Makefile & *make* *

version 1.3

Alban Mancheron
<alban.mancheron@martinique.univ-ag.fr>

3 *avril* 2007

1 Présentation

Ce que l'on appelle *Makefile* est en fait un fichier utilisable par la commande *make*. Par convention, ce fichier se nomme *Makefile*. Il est possible de le nommer autrement, mais alors il faudra spécifier le nom de fichier à l'outil *make*. Cet outil permet l'éxécution de scripts conditionnés par des dépendances.

1.1 Intérêt

Lors du développement d'une application, aussi petite soit-elle, nous sommes amenés à répéter un très grand nombre de fois certaines commandes (compilation, sauvegarde, effacement, ...). L'intérêt premier de l'outil *make* est de simplifier ces diverses commandes par une syntaxe plus courte et plus générique. De plus, certaines opérations comportent un riques de « mauvaise manipulation » (qui n'a jamais effacé un fichier par erreur, sinon perdu tout son travail avec la commande magique : *rm *.**) qui peuvent alors être évitées. En outre, lorsque le développement est modulaire (ce qui est quasiment tout le temps le cas), l'ordre de compilation peut être source d'erreurs si l'on ne gère pas les phénomènes de dépendances. Il est possible de laisser cette gestion à l'outil *make*. Cela fait donc trois bonnes raisons d'utiliser les fichiers *Makefile*. Enfin, lorsque l'on diffuse une application, plutôt que d'essayer de fournir les versions compilées sous toutes les plateformes existantes, ou encore de laisser le destinataire saisir toutes les commandes de compilation et d'installation, il suffit de fournir les fichiers sources ainsi que le *Makefile*.

1.2 Convention

Avant de continuer plus avant, il est préférable de fixer quelques conventions :
- sauf précision explicite (*i.e.* : en <u>souligné</u>), nous utiliserons indifféremment {*make*, outil *make*, *Makefile*, fichier *Makefile*} ;
- les exemples présentés respecterons les normes GNU, et sont valides sous Unix/Linux.

2 *Makefile* de base

Dans cette partie sont présentées les notions permettant de réaliser un *Makefile* minimal.

2.1 Structure du fichier

On peut découper un <u>fichier *Makefile*</u> en deux parties : la déclaration de variables et la déclaration des cibles.

* Relecture : Erwan Moreau <erwan.moreau@univ-nantes.fr>.

2.1.1 Les Variables

Les variables sont identifiées par un nom unique (par convention en majuscule). Elles sont non typées, et sont substituées par leur valeur lors de leur utilisation. Leur principal intérêt est de rendre le *Makefile* portable vers n'importe quel environnement (Linux/Unix, Windows, Mac, Sparc, ...). Néanmoins, elles permettent aussi de faciliter l'élaboration du *Makefile*.
La déclaration et l'affectation d'une variable suit le schéma :

```
<MA_VARIABLE> = <SA_VALEUR>
```

Pour utiliser une variable, il faut lui substituer sa valeur :

```
$(<MA_VARIABLE>)
```

Par exemple :

```
CC = gcc

exemple1.o : exemple1.c
	$(CC) -c exemple1.c
```

Il est aussi possible d'opérer des opérations sur cette substitution.
Par exemple[1] :

```
CC       = gcc
NOM_FICH = exemple1.c

$(NOM_FICH:.c=.o) : $(NOM_FICH)
	$(CC) -c $(NOM_FICH)
```

Enfin, il existe plusieurs variables prédéfinies telles que : `$^`, `$@`, `$<`. Nous reverrons leur utilisation ultérieurement.

2.1.2 Les Cibles

Les cibles sont la partie « utile » du *Makefile*. Leur déclaration suit le schéma suivant :

```
<CIBLE_1> <CIBLE_2> ... <CIBLE_X> : <DEP_1> <DEP_2> ... <DEP_Y>
        <ACTION_1>
        <ACTION_2>
           ...
        <ACTION_Z>
```

Cela signifie que pour « atteindre » une des cibles `<CIBLE_1>`, `<CIBLE_2>`, ..., `<CIBLE_X>`, il faut d'abord « atteindre » chacune des dépendances `<DEP_1>`, `<DEP_2>`, ..., `<DEP_Y>`. Si une des dépendance a été actualisée, ou si la cible ne correspond pas à un fichier existant dans le répertoire courant, alors il faut actualiser la cible en exécutant les actions `<ACTION_1>`, `<ACTION_2>`, ..., `<ACTION_Z>`. Pour atteindre une cible, il faut exécuter l'utilitaire *make* suivi de la cible à atteindre. Si aucune cible n'est donnée, par défaut, *make* prendra la première cible du fichier *Makefile*. Ce fonctionnement est illustré sur l'exemple 1.

Fichier 1 – Un exemple simple

```
CC        = gcc
PROG_NAME = exemple
SOURCE    = fich1.c fich2.c fich3.c

all : $(PROG_NAME)

```

[1] équivalent à l'exemple précédent.

```
fich1.o : fich_1.c
        $(CC) -c fich1.c

fich2.o : fich_1.c fich_2.c
        $(CC) -c fich2.c

fich3.o : fich1.c fich2.c fich3.c
        $(CC) -c fich3.c

$(PROG_NAME) : $(SOURCE:.c=.o)
        $(CC) $(SOURCE:.c=.o) -o $(PROG_NAME)
```

Si je lance *make* (ce qui est équivalent ici à `make all`), pour atteindre la cible `all`, il faut vérifier si la dépendance `exemple` est à jour. Ceci équivaut à lancer `make exemple`. Lançons alors `make exemple`. Ceci revient à « atteindre » la cible `exemple` (après traduction des caractères alchimiques) :

```
exemple : fich1.o fich2.o fich3.o
        gcc fich1.o fich2.o fich3.o -o exemple
```

Vérifier que `fich1.o`, `fich2.o` et `fich3.o` sont à jour revient à lancer successivement `make fich1.o ; make fich2.o ; make fich3.o`. Simulons `make fich1.o`, nous essayons alors d'« atteindre » la cible `fich1.c`. Il n'existe pas de cible `fich1.c` dans le *Makefile*. Donc la dépendance est considérée comme satisfaite. S'il n'existe pas de fichier intitulé `fich1.o` dans le répertoire courant, ou s'il existe mais que sa date[2] de dernière modification est plus ancienne que la date de dernière modification de `fich1.c`, alors, il faut actualiser le fichier `fich1.o`, et ce en exécutant la liste d'action donnée par la cible, à savoir : `gcc -c fich1.c`. Idem pour `fich2.o` et `fich3.o`.

La dépendance de `exemple` est alors considérée comme satisfaite. S'il n'existe pas de fichier intitulé `exemple` dans le répertoire courant, ou s'il existe mais que sa date de dernière modification est plus ancienne que la date de dernière modification de `fich1.o` ou celle de `fich2.o` ou encore celle de `fich3.o`, alors il faut actualiser le fichier `exemple`, et ce en exécutant la liste d'action donnée par la cible, à savoir : `gcc fich1.o fich2.o fich3.o -o exemple`.

La dépendance de `all` est alors considérée comme satisfaite. Comme il n'existe pas de fichier intitulé `all` dans le répertoire courant, il faut l'actualiser, et ce en exécutant la liste d'action donnée par la cible, à savoir rien.
Si maintenant je modifie le fichier `fich3.c`, et que je relance la commande *make*, alors je vais engendrer les commandes :

```
gcc -c fich3.c
gcc fich1.o fich2.o fich3.o -o exemple
```

Si cette fois je modifie le fichier `fich1.c`, et que je relance la commande *make*, alors je vais engendrer les commandes :

```
gcc -c fich1.c
gcc -c fich2.c
gcc -c fich3.c
gcc fich1.o fich2.o fich3.o -o exemple
```

2.2 Lisibilité et Astuces

Cette partie présente quelques fonctionnalités avancées de la syntaxe des fichiers *Makefile*. Bien qu'à première vue, le « code » fourni puisse sembler difficile d'accès, il ne s'agit que de « trucs & astuces ». Aussi, prenez courage et temps, c'est un investissement.

[2] A comprendre au sens large – date & heure.

2.2.1 Comment taire nos commentaires

Comme tout fichier de programmation, le *Makefile* peut être[3] structuré. Cela devient rapidement nécessaire. En effet, la syntaxe du *Makefile* a tendance à vite devenir hautement alchimique. C'est pourquoi il peut être utile de commenter son *Makefile*, et comme les auteurs de ce fabuleux outil ont pensé à tout, c'est possible grâce au symbole #. Ainsi un commentaire commence par # et se termine avec la fin de la ligne.

2.2.2 Les Variables prédéfinies

Nous avons vu précédemment qu'il existait quelques variables prédéfinies. Parmi celles-ci nous allons voir plus en détail `$^`, `$@`, `$<`. Soit la cible :

```
cible : dep1 dep2 dep3
	@echo $@
	@echo $^
	@echo $<
```

la commande `make cible` produira[4] l'affichage suivant[5] :

```
cible
dep1 dep2 dep3
dep1
```

Ces variables sont donc dépendantes du contexte. `$@`, `$^` et `$<` prennent respectivement pour valeur la cible en cours, l'ensemble des dépendances de la cible en cours, la première dépendance de la cible en cours.

2.2.3 Quelques Cibles bien pratiques

Le fichier 2 donne en vrac quelques lignes qui sont utiles dans la plupart des *Makefile*.

Fichier 2 – Un exemple plus complexe

```
##########################################################################################
#                                                                                        #
#  Quelques infos :                                                                      #
#    - Nom Prénom                                                                        #
#    - Date                                                                              #
#    - Intitulé du projet                                                                #
#    - Matière/Encadrant                                                                 #
#                                                                                        #
#  Utilisation :                                                                         #
#    - make clean (supprime les fichiers issus de la programmation)                      #
#    - make archive (crée une archive contenant les sources de l'application)            #
#    - make arch (crée une archive pour la diffusion de l'application)                   #
#    - make save (crée une sauvegarde dans un répertoire au format JJ_MM_AAAA/)          #
#                                                                                        #
##########################################################################################

                                        #############
                                        # Variables #
                                        #############

########
# La Compilation
CC        = $(MSQ)gcc
FLAGS     = -Wall
LIBS      =
```

[3]doit être ?

[4]N'hésitez pas à le tester.

[5]Rappel : en B-Shell, une commande précédée du caractère @ n'apparaît pas sur la sortie standard.

```

########
# Les Fichiers
PROGNAME = exemple # Nom de l'application
ARCH     = $(PROGNAME).tar.gz
ARCH_SRC = $(PROGNAME)_src.tar.gz
SOURCE   = fich1.c fich2.c fich3.c \ # Ce symbole permet de passer à la ligne
           fich4.c                   # Je suppose ici que je programme en 'C'
HEADERS  = config.h tools.h          # Si j'ai des fichiers .h particuliers
THIS     = Makefile
VERSION  = `eval\ date\ +%d_%m_%Y`

########
# Le Shell
MSQ      = @
RM       = $(MSQ)rm -f
LS       = $(MSQ)ls -a --color
MV       = $(MSQ)mv
MKDIR    = $(MSQ)mkdir -p
CLEAR    = $(MSQ)clear
MAKE     = $(MSQ)make
TAR      = $(MSQ)tar -czf

########
# La VF
MSG      = $(MSQ)echo -e
MSG1     = $(MSG) "Compilation de $<\t==>\t$@"
MSG2     = $(MSG) "Edition de lien de $^\t==>\t$@"
MSG_OK   = $(MSG) "\t\t\t.....................Ok"

                                   ##########
                                   # Cibles #
                                   ##########

########
# Cibles Usuelles
all : $(PROGNAME)
arch : $(ARCH)
archive : $(ARCH_SRC)
save : $(VERSION)

########
# Les Archives
$(ARCH) : $(THIS) $(PROGNAME) $(HEADERS) $(SOURCE:.c=.h) $(SOURCE:.c=.o)
	$(MSG) "Création de l'archive $@\n\tAjout des fichiers : $^"
	$(TAR) $@ $^
	$(MSG_OK)

$(ARCH_SRC) : $(THIS) $(SOURCE) $(SOURCE:.c=.h) $(HEADERS)
	$(MSG) "Création de l'archive $@\n\tAjout des fichiers : $^"
	$(TAR) $@ $^
	$(MSG_OK)

$(VERSION) : $(ARCH_SRC)
	$(MSG) "Création du répertoire de sauvegarde : $@"
	$(RM) -rf $@
	$(MKDIR) $@
	$(MSG) "Sauvegarde de l'Archive $<"
	$(MV) $< $@
	$(MSG_OK)

########
# Le Nettoyage
cls :
	$(MSG) "On fait le ménage"
	$(RM) \#*
```

```
	$(RM) *~
	$(RM) core
	$(RM) *.o
	$(RM) $(PROGNAME)
	$(RM) $(ARCH)
	$(RM) $(ARCH_SRC)
	$(MSG_OK)

clean : cls
	$(CLEAR)
	$(LS)

########
# La Compilation
fich1.o : fich1.c
	$(MSG1)
	$(CC) -c $(FLAGS) $(LIBS) $<
	$(MSG_OK)

fich2.o : fich2.c
	$(MSG1)
	$(CC) -c $(FLAGS) $(LIBS) $<
	$(MSG_OK)

fich3.o : fich3.c
	$(MSG1)
	$(CC) -c $(FLAGS) $(LIBS) $<
	$(MSG_OK)

fich4.o : fich4.c
	$(MSG1)
	$(CC) -c $(FLAGS) $(LIBS) $<
	$(MSG_OK)

$(PROGNAME) : $(SOURCE:.c=.o)
	$(MSG2)
	$(CC) $(FLAGS) $(LIBS) $^ -o $@
	$(MSG_OK)
```

3 *Makefile* avancé

Dans le fichier précédent, nous pouvons constater que dans la dernière partie[6] chaque fichier d'extension `.o` est généré de la même manière à partir du fichier d'extension `.c` correspondant. De plus, nous pouvons remarquer que nous n'avons pas tenu compte des dépendances entre les fichiers. Voyons comment remédier à ces deux aspects.

3.1 Généralisation des cibles

Plutôt que d'écrire une cible par module, il est possible d'écrire une cible générique qui aura le même comportement. Cette fonctionnalité passe par une cible particulière : `.SUFFIXES` qui sert de descripteur d'extensions. Ici nous ne considérons que les extensions `.c`, `.h` et `.o`. Nous devons donc modifier notre dernière partie en ajoutant cette cible :

```
.SUFFIXES : .c .h .o
```

Enfin, il nous faut réécrire la cible générique qui génère le fichier d'extension `.o` à partir du fichier d'extension `.c` correspondant. Cette cible s'écrit :

[6] Intitulée « La Compilation ».

```
.c.o :
	$(MSG1)
	$(CC) -c $(FLAGS) $(LIBS) $<
	$(MSG_OK)
```

3.2 Pré-processeur

Pour règler le problème des dépendances entre fichiers, quand c'est possible, le plus simple est de laisser le compilateur les générer. Sinon, nous pouvons les décrire manuellement dans un fichier en suivant la syntaxe : <CIBLE> : <DEP_1> <DEP_2> ... <DEP_N>.
Pour les générer automatiquement, il faut utiliser l'option `-MM` avec `gcc`.
Ensuite, il faut préciser au *Makefile* qu'il doit tenir compte du fichier décrivant ces dépendances. Cela se fait avec la commande de pré-processeur `include`.

Supposons que nous ayons 4 fichiers `fich1.c`, `fich2.c`, `fich3.c` et `fich4.c`. Alors, nous pouvons calculer leurs dépendances en utilisant :
`gcc -MM fich1.c fich2.c fich3.c fich4.c >> .depend`
Ainsi, il nous suffit d'écrire dans le *Makefile* : `include .depend`.

Nous pouvons automatiser cela en demandant au *Makefile*, lors de l'éxécution de *make*, d'inclure le fichier `.depend` s'il existe, sinon de générer le fichier de dépendances, puis de relancer l'exécution de *make*. Il existe une commande de pré-processeur permettant de vérifier l'existence d'un fichier :

```
ifeq ($(wildcard <NOM_FICH>), )
<si le fichier n'existe pas>
else
<si le fichier existe>
endif
```

Pour réaliser cette automatisation, il nous suffit alors d'inclure au début des cibles :

```
ifeq ($(wildcard .depend), )
all : .depend
	make --no-print-directory
else
all : $(PROGNAME)
endif
```

Puis de rajouter la cible :

```
.depend : $(SOURCE)
	$(MSG) "Recherche des dépendances des fichiers ..."
	$(RM) .depend
	$(CC) -MM $^ >> $@
	$(MSG_OK)
```

Ce qui nous donne le fichier final :

Fichier 3 – Un exemple complet

```
################################################################################
#                                                                              #
#  Quelques infos :                                                            #
#    - Nom Prénom                                                              #
#    - Date                                                                    #
#    - Intitulé du projet                                                      #
#    - Matière/Encadrant                                                       #
#                                                                              #
#  Utilisation :                                                               #
#    - make clean (supprime les fichiers issus de la programmation)            #
#    - make archive (crée une archive contenant les sources de l'application)  #
#    - make arch (crée une archive pour la diffusion de l'application)         #
```

```
#    - make save (crée une sauvegarde dans un répertoire au format JJ_MM_AAAA/) #
#    - make dep (Calcul des dépendances des fichiers sources)                  #
#                                                                              #
################################################################################

                               #############
                               # Variables #
                               #############

########
# La Compilation
CC       = $(MSQ)gcc
FLAGS    = -Wall
LIBS     =
DEP_FLAG = -MM

########
# Les Fichiers
PROGNAME = exemple # Nom de l'application
ARCH     = $(PROGNAME).tar.gz
ARCH_SRC = $(PROGNAME)_src.tar.gz
SOURCE   = fich1.c fich2.c fich3.c \ # Ce symbole permet de passer à la ligne
           fich4.c                   # Je suppose ici que je programme en 'C'
HEADERS  = config.h tools.h          # Si j'ai des fichiers .h particuliers
THIS     = Makefile
VERSION  = `eval\ date\ +%d_%m_%Y`
DEP_FILE = .depend

########
# Le Shell
MSQ      = @
RM       = $(MSQ)rm -f
LS       = $(MSQ)ls -a --color
MV       = $(MSQ)mv
MKDIR    = $(MSQ)mkdir -p
CLEAR    = $(MSQ)clear
MAKE     = $(MSQ)make
TAR      = $(MSQ)tar -czf

########
# La VF
MSG      = $(MSQ)echo -e
MSG1     = $(MSG) "Compilation de $<\t==>\t$@"
MSG2     = $(MSG) "Edition de lien de $^\t==>\t$@"
MSG_OK   = $(MSG) "\t\t\t......................Ok"

                                 ##########
                                 # Cibles #
                                 ##########

########
# Cibles Usuelles
ifeq ($(wildcard $(DEP_FILE)), )
all : $(DEP_FILE)
	$(MAKE) --no-print-directory
else
all : $(PROGNAME)
endif
dep : $(DEP_FILE)
arch : $(ARCH)
archive : $(ARCH_SRC)
save : $(VERSION)

########
# Les Archives
$(ARCH) : $(THIS) $(PROGNAME) $(HEADERS) $(SOURCE:.c=.h) $(SOURCE:.c=.o)
```

```
	$(MSG) "Création de l'archive $@\n\tAjout des fichiers : $^"
	$(TAR) $@ $^
	$(MSG_OK)

$(ARCH_SRC) : $(THIS) $(SOURCE) $(SOURCE:.c=.h) $(HEADERS)
	$(MSG) "Création de l'archive $@\n\tAjout des fichiers : $^"
	$(TAR) $@ $^
	$(MSG_OK)

$(VERSION) : $(ARCH_SRC)
	$(MSG) "Création du répertoire de sauvegarde : $@"
	$(RM) -rf $@
	$(MKDIR) $@
	$(MSG) "Sauvegarde de l'Archive $<"
	$(MV) $< $@
	$(MSG_OK)

########
# Le Nettoyage
cls :
	$(MSG) "On fait le ménage"
	$(RM) \#*
	$(RM) *~
	$(RM) core
	$(RM) *.o
	$(RM) $(PROGNAME)
	$(RM) $(ARCH)
	$(RM) $(ARCH_SRC)
	$(RM) $(DEP_FILE)
	$(MSG_OK)

clean : cls
	$(CLEAR)
	$(LS)

########
# La Compilation
.SUFFIXES : .c .h .o

$(DEP_FILE) : $(SOURCE)
	$(MSG) "Recherche des dépendances des fichiers ..."
	$(RM) $(DEP_FILE)
	$(CC) $(DEP_FLAG) $^ >> $@
	$(MSG_OK)

.c.o :
	$(MSG1)
	$(CC) -c $(FLAGS) $(LIBS) $<
	$(MSG_OK)

$(PROGNAME) : $(SOURCE:.c=.o)
	$(MSG2)
	$(CC) $(FLAGS) $(LIBS) $^ -o $@
	$(MSG_OK)
```

4 Et après...

Vous trouverez ici très peu d'information ; tout simplement, parce qu'une fois que l'on sait écrire un *Makefile*, l'objectif est de ne plus avoir à le faire et d'utiliser des outils qui s'en charge pour vous... Toujours pareil, l'informaticien qui travaille ne doit le faire que par fainéantise.

4.1 Quelques références

Pour plus d'information :
- info Makefile
- Linux in a Nutshell, ed. O'Reilly Paris, 1997, Jessica Perry HEKMAN, Alain NADEAU, Traduit par Jean-Michel VANSTEENE.

4.2 Génération automatique et Génie Logiciel

Pour ceux qui sont intéressés, vous pouvez aussi regarder les utilitaires `autoconf` et `automake`, qui génèrent automatiquement un *Makefile*.

Bon Courage...