Développement sous Unix

Fabrice Bellet
Laboratoire CREATIS

Fabrice.Bellet@creatis.insa-lyon.fr

Le développement d'applications dans un environnement Unix nécessite des connaissances de l'environnement de développement et des différents outils à disposition du développeur pour simplifier son cycle de développement. Cette connaissance passe par la maîtrise de l'utilisation du compilateur (dans le cas qui nous interesse, il s'agit du compilateur C), ses options, les principes de création de librairies de programmes. L'automatisation du développement nécessite l'utilisation des Makefiles pour automatiser les étapes de génération de code. Enfin, une connaissance des systèmes Unix et de ses interpréteurs de commandes s'avère indispensable.

Cette documentation ne se veut pas exhaustive mais fournit plutot quelques points de départ et quelques outils pour la personne curieuse qui souhaiterais approfondir certains points par la suite.

Les points abordés ici seront davantage ce que je considère comme des méthodes au mieux, ou des astuces au pire qui permettent d'exploiter un environnement Unix. Il s'agit du fruit de mon expérience, avec tout ce que cela peut avoir de partial et de subjectif.

1. Introduction

Ce document a pour objectif de vous familiariser avec le cycle de développement d'applications en environnement Unix. Il est composé de trois parties:

- La première partie concerne la mise en oeuvre des outils de compilation.
- La deuxième partie abordera le fonctionnement des *Makefiles*.
- La troisième partie présentera Unix dans ses commandes de base, et ses shells de commande.

2. Compilation

La *compilation* d'un programme est un ensemble de traitements successifs sur un fichier source dont l'étape ultime est de générer un fichier exécutable par la machine. Les traitements successivement effectués sur le fichier source sont les suivants:

Pré-processing

Le *pré-processing* consiste à substituer toutes les macros présentes dans le code par leur valeur: #ifdef, #define, #include, etc. Voici un petit exemple de code sur lequel le préprocesseur va fonctionner:

```
/*
 * Exemple d'utilisation de macros
 */

#include <stdio.h>
#define BORNE_MAX       100
#define MAX(a,b)        ((a)>(b)?(a):(b))

int table [BORNE_MAX];
```

```
int main() {
  int i;

  for (i=0; i < BORNE_MAX; i++)
          table [i]=i;
#ifdef DEBUG
  printf ("DEBUG : table[10]=%d\n", table[10]);
#endif
  return 0;
}
```

On peut affecter la valeur de macros a la compilation avec l'option **-DMON_SYMBOL** ou **-DMON_SYMBOL=valeur**. On utilisera généralement des noms de symboles en majuscules pour les différencier des simples variables.

Les personnes de nature curieuse pourront examiner le code obtenu à l'issue du pré-processing avec l'option **-E** du compilateur:

```
% gcc -E fichier.c > fichier.E
```

Compilation

L'étape de compilation par elle-même. Elle travaille sur le fichier résultat du préprocessing, et produit un fichier texte contenu du code en langage d'assemblage spécifique à la machine sur laquelle vous compilez.

Assemblage

L'étape d'*assemblage* prend le fichier précédent, et génère du code machine. Le fichier produit est appelé *fichier objet*, et se reconnait en général par son extension `.o`.

Edition de liens

L'*édition de liens* prend un ensemble de fichiers objets pour produire un programme exécutable.

Heureusement, dans la grande majorité des cas, on n'a pas à se préoccuper de tous ces fichiers intermédiaires, car ils sont gérés de façon transparente par le compilateur, à l'exception des fichiers objets. Ces fichiers et ces formats temporaires ne sont pas visibles à l'utilisateur sauf cas spécial. Mais il n'est pas inutile de savoir qu'ils existent.

Examinons un petit programme d'exemple:

```
/*
 * main.c
 */

#include <stdio.h>
#include <file1.h>

int main()
{
   printf ("%d\n",file1_proc (10));
   return 0;
}


/*
 * file1.h
 */

extern int file1_proc (int i);


/*
 * file1.c
 */

int file1_proc (int i)
{
   return i+1;
}
```

```
% gcc -c file1.c
% gcc -I. -c main.c
% gcc -o exemple main.o file1.o
% exemple
11
% gcc -v -c file1.c
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/specs
gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/cpp -lang-c -v -undef -D__GNUC__=2 -D__GNUC_MINOR__
GNU CPP version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release) (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/i386-redhat-linux/include
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include
 /usr/include
End of search list.
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/cc1 /tmp/cc2jRiec.i -quiet -dumpbase file1.c -versi
GNU C version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release) (i386-redhat-linux) compiled by GNU C
 as -V -Qy -o file1.o /tmp/cc2F6SSd.s
GNU assembler version 2.9.1 (i386-redhat-linux), using BFD version 2.9.1.0.24
% gcc -v -o exemple main.o file1.o
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/specs
gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.
```

Le compilateur C est appelé par la commande **cc** ou **gcc**. Son fonctionnement peut être modifié par une impressionnante floppée d'options sur la ligne de commande. Pour avoir une aide exhaustive sur un compilateur donné, il est recommandé de se reporter au manuel en ligne (accessible par la commande **`man gcc`**). Les options essentielles sont les suivantes:

`-c`

Cette option indique au compilateur de s'arrêter après la génération du fichier objet (de ne pas faire l'édition de liens). Elle permet la compilation séparée.

`-Idirectory`

Cette option indique le chemin de recherche pour trouver les fichiers inclus dans les macros #include <fichier.h>. Un certain nombre de répertoires sont prédéfinis et n'ont pas besoin d'être spécifiés. L'ordre des répertoires de recherche est important. Une variante existe: les macros #include "fichier.h" se limitent au répertoire courant.

`-o executable`

Cette option indique le nom de l'exécutable qui sera généré à l'édition de liens. Par défaut, si on ne précise rien, l'exécutable est appelé `a.out` en référence au format dans lequel est écrit ce fichier.

Note: Le format *a.out* a disparu de la circulation au profit du format *ELF*, plus portable et plus extensible.

Toute liste de fichiers sera considérée comme des fichiers devant être traités dans le processus de compilation. Le traitement à leur appliquer dépendra de leur suffixe. Par exemple **`gcc file1.c file2.o`** va compiler `file1.c`, générer temporairement `file1.o`, lier ensuite `file1.o` et `file2.o` dans un exécutable nommé `a.out`.

D'autres options de compilation peuvent s'avérer utiles:

`-g`

Cette option rajoute des informations supplémentaires dans chacun des fichiers produits pour faciliter la tache de debuggage. Cela permet en utilisant un debugger (par exemple **gdb**) d'avoir accès aux noms de variables utilisées dans le source, de savoir pour chaque instruction machine exécutée à quel fichier source, et à quelle ligne de code cela correspond. La taille des fichiers ainsi générés augmente sensiblement, on évitera donc de laisser trainer ces options en cycle de production ou lorsque l'on fournit un produit fini.

-On

Cette option indique si l'on souhaite produire un code optimisé. La valeur n indique le niveau d'optimisation souhaité, de zéro (pas d'optimisation, jusqu'à 6 ou plus, ceci dépend du compilateur). Chaque niveau correspond à la sélection d'un groupe d'options d'optimisation indépendantes. On peut choisir une optimisation specifique en précisant l'option **-foptimisation_specifique** ou au contraire pour interdire une optimisation spécifique par **-fno_optimisation_specifique**. Chaque option d'optimisation est décrite abondamment dans les pages du manuel en ligne du compilateur. Il est possible de mixer les options d'optimisation et de débugage (**-g -O2** par exemple), mais cela n'est pas recommendé car cela rend le code délicat à débugger. L'optimiseur se réserve le droit de supprimer des variables inutiles, de précalculer des expressions, de changer l'ordre d'exécution des instructions. Toutes ces modifications perturbent l'utilisateur qui s'attend à avoir une exécution séquentielle des lignes de son code source sous le debugger.

-DSYMBOL[=valeur]

Cette option permet de définir des constantes ayant une valeur pour le préprocesseur. L'exemple précédent montre que le compilateur ajoute lui-même un grand nombre de ces constantes, permettant de caractériser le système sur lequel la compilation s'exécute (compilation conditionnelle).

-Ldirectory -llibrary

Ces options sont utilisées pour l'édition de liens. Elles permettent d'inclure des fichiers objets supplémentaires à ceux que l'on a compilé. Ces fichiers objets sont regroupés dans un fichier unique, nommé *librairie*. Une librairie est donc un ensemble de fichiers `.o`. Elle peut être liée statiquement ou dynamiquement à l'exécutable.

Dans le premier cas, le contenu de la librairie est ajouté dans le fichier exécutable une fois pour toute. Cela rend l'exécutable *self-contained*. Tout le code dont il a besoin pour fonctionner est inclus dans le fichier executable. Le désavantage de cette méthode est que la mise à jour de la librairie ne pourra se faire que par une recompilation de notre fichier source qui est à la charge de l'utilisateur.

Dans le second cas, l'exécutable ne contient qu'un lien vers cette librairie. Il est donc de plus petite taille. Il n'y a cependant pas de miracle, le code de la librairie devra pourtant être chargé au moment de l'exécution (on parle de *runtime*). Pour cela, une variable d'environnement LD_LIBRARY_PATH contient une liste de répertoire dans laquelle le loader dynamique recherchera la librairie souhaitée au lancement de l'exécutable. L'intérêt est d'avoir des exécutables qui ne dupliquent pas le code d'une même librairie, et de pouvoir mettre à jour une librairie sans avoir besoin de recompiler les programmes qui l'utilisent (pour peu que cette librairie continue d'utiliser la même *API*, *Application Programming Interface*. Des numéros de versions majeurs et mineurs sont utilisés pour gérer la compatibilité entre les versions d'une librarie).

L'option **-Ldirectory** indique le directory où le linker doit rechercher la librairie (au moment de l'édition des liens), et l'option **-llibrary** precise le nom de cette librairie. Selon les architectures, le fichier aura comme nom `liblibrary.so` pour une librairie dynamique ou `liblibrary.a` pour une librairie statique.

Un petit exemple sur la création de librairies:

```
% gcc -c fichier1.c fichier2.c fichier3.c
% ar cr libessai.a fichier1.o fichier2.o fichier3.o
% ranlib libessai.a
% gcc -c main.c
% gcc -static -o executable main.o -L. -lessai
% gcc -static -o executable main.o libessai.a
```

Les deux dernières commandes sont équivalentes. Cet exemple illustre la création d'une librairie statique et son utilisation pour générer `executable`.

```
% gcc -c -fPIC fichier1.c fichier2.c fichier3.c
% gcc -shared -o libessai.so fichier1.o fichier2.o fichier3.o
% gcc -c main.c
% gcc -o executable main.o -L. -lessai
```

Cet exemple illustre la création d'une librairie dynamique et son utilisation pour générer `executable`.

Selon les cas, les systèmes Unix utilisés, et le type de compilateur utilisé, la syntaxe peut varier. Ces exemples s'appuient sur les outils GNU (http://www.gnu.org). Pour avoir les syntaxes à s'appliquant à un système praticulier, il faut se référer aux documentations, en particulier les pages de manuel de **cc**, et **ld** pour le linker dynamique.

3. Makefiles

3.1. Introduction

La partie précédente a montré aux naifs qui en doutaient encore qu'une gestion de projet nécessitait de mettre en oeuvre la compilation séparée des composantes de l'application. Le compilateur est invoqué pour chaque fichier source et une fois supplémentaire pour l'édition de liens, dans le cas où l'on ne génère pas de librairies intermédiaires.

Lorsqu'un seul des fichiers sources est modifié, il n'est pas forcément utile de recompiler *tous* les fichiers sources pour regénérer l'application. On gagnera du temps à ne recompiler que les fichiers sources sur lesquels la modification du code effectuée a un réel impact. C'est à ce moment-là que le *Makefile* entre en jeu.

Le principe du Makefile est de construire un graphe de dépendance des fichiers constituant l'application, surchargé pour chaque fichier par la commande qui permet de le regénérer à partir des fichiers dont il dépend.

3.2. Un exemple simple

Le `Makefile` le plus simple à comprendre est celui où toutes les dépendances apparaissent explicitement. Exemple:

Example 1. Un exemple simple de Makefile

```
all : executable
executable : file1.o file2.o
        gcc -o executable file1.o file2.o
file1.o : file1.c file1.h
        gcc -c file1.c
file2.o : file2.c file1.h file2.h
        gcc -c file2.c
clean :
        rm file1.o file2.o executable core
```

On lance l'interpréteur de Makefile par la commande **make**. Par défaut, la première dépendance rencontrée tentera d'être résolue : **`all`**. Cette *target* nécessite de remettre à jour toutes les conditions situées à droite des `:`. **make** passe donc à l'unique target suivante, `executable`

D'après la deuxième ligne du `Makefile`, on voit que `executable` dépend des deux fichiers objets `file1.o` et `file2.o`. Récursivement, les fichiers dont dépendent `file1.o` et `file2.o` sont recherchés. Si un des fichiers dépendant est plus récent qu'un des fichiers cible au cours de cette recherche, le fichier cible doit être regénéré, et la règle de compilation associée est exécutée.

```
% make clean
rm file1.o file2.o executable core
rm: cannot remove 'core': No such file or directory
% make
gcc -c file1.c
gcc -c file2.c
gcc -o executable file1.o file2.o
% touch file2.h
% make
gcc -c file2.c
gcc -o executable file1.o file2.o
% touch file2.o
% make
gcc -o executable file1.o file2.o
% touch file1.h
% make
gcc -c file1.c
gcc -c file2.c
gcc -o executable file1.o file2.o
```

3.3. Règles génériques

Des règles génériques évitent d'avoir à écrire ligne de Makefile pour chaque fichier source.

Example 2. Un autre exemple de Makefile

```
CC = gcc
CFLAGS = -O2 -c
OBJS = file1.o file2.o

all : executable
.c.o :
        $(CC) $(CFLAGS) $<
executable : $(OBJS)
        $(CC) -o $@ $(OBJS)
```

La règle **`.c.o:`** s'applique pour tous les fichiers `.o` et ont comme unique dépendance le fichier `.c` correspondant. $< est alors substitué par le nom du fichier sur lequel cette règle est instanciée.

On englobe pratiquement le premier exemple de `Makefile`, à la seule différence que l'on ne tient pas compte des dépendances sur les fichiers `.h`. La commande **makedepend** permet d'automatiser la génération de ces dépendances supplémentaires.

Example 3. Les dépendances supplémentaires

```
% cat Makefile
CC = gcc
CFLAGS = -O2 -c
OBJS = file1.o file2.o
SRCS = file1.c file2.c

all : executable
.c.o :
        $(CC) $(CFLAGS) $<
executable : $(OBJS)
        $(CC) -o $@ $(OBJS)CC = gcc
depend :
        makedepend -I. $(SRC)
% cat file1.c
#include "file1.h"
main() {}
% cat file2.c
#include "file2.h"
#include "file1.h"
% makedepend -I. file1.c file2.c
% cat Makefile
CC = gcc
CFLAGS = -O2 -c
OBJS = file1.o file2.o
SRCS = file1.c file2.c

all : executable
.c.o :
        $(CC) $(CFLAGS) $<
executable : $(OBJS)
        $(CC) -o $@ $(OBJS)CC = gcc
depend :
        makedepend -I. $(SRC)

# DO NOT DELETE

file1.o: ./file1.h
file2.o: ./file2.h ./file1.h
```

3.4. Plus loin

Les makefiles peuvent servir à automatiser toutes les tâches nécessitant d'appliquer plusieurs traitements consécutifs à un document. A titre d'exemple, ce document a été généré en utilisant un `Makefile`. Le fichier source est un document SGML, il est traité par la commande **jade** pour produire un document HTML et un document TeX. Le document TeX est ensuite compilé à son tour. Le makefile utilisé dans ce cas est le suivant:

```
all : development.ps index.html

index.html : development.sgml
        jade -ihtml -t sgml -d formation.dsl\#html development.sgml
development.tex : development.sgml
        jade -t tex -d formation.dsl\#print development.sgml
development.dvi : development.tex
        jadetex $<
development.ps : development.dvi
        dvips -o $@ $<
```

4. Unix

4.1. Introduction

Unix est une marque déposée par la société AT&T d'un systeme d'exploitation d'ordinateur.

> **Note:** A définir.

L'objectif d'un systeme d'exploitation est de faciliter l'interface entre l'utilisateur et le matériel. Par des abstractions, un OS (Operating System, abbrévation de système d'exploitation) masque à l'utilisateur les ressources matérielles et les lui présente sous une forme unifiée. Ceci est valable pour le/les processeurs, la mémoire, les périphériques, les disques, etc. Ces abstractions se font par le biais d'une API qui standardise les interactions entre l'utilisateur et le matériel.

Unix est un système multi-utilisateur, multi-taches, préemptif (termes a définir). Il repose donc à ce titre sur des principes de partage des ressources entre les différents utilisateurs.

4.2. Gestion de fichiers

On distingue plusieurs types de fichiers, parmi lesquels les deux plus importants : les fichiers *réguliers* et les *répertoires*. Le séparateur de noms de répertoires est le symbole **/**. Peu ou pas de contraintes existent sur les noms de fichiers. Leur longueur est généralement limitée à 512 ou 1024 caractères selon les systèmes, ce qui laisse le temps de voir venir.

4.2.1. Gestion des répertoires

mkdir: création
rmdir: destruction
pwd: répertoire courant
cd: changement du répertoire courant

4.2.2. Attributs des répertoires et des fichiers

ls

Un utilisateur est identifié sous Unix pour un nom et un numéro d'utilisateur, ainsi que par son appartenance à un ou plusieurs groupes. Les fichiers Unix disposent donc de droits d'acces à ces deux niveau. Les droits se décomposent classiquement en droits de lecture (r), écriture (w) et exécution (x) et sont définis respectivement pour l'utilisateur, pour les membres du groupe et pour les autres utilisateurs (chaine **`rwxr-xr-x`** dans le listing).

Les droits sur répertoires ont des significations particulieres, indiquant la possibilite de lire le contenu d'un répertoire (r), de créer de nouveaux fichiers dans un répertoire (w), et la possibilité de faire un **cd** dans ce répertoire (x).

On change les droits des fichiers avec la commande **chmod** qui a la syntaxe suivante: **`chmod [ugoa][+-=][rwx] files`**`....`

Example 4. Les droits des fichiers

```
% cd /
% ls -l
total 108
drwxr-xr-x   2 root     root         4096 Oct 24 16:02 bin/
drwxr-xr-x   2 root     root         4096 Oct 24 17:02 boot/
drwxr-xr-x   6 root     root        36864 Oct 26 15:26 dev/
drwxr-xr-x  41 root     root         4096 Oct 26 22:30 etc/
-rw-r--r--   1 root     root            2 Sep 24 23:53 fonts.dir
drwxr-xr-x   6 root     root         4096 Feb  6  1996 home/
drwxr-xr-x   4 root     root         4096 Oct 24 16:00 lib/
drwxr-xr-x   2 root     root        16384 Sep 24 23:27 lost+found/
drwxr-xr-x   2 root     root         4096 Sep 20 17:13 misc/
drwxr-xr-x   4 root     root         4096 Oct  9  1998 mnt/
drwxr-xr-x   2 root     root         4096 Aug 23 18:03 opt/
dr-xr-xr-x  70 root     root            0 Oct 26 15:25 proc/
drwxr-x---   9 root     root         4096 Oct 27 11:06 root/
drwxr-xr-x   3 root     root         4096 Oct 24 17:17 sbin/
drwxrwxrwt  13 root     root         4096 Oct 27 14:45 tmp/
drwxr-xr-x  21 root     root         4096 Aug  2  1998 usr/
drwxr-xr-x  21 root     root         4096 Oct 24 16:02 var/
```

Les manipulations possibles de fichiers sont:

mv: renommer, déplacer
cp: copier
rm: effacer
cat ou **more**: visualiser

4.3. Shell

L'interaction avec l'utilisateur passe par un interpréteur de commandes (aussi appelé *shell*). Il s'agit d'un processus Unix, un programme, qui interprete séquentiellement chaque commande entrée au clavier. Chaque commande est un processus Unix. (

> **Note:** parler des filiations entre les processus

). On termine l'exécution d'un shell de commande par la commande **exit**.

Les processus lancés par le shell, peuvent fonctionner en mode synchrone (par défaut) ou asynchrone. Exemple:

Example 5. Exemples de processus en tâche de fond

```
% sleep 2; echo coucou; sleep 3; echo coucou
% sleep 2 & echo coucou & sleep 3 & echo coucou
```

La commande **ps** permet de lister tous les processus.

```
% ps axu
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2  1104  464 ?        S    00:39   0:03 init
root         2  1.1  0.0     0    0 ?        SW   00:39  10:14 [kapmd]
root         3  0.0  0.0     0    0 ?        SW   00:39   0:00 [kswapd]
root         4  0.0  0.0     0    0 ?        SW   00:39   0:00 [kflushd]
```

```
bin        157  0.0  0.2  1200  396 ?        S    00:40   0:00 portmap
root       173  0.0  0.2  1100  496 ?        S    00:40   0:01 /usr/sbin/apmd -p
root       226  0.0  0.2  1160  524 ?        S    00:40   0:00 syslogd -m 0
root       237  0.0  0.4  1416  768 ?        S    00:40   0:00 klogd
daemon     253  0.0  0.2  1132  484 ?        S    00:40   0:00 /usr/sbin/atd
root       269  0.0  0.3  1308  600 ?        S    00:40   0:00 crond
bellet   28007  0.1  2.3  7372 4412 ?        S    09:48   0:31 gvim development.sgml
bellet   28013  0.0  2.0  7180 3976 ?        S    09:56   0:01 gvim corba.sgml
bellet   28418  0.0  0.7  2440 1520 ?        S    14:35   0:00 rxvt
bellet   28419  0.0  0.6  2252 1304 pts/3    S    14:35   0:00 -csh
bellet   28488  0.0  1.2  3608 2364 pts/3    S    14:40   0:00 gv development.ps
bellet   28497  0.0  1.6  5136 3104 pts/3    S    14:44   0:01 gs -dNOPLATFONTS -sDEVICE=x11alpha -d
bellet   28559  0.0  0.7  2444 1520 pts/3    S    15:14   0:00 rxvt
bellet   28560  0.0  0.5  2112 1144 pts/4    S    15:14   0:00 -csh
bellet   28627  0.0  0.4  2548  912 pts/0    R    15:42   0:00 ps axu
```

```
% ps axgf
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:03 init
    2 ?        SW    10:14 [kapmd]
    3 ?        SW     0:00 [kswapd]
    4 ?        SW     0:00 [kflushd]
  157 ?        S      0:00 portmap
  173 ?        S      0:01 /usr/sbin/apmd -p 10 -w 5 -W -u
27981 ?        S      0:07 /usr/lib/netscape/netscape-communicator -irix-session-management corba-gr
27996 ?        S      0:00  \_ (dns helper)
28007 ?        S      0:31 gvim development.sgml
28013 ?        S      0:01 gvim corba.sgml
28418 ?        S      0:00 rxvt
28419 pts/3    S      0:00  \_ -csh
28488 pts/3    S      0:00      \_ gv development.ps
28497 pts/3    S      0:01      |   \_ gs -dNOPLATFONTS -sDEVICE=x11alpha -dNOPAUSE -dQUIET -dSAFER
28559 pts/3    S      0:00      \_ rxvt
28560 pts/4    S      0:00          \_ -csh
```

On termine un processus avec la commande `kill`. On peut envoyer à un processus plusieurs types de signaux (`kill -l` pour avoir la liste). Le plus connu est le signal 9 (KILL) pour tuer un processus lancé en tâche de fond. Les shells modernes permettent aussi d'utiliser des numéros de jobs locaux au shell, ce qui évite de faire la commande **ps**. Exemple:

```
% yes > /dev/null &
[1] 28646
% yes > /dev/null &
[2] 28647
% ps axg | grep yes
28646 pts/0    R      0:43 yes
28647 pts/0    R      0:14 yes
% kill -STOP %1
[1]  + Suspended (signal)            yes > /dev/null
% ps axg | grep yes
28646 pts/0    T      1:01 yes
28647 pts/0    R      0:48 yes
% kill -CONT %1
[1]    yes > /dev/null &
% kill -TERM %1
[1]    Terminated                    yes > /dev/null
71.700u 0.220s 2:44.82 43.6%    0+0k 0+0io 88pf+0w
% fg %2
yes > /dev/null
CTRL-Z

Suspended
% jobs
[2]  + Suspended                     yes > /dev/null
% bg %2
```

```
[2]    yes > /dev/null &
% jobs
[2]    Running                       yes > /dev/null
% kill -KILL %2
[2]    Killed                        yes > /dev/null
160.400u 0.400s 3:47.71 70.6%   0+0k 0+0io 88pf+0w
```

La commande **top** permet d'avoir la liste remise régulièrement à jour des processus sur la machine, classé par utilisation du temps CPU. Elle est particulièrement pour contrôler que des processus oubliés ne continuent pas à tourner inutilement sur la machine. Exemple:

```
  3:57pm  up 15:18,  4 users,  load average: 0.17, 0.54, 0.32
60 processes: 59 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  0.0% user,  0.9% system,  0.0% nice, 99.0% idle
Mem:  191208K av, 178356K used,  12852K free,      0K shrd,   2052K buff
Swap: 385452K av,      0K used, 385452K free                 38648K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
28654 bellet    16   0  1064 1064   856 R       0  0.5  0.5   0:00 top
25525 bellet     3   0  1588 1588  1112 S       0  0.3  0.8   0:02 rxvt
    1 root       0   0   464  464   392 S       0  0.0  0.2   0:03 init
    2 root       0   0     0    0     0 SW      0  0.0  0.0  10:14 kapmd
    3 root       0   0     0    0     0 SW      0  0.0  0.0   0:00 kswapd
    4 root       0   0     0    0     0 SW      0  0.0  0.0   0:00 kflushd
  157 bin        0   0   396  396   316 S       0  0.0  0.2   0:00 portmap
  173 root       0   0   496  496   424 S       0  0.0  0.2   0:01 apmd
  226 root       0   0   524  524   424 S       0  0.0  0.2   0:00 syslogd
  237 root       0   0   768  768   392 S       0  0.0  0.4   0:00 klogd
  253 daemon     0   0   484  484   404 S       0  0.0  0.2   0:00 atd
  269 root       0   0   600  600   504 S       0  0.0  0.3   0:00 crond
  285 root       0   0   512  512   432 S       0  0.0  0.2   0:00 inetd
```

4.4. Utilisation avancée du C-Shell

Plusieurs shells (interpréteurs de commandes) existent sous Unix. Ils possedent chacun leur syntaxe et leurs *built-ins* spécifiques, mais ils se composent en deux grandes catégories:

Compatibles Bourne Shell

> Les shells compatibles avec le Bourne Shell (`/bin/sh`): **ksh**, **bash**, **zsh**, etc. Ils se distinguent entre eux par leur facilité d'édition des commandes, par la gestion ou pas d'un historique, par leur license d'utilisation entre autres. Ils sont généralement dédiés à l'écriture de scripts système.

Compatibles C-Shell

> Les shells de la famille des C-shells: csh, tcsh. Leur syntaxe est proche de celle du langage C, ce qui leur a donné ce nom.

L'utilisateur a la possibilité de changer son shell courant avec la commande **chsh** ou **passwd**. Pour qu'un shell soit autorisé, il doit être référencé dans le fichier `/etc/shells` pour des raisons de sécurité.

Les shells ne sont pas une simple boucle d'attente des commande de l'utilisateur. Ils sont des langages de commandes à part entiere, avec leurs variables, leurs structures de controle.

4.4.1. Les variables

Les variables ne sont pas typées.

```
% set X=123
% set Y=toto
% echo $X $Y
123 toto
```

Les variables peuvent être visibles uniquement dans le shell courant (**set**) ou bien dans tous les processus fils (**setenv**). **setenv** sert pour affecter les variables d'environnement, par exemple DISPLAY.

```
% set V1=1
% setenv V2 2
% csh
% echo $V1
V1: Undefined variable.
% echo $V2
2
% exit
% echo $V1 $V2
1 2
```

4.4.2. Les structures de controle

Les boucles, les itérations, les tests sont au rendez-vous:

```
% foreach i ( 1 2 3 4 )
? echo $i
? end
1
2
3
4

% foreach file ( * )
? echo $file
? end
fichier.c
fichier.o
Makefile

% if ( $i < 4 ) then
?    echo oui
? else
?    echo non
? fi
oui

% while ( $condition )
? instruction
? instruction
? end
```

4.4.3. Substitution de variables

```
% set variable = 123
% echo $variable
123
% set v2 = un nom avec un espace
% echo $v2
un
% set v2 = "variable vaut $variable"
% echo $v2
variable vaut 123
% set v2 = 'variable vaut $variable'
% echo $v2
variable vaut $variable
% set v2 = variables\ vaut\ $variable
% echo $v2
variable vaut 123
```

```
% set d  = "la date est `date`"
% echo $d
la date est Wed Oct 27 17:16:05 MET DST 1999
```

Les simples quotes interdisent l'évaluation du contenu à la différence des doubles quotes.

L'anti-slash \ interdit l'interprétation du caractère qui le suit.

```
% set v2 = xxxx\'yyyy
% echo $v2
% xxxx'yyyy
```

Il est possible de compléter des noms de fichiers.

```
% set v2 = fichier.*
% echo $v2
fichier.c fichier.o
```

4.4.4. Les flots standards d'entrée/sortie

Tout processus Unix lit les données en entrée sur un fichier `stdin`, écrit les données en sortie sur un fichier `stdout`, et écrit les messages d'erreur sur un troisième fichier `stderr`. Clavier et écran ne sont, pour le système Unix, que des fichiers particuliers, en lecture seule pour le premier et en écriture seule pour le deuxième. Ces trois flots sont représentés en csh par les symboles suivants:

Table 1. Les flots d'entrées/sorties

Flots	Symbole
stdin	**`<`**
stdout	**`>`**
stdout (append)	**`>>`**
stdout + stderr	**`>&`**
stdout + stderr (append)	**`>>&`**

On connecte les flux d'entrée/sorties de deux processus avec un "pipe" Unix:

```
% cat development.sgml | wc
    571    3147   25034
% make |& more
```

On utilise un sous-shell pour séparer `stdin` et `stdout`:

```
% ( make > make.log ) >& make.err
```

4.4.5. Intérêt des sous-shells

L'utilisation des *back-quotes* permet de laner des commandes dans des sous-shells et de substituer la commande par son résultat dans le shell courant.

```
% foreach f (`cat file_list`)
? rm -f $file
? end
```

4.4.6. Substitutions des noms de fichiers

Il existe un certain nombre de regles de substitution, qui, même si une certaine ressemblance existe, ne s'apparentent pas à des expression régulières.

- `*`: plusieurs caractères différents de ".".
- `?`: un seul caractère different de ".".
- `[abc]`: une alternative de caractères.
- `{chaine1,chaine2}`: une alternative de chaines de caractères.

Exemples:

- `*.*`: tous les fichiers contenant un point.
- `.??*`: tous les fichiers commencant par un point et avec au moins deux autres caractères. Cela permet de lister les fichiers cachés (commencant par `.`) et d'exclure `.` et `..` (le répertoire courant et le répertoire parent respectivement).
- `~/src/[12]/Makefile`: permet de matcher les deux fichiers `/home/bellet/src/1/Makefile` et `/home/bellet/src/1/Makefile`. En effet, le symbole ~ est un raccourci pour la variable d'environnement HOME, indiquant le home-directory de l'utilisateur.
- `*.[hc]`: permet de matcher tous les fichiers `.c` et `.h` du répertoire courant.

4.4.7. La galere des expressions numériques

Il faut utiliser la commande externe **expr**.

```
% set a = 3
% set b = `expr \( 1 + 3 \) \* $a`
% echo $b
12
```

Faire un compteur est possible:

```
% set a = 0
% while ( $a < 10 )
?   echo Traitement du fichier numéro $a
?   set a = `expr $a + 1`
? end
```

4.4.8. Les conditions

L'écriture des conditions est tres similaire à la syntaxe du langage C. Encore une chance, ce shell a été écrit pour ca!

```
% if ( $x == "chaine" ) ...
% if ( $x <= 10 && $y == 2 ) ...
% if ( $x != 1 || "$2" == `hostname` ) ...
% if ( -r fichier1 && -x fichier2 && -d fichier3 ) ...
```

Un autre moyen d'écrire des tests conditionnels consiste à utiliser une caractéristique des processus Unix, qui retournent un entier lorsqu'ils se terminent indiquant si la commande s'est bien exécutée ou pas. Si la commande s'est bien terminée, la variable d'environnement $status vaut 0, sinon tout autre valeur correspond à un code d'erreur. Voir les pages de manuel de chaque commande Unix pour connaitre la signification des codes d'erreur retournés. Cet entier est la valeur renvoyée par les programmes en C écrit scrupuleusement:

```
int main() {
        ... plein de choses;
        procedure_truc();
        ... encore plein d'autres choses;
        /*
         * fin normale du programme
```

```
         */
        return 0;
}
void procedure_truc () {
        ...
        if (erreur) {
           fprintf (stderr,"erreur irrecuperable. bye.\n");
           /*
            * fin anormale
            */
           exit (1);
        }
}
```

Ainsi, les primitives de comparaison && et || utilisent la valeur de $status pour fonctionner. On peut écrire des conditions également et de façon très compactes ainsi:

```
% echo azerty | grep -q aze && echo found
found
% echo azerty | grep -q qwe && echo found
% echo azerty | grep -q qwe || echo not found
not found
```

4.4.9. Gestions de flot d'entrées/sortie et grep

La commande **grep** lit des données depuis un fichier ou depuis l'entrée standard, et affiche sur sa sortie standard les lignes contenant la chaine de caractères passée en paramètre sur la ligne de commande.

Les instructions `grep` et `egrep` permettent de faire des recherches d'expression régulières dans le flux `stdin`. Par exemple on peut effectuer une recherche de deux chaines de caractères dans un fichier en utilisant une expression régulière dans la commande **egrep**. On notera qu'il est nécessaire de mettre l'expression régulière entre quotes, afin que le **|** ne soit pas interprété comme le pipe Unix dans cet exemple:

```
% egrep 'NNTPSERVER|PATH' ~/.cshrc
setenv  NNTPSERVER      demo2.univ-lyon1.fr
setenv  LD_LIBRARY_PATH ${HOME}/creatis/lib:/usr/local/lib
setenv  CLASSPATH       /usr/lib/netscape/java/classes/java40.jar
```

On peut faire un ET avec un pipe Unix cette fois-ci. Dans cet exemple, on va matcher par exemple la ligne qui contient les deux chaines passées a chacun des **grep**:

```
% grep define *.[hc] | grep MA_CONSTANTE
fichier.c:#define MA_CONSTANTE 10        /* Ma constante */
```

L'option **-i** est très utile pour ne pas différencier majuscules et minuscules dans la recherche.

A savoir sur les expressions régulières:

Table 2. Expression régulières

Symbole	Signification
.	un caractère quelconque
|	une alternative
()	une sous-expression régulière
*	le caractère ou l'expression précédents se répètent n fois, avec $n \geq 0$
+	le caractère ou l'expression précédents se répètent n fois, avec $n > 0$
^	le début de la ligne

Symbole	Signification
$	la fin de la ligne

4.4.10. La commande find

La commande **find** permet de rechercher des fichiers selon certains critères. Sa syntaxe est : `find [chemin] [exp1] [exp2] ...` , où `chemin` est le répertoire de départ de la recherche. Les principales expressions utiles sont:

- `-name "filename"`: des substitutions sont possibles.
- `-print` ou `-ls`: affiche les fichiers trouvés.
- `-type f` ou `-type d`: matche les fichiers ou les répertoires.
- `-exec` execute une commande sur chaque fichier matché.

Exemple d'utilisation, les trois dernières commandes sont équivalentes:

```
% find ~ -name "*.o" -print

% find -name core -exec rm -f {} \;
% /bin/rm -f `find -name core -print`
% find -name core -print | xargs rm -f
```

4.4.11. sed, stream editor

C'est une commande très complexe. Elle est intéressante pour substituer des expressions entre le `stdin` et le `stdout`. Exemple:

```
% sed '30,50s/nom1/nom2/g' < fichier.c > fichier.new
```

`30,50` indique les numéros de lignes du flot d'entrée concernés par la substitution. `g` autorise plusieurs substitutions sur la même ligne. Un détail important dans l'écriture des expressions régulières est que la substitution concernera toujours la chaîne la plus longue possible.

L'intérêt de **sed** est qu'il est utilisable sous l'éditeur **vi**, en mode commande, accessible par `Escape`. Il peut aussi servir d'excellent outils de filtre.

4.4.12. awk

awk, à la différence de **sed** est un véritable langage de traitement de flot, et pas seulement un filtre d'expressions régulières. Quelques exemples d'utilisation:

Pour récupérer la troisième colonne d'un tableau de résultats:

```
% cat fichier | awk '{print $3}'
```

`print` peut etre remplacé par un `printf` pour une meilleure mise en forme, avec les mêmes arguments qu'en langage C.

Pour faire la somme des valeurs d'une colonne:

```
% awk 'BEGIN{n=0}{n+=$3}END{print n}' < fichier
```

Il existe une abondante documentation sur le sujet.

Le séparateur de champs est l'espace par défaut. Il est paramétrable.

4.4.13. Quelques fonctions largement utilisées dans les shells

basename retourne le nom d’un fichier en lui otant les répertoire qui précedent son nom et éventuellement l’extension de son nom de fichier.

```
% basename /a/b/c/d.txt .txt
d
```

Une application peut etre de changer toutes les extensions des fichiers d’un repertoire:

```
foreach file ( *.c )
?  mv $file `basefile $file .c`.old
?  end
```

On peut faire la meme chose avec un **sed**.