        numero += 1;
    }

    printf("Bye!\n");
    free(line);
    return EXIT_SUCCESS;
}
```

Remarques :

1. La détection de l'arrêt se fait de façon un peu sommaire, en regardant si les 4 caractères `exit` figurent au début de la ligne.
2. Le test `if (length < 0)` détecte si l'utilisateur a fermé l'entrée standard en tapant `contrôle-D` dans la console.
3. Par mesure d'hygiène, on libère le tampon `line` à la fin du programme.

2.3 Exemple d'exécution

Si on fait tourner le programme, on remarque que `getline()` a aussi récupéré dans le tampon `line` le caractère de saut de ligne tapé par l'utilisateur. Ce qui fait afficher une ligne blanche.

```
$ ./demo-boucle
-- Démo Boucle
tapez exit pour arrêter
1> premiere ligne
```

```
=> premiere ligne

2> la seconde ligne
=> la seconde ligne

3> exit
=> exit

Bye!
```

2.4 Compilation et exécution

Le source a été compilé avec les options suivantes :

```
gcc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused \
    -D_XOPEN_SOURCE=700 \
    demo-boucle.c   -o demo-boucle
```

qui permettent l'utilisation de la bibliothèque POSIX (= XOPEN).

2.5 Vérification des fuites mémoires

On peut vérifier que ce programme n'a pas de fuite mémoire, en le lançant sous contrôle de `valgrind` :

```
$ valgrind -s -q --leak-check=full ./demo-boucle
--- Démo Boucle
tapez exit pour arrêter
1> une ligne
=> une ligne

2> une seconde ligne
=> une seconde ligne

3> exit
=> exit

Bye!
==8085== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Si vous mettez en commentaire la libération de `line`, vous obtiendrez un message du type

```
==7873== 120 bytes in 1 blocks are definitely lost in loss record 1 of 1
==7873==    at 0x483877F: malloc (vg_replace_malloc.c:307)
==7873==    by 0x48D373F: getdelim (iogetdelim.c:62)
==7873==    by 0x1091D1: main (demo-boucle.c:18)
==7873==
==7873== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

3 Seconde étape : représenter un tableau de chaînes

Notre programme devra découper la ligne qui est tapée par l'utilisateur. Par exemple, à partir de `"ls -l /tmp"`, il faudra obtenir un tableau contenant 3 chaînes `"ls"`, `"-l"` et `"/tmp"`.

Comme le langage C ne propose pas de conteneurs, on va réaliser nous même un type de données "tableau extensible de pointeurs".

Ce type de données sera compilé séparément, il est

- décrit par un fichier d'en-tête `ptr-array.h`, avec une définition de structure `ptr_array`, et des fonctions qui agissent dessus (avec préfixe `pa_`)
- implémenté dans `ptr-array.c`.

On choisit de faire un tableau de pointeurs "génériques" (`void *`), parce que c'est plus général que de se spécialiser sur un tableau de pointeurs de caractères, et que ce n'est absolument pas plus compliqué.

Ça répond même à une question existentielle : si un tableau contient des chaînes, ces chaînes appartiennent-elles au tableau ? Autrement dit, faut-il les libérer quand on libère le tableau ?

Si on part comme ici sur la base d'un tableau de *pointeurs*, la réponse est clairement non. Quand le tableau disparaît, les pointeurs aussi, mais libérer les objets pointés, c'est pas la responsabilité du tableau.

C'est toujours une bonne idée de commencer par écrire des tests pour les modules que l'on réalise. Ça permet de voir de quoi on a besoin.

3.1 Code des tests, fonctionnalités (`tests-ptr-array.c`)

```c
// tests-ptr-array.c

#include <stdio.h>
#include <assert.h>

#include "ptr-array.h"

void basic_test()
{
    printf("- basic_test()\n");
    struct ptr_array sa = pa_new();

    assert("new PtrArray size is zero" &&
           pa_size(&sa) == 0);

    char *s[21] = {
        "zero", "one", "two", "three", "four",
        "five", "six", "seven", "eight", "nine",
        "ten", "eleven", "twelve", "thirteen", "fourteen",
```

```
        "fifteen", "sixteen", "seventeen", "eighteen", "nineteen",
        "twenty"
    };

    for (size_t i = 0; i < 21; i++) {
        pa_add(&sa, s[i]);
        assert("adding increases size"
               && pa_size(&sa) == i+1);
        assert("data is present at end"
               && pa_get(&sa, i) == s[i]);
    }

    for (size_t i = 0; i < 21; i++) {
        assert("all added data is there"
               && pa_get(&sa, i) == s[i]);
    }

    pa_delete(&sa);

}

int main()
{
    printf("# Tests ptr_array\n");
    basic_test();
    printf("Ok\n");
}
```

Les fonctionnalités testées :

- `pa_new()` : retourne un tableau de pointeurs initialisé à vide ;
- `pa_size()` : indique le nombre de pointeurs présents dans le tableau ;
- `pa_add()` : ajoute un pointeur à la fin du tableau ;
- `pa_get()` : retourne le n-ième pointeur (indices à partir de 0) ;
- `pa_delete()` : libère le contenu d'un tableau de pointeurs.

En gros le test

- initialise un tableau (vide),
- vérifie que sa taille est 0,
- y ajoute 21 pointeurs en vérifiant à chaque étape
  - que la taille a augmenté
  - que l'élément ajouté est présent à la fin
- puis vérifie que les 21 éléments sont présents à leur place.

3.2 Fichier d'en-tête `ptr-array.h`

Un "`ptr_array`" est matérialisé par une structure contenant

- `array` : un tableau de pointeurs, alloué dynamiquement, et qui sera ré-alloué quand on aura besoin de l'agrandir ;
- `capacity` : le nombre de pointeurs que peut contenir le tableau alloué ;

- `size` : le nombre de pointeurs utilisés pour l'instant.

```
// ptr-array.h

#ifndef PTR_ARRAY_H
#define PTR_ARRAY_H

#include <stddef.h>

struct ptr_array {
    size_t size;
    size_t capacity;
    void **array;
};

void pa_init(struct ptr_array *pa);
struct ptr_array pa_new();

void pa_delete(struct ptr_array *pa);
void pa_add(struct ptr_array *pa, void *ptr);

size_t pa_size(const struct ptr_array *pa);
void * pa_get(const struct ptr_array *pa, size_t index);

#endif
```

Remarques

- On a utilisé `const` autant que possible, pour les fonctions qui reçoivent l'adresse d'un objet qu'elles ne sont pas censées modifier.
- La fonction `pa_new()` semble faire double emploi avec `pa_init()` : en fait elle permet de combiner élégamment définition et initialisation d'un tableau, comme dans

```
struct ptr_array sa = pa_new();
```

3.3 Implémentation des fonctions `ptr-array.c`

C'est une implémentation classique d'un tableau extensible

- lors de son initialisation, sa capacité est fixée à un petit nombre d'éléments (ici 8 pointeurs) ;
- quand il est plein et qu'on veut ajouter un élément supplémentaire, la capacité est doublée par réallocation.

```
// ptr-array.c

#include <stdlib.h>
#include <memory.h>

#include "ptr-array.h"

#ifndef PTR_ARRAY_MIN_CAPACITY
```

```
    #define PTR_ARRAY_MIN_CAPACITY 8
#endif

struct ptr_array pa_new() {
    return (struct ptr_array) {
        .size = 0,
        .capacity = PTR_ARRAY_MIN_CAPACITY,
        .array = malloc(PTR_ARRAY_MIN_CAPACITY
                        * sizeof(void *))
    };
}

void pa_init(struct ptr_array *pa)
{
    *pa = pa_new();
}

void pa_delete(struct ptr_array *pa)
{
    pa->size = 0;
    pa->capacity = 0;
    free(pa->array);
    pa->array = NULL;
}

void pa_add(struct ptr_array *pa, void *ptr)
{
    if (pa->size == pa->capacity) {
        pa->capacity *= 2;
        pa->array = realloc(pa->array,
                            pa->capacity * sizeof(void *);
    }
    pa->array[pa->size++] = ptr;
}

size_t pa_size(const struct ptr_array *pa)
{
    return pa->size;
}

void * pa_get(const struct ptr_array *pa, size_t index)
{
    return pa->array[index];
}
```

Ceci garantit que le coût amorti moyen d'un ajout est élémentaire O(1).

Pour ceux qui ne sont pas familiers avec ces notions

- les ré-allocations se produisent quand on arrive à 8+1, 16+1, 32+1, etc.
- quand on arrive par exemple à 1024+1, en tout on aura eu des allocations

de 8, 16, 32, … 1024 et 2048.
- le coût d'une ré-allocation est proportionnel à sa taille (il faut recopier l'ancien contenu)
- le coût total est donc de 8+16+…+2048, ce qui fait un peu moins de 4096, pour 1024+1 à 2048 éléments.
- donc le coût moyen par élément est entre 4096/2048 et 4096/1024, c'est à dire entre 2 et 4.

Remarques :

- noter, dans `pa_new()`, le retour d'une structure anonyme ;
- dans le test, on a choisi d'ajouter 21 éléments pour que se produisent
  - l'allocation initiale de 8 pointeurs,
  - le passage à 16 lors du 9-ième ajout,
  - le passage à 32 lors du 17-ième ajout.

3.4 Compilation et exécution des tests

On effectue une compilation séparée

```
$ gcc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g   -

$ gcc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g   -

$ gcc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g
```

Les tests se déroulent sans erreurs, et ne montrent pas de fuite mémoire :

```
$ valgrind -s -q --leak-check=full ./ptr_array
# Tests ptr_array
- basic_test()
Ok
==8680== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

4 Troisième étape : découpage de la ligne

Ici aussi nous commençons par les tests.

Ce que nous voulons c'est une fonction `split_line()` qui

- reçoit une chaîne de caractères
- retourne une structure `split_line_result` contenant
  - dans le champ `strings` le tableau de chaînes résultant du découpage de la ligne,
  - des indications sur d'éventuelles erreurs de syntaxe.

On ne les utilisera pas ici, mais les erreurs pourraient être des guillemets manquants etc. dans une version plus réaliste d'un shell.

4.1 Tests du découpage (`test-split-line.c`)

```c
// test-split-line.c

#include <stdio.h>
#include <string.h>

#include <assert.h>

#include "split-line.h"

void test_empty(const char line[])
{
    struct split_line_result r = split_line(line);

    assert("empty line has no words" &&
           pa_size(& r.strings) == 0);
    split_line_result_delete(& r);
}

void basic_test()
{
    printf("- basic_test()\n");
    char line[] = "  one  two three";
    struct split_line_result r = split_line(line);
    assert("example size is 3"
           && pa_size(&r.strings) == 3);
    assert("first word is 'one'" &&
            strcmp(pa_get(&r.strings, 0), "one") == 0);
    assert("second word is 'two'" &&
            strcmp(pa_get(&r.strings, 1), "two") == 0);
    assert("third word is 'three'" &&
            strcmp(pa_get(&r.strings, 2), "three") == 0);
    split_line_result_delete(& r);

}

void test_empty_lines()
{
    printf("- test_empty_lines()\n");
    test_empty("");
    test_empty("\n");
    test_empty("    \n");
}

int main()
{
    printf("# tests split-line\n");
    test_empty_lines();
    basic_test();
```

```
    printf("Ok\n");
}
```

Comme on le voit, il y a

- des tests sur différentes formes de lignes vides (avec des espaces, des sauts de ligne),
- une vérification du découpage d'une ligne de 3 mots.

La fonction `split_line_result_delete()` est chargé de libérer les ressources contenues dans une structure `split_line_result`.

4.2 Le fichier d'entête (`split-line.h`)

```
// split-line.h

#ifndef SPLIT_LINE_H
#define SPLIT_LINE_H

#include "ptr-array.h"

struct split_line_result {
    struct ptr_array strings;
    const char *error_message;
    size_t error_position;
};

struct split_line_result split_line(const char line[]);
void split_line_result_delete(struct split_line_result *r);

#endif
```

4.3 Le code du découpage (`split-line.c`)

Ici on se contente d'un découpage très sommaire : on considère que la ligne se décompose en mots qui étaient séparés par des espaces.

```
// split-line.c

#include <malloc.h>
#include <string.h>
#include <stdbool.h>
#include <ctype.h>

#include "split-line.h"

static bool is_ending_char(char c)
{
    return (c == '#') || (c == '\0');
}

static bool is_word_char(char c)
```

```
{
    return !( isspace(c) || is_ending_char(c));
}

struct split_line_result split_line(const char line[])
{
    struct split_line_result r = {
        .error_message = NULL,
        .error_position = 0,
        .strings = pa_new()
    };

    int i = 0;

    for (;;) {
        // ignore spaces
        while(isspace(line[i])) {
            i++;
        };
        if (is_ending_char(line[i])) {
            break;
        }
        int start_index = i;
        do {
            i += 1;
        } while(is_word_char(line[i]));

        pa_add(& r.strings,
               strndup(& line[start_index],
                       i - start_index));
    }

    return r;
}

void split_line_result_delete(struct split_line_result *r)
{
    const size_t n = pa_size(& r->strings);
    for (size_t i = 0; i < n; i++) {
        free(pa_get(& r->strings, i));
    }
    pa_delete(& r->strings);
}
```

Explications :

1. La boucle `for` de `split_line()` remplit un tableau de chaînes :
    - elle repère le premier caractère non-blanc,
    - s'arrête si on est arrivé au bout (fin de la ligne, ou début d'un commentaire),

- cherche la fin du mot,
- ajoute une copie du mot dans le tableau par `strndup()` - qui fait partie de la bibliothèque POSIX -,
- et recommence.

2. Comme `strndup()` alloue dynamiquement des chaînes, la fonction `split_line_result_delete()` s'occupera de les libérer, avant de libérer le tableau de pointeurs lui-même.

5 L'exécution des commandes

La fonction `main()` du programme final est similaire à ce qui a été vu au début de ce document, si ce n'est qu'elle va

- appeler `split_line()` pour découper la ligne
- afficher un message si il y a eu une erreur de syntaxe dans la ligne tapée,
- sinon appeler une fonction `execute_command()` en lui donnant le tableau des mots de la ligne, ce qui nous fournira un résultat de type `struct command_result`. qui indique comment la commande s'est déroulée.

5.1 La structure `command_result`

```
struct command_result {
    bool exit_shell;
    int  status;
    int  errnum;
};
```

- `exit_shell` est un booléen qui dit si il faut arrêter la boucle,
- `status` sera la valeur retournée par l'`exit()` du `main()` ;
- `errnum` contient le "`errno`" qui est positionné par les commandes système. Il nous servira à afficher le message d'erreur approprié.

5.2 La fonction `main()`

La fonction `main()` effectue la boucle de dialogue. avec l'utilisateur, et le traitement de la ligne tapée :

- découpage de la ligne
- affichage éventuel d'un message en cas d'erreur
- appel de la fonction `execute_command()`
- affichage éventuel de message d'erreur si

```
#define SHELL_NAME "Mini Shell V1.1"


int main()
{
    char *line = NULL;
    size_t line_size = 0;
    printf("--- Hello, this is " SHELL_NAME "!\n");
```

```
    int numero = 1;
    for (bool run_loop = true; run_loop; ) {
        printf("%d> ", numero);
        int length = getline(& line, &line_size, stdin);
        if (length < 0) {
            break;
        }
        struct split_line_result slr = split_line(line);
        if (slr.error_message != NULL) {
            printf("Syntax Error at char %zu: %s\n",
                   slr.error_position, slr.error_message);
        } else {
            if (pa_size(& slr.strings) != 0) {
                struct command_result cr = execute_command(& slr.strings);
                numero += 1;  // doesn't advance if empty command
                if (cr.errnum != 0) {
                    printf("Error: %s\n", strerror(cr.errnum));
                }
                if (cr.exit_shell) {
                    run_loop = false;
                }
            }
        }
        split_line_result_delete(& slr);
    }

    printf("Bye!\n");
    free(line);
    return EXIT_SUCCESS;
}
```

5.3 Les commandes internes

Les fonctions qui exécutent les commandes ont le même prototype que `execute_command()`

```
/ ----------------------------------------
// internal commands
//

struct command_result execute_internal_exit(
    const struct ptr_array *args)
{
    int status = pa_size(args) == 1
                 ? 0
                 : atoi(pa_get(args, 1));
    return (struct command_result) {
        .exit_shell = true,
        .status = status,
        .errnum = 0
```

```
    };
}

struct command_result execute_internal_cd(
    const struct ptr_array *args)
{
    char *dest = pa_size(args) == 1
                 ? getenv("HOME")
                 : pa_get(args, 1);
    int status = chdir(dest);
    return (struct command_result) {
        .exit_shell = false,
        .status = status,
        .errnum = errno
    };
}

struct command_result execute_internal_help(
    const struct ptr_array *args)
{
    printf("Commands: \n"
           "\texit [value] leave the shell\n"
           "\tcd  [dir]     change current directory (default: home)\n"
           "\thelp  | ?     this message\n"
          );
    return (struct command_result) {
        .exit_shell = false,
        .status = EXIT_SUCCESS,
        .errnum = 0
    };
}
```

Commentaires

- la commande interne “exit” a un paramètre optionnel qui est une valeur à retourner (par défaut 0). Elle positionne l’indicateur de sortie.
- la commande “cd” a un paramètre optionnel, le chemin du répertoire destination (par défaut le répertoire d’accueil qui est indiqué dans la variable d’environnment `HOME`). L’appel système `chdir` pouvant échouer, le code de retour et le numéro d’erreur sont rapportés à l’appelant.
- la commande “help” affiche juste quelques messages.
- on ne fait guère d’effort pour s’occuper des paramètres excédentaires !

5.4 La fonction `execute_command()`

Cette fonction doit déterminer si le premier mot est le nom d’une des commandes internes, et

- si oui lancer la fonction qui s’en occupe,
- sinon il s’agit de faire exécuter un programme (commande externe).

Comme un shell possède de nombreuses commandes internes, on évite de coder

une série de tests emboîtés, au profit d'une table de correspondance entre des noms de commandes, et des pointeurs vers les fonctions qui s'en occupent :

```
// ------------------------------------------------
// table of internal commands :
//

typedef struct command_result (*FUNCTION)(
    const struct ptr_array *
);

struct {
    const char *name;
    FUNCTION function;
} commands_table[] = {
    {"exit", &execute_internal_exit},
    {"cd", &execute_internal_cd},
    {"help", &execute_internal_help},
    {"?", &execute_internal_help},

    {NULL, NULL}
};
```

qui sera beaucoup plus facile à maintenir (et permet des synonymes, commme `"?"` pour `"help"`).

La fonction `execute_command()`

- cherche dans cette table la fonction qui correspond au premier mot de la ligne
- l'exécute en lui passant les paramètres.

Si le nom n'y figure pas, il s'agit d'une commande externe, qui sera traitée par la fonction `execute_external_command` (voir plus loin)

```
struct command_result execute_command(
    struct ptr_array *args
)
{
    const char * name = pa_get(args, 0);
    FUNCTION function = &execute_external_command;
    for (int i = 0; commands_table[i].name != NULL; i++) {
        if (strcmp(commands_table[i].name, name) == 0) {
            // internal command found
            function = commands_table[i].function;
            break;
        }
    }
    // apply function to args
    return (*function)(args);
}
```

5.5 Lancement d'un programme par `execute_external_command()`

Pour exécuter une commande externe, le shell

- crée un processus fils qui appelle `execvp()` en lui donnant en paramètre un tableau de chaines avec le nom de l'exécutable et les arguments, suivis par un pointeur nul.
- attend la fin de ce processus fils
- retourne le status renvoyé par le processus fils.

```
struct command_result execute_external_command(
    const struct ptr_array *args
)
{
    pid_t child_pid = fork();
    if (child_pid == 0) {
        // build array
        size_t nb_args = pa_size(args);
        char **a = malloc(sizeof(char *)
                          * (nb_args + 1));
        for (size_t i = 0; i < nb_args; i++) {
            a[i] = pa_get(args, i);
        }
        a[nb_args] = NULL;
        execvp(a[0], a);

        // if failed
        perror("execvp failed");
        exit (EXIT_FAILURE);
    }
    int wait_status;
    waitpid(child_pid, &wait_status, 0);
    return (struct command_result) {
        .exit_shell = false,
        .errnum = 0,
        .status = WEXITSTATUS(wait_status)
    };
}
```

Appels système utilisés :

- `fork()` crée un processus "fils" qui est une copie de celui qui l'appelle (= père). Il retourne 0 au fils, et le numéro du fils au père.
- `exit()` termine le processus qui l'appelle.
- `waitpid` bloque un processus (ici le père) en attente de la fin d'un autre dont on donne le numéro (ici le fils).
- `execvp()` tente de remplacer le processus courant par l'exécution d'un programme en lui transmettant des arguments. L'exit du programme terminera le processus. En cas d'échec de lancement (fichier absent, non exécutable, etc) l'appelant continue.

Compléments :

- il existe une famille de fonctions “`exec`”.
  - `execv` prend comme paramètre le **chemin d'accès** de l'exécutable et un tableau de paramètres ;
  - celle de l'exemple (`execvp`) prend comme paramètre un **nom de commande** et un tableau de paramètres. L'exécutable est cherché dans les répertoires indiqués par la variable d'environnement `PATH`.
  - Voir aussi `execl` et `execlp` qui utilisent une **liste** de paramètres au lieu d'un **tableau**.
- ici on n'a utilisé que le premier paramètre de `waitpid`.
  - le troisième permet d'attendre un **changement d'état** du processus fils (pas seulement sa fin).
  - Le second, si il n'est pas `NULL`, est l'adresse d'un entier qui permettra par exemple de récupérer le code laissé par l'`exit` du processus fils. Voir la page de manuel.

6 Exemple d'exécution

```
$ ./mini-shell
--- Hello, this is Mini Shell V1.1!
1> ls
demo-boucle.c  ptr-array.h      tests-ptr-array.c
Makefile       ptr-array.o      tests-ptr-array.d
mini-shell     split-line.c     tests-split-line
mini-shell.c   split-line.d     tests-split-line.c
mini-shell.d   split-line.h     tests-split-line.d
ptr-array.c    split-line.o
ptr-array.d    tests-ptr-array
2> help
Commands:
        exit [value] leave the shell
        cd  [dir]     change current directory (default: home)
        help  | ?     this message
3> cd
4> pwd
/home/billaud
5> cd /tmp
6> pwd
/tmp
7> exit 123
Bye!

$ echo $?
123
```

La dernière commande montre que le status donné en argument dans notre mini-shell a bien été transmis.

7 Conclusion

On a présenté ici la construction d'un mini-shell très rudimentaire avec possibilités de lancer des commandes externes, et quelques fonctions internes.

Dans un vrai shell, il y a des variables, des boucles, des redirections, de la gestion des processus etc. On en est très loin. Il ne faut pas cacher que ça ne serait pas facile, ne serait ce qu'en raison de la syntaxe étrange des langages de commande habituels.

L'important ici est la construction à partir d'un découpage en modules : conteneur `ptr-array`, analyseur `split-line`, et leur développement à partir de tests. qui sont testés continuellement, en particulier l'absence de fuites mémoires.

8 Annexes

8.1 Code entier `mini-shell.c`

Le code est présenté en entier, avec les directives `include` nécessaires, et dans l'ordre :

```
// mini-shell.c

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

#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

#include "ptr-array.h"
#include "split-line.h"

#define SHELL_NAME "Mini Shell V1.1"


// all commands receive an array of args
// and return a command_result

struct command_result {
    bool exit_shell;
    int  status;
    int  errnum;
};

typedef struct command_result (*FUNCTION)(
    const struct ptr_array *
```

```
);

// ----------------------------------------
// table of internal commands
//

struct command_result execute_internal_exit(
    const struct ptr_array *args)
{
    int status = pa_size(args) == 1
                 ? EXIT_SUCCESS
                 : atoi(pa_get(args, 1));
    return (struct command_result) {
        .exit_shell = true,
        .status = status,
        .errnum = 0
    };
}

struct command_result execute_internal_cd(
    const struct ptr_array *args)
{
    char *dest = pa_size(args) == 1
                 ? getenv("HOME")
                 : pa_get(args, 1);
    int status = chdir(dest);
    return (struct command_result) {
        .exit_shell = false,
        .status = status,
        .errnum = errno
    };
}

struct command_result execute_internal_help(
    const struct ptr_array *args)
{
    printf("Commands: \n"
           "\texit [value] leave the shell\n"
           "\tcd  [dir]     change current directory (default: home)\n"
           "\thelp  | ?     this message\n"
          );
    return (struct command_result) {
        .exit_shell = false,
        .status = 0,
        .errnum = 0
    };
}

// ----------------------------------------------------
// table of internal commands :
```

```c
//
struct {
    const char *name;
    FUNCTION function;
} commands_table[] = {
    {"exit", &execute_internal_exit},
    {"cd", &execute_internal_cd},
    {"help", &execute_internal_help},
    {"?", &execute_internal_help},

    {NULL, NULL}
};

// ------------------------------------------------


struct command_result execute_external_command(
    const struct ptr_array *args
)
{
    pid_t child_pid = fork();
    if (child_pid == 0) {
        // build array
        size_t nb_args = pa_size(args);
        char **a = malloc(sizeof(char *)
                          * (nb_args + 1));
        for (size_t i = 0; i < nb_args; i++) {
            a[i] = pa_get(args, i);
        }
        a[nb_args] = NULL;
        execvp(a[0], a);
        // if failed
        perror("execvp failed");
        exit (EXIT_FAILURE);
    }
    int wait_status;
    waitpid(child_pid, &wait_status, 0);
    return (struct command_result) {
        .exit_shell = false,
        .errnum = 0,
        .status = WEXITSTATUS(wait_status)
    };
}

struct command_result execute_command(
    struct ptr_array *args
)
{
    const char * name = pa_get(args, 0);
    FUNCTION function = &execute_external_command;
```

```
    for (int i = 0; commands_table[i].name != NULL; i++) {
        if (strcmp(commands_table[i].name, name) == 0) {
            // internal command found
            function = commands_table[i].function;
            break;
        }
    }
    // apply function to args
    return (*function)(args);
}

int main()
{
    char *line = NULL;
    size_t line_size = 0;
    int final_status = EXIT_SUCCESS;
    printf("--- Hello, this is " SHELL_NAME "!\n");

    int numero = 1;
    for (bool run_loop = true; run_loop; ) {
        printf("%d> ", numero);
        int length = getline(& line, &line_size, stdin);
        if (length < 0) {
            break;
        }
        struct split_line_result slr = split_line(line);
        if (slr.error_message != NULL) {
            printf("Syntax Error at char %zu: %s\n",
                   slr.error_position, slr.error_message);
        } else {
            if (pa_size(& slr.strings) != 0) {
                struct command_result cr = execute_command(& slr.strings);
                numero += 1;  // doesn't advance if empty command
                if (cr.errnum != 0) {
                    printf("Error: %s\n", strerror(cr.errnum));
                }
                if (cr.exit_shell) {
                    run_loop = false;
                    final_status = cr.status;
                }
            }
        }
        split_line_result_delete(& slr);
    }

    printf("Bye!\n");
    free(line);
    return final_status;
}
```

8.2 `Makefile` utilisé

Un `Makefile` est utilisé pour

- **générer automatiquement les dépendances des sources** option `-MMD` de `gcc`, inclusion par `-include $(wildcard *.d)`),
- lancer systématiquement tous les tests unitaires à chaque recompilation.

La composition de chaque exécutable est décrite par une ligne. Exemple

`mini-shell: split-line.o ptr-array.o`

dit que l'exécutable `mini-shell` nécessite les fichiers objets `split-line.o` et `ptr-array.o` (en plus de `mini-shell.o`).

Le source du `Makefile` :

```
CFLAGS = -std=c17
CFLAGS += -Wall -Wextra -pedantic -Werror -Wno-unused
CFLAGS += -MMD
CFLAGS += -D_XOPEN_SOURCE=700
CFLAGS += -g

VALGRIND_OPTIONS = -s -q --leak-check=full


TESTS = tests-ptr-array
TESTS += tests-split-line

# EXECS = demo-boucle
EXECS += mini-shell

all : tests execs

tests : $(TESTS)
    for p in $(TESTS) ; do valgrind $(VALGRIND_OPTIONS) ./$$p ; done


execs : $(EXECS)
    for p in $(EXECS) ; do ./$$p ; done


# composition des exécutables

tests-ptr-array:    ptr-array.o
tests-split-line:   split-line.o   ptr-array.o

mini-shell: split-line.o ptr-array.o

# dépendances automatiques

-include $(wildcard *.d)
```

```
# utilitaires

clean:
    $(RM) *~ *.o *.d

mrproper: clean
    $(RM) $(EXECS) $(TESTS)
```

8.3 Exemple de recompilation complète

En lançant `make` après avoir nettoyé le répertoire, on obtient

- la compilation séparée des différents modules
- la fabrication des exécutables
- l'exécution des tests unitaires
- le lancement du mini-shell

```
$ make
cc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g   -c -
cc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g    tes
cc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g   -c -
cc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g    tes
for p in tests-ptr-array  tests-split-line ; do valgrind -s -q --leak-check=full ./$p ; do
# Tests ptr_array
- basic_test()
Ok
==4708== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
# tests split-line
- test_empty_lines()
- basic_test()
Ok
==4709== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
cc -std=c17 -Wall -Wextra -pedantic -Werror -Wno-unused -MMD -D_XOPEN_SOURCE=700 -g    min
for p in mini-shell ; do ./$p ; done
--- Hello, this is Mini Shell V1.1!
1>
```