1ère NSI — Thème 7: Architecture Système

Systèmes d'Exploitation & Réseau

Lycée Fustel de Coulanges, Massy

Marc Biver, avril 2024, *v0.1*

Ce document reprend les notions abordées en cours et pratiquées en TP / TD; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Ses deux principales sources d'inspiration sont, d'une part, le cours mis en ligne par Charles Poulmaire et Pascal Remy[1] *et, d'autre part, le manuel de NSI en 1ère édité chez Ellipses*[2]*. Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.*

Document de cours incluant les réponses aux exercices.

1. Accessible ici.
2. Numérique et sciences informatiques, Spécialité NSI 1re: 30 leçons avec exercices corrigés, par Thibaut Balabonski, Sylvain Conchon, Jean-Christophe Filliâtre, et Kim Nguyen.

Table des matières

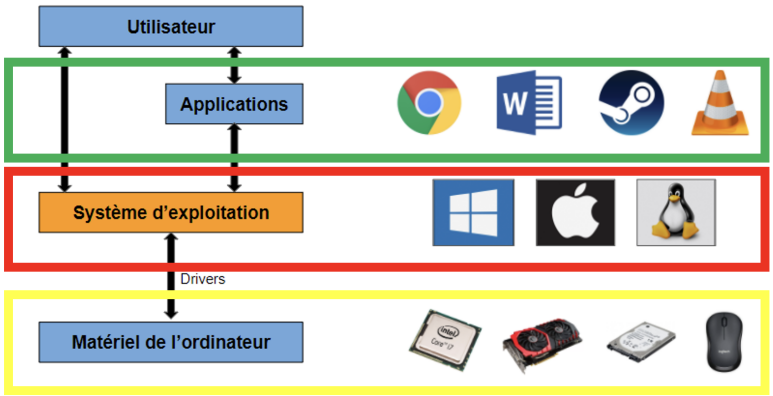
1 Point d'étape – où est-on / où va-t-on ?

1.1 Ce qu'on a couvert jusqu'à présent

- Rudiments de l'architecture physique d'un ordinateur – le modèle de Von Neumann.
- Mise en jambes sur de l'écriture de code : réalisation d'une page Web en HTML.
- Introduction à Python, et plus spécifiquement :
  - Ce qu'on appelle ses "constructions élémentaires" – variables, fonctions, conditions & embranchements, boucles...
  - Les types et valeurs de base : entiers (naturels et relatifs), flottants (réels), chaînes de caractères et booléens (qu'on n'a que brièvement abordés pour l'instant).
  - Un type dit "construit" – les listes.
- Un peu de théorie : la représentation des types et valeurs de base en machine (les entiers naturels et relatifs, les réels, les alphanumériques).
- Un peu plus de théorie : introduction à la logique booléenne (que l'on n'a couverte que très rapidement – on y reviendra en fin d'année).
- Un retour à la pratique : le traitement de données en table, la manipulation de fichiers dans Python, et des types de données plus complexes — les dictionnaires, les listes de dictionnaires...
- Une introduction à l'algorithmique, avec des allers-retours entre théorie et pratique : étude des principes d'algorithmes, rédaction de leur pseudo-code, implémentation en Python.

1.2 Ce dont on va parler dans ce nouveau chapitre

On va revenir ici au tout premier cours qu'on a fait cette année, le modèle de von Neumann, et "remonter" de là :



Dans le schéma ci-dessus on trouve :

Dans le cadre vert : La "couche applicative" — les programmes et applications qu'utilisent les utilisateurs finaux et que, cette année, nous avons commencé à apprendre à développer en Python ;

Dans le cadre rouge : Un des objets de ce cours, la couche intermédiaire située entre le matériel et l'applicatif, l'OS ("Operating System") ou système d'exploitation en français ;

Et enfin dans le cadre jaune : Les composants matériels, physiques de l'ordinateur que l'on a étudiés en début d'année et qu'on représente habituellement dans le modèle de von Neumann – et qui par lesquels nous allons commencer dans ce cours en étudiant les langages machine et assembleur.

Spécifiquement, dans cette thématique du cours, on va parcourir des éléments du schéma précédent et en "sortir" également un petit peu :

- Présentation des niveaux de langage les plus proches de la machine — le langage machine et le langage assembleur ;
- Pratique : visualisation de quelques opérations effectuées en langage assembleur (x86) et simulation quelques programmes simples ;
- Introduction aux systèmes d'exploitation – ce qu'ils sont, à quoi ils servent, pourquoi nos outils informatiques contemporains (ordinateurs, smartphones, tablettes, mais aussi voitures, télévisions, trains...) ne pourraient pas fonctionner sans ;
- Pratique : exercices de lignes de commande en shell (interface système) ;
- Réseaux et internet — introduction au modèle OSI ;
- Pratique : exercices de configuration réseau dans le logiciel de simulation Filius.

1.3 Comment réviser / préparer les contrôles ?

Comme pour les autre chapitres de cours de NSI, les deux choses les plus importantes que vous devez acquérir pour être en mesure de bien réussir les contrôles sont :

- Connaître et comprendre les notions présentées dans le cours (et en particulier les définitions) ;
- Être capable de faire tous les exercices présents dans ce cours et également dans le cahier d'exercices qui l'accompagne.

Comme pour les chapitres précédents, chaque section de ce cours se termine par un encart (sur fond vert) intitulé "À Savoir ⇄ À Réviser" dans lequel je vous explique plus en détails ce qu'il y a à retenir de la section en question.

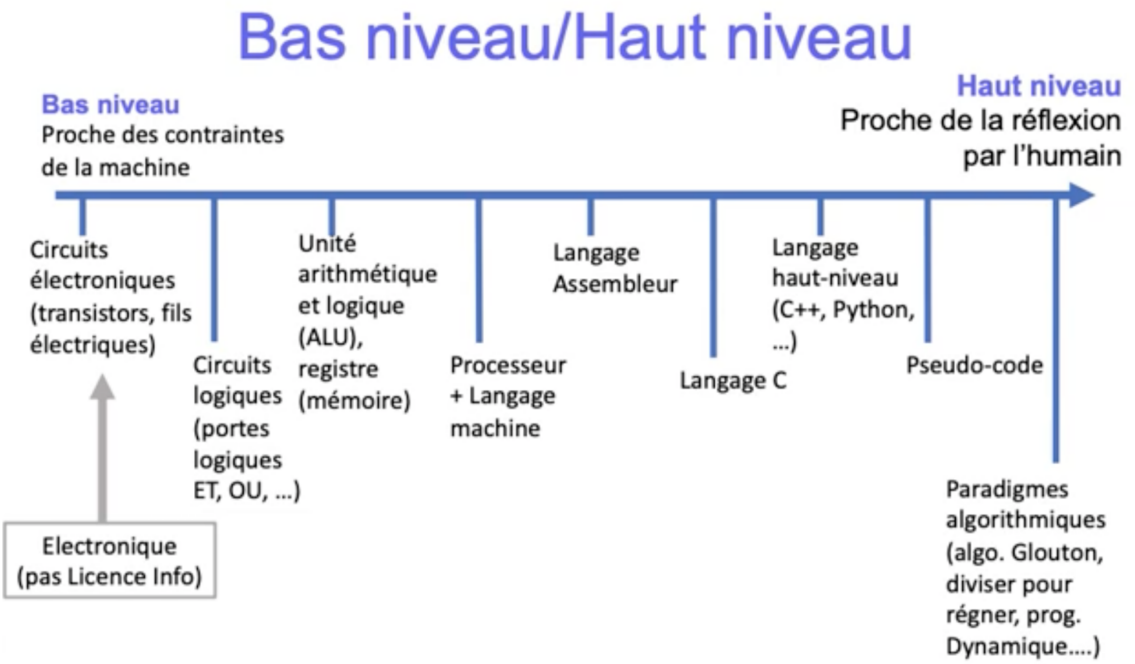
En tout état de cause mon conseil principal pour les révisions est de prendre la version de ce cours et du cahier d'exercices qui ne contient ***pas*** les solutions, vous exercer à faire les exercices qui sont dedans, puis vérifier les réponses dans la version qui contient les solutions.

Bon courage !

2 Les langages de bas niveau

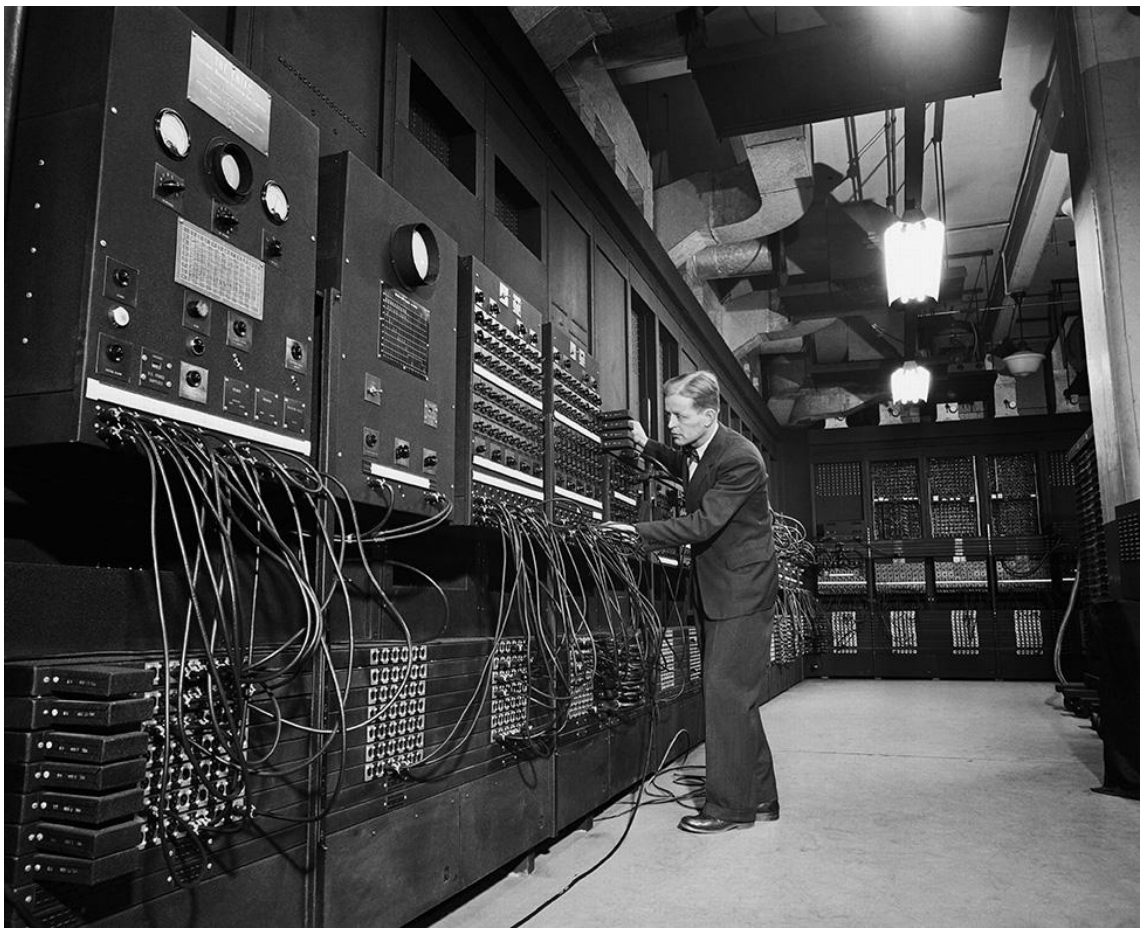
2.1 Un peu d'Histoire...

On a vu en tout début de cours sur la programmation que les langages informatiques se distinguent (entre autres) par ce qu'on appelle leur "niveau" : plus celui-ci est dit "bas" plus on est proche des objets physiques qui constituent la machine, plus au contraire il est "haut" plus on s'en éloigne pour arriver dans des vocables / syntaxes / grammaires compréhensibles par l'être humain — par exemple le langage Python ou, plus élevé encore, le pseudo-code.



Aux origines de l'informatique, les "langages" n'existaient pas (les claviers, d'ailleurs, non plus) : toute la programmation se faisait par la câblage manuel de la machine pour qu'elle réalise une tâche spécifique. Lorsqu'on souhaitait qu'elle en réalise une autre, le câblage devait être modifié, comme dans la photographie ci-dessous [3]. A l'époque, les rôles de concepteur, constructeur, programmeur, opérateur et utilisateur étaient donc confondus — c'était un seul et même ingénieur qui faisait tout (ou, pour se référer au schéma ci-dessus, les tâches correspondant à la totalité des niveaux étaient réalisées par la même personne).

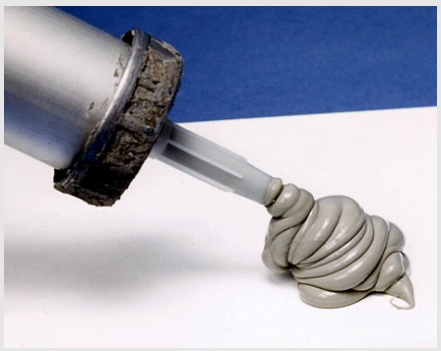
3. Photographie du calculateur ENIAC prise en 1946, obtenue textsur le site du CNRS.



C'est dans les années 1950 et surtout 1960 avec l'utilisation de plus en plus généralisée des transistors que la nécessité de câbler les machines a progressivement disparu, leur miniaturisation a été engagée, et les langages pour communiquer avec les composants micro-électroniques sont nés.

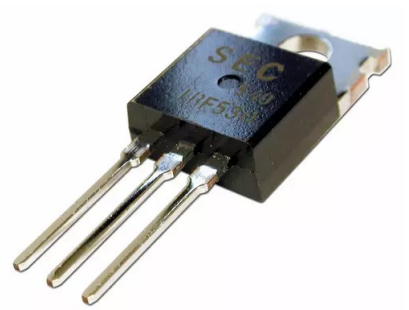
→*Pourquoi la région de Californie où sont localisées la plupart des grandes entreprises de technologie est-elle nommée la Silicon Valley ?*

C'est une bonne question quand on pense que le silicone est une forme de mastic ou de caoutchouc...



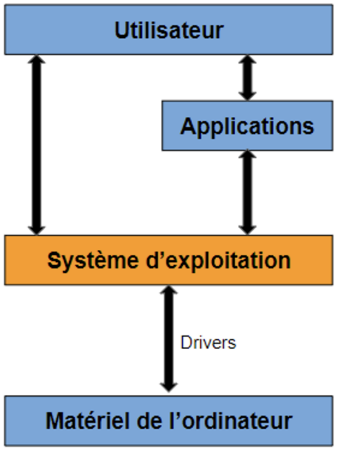
C'est en fait simplement un problème de traduction : l'élément silicium, avec lequel on fabrique beaucoup de semi-conducteurs et notamment les transistors, se dit "*silicon*" en anglais.

Pour faire simple, un transistor est un interrupteur (ayant donc deux positions, correspondant au 0 et au 1) qui est pilotable par un courant électrique : on peut donc connecter électriquement plusieurs transistors ensemble (pour former un circuit), les uns pilotant les autres, ce qui permet c'est de constituer dans un premier temps des portes logiques telles que celles que l'on a vues au chapitre sur les booléens (AND, OR, XOR...). Ces circuits prennent en entrée une ou plusieurs tensions électriques et en font ressortir une en fonction du câblage réalisé — ce qui, logiquement, se traduit par un ou plusieurs bits en entrée et un bit unique en sortie (vous vous souvenez des tables de vérité qu'on avait étudiées ? C'est exactement de cela que l'on parle ici).

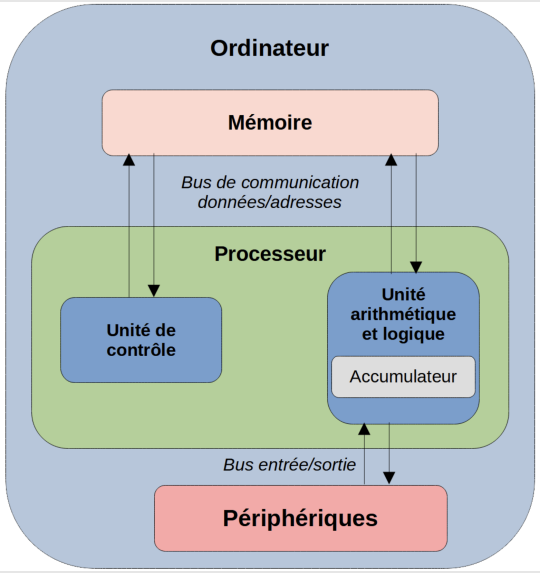


2.2 Quelle langue parle mon ordinateur ?

On a vu plus haut le schéma ci-dessous qui illustre la position des systèmes d'exploitation – entre le matériel et l'applicatif : on va s'intéresser dans ce chapitre à la partie basse de ce schéma.



→*Mais avant de nous lancer dans cette histoire de langage, si on commençait par un petit rappel : quels sont les éléments principaux de l'architecture de von Neumann qu'on a vue en début d'année — indice : il y en a 3... ?*



Ce modèle nous fournit une représentation simplifiée mais exacte de l'architecture matérielle d'un ordinateur (ou, plus généralement, de n'importe quel matériel qui s'appuie sur l'informatique). Les différents éléments qui le constituent ont les rôles suivants :

Unité Arithmétique et Logique : elle effectue les opérations de base ;

Unité de Contrôle : elle assure le séquencement des opérations à effectuer ;

Mémoire : elle contient à la fois les données qui vont être utilisées et les programmes qui s'en chargeront et indiqueront à l'unité de contrôle quelles opérations vont être nécessaires. Elle se divise en deux parties :

a. La mémoire volatile, ou mémoire vive utilisée en cours d'exécution de programmes ;
b. La mémoire permanente qui contient les programmes et données que l'on va conserver sur l'appareil.

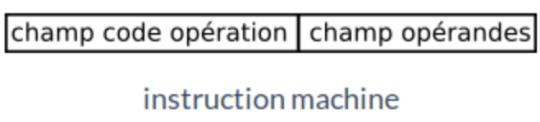
Périphériques : ce sont des dispositifs d'entrée / sortie qui vont permettre de communiquer avec l'extérieur. Il y en a une multitude : clavier, imprimante, carte réseau, micro, enceinte, écran, souris,...

L'enjeu de ce chapitre est de s'intéresser au langage que comprend le cerveau de notre ordinateur, son processeur : les "instructions machine".

Définition: Instruction machine

Une **instruction machine** est une chaîne binaire composée principalement de deux parties :

— Le champ "code opération" qui indique au processeur le type de traitement à réaliser ;
— Le champ "opérandes" indique la nature des données sur lesquelles l'opération désignée par le "code opération" doit être effectuée.



Ces instructions machines sont donc des mots binaires qui, à l'instar des données qu'elles manipulent, sont stockées dans la mémoire de l'ordinateur, dans des cases mémoire possédant chacune une adresse (ces cases, selon l'architecture de la machine, peuvent être de 8, 16, 32, ou 64 bits). On peut se représenter cette mémoire comme une pile de cases contenant chacune un élément qui sera soit une instruction soit une donnée.

On peut imaginer par exemple dans une architecture en 32 bits l'instruction suivante : `11100011 10100000 0000010 00010000` où les deux premiers octets constitueraient le code opération (disons par exemple "copie") puis les deux suivants les adresses mémoires des informations à copier (source / cible).

Un programme informatique deviendrait donc une suite de telles instructions — par exemple, un programme extrêmement simple n'occupant que 9 cases mémoire pourrait s'écrire :

```
11100011 10100000 00000000 00000100
11100101 10001111 00000000 01101100
11100011 10100000 00000000 00001000
11100101 10001111 00000001 00011000
11100101 10011111 00000000 01100000
11100011 01010000 00000000 00001010
00011010 00000000 00000000 00000010
11100011 10100000 00000000 00001001
11100101 10001111 00000001 00000100
```

Alors, me direz-vous, c'est bien gentil tout ça — mais pas franchement clair... C'est ce qui nous amène au langage assembleur.

2.3 Langage assembleur

Pourquoi s'intéresser à cela ?

Il n'est pas question ici de faire de vous des programmeurs en assembleur, ça n'aurait strictement aucun intérêt, même si c'est un métier qui existe et est vital dans bien des domaines — le développement de compilateurs, le développement de pilotes (drivers), la programmation de certaines tâches sur des systèmes embarqués... Mais ce métier est extrêmement spécialisé, et si vous devez y venir ce ne sera que dans plusieurs années.

L'intérêt de cette introduction est plutôt de vous ouvrir une fenêtre sur le fonctionnement interne des machines que l'on manipule depuis le début de l'année, vous faire comprendre ce que, en réalité, elles savent faire, et donc comment les programmes que vous concevez et réalisez en Python sont, dans la pratique, mis en œuvre par votre ordinateur.

Les mnémoniques

Le "code opération" dont on a parlé plus haut est nécessairement quelque chose d'immuable : il a dû être programmé dans le processeur à sa construction. Il y en a en fait relativement peu — il peut s'agir de :

- Opérations arithmétiques : addition, soustraction, multiplication...
- Transferts de données d'une "case" mémoire à une autre.
- Rupture de séquence (si l'on effectue une boucle par exemple, ou si on applique une condition et qu'en conséquence on ne va plus enchaîner les instructions dans l'ordre prévu).
- Opérateurs de comparaison.

Ces opérations se traduisent en langage assembleur en ce que l'on appelle des "mnémoniques" : des codes de trois lettres de long plus ou moins explicite correspondant exactement et de manière unique à une opération élémentaire que sait réaliser le processeur et qui donc sera traduit en le code opération correspondant. Par exemple :

ADD Addition de deux valeurs ;

MOV Copie d'une valeur d'une case mémoire vers une autre ;

CMP Compare deux valeurs ;

etc...

Il va de soi, puisque ces opérations sont programmées "en dur" dans les processeurs des machines, que les langages assembleur sont spécifiques à l'architecture du processeur pour lesquels ils sont conçus. Cela signifie que chaque type de processeur a son propre langage assembleur, qui utilise des mnémoniques et des instructions adaptées aux capacités et à la structure interne de ce processeur particulier. Celui que nous utilisons ici est une version du langage "x86" adapté aux processeurs Intel et AMD équipant la plupart des PCs modernes.

Vous constaterez qu'en passant d'un processeur à un autre, d'un simulateur à un autre même au sein de la famille des "x86" il existe de petites variations dans les mots utilisés et même dans la syntaxe (exemple avec la commande `STR` qui sert à stocker une valeur en mémoire et que dans différents systèmes on pourra trouver écrite `STO` ou `STA`). Il est cependant (comme c'est le cas pour `STR`) en général facile de deviner les transpositions de l'une à l'autre.

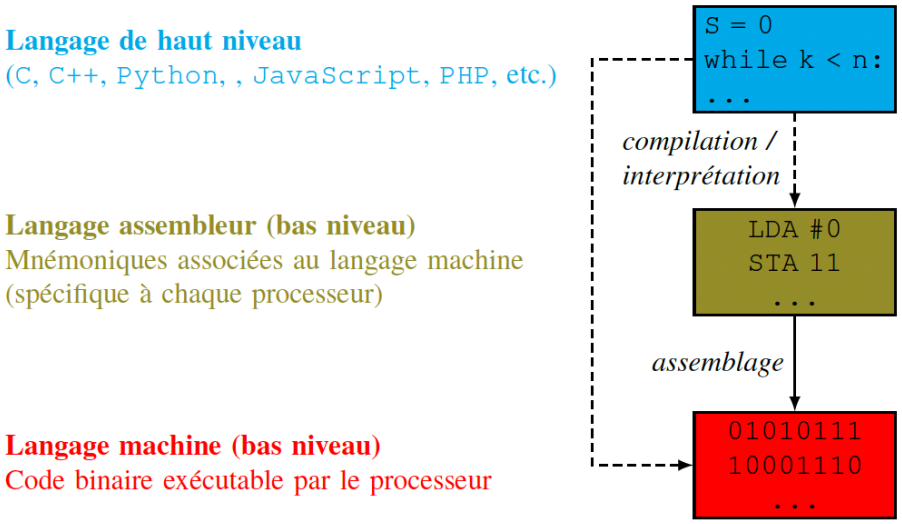
Fonctionnement d'un programme en assembleur:

Un programme en assembleur est donc une succession d'instructions utilisant des mnémoniques — il pourrait ressembler à :

```
        MOV R0, #4
        STR R0,30
        MOV R0, #8
        STR R0,75
        LDR R0,30
        CMP R0, #10
        BNE else
        MOV R0, #9
        STR R0,75
        B endif
else:
        LDR R0,30
        ADD R0, R0, #1
        STR R0,30
endif:
        MOV R0, #6
        STR R0,23
        HALT
```

C'est peu lisible — mais ça l'est infiniment plus que l'étape suivante qui est une traduction de ce programme en instructions machine écrites en binaire.

La chaine globale d'exécution d'un programme en langage informatique comme Python par exemple peut donc se représenter ainsi :



Définition: Compilation, Interprétation, Assemblage

Voilà trois termes qui, sommairement, pourraient tous trois se définir comme des formes de "traduction" mais qui dans les faits, on devrait l'avoir compris d'après ce qui vient d'être présenté, sont tout à fait distincts les uns des autres :

Compilation Processus qui consiste à convertir le code source écrit dans un langage de programmation de haut niveau (comme C ou Java) en code machine directement exécutable par le processeur d'un ordinateur. Ce processus est réalisé par un programme appelé compilateur. Le compilateur

effectue cette conversion en une seule fois, générant un fichier exécutable que l'on peut lancer indépendamment du compilateur.

Interprétation L'interprétation est une méthode d'exécution de programmes où le code source est exécuté directement, instruction par instruction, par un autre programme appelé interpréteur. Contrairement à la compilation, l'interprétation ne produit pas de fichier exécutable permanent ; l'interpréteur lit le code source, le traduit en instructions machine au fur et à mesure et exécute ces instructions immédiatement. Cela permet une plus grande flexibilité et facilité de débogage, mais peut réduire la vitesse d'exécution comparée à un programme compilé. C'est ce que nous utilisons avec notre code Python dans le cadre de ce cours.

Assemblage L'assemblage est le processus de conversion du code source écrit en langage assembleur (un langage de bas niveau proche du langage machine mais légèrement abstrait avec des mnémoniques) en code machine binaire. Ce processus est effectué par un programme appelé assembleur. Le langage assembleur permet aux programmeurs de contrôler plus directement le matériel et est souvent utilisé pour des tâches nécessitant une optimisation de performance ou pour interagir directement avec le matériel de l'ordinateur.

On notera ici une différence importante entre les deux premières notions qui consistent en une interprétation d'un langage pour passer à des instructions machine et nécessite de l'intelligence (une seule instruction Python par exemple pouvant être traduite en plusieurs instructions machine) et la troisième qui n'est en fait qu'une traduction "mot à mot".

Mise en pratique sur un simulateur

Nous allons dans cette section utiliser le simulateur mis en ligne par Peter Higginson.

Exercice 1: Premières instructions en assembleur

Cet exercice est à réaliser dans un document texte (format de votre choix) et à me soumettre soit par mail soit dans mon casier sur l'ENT.

a. Dans un moteur de recherche, rechercher "simulateur aqa peter higginson" et vous rendre sur la page correspondante.

b. Effectuez une par une les opérations suivantes dans le simulateur (en utilisant le bouton `STEP`) et expliquez ce qu'elles font :

   a. `MOV R3,#78`
   b. `STR R3,50`
   c. `LDR R1,50`
   d. `ADD R2,R1,#30`
   e. `ADD R0,R1,R2`

c. Sur la base de vos essais ci-dessus, quelle est la différence entre `50` et `#50` ?

d. Peut-on réaliser les deux premières opérations ci-dessus en une seule fois (faites l'essai !) ?

e. En quoi selon vous `MOV`, `LDR` et `STR` sont-ils des mnémoniques — à quels mots (en anglais) font-ils référence ?

a. Vous vous connectez donc au simulateur...

b. a. On place (`MOV` = "move") la valeur 78 dans le registre R3 du processeur ;

   b. On stocke (`STR` = "store") la valeur contenue dans le registre R3 dans la case mémoire 50 ;

   c. On charge (`LDR` = "loader") la valeur qui était dans la case mémoire 50 dans le registre R1 — en d'autres termes on réalise l'opération exactement inverse de la précédente ;

Exercice 2: Etudions à présent un programme complet

Soit le programme suivant :

```
INP R0,2
INP R1,2
CMP R1,R0
BGT HIGHER
OUT R0,4
B DONE
HIGHER:
OUT R1,4
DONE:
HALT
```

Copiez le dans le simulateur et lancez-le plusieurs fois.

a. Que fait ce programme — quelle serait sa docstring si vous deviez la rédiger ?

b. A quoi servent les mnémoniques `INP` et `OUT` ?

c. Et `B` / `BGT` ?

d. A quoi sert selon vous le premier registre (intitulé `PC`) ?

e. Modifiez-le pour qu'il ne prenne qu'un nombre en entrée et renvoie le plus petit entre celui-ci et la valeur stockée en mémoire à la case 50 (indice : vous aurez besoin de consulter les mnémoniques utilisés à l'exercice précédent).

Exercice 3: Etudions la correspondance avec Python

Soit le code Python suivant :

```
x = 4
y = 8
if x == 10:
  y = 9
else :
  x=x+1
z=6
```

Son interprétation en assembleur est la suivante :

```
MOV R0, #4
STR R0,30
MOV R0, #8
STR R0,75
LDR R0,30
```

```
CMP R0, #10
BNE else
MOV R0, #9
STR R0,75
B endif
else:
LDR R0,30
ADD R0, R0, #1
STR R0,30
endif:
MOV R0, #6
STR R0,23
HALT
```

a. Analysez la correspondance entre les deux programmes : dressez un tableau faisant correspondre chaque ligne Python avec celle(s) en assembleur.

b. À quoi correspondent les adresses mémoires 23, 75 et 30 ?

N'hésitez pas à utiliser le simulateur pour vous aider !

À Savoir ⇄ À Réviser:

Ce qu'il faut retenir de cette section sur le langage assembleur est :

- Comprendre et pouvoir sommairement expliquer l'évolution historique de l'informatique qui a amené à l'avènement des langages informatiques ;
- Se remémorer le modèle de von Neumann ;
- Comprendre et pouvoir expliquer les notions de langage machine et langage assembleur ;
- Comprendre et pouvoir expliquer les différents niveaux de langage informatique et les processus de passage de l'un à l'autre (compilation...) ;
- Sans connaître par cœur les mnémoniques que nous avons utilisés ici, être capable de réaliser des exercices du type des trois précédents — et notamment le dernier dans lequel il s'agit de lire du code assembleur et de le relier à son code Python correspondant.

3 Les systèmes d'exploitation

3.1 Présentation générale

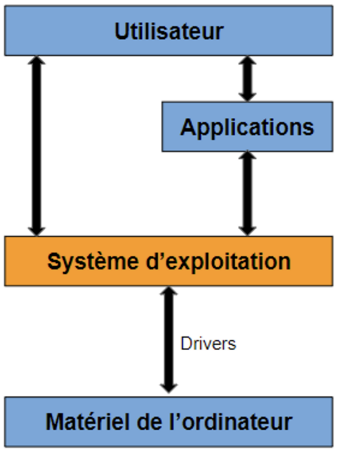
Introduction

Explication video:

Commençons pas une petite vidéo Youtube de l'INRIA présentant les rôles principaux d'un système d'exploitation.

Fonctions d'un système d'exploitation

On en revient toujours au même schéma dans lequel, par rapport à la section précédente, on "monte d'un cran"...



L'OS (*Operating System* en anglais) ou système d'exploitation est principalement un intermédiaire entre le matériel physique et les applications — même si (et c'est ce que nous allons faire en séance) les utilisateurs peuvent également y accéder directement. L'utilisateur utilise des applications qui communiquent avec l'OS ; ce dernier leur fournit les ressources matérielles dont elles ont besoin.

Un OS a notamment les rôles suivants :

Gestion du processeur : il est chargé de gérer l'allocation (le partage) du temps de traitement du processeur entre les différents programmes ;

Gestion de la mémoire et de l'espace physique alloué à chaque application ;

Gestion des entrées / sorties : il unifie unifie et contrôle l'accès des programmes aux ressources matérielles par l'intermédiaire des pilotes (ou *drivers*) ;

Gestion des applications : il est chargé de la bonne exécution des applications en leur affectant les ressources nécessaires à leur bon fonctionnement ;

Gestion des droits : il est chargé de la sécurité liée à l'exécution des programmes en garantissant que les ressources ne sont utilisées que par les programmes et utilisateurs possédant les droits adéquats ;

Gestion des fichiers : il gère la lecture et l'écriture dans le système de fichiers et les droits d'accès aux fichiers par les utilisateurs et les applications.

Pour réaliser ces tâches, un système d'exploitation est principalement constitué de :

Un noyau (kernel) : c'est la partie la plus importante de l'OS dans laquelle se trouve les programmes de gestion de la mémoire, du processeur, etc. et qui fournit aux logiciels une interface pour utiliser le matériel ;

Un système de fichiers (FS, file system) : une façon d'organiser les données sur un support de stockage ;

Un interpréteur de commande (shell) : il permet la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes.

Différents systèmes d'exploitation

Comme pour tous les logiciels, il existe des OS libres et d'autres dits "propriétaires" :

- Les deux systèmes d'exploitation les plus utilisés dans les ordinateurs personnels, Windows et MacOS sont tous deux propriétaires — ils sont payants (même si leur prix est généralement inclus à celui de la machine que l'on achète), appartiennent à une société privée (Microsoft et Apple), et ne peuvent être modifié ;
- UNIX est un système à l'origine développé par AT&T Bell Labs dans les années 60 et sur lequel de très nombreux autres OS se sont fondés (Linux et MacOS en particulier). Aujourd'hui il en existe de multiples versions propriétaires (IBM, HP, Sun...) d'UNIX.
- Linux est un système d'exploitation open source inspiré d'Unix.La communauté autour de Linux est très active, contribuant à une multitude de distributions différentes adaptées à divers besoins et préférences. Une des distributions les plus connues, Ubuntu, est connue pour sa simplicité d'installation et d'utilisation – et c'est celle que nous utiliseons sur les ordinateurs NSI ;
- Une variante connue de Linux est Chromium OS qui est l'OS des Chromebooks ;
- Il existe une foule d'autres OS adaptés à différents contextes — les OS de smartphones et tablettes (iOS et Androïd), des OS des systèmes embarqués dans les voitures, les trains, les avions, des OS pour certains serveurs spécialisés...

A retenir

À Savoir ⇄ À Réviser:

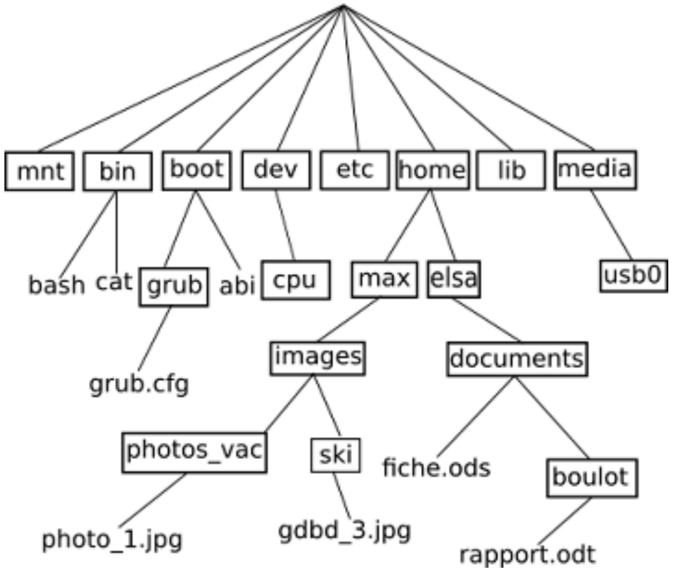
Ce qu'il faut retenir de cette section sur les OS :

- Comprendre et expliquer leur positionnement dans le fonctionnement d'un ordinateur ;
- Comprendre et expliquer quels sont leurs rôles principaux ;
- Comprendre et expliquer les trois composantes centrales qui les constituent ;
- Citer les principaux OS commerciaux (propriétaires) et libres.

3.2 FS & shell UNIX

Un système de fichiers est une façon d'organiser et de stocker une arborescence sur un support (disque, disquette, cd ...). Chaque OS propriétaire a développé sa propre

organisation. Dans le cas d'UNIX (et donc de Linux) la structure en est une d'un arbre comme illustré ci-dessous.



Une des possibilités qu'offre l'interface de ligne de commande (le *shell*) UNIX est de se déplacer à l'intérieur de cette arborescence, de lire des fichiers, de lister le contenu de répertoires, de déplacer ou copier des éléments, etc. Historiquement, c'était le seul moyen d'effectuer ces opérations — mais maintenant la plupart sont réalisables au moyen de la souris dans l'interface graphique. Pourquoi en ce cas, me direz-vous, nous embêter avec ça ? Plusieurs raisons :

- Ce que font les interfaces graphique n'est qu'une adaptation d'une partie de ces commandes — elles ne permettent pas de tout faire ;
- En ligne de commande on peut accéder partout dans l'arborescence, même dans les répertoires dits "cachés" — ce n'est pas le cas dans l'interface graphique ;
- Pour manipuler des fichiers depuis du code (Python par exemple) il n'y a pas d'autre moyen que de connaître ces commandes ;
- Et enfin (même si vous ne me croirez pas tout de suite), à l'usage, c'est *beaucoup* plus rapide et efficace de travailler en ligne de commande que dans une interface graphique !

Pour découvrir les fonctionnalités (ou tout au moins une partie d'entre elles) qui nous sont rendues disponibles par le shell Linux, nous allons effectuer trois activités les introduisant :

a. Un jeu en ligne, nommé Terminus (un TP développé par Charles Poulmaire) ;
b. Une série d'activités à effectuer directement sur votre machine ;
c. Pour ceux qui avancent bien, un petit défi.

Exercice 4: Jeu "Terminus"

Avant de nous lancer dans des manipulations "réelles" dans shell, nous allons nous initier à ces commandes au moyen d'un petit jeu en ligne, Terminus.

Rendez-vous sur l'espace documentaire partagé de l'ENT et effectuez le TP développé par Charles Poulmaire sur la base du jeu en ligne Terminus. Le fichier contenant les activités s'appelle **TP1-Terminus.pdf** et il se trouve dans le

même répertoire que ce cours (dans "1NSIGr2 / COURS / 07_ArchitectureSystème").

L'activité est à réaliser sur papier en parallèle de l'avancement du jeu et sera à rendre.

Exercice 5: Activités shell

Rendez-vous sur l'espace documentaire partagé de l'ENT et effectuez le second TP que vous y trouverez — le fichier contenant les activités s'appelle **02_TP2-Shell.pdf** et il se trouve dans le même répertoire que ce cours (dans "1NSIGr2 / COURS / 07_ArchitectureSystème").
Les activités seront à réaliser directement dans le terminal de votre machine NSI.