# STI 1ère année – Programmation Système

## TP 1: Mini-GLIBC, commandes système et SHELL

J. Briffaut

## 1 Mini-GLIBC : Bibliothèque

Dans cet exercice, on se propose de ré-implémenter quelques fonctions de la GLIBC afin de concevoir une mini bibliothèque système qui nous permettra ensuite d'implémenter des commandes système puis un terminal.

Pour cela, nous n'utiliserons que des appels système. Attention, vous devez gérer proprement les erreurs !

Attention à n'utiliser aucune fonction de la GLIBC dans votre code, et uniquement les fonctions de votre librairie au fur et à mesure que vous les définirez !!!

**Exercice 1** Créer un fichier `mini_lib.h` dans lequel vous mettrez, au fur et à mesure, toutes les fonctions que vous définirez au cours de ce TP.

**Exercice 2** Créer un fichier `main.c` qui vous permettra de tester vos fonctions

### 1.1 Gestion de la mémoire

**Exercice 3** Créer un fichier `mini_memory.c`.

**Exercice 4** Écrire une fonction

```
void* mini_calloc (int size_element, int number_element)
```

Cette fonction prend en paramètre le nombre d'éléments d'une taille spécifique de mémoire à allouer, et renvoie un pointeur sur l'espace mémoire alloué. Elle devra utiliser la fonction système ;`sbrk`.

**Exercice 5** Modifier votre fonction pour qu'elle initialise le buffer avec des '\0' avant de le renvoyer. Pourquoi faire cette opération ?

**Exercice 6** Que fait la fonction `free` en C ? libère-t'elle vraiment la mémoire ?

Nous allons modifier `mini_calloc` pour mémoriser les zones mémoires allouées, leurs tailles, et l'état (0 :libre, 1 :utilisé). Ce qui permettra de réutiliser une zone libérée si une zone a été libérée et a une taille suffisante.

**Exercice 7** Définir une structure `malloc_element` qui contient :

- un pointeur (void *), qui correspond à une zone allouée par `mini_calloc`
- la taille de cette zone `size_element * number_element`
- son statut (0 :libre, 1 :utilisé)
- un pointeur vers le prochain élément `next_malloc` (`NULL` s'il n'y a pas de suivant). Et oui nous allons utiliser une liste !

**Exercice 8** Ajouter une variable globale `malloc_list` qui correspondra à notre liste de zones mémoire allouées. Initialiser cette variable à `NULL`.

**Exercice 9** Modifier `mini_calloc` pour ajouter la zone allouée à la liste `malloc_list`.

**Exercice 10** Ajouter la fonction `void mini_free(void* ptr)` qui passera la zone allouée à non-utilisée (si la zone avait été allouée précédemment)

**Exercice 11** Modifier `mini_calloc` pour réutiliser une zones libérée, si la taille demandée est plus petite ou égale à la taille d'une zone libérée.

**Exercice 12** Proposer une solution (à implémenter dans le `main.c` pour tester `mini_calloc` et `free` )

**Exercice 13** Écrire une fonction `void mini_exit()`. Cette fonction mettra fin au programme en utilisant l'appel système `_exit`. Terminer votre programme de test avec `mini_exit` .

**Exercice 14** Ajouter les prototypes de `mini_calloc`, `mini_free` et `mini_exit` a `mini_lib.h`.

## 1.2 Gestion des chaînes de caractères

**Exercice 15** Créer un fichier `mini_string.c`.

1. définir une variable globale `BUF_SIZE` qui fixera la taille du tampon à 1024.
2. déclarer une variable globale `buffer` (notre tampon) de type chaîne de caractères.
3. Nous aurons besoin de connaître le nombre d'éléments stockés dans le tampon : déclarer une variable globale `ind` de type entier qui sera l'index de la première case vide du tampon. Instancier `ind` par la valeur `-1` pour exprimer le fait que le `buffer` n'a pas encore été instancié.

**Exercice 16** Écrire une fonction `void mini_printf(char*)`, similaire à printf. Lors de son premier appel, `mini_printf` initialisera le tampon en lui allouant de la mémoire. En général, `mini_printf` utilisera la chaîne de caractères passée en argument pour remplir le tampon. Lorsque le tampon est rempli ou que le dernier caractère ajouté au tampon est un saut de ligne (`'\n'`), `mini_printf` affichera le contenu du tampon à l'écran à l'aide de l'appel système `write`, puis videra le tampon.

**Exercice 17** Tester vos fonctions dans `main.c`, en particulier, essayer d'afficher une unique chaîne de caractères qui ne contient pas de saut de ligne. Quel problème reste-t-il à régler ?

**Exercice 18** En modifiant `mini_exit`, résoudre le problème décrit à la question précédente. Exécuter et analyser votre code avec les commandes `strace`.

**Exercice 19** Écrire une fonction `int mini_scanf(char* buffer, int size_buffer )`, similaire à scanf. Cette fonction liera depuis l'entrée standard, au plus `size_buffer` caractères) et stockera le contenu dans `buffer`. Elle retournera le nombre de caractères lus.

**Exercice 20** Ajouter un test dans votre `main.c` pour cette fonction. Que se passe t'il si le nombre de caractères saisis est égal à la taille du buffer ? Proposer et implémenter une solution.

**Exercice 21** Ajouter les fonctions suivantes :

1. `int mini_strlen(char* s)` : retourne la taille de la chaîne `s`, i.e. le nombre de caractère jusqu'a un `'\0'`

2. `int mini_strcpy(char* s, char *d)` : copie la chaîne s dans d, retourne le nombre de caractères copiés
3. `int mini_strcmp(char* s1, char* s2)` : compare les chaînes s1 et s2, retourne 0 si identique

**Exercice 22** Quels sont les problèmes de ces fonctions (notamment en terme de sécurité) ? Proposer une correction de ces fonctions.

**Exercice 23** Ajouter une fonction `void mini_perror(char * message)` qui affiche un message d'erreur suivi du code `errno` .

**Exercice 24** Ajouter les prototypes de vos fonctions à `mini_lib.h`.

### 1.3 Gestion des Entrées/Sorties

Nous allons implémenter les 2 fonctions de lecture/écrire bufferisées équivalentes à `fread/fwrite`.

**Exercice 25** Créer un fichier `mini_io.c`.

**Exercice 26** Définir une variable globale `IOBUFFER_SIZE` à 2048.

**Exercice 27** Définir une structure `MYFILE` qui permettra de représenter un fichier ouvert et ses buffers de lecture/écriture :

- `int fd` : le descripteur de fichier obtenu via `open`
- `void * buffer_read` : le buffer contenant les données lues
- `void * buffer_write` : le buffer contenant les données à écrire
- `int ind_read` : l'index actuel dans le `buffer_read`, -1 au départ pour indiquer que le buffer n'est pas alloué
- `int ind_write` : l'index actuel dans le `buffer_write`, -1 au départ pour indiquer que le buffer n'est pas alloué

**Exercice 28** En utilisant l'appel système `open`, implémenter la fonction `MYFILE* mini_fopen (char* file, char mode)` qui ouvrira le fichier avec le mode spécifié et retournera une structure MYFILE correspondant à votre fichier. Le mode correspondra à

r ouverture en lecture
w ouverture en écriture
b ouverture en lecture/écriture
a ouverture en ajout à la fin du fichier

**Exercice 29** En utilisant l'appel système `read`, implémenter la fonction `int mini_fread (void* buffer,int size_element, int number_element, MYFILE* file)`. Cette fonction utilisera une lecture bufferisée, i.e. elle liera toujours `IOBUFFER_SIZE` éléments avec `read` et les stockera dans `buffer_read`. Lors de l'appel a `mini_fread`, ce sont les éléments de `buffer_read` qui seront recopiés dans `buffer`, un `read` ne sera déclenché que si l'index `ind_read` est arrivé à la fin de `buffer_read`.

`buffer_read` sera alloué au premier appel de `mini_fread`, d'une taille `IOBUFFER_SIZE` et l'index `ind_read` sera positionné à 0. Cet index sera incrémenté à chaque lecture.

Cette fonction renvoie -1 en cas d'erreur ou le nombre de caractères lus.

**Exercice 30** Ajouter le code à votre `main` pour tester cette fonction.

**Exercice 31** En utilisant l'appel système `write`, implémenter la fonction `int mini_fwrite (void* buffer,int size_element, int number_element, MYFILE* file)`. Cette

fonction recopiera les éléments dans `buffer_write` et mettra a jour l'index `ind_write`. L'écriture, via un `write`, ne sera déclenchée que lorsque le `buffer_write` sera plein.

`buffer_write` sera alloué au premier appel de `mini_fwrite`, d'une taille `IOBUFFER_SIZE` et l'index `ind_write` sera positionné à 0. Cet index sera incrémenté à chaque écriture.

Cette fonction renvoie -1 en cas d'erreur ou le nombre de caractères écrits.

**Exercice 32** Ajouter le code à votre `main` pour tester cette fonction.

**Exercice 33** Ajouter une fonction `int mini_fflush(MYFILE* file)` qui va forcer l'écriture (via un `write`) des données non-écrites présentes dans `buffer_write`. Cette fonction renvoie -1 en cas d'erreur ou le nombre de caractère écrit.

**Exercice 34** Que se passe-t'il si le programme se termine alors que le buffer d'écriture n'était pas plein ? Ajouter le code permettant de corriger ce problème dans `mini_exit` (Pensez a ajouter une liste des fichiers ouverts pour pouvoir tous les flusher)

**Exercice 35** Ajouter la fonction `int fclose(MYFILE* file)` qui ferme le fichier, i.e, le flush, le supprime de la liste, et utilise l'appel système `close` pour le fermer. Cette fonction retourne -1 en cas d'erreur.

**Exercice 36** Ajouter la fonction `int mini_fgetc(MYFILE* file)` qui renvoie un caractère lu, -1 en cas d'erreur.

**Exercice 37** Ajouter la fonction `int mini_fputc(MYFILE* file, char c)` qui écrit un caractère, -1 en cas d'erreur.

**Exercice 38** Ajouter le code à votre `main` pour tester cette fonction.

## 2 Commandes Système

En utilisant vos fonctions, implémenter les programmes suivants :

**Exercice 39** `mini_touch file` qui crée un fichier vide `file` si il n'existe pas.

**Exercice 40** Implémenter la commande `mini_cp src dest` qui réalise une copie du fichier `src` dans `dest`. Proposer une méthode pour benchmarker votre programme (et le comparer au résultat du TD précédent)

- `mini_echo chaine` qui affiche à l'écran la chaîne passée en paramètre (qui peut contenir des espaces !)
- `mini_cat` : prend en paramètre un fichier ; affiche son contenu
- `mini_head -n N` : prend en paramètre un fichier ; affiche les N premières lignes
- `mini_tail -n N` : prend en paramètre un fichier ; affiche les N dernières lignes
- `mini_clean` : prend en paramètre un fichier ; crée un fichier vide si il n'existe pas, ou le remet à zéro si il existe
- `mini_grep` : prend en paramètre un mot et un nom de fichier, affiche toutes les lignes contenant ce mot
- `wc` : prend en paramètre un nom de fichier, affiche le nombre de **mots** du fichier

**Nous implémenterons d'autres fonctions lors du prochain TP**

## 3 SHELL

En utilisant vos fonctions, implémenter les programmes suivants :

**Exercice 41** Créer un programme `mini_shell` qui va simuler un mini-shell. Une fois lancé, ce programme attendra sur l'entrée standard que l'on entre une commande.

Cette commande sera ensuite exécutée. Le processus père de votre mini-shell attendra alors, que le processus fils se termine, avant de demander, à nouveau, à l'utilisateur d'entrer une commande. Vous devez utiliser les appels système `fork` et `wait`

Si la commande correspond à `exit`, votre shell s'arrête. Si il y a un problème lors de l'exécution de la commande, vous afficherez l'erreur correspondante.

**Nous implémenterons d'autres fonctionnalités lors du prochain TP**

# 4 A rendre

Sur celene, une archive `TP_miniglibc.zip` contenant :
— README.txt : réponses aux questions, commentaires supplémentaires ...
— src/ : avec les fichiers .c/.h de votre TP
— Makefile : pour compiler le tout, si vous savez l'écrire