Introduction aux Outils de Développement GNU

Imed Chihi (Synapse) <imed@suse.com.tn>

9 Novembre 2003

Abstract

Ce document est une simple introduction à l'environnement de développement avec les outils du projet GNU. Il ne fait que initier le développeur à l'intéraction avec le compilateur et donne des notions sur la culture et les habitudes de développement sous Linux et Unix. Ce document suppose une connaissance basique du langage de programmation C et du shell Linux/Unix. Prière d'adresser toute proposition d'améliorations ou rapport d'omissions ou inexactitudes à l'auteur.

Change log

```
20030108 ichihi   Initial revision
20031109 ichihi   Added a section on setting up a CVS server
```

1 Le compilateur gcc

GCC est le GNU Compiler Collection, il s'agit de tout un environnement de compilation pour les langages C, C++, Fortran, Java et Objective-C. Les exemples suivants traiterons des cas en C, il existe sur le web de très bons tutoriels sur le sujet, le livre de Brian Kernighan (et Dennis Ritchie) "The C Programming Language" est aussi une référence incontournable.

1.1 Ahlan (Hello World)

L'exemple classique des livres de programmation est ce bout de code qui affiche tout bonnement la chaine "Ahlan" ("Hello World"). Vous pouvez utiliser votre éditeur de texte préféré pour créer les sources.

```
$ cat ahlan-0.c
#include <stdio.h>
int main() {
printf("Ahlan\n");
return 0;
}
```

Il est généralement coutume d'ajouter une entête à chaque source de fichier spécifiant des informations cruciales comme son objet, son auteur et la date du début de développement.

```
$ cat ahlan.h
/* ahlan.h - Définit une constante
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * Ce programme est un logiciel libre, vous pouvez le
 * distribuer et/ou le modifier sous les termes de la GNU GPL.
 */

#define PI 3.14
#include <stdio.h>
#include <math.h>
```

Nous saisissons l'occasion pour créer un second fichier (ahlan.h) pour y faire des déclarations.

```
$ cat ahlan-1.c
/* ahlan-1.c - Affiche sin(pi)
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * Ce programme est un logiciel libre, vous pouvez le
 * distribuer et/ou le modifier sous les termes de la GNU GPL.
 */

#include "ahlan.h"

int main() {
printf("Ahlan, sinus pi vaut %f\n", sin(PI));
return 0;
}
```

Le format de la date à la Japonaise (année-mois-jour) est le format ISO de représentation de la date. Il s'agit, en effet, du standard ISO-8601. Une adresse e-mail sur chaque fichier source serait très appréciée. Dans le cas de GNU GPL (*General Public License*), la notice sur la licence ainsi que le nom de l'auteur ne doivent, en aucun cas, être altérés.

1.2 indent

indent(1) est un outil de formatage syntaxique des sources C, il est souvent documenté dans la section 1 du manuel Linux/Unix. Ainsi, vous pouvez utiliser:

```
$ man 1 indent
```

pour voir la page de manuel de cet outil. Lorsque la littérature Unix mentionne des commandes (comme indent(1)), des fonctions ou des fichiers de configuration, souvent elle référence aussi la section du manuel dans laquelle la commande en question est documentée. Vous verrez donc des textes mentionnant gzip(1), read(2), lilo.conf(5), etc.

Par exemple pour formater notre source selon les conventions GNU, on peut utiliser:

```
$ cat ahlan-1.c | indent --gnu-style
```

1.3 Compilation

Comme nous l'avons déjà évoqué, GCC est bien plus qu'un compilateur. En effet, le passage depuis un code source en C vers un programme exécutable par la machine est un processus à plusieurs étapes. Dans le cas de GCC, par exemple, il s'agit de faire:

1. le pré-traitement (preprocessing): le langage du préprocesseur, qui n'est pas spécifique à C, sert à controler le processus de comilation même, il sert entre autres à faire des définitions de symboles et à faire des compilations conditionnelles selon l'architecture. C'est une manière pour préparer le source à être compilé. Dans le noyau de Linux on trouve, par exemple, des bouts de code en assembleur spécifique au processeur qui sont inclus selon l'architecture sur laquelle le noyau est entrain d'être compilé. Essayez la commande `cpp fichier.c` ou bien `gcc -E fichier.c`,
2. la compilation: cette étape consiste à convertir (compiler) un source C (ou autre) pré-traité (preprocessed) et d'en générer un code assembleur spécifique au processeur spécifié. Essayez la commande `gcc -S fichier.c` et consultez `fichier.s`,
3. l'assemblage: c'est la convertion (compilation) d'un source assembleur en code binaire (extension .o). Essayez la commande `gcc -c fichier.c` ou `gcc -c fichier.s`,
4. l'édition de liens: il s'agit de la dernière étape pour générer un programme exécutable par la machine. L'essentiel de cette étape consiste à résoudre les références à des symboles (noms de varaibles ou de fonctions) existants dans plusieurs modules assemblés (.o) et des librairies.

Ainsi pour compiler notre exemple décrit plus haut nous pouvons faire:

```
$ gcc -E ahlan-1.c > ahlan-1.i
$ gcc -S ahlan-1.i
$ gcc -c ahlan-1.s
$ gcc -o ahlan-1 ahlan-1.o -lm
$ ./ahlan-1
```

Heureusement que GCC sache arranger tout ça seul, et on peut juste utiliser:

```
$ gcc -o ahlan-1 ahlan-1.c -lm
$ ./ahlan-1
```

Notez l'utilisation de -lm pour indiquer au compilateur qu'il doit chercher certains symboles dans une librairie appelée libm.so.

1.4 Librairies

Dans des conditions usuelles, tous les programmes générés par GCC sont, par défaut, dynamiquement liés (*dynamically linked*) avec libc et ld-linux. libc est la librairie standard du langage C qui contient des fonctions définies par le standard ainsi que des points d'entrée vers les appels systèmes du noyau. ld-linux est la librairie contenant le code responsable de la gestion des librairies dynamiques comme le chargement automatique. Pour vérifiez la liste de librairies dont un programme dépond, utilisez:

```
$ ldd ./ahlan-1
      libc.so.6 => /lib/libc.so.6 (0x4002d000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Pour compiler un programme avec les librairies statiquement liées (*statically linked*), utilisez l'option static:

```
$ gcc -static -o ahlan-1 ahlan-1.c -lm
$ ldd ./ahlan-1
      not a dynamic executable
```

Comme noté plus haut, le compilateur suppose l'existance d'une librairie nommée libm.so en voyant l'option -lm et libX11.so en voyant -lX11. Les chemins vers ces librairies sont gérés par ldconfig(8). Par exemple pour lister toutes les librairies enregistrées, on peut utiliser:

```
$ ldconfig -v
```

Les chemins des répertoires contenant des librairies dynamiques sont listés sous /etc/ld.so.conf. Comme ldconfig met à jour un cache (/etc/ld.so.cache), il peut exiger des privilèges root pour certaines opérations. Pour ajouter un répertoire de librairies dans le contexte d'un seul utilisateur, il suffit de définir la variable LD_LIBRARY_PATH:

```
$ export LD_LIBRARY_PATH=/home/imed/src/libs/
```

Essayons de construire une librairie dynamique contenant les fonctions suivantes:

```
$ cat libkaren.c
/* libkaren.c - Librairie de comparaison d'entiers
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * Ce programme est un logiciel libre, vous pouvez le
 * distribuer et/ou le modifier sous les termes de la GNU GPL.
 */

int max(int a, int b) {
  return (a < b) ? b : a;
}

int min(int a, int b) {
  return (a < b) ? a : b;
}
```

```
$ cat libkaren.h
/* libkaren.h - Déclarations des fonctions de la librairie libkaren.so
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * Ce programme est un logiciel libre, vous pouvez le
 * distribuer et/ou le modifier sous les termes de la GNU GPL.
 */

int max(int a, int b);
int min(int a, int b);

$ gcc -fPIC -shared -o libkaren.so libkaren.c
```

L'option -fPIC indique au compilateur de générer des références d'adresses relatives et indépendantes (*Position Independent Code*) de l'adresse à laquelle le programme sera chargé. L'option -shared indique à l'editeur de liens (*linker*) qu'il s'agit d'un code partagé qui sera lié à d'autres modules plus tard lors du chargement du programme et qu'il ne faut pas se soucier trop des symbôles non résolus à ce niveau. Nous pouvons maintenant utiliser la librairie comme nous avons utilisé libm.so plus haut.

```
$ cat karen.c
/* karen.c - Test de comparaisons utilisant libkaren
 * Imed Chihi (Synapse) <imed@suse.com.tn>
 * 2003-01-03
 *
 * Ce programme est un logiciel libre, vous pouvez le
 * distribuer et/ou le modifier sous les termes de la GNU GPL.
 */

#include "libkaren.h"
#include <stdio.h>
int main() {
  printf("Le max de 4 et 2 est %d, le min est %d\n", \
  max(4, 2), min(4, 2));
  return 0;
}

$ gcc -o karen karen.c -L. -lkaren
$ LD_LIBRARY_PATH=. ./karen
```

L'option -L. indique à l'éditeur de liens un chemin supplémentaire à ceux reconnus par ldconfig(8) où il devra chercher les librairies dynamiques. On aurait pu utiliser la variable LD_LIBRARY_PATH d'une façon similaire.

2 make

make(1) est un outil de gestion de la compilation de projets à plusieurs modules. Il débarasse le développeur des recompilations inutiles lors de modifications de source. Une fois configuré, il sait gérer les dépendances et réalise, en effet, tout le processus de compilation. Pour l'exemple de karen.c, le fichier Makefile devrait ressembler à ceci:

```
$ cat Makefile
# Makefile - Fichier make principal
# Imed Chihi (Synapse) <imed@suse.com.tn>
# 2003-01-08
#

SRCS= karen.c
CC= gcc
LIBSRCS= libkaren.c
CFLAGS= -O2 -g -Wall
LIBSPATH= -L.
LIBS= -lkaren
LDFLAGS= -fPIC -shared
RM= /bin/rm -rf

karen: $(SRCS) libs
        $(CC) $(CFLAGS) -o karen $(SRCS) $(LIBSPATH) $(LIBS)

libs: libkaren.so
```

```
libkaren.so: $(LIBSRCS) libkaren.h
        $(CC) $(LDFLAGS) -o libkaren.so $(LIBSRCS)

clean:
        $(RM) libkaren.so karen
```

```
$ make
gcc -fPIC -shared -o libkaren.so libkaren.c
gcc -O2 -g -Wall -o karen karen.c -L. -lkaren
$ make
gcc -O2 -g -Wall -o karen karen.c -L. -lkaren
$ vi libkaren.h
$ make
gcc -fPIC -shared -o libkaren.so libkaren.c
gcc -O2 -g -Wall -o karen karen.c -L. -lkaren
```

Il s'agit, en effet, d'un graphe de dépendances comme illustré dans la figure 1. make(1) est très exigeant lorsqu'il s'agit du formatage et du respect des espaces et des tabulations dans Makefile. Chaque noeud dans l'arbre peut être traduit dans le fichier Makefile par une destination (*target*) comme all, libs et karen dans notre cas.

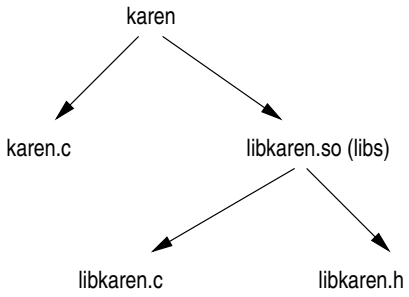

Figure 1: Graphe de dépendances du projet "karen".

3 Gestion de configuration

La gestion de configuration est un terme utilisé pour référencer tous les systèmes et techniques utilisés pour gérer les différentes versions d'un texte qu'il soit un document, un fichier de configuration ou un code source. Essentiellement, tout ce qui peut être modifié à travers le temps par une ou plusieurs personnes. L'outil de gestion de configuration le plus populaire sous Unix été SCCS mais il n'est plus utilisé maintenant, RCS est une évolution de SCCS et CVS est de loin l'outil de gestion de configuration le plus populaire de nos jours.

3.1 RCS et CVS

RCS(1) est le *Revision Control System*, c'est un système de gestion des versions d'un projet de développement comprenant plusieurs modules dans plusieurs fichiers. Il sert, entre autres, à suivre les changements effectués dans chaque fichier et de restituer le projet dans n'importe quel état antérieur du processus de développement. CVS(1) (*Concurrent Versioning System*) est le successeur de RCS, il ajoute des extensions pour supporter plusieurs développeurs simultanés faisant des modifications concurrentes aux fichiers d'un projet. Il s'agit de l'outil privilégié de gestion de configuration dans le monde Open Source et des connaissances de CVS sont quasiment indispensables dans un environnement de développement de plusieurs développeurs. Ce paragraphe va introduire CVS d'une manière brève.

Pour utiliser CVS, il faut tout d'abord créer un *repository* de source:

```
$ export CVSROOT=~/cvs
$ cvs init
```

Maintenant on doit créer un projet, appelé karen, dans ce repository:

```
$ mkdir ~/cvs/karen
```

A ce niveau on doit créer notre répertoire de travail dans lequel nous pouvons travailler sur une copie des fichiers pour la remettre après dans le repository:

```
$ cd ~/src/
```

On demande une copie de l'état actuel du repository, c'est vide pour le moment:

```
$ cvs checkout karen
```

On crée les fichiers du projet et on les ajoute au repository:

```
$ vi libkaren.c libkaren.h karen.c Makefile
$ cd ..
$ cvs add karen/Makefile; cvs add karen/karen.c; cvs add karen/libkaren.h;
cvs add karen/libkaren.c
$ cvs commit
```

Nous devons ajouter un commentaire du genre "Version initiale" au niveau du commit. A ce niveau nous avons nos fichiers dans le repository et nous pouvons travailler dessus avec plusieurs autres développeurs. CVS peut s'occuper même de certains cas de conflits où plusieurs développeurs modifient le même fichier. A chaque fois que des modifications signifiantes sont faites sur un fichier, il faut lancer:

```
$ cvs update
$ cvs commit
```

Si, entre temps, un autre développeur a modifié le source dans le repository, il sera signalé par le cvs update. Si, par malchance, il a modifié le fichier sur lequel nous travaillons, ceci sera aussi signalé, si les parties modifiées du fichier sont assez éloignées, CVS suppose que les changements ne sont pas conflictuels et les fusionne. Autrement, il modifie la copie locale du source pour mettre en évidence le conflit des deux modifications qui doit être résolu à la main et par commun accord entre les développeurs en question. La commande history donne le détail de toutes les transactions faites sur le repository:

```
$ cvs history -ae
O 2003-01-07 22:19 +0000 jamel karen =karen= ~/doc/dev/gnudev/cvs/*
A 2003-01-07 22:24 +0000 jamel 1.1 karen.c karen == ~/doc/dev/gnudev/cvs
A 2003-01-07 22:24 +0000 jamel 1.1 libkaren.c karen == ~/doc/dev/gnudev/cvs
A 2003-01-07 22:24 +0000 jamel 1.1 libkaren.h karen == ~/doc/dev/gnudev/cvs
O 2003-01-07 22:38 +0000 imed  karen =karen= ~/cvs/*
O 2003-01-07 22:38 +0000 imed  karen =karen= ~/cvs/*
M 2003-01-07 22:38 +0000 imed  1.2 karen.c karen == ~/cvs
C 2003-01-07 22:39 +0000 jamel 1.2 karen.c karen == ~/doc/dev/gnudev/cvs/karen
M 2003-01-07 22:40 +0000 jamel 1.3 karen.c karen == ~/doc/dev/gnudev/cvs
U 2003-01-07 22:40 +0000 imed  1.3 karen.c karen == ~/cvs/karen
M 2003-01-07 22:41 +0000 imed  1.4 karen.c karen == ~/cvs
U 2003-01-07 22:41 +0000 jamel 1.4 karen.c karen == ~/doc/dev/gnudev/cvs/karen
M 2003-01-07 22:42 +0000 imed  1.5 karen.c karen == ~/cvs
G 2003-01-07 22:42 +0000 jamel 1.5 karen.c karen == ~/doc/dev/gnudev/cvs/karen
M 2003-01-07 22:42 +0000 jamel 1.6 karen.c karen == ~/doc/dev/gnudev/cvs
```

Si vous trouvez cet affichage peu compréhensible, vous pouvez opter pour les multiples clients CVS comme WinCVS et jCVS. webcvs est une interface de consultation web très belle.

3.2 Mise en place d'un serveur CVS

Le binaire CVS fait fonction de client et serveur en même temps, il est disponible sur toutes les distributions Linux modernes. Seulement la configuration est un peu plus compliquée et, malheureusement, il est difficile de trouver un document sur le web qui explique tout sur cet aspect d'une façon compacte et claire. En tout cas voilà ce que je fait d'habitude sur SuSE Linux:

1. Création du repository
   ```
   # mkdir /usr/local/src
   # cvs -d /usr/local/src init
   ```

2. Définition des modules
```
# mkdir /tmp/junk
# cd /tmp/junk
# cvs -d /usr/local/src checkout CVSROOT/modules
# cat >> CVSROOT/modules
ps synapse/ps
baytar suse/src/baytar
^d
# cvs -d /usr/local/src commit
```
Ajouter un commentaire sur l'opération puis quitter vi
```
# export CVSROOT=/usr/local/src
# cvs release -d CVSROOT
```
Répondre "y" pour yes
```
# cd /usr/local/src/
# mkdir -p synapse/ps; etc.
```

3. Changement des permissions, je suppose ici que les utilisateurs de CVS appartiennent tous au groupe "cvsauthor"
```
# chgrp -R cvsauthor synapse suse
# chmod -R g+w synapse suse
```

4. Activation du service: éditer /etc/xinetd.d/cvs pour que la clause "disable" devienne "no", le mien ressemble à ceci:
```
$ cat /etc/xinetd.c/cvs
# CVS pserver (remote acces to your CVS repositories)
# Please read the section on security and passwords in the CVS manual,
# before you enable this.
# default:  off
service cvspserver
{
disable = no
socket_type = stream
protocol = tcp
wait = no
user = root
server = /usr/bin/cvs
server_args = -f --allow-root=/usr/local/src pserver
}
$
```
..et lancer:
```
# /etc/init.d/xinetd restart
```

5. Création des comptes: utiliser htpasswd(1) qui vient avec Apache pour générer les mots de passe cryptés,
```
$ cd /tmp/
$ htpasswd -c -d passwd imed
$ htpasswd -d passwd jamel
# cp /tmp/passwd /usr/local/src/CVSROOT/
```

6. Voilà c'est fini, depuis une machine sur le réseau vous pouvez essayer:
```
$ export CVSROOT=:pserver:imed@cvs.suse.com.tn:/usr/local/src
$ mkdir cvs ; cd cvs
$ cvs checkout ps
```
Créer et/ou modifier des fichiers sources, puis:
```
$ cvs commit
```

Ce qu'on appelle accès CVS anonyme est tout bonnement un compte avec un mot de passe vide.

3.3 diff et patch

diff(1) sert à comparer des fichiers et à générer une liste des différences. C'est très utile si un développeur veut envoyer ses modifications d'un projet à des utilisateurs ou d'autres développeurs sans avoir à envoyer tout le source. Supposons que nous allons faire un duplicata du répertoire karen appelé karen.orig et qu'on fasse des changements aux fichiers sous le répertoire karen, en ajoutant par exemple des commentaires.

```
$ cd ~/src
$ ls
karen karen.orig
$ diff -u --recursive --new-file karen.orig karen > karen-comments.patch
```

Ceci doit générer un fichier contenant toutes les modifications apportées aux fichiers du projet dans un format universel (-u) qui facilite leur positionnement.

Un développeur désirant appliquer (*patch*) ce patch à son source pour le rendre à jour ou bien pour corriger un bug, peut utilser la commande patch(1):

```
$ cd ~/src/karen/
$ patch -p1 < ../karen-comments.patch
```

Les développeurs du noyau Linux utilisent beaucoup ce méchanisme de diff et patch.