Gnu Make

- utilitaire qui détermine automatiquement les parties d'un programme à recompiler
- peut-être utilisé pour tous les langages de programmation à condition que le compilateur puisse être lancé par une ligne de commande
- peut-être utilisé en dehors des taches de programmation pour toute tache nécessitant une mise à jour partielle de certains fichiers.
- création d'un fichier nommé "makefile" qui décrit les relations entre les fichiers et contient les commandes de mise à jour de ces fichiers
- exécution par la commande "make"

1. Aperçu rapide de make

- un fichier makefile consiste en des **règles** de la forme :

```
cible : dépendances
<tab>  command
<tab>  command
….
```

- *cible* : généralement le nom d'un fichier généré (exécutable, fichier objet), peut aussi être le nom d'une action telle que "clean"
- *dépendances* : fichiers nécessaires en entrée pour générer la cible (généralement plusieurs)
- *commande* : commande permettant de générer la cible.

- Un *makefile* peut aussi contenir du texte autre que des règles.

Exemple : exécutable "edit" dépendant de 8 fichiers objets qui dépendent eux-mêmes de 8 sources C et 3 fichiers d'entête.

Schéma

```
edit : main.o kdb.o command.o display.o insert.o \
      search.o files.o utils.o
      gcc –o edit main.c kdb.c command.c display.c insert.c \
            search.c files.c utils.c

main.o : main.c defs.h
      gcc –c main.c
kdb.o : kdb.c defs.h command.h
      gcc –c kdb.c
command.o : command.c defs.h command.h
      gcc –c command.c
display.o : display.c defs.h buffer.h
      gcc –c display.c
insert.o : insert.c defs.h buffer.h
      gcc –c insert.c
search.o : search.c defs.h buffer.h
      gcc –c search.c
files.o : files.c defs.h buffer.h command.h
      gcc –c files.c
utils.o : utils.c defs.h
      gcc –c utils.c
clean :
      rm edit main.o kdb.o command.o display.o \
            insert.o search.o files.o utils.o
```

• utilisation :
  - générer l'exécutable "edit" : make
  - faire le ménage : make clean (n'est pas exécuté sinon)

• fonctionnement de make :
  - par défaut, make commence par la 1ère cible (edit dans l'exemple), attention : si clean en 1er, effacement par défaut.
  - pour générer cette cible, il doit exécuter les règles pour les fichiers dont "edit" dépend.
  - les ".o" sont recompilés si les sources ou fichier d'entête sont plus récents que le ".o", ou si le ".o" n'existe pas
  - par exemple : modification de insert.c
    - compilation de insert.o
    - édition de liens de edit

    modification de command.h

    compilation de kdb.o, command.o et files.o

    édition de liens de edit

• mais attention : si une cible correspond à un fichier existant, la date de modification de ce fichier sert de référence, sinon l'action sera toujours exécutée.

ex : dans l'exemple précédent, si on remplace `edit : ….` par `all : …` où `all` est une cible dite fictive, l'édition de liens sera toujours effectuée.

• une cible peut apparaître plusieurs fois

`main.o : main.c defs.h`
`  gcc –c main.c`
	idem que
	
	`main.o : main.c`
	`  gcc –c main.c`
	
	`main.o : defs.h`

2. Variables

2.1. variables automatiques

$@ : nom de la cible qui provoque l'exécution de la commande
$< : nom de la 1ère dépendance
$? : nom de toutes les dépendances qui sont plus récentes que la cible
$^ : nom de toutes les dépendances
$* : chaîne de caractères extraite par un filtre

exemple :

main : main.c
    gcc –o $@ $<

2.2. variables utilisateurs

• Utilisation :
- nombreuses répétition de noms de fichiers dans l'exemple (les ".o" pour les dépendances et l'édition de lien de edit)
- risque d'erreurs en cas de modification (ajout ou suppression d'un fichier)
- utilisation de variables (ou de macros) définissant une chaîne de caractères

• affectation :
var = value

référence : $(*var*), on peut omettre les parenthèses si le nom de la variable n'a qu'une seule lettre (H = 3 -> $H)

• Exemple :

```
OBJETS = main.o kdb.o command.o display.o \
      insert.o search.o files.o utils.o
```

puis dans le fichier makefile, remplacement de la liste des ".o" par $(OBJETS), par exemple :

```
edit : $(OBJETS)
      gcc –o edit $(OBJETS)
```

3. Les règles

3.1. Règles explicites

- 1ere règle : règle par défaut
- syntaxe générale

cible : dépendances
    commande1
    …
    - variables du shell : utilisation du caractère $ => $$ car $ réservé pour les références variables : $$HOME
    - pas de limitation de taille de ligne ou "\" puis nouvelle ligne
    - on peut utiliser les caractères génériques : *, ?, […]

ex : all : *.c
    echo *.c
make all affiche tous les fichiers .c du répertoire (mais *.c sinon

exemple :

```
clean :
    rm –f *.o
```

ou

```
objets = $$HOME/es.o $$HOME/main.o
all :
    @for i in $(objets); do \
        echo $$i; \
    done
```

• rem : une variable peut s'appeler *.c sans provoquer alors d'expansion

• attention : objets = *.o => objets est la chaîne *.o
ex : prog : $(objets)
    gcc –c prog $(objets)

- *.o sera expansé en tous les .o du répertoire
- MAIS si les ".o" ont été effacés, '*.o' n'est pas expansé => erreur
- solution possible : fonction "wildcard" et substitution de chaînes

la fonction *wildcard*
• indique qu'il faut explicitement expansé la notation, utilisation : $(wildcard masque)
    - remplacé par une liste de noms de fichiers séparés par des espaces correspondant aux masques.
    - si pas de correspondance => ignoré

exemple :
    objets = $(wildcard *.o)
    si aucun .o, objets est la chaîne vide
MAIS tous les *.o* n'existent peut-être pas => substitution de chaînes *.c* -> *.o*

```
objets = $(patsubst %.c, %.o, $(wildcard *.c))
prog : $(objets)
        gcc –o prog $(objets)
```

3.2. Les règles implicites

- il n'est pas nécessaire d'énoncer les commandes de compilation car il existe une règle implicite de mise à jour d'un ".o" à partir d'un ".c" à l'aide de la commande "cc –c"
- de +, le ".c" est automatiquement ajouté à la liste des dépendances. On peut donc l'omettre.
- si l'on utilise les règles implicites, on peut en plus regrouper les règles selon leurs dépendances plutôt que selon leurs cibles.

exemple :

```
OBJETS = main.o kdb.o command.o display.o \
        insert.o search.o files.o utils.o
edit : $(OBJETS)
        gcc –o edit $(OBJETS)
$(OBJETS) : defs.h
kdb.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

• rem : une variable CC contient le compilation C par défaut, un makefile ne contenant que CC=gcc (redéfinition du compilateur) provoque le changement des règles implicites en gcc -c

3.3. Règles à plusieurs cibles

• écrire plusieurs règles avec une seule cible avec des dépendances identiques et commandes similaires
• utile dans 2 cas :

- ajouter une dépendance à plusieurs cibles
  ex : kdb.o command.o files.o : command.h
- les commandes sont similaires pour toutes les cibles
  ex : bigoutput littleoutput : text.g
  generate text.g -$(subst output ,, $@) > $@
  est équivalent :
  bigoutput : text.g
  generate text.g –big > bigoutput
  littleoutput : text.g
  generate text.g –little > littleoutput

3.4. Définir des règles implicites

utilisation du '%'
exemple :

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

lancement : make prog.o

4. Quelques fonctions

• utilisation $(nom fonction param1, param2, …)
• wildcard : voir plus haut
• subst : $(subst motif, motif de remplacement, texte)
ex : $(subst ien, ienG, Bien tiens, il ne reste plus rien)
➔ BienG tienGs, il ne reste plus rienG
• patsubst : substitution avec possibilité de désigner un motif par %
ex :
$(patsubst version.%, %_version, version.01 version.02 version.03)
➔ 01_version 02_version 03_version
raccourci pour patsubst : $(variable:motif=motif de remplacement)
entrees = inputa inputb
sorties = $(entrees : input% = output%)

• strip : $(strip texte)
rend une chaîne en éliminant les blancs avant et après texte.
• shell : $(shell commande_shell)
$(shell ls *.c)
• filter : $(filter motif, texte)
ex : $(filter %.c, main.c vecteur.h vecteur.c interface.o)
-> main.c vecteur.c
• filter-out : complément de filter, extrait les éléments ne correspondant pas au motif
• sort : trie et élimine les doublons
• foreach, word, wordlist, dir, notdir
$(foreach x, pascal alfred, je m'appelle $(x)) => je m'appelle pascal je m'appelle alfred
$(wordlist 2, 4, salut tout le monde comment ca va) => tout le monde
$(dir Sources/fich.c Objets/fich.o) -> Sources/ Objets/
$(notdir Sources/fich.c Objets/fich.o) ->fich.c fich.o

5. Les conditionnelles

exemple :
libs_pour_gcc = -lgnu
normal_libs = -libc

```
prog : $(objets)
ifeq ($(CC), gcc)
        $(CC) –o prog $(objets) $(libs_pour_gcc)
else
        $(CC) –o prog $(objets) $(normal_libs)
endif
```

➔ autre forme *ifneq*
➔ 'else' facultatif
➔ existence d'une variable ou valeur vide
ifdef variable
ifndef variable

6. Le fichier Makefile

• contient 5 éléments :
- des règles explicites décrivant la manière de mettre à jour un ou plusieurs fichiers
- des règles implicites décrivant la manière de mettre à jour une classe de fichiers décrits par une propriété de leurs noms
- des définitions de variables (ou macros)
- des directives permettant de :
  - lire un autre fichier makefile (include)
  - décider d'utiliser ou non une partie du fichier makefile, selon les valeurs des variables (conditionnelle)
- des commentaires commençant par '#'

• make essaye de trouver les noms de fichiers dans l'ordre suivant : GNUmakefile, makefile, Makefile
- s'il ne trouve pas de makefile :
  - si pas de cible passé sur la ligne de commande => erreur,
  - sinon make essaye de la générer avec ses règles implicites
- si l'on veut utiliser un nom différent => option '-f nom', si on utilise plusieurs fois l'option, les fichiers sont concaténés dans l'ordre spécifié.
-

• l'inclusion d'autres makefiles :

include nom_fichier1 nom_fichier2 … attention : pas de tabulation sinon = commande

- utile pour les makefiles utilisant les mêmes variables ou certaines règles implicites
- recherche le fichier dans le répertoire courant puis dans /usr/local/include, /usr/gnu/include, /usr/include
- on peut donner d'autres chemin de recherche par –I
- autre méthode, au lancement : -include filename
- ne génère pas de message d'erreur si n'existe pas

7. Recherche de répertoire pour les dépendances

• permet d'organiser les fichiers aisément sans modifier les makefiles mais juste les chemins de recherche

• variable VPATH

VPATH = chemin chemin

séparateur ' ' ou ':'

exemple :

```
VPATH = src:../headers
prog.o : prog.c
```

interprété comme prog.o : src/prog.c

• directive vpath
- chemin de recherche pour une classe particulière de fichiers
- 3 formes :
  - vapth masque répertoires
  - vpath masque // efface les chemins associés au masque
  - vpath // efface tous les chemins

exemple : vpath %.h ../headers

8. Les cibles fictives

• n'est pas un nom de fichier
• permet d'exécuter certaines commandes par un appel explicite
ex :

```
clean :
      rm –f *.o temp
```

⇨ utilisation : make clean

attention : si il existe un fichier "clean", les commandes ne seront jamais exécutées car "clean" est toujours à jour (pas de dépendance)

⇨ solution : déclarer explicitement "clean" comme cible fictive :

.PHONY : clean
clean :
rm ….

(en anglais, phony signifie charlatan, inventé, faux, bidon)
• autre utilisation : génération de plusieurs cibles finales
exemple :

```
all : prog1 prog2 prog3
.PHONY : all
prog1 : prog1.o utils.o
      gcc –o prog1 prog1.o utils.o
prog2 : prog2.o
      gcc –o prog2 prog2.o
prog3 : prog3.o sort.o utils.o
      gcc –o prog3 prog3.o sort.o utils.o
```

• une cible fictive peut dépendre d'autres cibles fictives
exemple :

```
.PHONY : cleanall cleanobj cleandiff
cleanall : cleanobj cleandiff
      rm –f program
cleanobj :
      rm –f *.o
cleandiff :
      rm –f *.diff
```

9. Masques de règles statiques

• lorsque noms de cibles et noms de dépendances sont liés

cibles … : masque de cible : masques des dépendances
commandes

ex :

```
objets = prog.o bar.o
all : $(objets)
$(objets) : %.o : %.c
      gcc –c $< -o $@
```

• toutes les cibles doivent pourvoir correspondre au masque de cible, sinon -> warning
il est également possible d'utiliser la fonction "filter"

exemple :

```
fichiers = foo.elc bar.o lose.o
```

```
$(filter %.o, $(fichiers)) : %.o : %.c          => bar.o lose.o
      gcc -c $< -o $@
$(filter %.elc, $(fichiers)) : %.elc : %.el     => foo.elc
      emacs -f batch-compile $<
```

10. Quelques précisions sur les variables

10.1. Variables expansées récursivement

variables définies par "="

exemples :

```
message = $(truc)
truc = $(machin)
machin = Hello !

all :
      echo $(message)
   ⇨  affichage de Hello !
```

avantage : ex :

```
sources = $(chemin)/*.c
chemin = /home/toto/sources
```

⇨ sources sera expansée au moment de son utilisation

inconvénients :

```
sources = $(sources) prog.c
```

⇨ boucle infinie d'expansion (->erreur générée par make)

10.2. Variables expansées simplement

variables définies par ":=" : expansées uniquement au moment de la définition

exemple :

```
x := prog.c
y := $(x) entete.h
x := fichier.c
⇨ y := prog.c entete.h
⇨ x := fichier.c
```

Ajouter du texte à une variable : utilisation de "+="

ex :
```
objets = prog1.o prog2.o
objets += prog3.o
```
=>
```
objets = prog1.o prog2.o
objets := $(objets) prog3.o
```

➔ si la variable n'a pas été définie auparavant alors "+=" ⇔ "="

10.3. Définir ou modifier des variables au lancement de make

make variable=valeur

exemple :

prog :

gcc –c $(CFLAGS) prog.c

lancement : make CFLAGS="-g –O2"
attention : la définition au lancement écrase la définition interne au makefile de la même variable
pour l'éviter : override variable = valeur (ou :=)
ou pour ajouter des valeurs aux valeurs données au lancement
override variable += valeur

• les variables d'environnement présentes au lancement sont transformées automatiquement en variables make, peuvent être modifiées dans le makefile sauf si appel avec "-e"
ex :
HOME=/toto
si make, HOME du shell écrasé
si make –e, affectation ignorée

11. Divers

11.1. Génération automatique de dépendances

• pour les fichiers includes
gcc –M prog.c
=> donne les règles de dépendances (fichiers headers) prog.o : prog.c defs.h

• **makedepend** : ajoute les dépendances des .c par rapport aux .h
ex :
main.c #include <vector.h>
#include <matrix.c>
matrix.c #include <matrix.h>
makedepend matrix.c main.c
ajoute à la fin du makefile les lignes

DO NOT DELETE
main.o : vector.h matrix.h
matrix.o : matrix.h

La ligne contenant DO NOT DELETE est utilisée par makedepend pour savoir à quel endroit il peut écrire les dépendances. Si aucune chaîne de ce type n'est trouvé dans le fichier, alors les dépendances sont ajoutées à la fin du fichier, même si elles s'y trouvent déjà.

11.2. Appel récursif de make

utile en cas de makefiles séparés pour plusieurs sous-systèmes
ex : répertoire "sousdir" qui possède son propre makefile

```
soussyst :
	cd sousdir && make
```

ou

```
soussyst :
	make –C sousdir
```

• exporter des variables aux sous-makefiles

export variable
unexport variable
export variable=valeur <- définition + export
export <- exporte toutes les variables

11.3. mémorisation d'une suite de commandes

ex :

```
define compileplusfonc
gcc –c $(firstword $^)
gcc $(subst .c, .o,$(firstword $^)) fonctions.o –o $@
endef

prog : prog.c entete.h
	$(compileplusfonc)
```

12. Exemples de makefile

Exemple 1 : programme C

```
CC = gcc
# ================== Name changes happen here only ==============
TARGET = a.out

# ================================================================
# Flags: -O0 to avoid optimization, -O3 for best
# ------------------- Compilers options -------------------------
#
CCFLAGS = -g -O2
LIBS = -lforms \
-lXext -lX11 -lXmu -lXt -lm
LIBSDIR = -L$$HOME/Libs
INCLDIR = -I. -I/usr/include/g++-2
# ----------------------------------------------------------------
# This is processed automatically
SOURCES = $(wildcard *.c)
OBJECTS = $(SOURCES:.c=.o)

$(TARGET): $(OBJECTS)
      @echo "\n=== Linking ==="
      $(CC) $(CCFLAGS) $(LIBSDIR) -o $(TARGET) $(OBJECTS) $(LIBS)

.c.o:
      @echo "\n---" $@ "---"
      $(CC) $(INCLDIR) $(CCFLAGS) -c -o $@ $<

# ------------------------------
# --- This build dependencies
depend:
      makedepend $(INCLDIR) $(SOURCES)
# ------------------------------
clean:
      rm -f *~ *.o core
```

```
# ******************** Makedepend performs here ******************
#
# Be sure to let a blank line after this one
# DO NOT DELETE
```

Exemple 2 : création et installation d'une librairie *libtools.a*

```
C++ = g++
CC = gcc
AR = ar

LIBRARY_NAME = tools

ifdef TOP_DIR
include $(TOP_DIR)/Rules.make
else
INSTALL_LIB_DIR=lib
INSTALL_INCLUDE_DIR=include
endif

INCLUDE_DIR = -I.

ifdef $(INSTALL_INCLUDE_DIR)
INCLUDE_DIR += -I$(INSTALL_INCLUDE_DIR)
endif

CFLAGS = $(GLOBAL_CFLAGS)

########################
#       OBJECTS is all files *.cc and *.c

# all files with *.cc extension
ALL_CC = $(wildcard *.cc)
# all files with *.c extension
ALL_C = $(wildcard *.c)
# substitute .cc with .o
OBJECTS += $(patsubst %.cc, %.o, $(ALL_CC))
# substitute .c with .o
OBJECTS += $(patsubst %.c, %.o, $(ALL_C))

INSTALL_MODE= -m 0666
INSTALL_HEADERS= $(wildcard *.h)

# produce library

$(LIBRARY_NAME): $(OBJECTS)
      $(AR) -rsc lib$(LIBRARY_NAME).a $(OBJECTS)

# compile objects files

.c.o :
      $(CC) -c $< $(CFLAGS) $(INCLUDE_DIR) -o $@

.cc.o :
      $(C++) -c $< $(CFLAGS) $(INCLUDE_DIR) -D__cplusplus -o $@

# install library

install : $(LIBRARY_NAME)
      # create libraries dir if no exist
      install -d $(INSTALL_LIB_DIR)
```

```
      # create headers dir if no exist
      install -d $(INSTALL_INCLUDE_DIR)
      # install library
      rm -f $(INSTALL_LIB_DIR)/lib$(LIBRARY_NAME).a
      install $(INSTALL_MODE) lib$(LIBRARY_NAME).a $(INSTALL_LIB_DIR);
      # install headers
      @for i in $(INSTALL_HEADERS); do \
            rm -f $(INSTALL_INCLUDE_DIR)/$$i; \
                install $(INSTALL_MODE) $$i $(INSTALL_INCLUDE_DIR); \
         done

clean :
      @rm -f *.o lib$(LIBRARY_NAME).a *~
```

exemple d'utilisation :

- make
- make OBJECTS=test.o
- make TOP_DIR=/home/root

avec `/home/root/Rules.make`

```
ifeq ($(USER), root)
INSTALL_LIB_DIR=/usr/local/lib
INSTALL_INCLUDE_DIR=/usr/local/include
endif
```