Assembleur x86-64

P. Ezéquel

Table des matières

1 Introduction

AVERTISSEMENT : ce document est incomplet, mais correct. Il présente le minimum pour programmer des applications simples en assembleur x86-64.

1. Contexte : rappels architecture, langage machine, registres, mémoire
2. Algo du processeur : indéfiniment
   (a) lire l'opération en MC
   (b) lire les opérandes éventuelles, éventuellement en MC
   (c) exécuter l'opération
   (d) écrire les résultats éventuels, éventuellement en MC
   (e) mettre à jour CO
   (f) voir si interruptions
3. Assembleur :
   — traducteur d'un langage simple vers un langage encore plus simple
   — masque les atrocités de bas niveau : calculs d'occupation mémoire, d'adresses, réservation de mémoire, gestion de pile d'exécution,...
   — pas de structure de contrôle : seulement des GOTO (préhistoire de la prog...)
   — programmation sans filet : si ça ne marche pas, au mieux Seg Fault, au pire rien
   — nécessite d'avoir les idées claires : **algorithme préalable indispensable**
   — aucun intérêt pour la programmation de haut niveau avec les compilateurs modernes
   — indispensable pour les utilitaires de très bas niveau : boot, drivers,...
   — très grand intérêt pédagogique : montre l'envers du décor (pas très joli pour x86), oblige les étudiants à réfléchir et à utiliser les bons outils (`ddd`)
4. outil utilisé en TP : x86-64 en Linux avec `emacs`, `as` et `ddd`
5. rappel cycle de compilation de `C` avec `gcc` :
   (a) analyse lexicale : suite de caractères $\Longrightarrow$ suite de tokens
   (b) analyse syntaxique : suite de tokens $\Longrightarrow$ arbre syntaxique
   (c) génération de code : arbre syntaxique $\Longrightarrow$ programme
   (d) édition des liens : ensemble de programmes $\Longrightarrow$ exécutable

   En assembleur on n'a pas les étapes 5a et 5b.

2 Anatomie du x86-64

Différence RISC/CISC : RISC peu d'opérations, beaucoup de registres, CISC beaucoup d'opérations, peu de registres. x86-64 CISC : plusieurs centaines d'opérations, 64 registres en tout. SPARC propose (norme) de 64 à 528 registres entiers 64 bits et 80 registres flottants [1], et quelques dizaines d'opérations.

Sur la 60aine de registres du x86, seuls 24 nous intéressent.

2.1 Registres généraux

Ils sont 16. Dits généraux car le programmeur peut les utiliser (presque) comme il veut, sauf le pointeur de sommet de pile. Presque car ils ont une utilisation implicite dans certaines opérations.

Ils font 64 bits, cad 8 octets. Liste :

— `RAX` (Accumulateur) : seul registre vraiment général. `EAX` désigne les 4 premiers octets (sur 32 bits), `AX` désigne les 2 octets de poids faibles (sur 16 bits), `AH` le 2ème, `AL` le premier.
— `RBX` (Base) : pointeur vers la MC. Autres noms comme `RAX`.
— `RCX` (Compteur) : sert aux instructions de rotation/décalage et dans les boucles. Autres noms comme `RAX`.
— `RDX` (Données) : sert pour l'arithmétique et les E/S. Autres noms comme `RAX`.
— `RSP` (Sommet de Pile) : comme son nom l'indique. La largeur de la pile est de 8 octets (64 bits), cad que le processeur empile et dépile par paquets de 8 octets : on ne peut pas empiler ou dépiler seulement 1, 2 ou 4 octets.

1. soit au moins 144 registres seulement pour les données

— `RBP` (Base de Pile) : pointe à la base de la pile, utilisé comme borne pour explorer la pile.
— `RSI` (Source Index) : pointeur de début de zone source dans les opérations de transferts.
— `RDI` (Destination Index) : pointeur de début de zone destination dans les opérations de transferts.
— 8 autres registres, de `R8` à `R15`.

2.2 Registres de base de segments

Autrefois il n'y avait que 20 bits sur le bus d'adresse, soit seulement 2 MO adressables directement. Intel a utilisé la segmentation : une adresse est un couple (Base de segment, Offset), ce qui permet d'adresser (théoriquement) 4 TO avec 20 bits pour chaque valeur. De nos jours les bus font 64 ou 128 bits de large, la segmentation est devenue inutile, et ces registres de base de segments aussi. D'ailleurs Linux, MacOS, Windows les font pointer sur la même adresse (modèle de mémoire plate, *Flat Memory Model*). On les garde pour la compatibilité ascendante. Il y en a 6 :

— `SS` : Stack Segment (pile)
— `CS` : Code Segment (programme)
— `DS` : Data Segment (données du programme)
— `ES` : Extra segment, pour plus de données
— `FS` : encore plus de données Segment (F vient après E, quelle imagination chez Intel...)
— `GS` : encore encore plus de données Segment (G vient après F, quelle imagination chez Intel (bis))

Leur utilisation supposée est la suivante :

— toute instruction est lue dans `CS`
— tout accès indirect se fait dans `DS`
— tout accès basé sur `R?X` se fait dans `DS`
— tout accès basé sur `R?P` se fait dans `SS`
— toute fonction de chaîne se fait dans `ES`
— on ne peut pas écrire dans `CS` (!)

Aujourd'hui mode *Flat*, la mémoire est accessible globalement, linéairement, sans utilisation des segments : on peut donc oublier ce qui précède (sauf si on programme sans OS !)

2.3 Registre d'état

De longueur 64 bits, seuls les 32 premiers sont utilisés (les 32 derniers sont "réservés", d'après la documentation INTEL). Chaque bit porte une information sur l'état du processeur, ou sur le résultat de la dernière opération. Parmi les bits qui nous intéressent :

— `CF`, bit 0 (*Carry Flag*) : positionné par une opération arithmétique (addition, soustraction) qui a propagé une retenue (en fait c'est cette retenue). Utilisé par les opérations arithmétiques avec retenue.
— `ZF`, bit 6 (*Zero Flag*) : positionné si le résultat de la dernière opération a été 0.
— `SF`, bit 7 (*Sign Flag*) : positionné si le résultat de la dernière opération est négatif.
— `DF`, bit 10 (*Direction Flag*) : à 1, les opérations sur les chaînes vont aller dans le sens des adresses décroissantes, à 0 dans le sens des adresses croissantes.
— `OF`, bit 11 (*Overflow Flag*) : positionné si le résultat de la dernière opération **signée** est trop gros pour un registre.

2.4 Compteur ordinal

Nommé `RIP`, il contient l'adresse de la prochaine instruction. Ne peut pas être explicitement modifié par le programmeur, il y a des instructions de saut pour ça.

3 Instructions

Un programme, c'est essentiellement une suite d'instructions, à raison d'une par ligne. L'assembleur (en fait le langage d'assemblage) comprend des intructions à 0, 1, 2 ou 3 arguments. Il y a deux formats d'instruction :

1. le format `as`, qui est celui que nous utiliserons puisque c'est celui de l'assembleur `as` ;
2. le format Intel, qui est utilisé par Intel bien sûr dans ses manuels, mais aussi par d'autres assembleurs.

On a les conventions d'écriture suivantes :
— une instruction à 0 arguments est notée
```
instr
```
— une instruction à 1 argument est notée
```
instr arg
```
— une instruction à 2 arguments est notée
```
instr source, destination
```
— une instruction à 3 arguments est notée
```
instr aux, source, destination
```
Dans le format `as`, on peut spécifier quelle est la taille des arguments. Ceci se fait en ajoutant une lettre à la fin du nom de l'instruction, sur le format suivant :
— arguments sur un octet : `b` comme byte (octet)
— arguments sur 2 octets : `w` comme word (mot)
— arguments sur 4 octets : `l` comme long (long)
— arguments sur 8 octets : `q` comme quad (4 mots)
Un argument peut être (voir section 4)
— un registre,
— une adresse mémoire,
— une valeur.

3.1 Instructions arithmétiques

3.1.1 Opérations binaires

Elles prennent deux opérandes, **qui ne peuvent pas être toutes deux des adresses mémoires**. Une instruction de type

```
Op src, dest
```

est équivalente à `dest = dest Op src`.

1. addition : `add src, dest`.
2. addition avec retenue : `adc src, dest`. Comme l'addition, on ajoute le CF en plus (pour enchaîner des additions).
3. soustraction : `sub src, dest`
4. soustraction avec retenue : `sbb src, dest`. Comme la soustraction, on enlève le CF en plus (pour enchaîner des soustractions).
5. multiplication : `mul arg`. L'autre opérande et la destination dépendent de la taille de `arg`, voir le tableau ci-dessous. `imul` est la multiplication signée.

Taille en octets	1	2	4	8
Autre opérande	`AL`	`AX`	`EAX`	`RAX`
Moitié haute du résultat	`AH`	`DX`	`EDX`	`RDX`
Moitié basse du résultat	`AL`	`AX`	`EAX`	`RAX`

6. division : `div arg`. `arg` est le diviseur. Le dividende et la destination dépendent de la taille de `arg`, voir le tableau ci-dessous. `idiv` est la division signée.

Taille en octets	1	2	4	8
Dividende	`AX`	`DX:AX`	`EDX:EAX`	`RDX:RAX`
Reste	`AH`	`DX`	`EDX`	`RDX`
Quotient	`AL`	`AX`	`EAX`	`RAX`

`EDX:EAX` veut dire concaténation des 2 registres. Il faut propager à `EDX` le signe de `EAX`, au moyen de l'instruction `cdq` (Convert Double to Quad).

3.1.2 Opérations unaires

Elles prennent 1 seule opérande, qui peut être un registre ou une adresse mémoire.

7. opposé : `neg arg`. Remplace `arg` par son opposé.
8. incrément : `inc arg`. Équivalent à `arg++`
9. décrément : `dec arg`. Équivalent à `arg--`

3.2 Instructions logiques

Équivalentes à `dest = dest OP src` (sauf la négation bien sûr). Mêmes limitations que pour les opérations arithmétiques.

1. `and src, dest`
2. `or  src, dest`
3. `xor src, dest` (souvent utilisé avec `src` et `dest` identiques, pour mettre un registre à 0 rapidement : `xorl %eax,%eax` met `EAX` à 0)
4. `not arg`

3.3 Transfert d'information

1. `mov src, dest` affecte la valeur désignée par `src` à `dest`. `dest` est soit un registre soit une adresse mémoire, `src` peut aussi être une constante. Les deux ne peuvent pas être des adresses mémoires.
2. `push src` empile `src`, et met `RSP` à jour.
3. `pop dest` dépile dans `dest`, et met `RSP` à jour.
4. `movsb` copie l'octet d'adresse `DS:RSI` à l'adresse `DS:RDI`, incrémente `RSI` et `RDI` si `DF` est nul, les décrémente sinon. Avec `rep` devant, copie `RCX` octets. C'est la seule opération de transfert qui déplace de la mémoire vers la mémoire.

3.4 Instructions de contrôle de l'exécution

Il s'agit d'instructions de test et de changement du compteur ordinal, afin d'implanter des structures de contrôle de haut niveau : si... alors... sinon, Tantque, appel de *sous-programmes* (on dit des fonctions en C...).

3.4.1 Comparaisons

1. `test arg1, arg2` (`arg1` ne peut pas être une constante, `arg2` ne peut pas être en mémoire) : calcule le `ET` des deux arguments, met `ZF` à 1 s'il est égal à 0, met le résultat à la poubelle (utilisé en fait dans `test arg, arg`, pour faire le test `arg==0`, puisque le résultat de `arg ET arg` ne sera égal à 0 que si `arg` l'est)
2. `cmp arg1, arg2` : soustrait `arg1` de `arg2`, positionne le registre d'état. Comme il s'agit, finalement, d'une opération arithmétique, les 2 arguments ne peuvent pas être 2 adresses mémoires.

3.4.2 Sauts

1. `JMP adr` : saut inconditionnel à l'adresse `adr`.
2. `JE adr` : saut à l'adresse `adr` si la dernière comparaison a donné une égalité
3. `JNE adr` : saut à l'adresse `adr` si la dernière comparaison a donné une inégalité
4. `JZ adr` : saut à l'adresse `adr` si la dernière opération a donné 0 comme résultat
5. `JNZ adr` : saut à l'adresse `adr` si la dernière opération a donné un résultat non nul
6. `JL adr` : saut à l'adresse `adr` si la dernière comparaison a donné `arg2` plus petit que `arg1` (et il y a aussi `JG`, `JLE`, `JGE`...)

3.4.3 Appel de sous-programmes

L'appel se fait avec `call adr` où `adr` est l'adresse du sous-programme. `RIP` est empilé (c'est l'instruction qui suit l'appel du sous-programme), et le contrôle passe à l'adresse `adr`. Le sous-programme rend la main en exécutant `ret`, qui dépile le sommet de pile dans `RIP`. On peut nettoyer automatiquement la pile des arguments qu'on a passé au sous-programme (fonctionnement normal des fonctions C) en ajoutant un argument numérique à `ret`, qui est le nombre d'octets à dépiler après avoir dépilé l'adresse de retour.

4 Modes d'adressages

On nomme *mode d'adressage* une façon de spécifier un argument d'une instruction. Un argument peut être

1. une constante (numérique) : `$97` (en base 10) ou `$0x61` (en base 16) (adressage **immédiat**)
2. le contenu d'un registre : `%rax` (adressage **par registre**)
3. une adresse mémoire : il y a 3 possibilités
   (a) une étiquette du programme : `toto` (soit une donnée, soit une instruction du programme) (adressage **direct**)
   (b) l'adresse d'une donnée : `$compteur` (adressage **indirect**)
   (c) une référence à la mémoire :

   `déplacement(base,index,multiplicateur)`

   calcule l'adresse `base+index*multiplicateur+déplacement`. On peut omettre index, multiplicateur ou déplacement. Les usages les plus fréquents sont :
   — (base), pointeur sur la mémoire : `(%rcx)` désigne l'octet dont l'adresse est dans `RCX` (adressage **indirect**) ;
   — (base,index) : `(%rcx,%rax)` désigne l'octet dont l'adresse est `RCX + RAX` (adressage **indirect indexé**) ;
   — déplacement(base) : `-8(%rbp)` désigne l'octet situé 8 octets **au-dessous** de `RBP` (donc l'entier en fond de pile...).
   — base et `index` peuvent être un des registres `RAX`, `RBX`, `RCX`, `RDX`, `RSP`, `RBP`, `RSI`, `RDI`.
   — `déplacement` et multiplicateur sont des constantes.

5 Assembleur `as` et éditeur de liens `ld`

Nous utiliserons l'assembleur `as` et l'éditeur de liens `ld` dans les TP :
— `as` prend en entrée un fichier source, et produit un fichier objet **relogeable** ;
— `ld` prend en entrée un ou plusieurs fichiers relogeables et produit un fichier **exécutable**.

5.1 Assembleur `as`

Un programme `as` est divisé en **sections**. Deux sont fréquentes :
— `.data` contient les données initialisées du programme (équivalentes aux variables globales d'un programme `C`) ;
— `.text` contient le programme (il ne pourra pas être modifié à l'exécution).

Un programme comporte (au moins) une référence globale, signalée par `.globl` ou `.global` dans le texte du programme : il s'agit de l'adresse où va commencer l'exécution du programme. C'est cette adresse qui est utilisée par l'éditeur de liens pour élaborer un exécutable à partir de divers sous-programmes. Le programme principal **doit** avoir comme référence globale `_start` (valeur par défaut attendue par `ld` ; on peut en proposer une autre, c'est l'option `-e <symbole>` de `ld`).

5.1.1 Segment `data`

Commençant par le mot clé `.data`, c'est une suite de lignes

```
[étiquette:] .<type>  <expression>
```

où `type` est l'une des valeurs (liste non exhaustive)
— `byte` (octet, cad caractère),
— `word` (entier court sur 2 octets),
— `long` (entier sur 4 octets),
— `quad` (entier sur 8 octets),
— `string` (chaîne de caractères, que l'assembleur terminera par `\0`)

`étiquette` est le nom de la donnée (donc son adresse...).

`expression` est optionnelle : la déclaration des données sert en fait à réserver suffisamment de mémoire pour les ranger. Si rien n'est spécifié, rien n'est réservé : il y a donc intérêt à initialiser les variables locales...

5.1.2 Segment `text`

Commençant par le mot clé `.text`, c'est une suite de lignes

```
[étiquette:] <instruction>
```

Au moins une étiquette doit être présente, c'est celle qui indique la première instruction du programme. Les étiquettes sont les arguments symboliques des sauts (instructions `jmp`, `jz`,...).

5.1.3 Invocation et résultat

Pour lancer l'assemblage du programme contenu dans le fichier `fichier.s` et obtenir le module objet relogeable dans le fichier `fichier.o`, il suffit de donner en ligne de commmande

```
> as -a --gstabs -o fichier.o fichier.s
```

L'option `-a` demande un affichage de tout l'assemblage (code lu, code généré, tables des symboles) : souvent un problème lors de l'assemblage se traduit par des symboles non définis (sauf bien sûr en cas d'assemblage séparé). L'option `--gstabs` demande l'insertion de points d'arrêt qui seront exploités par le debugger (pour nous `ddd`).

5.2 Éditeur de liens `ld`

`ld` prend en entrée un ensemble de fichiers objets (issus de l'assemblage avec `as` ou de la compilation avec `gcc`) et produit un exécutable. On l'invoque par la ligne de commande

```
> ld [-e <symbole>] <fichier1> ... <fichierN> -o <exécutable>
```

où `<fichier1>`, ..., `<fichierN>` sont les fichiers objets et `<exécutable>` le fichier où le programme sera écrit. L'option `-e <symbole>` doit être présente si l'étiquette du programme principal n'est pas l'étiquette standard `_start`.

Pour utiliser les fonctions de la `libc`, qui sont quand même bien pratiques (par exemple `atoi`, `printf`, ...), il faut lier le programme avec la librairie dynamique, au moyen de la commande

```
> ld [-e <symbole>] -o programme --dynamic-linker /lib/ld-linux.so.2 -lc programme.o
```

Attention, l'ordre des arguments a de l'importance : `ld` est *vraiment* de très bas niveau [2].

6 Interface avec UNIX

Les programmes x86 reçoivent les arguments de la ligne de commande, et peuvent invoquer des appels système.

6.1 Arguments de la ligne de commande

Au lancement d'un programme, la pile d'exécution n'est pas vide, elle contient, du haut vers le bas de la pile :

- `argc`, le nombre d'arguments sur la ligne de commande,
- `argv[0]`,
- `argv[1]`,

  ⋮
- `argv[argc - 1]`,
- 0

Un programme a donc accès à la ligne de commande qui l'a invoqué, il suffit de la lire dans la pile...

2. ce qui est normal dans la mesure où il n'est pas censé être utilisé par des êtres humains...

6.2 Appels système

Il est possible d'invoquer les appels systèmes d'UNIX depuis un programme x86, de la façon suivante :
— chaque appel système possède un numéro, qui est placé dans `RAX`.
— Le passage des paramètres se fait au moyen des registres : ils sont placés, dans l'ordre, dans `RDI`, puis `RSI`, puis `RDX`, puis `R10`, puis `R8` et enfin `R9`. Si l'appel système a besoin de $n < 6$ arguments, seuls les n premiers registres sont utilisés. À ma connaissance il n'y a pas d'appel système POSIX comportant plus de 6 arguments.
— Il suffit alors d'exécuter l'appel système : `syscall`
— La valeur de retour de l'appel système est placée dans `RAX`.
— Le système utilise sa propre pile, la pile du processus appelant n'est pas modifiée.
— Les registres sont inchangés, sauf peut-être `RCX` et `R11`.

Dans le cadre de ce cours, nous n'utiliserons que les appels `exit` et `write`, de numéros 60 et 1 respectivement.

6.2.1 Utilisation de exit

Exemple de codage de l'instruction `exit(0)` :

```
movq $60, %rax    # rax <- 60 (numero de exit)
xorq %rdi, %rdi   # rdi <- 0
syscall           # appel systeme
```

6.2.2 Utilisation de write

L'appel système `write` a pour prototype

```
int write(int fd, void *buf, size_t longueur);
```

où
— `fd` est le numéro du descripteur de fichier considéré (`stdout` a le numéro 1),
— `buf` est l'adresse du premier octet à afficher,
— `longueur` est le nombre d'octets à afficher à partir de l'adresse `buf`.

Si on écrit sur `stdout`, les octets écrits sont supposés contenir des codes ASCII. Supposons qu'à l'adresse `$message` se trouve la chaîne de caractères

```
"Bonjour tout le monde!\n"
```

de longueur 23. Pour l'afficher, il suffit d'écrire le bout de code suivant :

```
movq $1, %rax         # rax <- 1        (numero de write)
movq $1, %rdi         # rdi <- 1        (stdout)
movq $message, %rsi   # rsi <- $message (buf)
movq $23, %rdx        # rdx <- 23       (longueur chaine)
syscall               # appel systeme
```

7 Deux programmes

7.1 Somme des éléments d'un tableau

7.1.1 Algorithme

début
  $I \leftarrow 0$;
  $S \leftarrow 0$;
  tant que $I \leq 3$ **faire**
    $S = S + T[I]$;
    $I = I + 1$
fin

7.1.2 Programme

```
.data  # début du segment de données

tableau:
      .quad   1
      .quad   5
      .quad   2
      .quad   18

somme:
      .quad 0

.text  # début du segment de code

.global _start
_start:
      xor %rax, %rax            # RAX = 0
      xor %rbx, %rbx            # RBX = 0
      mov $tableau, %rcx        # RCX = tableau

tant_que:
      cmp $4, %rax              # RAX == 4 ?
      je fin                    # alors fini
      add (%rcx, %rax,8), %rbx  # RBX += RCX[RAX]
      inc %rax                  # RAX++
      jmp tant_que
fin:
      mov %rbx, somme           # fini, résultat dans somme
      mov $60, %rax             # RAX = 60 (code de l'AS exit)
      xor %rdi, %rdi            # RDI = 0
      syscall                   # appel systeme
```

7.1.3 Module

On va faire un module réutilisable du programme précédent, qui contiendra le sous-programme de somme d'un tableau. Le sous-programme attend le tableau (cad l'adresse de son premier octet) et le nombre d'éléments du tableau sur la pile, dans cet ordre. Le résultat sera placé dans `RAX`.

```
.text

.global somme_tableau
somme_tableau:
      xor %rax, %rax           # RAX = 0
      xor %rbx, %rbx           # RBX = 0
      mov 8(%rsp), %rcx        # RCX = sommet de pile

tant_que:
      cmp 16(%rsp), %rbx        # RBX == N ?
      je fin                    # si oui fini
      add (%rcx,%rbx,8), %rax   # RAX += RCX[RBX]
      inc %rbx                  # RBX++
      jmp tant_que
fin:
      ret $16                    # retour, et dépiler les arguments
```

7.2 Affichage des arguments de la ligne de commande

On se propose d'écrire un programme qui affiche ses arguments, à raison d'un par ligne, sur la sortie standard. Le programme va utiliser un module d'affichage de chaîne de caractéres.

7.2.1 Module d'affichage d'une chaîne de caractères

Le sous-programme affiche sur `stdout` la chaîne de caractères dont l'adresse de début est dans `RSI`. L'algorithme est le suivant :

```
lgr_chaine = 0;
p = chaine;
tantque *p != 0 faire
    p ++;
    lgr_chaine ++;
ftq
write(1,p,lgr_chaine);
```

D'où le programme, où on utilise `RBX` pour le pointeur `p` et `RDX` pour `lgr_chaine` :

```
        .text
.global affiche_chaine

affiche_chaine:
        xor %rdx, %rdx   # RDX = 0
        mov %rsi, %rbx   # RBX = RSI
boucle:
        movb (%rbx), %al  # AL = *RBX
        test %al, %al     # AL == 0 ?
        jz fin            # alors fini
        inc %rdx          # RDX ++
        inc %rbx          # RBX ++
        jmp boucle
fin:                      # write(1,RSI,RDX)
        movl $1, %rax     # RAX = 1 (code de l'AS write)
        movl $1, %rdi     # RDI = 1
        syscall           # appel au noyau UNIX
        ret               # retour
```

7.2.2 Programme

Enfin le programme, utilisant le module d'affichage de chaîne. Comme la pile contient `argc` en sommet de pile, il faut commencer par le dépiler. Ensuite, on affiche chacune des chaînes de `argv` jusqu'à arriver à `NULL`. Le programme est donc le suivant :

```
       .text
.global debut
debut:
       pop %rsi                # RSI = argc
boucle:
       pop %rsi                # RSI = argv suivant
       test %rsi,%rsi          # RSI == 0 ?
       jz   exit               # alors aller a EXIT
       call affiche_chaine     # afficher la chaine dans RSI
       jmp boucle
exit:                          # exit(0)
       mov $60, %rax           # RAX = 60 (code de l'AS exit)
       mov $0, %rdi            # RDI = 0
       syscall                 # appel au noyau UNIX
```

Après assemblage, édition de liens et première exécution, on notera que le programme n'affiche pas les arguments à raison d'un par ligne, comme prévu : je laisse le soin au lecteur de le modifier en conséquence...