La chaîne de compilation

Cédric Castagnède
Service Expérimentation et Développement (SED)

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche
BORDEAUX - SUD-OUEST

Introduction

Sommaire

- Le monde de la programmation
  - Interprétation
  - Machines virtuelles
  - Compilation
- Du source au binaire...
  - Pré-traitement
  - Compilation en langage assembleur
  - Conversion langage machine
  - Edition de liens
- Édition de liens
  - Introduction et comparaison
  - Édition de liens statiques
  - Édition de liens dynamiques

Le monde de la programmation

Les langages du monde informatique

- Langage de programmation :
  - description d'un ensemble d'actions consécutives qu'un ordinateur doit exécuter
  - Intelligible par le programmateur (en théorie...)

- Langage machine :
  - ensemble de données binaires
  - Intelligible par le processeur

- Il faut donc avoir un mécanisme pour passer du langage de programmation au langage machine

Interprétation

- Pas de conversion dans un autre langage
- Implémentation nécessite un interprète à chaque exécution
- Traduction à la volée du code source en code machine

- Les implémentations pouvant être interprétées :
  - facilite le développement
  - permettent, en général, l'exécution d'un programme incomplet
  - sont portables
- L'interpréteur peut être la première cause possible de lenteur

- Exemple : Script, PHP, JavaScript, Python, Perl, Scheme...

Les machines virtuelles

- Permettent d'interpréter et d'exécuter un « bytecode »
- Le code source est compilé en « bytecode » pour être interprété dans la machine virtuelle
- Le « bytecode » produit les mêmes résultats quelle que soit la plateforme

- Exemple : GNU Guile, Java, Python, Tcl...

Compilation

- Implémentation nécessite un compilateur pour générer un exécutable
- Traduction « une bonne fois pour toute » du code source en code machine

- Les implémentations pouvant être compilés fournissent des binaires :
  - indépendant du compilateur suite à la compilation
  - qui ne nécessite pas de programme pour s'exécuter
  - sont rapide à l'exécution
- Nécessite de re-compiler à chaque modifications

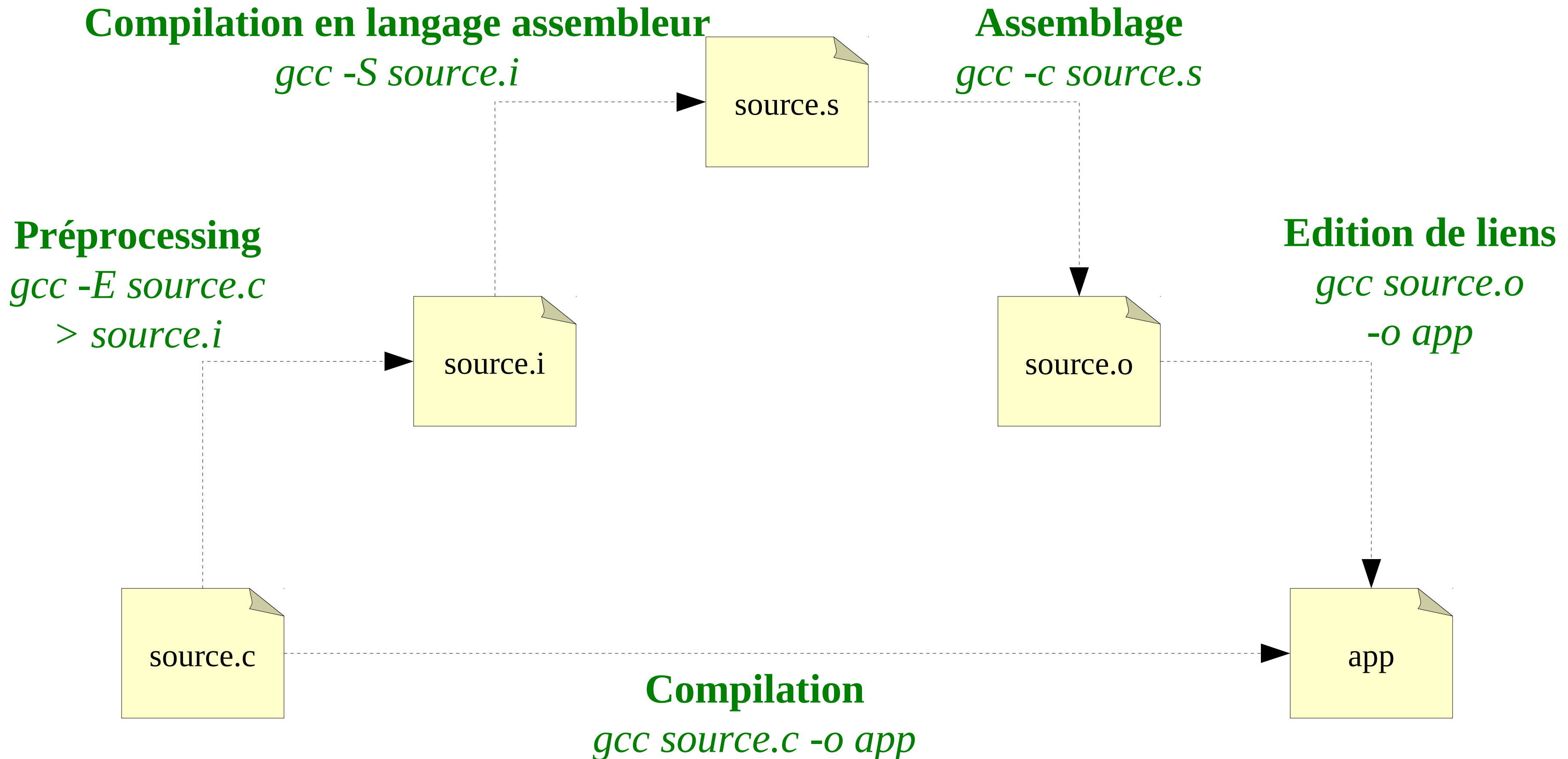
- Exemple : Fortran, C, C++, Java, Python, Scheme...

Abus de langage

- On parle souvent de « langage interprété » et, par opposition, « langage compilés » et c'est un abus de langage
  - Exemple : « langage interprété » = langage dont les implémentations utiliseront dans la plupart des cas un interprète
  - On réalise une implémentation d'un langage
  - L'implémentation utilisera un interprète ou un compilateur pour être exécuter
- Certaines fonctionnalité d'un langage ne peuvent être compilées :
  - `eval` pour Scheme, Perl, ...
  - `exec` pour Python
  - Mais cela peut être contourné

La chaîne de compilation

Illustration des étapes avec GCC



Pré-traitement

- Nécessite une analyse lexicale :
  - Élimination des commentaires, des espaces...
  - Reconnaissance des opérateurs et mots-clés
  - Reconnaissance des chaînes de caractères, les constantes numériques...
  - Vérifie les erreurs syntaxiques

- Réalise les préprocesseurs lexicaux (substitution) :
  - Réalise les inclusions
  - Remplace les macros existantes (#define)
  - Permet la compilation conditionnelle (#ifdef, #else, #endif, …)

Compilation en langage assembleur

- Transforme le fichier issue du pré-traitement en langage assembleur
- Fichier texte lisible
- Compréhensible si on connaît l'assembleur

Conversion langage assembleur en code machine

- Transforme le fichier en langage assembleur en code machine binaire
- Obtention du fichier objet
- Pas directement lisible
  - Il faut utiliser la commande : od -x fichier.o
  - od pour octal dump : affiche sur la sortie standard les octets en héxadécimal)

Edition des liens

- Le fichier objet est :
  - incomplet
  - possède seulement les prototype de fonction
- Édition de liens doit réunir les fichiers objet et les fonctions des bibliothèques pour produire un fichier exécutable
- Le fichier exécutable n'est pas utilisable sur n'importe quel machine

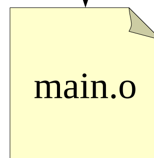
Édition de liens

Pourquoi l'édition de liens ?

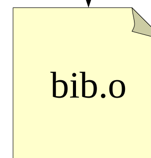
```
extern void mafonction(void);

int main (void)
{
  mafonction();
  return 0;
}
```

gcc -c main.c

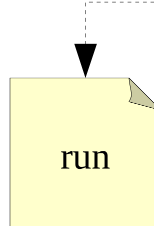


gcc -c bib.c




```
#include <stdio.h>
void mafonction(void)
{
  printf("Reveillez-vous au fond !!!\n");
  return;
}
```

???



- Les fichiers sources peuvent être compilés séparément
- Les fichiers objets doivent être assemblés mais comment ?

Identifier les symboles des fichiers objet

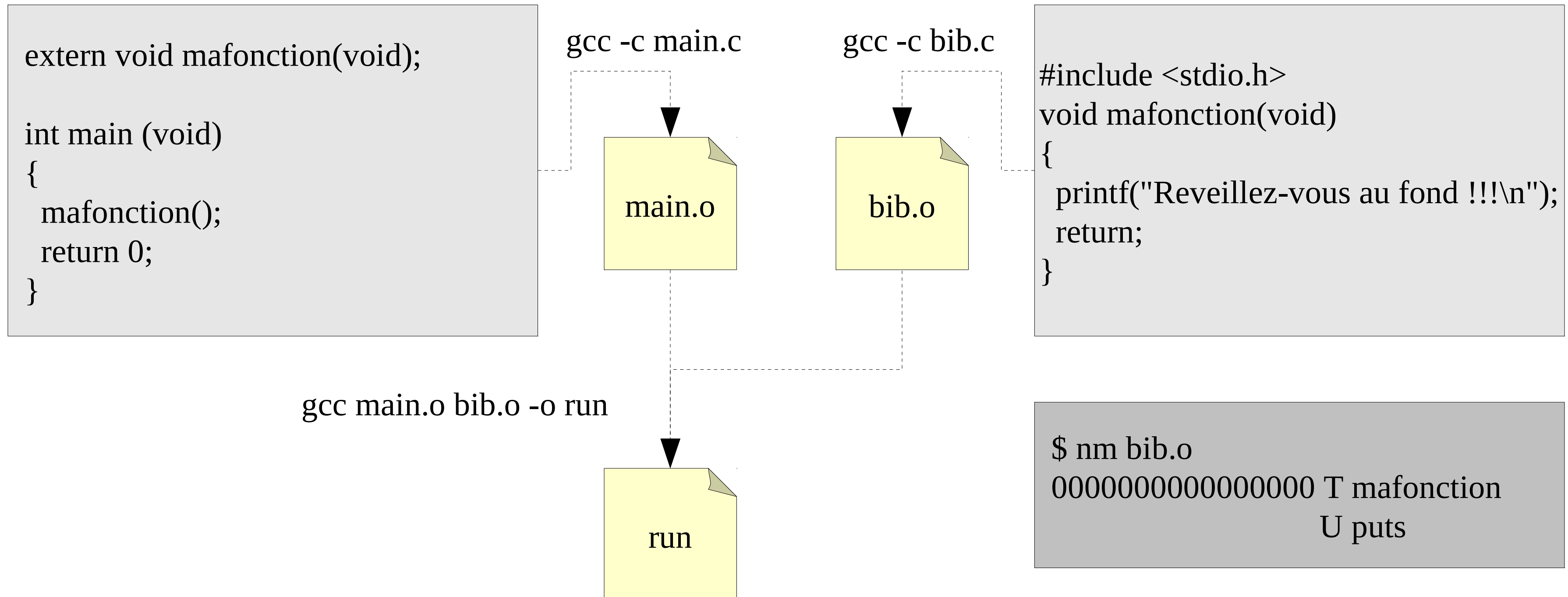
```
$ nm main.o
        0000000000000000 T main
                         U mafonction
```

Edition de liens : statique versus dynamique

- Edition de liens
  - Assembler un ensemble de fichiers objets
  - But : générer un fichier exécutable
  -
- Edition de liens statique
  - Intégration des bibliothèques dans l'exécutable
  - Aucun changement possible à l'exécution

- Edition de liens dynamique
  - Les bibliothèques dynamiques sont liées à l'exécution
  - Les bibliothèques peuvent être inter-changées

Edition de lien statique



- Génération d'un exécutable en assemblant tous les fichiers objets
- Tous les dépendances de bib.o sont satisfaites !!!

Que se passe-t-il sans bib.o ?

- Les symboles des fichiers objets ne sont pas satisfaits

```
extern void mafonction(void);

int main (void)
{
  mafonction();
  return 0;
}
```



```
$ gcc main.o -o run
main.o: In function `main':
main.c:(.text+0x5): undefined reference to `mafonction'
collect2: ld returned 1 exit status
```

Que faire s'il y a beaucoup de .o ?

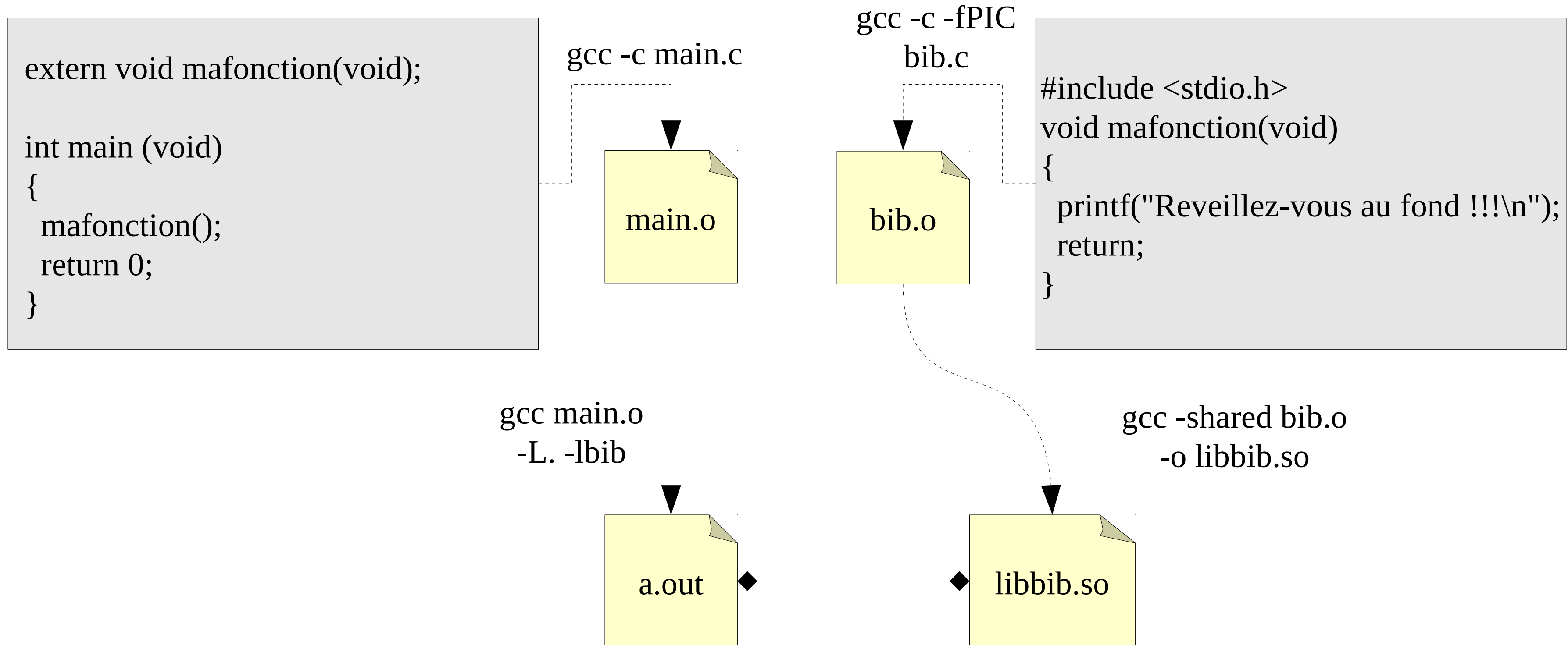
- Créer une archive de fichiers objets avec ar
- ar crée une archive contenant les fichiers objet et un index contenant les les symboles définis dans les fichiers objets.

```
$ ar r allbib.a bib.o
ar : creating allbib.a
$ ar t allbib.a
bib.o
```

```
$ nm -s allbib.a
Archive index:
message in bib.o

bib.o:
0000000000000000 T mafonction
                 U puts
$ nm -s –define-only allbib.a
$ gcc main.o allbib.a -o run
```

Edition de liens dynamique



- Création d'une bibliothèque dynamique
- Création d'un fichier exécutable lié dynamique avec cette dernière
- -fPIC : Génère du code indépendant de son adresse de chargement

Pourquoi l'édition de liens dynamique ?

- Mettre à jour la bibliothèque dynamique sans avoir à recompiler l'application
- Changer facilement la bibliothèque dynamique entre deux exécution
- La bibliothèque dynamique n'est pas dupliquée lors de plusieurs exécution simultanées de l'application

Identifier à posteriori l'édition de liens dynamique (1/3)

- La commande 'file' permet, entre autre, de savoir quelle édition de liens a été utilisée pour générer l'application :
  - dynamically linked : lié dynamiquement
  - statically linked : lié statiquement
  - Stripped : avec symbole de débogage
  - not stripped : sans symbole de

```
$ file run
run: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
```

Identifier à posteriori l'édition de liens dynamique (2/3)

- La commande 'ldd' permet de lister les bibliothèques dynamiques de l'exécutable
- On peut identifier notre bibliothèque dynamique
- Celle-ci n'est pas présente dans LD_LIBRARY_PATH

```
$ ldd run
        linux-vdso.so.1 =>  (0x00007fff223ff000)
        libbib.so => not found
        libc.so.6 => /lib/libc.so.6 (0x00007f1e98bf5000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1e98f96000)
```

Identifier à posteriori l'édition de liens dynamique (3/3)

- La commande 'objdump' affiche les symboles dynamiques des objets dynamiques
- 'mafonction' est défini seulement dans la bibliothèque dynamique

```
$ objdump -T libbib.so | grep mafonction
00000000000005ac g    DF .text         0000000000000012  Base        mafonction
$ objdump -T run | grep mafonction
0000000000000000      DF *UND*     0000000000000000              mafonction
```

Lancement de l'exécution

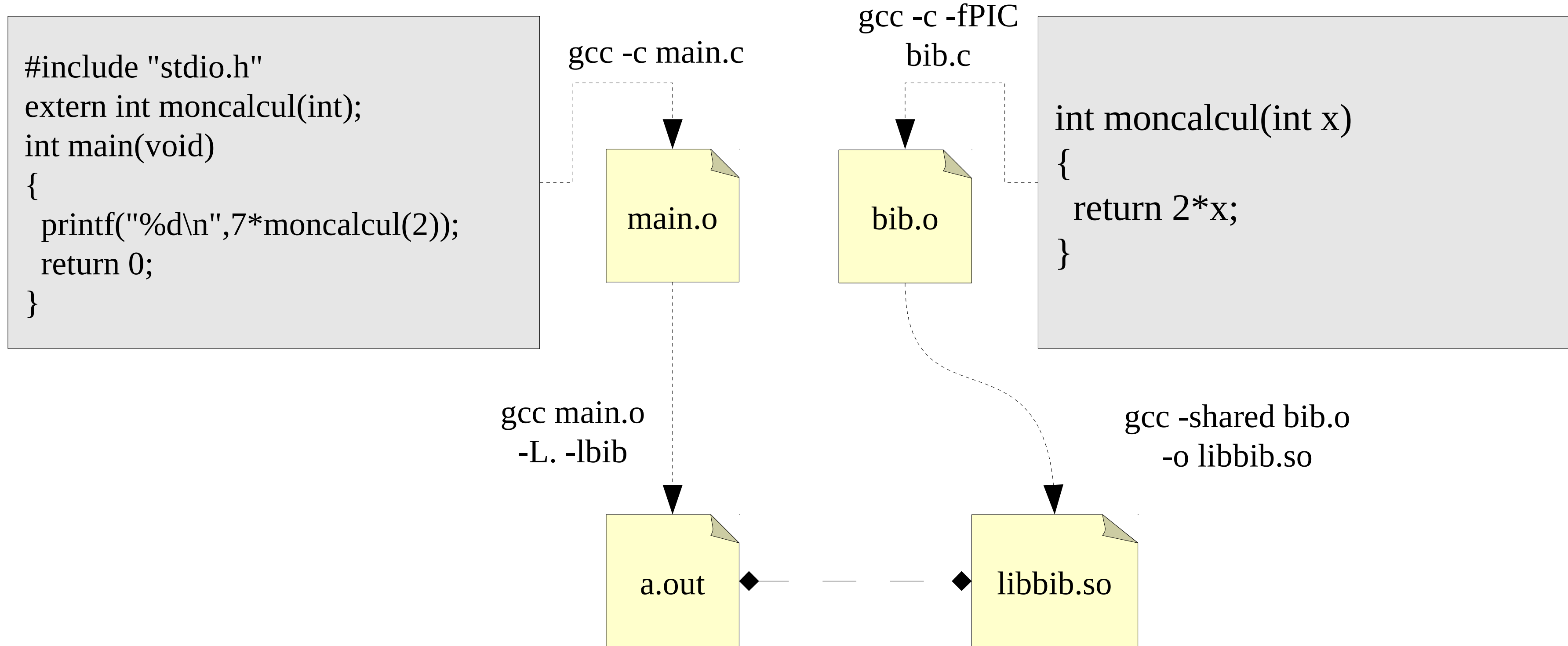
- Soit le chemin vers la bibliothèque dynamique doit être positionné dans l'environnement

```
$ ./run
./run: error while loading shared libraries:
libbib.so: cannot open shared object file: No such file or directory
$ export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
$ ./run
Reveillez-vous au fond !!!
```

- Soit on définit LD_LIBRARY_PATH en exécutant le programme

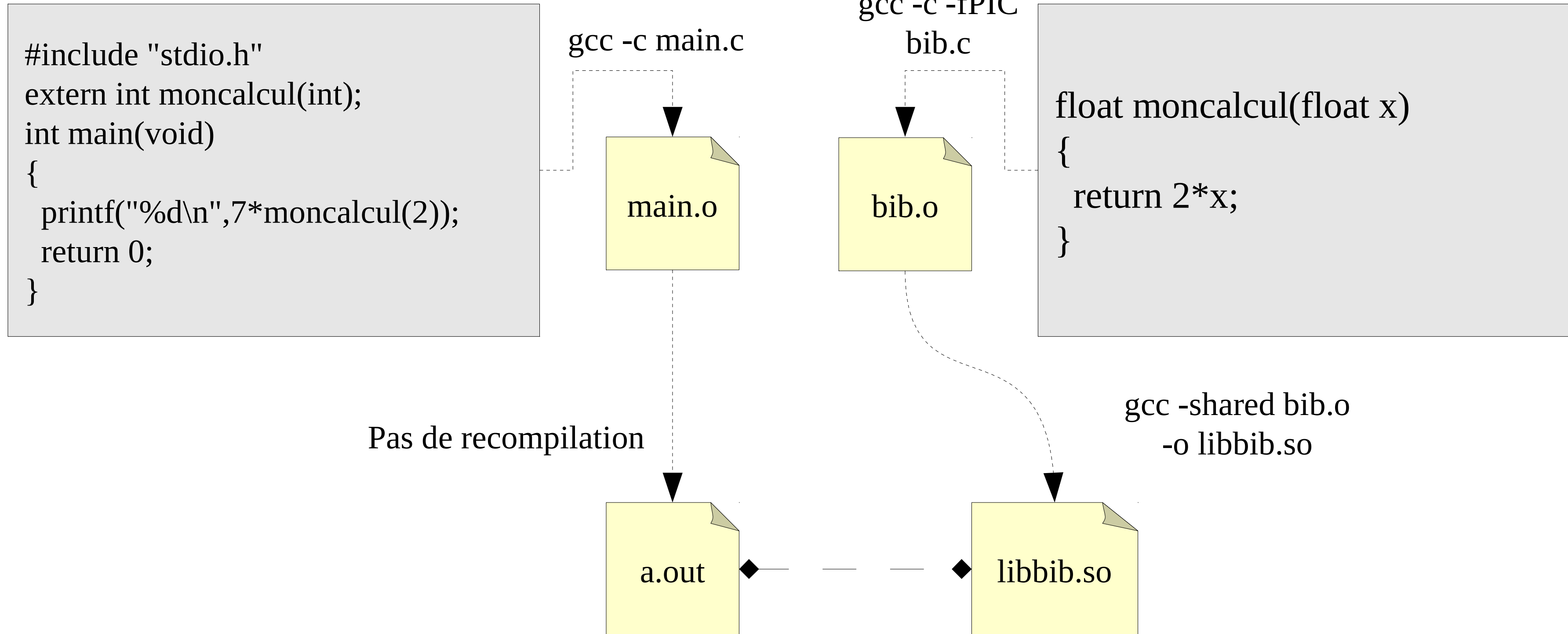
```
$ LD_LIBRARY_PATH=$PWD/lib1 ./run
Reveillez-vous au fond !!!
$ LD_LIBRARY_PATH=$PWD/lib2 ./run
Vous pouvez vous rendormir !!!
```

Et si on casse tout... (1/3)




```
$ LD_LIBRARY_PATH=$PWD ./run
28
```

Et si on casse tout... (2/3)



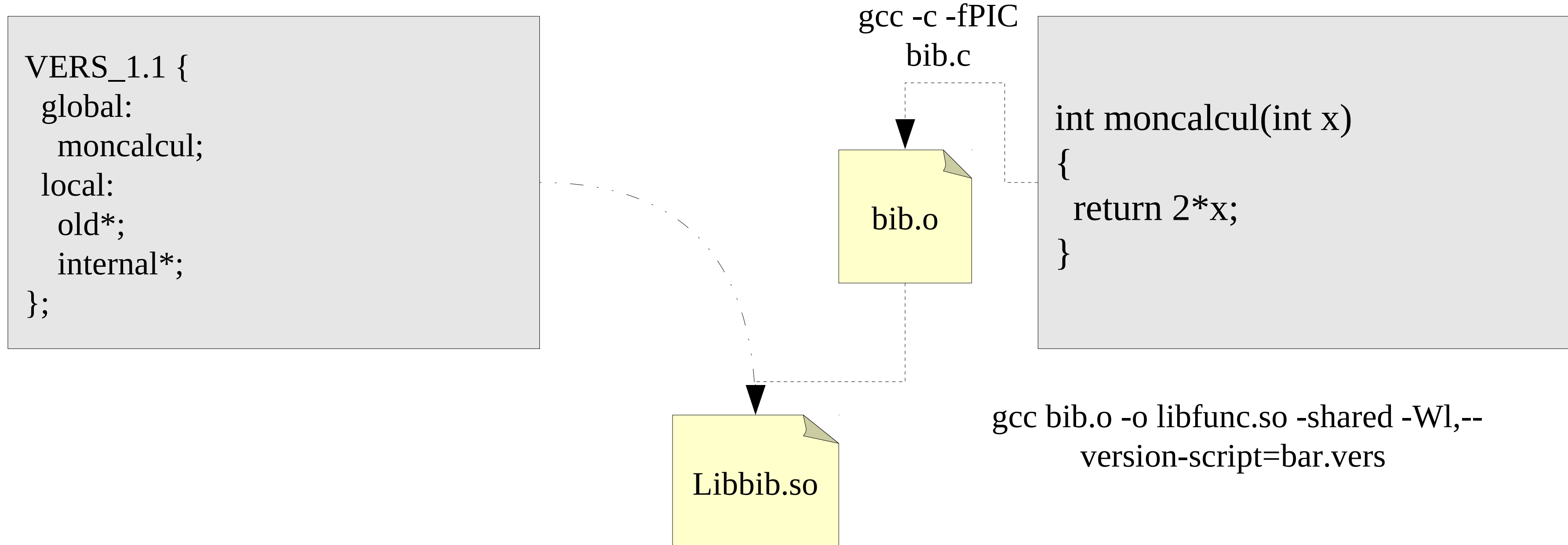

```
$ LD_LIBRARY_PATH=$PWD ./run
-1849464968
```

Et si on casse tout... (3/3)

- ABI (interface binaire-programme) d'un bibliothèque dynamique :
  - Prototype de fonction
  - Type des données des variables globales
  - Architecture des structures de données
  - Constantes

Des changements casse l'ABI,
effectuez les corrections et recompilez

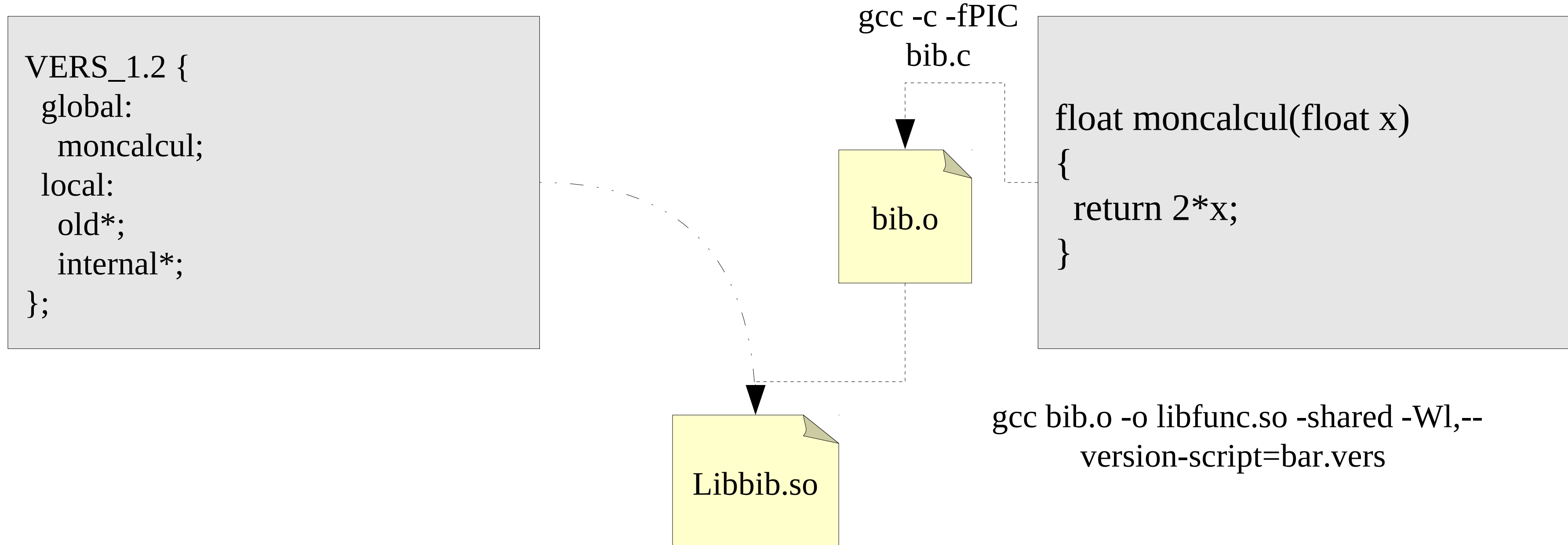
Comment se protéger ? (1/3)



- Positionner un « tag » sur les fonctions lors de la génération de la bibliothèque dynamique

```
$ objdump -T libbib.so | grep moncalcul
00000000000005cc g    DF .text        000000000000000e  VERS_1.1    moncalcul
```

Comment se protéger ? (2/3)



- Positionner un « tag » sur les fonctions lors de la génération de la bibliothèque dynamique

```
$ objdump -T libbib.so | grep moncalcul
00000000000005cc g    DF .text        000000000000000e  VERS_1.2    moncalcul
```

Comment se protéger ? (3/3)

- On compile l'application en utilisant la bibliothèque dynamique possédant le « tag » VERS_1.1 et on l'exécute

```
$ objdump -T run | grep moncalcul
0000000000000000      DF *UND*     0000000000000000  VERS_1.1    moncalcul
$ LD_LIBRARY_PATH=$PWD ./run
28
```

- Sans recompiler, on exécute l'application en utilisant la bibliothèque dynamique VERS_2.2

```
$ LD_LIBRARY_PATH=$PWD ./run
./run: /home/castagne/dev/demo2c/lib2/libfunc.so:
version `VERS_1.1' not found (required by ./run)
```

Et si on veut fournir les deux versions...

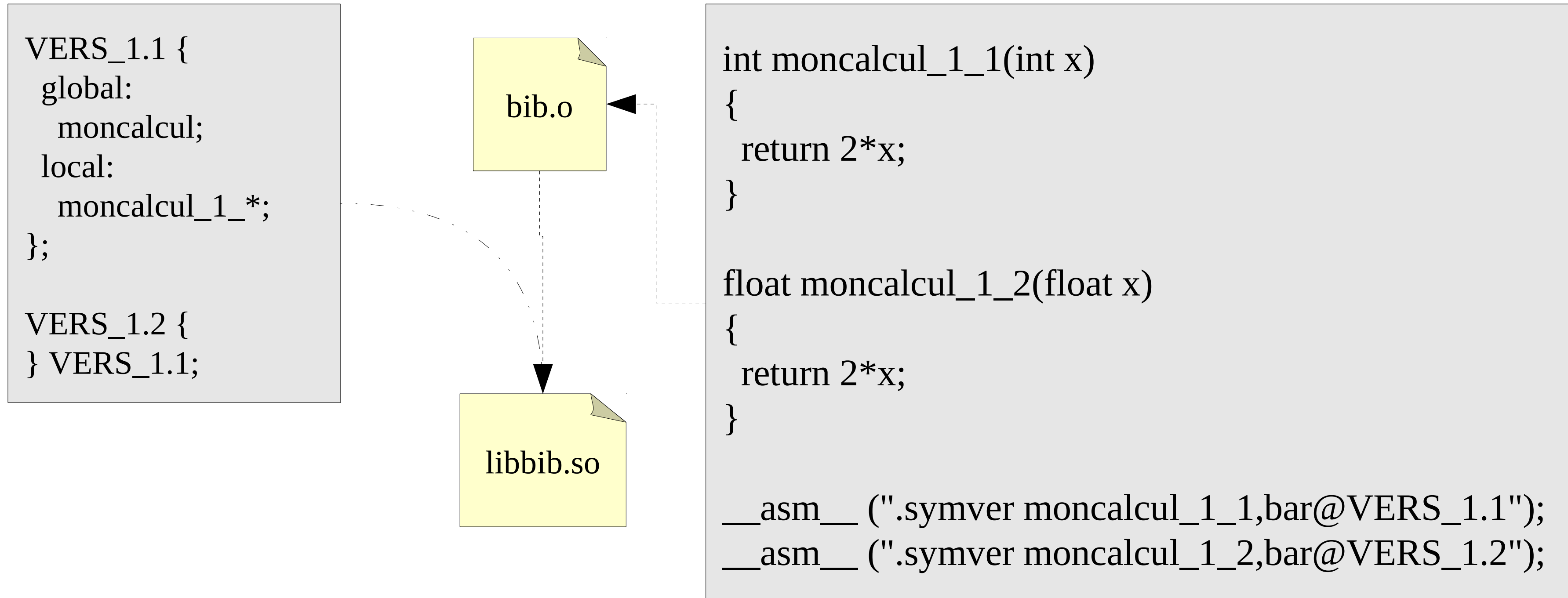
- On compile l'application en utilisant la bibliothèque dynamique possédant le « tag » VERS_1.1 et on l'exécute

```
$ objdump -T run | grep moncalcul
0000000000000000      DF *UND*     0000000000000000  VERS_1.1    moncalcul
$ LD_LIBRARY_PATH=$PWD ./run
28
```

- Sans recompiler, on exécute l'application en utilisant la bibliothèque dynamique VERS_2.2

```
$ LD_LIBRARY_PATH=$PWD ./run
./run: /home/castagne/dev/demo2c/lib2/libfunc.so:
version `VERS_1.1' not found (required by ./run)
```

Et si on veut fournir les deux versions...




```
$ objdump -T libfunc.so | grep moncalcul
000000000000064a g    DF .text    0000000000000014 (VERS_1.2)   moncalcul
000000000000063c g    DF .text    000000000000000e (VERS_1.1)   moncalcul
```

Conclusion

Questions ???

Références

- http://pauillac.inria.fr/~maranget/X/compil/poly/index.html
- http://sourceware.org/binutils/docs-2.21/binutils/index.html
- http://darcs.olsner.se/Linker/linker-book
- http://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_toc.html