UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Master M1 Informatique, CSE
TD & TP Surcouche d'entrées / sorties
Année 2020-2021

Résumé

L'écriture d'une bibliothèque d'entrées / sorties reposes sur deux idées principales : d'une part minimiser le nombre d'appels système permettant d'effectuer la tâche voulue afin de maximiser la performance et d'autre part effectuer le travail de formatage des données en dehors du noyau, car il n'a pas besoin de privilèges particuliers. Les objectifs de cette séance de TD/TP sont donc les suivants :

— développer une bibliothèque d'entrées-sorties de niveau utilisateur fournissant plus de fonctionnalités que les appels systèmes mis à disposition par le noyau ;
— fournir un mécanisme de gestion de tampon, analogue à celui mis en œuvre dans les fonctions déclarées dans `stdio.h`, permettant de factoriser les appels système ;
— comprendre la fabrication et l'utilisation d'une bibliothèque statique ou dynamique.

Ce TP sera noté et est à rendre en binôme sur Moodle pour le 4 décembre 2020.

I. Entrée / Sorties avec tampon

L'objectif pour vous ici est de programmer la gestion d'un tampon d'entrées / sorties permettant de factoriser les appels système. L'idée générale est de faire en sorte que tous les accès à un fichier ouvert en lecture ou en écriture se fassent par le biais d'un tampon dont vous gèrerez l'allocation, la libération, le remplissage et le vidage. Un appel système ne sera nécessaire que lorsque :

— un accès en lecture à un fichier ouvert en lecture est effectué et le tampon est vide : dans ce cas il faut effectuer un appel système lisant un bloc de données d'au plus la taille du tampon pour essayer de remplir celui-ci au maximum ;
— un accès en écriture à un fichier ouvert en écriture est effectué et le tampon est plein : dans ce cas il faut effectuer un appel système écrivant le tampon dans le fichier sous jacent afin de libérer de la place pour l'écriture demandée.

Votre rôle est donc de définir une structure de données `FICHIER` et les fonctions associées en respectant une interface de programmation qui devra être respectée par les utilisateurs de votre bibliothèque d'entrées / sorties. Cette interface vous est imposée et ne comprend dans un premier temps que les 4 fonctions suivantes :

— `FICHIER *ouvrir(char *nom, char mode);`
ouvre un fichier dont le `nom` est passé comme premier paramètre. Le second paramètre indique le `mode` d'ouverture du fichier. Il peut être ouvert en lecture (`mode == 'L'`) ou en écriture (`mode == 'E'`). Aucun autre mode n'est accepté. Renvoie un pointeur sur un objet de type `FICHIER`. La fonction `ouvrir` renvoie `NULL` si le fichier ne peut pas être ouvert.
Votre travail ici est d'allouer la structure de type `FICHIER` contenant un tampon et un descripteur de fichier obtenu à l'aide de la fonction `open`.
— `int fermer(FICHIER *f);`
ferme le fichier pointé par `f` (qui est un pointeur renvoyé par `ouvrir`). La structure de donnée allouée à l'ouverture pourra être soit libérée soit réutilisée pour un autre fichier.
— `int lire(void *p, unsigned int taille, unsigned int nbelem, FICHIER *f);`
lit `nbelem` éléments de données tenant chacun sur `taille` octets, depuis le fichier pointé par `f` et les stocke à l'emplacement mémoire pointé par `p`. La fonction `lire` retourne le nombre d'éléments lus. Le fichier doit avoir prélablement été ouvert en mode `'L'`. Cette fonction devra lire depuis le tampon contenu dans `f` en remplissant ce tampon à l'aide d'un `read` uniquement en cas de besoin.
— `int ecrire(void *p, unsigned int taille, unsigned int nbelem, FICHIER *f);`
écrit `nbelem` éléments de données tenant sur `taille` octets stockés à l'emplacement mémoire pointé par `p` dans le fichier pointé par `f`. La fonction `ecrire` retourne le nombre d'éléments écrits. Le fichier doit avoir été ouvert en mode `'E'`. De manière analogue au cas de `lire`, les données devront être écrites dans le tampon contenu dans `f` qui ne devra être lui-même écrit à l'aide d'un `write` qu'en cas de besoin.

Voici un petit exemple d'utilisation de la bibliothèque. Cet exemple ouvre deux fichiers dont les noms sont passés en paramètre. Ce programme copie le contenu du premier fichier dans le second.

```
#include <unistd.h>
#include "stdes.h"

int main (int argc, char **argv)
{
  FICHIER *f1;
  FICHIER *f2;
  char c;

  if (argc != 3)
      exit (-1);
  f1 = ouvrir (argv[1], 'L');
  if (f1 == NULL)
      exit (-1);
  f2 = ouvrir (argv[2], 'E');
  if (f2 == NULL)
      exit (-1);

  while (lire (&c, 1, 1, f1) == 1)
    {
      ecrire (&c, 1, 1, f2);
    }
  fermer (f1);
  fermer (f2);
}
```

II. Entrées-Sorties formatées

Nous souhaitons maintenant compléter cette interface avec des fonctions d'entrées-sorties formatées (du type `fprintf` ou `fscanf`). Une des particularités de ces fonctions est qu'elles ont un nombre variable de paramètres. Vous utiliserez pour cela les fonctions `va_start`, `va_end` et `va_arg` dont les prototypes se trouvent dans `stdarg.h`. Notre interface va être étendue avec deux nouvelles fonctions ;

```
int fecriref (FICHIER *fp, char *format, ...);
int fliref (FICHIER *fp, char *format, ...);
```

Le contenu des formats est similaire à ceux que vous utilisez avec les fonctions classiques de `<stdio.h>`. Trois types de données vont pouvoir être manipulés.
— caractère : %c
— chaîne : %s
— entier : %d
Voici un petit exemple illustrant l'utilisation de la fonction `fecriref`.

```
#include <unistd.h>
#include "stdes.h"

int main (int argc, char **argv)
{ FICHIER *f1, *f2;
  if (argc != 2) exit (-1);
  f1 = ouvrir (argv[2], 'E');
  if (f1 == NULL) exit (-1);

  fecriref (f1, "  %c  %s 12\n", 'a', "bonjour");
  fecriref (f1, " %d \n", -1257);

  fermer (f1);
}
```

L'exécution du programme génère un fichier dont le contenu est le suivant :

```
bash$ test_format resultat
bash$ cat resultat
  a  bonjour 12
```

```
 -1257
bash$
```

Pour implémenter ces deux nouvelles fonctions, vous pourrez (devrez) vous servir des fonctions `lire` et `ecrire` définies dans la partie précédente.

III. Manipulation de bibliothèques

La dernière partie de cette fiche est dédiée à la génération d'une bibliothèque statique et d'une bibliothèque dynamique. Une bibliothèque statique est reliée au programme exécutable pendant la phase d'édition de lien, c'est à dire avant l'exécution du programme. Une bibliothèque statique est habituellement stockée dans un fichier ayant comme extension `.a`. L'avantage d'une bibliothèque est qu'il n'est plus nécessaire de spécifier la liste des modules objets. Dans le cas de ce TP, la bibliothèque n'est constituée que d'un seul fichier objet mais une bibliothèque peut être constituée à partir d'un nombre arbitraire de fichiers objets. L'inconvénient d'une bibliothèque statique est que le programme exécutable contient la bibliothèque et donc que la taille du programme final peut être importante. Voici la séquence de commande utilisée pour générer une bibliothèque statique :

```
bash$ gcc -c stdes.c
bash$ ar q libstdes.a stdes.o
bash$ gcc -c test_format.c
bash$ gcc -o test_format test_format.o -L. -lstdes
```

La commande `ar` permet de manipuler des bibliothèques statiques. (`man ar` pour obtenir plus d'informations). L'option `-L` permet d'indiquer à `gcc` dans quel répertoire(s) chercher les bibliothèques. L'option `-l` permet de spécifier un nom de bibliothèque statique à intégrer au programme durant l'édition de liens. On ne spécifie ni le préfixe `lib`, ni l'extension `.a`.

A l'inverse, une bibliothèque partagée est reliée au programme au moment de son exécution. Le code de la bibliothèque n'est donc pas inclus dans le fichier du programme exécutable. Un des avantages est que si la bibliothèque venait à être modifiée (correction de bug par exemple), la nouvelle version serait prise en compte sans qu'il n'y ait d'édition de lien à refaire sur les programmes utilisant cette bibliothèque partagée. Cette caractéristique peut aussi être un inconvénient si la signature des fonctions de la bibliothèque est changée : dans ce cas il faut corriger et recompiler les programme qui l'utilisent afin qu'ils fonctionnent. En outre une bibliothèque partagée doit être installée dans le système pour qu'un programme qui l'utilise puisse s'en servir.

La commande `ldd` affiche les bibliothèques partagées nécessaires pour l'exécution d'un programme. La variable d'environnement `LD_LIBRARY_PATH` permet de spécifier une liste de répertoires dans lesquels sont recherchées les bibliothèques partagées. Sur l'exemple suivant, nous avons ajouté le répertoire de travail comme répertoire où devront être recherchées les bibliothèques partagées.

```
bash$ gcc -c stdes.c
bash$ gcc -shared -o libstdes.so.1 stdes.o
bash$ gcc -c test_format
bash$ gcc -o test_format test_format.o libstdes.so.1
bash$ test_format resultat
test_format: error while loading shared libraries:
libstdes.so.1: cannot open shared object file: No such file or directory
bash$ ldd test_format
linux-gate.so.1 =>  (0xffffe000)
libstdes.so.1 => not found
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e63000)
/lib/ld-linux.so.2 (0xb7fc6000)
bash$ export LD_LIBRARY_PATH=.
bash$ ldd test_format
linux-gate.so.1 =>  (0xffffe000)
libstdes.so.1 => ./libstdes.so.1 (0xb7faf000)
libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7e4f000)
/lib/ld-linux.so.2 (0xb7fcd000)
bash$ test_format resultat
bash$ cat resultat
  a  bonjour 12
 -1257
bash$
```