ENSTA ParisTech
École nationale supérieure de techniques avancées

Compilation séparée avec `make`

Cours IN201
Systèmes d'exploitation

Marc Baudoin
<babafou+in201@babafou.eu.org>

La compilation séparée est une caractéristique fondamentale du langage C. Elle permet de diviser un programme en plusieurs modules, chacun étant contenu dans un fichier source dédié. Cela permet de structurer le programme et, dans le meilleur des cas, de pouvoir réutiliser certains modules dans d'autres programmes. Cependant, dès que le nombre de modules est suffisamment important, il devient fastidieux de taper les commandes de compilation. On aimerait pouvoir disposer d'un outil qui se charge tout seul de la compilation, qui saurait quels modules recompiler et comment le faire. Cet outil magique existe, il s'appelle `make`.

1 Problèmes posés par la compilation séparée

Dans la suite de ce document, nous utiliserons l'exemple d'un fichier exécutable nommé `prog`, obtenu à partir de trois fichiers sources, `fichier1.c`, `fichier2.c` et `fichier3.c` et devant être lié avec la bibliothèque mathématique. La compilation manuelle de ce programme s'effectue donc grâce à la commande :

```
$ cc -o prog fichier1.c fichier2.c fichier3.c -lm
```

Outre le fait que cette commande soit longue à taper, cette façon de procéder impose, si l'on ne modifie qu'un seul des trois fichiers C, de compiler également les deux autres, ce qui représente une perte de temps.

Pour améliorer cela, on peut utiliser des fichiers objets. Un *fichier objet* est le résultat de la compilation d'un fichier C mais sans la phase d'édition de liens. Un fichier objet s'obtient grâce à l'option `-c` de `cc` et génère un fichier de même nom que le fichier C mais avec une extension `.o`.

En utilisant des fichiers objets, la compilation de `prog` s'effectue grâce aux commandes :

```
$ cc -c fichier1.c
$ cc -c fichier2.c
$ cc -c fichier3.c
$ cc -o prog fichier1.o fichier2.o fichier3.o -lm
```

Ceci est plus compliqué qu'auparavant mais, si l'on ne modifie que `fichier1.c`, la recompilation de `prog` ne nécessite plus qu'une compilation et une édition de liens, ce qui permet d'éviter deux compilations :

```
$ cc -c fichier1.c
$ cc -o prog fichier1.o fichier2.o fichier3.o -lm
```

Cette méthode est donc plus optimale mais nécessite de taper plus de commandes.

2 La commande `make`

Heureusement, la commande `make` permet d'automatiser tout cela. Il suffit de lui expliquer comment compiler `prog`, de taper `make`, et la compilation s'effectuera automatiquement en exécutant uniquement les commandes nécessaires.

> http://en.wikipedia.org/wiki/Make_(software)
> http://en.wikibooks.org/wiki/Make

2.1 Le fichier `Makefile`

Les informations permettant à `make` de compiler `prog` doivent se trouver dans un fichier appelé `Makefile` (ou bien `makefile`). Un fichier `Makefile` est un fichier texte. Voici un exemple de fichier `Makefile` permettant de compiler `prog` :

```
1  prog : fichier1.o fichier2.o fichier3.o
2  ————→cc -o prog fichier1.o fichier2.o fichier3.o -lm
3
4  fichier1.o : fichier1.c
5  ————→cc -c fichier1.c
6
7  fichier2.o : fichier2.c
8  ————→cc -c fichier2.c
9
10 fichier3.o : fichier3.c
11 ————→cc -c fichier3.c
```

Un fichier `Makefile` est composé de *règles* (il y en a quatre dans notre exemple), qu'on sépare habituellement par des lignes blanches (ce n'est pas obligatoire mais cela permet d'aérer le fichier `Makefile` et de mieux visualiser les règles).

Chaque règle est de la forme :

```
cible : dépendance1 dépendance2 ...
⟶commande
⟶commande
⟶...
```

La première ligne d'une règle indique sa *cible*, qui est généralement le nom d'un fichier à construire. La cible est suivie d'un deux-points (les espaces autour du deux-points ne sont pas obligatoires mais permettent là encore d'aérer les choses) puis d'une liste de fichiers, appelés *dépendances*, à partir desquels la cible est construite.

Les lignes suivantes indiquent les commandes (il n'y en a souvent qu'une) à exécuter pour construire la cible à partir de ses dépendances. Chacune de ces lignes doit absolument débuter par une tabulation, représentée dans ce document par une longue flèche. Attention à respecter scrupuleusement ce format car la commande `make` est très stricte à ce sujet.

Ainsi, la règle :

```
fichier1.o : fichier1.c
⟶cc -c fichier1.c
```

signifie que le fichier `fichier1.o` dépend du fichier `fichier1.c`, c'est-à-dire qu'il est obtenu à partir de celui-ci, et que la commande à exécuter pour obtenir `fichier1.o` à partir de `fichier1.c` est `cc -c fichier1.c`.

2.2 Utiliser la commande `make`

Une fois le fichier `Makefile` créé, il suffit de taper `make` :

```
$ make
cc -c fichier1.c
cc -c fichier2.c
cc -c fichier3.c
cc -o prog fichier1.o fichier2.o fichier3.o -lm
```

Les commandes exécutées par `make` sont affichées au fur et à mesure.

2.3 Comment fonctionne la commande `make` ?

Lorsqu'on l'exécute, la commande `make` va lire les informations contenues dans le fichier `Makefile`. Elle considère la première cible rencontrée comme la cible à construire. Dans notre exemple, il s'agit de `prog`, qui dépend de `fichier1.o`, `fichier2.o` et `fichier3.o`, qui

dépendent eux-mêmes respectivement de `fichier1.c`, `fichier2.c` et `fichier3.c` (ces derniers ne dépendent de rien).

Les dépendances de certaines règles peuvent donc être aussi les cibles d'autres règles. On représente habituellement les relations entre cibles et dépendances sous la forme d'un arbre. Notre exemple peut être représenté par l'arbre des dépendances indiqué dans la figure 1.

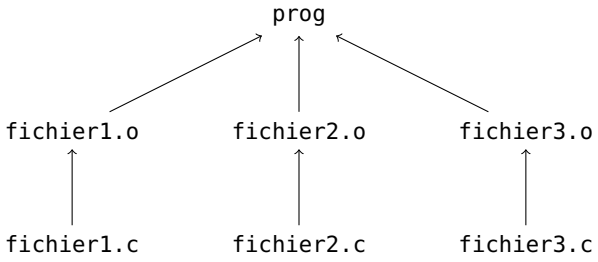

FIGURE 1 – *Arbre des dépendances*

Maintenant, si l'on modifie `fichier1.c` et lui seul, il faut le recompiler pour obtenir un nouveau `fichier1.o`. En revanche, `fichier2.c` et `fichier3.c` n'ayant pas été modifiés, il n'est pas nécessaire de les recompiler. Enfin, il faut refaire l'édition de liens de `fichier1.o`, `fichier2.o` et `fichier3.o` pour obtenir `prog`.

En pratique, `make` se rend compte de ce qu'il faut faire grâce aux dates de dernière modification de ces fichiers. Puisqu'on vient de modifier `fichier1.c`, la date de dernière modification de ce fichier est postérieure à celle de `fichier1.o`. `make` le recompile alors en utilisant la commande appropriée (celle qui figure dans le fichier `Makefile`). Comme `prog` dépend de `fichier1.o` et que celui-ci lui est postérieur (on vient de le modifier en recompilant `fichier1.c`), `make` effectue aussi l'édition de liens :

```
$ make
cc -c fichier1.c
cc -o prog fichier1.o fichier2.o fichier3.o -lm
```

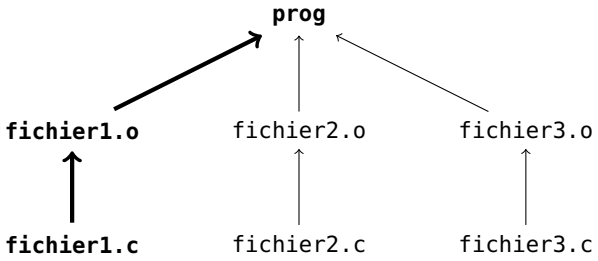

FIGURE 2 – *Principe de fonctionnement de la commande* `make`

Ce mécanisme est illustré de manière plus imagée dans la figure 2. La commande `make` remonte l'arbre des dépendances depuis les feuilles (parties terminales ne dépendant de rien)

jusqu'à la racine (c'est la cible à atteindre). Dès qu'il trouve un fichier dont la date de dernière modification est plus récente que celle du fichier qui suit, il exécute la commande appropriée, puis poursuit sa montée.

3 Un fichier `Makefile` plus complet

3.1 Les commentaires

Il est possible (et même recommandé) d'indiquer des commentaires dans un fichier `Makefile`. Un commentaire commence par un croisillon `#` et s'étend jusqu'à la fin de la ligne :

```
1 # fichier Makefile permettant de compiler prog
```

3.2 Les variables

Imaginons qu'on veuille utiliser le compilateur `pcc` au lieu de `cc`. Il faut donc changer `cc` en `pcc` partout dans le fichier `Makefile`. Le nôtre n'est pas bien long mais l'opération serait fastidieuse avec un fichier `Makefile` plus complexe. C'est pourquoi `make` permet l'emploi de variables dans un fichier `Makefile`. Une variable se déclare ainsi :

```
VARIABLE = valeur
```

Le nom d'une variable est habituellement en capitales. Par exemple : `CC`, `CFLAGS` ou `LDLIBS`.

Les espaces autour du signe égal ne sont pas obligatoires.

La valeur d'une variable s'utilise ensuite en indiquant un symbole dollar `$` suivi du nom de la variable entre parenthèses ou entre accolades. Par exemple : `$(CC)`, `$(CFLAGS)` ou `$(LDLIBS)`.

Ainsi, voici le début d'un fichier `Makefile` permettant de changer facilement le compilateur utilisé, les options du compilateur ainsi que les bibliothèques pour l'éditeur de liens :

```
3  CC     = cc
4  CFLAGS = -O
5  LDLIBS = -lm
6
7  all : prog
8
9  prog : fichier1.o fichier2.o fichier3.o
10 ————→$(CC) $(CFLAGS) -o prog fichier1.o fichier2.o fichier3.o $(LDLIBS)
```

Plus généralement, on utilise des variables pour tout ce qui peut être amené à changer : compilateur, options de compilation, bibliothèques, etc. Quelques variables habituellement utilisées dans un fichier `Makefile` sont indiquées dans le tableau 1. L'utilisation de ces noms de variables n'a rien d'obligatoire mais il s'agit d'une convention très répandue qui facilite la compréhension des fichiers `Makefile` et qu'il est donc préférable de respecter. On peut bien

entendu définir ses propres variables avec les noms de son choix si aucune de ces variables ne correspond.

Variable	Signification
`CC`	compilateur C
`CFLAGS`	options du compilateur C
`CPP`	préprocesseur C
`CPPFLAGS`	option du préprocesseur C
`CXX`	compilateur C++
`CXXFLAGS`	options du compilateur C++
`LD`	éditeur de liens
`LDFLAGS`	options de l'éditeur de liens
`LDLIBS`	bibliothèques pour l'éditeur de liens

TABLE 1 – *Variables habituellement utilisées dans un fichier* `Makefile`

3.3 Quelques cibles utiles

Une cible n'est pas toujours un nom de fichier. Cela peut être également une chaîne de caractères quelconque permettant soit la construction de plusieurs dépendances (c'est le cas de la cible `all` que nous allons aborder au paragraphe 3.3.1) soit l'exécution de commandes sans condition de dépendance (c'est le cas de la cible `clean` que nous allons aborder au paragraphe 3.3.2).

Une cible `cible` se construit grâce à la commande :

```
$ make cible
```

3.3.1 La cible `all`

Notre exemple n'aboutit à la création que d'un seul fichier exécutable, `prog`. La cible `prog` est donc la première dans le fichier `Makefile`. Mais comment faire si l'on doit construire deux fichiers exécutables, `prog1` et `prog2`, puisque seule une des deux cibles correspondantes pourra figurer en première position dans le fichier `Makefile` ?

La cible `all` permet de résoudre ce problème, en la plaçant en premier dans le fichier `Makefile` et en la faisant dépendre de `prog1` et de `prog2` :

```
all : prog1 prog2
```

La cible `all` n'a généralement pas de commande associée car elle n'est utilisée que pour construire plusieurs autres cibles.

3.3.2 La cible `clean`

La cible `clean` est utilisée pour faire le ménage. Sa fonction est de supprimer tous les fichiers qui peuvent être recréés afin de ne conserver que ceux qui sont indispensables. Dans notre exemple, on peut supprimer le fichier exécutable `prog` et les fichiers objets `*.o`. Généralement, on en profite pour supprimer un éventuel fichier `core`.

Pour notre exemple, la cible `clean` s'écrit donc ainsi :

```
21 clean :
22 ⟶rm -f prog *.o core
```

La cible `clean` n'a généralement pas de dépendances.

L'option `-f` de la commande `rm` permet d'éviter l'affichage d'un message d'erreur si l'un des fichiers n'existe pas (ce qui est normalement le cas pour le fichier `core`).

3.4 Le fichier `Makefile` utilisant ces propriétés

Voici donc ce à quoi ressemble maintenant le fichier `Makefile` pour notre exemple :

```
1  # fichier Makefile permettant de compiler prog
2
3  CC     = cc
4  CFLAGS = -O
5  LDLIBS = -lm
6
7  all : prog
8
9  prog : fichier1.o fichier2.o fichier3.o
10 ⟶$(CC) $(CFLAGS) -o prog fichier1.o fichier2.o fichier3.o $(LDLIBS)
11
12 fichier1.o : fichier1.c
13 ⟶$(CC) $(CFLAGS) -c fichier1.c
14
15 fichier2.o : fichier2.c
16 ⟶$(CC) $(CFLAGS) -c fichier2.c
17
18 fichier3.o : fichier3.c
19 ⟶$(CC) $(CFLAGS) -c fichier3.c
20
21 clean :
22 ⟶rm -f prog *.o core
```