Construction de programmes

Make et autres outils

Philippe Collet
2007-2008

D'après un cours de L. Pierre

Plan

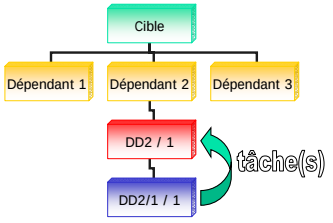
- Motivations et Principes
- Historique
- Make : fonctionnement explicite
- Make : variables, directives, etc.
- Make : fonctionnement implicite
- Outillage associé

Motivations

- Gérer les dépendances entre fichiers à
  - construire / compiler / relier / exécuter
- Automatiser la construction / l'exécution en
  - Reconstruisant ce qui est nécessaire
  - Et uniquement suffisant
  - ☞ Rapidité de construction / processus « à la main »
  - ☞ Complétude et optimisation de la construction
- Exemples :
  - .o/.c/.h, bibliothèque, archive, installation,
  - génération de documentation, .tex/.dvi/.ps, ...

Principes

- Un graphe de dépendances va être construit à partir :
  - De cibles (fichier ou simple nom)
  - De leur dépendances
  - Des tâches à réaliser
- Qu'est qu'un dépendant ?
  - Un élément qui nécessite l'exécution des tâches s'il n'est pas à jour par rapport à la cible
- Comment détermine-t-on qu'on est à jour ?
  - Par les dates de dernière modification



Principes (suite)

- L'ensemble des informations est centralisé dans un fichier, interprété par la commande make
  - Nom par défaut : makefile ou Makefile
  - Constitué de règles
- Une règle a la forme :

```
cible : liste des dépendants
<tab>   tâches (en sh)
```

- Des raccourcis et des variables d'environnement permettent de décrire plus facilement les règles, les cibles, les dépendances, les tâches

Historique

- Make, quasiment aussi vieux que les compilateurs C sous Unix
- GNU make
  - Développement de Linux => assimilation a GNU make
  - Compatibilité ascendante, mais
  - Quelques particularités donc
  - ATTENTION aux make/makefiles sur plates-formes Unix propriétaires
- Cours et TP basés sur GNU make

Mon premier makefile

```
all: helloworld

helloworld: helloworld.o
<tab> gcc –o helloworld helloworld.o

helloworld.o : helloworld.c helloworld.h
<tab> gcc -c helloworld.c
```


Syntaxe d'une règle

- Schéma général

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2 prerequis3 ... prerequisD
   commande1 ; commande2 ; commande3 ; ... ; commandeC
```

- ou bien

```
cible1 cible2 ... cibleN : prerequis1 prerequis2 ... prerequisD ; commande1 ;
   commande2 ; ... ; commandeC
```

- ou encore

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2 prerequis3 ... prerequisD
   commande1
   commande2
   commande3 ...
   commandeC
```

- Tous les cas intermédiaires sont possibles...
- Les cibles doivent être séparées par des espaces (blancs ou tabulations), ainsi que les dépendants.
- Deux commandes consécutives sur une même ligne doivent être séparées par des points-virgules « ; ».

Interprétation d'une règle

- Pour décider d'exécuter les tâches d'une règle, il faut vérifier au préalable si un des dépendants nécessite une mise en cohérence par rapport à la cible.
- Les commandes qui composent la tâches sont alors exécutées séquentiellement
- Si une commande provoque à une erreur, alors la cible n'est pas rendue cohérente et TOUT le processus s'arrête.

- Une cible est incohérente lorsque :
  - Un de ses dépendants est incohérent,
  - la cible n'existe pas,
  - sa date de dernière modification est antérieure à celles de ses dépendants

Fonctionnement général

- Deux phases
  - Construction
    - Lecture du makefile et de tous les makefile inclus
    - Stockage des couples macros/valeurs
    - Stockage et ordonnancement des règles implicites et explicites
    - Construction du graphe de dépendances
  - Utilisation
    - Détermination des cibles à reconstruire
    - Invocation des règles dans l'ordre

Un exemple plus conséquent

```
// convertir.h
// taux de conversion
#define TAUX 6.55957
// type de conversion
#define EURO 0
#define FRANC 1
// prototypes des routines de convertir.c
void franc_euro(double franc);
void euro_franc(double euro) ;
```

```
/* convertir.c routines de conversion francs-euro, euro-francs */
#include <stdio.h>
#include "convertir.h"
/* x francs = x/6.55957 euros */
void franc_euro(double franc) {
   printf("\t%.4lf francs = %.4lf euros\n", franc,
          franc / TAUX);
}
/* x euros = x*6.55957 francs */
void euro_franc(double euro) {
   printf("\t%.4lf euros = %.4lf francs\n", euro,
          euro * TAUX);
}
```


Exemple (2)

```
// fichier outils.h
#define CONTINUER 1
#define SORTIR 0
// prototypes de outils.c
void Message(char *message, short etat);
void LireDouble(char *phrase, double *nombre);
void LireShort(char *phrase, short *nombre);
```

```
// fichier outils.c
#include <stdio.h>
#include "outils.h"
void Message(char *message, short etat) {
  printf("%s", message);
  if(etat == SORTIR) {
     printf("\n ... interruption du programme \n");
     exit(0);
  }
}
void LireDouble(char *phrase, double *nombre) {
  Message(phrase, CONTINUER);
  if(scanf("%lf", nombre) == 0)
    Message("Vous n'avez pas entre un float",
            SORTIR);
}
void LireShort(char *phrase, short *nombre) {
  Message(phrase, CONTINUER);
  if(scanf("%hd", nombre) == 0)
    Message("Vous n'avez pas entre un entier",
            SORTIR);
}
```

Exemple (3)

```c
// fichier convert2sens.c
#include <stdio.h>
#include "convertir.h"
#include "outils.h"
int main (int argc, char **argv) {
  double val;
  short action;
  short type;
  action = CONTINUER; // initialisation
  do {
    LireDouble("entrez un nombre : ",  &val);
    if(val == 0)
      action = SORTIR;
    else {
      LireShort( "convertir en euros (0) / francs (1) ? ", &type);
      switch (type) {
       case EURO  : franc_euro(val); break;
       case FRANC : euro_franc(val); break;
       default : Message("Euros (0) /Francs (1) ?", CONTINUER);
       }
    }
  } while (action == CONTINUER);
  return 2;
}
```

```
gcc convertir.c outils.c convert2sens.c -o convertir
```


Makefile (explicite) associé

```
all: convertir

convertir: convertir.o outils.o convert2sens.o
        gcc convertir.o outils.o convert2sens.o -o convertir

convert2sens.o : convert2sens.c
        gcc -c convert2sens.c

convertir.o : convertir.c
        gcc -c convertir.c

outils.o : outils.c
        gcc -c outils.c
```


Fonctionnement

- Make nomDeCible
- Make –f monMakefile
- Make –i
  - Ignorer globalement toutes les erreurs qui peuvent survenir
- Make –n
  - Montrer ce que l'on ferait, mais ne pas le faire...
- Make
  - Makefile ou makefile dans le répertoire courant
  - Reconstruction de la première cible trouvée

Utilisation du makefile

```
matrix% ls
convert2sens.c  convertir.c  convertir.h  makefile  outils.c  outils.h
matrix% make convertir
gcc -c convertir.c
gcc -c outils.c
gcc -c convert2sens.c
gcc convertir.o outils.o convert2sens.o -o convertir
matrix% ls
convert2sens.c  convertir*   convertir.h  makefile  outils.h
convert2sens.o  convertir.c  convertir.o  outils.c  outils.o
```

```
matrix% make convertir
`convertir' is up to date.
```

```
matrix% touch *.c
matrix% make convertir
gcc -c convertir.c
gcc -c outils.c
gcc -c convert2sens.c
gcc convertir.o outils.o convert2sens.o -o convertir
```

Et les .h ?

- On modifie `convertir.h` pour passer la constante TAUX à 6.55
  - l'exécution de *make* ne provoquerait aucun changement !
- Prendre en compte les dépendances par rapport à des fichiers .h :
  - Ajout de cibles (*possible, car il n'y a pas de commande*)

```
convertir.o convert2sens.o : convertir.h
outils.o convert2sens.o : outils.h
```

  - *Attention, s'il y a plusieurs occurrences de la même cible avec des commandes, il faut utiliser « :: » à la place de « : »*
  - Ou modification des cibles déjà en place (préférable !)

```
convert2sens.o : convert2sens.c convertir.h outils.h
  gcc -c convert2sens.c
outils.o : outils.c outils.h
  gcc -c outils.c
convertir.o : convertir.c convertir.h
  gcc -c convertir.c
```


Autres éléments de syntaxe

- Passage à la ligne
  - Les cibles et les prérequis doivent se trouver sur une même ligne.
  - Sauf en utilisant le caractère « \ »
    - Lorsque make lit un fichier makefile, il ignore toutes les occurrences du caractère « \ » suivi d'un caractère « \n » (newline).
  - Donc la règle :

```
cible1 cible2 \
cible3 ... cibleN : \
prerequis1\
prerequis2 prerequis3 \
... prerequisD ; commande1 ; commande2 commande3 ; ... ; commandeB ; \
commandeC
```

  - est l'équivalent de la règle :

```
cible1 cible2 cible3 ... cibleN : prerequis1 prerequis2 prerequis3 ...
   prerequisD ; commande1 ; commande2 commande3 ; ... ; commandeB; commandeC
```


Autres éléments de syntaxe (2)

- Commentaire
  - Toute suite de caractères après un « # ».
- Le caractère TAB
  - Une ligne commence par TAB, ce qui suit est interprété comme une commande shell.
  - Une série d'espace n'a pas le même effet
  - Un TAB « esthétique » en début de ligne peut jouer des tours…
- Commande invisible
  - Le caractère @ placé devant une commande *empêche l'affichage de l'intitulé* de la commande à la console.

```
clean :
    @-rm *.o
    @ls
```

```
matrix% make clean
convert2sens.c    convertir.c    convertir.h     makefile
outils.c    outils.h
```


Autres éléments de syntaxe (3)

- Comment exécuter des commandes dans le même shell ?
  - A priori, chaque commande de la cible est exécutée dans un shell séparé.
  - Pour que des commandes soient exécutées dans le même shell il faut les placer sur la même ligne avec le séparateur de commandes « ; »

```
clean :
        cd $(HOME)/bin
        rm convertir
```

```
matrix% make clean
cd /u/profs/collet/bin
rm convertir
rm: convertir: No such file or directory
make: *** [clean] Error 1
```

```
clean :
        cd $(HOME)/bin ; rm convertir
```

```
matrix% make clean
cd /u/profs/collet/bin ; rm convertir
```

Les cibles « Phony »

- Cibles « étiquette »
  - Exemples
    - all
    - install
    - clean
    - veryclean
- « convertir » est-il « phony » ?
  - Non, car un fichier a ce nom est créé
- Directive :
  - .PHONY: all, clean
- Voir plus loin pour les directives...

Variables

- Il y a plusieurs types de variables pour make :
  - les variables passées par commande 1
  - les variables définies dans le makefile 2
  - les variables d'environnement 3
  - les variables automatiques
  - *Certaines variables ont des valeurs par défaut dans make* 4
- **Les variables passées par commande**
  - make TOTO=objet.c
  - dans le fichier makefile, toutes les occurrences de $(TOTO) ou de ${TOTO} seront remplacées par objet.c.
  - De manière générale, dans toute la suite, pour référencer les variables, on pourra utiliser les parenthèses et les accolades de façon interchangeable.

Variables d'environnement

- Toutes les variables du shell lançant make sont accessibles
  - echo $(VARIABLE) renvoie la valeur de la variable d'environnement VARIABLE.
  - Par exemple les variables USER, HOSTTYPE ou OSTYPE sont des variables du shell.
- Si la commande make est lancée par un shell donné, (bash, zsh, sh, csh, ksh, tcsh ou autre) toutes les variables d'environnement de ce shell peuvent être utilisées dans le makefile.
- Toutes les occurrences de $(USER), $(OSTYPE), etc., dans le makefile, seront remplacées par la valeur qu'elles avaient dans le shell.
- La seule exception est la variable SHELL qui, par défaut, vaut toujours /bin/sh, mais qui peut être modifiée dans le makefile.
  - Permet d'assurer qu'un makefile, appelé par une même ligne commande, ne puisse pas avoir deux comportements différents selon que celui qui lance la commande préfère ksh à bash ou à tout autre shell.

Variables définies dans le makefile

`var=valeur1 valeur2 valeur3 ... valeurN`

- **substitution** (en anglais *expansion*) de la variable var:
  - toutes les occurrences de `$(var)` ou de `$var` seront remplacées par `valeur1 valeur2 ... valeurN`.
- **Attention, la substitution est récursive !**

```
TITI = $(TUTU)
TOTO = $(TITI)
TUTU = coucou
all:
    echo $(TOTO)
TUTU=NonNon!!!
```

```
matrix% make
echo NonNon!!!  NonNon!!!
```

Variables de makefile (suite)

- Que se passe-t-il si on écrit
  `PATH = $(PATH) /usr/sbin`
  - Théoriquement, ça boucle...
- Dans les autres versions de make, autres que GNU make, seules les substitutions récursives sont possibles
- GNU make permet cependant de faire de la substitution simple :
  `PATH := $(PATH) /usr/sbin`
  - **ATTENTION, ceci n'est donc pas portable**

Manipulation de variables

- $variable correspond à une variable du makefile...
- Le $$ peut être utilisé pour *passer une expression de variable* commençant par $ au shell.

```
affiche:
	for fic in *.c; do ls $$fic; done
```

```
matrix% make affiche
for fic in *.c; do ls $fic; done
convert2sens.c  convertir.c  outils.c
```

- Pour pouvoir ajouter quelque chose à une variable (à substitution récursive ou simple), on utilise l'opérateur binaire « += »
  ```
  variable = valeur
  variable += davantage
  ```
- est équivalent à
  ```
  temp = valeur
  variable = $(temp) davantage
  ```
- sauf qu'aucune variable temp n'est définie

Application à notre makefile

```
CC = gcc
CFLAGS = -c
OBJS = convertir.o outils.o convert2sens.o
```

```
convertir: $(OBJS)
	$(CC) $(OBJS) -o convertir
convert2sens.o: convert2sens.c convertir.h outils.h
	$(CC) $(CFLAGS) convert2sens.c
. . .
```


Jokers et règles à motif

- * et ?, [a-z] et autres peuvent être utilisés dans
  - Les commandes shell (comme d'habitude)
  - Les cibles !
  - Les prérequis !
- On peut ainsi créer des règles à motif...

```
*.o : *.c
   gcc -c *.c
```

Variables automatiques

- Ces variables sont définies et remises à jour par make.
- Elles ne peuvent être utilisées que dans une règle et désignent une des cibles, les prérequis, un sous-ensemble de prérequis, etc.
- $@ : remplacé par le nom du fichier cible.
  - S'il y a plusieurs noms dans la cible, alors $@ est remplacé par celui des noms dans la cible qui a déclenché l'application de la règle.

```
convertir: $(OBJS)
	$(CC) $(OBJS) -o $@
```

- S* : préfixe de la cible
- $^ : remplacé par la liste des prérequis, chaque nom étant séparé par une espace.
  - Si un certain prérequis a été répété plusieurs fois, alors il n'apparaît qu'une seule fois dans $^.
  - Si on veut que l'ordre des éléments et le nombre des répétitions soient préservés, il faudra utiliser $+ au lieu de $^.

Variables automatiques (suite)

- $< désigne le premier prérequis.
  - nom du fichier qui a causé l'action associée à la cible (celui qui serait utilisé par la règle implicite)

```
convert2sens.o : convert2sens.c
	$(CC) -c $<
```

- $? remplacé par la liste des prérequis qui sont plus jeunes qu'au moins une des cibles.
  - En fait $? désigne la liste des prérequis qui sont responsable de l'application de la commande de la règle.

```
print : *.c
	@echo $? : fichiers C
```

```
matrix% make print
convert2sens.c convertir.c outils.c : fichiers C
```

- Il existe d'autres variables automatiques ($%, $+, etc.)

Make récursifs

- Considérons des sous-répertoires (à n'importe quel niveau) subrep_1, subrep_2, subrep_3, ..., subrep_n.
- Chacun de ces sous-répertoires contient un makefile
- Pour utiliser les makefiles dans chaque répertoire
  - on peut aller dans chacun de ces répertoires (cd subrep_i)
  - taper les commandes make adéquats

```
make arg_1 arg_2 arg_3 ... arg_N
```

- L'appel récursif est possible

```
make -C subrep_i arg_1 arg_2 arg_3 ... arg_N
```

- Lors de l'appel récursif de make, la commande « make » utilisée est celle définie *par défaut* sur le système...

```
${MAKE} -C subrep_i arg_1 arg_2 arg_3 ... arg_N
```


Directives non conditionnelles

- Inclusion
  - include <nom_fichiers>
  - make, en lisant cette directive, cesse temporairement de lire le makefile auquel appartient cette directive et lit, dans l'ordre, le ou les fichiers dont le nom appartient à la liste <nom_fichiers>.
  - Cette liste d'arguments <nom_fichiers> peut contenir des motifs, des noms de variables, etc.

```
include toto titi *.mk $(MKS)
```

Directives non conditionnelles (2)

- Définition de variable (avec des sauts de ligne)

```
define <nom_variable>
...
...
endef
```

  - La directive define doit être suivie du nom de la variable (et de rien d'autre) écrit sur la même ligne.
  - La valeur de la variable est constituée de toutes les lignes qui se trouvent entre la ligne qui contient la directive define et la ligne qui contient endef.

```
define sauvegarde
tar -cvf ../backup.tar *
gzip ../backup.tar
echo 'termine'
endef
```

  - Même utilisation que les variables

Directives non conditionnelles (3)

- Les directives `export` et `unexport`
  - héritage (ou non) de la valeur de toutes les variables qui étaient définies dans le make global.

`export var`

  - la variable var sera définie dans tous les sous-makes qui pourront être lancés à partir du makefile auquel appartient la directive.

`export`

  - Action sur toutes les variables

`export`

`unexport var`

  - toutes les variables soient héritées sauf la variable var

Directives conditionnelles

- ifeq et ifneq

```
ifeq (argument1,argument2)
texte1
else
texte2
endif
```

  - Toutes les variables dans les arguments argument1 et argument2 sont substituées.
  - Si les deux arguments sont égaux, la conditionnelle est équivalente à texte1 (texte2 n'est pas pris en compte), et s'ils sont différents, la conditionnelle est équivalente à texte2.
- ifdef et ifndef
  - Teste si une variable est définie ou pas.

```
ifdef DEBUG
  CFLAGS += -g
  endif
```

- Tout ça à ne pas confondre avec if...

Traitement de chaînes

- Syntaxe générale

```
$(fonction argument1, argument2, ..., argumentN)
```

- `$(shell cmd)`
  - Cette fonction peut créer des chaînes de caractères résultant de l'exécution d'une commande shell.

`$(shell ps)` a pour valeur

```
PID TTY TIME CMD 1234 pts/0 00:00:01 bash 10999 pts/0 00:00:03 emacs
   11219 pts/0 00:00:00 make 11220 pts/0 00:00:00 ps
```

  - Si une commande shell produit un résultat sur plusieurs lignes, la fonction shell fournit un résultat sur une seule ligne.
- `$(subst c1,c2,chaine)`
  - La fonction subst permet de remplacer une chaîne de caractères par une autre dans une liste de noms.

`$(subst ien,ienG, Tiens bien il ne reste plus rien)` vaut

```
TienGs bienG, il ne reste plus rienG
```

Traitement de chaînes (2)

- `$(patsubst c1,c2,chaine)`
  - Remplacement avec motif
  - Le premier argument de patsubst peut contenir un caractère %.
  - Par exemple « const.% » représente tous les mots qui commencent par « const. » et qui se terminent par une chaîne quelconque X.
  - Si le deuxième élément comprend aussi un caractère « % », ce « % » sera remplacé par la chaîne X.

`$(patsubst %ien, %ienG, Tiens bien il ne reste plus rien)` donne toujours

```
TienGs bienG il ne reste plus rienG
```

Par contre

`$(patsubst` **`Version.%, %_Version`**`, Version.01 Version.02 Version.03 ... Version.N)` vaut

```
01_Version 02_Version 03_Version ... N_Version
```

`$(patsubst argument1, argument2, $(variable))` est équivalent à `$(variable:argument1=argument2)`

Traitement de chaînes (3)

- Plein d'autres fonctions...
  - `strip` : pour débarasser une chaine des espaces superflus en début et fin.
  - `filter` et `filter-out` : extraire d'une liste les mots correspondants à un motif donné
  - `sort` : trie dans l'ordre lexicographique et élimine les doublons
  - `dir` et `notdir` : extraire les répertoires ou les fichiers d'une liste
  - `foreach` : parcours de liste avec application de substitution à chaque fois différente
  - ...

Règles implicites

```
convert2sens.o : convert2sens.c
        $(CC) -c $<
```

- Est une règle intéressante, mais on pourrait l'appliquer à tout fichier .c pour le lier au .o correspondant
- Il existe en fait une telle règle, dite **implicite**

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

```
CC = gcc
SRCS = outils.c convertir.c convert2sens.c
OBJS = $(SRCS:.c=.o)
all: convertir
convertir: $(OBJS)
       $(CC) $(OBJS) -o $@
```


Règles et directives de suffixe

- Il est possible de changer la liste des suffixes utilisables :

```
.SUFFIXES :            # vide la liste des suffixes
.SUFFIXES :  .o .c     # nouvelle valeur
```

ou alors

```
.SUFFIXES : .x $(SUFFIXES)  # valeur ajoutee a SUFFIXES
```

- Si on décidait de nommer les fichiers objets .ob, cela donnerait

```
SRCS = convertir.c outils.c convert2sens.c
OBJS = $(SRCS:.c=.ob)
all: convertir
convertir: $(OBJS)
    $(CC) $(OBJS) -o $@
.SUFFIXES : .ob .c
.c.ob :
    $(CC) -c  $< -o $*.ob
```

- Les règles de suffixe *ne peuvent pas avoir de dépendances explicites*.
- Par exemple, on ne pourrait pas avoir la règle

```
.c.o: moninclude.h
     $(CC) -c $<
```

Génération de makefile

- Utilisation des options du préprocesseur
  - Lorsqu'on compile avec l'option -M on obtient une règle utilisable par make qui décrit les dépendances du fichier.

```
matrix% gcc -M convert2sens.c
convert2sens.o: convert2sens.c /usr/include/stdio.h \
 /usr/include/sys/feature_tests.h /usr/include/sys/va_list.h \
 convertir.h outils.h
```

  - L'option –MM donne les dépendances hormis les dépendances *système*

```
matrix% gcc -MM convert2sens.c
convert2sens.o: convert2sens.c convertir.h outils.h
```


Génération de makefile (2)

- makedepend
  - permet d'obtenir les dépendances pour les fichiers .h inclus par le fichier source

```
matrix% makedepend -f monmake convert2sens.c convertir.c outils.c
```

```
convertir : $(OBJS)
    $(CC) $(OBJS) -o $@
clean :
    -rm *.o
%.o : %.c
    $(CC) -c -o $@ $<
```

```
# DO NOT DELETE
convert2sens.o: /usr/include/stdio.h /usr/include/sys/feature_tests.h
convert2sens.o: /usr/include/sys/va_list.h convertir.h outils.h
convertir.o: /usr/include/stdio.h /usr/include/sys/feature_tests.h
convertir.o: /usr/include/sys/va_list.h convertir.h
outils.o: /usr/include/stdio.h /usr/include/sys/feature_tests.h
outils.o: /usr/include/sys/va_list.h outils.h
```

  - Si makedepend trouve une ligne "# DO NOT DELETE" dans le fichier, il efface tout ce qui suit jusqu'à la fin du fichier et place les dépendances après cette ligne. S'il ne la trouve pas, il l'ajoute ainsi que les dépendances à la fin du fichier

Génération de makefile (3)

- 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
  - adaptation automatique au système d'exploitation, aux différents compilateurs, aux bibliothèques (recherche de la présence des fonctions) et utilitaires présents sur le système (yacc/bison, cc/gcc, lex/flex, awk/gawk,...)
  - fichiers Makefile.in (template), configure.in (paths)
  - très employé dans les distributions GNU
- genmake
  - Outil graphique (non GNU)
  - construction automatique d'un Makefile pour une seule cible à partir des sources C et des include dans le répertoire courant du projet

Sources

- Cours de C avancé et Environnement de Programmation (L. Pierre, 2003-2004)
- Manuels Unix/Linux