Architecture des ordinateurs

ESIPE - IR1 2015-2016

Fiche de TP 1

Introduction à la programmation en assembleur

Table des matières

1 Le processeur et les registres **2**
- 1.1 Opérations sur les registres 3
- 1.2 Lecture-écriture en mémoire 4

2 Premier programme **7**
- 2.1 Explication du code 7
- 2.2 Assemblage du programme 10

3 Débogage **12**
- 3.1 Déboguer avec `Gdb` 12
- 3.2 Environnement d'exécution 14

Le but de ce TP est d'introduire les bases de la programmation en assembleur sur des processeurs 32 bits d'architecture x86, sous Linux, et en mode protégé. L'assembleur Nasm (Netwide Assembler) sera utilisé en ce but.

Cette fiche est à faire en deux séances (soit 4 h), et en binôme. Il faudra :

1. réaliser un — bref — rapport à rendre **au format pdf** contenant les réponses aux questions de cette fiche ;
2. écrire les — différents — fichiers sources des programmes demandés. Veillez à nommer correctement vos fichiers sources. Les sources des programmes doivent **impérativement** être des fichiers compilables par Nasm ;
3. réaliser une archive **au format zip** contenant les sources des programmes et le rapport. Le nom de l'archive doit être sous la forme `IR1_Archi_TP1_NOM1_NOM2.zip` ;
4. déposer l'archive sur la plate-forme de rendu.

Tous les fichiers complémentaires nécessaires à ce TP se trouvent sur la plate-forme.

Le livre de Paul Carter *PC Assembly Language* est disponible gratuitement à l'adresse

http://www.drpaulcarter.com/pcasm/

Il est important de le consulter au fur et à mesure de l'avancement du TP.

La première partie du sujet présente l'architecture 32 bit de type x86 : les registres, les opérations sur les registres et les lectures/écritures en mémoire. La deuxième partie présente un premier programme écrit en Nasm. Elle décrit comment compiler un programme. Enfin, la troisième partie explique comment le déboguer. Elle présente également en détail ce qui se passe lorsqu'un programme est chargé en mémoire sous Linux.

1 Le processeur et les registres

Rappel 1. La figure 1 répertorie les noms des tailles de données habituelles.

Type	Taille en octets	Taille en bits
byte (ou octet)	1	8
word	2	16
dword	4	32

FIGURE 1 – Taille des types de données usuels.

Les processeurs 32 bits d'architecture x86 travaillent avec des registres qui sont en nombre limité et d'une capacité de 32 bits (soit 4 octets). Parmi ces registres, les registres appelés `eax`, `ebx`, `ecx`, `edx`, `edi` et `esi` sont des registres à usage général. Les registres `esp` et `eip` servent respectivement à conserver l'adresse du haut de la pile et l'adresse de l'instruction à exécuter. Ces derniers registres seront étudiés et exploités dans un prochain TP. Les registres `eax`, `ebx`, `ecx` et `edx` sont subdivisés en sous-registres. La figure 2 montre la subdivision de `eax` en `ax`, `ah` et `al`. Le premier octet (celui de poids le plus faible) de `eax` est accessible

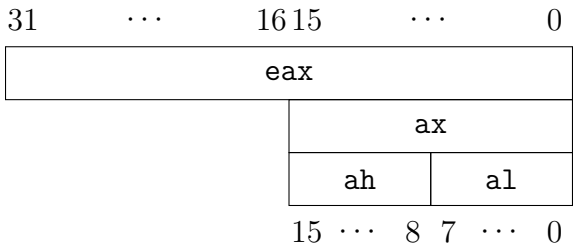

FIGURE 2 – Subdivision du registre `eax`.

par le registre `al` (de capacité 8 bits), le deuxième octet de poids le plus faible est accessible par le registre `ah`. Les 16 bits de poids faible de `eax` sont accessibles par le registre `ax` (qui recouvre `al` et `ah`). Noter que les 2 octets de poids fort ne sont pas directement accessibles par un sous-registre. De même pour `ebx`, `ecx`, et `edx`, on dispose des registres analogues `bx`, `bh`, `bl`, `cx`, `ch`, `cl` et `dx`, `dh`, `dl`.

Important 1. Lorsque l'on modifie le registre `al`, les registres `ax` et `eax` sont eux aussi modifiés.

Question 1. Quelles sont les valeurs minimales et maximales qui peuvent être enregistrées respectivement dans les registres `eax`, `ax`, `ah` et `al` ? Les valeurs données doivent être exprimées en décimal non signé, en décimal signé et en hexadécimal (non signé).

1.1 Opérations sur les registres

Nous allons présenter les opérations permettant d'affecter une valeur à un registre (instruction `mov`) et d'effectuer des calculs (par exemple, instructions `add` et `sub`).

Important 2. Dans la syntaxe de Nasm, c'est toujours le premier argument qui reçoit la valeur du calcul. C'est la syntaxe Intel (il existe d'autres assembleurs qui emploient la convention inverse, on parle alors de la syntaxe ATT). Voici quelques exemples d'écriture dans des registres :

```
1       mov eax, 3              ; eax = 3
2       mov eax, 0x10a          ; eax = 0x10a = 266
3       mov eax, 0b101          ; eax = 0b101 = 5
4       mov ebx, eax            ; ebx = eax
5       mov ax, bx              ; ax = bx
```

Remarque 1.
1. Il existe plusieurs formats pour spécifier une valeur. Par défaut, le nombre est donné en décimal. Une valeur préfixée par `0x` correspond à un nombre donné en hexadécimal. Ainsi `0x1A` correspond en décimal à 26. Le préfixe `0b` correspond au binaire.
2. On ne peut pas copier la valeur d'un registre dans un registre de taille différente. Ainsi, l'instruction `mov eax, bx` produira une erreur.
3. Google est utile pour faire facilement les conversions. Par exemple, la recherche `0x1a in decimal` retournera `0x1a = 26`.

Astuce 1. Connaissant la valeur de `eax` en **hexadécimal**, il est facile de connaître la valeur de `ax`, `ah` et `al`. Par exemple, si `eax = 0x01020304` alors `ax = 0x0304`, `ah = 0x03` et `al = 0x04`.

Question 2. Quelles sont les valeurs (en hexadécimal) de `eax`, `ax`, `ah` et `al` après l'instruction `mov eax, 134512768` ? Quelles sont ensuite les valeurs (en hexadécimal) de `eax`, `ax`, `ah` et `al` après l'instruction `mov al, 0` ?

Les instructions `add` et `sub` ont le comportement attendu : le résultat de l'addition et de la soustraction est écrit dans le premier argument. En cas de dépassement de la capacité, la retenue est signalée par le drapeau `carry` du processeur (ceci sera vu en détail dans un prochain TP). Nous avons ainsi par exemple,

```
1       add eax, 3              ; eax = eax + 3
2       add ah, 0x1c            ; ah = ah + 28
3       add ebx, ecx            ; ebx = ebx + ecx
4       sub eax, 10             ; eax = eax - 10
5       add ah, al              ; ah = ah + al
```

Question 3. Après l'exécution des instructions suivantes, quelles sont les valeurs des registres `eax`, `ax`, `ah` et `al` ?

```
1       mov eax, 0xf00000f0
2       add ax, 0xf000
3       add ah, al
```

1.2 Lecture-écriture en mémoire

Important 3. En mode protégé, la mémoire peut être vue comme un tableau de 2^{32} cases contenant chacune *un octet*. Le numéro (ou l'indice) d'une case est appelé son *adresse*. La mémoire est représentée par un tableau vertical dont les cases indexées de la plus petite adresse à la plus grande. Une adresse est codée sur 32 bits. Le contenu des registres 32 bits (comme `eax`, `ebx`, *etc.*) peut représenter un nombre ou adresse en mémoire.

La figure 3 illustre un exemple fictif d'état de la mémoire.

Question 4. Combien de valeurs différentes peut représenter une case de la mémoire ?

Question 5. Quelle est la quantité (exprimée en gibioctets) de mémoire adressable sur 32 bits ?

Question 6. Combien de cases se situent avant la case d'indice `0x0000100a` dans la mémoire ?

Adresse	Valeur
0x00000000	3
0x00000001	30
0x00000002	90
0x00000003	10
0x00000004	16
0x00000005	9
⋮	⋮
0x00000010	127
⋮	⋮
0xfffffffe	30
0xffffffff	3

FIGURE 3 – Exemple d'état de la mémoire.

1.2.1 Lecture en mémoire

La syntaxe générale pour lire la mémoire à l'adresse `adr` et stocker la valeur dans le registre `reg` est la suivante (les crochets font partie de la syntaxe) :

```
1       mov reg, [adr]
```

Le nombre d'octets lus dépend de la taille de `reg`. Par exemple, 1 octet sera lu pour `al` ou `ah`, 2 octets seront lus pour `ax` et 4 pour `eax`. Un autre exemple :

```
1       mov al, [0x00000003]  ; al reçoit l'octet stocké à l'adresse 3
2                             ; dans l'exemple, al = 10 = 0x0a;
3       mov al, [3]           ; Instruction équivalente à la précédente.
```

Question 7. Expliquer la différence entre `mov eax, 3` et `mov eax, [3]`.

Important 4. Quand on lit plus d'un octet, il faut adopter une convention sur l'ordre dont les octets sont rangés dans le registre. Il existe deux conventions *little-endian*(*petit boutisme*) et *big-endian* (*grand boutisme*). La convention employée dépend du processeur. Pour nous se sera toujours *little-endian*.

Considérons par exemple l'instruction suivante, avec la mémoire dans l'état représenté par la figure 3 :

```
1       mov eax, [0x00000000]
```

Le nombre d'octets lus dépend de la taille du registre. Ici on va lire les 4 octets situés aux adresses 0, 1, 2 et 3. Dans l'exemple de mémoire, ces octets valent respectivement 3 (= `0x03`), 30 (= `0x1e`), 90 (= `0x5a`) et 10 (= `0x0a`). Les deux choix possibles pour les ranger dans `eax` sont :

`eax` = `0x0a5a1e03` convention *little-endian*
`eax` = `0x031e5a0a` convention *big-endian*

Important 5. Les processeurs Intel et AMD utilisent la convention *little-endian*. Dans cette convention, les octets situés aux adresses basses deviennent les octets de poids faible du registre.

Au lieu de donner explicitement l'adresse où lire les données, on peut lire l'adresse depuis un registre. Ce registre doit nécessairement faire 4 octets. Par exemple,

```
1       mov eax, 0      ; eax = 0
2       mov al, [eax]   ; al reçoit l'octet situé à l'adresse contenue dans eax
3                       ; dans l'exemple, al = 0x03.
```

Question 8. Dans l'exemple de mémoire de la figure 3, quelle est la valeur de `ax` après l'instruction `mov ax, [1]` ?

Question 9. Quelle est la valeur de `eax` à la fin de la suite d'instructions suivante, en supposant que la mémoire est dans l'état de la figure 3 ?

```
1       mov eax, 5
2       sub eax, 1
3       mov al, [eax]
4       mov ah, [eax]
```

1.2.2 Écriture en mémoire

La syntaxe générale pour écrire en mémoire à l'adresse `adr` la valeur du registre `reg` est la suivante (les crochets font partie de la syntaxe) :

```
1       mov [adr], reg
```

L'écriture suit la même convention que la lecture (*little-endian* en ce qui nous concerne).

Question 10. Quel est l'état de la mémoire après l'exécution de la suite d'instructions qui suit ?

```
1       mov eax, 0x04030201
2       mov [0], eax
3       mov [2], ax
4       mov [3], al
```

On peut aussi directement affecter une valeur en mémoire sans la stocker dans un registre. Il faut alors préciser la taille des données avec les mots-clés `byte` (1 octet), `word` (2 octets) et `dword` (4 octets).

Question 11. Quel est l'état de la mémoire après avoir effectué la suite d'instructions suivante ?

```
1       mov dword [0], 0x020001
2       mov byte [1], 0x21
3       mov word [2], 0x1
```

Question 12. Quelle est la valeur de `eax` à la fin de la suite d'instructions suivante dans la convention *little-endian* et *big-endian* ?

```
1       mov ax, 0x0001
2       mov [0], ax
3       mov eax, 0
4       mov al, [0]
```

2 Premier programme

Notre premier programme `Hello.asm` affiche le traditionnel message `Hello world`. Pour le compiler, saisir dans un terminal et dans le répertoire contenant le fichier `Hello.asm` les commandes

```
1       nasm -g -f elf32 Hello.asm
2       ld -o Hello -e main Hello.o
```

La première commande crée un fichier objet `Hello.o` et la dernière réalise l'édition des liens pour obtenir un exécutable `Hello`. Par ailleurs,

- `-f elf32` est une option d'assemblage. Elle permet à Nasm de fabriquer du code objet 32 bit pour Linux.
- `-g` permet d'ajouter au programme des informations qui serviront au débogueur.

Pour exécuter le programme, il suffit de lancer la commande `./Hello`.

2.1 Explication du code

Nous allons maintenant expliquer en détail la programme dont le contenu est donné ci-dessous.

```
1       ; Premier programme en assembleur
2
3       SECTION .data ; Section des données.
4       msg :
5           db "Hello World", 10 ; La chaîne de caractères a afficher,
6                                ; 10 est le code ASCII du retour a la ligne.
7
8       SECTION .text            ; Section du code.
9       global main              ; Rend l'étiquette visible de l'extérieur.
10      main :                   ; Étiquette pointant au début du programme.
11          nop                  ; Instruction vide
12          mov edx, 0xc         ; arg3, nombre de caractères a afficher
13                               ; (équivalent a mov edx, 12).
14          mov ecx, msg         ; arg2, adresse du premier caractère
15                               ; à afficher.
16          mov ebx, 1           ; arg1, numéro de la sortie pour l'affichage,
17                               ; 1 est la sortie standard.
18          mov eax, 4           ; Numéro de la commande write pour
19                               ; l'interruption 0x80.
20          int 0x80             ; Interruption 0x80, appel au noyau.
21          mov ebx, 0           ; Code de sortie, 0 est la sortie normale.
22          mov eax, 1           ; Numéro de la commande exit.
23          int 0x80             ; Interruption 0x80, appel au noyau.
```

2.1.1 Structure du programme

Le programme est composé de deux sections :

Section .data (lignes 3 – 6) contient les données du programme, dans cet exemple, la chaîne de caractères à afficher.

Section .text (lignes 8 – 23) contient le code du programme.

Lors de son exécution, le programme est chargé en mémoire comme décrit dans la figure 4. L'adresse à laquelle débute le code est toujours la même : `0x08048080`. En revanche, l'adresse à laquelle commence les données dépend de la taille du code (plus le code est long, plus l'adresse de la case à laquelle commence les données est élevée).

Important 6. Lors de l'écriture d'un programme, l'adresse à laquelle est stockée la chaîne de caractères n'est pas connue. Il est difficile de la connaître car c'est le système qui l'attribue. C'est pour remédier à ce problème que l'on met l'étiquette `msg` au début de la ligne 2. Dans le programme, `msg` représentera l'adresse du premier octet la chaîne de caractères.

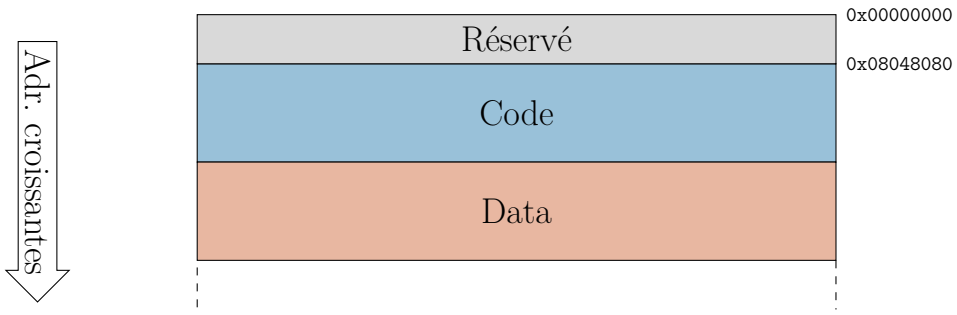

FIGURE 4 – Mémoire occupée par le programme lors de son exécution.

2.1.2 Section .data

La primitive `db` permet de déclarer une suite d'octets. La chaîne de caractères donnée entre guillemets correspond simplement à la suite des octets ayant pour valeur le code ASCII des caractères. Par exemple, le code ASCII de `'H'` est 72 et celui de `'e'` est 101. On aurait pu écrire, de manière équivalente,

```
1       msg : db 72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 10
```

ou bien encore

```
1       msg : db 0x48,0x65,0x6c,0x6c,0x6f,0x20,0x57,0x6f,0x72,0x6c,0x64,0x0a
```

2.1.3 Section .text

La section `.text` contient le code. L'instruction de la ligne 9 permet de rendre visible l'étiquette `main`, et en particulier, dans `Hello.o`. On peut ainsi y faire référence dans la commande `ld -o Hello -e main Hello.o` pour signaler que le point d'entrée du programme est cette étiquette.

Le premier bloc, allant de la ligne 11 à la ligne 19, permet d'afficher la chaîne de caractères et le deuxième bloc, allant de la ligne 20 à la ligne 22 permet de quitter le programme.

Il n'y pas dans le jeu d'instructions du processeur d'instruction spécifique pour afficher un message. Pour afficher un message à l'écran, il faut faire appel au système d'exploitation (dans notre cas, le noyau Linux). C'est le rôle de l'instruction `int 0x80`. Cette opération va passer la main au noyau pour réaliser une tâche. Une fois cette tâche réalisée, le noyau reprend l'exécution du programme à la ligne suivant l'interruption. Ce procédé est appelé un *appel système*. L'instruction `int 0x80` peut servir à faire toutes sortes de tâches : quitter le programme, supprimer un fichier, agrandir la taille du tas, exécuter un autre programme. Le noyau détermine la tâche à accomplir en regardant la valeur des registres `eax`, `ebx`, *etc.*, qui jouent alors le rôle de paramètres. En particulier, la valeur du registre `eax` correspond au numéro de la tâche à accomplir. Par exemple, `1` correspond à `exit` et `4` correspond à `write`.

Pour connaître le numéro de chacun des appels systèmes, il faut regarder dans les sources du noyau. Par exemple pour le noyau 2.6, on le trouve dans le fichier `unistd_32.h` qui commence par les lignes suivantes :

```
1       #define __NR_restart_syscall 0
2       #define __NR_exit 1
3       #define __NR_fork 2
4       #define __NR_read 3
5       #define __NR_write 4
6       #define __NR_open 5
7       #define __NR_close 6
8       #define __NR_waitpid 7
9       #define __NR_creat 8
10      #define __NR_link 9
11      #define __NR_unlink 10
12      #define __NR_execve 11
13      #define __NR_chdir 12
14      #define __NR_time 13
15      #define __NR_mknod 14
16      #define __NR_chmod 15
17      #define __NR_lchown 16
18      #define __NR_break 17
```

Pour l'appel système `write`, les registres `ebx`, `ecx`, `edx` sont utilisés comme suit :

- `ebx` contient le numéro de la sortie sur laquelle écrire. La valeur `1` correspond à la sortie standard.
- `ecx` contient l'adresse mémoire du premier caractère à afficher
- `edx` contient le nombre total de caractères à afficher.

Si l'on revient sur le premier bloc du code (lignes 11 – 19), on demande au système d'afficher (car `eax = 4`) sur la sortie standard (car `ebx = 1`) les 12 caractères (car `ecx = 12`) commençant à l'adresse `msg`. Le deuxième bloc (ligne 20 – 22) quitte le programme avec l'appel système `exit` qui a pour code `1`. Le code de retour est donné dans `ebx`. La valeur `0` signifiant qu'il n'y a pas eu d'erreur.

Question 13. Modifier `Hello.asm` pour afficher `L'ordinateur va s'eteindre !`. Le nouveau programme doit s'appeler `Q13.asm`.

Complément 1. Pour connaître les paramètres attendus par un appel système, il est possible d'utiliser la section 2 des pages de `man`. Par exemple, il faut taper `man 2 write` pour l'appel système `write`. La documentation donne le prototype de l'appel système pour le langage `C`. On peut l'adapter à l'assembleur en sachant que le premier argument va dans `ebx`, le second `ecx`, et ainsi de suite.

2.2 Assemblage du programme

Dans ce paragraphe, nous allons voir en détail comment le code de l'exécutable est assemblé en partant de code du programme. En utilisant la commande

```
1       nasm   Hello.asm -l Hello.lst -g -f elf -F dwarf
```

on obtient dans `Hello.lst` un listing donnant le code machine des différentes instructions. Le voici :

```
1         1
2         2
3         3                                  SECTION .data
4         4                                  msg :
5         5 00000000 48656C6C6F20576F72-         db "Hello World", 10
6         6 00000009 6C640A
7         7
8         8
9         9                                  SECTION .text
10       10                                  global main
11       11                                  main :
12       12 00000000 BA0C000000                 mov edx, 0xc
13       13
14       14 00000005 B9[00000000]               mov ecx, msg
15       15
16       16 0000000A BB01000000                 mov ebx, 1
17       17
18       18 0000000F B804000000                 mov eax, 4
19       19
20       20 00000014 CD80                       int 0x80
21       21 00000016 BB00000000                 mov ebx, 0
22       22 0000001B B801000000                 mov eax, 1
23       23 00000020 CD80                       int 0x80
```

À cette étape, on connaît les octets correspondant à la section `.data`. En hexadécimal, cela donne : `48656C6C6F20576F726C640A`.

Les octets correspondant aux différentes instructions sont connus. Seules les adresses des labels ne sont pas connues ; pour l'instant, l'adresse `msg` ne l'est pas. En particulier, l'instruction `mov ecx, msg` est codée par `B9[00000000]`. Les crochets signifient que l'adresse n'est pas encore finalisée.

On remarque que chaque instruction est précédée de son adresse. On peut par ailleurs déduire la taille en octets d'une instruction en regardant la 3^{e} colonne du `Hello.lst` puisque celle-ci contient son codage machine. Par exemple, en ligne 18, l'instruction `mov eax, 4` commence à l'adresse `0x0F` mémoire et se traduit en `0xB804000000`. Elle tient donc sur cinq octets.

Ceci étant dit, mesurons la taille de notre code. La dernière instruction est à l'adresse `0x20` et tient sur deux octets. La dernière adresse utilisée par le code est donc `0x21`, ce qui représente 33 en décimal. Notre code s'étend donc sur $34 = 33 + 0 + 1$ octets. Cependant, pour des raisons d'alignement, tout code doit toujours occuper un nombre d'octets multiple de 4. Notre code occupera 36 octets (les deux octets de remplissage valent `0`).

Le calcul de l'adresse des étiquettes est réalisé par `ld`. Reprenons la figure précédente. Le bloc de données va commencer à l'adresse : `0x08048080` + 36 = `0x080480a4`. L'adresse `msg` est donc `0x080480a4`. L'instruction `mov ecx, msg` sera codée par `B9a4800408`.

```
BA 0C 00 00 00 B9 a4 80 04 08
BB 01 00 00 00 B8 04 00 00 00
CD 80 BB 00 00 00 00 B8 01 00
00 00 CD 80 00 00 48 65 6C 6C
6F 20 57 6F 72 6C 64 0A
```

Question 14. Commenter les différences (éventuelles) entre les fichiers `Hello.lst` et `Q13.lst` en les expliquant.

3 Débogage

3.1 Déboguer avec Gdb

Nous allons utiliser le débogueur Gdb qui permet de suivre pas à pas l'exécution d'un programme et de connaître les valeurs dans la mémoire et dans les registres. Un système de déboguage est indispensable dans un programme en assembleur étant donné que l'affichage d'une donnée n'est pas simple à réaliser.

Pour déboguer notre programme `Hello.asm`, nous suivant la procédure suivante :

1. lancer `gdb Hello`. Le débogueur est maintenant lancé sur le programme.
2. `Gdb` est capable de désassembler le programme grâce à la commande `disassemble main`. L'affichage par défaut ne suit pas la syntaxe que nous utilisons, pour changer cela, on utilise la commande `set disassembly-flavor intel`.
3. Avant de lancer le programme, il faut spécifier des points d'arrêt. On peut indiquer un point d'arrêt par `break *label +n` où `n` désigne un nombre d'octets. Avec `disassemble main`, on peut savoir où se situent les instructions par rapport à l'étiquette `main`. Un programme lancé par `gdb` va stopper au niveau d'un point d'arrêt *avant* d'exécuter l'instruction. Par ailleurs, `Gdb` ne permet pas qu'on arrête le programme avant la première instruction ; pour cette raison, il faut ajouter une instruction vide `nop` en début de programme.
4. Créer un point d'arrêt au début du programme avec l'instruction `break *main+1`, puis lancer le programme avec `run`. Le programme s'arrête après l'instruction vide.

La table 1 contient une description brève des commandes les plus utiles de Gdb.

Question 15. Qu'affiche — textuellement — `info registers` ? A quoi sert cette commande ?

Commande	Déscription
`run`	Démarre le programme
`quit`	Quitte Gdb
`cont`	Continue l'exécution jusqu'au prochain point d'arrêt
`break [addr]`	Ajoute un point d'arrêt à l'adresse `addr`
`delete [n]`	Supprime le `n`e point d'arrêt
`delete`	Supprime tous les points d'arrêt
`info break`	Liste tous les points d'arrêt
`step` ou `stepi`	Exécute la prochaine instruction
`stepi [n]`	Exécute les `n` prochaines instructions
`next` ou `nexti`	Exécute la prochaine instruction, sans compter le corps des fonctions
`nexti [n]`	Exécute les `n` prochaines instructions, sans compter le corps des fonctions
`where`	Montre où l'exécution s'est arrêtée
`list`	Affiche quelques ligne du fichier source
`list [n]`	Affiche quelques ligne du fichier source à partir de la ligne `n`
`info registers`	Affiche les contenus des registres
`print/d [expr]`	Affiche `expr` en décimal
`print/x [expr]`	Affiche `expr` en hexadécimal
`print/t [expr]`	Affiche `expr` en binaire
`x/NFU [addr]`	Affiche le contenu de la mémoire à l'adresse `addr` au format donné
`display [expr]`	Affiche le contenu de `expr` à chaque arrêt du programme
`info display`	Donne la liste des affichages automatiques
`undisplay [n]`	Supprime le `n`e affichage automatique
`disas [addr]`	Désassemble à partir de l'adresse `addr`

TABLE 1 – Les principales commandes de Gdb.

Il est aussi possible de n'afficher qu'un seul registre grâce à `print/x $eax`, on utilise la lettre `x` pour un affichage en hexadécimal, `d` pour un affichage décimal ou `t` pour un affichage binaire.

L'instruction `x/12cb` permet l'affichage de la mémoire à partir d'une adresse donnée. Les informations après le `/` signifient que l'on affiche 12 éléments en format ASCII (caractère `c`) chacun de taille 1 octet (caractère `b`). Les autres formats d'affichage sont `d` pour décimal, `x` pour hexadécimal et `t` pour binaire. Les autres tailles possible sont `h` pour 2 octets et `w` pour 4 octets.

Question 16. Qu'affiche `x/12cb &msg` ?

Question 17. Expliquer la différence d'affichage entre `x/3xw &msg` et `x/12xb &msg`. Comment afficher uniquement le premier caractère de `Hello World` ? Comment afficher uniquement le 3e caractère de `Hello World` ?

Question 18. Utiliser la commande `next` (ou `nexti`) pour continuer l'exécution du programme ligne à ligne. Après tout modification d'un registre, afficher sa valeur grâce à `print`.

Question 19. Refaire le même travail sur le programme `Myst.asm`. À chaque étape, afficher la valeur des registres modifiés ou de l'espace mémoire modifié.

3.2 Environnement d'exécution

Nous allons utiliser `gdb` pour explorer la mémoire autour de notre programme. L'organisation de la mémoire pour un programme en cours d'exécution est illustrée dans la figure 5. La partie *Environnement* contient les arguments passés au programme ainsi que les variables

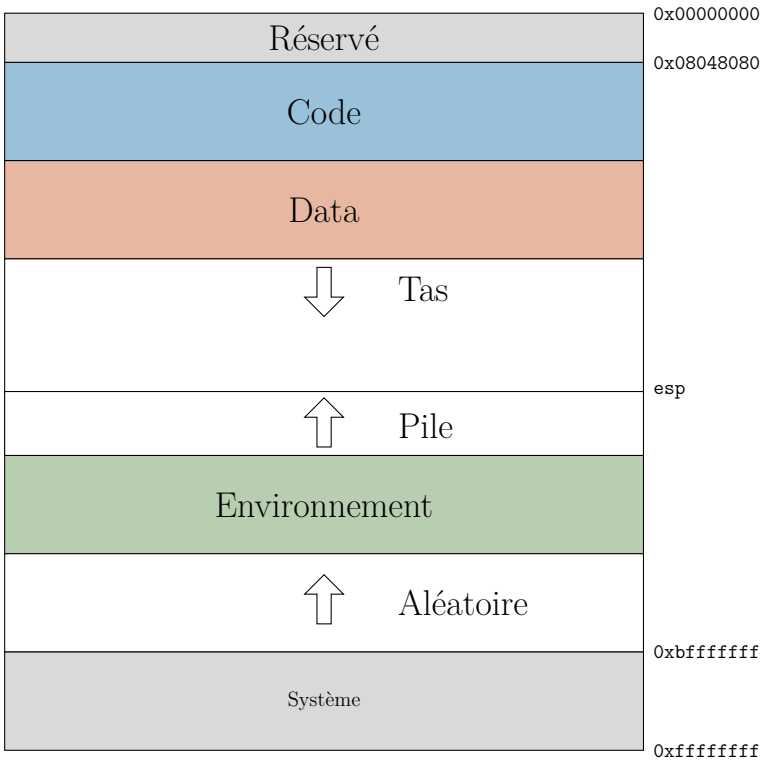

FIGURE 5 – Organisation de la mémoire sous Linux en mode protégé.

d'environnement. Pour passer des arguments au programme à débuguer, on les ajoute après `run` dans `gdb`.

Toutes les valeurs sont sur 4 octets. `Argc` est le nombre d'arguments du programme majoré de 1. La valeur `Arg[0]` est l'adresse du premier caractère d'une chaîne terminé par un `0` qui est le nom du programme. La valeur `Arg[1]` est l'adresse d'une chaîne correspondant au premier argument.

esp	Argc
esp+ 4	Arg[0]
esp+ 8	Arg[1]
⋮	⋮
esp+ 4n + 4	Arg[n]
esp+ 4n + 8	0
esp+ 4n + 12	Env[0]
⋮	⋮

Question 20. Utiliser `gdb` pour accéder à ces différentes valeurs après avoir ajouté deux arguments au programme. Retrouver les chaînes de caractères associées en mémoire.