LISV Laboratoire d'ingénierie des systèmes de Versailles | UVSQ université PARIS-SACLAY

Architecture des ordinateurs
TP 3 - CPU 80386

Halim Djerroud

révision 1.0

Exercice 1 :

1. Créer un fichier "printf_example.s" et copier le code suivant :

```
    .data
str1:   .asciz      "Hello Mon premier programme en assembleur \n"
str2:   .asciz      "La valeur de eax = %d \n"

    .text
    .global main

main:
        pushl   %ebp
        movl    %esp,  %ebp

        /* Imprimer str1 en utilisant la fonction printf de la libc */
        pushl   $str1
        call    printf

        /* Utiliser printf avec un parametre  */
        movl    $42, %eax
        pushl   %eax
        pushl   $str2
        call    printf
        addl    $4, %esp
        leave
        ret
```

   Compilez et exécuter le programme, si vous utilisez gcc en ligne de commande au lieu de nasm, ci-après les commandes pour compiler votre programme en 32 bits :
   `gcc -m32 -no-pie printf_example.s -o printf_example`
   Attention : l'option -m32 permet d'utiliser l'architecture 32 bits sur une machine 64 bits. Pour exécuter votre programme :
   `./printf_example`

2. Écrire un programme en assembleur qui affiche à l'écran un le message suivant : "L'assembleur est cool ! " en utilisant la fonction de la librairie standard `printf`.

3. Écrire une programme en assembleur qui affiche les nombre de 1 à 10
   :

```
.data
str:    .asciz "--> %d \n"
.text
.global main
main:
    movl %esp, %ebp #for correct debugging
    movl    $1,    %eax
```

```
loop:
    cmpl    $10,    %eax
    jg      end_loop
    pushl   %eax
    pushl   $str
    call    printf
    popl    %eax
    popl    %eax
    addl    $1,     %eax
    jmp     loop
end_loop:
    xorl  %eax, %eax
    ret
```

4. Écrire une programme en assembleur qui affiche les nombres paires de 1 à 20
   Solution :

```
.data
str: .asciz "%d\n"
x: .int 0
y: .int 2
.bss
.text
.global main
main:
    movl %esp, %ebp
    xorl  %eax, %eax
    movl $1, %eax
    movl $20, %ebx
for:
    movl $0, %edx
    movl %eax, x
    idiv y
    movl x, %eax
    cmp $0, %edx
    je paire
    inc %eax
    jmp for
paire:
    pushl %eax
    pushl %ebx
    pushl %eax
    pushl $str
    call printf
    addl $8, %esp
    popl %ebx
    popl %eax
    cmp $20, %eax
    je end
    inc %eax
    jmp for
end:
    ret

### Solution 2
.data
str:    .asciz "val = %d\n"
.text
```

```
.global main
main:
    movl    %esp, %ebp
    movl    $0,    %ecx
debut_boucle:
    incl    %ecx
    cmpl    $10,    %ecx
    jg      fin_boucle
    movl    %ecx,   %eax
    andl    $1,     %eax
    jnz     debut_boucle
    pushl   %ecx
    pushl   $str
    call    printf
    popl    %ecx
    popl    %ecx

    jmp     debut_boucle
fin_boucle:
    xorl    %eax, %eax
    ret
```

5. Écrire une programme en assembleur qui affiche les nombres premiers entre 2 à 100 :

```
.data
str: .asciz "%d \n"
premier: .int 1
x: .int 0
.bss
.text
.global main
main:
    movl %esp, %ebp
    movl $1, %ebx
    movl $2, %eax
for:
    movl $1, premier
    cmpl $100, %eax
    jg end
    movl $1, %ebx
    for2:
        incl %ebx
        cmpl %eax, %ebx
        jg for_suite
        movl %eax,x
        movl $0, %edx
        idiv %ebx
        movl x, %eax
        cmpl $0, %edx
        jne for2
        cmpl %eax, %ebx
        je for2
        movl $0, premier
        jmp for2
for_suite:
    cmpl $0, premier
    jne not_equal
    addl $1, %eax
```

```
    jmp for
not_equal:
    pushl %eax
    pushl $str
    call printf
    popl %eax
    popl %eax
    addl $1, %eax
    jmp for
end:
    xorl  %eax, %eax
    ret
```

Exercice 2 : Les sous-programmes (les fonctions)

1. Reprendre le dernier exercice de la partie précédente. Séparer votre programme en deux parties :
   (a) Une fonction qui prend en paramètre un entier et retourne en (0 ou 1) pour indiquer si l'entier donné est premier ou pas
   (b) une seconde fonction qui parcoure tous les entiers impairs de (3 à 199) et affiche uniquement les entiers premier, en utilisant la fonction précédente

   Solution :

```
int prime(int x){
  for (i=2; i<x ; i++){
    if(x%i == 0)
      return 0;
  }
  return 1;
}
```

```
.data
str:     .asciz  "--> %d \n"
.text
.global main

prim:
    pushl   %ebp
    movl    %esp,    %ebp
    movl    8(%ebp), %ebx
    movl    $2, %ecx

loop:
    cmpl    %ebx,   %ecx
    je      end_loop_prime
    xorl    %edx,   %edx
    movl    %ebx,   %eax
    divl    %ecx
    cmpl    $0,     %edx
    jz      end_loop_not_prime
    incl    %ecx
    jmp     loop
end_loop_not_prime:
    xorl    %eax,   %eax
    jmp quit
end_loop_prime:
    movl    $1,   %eax
quit:
```

```
    popl    %ebp
    ret

    /************************/
    /*         MAIN         */
    /************************/
main:
    movl %esp, %ebp
    movl    $3, %ecx
loop_main:
    cmpl    $199,   %ecx
    jg      loop_main_end
    pushl   %ecx
    call    prim
    popl    %ecx

    cmpl    $0, %eax
    je      after_print
    pushl   %ecx
    pushl   $str
    call    printf
    popl    %ecx
    popl    %ecx
 after_print:
    incl    %ecx
    jmp     loop_main

loop_main_end:
    ret
```

Exercice 3 : Les modes d'adressage et les tableaux

Les tableau :

1. Écrire une fonction `print_tab` qui prend en paramètre l'adresse d'un tableau d'entiers et sa taille et affiche l'ensemble de ses éléments sous la forme suivante : $[e_1, e_2, ..., e_n]$

   Solution :

```
#include <stdio.h>
#define N 5
int tab[N]={5,4,6,7,8};

void print_tab(int* t, int size){
  printf("[");
  for(int i =0 ; i < size; i++){
    printf("%d,", t[i]);
  }
  printf("]\n");
}

int main(){
  print_tab(tab, N);
  return 0;
}
```

```
.data
str:    .asciz  "%d,"
.align   4
```

```
tab:    .int 65,12,45,16,24,88,79
size:   .int 7

.text

print_tab:
    pushl   %ebp                # prologue
    movl    %esp,   %ebp

    movl    8(%ebp), %ebx
    movl    12(%ebp),%edx
    pushl   %ebx                # sauvegarder %ebx convention GCC
    xorl    %ecx,   %ecx        # mise à 0 du coupteur
boucle:
    cmpl    %edx,   %ecx
    jge     fin_boucle
    pushl   %edx
    pushl   %ecx
    pushl   (%ebx,%ecx,4)       # %ecx * 4 + %ebx
    pushl   $str
    call    printf
    addl    $8, %esp
    popl    %ecx
    popl    %edx
    incl    %ecx
    jmp     boucle
fin_boucle:
                                # épilogue
    popl    %ebx                # restaurer %ebx pour la convention GCC
    popl    %ebp                # restaurer %ebp
    ret

.global main
main:
    movl %esp, %ebp #for correct debugging

    pushl   size
    pushl   $tab
    call    print_tab
    addl    $8,     %esp

    xorl    %eax, %eax
    ret
```

2. Écrire une fonction `read_tab` qui prend en paramètre l'adresse d'un tableau d'entiers et sa taille et demande à l'utilisateur de saisir les valeurs (astuce : utilisez `printf` et `scanf`)

   Solution :

```
#include <stdio.h>
#define N 5
int tab[N];

void read_tab(int* t, int size){
  printf("saisir les valeurs du tableau : \n");
  for(int i =0 ; i < size; i++){
    scanf("%d,", &t[i]);
  }
}
```

```c
void print_tab(int* t, int size){
  printf("[");
  for(int i =0 ; i < size; i++){
    printf("%d,", t[i]);
  }
  printf("]\n");
}

int main(){
  read_tab(tab,N);
  print_tab(tab, N);
  return 0;
}
```

```
.data
str_scanf:  .asciz "%d"
str_read:   .asciz "Saisir les veleurs du tableau : \n"
str:        .asciz "%d \n"
size:       .int 4

.bss
        .lcomm tab, 20

.text
.global main

/*****************/
/*   read_tab    */
/*****************/
read_tab:
    #prologue
    pushl   %ebp
    movl    %esp,   %ebp
    movl    12(%ebp), %edi
    movl     8(%ebp), %esi

    pushl   $str_read
    call    printf
    addl    $4,     %esp

    movl    $0      , %ecx
loop_read:
    cmpl    %ecx,     %edi
    jle     end_loop_read
    pushl   %ecx
    leal   (%esi, %ecx, 4), %ebx
    pushl   %ebx
    pushl   $str_scanf
    call    scanf
    addl    $8,     %esp
    popl    %ecx
    incl    %ecx
    jmp loop_read
end_loop_read:

    # épilogue
    popl    %ebp
    xorl    %eax,   %eax    # return 0
```

```
    ret

/******************/
/*   print_tab    */
/******************/
print_tab:
    pushl   %ebp
    movl    %esp,   %ebp
    movl    12(%ebp), %edi
    movl     8(%ebp), %esi
    movl    $0      , %ecx
loop:
    cmpl    %ecx,     %edi
    jle     end_loop
    pushl   %ecx
    pushl   (%esi, %ecx, 4)
    pushl   $str
    call    printf
    addl    $8,     %esp
    popl    %ecx
    incl    %ecx
    jmp loop
end_loop:
    xorl    %eax,   %eax    # return 0
    pop     %ebp
    ret

/******************/
/*      main      */
/******************/
main:
    movl %esp, %ebp

    #appel de la fonction read_tab avec les deux params
    pushl size
    pushl $tab
    call  read_tab
    addl  $8,     %esp

    #appel de la fonction print_tab avec les deux params
    pushl size
    pushl $tab
    call print_tab
    addl    $8,     %esp

    xorl  %eax, %eax
    ret
```

3. Écrire une fonction `sum_tab` qui prend en paramètre l'adresse d'un tableau d'entiers et sa taille et retourne la somme des ses éléments

   Solution :

```
/******************/
/*    sum_tab     */
/******************/
sum_tab:
    pushl   %ebp
```

```
    movl    %esp,   %ebp
    movl    12(%ebp), %edi
    movl     8(%ebp), %esi

    xorl    %eax,   %eax
    xorl    %ecx,   %ecx
loop_sum:
    cmpl    %ecx,     %edi
    jle     end_loop_sum

    addl    (%esi, %ecx,4), %eax

    incl    %ecx
    jmp loop_sum
end_loop_sum:
    pop     %ebp
    ret


/*****************/
/*      main     */
/*****************/
main:
    movl %esp, %ebp

    #appel de la fonction read_tab avec les deux params
    pushl size
    pushl $tab
    call  read_tab
    addl  $8,     %esp

    #appel de la fonction print_tab avec les deux params
    pushl size
    pushl $tab
    call print_tab
    addl    $8,     %esp

    #appel de la fonction sum_tab avec les deux params
    pushl size
    pushl $tab
    call  sum_tab
    addl  $8,     %esp

    #afficher la somme
    pushl %eax
    pushl $str_sum
    call  printf
    addl  $8,     %esp

    xorl  %eax, %eax
    ret
```

4. Écrire une fonction qui prend en paramètre l'adresse d'un tableau d'entiers et sa taille et retourne la position du plus petit élément

   Solution :

   ```
   todo
   ```

5. Écrire une fonction qui prend en paramètre l'adresse d'un tableau d'entiers et sa taille et inverse ses

éléments. Exemple :

```
Tableau en entrée : [1,5,7,8,2,4,8]
Tableau inversé   : [8,4,2,8,7,5,1]
```

Solution :

```
todo
```

6. Écrire une fonction qui retourne la position de la première occurrence d'une valeur, si elle existe, dans un tableau. *Solution :*

```
todo
```

7. Écrire une fonction qui retourne **affiche** nombre d'occurrence d'une valeur dans un tableau ainsi que les positions où elle apparaît dans un tableau
   Solution :

```
todo
```

8. Écrire une fonction qui teste si un tableau est ordonné.
   Solution :

```
todo
```

9. Écrire une fonction qui insère une valeur dans un tableau trié.
   Solution :

```
todo
```

10. Écrire une fonction qui décale les valeurs dans un tableau d'une position vers la gauche
    Solution :

```
todo
```

11. Dans la section data on déclare la matrice suivante :

```
.data
tab:    .int 1,5,9
        .int 4,8,7
```

Écrire un programme qui permet de parcourir la matrice `tab` en utilisant deux boucles imbriquées :

```
.data
str:    .asciz "--> %d \n"
tab:    .int 1,5,9
        .int 4,8,7

l:      .int 0
c:      .int 0

.text
.global main
main:
    movl %esp, %ebp
    xorl  %eax, %eax

loop1:
    cmpl    $2,    l          #pour aller de 0 à 1 (deux lignes)
    jge     end_loop1
    movl    $0,     c
loop2 :
    cmpl    $3,    c          #pour aller de 0 à 2 (trois colonnes)
    jge     end_loop2

    movl    l,      %ecx
```

```
    imull   $3,     %ecx
    addl    c,      %ecx
    imull   $4,     %ecx
    movl    $tab,   %ebx
    movl    (%ebx, %ecx) , %eax # on met la valeur dans %eax pour faciliter le debug

    pushl   %eax
    pushl   $str
    call    printf
    addl    $8,     %esp
    addl    $1,     c
    jmp     loop2
end_loop2:

    addl    $1, l
    jmp     loop1
end_loop1:
    ret
```

Les chaînes de caractères :

1. Écrire une fonction `str_len` qui permet trouver le nombre de caractères dans une chaîne de caractères (la taille d'une chaîne).

   Solution :

```
.data
str:        .asciz "sum chars = %d"
my_char :    .asciz "Hello Worlds"

.text
.global main

str_len:
    pushl   %ebp
    movl    %esp,       %ebp
    movl    8(%ebp),    %esi # @ de my_char
    xorl    %eax,       %eax

sum_loop:
    movb    (%esi,%eax),    %dl
    cmpb    $0,             %dl
    jz      end_sum_loop
    incl    %eax
    jmp     sum_loop
end_sum_loop:
    pop     %ebp
    ret

main:
    movl    %esp, %ebp
    pushl   $my_char
    call    str_len
    addl    $4,     %esp
    pushl   %eax
    pushl   $str
    call    printf
    addl    $8,     %esp
    xorl    %eax, %eax
```

```
    ret
```

2. Écrire une fonction `copy_str` qui permet copier une chaîne de caractères dans une autre, les adresses des deux chaînes sont données en paramètre

   Solution :

```
.data
str_print: .asciz "str = %s\n"
string1:   .asciz "Copy this string"

.bss
.lcomm  string2, 100

.text
.global main

/**
  * en paramète l'adresse de dest puis src
  */
copy_str:
    pushl   %ebp
    movl    %esp,       %ebp
    movl    12(%ebp),   %edi # @ destination
    movl     8(%ebp),   %esi # @ source
    xorl    %eax,       %eax

copy_str_loop:
    movb    (%esi,%eax),    %dl
    cmpb    $0,             %dl
    jz      end_copy_str_loop
    movb    %dl,            (%edi,%eax)
    incl    %eax
    jmp     copy_str_loop
end_copy_str_loop:
    pop     %ebp
    ret

main:
    movl    %esp, %ebp
    pushl   $string2
    pushl   $string1
    call    copy_str
    addl    $8,     %esp
    pushl   $string2
    pushl   $str_print
    call    printf
    addl    $8,     %esp
    pushl   $string1
    pushl   $str_print
    call    printf
    addl    $8,     %esp
    xorl    %eax, %eax
    ret
```

3. Écrire une fonction qui permet concaténer deux chaînes de caractères.

   Solution :

```
    todo
```

4. Écrire une fonction `str_cmp` qui permet comparer deux chaînes de caractères `str1` et `str2` selon l'ordre lexicographique. La fonction doit renvoyer `1` si `str1 > str2`, `-1` si `str1 < str2`, `0` sinon.

   Solution :

   ```
   todo
   ```

5. Écrire une fonction qui permet convertir les lettres minuscules dans une chaîne de caractères en lettres majuscules.

   Solution :

   ```
   todo
   ```

6. Écrire une fonction qui `rev_str` permet inverser une chaîne de caractères

   Solution :

```
.data
str_print:  .asciz  "str = %s \n"
str:        .asciz  "abcdef"

.text
.global main
rev_str:
    pushl   %ebp
    movl    %esp,       %ebp
    movl    8(%esp),    %esi    #adresse de la str
    xorl    %ecx,       %ecx
    xorl    %edx,       %edx
    xorl    %edi,       %edi
loop_push_str:
    movb    (%esi,%ecx),    %dl
    cmp     $0,             %dl
    jz      end_loop_push_str
    decl    %esp
    movb    %dl,    (%esp)
    incl    %ecx
    jmp     loop_push_str
end_loop_push_str:

loop_pop_str:
    cmpl    $0,         %ecx
    jle     end_loop_pop_str
    movb    (%esp),     %dl
    movb    %dl,        (%esi,%edi)
    incl    %esp
    incl    %edi
    decl    %ecx
    jmp     loop_pop_str
end_loop_pop_str:

    xorl    %eax,   %eax
    popl    %ebp
    ret

main:
    movl %esp, %ebp #for correct debugging
    # write your code here
    xorl    %eax, %eax
    pushl   $str
    call    rev_str
    addl    $4,     %esp
```

```
        pushl   $str
        pushl   $str_print
        call    printf
        addl    $8,     %esp

        ret
```

7. Écrire une fonction qui permet vérifier si une chaîne de caractères est un palindrome
   Solution :

   ```
   todo
   ```

8. Écrire une fonction qui permet trouver le caractère le plus fréquent dans une chaîne de caractères
   Solution :

   ```
   todo
   ```

9. Écrire une fonction qui calcule le nombre d'alphabets, le nombre de chiffres et le nombre de caractères spéciaux dans une chaîne de caractères. On supposes que les paramètres sont données par référence

Utilisation des conventions

1. Pour chacune des fonctions écrite précédemment (exercice 3) vous devez l'appeler et la tester depuis un programme C.

Exercice 4 : Les appels systèmes

Les appels systèmes sont listé ci-dessus (dans ce document on utilise uniquement les appels 32 bits). La liste complète est fourni dans les documents de cours sans la section *ressources*
ou dans les headers du noyau linux ici :
/usr/include/x86_64-linux-gnu/asm/unistd_32.h

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
....
```

Exemple :

```
/***  DATA SECTION ***/
.data

str:    .asciz "Hello World\n"
        len = . - str - 1

/***  TEXT SECTION ****/

.text
        .global _start
```

```
_start:
        /** Appel System **/
        movl $0x4, %eax
        movl $0x1, %ebx
        movl $str, %ecx
        movl $len, %edx
        int $0x80

        /** EXIT **/
        movl $0x1, %eax
        movl $0x0, %ebx
        int $0x80
```

Compilation

```
as -o helloWorld.o helloWorld.s
ld -o helloWorld helloWorld.o
```

Exercice :

1. Écrire un programme en assembleur qui affiche à l'écran un le message suivant : "L'assembleur est cool ! " en utilisant les appels systèmes adéquats.
2. Écrire un programme qui permet de créer un fichier texte vide "test.txt"
3. Écrire un programme qui permet de lire un fichier texte préalablement rempli "myfile.txt".