Architecture des ordinateurs

IR1 Informatique 2011-2012

TP 3

Objectif de cette séance

L'objectif de cette séance est d'apprendre à utiliser la pile et à écrire des fonctions en assembleur. En particulier, nous verrons comment réaliser des fonctions qui suivent les conventions d'appel du C.

Cette fiche est à faire en binôme. Il faudra :

1. réaliser un — bref — rapport à rendre **au format pdf** contenant les réponses aux questions de cette fiche ;
2. réaliser une archive **au format zip** contenant les sources des programmes ;
3. envoyer le tout à l'adresse de votre chargé de TP

   `Samuele.Giraudo@univ-mlv.fr` ou `vcarnino@gmail.com`

   par un courriel dont le sujet est sous la forme

   `IR1 Archi TP3` $NOM1$ `;` $NOM2$

Tous les fichiers nécessaires pour ce TP se trouvent à l'adresse

`http://igm.univ-mlv.fr/~giraudo/Enseignements/IR1/Archi/Archi.html`.

1 La pile

La pile désigne une zone de la mémoire qui se trouve avant l'adresse `0xBFFFFFFF`. Par convention, la pile sera utilisée pour stocker exclusivement des `dword` (4 octets). L'adresse du dernier `dword` est enregistrée dans le registre `esp`. Cette zone est appelée *haut de pile*.

Important 1. La pile *grandit* vers le bas : plus on ajoute d'éléments dans la pile plus la valeur de `esp` diminue.

On dispose de deux instructions pour faciliter la lecture et l'écriture dans la pile : `push` et `pop`. L'instruction `push eax` ajoute le `dword` contenu dans `eax` en haut de la pile (et modifie donc `esp` en conséquence). L'instruction `pop eax` enlève le `dword` se trouvant en haut de la pile et le copie dans le registre `eax` (et modifie donc `esp` en conséquence).

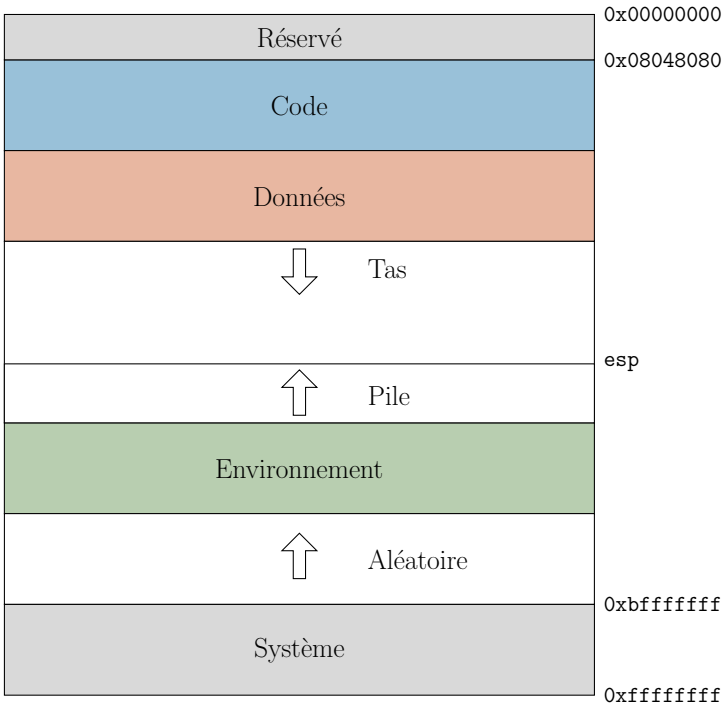

Figure 1: Organisation de la mémoire sous Linux en mode protégé.

Question 1. Pour la suite d'instructions suivante, donner instruction par instruction les valeurs des registres `esp` et `eax` ainsi que l'état de la pile.

```
1 mov eax, 0xABCDEF01
2 push eax
3 mov eax, 0x01234567
4 push eax
5 pop eax
6 pop eax
```

Question 2. Donner des suites d'instructions utilisant uniquement les instructions `mov` et `add` afin de simuler respectivement les d'instructions `push eax` et `pop eax`.

Question 3. Dans le code suivant, le registre `ebp` est utilisé pour sauvegarder la valeur de `esp`. Nous verrons plus tard que c'est en fait le rôle traditionnel de `ebp`. Donner l'état des registres `eax`, `ebx`, `esp` et de la pile avant et après l'exécution des instructions aux étiquettes `un`, `deux`, `trois` et `quatre`. L'état de la pile doit être donné sous la forme suivante :

Adresse	Valeur
ebp - 4	...
ebp - 8	...
⋮	⋮

```
1       mov ebp, esp
```

```
2       mov eax, 0
3       mov ebx, 0
4       push dword 12
5       push dword 13
6       push dword 15
7   un :
8       pop eax
9       pop ebx
10      add eax, ebx
11  deux :
12      push eax
13      push ebx
14      mov dword [esp+8], 9
15  trois :
16      pop eax
17      pop ebx
18      pop ebx
19  quatre :
```

Question 4. Écrire un programme qui lit des entiers au clavier tant qu'ils sont différents de -1, et affiche la liste des entiers lus dans l'ordre inverse. Par exemple, sur l'entrée

`2 5 6 7 1 3 2 4 -1`,

le programme affichera

`4 2 3 1 7 6 5 2`.

Astuce 1. Il est possible d'utiliser la pile pour stocker les entiers lus.

Bonus 1. Réaliser un programme qui lit une suite d'entiers terminée par -1 et qui affiche cette liste triée dans l'ordre croissant. Un simple tri à bulles ou un tri par sélection suffira.

2 Appels de fonction

Nous allons introduire le mécanisme de base pour écrire des fonctions en assembleur. Les deux instructions que l'on utilise sont `call` et `ret` :

- l'instruction `call label` empile l'adresse de l'instruction suivante sur la pile et saute à l'adresse `label`. Le fait de se souvenir de cette adresse permet de reprendre l'exécution des instructions après que les instructions correspondant au saut aient été exécutées.
- L'instruction `ret` dépile le `dword` du haut de pile et effectue un saut à cette adresse. C'est l'adresse empilée précédemment par l'instruction `call`.

Question 5. Donner une suite d'instructions n'utilisant que `push` et `jmp` qui soit équivalente à l'instruction `call print_int` dans le contexte suivant :

```
1       call print_int
2   suite :
3       mov eax, 0
```

Question 6. Donner l'évolution de la pile au cours de l'exécution des instructions suivantes. On pourra éventuellement expliquer le comportement de ces instructions sous la forme d'un schéma.

```
1  debut :
2      call pointA
3  point1 :
4      call pointB
5  point2 :
6      mov ebx, 0
7      mov eax, 1
8      int 0x80
9  pointA :
10     call pointB
11 point3 :
12     ret
13 pointB :
14     ret
```

Question 7. Expliquer l'élément problématique — s'il existe — contenu dans le code suivant :

```
1  fonction :
2      call read_int
3      push eax
4      pop ebx
5      push ecx
6      ret
```

Question 8. Expliquer le comportement du programme `Erreur.asm`.

Question 9. Réécrire la fonction `print_string` en utilisant l'appel système `write`. Rappelons que cet appel système affiche les caractères de la mémoire situés à partir de l'adresse contenue dans `eax`, et arrête l'affichage lorsque un octet nul est lu. Faire en sorte qu'après l'exécution de la fonction, les registres soient dans le même état qu'avant l'exécution de la fonction.

3 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 sauvegarde la valeur de `ebp` dans la pile et on copie la valeur de `esp` dans `ebp`. On accède ainsi aux paramètres de la fonction par l'intermédiaire de `ebp`. De plus, à la fin de la fonction, la valeur de `ebp` est restaurée.
- Le résultat retourné par la fonction doit être copié dans un registre. Les valeurs des autres registres doivent être les mêmes avant et après l'appel à la fonction.

Un squelette possible de fonction est le suivant :

```
1  fonction :
2      push ebp       ; Valeur de ebp empilee pour la sauvegarder.
3      mov ebp, esp   ; Sauvegarde du haut de pile dans ebp.
4
5      ; Debut du corps de la fonction.
6      ; argument 1 @ ebp + 8
7      ; argument 2 @ ebp + 12
8      ; argument k @ ebp + 4 * (k + 1)
9      ; Fin du corps de la fonction.
10
11     pop ebp        ; Valeur de ebp restauree.
12     ret
```

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

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

Au début du corps de la fonction, l'état de la pile est le suivant :

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

Question 10. Donner les avantages de passer les arguments d'une fonction par la pile plutôt que par les registres.

Question 11. Écrire une fonction respectant les conventions d'appel du C, prenant deux nombres a et b en argument et renvoyant leur différence $a - b$ dans `eax`. Donner le code pour appeler cette fonction avec les arguments $a := 1200$ et $b := 34$.

Question 12. Modifier la fonction précédente pour qu'elle dépile les arguments avant de sortir.

Question 13. Écrire une fonction 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.

Bonus 2. Modifier la fonction précédente afin qu'elle puisse accepter une suite d'entiers non bornés (mais bien entendu bornés par la limite de ce qui est encodable sur 32 bits).

Question 14. Écrire une fonction 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 retourné dans `eax`. Donner également une version récursive de cette fonction.

Question 15. Réaliser une fonction qui respecte les conventions d'appel du C et qui calcule de manière récursive le n-ième élément F_n de la suite de Fibonacci. Cette suite est définie par $F_0 := 0$, $F_1 := 1$ et pour $n \geq 2$, $F_n := F_{n-1} + F_{n-2}$. Le résultat sera renvoyé dans le registre `eax`.

Bonus 3. Modifier la fonction précédente de sorte à ce qu'elle prenne deux arguments entiers supplémentaires a et b et qui retourne le n-ième élément $F(a,b)_n$ de la suite définie par $F(a,b)_0 := 0$, $F(a,b)_1 := 1$ et pour $n \geq 2$, $F(a,b)_n := a \times F(a,b)_{n-1} + b \times F(a,b)_{n-2}$.