Logiciel de Base

Ensimag 1A Apprentissage

Examen — Juin 2010

Consignes :

- Durée : 2h.
- Tous documents autorisés.
- Le barème est donné à titre indicatif.
- On attend des réponses courtes et pertinentes, inutile de recopier le cours.
- Les parties sont indépendantes les unes des autres. La plupart des questions du problème sont également indépendantes. Pensez à lire le sujet en entier avant de commencer à répondre.

Consignes relatives à l'écriture de code C et assembleur Pentium :

- Pour chaque question, une partie des points sera affectée à la clarté du code et au respect des consignes ci-dessous.
- Pour les questions portant sur la traduction d'une fonction C en assembleur, on demande d'indiquer en commentaire chaque ligne du programme C original avant d'écrire les instructions assembleur correspondantes.
- Pour améliorer la lisibilité du code assembleur, on utilisera systématiquement des constantes (i.e. déclarations du type x=42) pour les déplacements relatifs à `%ebp` (i.e. paramètres des fonctions et variables locales). Par exemple, si une variable locale s'appelle `var` en langage C, on y fera référence avec `var(%ebp)`.
- Sauf indication contraire dans l'énoncé, on demande de traduire le code C en assembleur de façon systématique, sans chercher à faire la moindre optimisation : en particulier, **on stockera les variables locales dans la pile** (pas dans des registres), comme le fait le compilateur C par défaut.
- On respectera les conventions de gestions des registres Intel vues en cours, c'est à dire :
  - `%eax`, `%ecx` et `%edx` sont des registres scratch ;
  - `%ebx`, `%esi` et `%edi` ne sont pas des registres scratch.

1 Exercices sur le langage d'assemblage et GDB (2 points)

On considère le programme assembleur suivant :

```
.text
        .globl main
```

```
main:
        enter $0, $0

        movl $c1, %eax
debut_while:
        cmpl $0, %eax
        je fin_while

        movl 0(%eax), %ecx // ecx = eax->val
arret_ici:
        movl 4(%eax), %eax // eax = eax->suiv
        jmp debut_while
fin_while:

fin_main:
        xorl %eax, %eax
        leave
        ret

.data
c3:     .long 0x666
        .long c4
c1:     .long 0x42
        .long c2
c2:     .long 0x12
        .long c3
c4:     .long 0x1234
        .long 0
```

On assemble ce programme et on l'exécute dans GDB. Une trace incomplète est donnée ci-dessous :

```
(gdb) break arret_ici
Breakpoint 4 at 0x8048364: file liste.S, line 13.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: a.out

Breakpoint 4, arret_ici () at liste.S:13
// On affiche %eax en hexa.
(gdb) print /x $eax
$16 = 0x8049538
(gdb) print /x $ecx
$17 = 0x________
// On affiche les 8 mots longs suivant l'adresse 0x8049530
(gdb) x/8x 0x8049530
```

```
0x8049530: 0x00000666  0x________  0x00000042  0x08049540
0x8049540: 0x00000012  0x________  0x________  0x________
// On avance (exécution de movl 4(%eax), %eax)
(gdb) next
(gdb) print /x $eax
$18 = 0x________
```

Question 1 *6 valeurs ont été remplacées par des* `________`*. Donnez ces 6 valeurs, avec pour chacune une explication d'une ou deux phrases.*

```
(gdb) break arret_ici
Breakpoint 4 at 0x8048364: file liste.S, line 13.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: a.out

Breakpoint 4, arret_ici () at liste.S:13
(gdb) print /x $eax
$16 = 0x8049538
(gdb) print /x $ecx
$17 = 0x42
(gdb) x/8x 0x8049530
0x8049530 <c3>: 0x00000666 0x08049548 0x00000042 0x08049540
0x8049540 <c2>: 0x00000012 0x08049530 0x00001234 0x00000000
(gdb) next
(gdb) p /x $eax
$18 = 0x8049540
```

2 Implémentation d'une fonction simple en assembleur (1 point)

Soit la fonction C suivante :

```
void incrementer(int * i) {
        (*i) += 1;
}
```

Question 2 *Traduire cette fonction en assembleur.*

```
//void incrementer(int * i) {
incrementer:
```

```
i=8
        pushl %ebp
        movl %esp, %ebp
//      (*i) += 1;
        movl 8(ebp), %eax
        addl $1, (%eax)
//}
        leave
        ret
```

3 Liste chaînée (1,5 points)

Soit une liste chaînée définie par :

```
struct cellule {
        int val;
        struct cellule *suiv;
};
```

Question 3 *Écrire une fonction* `incrementer_liste(...)` *(la plus simple possible) qui incrémente chaque élément d'une liste (i.e. qui leur ajoute 1).*

```
// Passage par valeur :
// on modifie les elements, pas les pointeurs.
void incrementer_liste(struct cellule *l)
{
        while (l != NULL) {
                l->val += 1;
                l = l->suiv;
        }
}
```

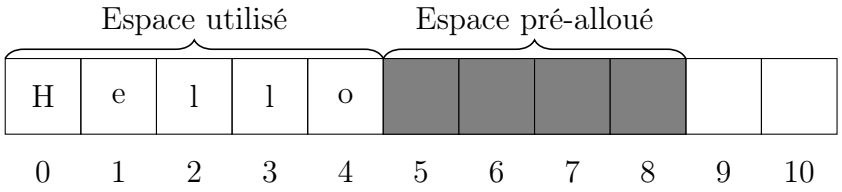
4 Problème : Implémentation d'un type « string » intelligent (16 points)

En C, il n'y a pas à proprement parler de type « string », on utilise à la place le type « char * ». On cherche dans cet exercice à implémenter un type « string » pour représenter des chaînes de caractères, de manière plus intelligente. Le type est défini comme ceci :

```
struct string {
        /* Les caracteres de la chaine */
        char *elems;
```

```
        /* Taille du tableau alloue pour elems */
        size_t alloc_size;
        /* Nombre de caracteres utilises dans elems */
        size_t used_size;
};
```

Le principe est d'avoir un tableau pré-alloué, d'une taille supérieure ou égale à la taille du tableau réellement utilisé. L'intérêt est que pour ajouter des caractères en fin de chaîne, on pourra utiliser directement l'espace pré-alloué s'il est suffisant, et éviter d'avoir à appeler systématiquement `malloc`. Ci-dessous, un exemple où le champ `elem` contient le tableau `Hello`, avec `used_size == 5` et `alloc_size == 9`. On pourra ici ajouter 4 caractères en fin de chaîne sans refaire d'allocation, et il faudra ré-allouer le tableau si on veut ajouter un 5ème caractère.



Quand la place n'est plus suffisante, on crée un tableau plus grand avec `malloc`, on recopie l'ancien tableau dedans avec `memcpy`. Pour rappel, `man memcpy` dit :

```
void *memcpy(void *dest, const void *src, size_t n);
// The memcpy() function copies n bytes
// from memory area src to memory area dest.
```

Pour créer une string, on pré-alloue par défaut 5 caractères :

```
void init_string (struct string *v)
{
        v->alloc_size = 5;
        v->elems = malloc(5);
        v->used_size = 0;
};
```

Question 4 *Traduire cette fonction en langage d'assemblage.*

```
// void init_string (struct string *v)
// {
init_string:
        pushl %ebp
        movl %esp, %ebp
        subl $(4 + 0), %esp
        andl $-16, %esp
v=8
elems = 0
```

```
alloc_size = 4
used_size = 8

//      v->alloc_size = 5;
        movl v(%ebp), %eax
        movl $5, alloc_size(%eax)
//      v->elems = malloc(5);
        movl $5, 0(%esp)
        call malloc
        movl v(%esp), %ecx
        movl %eax, elems(%ecx)
//      v->used_size = 0;
        movl v(%esp), %ecx
        movl $0, used_size(%ecx)
// };
        leave
        ret
```

On donne le code de la fonction `delete_string` (il n'est pas demandé de la traduire) :

```
void delete_string (struct string *v)
{
        free(v->elems);
        v->elems = NULL;
        v->alloc_size = 0;
        v->used_size = 0;
}
```

On va maintenant implémenter la fonction `push_back` qui ajoute un élément en fin de chaîne. Une implémentation de cette fonction en langage d'assemblage est :

```
.text
        .globl push_back_asm
push_back_asm:
        pushl %ebp
        movl %esp, %ebp
        subl $(4 + 8), %esp
        andl $-16, %esp
elems = 0
alloc_size = 4
used_size = 8

str = 8
elem = 12
new_alloc_size = -4

        //
```

```
        movl str(%ebp), %eax
        movl alloc_size(%eax), %eax
        movl str(%ebp), %ecx
        movl used_size(%ecx), %ecx
        cmpl %eax, %ecx
        jne else
        //
        movl str(%ebp), %eax
        movl alloc_size(%eax), %eax
        shl %eax // multiplier eax par 2
        addl $1, %eax
        movl %eax, new_alloc_size(%ebp)
        //
        movl str(%ebp), %eax
        movl %eax, 0(%esp)
        movl new_alloc_size(%ebp), %eax
        movl %eax, 4(%esp)
        call realloc_string
        //
else:
        //
        movl str(%ebp), %eax
        movl elems(%eax), %ecx
        movl used_size(%eax), %edx
        movl elem(%ebp), %eax
        /* parametre sur 8 bits passe sur 32 bits :
           On a empile des 0 sur les bits de poids fort. */
        movb %al, (%ecx, %edx)
        //
        movl str(%ebp), %eax
        addl $1, used_size(%eax)

        leave
        ret
```

(ce code a été écrit à partir d'une fonction C, l'auteur a laissé les `//` qui délimitent les lignes de C traduites pour vous aider. Si ces `//` vous perturbent, vous pouvez les ignorer)

La fonction utilise une fonction `realloc_string` implémentée en C comme ceci :

```
void realloc_string (struct string *str, size_t new_alloc_size)
{
        char *new_elems;
        new_elems = malloc(new_alloc_size);
        memcpy(new_elems, str->elems, str->alloc_size);
        str->alloc_size = new_alloc_size;
        free(str->elems);
        str->elems = new_elems;
```

```
}
```

Question 5 *Dessinez la pile pendant un appel de* `push_back_asm`*.*

Question 6 *Traduire la fonction* `push_back_asm` *en une fonction* `push_back()` *en langage C.*

```
void push_back (struct string *str, char elem)
{
        size_t new_alloc_size;
        if (str->alloc_size == str->used_size) {
                new_alloc_size = (str->alloc_size << 1) + 1;
                realloc_string(str, new_alloc_size);
        }
        str->elems[str->used_size] = elem;
        str->used_size++;
}
```

Question 7 *Traduire la fonction* `realloc_string` *en langage d'assemblage.*

```
// void realloc_string (struct string *str, size_t new_alloc_size)
// {
realloc_string:
        push %ebp
        movl %esp, %ebp
//      char *new_elems;
        subl $(4+12), %esp
        andl $-16, %esp
//      new_elems = malloc(new_alloc_size);
        movl new_alloc_size(%ebp), %eax
        movl %eax, 0(%esp)
        call malloc
        movl %eax, new_elems(%ebp)
//      memcpy(new_elems, str->elems, str->alloc_size);
        movl new_elems(%ebp), %eax
        movl %eax, 0(%esp)
        movl str(%ebp), %eax
        movl elems(%eax), %ecx
        movl %ecx, 4(%esp)
        movl alloc_size(%eax), %ecx
        movl %ecx, 8(%esp)
//      str->alloc_size = new_alloc_size;
        movl new_alloc_size(%ebp), %eax
        movl str(%ebp), %ecx
```

```
        movl %eax, alloc_size(%ecx)
//      free(str->elems);
        movl str(%ebp), %eax
        movl elem(%eax), %eax
        movl %eax, 0(%esp)
        call free
//      str->elems = new_elems;
        movl new_elems(%ebp), %eax
        movl str(%ebp), %ecx
        movl %eax, elem(%ecx)
// }
        leave
        ret
```

Question 8 *Écrire une fonction* `main` *qui crée une structure de type « string », l'initialise, ajoute les caractères 'a' puis 'b' en fin de chaîne, puis détruit la « string ». Le code utilisera la pile autant que possible (i.e. évitera les* `malloc` *inutiles).*

```
int main(void)
{
        struct string v1;
        init_string(&v1);
        push_back(&v1, 'a');
        push_back(&v1, 'b');
        delete_string(&v1);
}
```

On va maintenant implémenter la fonction

```
void add_end (struct string *str, const char *to_add);
```

qui ajoute une chaîne de caractère C en fin de string. Un exemple d'utilisation serait :

```
        struct string hello;
        init_string(&hello);
        add_end(&hello, "Hello␣");
        add_end(&hello, "world!");
```

La stratégie de réallocation est la suivante :

- Si l'espace pré-alloué est suffisant pour contenir les caractères de `to_add`, on ne réalloue rien.
- Sinon, on réalloue un tableau de taille :
  `MAX(new_size_min, (str->alloc_size * 2) + 1)`
  `new_size_min` est la taille minimale pour contenir les deux chaînes concaténées, et le MAX permet éventuellement de pré-allouer un tableau un peu plus grand que nécessaire (ce qui permet d'économiser de futures allocations).

Question 9 *Implémentez la fonction* `add_end` *en langage C. Il faudra définir la macro* `MAX(a, b)` *proprement, et l'utiliser.*

```
#define MAX(a, b) ((a) >= (b) ? (a) : (b))

void add_end (struct string *str, const char *to_add)
{
        size_t new_alloc_size;
        size_t new_size_min = str->used_size + strlen(to_add);
        if (new_size_min > str->alloc_size) {
                new_alloc_size = MAX(new_size_min, (str->alloc_size * 2
                realloc_string(str, new_alloc_size);
        }
        memcpy(str->elems + str->used_size, to_add, strlen(to_add));
        str->used_size += strlen(to_add);
}
```

On va maintenant implémenter la fonction

```
void concat(const struct string s1,
            const struct string s2,
            struct string *result);
```

qui prend en argument deux strings s1 et s2 et renvoie leur concaténation dans `result`.

Question 10 *Implémentez cette fonction en langage C. On utilisera les fonctions C standard* `malloc` *et* `memcpy`*, mais aucun autre appel de fonction.*

```
void concat(const struct string s1, const struct string s2,
            struct string *result)
{
        result->alloc_size = s1.used_size + s2.used_size;
        result->used_size = result->alloc_size;
        result->elems = malloc(result->alloc_size);
        memcpy(result->elems, s1.elems, s1.used_size);
        memcpy(result->elems + s1.used_size, s2.elems, s2.used_size);
}
```

On considère maintenant la fonction suivante :

```
ma_fonction:
i=-4
        pushl %ebp
        movl %esp, %ebp
        subl $(4 + 8), %esp
        andl $-16, %esp
```

```
str=8
        //
        movl $0, i(%ebp)
        movl $0, %ecx
debut:
        //
        movl 16(%ebp), %ecx
        cmpl %ecx, i(%ebp)
        jae fin
        //
        movl $fmt_c, 0(%esp)
        movl 8(%ebp), %ecx
        movl i(%ebp), %eax
        movb (%eax, %ecx), %cl
        movl %ecx, 4(%esp)
        call printf
        //
        addl $1, i(%ebp)
        //
        jmp debut
fin:
        //
        movl $fmt_n, 0(%esp)
        call printf
        //
        leave
        ret
.data
fmt_c:  .asciz "%c"
fmt_n:  .asciz "\n"
```

Question 11 *Que fait cette fonction ? Traduisez-la en langage C.*

Cette fonction affiche le contenu de la string, caractère par caractère.

```
void print_string(struct string str) {
        for (unsigned i = 0; i < str.used_size; i++) {
                printf("%c", str.elems[i]);
        }
        printf("\n");
}
```

Question 12 *Écrire une variante de cette fonction, toujours en assembleur, où la variable* `i` *n'est pas stockée dans la pile, mais dans un registre.*

Le plus simple est d'utiliser un registre non-scratch, pour ne pas avoir de problème avec printf. On choisit ebx.

```
        // void print_string(struct string str) {
        .globl print_string_asm_opt
print_string_asm_opt:
i=-4
        pushl %ebp
        movl %esp, %ebp
        pushl %ebx
        subl $(0 + 8), %esp
        andl $-16, %esp

str=8
        // for (unsigned i = 0;
        movl $0, %ebx
        movl $0, %ecx
debut2:
        // i < str.used_size;
        movl 16(%ebp), %ecx
        cmpl %ecx, %ebx
        jae fin2
        //      printf("%c", str.elems[i]);
        movl $fmt_c, 0(%esp)
        movl 8(%ebp), %ecx
        movl %ebx, %eax
        movb (%eax, %ecx), %cl
        movl %ecx, 4(%esp)
        call printf
        // i++) {
        addl $1, %ebx
        // }
        jmp debut2
fin2:
        // printf("\n");
        movl $fmt_n, 0(%esp)
        call printf
        // }
        movl 4(%ebp), %ebx
        leave
        ret
```