# Éditions de liens

Romaric DAVID

LEM2I - Décembre 2011

## Plan

- Compilation séparée
- Bibliothèques statiques
- Bibliothèques dynamiques
- Paramétrage de l'édition de liens
- Étude de cas

## 1 La compilation séparée

Dans le cours précédent, nous avons examiné le travail du compilateur sur un unique fichier source. Quand on écrit un code long, il est préférable de le séparer en plusieurs fichiers **source** afin de faciliter la maintenance (un fichier trop gros serait illisible), la réutilisabilité (si on souhaite reprendre seules quelques fonctions du code) et sa distribution. Cela permet également de réduire le temps de compilation.

À la compilation, ces fichiers source seront compilés séparément **(compilation séparée)**, en plusieurs fichiers objets. Chacun de ces fichiers objets ne contient donc qu'une fraction du code du programme.

Dans la dernière phase de la compilation, ces fichiers objets seront assemblés pour ne former qu'un exécutable. Cela constitue la phase d'édition le liens.

#### Compilation séparée

- La compilation séparée consiste en la création de plusieurs *fichiers objets* (un par ensemble de fonctions)
- À partir de ces fichiers objets, on peut assembler un exécutable : `gfortran a.o b.o c.o -o monprog`
- Ces processus est l'édition de liens

On peut souhaiter distribuer le code objet afin de son programme afin de le distribuer (pas forcément gratuitement). Si l'on a écrit des fonctions de base (solveur, traitement d'image..), ces fonctions seront réutilisées dans des programmes écrits par d'autres personnes.

La manière la plus simple de redistribuer son code est de fournir les fichiers objet (`.o`). Néanmoins, cela peut être complique si plusieurs fichiers objet sont nécessaires. De plus, la décomposition en fichiers objets traduit plutôt la manière que l'on a eu de construire son code plutôt que son utilisation. C'est pourquoi il est possible de rassembler *dans un fichier unique* un ensemble de fichiers objets.

Ce fichier unique peut être construit de deux manières. La première consiste à en faire un paquet destiné à être recopié tel quel à la fin de l'exécutable qui l'utilisera. On parle alors de *bibliothèque statique*. On utilise pour cela l'utilitaire `ar`.

**Distribution de code objet : un procédé simple**

- Problématique : rassembler les différents fichiers objet.
- Un procédé simple : la bibliothèque statique (archive)
- Assemblage de fichiers objets à intégrer lors de l'édition de liens.
- Édition de liens : résolution des branchements dans le code (appel de fonction = branchement) à la compilation

Utilisation de `ar`

- `ar cr archive.a fichier1.o fichier2.o`
- Construction d'un index des fonctions présentes dans l'archive, afin d'accélérer l'édition de liens.
- La mise à jour des composants de l'archive (fichiers objet) est possible individuellement : `ar r archive.a objet.o`

# 2 Bibliothèques statiques

**Utilisation de la bibliothèque**

- Utilisation du nom/chemin complet à la compilation
- Indication à l'éditeur de liens : emplacement et nom ($\Rightarrow$ convention sur le nom)

Résultat identique : recopie de l'ensemble du code objet dans l'exécutable.

- Convention de nommage : on parle de bibliothèque (*library*)
- Un nom de fonction présent dans une bibliothèque est appelé un symbole.
- Si l'on nomme l'archive `lib`**mesfonctions**`.a`
- $\Rightarrow$Utilisation abrégée par `-lmesfonctions`

- Indiquer l'emplacement : `-L/chemin/vers/la/bibliothèque`

Le compilateur utilise certains chemins de recherche des bibliothèques par défaut. Par exemple, pour le compilateur commercial intel V12, on trouve entre autres

```
/opt/intel/composerxe-2011.1.107/compiler/lib/intel64 \
/usr/lib/gcc/x86_64-redhat-linux/4.1.2 \
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../lib64 \
/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../.. \
/lib64 \
/lib \
/usr/lib64 \
/usr/lib \
```

L'ordre dans l'édition de liens est importante : les symboles d'une bibliothèque ne sont pris en compte que s'ils sont utilisés par un objet ou une bibliothèque présents **avant** dans la ligne de commande.

Ainsi, si une bibliothèque b2 dépend de (utilise des symboles présents dans) une bibliothèque b1, b1 devra figurer après b2 dans la phase d'éditions de liens statiques. Cela est problématique s'il existe des dépendances circulaires : certaines bibliothèques devront figurer 2 fois (ou plus...) sur la ligne de commande.

**Quelques ennuis...**

- Ordre à l'édition de liens
- Dépendances circulaires
- Utiliser le groupement de bibliothèques -Wl,–start-group a.a b.a -Wl,–end-group

**Caractéristiques des bibliothèques statiques**

- Elles conduisent à un exécutable auto-suffisant (appelé exécutable statique)
- Ainsi, l'exécutable est plutôt indépendant de l'état du système (et de l'environnement du processus)
- Inconvénient : toute modification dans la bibliothèque (correction de bugs ...) $\Rightarrow$ exige une recompilation de code final de l'utilisateur !
- Si plusieurs codes utilisent les mêmes bibliothèques, le code de ces bibliothèques est présent plusieurs fois en mémoire.

Les bibliothèques statiques, si elles permettent de distribuer un ensemble de fonctions, constituent un procédé limité :

- Dans sa consommation de ressources (taille du code, consommation mémoire de plusieurs code utilisant la même bibliothèque, par exemple les processus d'une application parallèle)
- Dans sa difficulté de mise à jour : pour mettre tous les codes utilisant une bibliothèque statique à jour après mise à jour de cette dernière, il faut les recompiler.

# 3 Bibliothèques dynamiques

Plutôt que de recopier à la compilation le contenu de la bibliothèque, il est possible d'y faire simplement *référence*. On note dans l'exécutable l'ensemble des bibliothèques dont le code a besoin (dépendances). Les adresses mémoire des branchements correspondant aux appels de fonctions de la bibliothèque sont, à la compilation, être remplacées par des blancs. À l'exécution du code, ces blancs seront alors remplacés par la bonne adresse par un programme appelé chargeur dynamique (programme `ld.so` Bien évidemment, ld.so est lui-même un exécutable statique.

**Distribution de code objet : un procédé élégant**

- Création de bibliothèques dynamiques (option `-shared` sous Linux et `-dylib` sous Mac Os X)
- Compilation du code avec références à ces bibliothèques (options `-L` et `-l` , identiques à la compilation statique
- -L indique l'emplacement des bibliothèques lors de la compilation du programme
- -ltruc $\Rightarrow$ Fichier libtruc.so. Il s'agit d'un raccourci syntaxique
- on peut aussi indiquer directement /usr/lib/libtruc.so.1 par exemple
- À l'exécution, calcul des branchements
- Si plusieurs programmes utilisent les mêmes bibliothèques, ils se partagent le code de ces bibliothèques.

Un code utilisant des bibliothèques dynamiques a besoin d'un environnement bien huilé pour fonctionner. En particulier, comme les branchements seront résolus à l'exécution, il faut s'assurer que les bibliothèques nécessaires sont accessibles. Pour cela, les chemins de recherche de ces bibliothèques sont configurés dans le système et dans l'environnement utilisateur.

**Exécution d'un programme dépendant de bibliothèques dynamiques**

- Visualisation des dépendances : `ldd`
- Variables d'environnement LD_LIBRARY_PATH (utilisateur)
- Chemins de recherche standard du système : `/etc/ld.so.conf`
- `/usr/lib puis /lib`

ld.so parcourt successivement tous les emplacements indiqués à la recherche des bibliothèques dont dépend le programme (cette information provient de la compilation du programme). Si une bibliothèque vient à manquer, un message du type

`./adyn: error while loading shared libraries: liba.so: cannot open shared object file: No such file or directory` est affiché et l'exécution du programme s'arrête.

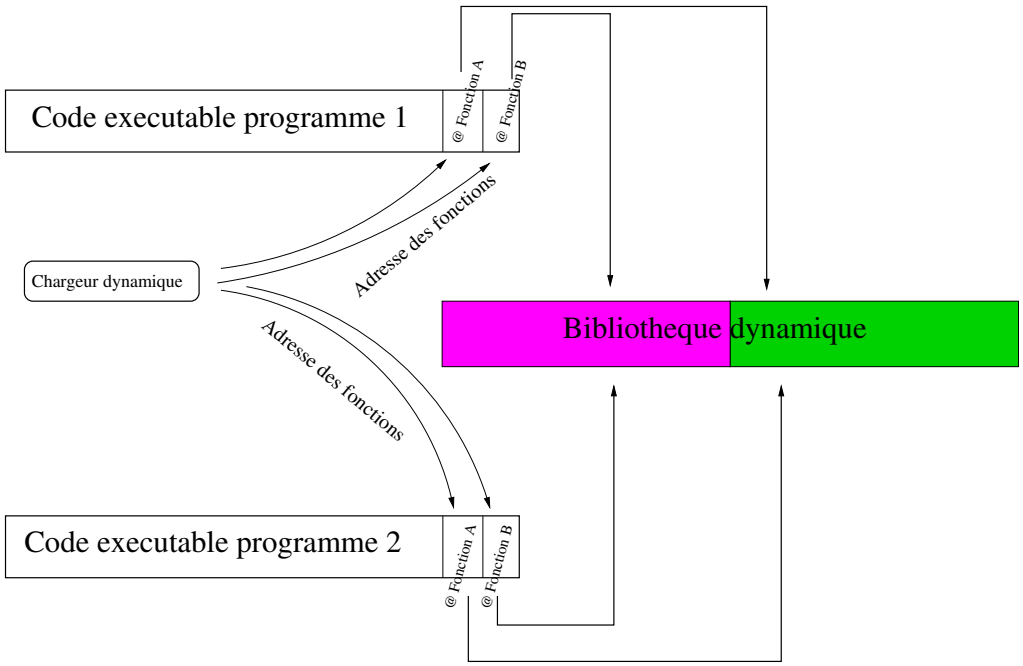
Une des difficultés lors de l'exécution d'un programme utilisant des bibliothèques partagées est le paramétrage correct de l'environnement du chargeur dynamique. En effet, devant le grand nombre de bibliothèques disponibles, il est probable que leurs chemins ne soient pas tous mentionnés dans les fichiers de configuration du chargeur dynamique. De même, plusieurs versions d'une même bibliothèque, pas toujours compatibles entre elles, peuvent cohabiter sur un système. La bonne propagation de l'environnement utilisateur est donc primordiale pour s'assurer que la bonne bibliothèque sera accessible. Cela peut être compliqué à gérer si plusieurs programmes d'un même utilisateur dépendent de plusieurs versions différentes de bibliothèques. Pour cette raison, on peut embarquer dans un exécutable l'information sur l'emplacement des bibliothèques dynamiques néces-saires et utiliser l'environnement module.

**Environnement d'exécution**

Problème : comment fixer à l'exécution le chemin de recherche des bilbiothèques ?

- Pour le chargeur dynamique (`LD_LIBRARY_PATH` ), dans l'environnement utilisateur
- Embarqué dans l'exécutable :
  - `-rpath` à l'édition de liens
  - Plus directement `-Wl,-rpath=...` à la compilation
  - ou variable `LD_RUN_PATH`

**The big picture**



# 4 Paramétrage de l'édition de liens

## 4.1 Statique ou dynamique ?

Vous aurez noté qu'à l'édition de liens, la référence à des bibliothèques statiques et dynamiques se fait par la même syntaxe. Que se passe-t-il si à la fois un fichier libtruc.a et un fichier libtruc.so sont présents ? Par défaut, l'éditeur de liens choisira la version dynamique de la bibliothèque. Bien évidemment, ce comportement est paramétrable.

**Statique ou dynamique ?**

- Par défaut, choix de la bibliothèque dynamique
- Option -Wl,-Bstatic : les bibliothèques indiquées après cette option seront liées statiquement.

Comme les bibliothèques dynamiques sont chargées à l'exécution, il est possible de modifier la version qui sera effectivement utilisée par une version améliorée (correction de bugs) ou une version incluant des routines de débogage, sans toucher au programme principal. On peut également utiliser une bibliothèque dont le rôle est d'encapsuler la bibliothèque d'origine. On parle alors de bibliothèque d'interposition.

**Fonctionnalités avancées**

- Mise à jour du système facilitée
- Bibliothèques d'interposition : permettent d'intercepter les appels (debug, profiling). Exemple : vampirtrace

## 4.2 Linker scripts

La commande ldd nous a permis de déterminer de quelles bibliothèques dépendait un exécutable donné. Les bibliothèques peuvent aussi dépendre (implicitement ou explicitement) de bibliothèques. En cas de dépendances explicites, celles-ci seront indiquées également par ldd. À l'exécution, les bibliothèques correspondantes seront mise en mémoire par le chargeur dynamique. Avantage : à l'édition de liens, il suffit de spécifier une bibliothèque sans préciser toutes ces dépendances.

Dans d'autres cas, certaines fonctions d'une bibliothèque se trouvent dans d'autres bibliothèques qui doivent être spécifiées par l'utilisateur. Par exemple, si une bibliothèque `b` dépend d'une fonction $f_b$ qui peut être définie dans deux bibliothèque(par ex. version accélérée et version normale), le choix de l'implémentation de $f_b$ incombe à l'utilisateur. Certaines combinaisons de bibliothèques sont pré-configurables et utilisables par l'éditeur de liens. On parle alors de *linker script*. Les linker scripts concernent aussi bien l'édition dynamique que statique.

**Scripts du linker**

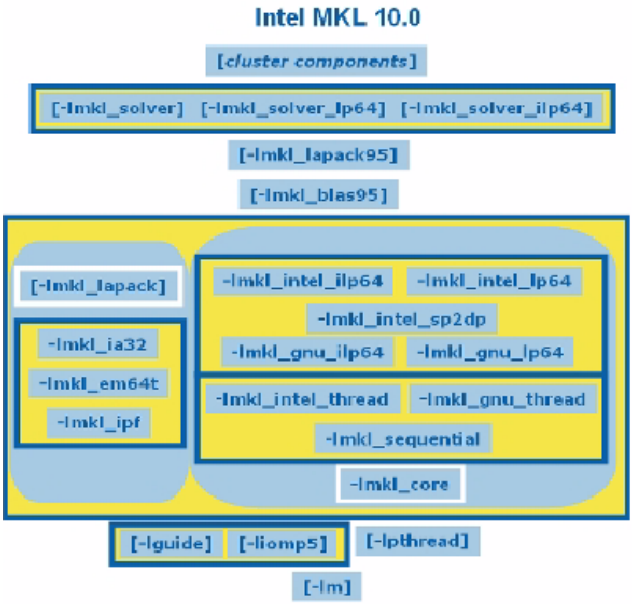
- Tout fichier n'étant pas une archive ou une bibliothèque dynamique sera interprété comme un script
- Script : ensemble de commandes destinées à l'éditeur de liens
- Commandes `INCLUDE` ou `GROUP`

# 5 Étude de cas : bibliothèques mathématiques Intel

La bibliothèqueIntel MKL comprend un ensemble de fonctions d'algèbre Linéaire (BLAS, Lapack, Scalapack, ...). Elle est fournie sous forme de bibliothèquepartagées ou statique. Certaines bibliothèques existent en version parallélisée (avec des threads pour Lapack) ou séquentielle. Des symboles de même nom existent dans les bibliothèques parallèles et séquentielles. Le choix entre les deux versions se fait à l'édition de liens. Des combinaisons prédéfinies sont proposées sous la forme de scripts de l'éditeur de liens.

**Aperçu de la bibliothèque**

Les dépendances entre les différentes composantes peuvent se représenter comme ceci :




Lapack dépend ainsi d'une combinaison de bibliothèques fonction de :

- Le modèle de programmation utilisé (taille des entiers en 64 bits)
- le compilateur utilisé pour les bibliothèques de bas niveau

d'une part et

- du modèle d'exécution (séquentiel, thread)

d'autre part. L'ensemble de ces bibliothèques s'appuie ensuite sur les fonctions de base Mkl.

La combinaison de toutes ces variantes dépend des caractéristiques du programme que l'on va ensuite utiliser. Certaines combinaisons (les plus répandues) sont pré-configurées dans des scripts du chargeur dynamique.

**Deux exemples de compilation**

- Version parallèle (threads) de la bibliothèque avec le compilateur intel : `-L/chemin-mkl -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core`
- Version séquentielle `-L/chemin-mkl -lmkl_intel_lp64 -lmkl_sequential -lmkl_core`

# 6 Conclusion

Les bibliothèques, surtout dynamiques, constituent un composant fondamental des applications. Leur création et mise en oeuvre est facile. Pour ces raisons, il est important dès la conception de modulariser son code en le construisant à base de bibliothèques. Cela permettra par la suite de mieux le distribuer et de l'interfacer avec d'autres langages.

**En résumé…**

- Construction de bibliothèques : tâche facile
- Codes parallèles : préférer les bibliothèques dynamiques
- Nécessite environnement utilisateur bien huilé
- Porte ouverte à l'interfaçage avec d'autres langages, comme python par exemple