Université d'Aix-Marseille III - Licence math-info 1ère année

I2 - Notes de cours (2) : Utilisation du système d'exploitation Unix

Nous allons ici présenter le système d'exploitation Unix dans la perspective de son utilisation courante. Ultérieurement, nous élargirons notre propos en présentant la manière dont les systèmes d'exploitation gèrent les processus et la mémoire.

Qu'est-ce qu'un système d'exploitation ?

Un système d'exploitation [operating system] est un programme de contrôle qui gère les ressources de votre machine : mémoire, programmes, périphériques d'entrées-sorties (écran, clavier, disque dur, souris, imprimante, ...), etc. Il sert d'interface entre, d'une part, l'utilisateur et ses programmes, et d'autre part, la machine en tant qu'ensemble de composants matériels.

Il est composé de plusieurs programmes destinés à des tâches spécifiques que l'utilisateur peut aussi exécuter dans ses propres programmes. Ex : lire un caractère au clavier, charger un fichier, etc.

UNIX est un système d'exploitation *multi-processus* (plusieurs programmes peuvent s'exécuter simultanément sur la machine) et *multi-utilisateurs* (plusieurs utilisateurs peuvent accéder simultanément à la machine). Les utilisateurs partagent certaines ressources de la *machine hôte* (microprocesseur, mémoire, espace disque) en y accédant à partir d'un *terminal* (machine utilisée pour la communication avec l'hôte). Linux est une implémentation de système UNIX.

1 Utilisation d'Unix

1.1 Connexion au système

Pour pouvoir utiliser le système, il faut préalablement s'y connecter (se *loguer*). Vous disposez d'un *compte* personnel identifié par votre *nom de login* (public) et d'un *mot de passe* pour garantir que vous soyez le seul à accéder à votre compte.

Vous pourrez changer votre mot de passe grâce à la commande `passwd` (dans un shell).

Pour vous déconnecter du système (se *déloguer*), il faudra utiliser la commande `exit`[1].

1.2 Le shell

Un *shell* est un programme interpréteur de commandes UNIX qui permet d'exploiter les ressources du système. Dans un environnement textuel, vous vous retrouverez dans un shell dès que vous vous serez connecté. Le shell affiche une *invite* [*prompt*] pour indiquer que vous pouvez rentrer la commande au clavier. Cette invite peut être '%' ou '$' suivant le type de shell que vous utilisez (sh[2], ksh, csh, bash, zsh, ...). Exemple d'exécution de commandes :

```
% date
ven sep 15 12:32:15 CEST 2000
% who
prcovic  tty3     Sep 15 09:23
%
```

En plus des caractères alphanumériques normaux, vous pouvez utilisez des caractères spéciaux, accessibles en maintenant la touche *Ctrl* appuyée et en tapant une lettre du clavier, qui produiront différents effets :

1. Sauf si le shell est dans une fenêtre graphique et alors c'est seulement la fenêtre qui se ferme.
2. Remarque : sh équivaut à bash sous Linux.

^C : arrêt de la commande en cours d'exécution, ^D : caractère fin de fichier, ^H : effacement du caractère avant le curseur, ^W : effacement du mot précédent, ^U : effacement de la ligne, ^S : arrêt de l'affichage, ^Q : reprise de l'affichage.

Dans les shells récents, les flèches haut et bas permettent de remonter et de descendre dans la liste des commandes qu'on a entrées précédemment. Par ailleurs, l'appui sur la touche Tab permet de compléter un nom de commande ou de fichier dont on a déjà tapé le début.

1.3 Le système de fichiers

Les fichiers sont des regroupements de données ou des programmes qui sont stockés en dehors de la mémoire principale et qui sont désignés par un nom. Il existe 3 types de fichiers : les *fichiers ordinaires*, les *répertoires* [directory] et les *fichiers spéciaux*. Les répertoires contiennent les informations sur les fichiers qu'ils regroupent. Les fichiers spéciaux sont des représentations internes de dispositifs physiques (clavier, écran, imprimante, ...). Ils permettent d'accéder à ces dispositifs comme s'ils étaient des fichiers ordinaires.

1.3.1 Les fichiers ordinaires

Il existe deux grands types de fichiers ordinaires : les fichiers texte et les fichiers binaires. Les fichiers texte (= fichiers ASCII) sont affichables. Ils contiennent des textes qui servent de données mais sont aussi éventuellement exécutables (scripts). Les fichiers binaires contiennent des données non textuelles qui ne sont pas affichables lisiblement. Ces fichiers sont destinés à être utilisés par des programmes. Ils peuvent aussi être exécutables (et ce sont alors des programmes).

1.3.2 Les répertoires

Les répertoires contiennent des fichiers ou des répertoires, créant ainsi une hiérarchie arborescente de fichiers.

Le répertoire racine [root] est désigné par '/'. Il contient entre autres les répertoires `bin` (fichiers binaires), `usr` (fichiers utilisateurs), `tmp` (fichiers temporaires), `home` (répertoires des comptes des utilisateurs), `dev` (fichiers spéciaux), etc.

On peut faire référence à un fichier ou un répertoire grâce à son chemin absolu (à partir de la racine) ou son chemin relatif (à partir du répertoire où l'utilisateur se trouve). Un chemin est représenté par la suite des répertoires emboîtés qui permettent de parvenir au fichier désigné. On sépare les noms de répertoire par '/'. Pour désigner qu'on a affaire à un chemin absolu, il faut le préfixer par '/'. Le répertoire d'accueil, le répertoire courant et le répertoire parent sont respectivement désignés par '∼', '.' et '..'.

On peut désigner un fichier grâce à des symboles qui remplacent des lettres. Le symbole '*' correspond à une suite de caractères, '?' à un seul caractère. '[' et ']' délimitent un ensemble de caractères possibles, '{' et '}', un ensemble de mots possibles. Ex : `[abc]*` représente les noms commençant par 'a','b' ou 'c'.

1.3.3 Propriétés des fichiers

Les droits d'accès en lecture, écriture et exécution d'un fichier sont modulables en fonction de l'ensemble d'utilisateurs auquel vous appartenez : vous, votre groupe ou tout le monde. La commande `chmod` permet d'autoriser ou d'interdire un type d'accès donné à un ensemble d'utilisateurs donné. La commande `ls` avec l'option `-l` permet d'afficher les différents droit d'accès attachés à chacun des fichiers.

Un fichier dont le nom commence par '.' est un *fichier caché* : il n'apparaît pas dans la liste des fichiers qu'affiche la commande `ls` (sauf avec l'option '-a').

1.4 Les commandes du shell

Pour avoir une liste de commandes, tapez `ls /bin` ou `ls /usr/bin` dans votre shell. Pour avoir l'aide d'une commande, tapez `man` suivi du nom de la commande. Pour avoir une courte description d'une com-

mande, tapez `whatis` suivi du nom de la commande. Pour rechercher une commande dont l'aide pourrait contenir un mot particulier, tapez `apropos` suivi de ce mot.

1.4.1 Syntaxe des commandes

Tout ce qui se trouve entre le *prompt* et le *retour-chariot* est la *ligne de commande*. Plusieurs commandes peuvent être entrées sur la ligne de commande en les séparant par des points-virgules.

Ex : `date ; users`

Une commande peut avoir deux types d'arguments : les *options* et les *paramètres*. Les options modulent l'action de la commande. Elles sont préfixées par le signe '-' et elles doivent apparaître avant les paramètres. Les paramètres sont des données d'entrée de ces commandes (en général, des noms de fichiers).

Ex : `ls -l -F fichier1 fichier2`

1.4.2 Redirection et tube

Les commandes UNIX utilisent l'*entrée standard* (par défaut : le clavier) et la *sortie standard* (par défaut : l'écran) pour lire les données d'entrée et écrire les données résultats.

On peut rediriger la sortie standard vers un fichier en utilisant le caractère '>'. Par exemple, `ls > fichier` écrira le contenu du répertoire courant dans `fichier` et pas à l'écran. Si ce fichier n'existe pas, il sera créé. S'il existe, il sera remplacé. Si on utilise '>>' à la place de '>' alors les données produites par la commande seront ajoutées à la fin du fichier déjà existant.

On peut rediriger l'entrée standard à partir d'un fichier en utilisant le caractère '<'. Ex : `mail truc@domaine.fr < fichier` envoie un message à `truc@domaine.fr` dont le contenu est dans `fichier`.

On peut rediriger la sortie standard d'une commande vers l'entrée standard d'une deuxième commande grâce à un tube [pipe] en utilisant le caractère '|'. Ex : `sort | lpr` trie un fichier entré au clavier, passe le fichier résultat à la commande `lpr`, qui l'imprime.

On peut rediriger la sortie standard vers plusieurs fichiers grâce à un té [tee] en utilisant la commande `tee`. Ex : `cat | tee fichier1 | sort` lit un fichier au clavier, le dirige dans `fichier` et vers la commande `sort` qui le trie et l'affiche à l'écran.

1.4.3 Processus en tâche de fond

Quand on lance une commande, le shell rend la main une fois que la tâche est terminée. Pour reprendre la main immédiatement après avoir lancé la commande, ajouter le caractère '&' après la commande. Le processus continuera de s'exécuter en tache de fond. La liste des processus en cours d'exécution est donnée par la commande `ps`. On peut stopper prématurément un processus grâce à la commande `kill`.

1.5 Les principales commandes

Commandes de fichiers et répertoires

Commande	Action	Options importantes
ls	liste des fichiers d'un répertoire	a : avec fichiers cachés; l : infos complètes
cp	copie de fichiers	i : interactif ; R : copie récursive
mv	déplacements de fichiers	i : interactif
rm	destruction de fichier	i : interactif; R : récursif
touch	mise à jour date du fichier	
chmod	change permissions du fichier	
cd	change le répertoire courant	
pwd	affiche le chemin du répertoire courant	
mkdir	crée un répertoire	
rmdir	détruit un répertoire vide	

Commandes de manuel

Commande	Action	Options importantes
man	manuel	f : idem whatis; k : idem apropos
whatis	manuel résumé	
apropos	manuel se référant à des mots	

Commandes d'affichage et de traitements de fichiers texte

Commande	Action	Options importantes
cat	rien (concaténation)	n : numérote les lignes
colrm	détruit des colonnes	
crypt	encode/décode le fichier	
cut	extrait des colonnes	c : positions; f : champs ; d : séparateur
head	affiche le début	c : nombre de caractères ; n : nombre de lignes
(e)grep	extrait des lignes	e : motif ; f : fichier ; v : action inverse
more	affiche le contenu, par écran	
paste	combine des colonnes	d : délimiteurs
rev	inverse l'ordre des caractères	
sort	trie les lignes	r : inverse ordre ; m : joint des fichiers triés
spell	vérifie l'orthographe anglaise	
tail	affiche la fin	
tee	démultiplie la sortie standard	a : ajouter après le fichier sortie
tr	traduit des caractères	d : supprime les caractères
uniq	supprime les lignes identiques	u/d : lignes uniques/dupliquées
wc	compte lignes, mots et caractères	l : lignes ; w : mots ; c : caractères

Commandes d'impression

Commande	Action	Options importantes
lpr	imprime un fichier	P : nom imprimante ; # : nombre copies
lpq	liste les fichiers en cours d'impression	P : nom imprimante
lprm	enlève un fichier de la queue d'impression	P : nom imprimante

Commandes liées à Internet

Commande	Action	Options importantes
mail	lit ou envoie du courrier	f : fichier de stockage, s : subject
ftp	transfert de fichiers entre machines	
host	affiche adresse d'une machine	
hostname	affiche nom de sa machine	
ping	vérifie l'accessibilité d'une machine	
telnet	connexion à distance	l : nom utilisateur

Autres commandes diverses

Commande	Action	Options importantes
emacs	éditeur de texte	
vi	éditeur de texte (moins évolué)	
history	historique des commandes	
alias	création de l'alias d'une commande	
unalias	annule l'alias	
cal	calendrier	y : toute l'année courante
date	lire ou configurer la date	
du	stats sur l'espace disque utilisé	a : tous les fichiers
login	nouveau login	f : garde l'authentification courante
exit	termine un shell	
passwd	changement de mot de passe	
tty	affiche configuration du terminal	
compress	compresse un fichier	f : fichier sortie
uncompress	décompresse un fichier	
tar	archivage de fichiers	'xvf'/'cvf' : extrait/crée l'archive
finger	infos sur compte	
w	liste des processus en cours	
ps	liste de ses processus	a : autres utilisateurs aussi
who	liste des personnes connectées	
whoami	qui je suis	
echo	envoie un texte à la sortie standard	

2 Programmation du shell

2.1 les variables de l'environnement

On peut définir des variables dans l'environnement du shell. Syntaxe[3] : `variable=valeur` (pas d'espace autour du '='). En préfixant son nom par '$', on obtient sa substitution par sa valeur pendant l'évaluation de la ligne de commande. On peut affecter à une variable le résultat d'une commande en entourant cette dernière par des ‘.

```
$ MAVAR=abcd
$ echo $MAVAR
abcd
$ echo MAVAR
MAVAR
$ MAVAR=date
$ $MAVAR
mar oct 31 17:07:55 CET 2000
$ echo $MAVAR
date
$ MAVAR=‘date‘
$ echo $MAVAR
mar oct 31 17:07:58 CET 2000
```

La portée des variables est locale : elle n'existe qu'à l'intérieur du shell où elles ont été définies. On peut étendre leur portée aux shells fils du shell où elles ont été créées grâce à la commande `export VAR`. La modification de la variable dans un shell fils laisse inchangée celle du shell père.

3. Cette syntaxte dépend du shell. Celle que nous indiquons est valable pour bash.

les variables prédéfinies du shell

Les commandes du shell utilisent des variables déjà définies lors du lancement du shell.

HOME : le répertoire d'accueil, PATH : les répertoires (séparés par ':') où chercher les commandes, MAIL : le fichier recevant les mails en attente, PS1 : l'invite, LOGNAME : nom de login, RANDOM : valeur aléatoire, etc.

On obtient la liste des variables prédéfinies grâce à la commande `env`, la liste de toutes les variables grace à `set`.

Substitution conditionnelle

L'utilisation d'accolade autour du nom de la variable permet des substitutions conditionnelles.

- Sans condition : `${var}`

```
$ echo $PATH
/bin:/usr/bin
$ PATH=${PATH}:/usr/local/bin
$ echo $PATH
/bin:/usr/bin:/usr/local/bin
```

- Avec valeur par défaut : `${var-val}` substitue `$var` ou `val` si `var` n'existe pas.
- Avec affectation de valeur par défaut : `${var=val}` substitue `$var` ou `val` si `var` n'existe pas et alors affecte `val` à `var`.
- Avec valeur de remplacement : `${var+val}` substitue `val` si `var` existe.
- Que la valeur par défaut : `${var?val}` substitue `val` si `var` n'existe pas.

Si on veut disposer de ses propres variables définies au moment du login, alors on en place la liste dans le fichier système `.profile`.

Les variables spéciales

En plus de variables d'environnement prédéfinies, le shell utilise des variables spéciales contenant diverses informations courantes.

$0 contient le nom de la commande courante, $1, son premier paramètre, $2, son deuxième paramètre, ... jusqu'à $9. Les paramètres suivant sont accessibles grace à la commande `shift`.

$? contient le code de retour d'une commande. Cette valeur est 0 si la commande s'est bien déroulée.

2.2 Les fonctions

Une fonction est une suite de commandes regroupées sous un nom. On la définit ainsi :

```
$ nom_fonction()
{ commande1;
  commande2;
  ...
}
```

Exemple : la fonction suivante prend en paramètre deux fichiers dont le deuxième sera le double du premier.

```
$ double()
{ cp $1 fichiertmp;
  cat $1 >> fichiertmp;
  mv -f fichiertmp $2
}
```

La commande `expr`

La commande `expr` permet d'évaluer des expressions arithmétiques ou lexicales.
Ex : `$ expr 3 + 5` renvoie 8.

On peut utiliser l'addition (+), la soustraction (-), la multiplication (*), la division (/), le modulo (%). On peut comparer des entiers ou des mots avec des opérateurs (<, >, <=, >=, ==, !=). L'opérateur | renvoie la valeur de son premier argument s'il n'est pas nul, sinon il renvoie la valeur de son deuxième argument. l'opérateur & renvoie zéro si son premier argument est nul, sinon il renvoie son deuxième argument.

2.3 Les scripts

Un script est un fichier texte exécutable. Il contient un ensemble de commandes à exécuter par le shell. Une fois le texte édité et sauvegardé, il faut le rendre exécutable grâce à la commande `chmod`.

Remarque : Un script est comme une fonction dont on aurait le texte dans un fichier.

Exemple :

```
$ cat > inv
echo $2 $1
^d
$ chmod +x inv
$ inv un deux
deux un
```

Les paramètres du script

$0 fait référence au nom du script,$1, ... à $9 aux paramètres. S'il y a plus de 10 paramètres, on peut utiliser la commande `shift`. Celle-ci décale vers la gauche le contenu des variables contenant les paramètres : $2 devient $1, $3 devient $2, etc.

Les autres variables spécifiques sont : $? qui contient le code de retour du script, $# qui contient le nombre de paramètres de la commande, $* qui contient la liste des paramètres, $@ qui contient la liste des paramètres sauf ceux se trouvant entre guillemets.

2.4 Les instructions

Instructions de lecture/écriture

L'instruction `echo` permet de transmettre des données en sortie standard. L'instruction `read` lit les données sur l'entrée standard.

Opérateurs d'exécution conditionnelle

L'opérateur && (syntaxe : `cmd1 && cmd2`) n'exécute `cmd2` que si `cmd1` a fourni un résultat. Ex : `ls rep && cd rep` fait se déplacer dans `rep` si ce dernier existe.

L'opérateur || n'exécute `cmd2` que si `cmd1` n'a pas fourni de résultat. Ex : `ls rep || echo 'repertoire inexistant'`.

La commande `test`

La `test` permet de faire des tests sur des nombres, des chaînes de caractères ou des types de fichiers.

Pour les tests sur des nombres, la syntaxe est `test <entier> <option> <entier>`. `<option>` peut être `-eq, -ne, -gt, -ge` ou `-le`.

Pour les fichiers, la syntaxe est `test <option> <fichier>`. `<option>` peut être `-s` (le fichier existe est n'est pas vide), `-d` (le fichier est un répertoire), `-f` (le fichier est ordinaire), `-w` (le fichier a le droit d'écriture) ou `-r` (le fichier a le droit en lecture).

Pour les chaînes, la syntaxe est `test <chaine> = <chaine>` (égalité),`test <chaine> != <chaine>` (différence), `test -z <chaine>` (chaîne de longueur nulle) ou `test -n <chaine>` (chaîne de longueur non nulle).

On peut combiner des tests avec les opérateurs `-a` (ET logique) et `-o` (OU logique). Ex : `$ test -z C1 -a -n C2` (teste si C1 est nulle et C2 est non nulle).

L'instruction `if`

Syntaxe :

```
if <commande>
then
   <commande>
[elif <commande>
   <commande>
 elif ...
   ...
 else
   <commande>]
fi
```

Cette instruction exécute <`commande`> et exécute ce qui suit le `then` si $? est égal à 0, sinon elle exécute ce qui suit le `else`. `elif` est une contraction de `else if`.

Les instructions `while` et `until`

Syntaxe :

```
while <commande>; do <commande> done
until <commande>; do <commande> done
```

La commande `while` (resp. `until`) répète la commande située après de `do` tant que (resp. jusqu'à ce que) le résultat de la commande située après le `while` est (resp. soit) vraie.

Ex : `$ A=5 ; while ‘expr $A < 10 >/dev/null‘ ; do  echo $A ; A=‘expr 1 + $A‘ ; done`

L'instruction `for`

Syntaxe :

```
for <var> in <ensemble_de_mots> ; do <commande> ; done
```

La commande `for` exécute la commande située après le `do` pour chaque valeur respective de la variable $< var >$ dans $< ensemble\_de\_mots >$.

Ex : `for i in 1 2 3 4; do expr $i + $i ; done` affiche 1, 4, 9 et 16.

`for i in *a* ; do ls $i ; done` effectue un 'ls' sur tous les fichiers et répertoires dont le nom contient la lettre 'a' dans le répertoire courant.

3 Compiler du C sous Unix

3.1 Programmation modulaire

La plupart des problèmes informatiques sont trop compliqués pour être résolus globalement et il est très utile de les décomposer en sous-problèmes de résolution plus aisée. En conséquence, il est utile de pouvoir écrire ses programmes sous forme de modules résolvant une partie de la tâche globale à effectuer. Lorsqu'il s'agit d'un gros projet informatique, plusieurs personnes peuvent participer et écrire chacun de son côté un ou plusieurs modules qu'il faudra rassembler plus tard.

Comme module courant en C, on peut citer les bibliothèques [library] de fonctions prédéfinies : stdio, stdlib, math, etc. La spécificité des bibliothèques de fonctions est de regrouper un ensemble de fonctions suffisamment générales pour être utilisables dans plusieurs projets informatiques différents. On peut évidemment créer ses propres bibliothèques.

3.2 Utilisation des modules en C

Afin qu'un module puisse utiliser les fonctions d'un autre module, il faut que leur existence soit connue du module qui les utilise. C'est pourquoi un module est généralement composé de deux fichiers : un *en-tête* [header] qui contient les déclarations de fonctions et de constantes qui pourront être utilisées par d'autres modules et un *corps* [body] qui contient le code des fonctions. Par convention, les fichiers en-tête ont pour extension *.h et les fichiers contenant le corps du module ont pour extension *.c.

Lorsqu'un module A (composé des fichiers A.h et A.c) a besoin d'utiliser une ou plusieurs fonctions d'un module B, il doit inclure le fichier B.h grâce à la directive d'inclusion de fichier du préprocesseur `#include<B.h>` au début du fichier A.c. De la même manière, chaque corps de module doit inclure son propre fichier en-tête.

3.3 Compiler du C

Il existe plusieurs compilateurs C et C++ : cc, gcc, g++, etc. Pour générer un fichier exécutable à partir d'un ensemble de fichiers-texte (codes source) écrits en C, il y a deux étapes : la génération du code objet (binaire) de chaque module et l'édition de liens, qui unissent les fichiers objets en un seul programme (un exécutable).

A partir de la racine du répertoire contenant tous les fichiers du projet, on créera les sous-répertoires `src` pour mettre les codes sources, `obj` pour mettre les codes objets et `include` pour mettre les fichiers en-tête.

Pour créer directement un exécutable à partir d'un code source qui tient en un seul module : `gcc prg.c`. Le code exécutable généré a pour nom `a.out`. Pour spécifier un autre nom, utiliser l'option `-o`. Ex : `gcc src/prg.c -o app`.

Pour créer un code objet (non exécutable) à partir d'un code source, utiliser l'option `-c`. Ex : `gcc -c src/prg.c -o obj/prg.o`.

L'option `-I`*dir* ajoute le répertoire *dir* à la liste des répertoires à explorer pour les fichiers en-tête à inclure. Ex : `gcc -c src/prg.c -I$$HOME/Projet/include/ -o obj/prg.o`.

Pour créer une librairie à partir de plusieurs fichiers objets, utiliser les commandes d'archivage `ar` puis `ranlib`. Exemple courant d'utilisation : `ar r obj/libmabib.a obj/module1.o obj/module2.o ; ranlib obj/libmain.a` permet de créer la bibliothèque *mabib* (dont le nom du fichier est `libmabib.a`) à partir de `module1.o` et `module2.o`.

Pour créer un exécutable à partir d'un code source (contenant généralement le programme principal) et d'une librairie (contenant les codes objets des autres modules), on pourra faire ceci : `gcc src/main.c -I$$HOME/Projet/include/ -L$$HOME/Projet/obj/ -o app -lmabib`. L'option `-l`*library* utilise la bibliothèque *library* dont le nom est lib*library*.a. L'option `-L`*dir* ajoute le répertoire *dir* à la liste des répertoires à explorer pour `-l`.

3.4 Le fichier `makefile`

La compilation d'un module peut prendre plusieurs secondes. Un programme peut nécessiter plusieurs centaines de modules. La compilation de la totalité des modules peut donc prendre plusieurs dizaines de minutes. Lorsqu'on modifie le code source d'un module, on veut que seuls ce module et les modules qui dépendent de lui soient recompilés puis que l'édition de liens soit faite avec les autres modules déjà compilés afin de générer le programme final. Le fichier `makefile` (à créer soi-même à la racine du répertoire contenant tous les fichiers relatifs au programme) est un fichier-texte qui contient les relations de dépendances entre les modules et permet de compiler uniquement les modules nécessaires à la création du programme complet.

Le fichier `makefile` contient une suite de triplets (fichier père, liste de fichiers fils, commande à effectuer). La syntaxe d'un triplet est :
fichier-père: *fichier-fils1 fichier-fils2 [...] retour-chariot tabulation commande*

Exemple de `makefile` pour un projet regroupant 2 modules, soit 5 fichiers au départ (module1.h, module1.c, module2.h, module2.c et main.c):

```
app: src/main.c obj/libmabib.a include/module1.h include/module2.h
	gcc src/main.c \
		-I$$HOME/Projet/include/ \
		-L$$HOME/Projet/obj/ \
		-o app \
		-lmabib

obj/libmabib.a: obj/module1.o obj/module2.o
	ar r obj/libmabib.a obj/module1.o obj/module2.o ; \
	ranlib obj/libmabib.a

obj/module1.o: src/module1.c include/module1.h
	gcc -c src/module1.c \
		-I$$HOME/Projet/include/ \
		-o obj/module1.o

obj/module2.o: src/module2.c include/module2.h
	gcc -c src/module2.c \
		-I$$HOME/Projet/include/ \
		-o obj/module2.o
```

Lorsque qu'on lance la commande `make` dans le répertoire où se trouve le fichier `makefile`, ce dernier est lu. Si un ou plusieurs fichiers-fils ont été modifiés depuis la dernière commande `make` alors les commandes correspondantes sont exécutées. Leur exécution modifiant des fichiers-père qui peuvent être aussi des fichiers-fils, l'exécution des commandes se propage des fils vers les pères. Pour l'exemple du fichier `makefile` ci-dessus, si `module2.c` a été modifié, l'exécution de `make` permet la recompilation de `module2.o` qui entraîne la recréation de `libmabib.a` qui entraîne la recompilation de `app`.

Bibliographie

''UNIX, Guide de l'étudiant'', Harley Hahn, Collection Science Informatique, Dunod.
''La machine UNIX'', Louis Léon, Cépaduès.
''Filtres et utilitaires UNIX'', Richard Stoeckel, Armand Colin.
''Linux in a Nutshell'', Jessica P. Hekman, Ed. O'Reilly.
''Systèmes d'exploitation'', Andrew Tanenbaum, Interéditions.