System

- Numero: 2
- Prof: Laskar Gabriel
- Date: 17 Octobre 2017

Cours d'assembleur

Ceci est un cours d'assembleur. On va donc parler:

- *x86-64*
- ELF
- Memoire
- assembler (syntaxe)

Il n'y a pas de livre pour apprendre ce cours. Sur le net on trouve bcp de syntaxe *Intel* et 32bits. On va utiliser la syntaxe *gcc*.

En ressource utile on a *Software Developer Manual* (4000 pages)

1. User (Chap 3 et 5)
2. Ref
3. System (on s'en fou)

Il faut le lire. Commencer par le chapitre 3 et 5.

On a ausse le manuel de l'assembleur. *GNU AS* (gas manual). `info as`

Registre

- RIP (Register **Instruction Pointer**)
- RSP, RBP (Register **Stack Pointer**, Un truc louche)
- RDI, RSI (on en fait un peu se qu'on veut)
- RAX, RBX, RCX, RDX (on en fait un peu se qu'on veut)
- r8 -> r15 (on en fait un peu se qu'on veut)
- eflags (Registre de conditions)
  - Flag `Z C O C`

Ces registres la sont accessible de plusieurs manieres:

```
 ----------------------------
|          RAX 64b           | (Version 64)
 ----------------------------
 -------------
|    EAX      | (Version 32)
 -------------
 ------
|  AX  | (Version 16)
 ------
```

Ceci fonctionne uniquement pour les registres de A a D.

Ecrivons du code !

```
if (a != 0)
  return 1;
else
  return 0;
```

L'instruction pour retourner une valeur en assembleur c'est `ret`. On return le registre `rax`. C'est historique puis il faut en choisir un...

```
  cmp   %rax, $0 (On suppose que a est dans rax)
  je    toto (Jump vers toto si cmp est vrai)
  mov   $1, %rax
toto:
  ret (On renvoi rax)
```

On a des instructions de taille variable (Car on a du 8, 16, 32, 64 bits) C'est chiant mais ca peut etre une bonne idee car on a les petites instructions qu'on utilise souvent et les grosses moins.

Du coup on peut gagner en taille de code. Du code plus dense = plus vite. se qui coute chere ce n'est pas executer l'instruction mais c'est aller la chercher.

- Vitesse d'un proc: L'ordre du Ghz
- Vitesse d'un bus: 1000 fois plus lent.

En 64bits les operations 32bits sont signed extend. Chez Intel `1 word = 16 bits`

Un label est un moyen de donner un nom a une adresse. On peut donc jump dessus.

ELF

C'est un format d'executable. C'est un moyen de stocker des donnees de manieres organise. On peut faire des bibliotheques dynamiques.

- Executable
- Shared Object
  - .so and others
- Relocatable Object
- Coredump

Static library ne sont pas des *ELF*.

- cpp, Preprocesseur
- cc
- as, Assembleur va donner les .o
- ld, Linker de tous les .o

C'est une compilation incremental. Ca permet de ne pas tout recompiler a chaque modif. Mais dans un `.o` on a pas les adresses. On a des refs vers des fonctions externes. C'est le link qui va regler ces problemes pour completer les trous.

Un `ELF` a un header contenant des infos plus ou moins utile, ex:

- Le type: `e_type`
- Le debut du programme: `e_entry`
- Des tags:
  - Tags de sections, permet de manipuler en static l'ELF.
    * Il va contenir une table de symboles.
  - Tags de segmants, permet de manupuler dynamiquement ELF.

Pour manipuler des `ELF` on peut utiliser:

- `nm` Pour avoir la table des symboles
- `objdump` Pour avoir le code assembleur
- `readelf` Pour voir la table de segments
- `gdb`
- `objcopy` Assez cool d'apres multun
- `ld` / `gas`

Tous ces outils sont des `bin utils`.

Toolchain:

- binutils

- gcc
- libc:
  - headir
  - libc.so
  - ld.so