L'assembleur sous Linux

Ou comment faire une intro 4kb Linux avec la Xlib

Table

- 1. Introduction
- 2. Le bon vieux "Hello World"
- 3. Un problème de taille
  - 3.1. Retirer les symbole : strip
  - 3.2. Se défaire de main et utiliser les appels systèmes
  - 3.3. Ne garder que l'essentiel
  - 3.4. Une astuce supplémentaire
  - 3.5. Ca nous donne quoi tout ça ?
- 4. Utilisation de la Xlib
  - 4.1. Initialisation
    - 4.1.1. Les fonctions externes
    - 4.1.2. Les variables
    - 4.1.3. La fonction d'initialisation
  - 4.2. Blitter notre buffer
  - 4.3. Conversion de couleurs
- 5. Le temps
  - 5.1. Initialiser le compteur
  - 5.2. Connaître le temps écoulé
- 6. Le son avec OSS
- 7. Aller plus loin
- A. Annexes
  - A.1. Hello World avec les appels systèmes
- B. Liens

1. Introduction

Je tiens d'abord à préciser une chose : ce texte ne s'adresse pas à ceux qui veulent apprendre l'assembleur. Il est plutôt destiné à ceux qui le connaissent déjà et qui désirent découvrir comment l'utiliser sous linux. Néanmoins, comme on n'apprend jamais mieux qu'en lisant des sources (en tout cas, en ce qui concerne ce "langage" qu'est l'assembleur), vous êtes tous, du débutant au confirmé, invité à lire cette doc et à me faire part de vos suggestions/remerciements/insultes/autres soit en laissant un commentaire sur alrj.org, soit par mail à l'adresse allergy@alrj.org.

Les outils nécessaires seront, outre votre éditeur de texte favori, l'assembleur *Nasm* ainsi que *ld* (ou *gcc*, qui l'inclut). Comme vous pouvez le constater, point de syntaxe AT&T ici, juste du "presque Intel".
Bien sûr, vous pouvez aussi vous munir d'un debugger et d'un compilateur C (pour voir le code qu'il pond).

Si vous êtes prêt, commençons.

2. Le bon vieux "Hello World"

Je vais commencer par donner l'exemple, que je commenterai ensuite. Lorsqu'il s'agit d'un programme aussi petit, ca permet d'avoir directement une vue d'ensemble.

Voici donc le code de notre Hello World :

```
1       BITS 32
2
3       EXTERN puts
4
5       SECTION .data
6               chaine      db "Hello world !", 0
7
8       SECTION .text
9               GLOBAL main
10
11              main:
12              push dword chaine
13              call puts
14              add esp, 4
15              ret
```

Ce simple exemple nous donne déjà une belle quantité d'informations.
On y découvre, en lignes 5 et 8, des `SECTIONS`. Ce sont les segments de notre futur exécutable. Nous utilisons ici les sections `.data` et `.text`.

- `.data` est utilisée pour les données initialisées. C'est à dire, les données qui doivent avoir une valeur précise au lancement du programme. Dans cet exemple, c'est la chaine de caractère que nous stockons.
- `.text` contient le code de notre exécutable.

Il est à noter qu'il existe bien d'autres types de segments (sections), comme `.bss` qui contient les données **non**-initialisées. Le système leur réservera de la place en mémoire lors de l'exécution du programme, mais elles ne prennent pas le moindre octet dans le binaire lui-même.

Aux lignes 3 et 9, nous pouvons remarquer les mots `EXTERN` et `GLOBAL`. Ils sont complémentaires. Le premier permet de spécifier une fonction qui se trouve en fait à l'extérieur de notre source. Le second spécifie quels sont les symboles qui seront utilisables depuis l'extérieur.
Nous devons ici déclarer `puts` en `EXTERN` car il sera utilisé par notre programme.
`main` est déclaré en `GLOBAL` car il sera appelé par le système : c'est le point d'entrée du programme (comme en C).

Enfin, les lignes 12 à 14 de cet exemple montrent comment sont effectués les appels aux fonctions. Les arguments sont `push`és sur la pile (ligne 12) avant l'appel (ligne 13), et c'est l'appelant qui les dépile (ligne 14).
L'exemple ici ne le montre pas, mais les arguments sont mis sur la pile *en commençant par le dernier*. C'est à dire que si nous avons une fonction qui prend deux arguments :

```
int ma_fonction(int arg1, int arg2)
```

il faudra d'abord empiler `arg2` puis `arg1`. Le code de retour est habituellement donné dans `EAX`.

Pour compiler ce programme :

```
$ nasm -f elf hello.asm
$ gcc hello.o -o hello
$ ./hello
Hello world !
$
```

Vous l'aurez deviné, le `$` représente le prompt.

3. Un problème de taille

Si vous faites de l'assembleur avec pour objectif la réalisation d'une intro en 4kb, vous aurez constaté un petit problème : ce simple "hello world" est gros. Très gros, même.

```
$ ls -l hello
-rwxr-xr-x    1 allergy  allergy      4758 avr  3 22:11 hello
```

4758 bytes, c'est trop pour une 4kb, pour laquelle la limite est de 4096 bytes...
Mais tout n'est pas perdu. Certains vous diront qu'il faut commencer à se prendre la tête avec les en-têtes ELF, utiliser l'option `-f bin` de nasm pour générer directement tout l'exécutable, etc. Rassurez-vous, ces mesures ne sont à prendre qu'en dernier recours.

Tentons de déterminer ce qui prend de la place dans ce programme :

- Bien évidemment le code et les données, bref, les trucs à nous
- Chaque fonction externe doit être référencée
- D'éventuelles infos de débuggage
- Les en-têtes ELF
- Des "commentaires", mis là par le compilateur

Réduire la taille du code est bien évidemment votre travail. C'est à vous de trouver les optimisations nécessaires. De même, évitez autant que possible l'utilisation des fonctions externes. Il reste bien sûr des cas où elles sont inévitables, mais chaque fois que vous pouvez le faire, tentez de les éliminer.

Voyons maintenant quelques méthodes qui nous permettront de réduire la taille de notre binaire. Je ne parlerai ici que des méthodes applicable de manière générale, pas des optimisations propres à ce programme-ci.

3.1. Retirer les symboles : strip

La première chose à faire est de *strip*er l'exécutable (`man strip` vous en dira plus long sur cette commande, si vous ne la connaissez pas encore).

```
$ strip -s hello
$ ls -l hello
-rwxr-xr-x    1 allergy  allergy      2956 avr  5 06:24 hello
```

C'est bien, mais ce n'est pas encore tout à fait ça.
Cela nous laisse un peu plus d'un kilo-octet pour le code et les données. Comptez l'initialisation graphique, peut-être également une gestion de la carte son, il ne vous reste vraiment plus grand chose.

Il doit donc forcément être possible de faire mieux.
Effectivement.

3.2. Se défaire de main et utiliser les appels systèmes

Je sais, le titre a l'air un peu compliqué, mais ne vous inquiétez pas, c'est presque aussi simple.
Vous savez peut-être que `main` n'est le point d'entrée du programme que parce que c'est le comportement par défaut de *ld*. Heureusement, on peut lui demander de fonctionner différemment.

Avec le fonctionnement par défaut, `main` n'est qu'une fonction comme les autres. Le point d'entrée réel de l'exécutable se trouve ailleurs et s'appelle `_start`. La sortie de notre programme, qui était un simple `ret` ne fait que quitter la fonction main.
Cela signifie qu'en réalité, il y a une partie du code, dans cet exécutable, qui n'est pas de nous et qui se charge d'appeler notre `main` ! Il est bien évident que nous devons nous en débarrasser.

Opérons quelques modifications à notre code.

```
1       BITS 32
2
3       EXTERN puts
4
5       SECTION .data
6               chaine      db "Hello world !", 0
7
8       SECTION .text
9               GLOBAL _start
10
11              _start:
12              push dword chaine
13              call puts
14              add esp, 4
15              mov eax, 1
16              int 0x80
```

Comme vous le constatez, pas grande différence. Aux lignes 9 et 11, nous avons remplacé le `main` par `_start` et le `ret` de la ligne 15 est devenu un étrange `mov eax, 1 | int 0x80`. Et voilà, vous venez de faire un appel système.

Sous Linux, ces appels se font par l'intermédiaire de l'interruption 0x80. L'appel numéro 1 est celui qui demande au système la fermeture du programme. Rien de bien compliqué jusqu'à présent.

Cependant, si vous tentez de compiler ceci de la même manière que pour l'exemple du chapitre 2, vous risquez d'avoir une mauvaise surprise au moment de l'édition des liens. Voici la marche à suivre :

```
$ nasm -f elf hello.asm
$ gcc -nostdlib -lc hello.o -o hello
$ ./hello
Hello world !
$ ls -l hello
-rwxr-xr-x    1 allergy  allergy      2028 avr  5 07:01 hello
```

L'argument `-nostdlib` permet de s'affranchir du code ajouté lors de l'édition des liens. Plus de `main`, place à `_start`.
Le `-lc` est l'habituelle demande de lien avec une bibliothèque externe, nécessaire ici car c'est elle qui contient `puts`.

N'oublions pas ce que nous avons vu précédemment, et *strip*ons le binaire obtenu :

```
$ strip -s hello
$ ls -l hello
-rwxr-xr-x    1 allergy  allergy      1404 avr  5 07:15 hello
```

Avouez que c'est déjà mieux ! Il reste maintenant un peu plus de 2.5kb de libre pour votre code.
Et pourtant, il est tellement simple de faire mieux...

3.3. Ne garder que l'essentiel

Au long de mes pérégrinations sur le net, je suis tombé sur la page des ELFkickers (voir la section des liens). C'est une compilation d'outils très pratiques, qui tournent tous autour du format ELF.
L'un d'entre eux, en particulier, a retenu mon attention : *sstrip*. Dans la même veine que *strip*, il permet un travail plus en profondeur. Il retire non seulement les symboles inutiles, mais simplifie également les en-têtes ELF et retire les sections inutiles.

Voyons ce que ça donne :

```
$ sstrip hello
$ ls -l hello
-rwxr-xr-x    1 allergy  allergy       612 avr  5 07:35 hello
$ ./hello
Hello world !
```

612 octets, et il fonctionne toujours ! Et pourtant, ce n'est même pas fini.

3.4. Une astuce supplémentaire

Vous vous en doutez sans doute, un bon moyen pour mettre plus de données dans un fichier de petite taille passe par la compression. Le problème, c'est qu'un décompresseur, ça prend de la place. Sauf bien sûr s'il n'est pas réellement inclus dans l'exécutable. Et il existe un compresseur disponible sur presque toute machine Linux : *gzip*. Alors pourquoi ne pas lui demander de faire le travail pour nous ?

Mais il est toujours plus propre d'avoir un seul fichier exécutable, plutôt qu'un exécutable et un fichier compressé. La solution (volée à titou^shagreen) est de regrouper les deux : un seul fichier qui consiste en deux partie : un script de décompression et d'exécution ainsi que le binaire compressé.

Je vous donne cette astuce telle quelle, sans entrer dans les détails. Si vous avez du mal à comprendre son fonctionnement, n'hésitez pas à me contacter.

```
$ cat compress.sh
#!/bin/sh
dd if=$0 bs=1 skip=68|gzip -cd>A
chmod +x A;./A
rm A
exit

$ gzip -9 hello
$ cat compress.sh hello.gz > hello
$ chmod a+x hello

$ ./hello
349+0 enregistrements lus.
349+0 enregistrements écrits.
Hello world !

$ ls -l hello
-rwxr-xr-x    1 allergy  allergy       417 avr  5 19:32 hello
```

Quelques petits commentaires, cependant.

Certains esprits grincheux contesteront et diront que l'utilisateur n'a pas forcément un accès en écriture dans le répertoire (et ils auront raison). Or, c'est indispensable pour écrire le fichier décompressé. La solution est de remplacer le fichier `A` par `/tmp/A`. Ca augmente légèrement la taille du script, mais dans une bien faible mesure. N'oubliez cependant pas de modifier également la valeur de `skip` lors du *dd*.

L'autre désagrément est celui impliqué par la commande *dd* : les `349+0 enregistrements....` Soit on s'arrange avec une redirection, soit on se dit que ça n'est pas dérangeant.

3.5. Ca nous donne quoi tout ça ?

Nous en arrivons donc à un exécutable de 417 octets. Il est évidemment possible de faire **beaucoup** mieux. Par exemple, remplacer le `puts` par un appel système, comme proposé au chapitre 3.2. On évite alors le lien avec la *libc* et le `EXTERN`. Comme je suis gentil et serviable, j'ai mis en annexe, pour ceux qui seraient intéressés, un exemple de Hello World qui utilise les appels systèmes.
Il est aussi possible de jouer avec les en-têtes, et arriver au final à un exécutable de moins de 100 bytes.
Mais j'estime que cette dernière solution n'a pas sa place dans ce tutoriel, qui se veut plus généraliste. Peut-être une prochaine fois ?

4. Utilisation de la Xlib

Ce point-ci offre moins d'intérêt au point de vue assembleur pur. Certes, il est indispensable à la réalisation d'une animation graphique, mais il consiste surtout à appeler quelques fonctions externes. Il faudra cependant faire attention : lorsque cette partie est réalisée en C, on ne se rend pas toujours compte que certains appels sont en réalité des macros et non des fonctions. Il est parfois intéressant d'écrire une première fois l'initialisation en C et de demander une sortie assembleur à *gcc* (grâce à l'option `-S`) pour voir de quelle manière il travaille.

4.1. Initialisation

4.1.1. Les fonctions externes

Avant tout, nous devons déclarer quelques fonctions externes.

```
EXTERN XOpenDisplay
EXTERN XCreateSimpleWindow
EXTERN XMapRaised
EXTERN XCreateImage
EXTERN XPutImage
EXTERN malloc
```

4.1.2. Les variables

Nous aurons également besoin de quelques variables. Elles n'auront pas à avoir de valeur précise lors du lancement du programme, leur place est donc tout indiquée : le segment `.bss`, comme expliqué au point 2.

```
SECTION .bss
; pour la Xlib :
display:                resd 1
sreen:                  resd 1
win:                    resd 1
root:                   resd 1
gc:                     resd 1
ximage:                 resd 1

; Deux buffer :
buffer16:               resd 1
buffer32:               resd 1
```

Nous avons ici deux variables différentes, pour les buffers. Il est en effet plus facile de travailler en 32 bits en interne et de faire une conversion après, si nécessaire

4.1.3. La fonction d'initialisation

Il n'y aura que très peu de commentaires, dans cette section. À la place, je donnerai chaque fois le code C correspondant.

La première chose à faire est d'obtenir le `display` :

```
display = XOpenDisplay (NULL);
```

```
xor eax,eax
push eax
call XOpenDisplay
add esp, 4
mov [display], eax
```

Il faut maintenant obtenir le `screen`, la fenêtre `root` et le contexte graphique (`gc`). En C, trois macros nous permettent de les avoir facilement. Ces valeurs sont en fait des membres de structures. Si vous êtes motivés, allez voir dans les headers de la xlib. Sinon, faites un copier/coller de ce que je vous donne ici :

```
screen = DefaultScreen (display);
root = DefaultRootWindow (display);
gc = DefaultGC (display, screen);
```

```
mov edx, [eax+132]      ; eax vaut toujours [display]
mov [screen], edx

imul edx, 80
mov ebx, [eax+140]
mov ecx, [ebx+edx+8]
mov [root], ecx

mov edx, [ebx+edx+44]
mov [gc], edx
```

Une fois qu'on a tout ceci, on peut enfin faire quelque chose de visible ! Je parle de l'ouverture d'une fenêtre, bien sûr. Une fois de plus, référez-vous à la mage de manuel pour plus de détails sur les paramètres de la fonction.

```
win = XCreateSimpleWindow (display, root, 10, 10, width, height, 0, 0, 0);
```

```
xor ebx, ebx
push ebx                        ; background : 0
push ebx                        ; border : 0
push ebx                        ; border_width : 0
push d 480                      ; hauteur
push d 640                      ; width
mov bl, 10
push ebx                        ; y : 10
push ebx                        ; x : 10
push ecx                        ; fenêtre parent : root
push eax                        ; le display : eax
call XCreateSimpleWindow
add esp, (9*4)
mov [win],eax
```

Bien. Une fois que nous avons une fenêtre, il faut encore la "mapper" au `display`. Cette fonction permet également de faire passer notre fenêtre au premier plan.

```
XMapRaised (display, window);
```

```
push eax                ; eax vaut [win]
push dword [display]
call XMapRaised
add esp, 8
```

Une fois ceci fait, il ne reste plus grand chose. Nous devons créer un buffer un peu spécial qui sera lu par le serveur X pour remplir notre fenêtre. C'est ce qu'on appelle une `ximage`.
Ici, il faudra faire un peu attention. Cette fonction demande une valeur pour les bits par pixels. Comme je sais que mon écran est en 16bpp, j'ai mis cette valeur en dur. En faisant un peu de recherches, je ne doute pas que vous trouverez comment obtenir la profondeur de couleur de l'écran au run-time :)
`CopyFromParent` est en fait un #define qui vaut 0.

```
ximage = XCreateImage (display, CopyFromParent, depth, ZPixmap, 0,
                       NULL, width, height, 32, 0);
```

```
xor eax, eax
push eax                ; padding : 0
push dword 32           ; alignement
push dword 480          ; hauteur
push dword 640          ; largeur
push eax                ; 0
push eax                ; 0
push dword 2            ; ZPixmap
push dword 16           ; ou 32 : bits par pixel
push eax                ; CopyFromParent : 0
push dword [display]    ; display
call XCreateImage
add esp, (4*10)
mov [ximage], eax
```

Enfin ça y est, notre fenêtre est prête ! Il ne nous reste dès lors qu'à allouer un buffer dans lequel nous dessinerons, avant de l'envoyer à notre `ximage`.
Nous allons en fait allouer *deux* buffers : l'un en 32 bits, dans lequel nous dessinerons, l'autre en 16bpp, que nous utiliserons au cas où nous devrions faire une conversion.

```
buffer16 = malloc(largeur * hauteur * 2);
buffer32 = malloc(largeur * hauteur * 4);
```

Notez que je regroupe les deux appels à `malloc` en un seul :

```
push dword (640 * 480 * 6)
call malloc
mov [buffer16], eax
add eax, (640 * 480 * 2)
mov [buffer32], eax
add esp, 4
```

4.2. Blitter notre buffer

Tout ça, c'est bien joli, mais tant que nous n'avons pas de fonction pour envoyer notre buffer à l'écran, nous n'irons pas loin. Heureusement (vous vous en doutiez), il existe une fonction pour ça. Avant de l'appeler, il faudra juste prendre soin de spécifier quel buffer sera utilisé par l'`ximage`.

```
ximage->data = buffer;
XPutImage(display, window, gc, ximage, 0, 0, 0, 0, largeur, hauteur);
```

```
mov ebx, [buffer16]     ; ou [buffer32] si vous êtes en 32bpp
mov eax, [ximage]
mov [eax+16], ebx

xor eax, eax
push dword 480          ; hauteur
push dword 640          ; largeur
push eax                ; dest_y : 0
push eax                ; dest_x : 0
push eax                ; src_y : 0
push eax                ; src_x : 0
push dword [ximage]     ; ximage
push dword [gc]         ; gc
push dword [win]        ; window
push dword [display]    ; display
call XPutImage
add esp, (10*4)
```

4.3. Conversion de couleurs

Si, comme moi, votre affichage est en 16 bits et que vous utilisez le buffer en 32 bits, vous aurez besoin d'une fonction qui se charge de convertir tout ça.
En voici une, probablement très peu optimisée (sauf peut-être en taille, et encore) mais qui a le mérite d'exister.
Non, n'insistez pas, je ne ferai pas de commentaires.

```
Convert_32_to_16:
        mov esi, [buffer32]
        mov edi, [buffer16]
        mov ecx, (640 * 480)

        .Lconv:
                xor ebx,ebx
                lodsd
                shr al, 3
                mov bl, al
                shr ax, 5
                and ax, 11111100000b
                or bx, ax
                shr eax, 16
                and al, 11111000b
                or bh, al
                mov [edi], bx
                add edi, 2

                dec ecx
        jnz .Lconv
ret
```

Je vous laisse lâchement vous débrouiller pour les autres conversions, si vous estimez en avoir besoin.

5. Le temps

Avoir un moyen pour connaître le temps écoulé depuis le début du programme est très utile. Cela permet d'avoir la même vitesse d'exécution quelle que soit la machine.

5.1. Initialiser le compteur

La première chose à faire est de demander l'heure au système, par l'intermédiaire de `gettimeofday`. Cette fonction nous donnera un nombre de secondes et un nombre de micro-secondes, que nous aurons soin de sauvegarder. Cela correspondra pour nous au moment du lancement de notre programme.

Nous aurons besoin d'un `EXTERN` et de quelques variables. Notez que leur ordre est important : la fonction `gettimeofday` demande un pointeur sur une structure comportant deux entiers.

```
EXTERN gettimeofday

SECTION .bss
        start_time_sec:         resd 1  ; secondes au lancement
        start_time_usec:        resd 1  ; micro-secondes

        now_time_sec:           resd 1  ; secondes actuellement
        now_time_usec:          resd 1

        Ticks:                  resd 1  ; nombre de 'ticks' écoulés
```

Et le code :

```
push dword 0
push dword start_time_sec
call gettimeofday
add esp, (2*4)
```

5.2. Connaître le temps écoulé

Pour connaître le temps écoulé, il suffira dès lors de faire un nouvel appel à `gettimeofday` et de soustraire la valeur de départ à celle nouvellement obtenue.
Mais comme les valeurs que nous avons sont scindées en secondes et micro-secondes, quelques calculs seront nécessaires. Pour obtenir une valeur en millisecondes (que j'appelle *ticks*), le calcul sera le suivant :

$Ticks = (now\_time\_sec - start\_time\_sec) * 1000 + (now\_time\_usec - start\_time\_usec) / 1000$

Voici le code correspondant :

```
; appel à gettimeofday
push dword 0
push dword now_time_sec
call gettimeofday
add esp, (2*4)

; Maintenant, faire le calcul
; secondes * 1000 :
mov eax, [now_time_sec]
mov ebx, 1000
sub eax, [start_time_sec]
mul ebx                         ; eax = (now.sec - start.sec) * 1000
mov ecx, eax

; micro-secondes / 1000 :
mov eax, [now_time_usec]
sub eax, [start_time_usec]
cdq
idiv ebx                        ; eax = (now.usec - start.usec) / 1000

; additionner les deux valeurs
add eax, ecx

; et enregistrer le résultat
mov [Ticks], eax
```

Pour ma part, j'ai simplement mis le code d'initialisation avec celui de la Xlib, et le calcul du temps écoulé se fait lors du blit.
Notez qu'avec cette méthode, la granularité du timer est de 10ms. C'est la raison pour laquelle on ne calcule pas le temps écoulé depuis le dernier blit, mais bien depuis le début du programme, pour éviter d'avoir de trop grosses erreurs.

6. Le son avec OSS

Comme pour l'utilisation de la Xlib, il n'y a pas de réel problème en ce qui concerne la gestion du son. De plus, l'avantage ici est que tout est faisable par l'intermédiaire d'appels systèmes. Pas besoin de lier notre programme à une bibliothèque, nous évitons donc les pertes de place.

De manière générale, lorsqu'on doit gérer le son via OSS, on opère de la manière suivante :

1. Ouverture du périphérique (*/dev/dsp* généralement)
2. Configuration des paramètres (fréquence, mono/stéréo et échantillonage)
3. Ecriture des samples

Voyons comment ça se passe dans la pratique.
Premièrement, nous aurons besoin de quelques variables.

```
SECTION .bss
file_desc       resd 1          ; le descripteur de fichier pour /dev/dsp
data            resd 22050      ; une seconde de son

%assign AFMT_U8 8

SECTION .data
devdsp          dd "/dev/dsp", 0 ; le nom du périphérique
channels        dd 0            ; mono (1 pour stéréo)
format          dd AFMT_U8      ; 8 bits non signé
rate            dd 22050        ; 22 kHz
```

Vous vous demandez sans doute d'où je tire la valeur atribuée à `AFMT_U8`. Je ne l'ai bien sûr pas inventée. Elle est définie dans un des fichiers d'en-tête de OSS. J'ai donc simplement écrit un petit programme en C, chargé d'afficher cette valeur. Vous trouverez plus de détails sur OSS en lisant l'Open Sound System Programmer's Guide dont la référence est disponible dans la section Liens.

Une fois tout ceci déclaré, il ne nous reste plus qu'à coder.
L'ouverture de */dev/dsp* se fait par l'intermédiaire de l'appel système 5.

```
mov eax, 5                      ; open
mov ebx, devdsp                 ; l'adresse du nom du périphérique
mov ecx, 1                      ; pour ouvrir en write-only
int 0x80
test eax,eax
js error                        ; à vous de travailler...
mov [file_desc], eax
```

Une fois ceci fait, nous devons régler le périphérique avec les *IOCTL*s. Vous trouverez une liste des valeurs des *IOCTL*s dans votre fichier */usr/src/linux/asm/ioctl.h*.

```
; On passe en mono...
mov eax, 54 ; ioctl
mov ebx, [fd]
mov ecx, SNDCTL_DSP_STEREO  ; l'ioctl voulu
mov edx, channels           ; pointeur sur les données
int 0x80
test eax,eax
js error

; ... 8 bits ...
mov eax, 54 ; ioctl
mov ebx, [fd]
mov ecx, SNDCTL_DSP_SETFMT
mov edx, format
int 0x80
test eax,eax
js error

; ... 22 kHz
mov eax, 54 ; ioctl
mov ebx, [fd]
mov ecx, SNDCTL_DSP_SPEED
mov edx, rate
int 0x80
test eax,eax
js error
```

Comme vous l'aurez peut-être constaté, les valeurs données pour les paramètres sont passés par référence. C'est parce que l'appel à *ioctl* doit pouvoir modifier leur valeur. Il ne se gênera d'ailleurs pas pour le faire. Si la valeur demandée est impossible à obtenir, il en mettra une autre, qu'il sauvera dans votre variable. C'est donc à vous de vous assurer des paramètres réellement utilisés, chose que je ne fais pas ici.

Nous allons maintenant nous créer un petit sample -- très basique, je vous rassure -- dans `data`. C'est lui que nous enverrons à la carte son.

```
mov edi, data
mov ecx, 22050
xor al,al
.create_data:
        stosb
        inc al
        dec ecx
jnz .create_data
```

Nous sommes presque au bout de nos peines, il ne reste plus qu'à envoyer ce sample à la carte son.

```
mov eax, 4  ; write
mov ebx, [fd]
mov ecx, data       ; pointeur sur les données
mov edx, 22050      ; nombre de bytes à écrire
int 0x80
test eax,eax
js error
```

Et voilà !

7. Aller plus loin

Si vous avez compris tout ce que j'ai raconté, il ne devrait pas vous être difficile de continuer à découvrir par vous-mêmes.
Lors de la réalisation d'une intro, il sera peut-être avantageux d'utiliser (si les règles le permettent) une bibliothèque comme SDL, qui offre l'avantage de minimiser le nombre d'appels externes pour l'initialisation. Vous pouvez aussi utiliser OpenGL, ou toute autre bibliothèque qui vous semblera correspondre à vos besoins.
Gardez cependant à l'esprit que chaque fonction externe utilisée prendra de la place dans votre exécutable final

Si votre but était surtout d'apprendre comment interfacer l'assembleur avec les bibliothèques existantes, j'espère que cette petite introduction vous aura servi. Vous l'aurez remarqué, c'est assez simple.

Et n'oubliez jamais : c'est toujours intéressant de voir comment gcc transforme votre code C en assembleur.

Bon amusement !

Allergy

A. Annexes

A.1. Hello World avec les appels systèmes

Le code de *hello_syscall.asm* :

```
        BITS 32

        SECTION .data
                chaine      db "Hello world !",10

        SECTION .text
                GLOBAL _start

                _start:
                mov eax, 4
                mov ebx, 1
                mov ecx, dword chaine
                mov edx, 14
                int 0x80
                mov eax, 1
                int 0x80
```

Et ce que ca donne :

```
$ nasm -f elf hello_syscall.asm
$ gcc -nostdlib hello_syscall.o -o hello_syscall
$ ls -l hello_syscall
-rwxr-xr-x    1 allergy  allergy       861 avr  9 00:00 hello_syscall

$ strip -s hello_syscall
$ ls -l hello_syscall
-rwxr-xr-x    1 allergy  allergy       484 avr  9 00:01 hello_syscall

$ sstrip hello_syscall
$ ls -l hello_syscall
-rwxr-xr-x    1 allergy  allergy       174 avr  9 00:01 hello_syscall

$ ./hello_syscall
Hello world !
```

Comme vous le voyez, une simple dépendance externe peut prendre beaucoup de la place. Notez qu'avec un exécutable de 174 octets, la compression fait passer la taille à 197 octets.

B. Liens

- **Nasm** : http://nasm.2y.net/
- **ELFkickers** : http://www.muppetlabs.com/~breadbox/software/elfkickers.html
- **Liste des appels systèmes** : http://home.snafu.de/phpr/lhpsysc0.html
- **Tout sur l'assembleur sous Linux** : http://www.linuxassembly.org/
- **OSS Programmer's guide (PDF)** : http://www.opensound.com/pguide/oss.pdf
- **Linux scene** : http://www.lnxscene.org/
- **Simple DirectMedia Layer** : http://www.libsdl.org/