Make

Didier DONSEZ

Université Joseph Fourier (Grenoble 1)

IMA – LSR/ADELE

Didier.Donsez@imag.fr, Didier.Donsez@ieee.org

Motivation : L'organisation des Développements

- Développements multi-plateformes de gros codes en groupe.
  - plusieurs fichiers de sources dans différents langages
  - Pouvoir générer AUTOMATIQUEMENT un(des) exécutable(s) à jour à partir des dernières modifications
    pour une plateforme donnée.

- Les outils
  - make : lanceur de commandes (compilation) en fonction de dépendances
  - imake: générateur de Makefile indépendant de la machine cible
  - SCCS,RCS, CVS, … : gestion de versions
  - Bibliothèques

L'Utilitaire Make

- Motivation
- agit selon des règles
- contenu d'un Makefile
- Règles standards
- Déclarations de macro
- Caractères joker
- Substitution de chaînes dans les macros
- Règles de suffixe
- Archives et règles de suffixe
- "règles spéciales"
- makedepend
- imake

Motivations

- Simplifier le déclenchement des suites d'opérations souvent répétées sur des fichiers
- Optimiser la recréation d'exécutables et de bibliothèques dépendant de fichiers modifiés

Make : agit selon des règles

- make cible
  - applique des règles "universelles" (dans /usr/share/lib/make/make.rules)
  - puis des règles "ponctuelles" le plus souvent dans ./Makefile ou ./makefile
  - Remarque:
    - on peut se passer de ./Makefile, ./makefile si opération tellement simple que les règles "universelles" suffisent
- make -f autre_fichier_de_commandes cible
- make -r cible
  - pour ne pas utiliser les règles "universelles"
  - [ci-après, Makefile désignera n'importe quel fichier de règles ponctuelles]

Contenu d'un Makefile

- # commentaires
- lignes blanches
- règles standards avec cible réelle
- règles standards avec pseudo-cible
- déclarations de macro
- règles de suffixe
- règles "spéciales"

Règles standards

Règle avec dépendances

- Les commandes seront exécutées si des dépendances ont été modifiées ultérieurement à la dernière modification de cible

```
cible : fichier_1 fichier_2 fichier_3
< tab > commande
< tab > commande
< tab > ...
```

- Remarque : avant d'exécuter ces commandes, make va éventuellement recréer les dépendances

Règles standards

- Règle simple (sans dépendances)
  - Les commandes seront exécutées si cible n'existe pas
- cible réelle
  - l'exécution des commandes visent à créer le fichier cible

```
cible :
< tab > mkdir cible
< tab > commande
< tab > ...
```

- pseudo-cible
  - le fichier cible n'est jamais créé

```
clean :
< tab > rm -f *.o
```

Règles standards
Dépendances supplémentaires

- Règle de dépendance parasite
  - sans commande pour ajouter d'autres dépendances à une règle déjà définie

```
cible : fichier_1 fichier_2 fichier_3
< tab > commande
< tab > ...
cible : dep1.h dep2.h
```

Règles standards
Dépendances supplémentaires

- Double dépendance
  avec des suites de commandes différentes
  - Attention au ::

```
cible :: fichier_1 fichier_2 fichier_3
< tab > commande_a
< tab > commande_b
< tab > ...
cible :: fichier_11 fichier_12 fichier_13
< tab > commande_m
< tab > commande_n
< tab > ...
```

Caractères Joker

- `$@` cible (lignes de commandes)
- `$?` l'ensemble des dépendances plus récentes que la cible
- `$<` première dépendance de la liste des dépendances
- `$^` toutes les dépendances
- `$*`la chaîne correspondant au pattern `%`

```
echantillons : echantillons.dos
< tab > dos2unix $? $@ 2>/dev/null
mylib.a : obj1.o obj2.o obj3.o
< tab > ar r $@ $?
myprog : obj1.o obj2.o obj3.o
< tab > cc -o $@ $^ -lsocket -lnsl
obj/%.o: src/%.c
< tab > ${CC} -o obj/%.o -c src/$*.c
```

Macros (i)

- Motivation : paramètrisation
- Déclaration

```
CFLAG = -O2
CC = gcc
MYOBJS = obj1.o obj2.o obj3.o
LIBS = -lsocket -lnsl
INSTALL = ${HOME}/bin
```

- Usage

```
myprog : ${MYOBJS}
< tab > ${CC} ${CFLAG} -o $@ ${MYOBJS} ${LIBS}
```

- Macros imbriquées

```
SYSLIBS = -lsocket -lnsl
LIBS = ${MYLIBDIR} ${MYLIBS} ${SYSLIBS}
```

Macros (ii)

- Surchage d'une macro depuis la ligne de commande

```
make "MYOBS = obj1.o obj2.o obj3_bis.o"
```

- Utilisation des variables d'environnement

```
install: ${EXEC}
< tab > cp $< ${HOME}/bin
```

- Priorités des Macros (du plus au moins prioritaire)
  - macros déclarées à la ligne de commande
  - macros déclarées dans le Makefile
  - macros déclarées en variables d'environnement (setenv)
  - macros "universels" (dans /usr/share/lib/make/make.rules)

Macros (iii)

- "macro string substitution"
  - possibilité de substituer des chaînes de caractères mais seulement en début et en fin de mot

```
SRCS = m1.c m2.c m3.c
OBJS = ${SRCS: .c=.o}
DEBUG_OBJS = ${SRCS: .c=_dbg.o}
```

Règles de suffixe (i)

- Règles basées sur les suffixes
  - La force de Make : remonter la chaîne des dépendances tant que des règles (ponctuelles ou universelles) peuvent être appliquées
- Exemple

```
.c.o:
<tab> ${CC} ${CFLAGS} -c $<
.c:
<tab> ${CC} ${CFLAGS} ${LDFLAGS} $< -o $@
.y.c:
<tab> $(YACC.y) $<
<tab> mv y.tab.c $@
```

- Priorités entre suffixe

```
.SUFFIXES: .o .c .cc .y .l .s .sh .S .ln \
.h .f .F .mod .sym .def .p .r \
.f90 .ftn .cps .C .Y .L
```

Règles de suffixe (ii)

- Exemple n'utilisant que des règles de suffixe

```
SUFFIXES = .ps .dvi .tex
.SUFFIXES: ${SUFFIXES}
.tex.dvi:
<tab> latex $<
.dvi.ps:
<tab> dvips -o $@ $<
.tex.ps:
<tab> latex $<
<tab> dvips -o $@ $<
```

Les archives

- Rappel : une archive d'objets (nommé libXX.a) facilite la gestion d'un ensemble de fichiers objet (.o)
- Règle de suffixe de construction d'une archive

```
.c.a:
< tab > ${CC} -c ${CFLAGS} -c $<
< tab > ar rv $@ $*.o
< tab > rm -f $*.o
```

- make permet de référencer un objet intégré dans une archive
- libXX.a(util.o) représente l'objet util.o dans l'archive libXX.a
- exemple

```
prog : main.o libXX.a(util.o)
< tab >${LINK.c} -o $@ $*
```

Règles spéciales

- Change le comportement de make

```
.DEFAULT:
@ echo "this is the default target"
.SILENT:
# ( = make -s)
.IGNORE:
# ( = make -i)
.SUFFIXES:
# ( voir des exemples)
```

- …

/usr/share/lib/make/make.rules

- Contient des définitions (par défaut) de
  - Macros
  - Règles de suffixes
- Question
  - `'make prog'` est suffisante pour obtenir l'exécutable prog à partir de prog.c . Pourquoi ?
- Réponse : cette règle de suffixe

```
.c:
<tab> ${CC} ${CFLAGS} ${LDFLAGS} $< -o $@
```

Les options de make

- `make [options] [définition macros] [cibles]`
- `make f1.o f2.o` *(les cibles f1.o et f2.o)*
- `make "EXEC=app.exe" "CC=gcc"` *(macro)*
- `make -f appli.mk`
- `make -n` *(mise au point du makefile)*
- `make -ts` *(touch et .SILENT)*
- `make -p` *(liste des macros)*
- `make -i` *(.IGNORE)*
- `make -d` *(debug interne à make)*

Utilitaire makedepend

- makedepend [options] liste de sources .c
  - Ecrit dans un Makefile des règles de dépendance entre des fichiers objet et les fichiers en-tête (.h) inclus dans les sources .c
  - Utilise le préprocesseur du C
    - `#include, #ifdef, -I..chemin`
- Exemple

```
SRCS = main.c fnc.c affich.c
CFLAGS = -Dxxxxx, -Ixxxxxxx
depend:
<tab> makedepend -- ${CFLAGS} -- ${SRCS}
```

  - Ajoute à la fin du Makefile un bloc de lignes :

```
# DO NOT DELETE THIS LINE -- make depend depends on it.
main.o : fnc.h affich.h
fnc.o : affich.h
```

Utilitaire Imake

Quelques conventions de nommage de cibles pour un projet

- all: compile tous les fichiers source pour créer l'exécutable principal
- install: exécute *all*, puis copie l'exécutable, les bibliothèques, les données, et les fichiers en-tête dans les répertoires de destination
- uninstall: détruit les fichiers créés lors de l'*install*
- info: génère un fichier *info*
- man: génère les pages du manuel
- dist: crée un fichier *tar* de distribution
- clean: détruit tout les fichiers créés par *all*
- makedepend: calcule les dépendances et les ajoute au Makefile

Autres

- genmake
- construction automatique d'un Makefile pour une seule cible à partir des sources C et des include dans le répertoire courant du projet
- makedepend
- construction automatique des dépendances des fichiers include qui seront rajoutées ou modifiées en fin du Makefile
- imake, xmkmf
- imake : utilisation d'un fichier modèle (template) Imakefile
- xmkmf : script spécifique à X11 utilisant imake pour adapter les PATHS de X11
- fichier Imakefile
- autoconf, automake, configure

Conclusion
Points forts vs Points faibles

POINTS FORTS

- la prise en compte de règles de dépendance entre des fichiers
- l'utilisation de leur date de modification
- pour déterminer la nécessité de prendre des actions
- Versions parallèles de make

POINTS FAIBLES

- convivialité moyenne
- syntaxe ubuesque
- emploi d'"astuces"
- Si c'est surtout pour simplifier et non pour optimiser le déclenchement d'une suite d'opérations, un script est parfois plus commode
- fonctionnalités non standards (`make, gmake, nmake`)
- hétérogénéité (Unix vs Windows) des chemins (PATH), des séparateurs (; :) et des commandes (`cp` vs `copy`)
  - CygWin pâlie partiellement à ce problème

Bibliographie et Webgraphie

- Andy Oram, Steve Talbott, « Managing Projects with make, 2nd Edition », Ed Oreilly, Octobre 1991, 0-937175-90-0
- Documentation de GNU Make

- Cours sur Make
  - http://www.infres.enst.fr/~dax/polys/make/
  - http://www.april.org/groupes/doc/make/make.html
  - http://www.cmi.univ-mrs.fr/~contensi/coursC/environnement/make.html