

Logiciel de Base : examen de première session

ENSIMAG 1A

Année scolaire 2008–2009

Consignes générales :

- Durée : 2h. Tous documents et calculatrices autorisés.
- Le barème est donné à titre indicatif.
- Les exercices sont indépendants et peuvent être traités dans le désordre.
- **Merci d’indiquer votre numéro de groupe de TD et de rendre votre copie dans le tas correspondant à votre groupe.**

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 (directive `.equ`) 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*.

Ex. 1 : Parcours de tableaux (12 pts)

Le Crible d’Ératosthène¹ est un algorithme itératif permettant de générer les nombres premiers inférieurs ou égaux à un entier N donné.

On travaille ici sur une version de cet algorithme utilisant des tableaux. Plus précisément, l’algorithme consiste à :

¹Ératosthène était un astronome, géographe, philosophe et mathématicien grec du III^e siècle avant J.-C.

- initialiser un tableau de booléens allant de 0 à N (inclus) avec la valeur *Faux* pour les cases 0 et 1 (qui ne sont pas des nombres premiers) et *Vrai* pour les autres ;
- parcourir ce tableau pour marquer avec la valeur *Faux* toutes les cases dont l'indice est un multiple d'un entier inférieur : on démontre qu'on peut arrêter le parcours dès qu'on a supprimé tous les multiples des entiers inférieurs ou égaux à \sqrt{N} ;
- afficher les indices des cases contenant la valeur *Vrai*, car les indices en question sont tous des nombres premiers.

Le programme principal de cet algorithme s'écrit donc en C (où N est un naturel supérieur ou égal à 2) :

```
int main()
{
    unsigned int tab[N + 1]; /* unsigned int est equivalent a unsigned */
    remplir(tab, N);
    supprimer_multiples(tab, N);
    afficher(tab, N);
    return 0;
}
```

Question 1 Ecrire en C la fonction void remplir(unsigned int tab[], unsigned int sup) qui initialise le tableau tab comme expliqué plus haut (on rappelle qu'en C il n'existe pas de type booléen : on utilisera donc 0 pour la valeur *Faux* et 1 pour la valeur *Vrai*).

```
void remplir(unsigned int tab[], unsigned int sup)
{
    unsigned int i;
    tab[0] = tab[1] = 0;
    for (i = 2; i <= sup; i++)
        tab[i] = 1;
}
```

Question 2 Traduire cette fonction remplir en assembleur. On rappelle qu'il est demandé de précéder chaque séquence d'instructions assembleur par la ligne de C correspondante, et d'utiliser des constantes pour les déplacements relatifs à %ebp.

```
/* void remplir(unsigned int tab[], unsigned int sup) */
.equ tab, 8
.equ sup, 12
remplir:
    /* unsigned int i; */
    .equ i, -4
    enter $4, $0
```

```

/* tab[0] = tab[1] = 0; */
movl tab(%ebp), %eax
movl $0, (%eax)
movl $0, 4(%eax)
/* for (i = 2; ... */
movl $2, i(%ebp)
remplir_bcl:
/* ... i <= sup; ... */
movl i(%ebp), %eax
cmpl sup(%ebp), %eax
ja remplir_bcl_fin
/* tab[i] = 1; */
movl tab(%ebp), %eax
movl i(%ebp), %edx
movl $1, (%eax, %edx, 4)
/* ... i ++ */
incl i(%ebp)
jmp remplir_bcl
remplir_bcl_fin:
leave
ret

```

Question 3 Traduire en C le code de la fonction `supprimer_multiples` donnée ci-dessous en assembleur. On conservera les mêmes noms de paramètres et de variables locales lors de la traduction. On rappelle quelques instructions utiles pour cette question :

- `enter $8, $0` : cette instruction est exactement équivalente à la séquence d'instructions :

```

pushl %ebp
movl %esp, %ebp
subl $8, %esp

```

- `mull x` : multiplie le contenu de la variable 32 bits `x` par le contenu du registre `%eax`, et stocke le résultat (sur 64 bits) dans `%edx:%eax`;
- `shll %eax` : décale le contenu du registre `%eax` d'un bit vers la gauche (ce qui revient en pratique à le multiplier par 2).

```

/* void supprimer_multiples(unsigned int tab[], unsigned int sup) */
.equ tab, 8
.equ sup, 12
supprimer_multiples:
/* unsigned int i, j */
.equ i, -4
.equ j, -8
enter $8, $0

```

```

    movl $2, i(%ebp)
sm_bcl1:
    movl i(%ebp), %eax
    mull i(%ebp)
    cmpl sup(%ebp), %eax
    ja sm_bcl1_fin
    movl i(%ebp), %eax
    shll %eax
    movl %eax, j(%ebp)
sm_bcl2:
    movl j(%ebp), %eax
    cmpl sup(%ebp), %eax
    ja sm_bcl2_fin
    movl tab(%ebp), %eax
    movl j(%ebp), %edx
    movl $0, (%eax, %edx, 4)
    movl i(%ebp), %eax
    addl %eax, j(%ebp)
    jmp sm_bcl2
sm_bcl2_fin:
    incl i(%ebp)
    jmp sm_bcl1
sm_bcl1_fin:
    leave
    ret

```

```

void supprimer_multiples(unsigned int tab[], unsigned int sup)
{
    unsigned int i, j;
    for (i = 2; i * i <= sup; i++)
        for (j = i * 2; j <= sup; j += i)
            tab[j] = 0;
}

```

Question 4 Ecrire en C la fonction `void afficher(unsigned int tab[], unsigned int sup)` qui affiche à l'écran les nombres premiers en fonction du contenu du tableau `tab`, comme expliqué plus haut.

```

void afficher(unsigned int tab[], unsigned int sup)
{
    unsigned int i;
    for (i = 0; i <= sup; i++)

```

```

    if (tab[i] == 1)
        printf("%u ", i);
    printf("\n");
}

```

Ex. 2 : Optimisation de code (8 pts)

On travaille dans cet exercice sur différentes implantations d'une fonction permettant de copier le contenu d'une zone mémoire dans une autre. Cette fonction existe dans la bibliothèque C standard :

```

/*
 * Recopie les n premiers octets de la zone memoire pointee par src
 * dans la zone memoire pointee par dst et renvoie l'adresse dst.
 */
char *memcpy(char *dst, char *src, unsigned int n)

```

Une implantation simpliste de cette fonction en C pourrait être :

```

char *copie_memoire(char *dst, char *src, unsigned int n)
{
    unsigned int i;
    for (i = 0; i < n; i++)
        dst[i] = src[i];
    return dst;
}

```

Question 1 Traduire cette version simpliste en assembleur. On traduira de façon systématique sans chercher à faire la moindre optimisation. Notamment, les variables locales seront stockées dans la pile et pas dans des registres.

```

/* char *copie_memoire_naif(char *dst, char *src, unsigned int n); */
.equ dst, 8
.equ src, 12
.equ n, 16
copie_memoire_naif:
    /* unsigned int i; */
    .equ i, -4
    enter $4, $0
    /* for (i = 0; ... */
    movl $0, i(%ebp)
cpn_bcl:
    /* ...; i < n; ... */

```

```

movl i(%ebp), %eax
cmpl n(%ebp), %eax
jae cpn_bcl_fin
/* dst[i] = src[i] */
movl i(%ebp), %eax
movl src(%ebp), %ecx
movl dst(%ebp), %edx
movb (%ecx, %eax), %cl
movb %cl, (%edx, %eax)
/* ... i ++ */
incl i(%ebp)
jmp cpn_bcl
cpn_bcl_fin:
/* return dst; */
movl dst(%ebp), %eax
leave
ret

```

Question 2 Ecrire une nouvelle version assembleur de cette même fonction, mais en stockant au début du programme les variables locales et les paramètres dans des registres, pour supprimer dans la suite du code autant d'accès mémoire que possible. On stockera src dans %esi, dst dans %edi, n dans %edx et i dans %ecx.

```

.equ dst, 8
.equ src, 12
.equ n, 16
copie_memoire_optimisee:
enter $0, $0
pushl %esi
pushl %edi
movl src(%ebp), %esi
movl dst(%ebp), %edi
movl n(%ebp), %edx
xorl %ecx, %ecx
cpo1_bcl:
cmpl %edx, %ecx
jae cpo1_bcl_fin
movb (%esi, %ecx), %al
movb %al, (%edi, %ecx)
incl %ecx
jmp cpo1_bcl
cpo1_bcl_fin:
movl dst(%ebp), %eax

```

```

popl %edi
popl %esi
leave
ret

```

En pratique, l'implantation de `memcpy` fournie par la bibliothèque C standard est plus rapide que les deux versions implantées dans les questions précédentes. On peut effectivement optimiser le code en utilisant des instructions dédiées du processeur Pentium :

- `shrl $X, %reg` : décale de `X` bits vers la droite le registre `reg` ;
- `movsd` : recopie le mot de 32 bits pointé par `%esi` dans la case mémoire pointée par `%edi` et ajoute ou soustrait 4 à `%esi` et `%edi` (le choix entre l'addition ou la soustraction se fait en fonction du bit `DF` du registre `%eflags`) ;
- `cld` : met à zéro le bit `DF` du registre `%eflags`, ce qui a pour effet en pratique de sélectionner l'addition de 4 à `%esi` et `%edi` pour l'instruction `movsd` ;
- `rep` : préfixe ajouté avant certaines instructions (*e.g.* `rep movsd`) et qui a pour effet de répéter l'exécution de l'instruction préfixée tant que `%ecx` est différent de 0, et qui soustrait automatiquement 1 à `%ecx` après chaque exécution de l'instruction.

Question 3 En utilisant les instructions présentées ci-dessus, donner une version assembleur optimisée de la fonction de copie mémoire. On supposera pour simplifier que la taille de la zone mémoire à recopier est un multiple de 4.

```

/* Suppose que le nombre d'octets a copier est un multiple de 4 */
.equ dst, 8
.equ src, 12
.equ n, 16
copie_memoire_movs:
    enter $0, $0
    pushl %esi
    pushl %edi
    movl dst(%ebp), %edi
    movl src(%ebp), %esi
    movl n(%ebp), %ecx
    shrl $2, %ecx
    cld
    rep movsd
    movl dst(%ebp), %eax
    popl %edi
    popl %esi
    leave
    ret

```

Pour les curieux, un test simpliste effectué sur *telesun* (recopie d'une zone mémoire de 1468006400 octets) donne les résultats suivants : version naïve en 5,98 sec., version optimisée en 1,93 sec., version avec les instructions dédiées et *memcpy* de la bibliothèque C toutes deux en 1,80 sec. En pratique, on copie rarement de très gros blocs de mémoire, mais plus souvent de petits blocs de façon répétitives. L'implantation de *memcpy* de la bibliothèque C est alors très performante grâce à la prise en compte des caractéristiques du pipeline et de la mémoire cache de la machine.

Comme suggéré ci-dessus, l'implémentation de *memcpy* dans la libc GNU sur plateforme Intel est encore plus optimisée que la version avec les instructions *movsd*, etc. La version disponible sur *telesun* dans `libc/sysdeps/i386/i586/memcpy.h` est principalement constituée de :

```

/* Written like this, the Pentium pipeline can execute the loop at a
   sustained rate of 2 instructions/clock, or asymptotically 480
   Mbytes/second at 60Mhz. */

#undef WORD_COPY_FWD
#define WORD_COPY_FWD(dst_bp, src_bp, nbytes_left, nbytes) \
do \
{ \
asm volatile ("subl $32,%2\n" \
"js 2f\n" \
"movl 0(%0),%%edx\n" /* alloc dest line */ \
"1:\n" \
"movl 28(%0),%%eax\n" /* alloc dest line */ \
"subl $32,%2\n" /* decr loop count */ \
"movl 0(%1),%%eax\n" /* U pipe */ \
"movl 4(%1),%%edx\n" /* V pipe */ \
"movl %%eax,0(%0)\n" /* U pipe */ \
"movl %%edx,4(%0)\n" /* V pipe */ \
"movl 8(%1),%%eax\n" \
"movl 12(%1),%%edx\n" \
"movl %%eax,8(%0)\n" \
"movl %%edx,12(%0)\n" \
"movl 16(%1),%%eax\n" \
"movl 20(%1),%%edx\n" \
"movl %%eax,16(%0)\n" \
"movl %%edx,20(%0)\n" \
"movl 24(%1),%%eax\n" \
"movl 28(%1),%%edx\n" \
"movl %%eax,24(%0)\n" \
"movl %%edx,28(%0)\n" \
"leal 32(%1),%1\n" /* update src ptr */ \
"leal 32(%0),%0\n" /* update dst ptr */ \
"jns 1b\n" \
"2: addl $32,%2" : \

```

```
"=r" (dst_bp), "=r" (src_bp), "=r" (nbytes_left) : \  
"0" (dst_bp), "1" (src_bp), "2" (nbytes) : \  
"ax", "dx"); \  
} while (0)
```

mais inclue aussi beaucoup d'optimisations pour les zones mémoires plus petites (cas particulier pour les cases non-alignées, ...).