Université d'Angers
Faculté des Sciences
2 Boulevard Lavoisier
49045 Angers Cedex 01

Cours Make
Utilisation

M2 Mention CCI

Version 1.0
8 janvier 2006

Jean-Michel Richer
Bureau H206
Tel : 02-41-73-52-34
Email : Jean-Michel.Richer@univ-angers.fr

Table des matières

Chapitre 1

Make

1.1 Make

1.1.1 Rappel

`make` est une commande UNIX qui a pour but d'automatiser la compilation de programmes modulaires. Cet utilitaire réduit le temps de construction d'un programme en ne recompilant que ce qui est nécessaire. On part du principe qu'un fichier objet `.o` n'a pas besoin d'être recompilé si le fichier source `.c` dont il découle n'a pas été modifié. Dans le cadre de gros projets de développement qui se composent de nombreux modules (ou packages) il est pénalisant de devoir recompiler l'ensemble des fichiers de chaque module pour obtenir un nouvel exécutable si on a modifié qu'un fichier d'un module. `make` se charge donc de ne recompiler que ce qui a été modifié en fonction des règles définies par le programmeur.

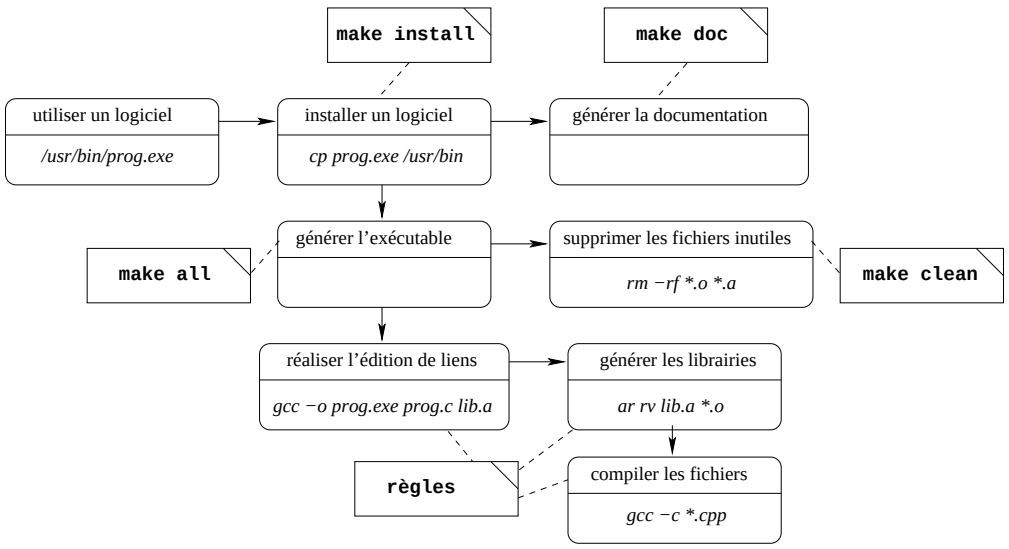

Fig. 1.1 – Processus d'installation d'un logiciel

`make` permet également d'automatiser l'ensemble du processus d'installation d'un logciel en passant par la compilation, l'installation de l'exécutable dans le répertoire approprié, la génération de la documentation, etc (cf. figure 1.1).

1.1.2 les règles

Les règles constituent le moyen de définir les relations de dépendance entre fichier source et objet. Une règle a la forme générique suivante :

```
cible    : sources
           action
```

- *cible* représente soit un programme à générer, soit un identifiant,
- *sources* est composé d'identifiants et/ou de fichiers sources,
- *action* représente la ou les commandes à exécuter.

Les identifiants sont des chaînes de caractères sans espace qui font référence à des noms de cibles que l'on exécutera obligatoirement, elles ne sont donc pas soumises à condition. Il existe un certain nombre d'identifiants par défaut qui ont une signification particulière :

all est exécuté par défaut si on lance la commande `make` sans arguments. Il correspond en quelque sorte au `main` en langage C.

clean est destinée à supprimer tous les fichiers inutiles une fois l'exécutable obtenu. En général on supprime tous les fichiers objets une fois que le programme est généré.

install permet d'installer le programme dans un répertoire approprié

dep gère les relations de dépendances de fichiers entête `.h` entre fichiers `.c`

1.1.3 utilisation de variables

Il est parfois intéressant de pouvoir utiliser des variables qui peuvent être instanciées en fonction des besoins de l'utilisateur. Une variable contient une chaine de caractère. On utilise une variable grâce à la syntaxe `$(VAR)` ou `${VAR}` :

```
RM = rm -rf
OBJS = a.o b.o c.o
GCC = gcc

prog.exe : $(OBJS)
    $(GCC) -o prog.exe $(OBJS)

clean:
    $(RM) *.o
```

Il existe des variables prédéfinies liées aux règles, ainsi

- `$@` représente nom complet de la cible
- `$*` représente le nom de la cible sans suffixe (dans le cas de l'utilisation de pattern, voir plus loin)
- `$<` représente le premier fichier source
- `$?` représente l'ensemble des fichiers sources

1.1.4 utilisation de conditions

On peut introduire des tests de condition afin de paramètrer des variables, comme par exemple :

```
ifeq "$(OS)" "linux 32"
   ARCH=linux
endif

ifeq ($(ARCH),linux)
  CC = gcc
else
  CC = cc
endif
```

Il existe également la conditionnelle `ifneq` qui fonctionne sur le même principe.

1.1.5 fonctions permettant de transformer du texte

Ces fonctions s'utilisent sous la forme suivante :

```
$(fonction arguments)
```

Voici un aperçu des fonctions disponibles :
- `$(subst from,to,text)` replace toute occurrence de `from` par `to` dans `text`
- `$(findstring find,in)` recherche la présence de `find` dans `in`. La fonction renvoie `find` si il existe dans `in`, dans le cas contraire on retourne une chaîne vide.
- `$(addprefix prefix,names)` ajoute le préfixe `prefix` devant chaque nom que contient la liste `names` où les éléments sont séparés par des espaces.
- `$(addsuffix suffix,names)` ajoute le suffixe `suffix` dérrière chaque nom que contient la liste `names` où les éléments sont séparés par des espaces.

Exemple 1 - On désire générer automatiquement l'ensemble des fichiers `.o` à partir de la liste des fichiers `.c` stokés dans la varibale `SOURCES` :

```
SOURCES = fichier1.c fichier2.c fichier3.c
OBJS = ($subst .c,.o,$(SOURCES))
```

on peut également écrire :

```
OBJS = $(SOURCES:.c=.o)
```

Exemple 2 - Les fichiers sources sont dans le répertoire courant et on désire stocker les fichiers objets dans le répertoire `obj` :

```
SOURCES = fichier1.c fichier2.c fichier3.c
OBJS = $(addprefix obj/,($subst .c,.o,$(SOURCES)))

obj/%.o: %.c
      gcc -c $< -o $@
```

1.1.6 les régles particulières (Pattern)

Il est possible de définir des règles particulières en utilisant le symbole `%`. Par exemple pour indiquer qu'à tout fichier d'extension `.o` correspond un fichier d'extension `.c` et qu'il faudra compiler le fichier source `.c` afin d'obtenir le fichier cible `.o`, on écrira :

```
%.o : %.c
   gcc -c $< -o $@
```

1.1.7 inclusion de fichiers

il est possible d'inclure d'autres fichiers qui contiennent des commandes `make` à l'intérieur d'un fichier `make`, il suffit d'utiliser la commande `include` :

```
include nom-du-fichier
```

1.1.8 invoquer make

Il existe plusieurs manière d'invoquer `make` :

make cible définitions-de-variables `-f` *makefile*

par exemple :

```
make
```

recherche l'existence dans le répertoire courant d'un fichier nommé `makefile` ou `Makefile` et commence par exécuter la cible nommée `all`.

```
make install GCC=gcc -f mymake
```

La commande précédente recherche le fichier `mymake` qui est supposé contenir des commandes `make` puis exécute la cible `install` en assignant une valeur à la variable `GCC`.

Exemple 3 - Voici un exemple de fichier `make` qui a pour but de compiler un exécutable `prog.exe` composé de fichiers sources situés dans le répertoire `src`. Les fichiers objets générés lors de la compilation des fichiers sources sont placés dans le répertoire `obj`.

```
GCC=gcc
CFLAGS=-Wall -ggdb
OFLAGS=-ggdb
RM=rm -rf

OBJS=a.o b.o c.o d.o prog.o

all: prog.exe

prog.exe : $(addprefix obj/,$(OBJS))
        $(GCC) $(OFLAGS) $? -o obj/$@

obj/%.o : src/%.c
        $(GCC) -c $< $(CFLAGS) -o $@

clean:
        $(RM) obj/*
```

1.1.9 hierarchie de makefile

Dans le cas où on doit compiler des fichiers situés dans des répertoires différents (cas d'une programmation modulaire), il est préférable de concevoir un `makefile` dans le répertoire principal du projet et d'autres makefile dans les répertoires sources :

```
\home\toto\projet\
+- makefile
+- module1\
|  +- fichier1.c
|  +- fichier2.c
|  +- makefile
+- module2\
|  +- fichier3.c
|  +- fichier4.c
|  +- makefile
```

Le fichier `makefile` situé dans le répertoire du projet appelera les fichiers `makefile` situés dans les sous-répertoires :

```
all : module1 module2

module1:
        @cd module1 ; \
        make -f makefile ; \
        cd ..

module2:
        @cd module2 ; \
        make -f makefile ; \
        cd ..
```

1.1.10 Le cas Java

En Java, le compilateur `javac` gère automatiquement les dépendances entres fichiers grâce aux instructions `import`. Il suffit normalement de compiler le fichier principal qui comporte la fonction `main` et le compilateur recompilera les fichiers importés.

Cependant, si on modifie un fichier source il peut être nécessaire de recompiler tous les fichiers qui importent ce fichier source.