

Architecture des ordinateurs

IR1 2011-2012

TP 1

Objectif de cette séance

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 binôme. Il faudra :

1. réaliser un — bref — rapport à rendre **au format pdf** contenant les réponses aux questions de cette fiche ;
2. réaliser une archive **au format zip** contenant les sources des programmes ;
3. envoyer le tout à l'adresse de votre chargé de TP

Samuele.Giraud@univ-mlv.fr ou vcarnino@gmail.com

par un courriel dont le sujet est sous la forme

IR1 Archi TP1 *NOM1* ; *NOM2*

Nasm est téléchargeable gratuitement à l'adresse

<http://sourceforge.net/projects/nasm>.

L'excellent livre (traduit en français) de Paul Carter intitulé *PC Assembly Language* est disponible gratuitement à l'adresse suivante :

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

Tous les fichiers nécessaires pour les séances de travaux pratiques se trouvent à l'adresse

<http://igm.univ-mlv.fr/~giraud/Enseignements/IR1/Archi/Archi.html>.

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.

Part I

Le processeur et les registres

Rappel 1.

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. Les registres `eax`, `ebx`, `ecx` et `edx` sont subdivisés en sous-registres. La figure suivante montre la subdivision de `eax` en `ax`, `ah` et `al`. Le premier octet (celui de poids le plus faible) de `eax` est accessible 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`.

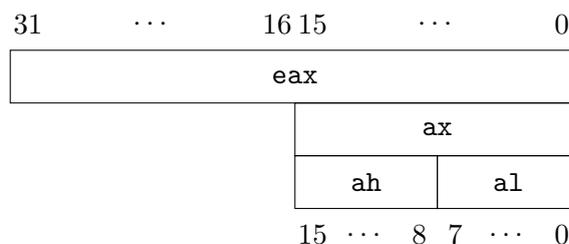


Figure 2: Subdivisions du registre `eax`.

Question 1. Quelles sont les valeurs maximales (en décimal et en hexadécimal) qui peuvent être stockées respectivement dans les registres `eax`, `ax`, `ah` et `al` ?

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

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` ?

Les instructions **add** et **sub** ont le comportement attendu : le résultat de l'addition et de la soustraction est stocké dans le premier paramètre. En cas de dépassement de la capacité, la retenue est marquée dans le drapeau **Carry** du processeur.

```
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 de **eax**, **ax**, **ah** et **al** ?

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

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 d'une case est appelé son *adresse*. Une adresse est codée sur 32 bits. Le contenu des registres 32 bits (comme **eax**, **ebx**, ...) peut représenter un nombre ou une adresse en mémoire.

La figure suivante donne un exemple fictif d'état de la mémoire :

Adresse	Valeur
0xffffffff	4
0xffffffe	8
⋮	⋮
0x00000005	124
0x00000004	2
0x00000003	10
0x00000002	90
0x00000001	30
0x00000000	3

Figure 3: Exemple d'état de la mémoire.

Question 4. Quelle est la quantité (exprimée en giga-octets) de mémoire adressable sur 32 bits ?

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`.

```
1   mov al, [0x00000003] ; al recoit l'octet stocke a l'adresse 3
2                               ; dans l'exemple, al = 10 = 0x0a;
3   mov al, [3]           ; Instruction equivalente a la precedente.
```

Question 5. 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* et *big-endian*. 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 précédente :

```
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.

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

Question 6. Dans l'exemple de mémoire, quelle est la valeur de **ax** après l'instruction `mov ax, [1]` ?

Question 7. 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 précédente ?

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

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 8. Quel est l'état de la mémoire après l'exécution de la suite d'instructions suivante ?

```
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 9. 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 10. 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]
```

Part II

Premier programme

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

```
1 nasm -g -f elf Hello.asm
2 ld -e main Hello.o -o Hello
3 chmod +x Hello
```

Les deux premières commandes créent un fichier `Hello` et la dernière le rend exécutable. Il ne sera nécessaire de taper la dernière ligne que la première fois que le fichier est créé. Pour exécuter le programme taper `./Hello`.

3 Explication du code

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

```
1 ; Premier programme en assembleur
2
3 SECTION .data ; Section des donnees.
4 msg :
5     db "Hello World", 10 ; La chaine de caracteres a afficher ,
6                           ; 10 est code ASCII retour a la ligne.
7
8 SECTION .text ; Section du code.
9 global main ; Rend l'etiquette visible de l'exterieur.
10 main : ; Etiquette pointant au debut du programme.
11     mov edx, 0xc ; arg3, nombre de caracteres a afficher
12                ; (equivalent a mov edx, 12).
13     mov ecx, msg ; arg2, adresse du premier caractere
14                ; a afficher.
15     mov ebx, 1 ; arg1, numero de la sortie pour l'affichage ,
16                ; 1 est la sortie standard.
17     mov eax, 4 ; Numero de la commande write pour
18                ; l'interruption 80h.
19     int 0x80 ; Interruption 0x80, appel au noyau.
20     mov ebx, 0 ; Code de sortie, 0 est la sortie normale.
21     mov eax, 1 ; Numero de la commande exit.
22     int 0x80 ; Interruption 0x80, appel au noyau.
```

3.1 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 – 22) contient le code du programme.

Lors de son exécution, le programme est chargé en mémoire comme décrit dans la figure suivante. L'adresse à laquelle débute le code est toujours la même : `0x08048080`.

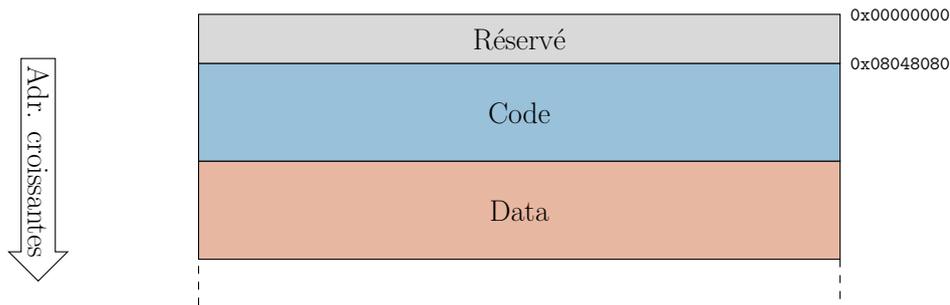


Figure 4: Mémoire occupée par le programme lors de son exécution.

En revanche, l'adresse à laquelle commence les données dépend de la taille du code.

Important 6. Lors de l'écriture d'un programme, l'adresse à laquelle est stockée la chaîne de caractères n'est pas connue. 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.

3.2 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,0x64,0xa
```

3.3 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 -e main Hello.o -o Hello` 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 a 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`, ... 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 11. Modifier `Hello.asm` pour afficher `Adieu monde cruel`.

Bonus 1. Écrire un programme qui exécute le programme `/bin/ls`. Pour cela, on utilisera l’appel système `execve`. Il s’utilise comme suit :

- `ebx` contient l'adresse de la chaîne à exécuter (terminée par un caractère de code ASCII 0),
- `ecx` doit contenir une adresse en mémoire à laquelle on trouvera un `dword` égal à l'adresse de la chaîne à exécuter suivi d'un `dword` valant 0.
- `edx` peut être égal à 0.

Complément 1. Pour connaître les paramètres attendus par un appel système, il faut 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.

4 Assemblage du programme

Dans cette section, 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 -g -f elf -l Hello.lst Hello.asm
```

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

```

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 BA0C00000             mov edx, 0xc
13 13
14 14 00000005 B9[00000000]         mov ecx, msg
15 15
16 16 0000000A BB0100000             mov ebx, 1
17 17
18 18 0000000F B80400000             mov eax, 4
19 19
20 20 00000014 CD80                 int 0x80
21 21 00000016 BB0000000             mov ebx, 0
22 22 0000001B B80100000             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` n'est pas connue. En particulier, l'instruction `mov ecx, msg` est codée par `B9[00000000]`. Les crochets signifient que l'adresse n'est pas encore finalisée.

Le code va occuper 34 octets en mémoire. Pour des raisons d'alignement, le code doit toujours occuper un nombre d'octets multiple de 4. Notre code occupera 36 octets (les deux octets manquant 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 12. Retrouver les octets chargés en mémoire dans l'exécutable `Hello`. Il est possible d'utiliser `xxd Hello` pour obtenir un affichage du fichier en hexadécimal.

Part III

Débugage

5 Débuguer avec Insight

Nous allons utiliser un front-end graphique pour `Gdb` appelé `Insight` qui permet de suivre pas à pas l'exécution d'un programme et de connaître les valeurs de la mémoire et des registres.

Important 7. Pour lancer `Insight`, taper `insight` dans un terminal mais jamais `insight &`.

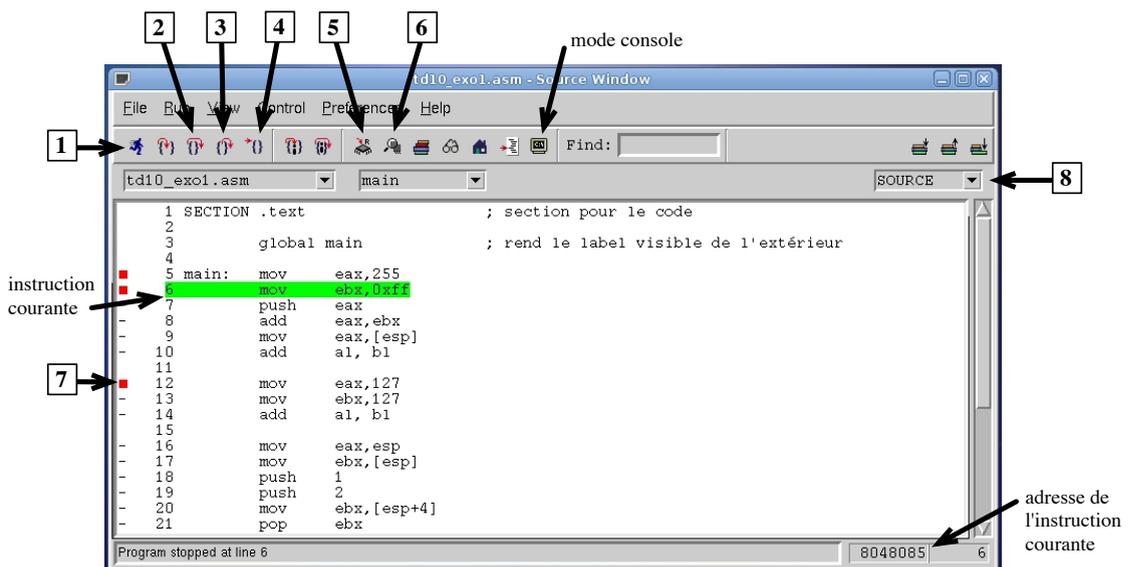


Figure 5: L'application `Insight`.

1. Ouvrir l'exécutable `Hello` (menu **File**, **Open...**). Il est possible de changer le style d'affichage du code assembleur en 8.

2. Repasser en affichage **Source** et lancer le programme [1]. Que se passe-t-il?
3. Pour pouvoir observer l'exécution du programme, il est nécessaire de placer des points d'arrêt (*breakpoints*). Pour cela, il suffit de cliquer sur le numéro de la ligne (sauf la 1ère) à laquelle on souhaite s'arrêter [7]. En cliquant à nouveau, on supprime le point d'arrêt. Il y a plusieurs façons d'avancer dans l'exécution du programme. On peut aller jusqu'au prochain breakpoint [4] si on en a placé plusieurs. Sinon, on peut avancer *pas à pas* (instruction par instruction) [2]. Enfin, on peut terminer l'exécution sans s'arrêter [3]. Placer un breakpoint sur la ligne 7. Relancer le programme. Que se passe-t-il?
4. Maintenant que le programme est lancé, il est possible d'observer l'état de la mémoire. Ouvrir la fenêtre des registres [5] et la fenêtre de la mémoire [6].

Question 13. Utiliser *Insight* pour suivre l'exécution de `Hello.asm`.

Question 14. En utilisant *Insight*, expliquer le comportement du programme `Myst.asm`.

6 Environnement d'exécution

Nous allons utiliser *Insight* pour explorer la mémoire autour de notre programme. L'organisation de la mémoire est donnée dans la dernière figure.

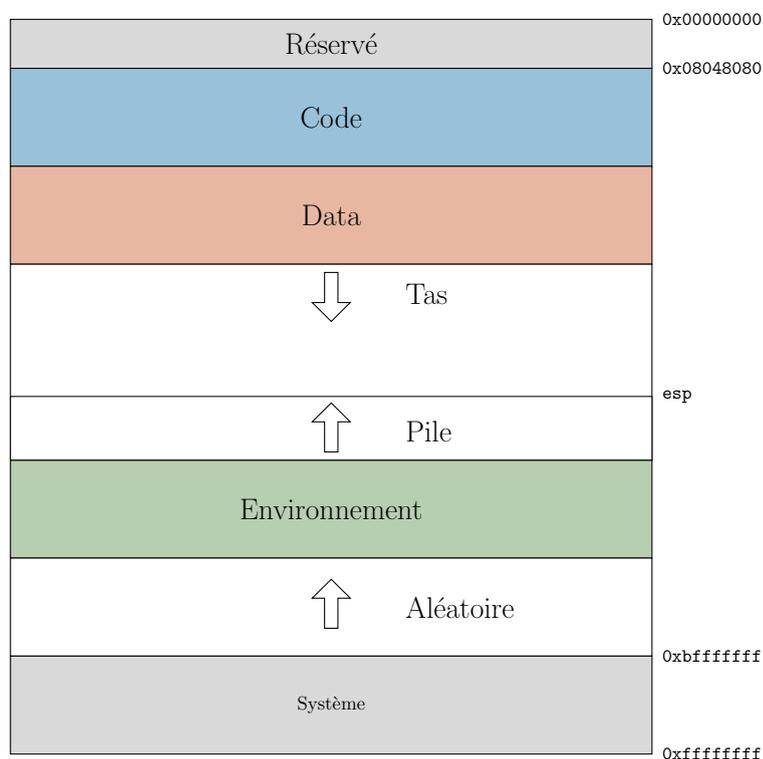


Figure 6: Organisation de la mémoire sous Linux en mode protégé.

La partie *Environnement* contient les arguments passés au programme ainsi que les variables d'environnement.

Astuce 2. Pour passer des arguments à un programme dans le terminal, il suffit de les taper à la suite (par exemple, `./Hello arg`). Pour passer des arguments dans `Insight`, taper `run arg` dans `Console Window`.

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

<code>esp</code>	<code>Argc</code>
<code>esp+ 4</code>	<code>Arg[0]</code>
<code>esp+ 8</code>	<code>Arg[1]</code>
<code>⋮</code>	<code>⋮</code>
<code>esp+ 4n + 4</code>	<code>Arg[n]</code>
<code>esp+ 4n + 8</code>	<code>0</code>
<code>esp+ 4n + 12</code>	<code>Env[0]</code>
<code>⋮</code>	<code>⋮</code>

Question 15. Utiliser `Insight` pour accéder à ces différentes valeurs après avoir ajouté deux arguments au programme. Retrouver les chaînes de caractères correspondantes en mémoire.

Astuce 3. Il est possible de taper `$esp` dans la fenêtre mémoire pour afficher la mémoire à l'adresse contenue dans `esp`.