Compilation et Optimisation de code
TP 1

ENSSAT – Systèmes Numériques 3

Camille Oudot
camille.oudot@gmail.com
Francois Charot, charot@irisa.fr

Exercice 1 : Compilation séparée, Makefiles

Récupérer `stat.tgz` sur l'ENT et le décompresser.

Le projet récupéré est composé de deux modules :

- **libstat** : une toute petite librairie qui doit calculer des statistiques sur des séries de données
- **libstat_test** : un programme de test pour valider la librairie ci-dessus

Étape 1 : Implémenter dans le projet **libstat** la fonction **moyenne arithmétique** (*mean*), dont la signature est définie dans **mean.h**. Créer le fichier **mean.c** pour y placer l'implémentation. On utilisera la formule itérative naïve :

$$\overline{X} = \frac{1}{n} \times \sum_{i=1}^{n} x_i$$

X étant la série statistique (`data_set`) et n le nombre de valeurs dans X (`data_set_len`).

Étape 2 : Implémenter dans le projet **libstat** la fonction **variance**, dont la signature est définie dans **variance.h**. Créer le fichier **variance.c** pour y placer l'implémentation. On utilisera la formule itérative naïve :

$$V(X) = \frac{1}{n} \times \sum_{i=1}^{n} (x_i - \overline{X})^2$$

Étape 3 : Implémenter dans le projet **libstat** la fonction **écart type** (*stddev*), dont la signature est définie dans **variance.h**. La placer dans le fichier **variance.c**. On utilisera la définition suivante :

$$\sigma(X) = \sqrt{V(X)}$$

Étape 4 : Compléter le fichier **Makefile** du module **libstat**, pour qu'il produise la bibliothèque partagée **libstat.so** à partir des deux fichiers sources créées ci-dessus.

Étape 5 : Compléter le fichier **Makefile** du module **libstat_test** pour qu'il produilse le programme **test** à parir de **test.c**. Ce programme devra être lié dynamiquement à la librairie **libstat.so**.

> **Remarque :** vous pouvez désigner la librairie du module **libstat** depuis le module **libstat_test** en utilisant un chemin **relatif** : `../libstat/libstat.so`.

Étape 3 : Compléter la cible **run** de ce même makefile pour qu'ell exécute le programme **test**.

Remarque : ne pas oublier de positionner la variable `LD_LIBRARY_PATH`. Pour lancer une commande en lui passant la valeur `foo` dans la variable d'environnement `BAR`, on peut précéder l'appel de la commande de `BAR=foo`, par exemple :

```
BAR=foo commande param1 param2 ...
```

Testez votre librarie avec `make run`, le résultat attendu est :

```
[PASS] Tested: "moyenne arithmétique"
[PASS] Tested: "variance"
[PASS] Tested: "écart type"

TOTAL: 3 tests, 3 passed, 0 failed
```

Exercice 2 : Analyse du plan mémoire d'un processus

L'objectif est d'écrire un programme en C qui mette en évidence les différents segments liés aux différents mécanismes d'allocation. À l'issue de cet exercice, vous saurez à quel endroit de la mémoire se trouvent les donnée et les variables que manipulent les programmes en langage C.

Q1 : Écrire un programme en C qui affiche l'adresse de la première instruction de la fonction `main()`, puis qui attend un appui sur entrée d'une pour se terminer (utiliser la fonction `getchar()`).

Remarque 1 : Dans le langage C, le nom d'une fonction utilisé dans un contexte de valeur représente son adresse en mémoire.

```c
float func(int x) { ... }
float x = func(42);  /* x reçoit la valeur de retour de la fonction
                        func appelée avec le paramètre 42 */
void *y = func;      /* y reçoit l'adresse de la fonction func */
```

Remarque 2 : Le format pour afficher une adresse avec la fonction `printf()` est `"%p"`.

```c
int i = 0;
printf("l'adresse de i est %p\n", &i);
```

Q2 : Compiler et exécuter ce programme. Chercher le segment du plan mémoire dans lequel se trouve le code de la fonction `main()`, (fichier **/proc/PID/maps**, où **PID** est l'identifiant du processus). Quelles sont les permissions du segment en question ? Ce segment est-il anonyme ou adossé à un fichier (voir remarque 2) ? Chercher également dans quelle section du fichier ELF a été placée la fonction `main()` (utiliser `objdump -h EXÉCUTABLE`).

Remarque 1 : Le pseudo-fichier **/proc/PID/maps** n'existe que pendant la durée de vie du processus en question, c'est pourquoi il est nécessaire de faire attendre notre programme avec la fonction `getchar()`.

Remarque 2 : Sous UNIX, les segments en mémoire sont soit adossés à des fichiers, soit anonymes. Être adossé à un fichier signifie que le contenu du segment mémoire correspond au contenu du fichier. C'est le système d'exploitation qui charge le contenu du fichier à l'adresse donnée lorsque le processus y accède. Un segment anonyme est un segment qui n'est adossé à aucun fichier : c'est une simple zone mémoire dans laquelle un processus peut lire ou écrire des données pendant sa durée de vie (selon les permissions du segment).

Q3 : Compléter le programme pour y afficher l'adresse :

- d'une variable globale initialisée
- d'une variable statique (qualificateur `static`) dans une fonction, initialisée
- d'une constante globale (qualificateur `const`) initialisée
- d'une variable globale non-initialisée

Pour chaque adresse, chercher dans quel segment du plan mémoire elle se trouve, ainsi que dans quelle section de l'exécutable ELF elle a été placée.

Q4 : Compléter le programme pour y afficher l'adresse :

- d'une variable locale
- d'un paramètre de fonction
- de zones allouée dynamiquement avec `malloc()`, l'une de 16 octets, l'autre de 512 kio

Pour chaque adresse, chercher dans quel segment du plan mémoire elle se trouve.

Q5 : Ajouter la variables globale suivante :

```
char bigvar1[1024*1024] = {[0 ... 1024*1024 - 1] = 42};
```

Que dire de la taille de l'exécutable obtenu ?

Ajouter la variable globale suivante :

```
char bigvar2[1024*1024];
```

Même question.