TD 3: (Re-)Compilation automatique à l'aide de Make*

ED6 — Licence 3 — Université Paris Diderot

Séances du 7 et 14 mars 2012

L'objectif de ces travaux dirigés est la découverte de l'outil Unix `make`, un gestionnaire de **dépendances entre fichiers**.

À l'issue de ce TD, vous devrez savoir :
- quels problèmes peuvent être entraînés par des dépendances mal calculées ;
- écrire un `Makefile` pour un projet C ;
- connaître les principaux mécanismes offerts par `make`.

1 Introduction

Un projet est formé d'un ensemble de fichiers sources dont l'interprétation à travers des processus de traduction mène à l'obtention d'un produit. Ce produit peut être un logiciel, un document, une trace d'exécution d'une batterie de tests,... La figure 1 illustre un tel processus de traduction dans le cadre d'un développement logiciel.

Lorsque l'on modifie un sous-ensemble de ces fichiers sources, il est nécessaire de recalculer la nouvelle version du produit. Une approche naïve de ce problème consisterait à relancer la totalité du processus de traduction pour calculer la nouvelle version du produit à partir de zéro. Cependant, cette idée ne passe pas à l'échelle dès que le projet atteint une dimension raisonnable : il est bien plus efficace de recalculer uniquement la partie du produit dont les sources ont été modifiées, c'est ce que l'on appelle la reconstruction incrémentale.

Pour savoir si un fichier source A doit être analysé de nouveau, il faut d'abord déterminer si le contenu de A a été modifié mais aussi si il existe un fichier B dont le contenu a été modifié et qui influence l'analyse de A. Exhibez les dépendances entre les fichiers d'un projet sert à répondre à cette question.

C'est à cette tâche qu'est dévolu l'outil `make`. Plus précisément, il permet de :
- déterminer les dépendances entre fichiers ;
- automatiser la construction incrémentale d'un projet ;
- centraliser les commandes de maintenance d'un projet.

Les informations spécifiques au projet sont décrites par le programmeur dans un `Makefile`.

`make` est un outil de développement standard sous UNIX (il est standardisé POSIX.2). Il existe de nombreuses versions de `make` qui fournissent des extensions à ce standard. Nous utiliserons `GNUMake`, le plus populaire, mais nous resterons dans le cadre du standard POSIX.

2 Un premier Make le

Question 2.1. *Récupérez l'archive* `http://upsilon.cc/~zack/teaching/1112/ed6/tp3-files.tar.gz`. *Dans un nouveau répertoire* `tp_make`, *décompressez l'archive :* `tar xvfz tp3-files.tar.gz`.

Créez un fichier `src/Makefile` *dont le contenu est :*

```
# Variable declarations
CC = gcc
CFLAGS = -Wall

# Directives
.PHONY: re clean

# Rules
all: main

compute_result.o: compute_result.c compute_result.h \
                  compute_struct.h tools.h
	$(CC) $(CFLAGS) -c compute_result.c
```

*Ce sujet de TD est inspiré d'un sujet proposé par Yann Régis-Gianas

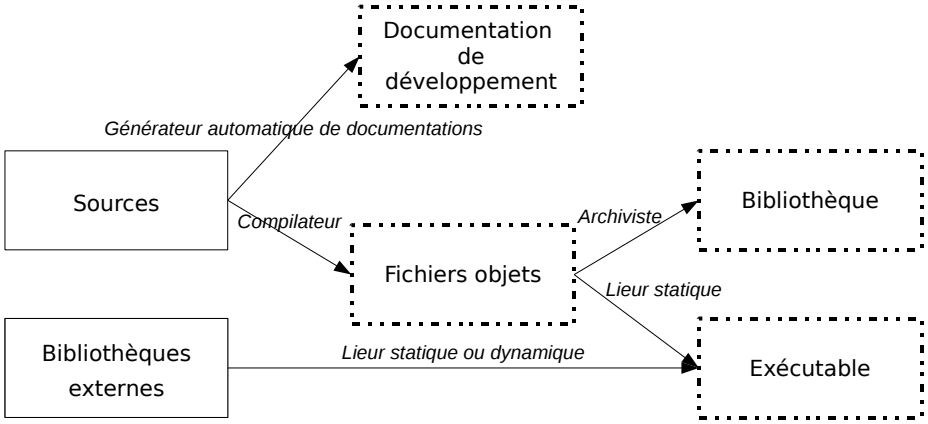

FIGURE 1 – Chaîne standard de compilation : les produits sont entourés en pointillé.

```
parse_num.o: parse_num.c parse_num.h tools.h
	$(CC) $(CFLAGS) -c parse_num.c

print_result.o: print_result.c print_result.h compute_struct.h tools.h
	$(CC) $(CFLAGS) -c print_result.c

main.o: main.c print_result.h compute_result.h parse_num.h
	$(CC) $(CFLAGS) -c main.c

main: main.o print_result.o parse_num.o compute_result.o
	$(CC) -o main print_result.o parse_num.o compute_result.o main.o

clean:
	rm -f main *.o *~ \#*

re: clean main
```

Nous allons d'abord comprendre ce `Makefile` et ensuite le modifier pour corriger les quelques erreurs qui s'y trouvent.

2.1 Compiler avec Make

Un `Makefile` est composé de règles, de déclarations de variables et de directives. Nous rentrerons dans le détail du format de ces différents composants un peu plus tard. Notez que les lignes commençant par un '#' sont des commentaires.

Question 2.2. *Identifiez la catégorie des différentes lignes du* `Makefile` *précédent.*

Question 2.3. *Dans le répertoire* `src`*, lancez make :*

```
$ cd ~/tp_make/src
$ make
```

La compilation du projet s'effectue. Vous pouvez lancer l'exécutable, c'est un programme qui effectue la division euclidienne d'un nombre nb donné en argument par 3 (il calcule p et q tel que nb = p * 3 + q).

Par exemple :

```
$ ./main 26
8 2
$
```

2.2 L'importance des regles

2.2.1 Pour maintenir les fichiers à jour

Le `Makefile` décrit les dépendances entre fichiers. Dans notre cas, on peut remarquer que les fichiers `main.o`, `print_result.o` et `compute_result.o` dépendent du fichier `compute_struct.h`. En effet, ce fichier définit une structure de données qui est utilisée lors du calcul et de l'affichage du résultat.

Question 2.4. *Modifiez le fichier* `compute_result.h`*, en rajoutant une ligne vide par exemple ou bien en lançant la commande :* `touch compute_result.h`*.*

Relancez `make`*.*

Comme le fichier `compute_result.h` *est plus récent que les fichiers qui en dépendent, le système recompile les fichiers* `main.o`*,* `print_result.o` *et* `compute_result.o` *mais pas le fichier* `parse_num.o` *car celui-ci n'utilise pas* `compute_result.h`*.*

2.2.2 Pour maintenir la cohérence du projet

Maintenir correctement les dépendances d'un projet n'est pas facultatif.

Question 2.5. *Dans le* `Makefile`*, supprimez la dépendance entre* `print_result.o` *et* `compute_struct.h`*.*

Modifiez `compute_struct.h` *en échangeant l'ordre des champs de la structure* `compute_struct_t`*.*

Enfin, relancez `make`*.*

Question 2.6. *La compilation se passe bien et pourtant le* ***programme est incorrect*** *!*

Vérifiez-le !

Question 2.7. *Vérifiez les dépendances de ce Makefile en dessinant le graphe induit par ses règles.*

2.3 Le format des regles

Une règle est définie par un ensemble de cibles, un ensemble de sources dont elles dépendent et une liste de commandes UNIX pour les produire :

```
cibles: dependances
[TAB]   commande UNIX
[TAB]   commande UNIX
[TAB]   ...
```

2.3.1 Tabulation en début de commande

L'oubli des tabulations au début de chaque ligne de commande est une erreur commune. Dans ces cas-là, `make` produit une erreur de la forme :

```
Makefile: 14: *** missing separator. Stop.
```

L'intérêt d'effectuer la compilation avec make depuis Emacs au moyen de `M-x compile`, est que s'il y a une telle erreur dans le Makefile, `M-g n` placera directement le curseur à la ligne du Makefile contenant cette erreur.

2.3.2 Cibles multiples

La syntaxe des règles suggère que les cibles peuvent être multiples. Un même graphe de dépendances peut donc être écrit de plusieurs façons.

Question 2.8. *Testez le Makefile suivant :*

```
CC = gcc
CFLAGS = -Wall

.PHONY: re clean

all: main

parse_num.o print_result.o compute_result.o: tools.h

print_result.o compute_result.o: compute_struct.h

compute_result.o: compute_result.c compute_result.h
	$(CC) $(CFLAGS) -c compute_result.c

parse_num.o: parse_num.c parse_num.h
	$(CC) $(CFLAGS) -c parse_num.c

print_result.o: print_result.c print_result.h
	$(CC) $(CFLAGS) -c print_result.c

main.o: main.c print_result.h compute_result.h parse_num.h
	$(CC) $(CFLAGS) -c main.c

main: main.o print_result.o parse_num.o compute_result.o
```

```
	$(CC) -o main print_result.o parse_num.o compute_result.o main.o

clean:
	rm -f main *.o *~ \#*

re: clean main
```

Question 2.9. *Vérifiez que ce Makefile est équivalent au premier en dessinant son graphe de dépendances.*

`.PHONY` permet de déclarer des règles qui ne produisent pas de fichiers mais qui sont interprétés comme des commandes. En effet, ici, la règle "clean" ne produit pas de fichiers – bien au contraire, elles les détruits. On peut y faire appel à tout moment en tapant `make clean`.

2.3.3 Règles à motif

Les dépendances peuvent être exprimées à l'aide de règles générales définies par le biais de motifs (*patterns* en anglais). Ainsi, on peut décrire comment construire un fichier objet à partir d'un fichier source :

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

Cette règle signifie que, pour construire un fichier objet à partir d'un fichier source, il faut compiler le fichier .c par la commande `$(CC) $(CFLAGS) -c $<`. La séquence `$<` est une variable définie automatiquement pour toute règle et qui symbolise le premier fichier de la liste de dépendances.

Question 2.10. *Modifiez le Makefile en rajoutant cette règle.*

Il y a d'autres variables automatiques :
- `$<` : le premier fichier de la liste de dépendances ;
- `$^` : tous les fichiers de la liste de dépendances (en fusionnant les doublons) ;
- `$+` : tous les fichiers de la liste de dépendances (sans fusionner les doublons) ;
- `$@` : la cible ;

Question 2.11. *Par exemple, on peut modifier la règle précédente en :*

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

Question 2.12. *De même, la création de l'exécutable (liaison) peut s'écrire :*

```
main: main.o print_result.o parse_num.o compute_result.o
	$(CC) $(CFLAGS) -o $@ $^
```

2.3.4 Format des commandes

Les commandes qui sont exécutées par les règles sont interprétées par un shell UNIX.

Question 2.13. *Modifiez la règle de compilation pour afficher la dernière date de création du fichier cible :*

```
%.o : %.c
	@ (test -f $@ && \
	 echo Last modification of $@ was: \
	 `stat -c %y $@`.) || \
	 echo $@ does not exist.
	 $(CC) $(CFLAGS) -c $< -o $@
```

Plusieurs choses à noter :
- `@` spécifie à `make` de ne pas afficher la commande avant de l'exécuter.
- les \ servent à continuer la ligne (comme en shell). Cette règle contient donc deux commandes. La première servant à faire l'affichage de la date et la seconde qui est la compilation à proprement parler.
- les commandes shell sont interprétées par défaut par `sh`. (On peut spécifier un autre shell à l'aide de la variable d'environnement SHELL.)

La dernière utilisation des règles est l'automatisation de tâches. On peut créer des règles qui ne dépendent de rien et dont le rôle est d'exécuter une commande shell. Par exemple, la règle clean du Makefile permet de nettoyer le projet en ne laissant que les sources.

On peut appeler ces règles directement par une commande shell : `make clean`

Lorsque `make` est invoqué sans argument, c'est la première règle du fichier qui est exécutée. Ainsi, dans notre cas, la règle 'all' est exécutée. Comme elle dépend de l'existence du fichier 'main', la compilation du projet est lancée.

La règle "re" du Makefile est une erreur commune à ne surtout pas commettre. Souvent, lorsque l'on a mal spécifiée ses dépendances, on a tendance à compiler le projet par la commande : `make clean && make`. C'est exactement ce que fait la règle "re" : recompiler entièrement le projet. Or, cette utilisation de `make` remet en cause tous ses avantages, c'est-à-dire la capacité de ne recompiler que le strict nécessaire. Lors de gros projets, il est donc essentiel d'avoir bien défini ses dépendances, sinon, des attentes de l'ordre de dizaine de minutes sont à prévoir à chaque compilation !

En plus, les dépendances sont déclaratives, donc vous n'avez aucunes garanties que "clean" sera exécuté avant "main", surtout dans le cas d'utilisations parallèles de make (voir l'option "-j").

Question 2.14. *Supprimez la règle 're', ce genre de règle est à bannir.*

2.4 Variables

On définit une variable suivant le format :

```
NOM = VALEUR
```

On accède au contenu d'une variable en la déréférençant à l'aide du `$`, ainsi `$(NOM)` représente la chaîne de caractères "VALEUR".

Les variables permettent par exemple de centraliser des informations dépendantes du système d'exploitation sur lequel le projet est compilé. Par exemple, dans le `Makefile`, on a stocké le nom du compilateur à l'intérieur de la variable CC. Ainsi, si on veut utiliser un autre compilateur C, il suffit de modifier cette valeur. Pour rendre les `Makefile`-s plus facile à adapter en fonction des systèmes, on a l'habitude de mettre les déclarations de variables en tête du fichier.

Les variables peuvent aussi servir à contenir une liste de fichiers pour y faire référence par la suite.

Question 2.15. *Ainsi, dans le `Makefile`, définissez la variable $(OBJECTS) représentant les fichiers objets du projet ainsi que la variable $(PROGRAM) définissant le programme exécutable final :*

```
PROGRAM = main
OBJECTS = main.o print_result.o parse_num.o compute_result.o
```

La règle de création de l'exécutable devient donc :

```
$(PROGRAM): $(OBJECTS)
	$(CC) -o $@ $+
```

Attention, on pourrait être tenté de définir la variable `$(OBJECTS)` ainsi :

```
OBJECTS = *.o
```

Question 2.16. *Essayez de définir OBJECTS de cette façon. Que se passe-t-il ? Pourquoi ?*

On peut tout de même déduire automatiquement la liste des objets à produire en utilisant les fonctions de manipulation des variables de `make`.

Par exemple, la fonction `$(wildcard ...)` permet d'effectuer une interprétation de motif (ou pattern) shell au moment de la définition de la variable :

```
SOURCES = $(wildcard *.c)
```

La fonction `$(patsubst PATTERN, REPLACEMENT, TEXT)` permet de remplacer toutes les occurrences d'un pattern de mot dans un texte.

Question 2.17. *Ainsi, définissez :*

```
OBJECTS=$(patsubst %.c,%.o, $(SOURCES))
```

Il existe aussi `$(subst FROM,TO,TEXT)` qui permet de faire un remplacement textuel. Pour plus d'informations sur ces fonctions reportez-vous à la documentation de `make`.

2.5 Directives

Les directives sont des instructions spéciales qui agissent sur le comportement de `make`. La directive `include` permet d'inclure un `Makefile` au sein d'un autre `Makefile`. Cela permet par exemple de centraliser les informations de configuration dans un unique fichier et de l'inclure en tête de chaque `Makefile`.

Question 2.18. *Éditez un nouveau fichier `~/tp_make/Makefile` et copiez-collez nos définitions de variables dépendantes du projet.*

2.6 Options de Make

Si vous voulez spécifier à `make` quel fichier interprété comme `Makefile`, il suffit d'utiliser l'option `-f` : `make -f fichier`.

Question 2.19. *Copiez votre fichier src/Makefile dans un fichier du nom de votre choix et lancez* `make` *en utilisant ce dernier :*

```
$ cp Makefile monfichier
$ make -f monfichier
```

Lorsque l'on ne comprend pas le comportement de `make`, on peut le lancer en mode déboguage à l'aide de l'option "-d" : `make -d`.

Question 2.20. *On peut remarquer que* `make` *teste beaucoup de règles implicites de production. L'utilisation de ce comportement est l'objet de la section suivante. Que se passe-t-il si vous passez simultanément les options "-r" et "-d" ?*

L'option "-k" permet de continuer la compilation des fichiers même en cas d'erreur de certaines compilations.

On peut enfin noter l'option "-j" qui permet de lancer plusieurs compilations en parallèle lorsque c'est possible. Sur des systèmes multiprocesseurs/multicœurs, on peut ainsi gagner beaucoup de temps.

3 Utilisation courante

Cette section explique une utilisation de `make` qui repose sur des variables et des règles définies implicitement.

3.1 Regles et variables implicites

Comme la compilation de C est une tâche très courante sur les systèmes UNIX, la règle qui décrit comment compiler un fichier .c en .o est déjà définie à l'intérieur de make.

Question 3.1. *Supprimez la règle correspondante du Makefile, et lancez :*

```
$ make clean
$ make
```

La compilation des fichiers sources s'est effectuée sans qu'on ait eu besoin d'écrire la règle ! En fait, la règle correspondante suit la forme suivante :

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

Les noms de variable CC, CPPFLAGS, CFLAGS suivent des règles de nomenclatures standards sous UNIX :

- `CC` : le compilateur C ;
- `CPPFLAGS` : les options du préprocesseur (comme `-I`) ;
- `CFLAGS` : les options du compilateur (comme `-O2`).

Ainsi, on peut définir ces variables au moment de l'invocation de `make` :

```
$ make clean
$ make CFLAGS=-O2
$
```

De la même manière, la création d'exécutable est gérée par défaut par `make`.

Question 3.2. *Supprimez la commande de création de l'exécutable et faites ensuite :*

```
$ make clean
$ make
```

La règle implicite correspondante est de la forme :

```
%: %.o
    $(CC) -o $(CFLAGS) $(LDFLAGS) $@ $^
```

`LDFLAGS` correspond aux options du programme qui effectue la liaison (usuellement `ld`). On peut spécifier ici les options de type "-Ldirectory" ou bien "-llibrary".

3.2 Applications recursives

Il est souvent utile de faire une application récursive de `make` dans les sous-répertoires d'un projet.

Par exemple, on peut définir un `Makefile` à la racine du projet pour factoriser l'appel de la règle `clean` dans tous les sous-répertoires du projet :

```
SUBDIRS = src doc test
clean:
     for dir in $(SUBDIRS); do \
       $(MAKE) clean -C $$dir; \
     done
```

Question 3.3. *Effectuez une copie de votre environnement de travail pour y pouvoir revenir plus tard.*

Placez les sources .c et .h relatives à la computation du résultat (`compute_*`) dans un sous-répertoire `compute/` et les sources relatives au parsing et à l'affichage du résultat (`print_*` et `parse_*`) dans un sous-répertoire `frontend/`.

Écrivez le `Makefile`-s pour les sous-répertoires `compute/` et `frontend/` et le `Makefile` pour la racine. Est-ce que un changement dans un fichiers dans les sous-répertoires force la recompilation de `main` dans la racine ?

Avant de poursuivre, revenez à la version de votre environnement de travail sans les sous-répertoires et sans `Makefile` récursive.

3.3 Generation automatique des dependances

Il est possible de définir automatiquement la plupart des dépendances d'un projet à l'aide de programmes comme `makedepend` ou `gcc`.

Question 3.4. *Voici l'exemple fourni dans la documentation de make :*

```
%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm -f $@.$$$$
```

Cherchez la documentation de l'option "-M" de `gcc` et essayer de l'utiliser avec les fichiers .c de votre projet.

Il créé un fichier .d pour chaque fichier source.

Il suffit ensuite d'inclure les fichiers générés à la fin du `Makefile` (vous devez pouvoir le faire sans aide).

3.4 Un Make le \generique"

Après toutes ces explications, nous aboutissons à un `Makefile` générique qui devrait vous servir pour des projets C de taille moyenne :

```
# Project specific variables
PROGRAM=
SOURCES=
SUBDIRS=
CC=
CFLAGS=
CPPFLAGS=
LDFLAGS=

# Generic part of the Makefile
OBJECTS=$(patsubst %.c,%.o, $(SOURCES))

.PHONY: all clean

all: $(PROGRAM)

%.d: %.c
        $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
        sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
        rm -f $@.$$$$

$(PROGRAM):$(OBJECTS)

clean:
     for dir in $(SUBDIRS); do \
       $(MAKE) clean -C $$dir; \
     done;
     rm -f main *.o *~ \#*

include $(SOURCES:.c=.d)
```

4 Batteries de test

Le sous-répertoire `tests/` contient une batterie de tests pour notre programme de division euclidienne. Chaque test est formé par :

- un fichier avec extension `.in`, qui contient un entrée pour `main`. Le format est une liste des arguments à passer sur la ligne de commande ; par exemple : "`15`" si on veut tester `main` en appelant "`main 15`"
- un fichier avec le même nom de son homologue `.in`, mais extension `.expected`, qui contient la sortie correcte quand on appelle `main` avec l'entrée de test. Par exemple : "`3 0`" pour l'entrée "`15`".

Question 4.1. *Lisez la documentation de l'outil* `diff`*, avec* `man diff` *ou dans Emacs avec* `M-x man`*. Comment pouvez vous vérifier si deux fichiers ont le même contenu avec* `diff` *?*

Question 4.2. *À l'aide de* `diff`*, vérifiez que la batteries de tests corresponde au comportement de* `main`*.*

Question 4.3. *Implémentez une règle* `test` *dans votre* `Makefile` *qui lance tous les tests disponibles dans le sous-répertoire* `tests/`*. L'invocation de* `make` *doit retourner un message d'erreur si au moins un de tests ne correspond pas au comportement de* `main`*.*

Question 4.4. *Vérifiez votre règle en ajoutant un test dont le résultat ne doit pas correspondre à une implémentation correcte de la division euclidienne.*

5 Vers un Make le portable

Pour pouvoir compiler son projet sur plusieurs environnements. Il faut se donner les moyens d'adapter les commandes du Makefile en fonction des caractéristiques du système d'exploitation ou des bibliothèques installées.

Par exemple, sous certains UNIX, le compilateur C n'est pas gcc, mais cc (c'est le cas de la machine de l'UFR ouindose.informatique.univ-paris-diderot.fr – à laquelle vous avez accès – qui est sous SUN/OS SOLARIS) c'est pour ça qu'on utilise le contenu de la variable $(CC) et non pas gcc pour compiler.

Aussi, certains modules du projet, par exemple une interface graphique, sur certains systèmes, ne peuvent pas être *compilables*. Make offre la possibilité d'interpréter certaines parties des `Makefile`-s conditionnellement à l'aide des commandes `ifeq VAR VALUE` et `ifdef VARIABLE` qui permettent de tester la valeur d'une variable ou bien, simplement, son existence.

On peut donc rajouter ces lignes dans un Makefile :

```
# If we are working with gcc, we can compile some
# other features.
ifeq "CC" "gcc"
OBJECTS= ext_features.o $(OBJECTS)
endif

# If the Qt library is present, we can compile a GUI.
ifdef QT_IS_PRESENT
OBJECTS= gui.o $(OBJECTS)
endif
```

Question 5.1. *Utilisez ces instructions de compilation conditionnelle pour compiler l'unité* `print_result_with_system.o` *à la place de* `print_result.o` *suivant la valeur d'une variable* `USE_SYSTEM_TO_PRINT`*.*

6 Pour en savoir plus

La documentation de make s'obtient ainsi :

```
info make
```

Pour des projets de plus grande dimension et nécessitant une importante configurabilité, on utilise les `autotools`, c'est à dire `autoconf` et `automake`.