

RAPPORT DE TP – ENSIBS Cybersécurité du Logiciel – 3^e année

TP ASSEMBLEUR

« Assembleur x86 »

TP réalisé par

Robin MARCHAND
Lucas CHAPRON

TP encadré par

Sébastien GUILLET

Table des matières

I.	INTRODUCTION	4
II.	DÉVELOPPEMENT	4
1.	CONTEXTE.....	4
2.	APPEL DE FONCTIONS.....	4
i.	Analyse du projet HelloWorld	4
ii.	Écriture d'un programme faisant appel à MessageBox.....	5
3.	MODES D'ADRESSAGES.....	6
i.	Routine toMaj d'une chaîne de caractères	6
ii.	Routine count d'une chaîne de caractères	9
4.	VARIABLES LOCALES	11
i.	Traduction de la fonction myst en ASM	11
ii.	Suite de Fibonacci et vérification du bon fonctionnement du programme ASM.....	12
iii.	Écriture d'une fonction de comptage de caractères	14
5.	UN PEU DE CALCUL	17
i.	Écriture d'une fonction pour avoir les diviseurs d'un nombre.....	17
ii.	Écriture d'une fonction récursive calculant la factorielle	20
6.	UN PEU DE LECTURE	22
i.	Analyse de l'article	22
III.	CONCLUSION	23

Table des figures

Figure 1 : Appel HelloWorld sur x32dgb	4
Figure 2 : Exécution de HelloWorld.exe	4
Figure 3 : Programme C : HelloWorld	5
Figure 4 : Programme ASM : MessageBox.....	5
Figure 5 : Test MessageBox	5
Figure 6 : Programme C : MessageBox.....	6
Figure 7 : Routine toMaj	6
Figure 8 : Programme ASM : toMaj.....	7
Figure 9 : Test toMaj	7
Figure 10 : Blocs ida toMaj	8
Figure 11 : Programme C : toMaj.....	8
Figure 12 : Routine ASM : cptCarac	9
Figure 13 : Programme ASM : cptCarac	9
Figure 14 : Blocs ida cptCarac	10
Figure 15 : Programme C : cptCarac	10
Figure 16 : Programme C : FUN_00401000	10
Figure 17 : Test cptCarac	10
Figure 18 : Programme C : myst	11
Figure 19 : Routine ASM : myst initialisation	11
Figure 20 : Routine ASM : myst boucle for	12
Figure 21 : Routine ASM : myst retour.....	12
Figure 22 : Test Fibonacci	12
Figure 23 : Blocs ida Fibonacci	13
Figure 24 : Programme C : Fibonacci	13
Figure 25 : Routine ASM : count initialisation	14
Figure 26 : Routine ASM : count boucle for	14
Figure 27 : Routine ASM : count vérification terminaison	15
Figure 28 : Routine ASM : count retour.....	15
Figure 29 : Test count.....	15
Figure 30 : Blocs ida count.....	16
Figure 31 : Programme C : count	16
Figure 32 : Programme ASM : diviseur initialisation	17
Figure 33 : Programme ASM : diviseur	18
Figure 34 : Test diviseur	18
Figure 35 : Blocs ida diviseur.....	19
Figure 36 : Programme C : diviseur	19
Figure 37 : Routine ASM : factorielle	20
Figure 38 : Programme ASM : factorielle	21
Figure 39 : Blocs ida factorielle	21
Figure 40 : Programme C : factorielle	21
Figure 41 : Programme C : factorielle FUN_00401000	22

I. INTRODUCTION

Dans le cadre de notre première année du cycle ingénieur en Cybersécurité du Logiciel à l'ENSIBS, il nous est proposé un TP de 12h nous permettant de mettre en pratique nos connaissances et nos compétences pour le développement de programme en assembleur (MASM).

II. DÉVELOPPEMENT

1. CONTEXTE

Étant tous les deux passionnés de *Reverse Engineering*, nous n'avons pas eu beaucoup de difficultés durant ce projet. Le code bas niveau nous passionne vraiment et nous avons pris du plaisir à travailler dessus.

Pour chaque exercice de code, nous avons choisi d'utiliser *Ida* et *Ghidra* pour décompiler et expliquer plus en détail les différents bouts de code.

Même si l'utilisation de ce genre de logiciel peut se révéler inutile dans le Reverse de la plupart des Malwares.

2. APPEL DE FONCTIONS

i. Analyse du projet HelloWorld

On nous demande donc d'ouvrir le projet *HelloWord* puis de le compiler avec le fichier .bat
On utilise donc *x32dbg* pour regarder comment est réalisé l'appel à la fonction :

00401000	6A 2A	push 2A	EntryPoint
00401002	68 00304000	push helloworld.403000	403000:"Hello world : %d\n"
00401007	FF15 0C204000	call dword ptr ds:[<&printf>]	
0040100D	68 12304000	push helloworld.403012	403012:"Pause\r\n"
00401012	FF15 08204000	call dword ptr ds:[<&system>]	
00401018	83C4 04	add esp,4	
0040101B	B8 00000000	mov eax,0	
00401020	50	push eax	call \$0
00401021	E8 00000000	call <JMP.&ExitProcess>	JMP.&ExitProcess
00401026	FF25 00204000	jmp dword ptr ds:[<&ExitProcess>]	

Figure 1 : Appel HelloWorld sur x32dbg

Une fois l'exécution du programme, on peut remarquer que le programme exécute une suite d'instruction pour afficher *Hello World : 42*

On peut observer un ajout sur la pile du nombre de *0x2a* ce qui correspond bien au nombre 42.

Ensuite, le programme rajoute sur la pile le contenu de la variable helloworld vers l'adresse *403000*. En regardant dans le débogueur, on observe le contenu de l'adresse 403000 : *"Hello World : %d\n"*. Donc il contient bien l'appel du string *Hello Word* : et du int 42 représenté par *%d*.

Ensuite, sur 00401007, on voit bien l'appel de *printf* qui permet d'afficher le message voulu.

On peut remarquer également un appel de *&ExitProcess* qui permet d'afficher le message *Appuyez sur une touche* pour continuer à la fin du programme.

```
Hello World : 42
Appuyez sur une touche pour continuer...
```

Figure 2 : Exécution de HelloWorld.exe

Le programme en C ressemblerait donc à cela :

```
void entry(void)
{
    printf(s_Hello_World : _%d_00403000,0x2a);
    system(s_Pause_00403012);
    ExitProcess(0);
    return;
}
```

Figure 3 : Programme C : HelloWorld

ii. Écriture d'un programme faisant appel à MessageBox

Dans cette dernière partie de la partie A, il nous est demandé de créer un fichier *MessageBox.asm*. Ce programme va nous permettre d'afficher une fenêtre Windows avec un message stocké à l'intérieur.

Avec un peu de recherches, on trouve comment afficher une fenêtre du type *MessageBox*. Cette fonction prend exactement 4 arguments :

On peut donc commencer à coder le programme en ASM :

```
; -----
; Variables non-initialisees (bss)
; -----
.DATA?

; -----
; Section du code
; -----
.CODE
start:
    push MB_OK                ; On place le 4e argument sur la pile (bouton OK)                ; On peut aussi mettre MB_OKCANCEL
    push offset MsgBoxTitre    ; On place le 3e argument sur la pile (titre de la MessageBox)            ; On peut aussi mettre NULL
    push offset MsgBoxTexte    ; On place le 2de argument sur la pile (texte de la MessageBox)        ; On peut aussi mettre NULL
    push NULL                  ; On place le 1re argument sur la pile (le parent de la MessageBox, ici null car elle n'a pas de parent) ; On peut aussi mettre NULL
    call MessageBox            ; Appel a la MessageBox(HWND hwnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType) ; On peut aussi mettre MessageBoxA

    ; On aurait pu utiliser ceci pour ne pas pousser sur la pile les arguments :
    ; "invoke MessageBox, NULL, addr MsgBoxTexte, addr MsgBoxTitre, MB_OK"

    invoke ExitProcess,NULL    ; On quitte le programme
                                ; push NULL
                                ; call ExitProcess

end start
```

Figure 4 : Programme ASM : MessageBox

Comment nous pouvons le voir, la Box aura *TP1 MARCHAND / CHAPRON* comme titre et en texte *MessageBox => OK pour quitter*

Un bouton OK permettra également de fermer la fenêtre :

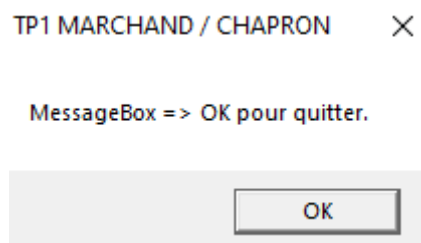


Figure 5 : Test MessageBox

Le programme en C ressemblerait donc à cela :

```
void entry(void)
{
    MessageBoxA((HWND)0x0,s_MessageBox=>_OK_pour_quitter._00403017,s_TP1_MARCHAND/_CHAPRON_00403000,0);
    ExitProcess(0);
    return;
}
```

Figure 6 : Programme C : MessageBox

3. MODES D'ADRESSAGES

i. Routine toMaj d'une chaîne de caractères

Dans cette partie, on nous demande de réaliser une routine pour mettre en majuscule une chaîne de caractères.

Pour cela, nous avons créé une routine qui va récupérer un caractère sur la pile et va le transformer en un caractère majuscule.

Il y a plusieurs moyens de réaliser cet exercice, avec mon camarade, nous avons décidé de procéder comme tel :

- La routine va regarder dans le registre « AL » de taille 8 bits.
- Puis vérifie que la valeur est supérieure à « a » (97 en ASCII) et inférieur à « z » (122 en ASCII)
 - Si c'est le cas alors il s'agit d'une lettre de l'alphabet minuscule et on peut lui soustraire 32 pour récupérer un code ASCII se situant en 65 et 90, représentant les majuscules en ASCII.
- Si la valeur n'est pas comprise dans les bornes 97 et 122, alors il s'agit soit d'un caractère déjà en majuscule, soit d'un caractère non alphabétique et alors nous ne modifions pas le caractère sur la pile.

```
toMaj proc
    cmp     AL,'a'
    jnb     sortie
    cmp     AL,'z'
    ja      sortie
    sub     AL,32

    sortie:
        ret
toMaj endp
```

Figure 7 : Routine toMaj

On la place dans un sous-programme que l'on va appeler.

La chaîne de caractères est mise dans le registre "EBX", puis le premier char de la chaîne est récupéré et mise sur la pile, on vérifie ensuite qu'il n'est pas nul.

⇒ On vérifie si c'est la fin de la chaîne

```

; -----
; Variables initialisees
; -----
.DATA
    Chaine db "Chaine de test ~# ", 13, 10, 0 ; 13 = CR, 10 = LF, 0 = fin de chaine
    strCommand db "Pause", 13, 10, 0 ; 13 = CR, 10 = LF, 0 = fin de chaine

; -----
; Variables non-initialisees (bss)
; -----
.DATA?

; -----
; Section du code
; -----
.CODE

; Routine toMaj
toMaj proc
    cmp AL, 'a' ; AL == le caractère à modifier. ; AH == le caractère à remplacer
    jnb sortie ; Compare le caractère avec 'a' (97 en ASCII). ; si c'est le cas, on passe à la suite
    cmp AL, 'z' ; Si le caractère est plus petit, (donc un caractère en dessous du code ASCII) retourne le caractère. ; sinon, on passe à la suite
    jnb sortie ; Compare le caractère courant avec 'z' (122 en ASCII). ; si c'est le cas, on passe à la suite
    sub AL, 32 ; Si le caractère est plus grand, (donc un caractère au dessus du code ASCII) retourne le caractère. ; sinon, on passe à la suite
    ; Si le caractère est donc une minuscule, on va soustraire 32 au code ASCII pour recuperer une majuscule. ; sinon, on passe à la suite

    ; ASCII majuscule vont de 65 à 90 donc pour avoir un caractère minuscule en majuscule on soustrait 32.
    ; au caractère minuscule.

    sortie: ret ; Retourne le caractère modifié
toMaj endp ; Fin de la routine

start:
    push offset Chaine ; Charge la chaine de caractère dans la pile
    call crt_printf ; Affiche le message de base.

    mov ebx, offset Chaine ; ebx = adresse de la chaine de caractère

    begin_loop: ; Boucle infinie
        mov AL, [ebx] ; On récupère le premier caractère.
        cmp AL, 0 ; Si le caractère est 0, on sort de la boucle.
        jz end_loop ; Si on est en fin de phrase, on quitte la boucle.
        call toMaj ; Appel à la routine pour mettre en majuscule la lettre ou non.
        mov [ebx], AL ; On remplace le caractère original pour le nouveau (on majuscule ou non).
        inc ebx ; On itère sur la chaine.
        jmp begin_loop ; Retour au début de la boucle.

    end_loop: ; Fin de la boucle
        push offset Chaine ; Charge la chaine de caractère dans la pile
        call crt_printf ; Affiche le message en majuscule.

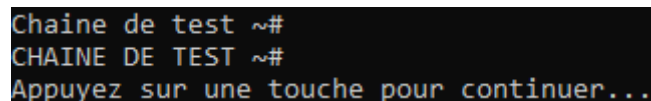
        invoke crt_system, offset strCommand ; Appel à la fonction systeme pour faire la pause.
        invoke ExitProcess, NULL ; Quitte le programme.

end start ; Fin de la routine

```

Figure 8 : Programme ASM : toMaj

Qui réalise le programme tel que :



```

Chaine de test ~#
CHAINE DE TEST ~#
Appuyez sur une touche pour continuer...

```

Figure 9 : Test toMaj

Comme nous pouvons le voir, les caractères compris entre ASCII $97 < x < 122$ sont bien transformés en majuscule. En revanche, les caractères spéciaux, comme “~” et “#” ne sont pas touchés.

Le programme est donc divisé comme ci-dessous :

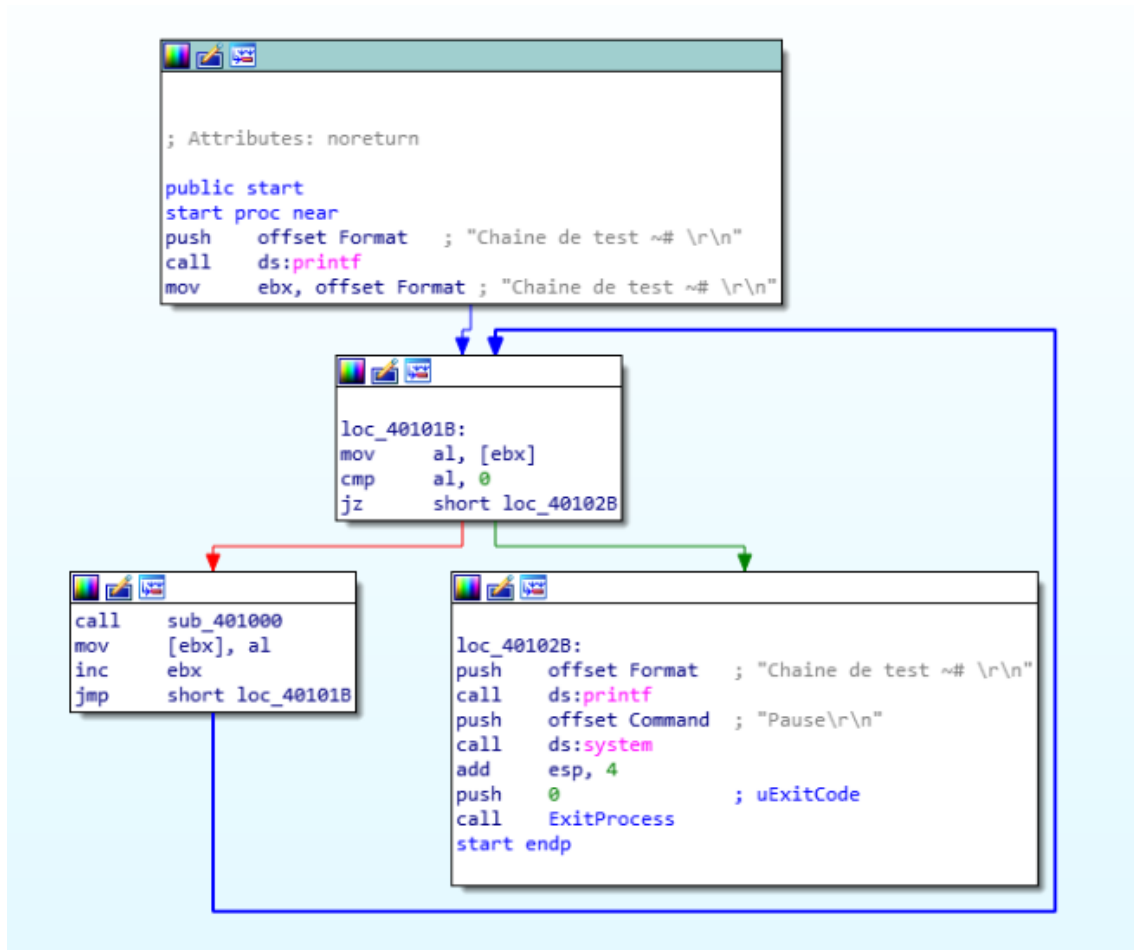


Figure 10 : Blocs ida toMaj

Le programme en C ressemblerait donc à cela :

```

void entry(void)
{
    char cVar1;
    char *pcVar2;

    printf(s_Chaine_de_test_~#_00403000);
    for (pcVar2 = s_Chaine_de_test_~#_00403000; *pcVar2 != '\0'; pcVar2 = pcVar2 + 1) {
        cVar1 = FUN_00401000();
        *pcVar2 = cVar1;
    }
    printf(s_Chaine_de_test_~#_00403000);
    system(s_Pause_00403015);

    ExitProcess(0);
}

```

Figure 11 : Programme C : toMaj

ii. Routine count d'une chaîne de caractères

Dans cette deuxième question, on nous demande de compter le nombre de caractères dans un string. Nous aurions pu exécuter ce programme dans le même fichier que la question précédente, mais pour un souci de compréhension, nous allons faire un fichier à part.

On peut utiliser le registre **EBP** pour le récupérer ainsi que **EAX** pour le modifier sur la pile.

Puis le string est placé dans le registre **EBX**

On récupère le premier char du string puis on call la routine qui va itérer le compteur présent sur la pile.

Voici la routine cptCarac :

```
; Routine cptCarac
cptCarac proc
    mov eax,[ebp-4]          ; Récupère depuis la pile l'état du compteur et le met dans le registre EAX.
    inc eax                 ; Incrémente EAX.
    mov [ebp-4], eax        ; Replace le compteur sur la pile.
    ret
cptCarac endp ; Fin de la routine cptCarac
```

Figure 12 : Routine ASM : cptCarac

Pour le déroulement, on passe au char suivant jusqu'à la fin du string et on affiche le message de sortie.

```
start: ; Début de la routine start
    push offset strMsgEntree          ; Push de la chaîne de caractère dans la pile.
    call crt_printf                   ; Affiche la chaîne de caractère.

    mov eax, -2                      ; Initialisation du compteur à -2 pour ne pas compter les caractères CRLF.
    mov [ebp-4], eax                 ; Le compteur est placé sur la pile.

    mov ebx, offset strMsgEntree      ; Met la chaîne d'entrée dans le registre ebx.

begin_loop: ; Début de la boucle
    mov AL, [ebx]                    ; Récupère le premier caractère.
    cmp AL, 0                        ; Si on est en fin de phrase, on quitte la boucle.
    jz end_loop                     ; Sinon, on passe à la suite.

    call cptCarac                    ; Appel à la routine pour itérer le compteur.
    inc ebx                          ; Itère sur la chaîne.
    jmp begin_loop                  ; Retour au début de la boucle.

end_loop: ; Fin de la boucle
    push [ebp-4]                     ; Push du compteur dans la pile.
    push offset strMsgSortie         ; Push de la chaîne de caractère dans la pile.
    call crt_printf                   ; Affiche le message en majuscule.

    invoke crt_system, offset strCommand ; Invoke de la commande system.
    invoke ExitProcess, NULL          ; Invoke de la commande ExitProcess.

end start ; Fin de la routine start
```

Figure 13 : Programme ASM : cptCarac

Voici le programme final.

Le programme est donc divisé comme ci-dessous :

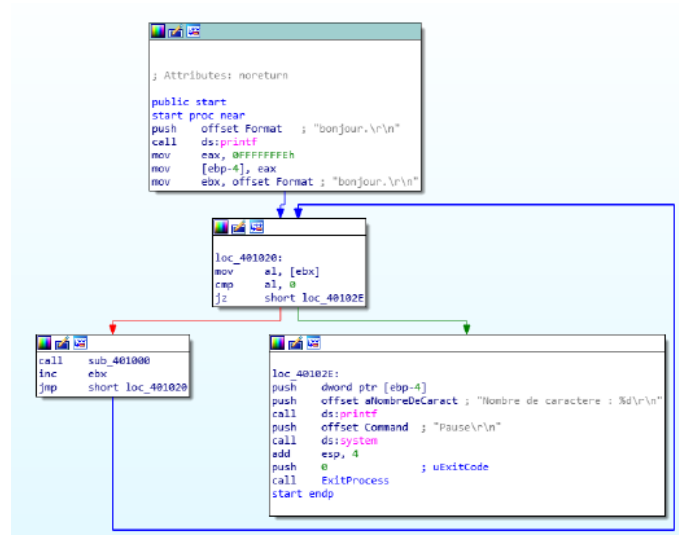


Figure 14 : Blocs ida cptCarac

Le programme en C ressemblerait donc à cela :

```

void entry(void)
{
    char *pcVar1;
    int unaff_EBP;
    printf(s_bonjour._00403000);
    *(undefined4 *) (unaff_EBP + -4) = 0xffffffff;
    for (pcVar1 = s_bonjour._00403000; *pcVar1 != '\0'; pcVar1 = pcVar1 + 1) {
        FUN_00401000();
    }
    printf(s_Nombre_de_caractere_ : _d_0040300b, *(undefined4 *) (unaff_EBP + -4));
    system(s_Pause_00403026);
    ExitProcess(0);
}

```

Figure 15 : Programme C : cptCarac

```

void FUN_00401000(void)
{
    int unaff_EBP;
    *(int *) (unaff_EBP + -4) = *(int *) (unaff_EBP + -4) + 1;
    return;
}

```

Figure 16 : Programme C : FUN_00401000

On a bien nos 2 fonctions, la fonction **entry** appelle bien la fonction **FUN_00401000** qui permet d'effectuer des opérations d'itérations.

Puis **entry** appelle le message de fin :

```

TP Marchand/Chapron
Nombre de caractere : 19
Appuyez sur une touche pour continuer...

```

Figure 17 : Test cptCarac

On a bien nos 19 caractères.

4. VARIABLES LOCALES

i. Traduction de la fonction myst en ASM

```
int myst(int n) {
    int i, j, k, l;
    j = 1;
    k = 1;
    for (i = 3; i <= n; i++) {
        l = j + k;
        j = k;
        k = l;
    }
    return k;
}
```

Figure 18 : Programme C : myst

En analysant rapidement cette fonction, on trouve qu'elle permet de calculer la suite de Fibonacci en fonction du paramètre n.

On compte également 4 variables qui devront être initialisées :

Valeurs	@
L	EBP-16
K	EBP-12
J	EBP-8
I	EBP-4
Registre EBP	EBP
n	EBP+8

On peut réserver 16 octets dans la pile pour y stocker nos variables.

On initialise nos 4 variables présentes dans le tableau ci-dessus à l'aide du registre **EAX**

```
.CODE
myst :
    push ebp                ; Sauvegarde de l'adresse de base du pointeur
    mov ebp,esp             ; Chargement de l'adresse du stack dans le registre de base du pointeur
    sub esp, 16             ; Sauvegarde de 16 octets pour les variables locales (4 octets chacun)
    mov eax, 3              ; Initialisation de la variable locale i
    mov [ebp-4], eax        ; Sauvegarde de la valeur de i dans la zone mémoire
    mov eax, 1              ; Initialisation de la variable locale j
    mov [ebp-8], eax        ; Sauvegarde de la valeur de j dans la zone mémoire
    mov eax, 1              ; Initialisation de la variable locale k
    mov [ebp-12], eax       ; Sauvegarde de la valeur de k dans la zone mémoire
    xor eax, eax            ; Initialisation de la variable locale l
    mov [ebp-16], eax       ; Sauvegarde de la valeur de l dans la zone mémoire
```

Figure 19 : Routine ASM : myst initialisation

On initialise bien nos valeurs comme dans le tableau.

L'étape d'après consiste à effectuer les opérations nécessaires sur le programme.

Au début de la boucle, on vérifie que i soit inférieur à égal à n. Si ce n'est pas le cas, on effectue un branchement vers la fin de fonction.

À la fin des opérations, on place un branchement pour remonter vers le début de la boucle.

```

for_1:                ; Boucle for
    cmp [ebp-4], ebx   ; Vérification de i<=n
    jg end_for_1       ; Si i>n, on sort de la boucle
    mov eax,[ebp-8]     ; Récupération de la valeur j
    add eax,[ebp-12]    ; j = j + k
    mov [ebp-16],eax    ; l = j
    mov eax,[ebp-12]    ; Récupération de la valeur k
    mov [ebp-8],eax     ; j = k
    mov eax,[ebp-16]    ; Récupération de la valeur l
    mov [ebp-12],eax    ; k = l
    mov eax,[ebp-4]     ; Récupération de la valeur i
    inc eax            ; i++
    mov [ebp-4], eax    ; Sauvegarde de la valeur de i dans la zone mémoire
    jmp for_1          ; On recommence la boucle

```

Figure 20 : Routine ASM : myst boucle for

Une fois la boucle terminée, on peut placer k dans la variable de retour en détruisant les variables locales et en restaurant les adresses de retour :

```

end_for_1:            ; Fin de la boucle for
    mov eax, [ebp-12]  ; Déplacement de k dans le registre de retour
    mov esp, ebp      ; Nettoyage des variables locales
    pop ebp           ; Restauration de l'adresse de base du pointeur
    ret               ; Retour de la fonction

```

Figure 21 : Routine ASM : myst retour

ii. Suite de Fibonacci et vérification du bon fonctionnement du programme ASM

Ce programme calcule la suite de Fibonacci en fonction de n

On initialise la valeur de n à 12 puis on lance le programme :

Le resultat de Fibonacci pour 12 est : 144

Figure 22 : Test Fibonacci

On peut également décompiler le programme en blocs/C pour comprendre d'une manière plus précise comment fonctionne ce dernier et quelles étapes sont appelées.

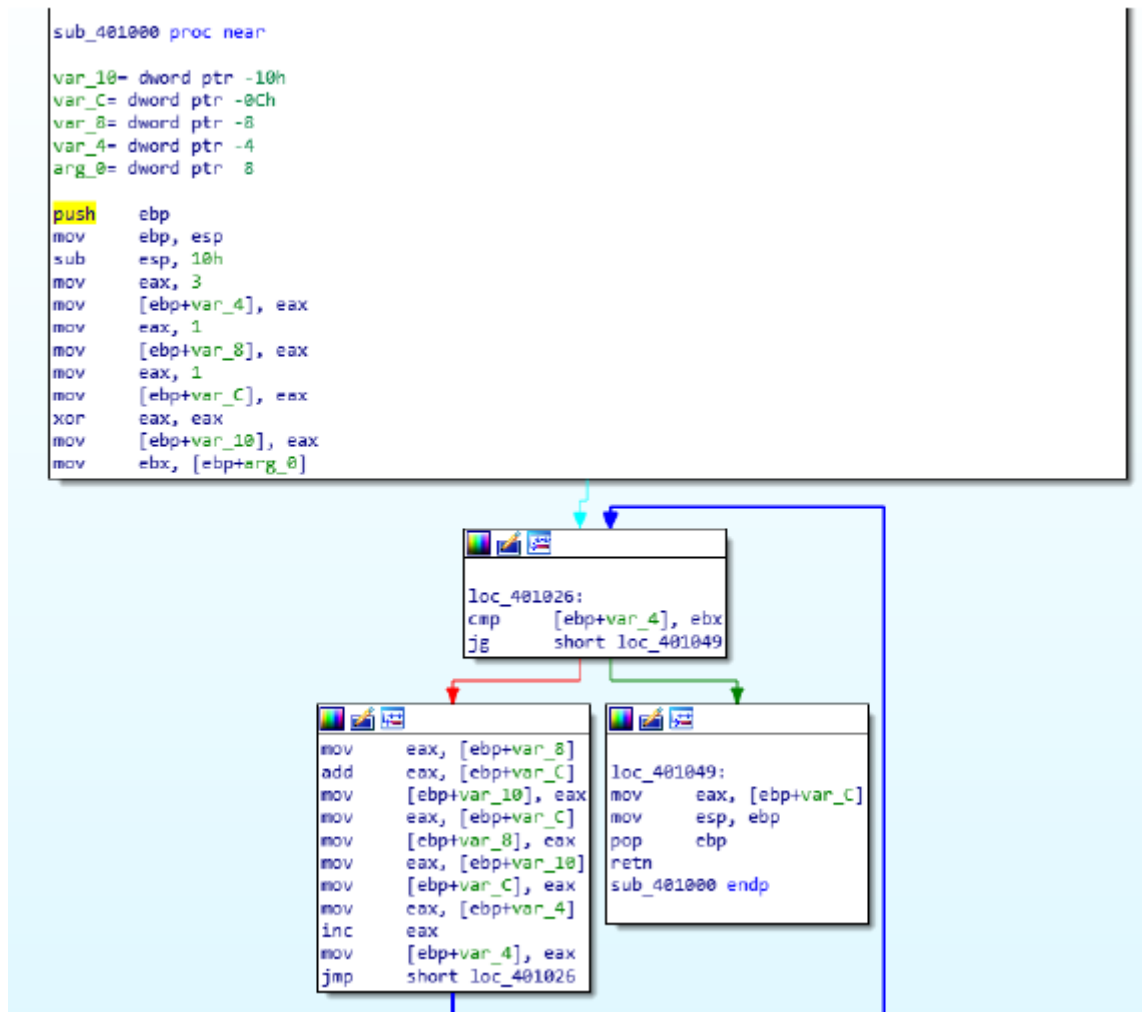


Figure 23 : Blocs ida Fibonacci

```

int __cdecl FUN_00401000(int param_1)
{
    int iVar1;
    int local_10;
    int local_c;
    int local_8;
    local_c = 1;
    local_10 = 1;
    for (local_8 = 3; local_8 <= param_1; local_8 = local_8 + 1) {
        iVar1 = local_c + local_10;
        local_c = local_10;
        local_10 = iVar1;
    }
    return local_10;
}

```

Figure 24 : Programme C : Fibonacci

On retrouve bien le code en C extrait du code de l'exécutable.

iii. Écriture d'une fonction de comptage de caractères

Dans cette deuxième étape on nous demande de coder un autre programme qui cherche et compte le nombre de *a*, *b* et *c* dans *EAX*, *EBX* et *ECX*.

On peut ici, utiliser 3 variables locales, une pour chaque lettre. On construit un tableau comme pour l'exercice précédent.

Valeurs	@
c	EBP-12
b	EBP-8
a	EBP-4
Registre EBP	EBP
String	EBP+8

On peut donc facilement coder la partie *count* pour ce programme :

```
count:
    push ebp                ; Sauvegarde de l'adresse de base du pointeur
    mov ebp, esp            ; Chargement de l'adresse du stack dans le registre de base du pointeur
    sub esp, 12             ; Sauvegarde de 12 octets pour les variables locales (4 octets chacun)
    xor eax, eax            ; Initialisation de la variable locale i
    mov [ebp-4], eax        ; Initialisation de la variable locale a
    mov [ebp-8], eax        ; Initialisation de la variable locale b
    mov [ebp-12], eax       ; Initialisation de la variable locale c
    mov eax, [ebp+8]        ; Récupération de l'argument n
    mov ecx, 1              ; Initialisation de la variable locale i
```

Figure 25 : Routine ASM : count initialisation

On peut ensuite créer une boucle qui va récupérer le premier caractère du string, stocké dans *EAX*.

On stocke ce dernier dans un registre de taille 8 bits.

Ensuite vient l'étape de comparaison, la comparaison se fait avec la valeur ASCII de *a*, *b* et *c* :

- Si c'est égal, on incrémente la valeur locale dans la pile et on saute à la fin de la boucle.
- Sinon on passe à la lettre suivante.

On en déduit le programme suivant :

```
for_2:                ; Boucle for
    mov bl, [eax]      ; On récupère le caractère de la chaîne
is_A:                ; Boucle if
    cmp bl, 97         ; i = 'a'
    jne is_B           ; Si i!='a', on passe à la suite
    add [ebp-4], ecx   ; a = a + i
    jmp end_for_2      ; On sort de la boucle for
is_B:                ; Boucle if
    cmp bl, 98         ; i = 'b'
    jne is_C           ; Si i!='b', on passe à la suite
    add [ebp-8], ecx   ; b = b + i
    jmp end_for_2      ; On sort de la boucle for
is_C:                ; Boucle if
    cmp bl, 99         ; i = 'c'
    jne end_for_2      ; Si i!='c', on sort de la boucle
    add [ebp-12], ecx  ; c = c + i
    jmp end_for_2      ; On sort de la boucle
end_for_2:           ; Fin de la boucle for
```

Figure 26 : Routine ASM : count boucle for

À la fin de cette boucle *for_2*, on incrémente *EAX* afin de le faire pointer sur le prochain caractère.
Puis :

- S'il est égal à 0, on remonte dans la boucle.
- Sinon on continue vers la fin de la fonction.

```
inc eax                ; i++
cmp bl,0               ; On vérifie si i<=n
jne for_2              ; Si i<=n, on recommence la boucle
```

Figure 27 : Routine ASM : count vérification terminaison

On charge donc chaque compteur de lettre dans un registre de retour.

Comme pour la fonction précédente, on détruit les variables locales et on restaure l'adresse de retour.

```
mov eax, [ebp-4]       ; Déplacement de a dans le registre de retour eax
mov ebx, [ebp-8]       ; Déplacement de b dans le registre de retour ebx
mov ecx, [ebp-12]      ; Déplacement de c dans le registre de retour ecx
mov esp, ebp           ; Nettoyage des variables locales
pop ebp                ; Restauration de l'adresse de base du pointeur
ret
```

Figure 28 : Routine ASM : count retour

La chaîne de caractères 'TP Marchand/Chapron' contient 3 'a', 0 'b', 1 'c'

Figure 29 : Test count

On trouve bien :

- 3 fois la lettre *a*
- 0 fois la lettre *b*
- 1 fois la lettre *c*

On peut également décompiler le programme en blocs/C pour comprendre d'une manière plus précise comment fonctionne ce dernier et quelles étapes sont appelées.

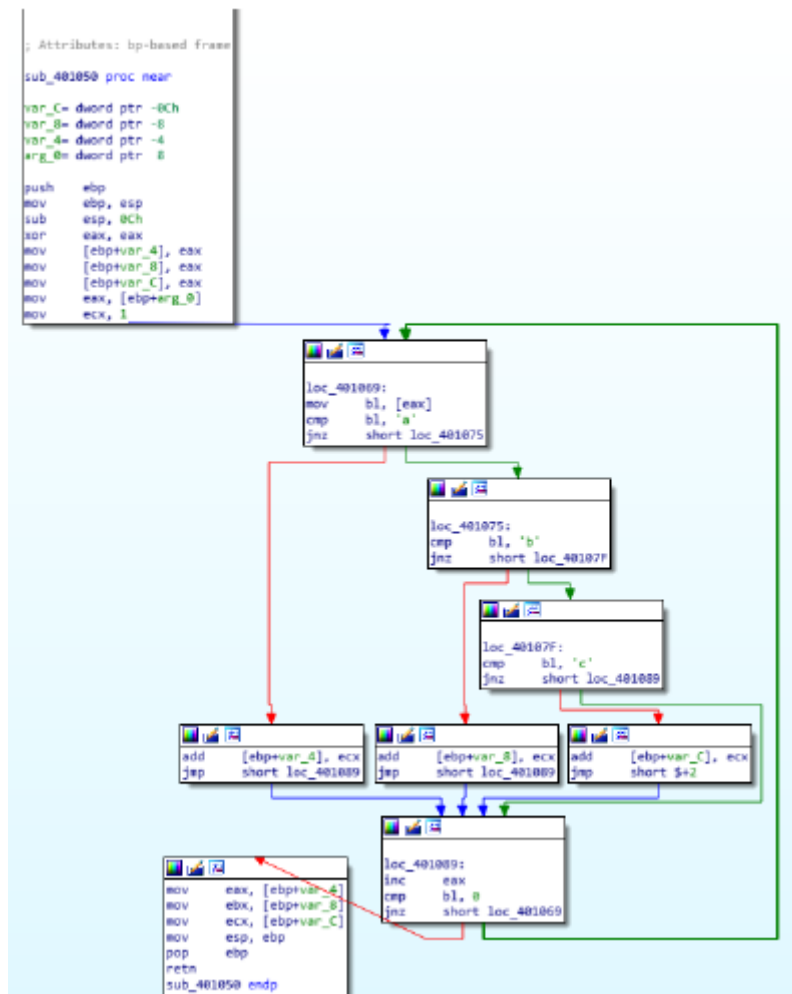


Figure 30 : Blocs ida count

```

int __cdecl FUN_00401050(char *param_1)
{
    char cVar1;
    int local_8;
    local_8 = 0;
    do {
        cVar1 = *param_1;
        if (cVar1 == 'a') {
            local_8 = local_8 + 1;
        }
        param_1 = param_1 + 1;
    } while (cVar1 != '\0');
    return local_8;
}

```

Figure 31 : Programme C : count

Ici on effectue des tests avec la variable *a*

On retrouve bien notre *cmp* :

```
0040106b 80 fb 61      CMP     BL, 0x61
```

Avec 0x61 => ASCII 97 => A

Puis avec 0x62 et 0x63

5. UN PEU DE CALCUL

i. Écriture d'une fonction pour avoir les diviseurs d'un nombre

Pour cette partie nous allons utiliser *scanf* qui est une fonction de la bibliothèque standard du langage C.

Cela va nous permettre de lire une valeur entrée au clavier en lui indiquant la variable qui va stocker la valeur et le format (%d dans notre cas représentant un entier).

nNombre est la variable contenant la valeur entrée au clavier.

.CODE

start:

```
push offset strMsg          ; Empile le message de début.
call crt_printf             ; Affiche "Entrez un nombre : ".

push offset nNombre         ; Premier paramètre de scanf.
push offset strFormat       ; Deuxième paramètre de scanf.
call crt_scanf              ; scanf ("%d",&nNombre).

push offset strMsgSortie    ; Empile le message de sortie.
call crt_printf             ; Affiche "Diviseur du nombre choisi : ".

mov eax, 1                  ; Initialise le compteur.
mov [ebp-4], eax            ; Initialisation de i à 1.
```

Figure 32 : Programme ASM : diviseur initialisation

On place également des *printf* pour afficher les messages dans la console pour l'utilisateur.

On crée ensuite une boucle *for_loop* puis on récupère l'entier *nNombre* et on le met dans le registre *EBX* et on fait une comparaison entre *i* et *EBX*

- Si $i > nNombre$ cela signifie que l'on a essayé tous les nombres plus petits ou égaux à «nNombre»
 - On sort de la boucle et on quitte le programme
- Sinon on divise le *nNombre* par *i* ⇒ on récupère le résultat dans le registre EDX.
- Si *EDX* vaut 0 ⇒ on rentre dans une boucle qui va afficher le diviseur sur la sortie standard.
- Sinon on incrémente *i* et on remonte au début de la boucle.

```

for_loop:                ; Boucle for.
    mov ebx, nNombre      ; On stocke le nombre dans ebx.

    cmp [ebp-4], ebx      ; Si i est plus petit ou égal à n on quitte la boucle.
    jg end_for            ; Sinon on passe à la suite.

    xor edx, edx          ; Initialise le registre EDX avec la valeur 0.
    mov eax, nNombre      ; Ajoute le nombre entré dans le registre EAX.
    mov ecx, [ebp-4]      ; Ajoute i dans le registre ECX.
    div ecx               ; Fait le calcul nNombre mod i et ajoute le résultat dans EDX.

    cmp edx, 0            ; Si le résultat est égal à 0 on passe à la suite.
    je afficher_sortie    ; Si le résultat vaut 0, alors i est un diviseur de nNombre.

increment_i:              ; On incrémente i.
    mov eax, [ebp-4]      ; On récupère i et on le met dans le registre EAX.
    inc eax               ; Incrémente EAX.
    mov [ebp-4], eax      ; On remet la valeur de EAX dans i.
    jmp for_loop          ; Retour dans la boucle pour.

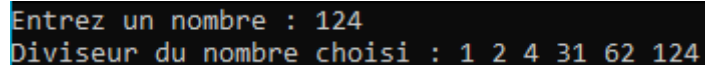
afficher_sortie:          ; Affiche le résultat.
    push [ebp-4]          ; Empile i.
    push offset strSortie ; Empile le message de sortie.
    call crt_printf        ; Affiche un des diviseurs du nombre entré.
    jmp increment_i       ; Retour dans la boucle for.

end_for:
    invoke crt_system, offset strCommand ; Affiche le message de sortie.
    invoke ExitProcess, NULL             ; Quitte le programme.

```

Figure 33 : Programme ASM : diviseur

Le programme final ressemble donc à celui-ci-dessus et voici un test de ce dernier :



```

Entrez un nombre : 124
Diviseur du nombre choisi : 1 2 4 31 62 124

```

Figure 34 : Test diviseur

D'après d'autre algorithme sur internet on trouve bien les mêmes diviseurs de **124** : **1**, **2**, **4**, **31**, **62** et **124**.

On peut également décompiler le programme en blocs/C pour comprendre d'une manière plus précise comment fonctionne ce dernier et quelles étapes sont appelées.

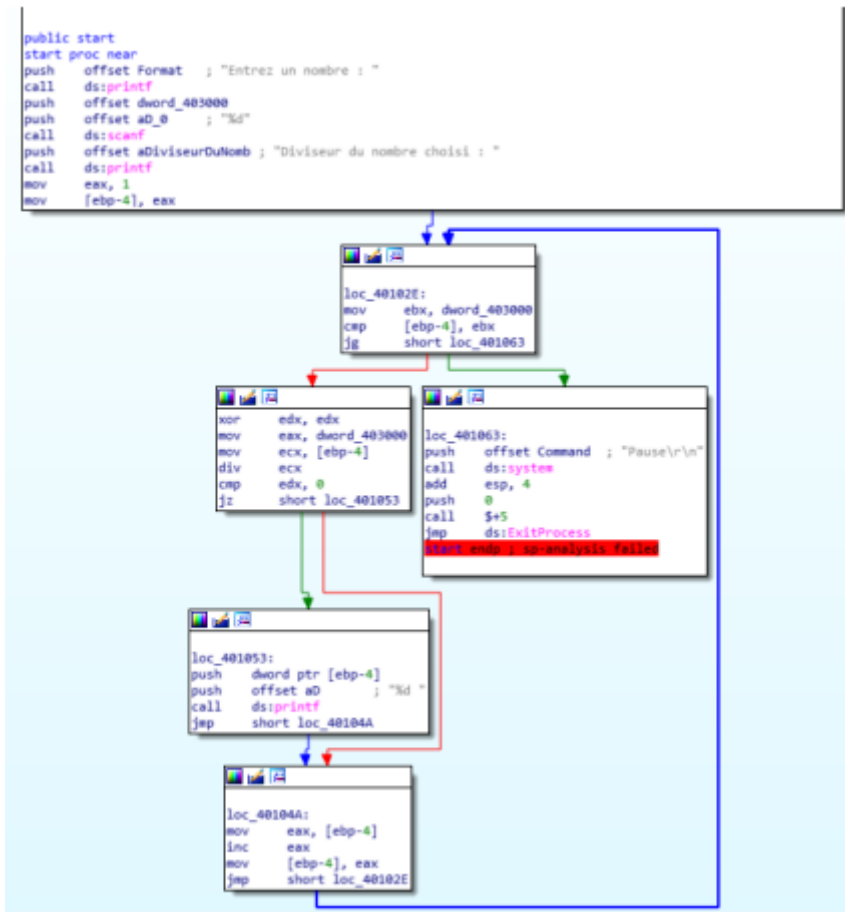


Figure 35 : Blocs ida diviseur

```

void entry(void)
{
    int unaff_EBP;
    printf(s_Entrez_un_nombre_:_00403007);
    scanf(&DAT_00403004,&DAT_00403000);
    printf(s_Diviseur_du_nombre_choisi_:_0040301b);
    *(undefined4 *) (unaff_EBP + -4) = 1;
    while (*(uint *) (unaff_EBP + -4) == DAT_00403000 ||
        (int) *(uint *) (unaff_EBP + -4) < (int) DAT_00403000) {
        if (DAT_00403000 % *(uint *) (unaff_EBP + -4) == 0) {
            printf(&DAT_00403038, *(undefined4 *) (unaff_EBP + -4));
        }
        *(int *) (unaff_EBP + -4) = *(int *) (unaff_EBP + -4) + 1;
    }
    system(s_Pause_0040303c);
    ExitProcess(0);
    return;
}

```

Figure 36 : Programme C : diviseur

ii. Écriture d'une fonction récursive calculant la factorielle

Dans cette dernière question, on nous demande de coder une fonction récursive qui calcule la factorielle d'un entier.

On peut créer une routine *Facto*, sauvegarder l'adresse de retour et la stocker dans la pile.

On récupère par le nombre entrée par l'utilisateur *n* :

- Si *n* > 0 on décrémente *n* et on empile sa valeur sur la pile. Puis on fait un appel récursif à la routine *Facto*.
- Sinon on retourne *EAX* en dépilant *EBP*.

Après chaque retour des appels récursifs, on récupère le *n* actuel ainsi que *Facto(n+1)* qu'on multiplie.

On quitte la fonction en retournant *EAX* et en nettoyant la pile.

```
; Routine Facto
Facto proc ; Routine Facto
    push ebp                ; Sauvegarde de la pile
    mov ebp,esp             ; Initialisation de la pile
    mov eax,[ebp+8]         ; Récupère n

    cmp eax,1               ; n > 0?
    ja continuer            ; True : on continue.
    jmp retour_recur        ; False : retour récursif.

continuer:                  ; On continue
    dec eax                 ; n--
    push eax                ; Empile EAX.
    call Facto              ; Appel récursif de Facto(EAX).

    mov ebx,[ebp+8]         ; Exécuté après tous les retours récursifs de Facto.
    mul ebx                 ; Récupère n.
                             ; EAX = EAX * EBX.

retour_recur:              ; Retour de la fonction.
    pop ebp                 ; Retourne EAX.

    ret 4                   ; Nettoyage de la pile.

Facto endp                  ; Fin de la routine Facto.
```

Figure 37 : Routine ASM : factorielle

On a donc notre routine.

Pour la partie *main* du programme, on va demander à l'utilisateur d'entrer un nombre dans la console.

On récupère cette valeur et on la stocke dans *nNombre*.

On va mettre *nNombre* dans le registre *EAX* et appeler la routine *Facto*.

Une fois la routine réalisée, on va mettre le registre *EAX* contenant le retour de la routine *Facto* ainsi que le message de sortie sur la pile et ainsi appeler *printf* afin d'afficher la factorielle du nombre que l'utilisateur a entré.

```

start:                                ; Routine de départ.
    push offset strMsg                ; Empile le message de début.
    call crt_printf                   ; Affiche "Entrez un nombre : ".

    push offset nNombre               ; Premier paramètre de scanf.
    push offset strFormat             ; Deuxième paramètre de scanf.
    call crt_scanf                    ; scanf ("%d",&nNombre).

    mov eax, nNombre                  ; On met le nombre saisi dans EAX.

    push eax                          ; Empile EAX.
    call Facto                        ; Facto(EAX).

    push eax                          ; Récupère la factorielle calculé .
    push offset strMsgSortie          ; Empile le message de sortie.
    call crt_printf                   ; Affiche "factorielle du nombre choisi : ".

    invoke crt_system, offset strCommand ; Affiche la commande pause.
    invoke ExitProcess, NULL           ; Fin du programme.

end start

```

Figure 38 : Programme ASM : factorielle

On peut également décompiler le programme en blocs/C pour comprendre d'une manière plus précise comment fonctionne ce dernier et quelles étapes sont appelées.

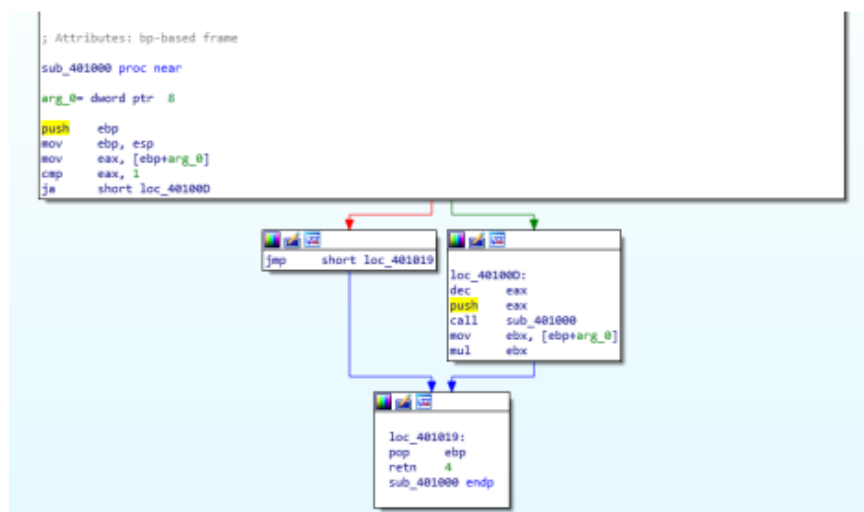


Figure 39 : Blocs ida factorielle

```

void entry(void)
{
    undefined4 extraout_ECX;
    undefined4 extraout_EDX;
    undefined8 uVar1;
    printf(s_Entrez_un_nombre_:_00403007);
    scanf(&DAT_00403004,&DAT_00403000);
    uVar1 = FUN_00401000(extraout_ECX,extraout_EDX,DAT_00403000);
    printf(s_Factorielle_du_nombre_choisi_:_%_0040301b,(int)uVar1);
    system(s_Pause_00403040);
    ExitProcess(0);
    return;
}

```

Figure 40 : Programme C : factorielle

On a bien l'appel à la fonction *FUN_00401000* :

```
{
  longlong lVar1;
  ulonglong uVar2;
  if (1 < param_3) {
    uVar2 = FUN_00401000(param_1,param_2,param_3 - 1);
    lVar1 = (uVar2 & 0xffffffff) * (ulonglong)param_3;
    param_2 = (undefined4)((ulonglong)lVar1 >> 0x20);
    param_3 = (uint)lVar1;
  }
  return CONCAT44(param_2,param_3);
}
```

Figure 41 : Programme C : factorielle FUN_00401000

6. UN PEU DE LECTURE

i. Analyse de l'article

Le processeur possède un niveau de privilège *ring* codé sur 2 bits.

4 valeurs possibles : *4*, *3*, *2* et *1*. Plus le *ring* est petit, plus les privilèges sont élevés.

Donc plus le *ring* est grand, les actions sont plus restreintes.

Le *ring* 0 est assez connu puisqu'il s'agit du mode superviseur.

Le niveau courant du processeur *CPL* est stocké sur les deux premiers bits des registres *CS* et *SS*.

Règle principale : On ne peut pas exécuter des instructions de niveaux inférieur à celui du *CPL*. Comment changer le *CPL* lors d'exécution de code utilisateur pour appeler les instructions *Kernel* ?

Pour comprendre comment se fait la transition vers le mode *Noyau*, l'auteur de l'article réalise un programme en C contenant un appel système.

Il va ensuite compiler puis débbugger le programme à l'aide d'*OllyDbg*. Il remonte chaque appel. La fonction permettant de rentrer le mode *Noyau* est *KiFastSystemCall*, elle prend en paramètre le numéro de la fonction noyau qui a besoin d'être appelé.

En regardant le code de la fonction *KiFastSystemCall*. On se rend compte que c'est l'instruction *SYSENTER* qui réalise l'entrée. Ce n'est pas une interruption matérielle ! *SYSENTER* n'utilise pas de table de pointeurs mais possède des registres spécifiques : *MSR*. Ils définissent tout ce qu'il faut pour faire un appel système. Ils sont accessibles en lecteur écriture avec les instructions *rdmsr*, *wrmsr*.

Une fois les paramètres vérifiés, la fonction *KiFastCallEntry* permet d'exécuter la fonction souhaitée avec l'indice d'appel système mis en paramètre de *KiFastSystemCall*.

Pour retourner dans l'userland, l'appel à *KiServiceExit* va être fait, à la fin de *KiFastCallEntry*, et va lui-même appeler *kiSystemCallExit* ou *kiSystemCallExit2*.

III. CONCLUSION

C'est avec l'analyse de cet article que s'achève ce TP dans la matière "Langage d'assemblage et microprocesseur". Ce TP nous a permis de remettre à jour des concepts oubliés en code bas niveau. Si on devait refaire ce TP on prendrait plus de temps pour faire une partie qui vérifie l'entrée de l'utilisateur (affiche un message d'erreur si l'entrée est un string pour diviseur ou factorielle). Ce fût un TP très prenant et intéressant à notre sens et à le mérite d'exister.