Systèmes d'exploitation - Compilation séparée

Pierre Gançarski

DUT Informatique - S31

ATTENTION
Ces transparents ne sont qu'un guide du cours : de nombreuses explications et illustrations manquent.

De nombreuses précisions seront données au tableau et à l'oral pendant le cours.

Université de Strasbourg

IUT Robert Schuman
UNIVERSITÉ DE STRASBOURG

Plan

Compilation d'un programme

Exemple

Soit le code C suivant dans un fichier `Prog.c`

```c
#include <stdio.h>
#define N 10

int incr (int glob) { glob ++;  return glob; }

void affiche(int glob) {  printf ("Glob = %d \n", glob);
     }

int main(int argc, char *argv[]){
  int glob = N;
  glob = Incr(glob);
  affiche(glob);
}
```

- Comment le compilateur va-t-il créer un fichier `a.out` exécutable ? :

Compilation d'un fichier

Rappel

- Les deux premières étapes de la compilation d'un programme `P.c` consiste en :
  1. Analyse textuelle : le compilateur traite les #define
  2. Traduction : Le compilateur traduit le code. Pour les fonctions (`main` compris) il leur associe une adresse correspondant au code compilé de la fonction. Enfin, chaque appel à une fonction est remplacé par un saut à l'adresse de la fonction.
- Problème - Comment le compilateur va-t-il traiter le cas d'une fonction pour laquelle il n'a pas de code (donc pas d'adresse pour cette fonction) : `printf ("Glob ...);`

Compilation d'un fichier

Rappel

- La compilation d'un programme `P.c` → trois étapes :
  1. Analyse textuelle
  2. Traduction :
     * Le compilateur traduit le code des fonctions (`main` compris) en leur associant une adresse dans le fichier compilé : chaque appel à une telle fonction est remplacé par un saut à l'adresse de la fonction.
     * Pour une fonction non définie dans ce fichier (par exemple `printf`) il va chercher le profil de la fonction dans tous les fichiers (généralement `.h`) donnés par les `#include` :
       - S'il le trouve (par exemple ici dans `stdio.h`), il marque la fonction comme connue. Par contre, l'adresse du saut sera marquée comme inconnue.
       - Pour pour chaque fonction pour laquelle il ne trouve pas de profil, il émet une erreur ou un warning : appel à une fonction inconnue.

Compilation d'un fichier

Rappel

- La compilation d'un programme `P.c` → trois étapes :
  1. Analyse textuelle
  2. Traduction
  3. Edition des liens : supprimer les adresses inconnues dans les sauts du code compilé. Trois possibilités
     - Il trouve le code dans un des fichiers dits "objet" passés en paramètres : il s'agit de fichiers (généralement `.o`) ne contenant que des codes de fonctions sans main → Il le recopie le code dans le fichier compilé a.out et met à jour l'adresse du saut
     - Il trouve le code dans une des librairies données en paramètres (la libC est toujours considérée comme paramètre) → Il recopie le code (librairie statique) ou met à jour l'adresse du saut dans cette librairie (librairie dynamique - cf plus loin)
     - Il ne trouve pas de code → il émet une erreur d'édition de lien à une fonction inconnue.

Plan

Compilation séparée

Exemple

Soit le code C suivant dans un fichier `Prog.c`

```c
#include <stdio.h>

int incr (int glob) { glob ++;  return glob; }

void print(int glob) {  printf ("Glob = %d \n", glob); }

int main(int argc, char *argv[]){
  int glob = 1;
  glob = Incr(glob);
  print(glob);
}
```

- Idée : Utiliser la capacité du compilateur à éditer des liens sur des fichiers externes

Compilation séparée

Séparation des fonctions et du `main`

- Idée : Utiliser la capacité du compilateur à éditer des liens sur des fichiers externes → regrouper les fonctions dans un fichier `Func.c`

```c
#include <stdio.h>
int incr (int glob) { glob ++;  return glob; }
void print(int glob) {  printf ("Glob = %d \n", glob); }
```

et le main dans un autre fichier `Main.c`

```c
int main(int argc, char *argv[]){
  int glob = 1;
  glob = Incr(glob);
  print(glob);
}
```

Compilation séparée

Compilation des fonctions externes

- Idée : Utiliser la capacité du compilateur à éditer des liens sur des fichiers externes → Compiler le fichier `Func.c` indépendamment du `main`

```c
// Fichier func.c
#include <stdio.h>
int incr (int glob) { glob ++;  return glob; }
void print(int glob) {  printf ("Glob = %d \n", glob); }
```

- Compiler séparément les fichiers externes
  1. Les codes des fonction sont dans un ou plusieurs fichiers `.c`. On les compile avec l'option `-c` qui signale qu'il n'y a pas main et pas d'édition de liens : `gcc -c Func.c`
     → Résultat : fichier `Func.o`

Compilation séparée

Compilation du `main`

- Idée : Utiliser la capacité du compilateur à éditer des liens sur des fichiers externes → Compiler le fichier `main.c` et effectuer l'édition de liens

```
int main(int argc, char *argv[]){
  int glob = 1;
  glob = Incr(glob);
  print(glob);
}
```

- Compiler le main
  1. Fichier `Func.o`
  2. Le fichier principal ne contient plus les codes des fonctions : l'édition de liens se chargera de les ajouter au fichier compilé a.out
     Exemple : `gcc -c Prog.c Func.o` → Résultat : fichier `a.out`

Bibliothèques

Compilation séparée

- Permet de décomposer un programme en plusieurs fichiers
  - plus simple à maintenir
  - permet de partager du code
- Pour cela :
  1. Le code C du programme est éclaté sur plusieurs fichiers :
     - Les fonctions sont réparties sur un ensemble de fichiers
     - Le `main` est isolé dans un fichier
  2. Chacun de ces fichiers est compilé indépendamment
  3. L'édition des liens reliera tous les codes lors de la compilation du fichier contenant le `main`

Plan

Bibliothèque

Bibliothèques binaires

- Une bibliothèque binaire est une collection de fichiers-objets `.o` regroupés en une seule entité
- Deux types de bibliothèques :
  - statiques (`.a`) : liées lors de la compilation
  - partagées (`.so`) (*shared object*) : chargées et liées dynamiquement lors de l'exécution

Bibliothèque

Bibliothèques statiques

- Création : `libtool -static -o libcprog.a p1.o ...pn.o`
- Utilisation :
  - `gcc P.c -libprog.a` si `libprog.a` est dans le répertoire courant
  - `gcc P.c -lprog` si `libprog.a` est dans le chemin de recherche des bibliothèques (cf. plus loin)
- Le code des fonctions (des `.o`) appelées dans le programme est ajouté au code exécutable.

Bibliothèques

Bibliothèques dynamiques

- Les fonctions ne sont pas intégrées dans l'exécutable, mais partageables entre tous les processus : les fonctions doivent être **réentrantes**
- Correspondent aux bibliothèques `.dll` (Dynamic Loadable Library) en MS Windows et `.so` (soname) en Unix
- Chargement
  - Implicite : Lorsqu'un processus qui s'exécute appelle une fonction d'une telle bibliothèque, celle-ci est chargée automatiquement en mémoire si elle n'y est pas déjà → la bibliothèque est alors ajoutée via un segment partagé
  - Explicite : Le processus va ouvrir les bibliothèques et chercher les fonctions et les variables de façon explicite → utilisation de quatre fonctions fournies par la bibliothèque `libdl.so` :
    - `void *dlopen(const char *filename, int flag)` : Ouvre une bibliothèque dynamique
    - `void dlclose(void *plib)` : Ferme la bibliothèque
    - `void *dlsym(void *plib, const char *symbolname)` : Recherche d'une fonction ou d'une variable
    - `char *dlerror()` : Décode une erreur

Bibliothèques

Bibliothèques dynamiques

- Création : `gcc -shared P1.c ...Pn.c. -o libprog.so`
- Utilisation : `gcc Prog.c -libprog.so` (ou `gcc P.c -lprog` si `libprog.so` est dans le chemin de recherche des bibliothèques)
- La recherche de librairies (`.a` ou `.so`) se fait de plusieurs manières :
  - Répertoires par défaut (`/lib`, `/usr/lib`, ...)
  - Ajout : `/etc/ldconfig` → fichier `/etc/ld.so.conf`
  - Variable d'environnement `LD_LIBRARY_PATH` (propre à chaque utilisateur)

Plan

Makefile

Principe

- Automatisation de la compilation via un fichier précisant des cibles avec leurs dépendances et leur mécanisme de construction
  Exemple : fichier `makefile.txt` (Utilisation : `make -f makefile.txt Main`)

  Func.o : Func.c
  gcc -Wall -c Func.c
  Main : Main.c Func.o MyIncludes.h
  gcc -Wall Main.c Func.o -o Main
- Une règle est appliquée si une des dépendances (ex : `Func.c`) est plus récente que la cible (ex : `Func.o`)
- Récursivité : Pour une cible donnée, chaque dépendance est vérifiée : s'il existe une cible la concernant, celle-ci est vérifiée. Après avoir vérifier toutes ces dépendances, si une ou plusieurs d'entre elles ont été reconstruites, alors la cible est plus "vieille" que celles-ci et donc la règle s'applique $\rightarrow$ la cible est reconstruite

Makefile

Makefile enrichi

- Cibles complémentaires classiquement ajoutées :
  - all : qui regroupe toutes les dépendances des exécutables à produire.
  - clean : qui supprime tous fichiers intermédiaires générés
  - mrproper : qui supprime tout ce qui peut être régénéré et permet une reconstruction complète du projet

  Exemple :

```
clean :     #supprimer les fichier *.o
        rm -rf *.o
```

- Variables sous la forme NOM=VALEUR et s'utilise via $(NOM).
  - CC désigne le compilateur utilisée (CC=gcc).
  - CFLAGS regroupe les options de compilation (CFLAGS= -Wall)
  - LDFLAGS regroupe les options de l'édition de liens.
  - EXEC donne le nom des exécutables à générer.

Makefile

Makefile enrichi

- Cibles complémentaires
- Variables sous la forme NOM=VALEUR s'utilisant via $(NOM).
- Variables prédéfinies.

$@	Nom de la cible
$<	Nom de la première dépendance
$^	Liste des dépendances
$?	Liste des dépendances plus récentes que la cible
$*	Nom du fichier (sans son suffixe)

Exemple :

```
hello.o : hello.c hello.h
	$(CC) -o $@ -c $< $(CFLAGS)
main : main.c hello.o
	$(CC) -o $@ $< $^ $(CFLAGS)
```

Makefile

Makefile enrichi

- Cibles complémentaires
- Variables sous la forme NOM=VALEUR s'utilisant via $(NOM).
- Variables prédéfinies.
- Règles génériques telle que :
  %.o : %.c
  $(CC) -o $@ -c $< $(CFLAGS)
  qui décrit comme construire un .o à partir d'un .c.
- "Parcours" d'une liste de cibles :
  EXEC=p1 p2 p3
  $(EXEC) : % : %.c $(LDEP)
  $(CC) -o $@ $^ $(CFLAGS) $(LDEP)

Makefile

Makefile enrichi

- Liste des fichier à générer

```
# L'exécutable à générer :
EXEC=calculatrice

# Les fichiers .o a générer si nécessaire
ODEP=calculatrice_f_ext.o calcule.o

#Les librairies à construire
LDEP=libcalc.a

all: $(EXEC)
```

Makefile

Makefile enrichi

- Liste des fichier à générer
- Les cibles correspondantes

```
#Variables personnalisées
CC=gcc
CFLAGS=-Wall
LDFLAGS=-lm

libcalc.a: $(ODEP)
	libtool -static -o $@ $^

%.o: %.c
	$(CC) -c $^ $(CFLAGS)
```

Makefile

Makefile enrichi

- Liste des fichier à générer
- Les cibles correspondantes
- Génération des exécutables

```
$(EXEC): %: %.c $(LDEP)
	$(CC) $(CFLAGS) $(LDEP) -o $@ $^
```

- Les "utilitaires"

```
$(EXEC): %: %.c $(LDEP)
	$(CC) $(CFLAGS) $(LDEP) -o $@ $^

#supprimer les fichier *.o
clean:
	rm -rf *.o *.a

#supprimer les fichiers *.o et les exécutables
mrproper: clean
	rm -rf $(EXEC)
```