Make et Makefile : petit résume rapide

(*Vos commentaires sont les bienvenus : mailto:jciehl@bat710.univ-lyon1.fr*)

Make est outil très général permettant, entre autre, d'automatiser la compilation d'un projet.

Supposons que le projet soit constitué des sources C suivants : main.c, structure.c, et operation.c . Il est possible de compiler ce projet de trois manières différentes, les parties suivantes précisent ces différentes étapes.

1. une seule commande

```
gcc -o prog -Wall main.c structure.c operation.c
```

(rappel des options de gcc : `man gcc`, `info gcc`, ou *gnome help system* sujet `info:gcc`)

2. une commande pour chaque fichier source et une autre pour créer l'éxécutable :

```
gcc -c -Wall main.c
gcc -c -Wall structure.c
gcc -c -Wall operation.c
```

Les commandes précédentes compilent chaque fichier source et créent les fichiers objets correspondants qui s'appellent : main.o, structure.o et operation.o. Un fichier objet est le résultat de la compilation d'un fichier source et contient les instructions machines associées à chaque fonction du fichier source, ainsi que la liste des fonctions appelées qui ne se trouvent pas dans le fichier source (par exemple, les fonctions des librairies standards printf, scanf, malloc, etc. et les fonctions que vous avez écrites mais qui se trouvent dans les autres fichiers sources (voir l'utilisation du mot clé extern)).

```
gcc -o prog main.o structure.o operation.o
```

Cette dernière commande crée l'éxécutable prog en assemblant (cette partie de la compilation s'appelle l'édition de liens) les ensembles d'instructions associés à chaque fichier objet. C'est à ce moment que le compilateur vérifie qu'il connait l'ensemble d'instructions associé à chaque fonction.

Le cycle de travail sur le projet nécessitera généralement plusieurs corrections d'erreurs. Il faudra donc, après chaque correction, recompiler le fichier source modifé (après avoir enregistré les modifications ...). La compilation de ce fichier source produira un fichier objet et il faudra bien sur recréer l'éxécutable.

Prenons un exemple :
supposons que l'on ait modifié main.c, la commande suivante permet de le recompiler (et de créer une nouvelle version de main.o) :

```
gcc -c -Wall main.c
```

il faut encore reconstruire l'éxécutable, afin de tenir compte des modifications apportées à main.c (et donc main.o). Par contre, les objets structure.o et operation.o n'ont pas été modifiés, il n'est donc pas nécessaire de les recompiler.

```
gcc -o prog main.o structure.o operation.o
```

On peut maintenant vérifier le bon fonctionnement du programme.

Un projet "moyen" comporte rapidement une dizaine de fichiers sources, et il devient vite pénible de taper toutes les commandes nécessaires à la compilation du projet. De plus, le risque d'erreur ou d'oubli augmente très rapidement, d'ou l'utilité, voire la nécessité, d'un outil permettant d'automatiser la construction du projet.

3. Make

L'utilisation de make est relativement simple, une fois que l'on a compris que son rôle est de produire automatiquement la séquence de commandes permettant de construire un projet. Pour créer l'éxécutable, la première fois, il faut que make génère la séquence de commandes décrite dans la partie 2 :

```
gcc -c -Wall main.c
gcc -c -Wall structure.c
gcc -c -Wall operation.c
gcc -o prog main.o structure.o operation.o
```

De la même manière, après la modification de main.c, make doit générer la séquence de commandes suivante :

```
gcc -c -Wall main.c
gcc -o prog main.o structure.o operation.o
```

Le raisonnement qui nous a permis d'écrire cette séquence de commande est basé sur les dépendances (ou les relations) entre ces fichiers :

main.c produit main.o
et
main.o, structure.o et operation.o permettent de construire le projet prog.

Il suffit de décrire ces relations à Make, ainsi que les commandes associées pour qu'il puisse produire la séquence de commandes correcte. Ces relations sont décrites dans un fichier texte, nommé Makefile ou makefile. Ce fichier est constistué des descriptions des relations entre les fichiers sources et les fichiers objets ainsi que des relations entre les fichiers objets et le projet.

Une relation s'écrit de la manière suivante dans le fichier makefile :

```
produit : source
    commande
```

Cette règle indique plusieurs choses à make. Premièrement, que `produit` est crée à partir de `source` et que c'est `commande` qui permet de le faire. On peut donc décrire la relation entre main.c et main.o, qui est produit par la compilation de main.c :

```
main.o : main.c
    gcc -c -Wall main.c
```

De même, la relation entre les fichiers objets et le projet s'écrit :

```
prog: main.o structure.o operation.o
    gcc -o prog main.o structure.o operation.o
```

Pour achever la création du makefile, il ne reste plus qu'à écrire les règles pour structure et operation :

```
structure.o : structure.c
    gcc -c -Wall structure.c

operation.o : operation.c
    gcc -c -Wall operation.c
```

Un dernier détail, comme le makefile est composé de plusieurs relations, il faut indiquer à make laquelle construire en priorité : par convention, c'est tout simplement la première. Il suffit donc d'écrire la règle décrivant la construction du projet au début du fichier makefile. Le makefile complet ressemblera donc à :

```
prog: main.o structure.o operation.o
    gcc -o prog main.o structure.o operation.o

main.o : main.c
    gcc -c -Wall main.c

structure.o : structure.c
    gcc -c -Wall structure.c

operation.o : operation.c
    gcc -c -Wall operation.c
```

Lors d'une modification, make se base sur la date de modification des fichiers pour déterminer les mises à jours à effectuer. Dans le scénario précédent, le projet complet est compilé et éxécuté. Lors de ce premier appel à make, ni les fichiers objets, ni l'éxécutable n'existent, ils sont donc créés en utilisant la commande associée à leur règle. Ensuite, main.c est modifié, main.c devient donc plus récent que main.o et prog. Make analyse les régles définies dans le makefile et vérifie les dates des fichiers. Comme main.c est plus récent que main.o, ce dernier ne peut être le produit de la version actuelle de main.c, il faut donc le recompiler, la commande associée à la règle main.o sera donc éxécutée (`gcc -c -Wall main.c`). Après la compilation, la date de main.o est plus récente que celle de prog, il faudra donc récréer prog avec la commande correspondante (`gcc -o prog main.o structure.o operation.o`).

3. Quelques idées d'utilisation

Make ne pose aucune contrainte sur la compilation. Il ne sait pas qu'il est en train de compiler un projet, vous pouvez donc en profiter pour lui faire faire autre chose, comme effacer les fichiers temporaires, créer une archive de votre projet, ou définir plusieurs versions du même projet ou de plusieurs projets dans le même makefile.

Les sections suivantes présentent quelques règles à ajouter au makefile. Pour indiquer à make que l'on veut éxécuter une règle particulière il suffit de lui indiquer : `make regle`, ce qui éxécutera les commandes associées à la règle `regle`.

3.1 Effacer les fichiers temporaires

Lors de modifications importantes d'un projet, on a souvent besoin de recompiler la totalité des sources, ou de nettoyer le répertoire de travail. Il est toujours possible de taper une commande rm dans un terminal, mais on peut aussi le faire faire à make. Comment écrire cette règle ? Quelle relation existe-t-il entre un fichier et un fichier effacé ? La réponse est simple : aucune. Mais on peut quand même écrire une règle qui éxécute toujours sa commande, il suffit de ne pas indiquer de source et de donner à la règle un nom qui n'est pas un fichier.

Pour effacer les fichiers objets du projet, il suffit de d'écrire la règle suivante qui s'appelle clean par convention :

```
clean :
    rm -f prog *.o
```

L'option -f (force) indique juste à rm de ne pas signaler les erreurs éventuelles (plus de détail sur les options de rm, `man rm`). Le *.o est un joker qui indique à rm d'effacer tous les fichiers se terminant par .o, c'est à dire les fichiers objets.

`make clean` éxécutera donc la commande `rm -f prog *.o` associée à la règle `clean`.

3.2 Archiver le projet

Les règles sans sources (voir ci-dessus) permettent d'éxécuter une commande quelconque, on peut en profiter pour archiver le projet, par exemple. On peut compléter le makefile par la règle :

```
zip:
    tar -zcvf prog.tar.gz main.c structure.c operation.c Makefile
```

La commande tar permet de créer une archive compressée, il suffit d'indiquer le nom de l'archive à créer et la liste des fichiers à archiver (plus de détails sur l'utilisation de tar, `man tar`). Vous pouvez aussi envoyer l'archive par mail (`man mail`) ou la copier sur disquette.

`make zip` éxécutera donc la commande `tar -zcvf prog.tar.gz main.c structure.c operation.c Makefile` associée à la règle `zip`.

3.3 Plusieurs versions du projet

Dans le TP3, il était demandé de fournir plusieurs versions du même programme. On peut créer plusieurs projets séparés, ou tout simplement décrire les différentes versions dans le même makefile. Le TP3 était composé des sources C suivants : blob.c tga.c tri_blob.c. Voici le fichier Makefile fourni avec le sujet :

```
blob: blob.o tga.o tri_blob.o
    gcc -o blob blob.o tga.o tri_blob.o -lm

blob.o: blob.c
    gcc -Wall -c blob.c

tga.o: tga.c
    gcc -Wall -c tga.c

tri_blob.o: tri_blob.c
    gcc -Wall -c tri_blob.c

tarball:
    tar -zcvf blob.tar.gz blob.c tga.c tri_blob.c blob.h tga.h Makefile

clean:
    rm *.o blob
```

Ce Makefile crée un éxécutable nommé blob à partir de blob.o, tga.o et de tri_blob.o. Le travail demandé consistait principalement a écrire de nouvelles fonctions de tri. Supposons que le tri par insertion se trouve dans le source tri_insertion.c, et que le tri par tas se trouve dans tri_tas.c. On veut créer une version de blob, blob_insertion, compilée avec le tri par insertion et une autre compilée avec le tri par tas, blob_tas. Les deux nouvelles règles :

```
blob_insertion: blob.o tga.o tri_insertion.o
        gcc -o blob_insertion blob.o tga.o tri_insertion.o -lm

tri_insertion.o:        tri_insertion.c
        gcc -Wall -c tri_insertion.c
```

permettent de créer l'éxécutable blob_insertion qui est bien blob compilé avec la fonction de tri par insertion. De même, il suffit de rajouter les règles suivantes pour la version tri par tas :

```
blob_tas:       blob.o tga.o tri_tas.o
        gcc -o blob_tas blob.o tga.o tri_tas.o -lm
```

```
tri_tas.o:      tri_tas.c
        gcc -Wall -c tri_tas.c
```

Au final, le Makefile ressemblera à :

```
blob:   blob.o tga.o tri_blob.o
        gcc -o blob blob.o tga.o tri_blob.o -lm

blob_insertion: blob.o tga.o tri_insertion.o
        gcc -o blob_insertion blob.o tga.o tri_insertion.o -lm

blob_tas:       blob.o tga.o tri_tas.o
        gcc -o blob_tas blob.o tga.o tri_tas.o -lm


tri_tas.o:      tri_tas.c
        gcc -Wall -c tri_tas.c

tri_insertion.o:        tri_insertion.c
        gcc -Wall -c tri_insertion.c

blob.o: blob.c
        gcc -Wall -c blob.c

tga.o: tga.c
        gcc -Wall -c tga.c

tri_blob.o: tri_blob.c
        gcc -Wall -c tri_blob.c

tarball:
        tar -zcvf blob.tar.gz blob.c tga.c tri_blob.c blob.h tga.h Makefile

clean:
        rm *.o blob
```

Les trois premières règles (blob, blob_insertion, blob_tas) correspondent aux trois versions du projet et les règles suivantes décrivent la compilation des différents sources des projets.

4. Makefile avancé

La rédaction d'un makefile est relativement délicate, il est courant d'oublier de rajouter un objet dans les sources d'une règle ou dans la liste des objets à compiler pour créer l'éxécutable. De même, l'écriture des règles de compilation est particulièrement longue. Heureusement make dispose de fonctionnalités supplémentaires permettant d'automatiser une partie de la rédaction. Les deux sections suivantes présentent l'utilisation des variables afin de simplifier l'écriture des makefile.

4.1 variables

Des variables existent dans make, ce sont simplement des chaînes de caractères. On peut par exemple définir une variable indiquant quel compilateur utiliser, quelles options de compilation utiliser ou encore une liste de fichiers.

Pour définir une variable, il suffit de lui donner un nom et une valeur. Un nom de variable est une suite de caractères ne contenant pas `:', `#' et `='. Une valeur est une chaîne de caractères quelconque. Pour substituer une variable à sa valeur, il suffit d'écrire $nomvariable. Quelques exemples :

```
CC= gcc
```

```
blob:   blob.o tga.o tri_blob.o
        $(CC) -o blob blob.o tga.o tri_blob.o -lm
...
```

ou

```
CC= gcc
objets= blob.o tga.o tri_blob.o

blob:   $(objets)
        $(CC) -o blob $(objets) -lm
...
```

Les parenthèses autour des noms de variables ne sont pas obligatoires mais permettent d'éviter certains problèmes lors de la substitution de la variable par sa valeur (c'est un simple remplacement de caractères).

4.2 variables automatiques et règles implicites

Il existe un autre type de variables : les variables automatiques, leurs noms sont imposés et make défini leurs valeurs. Voici les plus utilisées :

`$@` : produit (ou but) de la règle
`$<` : nom de la première dépendance (ou source)
`$?` : toutes les dépendances plus récentes que le but
`$^` : toutes les dépendances
`$+` : idem mais chaque dépendance apparait autant de fois qu'elle est cité et l'ordre d'apparition est conservé.

La variable `$@` représente le produit (ou le but) d'une règle, en reprenant l'exemple précédent :

```
CC= gcc
objets= blob.o tga.o tri_blob.o

blob:   $(objets)
        $(CC) -o $@ $(objets) -lm
```

La variable `$^` représente toutes les dépendances d'une règle, par exemple tous les objets lors de la création du projet :

```
CC= gcc
objets= blob.o tga.o tri_blob.o

blob:   $(objets)
        $(CC) -o $@ $^ -lm
```

La variable `$<` représente la première dépendance, on peut l'utiliser pour compiler un source C :

```
CC= gcc -Wall
blob.o: blob.c
        $(CC) -o $@ -c $<
```

L'utilisation des variables automatiques est particulièrement puissante lors de la définition de règles implictes qui s'appliquent à plusieurs fichiers. Par exemple, il possible, avec une seule règle implicite, de compiler tous les sources C :

```
%.o: %.c
     $(CC) -o $@ -c $<
```

Le but est défini par un *pattern*, ou un joker : %.o . Cette règle indique que chaque fichier .o nécessaire à la création du projet est obtenu à partir du fichier .c correspondant et que la commande `$(CC) -o $@ -c $<` permet de créer le fichier objet à partir du fichier source. Les variables automatiques permettent de retrouver le nom du fichier concerné par la règle.

En résumé, il est donc possible de compiler un projet quelconque avec un Makefile de quelques lignes, par exemple blob :

```
CC= gcc        # compilateur
CFLAGS= -Wall  # options de compilation pour les sources C

objets= blob.o tga.o tri_blob.o
blob: $(objets)
    $(CC) -o $@ $^ -lm

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

4.3 Génération automatique des noms de fichiers

Il est possible d'utiliser des substitutions dans les noms de fichiers, par exemple pour générer automatiquement les noms des fichiers objets à partir des noms des fichiers sources, en remplaçant l'extension .c par .o :

```
SRC= main.c struct.c operation.c

# nommage automatique des fichiers objets d'apres les noms des sources C
OBJ= $(SRC:.c=.o)
```

dans cet exemple `$(OBJ)` contiendra `main.o struct.o operation.o`

exemple du makefile précedent avec nommage automatique des objets a partir des fichiers sources .c :

```
CC= gcc        # compilateur
CFLAGS= -Wall  # options de compilation pour les sources C

sources= blob.c tga.c tri_blob.c
objets= $(sources:.c=.o)

blob: $(objets)
    $(CC) -o $@ $^ -lm

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

5. Génération automatique des dépendences

Lorsque un projet prends un petit peu d'importance, de nombreux headers (.h) sont utilisés. Lorsque l'on modifie un header, il est nécessaire de recompiler tous les sources qui l'utilisent. Il est évidement possible d'ajouter les fichiers .h comme dépendance des sources, mais c'est particulièrement fastidieux. Le compilateur est le mieux placé pour savoir quels sont les headers utilisés par chaque source et il est même possible de récupérer ces dépendances au format des régles de make. Il suffit de stocker cette nouvelle règle dans un fichier (.d par convention) et de l'inclure dans le makefile. Pour vous convaincre, essayer sur un de vos sources :

```
    gcc -o source.d  -MM source.c

    cat source.d
```

regardez les options -MM dans le manuel de gcc

(la suite, bientôt ...)

exemple de makefile complet :

3 projets différents sont décrits,
archivage de l'ensemble,
numérotation automatique du projet,
génération automatique de changelog
...

```
# correction TP5 systemes d'exploitation 2003 : partage de fichiers
CFLAGS= -g -Wall -D_REENTRANT

LDFLAGS=

LIB= -lpthread

CC= gcc $(CFLAGS)
LD= gcc

# serveur d'annonce
ASRC=annonce.c \
    annonce_sha1_f.c \
    mtfifo.c \
    critique.c \
    socket.c \
    fichier.c \
    util.c \
    debit.c \
    version.c \
    sha1.c \
    sha1_f.c

# nommage automatique des fichiers objets d'apres les noms des sources C
AOBJ= $(ASRC:.c=.o)

# nommage automatique des fichiers de dependance d'apres les noms des sources C
ADEP= $(ASRC:.c=.d)

# serveur de fichiers
FSRC=serveur_fichier.c \
    serveur_fichier_sha1.c \
    serveur_fichier_sha1_f.c \
    mtfifo.c \
    critique.c \
    socket.c \
    fichier.c \
    util.c \
    debit.c \
    version.c \
    sha1.c \
    sha1_f.c

# nommage automatique des fichiers objets d'apres les noms des sources C
```

```
FOBJ= $(FSRC:.c=.o)

# nommage automatique des fichiers de dependance d'apres les noms des sources C
FDEP= $(FSRC:.c=.d)

#client
CSRC=client_sha1.c \
    client_sha1_f.c \
    mtfifo.c \
    critique.c \
    socket.c \
    fichier.c \
    util.c \
    debit.c \
    version.c \
    sha1.c \
    sha1_f.c

# nommage automatique des fichiers objets d'apres les noms des sources C
COBJ= $(CSRC:.c=.o)

# nommage automatique des fichiers de dependance d'apres les noms des sources C
CDEP= $(CSRC:.c=.d)

# ensemble des fichiers de dependance
DEP= $(ADEP) $(FDEP) $(CDEP)


# numero de version auto
include build

#
BIN=annonce  \
    serveur_fichier \
    client

#
VER=annonce.version \
    serveur_fichier.version \
    client.version


.PHONY: all
all: $(BIN) $(VER) changelog

# numerotation des compilations completes
build:
    echo BUILD= 1 > build

rebuild: build
    @echo BUILD= `expr $(BUILD) + 1` > build

# genere un fichier C avec une chaine de caracteres decrivant la version du projet
version.c: build
    @echo "char version_id[]= \"build $(BUILD)\";"  > $@

# conserve les commentaires sur les versions du projet
changelog: build
    @echo -- changelog build $(BUILD)
    @echo -e --\\n$(USER)@`hostname -s` --  build $(BUILD)  $(HOSTTYPE) --  `date` > $@.tmp
    @if test -f $@; then cat $@ >> $@.tmp; fi
    @mv $@.tmp $@


annonce.version: annonce
```

```
    @echo $< --  $(USER)  build $(BUILD)  --  `date` > $@

annonce: $(AOBJ)
    @echo -- build $(BUILD)
    $(LD) $(LDFLAGS) -o $@ $+ $(LIB)

#
serveur_fichier.version: serveur_fichier
    @echo $< --  $(USER)  build $(BUILD)  --  `date` > $@

serveur_fichier: $(FOBJ)
    @echo -- build $(BUILD)
    $(LD) $(LDFLAGS) -o $@ $+ $(LIB)

#
client.version: client
    @echo $< --  $(USER)  build $(BUILD)  --  `date` > $@

client: $(COBJ)
    @echo -- build $(BUILD)
    $(LD) $(LDFLAGS) -o $@ $+ $(LIB)


#
%.o: %.c
    $(CC) -o $@ -c $<

%.d: %.c
    $(CC) -MM -MD -o $@ $<


.PHONY: clean
clean: rebuild
    rm -f $(BIN) $(OBJ) $(DEP) *.i *.s

.PHONY: tarball
tarball:
    @echo -- build $(BUILD)
    -tar -zcf annonce_b$(BUILD).tar.gz Makefile build changelog *.[ch]

# inclusion des dependances
-include $(DEP)
```

6. Conclusion

Il existe de nombreuses fonctionnalités, lisez le manuel : `info make`