Architecture des systèmes

Chapitre 2 : Assembleur x86

Objectifs du chapitre

- Assembleur x86
  - fonctionnement d'un microprocesseur
    - réalisation de calculs
    - accès à la mémoire
  - programmation d'un microprocesseur
    - langage : assembleur x86
  - intérêt de l'assembleur
    - relation avec les langages de programmation évolués

Objectifs du chapitre

- Pourquoi connaître l'assembleur ?
  - comprendre certains fonctionnements avancés des autres langages (compilation, déboguage, pointeurs, etc.)
  - langage de génération 1
- Pourquoi programmer en assembleur ?
  - optimiser certaines parties critiques (systèmes d'exploitation, primitives graphiques)
  - écrire des modules bas niveau (gestionnaires de périphériques)

Plan

- 1. Programmation assembleur x86
- 2. Registres et mémoire
- 3. Instructions assembleur
- 4. Lien avec les langages évolués

1. Programmation assembleur x86

1. Programmation assembleur x86

- Assembleur
  - langage de génération 1
- Avantage
  - exécution très rapide
- Inconvénients
  - très bas niveau (très peu *user-friendly*)
  - pas portable
  - beaucoup de contraintes de programmation (langage très technique)

1. Programmation assembleur x86

- Un assembleur par architecture de processeur
  - famille x86
    - famille de processeurs incluant Intel Pentium, Intel i7, AMD Phenom, AMD Athlon, etc.
    - processeur 8086 : 1978
  - manipule 8 bits, 16 bits ou 32 bits
  - syntaxe Intel (et non pas syntaxe AT&T)
  - environnement : système d'exploitation Linux, logiciel nasm

1. Programmation assembleur x86

- Programme Hello world

```
section .data

msg db 'Hello world!',10
len equ $-msg

section .text

global _start
_start:
  mov eax, 4
  mov ebx, 1
  mov ecx, msg
  mov edx, len
  int 80h
  mov eax, 1
  mov ebx, 0
  int 80h
```

```
all: hello-world

hello-world.o: hello-world.asm
	nasm -f elf hello-world.asm

hello-world: hello-world.o
	ld -m elf_i386 -o hello-world hello-world.o
```

```
alex$ vim hello-world.asm
alex$ make
nasm -f elf hello-world.asm
ld -m elf_i386 -o hello-world hello-world.o
alex$ ./hello-world
Hello world!
alex$
```

1. Programmation assembleur x86

- Désassemblage
  - objdump -d hello-world -M intel
- Déboguage
  - gdb hello-world

2. Registres et mémoire

2. Registres et mémoire

- Registres
  - variables internes au processeur
  - accès très rapide, mais faible nombre de registres
  - les registres sont spécialisés
- Registres de données
  - registres 32 bits : EAX, EBX, ECX, EDX
  - très utilisés (manipulation des données, appel de fonctions, etc.)
  - chacun comprend un registre 16 bits (AX, BX, CX, DX)
  - chaque registre 16 bits comprend deux registres 8 bits (AH, AL, BH, BL, CH, CL, DH, DL)

2. Registres et mémoire

- Mémoire : décomposée en segments (blocs) et en offsets (index)
  - variables globales, variables locales, allocation dynamique
  - pile d'appels de fonction (schéma)
- Registres de segments
  - code : CS
  - données : DS
  - supplémentaires : ES, FS, GS
  - pile : SS

2. Registres et mémoire

- Registre d'index
  - code : IP
  - données : DI, SI
  - pile : SP, BP
- Registre des indicateurs
  - indicateur de zéro : ZF
  - indicateur de signe : SF
  - indicateur d'interruptions : IF
  - etc.

2. Registres et mémoire

- Adressage
  - adressage direct : le contenu de la case 0x100 se dit [0x100] (ou DS:[0x100])
  - adressage indirect : le contenu de la case dont l'adresse est la valeur de DI (ou de SI) : [DI] (très utilisé)
  - adressage basé : le contenu de la case dont l'adresse est la valeur de BX : [BX] (ou avec BP, mais utilise le segment SS dans ce cas) (peu utilisé)
  - adressage indirect basé avec déplacement : [DI+BX+4] (très peu utilisé)

3. Instructions assembleur

3. Instructions assembleur

- Instructions : mnémoniques + opérandes
  - affectation, opérations logiques, opérations arithmétiques
  - comparaisons, branchements, boucles
  - fonctions et interruptions

3. Instructions assembleur

- Affectation
  - MOV destination, valeur
- Remarques
  - “valeur” peut être une constante, un registre, une zone mémoire
  - une zone mémoire peut être désignée par : [constante], [registre], [registre+constante], [registre+BP+constante]
  - on peut préciser le segment en utilisant segment: [offset], ou [segment:offset]

3. Instructions assembleur

- Contraintes
  - on ne peut pas désigner deux zones mémoire dans un MOV, il faut passer par un registre intermédiaire
  - on ne peut pas modifier avec un MOV certains registres (IP, registres de segments)
  - implicitement, DS est utilisé, sauf avec BP

3. Instructions assembleur

- Instructions logiques
  - AND destination, valeur
  - OR destination, valeur
  - NOT destination
  - XOR destination, valeur

3. Instructions assembleur

- Instructions arithmétiques
  - ADD destination, valeur
  - SUB destination, valeur
  - IMUL registre1, opérande2 : multiplie registre1 et opérande2, et stocke le résultat dans registre1
  - IDIV opérande : divise le contenu de EDX:EAX par l'opérande, place le quotient dans EAX et le reste dans EDX

3. Instructions assembleur

- Comparaisons
  - CMP a, b
- Branchements
  - inconditionnels : JMP destination
  - conditionnels : JE dest, JNE dest, JG dest, JGE dest, JL dest, JLE dest,
- Remarques
  - JMP segment:[offset]

3. Instructions assembleur

- Boucle
  - LOOP label : décrémente CX et effectue un saut au label désigné

3. Instructions assembleur

- Fonctions
  - pour appeler la fonction : call label
  - pour sortir de la fonction : ret
- Pile
  - utilisée pour les arguments des fonctions, et pour sauvegarder le contexte lors d'interruptions
  - utilise SS:ESP
  - empilement : push valeur (ESP diminue)
  - dépilement : pop valeur (ESP augmente)

3. Instructions assembleur

- Interruptions
  - fonctions du système
  - pour appeler une interruption : int numéro
  - pour sortir d'une interruption : iret
- Quelques interruptions
  - 0x0 à 0x7 : processeur
  - 0x8 à 0xf : périphériques
  - 0x10 : vidéo
  - 0x13 : accès aux disques

3. Instructions assembleur

- Quelques interruptions (suite)
  - 0x16 : clavier
  - 0x1C : horloge
  - 0x20 : DOS : terminer un programme
  - 0x21 : DOS : API
  - 0x28 : DOS : boucle d'attente du shell

3. Instructions assembleur

- Interruption 80h : Unix : API
  - utilisée pour tous les appels systèmes Unix
- Appels systèmes sous Linux
  - unistd.h contient les numéros de chaque appel système
  - le numéro de l'appel système est dans EAX
  - les paramètres sont passés *via* EBX, ECX, EDX, ESI, EDI, EBP (dans l'ordre)

3. Instructions assembleur

- Exemple 1 : afficher un message
  - appel système write : ssize_t write(int fd, const void *buf, size_t count);
  - l'appel système est le 4
  - MOV EAX, 4
  - MOV EBX, 1
  - MOV ECX, message
  - MOV EDX, longueur
  - INT 80h

3. Instructions assembleur

- Exemple 2 : quitter
  - appel système exit : void _exit(int status);
  - l'appel système est le 1
  - MOV EAX, 1
  - INT 80h

3. Instructions assembleur

- Pseudo-instructions
  - utilisées par le compilateur
  - label : noms suivis du symbole ':'
  - déclaration d'une variable : pour des octets : nom db valeur (dw pour des mots, dd pour des doubles mots), ? pour des valeurs non initialisées
  - déclaration d'un tableau : db 10 dup(0), déclare 10 octets initialisés à 0
  - adresse d'une variable : variable
  - contenu d'une variable : [variable], éventuellement avec byte ou word (pour spécifier le type)

4. Lien avec les langages évolués

4. Lien avec les langages évolués

- Programmer en assembleur / C
  - méthode 1 : programmer en assembleur complètement, avec accès à la libc (ou à toute autre bibliothèque)
  - méthode 2 : programmer en C avec des fonctions en assembleur
  - méthode 3 : programmer en C, obtenir le code assembleur, et l'optimiser

4. Lien avec les langages évolués

- Méthode 1 : accès à la libc en assembleur
  - utilisation de “extern”
  - utilisation de “main” au lieu de “_start” (car “_start” est redéfini par le linker)
  - mais, la fonction doit respecter l'interface imposée (pour ne pas avoir d'effet de bord)

```
extern puts, exit

section .data

msg db 'Hello world!', 10
len equ $-msg

section .text

global main
main:
        ; (manque préambule)
        mov edi, msg ;ou push word msg
        call puts
        ; (manque postambule)
        mov eax, 0
        ret
```

4. Lien avec les langages évolués

- Méthode 1 (suite)
  - il faut sauvegarder le cadre de pile en début de fonction, et le restaurer en fin de fonction
  - Remarque : pour la version avec push, nécessite que la libc soit compilée en 32 bits

```
push ebp
mov ebp, esp
push ebx
push edi
push esi


# prendre soin de
# remettre esp à sa
# valeur initiale
# auparavant
pop esi
pop edi
pop ebx
mov esp, ebp
pop ebp
```

4. Lien avec les langages évolués

- Méthode 1 (suite)
  - Compilation avec gcc : “nasm -f elf64 methode1.asm” puis “gcc -m elf_x86_64 -o methode1 methode1.o”

```
push ebp
mov ebp, esp
push ebx
push edi
push esi


# prendre soin de
# remettre esp à sa
# valeur initiale
# auparavant
pop esi
pop edi
pop ebx
mov esp, ebp
pop ebp
```

4. Lien avec les langages évolués

- Méthode 2 : programmer en C avec des fonctions assembleur
  - utiliser “asm(...)”
  - compiler avec “gcc -masm=intel -o methode2 methode2.c”

```
#include <stdio.h>

int main() {
	int value = 100;
	asm(
		"mov eax, %0;"
		"inc eax;"
		"mov %0, eax;"
		:"=r"(value):"r"(value)
	);
	printf("Hello world\n”);
	printf(“value is %d (should not be ”, value);
	printf(“equal to 100 anymore)\n");
	return 0;
}
```

4. Lien avec les langages évolués

- Méthode 2 (suite)
  - Pour accéder aux variables en écriture, on utilise “=r” (pour registre)
  - Pour accéder aux variables en lecture, on utilise “r”
  - Dans la partie assembleur, on utilise %0, %1, etc.

```
#include <stdio.h>

int main() {
        int value = 100;
        asm(
                "mov eax, %0;"
                "inc eax;"
                "mov %0, eax;"
                :"=r"(value):"r"(value)
        );
        printf("Hello world\n”);
        printf(“value is %d (should not be ”, value);
        printf(“equal to 100 anymore)\n");
        return 0;
}
```

4. Lien avec les langages évolués

- Méthode 3 : optimiser le code assembleur produit par du C
  - gcc -S -masm=intel methode3.c
  - modifier le fichier methode3.s
  - gcc -o methode3 methode3.s