TP1 : mémoire, copie de fichier

Matthieu Boutier (boutier@pps.univ-paris-diderot.fr)

Attention ! !

— Vérifiez *systématiquement* la valeur de retour des fonctions, et faites ce qu'il faut (`perror(3), exit(3)` [1]).
— *compilez* vos programmes le plus souvent possible,
— *testez* les à chaque question.

1 Programmation système

1.1 Entrée et sortie standard

Les descripteurs de fichiers correspondant à l'entrée standard, la sortie standard et la sortie d'erreur sont respectivement désignés par les macros `STDIN_FILENO` (== 0), `STDOUT_FILENO` (== 1) et `STDERR_FILENO` (== 2).

1. Créez un programme `mycat.c`, avec une macro `BUFFER_SIZE` égale à 4096.
2. En utilisant `read(2)` et `write(2)`, écrivez un main qui lit une fois sur l'entrée standard, puis écrit ce qui a été lu sur la sortie standard (conseil : utilisez un tampon déclaré sur la pile.)
3. Répétez l'opération indéfiniment jusqu'à ce qu'une fin de fichier soit détectée, ou qu'une erreur se produise [2].
4. Testez votre programme en redirigeant l'entrée standard par un fichier texte : `./mycat < exemple.txt`.
5. Utilisez la commande `time` pour tester l'efficacité de votre programme. Comparez avec une taille de tampon de 1.

1.2 Copie de fichier

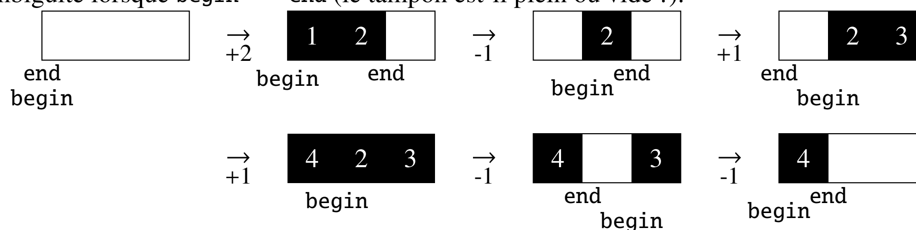
1. Créez un programme `mycp.c`, dont le main ouvre un fichier `entree.txt` en lecture, puis le ferme, respectivement avec `open(2)` et `close(2)` [3]. On appellera `fd_rd` le descripteur de fichier (pour *file descriptor read*) [4].
2. Ajoutez des instructions pour que votre fonction lise les `BUFFER_SIZE` premiers octets du fichier, et affiche le nombre d'octets lus.
3. Ajoutez encore du code pour ouvrir un fichier `sortie.txt` en écriture (avec `O_CREATE` et `O_TRUNC`, mode = 0644), puis le fermer.

1. Le "(3)" désigne le numero de page de manuel : on tapera `man 3 exit` pour avoir la page de manuel de la fonction `exit(3)`, et `man 2 read` pour celle de `read(2)`.
2. On pourra quitter le programme avec `^D` (fin de transmission) ou `^C` (fin de texte).
3. Fermer un descripteur de fichier est aussi important que de libérer une zone mémoire.
4. Si vous êtes curieux, vous pouvez afficher sa valeur.

4. N'affichez plus le contenu du fichier, mais écrivez les octets lus dans `sortie.txt`.
5. Mettez une boucle quelque part pour que votre programme copie tout le fichier.
6. Utilisez les arguments du programme (`argv`) pour les fichiers d'entrée et de sortie.
7. Testez votre programme avec une taille de tampon de 3, puis de 1 [5].
8. Nous aimerions garder le temps de dernière modification (après tout, une copie ne change pas le contenu). Utilisez les appels système `fstat(2)` et `futimes(2)` pour mettre le temps de dernière modification de `sortie.txt` égal à celui de `entree.txt`.

2 Un tampon circulaire

Parfois, les lectures et écritures peuvent être partielles, c'est-à-dire ne pas remplir ou vider totalement un tampon. Nous allons implémenter un tampon circulaire, avec la possibilité d'y ajouter ou d'en retirer un élément, comme illustré ci-dessous. Remarquez l'ambiguïté lorsque `begin == end` (le tampon est-il plein ou vide ?).



1. Créez un fichier `cbuffer.h` qui contient la structure ci-dessous, et un fichier `cbuffer.c` qui contiendra le code des questions suivantes (n'oubliez-pas d'ajouter les déclarations des fonctions dans le `.h` au fur et à mesure).

```
#define BUFFER_SIZE 3

struct cbuffer {
  char buffer[BUFFER_SIZE];
  char *begin; /* == end si le tampon est vide */
  char *end; /* == NULL si le tampon est plein */
};
```

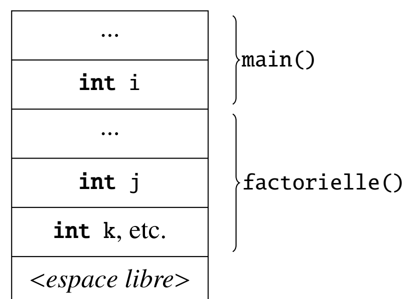
2. Écrivez une fontion `void cbuf_init(struct cbuffer *buf)` qui initialise le tampon `buf` passé en paramètres : on fera pointer `begin` et `end` au début du tampon. On pourra aussi initialiser le contenu du tampon à 0 (utile pour dégoguer).
3. Écrivez `int cbuf_putc(struct cbuffer *str, char c)` qui écrit l'octet `c` dans le tampon. Retourne 0 en cas de succès, -1 en cas d'échec.
4. Écrivez `int cbuf_getc(struct cbuffer *str)` qui retourne l'octet pointé par `begin`, et le supprime [6], ou -1 si le tampon est vide.
5. Écrivez `int cbuf_get_free_space(struct cbuffer *str)` qui renvoie le nombre d'octets libres dans le tampon.

5. Utile pour chasser des bugs potentiels.
6. On pourra écrire une espace ' ' à la place pour le debug, mais c'est facultatif.

6. Testez votre programme avec `BUFFER_SIZE` à 1. Si tout marche bien, vous pouvez mettre la valeur finale de la taille du buffer à 4096 : vous avez maintenant un beau tampon circulaire.
7. (A faire chez vous, si vous trouvez ça fun.) Il est bon de pouvoir écrire et lire plus d'un octet à la fois. Écrivez sur le même modèle `int cbuf_put(struct cbuffer *str, char *mem, int n` et `int cbuf_get(struct cbuffer *str, char *mem, int n` (on retournera le nombre d'octets effectivement lus ou écrits).

3 Allocation sur la pile, utilisation du débogueur

La pile est la zone mémoire où sont sockées les variables locales (et arguments des fonctions). On l'appelle pile... par ce que c'est une pile (par adresses décroissantes).




```c
#include <stdlib.h>

int factorielle(int j) {
    int k;
    int result = 1;
    long long_result = 1;
    for (k = 2; k <= j; k ++) {
        long_result *= k;
        result *= k;
        if (long_result != result)
            abort();
    }
    return result;
}

int main()
{
    int i;
    for (i = 0; i < 20; i ++)
        factorielle(i);
    return 0;
}
```

1. Compilez le programme précédent avec la commande habituelle : `gcc -g -Wall bug.c -o bug`. Exécutez-le. Oh, il ne marche pas, quelle surprise ! Heureusement, nous avons compilé avec `-g` : nous allons pouvoir le déboguer.

2. Utilisez `gdb` avec le nom de notre programme compilé : `gdb bug`. Il est maintenant chargé.
3. Tapez `break main` : à chaque fois que le programme passe par la fonction `main`, il s'arrêtera au début de celle-ci [7].
4. Tapez `run` pour lancer le programme. Vous pouvez exécuter les lignes de code une par une avec `next` (out `n`), et rentrer dans les fonctions appelées avec `step` (ou `s`).
5. Regardez la valeur des variables à portée avec `print`.
6. Bon, tout à l'air de bien se passer, allons directement à l'erreur : tapez `continue` (ou `c`).
7. Le programme s'est arrêté quelque part par un `SIGABRT`, dans une fonction au nom étrange (sur ma machine, `__kill()`. Pour voir où nous sommes dans le programme, regardons l'état de notre pile : tapez `backtrace` (ou `bt`). Voilà la liste des fonctions appelées, avec l'adresse de l'emplacement mémoire de leur code source.
8. Remontons la pile à l'aide de `up`, jusqu'à être dans le `main` (jusqu'à ce qu'il nous dise qu'il ne puisse plus rien faire).
9. `gdb` nous informe que le programme s'est arrêté dans le `main` à la ligne 20, pendant l'appel à `factorielle`. Nous pouvons savoir avec quelle valeur en utilisant `print i`. Affichez aussi l'adresse mémoire de i : `print &i`. Remarquez que les adresses du code des fonctions sont très éloignées de celles de la pile.
10. Redescendons dans notre pile avec `down` : nous sommes dans la fonction factorielle, qui a échouée au `abort`.
11. Regardez pourquoi `abort` a été appelée, et comprenez pourquoi en regardant les différentes valeurs des variables.
12. Regardez aussi l'adresse des variables, et comparez-les avec celle de `i`, qui était dans le main.

Sachez que `gdb` permet d'autres choses, comme d'inspecter des zones de mémoire (`x`), d'appeler des fonctions du programme (`call`), etc.

7. On peut aussi stopper le programme sur d'autres fonctions, ou à un numéro de ligne précis (`break 31`), etc.