



ESIR SYS1 CC2 2016–2017
12 avril 2017

Nom et Prénom

Y Y

numéro d'étudiant : 999999992

Durée : 1h, Notation : sur 20 points

1 Première partie : QCM (10 points)

Instructions :

Cochez clairement la case de la réponse que vous pensez être juste. Il y a **une seule réponse** juste par question.

Barème :

+0.5 pour chaque réponse correcte

−0.5/ m pour chaque réponse fausse (où $m+1$ est le nombre de réponses possibles)

Question 1 Si l'exécutable `a.out` utilise la librairie partagée (**shared library**) `libfoo.so`, trouvera-t-on le contenu de `libfoo.so` dans le fichier `a.out` ?

☐ oui

☐ non

Question 2 Combien de commandes unix la ligne suivante combine-t-elle ?

```
sed 's/[;";.: !?-\]/\n/g' pg84.txt | sort -f | uniq -c | sort -nrk1 > xyk
```

☐ 4

☐ 9

☐ 5

☐ 10

Question 3 Comment désalloue-t-on explicitement de la mémoire sur le tas en Java ?

☐ On ne peut pas désallouer explicitement la mémoire du tas en Java.

☐ avec la méthode `free()`

☐ avec l'opérateur `delete`

Question 4 Dans un ordinateur moderne, si deux processus modifient le contenu d'une même adresse (par exemple `0x00400000`), s'agit-il en général :

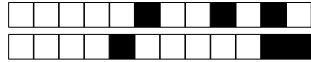
☐ de la même 'case' mémoire physique (les modifications d'un processus sont visibles par l'autre processus).

☐ de deux 'cases' mémoire physiques distinctes (les modifications d'un processus ne sont pas visibles par l'autre processus).

Question 5 Dans un processeur moderne :

☐ le cache L2 est plus rapide que le cache L1.

☐ le cache L1 est plus rapide que le cache L2.



Question 6 On considère l'extrait de code en C suivant :

```
int16_t i = 10;
char* ptr_i = (char*)&i;
*(ptr_i+1) = 2 ;
printf("%i\n", (int)i );
```

La ligne `printf("%i\n", (int)i);` permet d'imprimer un entier. Quelle sortie produit cet extrait de programme lorsqu'il s'exécute sur un processeur Intel de la famille x86 ?

- ☐ 522
- ☐ 10
- ☐ 256

Question 7 Soit la ligne de commandes suivante :

```
a=Salut ; echo '$a'
```

Qu'imprime-t-elle sur la console ?

- ☐ \$a
- ☐ Salut
- ☐ 'Salut'

Question 8 Le code suivant compile-t-il en C ?

```
#include <stdio.h>
int main( int argc, char** argv) {
    int x = "Bonjour" - "Au revoir";
    printf("%i\n", x );
} // EndMain
```

- ☐ non
- ☐ oui

Question 9 À quoi la seconde ligne du code suivant est-elle équivalente ?

```
char* s = "Hi!";
*(s+1) = 1 ;
```

- ☐ `s[1] = 1`
- ☐ `s = 0`
- ☐ `s = s + 1`



Question 10 Soit le code suivant

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void foo(char** c) {
    char a_string[] = "SYS1 c'est top!";
    *c = malloc(strlen(a_string)+1);
    strcpy(*c,a_string);
}

int main(int argc, char** argv) {
    char* c;
    printf("%s\n",c);
}
```

Dans quel région mémoire la chaîne contenue dans `a_string[]` est-elle stockée ?

- ☐ sur la pile
- ☐ dans le segment BSS
- ☐ dans le segment de texte
- ☐ sur le tas

Question 11 Soit le code assembleur suivant

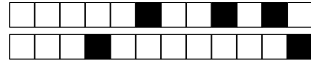
```
MOV BX,5
MOV CX,6
PUSH BX
PUSH CX
MOV AX,7
MOV BX,8
POP BX
POP AX
```

À la suite de ce code :

- ☐ AX==7, BX==8
- ☐ AX==5, BX==6
- ☐ AX==7, BX==5
- ☐ AX==8, BX==7

Question 12 Comment réserver 16 octets (10h en hexadécimal) sur la pile en x86 16 bits ?

- ☐ sub bp,10h
- ☐ add sp,10h
- ☐ add bp,10h
- ☐ sub sp,10h



Question 13 Soit le code suivant

```
#include <stdio.h>
```

```
void foo(char* i) {  
    i += 1;  
}
```

```
int main( int argc, char** argv) {  
    char i = 'A';  
    foo(&i);  
    printf("%c\n",i);  
}
```

Quelle valeur ce code imprime-t-il sur la console ?

- ☐ A
- ☐ b
- ☐ B

Question 14 On considère les instructions assembleur TASM suivantes (assembleur 16bits) :

```
PUSH BP  
MOV BP,SP
```

À quoi ces deux lignes peuvent-elles correspondre ?

- ☐ au prologue d'une fonction
- ☐ à l'épilogue d'une fonction
- ☐ à l'invocation d'une fonction

Question 15 Si le répertoire de travail est `/Users/ftaiani/`, quel répertoire représente `../ftaiani/Desktop/../../../../?`

- ☐ `/Users/ftaiani/Desktop/`
- ☐ `/Users/`
- ☐ `/Users/ftaiani/`

Question 16 Quel est l'ordre de grandeur de la taille d'un cache de niveau 1 (L1) sur un processeur moderne ?

- ☐ 1 megaoctet
- ☐ 1 gigaoctet
- ☐ 10 octets
- ☐ 64 kilooctets

Question 17 On considère le nombre hexadécimal 64 bits `0x44332211` stocké en mémoire sur une architecture Little Endian. Quel est la séquence d'octets qui encode ce nombre ?

- ☐ `0x44,0x33,0x22,0x11`
- ☐ `0x11,0x22,0x33,0x44`

Question 18 Où sont allouées les variables automatiques ?

- ☐ sur la pile
- ☐ sur le tas
- ☐ sur le segment BSS



Question 19 Soit le code assembleur TASM suivant :

```
.MODEL SMALL      ; memory model
.STACK            ; memory space for the stack
.DATA             ; data segment
    MESSAGE DB 'foo is executing',13,10,'$'
.CODE

foo:  MOV DX, offset MESSAGE ; DX contains start of string to be printed
      MOV AH, 09H           ; 09H is the number of the DOS print routine
      INT 21H               ; calling DOS print routine
      RET

start: MOV AX, @DATA         ; initialising
      MOV DS, AX            ; stack segment (needed on 16bit)

      CALL foo

      MOV AH, 4CH           ; 4CH is the number of the DOS exit routine
      INT 21H               ; calling DOS exit routine
END start
```

Ce code est sensé imprimer la chaîne de caractère 'foo is executing', puis terminer. Ce code est-il correct ?

☐ non

☐ oui

Question 20 Soit le code assembleur suivant implémentant une boucle :

```
MOV BX,25
MOV AX,10
lp:  ; body of the loop
      INC AX
      CMP AX,BX
      JGE lp
```

Combien de fois le corps de la boucle (*body of the loop*) va-t-il être exécuté ? (On supposera que ni AX ni BX ne sont modifiés dans ce corps de boucle.)

☐ 16

☐ 0

☐ 15

☐ 1



2 Deuxième partie : Questions ouvertes et problèmes (10 points)

Question 21 On considère un petit programme, dont le code est réparti entre deux fichiers source, `mon_petit_programme.c` et `ma_procedure.c`. Le fichier `mon_petit_programme.c` contient le code suivant :

```
int main() {
    ma_procedure(5);
}
```

et le fichier `ma_procedure.c` le code qui suit :

```
#include <stdio.h>
void ma_procedure(int nb) {
    printf("Le nombre est: %d\n",nb);
}
```

On compile ce programme avec les commandes suivantes. (Les '#' introduisent des commentaires.)

```
$ gcc -c mon_petit_programme.c      # génère mon_petit_programme.o
$ gcc -c ma_procedure.c             # génère ma_procedure.o
$ gcc ma_procedure.o mon_petit_programme.o -o mon_petit_executable
```

À la suite de ces commandes, le programme peut être exécuté sans erreur :

```
$ ./mon_petit_executable
Le nombre est: 5
```

On rappelle que l'option `-c` de `gcc` compile et assemble les fichiers sources, mais n'opère aucune édition de lien :

-c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

L'option `-o` indique dans quel fichier doit être produit l'exécutable (plutôt que le fichier exécutable par défaut `a.out`). On s'intéresse au statut des symboles `main`, `ma_procedure` et `printf` dans les fichiers objets `mon_petit_programme.o` et `ma_procedure.o` et dans l'exécutable `mon_petit_executable`. Pour les étudier on exécute les commandes suivantes :

```
$ nm mon_petit_programme.o | grep -E " (main|ma_procedure|printf)"
                 U ma_procedure
0000000000000000 T main
$ nm ma_procedure.o | grep -E " (main|ma_procedure|printf)"
0000000000000000 T ma_procedure
                 U printf
$ nm mon_petit_executable | grep -E " (main|ma_procedure|printf)"
00000000004004e4 T ma_procedure
0000000000400508 T main
                 U printf@@GLIBC_2.2.5
```

Pour rappel, la commande Unix `nm` permet de lister les symboles mentionnés dans un fichier objet ou un exécutable au format ELF.

1. Que signifient les lettres U et T dans les sorties de `nm` ?
2. Expliquez l'évolution du statut de chaque symbole (`main`, `ma_procedure` et `printf`) dans les sorties successives de `nm`.

[3 points]

☐ A+ ☐ A ☐ B ☐ C ☐ D ☐ E ☐ F



+74/7/58+





Question 22 On considère le programme C suivant.

```
int foo(int max) {
    int k; int r = 0;
    for(k=0; k<max; k++) { r = r + k ; }
    return r ;
} // EndMain

int main() {
    int r = foo(64);
    printf("%d\n",r);
} // EndMain
```

Après compilation (gcc loop_in_c.c), ce programme calcule la somme des entiers compris entre 0 et 63 :

```
$ gcc loop_in_c.c
$ ./a.out
2016
```

On décompile l'exécutable obtenu (en utilisant objdump -S -Intel -d a.out) pour mieux comprendre l'utilisation de la pile par le compilateur. On obtient le code assembleur suivant pour la procédure foo :

```
000000000400534 <foo>:
400534:    55                push    rbp
400535:    48 89 e5          mov     rbp, rsp
400538:    48 83 ec 20       sub     rsp, 0x20
40053c:    89 7d ec          mov     DWORD PTR [rbp-0x14], edi
40053f:    c7 45 fc 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
400546:    c7 45 f8 00 00 00 mov     DWORD PTR [rbp-0x8], 0x0
40054d:    eb 0a            jmp     400559 <foo+0x25>
40054f:    8b 45 f8          mov     eax, DWORD PTR [rbp-0x8]
400552:    01 45 fc          add     DWORD PTR [rbp-0x4], eax
400555:    83 45 f8 01       add     DWORD PTR [rbp-0x8], 0x1
400559:    8b 45 f8          mov     eax, DWORD PTR [rbp-0x8]
40055c:    3b 45 ec          cmp     eax, DWORD PTR [rbp-0x14]
40055f:    7c ee            jle     40054f <foo+0x1b>
...
40056b:    8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
40056e:    c9               leave
40056f:    c3               ret
```

La notation DWORD PTR [rbp-0x4] est équivalente à la notation rbp[-0x4] en Turbo Assembleur, et référence une zone mémoire sur 32 bits (DWORD PTR signifiant pointeur vers un double mot, soit 32 bits). Les registres rxx sont sur 64 bits, et les registres exx sur 32. La notation 0x.. représente un nombre en notation hexadécimale (équivalente à ..H en Turbo Assembleur).

Quel sont le prologue et l'épilogue de la procédure foo. foo réserve-t-elle de la place sur la pile ? Si oui, combien d'octets, et avec quelle(s) instruction(s) ? [2 points]

☐ A ☐ B ☐ C ☐ F



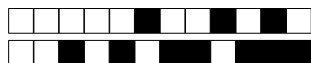
Question 23 Dans le programme de la question 2, la procédure `main` appelle la procédure `foo` à travers le code assembleur suivant :

```
400578:      bf 40 00 00 00      mov     edi,0x40
40057d:      e8 b2 ff ff ff      call    400534 <foo>
```

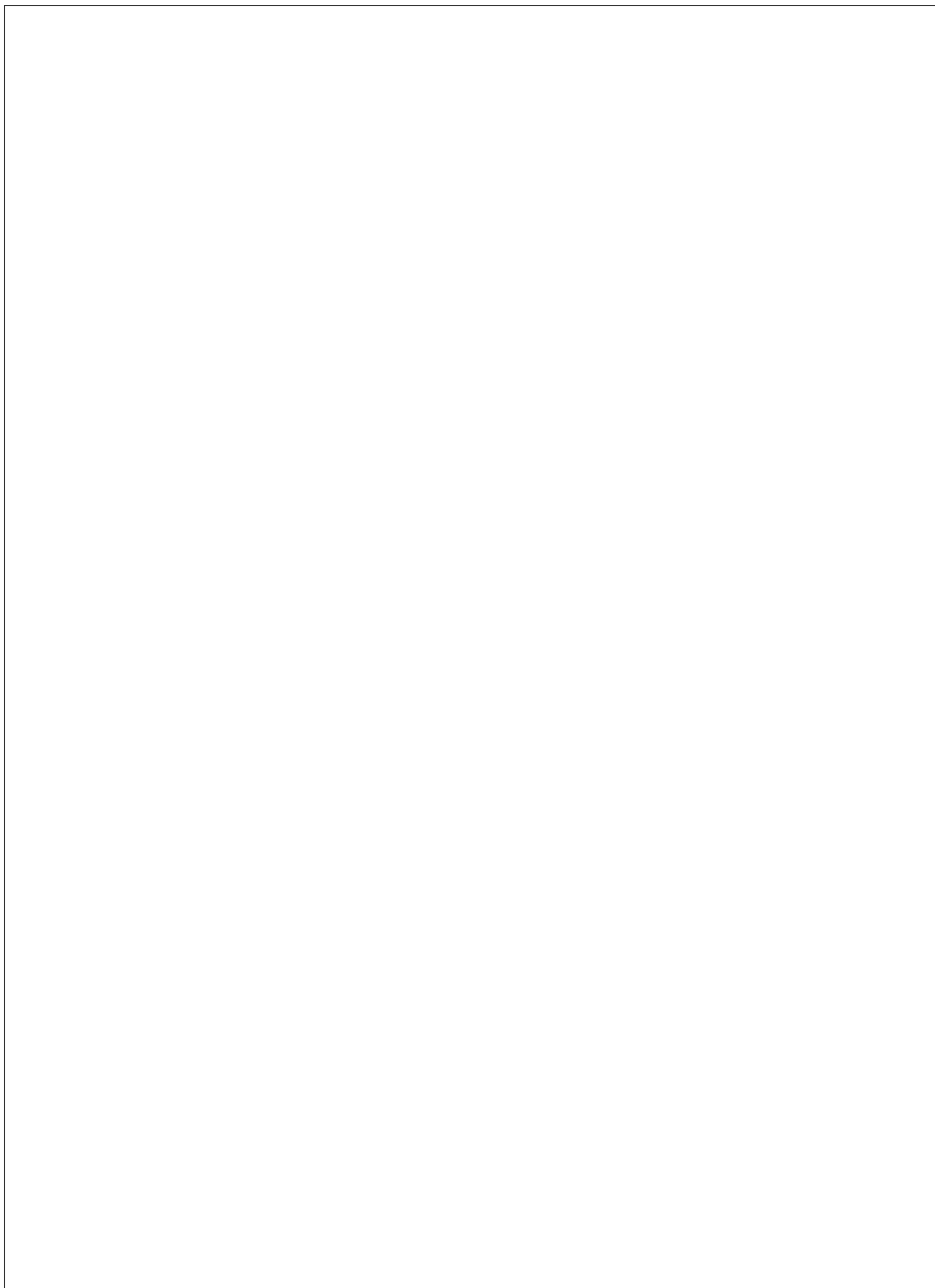
En étudiant le code ci-dessus et le code assembleur de la procédure `foo` de la question 2, répondez aux questions suivantes :

1. Lors de l'appel de `foo` par `main` à la ligne `int r = foo(64);`, comment le paramètre d'entrée `max` est-il passé à `foo`. S'agit-il d'un passage par valeur ou par adresse? Par la pile ou par registre? **[1 points]**
2. Dans le code assembleur de `foo` fourni à la question 2, comment sont stockées les variables `max`, `r`, et `k`, respectivement. Quelles sont leurs adresses? Représentez le schéma de pile utilisé par `foo`. **[2 points]**

☐ A+ ☐ A ☐ B ☐ C ☐ D ☐ E ☐ F



+74/10/55+





Question 24 On compile maintenant le programme C de la question 2 avec le premier niveau d'optimisation du compilateur gcc (gcc -O1 loop_in_c.c). Le code assembleur de foo, une fois décompilé, devient :

```
0000000000400534 <foo>:
 400534:    53                push    rbx
 400535:    bb 00 00 00 00    mov     ebx,0x0
 40053a:    b8 00 00 00 00    mov     eax,0x0
 40053f:    85 ff             test    edi,edi
 400541:    7e 09             jle     40054c <foo+0x18>
 400543:    01 c3             add     ebx,eax
 400545:    83 c0 01          add     eax,0x1
 400548:    39 f8             cmp     eax,edi
 40054a:    75 f7             jne     400543 <foo+0xf>
 40054c:    .. ..           ...
 400556:    89 d8             mov     eax,ebx
 400558:    5b               pop     rbx
 400559:    c3               ret
```

Dans ce code, l'instruction `test edi,edi` permet de tester si le registre `edi` est nul (équivalent à `cmp edi,0`). En étudiant ce code, expliquez comment les variables `max`, `r`, et `k` sont maintenant stockées. Comment la boucle `for(k=0; k<max; k++) { .. }` est-elle implémentée? Pensez-vous que ce code puisse s'exécuter plus rapidement que la version compilée sans `-O1`? Pourquoi? [2 points]

☐ A ☐ B ☐ C ☐ D ☐ F



+74/12/53+