Introduction aux Makefile

REF. OSP002

GAYET Thierry
Thierry.Gayet@laposte.net

Plan de la présentation

- Introduction
- Outils de transformation
- Règles de transformation
- Make
- Création d'un makefile pas à pa
- Conclusion

Introduction 1/2

Sous Unix il est courant de faire appel aux commandes suivantes :

```
# make                  Interprétation du fichier Makefile
                         du chemin courant
```

Ou bien :

```
# ./configure           Génération d'un script propre à la
                        plateforme
# make                  Compilation
# make install          Installation du programme
```

Cette commande lance immédiatement une *compilation automatique* en *gérant les dépendances* (date de la dernière modification, ...) en ne régénérant que ce qui est nécessaire.

« C'est en gros une sorte d'automate intelligent ,
Un parseur de règles »

Introduction 2/2

Le programme **make** est un programme permettant de réaliser des transformations d'un format à un autre. Dans 90%des cas il est utilisé pour l'automatisation des tâches de compilation et de linkage.

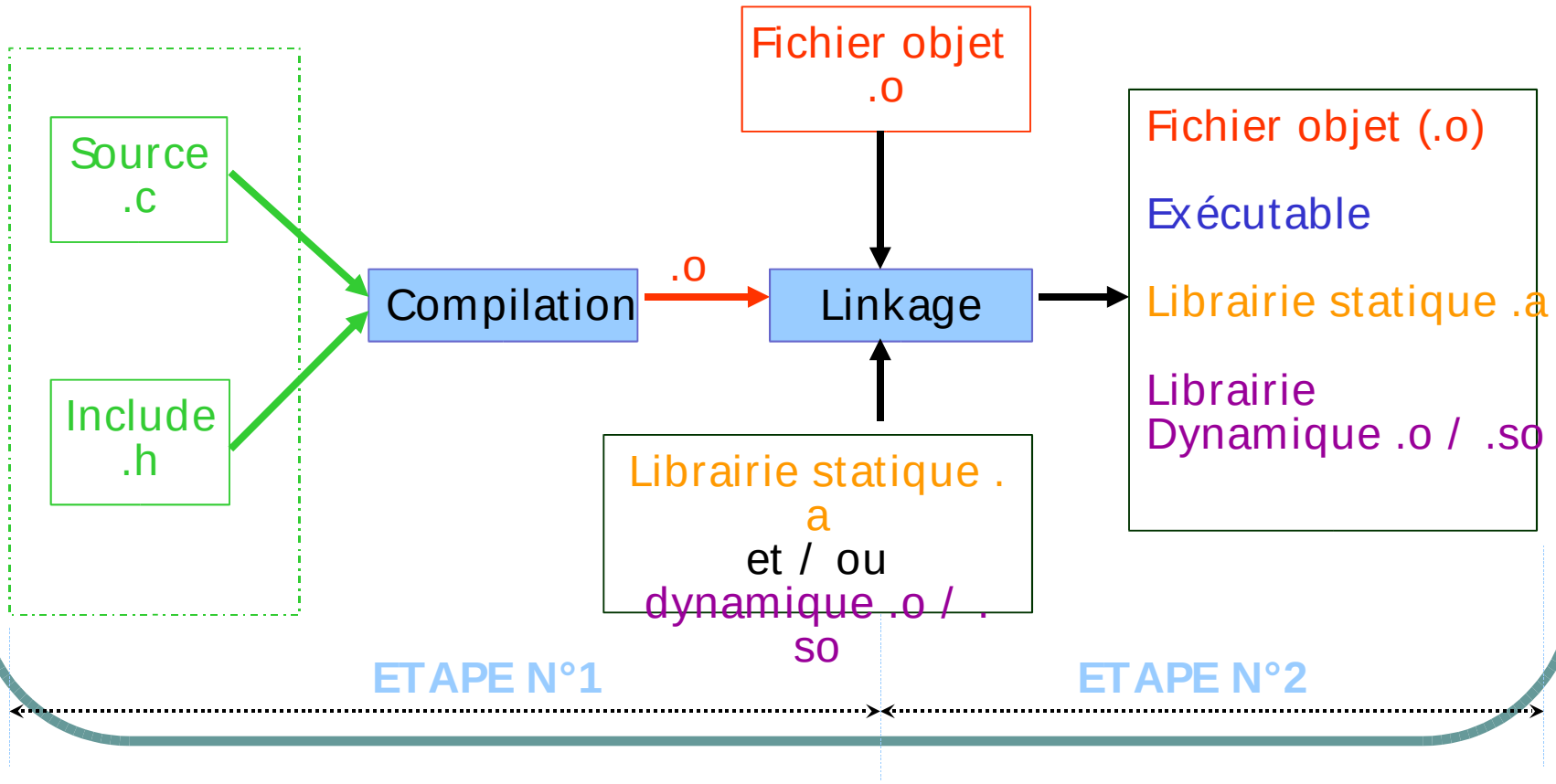
Make a pour rôle de prendre en compte tout ce que nous venons de voir, mais aussi :

- de gérer en plus les dépendances sur ce qui est déjà crée et ce qui ne l'est pas.
- de prendre en compte les plate formes (si besoin).
- de gérer aussi bien la compilation que le linkage.
- de créer des exécutables, des librairies statiques et dynamiques, etc ...

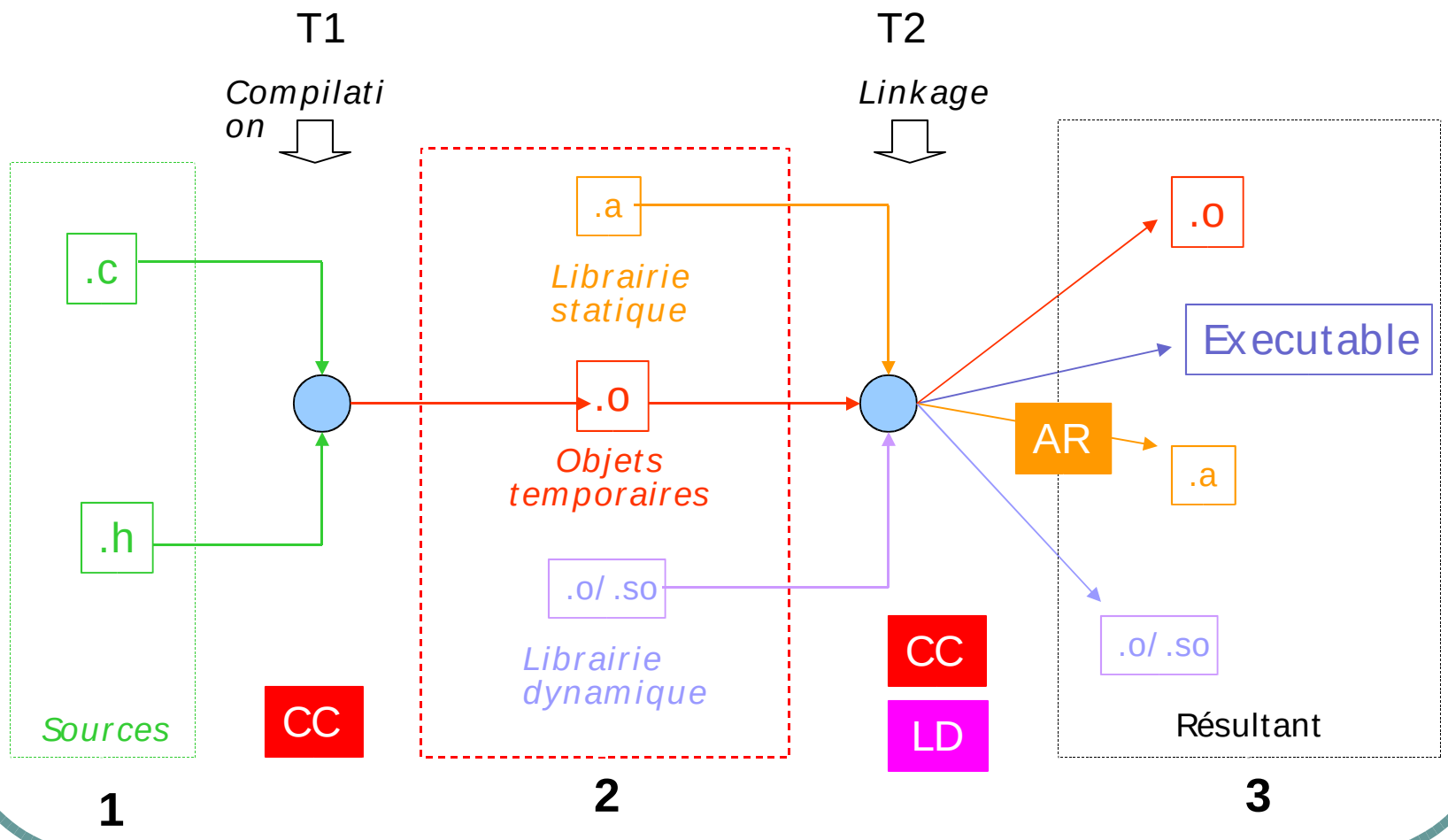
LES OUTILS DE TRANSFORMATION

Outils de transformation 1/ 4

Vue complète de la chaîne de création (compilation et linkage) :



Outils de transformation 2/ 4



Outils de transformation 2/4

Compilateur C :

GNU Linux	gcc						
SUN OS/Solaris	acc	ou	gcc				
IBM Aix	cc_r	ou	cc	ou	gcc		

CC

(*) le compilateur gnu gcc existe pour toutes les plateformes.

Linker / editeur de liens :

GNU Linux	gcc					et	ld	
SUN OS/Solaris	acc	ou	gcc		et	ld		
IBM Aix	cc_r	ou	cc	ou	gcc	et	ld	

LD

Outils de transformation 3/4

Outils de création de librairie statique :

Pour toute les plate-forme : ar

AR

ranlib

RL

Note sur les exécutables :

Une fois un exécutable généré, il est conseillé de le rendre réellement exécutable (s'il ne l'est déjà) :

chmod u=rwx result

(u=rwx peut être remplacé par son code octal : 04755)

REGLES DE TRANSFORMATION

ET DE DEPENDANCE

Règles de création 1/5

Compilation simple d'un source .c/ .h pour obtenir un objet .o :

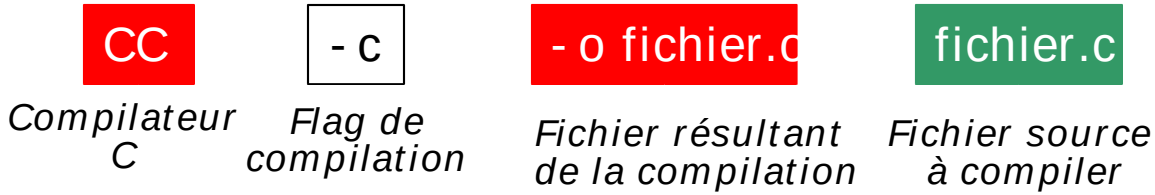
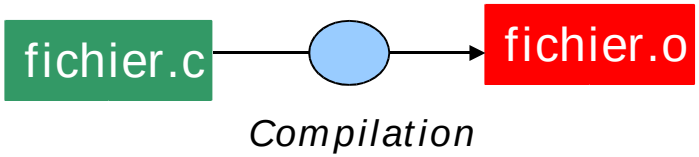


Diagramme de transformation :



Quand bien même il peut y avoir un outil ou une règle pour une transformation, Il peut être du goût de chacun d'utiliser un compilateur ou un linkeur plutôt qu'un Autre.

Règles de création 2/5

Linkage d'un seul fichier source .c :

A partir du moment où un exécutable est requis, il est nécessaire de faire intervenir la phase de linkage.

CC - c - o result fichier.c *2 étapes en 1*

ou

CC - c - o result.o fichier.c *Phase de compilation*

LD - o result fichier.o *Phase de linkage*

Diagramme de transformation :

Règles de création 3/5

Linkage de n fichiers sources .c :

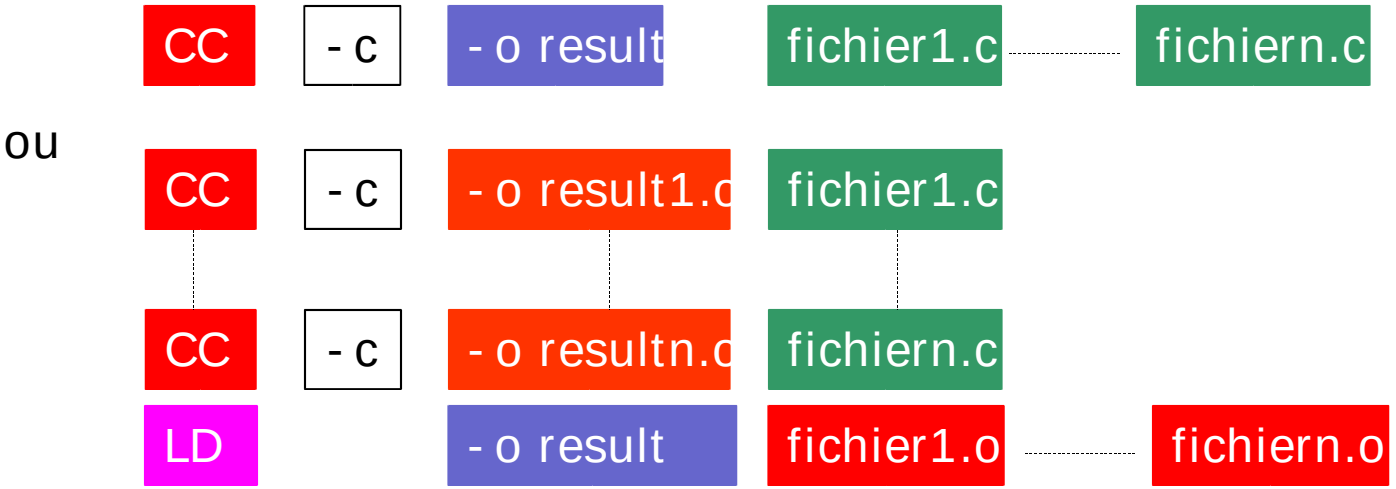
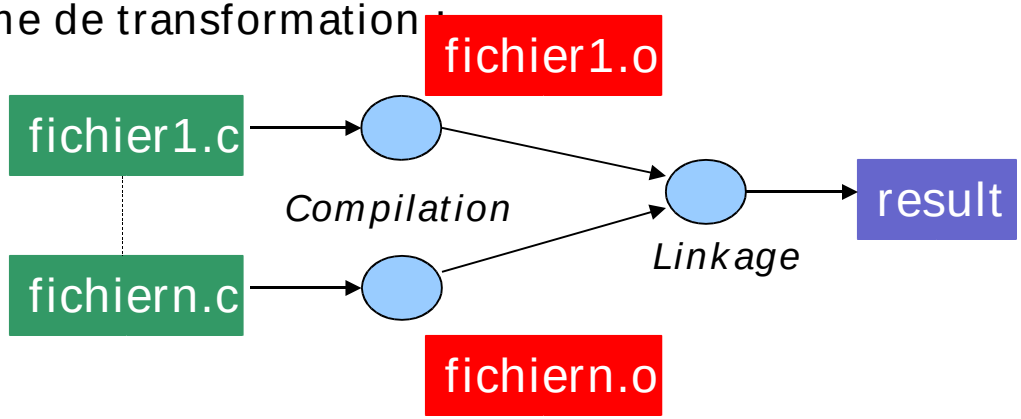


Diagramme de transformation :



Règles de création 4/5

Création d'une librairie statique .a (toute plateforme) :

```
AR  - rv  lib_statique.a  *.o
```

Il est juste nécessaire de lancer le programme ranlib sous SUN OS afin de Remettre à jour la table de symboles de la librairie :

```
RL  lib_statique.a
```

Extraction des objets .o d'une librairie statique .a :

```
AR  - x  lib_statique.a
```

Règles de création 5/5

Création d'une librairie dynamique :

GNU Linux :

CC	- c	- shared	- o lib_dynamic.so	fichier.c
CC		- shared	- o lib_dynamic.so	fichier.o

SUN OS/ Solaris / IBM Aix :

CC	- c		- o lib_dynamic.so	fichier.c	
LD	- dY	- G	- o lib_dynamic.so	fichier.o	*SUN*
LD	- dY	- G	- o	fichier.o	*IBM*

GNU

LE PROGRAMME MAKE

Appel

Le binaire du programme **make** est situé dans le répertoire : `/usr/bin`

Par défaut ce programme prend en compte le fichier makefile ou Makefile du répertoire courant. Il suffit donc de lui faire appel :

```
# make
```

Néanmoins, il est possible de lui spécifier un autre nom de fichier :

```
# make –f Makefile2
```

Si make ne trouve aucun fichier, ce dernier afficher le message suivant :

```
make: *** No targets specified and no makefile found.  Stop.
```

Point d'entrée

Dans la majorité des cas le Makefile est appelé directement, c'est-à-dire :

```
# make
```

Dans ce cas, le Makefile ira au point d'entrée par défaut (**do**). Cela se représentera dans le fichier Makefile par le préfixe suivant :

```
do:
```

Cependant, il peut être parfois nécessaire de lui passer des noms afin de lui préciser d'autres points d'entrée :

```
# make clean     ou encore     # make install
```

Le premier appel est souvent utilisé pour permettre de nettoyer l'environnement de développement. Le second pour installer une application sur une destination précise.

Dans ces derniers cas le point d'entrée est respectivement `clean:` ou `install:` et non `do:`

Commentaires

Suivant la philosophie du Shell Unix, un commentaire dans un Makefile est symbolisé par un dièse : #

Exemple de commentaire :

```
# Cette ligne est en commentaire jusqu'au retour chariot
```

Contrairement au langage C, un commentaire nécessitant plusieurs lignes, ne pourra utiliser les symboliques suivantes :

```
/*
   Commentaire
   sur plusieurs
   lignes
*/
```

FAUX

```
//
// Commentaire
// sur plusieurs
// lignes
//
```

FAUX

```
#
# Commentaire
# sur plusieurs
# lignes
#
```

CORRECT

Un message est prioritaire sur tout le reste ; en d'autres terme un dièse inhibe le reste de la lign

Macros 1/2

Les fichiers makefile permettent l'utilisation intrinsèque de macros. Ces dernières permettent de clarifier la syntaxe en factorisant les informations :

Exemple de définition de macros :

```
# Définition d'une macro SRC qui sera égale à ./src
SRC = ./src
```

Par convention, le nom des macros est en majuscule !!

La lecture et donc l'exécution d'un Makefile étant linéaire, cette macro peut être utilisée (après la définition) en utilisant la syntaxe suivante : $(NOM_VARIABLE)

```
src_test = $(SRC)/test
```

Cela créera : `src_test = ./src/test` *(le contenu de SRC sera substitué dans la chaîne)*

D'autre part, cela permet de récupérer le contenu d'une variable d'environnement :

CC = $(COMPILATEUR_C)

(ici la variable d'environnement compilateur_c sera remplacée par son) contenu préalablement positionné)

Macros 2/2

Technique utilisable pour s'assurer qu'une chaîne soit bien assigné à une valeur par défaut :

```
ARCHREF = Linux
ARCH    = $(ARCHREF$(CIBLE))$(CIBLE)
```

Ici si la variable d'environnement CIBLE existe, la condition $(ARCHREF$(CIBLE)) ne sera pas validée et ARCH sera égal à CIBLE. Dans le cas contraire il sera égal à ARCHREF.

En détaillant, la syntaxe suivante $(ARCHREF$(CIBLE)) revient à rechercher la chaîne CIBLE dans la seconde chaîne ARCHREF . Il est donc logique que la recherche soit infructueuse puisque ARCH est différent de ARCHREF dans le cas où CIBLE est positionné. Dans le second cas, CIBLE étant nul, la recherche est inactive puisque qu'il n'y a rien à rechercher dans ARCHREF ; ARCH est donc égal à ARCHREF.

Il est à noter que sous Unix, une chaîne inexistante ne retourne pas d'erreur mais une chaîne vide. En effet :

```
ARCH = $(CIBLE)
```

Si CIBLE n'existe pas ARCH sera équivalent à un chaîne vide

D'un makefile à l'autre

Dans un makefile il est possible d'appeler une série de commande :

```
RMOBJ = @rm –f *.o

clean:
        $(RMOBJ)
```

La commande en ligne apparaîtra ici. La faire précéder par un arobase @ permet de ne pas l'afficher tout en continuant de l'exécuter.

Pour cumuler plusieurs commande sur une seule ligne, il est possible d'utiliser la syntaxe suivante :

```
@(cd src; rm –f *.o; make)
```

va dans le répertoire src, supprime les .o puis lance le makefile

Il est possible à partir d'un makefile d'en appeler un second puisque make est aussi une commande.

Variables automatiques

Make autorise un certain nombre de variables automatique dont les plus courantes sont :

$@ Fichier destination après transformation

$< Fichier source avant transformation

Définition d'une transformation

Quand bien même make accepte un certain nombre de règles implicites, il lui est possible de redéfinir ces dernière. Avec la même logique, il est possible de lui en ajouter.

Définition personnalisée de règle de transformation d'un format à un autre ; d'une extension à un autre :

```
.SUFFIXES: .o .c

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

Ici lorsque make trouvera un .c et qu'il souhaite obtenir un .o, il passera dans la règle ci- dessus. Cette dernière est assez générale via l'utilisation des variables automatiques.

Premier Makefile 1/2

```
# Exemple de Makefile :

# Répertoire de travail :
SRC  = ./src
INC  = ./src/include
BIN  = ./bin

# Nom de l'exécutable à générer :
BINTEST = $(BIN)/test

# Compilateur C Ansi :
CC  = gcc

# Flags de compilation / débug / warning
CFLAG = -g -Wall -O2 -I$(INC)

do: $(BINTEST)

$(BINTEST)
        $(CC) -c $(CFLAG) $(SRC)/test.c -o $@
```

Point d'entrée →

Chaîne de création :

test = test.c + test.h

Appel du make :

```
# make
```

Source : **test.c** + **test.h**	**$<**
Binaire cible : **test**	**$@**

.c / .h → **.o** → **bin**

Equivalent à :

```
# gcc -c -g -Wall -O2 -I../src/include
./src/test.c -o ../bin/test
```

*(Ici **$@** est égal à **BINTEST**)*

Premier Makefile 2/2

Commande :

```
# gcc -c -o ./bin/test ./
   src/test.c
```

Dans la mesure où il n'y a qu'un seul fichier (**test.c** / **test.h**) à compiler il est possible de générer l'exécutable souhaité (test) en une seule commande.

CC gcc

LD gcc

Fichier d'entrée : **test.c**

Fichier de sortie : **test**
Type : exécutable

Avant tout il faut avoir bien en tête le schéma de transformation d'un fichier .c / .h vers un fichier exécutable.

APPRENTISSAGE

PAR L'EXEMPLE

Préambule

La philosophie pour arriver à écrire un Makefile est la suivante :

Quels fichiers sources avons nous ? Quel format de fichier ? Quel langage ? Y a-t-il une arborescence précise pour le projet (include/ librairies/ binaire/ source) ?

Que voulons nous générer comme fichiers (exécutable, librairie statique/ dynamique, ...) ?

Connaissons nous toutes les règles de transformation ?

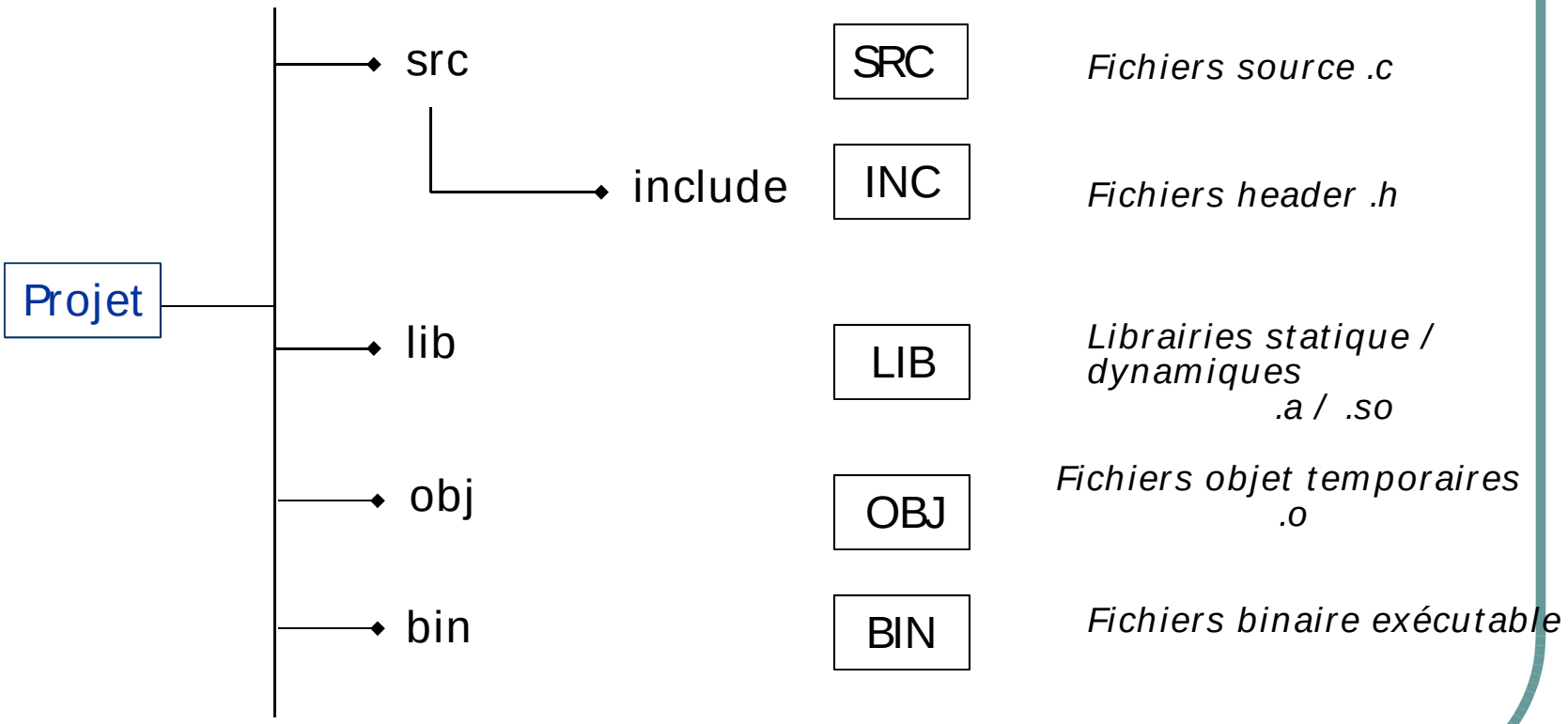
Souhaitons nous un fonctionnement multi-plateforme / multi système d'exploitation ?

Tout comme dans une majorité de domaine, il est possible de générer une multitude de Makefile différents mais qui réaliseront exactement la même chose. Il en va de chacun pour personnaliser ce dernier et utiliser les outils que nous avons l'habitude d'utiliser.

« Nous allons donc prendre un exemple
et le retranscrire pas à pas au sein d'un Makefile »

Arborescence

Exemple standard d'une arborescence :



Ce que l'on souhaite

Disposant de plusieurs fichiers sources, nous souhaitons générer un binaire et une librairie statique.

Les source sont en langage C **.c** sont localisés dans le répertoire ./src

Les includes .h dans le répertoire ./src/include

Les objets temporaires dans : ./obj

La librairies dans ./lib

Nom souhaité : **libtest.a**

Et enfin le binaires **exécutables** dans ./bin :

Nom souhaité : **test**

Les sources

Les fichiers utilisés pour l'exemple :

Les sources .c :

./ src/ test1.c
./ src/ test2.c
./ src/ test3.c

./ src/ libtest.c

Pour le binaire

c-an
si

Pour la librairie statique

Le source .h :

./ src/ include/ test.h

Création pas à pas d'un Makefile

Bien que les Makefiles soient adaptés à la compilation / Linkage de n fichiers ils sont d'autant plus adaptés aux programmes découpés de façon modulaire :

./bin/test = ./src/**test1.c** + ./src/**test2.c** + ./src/**test3.c** + /src/include/**test.h**

./lib/lib**test.a** = ./src/lib**test.c** + /src/include/**test.h**

Pour ce faire, il est possible de créer les chaînes de fabrication :

./obj/lib**test.o** = ./src/lib**test.c** + ./src/include/**test.h** (compilation)

./bin/**test** = ./obj/**test1.o** + ./obj/**test2.o** + ./obj/**test3.o** + **/lib/libtest.a** *(linkage)*

./obj/**test1.o** = ./src/**test1.c** + ./src/include/**test.h** (compilation)

./obj/**test2.o** = ./src/**test2.c** + ./src/include/**test.h** (compilation)

./obj/**test3.o** = ./src/**test3.c** + ./src/include/**test.h** (compilation)

Création pas à pas d'un Makefile

Les commandes de compilation unitaire peuvent être retranscrites de la façon suivante :

./obj/**libtest.o** = ./src/**libtest.c** + ./src/include/**test.h**

gcc -c -o ./obj/**libtest.o** –i./src/include ./src/**libtest.c**

et

./obj/**testx.o** = ./src/**testx.c** + ./src/include/**test.h**

gcc -c -o ./obj/**testx.o** –i./src/include ./src/**testx.c** (1)

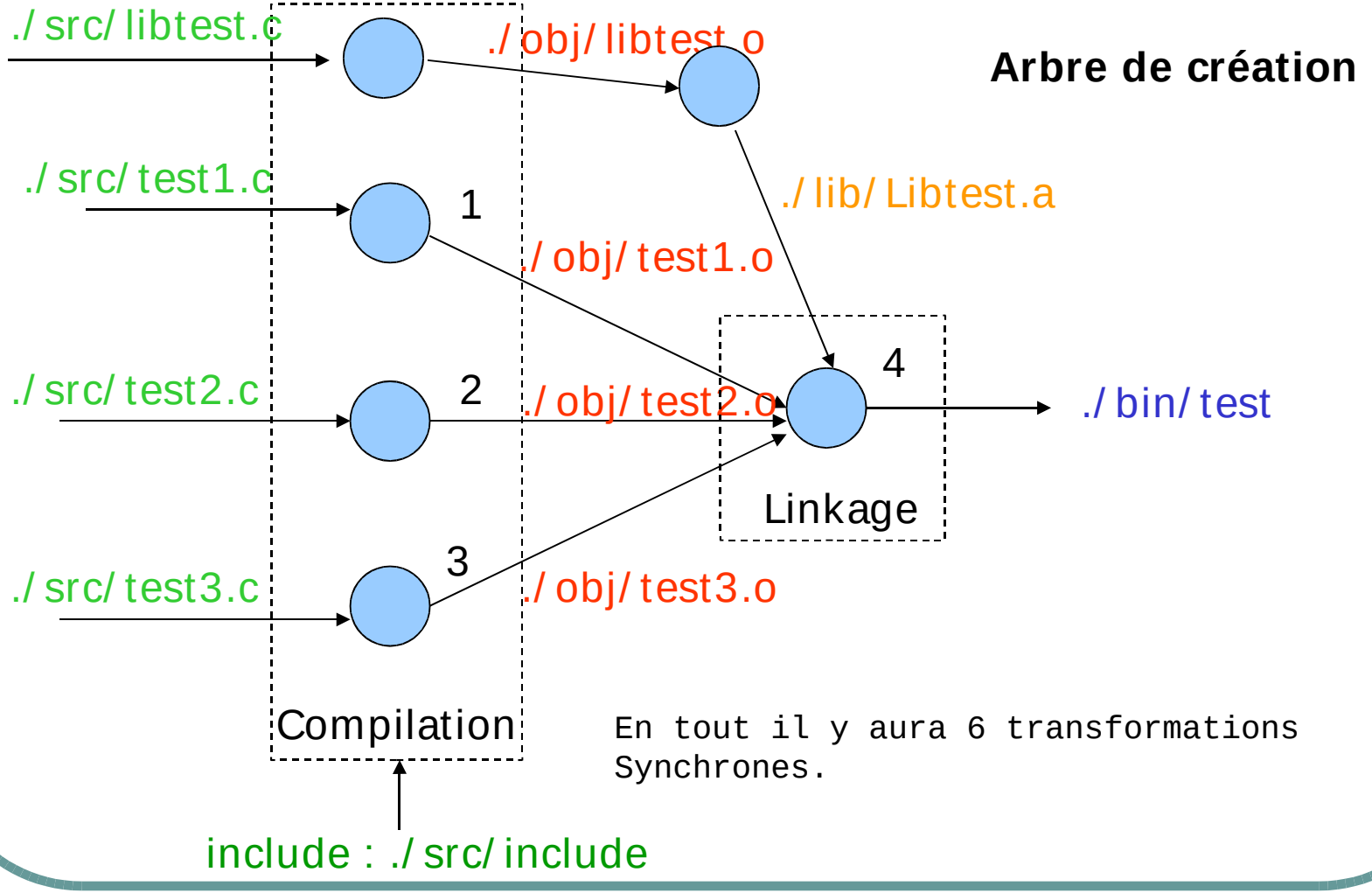
(1) *avec x compris entre 1 et 3.*

De même pour la commande de linkage :

./bin/**test** = ./obj/**test1.o** + ./bin/**test2.o** + ./bin/**test3.o** + /lib/**libtest.a**

gcc -o ./bin/**test** ./obj/**test1.o** ./bin/**test2.o** ./bin/**test3.o** /lib/**libtest.a**

Création pas à pas d'un Makefile



Enfin le makefile


```
# Makefile du test :

# Répertoire de travail :
SRC        = ./src
INC        = ./src/include
LIB        = ./lib
OBJ        = ./obj
BIN        = ./bin
TARGETDIR = /usr/local/progtest

# Nom des fichiers à générer :
BINTEST = $(BIN)/test
LIBTEST = $(LIB)/libtest.a
TYPEOS = LINUX

# Compilateur C Ansi :
CC         = gcc
AR         = @ar –rv
RMOBJ    = @rm –f $(OBJ)/*.o
ECHO     = @echo
MKREP    = @mkdir $(TARGETDIR)
MKDROIT = @chmod u=rwx
CPPROG  = @(cp $(BIN)/* $(TARGETDIR); cp $(LIB)/* $(TARGETDIR))

# Flags de compilation / débug / warning
CFLAG = -g –Wall –O2 –I$(INC) -D$(TYPEOS)
```

...

Suite du Makefile

make → *Traitement linéaire/ séquentiel*

Point d'entrée par défaut

```
do: $(LIBTEST) $(BINTEST)

$(LIBTEST):
        $(CC) –c $(CFLAG ) $(SRC)/ libtest.c –o $(OBJ)/ libtest.o
        $(AR) $@ $(OBJ)/ libtest.o

$(BINTEST):
        $(CC) –c $(CFLAG ) $(SRC)/ test1.c –o $(OBJ)/ test1.o
        $(CC) –c $(CFLAG ) $(SRC)/ test2.c –o $(OBJ)/ test2.o
        $(CC) –c $(CFLAG ) $(SRC)/ test3.c –o $(OBJ)/ test3.o
        $(CC) $(LIBTEST) $(OBJ)/ test1.o $(OBJ)/ test2.o $(OBJ)/ test3.o - o $@
        $(MKDROIT) $@
        $(ECHO) «Programme compilé.»

clean:
        $(RMOBJ)
        $(ECHO) «Environnement nettoyé.»

install:
        $(MKREP)
        $(CPPROG)
        $(ECHO) «Programme installé.»

start:
        $(ECHO) «Lancement du programme»
        $(BINTEST)
```

1

2

Génère la librairie statique.

Génère le binaire en utilisant la librairie.

make clean →

make install →

make start →

Seconde méthode

Une autre possibilité aurait été d'écrire :

```
$(LIBTEST):    $(OBJ)/libtest.o
               $(AR) $@ $(OBJ)/libtest.o

$(BINTEST): $(OBJ)/test1.o $(OBJ)/test2.o $(OBJ)/test3.o
            $(CC) $(LIBTEST) $(OBJ)/test1.o $(OBJ)/test2.o $(OBJ)/test3.o - o $@
            $(MKDROIT) $@
            $(ECHO) «Programme compilé.»
```

```
.SUFFIXES: .o .c

.c.o:
        $(ECHO) "génération de" $@ "à partir de" $<
        $(CC) -c $(CFLAG) -o $@ $<
```

Pour la génération de **libtest** ou **bintest**, make regardera s'il possède l'équivalent en .c qu'il sait compilé par la règle définit par le suffixe.

Résumé

4 Point d'entrée :

make

génère la librairie **PUIS** le binaire

make clean

nettoie l'environnement des objets et binaires précédents

make install

crée le répertoire et copie le programme dans un emplacement choisi

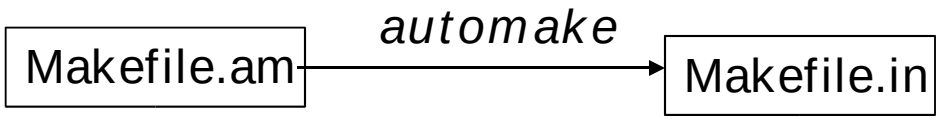
make start

lance l'exécution du programme

Evolutions

Une évolution possible des makefiles est l'utilisation des programmes gnu open source Suivant :

- automake
- autoconf



En effet, ils permettent de créer un fichier de configuration (Makefile.in) utilisable par la suite dans les Compilations. De plus, il vérifie si la totalité de la chaîne complète que make utilisera par La suite (présence des binaires et librairies utiles pour les transformations, version des Librairies).

Une fois le fichier de config généré (via ./configure), il est possible de lancer successivement sa compilation (make) et enfin son installation (make install).

http://www.gnu.org/software/automake/
http://www.gnu.org/software/autoconf/
http://www.amath.washington.edu/~lf/tutorials/autoconf/

Autres utilisations

Comme nous l'avons vu les makefile sont utilisés principalement pour le développemen

Néanmoins, d'un point de vue conceptuel, il en résulte une transformation de formats.

Il est possible de l'utiliser pour les utilisations suivantes :

- transformation du latex (.tex) en Postscript (.ps) ou Acrobat (.pdf)
- transformation du .xml avec un .xsl+.dtd pour générer diverse sorties.
- etc...

Conclusion

Jespère que ce document vous rendra service.

Pour davantage d'aide sur les différentes commandes :

```
# man commande

# commande --help

Les how to sur la compilation (glibc, make, ld, gcc, gdb …).

Les forums dédiés (Newsgroups).

les sites dédiés.

Et bien entendu google.fr, gnu.org, freshmeat.net
```

Bibliographie

- Un livre de référence :

Managing Projects with GNU make, 3rd Edition
By Robert Mecklenburg
3rd Edition November 2004
ISBN: 0-596-00610-1
300 pages
Edition O'REILLY

- Deux liens à retenir :

http://www.gnu.org/software/make/manual/make.html
http://www.madchat.org/coding/c/c.ansi/
http://www.oreilly.com/catalog/make3/index.html

Making of...

Diaporama réalisé à partir des deux projets open source suivant :

 OpenOffice.fr

 Gimp

Fin

Bonne programmation