DESS CCI : examen Langage Machine, Décembre 2006

1 procédures

La procédure chercher est récursive. Il existe autant d'exemplaires de wanted et indice que d'appels de chercher en cours. A chaque appel de chercher, un nouveau bloc de paramètres (contenant un exemplaire de wanted) et un nouveau bloc local (contenant un exemplaire de indice) sont empilés, ce qui explique que les adresses affichées décroissent. La taille mémoire totale allouée dans la pile à chaque appel est constante : ici 64 octets (0x40).

La différence entre les adresses de wanted et indice est constante : 0x7fffcc - 0x7fffa8 = 0x7ffe0c - 0x7ffde8 = 0x24 = 36, soit 9 mots de 32 bits. Quatre de ces mots correspondent au stockage des anciennes valeurs de fp, sp, lr et à l'adresse du corps de chercher. Quatre autres correspondent aux sauvegardes des registres r0 à r3 qui seront modifiés par la procédure lorsqu'elle s'appelera à nouveau elle-même. On peut supposer que la procédure utilise aussi un autre registre (pour stocker les valeurs lues en mémoire) qu'il faut sauvegarder également.

Lorsque la valeur n'a pas été trouvée dans le tableau, le programme va parcourir dans la pile les mots qui suivent le dernier élément du tableau, à savoir les dernières variables locales, puis les sauvegardes de registres, puis le tableau des paramètres reçus de l'appelante (sauf les 3 premiers).

Indice est stocké à l'adresse qu'aurait occupé l'élément tab[4] s'il avait existé. Wanted est stocké à l'adresse qu'aurait occupé l'élément tab[13] s'il avait existé. Wanted contenant la valeur recherchée, l'exécution du programme se termine sur l'indice 13. S'il y avait eu un paramètre supplémentaire avant wanted, la recherche se serait terminée à l'indice 14.

2 Variables

Les variables sont toutes déclarées avec une valeur initiale et seront donc stockées dans la section data, en faisant attention à respecter les contraintes d'alignement lorsqu'on passe d'une variable à une variable de plus grand format. Message occupe 9 octets : 8 caractères de la chaîne plus la marque de fin (un octet à 0).

```
            .data
us:         .half    0x1234     @ ou .short 0x1234
            .balign  4          @ ou .skip 2 pour  aller à 4X
l:          .word    0x12345678
c1:         .byte    0x61
c2:         .byte    0x62
c3:         .byte    0x63
```

```
            .balign  2          @ ou .skip 1 pour aller à 2X
s:          .half    0x6100
message:    .asciz   "bonjour!"
            .balign  4
p:          .word    c1
```

Le programme va essayer d'afficher caractère par caractère le contenu des octets c1 et suivant, jusqu'à rencontrer une marque de fin de chaîne, à savoir un octet à 0. C1 contient le code ASCII de 'a', c2, celui de 'b' et c3 celui de 'c'.

Avec une machine big endian, l'octet occupant l'adresse de s en contient les bits de poids forts, en l'occurence 0x61 (code de 'a') et l'octet d'adresse s+1 contient alors 00. Avec une machine little endian, l'octet d'adresse s contiendra 0 et celui d'adresse s+1 contiendra 'a'. Dans les deux cas, la suite d'octets forme une chaîne de caractères terminée par un 0. Le programme affichera donc "abca" en big endian et "abc" en little endian.

```
i1:     ldr    r1,= c2       @ r1 = &c2
i2:     ldrsb  r0, [r1]      @ r0 = c2 = *&c2
i3:     ldr    r1,= p        @ r1 = &p
i4:     ldr   r1, [r1]       @ r1 = p = *&p
i5:     strb  r0, [r1]       @ **&p = *&c2
        .ltorg

@ rappel :
         ldr   r1, =C2
@ equivaut a
ici:     ldr   r1, [pc, #((relais -ici)/4-2)]
         ...
relais:  .word c2
```

L'affectation se traduit par cinq instructions load ou store, dont chacune représente deux accès mémoire : l'un pour lire le code opération de l'instruction dans la section text, et l'autre pour transférer le contenu du mot accédé de ou vers le registre. Au total : 7 lectures dans text (5 fois code-op plus 2 accès aux .word dans ltorg), 2 lectures et 1 écriture dans data. Aucun accès à bss (pas de variable déclarée sans initialisation).

Instructions i1 et i3 : le contenu lu dans le registre vient de la zone .ltorg incluse dans la section text. Instructions i2, i4 : le contenu est lu dans une variable stockée dans la section data. Instruction i5 : le contenu est écrit dans une variable stockée dans la section data.

3 Entiers en binaire

3.1 exemples

On peut consulter avec profit le diagramme circulaire des entiers sur 4 bits à la fin de la documentation décrivant le jeu d'instructions ARM.

Contenu binaire	Valeur entière représentée			
	si entier naturel	si entier relatif	naturel 12	relatif 12
0110	6	+6	0x006	0x006
0101	5	+5	0x005	0x005
1101	13	-3	0x00d	0xffd
1110	14	-2	0x00e	0xffe

Le bit de poids fort de la représentation en binaire d'un entier relatif est son bit de signe. Il est à un pour tous les entiers négatifs.

Tous les entiers multiples de 4 ont les deux bits de poids faible à 0.

Cette expression correspond à 8 * (x/8) soit le reste de la division entière de x par 8 (**i%8**). Elle est nulle si et seulement si x est multiple de 8 (les trois bits de poids faible de x sont à 0).

3.2 Binaire en C

L'algorithme examine chacun des bits de x du poids fort au poids faible. A chaque tour de boucle,

- la condition $x < 0$ est vraie si et seulement si le bit de poids fort de x est à 1 : on écrit le caractère '0' ou '1' correspondant,
- le bit de poids fort de x est remplacé par le bit de rang immédiatement inférieur par le décalage d'un bit à gauche.

Comme toute variable ou tableau déclaré statiquement et sans initialisateur, le tableau chaîne sera stocké dans bss et tous ses éléments seront intialisés à 0 au chargement du programme en mémoire.

La dernière affectation dans printbin a pour but de terminer la chaîne de caractères par un zéro, qui la marque de fin de chaîne en C. Sa suppression n'a pas d'impact ici puisque l'élément du tableau chaine qui suit le dernier caractère écrit par printbin est déjà à 0.

Il suffit d'initialiser les éléments du tableau chaîne à autre chose que 0 avant l'appel de sprintbin pour que la supression de $*p = 0$ donne un affichage erroné (incluant le contenu de msgerr). .

```
/* declaration avec initialisation statique */
```

```
char chaine [MAX] = "12345678901234567890123456789012345678901234567890";


char chaine [MAX];
...
/* ou initialisation dynamique dans le corps de main */
strcpy (chaine,"12345678901234567890123456789012345678901234567890");

/* avant appel de sprintbin */

@ affectation des registres
@ r0 : x
@ r1 : ecriture
@ r4 : p
@ r5 : i
@ r6 : temporaire valeur

            .text
.global main
@ Comme printbin n'appelle pas d'autre procédure on peut
@ omettre la gestion de sauvegarde de fp,sp, etc

printbin:   @ empiler registres modifiés : r4,r5,r6
            sub  sp, sp, #4
            str  r4, [sp]
            sub  sp, sp, #4
            str  r5, [sp]
            sub  sp, sp, #4
            str  r6, [sp]

            mov  r4, r1    @ p = ecriture
            mov  r5, #0    @ i = 0
            bal  test      @ goto test
corps:      cmp  r5, #0    @ if (x >= 0) goto sinon1
            bge  sinon1    @ x signé : test bge pour >=
alors1:     mov  r6, #'1'   @ '1' ou 0x31 ou 49
            strb r6, [r4]  @ *p = '1'
            @ Le add qui suit pourrait etre factorise dans le finsi1
            add  r4, r4, #1@ p++
            bal  finsi1    @  goto finsi1
sinon1:     mov  r6, #'0'   @ '0' ou 0x30 ou 48
            strb r6, [r4]  @ *p = '0'
            add  r4, r4, #1@ p++
finsi1:     mov  r0, r0, LSL #1  @ x << 1
            add  r5, r5, #1      @ i++
```

```
            mov  r6, r5, LSR #2  @ r6 = i/4
            cmp  r5, r6, LSL #2  @ comparer i et 4*(i/4)
            bne  finsi2          @ if ((i%4) ==0) goto sinon2
alors2:     mov  r6, #' '        @ *p = ' '
            strb r6, [r5]
            add  r6, r6, #1      @ p++
finsi2:
test:       cmp r5, #32          @ if i<32 goto corps
            bcc corps            @ i unsigned : bcc pour <

            mov r6, #0
            strb r6, [r4]        @ *p = 0

            @ désempiler registres
            ldr  r6, [sp]
            add  sp, sp, #4
            ldr  r5, [sp]
            add  sp, sp, #4
            ldr  r4, [sp]
            add  sp, sp, #4

            mov  pc, lr


main :      @ ...
            @ appel de sprintbin
            ldr  r6, =x     @ x dans r0 = entier
            ldr  r0, [r6]
            ldr  r1, =chaine @ ecriture dans r1 = chaine
            bl   sprintbin
```

Nous avons l'invariant de boucle **p = ecriture + i**. La condition devient donc **while (p - ecriture) < 32)**. D'autre part, utiliser la nullité de x comme critère d'arrêt est incorrect : les chiffres à zéro en poids faible ne seraient pas affichés.

```
            sub   r6, r4, r1
            subS  r6, #32
            bcc   corps
```