TD3 – Bibliothèques statiques et dynamiques

1 Objectif

Cet exercice introduit aux différents types de *bibliothèques binaires*, qui sont des collections de fichiers-objets (.o) regroupés en une seule entité. On laisse alors à l'éditeur de liens le soin de trouver dans la bibliothèque les modules contenant les fonctions dont il a besoin : c'est la *résolution des références externes*.

LINUX (de même que la plupart des UNIX modernes) connaît deux types de bibliothèques, statiques (extension .a) et dynamiques (.so, comme *shared object*), et trois manières d'éditer les liens : statiquement avec chargement statique, statiquement avec chargement dynamique automatique, dynamiquement avec chargement dynamique explicite. Cet exercice explore les différentes possibilités.

1.1 Le code fourni

Pour cet exercice nous allons réutiliser les fichiers du Cache (TD 2). Vous trouverez dans le répertoire `Shared_Lib` les fichiers correspondant à la gestion générale du cache. Ils sont fournis sous la forme de fichiers-objets (.o), à l'exception des fichiers d'en-têtes (.h) et du programme principal (`main_Cache.c`). Cet ensemble de fichiers est complété par les quatre fichiers-sources, correspondant aux quatre stratégies de remplacement de bloc, développés au cours du TD 2.

Avec la Makefile fournie dans `Shared_Lib`, si vous exécutez la commande `make` sans paramètre, vous allez reconstruire les quatre exécutables correspondant aux quatre stratégies. Le but de cet exercice est de construire avec ces fichiers objets différentes bibliothèques et pour cela, vous allez modifier la Makefile du répertoire `Shared_Lib` (c'est donc aussi l'occasion d'améliorer la connaissance de `make` et de ses Makefiles). Les commentaires de cette Makefile préfixés d'un double dièse (##) indiquent comment faire ces modifications.

Attention Pour ce TD, utilisez les fichiers .o fournis dans le répertoire `Shared_Lib`, *pas ceux qui étaient fournis avec le sujet du Cache (TD 2)*. De même, les fichiers-sources qui implémentent les stratégies doivent être recompilés avec la Makefile de `Shared_Lib`. (cf. § 3.2 relatif aux options de compilation).

2 Bibliothèques et édition de liens statiques

2.1 Introduction aux bibliothèques statiques

Lors de la compilation d'un programme C, chaque fichier-source (.c) est compilé séparément, produisant un fichier-objet (.o). Puis l'éditeur de liens prend tous les fichiers-objets et les regroupe en un seul fichier binaire exécutable.

Cette phase d'*édition de liens* est cruciale. Elle permet d'opérationnaliser la communication entre les différents fichiers constituant le programme : certaines variables sont définies (allouées et initialisées) dans un fichier et utilisées dans d'autres (variables externes) ; de même certaines fonctions peuvent avoir leur corps défini dans un fichier alors qu'elles sont invoquées dans d'autres (fonctions externes). On parle de *symboles externes*, ou de *références externes*. L'établissement des correspondances entre définition et utilisation de ces symboles constitue la *résolution des références*, le rôle principal de l'éditeur de liens.

Sous UNIX, l'éditeur de liens est une commande nommée `ld` (comme *loader*), mais il est rare que vous ayez à l'utiliser directement. Ainsi la commande de compilation C (`cc` ou `gcc`) invoque-t-elle directement l'éditeur de liens si on lui donne des `.o` comme arguments :

```
gcc -o prog.exe main.o prog1.o prog2.o prog3.o
```

Ceci construit l'exécutable `prog.exe` en éditant les liens entre les quatre fichiers `.o` mentionnés.

Cependant, la discipline de programmation modulaire fait en sorte que, pour des applications conséquentes, le nombre de fichiers `.o` à lier ensemble peut devenir considérable, et qu'il peut même devenir humainement impossible de déterminer ceux qui sont réellement utilisés.

2.2 Création de bibliothèques statiques (archives)

La solution est donc de regrouper plusieurs `.o` dans un fichier unique, une bibliothèque. La commande pour ce faire est nommée `ar` :

```
ar -r libprog.a prog1.o prog2.o prog3.o
```

Ceci crée la bibliothèque `libprog.a` (il est d'usage de faire commencer le nom d'une bibliothèque par le préfixe `lib`) et lui donne les trois fichiers `.o` désignés comme contenu.

L'option `-r` (*replacement*) permet de créer la bibliothèque avec le contenu indiqué, mais aussi de remplacer dans une bibliothèque existante certains des `.o` par de nouvelles versions.

Il est pratique de doter la bibliothèque d'un index des références qui y sont définies ou utilisées, afin de faciliter la tâche de l'éditeur de liens. Ceci est le rôle de l'utilitaire `ranlib`, dont l'utilisation est très simple :

```
ranlib libprog.a
```

La création d'une *bibliothèque statique* est donc en général constituée d'un appel à `ar`, suivi d'un appel à `ranlib`.

2.3 Edition de liens avec une bibliothèque statique

Une fois la bibliothèque ainsi créée, on peut l'utiliser dans une commande d'édition de liens :

```
gcc -o prog.exe main.o libprog.a
```

si le fichier `libprog.a` est dans le répertoire courant : ou bien :

```
gcc -o prog.exe main.o -lprog
```

si `libprog.a` est dans *le chemin de recherche des bibliothèques*, un ensemble de répertoires (typiquement `/lib`, `/usr/lib`, `/usr/local/lib`, etc.) prédéfini à la configuration du système (l'option `-lxxx` recherche une bibliothèque nommée `libxxx.a` dans le chemin en question) ; ou encore :

```
gcc -o prog.exe main.o -L/users/jpr/lib -lprog
```

si on souhaite ajouter (option `-L`) le répertoire `/users/jpr/lib` au chemin de recherche susmentionné.

Ce type de bibliothèques est dit *statique*, comme l'édition de liens associée. On peut donc dire que le mécanisme est *doublement statique* : d'une part, les références sont résolues statiquement ;

d'autre part, le code des `.o` sélectionnés dans la bibliothèque est copié directement dans l'exécutable.

Remarque Il y a une différence fondamentale entre l'édition de liens directement avec des `.o`

```
gcc -o prog.exe main.o prog1.o prog2.o prog3.o
```

et l'édition de liens avec une bibliothèque

```
gcc -o prog.exe main.o libprog.a
```

Dans le premier cas, tous les `.o` sont en quelque sorte concaténés et leur somme forme l'exécutable. Dans le second cas, l'éditeur de liens extrait de la bibliothèque uniquement les `.o` qui sont réellement utilisés par le programme. L'utilisation d'une bibliothèque peut donc contribuer à réduire la taille des exécutables. Par ailleurs une bibliothèque peut avoir un contenu vaste, qui dépasse les besoins de telle application particulière.

2.4 Commandes de manipulation des bibliothèques statiques

La commande `ar` a de nombreuses options. En particulier, la commande `ar -t` permet de connaître le contenu d'une bibliothèque (la liste des `.o` qui la composent).

La commande `nm` (*namelist*) appliquée à une bibliothèque donne la liste des symboles qui y sont définis ou utilisés.

2.5 Création et utilisation de bibliothèques statiques pour le projet Cache

Modifiez la Makefile fournie dans le répertoire `Shared_Lib` afin de créer *quatre bibliothèques*, une pour chaque stratégie de remplacement de bloc, que vous nommerez de la manière suivante : `libCache_XXX-static.a`, où `XXX` désigne la stratégie : `XXX` est `RAND`, `FIFO`, `LRU` ou `NUR` (ces noms, de même que tous ceux qui suivent, sont ceux qui sont définis dans la Makefile fournie, respectez-les !).

Chaque bibliothèque doit contenir la gestion générale du cache (fichier `cache.o`), la gestion de la stratégie correspondante (`XXX_strategy.o`) et éventuellement, si nécessaire, la gestion de liste (`cache_list.o`).

- Utilisez ces quatre bibliothèques pour régénérer quatre exécutables de test nommés `tst_Cache_XXX-static.exe`.
- Vérifiez que les résultats d'exécution de ces fichiers sont identiques à ceux des fichiers générés sans bibliothèque (`tst_Cache_XXX.exe`).
- Comparez également leur taille à celle de ces exécutables générés sans bibliothèque (`ls -l`). Ces tailles sont-elles fondamentalement différentes ? La commande `size` vous permettra de déterminer dans quel(s) segment(s) se fait l'éventuel gain de taille. Dans le résultat de cette commande, `text` désigne la taille du segment de texte, `data` celle du segment de données initialisées, et `bss` celle du segment de données non initialisées. Donc, la taille totale du segment de données est `data + bss` et correspond à l'ensemble des variables externes et statiques (à l'exclusion évidente de ce qui sera alloué lors de l'exécution par `malloc()`, `sbrk()`, etc.).
- Vérifiez que le code des fonctions de `cache_list.c` n'a pas été inclus dans les exécutables correspondant aux stratégies qui n'en ont pas besoin (contrairement à ce qui se passe pour les exécutables générés sans bibliothèque). Pour cela, vous pouvez utiliser, par exemple, une commande shell comme

```
nm un_des_exécutables | grep Cache_List
```

3 Bibliothèques dynamiques et édition de liens statique

3.1 Motivation pour les bibliothèques dynamiques

Les bibliothèques statiques ont l'avantage de proposer une intégration sélective des fichiers objets à lier à l'exécutable. Cependant elles souffrent de plusieurs inconvénients :

- La taille des fichiers binaires exécutables est importante, à cause de la copie du code.
- La taille de l'espace mémoire des processus est également importante. Aucun partage de mémoire n'est possible entre des processus exécutant des programmes différents, mais utilisant la même bibliothèque statique. Certes, il s'agit de mémoire « virtuelle », mais il faudra quand même bien la ranger quelque part !
- Si la bibliothèque change de version, il faut refaire l'édition de liens de tous les exécutables qui lui sont liés, et ceci même si l'interface de la bibliothèque (prototypes des fonctions, types des variables externes) n'a pas changé.

Pour pallier ces inconvénients, on a inventé il y a longtemps (pratiquement depuis la mémoire virtuelle) la notion de *bibliothèque partagée*, appelée aussi sous UNIX (fichier-)objet partagé (d'où son extension `.so`, comme *shared object*). Le code d'une telle bibliothèque n'est pas recopié dans le binaire exécutable, mais partagé (sur disque comme en mémoire) entre tous les exécutables qui l'utilisent. Dès qu'un processus s'exécute qui nécessite la bibliothèque, cette dernière est chargée en mémoire (si elle n'y est pas déjà). D'où l'autre nom de *bibliothèque (à chargement) dynamique* (*Dynamic Loadable Library* de Windows, `.dll`).

Le point clé est de savoir comment sont résolues les références dans le cas des bibliothèques partagées : elles peuvent l'être statiquement (à l'édition de liens, avant l'exécution), ou dynamiquement (lors de l'exécution). Nous traitons le premier cas dans cette section, et le second dans la section suivante.

3.2 Création d'une bibliothèque dynamique

Alors que pour créer une bibliothèque statique, il suffit de produire des fichiers-objets sans option particulière, ce n'est pas le cas ici. Pour créer une bibliothèque dynamique, il faut d'abord compiler chaque fichier-source avec l'option `-fpic` ou `-fPIC`, comme dans :

```
gcc -c -fpic -ansi -pedantic -Wall prog1.c
```

Puis la bibliothèque elle-même est créée par une commande d'édition de liens spéciale :

```
gcc -shared -Wl,-soname,libprog.so -o libprog.so \
    prog1.o prog2.o prog3.o
```

man gcc (ou **info gcc**) vous dévoilera les dessous de cette formule magique...

3.3 Edition de liens statique

Une fois la bibliothèque créée, on peut l'utiliser dans une commande d'édition de liens statique, de la même manière que s'il s'agissait d'une bibliothèque statique. C'est-à-dire :

```
gcc -o prog.exe main.o libprog.so
```

si le fichier `libprog.so` est dans le répertoire courant ; ou bien :

```
gcc -o prog.exe main.o -lprog
```

si `libprog.so` est dans le chemin de recherche des bibliothèques ; ou encore :

```
gcc -o prog.exe main.o -L/users/jpr/lib -lprog
```

si on souhaite ajouter (option `-L`) le répertoire `/users/jpr/lib` au chemin de recherche susmentionné.

Les références sont alors résolues statiquement, et le fichier exécutable contient le résultat de cette résolution, mais *pas* le code correspondant. Celui-ci sera chargé dynamiquement.

3.4 Chargement dynamique automatique de bibliothèques dynamiques

Lorsqu'on exécute un programme lié à une bibliothèque statique, l'exécutable contient toute l'information. Ce n'est pas le cas avec une bibliothèque dynamique : il faut que l'on sache, lors du lancement du processus, où trouver le fichier `.so` pour pouvoir le charger. La recherche de ce fichier se fait de plusieurs manières :

- le système connaît un certain nombre de répertoires prédéfinis où chercher les `.so` (`/lib`, `/usr/lib`, `/usr/local/lib`, etc.) ;
- sous LINUX, on peut configurer ce chemin de recherche (utilitaire `/etc/ldconfig` et fichier de configuration `/etc/ld.so.conf`) mais cela nécessite des droits d'administrateur ;
- enfin, chaque utilisateur peut définir son propre chemin de recherche des bibliothèques grâce à la variable d'environnement `LD_LIBRARY_PATH`, qui est une suite de répertoires séparés par deux-points. Par exemple

```
export LD_LIBRARY_PATH=.:~/lib
```

Les répertoires indiqués seront examinés avant les répertoires système (ceux établis des deux premières manières).

Définissez votre variable d'environnement `LD_LIBRARY_PATH` de telle sorte qu'elle contienne au moins le répertoire courant.

3.5 Commandes de manipulation de bibliothèques dynamiques

La commande `nm`, permet de connaître les symboles définis et utilisés dans une bibliothèque dynamique (en fait `nm` fonctionne pour tout fichier-objet).

La commande `ldd` appliquée à un binaire exécutable liste les bibliothèques dynamiques dont cet exécutable dépend. Vous constaterez que, par défaut, tous les exécutables que vous produisez dépendent d'un certain nombre de bibliothèques dynamiques prédéfinies, dont la bibliothèque standard C (`/lib/libc.so`) et le *chargeur dynamique* (`/lib/ld-linux.so`). Ce dernier a pour charge de gérer les bibliothèques dynamiques utilisées par l'exécutable.

3.6 Création et utilisation de bibliothèques dynamiques pour le projet Cache

Faites exactement ce que vous avez fait précédemment (§ 2.5), mais en utilisant des bibliothèques dynamiques au lieu de bibliothèques statiques. Vous nommerez ces bibliothèques dynamiques `libCache_XXX-dynamic.so`, où `XXX` est le nom de la stratégie.

Attention, lors de l'exécution des programmes de test, vérifiez bien la valeur de votre variable d'environnement `LD_LIBRARY_PATH`.

4 Bibliothèques dynamiques et édition de liens dynamique

4.1 Motivation pour le chargement dynamique explicite de bibliothèques

Dans l'utilisation des bibliothèques dynamiques de la section précédente, la résolution des références restait statique. Il y avait un lien très fort entre la bibliothèque et l'exécutable. On pourrait souhaiter que la résolution des références soit elle-même dynamique. Cela permettrait d'avoir plusieurs bibliothèques de fonctionnalités semblables (par exemple d'interfaces voisines

voire identiques) mais d'implémentations différentes. On aimerait alors être capable de choisir une de ces bibliothèques à l'exécution, et de la changer dynamiquement.

C'est précisément le cas dans le projet de Cache : quatre bibliothèques (`libCache_XXX-dynamic.so`), implémentent de manières différentes la même interface, définie dans `strategy.h`. LINUX (et les UNIX modernes en général) permettent que la résolution des références se fasse à l'exécution, à charge au programmeur de charger et de résoudre *explicitement* les références qui l'intéressent dans son programme. Le système fournit pour cela une *bibliothèque générique de chargement dynamique* : `/lib/libdl.so`.

4.2 Création de bibliothèques dynamiques

La procédure est en tout point identique à celle décrite précédemment (§ 3.2) : la même bibliothèque dynamique peut être chargée aussi bien automatiquement qu'explicitement.

4.3 Edition de liens et chargement dynamique explicite de bibliothèques

Une fois la bibliothèque créée, on peut l'utiliser dans une commande d'édition de liens avec chargement dynamique explicite de la manière suivante :

```
gcc -o prog.exe main.o -ldl
```

Notez que cette commande d'édition de liens ne comporte aucune indication des bibliothèques qui seront explicitement et dynamiquement chargées. ***L'exécutable produit n'en dépendra donc pas.*** La seule bibliothèque ici est celle de chargement générique, `libdl.so`.

C'est donc au programmeur qu'il appartient de désigner, charger, et même résoudre les références. La bibliothèque `libdl.so` lui fournit pour cela quatre fonctions, dont le type est déclaré dans le fichier d'en-tête standard `<dlfcn.h>` (qu'il convient donc d'inclure) :

```
void *dlopen(const char *filename, int flag);
```

Ouvre la bibliothèque dynamique de nom `filename`, et retourne un pointeur (`plib`) qui permettra de référencer cette bibliothèque dans `dlclose()` et `dlsym()`. Si la bibliothèque ne peut être ouverte, le pointeur retourné est `NULL`. Une bonne valeur pour `flag` est `RTLD_LAZY` (voir `man dlopen` pour d'autres possibilités). La bibliothèque est recherchée de la même manière qu'en chargement automatique (§ 3.4), donc attention à `LD_LIBRARY_PATH`.

```
void dlclose(void *plib);
```

Ferme la bibliothèque représentée par `plib` qui doit donc être le résultat d'un `dlopen()` précédent.

```
void *dlsym(void *plib, const char *symbol_name);
```

Retourne un pointeur sur l'entité de nom `symbol_name` (ce peut être le nom d'une fonction ou d'une variables externe). Le type du pointeur retourné est `void *`; il appartient donc au programmeur de connaître le type exact de cette entité et d'effectuer le *cast* nécessaire.

```
char *dlerror();
```

Si la valeur de retour n'est pas le pointeur `NULL`, c'est un message indiquant la cause de l'erreur dans le dernier appel à l'une des quatre fonctions de `libdl.so`.

La page de `man` d'une quelconque de ces fonctions (par exemple, `man dlopen`) vous donnera tous les détails nécessaires, ainsi qu'un exemple d'utilisation.

4.4 Chargement explicite de bibliothèques dynamiques pour le projet Cache

Modifiez la Makefile pour créer quatre bibliothèques dynamiques de nom `libStrategyxxx.so`, où `xxx` est le nom de la stratégie. Chaque bibliothèque contient le fichier-objet de stratégie (`xxx_strategy.o`) et éventuellement, si nécessaire, `cache_list.o`. *Attention, ces bibliothèques ne contiennent pas* `cache.o`, contrairement aux cas précédents, mais uniquement ce qui dépend de la stratégie considérée.

Copiez le programme de test `main_Cache.c` dans `main_Cache-loadable.c`. Modifiez ce fichier pour que le nom de la stratégie puisse être passé en paramètre sur la ligne de commande. Le traitement de ce paramètre consistera à appeler une fonction `Load_Strategy(strat)`, où `strat` est le nom en question. Cette fonction `Load_Strategy()` devra être définie dans un nouveau fichier-source, `load_strategy.c`. Elle aura pour rôle de :

- charger dynamiquement la bibliothèque correspondant au nom de la stratégie,
- résoudre les références, grâce à `dlsym()`, à toutes les fonctions d'interface de la stratégie déclarées dans `cache.h` : `Strategy_Create()`, `Strategy_Close()`, etc.

Pour ne pas avoir à modifier `cache.c`, vous définirez, toujours dans `load_strategy.c`, toutes les fonctions d'interface de `cache.h`, mais en leur donnant un corps qui est un simple relais d'appel de la « vraie » fonction (celle de la bibliothèque dynamique), à travers le pointeur que vous a renvoyé `dlsym()`.

Modifiez la Makefile pour créer un exécutable unique, `tst_Cache-loadable.exe`, de cette manière :

```
gcc -rdynamic -o tst_Cache-loadable.exe \
main_Cache-loadable.o cache.o load_strategy.o -ldl
```

L'option `-rdynamic` permet à une bibliothèque chargée explicitement, de chercher ses références non résolues dans l'exécutable lui-même¹. Cette option est indispensable ici, car les bibliothèques de stratégie utilisent `Get_Free_Block()`, définie dans `cache.o`.

Examinez la taille de l'exécutable produit, et comparez-la à celle des exécutables obtenus précédemment. Le résultat peut vous paraître paradoxal. Essayez de le justifier et de l'analyser.

Déterminez, grâce à `ldd`, les bibliothèques dont dépend votre exécutable. Vous ne devez pas y trouver les bibliothèques spécifiques de stratégies (`libStrategy-xxx.so`).

Enfin exécutez le fichier obtenu en lui passant le nom des stratégies en paramètre :

```
% tst_Cache-loadable.exe NUR
... résultat de l'exécution avec stratégie NUR ...
% tst_Cache-loadable.exe LRU
... résultat de l'exécution avec stratégie LRU ...
% etc.
```

¹ Lorsqu'une bibliothèque est chargée, les autres bibliothèques dont elle dépend (i.e. dans lesquelles elle a des références à résoudre) le sont aussi. En revanche, par défaut, la bibliothèque ne résout pas de référence dans l'exécutable qui l'a chargée. L'option `-rdynamic` permet cette sorte de rétro-édition de liens.

5 Pointeurs sur fonctions

5.1 Notions de pointeur sur fonction

Soit une fonction dont le prototype est

```
int myfunc(int, double);
```

Supposons que l'on veuille un pointeur sur cette fonction. Le type de ce pointeur devra être *pointeur sur une fonction à deux arguments de types respectifs int et double et retournant un int*. La déclaration d'un objet de ce type sera en C :

```
int (*ptrfunc)(int, double);
```

On pourra initialiser cette variable lors de sa déclaration :

```
int (*ptrfunc)(int, double) = &myfunc;
```

ou lors d'une affectation :

```
ptrfunc = &myfunc;
```

Notez que l'opérateur « adresse-de » (&) n'est indispensable dans aucune des deux instructions précédentes : un nom de fonction, utilisé dans une expression, est assimilé à une valeur de type pointeur sur la fonction — c'est la même règle que pour les noms de tableaux.

On peut appeler la fonction à travers le pointeur, par :

```
int i = (*ptrfunc)(3, 5.75);
```

ou, plus simplement, par

```
int i = ptrfunc(3, 5.75);
```

5.2 Syntaxe des déclarations de pointeur sur fonction

Mais comment diable décoder une déclaration aussi abstruse que

```
int (*ptrfunc)(int, double);
```

Il suffit de la lire de la manière suivante, pas à pas :

- `ptrfunc` est un *pointeur*... car on peut lui appliquer l'opérateur d'indirection `*` ;
- le résultat `(*ptrfunc)` est une *fonction*... puisqu'on peut lui appliquer une paire de parenthèses avec une liste d'arguments ; *la fonction pointée prend deux paramètres, un int et un double* ;
- enfin le résultat de cette application, `(*ptrfunc)(int, double)`, est un `int`, donc ... *la fonction pointée retourne un int*.

La syntaxe de déclaration en C est donc la *syntaxe d'usage*. C'est déjà vrai pour une déclaration simple comme `int i;` qui peut se lire « `i`, utilisé dans une expression, est un `int` ».

De la même manière, dans

```
void (*pf1)();
void (*(pf2)(int))();
```

`pf1` est déclaré être un pointeur sur une fonction sans argument ni résultat et `pf2` est un pointeur sur une fonction à un argument entier qui retourne un pointeur du même type que `pf1` !

L'utilisation de quelques `typedef` peut éclaircir les choses :

```
typedef void (*PtrFunc1_type)();
typedef PtrFunc1_type (*PtrFunc2_type)(int);
void (*f(int))(); /* une fonction pointable par un PtrFunc2_type */
PtrFunc1_type g(int); /* une autre fonction pointable par un PtrFunc2_type */
PtrFunc2_type pf2 = f;
PtrFunc1_type pf1 = f(3);
pf1();
```

5.3 Transtypage de pointeurs sur fonction

On peut aussi transtyper un pointeur de ou vers un type pointeur sur fonctions (et vous aurez à le faire lors de l'utilisation de `dlsym()` § 4.3) :

```
void *p = ...;
void (*pf1)() = (void (*)())p;
```

ou, si l'on préfère, après le `typedef` précédent :

```
PtrFunc1_type pf1 = (PtrFunc1_type)p;
```

Remarque ISO C prévoit un *warning* lors des transtypages précédents, pourtant indispensables dans ce cas. Avec `gcc`, ce *warning* disparaît lorsque l'on supprime `-pedantic` des options de compilation. C'est pourquoi, exceptionnellement, cette suppression est faite dans la Makefile.