Assembleur Intel avec NASM sur machine virtuelle sans système d'exploitation

par Etienne Sauvage (http://esauvage.developpez.com/)

Date de publication : 20 mars 2012

Dernière mise à jour :

À la découverte de l'assembleur, le retour. Suite de **ceci**.

I - Un mauvais malloc pour KarlOS....3
I.1 - De l'utilisation de la mémoire....3
I.2 - De la gestion de la mémoire....3
I.3 - De l'allocation....4
I.3.a - Méthode radicale et infaillible....4
I.3.b - Méthode moins radicale et plus faillible....4
I.3.c - Méthode fumeuse....4
I.4 - De la sylviculture....4
I.4.a - Pourquoi réinventer la roue ?....4
I.4.b - Revenons à nos moutons....4
I.5 - De malloc....5
- - De l'algorithme....5
I.6 - Du code....6
II - Un shell pour KarlOS....6
II.1 - Du shell....6
II.2 - De la ligne de commande....6
II.3 - De l'analyseur....7
II.3.a - Trouver la fonction demandée....7
II.3.b - L'appeler....9
II.4 - De la lecture de la chaîne....9
II.5 - Du code....11
III - Des fonctions pour KarlOS....11
III.1 - Pour rigoler....11
III.2 - Table....12
III.3 - CPU....12
III.4 - Date....15
III.5 - Du code....17
IV - Remerciements....17

I - Un mauvais malloc pour KarlOS

Au chapitre précédent, nous avons obtenu la carte de la mémoire, qui nous était aussi utile qu'une carte de l'Ouganda à votre serviteur (qui n'y a jamais mis les pieds et qui ne compte absolument pas importuner les ougandais de son inopportune présence). Bref, la mémoire ça s'utilise, sinon c'est Alzeihmer.

I.1 - De l'utilisation de la mémoire

On utilise déjà de la mémoire. Le code de KarlOS est en mémoire. Nous avons déjà donné des adresses de chargement, nous lisons et écrivons dans des cases mémoire et ce depuis le début. Alors, de quoi qu'est-ce que je vous chante avec mes histoires de carte mémoire ?

Et bien, figurez-vous qu'un développeur, ou qu'un concepteur de système d'exploitation, aussi génial soit-il, n'est pas omniscient. Autrement dit, il ne peut pas tout prévoir, ni n'en a le temps. La quantité d'information et de temps dans un homme fini est nécessairement finie. Conséquemment, il a peu d'idée sur quelle partie de la mémoire est utilisée à un instant donné pour quel usage : sinon, il serait capable de prévoir tous les comportements de tous les utilisateurs, et il ne serait pas développeur, il serait maître du monde.

Imaginons le cas suivant : nous écrivons cette page web. L'ensemble des caractères tapés doit être stocké en mémoire. Cela prend de la place, et il ne faudrait pas que ces caractères soient stockés sur l'emplacement mémoire occupé par le gestionnaire d'interruptions clavier, par exemple. De surcroît, je mets au défi qui que ce soit de donner, avant la fin de la composition, l'espace mémoire occupé par cette même composition, à l'octet près. Indépendamment du débat entre détermination et indétermination de l'univers, je n'ai pas les moyens de connaître la quantité de mémoire dont j'ai besoin avant de faire les choses. Il s'ensuit qu'un système d'exploitation, même indigne de ce nom, doit avoir un système permettant de stocker des choses en mémoire sans dézinguer les autres choses dont on a encore besoin.

I.2 - De la gestion de la mémoire

Le paragraphe précédent, quoi qu'obscur, signifie que KarlOS se doit de gérer la mémoire. En simplifiant, cela consiste à marquer l'utilisation ou la disponibilité de la mémoire. Je veux stocker une chaîne de caractère. Je dis à KarlOS : "Hey, je veux de la mémoire !". Et je m'attends à ce que KarlOS me réponde : "Tiens, en v'là. Va donc voir au 31 rue de la forge 43189 TRIFOUILLY CEDEX.". Oui, je m'attends à recevoir l'adresse de la mémoire demandée. KarlOS doit donc savoir que le 31 rue de la forge à Trifouilly est vide d'occupant.

Poursuivons cette analogie. J'occupe, avec ma chaîne de caractères, le 31 rue de la forge à Trifouilly. Nous y vivons paisiblement, mais bon, vous savez ce que c'est, on vieillit, on souffre d'incontinence pianoteuse, on fait des cochonneries, et la chaîne grandit. Elle grandit tant et si bien qu'elle ne tiendra plus bien longtemps à cette adresse, parce que le 31 rue truc, il est prévu pour 256 octets et pas 3124. Et le 33, il est occupé par un terroriste qui n'a pas l'air d'apprécier nos débordements. On demandera donc à KarlOS de nous allouer une nouvelle adresse, plus grande.

Je continue avec ma grosse chaîne, mais maintenant, j'en ai marre, et je n'en ai plus besoin. Je rends les clefs. L'adresse que j'occupais doit maintenant être considérée comme vide. KarlOS, prévenu, reprend les clefs. Mécaniquement, ma chaîne disparaît.

Je suis maintenant victime du devoir et je disparais, purement et simplement, dézingué par le terroriste sus-cité, ou par KarlOS lui-même sous prétexte d'opération non-conforme, ne riez pas, je suis sûr que cela vous est déjà arrivé. KarlOS doit considérer mon adresse comme libre.

Cette analogie fonctionne bien, et montre que la gestion de la mémoire par le système d'exploitation, cela ressemble à la gestion foncière en République Populaire de Chine.

Pour résumer, la gestion de la mémoire doit se faire par, minimum syndical, trois fonctions : allocation, réallocation et libération. Aujourd'hui, ce sera bien assez de travail, on n'en fera qu'une, et encore, mal : l'allocation de mémoire.

I.3 - De l'allocation

I.3.a - Méthode radicale et infaillible

Oui. Une méthode radicale et infaillible existe, dépourvue de tout problème de segmentation, perte, disproportion, complexité. Elle n'a qu'un seul et unique problème : elle ne permet pas d'accéder à la mémoire.

I.3.b - Méthode moins radicale et plus faillible

Les systèmes d'exploitations, les vrais de la vraie vie, ont des méthodes difficiles et poussées pour allouer de la mémoire. Pensez donc : il s'agit, en gros, de savoir à tout instant si un octet particulier est utilisé ou pas. Si on utilise un bit par octet, c'est-à-dire que pour chaque octet, on stocke si oui ou non il est utilisé, on a besoin du huitième de la mémoire. Autrement dit, sur votre ordinateur disposant de 2 gigaoctets, il y a 256 mégaoctets pris rien que pour savoir si chaque octet est libre ou pas. Et encore, à ce niveau-là, on ne sait pas par qui : donc impossible de mettre à jour ces bits si ceux qui l'utilisent ne le font pas eux-même. Donc, zou, on dégage, pas la bonne méthode. C'est un problème actuel et difficile, dont la solution n'est ni parfaite ni visiblement unique.

I.3.c - Méthode fumeuse

Nous, on n'est pas des vrais. Nous, on est des rigolos, des fumistes et notre but est juste d'appréhender une possibilité pour le faire. Alors, autant qu'elle soit instructive et simple à mettre en oeuvre. Aussi allons-nous parler fumisterie, et donc herbe. Ah non, pardon, arbres.

I.4 - De la sylviculture

Voici le premier concept de programmation qui soit un peu indépendant du support. Toute notre progression est partie d'à peu près rien, tout de même. Maintenant, nous comprenons mieux comment fonctionne un ordinateur. Mais nous n'avons vu que des choses à peu près inutiles au commun des développeurs. Il serait temps de s'y essayer.

Si vous avez tenté de parler à un vrai développeur d'assembleur, de VESA, d'interruption et de mode protégé, je gage qu'on vous a regardé avec des yeux ronds. Ca c'est dans le meilleur des cas, si vous avez causé avec un dinosaure du Fortran. Si c'était un vieux programmeur C, vous avez récolté le dédain et la condescendance. Et si vous avez à faire à un programmeur objet, si vous croyez en un dieu, remerciez-le chaudement d'être toujours en vie : ces gens ont une sainte horreur de la programmation bas niveau, qu'ils accusent de tous les maux. D'ailleurs, à ce propos.

I.4.a - Pourquoi réinventer la roue ?

Parce que si on ne réinventait pas la roue en permanence, on en serait toujours à les utiliser à l'horizontale. C'est parce que Linus Torvald a réinventé un noyau qu'on a Linux. Et qu'on refasse les choses en permanence permet de montrer qu'elles sont humainement faisables. C'est parce que beaucoup de personnes font des avions qu'on sait toujours les faire.

I.4.b - Revenons à nos moutons

Donc, les vrais développeurs de la vraie vie utilisent des trucs, qu'on retrouve quel que soit le langage de programmation. Et, *in fine*, ça se retrouve en assembleur. Celui que nous allons aborder s'appelle un arbre.

Un arbre, ça a, en première approximation, des racines, un tronc, des branches, des branches, encore des branches, et des cochonneries au bout des branches. Comme nous sommes des informaticiens, nous avons la réputation d'être peu au fait des choses de la nature. Et bien, soyons à la hauteur de notre réputation et supprimons ce qu'on ne voit pas; *exit* les racines. Tant qu'à faire, comparons un tronc à une branche : oui, ça ressemble beaucoup quand même. Allez, on le prend en automne, quand il n'a plus ni feuilles, ni fleurs, ni fruits, ni futile future chose commençant par

un f, et finalement, on obtient ça : un arbre, c'est une branche qui porte des branches, qui portent des branches, qui elles-même portent des branches, ou pas. Chaque branche peut porter des branches, ou ne pas en porter. Voici un arbre numérique. Pas très sexy, mais un outil intéressant. Il y a un vocabulaire particulier pour ces arbres. La branche qui est au ras du sol s'appelle la racine. Oui, c'est un terme dévoyé par rapport à une vraie racine, mais on est comme ça, nous les informaticiens : la botanique, c'est à dévoyer. Chaque branche s'appelle aussi un noeud. La racine est donc un noeud, le noeud-racine. Oui, c'est lourd. Chaque branche qui part d'une autre branche est un fils de cette branche. C'est un noeud-fils. La branche qui a des fils est, du coup, un père : le noeud-père. Il n'y a malheureusement pas de noeud qui soit juste un noeud, donc pour les noeuds-noeud, vous repasserez, merci.

Si on a plusieurs noeuds-racine, on dispose d'une forêt. Tout à fait, on commence la forêt à deux arbres.

On peut décider arbitrairement qu'une branche ne peut avoir que deux fils au plus. On a alors défini un arbre binaire. Et tout arbre peut se réduire à un arbre binaire. Vous m'en ferez la démonstration.

I.5 - De malloc

Ah, *malloc*. Quelle étoile ! Quel phare éternel au firmament obscur du ciel pesant sur les épaules chastes du développeur ! Quel éclat, quel guide vers la nativité d'un programme fonctionnel ! J'espère que j'en fais trop, là.

malloc. Comme Memory ALLOCation. Allocation de mémoire. La base. La brique essentielle. On en était encore à bidouiller avec sa lampe à souder que *malloc* existait déjà. Les interfaces graphiques étaient encore inconcevables que *malloc* était là. Cette fonction a accompagné les vieux de la vieille aussi sûrement que ceux-ci accompagnaient l'empereur. Et, comme nos vénérables ancêtres, la jeune garde veut la cacher, la supprimer, la faire disparaître. Sauf que la vieille est essentielle, vitale, indispensable. Les jeunots ne font que la planquer, en fait. Elle existe toujours, même maquillée comme une voiture volée au travers des *new*, les *afficionados* du C++ apprécieront.

Bref, malloc est la fonction que nous allons tenter. Je n'en changerai pas le nom, tellement elle est essentielle.

- - De l'algorithme

Tout le monde s'en doute, malloc utilisera, quelque part, un arbre. Je me permets de préciser qu'*a priori*, pour un vrai système d'exploitation, ce n'est pas comme ça qu'il faut faire. Nous avons un malloc à faire dans un système d'exploitation en mousse et nous devons progresser en assembleur, nous allons donc en profiter pour toucher du doigt les arbres.

Première hypothèse : il sera demandé à malloc une quantité de mémoire suffisante pour que le nombre de blocs demandés soit négligeable devant la taille totale disponible. Seconde hypothèse : le gaspillage n'est pas répréhensible.

La mémoire va être représentée par une forêt. Chaque arbre de cette forêt représente un espace contigu, soit une ligne de la carte du **chapitre précédent**. Un arbre est constitué d'un seul noeud comme point de départ. Ce noeud est un espace de **x** bits.

- L'adresse du premier octet de la zone mémoire concernée.
- La taille de cette zone mémoire.
- La quantité de mémoire actuellement allouée dans cette zone mémoire.
- L'adresse du noeud-fils gauche.
- L'adresse du noeud-fils droit.

Soit **x** = 160. Notre malloc est en 32 bits, les adresses et tailles mémoire sont donc de 32 bits pour traiter toutes les possibilités. Toutes ces adresses, je vais les appeler des pointeurs, parce que j'en parle comme ça depuis des années, l'habitude est trop forte. L'adresse du noeud-fils gauche est donc le pointeur sur le fils gauche.

Disons que malloc va fonctionner comme cela : on l'appelle avec la taille demandée dans **ECX**. Au retour, dans l'accumulateur **EAX**, malloc va placer l'adresse qu'il a trouvée. Par ailleurs, l'initialisation de **EAX** aura l'amabilité d'être faite avant l'appel à malloc. Et l'adresse invalide sera -1. La forêt est stockée dans le deuxième segment, adresse 0. On lit ce qu'il y a à cette adresse, première racine de la forêt. Si c'est -1, merci, au revoir. Sinon, on va regarder ce qui se passe dans cet arbre. On l'affiche, d'abord. Ensuite, on lit le contenu du noeud. La dernière valeur est la taille de la mémoire réservée dans cet arbre. On l'ajoute à **ECX**. Si le résultat est supérieur à **EDX** (qui contient la taille du segment), alors il faut en trouver un autre : on n'a pas assez de place dans celui-ci. Sinon, nous allons trouver dans cet arbre le bon noeud. Pour ce faire, nous comparons la taille souhaitée à la moitié de la taille disponible. Si la taille souhaitée est plus grande que cette moitié, on va prendre tout le bloc. Sinon, on regarde fils par fils, en profondeur d'abord.

I.6 - Du code

L'ensemble du code de ce chapitre se trouve ici : **Tuto13**

II - Un shell pour KarlOS

Nous allons maintenant faire quelque chose qui va nous faire ressembler KarlOs à un vieux DOS des familles. Ouais, un vieux DOS ! Euh... C'était quoi, le DOS, déjà ?

II.1 - Du shell

Sur l'air des Acadiens :

Tous les unixiens, toutes les linuxiennes,
font du shell et pianotent des commandes.

Le shell, ou la coquille, est le "machin" qu'on manipule sur un ordinateur. Il y en a deux versions : l'ILC (Interface en Ligne de Commande) et l'IUG (Interface Utilisateur Graphique), que les rosbifs appellent CLI et GUI respectivement. Les utilisateurs d'Unix et dérivés ainsi que les vieux de la Vieille connaissent l'ILC. Les blancs-becs et les Apple-eux jouent plus volontiers avec une IUG. Le DOS ne disposait que d'une ILC. L'IUG développée pour DOS s'appelle Windows, *never forget*. Donc, nous avons annoncé **DOS-like**, ce qui implique une ligne de commande.

II.2 - De la ligne de commande

Une ligne de commande commence par n'importe quoi et se termine par un appui sur la touche "Entrée". Soit, en gros, un appel à la fonction "litChaine". La ligne de commande est la boucle principale de KarlOS. Nous allons donc avoir quelque chose comme :

```
 mov ecx, 0x00800 ;Taille maximum d'une ligne : 2048 caractères
 call malloc
 mov esi, edi
princ:
 call litChaine
 mov edi, esi
 call analyseur
 jmp princ
```

On réserve 2048 octets pour stocker notre ligne de commande, et on lit. La source est égale à la destination, la destination est égale à la source, litChaine modifie la destination, on la recharge avec la source. Cette chaîne, on la donne à l'analyseur. Et on recommence, *ad vitam aeternam* (l'éternité se terminant par un appui sur un interrupteur).

II.3 - De l'analyseur

La chaîne lue, on la passe à un analyseur, dont le rôle est de mettre en correspondance une chaîne de caractère et une fonction. Donc, deux étapes :

- Trouver la fonction demandée
- L'appeler

II.3.a - Trouver la fonction demandée

Pour trouver la fonction demandée, il faut déjà savoir quelles sont les fonctions dont nous disposons. Une fonction n'est pour nous qu'une simple adresse, or la fonction va être appelée par une chaîne de caractères. Hors de question d'entrer l'adresse de la fonction au clavier, il faut lui associer un nom.

Il nous faut d'abord l'ensemble des noms de fonctions :

```
CommandesDispos:
db 30,11,12,22,15,0 ;"table"
db 13,26,31,0 ;"cpu"
db 0
```

On a défini deux fonctions : **table** et **cpu**. Peu importe ce qu'elles font actuellement. La liste se termine par une chaîne vide*db 0* Ca évite d'avoir à indiquer quelque part la taille de la liste, car quand on lit une liste, le problème est toujours de savoir où on s'arrête. Pour résoudre ce problème, le monde n'a imaginé que deux et deux possibilités seulement :

- avant de parcourir la liste, indiquer le nombre d'éléments à lire,
- marquer la fin.

Si vous trouvez une autre idée, gloire et confettis vous submergeront pour les siècles des siècles. En attendant, dans KarlOS, pour la liste des fonctions, on va marquer la fin.

On va ensuite faire un tableau de correspondance entre un nom et une adresse. Pour se faciliter la tâche, les adresses sont des étiquettes, et on va les stocker dans le même ordre que la liste, avec la même convention (une adresse nulle à la fin). Théoriquement, nul n'est besoin de cette adresse nulle supplémentaire, mais quand on aime, on ne compte pas. Et en l'occurence ça me facilite la vie. Voilà la liste des adresses :

```
AdressesCommandes: dd 0, 0, 0
```

Comment ça, vide ? Oui, vide, parce que je ne sais pas faire faire des calculs à NASM à la compilation. Donc, à l'exécution, on va appeler une petite fonction :

```
chargeFonctions:
 push edi
 mov edi, AdressesCommandes
 mov eax, tabCaracteres
 stosd
 mov eax, CPUInfos
 stosd
 pop edi
 ret
```

On stocke chaque adresse, c'est pas la mort.

EDI contient une commande. Correspond-elle à une de nos fonctions ? Pour le savoir, il va falloir comparer. L'instruction miracle est **cmpsb**, complétée par un préfixe répéteur, **repe**. On compare la source à la destination et on avance d'un octet tant que les deux sont égaux. Cette instruction nécessite néanmoins d'avoir le nombre

maximal d'itérations dans **ECX**. Le nombre maximal d'itérations, c'est bien évidemment la taille de l'une des chaînes de caractères. N'importe laquelle fera l'affaire, puisqu'au pire, on s'arrêtera au caractère nul terminal. Voici donc comment je compare deux chaînes :

```
 call tailleChaineB
 jz .chaineVide
 repe cmpsb
 jz .appelFonction
```

Ah, oui calculer la taille d'une chaîne. Vous allez voir, je me suis foulé :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; tailleChaineB             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ECX renvoie la longueur de la chaîne EDI       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
tailleChaineB:
 push edi
 push eax
 or ecx, 0xFFFFFFFF
 xor al, al
 repne scasb
 neg ecx
 dec ecx
 pop eax
 pop edi
 ret
```

Allez, réfléchissez, j'ai fait dans le bizarre. Tiquez, hurlez, ne restez pas les bras ballants, faites quelque chose, quoi !

Bon, alors, pourquoi est-ce que je n'ai pas appliqué la même technique dans la comparaison de chaîne et dans le calcul de la taille ? Le **or ecx, 0xFFFFFFFF**, mettant le nombre maximum d'itérations à l'infini (=2^{32} - 1, comme il se doit) ?

Parce qu'en bon père de famille, je m'engage à occuper bourgeoise... pouf, pouf. Parce qu'en bon ingénieur, je prévois le futur, et je sais que je vais devoir comparer mon entrée avec la chaîne représentant la commande suivante : il faut donc que j'avance à la prochaine chaîne. Pour arriver à la prochaine chaîne, j'ajouterai la taille de la chaîne courante (la taille comprend le caractère nul final). Avec ce parcours, nous avons donc l'analyseur :

```
analyseur:
 pushad
 mov edi, AdressesCommandes
 call tailleChaineD
 mov eax, edi
 mov edi, CommandesDispos
.boucle:
 push esi
 push ecx
 push eax
 call tailleChaineB
 pop eax
 jz .chaineVide
 repe cmpsb
 jnz .chaineVide
 jz .appelFonction
.chaineVide:
 add edi, ecx
 pop ecx
 pop esi
 add eax, 4
 loop .boucle
.retour:
 popad
 ret
```

La fonction **tailleChaineD** retourne la taille d'une chaîne de doubles mots, c'est exactement **tailleChaineB** à deux variantes près (et encore, on pourrait limiter à une variante).

II.3.b - L'appeler

Par contre, maintenant, il nous faut les appeler. Ce qui est rigolo, c'est qu'on a ce qu'il est convenu d'appeler des **pointeurs de fonction**, ce qui donne des sueurs froides à bien des développeurs.

Dans la fonction ci-dessus, nous avons l'appel à l'appel de fonction. Relisez tranquillement, ça va aller. C'est le **jz .appelFonction**. Et attention, le code est *super*, mais ***super*** compliqué :

```
.appelFonction:
 call [eax]
 jmp .chaineVide
```

Les plus sagaces d'entre vous ont bien évidemment remarqué que ceci ne sert à rien. Les experts en sagacité ont incidemment remarqué que si deux fonctions portent le même nom, elles seront exécutées l'une après l'autre, vu que nous bouclons sur l'ensemble des fonctions à chaque fois. Bon, on supprime cette horreur pour avoir l'analyseur complet :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Analyseur syntaxique     ;;
;; ESI contient la chaîne à analyser  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
analyseur:
 pushad
 mov edi, AdressesCommandes
 call tailleChaineD
 mov eax, edi
 mov edi, CommandesDispos
.boucle:
 push esi
 push ecx
 push eax
 call tailleChaineB
 pop eax
 jz .chaineVide
 repe cmpsb
 jnz .chaineVide
 call [eax]
.chaineVide:
 add edi, ecx
 pop ecx
 pop esi
 add eax, 4
 loop .boucle
.retour:
 popad
 ret
```

II.4 - De la lecture de la chaîne

Il faut retravailler notre belle fonction, afin qu'elle ne puisse stocker plus de caractères que la zone allouée. Sinon, c'est le débordement, et on se retrouve à écrire dans le code système, ce qui est universellement considéré comme mauvais.

Donc, dans **ECX**, le nombre de caractères à lire. Ca donne la boucle à effectuer.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; litChaine              ;;
;;-------------------------------------------------------------;;
;; Ecrit dans la chaîne pointée par EDI les caractères écrits au  ;;
```

```
;; clavier, jusqu'à un retour chariot.         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
litChaine:
 push  eax
 push  ebx
 push ecx
 push  edi ; Sauvegarde d'EDI pour ne pas remonter avant le début de la chaîne

 mov  byte [derniereTouche], 0
.attend_clavier:
 cmp  byte [derniereTouche], 0
 jz .attend_clavier
 xor  eax, eax
 mov  al, [derniereTouche]
 mov  byte [derniereTouche], 0

 mov  ebx, traductionASCII
 add  ebx, eax
 mov  al, [ebx] ; Le caractère est dans AL

 cmp  al, 110   ;Traitement de l'espace arrière
 jne .testTab
 pop eax
 push eax
 cmp eax, edi
 jae .attend_clavier
 dec  edi
 mov al, 51
 call afficheCaractere
 sub  word [curseur+2], 0x10 ; On recule le curseur de 2 caractères
 js .bordGauche
 jmp .attend_clavier

.testTab:
 cmp  al, 111   ;Traitement de la tabulation avant
 je .tab

.testEntree:
 cmp  al, 112
 je .fin_attend_clavier

 test byte[touchesActives + 5], 0b0000100 ;On teste le shift gauche enfoncé
 jnz .shift
 test byte[touchesActives + 6], 0b1000000 ;On teste le shift droit enfoncé
 jz .RAS
.shift:
 cmp al, 114   ;On ne stocke pas un Shift gauche comme touche
 je .attend_clavier
 cmp al, 115   ;On ne stocke pas un Shift droit comme touche
 je .attend_clavier

 add al, 73
.RAS:
 stosb
 call afficheCaractere
.boucle:
 loop .attend_clavier
.fin_attend_clavier:
 xor  al, al  ;Stocke une fin de chaîne
 stosb
 mov  al, 51  ;Supprime le curseur à la fin de la ligne
 call afficheCaractere
 add  word [curseur], 7
 mov  word [curseur+2], 0
 dec  edi

.retour:
 pop  ebx
 pop  ecx
 pop  ebx
 pop  eax
 ret
```

```
.bordGauche:
 mov  ax, word [XResolution]
 sub  ax, 8
 mov  word [curseur+2], ax
 xor  ah, ah
 mov  al, byte [lignesParCaractere]
 sub  word [curseur], ax
 jmp .attend_clavier

.tab:
 mov  eax,
 (51<<24)+(51<<16)+(51<<8)+51 ; La tabulation avant est remplacée par 4 espaces dans la chaîne
 stosd
 push ecx
 mov eax, 51
 mov ecx, 4
.afficheTab:
 call afficheCaractere
 loop .afficheTab
 pop  ecx
 sub ecx, 4
 jle
 .fin_attend_clavier ; Ca teste si ZeroFlag == 0 & Sign Flag <> Overflow Flag. Comme OF == 0 a priori
 jmp .boucle
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Fin litChaine             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

II.5 - Du code

L'ensemble du code de ce chapitre se trouve ici : **Tuto14**

Pour la prochaine fois, je vous montre quelque chose qu'un langage de haut niveau ne pourra jamais faire... Ce qui vous expliquera pourquoi le code de ce chapitre est différent de celui que vous venez de lire...

III - Des fonctions pour KarlOS

On a fait une ligne de commande dans laquelle on a mis deux fonctions. On va voir maintenant ce qu'elles font, voire, on va en rajouter.

III.1 - Pour rigoler

Mais avant cela : rigolage, comme promis. Alors, je me suis dit, comme ça, dans le train : "Mais z'au fait, mon garçon, c'est quoi, la taille d'une chaîne de caractère ?" Réponse : c'est la position de la première occurence du caractère 0. Comme, ma foi, on fait un **SCASB**, appeler tailleChaineB, c'est appeler indexChaineB avec 0 dans **AL**. Et ça, personne n'en parle. Et en quoi c'est rigolo ? C'est rigolo parce que (faut toujours tout expliquer) avec un seul code, on a deux fonctions ! Oui, monsieur, deux fonctions pour le prix d'une ! Même le regretté Ritchie ne nous ferait pas un coup pareil. Regardez plutôt :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; tailleChaineB             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ECX renvoie la longueur de la chaîne EDI       ;;
;; AL est mis à 0             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
tailleChaineB:
 xor al, al
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; indexChaineB              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ECX renvoie l'index du caractère AL dans la chaîne EDI   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
indexChaineB:
 push edi
 or ecx, 0xFFFFFFFF
 repne scasb
 neg ecx
 dec ecx
 pop edi
 ret
```

Et, ça, aucun compilo au monde ne l'aurait fait... Pas que ce soit forcément bien, notez.

III.2 - Table

La toute première fonction codée pour la ligne de commande de KarlOS. Cela ne fait qu'afficher notre table de caractères.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; tabCaracteres              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Affiche les 122 premiers caractères de KarlOS     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
tabCaracteres:
 push ecx
 push eax
 mov ecx, 122
 xor eax, eax
.afficheAlphabet:
 call afficheCaractere
 inc al
 loop .afficheAlphabet
 call retourChariot
 pop eax
 pop ecx
 ret
```

III.3 - CPU

La toute deuxième (quoi, y'a bien la toute première) fonction codée pour la ligne de commande de KarlOS. Cela ne fait qu'afficher une malheureuse information sur le microprocesseur. Bon, on utilise l'instruction **CPUID**, qui est une instruction versatile, voire lunatique. Utilisez-la dans un langage évolué, celle-là, qu'on rigole un peu. Bon, je vous la livre en l'état.

Il faut d'abord tester si cette fonction est supportée par le processeur. Ce n'est pas évident. Ca se passe dans le registre EFLAGS. Le bit n°21 de ce registre (notez qu'on est nécessairement au moins en 32 bits) peut être modifié ssi **CPUID** existe. C'est la base du test. Mais, car il faut bien un "mais" sinon on s'ennuie, mais donc, aucune instruction ne permet de modifier ce registre. On aurait voulu nous mettre des bâtons dans les roues qu'on ne s'y serait pas pris autrement. La ruse (car tous les programmeurs de système sont des fourbes félons fielleux et fallots fomentant ferme de fabuleux mais faux complots, bienvenue chez nous, nouvel initié), la ruse consiste à sauver ce registre sur la pile. **PUSHFD** est notre premier homme. Comme c'est sur la pile, hé, vieux, 'suffit de le récupérer dans un vrai registre ! **POP EAX**, et nous avons EFLAGS dans un registre d'honnête homme. Il suffirait que **POPFD** ait le bon goût d'exister, et nous serions les plus heureux développeurs modifieurs d'EFLAG. Savourons notre bonheur, elle existe. Donc, on récupère dans un registre "normal" EFLAGS, on fait un **XOR** sur le bit 21, on le met dans le registre maudit, on le relit juste derrière, si la valeur est égale à celle avant le **XOR**, **CPUID** n'est pas implanté dans le processeur, merci d'avoir joué, repassez nous voir à la prochaine release.

```
 pushfd ; Sauve le registre EFLAGS
 pop eax ; Le récupère dans EAX
 mov ebx, eax ; Le copie dans EBX pour la comparaison postérieure
 xor eax, 00200000h ; on change l'état du bit 21
 push eax ; On le met sur la pile
 popfd ; Pour le remettre dans le registre EFLAGS
 pushfd ; On re-sauve le registre EFLAGS (c'est l'idée du test)
```

```
 pop eax ; Le récupère dans EAX
 cmp eax, ebx ; Et on le compare avec sa sauvegarde
 jz .NO_CPUID ; Si rien n'a changé, l'instruction CPUID n'est pas supportée.
```

Admettons que l'instruction existe. Que fait-elle ? Et bien, très chers frères, il faut savoir que cette instruction ô combien utlie à l'homme de bien, fut conçue tout près de nos verts bocages, au milieu des vergers de pommes et de poires, sous les douces effluves de Camembert, dans notre Normandie. Conséquemment, ce qu'elle fait, ça dépend. En gros, il en existe autant de versions qu'il y a de fondeurs de processeurs, de familles de processeurs et de modèles de processeurs. Néanmoins, elle fonctionne un peu comme une interruption. Suivant ce qu'on a dans le registre **EAX**, on obtient différents résultats. Avec 0 (alias "la bulle"), on obtient, partout, toujours et en tout lieu et nonobstant les conditions sus-citées, le nombre de fonctions dont dispose **CPUID**, mais aussi une chaîne de caractères identifiant le fondeur. On n'arrête pas le progrès des sciences forensiques. Vous noterez, au passage, le tropisme de la valeur zéro.

```
 xor eax, eax
 cpuid
 mov [nbFonctions], eax
 mov edi, chaine ;Affichage de la marque
 mov eax, ebx
 stosd
 mov eax, edx
 stosd
 mov eax, ecx
 stosd
 xor al, al
 stosb
 mov esi, chaine
 call conversionASCII
 call afficheChaine
 call retourChariot
```

Dans **EAX**, on récupère le nombre de fonctions disponibles dans **CPUID**. Donc, premier coup, on a la marque et jusqu'où on peut aller (cf juste au-dessus) :

```
.marqueAMD:
 inc al; AL = 2
.marqueIntel:
 inc al; AL = 1
 jmp .marqueInconnue

 xor eax, eax
 cmp [Intel.ebx], ebx
 je .marqueIntel
 cmp [AMD.ebx], ebx
 je .marqueAMD
.marqueInconnue:
 mov byte [Marque], al
 call chargeCPUIDFamille ; On charge les familles de processeurs du constructeur
```

La fonction chargeCPUIDFamille prépare un tableau dans lequel on trouvera les noms des familles de processeur du constructeur considéré.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EAX contient la marque du CPU, et est détruit ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
chargeCPUIDFamille:
 push edi
 push ecx

 mov edi, listeFamilles
 mov ecx, 16
 cmp al, 1 ;Test Intel
 je .setEAXIntel2
 cmp al, 2 ;Test AMD
 je .setEAXAMD2
.suite:
 call chargeListeSansTest
```

```
 pop ecx
 pop edi
 ret

.setEAXIntel2:
 mov eax, Intel.Familles
 jmp .suite

.setEAXAMD2:
 mov eax, AMD.Familles
 jmp .suite
```

En gros, on se contente d'initialiser les paramètres de ChargeListeSansTest, que je vous présente.

```
;; ECX contient le nombre de chaînes à indexer
;; EAX contient l'adresse de la liste des chaînes
;; EDI contient l'adresse de la liste d'index à mettre à jour
chargeListeSansTest:
 push eax
 push ebx
 mov ebx, eax
.chaineSuivante:
 stosd
 push ecx
 xchg edi, ebx
 xor al, al
 or ecx, 0xFFFFFFFF
 repne scasb
 mov eax, edi
 xchg edi, ebx
 pop ecx
 loop .chaineSuivante
 pop ebx
 pop eax
 ret
```

Cette fonction indexe les adresses de ECX chaînes de caractères consécutives en mémoire dans un tableau. Ca n'utilise que des registres, et encore, pas tous, et je n'ai pas plus élégant. L'idée est de stocker l'adresse du caractère suivant chaque caractère nul, plus le premier. Pour ce faire, on va échanger le contenu de EDI avec EBX souvent. EBX parce qu'il m'en fallait un, pas envie de le mettre sur la pile. Tout d'abord, on met dans EBX l'adresse des chaînes à indexer. On met dans EDI EAX, qui contient la même chose que EBX. On échange EDI et EBX, EDI contient maintenant les chaînes à indexer. SCASB va comparer l'octet à l'adresse EDI à la valeur dans AL (donc mettre le drapeau d'égalité au besoin), puis incrémenter EDI. En mettant AL à zéro et en répétant SCASB tant que le drapeau égalité n'est pas mis, EDI sera placé sur le premier caractère de la chaîne suivante, soit pile celui qu'on veut. Non, vraiment, des fois on dirait que les instructions ont été faites pour la programmation. On échange EBX et EDI, on n'oublie pas de mettre EAX à jour, et on est repartis !

Ceci étant fait, allons, courage camarade, essayons la fonction n°1. Chez Intel et AMD, elle renvoie des champs cabalistiques donnant l'identité du processeur dans EAX. Ces **gens bien documentés** nous donnent la structure de ce que nous cherchons. Occupons-nous d'abord de la famille. Car la familia, c'est important, disait mon parrain.

Pour l'obtenir, il faut appeler la fonction n°1 de CPUID.

```
 mov eax, 1
 cmp eax, [nbFonctions]
 jae .fin ; La fonction n'existe pas
 cpuid
```

On fera ensuite des déplacements de bits couplés à des ET logiques afin d'isoler les nombres qui nous intéressent. Dans le code de démonstration, vous noterez que je n'ai pas tout fait, loin s'en faut. CPUID a suivi l'histoire des compatibles PC depuis leurs 32 bits et en est la digne héritière : c'est fouillis au possible, et c'est la rencontre entre la technique et la réclame (le marketing pour les franglais). On notera qu'AMD, plus jeune qu'Intel en tant que concepteur de micro-processeurs compatibles PC, est sensiblement plus propre dans ses dénominations.

III.4 - Date

Je vous propose maintenant de voir un peu ce que les stockages de masse ont à nous dire. Ce qui n'a rien à voir avec le titre du chapitre. Et en fait, si. Camarade, accroche-toi au pinceau, j'ôte l'échelle. Nous allons tenter de communiquer avec le lecteur de disquettes. Exercice de longue haleine pour nous, mais commençons petitement. Petitement signifie : jouons un peu avec quelque chose de facile à voir qui dispose d'un point commun avec le sujet qui nous intéresse. Le point commun en question, c'est les ports d'E/S. On communique logiquement avec nos camarades disquettes par les ports d'entrée-sortie. Quelque chose dans le même genre est plus facile à appréhender du premier coup, il s'agit du CMOS. CMOS comme Complementary Metal Oxide Semiconductor, soit en français : transistors complémentaires à effet de champ à grille isolée. Ce qui est incompréhensible. On l'appelle comme ça pour des raisons qui m'échappent mais qui sont d'une pertinence confondante, pour peu qu'on soit anglo-saxon et timbré.

Je pourrai évoquer la physique des semi-conducteurs et des batteries, les casse-têtes d'ingénieurs pour fabriquer des mémoires rémanentes, l'histoire moderne des prospecteurs miniers à la recherche des filons purs, nouveaux chercheurs d'or du Klondike (oui, j'aime beaucoup Picsou) pour vous expliquer l'origine de ce terme et ce qu'il désigne. Mais je m'abstiendrai

Parce qu'entre nous, et n'en dites rien, profitez d'être dans la confidence des dieux, il s'agit d'une montre.

Oui, il y a trois bricoles en plus, mais franchement.

Entre nous.

C'est une montre.

Et on lit cette montre par les ports d'E/S.

Cette montre écoute le port 70h et parle sur le 71h. En gros, on demande une case mémoire sur le 70h et on récupère sa valeur sur le 71h.

Alors, ces cases mémoires, de quel choix mirifique disposons-nous ? J'ai trouvé **ceci**, qui a le bon goût de correspondre assez bien à ma réalité. Si, vous et moi, ne vivons pas dans le même multivers et que votre réalité ne correspond pas à la mienne, je peux faire un effort et faire le correctif.

Numéro	Fonction
0	RTC seconds. Contient les secondes de l'heure courante.
2	RTC minutes. Contient les minutes de l'heure courante.
4	RTC hours. Contient les heures de l'heure courante.
6	RTC day of week. Contient le jour de la semaine courant
7	RTC date day. Contient le numéro du jour courant.
8	RTC date month. Contient le mois courant.
9	RTC date year. Contient l'année courante (siècles exclus).
50	RTC date century. Contient le siècle courant.

Je n'ai écrit ici que les valeurs qui nous préoccupent au premier chef. Je puis donc affirmer qu'au moins une machine sur cette terre utilise ces valeurs. Elles sont codées en "BCD" (Binaire Codé Décimal) ou "DBC" (Décimal Codé Binaire) ou qu'importe le sigle pourvu qu'on ait la valeur. Le BCD est l'art de perdre de la place en mémoire : **Wikipédia**, qui est notre ami, explique assez bien de quoi il retourne. Nous, tout ce qu'on a besoin de savoir, c'est que les octets du tableau ci-dessus doivent être passés à cette fonction de conversion :

```
DCBVersBinaire:
 push bx
 push ax   ;On sauve le Décimal Codé Binaire
 and al, 0xF0 ;On récupère les dizaines
 push ax   ;On sauve les dizaines
 shr al, 1  ;Qu'on divise par 2 (ie on multiplie le _chiffre_ des dizaines par 8)
 pop bx
 shr bl, 3  ;Qu'on divise par 8 (ie on multiplie le _chiffre_ des dizaines par 2)
 add al, bl  ;On a donc le nombre des dizaines (8+2)*chiffre des dizaines
 pop bx
 and bl, 0xF  ;On récupère les unités
 add al, bl  ;Et voici la somme des 2, ie notre nombre
 pop bx
 ret
```

Allez, c'est parti, on tranforme notre bête de calcul, notre 64 bits Quad Core Tri Channels Moto Krote en montre à quartz. On commence par charger les chaînes de caractères intéressantes, comme, au pif, les noms des jours et des mois.

```
 mov eax, chMois
 mov edi, indexChMois
 mov ecx, 12
 call chargeListe
 mov eax, chJours
 mov edi, indexChJours
 mov ecx, 7
 call chargeListe
```

Et ensuite, on demande gentiment au CMOS, qui s'appelle aussi RTC (Real Time Clock, ou Regarde Ton Cadran - horloge temps réel), la date. On le fait dans une boucle parce que je n'aime pas le code dupliqué :

```
 mov edi, Annee
 xor ah, ah
 mov ecx, 4
.boucleDate:
 mov al, cl ; Registres de date du RTC du CMOS
 add al, 5
 out 0x70, al
 in al, 0x71
 call DCBVersBinaire
 stosb
 loop .boucleDate
```

Soyez chics, n'oubliez pas de prévoir de la place dans **Annee**, comme ceci :

```
Annee: db 0
Mois: db 0
JourDuMois: db 0
JourDeLaSemaine: db 0
Siecle : db 0
Heure : db 0
Minute : db 0
Seconde : db 0
```

Ca fait 8 octets. Croyez-moi, vous voulez nommer chaque octet. Bon, on récupère le siècle :

```
 mov al, 50 ; Registre de siècle du RTC du CMOS (peut-être)
 out 0x70, al
 in al, 0x71
 call DCBVersBinaire
 stosb
```

Et enfin l'heure par une boucle de 2 en 2 :

```
 mov ecx, 3
```

```
.boucleHeure:
 mov al, cl ; Registres d'heure du RTC du CMOS
 dec al
 shl al, 1
 out 0x70, al
 in al, 0x71
 call DCBVersBinaire
 stosb
 loop .boucleHeure
```

Reste à faire l'affichage. Et passer un coup de balai, c'est à dire transformer en fonction le code redondant.

III.5 - Du code

L'ensemble du code de ce chapitre se trouve ici : **Tuto15**

IV - Remerciements

Merci à l'équipe de developpez.com, et en particulier à **ClaudeLELOUP** pour sa relecture soigneuse.