Introduction

Les systèmes informatiques jouent un rôle de plus en plus important dans notre société. Depuis les premiers calculateurs à la fin de la seconde guerre mondiale, les ordinateurs se sont rapidement améliorés et démocratisés. Aujourd'hui, notre société est de plus en plus dépendante des systèmes informatiques.

Composition d'un système informatique

Le système informatique le plus simple est composé d'un **processeur** (**CPU** en anglais) ou unité de calcul et d'une mémoire. Le processeur est un circuit électronique qui est capable d'effectuer de nombreuses tâches :

- lire de l'information en mémoire
- écrire de l'information en mémoire
- réaliser des calculs

L'architecture des ordinateurs est basée sur l'architecture dite de Von Neumann. Suivant cette architecture, un ordinateur est composé d'un processeur qui exécute un programme se trouvant en mémoire. La mémoire contient à la fois le programme à exécuter et les données qui sont manipulées par le programme.

L'élément de base pour stocker et représenter de l'information dans un système informatique est le **bit**. Un bit (*binary digit* en anglais) peut prendre deux valeurs qui par convention sont représentées par :

- `1`
- `0`

Physiquement, un bit est représenté sous la forme d'un signal électrique ou optique lorsqu'il est transmis et d'une charge électrique ou sous forme magnétique lorsqu'il est stocké. Nous n'aborderons pas ces détails technologiques dans le cadre de ce cours. Ils font l'objet de nombreux cours d'électronique.

Le bit est l'unité de base de stockage et de transfert de l'information. En général, les systèmes informatiques ne traitent pas des bits individuellement **[1]**.

La composition de plusieurs bits donne lieu à des blocs de données qui peuvent être utiles dans différentes applications informatiques. Ainsi, un **nibble** est un bloc de 4 bits consécutifs tandis qu'un **octet** (ou **byte** en anglais) est un bloc de 8 bits consécutifs. On parlera de mots (*word* en anglais) pour des groupes comprenant généralement 32 bits et de long mot pour des groupes de 64 bits.

Le processeur et la mémoire ne sont pas les deux seuls composants d'un système informatique. Celui-ci doit également pouvoir interagir avec le monde extérieur, ne fut-ce que pour pouvoir charger le programme à exécuter et les données à analyser. Cette interaction se réalise grâce à un grand nombre de dispositifs d'entrées/sorties et de stockage. Parmi ceux-ci, on peut citer :

- le clavier qui permet à l'utilisateur d'entrer des caractères
- l'écran qui permet à l'utilisateur de visualiser le fonctionnement des programmes et les résultats qu'ils produisent
- l'imprimante qui permet à l'ordinateur d'écrire sur papier les résultats de l'exécution de programmes
- le disque-dur, les clés USB, les CDs et DVDs qui permettent de stocker les données sous la forme de fichiers et de répertoires
- la souris ou la tablette graphique qui permettent à l'utilisateur de fournir à l'ordinateur des indications de positionnement
- le scanner qui permet à l'ordinateur de transformer un document en une image numérique
- le haut-parleur avec lequel l'ordinateur peut diffuser différentes sortes de son
- le microphone et la caméra qui permettent à l'ordinateur de capturer des informations sonores et visuelles pour les stocker ou les traiter

Les systèmes informatiques peuvent prendre différentes formes, allant de minuscules systèmes embarqués à de gigantesques supercalculateurs. Les **raspberry pi** sont un exemple d'un système embarqué. Il s'agit de nano-ordinateurs, de la taille d'une carte de crédit. Possédant les mêmes composants que décrits ci-dessus, ils fonctionnent de la même façon que des systèmes plus imposants comme les ordinateurs personnels que l'on utilise au quotidien, seulement avec moins de ressources.

Unix

Unix est aujourd'hui un nom générique **[2]** correspondant à une famille de systèmes d'exploitation. La première version de Unix a été développée pour faciliter le traitement de documents sur mini-ordinateur.

Quelques variantes de Unix

De nombreuses variantes de Unix ont été produites durant les quarante dernières années. Il est impossible de les décrire toutes, mais en voici quelques unes.

- **Unix**. Initialement développé aux AT&T Bell Laboratories, Unix a été ensuite développé par d'autres entreprises. C'est aujourd'hui une marque déposée par `The Open group`, voir **http://www.unix.org/**
- **BSD Unix**. Les premières versions de Unix étaient librement distribuées par Bell Labs. Avec le temps, des variantes de Unix sont apparues. La variante développée par l'université de Berkeley en Californie a été historiquement importante car c'est dans cette variante que de nombreuses innovations ont été introduites dont notamment les piles de protocoles TCP/IP utilisés sur Internet. Aujourd'hui, **FreeBSD** et **OpenBSD** sont deux descendants de **BSD Unix**. Ils sont largement utilisés dans de nombreux serveurs et systèmes embarqués. **MacOS**, développé par Apple, s'appuie fortement sur un noyau et des utilitaires provenant de **FreeBSD**.
- **Minix** est un système d'exploitation développé initialement par **Andrew Tanenbaum** à l'université d'Amsterdam. **Minix** est fréquemment utilisé pour l'apprentissage du fonctionnement des systèmes d'exploitation.
- **Linux** est un noyau de système d'exploitation largement inspiré de **Unix** et *Minix*. Développé par **Linus Torvalds** durant ses études d'informatique, il est devenu la variante de Unix la plus utilisée à travers le monde. Il est maintenant développé par des centaines de

développeurs qui collaborent via Internet.
- **Solaris** est le nom commercial de la variante Unix de Oracle.

Dans le cadre de ce cours, nous nous focaliserons sur le système **GNU/Linux**, c'est-à-dire un système qui intègre le noyau **Linux** et les librairies et utilitaires développés par le projet **GNU** de la **FSF**.

Un système Unix est composé de trois grands types de logiciels :

- Le noyau du système d'exploitation qui est chargé automatiquement au démarrage de la machine et qui prend en charge toutes les interactions entre les logiciels et le matériel.
- De nombreuses librairies qui facilitent l'écriture et le développement d'applications
- De nombreux programmes utilitaires simples qui permettent de résoudre un grand nombre de problèmes courants. Certains de ces utilitaires sont chargés automatiquement lors du démarrage de la machine. La plupart sont exécutés uniquement à la demande des utilisateurs.

Le rôle principal du noyau du système d'exploitation est de gérer les ressources matérielles (processeur, mémoire, dispositifs d'entrées/sorties et de stockage) de façon à ce qu'elles soient accessibles à toutes les applications qui s'exécutent sur le système. Gérer les ressources matérielles nécessite d'inclure dans le systèmes d'exploitation des interfaces programmatiques (*Application Programming Interfaces* en anglais - **API**) qui facilitent leur utilisation par les applications. Les dispositifs de stockage sont une belle illustration de ce principe. Il existe de nombreux dispositifs de stockage (disque dur, clé USB, CD, DVD, mémoire flash, ...). Chacun de ces dispositifs a des caractéristiques électriques et mécaniques propres. Ils permettent en général la lecture et/ou l'écriture de blocs de données de quelques centaines d'octets. Nous reviendrons sur leur fonctionnement ultérieurement. Peu d'applications sont capables de piloter directement de tels dispositifs pour y lire ou y écrire de tels blocs de données. Par contre, la majorité des applications sont capables de les utiliser par l'intermédiaire du système de fichiers. Le système de fichiers (arborescence des fichiers) et l'API associée (**open(2)**, **close(2)**, **read(2) write(2)**) sont un exemple des services fournis par le système d'exploitation aux applications. Le système de fichiers regroupe l'ensemble des fichiers qui sont accessibles depuis un système sous une arborescence unique, quel que soit le nombre de dispositifs de stockage utilisé. La racine de cette arborescence est le répertoire `/` par convention. Ce répertoire contient généralement une dizaine de sous répertoires dont les noms varient d'une variante de Unix à l'autre. Généralement, on retrouve dans la racine les sous-répertoires suivants :

- `/usr` : sous-répertoire contenant la plupart des utilitaires et librairies installées sur le système
- `/bin` et `/sbin` : sous-répertoire contenant quelques utilitaires de base nécessaires à l'administrateur du système
- `/tmp` : sous-répertoire contenant des fichiers temporaires. Son contenu est généralement effacé au redémarrage du système.
- `/etc` : sous-répertoire contenant les fichiers de configuration du système
- `/home` : sous-répertoire contenant les répertoires personnels des utilisateurs du système
- `/dev` : sous-répertoire contenant des fichiers spéciaux
- `/root`: sous-répertoire contenant des fichiers propres à l'administrateur système. Dans certains variantes de Unix, ces fichiers sont stockés dans le répertoire racine.

Un autre service est le partage de la mémoire et du processus. La plupart des systèmes d'exploitation supportent l'exécution simultanée de plusieurs applications. Pour ce faire, le système d'exploitation partage la mémoire disponible entre les différentes applications en cours d'exécution. Il est également responsable du partage du temps d'exécution sur le ou les processeurs de façon à ce que toutes les applications en cours puissent s'exécuter.

Unix s'appuie sur la notion de processus. Une application est composée de un ou plusieurs processus. Un processus peut être défini comme un ensemble cohérent d'instructions qui utilisent une partie de la mémoire et sont exécutées sur un des processeurs du système. L'exécution d'un processus est initiée par le système d'exploitation (généralement suite à une requête faite par un autre processus). Un processus peut s'exécuter pendant une fraction de secondes, quelques secondes ou des journées entières. Pendant son exécution, le processus peut potentiellement accéder aux différentes ressources (processeurs, mémoire, dispositifs d'entrées/sorties et de stockage) du système. A la fin de son exécution, le processus se termine **[3]** et libère les ressources qui lui ont été allouées par le système d'exploitation. Sous Unix, tout processus retourne au processus qui l'avait initié le résultat de son exécution qui est résumée en un nombre entier. Cette valeur de retour est utilisée en général pour déterminer si l'exécution d'un processus s'est déroulée correctement (zéro comme valeur de retour) ou non (valeur de retour différente de zéro).

Dans le cadre de ce cours, nous aurons l'occasion de voir en détails de nombreuses librairies d'un système Unix et verrons le fonctionnement d'appels systèmes qui permettent aux logiciels d'interagir directement avec le noyau. Le système Unix étant majoritairement écrit en langage C, ce langage est le langage de choix pour de nombreuses applications. Nous le verrons donc en détails.

Pour vous permettre de mettre vos apprentissages en pratique, vous recevrez durant le quadrimestre un **raspberry pi**. Il est possible d'installer différents systèmes d'exploitation sur celui-ci. Nous utiliserons **raspbian** qui est lui aussi une variante de Unix.

Utilitaires

Unix a été conçu à l'époque des mini-ordinateurs. Un mini-ordinateur servait plusieurs utilisateurs en même temps. Ceux-ci y étaient connectés par l'intermédiaire d'un terminal équipé d'un écran et d'un clavier. Les programmes traitaient les données entrées par l'utilisateur via le clavier ou stockées sur le disque. Les résultats de l'exécution des ces programmes étaient affichés à l'écran, sauvegardés sur disque ou parfois imprimés sur papier. Le fonctionnement de nombreux utilitaires Unix a été influencé par cet environnement. A tout processus Unix, on associe :

- une entrée standard (**stdin** en anglais) qui est un flux d'informations par lequel le processus reçoit les données à traiter. Par défaut, l'entrée standard est associée au clavier.
- une sortie standard (**stdout** en anglais) qui est un flux d'informations sur lequel le processus écrit le résultat de son traitement. Par défaut, la sortie standard est associée au terminal.
- une sortie d'erreur standard (**stderr** en anglais) qui est un flux de données sur lequel le processus écrira les messages d'erreur éventuels. Par défaut, la sortie d'erreur standard est associée à l'écran.

Unix ayant été initialement développé pour manipuler des documents contenant du texte, il comprend de nombreux utilitaires facilitant ces traitements. Une description détaillée de l'ensemble de ces utilitaires sort du cadre de ce cours. De nombreux livres et ressources Internet fournissent une description détaillée. Voici cependant une brève présentation de quelques utilitaires de manipulation de texte qui peuvent s'avérer très utiles en pratique.

- **cat(1)** : utilitaire permettant notamment d'afficher le contenu d'un fichier sur la sortie standard
- **echo(1)** : utilitaire permettant d'afficher sur la sortie standard une chaîne de caractères passée en argument
- **head(1)** et **tail(1)** : utilitaires permettant respectivement d'extraire le début ou la fin d'un fichier
- **wc(1)** : utilitaire permettant de compter le nombre de caractères et de lignes d'un fichier
- **grep(1)** : utilitaire permettant notamment d'extraire d'un fichier texte les lignes qui contiennent ou ne contiennent pas une chaîne de caractères passée en argument
- **sort(1)** : utilitaire permettant de trier les lignes d'un fichier texte
- **uniq(1)** : utilitaire permettant de filtrer le contenu d'un fichier texte afin d'en extraire les lignes qui sont uniques ou dupliquées (requiert que le fichier d'entrée soit trié, car ne compare que les lignes consécutives)
- **more(1)** : utilitaire permettant d'afficher page par page un fichier texte sur la sortie standard (**less(1)** est une variante courante de **more(1)**)
- **gzip(1)** et **gunzip(1)** : utilitaires permettant respectivement de compresser et de décompresser des fichiers. Les fichiers compressés prennent moins de place sur le disque que les fichiers standard et ont par convention un nom qui se termine par `.gz`.
- **tar(1)** : utilitaire permettant de regrouper plusieurs fichiers dans une archive. Souvent utilisé en combinaison avec **gzip(1)** pour réaliser des backups ou distribuer des logiciels.
- **sed(1)** : utilitaire permettant d'éditer, c'est-à-dire de modifier les caractères présents dans un flux de données.
- **awk(1)** : utilitaire incluant un petit langage de programmation et qui permet d'écrire rapidement de nombreux programmes de manipulation de fichiers textes

La plupart des utilitaires fournis avec un système Unix ont été conçus pour être utilisés en combinaison avec d'autres. Cette combinaison efficace de plusieurs petits utilitaires est un des points forts des systèmes Unix par rapport à d'autres systèmes d'exploitation.

Shell

Avant le développement des interfaces graphiques telles que **X11**, **Gnome** ou **Aqua**, l'utilisateur interagissait exclusivement avec l'ordinateur par l'intermédiaire d'un interpréteur de commandes. Dans le monde Unix, le terme anglais **shell** est le plus souvent utilisé pour désigner cet interpréteur et nous ferons de même. Avec les interfaces graphiques actuelles, le shell est accessible par l'intermédiaire d'une application qui est généralement appelée `terminal` ou `console`.

Un **shell** est un programme qui a été spécialement conçu pour faciliter l'utilisation d'un système Unix via le clavier. De nombreux shells Unix existent. Les plus simples permettent à l'utilisateur de taper une série de commandes à exécuter en les combinant. Les plus avancés sont des interpréteurs de commandes qui supportent un langage complet permettant le développement de scripts plus ou moins ambitieux. Dans le cadre de ce cours, nous utiliserons **bash(1)** qui est un des shells les plus populaires et les plus complets. La plupart des commandes **bash(1)** que nous utiliserons sont cependant compatibles avec de nombreux autres shells tels que **zsh** ou **csh**.

Lorsqu'un utilisateur se connecte à un système Unix, en direct ou à travers une connexion réseau, le système vérifie son mot de passe puis exécute automatiquement le shell qui est associé à cet utilisateur depuis son répertoire par défaut. Ce shell permet à l'utilisateur d'exécuter et de combiner des commandes. Un shell supporte deux types de commande : les commandes internes qu'il implémente directement et les commandes externes qui font appel à un utilitaire stocké sur disque. Les utilitaires présentés dans la section précédente sont des exemples de commandes externes. Voici quelques exemples d'utilisation de commandes externes.

```
$ cat exemple.txt
Un simple fichier de textes
aaaaaaaaaa bbbbbb
bbbbb ccccccccccc
eeeee fffffffff
aaaaaaaaaa bbbbbb
$ grep fichier exemple.txt
Un simple fichier de textes
$ wc exemple.txt
       5      13      98 exemple.txt
```

La puissance du **shell** ne vient pas de sa capacité d'exécuter des commandes individuelles telles que ci-dessus. Elle vient de la possibilité de combiner ces commandes en redirigeant les entrées et sorties standards. Les shells Unix supportent différentes formes de redirection. Tout d'abord, il est possible de forcer un programme à lire son entrée standard depuis un fichier plutôt que depuis le clavier. Cela se fait en ajoutant à la fin de la ligne de commande le caractère < suivi du nom du fichier à lire. Ensuite, il est possible de rediriger la sortie standard vers un fichier. Cela se fait en utilisant > ou >>. Lorsqu'une commande est suivie de > `file`, le fichier `file` est créé si il n'existait pas et remis à zéro si il existait et la sortie standard de cette commande est redirigée vers le fichier `file`. Lorsqu'un commande est suivie de >> `file`, la sortie standard est sauvegardée à la fin du fichier `file` (si `file` n'existait pas, il est créé). Des informations plus complètes sur les mécanismes de redirection de **bash(1)** peuvent être obtenues dans le **chapitre 20** de **[ABS]**.

```
$ echo "Un petit fichier de textes" > file.txt
$ echo "aaaaa bbbbb" >> file.txt
$ echo "bbbb ccc" >> file.txt
$ grep -v bbbb < file.txt > file.out
$ cat file.out
Un petit fichier de textes
```

Les shells Unix supportent un second mécanisme qui est encore plus intéressant pour combiner plusieurs programmes. Il s'agit de la redirection de la sortie standard d'un programme vers l'entrée standard d'un autre sans passer par un fichier intermédiaire. Cela se réalise avec le symbole | (**pipe** en anglais). L'exemple

suivant illustre quelques combinaisons d'utilitaires de manipulation de texte.

```
$ echo "Un petit texte" | wc -c
      15
$ echo "bbbb ccc" >> file.txt
$ echo "aaaaa bbbbb" >> file.txt
$ echo "bbbb ccc" >> file.txt
$ cat file.txt
bbbb ccc
aaaaa bbbbb
bbbb ccc
$ cat file.txt | sort | uniq
aaaaa bbbbb
bbbb ccc
```

Le premier exemple est d'utiliser **echo(1)** pour générer du texte et le passer directement à **wc(1)** qui compte le nombre de caractères. Le deuxième exemple utilise **cat(1)** pour afficher sur la sortie standard le contenu d'un fichier. Cette sortie est reliée à **sort(1)** qui trie le texte reçu sur son entrée standard en ordre alphabétique croissant. Cette sortie en ordre alphabétique est reliée à **uniq(1)** qui la filtre pour en retirer les lignes dupliquées.

Tout shell Unix peut également s'utiliser comme un interpréteur de commande qui permet d'interpréter des scripts. Un système Unix peut exécuter deux types de programmes :

- des programmes exécutables en langage machine. C'est le cas de la plupart des utilitaires dont nous avons parlé jusqu'ici.
- des programmes écrits dans un langage interprété. C'est le cas des programmes écrits pour le shell, mais également pour d'autres langages interprétés comme **python** ou **perl**.

Lors de l'exécution d'un programme, le système d'exploitation reconnaît **[4]** si il s'agit d'un programme directement exécutable ou d'un programme interprété en analysant les premiers octets du fichier. Par convention, sous Unix, les deux premiers caractères d'un programme écrit dans un langage qui doit être interprété sont `#!`. Ils sont suivis par le nom complet de l'interpréteur qui doit être utilisé pour interpréter le programme.

Le programme **bash(1)** le plus simple est le suivant :

```
#!/bin/bash
echo "Hello, world"
```

L'exécution de ce script shell retourne la sortie suivante :

```
Hello, world
```

Par convention en **bash(1)**, le caractère `#` marque le début d'un commentaire en début ou en cours de ligne. Comme tout langage, **bash(1)** permet à l'utilisateur de définir des variables. Celles-ci peuvent contenir des chaînes de caractères ou des nombres. Le script ci-dessous utilise deux variables, `PROG` et `COURS` et les utilise pour afficher un texte avec la commande `echo`.

```
#!/bin/bash
PROG="LEPL"
COURS=1503
echo $PROG$COURS
```

Un script **bash(1)** peut également prendre des arguments passés en ligne de commande. Par convention, ceux-ci ont comme noms `$1`, `$2`, `$3`, ... L'exemple ci-dessous illustre l'utilisation de ces arguments.

```
#!/bin/bash
# $# nombre d'arguments
# $1 $2 $3 ... arguments
echo "Vous avez passe" $# "arguments"
echo "Le premier argument est :" $1
echo "Liste des arguments :" $@
```

L'exécution de ce script produit la sortie suivante :

```
Vous avez passe 2 arguments
Le premier argument est : LEPL
Liste des arguments : LEPL 1503
```

Concernant le traitement des arguments par un script bash, il est utile de noter que lorsque l'on appelle un script en redirigeant son entrée ou sa sortie standard, le script n'est pas informé de cette redirection. Ainsi, si l'on exécute le script précédent en faisant `args.sh arg1 > args.out`, le fichier `args.out` contient les lignes suivantes :

```
Vous avez passe 2 arguments
Le premier argument est : LEPL
Liste des arguments : LEPL 1503
```

bash(1) supporte la construction `if [ condition ]; then ... fi` qui permet notamment de comparer les valeurs de variables. **bash(1)** définit de nombreuses conditions différentes, dont notamment :

- `$i -eq $j` est vraie lorsque les deux variables `$i` et `$j` contiennent le même nombre.
- `$i -lt $j` est vraie lorsque la valeur de la variable `$i` est numériquement strictement inférieure à celle de la variable `$j`
- `$i -ge $j` est vraie lorsque la valeur de la variable `$i` est numériquement supérieure ou égale à celle de la variable `$j`
- `$s = $t` est vraie lorsque la chaîne de caractères contenue dans la variable `$s` est égale à celle qui est contenue dans la variable `$t`
- `-z $s` est vraie lorsque la chaîne de caractères contenue dans la variable `$s` est vide

D'autres types de test sont définis dans la page de manuel : **bash(1)**. Le script ci-dessous fournit un premier exemple d'utilisation de tests avec **bash(1)**.

```
#!/bin/bash
# Vérifie si les deux nombres passés en arguments sont égaux
if [ $# -ne 2 ]; then
    echo "Erreur, deux arguments sont nécessaires" > /dev/stderr
    exit 2
fi
if [ $1 -eq $2 ]; then
        echo "Nombres égaux"
else
        echo "Nombres différents"
fi
exit 0
```

Tout d'abord, ce script vérifie qu'il a bien été appelé avec deux arguments. Vérifier qu'un programme reçoit bien les arguments qu'il attend est une règle de bonne pratique qu'il est bon de respecter dès le début. Si le script n'est pas appelé avec le bon nombre d'arguments, un message d'erreur est affiché sur la sortie d'erreur standard et le script se termine avec un code de retour. Ces codes de retour sont importants car ils permettent à un autre programme, par exemple un autre script **bash(1)** de vérifier le bon déroulement d'un programme appelé. Le script **`src/eq.sh`** utilise des appels explicites à **exit(1posix)** même si par défaut, un script **bash(1)** qui n'en contient pas retourne un code de retour nul à la fin de son exécution.

Un autre exemple d'utilisation des codes de retour est le script **`src/wordin.sh`** repris ci-dessous qui utilise **grep(1)** pour déterminer si un mot passé en argument est présent dans un fichier texte. Pour cela, il exploite la variable spéciale `$?` dans laquelle **bash(1)** sauve le code de retour du dernier programme exécuté par le script.

```
#!/bin/bash
# wordin.sh
# Vérifie si le mot passé en premier argument est présent
# dans le fichier passé comme second argument
if [ $# -ne 2 ]; then
    echo "Erreur, deux arguments sont nécessaires" > /dev/stderr
    exit 2
fi
grep $1 $2 >/dev/null
# $? contient la valeur de retour de grep
if [ $? -eq 0 ]; then
        echo "Présent"
        exit 0
else
        echo "Absent"
        exit 1
fi
```

Ce programme utilise le fichier spécial `/dev/null`. Celui-ci est en pratique l'équivalent d'un trou noir. Il accepte toutes les données en écriture mais celles-ci ne peuvent jamais être relues. `/dev/null` est très utile lorsque l'on veut ignorer la sortie d'un programme et éviter qu'elle ne s'affiche sur le terminal. **bash(1)** supporte également `/dev/stdin` pour représenter l'entrée standard, `/dev/stdout` pour la sortie standard et `/dev/stderr` pour l'erreur standard.

Une description complète de **bash(1)** sort du cadre de ce cours. De nombreuses références à ce sujet sont disponibles **[Cooper2011]**.

Footnotes

[1] Dans certaines applications, par exemple dans les réseaux informatiques, il peut être utile d'accéder à la valeur d'un bit particulier qui joue par exemple le rôle d'un drapeau. Celui-ci se trouve cependant généralement à l'intérieur d'une structure de données comprenant un ensemble de bits.

[2] Formellement, Unix est une marque déposée par l'**Open Group**, un ensemble d'entreprises qui développent des standards dans le monde de l'informatique. La première version de Unix écrite en C date de 1973, **http://www.unix.org/what_is_unix/history_timeline.html**

[3] Certains processus sont lancés automatiquement au démarrage du système et ne se terminent qu'à son arrêt. Ces processus sont souvent appelés des *daemons*. Il peut s'agir de services qui fonctionnent en permanence sur la machine, comme par exemple un serveur web ou un *daemon* d'authentification.

[4] Sous Unix et contrairement à d'autres systèmes d'exploitation, le suffixe d'un nom de fichier ne joue pas de rôle particulier pour indiquer si un fichier contient un programme exécutable ou non. Comme nous le verrons ultérieurement, le système de fichiers Unix contient des bits de permission qui indiquent notamment si un fichier est exécutable ou non.