

## TD 2 d'Environnement Logiciel

Mercredi 15 septembre 2004

### **Le retour du TD1 Édition de lien, gcc, ld, make, objdump.**

Exemple d'un projet complet: *ecm*.

Ecm pour *Elliptic Curve Method* est un logiciel permettant de factoriser des nombres entiers via la méthode des courbes elliptiques. C'est surtout un projet contenant plusieurs modules en C que l'on va utiliser pour illustrer la compilation séparée.

- Récupérez les sources du projet avec la commande:  
`cvs -d:pserver:cvs@cvs-sop.inria.fr:/CVS/spaces co ecm`. Un répertoire `ecm` contenant le projet va être créé dans le répertoire courant.
- Combien de fichiers C composent le projet?
- Où se trouve le point d'entrée du programme?
- Tapez `make` et observez la sortie à l'écran. Avec `ls -la` avant et après l'exécution de `make`, regardez les fichiers créés par la commande. `make` est une commande très utile pour le développeur, voir <http://www.gnu.org/software/make/> pour plus d'informations.
- Tapez `make clean` et recommencez la compilation à la main (sans utiliser `make`) avec l'utilisation de `gcc` et `ld`.
- Plus fort: observez les diverses étapes de la compilation d'un fichier C de votre choix en utilisant `gcc -E`, `gcc -S`, `gas...`
- Après avoir tapé `make`, retapez `make`. Normalement, il ne se passe rien.
- Exécutez `touch median.o` puis `make`. Que se passe-t-il? Pourquoi?
- Éditez la fonction `main` du projet pour afficher un message de bienvenu de votre choix à chaque démarrage et tapez `make`. Quels étapes de la création du fichier exécutable final sont effectuées?
- Dissection avec `objdump`: à l'aide de la page de manuel d'`objdump`, regardez le contenu des fichiers objets produits.
- Et si les binutils n'existaient pas:  
<http://www.comms.scitech.susx.ac.uk/fft/programming/teensy.html>.

## Entrées-sorties en C

- Comparez les résultats de `printf("tic");`, de `fprintf(stdout, "tic");` et de `fprintf(stderr, "tic");`.
- Comparez les résultats de `{ fprintf(stdout, "tic"); while (1);}` et de `{ fprintf(stderr, "tic"); while (1);}`.
- Comparez les résultats de `{ fprintf(stdout, "tic"); fflush(stdout); while (1);}` et de `{ fprintf(stderr, "tic"); while (1);}`.
- Comparez les résultats de `{ fprintf(stdout, "tic\n"); while (1);}` et de `{ fprintf(stderr, "tic\n"); while (1);}`.
- Comparez `scanf("%d", &x);` et `fscanf("%d", &x);`.

Les objets `stdin`, `stdout` et `stderr` sont respectivement appelés *entrée standard*, *sortie standard* et *sortie standard d'erreur*. Ils correspondent aux *file descriptor* `STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO` (voir `/usr/include/unistd.h`).

### Fichiers

Les objets fichiers (ou plus exactement *stream*) ont pour type `FILE *`. Quel est le stype de `stdin`, `stdout` et `stderr`? (cf. `/usr/include/stdio.h`)

- Créez un fichier à partir d'un programme à l'aide de `f = fopen("fichier", "w");`. Cette fonction retourne un objet de type `FILE *`. Écrivez quelques lignes dans le programme en utilisant `fprintf` avec comme premier paramètre `f`. N'oubliez pas de fermer le fichier avec `fclose`.
- Que se passe-t-il si vous exécutez plusieurs fois votre programme? Remplacez `w` par `a` dans la commande `fopen` et réitérez l'expérience.
- À quoi correspondent les différents modes? Expliquez le comportement observé.
- Créez deux programmes `p1.c` et `p2.c`. Le programme `p1.c` demande un entier `n` à l'utilisateur puis écrit les entiers de 0 à `n - 1` dans un fichier `fichier_int` (avec éventuellement des informations supplémentaires si nécessaire). `p2.c` doit être capable de lire la suite d'entiers écrite dans le fichier `fichier_int` et de la stocker dans un tableau de `n` entiers alloué dynamiquement avec `malloc`, puis d'afficher le contenu du tableau.
- Écrivez un programme qui affiche le nombre de lignes d'un fichier. Comparez le résultat de votre programme avec la commande `wc -l`.
- Modifiez votre programme pour que le nom du fichier à analyser puisse être donné en paramètre. Rappel: pour un programme écrit en C et déclarant sa fonction `main` par `int main (int argc, char *argv[])`, `argc` correspond au nombre d'arguments présents sur la ligne de commande et le tableau `argv` contient ces arguments: `argv[0]` est le nom du programme, `argv[1]` le premier argument...

- Modifiez votre programme pour afficher également le nombre de mots et comparez avec `wc fichier`.

### Shell et redirections

- Essayez :

1. `sort < fichier.txt`
2. `echo 10 > num.txt`
3. `echo 20 >> num.txt`
4. `ls | wc -l`
5. `ps aux | grep 'whoami'`.

en expliquant ce qu'il se passe dans chaque exemple.

- Que constatez vous si vous exécutez `ls > /dev/null`? Quel est l'intérêt de la redirection vers `/dev/null`? Voir:  
[http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/faq/funnies.html#DEV-NULL](http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/funnies.html#DEV-NULL)
- Quel est le résultat de `cat < /dev/null > fichier.txt`? Qu'en déduisez vous sur un usage possible de cette commande?
- Les commandes `head` et `tail` affichent respectivement le début et la fin d'un fichier. Écrivez le script `milieu <debut> <fin> <fichier>` qui met à profit ces deux commandes pour afficher les lignes de `debut` à `fin` incluses du fichier `fichier`.

### Entrée sortie, encore

- Écrivez une version en C de `cat` en utilisant uniquement les fonctions d'entrée sortie de la bibliothèque standard `fread` et `fwrite`, et en lisant et écrivant des blocs d'un caractère.
- Écrivez une seconde version de `cat` en utilisant uniquement les appels systèmes `read` et `write`.
- En utilisant la commande `time` comparez les performances des deux versions pour une lecture/écriture d'un fichier raisonnablement volumineux.
- Interprétez les résultats obtenus. Quels sont les commandes de plus bas niveau: `read/write` ou `fread/fwrite`?
- Essayez le programme suivant:

```
#include <stdio.h>
#include <unistd.h>
```

```
int main ()
{
    fwrite ("Hello ", sizeof(char), 6, stdout);
    write (STDOUT_FILENO, "World !", 7);

    return 0;
}
```