Programmation Unix 2 – cours n°10

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Janvier 2017

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°10

`Makefile` (suite) :

1. Principes pour les règles
2. Les directives
3. Variables automatiques
4. Blocs de commandes

1 - Principes pour les règles

Rappel : la syntaxe générale d'une règle est

```
cible : dépendances
⟼ commande pour produire la cible
```

ou, de manière équivalente

```
cible : dépendances ; commande
```

`make fonctionne en 2 passes` :

1. il lit tout le fichier et mémorise les règles
2. il recherche la cible demandée

Principe n°1

Toutes les dépendances doivent

- exister en tant que cible dans le `Makefile`
- et/ou correspondre à un fichier existant

Exemple : on crée ce `Makefile` dans un répertoire vide :

```
a : b c
b : c
```

`make` échoue car il n'y a ni cible `"c"` ni fichier `"c"`.

Solutions :

- rajouter une règle `c :`
- ou créer un fichier `"c"` dans le répertoire.

Principe n°2

Si une cible est un fichier existant,
et si cette cible n'a pas de dépendances,
la commande n'est pas exécutée.

Exemple :

```
c :
	echo "do c"
```

```
$ make
do c
$ touch c
$ make
make: "c" est à jour.
```

Principe n°3

Une cible peut ne pas correspondre à un fichier.

Exemple : `all`, `clean`, `depend`

Appelées *cibles factices* (uk: phony targets)

Pour préciser des cibles factices dans le `Makefile` :

```
.PHONY : all clean

all : prog1 prog2

clean :
    rm -f *.o prog1 prog2
```

Intérêt : ne teste pas l'existence du fichier

- évite les problèmes (non exécution, principe n°2)
- efficacité

Principe n°4

`make` affiche chaque commande avant de l'exécuter :

```
a :
    echo "do a"
```

```
$ make
echo "do a"
do a
```

On peut supprimer l'affichage de la commande :

```
a :
    echo "do a"
```

```
$ make --silent
do a
```

Autre méthode :

```
a :
    @echo "do a"
```

```
$ make
do a
```

Principe n°5

Une cible n'est réalisée *qu'une seule fois* ;
si plusieurs dépendances aboutissent à la même cible,
la commande n'est exécutée que la première fois.

Exemple :

```
a : b c
    echo "do a"
b : c
    echo "do b"
c :
    echo "do c"
```

```
$ make --silent
do c
do b
do a
```

Principe n°6

Plusieurs cibles peuvent partager la même règle.

Exemple :

```
a : b
    echo "do a"
b c :
    echo "do b or c"
```

```
$ make --silent
do b or c
do a
```

Principe n°7

Une cible peut apparaître plusieurs fois,
avec des dépendances identiques ou différentes ;
les dépendances sont concaténées.

Cependant : un seul bloc de commande par cible.

Ex. :

```
a : b c
    echo "1"
a : c e
b c e :
```

Correct : équivalent à

```
a : b c e
    echo "1"
b c e :
```

Ex. :

```
a : b c
    echo "1"
a : c e
    echo "2"
b c e :
```

Incorrect :
il y a deux blocs de commandes
pour la cible "a".

Principe n°8

Pour mettre plusieurs blocs de commandes pour une même cible,
il suffit de mettre "::" à la place de ":"
pour toutes les occurrences de la cible.

Ex. :

```
a :: b c
    echo "a1"
a :: c e
    echo "a2"
b : ; echo "b"
c : ; echo "c"
e : ; echo "e"
```

```
$ make --silent
b
c
a1
e
a2
```

→ Les blocs sont exécutés dans l'ordre d'apparition.

Principe n°9

"::" rend une cible factice.

Il y a donc 2 façons de déclarer des cibles factices :

```
.PHONY : a

a : ....
    ....
```

ou

```
a :: ....
     ....
```

Principe n°10

Par défaut, `make` s'arrête dès qu'une commande échoue.
On peut désactiver l'arrêt en mettant `"-"` devant le bloc de commandes.

```
a : b c
    @echo "a"
b :
    @echo "b"
c :
    false
```

```
$ make
b
false
make: *** [c] Erreur 1
```

```
a : b c
    @echo "a"
b :
    @echo "b"
c :
    -false
```

```
$ make
b
false
make: [c] Erreur 1 (ignorée)
a
```

2 - Les directives

La commande `make` autorise l'emploi de directives :

- inclusion de fichiers
- variables
- conditionnelles
- etc

Toujours situées en début de ligne
Jamais dans un bloc de commandes.

Inclusion de fichiers

```
include fichier ...
```

Inclut le fichier, mais échoue si absent.

```
-include fichier ...
```

Inclut le fichier ; s'il est absent, ni erreur ni warning.

Usage typique : remplacement de `makedepend` par `gcc -MM` :

```
depend ::
    gcc -MM -std=c99 *.c >| .depend

-include .depend
```

Variable de `make`

```
variable = valeur
```

Définit une variable et sa valeur ; pas besoin d'accoler

```
variable += valeur
```

Concatène la valeur à la fin

Substituer la valeur : `$(variable)` ou `${variable}`
ou `$v` si une lettre

On ne peut modifier une variable dans un bloc de commande

Variables usuelles

```
SHELL = /bin/bash   # Important : shell utilisé par make
CC    = gcc
RM    = rm -f

CFLAGS = -Wall -W -std=c99 -O2
CPATHS = -I.
LFLAGS = -lm -lmabibliotheque
LPATHS = -L.

EXEC = monprogramme
OBJS = module1.o module2.o

$(EXEC) : $(OBJS)
    $(CC) $(OBJS) $(LPATHS) $(LFLAGS) -o $(EXEC)
```

`$(LFLAGS)` après `$(OBJS)`

Conditionnelle d'existence

```
ifdef variable       # ou ifndef
    ....
else                 # optionnel
    ....
endif
```

Inclut les lignes `....` selon que la variable est définie ou pas.

Ex. :

```
CFLAGS = -Wall -W -std=c99 -O2

# Décommenter pour débugger avec gdb
#DEBUG =

ifdef DEBUG
  CFLAGS += -g
endif

toto.o : toto.c
    gcc $(CFLAGS) -c toto.c
```

Conditionnelle de valeur

```
ifeq "valeur1" "valeur2"   # ou ifneq
    ....
else                       # optionnel
    ....
endif
```

Inclut les lignes `....` selon que les valeurs sont égales ou pas.

Variante : `ifeq (valeur1, valeur2)`

Ex. :

```
CFLAGS = -Wall -W

# CVER = ansi ou c99
CVER = c99

ifeq "$(CVER)" "c99"
  CFLAGS += -std=c99
else
  CFLAGS += -ansi
endif

toto.o : toto.c
    gcc $(CFLAGS) -c toto.c
```

3 - Variables automatiques

Variables définies par `make` selon le contexte

`$@`	cible courante
`$?`	dépendances plus récentes que `$@`
`$*`	préfixe de `$@` (vide si extension non reconnue)
`$<`	première dépendance
`$^`	liste des dépendances


S'utilisent uniquement dans le bloc de commandes

Exemple

```
mod1.o : mod1.c
    $(CC) $(CPATHS) $(CFLAGS) -c mod1.c

mod2.o : mod2.c
    $(CC) $(CPATHS) $(CFLAGS) -c mod2.c

prog1 : mod1.o mod2.o
    $(CC) mod1.o mod2.o $(LPATHS) $(LFLAGS) -o prog1
```

Peut se réécrire :

```
mod1.o : mod1.c
mod2.o : mod2.c

mod1.o mod2.o :
    $(CC) $(CPATHS) $(CFLAGS) -c $*.c

prog1 : mod1.o mod2.o
    $(CC) $^ $(LPATHS) $(LFLAGS) -o $@
```

Règle générique

Cible spéciale `.c.o :` (ancienne syntaxe)

tout `.o` dépend du `.c` du même nom

Dans l'exemple précédent,

```
mod1.o : mod1.c
mod2.o : mod2.c

mod1.o mod2.o :
    $(CC) $(CPATHS) $(CFLAGS) -c $*.c
```

devient :

```
.c.o :
    $(CC) $(CPATHS) $(CFLAGS) -c $*.c
```

Cible avec motif

Syntaxe GNU-make (la plus répandue) :

toute cible peut contenir un "%" (au plus),
qui correspond à toute chaîne non vide.

Application : `%.o : %.c` est plus clair que `.c.o :`

L'exemple précédent devient :

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

Suffixes (pour l'ancienne syntaxe)

`make` connaît une liste de suffixes pour $\neq$ langages

Pour les afficher :

```
show ::
    @echo "$(SUFFIXES)"
```

```
$ make show
.out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l
.s .S .mod .sym .def .h .info .dvi .tex .texinfo
.texi .txinfo .w .ch .web .sh .elc .el
```

Vider cette liste (n'altère pas la variable `SUFFIXES`)

```
.SUFFIXES:
```

Ajouter des suffixes à la liste (idem)

```
.SUFFIXES: .pdf .png
```

Exemple

Un `Makefile` qui convertit des `.png` en `.pdf` avec la commande `convert` de `ImageMagick` :

```
.SUFFIXES: .png .pdf

PDFS = pic1.pdf pic2.pdf

all :: $(PDFS)

.png.pdf :
    convert $*.png $*.pdf

clean ::
    rm -f $(PDFS)
```

```
PDFS = pic1.pdf pic2.pdf

all :: $(PDFS)

%.pdf : %.png
    convert $*.png $*.pdf

clean ::
    rm -f $(PDFS)
```

GNU make

La version GNU de `make` (par défaut sous Linux) ajoute de nombreuses possibilités :
http://www.gnu.org/software/make/manual/make.html

Exemple :

```
SRCS = $(wildcard *.png)
PDFS = $(patsubst %.png,%.pdf,$(SRCS))

all :: $(PDFS)

%.pdf : %.png
    convert $*.png $*.pdf

clean ::
    rm -f $(PDFS)
```

4 - Blocs de commande

Dans le bloc de commandes d'une règle :

- chaque ligne non vide commence par tab
- on peut sauter des lignes
- on peut commenter avec #

Le shell utilisé est défini par `SHELL` :

```
SHELL = /bin/bash
```


Toujours définir

Substitutions du shell

"$" déjà utilisé pour les variable de `make`

→ pour les substitutions de shell, on utilise "$$" :

```
SHELL = /bin/bash
FOO = bar

essai ::
    @echo "FOO = $(FOO)"
    @a="salut" ; echo "$$a"
    @echo "3 + 4 = $$((3+4))"
    @echo "la date est $$(date)"
    @echo "Le PID est $$$$"
```

Un shell par ligne

Dans un bloc de commande, `make` exécute chaque ligne dans un nouveau shell.

Inconvénient : perte de variables shell

```
SHELL = /bin/bash
essai ::
    @a="salut"
    @echo "a = $$a"
```

```
$ make essai
a =
```

Solution : réunir les lignes avec "\"

```
SHELL = /bin/bash
essai ::
    @a="salut" ;\
    echo "a = $$a"
```

```
$ make essai
a = salut
```

Makefile dans des sous-répertoires

Soient les sous-répertoires Bim, Bam et Boum ; chacun contient un Makefile avec les cibles all, clean et depend.

Makefile général :

```
SHELL = /bin/bash
MAKE = make

DIRS = Bim Bam Boum

all clean depend ::
    @for D in $(DIRS) ; do \
      (cd "$$D" && $(MAKE) $@) ;\
    done
```