Chapitre 1
Éléments d'architecture des ordinateurs

Dans ce premier chapitre, nous allons très succinctement décrire les principaux constituants matériels d'un ordinateur (le *hardware*) ainsi que les principes généraux qui régissent son système d'exploitation (le *software*). Ces généralités concerneront avant tout les ordinateurs personnels mais restent valables pour la plupart des machines numériques telles les tablettes ou les smartphones.

Introduction

L'informatique au sens où on l'entend aujourd'hui est née vers la fin des années 1930 des efforts conjugués de mathématiciens d'une part, et d'ingénieurs d'autre part.
À cette époque, certains logiciens, les plus connus étant Church et Turing, se posent la question suivante : *Qu'est-ce qu'un calcul ?*, et donnent naissance à la théorie de la calculabilité, qui vise à définir précisément ce qui est calculable de ce qui ne l'est pas. Church est sans doute le premier à avoir pensé pouvoir définir formellement ce que l'on s'accorde à reconnaître intuitivement comme un calcul en développant un nouveau formalisme : le « lambda-calcul », et en imposant l'idée, fondamentale en algorithmique, qu'une fonction peut être définie par des règles de calcul.
Turing quant à lui explore une autre approche en liant la notion de calcul à la notion de *machine* : il imagine une machine idéale pouvant fonctionner sans intervention humaine, qui n'avait pas (encore) de réalisation physique mais qui semblait plausible. Est alors calculable toute fonction qui peut être exécutée par une machine de Turing. Nous savons maintenant que ces deux approches sont équivalentes, et ceux d'entre vous qui suivront l'option informatique en complément de ce cours pourront effleurer chacune de ces deux théories à travers l'étude d'un langage fonctionnel, Caml, qui découle directement des concepts du lambda-calcul et en étudiant une partie de la théorie des automates née des travaux de Turing.

En parallèle à ces travaux théoriques, des ingénieurs cherchent à construire des machines électroniques ou électro-mécaniques capables d'exécuter des calculs complexes à grande vitesse. Parmi ces précurseurs un nom se détache, celui de Von Neuman. Au sein du projet ENIAC ce dernier décrit dans les années 1940 un schéma d'architecture de calculateur qui reste encore aujourd'hui d'une étonnante actualité. Rétrospectivement, on peut affirmer que bien qu'à cette époque les mathématiciens et les ingénieurs se soient mutuellement assez largement ignorés, ce sont les travaux de Von Neuman qui permettent la première réalisation concrète d'une machine de Turing.

1. Principaux composants d'un ordinateur

Un ordinateur peut être décrit comme une machine de traitement de l'*information* [1] obéissant à des programmes constitués de suites d'opérations arithmétiques et logiques : il est capable d'acquérir de l'information, de la stocker, de la transformer et de la restituer sous une autre forme.

- L'acquisition de l'information se fait par l'intermédiaire de *périphériques d'entrées* que sont le clavier, la souris, le micro, la webcam, le scanner, l'écran tactile, ...
- Le stockage de l'information se fait dans ce qu'on appelle généralement la *mémoire* d'un ordinateur. On peut distinguer la mémoire de masse (disque dur, clé USB, etc.) destinée à un stockage persistant même en absence d'alimentation électrique, de la mémoire vive (la RAM [2]) utilisée par le processeur pour traiter les données qui nécessite d'être alimentée électriquement.
- La transformation de l'information est le rôle du *processeur* (le CPU [3]), composé de deux éléments : l'unité de commande, responsable de la lecture en mémoire et du décodage des instructions et l'unité de traitement [4] qui exécute les instructions qui manipulent les données.

1. d'ailleurs, le mot *informatique* vient de la contraction des mots *information* et *automatique*.
2. pour Ramdom Access Memory, ou mémoire à accès direct en français.
3. de l'anglais Central Processing Unit.
4. aussi appelée Unité Arithmétique et Logique.

– Enfin, la restitution de l'information utilise les *périphériques de sorties* que sont l'écran, l'imprimante, les enceintes acoustiques, etc.

1.1 Le modèle de von Neumann

Les différents éléments décrits ci-dessus s'articulent suivant l'architecture conçue par Von Neuman et son équipe (voir la figure 1).

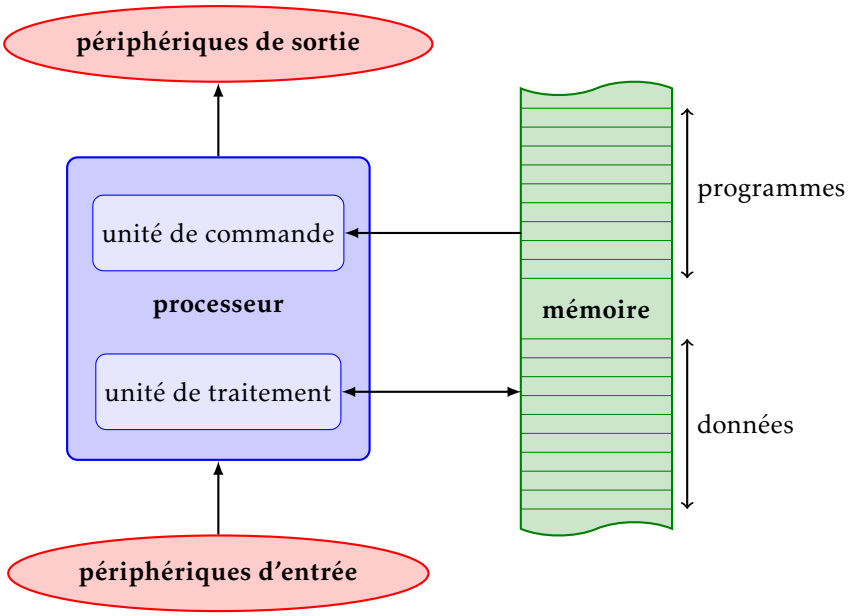


Figure 1 – L'architecture d'un ordinateur selon le modèle de von Neuman.

La circulation de l'information entre ces différents éléments (représentée par des flèches sur ce schéma) est assurée par des connexions appelées *bus* (bus de données, bus d'adresse, bus de signal lecture/écriture…) ; il s'agit tout simplement de fils conducteurs utilisés pour transmettre des signaux binaires.
Ceci reste bien évidemment schématique, et l'architecture actuelle des ordinateurs s'est enrichie depuis cette époque, en particulier sur deux points :

– les entrées et sorties, initialement commandées par l'unité centrale, sont depuis le début des années 1960 sous le contrôle de processeurs autonomes (c'est le cas par exemple des cartes graphiques chargées de produire une image sur un écran) ;

– les ordinateurs comportent maintenant des processeurs multiples, qu'il s'agisse d'unité séparées ou de « cœurs » multiples à l'intérieur d'une même puce. Cette organisation permet d'atteindre une puissance globale de calcul élevée sans augmenter la vitesse des processeurs, limitée par les capacités d'évacuation de la chaleur.

Néanmoins, les deux innovations majeures du modèle de Von Neuman restent d'actualité. La première innovation est la séparation nette entre l'unité de commande, qui organise le flot de séquencement des instructions, et l'unité arithmétique, chargée de l'exécution proprement dite de ces instructions. La seconde innovation, la plus fondamentale, est l'idée d'un programme *enregistré* : au lieu d'être codées sur un support externe, les instructions sont enregistrées dans la mémoire et ainsi, *un programme peut être traité comme une donnée par un autre programme*. Cette idée, présente en germe dans la machine de Turing, trouvait ici sa concrétisation.

1.2 La mémoire principale

Une information traitée par un ordinateur peut être de différents types (texte, nombre, image, son, …) mais elle est toujours représentée et manipulée par l'ordinateur sous forme binaire, c'est à dire par une suite de 0 et de 1 appelés *bits* (pour *binary digits*, chiffres binaires en anglais). Par la suite, il faudra prendre garde à bien distinguer un objet de sa représentation machine : deux objets distincts, disons un nombre et un caractère, peuvent avoir la même représentation machine, mais vont différer par ce qu'on appelle leur *type* dans un langage de programmation, voire tout simplement par le contexte dans lequel ils sont utilisés.

Mais avant de s'intéresser à la façon dont ces objets sont représentés en mémoire, nous allons décrire les différentes sortes de mémoires qui coexistent au sein d'un ordinateur, et qui se distinguent par leur capacité et leur vitesse.

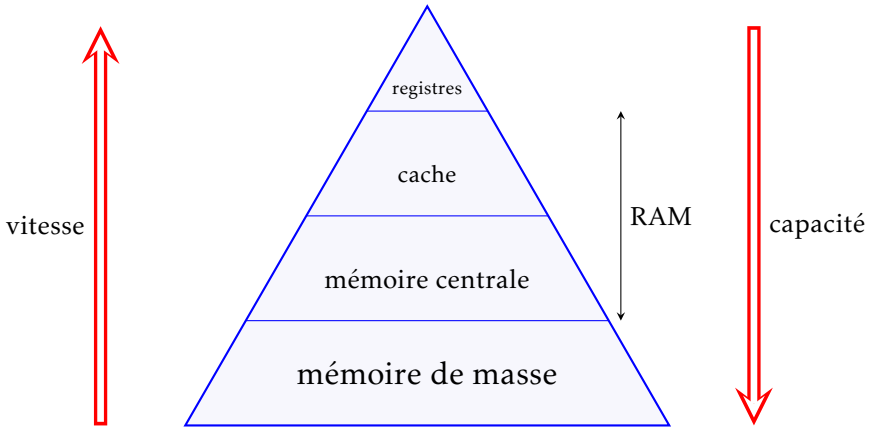


FIGURE 2 – Les différents types de mémoire dans un ordinateur.

- Les *registres* sont des emplacements mémoires internes au processeur dans lesquels ces derniers exécutent les instructions (on en reparlera plus loin). Selon les processeurs leur nombre varie d'une dizaine à une centaine, et leur capacité dépasse rarement quelques dizaines d'octets (en général 8, 16, 32 ou 64 bits par registre).
- La mémoire *cache* (ou mémoire tampon) est une mémoire rapide permettant de réduire les délais d'attente des informations stockées dans la mémoire centrale. En effet, cette dernière possède une vitesse bien moins importante que le processeur et stocker les principales données devant être traitées par le processeur dans le cache permet d'en accélérer le traitement. Il existe plusieurs niveaux de caches qui se distinguent par leur vitesse d'accès, le premier niveau ayant une vitesse se rapprochant de celle des registres. La capacité d'un cache se compte généralement en kilo-octets pour les deux premiers niveaux et en mega-octets pour le troisième.
- La *mémoire centrale* contient le code et les données des programmes exécutés par le processeur ; elle se compte actuellement en giga-octets. Elle est divisée en emplacements de taille fixe appelés *bytes* (en général 8 bits, c'est-à-dire un *octet*) : un byte est la plus petite unité adressable d'un ordinateur. Chaque mot-mémoire est repéré par un numéro qu'on appelle son *adresse*. Actuellement, les adresses sont codées dans des registres de 32 ou 64 bits, ce qui autorise au maximum 2^{32} ou 2^{64} adresses différentes. Sachant que 2^{32} octets correspondent approximativement à 4 Go, un processeur 32 bits ne peut donc en théorie gérer plus de 4 Go de mémoire vive. Ceci explique la généralisation actuelle des processeurs 64 bits pour accompagner l'augmentation de la mémoire vive associée à un ordinateur.
- Enfin, la *mémoire de masse* désigne tous les moyens de stockage pérenne dont on dispose : disque dur, clé USB, DVD, etc. accessibles en lecture/écriture ou en lecture seule (à proprement parler, cette mémoire de masse s'apparente donc plus à un périphérique d'entrée / sortie). Lors du lancement d'une application, les données essentielles sont copiées dans la mémoire centrale, et les programmes sont conçus pour minimiser les accès à la mémoire de masse, très lente comparativement aux autres mémoires (d'ailleurs il existe aussi un cache entre mémoire de masse et mémoire centrale pour accélérer ces transferts). La mémoire de masse tend de plus en plus à être mesurée en tera-octets.

Remarque. On aura noté que l'unité de base de mesure de la mémoire est l'*octet* (égal à 8 bits) et que les préfixes usuels (kilo, méga, giga ...) sont couramment utilisés. Cependant, ces derniers sont liés à la base 10 et en réalité assez mal adaptés à un univers dans lequel la base 2 est reine. C'est pourquoi ont été inventés des préfixes spécifiques à la mesure de la mémoire informatique. Partant du principe que $10^3 \approx 2^{10}$, on utilise les préfixes kibi, mébi, gibi ... dont la définition est donnée figure 3.
Malheureusement ces notations sont assez récentes (1998), les premiers informaticiens s'étant contenté de modifier l'usage des préfixes usuels, et nombreux sont encore ces derniers à parler de kilo-octet pour désigner en réalité un kibi-octet. On notera cependant que si l'erreur est faible s'agissant de ko (2,4%), celle-ci augmente significativement s'agissant de Go (7,4%) voire de To (10%).

préfixe	valeur théorique	mésusage
1 k (kilo)	$10^3 = 1\,000$	$2^{10} = 1\,024$
1 M (méga)	$10^6 = 1\,000\,000$	$2^{20} = 1\,048\,576$
1 G (giga)	$10^9 = 1\,000\,000\,000$	$2^{30} = 1\,073\,741\,824$

préfixe	valeur théorique
1 ki (kibi)	$2^{10} = 1\,024$
1 Mi (mébi)	$2^{20} = 1\,048\,576$
1 Gi (gibi)	$2^{30} = 1\,073\,741\,824$

FIGURE 3 – Multiples usuels en informatique.

Lecture et écriture
Seul le processeur peut modifier l'état de la mémoire ; il peut :

- écrire à un emplacement (le processeur donne une valeur et une adresse et la mémoire range la valeur à l'emplacement indiqué par l'adresse) ;
- lire un emplacement (le processeur demande à la mémoire la valeur contenue à l'emplacement dont il indique l'adresse).

Notons que les opérations de lecture et d'écriture portent en général sur plusieurs octets contigus en mémoire, qu'on appelle un *mot mémoire*. La taille d'un mot mémoire est de 4 octets pour un processeur 32 bits et de 8 octets pour un processeur 64 bits.

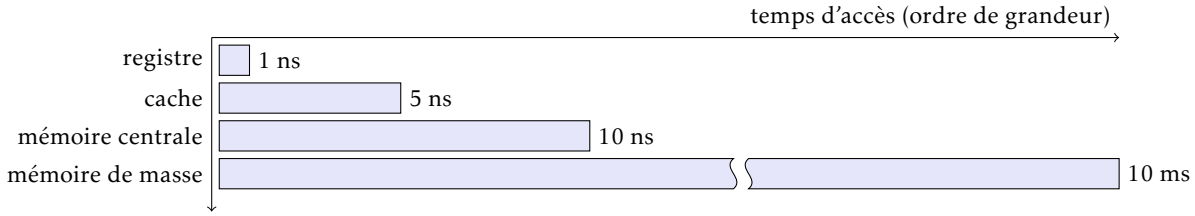


FIGURE 4 – Le *temps d'accès* est l'intervalle de temps entre la demande de lecture/écriture et la disponibilité de la donnée.

1.3 Le processeur

Un processeur, ou encore CPU (*Central Process Unit*), est composé d'une unité de commande et d'une unité de traitement.

L'*unité de commande* décode les instructions d'un programme et les transcrit en une succession d'instructions élémentaires envoyées à l'unité de traitement. Son travail est cadencé par une horloge, ce qui permet de coordonner les instructions et d'optimiser le traitement d'une succession d'instructions.

L'*unité de traitement* est composée d'une unité arithmétique et logique (UAL), d'un registre d'entrée et d'un accumulateur [5] (figure 5).

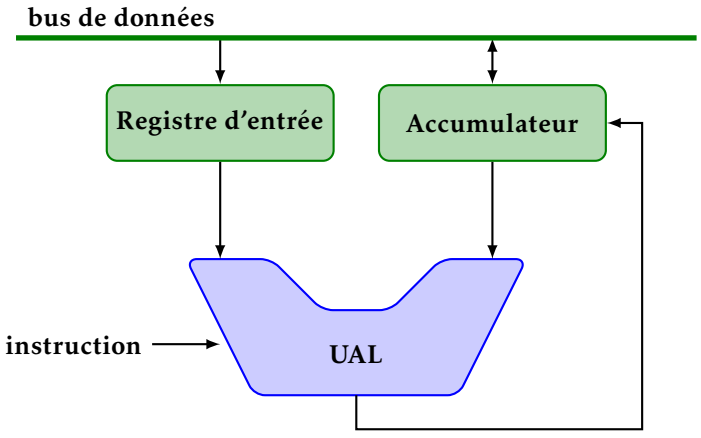


FIGURE 5 – Schéma d'une unité de traitement.

5. Cette représentation est très simplifiée et ne correspond pas à tous les modèles de processeurs, mais les principes restent les mêmes.

Suivant le contexte, un mot mémoire (une donnée) peut représenter un entier, une adresse, ou encore une instruction transmise au processeur. Ainsi, un programme n'est qu'une suite d'instructions, autrement dit un certain nombre de mots mémoire écrits en binaire. L'ensemble de ces instructions comprises par le processeur s'appelle le *langage machine* ; pour en faciliter la lecture et l'écriture, on représente les mots binaires correspondant aux instructions par des mnémoniques comme LOAD, ADD, JMP ... ; on parle alors de *langage assembleur* ; ce langage est en bijection stricte avec le langage machine.

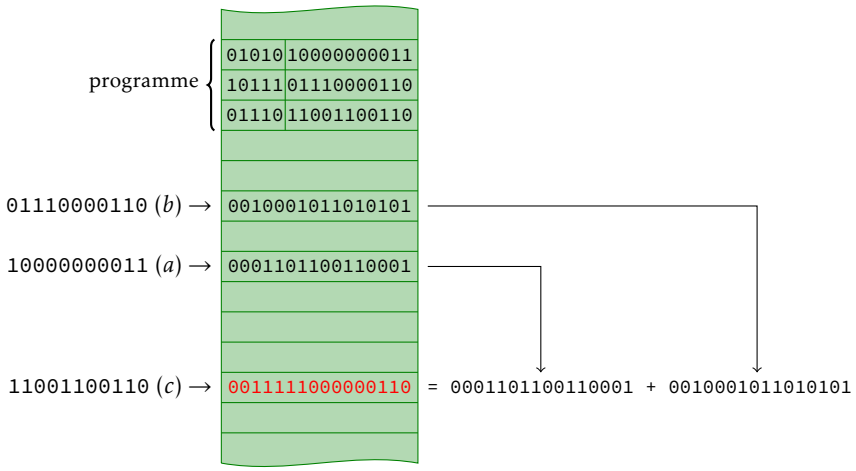
Un exemple de programme en assembleur
Pour illustrer le fonctionnement d'un processeur, nous allons imaginer une machine dont les mots-mémoire sont de 16 bits. Chaque instruction comporte un code d'opération sur 5 bits spécifiant l'opération à effectuer et une adresse sur 11 bits qui est celle de l'opérande de cette instruction. Ainsi il peut y avoir $2^5 = 32$ instructions différentes et $2^{11} = 2048$ emplacements en mémoire.
Nous allons écrire un programme en assembleur utilisant les trois instructions suivantes :

assembleur	machine	rôle
LOAD X	01010	charge dans l'accumulateur le contenu de l'emplacement d'adresse X
STO X	01110	range le contenu de l'accumulateur dans l'emplacement d'adresse X
ADD X	10111	charge dans le registre le contenu de l'emplacement d'adresse X et l'ajoute à l'accumulateur

Le programme que nous souhaitons réaliser consiste à réaliser l'opération : $c \leftarrow a + b$: ranger à l'adresse c la valeur de la somme des entiers stockés aux adresses a et b. Avec notre modèle les adresses sont décrites sur 11 bits ; choisissons arbitrairement $a = 10000000011$, $b = 01110000110$, $c = 11001100110$. En assembleur le programme va ressembler à ceci :

```
LOAD A
ADD B
STO C
```

En langage machine, ce programme sera stocké dans trois mots-machine consécutifs quelque part dans la mémoire :



Mais si l'introduction de l'assembleur dans les années 1950 est la marque d'un progrès vers l'abstraction, un défaut fondamental subsistait : un programme en assembleur est écrit en termes de *ce que sait faire la machine* ; l'invention des langages de programmation va permettre de changer de paradigme en rédigeant un programme en termes de *ce que veut faire l'utilisateur*. En outre, cette démarche présente l'avantage d'être indépendante de la machine : un langage de programmation s'exprime en termes indépendants de la nature du processeur présent dans l'ordinateur, contrairement à l'assembleur.

1.4 Langages de programmation

- **Fortran**

Fortran (1954) est le premier langage de haut niveau ayant connu une large diffusion. Il était destiné au calcul scientifique (la principale application à l'époque), et son usage s'est vite répandu auprès des scientifiques et des ingénieurs. L'importance de sa bibliothèque de fonctions mathématiques qui s'est constituée au fil du temps est

un des facteurs qui explique sa longévité, car ce langage reste encore très utilisé à l'heure actuelle (il est vrai au prix d'une importante évolution depuis sa naissance). D'ailleurs, certaines des bibliothèques mathématiques que nous utiliserons cette année utilisent du code écrit en Fortran, même si nous ne nous en rendrons pas compte.

Un langage compilé
Fortran est un langage *compilé*, c'est-à-dire qu'il nécessite l'usage d'un *compilateur*, un programme spécialisé chargé de traduire un code source écrit dans un langage de haut niveau vers le langage machine. En général, l'efficacité du code produit est considéré comme un facteur déterminant de qualité, et ce sont les efforts constants consacrés à l'amélioration des compilateurs Fortran qui sont une des autres cause de la longévité de ce langage.

Interprètes et compilateurs

Nous l'avons déjà dit, le seul langage compris par le processeur est le langage machine, langage composé d'un ensemble restreint d'opérations élémentaires, dont l'exécution est implémentée directement dans les composants du processeur. À l'inverse, un langage de programmation se place à un niveau d'abstraction indépendant de la machine, mieux à même de traduire les objectifs du programmeur. Ceci implique la nécessité d'utiliser une sorte de traducteur entre le langage de programmation et le langage machine. Schématiquement, il en existe de deux types :

- un *compilateur* est un programme qui traduit un code écrit dans un langage de programmation en langage machine. Le code produit (qu'on appelle un *exécutable*) peut être exécuté directement sur la machine sur laquelle il a été compilé.
- un *interprète* est un programme qui simule une machine virtuelle dont le langage machine serait le langage de programmation lui-même. Le code va être lu, analysé et exécuté instruction par instruction par l'interprète.

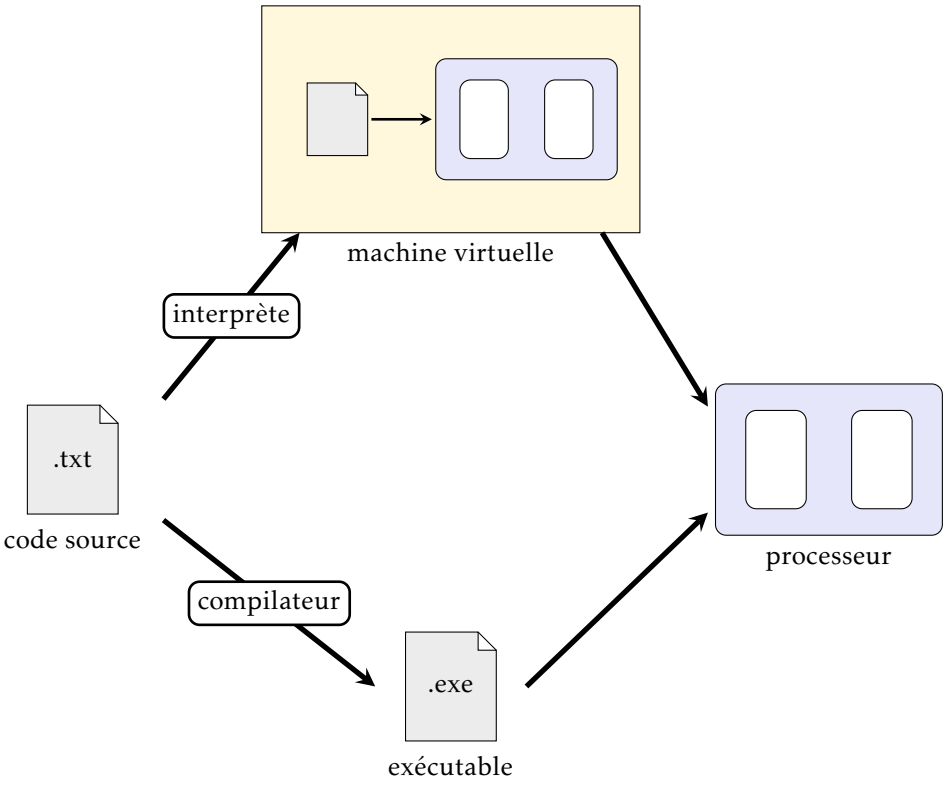


Figure 6 – Compilation et interprétation.

L'exécutable créé par le compilateur dépend de la machine sur lequel il a été créé, alors qu'un programme interprété est indépendant de la machine (puisque c'est l'interprète lui-même qui dépend de la machine). En général, un programme compilé est nettement plus rapide qu'un programme interprété puisque toute la phase d'analyse et de vérification du code source a déjà été faite lors de la phase de compilation.

- **Algol et Cobol**

À partir de 1955 un grand nombre de langages furent proposés, souvent pour répondre à des faiblesses du Fortran. L'Algol (pour Algorithmic Language) est le fruit d'un travail universitaire visant à définir un langage « universel » de programmation orienté vers le calcul scientifique, avec un accent sur la généralité, la sécurité et la rigueur de la définition, points faibles du Fortran. Finalement assez peu utilisé, il a néanmoins posé les bases de l'algorithmique et a influencé de nombreux langages plus récents : Pascal (1970), C (1972) et ses descendants, Java (1995), Python (1990) et plus généralement la plus-part des langages impératifs actuels.

Dans un autre registre, le langage Cobol (1959) a été créé pour répondre à un besoin croissant en informatique de gestion, domaine dans lequel Fortran était peu adapté. Bien que présentant de nombreux défauts, il est néanmoins resté pendant longtemps un langage de référence en matière d'informatique de gestion. Il n'a pas eu de véritable descendant, mais a connu de nombreuses évolutions.

- **Lisp**

La naissance du Lisp en 1958 est importante à plus d'un titre. C'est tout d'abord le premier langage fonctionnel inspiré du lambda-calcul de Church ; à ce titre on peut le considérer comme l'ancêtre de tous les langages fonctionnels et en particulier de Caml (1985), le langage qui sera utilisé en option informatique. C'est aussi le premier langage interprété.

- **Python**

Dans les années 1960-70 les chercheurs en informatique poursuivent en vain le mythe du « langage universel », jusqu'à prendre conscience de la vanité de cette recherche. Il n'existe pas un mais des langages de programmation, chacun ayant ses points forts et ses points faibles, ainsi que ses domaines d'application privilégiés.

Le langage que nous utiliserons pour illustrer ce cours est le Python (né en 1990). Il s'agit d'un langage de style majoritairement impératif, utilisé dans de nombreux domaines : développement web et internet, calcul numérique et scientifique, éducation. Il s'agit d'un langage interprété, à une nuance près : pour être plus efficace, un script Python est tout d'abord compilé en un langage intermédiaire entre le langage de programmation et le langage machine et qu'on appelle le *bytecode*. Ce dernier est ensuite interprété par une machine virtuelle, le processus complet étant transparent pour l'utilisateur standard. Les performances des interprètes de bytecode sont en général bien meilleures que celles des interprètes de langages de plus haut niveau car le bytecode est déjà plus proche du langage machine et moins intelligible par l'homme, mais il reste indépendant de la machine sur laquelle il a été créé.

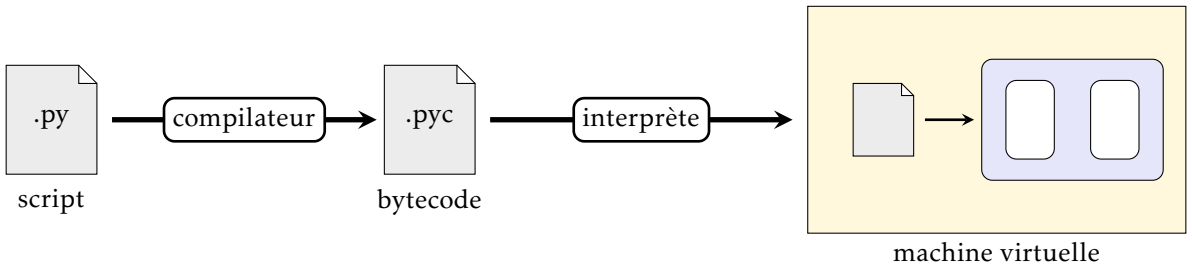


Figure 7 – Interprétation d'un script Python.

L'interprète de bytecode le plus utilisé en Python s'appelle CPython ; il est écrit en C, mais d'autres existent (Jython, écrit en Java par exemple).

2. Système d'exploitation

Le *système d'exploitation* est le premier programme exécuté lors de la mise en marche de l'ordinateur ; il permet d'accéder aux ressources matérielles de celui-ci, en particulier les organes d'entrées/sorties et la mémoire de masse (le disque dur). Le rôle principal du système d'exploitation est d'isoler les programmes des détails du matériel : un programme désirant afficher une image ne va pas envoyer directement des instructions à la carte graphique de l'ordinateur, mais plutôt demander au système d'exploitation de le faire. C'est ce dernier qui doit connaitre les détails du matériel (dans ce cas le type de carte graphique et les instructions qu'elle comprend). Cette répartition des rôles simplifie grandement l'écriture des programmes d'application.

De cette première tâche découle une seconde : celle d'assurer l'interface entre le ou les utilisateurs et la machine, en fournissant à chacun d'eux une machine virtuelle à travers laquelle chacun pourra interagir avec la machine.

Actuellement, les systèmes d'exploitation les plus connus appartiennent à l'une des deux familles suivantes : la famille UNIX (Mac OS X et Linux pour les ordinateurs, iOS et Android pour les tablettes et smartphones) et la famille WINDOWS. Dans la suite de cette partie, nous ne décrirons que certains aspects de la première de ces deux familles, étant entendu que les principes généraux que nous allons évoquer sont partagés par tous les systèmes d'exploitation actuels.

2.1 UNIX

Au début des années 1970, une équipe des laboratoires BELL donne naissance à un nouveau système d'exploitation : UNIX. Ce dernier est écrit dans un tout nouveau langage de programmation, le C, créé par un des membres de cette équipe, Dennis RITCHIE. Ce langage, proche de la machine physique (il permet notamment de manipuler directement des adresses mémoires) et pourvu pour cette raison de compilateurs très efficaces, devait par la suite devenir le langage privilégié pour l'écriture de logiciels destinés à interagir directement avec la machine, et en particulier les systèmes d'exploitation.

UNIX est organisé autour d'un noyau de base qui sert de support à un interprète de langage de commande, le *shell*[6]. Pour l'utilisateur, ce langage de commande permet de réaliser des tâches complexes par assemblage d'actions élémentaires par l'intermédiaire d'une interface textuelle (le *terminal*) ou graphique.

2.2 Session utilisateur

Les systèmes d'exploitation actuels des ordinateurs (et en particulier UNIX) sont multi-tâches et multi-utilisateurs : ils sont capables de partager des ressources selon une hiérarchie de droits d'accès. Chaque utilisateur se voit attribuer un compte comportant :

- un identifiant associé à un mot de passe ;
- un répertoire d'accueil personnel destiné à héberger tous les sous-répertoires et fichiers qui lui appartiennent.

L'utilisateur commande au système d'exploitation par l'intermédiaire d'une *interface*. Jadis purement textuelle (l'utilisateur n'avait d'autre choix que d'entrer ses commandes par l'intermédiaire de son clavier), le développement de l'informatique grand public a conduit à l'apparition d'interfaces graphiques se manipulant à l'aide d'un pointeur dirigé par une souris ou par un doigt (écran ou pavé tactile). Leur avantage est de permettre une utilisation plus intuitive du logiciel d'exploitation, mais une interface par ligne de commande reste plus efficace pour un utilisateur aguerri. C'est pourquoi les systèmes d'exploitation modernes continuent de proposer des interprètes de commandes textuelles : c'est le *Terminal* sous OS X ou Linux et *PowerShell* sous windows.

2.3 Hiérarchie des fichiers

Quel que soit le système d'exploitation, l'utilisateur interagit avec une machine virtuelle bien différente de la machine réelle. Dans cette machine virtuelle, l'ensemble des fichiers (applications ou données) est structuré sous forme arborescente : les feuilles de cet arbre sont les fichiers, les nœuds étant appelés des *répertoires*. On est bien loin de la machine réelle pour laquelle un fichier n'est pas nécessairement constitué d'emplacements consécutifs en mémoire (c'est le phénomène de *fragmentation*).
Sous UNIX par exemple, la racine de cet arbre est désignée par `/` ; c'est un répertoire qui contient lui-même d'autres répertoires : `bin` pour les exécutables, `dev` pour les périphériques, `etc` pour le système, `home` pour les utilisateurs, `tmp` pour les fichiers temporaires, `usr` pour les outils, etc. et ces répertoires contiennent eux-mêmes d'autres répertoires et fichiers (voir illustration figure 8).
Au sein de cette hiérarchie, chaque utilisateur possède une branche de l'arbre constitué d'un répertoire d'accueil qui contient les sous-répertoires et fichiers qui lui appartiennent. Dans l'exemple représenté ici, Alice et Bob possèdent tous deux un répertoire personnel rangé dans le répertoire `home`.

Chemin d'accès
Tout fichier est référencé par son chemin d'accès, c'est à dire la description du chemin qu'il faut parcourir dans l'arborescence à partir d'un certain répertoire pour atteindre le fichier en question. Le chemin est spécifié par les noms des répertoires séparés par le caractère « / » et suivi du nom du fichier.

6. Il en existe de multiples versions.

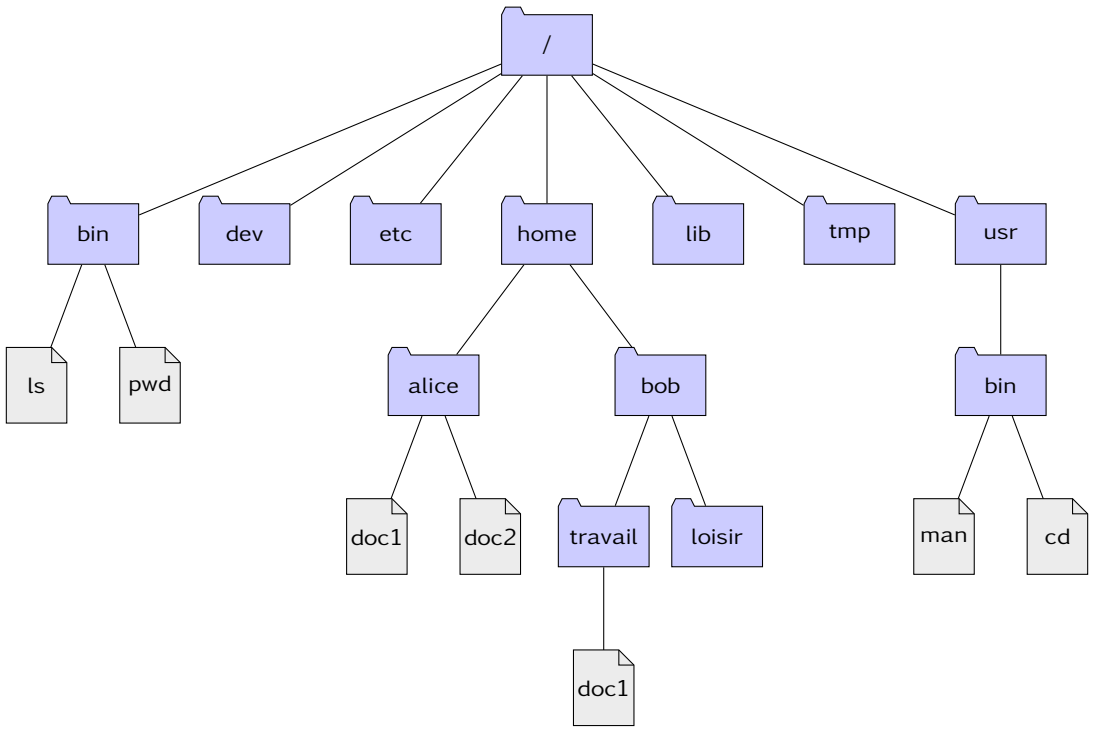


FIGURE 8 – un exemple partiel d'arborescence UNIX

Lorsqu'on commence la description du chemin par la racine, on dit que le chemin est *absolu*. Par exemple, l'un des deux documents possédé par Alice est référencé par le chemin : `/home/alice/doc1`. Il n'a bien entendu rien à voir avec le fichier `/home/bob/travail/doc1` possédé par Bob [7].

Il est aussi possible de décrire un fichier relativement à la position d'un autre répertoire ; on parle alors de chemin *relatif*. Sous UNIX par exemple, le symbole « ~ » désigne le répertoire d'accueil d'un utilisateur. Lors d'une session ouverte par Bob, le document qui se trouve dans le répertoire travail peut être décrit par : `~/travail/doc1`. Enfin, un fichier peut être décrit relativement au répertoire courant. Ce dernier est représenté par le caractère « `.` », et on peut remontrer dans la hiérarchie au répertoire père d'un répertoire donné en le désignant par « `..` ». Par exemple, si le répertoire courant est le répertoire loisir de Bob, on peut décrire le premier des deux fichiers d'Alice par : `../../alice/doc1`.

Il est possible d'explorer cette arborescente en utilisant un terminal (ou le powershell) et les quelques instructions suivantes [8] :

- `pwd` affiche le répertoire courant ;
- `ls` affiche le contenu du répertoire courant ;
- `cd` change le répertoire courant ;
- `mv` déplace (et renomme) un fichier ou un dossier ;
- `cp` copie (et renomme) un fichier ;
- `rm` supprime un fichier ;
- `mkdir` crée un nouveau répertoire (vide) ;
- `rmdir` supprime un répertoire vide.

Par exemple, si Alice veut créer un dossier `public` dans son répertoire courant et y déplacer le fichier `doc2` en le renommant `doc3` elle écrira dans son terminal :

```
mkdir ~/public

mv ~/doc2 ~/public/doc3
```

7. Dans le cas d'un système WINDOWS, le séparateur est le caractère « \ » et les périphériques désignés par une lettre suivie du caractère « : ». Un exemple de chemin d'accès dans ce cas serait `C:home\alice\doc1`.
8. Ce sont des instructions UNIX mais elles sont aussi utilisables dans Powershell.

2.4 Droits d'accès

À chaque fichier est associé un certain nombre d'attributs précisant sa nature (fichier ordinaire ou exécutable) ainsi que les droits d'accès qui sont attribués au propriétaire et aux autres utilisateurs. Ces droits d'accès peuvent être de trois types : droit en lecture (`r` – *read*), droit en écriture (`w` – *write*), droit d'exécution (`x` – *execute*). Le droit en lecture permet de lire ce fichier mais pas de le modifier (il faut pour cela bénéficier du droit d'écriture), le droit d'exécution permet d'utiliser un programme.
Pour chaque fichier il y a trois classes d'utilisateurs : `user` (le propriétaire du fichier), `group` (le groupe auquel appartient le fichier), `other` (tous les autres). Ainsi, les droits d'accès d'un fichier sont décrits par une succession de neuf symboles ; par exemple, un fichier dont le droit d'accès est `rwxr-xr--` est accessible en lecture/écriture/exécution (`rwx`) pour le propriétaire, en lecture/exécution pour son groupe (`r-x`) et en lecture seule pour les autres (`r--`).
Chacun de ces triplets peut aussi être décrit par un nombre binaire compris entre `000` (aucun droit) et `111` (tous les droits). Converti en base 10, on obtient un nombre compris entre 0 et 7 pour décrire les huit cas de figure possible. Les droits d'accès du fichier précédent peuvent donc être décrits par le nombre 754 car $7 = (111)_2$ = `rwx`, $5 = (101)_2$ = `r-x`, $4 = (100)_2$ = `r--`.
Dans le terminal la fonction `chmod` permet de changer les droits d'accès. Par exemple, si Alice veut attribuer le droit d'accès `r-xr-x--x` au fichier `doc1` elle écrira :

```
chmod 551 ~/doc1
```

Les répertoires possèdent des droits analogues, le droit de lecture permet de visualiser le contenu de ce répertoire et le droit d'écriture d'ajouter, de renommer et de supprimer des éléments de ce répertoire, à condition de pouvoir y accéder grace au droit d'exécution. Ainsi, les droits courants pour un répertoire vont être `rwx`, `r-x` et `-wx` car il ne sert pas à grand chose de pouvoir lire ou modifier un répertoire si on ne peut y accéder.

Alice souhaite maintenant autoriser Bob à déposer des fichiers dans son répertoire `public` mais pas en voir le contenu. Il lui suffit d'écrire dans son terminal :

```
chmod 733 ~/public
```

Bob peut maintenant copier son fichier `doc1` dans le répertoire public d'Alice en écrivant dans son propre terminal :

```
cp ~/travail/doc1 ../alice/public
```

En revanche, il ne pourra pas visualiser le contenu de ce répertoire en écrivant :

```
ls ../alice/public
```

car ses droits ne sont pas suffisants.