Architecture des ordinateurs

Fiche n°5

ESIPE - INFO 1 2024 –2025

Convention d'appel du C sous Linux

Cette fiche est à faire en une séance (soit 2 h, sans compter le temps de travail personnel), et en binôme. Il faudra

1. réaliser un rapport soigné à rendre **au format pdf** contenant les réponses aux questions de cette fiche (exercices marqués par ■), une introduction et une conclusion ;
2. écrire les différents fichiers sources des programmes demandés. Veiller à nommer les fichiers sources `Exx.asm` où `xx` est le numéro de l'exercice. Ceux-ci doivent **impérativement** être des fichiers compilables par Nasm ;
3. réaliser une archive **au format zip** contenant les fichiers des programmes et le rapport. Le nom de l'archive doit être sous la forme `AO_TP5_NOM1_NOM2.zip` où `NOM1` et `NOM2` sont respectivement les noms de famille des deux binômes dans l'ordre alphabétique ;
4. déposer l'archive sur la plate-forme de rendu.

Tous les fichiers complémentaires nécessaires à ce TP se trouvent sur le site
https://igm.univ-mlv.fr/~derycke/AO/
Le but de cette séance est de poursuivre l'écriture de fonction en suivant une convention quant à l'utilisation des registres et le passage des arguments.

1 Convention d'appel du C

La convention d'appel du C est essentiellement la suivante :

- les paramètres de la fonction sont passés par la pile et non dans les registres comme pour les fonctions `print_int` et `print_string`
- au début de la fonction, on empile la valeur de `ebp`, et on sauvegarde la valeur `esp` dans `ebp`. À la fin de la fonction, la valeur de `ebp` est restaurée
- l'éventuelle valeur de retour de la fonction doit être placée dans `eax`
- les valeurs des registres `eax`, `ecx` et `edx` peuvent être modifiées à l'envie. Les valeurs des autres registres doivent être les mêmes avant et après l'appel à la fonction. (on parle de registres *caller-save* et *callee-save*)

Un squelette possible de fonction est le suivant :

```
fonction :
    push ebp      ; Valeur de ebp empilee.
    mov ebp, esp ; Sauvegarde la tete de pile dans ebp.

    ; Debut du corps de la fonction.
    ; argument 1 @ ebp + 8
    ; argument 2 @ ebp + 12
    ; argument k @ ebp + 4 * (k + 1)
    ; Fin du corps de la fonction.

    pop ebp      ; Valeur de ebp restauree.
    ret
```

Pour appeler cette fonction, on utilise une suite d'instructions de la forme :

```
push arg3       ; On empile les arguments dans l'ordre inverse.
push arg2
push arg1
call fonction   ; Appel de la fonction.
add esp, 12     ; Depile d'un coup les trois arguments.
```

Au début de l'exécution du corps de la fonction, l'état de la pile, relativement à `ebp`, est

Table 1: État de la pile

Adresse	Valeur
⋮	⋮
`ebp` ancie	nne valeur de `ebp`
`ebp` + 4	adresse retour
`ebp` + 8	argument 1
`ebp` + 12	argument 2
`ebp` + 16	argument 3
⋮	⋮

Exercice 1 Donner les avantages de passer les arguments d'une fonction par la pile plutôt que par les registres. Essayer de réfléchir en termes d'avantages lors de l'écriture d'une fonction ainsi que lors de l'utilisation de fonctions.

Exercice 2 ■ Écrire une fonction `difference` respectant les conventions d'appel du C, prenant deux nombres a et b en argument et renvoyant leur différence $a - b$ dans `eax`. Écrire un programme qui demande deux entiers à l'utilisateur et affiche leur différence. La différence doit être calculée grâce à la fonction `difference`.

Exercice 3 Écrire une fonction `occurrence` qui respecte les conventions d'appel du C et qui lit au clavier :

- une suite d'entiers compris entre 0 et 100, terminée par un -1 ;
- puis un entier a compris entre 0 et 100.

La fonction doit renvoyer dans `eax` le nombre d'occurrences de a dans la suite. Pour mémoriser le nombre d'occurrences de chaque nombre entre 0 et 100, il faut utiliser dans cet exercice un tableau de 101 `dword` enregistré dans la pile. Écrire un programme qui appelle la fonction `occurrence` puis qui affiche le resultat de la fonction.

Exercice 4 Modifier la fonction et le programme précédent afin que la fonction `occurrence` puisse accepter une suite d'entiers non bornés (mais bien entendu bornés par la limite de ce qui est encodable sur 32 bits).

Exercice 5 ■ Écrire une fonction récursive `longueur` qui respecte les conventions d'appel du C et qui prend en argument l'adresse d'une chaîne de caractères terminée par un octet 0 et calcule sa longueur. Le résultat sera renvoyé dans `eax`.

Exercice 6 Réaliser une fonction `fibonacci` qui respecte les conventions d'appel du C et qui calcule de manière récursive le n^{ieme} élément F_n de la suite de Fibonacci. Cette suite est définie par $F_0 := 0$, $F_1 := 1$ et pour $n \geqslant 2$, $F_n := F_{n-1} + F_{n-2}$. Le résultat sera renvoyé dans le registre `eax`. La fonction sera appelé dans un programme qui lit un entier positif n au clavier et affiche la valeur F_n.

Remarque 1 Cet algorithme (récursif) de calcul d'un terme de la suite de Fibonacci est mauvais en complexité temporelle et spatiale. Il n'y a pas lieu de s'étonner si le programme ainsi écrit prend beaucoup de temps à l'exécution pour des petites valeurs d'entrée.

Exercice 7 ■ Modifier la fonction et le programme précédent de telle sorte que la fonction prenne deux arguments entiers supplémentaires a et b et qui renvoie le n^{ieme} élément $F(a,b)_n$ de la suite définie par $F(a,b)_0 := 0$, $F(a,b)_1 := 1$ et pour $n \geqslant 2$, $F(a,b)_n := a \times F(a,b)_{n-1} + b \times F(a,b)_{n-2}$. La nouvelle fonction doit s'appeler `suiterecursive`.

Exercice 8 ■ Proposer une signature en C pour chacune des fonctions précédentes.

---

2 De l'assembleur et du `C`

On se propose de s'assurer du bon respect de notre convention d'appel. Pour cela, on va créer un programme `C` utilisant les fonctions qu'on aura coder en assembleur et inversement.

Dans les deux cas, on aura (au moins) un module codé en assembleur et un module codé en `C` à compiler chacun en fichier object (`.o`) et assembler à la fin via le compilateur de `C` (`cc/gcc/clang`).

```
# Compilation du code assembleur (comme avant)
nasm -f elf32 ModuleASM.asm
# Compilation du code C (-m32 pour la compilation en 32bit)
```

```
cc -Wall -m32 -c -o ModuleC.o ModuleC.c
# Édition de liens
cc -m32 -no-pie ModuleC.o ModuleASM.o -o Exec
```

2.1 Rappel sur la modularisation

Lorsqu'un programme est constitué de plusieurs modules en assembleur, les fonctions d'un module destinées à être utilisées depuis d'autres modules doivent être déclarées visible à l'aide de la directive `global` dans le module et déclarées `extern` depuis les autres modules (éventuellement à l'aide d'un fichier entête comme le module `asm_io`).

```
extern mult      ; Fonction d'un autre module

global longueur ; Fonction visible du module
longueur:
    ...

routine:         ; Fonction non visible du module
    ...
```

En `C`, les fonctions de modules sont visibles par défaut (il faut utiliser le mot clé `static` pour obenir le comportement inverse). Cependant les fonctions hors d'un module doivent être décalaré avant d'être utilisé. Le même mot clé `extern` peut alors être utilisé[1].

```
// Fonction visible du module
int mult(int a, int b) { ... }
// Fonction non visible du module
static void error(void) { ... }
// Déclaration de fonction d'autres modules
extern int convertir(char *s);
int convertir(char *s); // équivalent
```

2.2 Exemple 1 : Appeler du `C` depuis l'assembleur depuis du `C`

Voici un exemple à deux modules disponible dans les fichiers fournis et compilable avec le script `CompileAsmC.sh` via la commande `$ ./CompileAsmC.sh exemple`.

```
section .text
extern constant
global somme
somme:
    call constant
    add eax, dword [esp + 4]
    ret
```

```
#include <stdio.h>
int somme(int);
int constant(void) {
    return 38;
}
int main(void) {
    int a = somme(4);
    printf("%d\n", a);
    return 0;
}
```

[1] il est nécessaire dans le cas des variables globales externes au module

Le point d'entrée du programme est le même qu'en `C`, la fonction `main` définie ici dans le code `C`. La fonction `main` appelle la fonction `somme` définie dans le module écrit en assembleur. La fonction `somme` appelle elle-même la fonction `constant` définie dans le module écrit en `C`.

Exercice 9 À l'aide de la commande `./CompileAsmC.sh exemple`, compiler et tester l'exemple précédent.

2.3 Tests de code assembleur

Exercice 10 ■ Écrire un code `C` testant (succinctement) les fonctions `difference`, `longueur` et `suiterecursive`. Les trois fonctions seront placées dans un même module en assembleur. La fonction principale sera placée dans le module en C.

2.4 Utilisation de la `libc`

On se propose maintenant d'utiliser directement des fonctionnalités de la bibliothèque standard du C depuis un code écrit en assembleur. En utilisant le compilateur du `C`, le point d'entrée `_start` est utilisé et est fourni par le compilateur. Ce dernier se charge d'appeler la fonction `main` et d'utiliser son code de retour comme code de retour du programme.

Ainsi, le `main` devient une fonction et il n'est plus nécessaire de réaliser un appel système pour terminer l'exécution du programme.

Exercice 11 Chercher la documentation des fonctions `open/close`, `read/write` dans le manuel.

Exercice 12 Écrire un programme lisant le contenu d'un fichier et l'écrivant dans la sortie standard.

Lorsque le compilateur du `C` est utilisé, le code fourni à l'étiquette `_start` récupère les arguments de la ligne de commande et les places comme arguments de la fonction `main(int argc, char *argv[])`. On peut alors récupérer ces arguments comme tout argument de fonction suivant la convention d'appel du `C`.

Exercice 13 Modifier le programme afin d'utiliser le premier argument de la ligne de commande pour le nom de fichier. Le programme aura comme code de retour 1 si aucun argument n'a été donné.
