Conservatoire National des Arts et Métiers
De Montpellier

Le Programmation des Ordinateurs

14 mai 2025

Table des matières

I- Introduction

- Les processeurs des ordinateurs sont programmés en langage machine dit langage de 1ère génération. Le langage machine est codifié en binaire.
- Etant donné que le binaire est indigeste pour l'humain, on rajoute une sur couche symbolique au langage machine, c'est l'assembleur dit langage de 2ème génération : les instructions vont être représentées par des mnémoniques et les nombre s'exprimeront en hexadécimal.
- Etant donné que l'assembleur, reflet du langage machine n'est pas très convivial et que chaque famille de processeurs possède son propre assembleur, il a fallu créer au dessus de l'assembleur, un langage plus puissant, mieux structuré et multiplateforme : un langage évolué dit langage de 3ème génération comme les langages C et Python.
- Un programme est constitué d'un ensemble d'instructions en langage machine, que l'on appelle un "exécutable".
- Les ordinateurs disposent d'un premier programme que l'on appelle Système d'Exploitation qui permet à l'utilisateur de démarrer/arrêter l'ordinateur et de lancer ses programmes.
- En langage compilé comme le langage C, un programme source va être traduit en langage machine et donc transformé en exécutable par le compilateur.
- Dans le cas d'un langage interprété comme Python, c'est l'interpréteur Python lui-même, écrit en langage machine qui sera l'exécutable et qui va à son tour exécuter le programme source. L'ordinateur ne comprends que le langage machine.

L'Asssembleur

- Pour illustrer nos propos, nous allons utiliser un assembleur 32 bits virtuel, proche de l'assembleur Intel :

a) Le processeur dispose de registres internes 32 bits (4 octets) :
IR (instruction register) : registre qui contient l'adresse de l'instruction suivante à exécuter.
IP (instruction pointer) : registre qui contient l'adresse de l'instruction suivante à exécuter.
A, **B**, **C**, **D** : registres de travail
DI (destination index) : registre de travail qui sert d'index dans les tableaux
CS (code segment) : les adresses des instructions vont être relatives à CS
DS (data segment) : les adresses des données vont être relatives à DS
SS (stack segment) : les adresses pile vont être relatives à SS
SP (stack pointer) : registre de pile, il pointe sur le sommet de la pile.
BP (base pointer) : registre de base, nous verrons plus loin que chaque fonction possède une zone de données contextuelles dans la pile, BP contient l'adresse de cette zone de données.
FLAGS : contient des indicateurs d'état et en particulier :
- **ZF** (Zero Flag) indique que le résultat de la dernière opération était 0
- **SF** (Signed Flag) indique que le résultat de la dernière opération était négatif
- **OF** (Overflow Flag) indique que le résultat de la dernière opération était un débordement

b) La mémoire sera découpée en mots de **32 bits**. Cependant les processeurs ne traitent pas les données directement en mémoire, mais ils vont les recopier dans leurs registres internes, cela pour des raisons de performance, l'accès aux registres internes étant nettement plus rapide (près de 10 fois) que celui de l'accès à la mémoire.

c) Certaines instructions du langage machine sont réservées au mode noyau. Le système d'exploitation (en mode noyau) pourra y accéder mais pas les applications (en mode utilisateur).

II- La Programmation

- Les ordinateurs ont été inventés par des mathématiciens. La programmation c'est du calcul mathématique.

II.1- Calculs Mathématiques

- Pour effectuer leurs calculs, les mathématiciens ont besoin des 4 opérations, du débranchement conditionnel et de mémoriser des données. Pour cela l'ordinateur va disposer d'un processeur (les 4 opérations + le débranchement) et de mémoire.

1- Les 4 opérations mathématiques

En langage C :

```
total = prix + taxes;
```

➢ idem pour moins (**-**), multiplier (*****) et diviser (**/**)

En assembleur :

```
MOV A,[345FFC]      ; lit dans le registre A le mot mémoire 345FFC* correspondant à prix
MOV B,[123454]      ; lit dans le registre B le mot mémoire 123454*  correspondant à taxes
ADD A,B             ; ajoute les registres A et B, résultat dans le registre A
MOV [345678],A      ; écrit le registre A dans le mot mémoire 345678*  correspondant à total
```

➢ idem pour moins **SUB**, multiplier **MUL** et diviser **DIV**

En langage machine :

```
A100345FFC          ; A1 est l'instruction* et 00345FFC  l'argument**
8B1D00123454        ; 8B1D est l'instruction et 00123454 l'argument
01D8                ; 01D8 est l'instruction, il n'y a pas d'argument
A300345678          ; A3 est l'instruction et 00345678  l'argument
```

** Les instructions sont sur 1,2,3 ou 4 octets*

*** Les valeurs numériques et les adresses mémoire s'expriment sur 4 octets (8 chiffres hexadécimaux)*

2- Le débranchement conditionnel

En langage C :

```
if (total >= 1000) {reduction = 100;}
else {reduction = 10;}
```

En assembleur :

```
      MOV A,[345678]        ; lit le mot mémoire correspondant à total  dans le registre A
      CMP A,3E8             ; compare A à la valeur directe 1000 (3E8 hexa)
      JL MOINS              ; si A est inférieur* on jump à l'étiquette MOINS
      MOV A, 64             ; met la valeur directe 100 dans A (64 hexa)
      JMP SUITE             ; jump à l'étiquette SUITE
MOINS: MOV A, A             ; met la valeur directe 10 dans A (A hexa)
SUITE: MOV [444444],A       ; on range le résultat à l'adresse mémoire de reduction
```

* Les étiquettes **SUITE**, **MOINS** permettent d'utiliser de façon symbolique des adresses en mémoire.

3- Les lectures/écritures de données en mémoire

En langage C :
écriture mémoire : `total = 1000;` //cela revient à l'affectation d'une variable
lecture mémoire : `valeur = total;` //cela revient à récupérer une variable

En assembleur : Instruction **MOV** déjà vue.

II.2- La montée en puissance des ordinateurs

- A partir du calcul mathématique, un ordinateur va être capable de traiter toutes sortes de données.

  a) Pour traiter du texte on va codifier les caractères (par exemple **A** est égal à **41 hexa** en codification **UTF-8**) et ainsi une phrase sera ramenée à une série de chiffres.

  b) Pour traiter les images, on va attribuer un triplet de valeurs à chacun de ses points : quantité de rouge, quantité de vert, quantité de bleu. Bien entendu plus on saisira de points sur l'image meilleure sera la restitution.

  c) Pour traiter un objet qui varie dans le temps comme le son, on va l'échantillonner, c'est-à-dire que l'on va saisir l'intensité du son à des intervalles de temps réguliers. Bien entendu plus l'intervalle d'échantillonnage sera petit, meilleure sera la restitution.

- A partir du calcul mathématique, un ordinateur va être capable de piloter toutes sortes d'équipements périphériques : disque dur, écran, clavier, imprimante, équipements de télécommunication, machines-outils …

  Pour piloter un périphérique il faut lui transmettre* des ordres et des données numériques et en retour le périphérique retransmettra un compte-rendu d'exécution et des données numériques à l'ordinateur.

  ** Les échanges se feront au travers d'une mémoire commune ordinateur-périphérique que l'on appelle registres de périphériques.*

  Pour l'ordinateur un périphérique est vu comme une extension mémoire.

- Avec l'introduction des disques durs les ordinateurs vont être capables de mémoriser de plus en plus d'informations, de la base de données au Big Data et à l'IA.

II.3- Aboutissement de la programmation

- Il est nécessaire de rajouter quelques fonctionnalités aux instructions de base la programmation, pour rendre celle-ci efficace, dont en particulier :

4- Les boucles itératives

Certains programmes effectuent un très grand nombre de fois des calculs analogues (des milliers, des millions de fois …), il n'est pas envisageable d'écrire des milliers de fois les mêmes instructions, il faut donc disposer d'un mécanisme permettant de factoriser les instructions.

En langage C :

```
for (i = 0 ; i < 1000 ; i++)  {…}
```

En assembleur :

```
        MOV C, 3E7          ; initialise le registre C à 999 (3E7 hexa)
BOUCLE:
        …
        DEC C               ; décrémente le registre C
        JNS BOUCLE          ; on jump à BOUCLE si C est positif ou nul (NS pour flag non signé)
```

Pour ls débranchements conditionnels on dispose d'un grand nombre d'instructions :
- **JZ** : jump si le résultat de l'opération précédente est égal à 0
- **JS** : jump si le résultat de l'opération précédente est égal à 0 est négatif
- **JO** : jump si le résultat de l'opération précédente est un débordement
- **JMP** : jump inconditionnel (**GOTO** en langage C)

Ainsi que ceux relatifs aux comparaisons :
- **JL** : jump si le résultat de la comparaison précédente est <
- **JLE** : jump si le résultat de la comparaison précédente est <=
- **JZ** : jump si le résultat de la comparaison précédente est =
- **JG** : jump si le résultat de de la comparaison précédente est >
- **JGE** : jump si le résultat de de la comparaison précédente est >=

5- Les tableaux

Certains programmes manipulent un très grand nombre de données analogues (des milliers, des millions …), il n'est pas envisageable d'utiliser une adresse mémoire aléatoire pour chacune de ces données, il faut disposer de mécanismes capables d'ordonner, de structurer les données en mémoire (le plus élémentaire étant le tableau de données).

En langage C :

```
int tableau [1000];
for (i = 0 ; i < 1000 ; i++){
    tableau[i] = i;
}
```

En assembleur :

```
      align 4              ;  aligne la mémoire sur 4 octets avant de déclarer le tableau
      tableau RESD 3E8     ; réserve une zone tableau* de 1000 mots en mémoire (3E8 hexa)
      MOV DI, tableau      ; DI est un registre spécial qui sert de pointeur mémoire
      MOV A, 0             ; initialise le registre A à 0
      MOV C, 3E7           ; initialise le registre C à 999 (3E7 hexa)
BOUCLE:
      MOV [DI], A                ; on écrit A au mot mémoire pointé par DI
      ADD DI,4             ;on ajoute 4 à l'adresse tableau (mots de 4 octets)
      INC A                ; incrémenter le registre A
      DEC C                ; décrémenter le registre C
      JNS BOUCLE           ; on jump à BOUCLE si C est positif ou nul (pas de flag SF)
```

** RESD n'est pas une instruction mais une directive d'assemblage destinée à réserver des mots de 32 bits à partir de l'adresse présente, le mnémonique tableau va représenter l'adresse présente.*

6- Appel / Retour de sous-programme
7- Empiler / Dépiler

Pour des raisons de clarté et de maintenance, il n'est pas envisageable de développer un programme complexe (de milliers ou de millions d'instructions) d'un seul tenant, il faut donc disposer d'un mécanisme permettant de découper les programmes en sous-programmes (ou fonctions), chaque sous-programme effectuant un traitement bien spécifique.

La gestion des sous-programmes va rendre nécessaire l'utilisation d'une pile. A chaque appel à un sous-programme donné, le système doit sauvegarder le contexte d'exécution de ce sous-programme et à chaque retour au programme appelant, le système doit libérer l'espace de ce contexte devenu inutile. Pour sauvegarder les contextes d'exécution des sous-programmes on va utiliser une pile dont le principe est "last in, first out".

Le contexte d'exécution d'un sous-programme est constitué principalement de :
- son adresse de retour au programme appelant
- ses arguments
- ses variables locales

En langage C :

```
ma_fonction (arg);                  // j'appelle le sous-programme ma_fonction,
…
void ma_fonction (int arg) {        // sous programme ma_fonction
    …
    return ;
}
```

En assembleur :

```
    MOV A,[444444]   ; lit l'argument cad le mot mémoire d'adresse 444444 dans le registre A
    PUSH A                   ; empile le registre A dans la pile*
    CALL FONCTION    ; CALL va faire un PUSH IP : instruction suivante = adresse retour
                     ; avant de jumper à FONCTION
    ADD SP 4         ; au retour on retire l'argument de la pile
    …
FONCTION:
    PUSH BP          ; sauvegarde le pointeur du contexte de l'appelant
    MOV BP,SP        ; positionne son propre pointeur de contexte,
                     ;      - au dessous de ce pointeur il y aura les arguments
                     ;      - et au dessus la fonction va rajouter ses variables locales
    …
    MOV SP, BP         ; Restaurer SP pour remettre la pile dans son état initial
    POP BP           ; Restaurer le pointeur de contexte de l'appelant
    RET              ; RET va retirer l'adresse retour de la pile (ADD SP 4)
                     ; avant de jumper à cette adresse retour
```

** Attention la pile progresse à l'envers en mémoire de l'adresse la plus haute à l'adresse la plus basse.*

III- Les Systèmes d'Exploitation

- Pour passer de la programmation mon-tâche à un système multitâche il faut disposer de 2 nouveaux mécanismes, les interruptions et la lecture-écriture des registres des périphériques.

III.1- Interruptions

Le mécanisme d'interruption va permettre au système d'exploitation de partager le temps processeur entre les différentes tâches (différents programmes), donnant ainsi une impression de simultanéité.

8- Lancer une interruption, retour de routine d'interruption
9- Masquer les interruptions, démasquer les interruptions

En langage C :

On ne peut pas accéder aux interruptions avec les instructions du langage C. C'est l'appel à une primitive système qui va générer l'interruption n° 80 hexa, dite interruption "logicielle" laquelle va permettre au programme d'activer la primitive dans le noyau (cf. Primitives Système).

Cependant si on utilise le langage C en mode noyau, il est possible d'inclure du code assembleur dans le code C et ainsi de gérer toutes les interruptions.

```
#include <stdio.h>
printf("Hello World\n") ;   // printf va lancer une interruption n° 80
                            // qui va activer la primitive sys_write du système
```

En assembleur :

- Il faut être en mode noyau

```
INT num     ; lancer l'interruption de numéro num*
            ; le fonctionnement est analogue à CALL sauf que
            ; INT va basculer vers la pile système**
            ; INT va également empiler FLAGS et le code segment CS***
IRET        ; le fonctionnement est analogue à RET sauf que
            ; IRET va également dépiler FLAGS et CS
            ; IRET va rebasculer vers la pile de l'application

CLI         ; masquer les interruptions
STI         ; démasquer les interruptions
```

** La routine d'interruption est le sous-programme associé à une interruption. La table des vecteurs d'interruption indexée par le numéro d'interruption, contient les adresses des routines d'interruption.*
*** La routine d'interruption fait partie du noyau Linux, elle utilise donc la pile système et non pas la pile de l'application.*
**** La routine d'interruption ne se trouve pas dans le même segment de code que le programme, c'est pourquoi il va falloir basculer les segments de code.*

Table des vecteurs d'interruption : 256 interruptions

```
…
128        <adresse appel système>   R3
150       <adresse routine>   R0
…
```

III.2- Pilotage des périphériques

Pour piloter les périphériques, on a besoin des 2 fonctionnalités complémentaires suivantes :

10- Lectures et écritures dans les registres des périphériques

En langage C :

On ne peut pas piloter les périphériques avec le langage C, Il faut faire appel à une primitive système (cf. ci-dessous). Cependant si on utilise le langage C en mode noyau, il est possible d'inclure du code assembleur dans le code C et ainsi de piloter les périphériques.

En assembleur :

- Il faut être en mode noyau

```
IN [123456]   ; Lit dans le registre A le mot d'adresse 123456  de la mémoire périphérique
OUT [456789]  ; Ecrit le registre A dans le mot d'adresse 456789  de la mémoire périphérique
```

Les accès aux périphériques peuvent se faire en mode bloquant : on interrompt le traitement durant l'attente de la réponse du périphérique ; ou en mode non-bloquant : on poursuit le traitement et on est alerté par une interruption lorsque le périphérique a répondu.

III.3- Primitives Système

Pour accéder aux ressources de la plateforme matérielle (écran/clavier, disques durs, lignes de télécommunication …) il faut être capable de piloter les périphériques or cela est réservé au système d'exploitation qui s'exécute en mode noyau. Les systèmes d'exploitation vont donc mettre à la disposition des programmeurs d'applications, des Bibliothèque de sous-programmes (Primitives Système, API[1] en anglais) pour accéder à ces ressources.

En particulier au travers de ces primitives système une application va pouvoir utiliser l'écran.

En langage C :

```
#include <stdio.h> ;                  // Bibliothèque d'API
printf("Hello World \n");             // printf va lancer une interruption n° 80
                                      // qui va activer la primitive sys_write du système
```

En assembleur :

```
message db 'Hello, world!', 0xA  ; texte du message
msg_len equ $ - message          ; longueur du message
…
MOV A,4                          ; numéro de l'API sys_write
MOV B,1                          ; numéro du périphérique de sortie, 1 = écran
MOV C, message                   ; pointeur sur le message
MOV D, msg_len                   ; longueur du message
INT 0x80                         ; lance la routine d'interruption
```

** Pour appeler une primitive système, on met les arguments dans les registres et puis on interpelle le système d'exploitation en lançant l'interruption 80 hexa qui est l'interruption dite logicielle, la seule autorisée en mode utilisateur.*

[1] Application Programming Interface