TLS-SEC **2019-2020**
Semaine d'harmonisation

Assembleur x86 - Harmonisation

4 octobre 2024

Ce TP associé au cours d'harmonisation d'assembleur dispensé au sein de la formation TLS-SEC. Les objectifs des manipulations déroulées sont : savoir développer un algorithme simple en assembleur `x86` ; utiliser les instructions système du processeur à l'aide l'assembleur en ligne C ; optimisations à l'aide d'instructions spécifiques et d'assembleur en ligne ; interfacer du code assembleur en ligne avec l'API et l'ABI de fonctions écrites en langage C ; interfacer de l'assembleur en ligne avec les services du noyau linux.

Structure du TP Ce TP est proposé avec un archive `lst.tgz` qui contient les sources nécessaires à la réalisation des exercices. Téléchargez ces sources et décompressez-les :

```
$ cd <mon-dossier-de-travail-prefere>
$ wget -c https://flash.enseeiht.fr/bemorgan/tp-assembleur/lst.tgz
$ tar xvzf lst.gz
$ ls lst
```

Le dossier `lst` contient un fichier `Makefile` pour la gestion des dépendances ainsi qu'un sous répertoire `src` contenant les sources des différentes questions. Chaque question n porte un nom de dossier de la forme `src/qn`.

On vous conseille de copier le dossier `lst` pour garder sous la main les sources originales et donc les références de ce document vers les lignes de code à modifier !

```
$ cp -vr lst lst_sol
```

1 Ma première fonction assembleur

Afin de mieux appréhender les aspects logiciels bas niveau et système de la formation, il est au minimum nécessaire d'être capable de mettre en œuvre un algorithme simple.

Dans notre cas, nous allons étudier la mathématique factorielle :

$$x\,! \Leftrightarrow \prod_{i=1}^{x} i | x \in \mathbb{N}$$

Voici pour exemple une implémentation impérative et itérative en langage C classique (`src/q0/q0.c`) :

```
unsigned int fac(unsigned int x) {
  unsigned int r = 1;
  unsigned int i;
  for (i = 2; i <= x; i++) {
    r *= i;
  }
  return r;
}
```

Nous proposons une fonction équivalente, `int (%rax) fac_asm(int (%rax) x)`, en assembleur `x86` telle que : l'argument de la fonction et la valeur retournée sont passés à l'aide du registre `%rax`.

Le dossier `src/q1` propose le squelette nécessaire à la réalisation de cet exercice. Il contient un fichier `src/bs.c` qui va appeler la fonction `fac_asm` en lui passant le bon paramètre et en récupérant sa valeur de retour. Le fichier `sec/q1.s` contient un squelette de définition de la fonction `fac_asm` que vous devez compléter.

Développez le corps de la fonction `q1` et constatez son fonctionnement.

☞ Dans le contexte de la fonction `fac_asm`, vous ne disposez pas du moyen de débogage simples sur la sortie standard (`printf()`). Usez et abusez donc du débogeur GDB (commande `gdb <exécutable>`) pour suivre l'exécution de votre programme pas à pas.

2 Utilisation d'instruction spécifiques au matériel

Certaines fonctionnalité du processeur, c'est à dire certaines instructions, ne sont pas directement utilisables à l'aide d'un langage système comme le langage C. En effet, c'est le cas de l'instruction `cpuid` qui permet d'identifier précisément le modèle de processeur qui est en train d'exécuter notre programme. Un autre exemple est aussi l'instruction `rdtsc`, qui permet de consulter un registre spécial donnant un *timestamp* précis. Cette deuxième instruction peut être utile dans le cas des systèmes temps réels.

L'instruction `cpuid` L'interface de programmation de cette instruction est la suivante :

		eax	ebx	ecx	edx
Interface générique	Entrées	fonction appelée			
	Sorties	sortie 1	sortie 2	sortie 3	sortie 4
Constructeur	Entrées	`0x0`			
	Sorties	max. `cpuid` value	"Genu"	"ntel"	"ineI"
Féquence (MHz)	Entrées	`0x16`			
	Sorties	base	max	bus	

> ☞ Le contenu des chaînes de caractères placées dans `e[bcd]x` n'est pas terminé par un caractère de fin de chaîne. En effet, la taille de registres est de 4 octets.

Modifiez le fichier `src/q2/q2.c` de façon à afficher le nom du constructeur de votre processeur ainsi que ses fréquences de fonctionnement. Vous devrez utiliser de l'assembleur en ligne.

Il est parfois difficile de déboguer le code assembleur en ligne. Il est possible d'extraire la sortie assembleur de `gcc` à l'aide de l'option `-S` de gcc, Ainsi il est possible d'accéder à l'assembleur en généré par la primitive d'assembleur en ligne.

```
$ gcc -S src/q2/q2.c -o /tmp/q2.s
# Rechercher le bloc d'assembleur généré
$ grep "#APP" -A9 -B5 /tmp/q2.s
# Ou en une seule ligne
$ gcc -S src/q2/q2.c -o /dev/stdout | grep "#APP" -A9 -B5
```

3 Copies mémoire optimisées

L'architecture `x86` dispose d'un jeu d'instruction *CISC* à taille variable et qui supporte les préfixes d'instruction. Ces préfixes influent sur l'exécution des instructions.

Un exemple de préfixe utile pour des optimisations de copies mémoire est le préfixe *Repeat String Operation Prefix*, ou préfixe de répétition pour les chaînes de caractères. Ce préfixe est noté `rep` et doit être placé avant l'instruction sur laquelle on veut avoir un effet de répétition. Par exemple, `rep lods %al` va répéter `cx` fois l'instruction : `al` = `MEM[si]`.

En plus de la répétition et à chaque itération, les registres d'index `si` et `di` sont incrémenté de la taille de mot considéré par l'instruction : 1 si un octet ; 2 si un mot *(word)*, etc.

Le préfixe `rep` est donc très intéressant pour les copie mémoires octet par octet continues, notamment en utilisation conjointe avec l'instruction `movsb`. En effet, `rep movsb` s'exécutera alors de la façon suivante : `MEM[di]` = `MEM[si]`, `cx` fois tel que `si` et `di` incrémentés à chaque itération.

Modifiez le fichier `src/q3/q3.c` pour développer la fonction `memcpy_asm()`, clone de la fonction de la bibliothèque standard C `memcpy` (`$ man 3 memcpy`). Cette fonction devra utiliser de l'assembleur en ligne et l'instruction préfixée `rep movsb`. Vous remplacerez l'appel à la fonction `memcpy` et constaterez un fonctionnement identique.

Fin du tour de chauffe Le but des exercices suivants et d'appréhender la différence entre les notions d'API, d'ABI et de produire de l'assembleur en ligne capable de s'interfacer directement avec le système (noyau) ou une autre fonction d'un programme C.

4 Notions d'API et d'ABI

L'API *(Application Programming Interface)* ou interface de programmation d'une fonction constitue le type d'une fonction. Le type d'une fonction est donné par le type de ses arguments et de sa valeur de retour, ainsi que leur quantité. En ce qui concerne les langages systèmes compilés comme le C, l'API d'une fonction est décrite par son prototype. L'interface d'API entre une fonction appelée et une fonction appelante est contrôlée par l'utilisateur du langage et sa cohérence est vérifiée lors de l'application de la sémantique des appels de fonctions. Le type de la fonction appelée est à ce moment là vérifié. Exemple d'API : `ssize_t strlen(const char *str);`

L'ABI *(Application Binary Interface*) ou interface de programmation binaire d'une fonction constitue la technique concrète que vont utiliser une fonction appelante et appelée pour communiquer des arguments et retourner une valeur de retour. C'est une notion plus bas niveau que la précédente. En effet, elle est normalement entièrement traitée par le compilateur et sa mise en œuvre doit être totalement transparente pour l'utilisateur d'un langage de programmation. Historiquement l'ABI des systèmes GNU / linux utilisait la pile pour renseigner les arguments et une valeur de retour. Cette ABI est assez lente car elle utilise la mémoire centrale et va ralentir considérablement l'appel d'une fonction. C'est pour ça que les ABI de type *fast call* on été proposées. Au lieu de communiquer les arguments sur la pile, le compilateur va utiliser les registres généraux. Le nombre de registres généraux et leur taille limitée vont conditionner l'utilisation de cette technique. Lorsque la place manquera ou que le nombre d'arguments sera trop important, le compilateur utilisera à nouveau la pile.

Les langages d'assemblage sont par construction très proches du matériel. Ils ne disposent pas des facilités d'appel de fonction que proposent les langages systèmes comme le C. Les programmes assembleur doivent par conséquent impérativement définir l'ABI des fonctions qu'il vont appeler. Enfin, un programme assembleur peut tout à fait appeler une fonction d'un autre langage, à condition de respecter l'ABI de destination.

5 S'interfacer avec un appel système linux

Cet exercice propose le développement d'un appel système en assembleur en ligne. Les programmes écrits en langage C n'ont pas habituellement à s'interfacer directement avec le noyau pour exécuter des appels systèmes. Ils s'appuient en général sur la bibliothèque standard C (`$ man 2 <appel-système>`). Par exemple, la fonction `int open(const char *pathname, int flags);` met en œuvre l'appel système open.

Dans cet exercice, nous allons nous concenter sur l'appel système de recouvrement, c'est à dire celui qui permet le chargement d'un exécutable, puis son exécution, par recouvrement du code du processeur courant. Cet appel système est `execve`. Il existe déjà une fonction de la bibliothèque standard C qui implémente cet appel. Le programme `src/q4/q4.c` utilise cette fonction.

Compilez le programme et constatez son fonctionnement. Observez les arguments de cette fonction et relevez précisément leur type.

Le noyau linux a standardisé l'interface de ses appels systèmes en suivant la norme `POSIX`. Aussi, dans les version modernes de `linux` l'instruction utilisée pour effectuer un appel système est l'instruction `syscall`. Cette fonction invoque explicitement le noyau pour exécuter un appel système.

1. L'instruction `syscall` se contente brancher dans le noyau avec un niveau de privilège plus élevé. Elle ne spécifie par l'interface bas niveau (ABI) à utiliser pour spécifier les arguments de l'appel.
2. Le type d'appel système à exécuter est donné par son numéro qui est aussi passé en paramètre.

Consultez le fichier du noyau linux `arch/x86/entry/entry_64.S` et plus particulièrement l'étiquette `entry_SYSCALL_64` afin de connaître l'ABI du noyau pour l'instruction `syscall`. Consultez soit le fichier du noyau linux `entry/syscalls/syscall_64.tbl`, ou le fichier de la libc `/usr/include/asm/unistd_64.h` pour déterminer le numéro de l'appel système `execve()`. Confirmez aussi la version de l'ABI.

Vous pouvez trouver le code source indexé de linux à l'adresse suivante :
`https://elixir.bootlin.com/linux/latest/source`

Une fois que vous avez bien déterminé le type des arguments, leurs valeurs et l'ABI à utiliser. Remplacez la ligne `11` du fichier `src/q4/q4.c` par un bloc d'assembleur en ligne équivalent. Vous pouvez utiliser autant de contraintes d'entrées que nécessaire, mais vous ne devez utiliser qu'une seule ligne d'assembleur ! !

6 S'interfacer avec une fonction C

L'ABI utilisée actuellement par les applications C `64-bit` compilées avec GCC sous GNU / linux est l'ABI *System V ABI* documentée ici :
`https://wiki.osdev.org/System_V_ABI#x86-64`.

Le listing C `src/q5/q5.c` définit une fonction `strlen_dumb` que la fonction main appelle afin de calculer la taille d'une chaîne de caractères.

Documentez-vous sur l'ABI *System V ABI*. Déterminez ensuite précisément l'interface de la fonction `strlen_dumb`, c'est à dire quels sont les registres où placer les argument de la fonction, puis où lire sa valeur de retour. Enfin, produisez un bloc d'assembleur en ligne équivalent à la ligne `38` pour appeler correctement la fonction `strlen_dump`.

Dans un deuxième temps, l'utilisateur déclare et définit une structure à seize éléments de 4 octets, `struct dumber`, de taille donc conséquente. La structure est ensuite passée en argument de la fonction d'affichage `display_dumber`. La structure est trop grosse pour être passée en paramètre par registre. Le compilateur aura normalement choisi de copier la structure sur la pile.

Vérifiez ce comportement à l'aide du débogeur `gdb`. Vous pouvez par exemple utiliser les commandes suivantes :

```
$ gdb out/q5/q5
(gdb) b 49
(gdb) run
(gdb) disass
```

Observez bien la gestion de la pile et la quantité mémoire précédemment allouée qui est affectée.

Nous vous rappelons que la pile d'un processeur `x86` "descend". Aussi, nous vous rappelons que les éléments d'une structure sont alloués aux adresses basses pour les éléments déclarées en premier.

Produisez un bloc d'assembleur en ligne qui prend en entrée une référence vers la structure mémoire `dumber`, alloue une copie dans la pile (décrementation d'`esp` puis copie) et appelle correctement la fonction `display_dumber`. Ce bloc d'assembleur doit remplacer la ligne `49` : il est interdit d'allouer la mémoire à l'aide du compilateur :)

☞ Utilisez la contrainte de référence mémoire `"m"` pour récupérer l'adresse de la structure, source de la copie sur la pile. N'oubliez pas qu'un pointeur peut être copié dans un registre à l'aide de l'instruction `lea`. Utilisez la contrainte registre `"r"` pour obtenir la taille de la structure à copier. Pensez aussi à `rep movsb` pour copier la structure sur la pile !