Langage C et Programmation Système
TP n° 6 : Implémentation d'une bibliothèque d'entrée/sortie

Le but de ce TP : Jusqu'ici, vous avez utilisé les fonctions d'entrée/sortie de la bibliothèque standard du C, qui sont déclarées dans `stdio.h`. Elles ont l'avantage d'être disponibles sur n'importe quelle plateforme supportant le langage C. Dans ce TP, on va "descendre d'une couche d'abstraction". Vous allez écrire vos propres implémentations de certaines fonctions d'entrée/sortie en vous servant des appels système Unix correspondants, qui, eux, ne sont pas disponibles sur toutes les plateformes. Au final, vous pourrez remplacer `#include <stdio.h>` par `#include "myio.h"` (« my input/output ») dans vos programmes `myecho`, `mycp` et `mycat` des TPs n° 2 et 3, et utiliser vos propres fonctions.

Exercice 1 : Préparation – L'en-tête `myio.h`

1. Crééz un fichier d'en-tête `myio.h`, et définissez-y les macros suivantes (avec `#define`) :

`MY_NULL` de valeur `0` : L'état d'un pointeur qui n'a pas de cible (« null pointer »).

`MY_EOF` de valeur `(-1)` : Notre indicateur de fin de fichier (« End Of File »).

`MY_FOPEN_MAX` de valeur `64` : Le nombre maximal de fichiers ouverts simultanément.

`MY_BUFSIZ` de valeur `1024` : La taille d'un buffer (mesurée en nombre de caractères).

2. Toujours dans `myio.h`, définissez la structure `MY_FILE`, qui sera notre équivalent de la structure `FILE` de la bibliothèque standard (`typedef struct {...} MY_FILE;`). Ajoutez les champs suivants à `MY_FILE` :

`fd:` Le descripteur de fichier (« file descriptor »), un entier de type `int` qui permettra d'identifier le fichier représenté par la structure.

`flags:` Un entier de type `int` dans lequel on encodera les droits d'accès et d'autres informations sur le fichier. (Chaque information correspondra à un bit de cet entier.)

`buf:` Un pointeur de type `unsigned char*` qui pointera sur un buffer (une mémoire tampon) dans lequel on stockera temporairement une partie du fichier.

`pos:` Un pointeur de type `unsigned char*` qui désignera la position du prochain caractère à lire ou écrire dans le buffer.

`count:` Un entier de type `int` qui indiquera le nombre de caractères ou de places libres restant dans le buffer (ou une erreur retournée par un appel système).

3. Déclarez trois pointeurs représentant les flux standard :

```
extern MY_FILE* mystdin;
extern MY_FILE* mystdout;
extern MY_FILE* mystderr;
```

Le mot-clé `extern` permet de *déclarer* une variable globale sans la *définir*. Cela veut dire que l'on indique le type de la variable (ici `MY_FILE*`), mais on ne lui réserve pas de place dans la mémoire. La définition (qui réserve de la mémoire) se fera dans un autre fichier.

Pourquoi déclarer, mais pas définir, ces trois variables dans `myio.h` ? (Pensez à ce que fait `#include "myio.h"`.)

4. Finalement, déclarez les fonctions qui seront disponibles à l'utilisateur de votre bibliothèque d'entrée/sortie : `myfopen`, `myfclose`, `myfgetc` et `myfputc`. Les signatures seront presque les mêmes que celles des fonctions correspondantes de la bibliothèque standard (`fopen`, `fclose`, `fgetc` et `fputc`, respectivement), avec la seule différence qu'il faut remplacer `FILE` par `MY_FILE`.

Exercice 2 : Ouvrir et fermer un fichier

Afin de pouvoir facilement identifier les composantes internes de la bibliothèque, nous adoptons la convention suivante : Si une macro, variable globale ou fonction n'est pas destinée à l'utilisateur final (en l'occurrence vous-même), son nom commencera par un tiret bas, et elle ne sera pas déclarée dans `myio.h`.

1. Crééz un fichier d'implémentation `myio.c`, commençant par `#include "myio.h"` et les définitions suivantes :

```
#define _READ 1     // Fichier en mode lecture.
#define _WRITE 2    // Fichier en mode ecriture.
#define _NOBUF 4    // On n'utilise pas de buffer.
#define _EOF 8      // Fin de fichier atteinte.
#define _ERROR 16   // Une erreur s'est produite.
```

   Chacune de ces macros représente un bit de l'entier `flags` d'un `MY_FILE f`. On peut manipuler les bits d'un entier à l'aide des opérations logiques bit à bit. Ainsi, pour déterminer si `f` est en mode lecture, il suffit de tester si l'expression

```
f.flags & _READ                    ("et" logique bit par bit)
```

   est égale à une valeur non nulle. (La valeur est soit `0`, soit `_READ`.)
   Autre exemple : pour indiquer que la fin de fichier a été atteinte, on peut exécuter

```
f.flags = f.flags | _EOF;          ("ou" logique bit par bit)
```

   ou, plus succinctement, `f.flags |= _EOF;`.

2. Définissez et initialisez partiellement une variable globale `_files` qui représentera le tableau de toutes les structures de fichier disponibles à votre bibliothèque, comme ceci :

```
MY_FILE _files[MY_FOPEN_MAX] =
                  { {0, _READ, MY_NULL, MY_NULL, 0},
                    {1, _WRITE, MY_NULL, MY_NULL, 0},
                    {2, _WRITE | _NOBUF, MY_NULL, MY_NULL, 0} };
```

   Ainsi, les trois premiers éléments du tableau sont initialisés à des structures de fichier correspondant à l'entrée standard (descripteur de fichier `0`, mode lecture), la sortie standard (descripteur `1`, mode écriture) et l'erreur standard (descripteur `2`, mode écriture, sans buffer), respectivement. Comme `_files` est une variable globale, tous les bits des autres éléments du tableau sont automatiquement initialisés à `0`. (Ce ne serait pas le cas pour une variable locale d'une fonction.)
   Définissez et initialisez maintenant les pointeurs `mystdin`, `mystdout` et `mystderr` (que vous avez déjà déclarés dans `myio.h`), de façon à ce qu'ils pointent sur les bons éléments du tableau.

3. Écrivez la fonction `MY_FILE* myfopen(const char* path, const char* mode)`. L'argument `path` représente le chemin d'accès du fichier à ouvrir, et `mode` doit commencer par `'r'`, `'w'` ou `'a'`. (La signification est la même que pour `fopen`, mais vous n'aurez pas à traiter les cas `"r+"`, `"w+"` et `"a+"`.) Votre fonction devra chercher un `MY_FILE f` libre (non alloué, ce qui est indiqué par `f.flags`) dans le tableau `_files`, et utiliser l'appel système `open` (cf. `man 2 open`) pour ouvrir le fichier demandé par l'utilisateur. En mode `"w"` ou `"a"`, si le fichier n'existe pas encore, créez le avec les permissions `-rw-r--r--`. Il faudra penser à initialiser les champs de `f` de manière sensée, en particulier à sauvegarder le descripteur de fichier retourné par `open` et le droit de lecture ou écriture indiqué par l'argument `mode`. (N'allouez pas encore de mémoire pour le buffer ; les fonctions de l'exercice 3 s'en occuperont.) En cas de succès, `myfopen` retournera un pointeur sur `f`, sinon `MY_NULL`.
4. De manière analogue, implémentez la fonction `int myfclose(MY_FILE* fp)` qui ferme le fichier représenté par `*fp`, par l'intermédiaire de l'appel système `close`, et retourne `0` en cas de succès, sinon `MY_EOF`. N'oubliez pas d'indiquer dans `fp->flags` que la structure `*fp` est à nouveau libre. Si `fp->buf` pointe sur un buffer, il faudra aussi libérer cette mémoire (avec `free`).
   Remarque : À ce stade, votre implémentation de `myfclose` n'est pas encore complète. Dans l'exercice 3, on vous demandera d'insérer une ligne de code supplémentaire.

Exercice 3 : Remplir et vider un buffer

Comme déjà insinué dans les exercices 1 et 2, un objet de type `MY_FILE` va donner un accès indirect à un fichier. Au lieu de directement lire ou écrire dans le fichier, on va lire ou écrire dans le buffer associé à ce fichier. Cette approche permet de limiter le nombre d'appels système `read` et `write`, qui sont beaucoup plus lents que l'accès au buffer.
Si on est en mode lecture et que le buffer est vide, il va falloir le remplir à partir du fichier. Similairement, en mode écriture, un buffer plein va devoir être copié dans le fichier, puis vidé. (Notez que, n'ayant pas implémenté les modes `"r+"`, `"w+"` et `"a+"` de `fopen`, vos fichiers ne pourront pas être simultanément en mode lecture et écriture.) Les deux opérations de manipulation de buffer seront réalisées par les fonctions `_fillbuffer` et `_flushbuffer`.

1. Écrivez une fonction de signature `int _fillbuffer(MY_FILE* fp)` qui remplit le buffer de la structure `*fp` à partir du fichier identifié par le descripteur `fp->fd`, et, en cas de succès, retourne le premier caractère du buffer, sinon `MY_EOF`. L'appel système à utiliser est `read` (cf. `man 2 read`). Si le buffer n'existe pas encore, allouez un tableau de `MY_BUFSIZ` caractères, à condition que le bit `_NOBUF` de `fp->flags` soit `0` ; dans le cas contraire, allouez un tableau d'un seul caractère. (Servez-vous de la fonction `malloc`.) Pensez à actualiser les champs de `*fp` pour lesquels cela est nécessaire. Pensez aussi à vérifier le contenu de `fp->flags` ; dans quels cas faudra-t-il directement retourner `MY_EOF` ?
2. Similairement, écrivez une fonction `int _flushbuffer(int c, MY_FILE* fp)` qui copie dans le fichier correspondant (en se servant de l'appel système `write`) le contenu du buffer de `*fp`, concaténé avec `c` si `c` n'est pas `MY_EOF`, et "vide le buffer" (il suffit de déclarer qu'il est vide). La fonction doit retourner `c` en cas de succès, sinon `MY_EOF`, et, comme `_fillbuffer`, elle doit allouer le buffer s'il n'existe pas encore. Il pourrait s'avérer utile de "mentir légèrement" (de `1`) concernant la place restante dans le buffer.

3. Il est maintenant temps de compléter `myfclose` (cf. exercice 2, question 4). Que faut-il faire avant de fermer un fichier ?

Exercice 4 : Mise en service et test de la bibliothèque

Votre bibliothèque devrait être presque fonctionelle. Il suffit maintenant de donner à l'utilisateur un moyen de lire et écrire les fichiers. Dans un premier temps, pour faire simple, on va procéder caractère par caractère, ce qui demande seulement quelques lignes de code. (On fera appel à `_fillbuffer` et `_flushbuffer`.)

1. Implémentez la fonction `int myfgetc(MY_FILE* fp)` qui se comportera comme `fgetc`, son modèle dans la bibliothèque standard. (Évidemment, elle retournera `MY_EOF` au lieu de `EOF` en cas d'erreur ou si la fin de fichier a été atteinte.)

2. De manière analogue, implémentez la fonction `int myfputc(int c, MY_FILE* fp)`.

3. Il est enfin possible de tester votre bibliothèque. Téléchargez les fichiers `mycp.c` et `Makefile` que vous trouverez sur DidEL, dans le répertoire correspondant à ce TP, [1] et enregistrez-les dans votre répertoire de travail. Si votre implémentation est correcte, et que vous avez respecté les noms de fichier demandés dans ce sujet, un simple `make` suffira pour compiler votre bibliothèque, ainsi qu'une version adaptée du programme `mycp`. (À part pour les messages d'erreur, cette version utilise `myio.h` au lieu de `stdio.h`. Autrement, elle devrait être similaire à celle que vous avez écrit pour le TP n° 3.)

Essayez de copier un fichier avec `mycp`, puis d'ajouter le contenu d'un fichier à la fin d'un autre en utilisant l'option `--append` de `mycp`.

Exercice 5 : Pour aller un peu plus loin ...

1. Jouez avec la valeur de `MY_BUFSIZ`, et testez si elle a une influence sur la vitesse d'exécution de `mycp`. Pour cela, créez un grand fichier (de l'ordre de grandeur d'une centaine de méga-octets) avec la commande Unix `dd`, et mesurez le temps d'exécution de `mycp` à l'aide de la commande `time` (cf. les pages de manuel `dd(1)` et `time(1)`).

2. Implémentez vos propres versions des fonctions `fgets` et `fputs` :

```
char* myfgets(char* s, int size, MY_FILE* fp);
int myfputs(const char* s, MY_FILE* fp);
```

(N'oubliez pas d'actualiser votre fichier d'en-tête.)

3. Adaptez à `myio.h` un programme de votre choix que vous avez écrit pour l'un des derniers TPs (par exemple `mycat`). Utilisez `myfputs` au lieu de `printf` (à moins que vous vouliez aussi implémenter un `myprintf`, ce qui serait un peu plus ambitieux).

4. Rendez votre bibliothèque plus utile en ajoutant vos propres versions des fonctions auxiliaires `clearerr`, `feof`, `ferror` et `fileno`. (Elles sont très faciles à implémenter.) Écrivez aussi votre propre version de `fflush` en vous servant de `_flushbuffer`.

5. Implémentez les fonctionnalités de `ftell` et `fseek` dans les fonctions suivantes :

```
long myftell(MY_FILE* fp);
int myfseek(MY_FILE* fp, long offset, int whence);
```

Il faudra faire attention à bien coordonner `myfseek` avec le buffer. Pour tester vos fonctions, vous pourrez, par exemple, adapter le programme de l'exercice 3 du TP n° 3.

1. Alternativement, ces fichiers sont aussi disponibles localement sous `/ens/reiter/sysc/tp6`.