La programmation système en Python

Arnaud Calmettes (nohar)

30 avril 2015

Table des matières

Avant-propos

Savez-vous comment fonctionne votre système d'exploitation ?

Par exemple, savez-vous de quelle façon les programmes qui tournent sur votre ordinateur sont capables de communiquer entre eux ? Connaissez-vous la différence entre un *thread* et un processus ? Ou même sans aller jusque là, avez-vous une idée de ce qu'il se passe *réellement* lorsque vous ouvrez un fichier pour le lire ?

Si ce n'est pas le cas, **comment voulez-vous comprendre ce que font vraiment les programmes que vous écrivez ?**

Même si cela ne saute pas aux yeux, la *programmation système* n'est pas seulement utile aux barbus qui bidouillent leur Linux dans une cave. Elle permet à n'importe quel développeur d'avoir une bonne intuition de l'efficacité de ses programmes, et de comprendre le fonctionnement des mécanismes de son système d'exploitation.

Contrairement à tous les autres cours que vous pourrez trouver sur la programmation système, celui-ci n'utilisera pas directement le langage C, mais *Python*. Son but n'est pas de vous livrer un savoir encyclopédique (il existe déjà des kilomètres de documentation pour ça), mais plutôt de découvrir toutes ces notions *utiles*, dont certaines que vous avez sûrement déjà utilisées dans vos programmes sans forcément le savoir ni les comprendre.

Ainsi, l'objectif de ce cours est de compléter votre culture informatique avec des notions qui feront de vous un meilleur développeur.

Inévitablement, on ne peut pas faire de programmation système sans se concentrer sur un système d'exploitation particulier. Dans ce cours, nous nous pencherons de près sur le fonctionnement des systèmes qui suivent la norme POSIX, parmi lesquels on trouvera la famille des UNIX (comme BSD ou Mac OS X) ainsi que GNU/Linux.

Pour suivre ce cours, vous devrez :

— Connaître les bases de la programmation en Python. Nous prendrons Python 3.4 comme version de référence.
— Disposer d'un ordinateur (ou d'un Raspberry Pi, ou même d'une machine virtuelle) sous GNU/Linux (ou l'un des descendants d'UNIX).
— Ne pas prendre peur à la vue d'une ligne de commande.

Alors, êtes-vous prêts à ne plus être un *utilisateur lambda* de votre OS ? La curiosité de percer ses petits secrets vous a-t'elle piqué au vif ? Ne frissonnez-vous pas de plaisir à l'idée de mettre les mains dans le cambouis ? Suivez-moi, et vous ne serez pas déçus !

1 Le noyau et la coquille

Commençons par le commencement. Dans ce chapitre, nous allons survoler rapidement la façon dont sont structurées les couches basses de votre système d'exploitation. Il s'agit principalement d'un rappel des notions sur lesquelles nous construirons les chapitres suivants.

1.1 Au plus proche du matériel : le noyau

Rassurez-vous, je vous ferai grâce d'un long couplet sur l'architecture des micro-processeurs. Ce n'est pas ce qui nous intéresse dans ce cours. Simplement, vous le savez comme moi, un ordinateur, c'est d'abord tout un tas de composants matériels soudés sur des cartes et connectés entre eux par des bus de données. Et puis n'oublions pas tous les périphériques reliés à ces cartes par des câbles. Bref, un ordinateur, c'est d'abord et avant tout *du matériel*.

En fait, *tous* les programmes que vous exécutez vont forcément, à un moment donné, devoir intéragir avec ce matériel. Au minimum, le microprocesseur devra exécuter les instructions du programme en allant piocher des données dans la mémoire vive (ou RAM). Mais pour qu'un programme serve réllemment à quelque chose, il faut bien qu'il ait des **entrées** et des **sorties**.

Par exemple, vous allez peut-être saisir des données sur votre *clavier*, ou bien utiliser une *souris* pour cliquer sur des boutons. Et pour voir ce que vous êtes en train de faire, ce même programme devra afficher quelque chose sur un *écran*. Peut-être même qu'il ira lire ou écrire des fichiers sur un *disque dur*, ou encore se connecter à internet pour échanger des données avec d'autres programmes qui tournent sur d'autres ordinateurs, en passant par *la carte réseau*...

Vous imaginez bien que tout cela ne se fait pas par magie. Pour qu'un programme utilise un périphérique matériel, **il doit forcément faire appel à du code qui sert à manipuler ce matériel.**

J'entends quelqu'un murmurer le terme "pilote", là bas au fond. J'y viens, patience !

1.1.1 Interagir avec le matériel. Oui, mais...

Imaginons un instant que vous ayez écrit un programme capable de communiquer directement avec l'un de vos périphériques. Par exemple votre disque dur. Vous avez mis quelques bonnes semaines et englouti des douzaines de cafetières, vous avez plongé votre nez dans la documentation technique du matériel et failli vous décourager une bonne vingtaine de fois, mais ça y est, votre programme sait lire et sauvegarder des données au bon endroit sur votre disque dur et il fonctionne à merveille. Ouf !

Vous avez donc quelque chose de semblable à la figure 1.1.

Imaginons maintenant que vous vouliez créer *un autre* programme qui ait besoin d'interagir avec ce disque dur. Manque de chance, le code de votre premier programme ne sait écrire que des fichiers de type `A` sur le disque, mais le second a besoin de lire des fichiers de type `B`. Vous voilà obligé de tout recommencer : les quelques semaines, la cafetière, la documentation technique... Sauf que cette fois-ci,

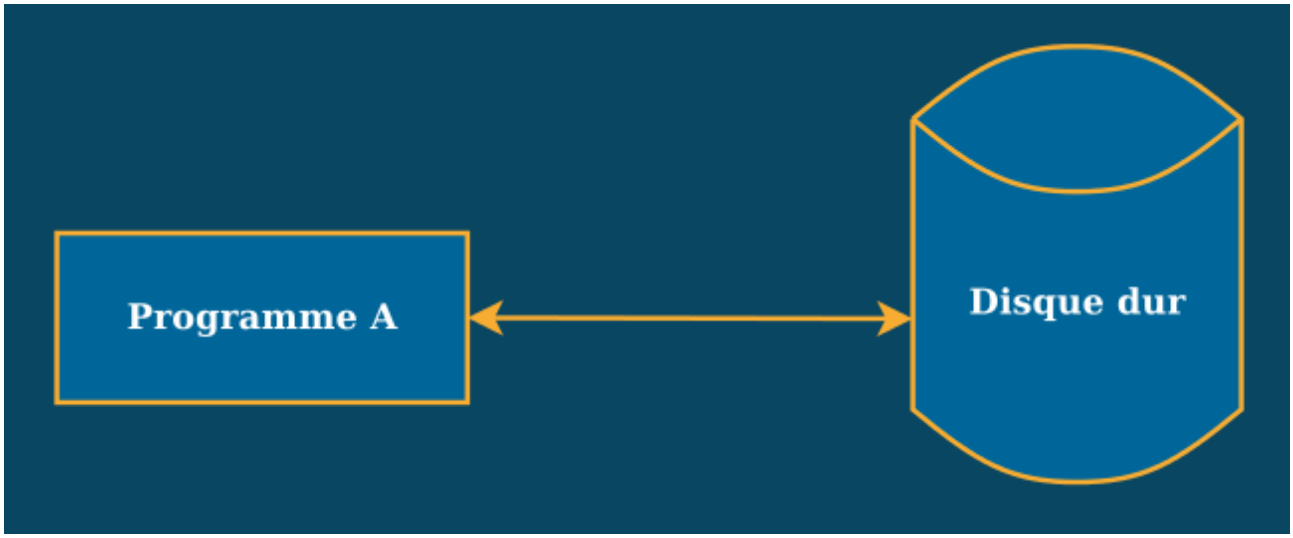

FIGURE 1.1 – Interaction directe avec le matériel

vous ne vous laisserez pas avoir ! Vous allez écrire une *bibliothèque de fonctions* qui sera capable de lire et d'écrire n'importe quelle donnée sur votre disque dur. De cette manière, les programmes n'auront qu'à mettre en forme leurs données selon le type de fichier `A` ou `B`, et laisser le soin à votre bibliothèque de les charger au bon endroit sur le disque. Cette bibliothèque, c'est ce que l'on peut appeler un **pilote de périphérique** (on parle également de *pilote* tout court, ou bien de *driver* en anglais).

Vous obtenez alors une architecture similaire à la figure 1.2.

Vous pouvez être fier de vous ! Vous avez partagé le code de votre pilote avec vos amis qui désirent interagir avec ce disque dur : ils n'ont même pas besoin de mettre le nez dans la documentation technique, ils n'ont plus qu'à se concentrer sur leurs programmes à eux.

Tout va bien dans le meilleur des mondes, les gens utilisent votre pilote pour lire et écrire des programmes sur le disque et celui-ci fonctionne parfaitement. En fait, il fonctionne tellement bien qu'un beau jour, au milieu de l'été, *le disque est rempli à ras bord de données*. Qu'à cela ne tienne, vous direz-vous, il suffit d'aller en acheter un deuxième. Et puis comme il y a une promo dans votre magazin, vous en trouvez un deux fois plus grand, deux fois plus rapide, deux fois plus joli et deux fois moins cher, une affaire en or ! Satisfait, vous rentrez chez vous en toute hâte, puis déballez ce petit bijou pour l'installer sur votre ordinateur, mais... il s'avère que votre nouveau disque dur n'est pas compatible avec le pilote que vous aviez développé pour le premier. La poisse !

Par chance, ce nouveau disque dur est livré avec une disquette [1] d'installation qui contient son pilote. Voilà qui vous fera gagner du temps. Mais le problème c'est que les programmes que vous avez écrits jusqu'à présent savent utiliser le pilote de votre premier disque dur, mais pas celui-ci, qui n'a absolument rien à voir.

Résigné, vous vous laissez choir dans votre fauteuil : tout est à refaire. Décidément, **interagir directement avec le matériel, c'est un véritable cauchemar**. Quand ce n'est pas le code de pilotage du disque qui est mélangé aux programmes, ce sont les programmes qui ne sont pas compatibles avec les autres pilotes de disque et qu'il faut modifier chaque fois qu'un nouveau modèle arrive sur le marché.

1. Ah oui, j'avais oublié de le préciser, mais nous sommes au début des années 1970. Vous n'imaginiez pas que quelqu'un développerait son propre pilote de disque dur de nos jours, tout de même !

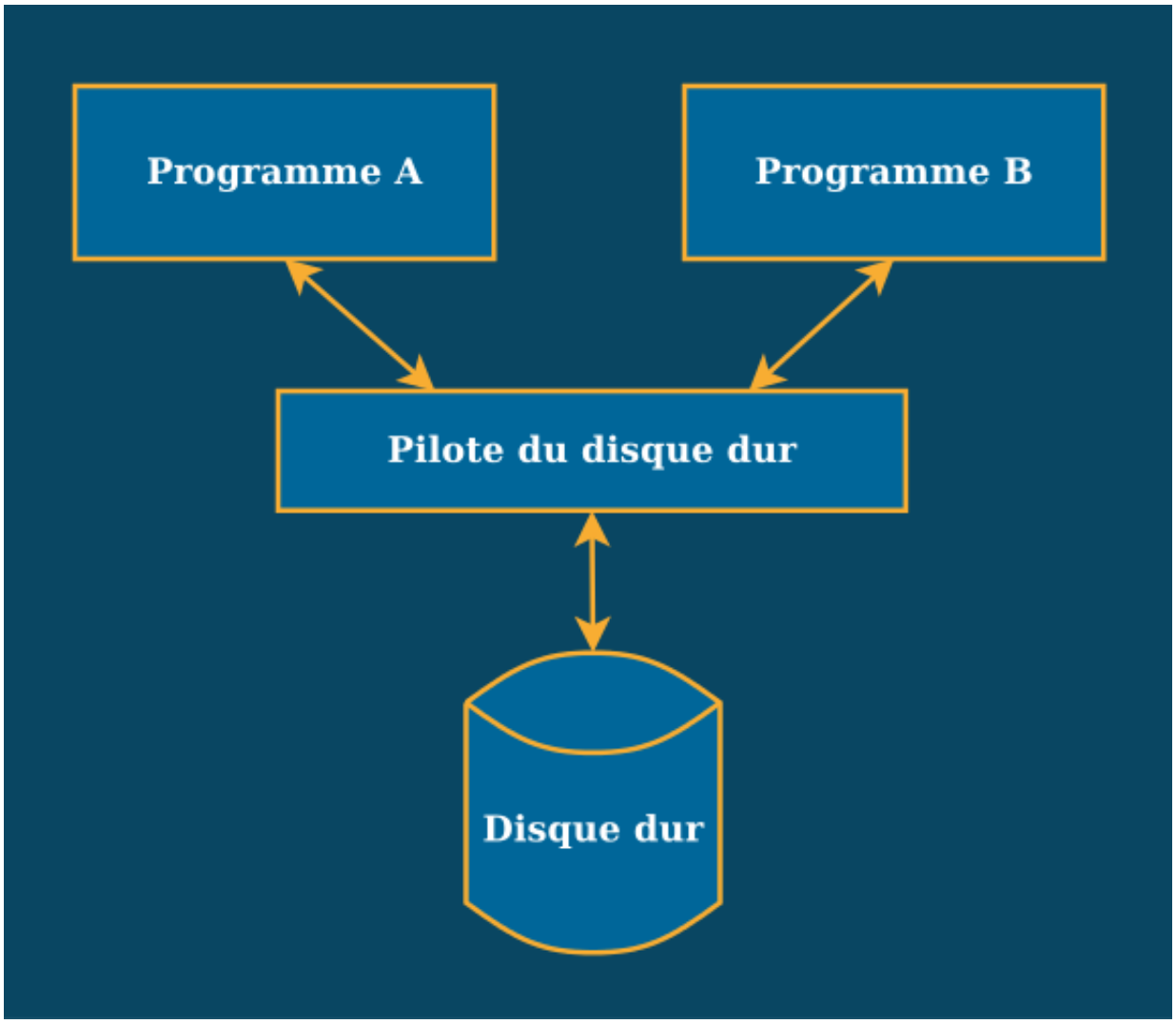

FIGURE 1.2 – Utilisation d'un pilote

Même avec la meilleure volonté du monde, ce n'est pas gérable. En fait, de nos jours, les programmes n'interagissent jamais directement avec le matériel, sauf à la rigueur sur certains systèmes embarqués.

1.1.2 Un noyau pour les gouverner tous, et dans l'abstraction les lier

Vous l'avez compris : un système d'exploitation digne de ce nom doit fournir aux programmes une interface commune, de façon à ce que ceux-ci n'aient jamais besoin de se soucier du matériel sur lequel ils agissent. Cette interface, on l'appelle *HAL* pour *Hardware Abstraction Layer* (*couche d'abstraction matérielle*). C'est cette couche qui s'occupe de charger et d'utiliser le bon pilote de périphérique en fonction du matériel qui est effectivement branché à l'ordinateur.

Néanmoins, cela n'est pas encore tout à fait suffisant. En effet, c'est bien joli de fournir aux développeurs une interface commune pour accéder au matériel, mais encore faut-il penser à *protéger* l'ordinateur contre le code qui fait n'importe quoi.

Imaginons qu'un premier programme demande à HAL de sauvegarder des données sur le disque dur. HAL va faire bien attention à trouver une adresse libre sur le matériel pour écrire ces données. Maintenant, devinez ce qu'il se passe si un second programme ne demande pas l'avis de HAL, et accède directement au disque dur pour écrire des données *à la même adresse...*

C'est pour cela que les systèmes d'exploitation modernes disposent d'un **noyau** (ou kernel) qui :

- **protège** le matériel en étant *le seul* autorisé à interagir avec lui,
- fournit aux programmes qui tournent sur la machine une **interface** leur permettant d'utiliser le matériel de façon contrôlée.

Ainsi, le système d'exploitation va faire la distinction entre deux types de codes :

- le code *utilisateur* (ou *non-privilégié*), qui n'a pas le droit d'accéder au matériel,
- le code *noyau* (ou *privilégié*) qui a tous les droits.

Lorsqu'un programme utilisateur a besoin d'exécuter du code privilégié, celui-ci va réaliser un **appel système**. Lors de cet appel système, le programme va passer en mode privilégié et exécuter le code correspondant puis, au retour de la fonction, il repassera automatiquement en mode utilisateur.

Ce fonctionnement est résumé sur la figure 1.3.

Les appels systèmes en eux-mêmes sont réalisés au moyen d'une interruption matérielle. En somme, leur nature et la façon dont ils sont réalisés dépend à la fois de la machine et du système d'exploitation.

1.1.3 Appels systèmes sous Unix

La famille Unix est la plus grande famille de systèmes d'exploitation qui existe à ce jour. Ces systèmes ont tous un fonctionnement commun, qui reproduit celui du système UNIX développé par AT&T durant les années 1970.

Dans cette famille de systèmes, on trouvera notamment GNU/Linux [2] et OpenBSD, qui sont des systèmes d'exploitation libres, mais également Mac OS X, qui peuple les ordinateurs de la marque Apple, ainsi qu'une myriade d'autres systèmes moins connus tels que Minix, Plan9 ou GNU/Hurd.

2. En toute rigueur, GNU/Linux est une réécriture complète d'UNIX, mais ça ne l'empêche pas de fonctionner de la même façon et de suivre en très grande partie les mêmes standards. C'est la raison pour laquelle on le place très volontiers dans la famille des systèmes Unix.

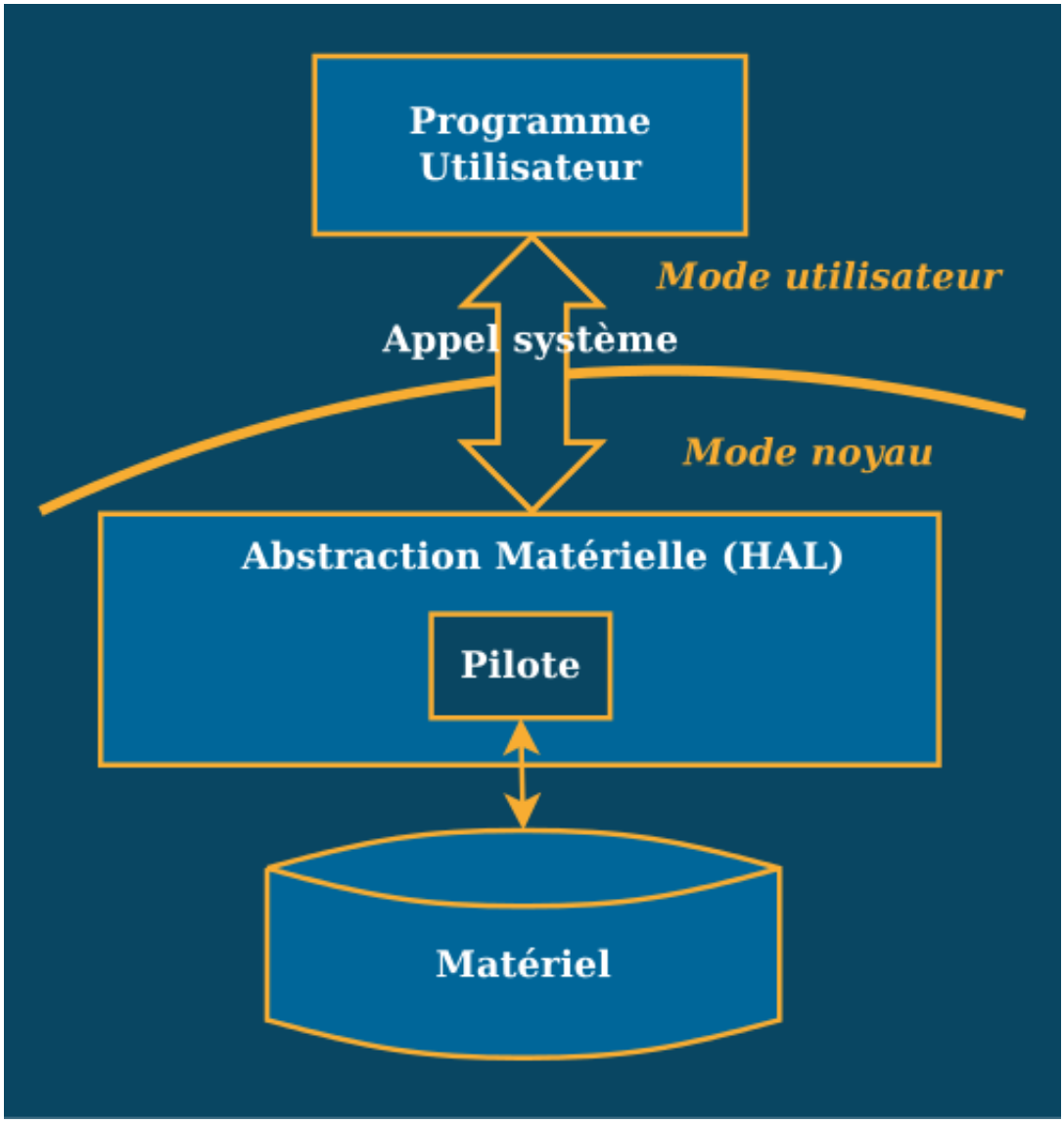

FIGURE 1.3 – Accès au matériel sur un système moderne

Les systèmes Unix proposent tous, dans leur bibliothèque C standard (que l'on appelle la `libc`), des fonctions permettant de réaliser des appels système.

La plus basique d'entre elles est la fonction `syscall()` dont la signature est la suivante :

```
int syscall (int number, ...)
```

En somme, pour réaliser un appel système depuis un programme C, il suffit de passer à cette fonction le *numéro* de l'appel système en question, ainsi que les arguments attendus par celui-ci (si tel est le cas), et celle-ci retournera un entier.

Essayons d'utiliser cette fonction depuis Python. Pour commencer, sachez qu'il est possible de charger la `libc` et d'en utiliser les fonctions directement depuis Python, en nous servant du module standard `ctypes`. Sous GNU/Linux, la `libc` sera nommée `libc.so.6`. Chargeons-la :

```
>>> import ctypes
>>> libc = ctypes.CDLL('libc.so.6')
>>> libc.syscall
<_FuncPtr object at 0x7f634a29a5c0>
```

La dernière ligne nous indique que Python a bel et bien accès à la fonction standard `syscall`. Il nous suffit maintenant de trouver un numéro d'appel système à tester. Prenons-en un qui n'a besoin d'aucun argument. Par exemple, récupérons le numéro du processus dans lequel notre console Python est en train de tourner au moyen de l'appel système `getpid`. Cet appel système correspond sur ma machine (Linux 3.13 sur une architecture `x86_64`) au numéro 39.

Essayons !

```
>>> libc.syscall(39)
7827
```

Cela semble avoir fonctionné, mais comment en être certain ?

1.1.4 Les *stubs* de la `libc` et le module standard `os` de Python

En fait, dans la plupart des cas, nous n'aurons jamais besoin d'utiliser cette fonction `syscall()` directement. Déjà, sachez que la `libc` définit un certain nombre de raccourcis, appelés *stubs*, pour la plupart des appels systèmes courants. Pour en revenir à notre exemple, il existe effectivement une fonction standard `getpid()` en C, qui réalise ce même appel système :

```
>>> libc.getpid()
7827
```

Avouez que c'est quand même plus pratique et portable que d'appeler un numéro arbitraire.

Maintenant, sachez que ces *stubs* sont également portés en Python. On en trouvera un très grand nombre, par exemple, dans le module `os` *dont c'est précisément le rôle*[3]. Ainsi, au lieu de charger la libc comme des barbares, nous aurions tout simplement pu utiliser la fonction `getpid()` de Python, qui réalise l'appel système du même nom !

3. Je parie que c'est la première fois que l'on vous présente le module `os` pour ce qu'il est vraiment : une collection d'appels systèmes.

```
>>> import os
>>> os.getpid()
7827
```

Constatez que nous obtenons bien le même résultat, que nous utilisions la fonction `syscall()`, le *stub* `getpid()` de la `libc` ou la fonction `os.getpid()` de la bibliothèque standard Python.

Gardez cependant cette fonction `syscall()` dans un coin de votre mémoire. Il nous arrivera peut-être un jour (ou dans un futur chapitre de ce cours) de devoir nous rappatrier sur cette fonction pour réaliser un appel système dont il n'existe ni fonction équivalente dans le module `os`, ni *stub* dans la `libc`. Dans ce cas, nous serons bien contents de pouvoir recourir à cette solution.

1.2 Le shell Unix

Un système d'exploitation, ce n'est pas seulement un noyau. C'est d'ailleurs pour cette raison que l'on ne devrait jamais parler de « Linux » tout seul lorsque l'on désigne le système d'exploitation GNU/Linux : pour qu'un système soit utilisable, il faut envelopper le noyau d'une couche logicielle qui sert *d'interface* entre celui-ci et l'utilisateur. Cette interface, dans le vocabulaire Unix, porte le nom de *shell* (comme le mot anglais qui désigne une *coquille*).

La notion de *shell*, en raison de nombreux abus de langages, est finalement assez vague, car elle prend un sens différent selon le contexte dans lequel on y fait référence. Dans le contexte d'un système d'exploitation, le shell est l'interface qui enveloppe le noyau. Cette interface peut être *graphique* ou en *ligne de commande*, tant qu'elle désigne la couche logicielle qui permet à l'utilisateur d'interagir avec son OS. On s'accordera toutefois à dire qu'un *shell Unix* désigne bel et bien une interface en ligne de commande, puisque la norme POSIX (qui sert à uniformiser le fonctionnement des systèmes Unix) définit le format et le comportement d'un certain nombre de commandes standard.

En revanche, il existe plusieurs *shells* qui permettent tous de travailler avec ces mêmes commandes standard. Le plus répandu à l'heure actuelle est `bash` (pour *Bourne-Again SHell*), puisque c'est celui qui vient par défaut avec les distributions de GNU/Linux et Mac OS X.

À la rigueur, pour suivre ce cours, vous pouvez bien utiliser le shell que vous voudrez. Personnellement, j'utilise `zsh`, mais toutes les lignes de commande que nous taperons dans ce cours seront compatibles avec `bash`. De toute manière, dès que nous aurons besoin de réaliser des choses élaborées, nous utiliserons Python.

Ce cours ne portant pas sur le shell mais sur la programmation système, toutes les commandes utilisées ne seront pas nécessairement détaillées. Le lecteur est invité à découvrir la fonction de celles qu'il ne connaît pas en explorant de lui-même, soit en affichant les messages d'aide (via `<cmd> --help`), soit en se retournant vers le manuel standard (`man <cmd>`).

1.2.1 Conventions de notation

Dans toute la suite de ce cours, nous distinguerons trois notations particulières pour différencier les lignes de commande. Les commandes à taper dans une console Python seront précédées, comme dans l'interpréteur standard, par trois chevrons fermants (`>>>`) :

```
>>> print("Hello, World!")
Hello, World!
```

Les commandes *shell* sous un utilisateur quelconque seront précédées d'un symbole *dollar* (`$`) :

```
$ whoami
arnaud
```

Les commandes *shell* tapées sous l'utilisateur spécial `root` seront précédées d'un symbole *pourcent* (`%`) :

```
% whoami
root
```

1.2.2 Utilisateurs et groupes

Tous les utilisateurs d'un système Unix sont identifiés par un numéro d'utilisateur unique : leur UID (User IDentifier).

Par exemple, sur ma machine, je dispose de l'UID 1000 :

```
$ id -u
1000
```

En fait, cet UID unique est associé à un *nom d'utilisateur* dans une base de données spéciale, ce qui est beaucoup plus facile à manipuler pour nous autres, simples mortels :

```
$ id -un
arnaud
```

Par ailleurs, chaque utilisateur fait partie d'un ou plusieurs **groupes**. La plupart des utilisateurs ont un *groupe primaire* qui leur est propre (leur GID), et appartiennent possiblement à plusieurs *groupes secondaires*. De la même manière que les utilisateurs, les groupes sont désignés par un numéro de groupe unique, et associés à un nom de groupe dans une base de données. Ces groupes servent à identifier les utilisateurs qui partagent des permissions communes (nous en reparlerons plus tard).

```
$ id
uid=1000(arnaud) gid=1000(arnaud) groups=1000(arnaud),4(adm),27(sudo),...
```

Dans cet exemple, on s'aperçoit que j'ai l'UID 1000, que mon groupe primaire (mon GID) est le numéro 1000, et que j'appartiens également aux groupes 4 et 27 (et plein d'autres que j'ai éludés).

La base de données des utilisateurs s'appelle généralement `passwd`. Elle est écrite en clair dans le fichier `/etc/passwd`. La base de données des groupes, quant à elle, se trouve classiquement dans le fichier `/etc/groups`.

1.2.3 Les modules `pwd` et `grp` de la bibliothèque standard

Plutôt que d'interagir directement avec ces fichiers, il est plus pratique d'utiliser respectivement les modules standard `pwd` et `grp` pour accéder en lecture aux bases de données des utilisateurs et des groupes. Leur documentation (pwd, grp) est suffisamment concise pour que je me permette de vous y renvoyer directement.

Dans l'exemple suivant, on utilise l'appel système `getuid` pour récupérer le UID de l'utilisateur du programme, et le saluer par son login que l'on retrouve dans la base de données `passwd` via le module `pwd` :

```
>>> import os, pwd
>>> def say_hello():
...     pwd_struct = pwd.getpwuid(os.getuid())
...     print("Hello, {}!".format(pwd_struct.pw_name))
...
>>> say_hello()
Hello, arnaud!
```

1.2.3.1 Exercices

1. Compléter cette fonction `say_hello` de façon qu'elle affiche également le GID de l'utilsateur (via l'appel système `getgid`) ainsi que le nom de ce groupe en vous servant du module `grp`.
2. Utiliser ces modules pour essayer de reproduire le comportement de la commande shell `id` (au moins l'appel de base, sans option). Pour connaître tous les groupes auxquels un utilisateur appartient, vous pouvez vous servir de l'appel système `os.getgrouplist` qui prend en arguments le nom d'utilisateur en toutes lettres ainsi que son GID.

```
>>> os.getgrouplist('arnaud', 1000)
[1000, 4, 24, 27, 30, 46, 102, 108, 124]
```

Nous aurons l'occasion de revenir sur les notions d'utilisateurs et de groupes dans plusieurs chapitres de ce cours. En particulier, chaque fois que nous parlerons des permissions, qui sont le principal mécanisme de sécurité des systèmes Unix.

2 Le système de fichiers

En tant qu'utilisateurs réguliers de nos ordinateurs, on pourrait se demander ce qu'il y a de *si* important à propos du système de fichiers pour lui dédier un chapitre entier. En effet, c'est plutôt intuitif : c'est une arborescence composée de répertoires avec des fichiers dedans...

Cependant, vous avez sûrement déjà entendu quelqu'un déclarer que « Unix c'est génial parce que *tout est fichier* ». Vous n'y avez sûrement pas prêté attention sur l'instant, mais cela fait pourtant toute la différence. Figurez-vous que le système de fichiers Unix va très au-delà de la simple représentation du contenu d'un disque. C'est **une puissante abstraction matérielle** à la fois intuitive pour l'utilisateur et extrêmement commode pour le développeur : pour preuve, il vous suffit de savoir lire ou écrire dans un fichier pour être capable d'écouter les évènements liés aux périphériques d'entrée, interagir avec les périphériques de sortie, et même surveiller les programmes qui sont en train de s'exécuter !

Cela vaut bien la peine de jeter un oeil sous le capot, vous ne trouvez pas ?

2.1 Arborescence des fichiers

2.1.1 L'interface utilisateur classique

Commençons par enfoncer des portes ouvertes. Un système de fichiers (*filesystem*) se présentera toujours à l'utilisateur sous la forme d'une structure arborescente :

- Les noeuds de l'arbre sont appelés des **répertoires** (*directories* en anglais), quoique l'appellation "dossier" (*folder*) a été largement popularisée par les systèmes d'exploitation de chez Microsoft.
- Les feuilles de l'arbre s'appellent des **fichiers** (*files*).
- La **racine** du système de fichiers (*filesystem root*), est désignée conventionnellement par une simple barre oblique (`/`).

La figure 2.1 illustre cette arborescence. Nous y voyons trois fichiers différents, dont les chemins sont respectivement `/usr/bin/python3.4` (l'interpréteur Python), `/bin/bash` (le shell `bash`) et `/home/arnaud/.zshrc` (mon fichier de configuration personnel pour le shell `zsh`).

Il y a d'énormes chances pour que vous manipuliez déjà des arborescences de fichiers depuis des années. Ainsi, je ne vous apprendrai rien en vous disant que la commande `cd` permet de se déplacer dans cette arborescence, que `pwd` permet d'afficher le chemin du répertoire courant et que `ls` permet de lister le contenu d'un répertoire :

```
$ cd /home/arnaud/doc/cours-systeme
$ pwd
/home/arnaud/doc/cours-systeme
$ ls
makefile  src
```

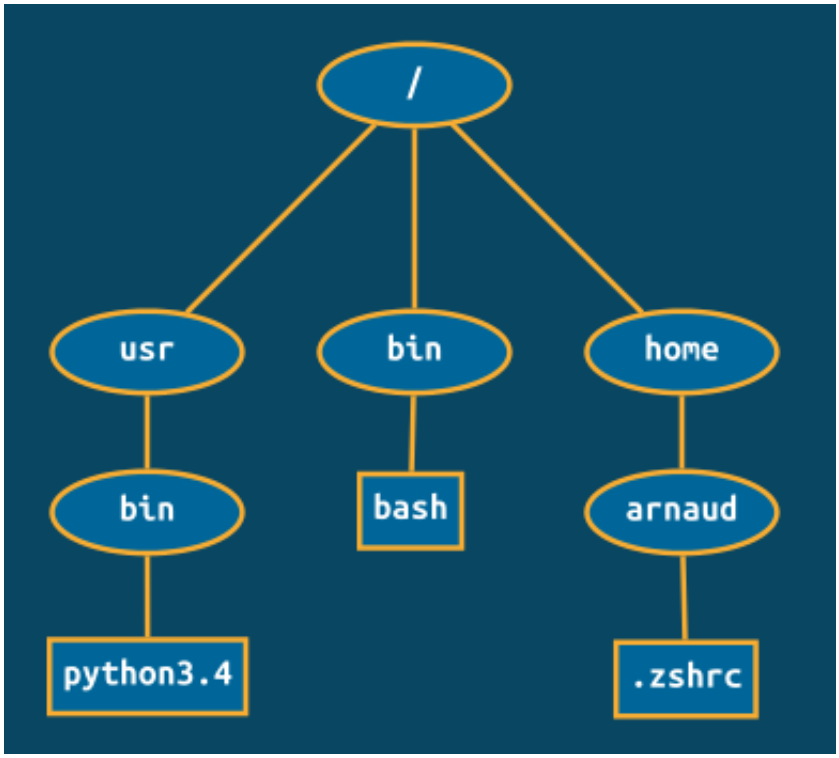

FIGURE 2.1 – Représentation abstraite du système de fichiers

Peut-être vous apprendrai-je en revanche que Python permet de réaliser les mêmes opérations, notamment grâce aux appels systèmes `os.chdir()` et `os.getcwd()`, ainsi que la fonction utilitaire `os.listdir()` :

```
>>> os.chdir('/home/arnaud/doc/cours-systeme')
>>> os.getcwd()
'/home/arnaud/doc/cours-systeme'
>>> os.listdir()
['src', 'makefile']
```

Toutes ces notions consistent à manipuler la représentation abstraite du système de fichier que le noyau nous expose. C'est largement suffisant pour un utilisateur occasionnel d'un ordinateur, mais je suis certain que le *hacker* qui sommeille en vous est bien plus curieux que ça, et désire en savoir plus sur ce qui se cache derrière cette abstraction.

2.1.2 La représentation bas niveau : les *inodes*

Peut-être avez-vous déjà utilisé l'option `-l` de la commande standard `ls` afin d'afficher le résultat sous la forme d'une liste détaillée :

```
$ ls -l
total 628
-rw-rw-r-- 1 arnaud arnaud    477 avril 17 22:51 makefile
drwxrwxr-x 3 arnaud arnaud   4096 avril 21 22:44 src
```

Cette commande fait apparaître pour chaque noeud fils du répertoire courant des informations telles que l'utilisateur et le groupe propriétaires du noeud, sa taille et sa date de dernière modification. On peut légitimement se demander d'où la commande `ls` tire toutes ces infos !

En réalité, tous les noeuds du système de fichiers (qu'il s'agisse de fichiers simples, de répertoires, ou d'autres choses encore) sont associés en interne à une structure que l'on appelle un *inode* (de l'anglais *index node* : *noeud d'index*). Cette structure recelle de nombreuses informations, et sert à *indiquer* où se trouvent les données du noeud correspondant.

Ces données peuvent prendre plusieurs formes. Dans le cas d'un système de fichiers *réel* (c'est-à-dire qui correspond vraiment au contenu d'un disque dur), on pourra trouver :

— Des données de fichiers brutes, qui ne sont rien d'autre que des séquences d'octets.
— Des données de répertoire. On peut se représenter un répertoire comme une collection d'**entrées de répertoire**. Chaque entrée associe le *nom* d'un noeud fils du répertoire au *numéro* de son inode.

Pour bien se représenter cette structure à bas niveau, la figure 2.2 détaille le chemin `/bin/bash` qui consiste à traverser deux répertoires (`/` et `/bin`) pour retrouver les données du fichier `bash`.

Le *numéro d'inode* est un nombre permettant d'identifier un noeud de l'arborescence de façon unique sur le périphérique auquel il appartient. Pour l'afficher, il suffit de passer l'option `-i` à la commande `ls`

```
$ ls -i
6422940 makefile  6422937 src
```

Ce dernier exemple nous montre que dans le répertoire courant, nous avons une entrée nommée `makefile` pointant sur l'inode numéro 6422940 et une entrée nommée `src` pointant sur l'inode numéro 6422937. Notez que je parle bien d'*entrées* : lorsque l'on traverse un répertoire, nous n'avons aucun moyen de savoir si une entrée donnée correspond à un fichier ou à un sous-répertoire. Pour avoir cette info, il faut aller lire les données des inodes.

2.1.3 L'appel système `stat` : lire le contenu des inodes

La structure réelle des inodes dépend du type de système de fichiers sur lequel ils stont stockés (et il en existe des tonnes : `ext4`, `reiserfs`, `ramfs`, `fat32`, `ntfs`...). Cela dit, quelle que soit la structure adoptée par le système de fichiers, il est possible d'accéder aux données qu'elle recèle en invoquant l'appel système `stat`, ou bien la commande POSIX du même nom.

Nous nous contenterons de réaliser cet appel système depuis Python, en utilisant la fonction standard `os.stat()`.

```
>>> stat_res = os.stat('/home/arnaud/doc/cours-systeme/makefile')
>>> stat_res
os.stat_result(st_mode=33204, st_ino=6422940, st_dev=64513, st_nlink=1,
               st_uid=1000, st_gid=1000, st_size=477, st_atime=1429727097,
               st_mtime=1429303879, st_ctime=1429303879)
```

Le retour de cet appel système est une structure complexe que Python représente par un tuple nommé. Prenons le temps de détailler ses champs dans le tableau ci-dessous.

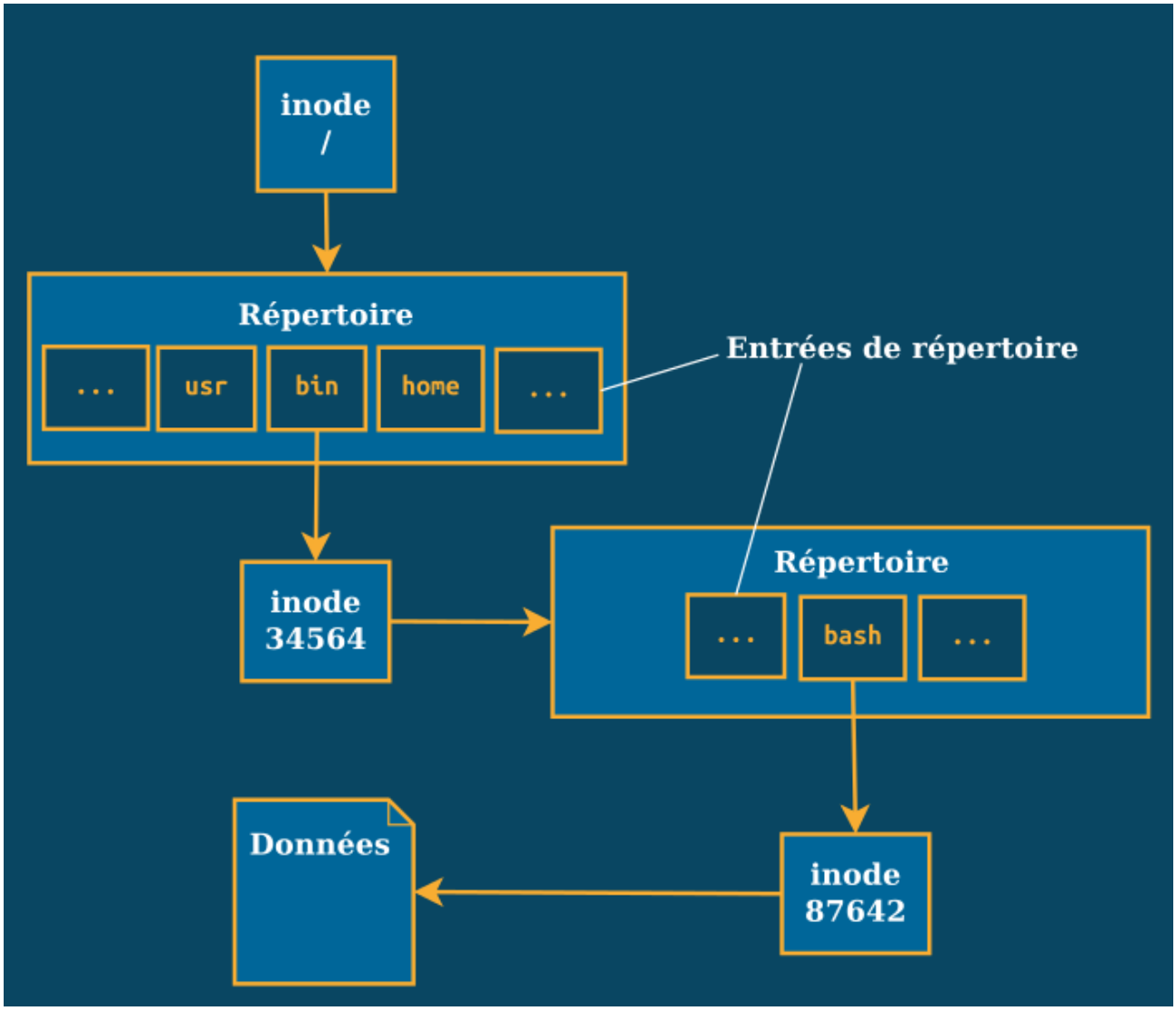

FIGURE 2.2 – Représentation physique du chemin `/bin/bash`

Champ	Signification
`st_mode`	Nombre codé sur 16 bits, décrivant le type et les permissions du fichier.
`st_ino`	Numéro d'inode. Identifiant unique de l'inode sur le périphérique.
`st_dev`	Numéro de *device*, qui identifie le périphérique de façon unique sur le système.
`st_nlinks`	Nombre d'entrées de répertoire qui pointent vers cet inode.
`st_uid`	Identifiant de l'utilisateur propriétaire du fichier.
`st_gid`	Identifiant du groupe propriétaire du fichier.
`st_size`	Taille du fichier en octets.
`st_atime`	*Access time*. Date du dernier accès, sous la forme d'un *timestamp Unix*. Ce champ est mis à jour chaque fois qu'un programme lit les données du fichier.
`st_mtime`	*Modification time*. Date de la dernière modification. Ce champ est mis à jour chaque fois qu'un programme écrit des données dans le fichier.
`st_ctime`	*Status change time*. Date de la dernière modification de cet inode. **Attention, ce n'est PAS la date de création du fichier.**

TABLE 2.1 – Résultat de l'appel système `stat`

Un *timestamp Unix* est une date représentée sur 32 bits. Il s'agit ni plus ni moins du nombre de secondes écoulées depuis le premier janvier 1970 à minuit (le timestamp 0), date que l'on surnomme *Epoch*. Pour décoder un timestamp en Python, il suffit d'utiliser le module `datetime` :

```
>>> from datetime import datetime
>>> d = datetime.fromtimestamp(1429303879)
>>> str(d)
'2015-04-17 22:51:19'
```

En somme, on peut en déduire qu'à l'exception du nom des fichiers, toutes les infos utiles se trouvent dans les inodes.

2.1.4 Type et permissions d'un fichier

Le champ le plus dense en informations que nous retourne l'appel système `os.stat()` est sans aucun doute `st_mode` : il y a tellement de choses encodées sur ces 16 bits que Python dédie un module standard entier (`stat`) à leur manipulation !

Concrètement, chaque bit de ce nombre est un *flag*, c'est-à-dire une option qui peut être activée ou non. Manipuler ce style de données est monnaie courante en C, mais reste assez anecdotique chez les Pythonistes [1].

Rappelons rapidement le fonctionnement des flags binaires. Imaginons un champ *vierge*, où tous les bits sont à 0 :

1. Il y a une explication très simple à cela. Les opérateurs bit-à-bit sont extrêmement rapides en C car les microprocesseurs savent les exécuter en une seule opération, mais pas en Python où les opérations sur les entiers, quelles qu'elles soient, sont comparativement très lentes.

```
>>> mode = 0
```

Nous pouvons vérifier si le flag `stat.S_IRGRP` (nous verrons sa signification plus loin) est activé au moyen d'un *ET bit-à-bit*, et l'activer au moyen du *OU bit-à-bit* :

```
>>> import stat
>>> bool(mode & stat.S_IRGRP)
False
>>> mode |= stat.S_IRGRP      # Activation du bit
>>> bool(mode & stat.S_IRGRP)
True
```

Le système va différencier trois types d'utilisateurs pour chaque fichier :

— son **propriétaire**, dont l'UID est égal à celui du fichier (`st_uid`),
— son **groupe propriétaire**, appartenant au groupe désigné par le GID du fichier (`st_gid`),
— les **autres**.

Pour chacun de ces types d'utilisateurs, on va définir trois *permissions* possibles :

— accès en lecture (l'utilisateur a le droit de lire le contenu du fichier),
— accès en écriture (l'utilisateur a le droit de modifier le fichier),
— accès en exécution (l'utilisateur a le droit d'exécuter le fichier).

Le tableau suivant établit la correspondance entre les valeurs des *flags* du module `stat`, l'indice visuel que l'on retrouvera un peu partout, et leur signification.

Flag	Mode	Octal	Signifiaction
`S_IRUSR`	`-r--------`	0400	Le propriétaire du fichier a le droit d'y accéder **en lecture**.
`S_IWUSR`	`--w-------`	0200	Le propriétaire du fichier a le droit d'y accéder **en écriture**.
`S_IXUSR`	`---x------`	0100	Le propriétaire du fichier a le droit **d'exécuter** le fichier.
`S_IRGRP`	`----r-----`	0040	Les membres du groupe propriétaire du fichier y ont accès **en lecture**.
`S_IWGRP`	`-----w----`	0020	Les membres du groupe propriétaire du fichier y ont accès **en écriture**.
`S_IXGRP`	`------x---`	0010	Les membres du groupe propriétaire du fichier ont le droit de l'**exécuter**.
`S_IROTH`	`-------r--`	0004	Les autres utilisateurs ont accès au fichier **en lecture**.
`S_IWOTH`	`--------w-`	0002	Les autres utilisateurs ont accès au fichier **en écriture**.
`S_IXOTH`	`---------x`	0001	Les autres utilisateurs ont le droit d'**exécuter** le fichier.

TABLE 2.2 – Masque des permissions sur un fichier

Bien évidemment, ces valeurs sont cumulatives. Ainsi, si l'on regarde celles de mon fichier `makefile` :

```
$ ls -l makefile
```

```
-rw-rw-r-- 1 arnaud arnaud 477 avril 17 22:51 makefile
```

Ce fichier m'appartient et appartient à mon groupe primaire. J'ai le droit de le lire et de le modifier, mon groupe primaire également, mais tous les autres utilisateurs ont seulement le droit de le lire. La représentation de ce mode, en octal, sera :

$$0400 + 0200 + 0040 + 0020 + 0004 = 0664$$

Cette représentation en octal vous semble peut-être sortir de nulle part, mais sachez qu'elle est utilisée aussi souvent que la notation visuelle dans les appels systèmes et les commandes shell qui permettent de manipuler le mode des fichiers, à l'instar de `chmod`[2].

Notez le module `stat` nous propose une fonction `filemode()` qui sert à représenter les permissions du fichier de façon visuelle :

```
>>> stat.filemode(os.stat('makefile').st_mode)
'-rw-rw-r--'
```

Les plus observateurs d'entre vous l'auront remarqué : cette représentation visuelle est préfixée d'un dixième symbole alors que nous n'en avons pour l'instant lu que 9. Qu'est-il donc supposé représenter ?

Voyons voir :

```
>>> stat.filemode(os.stat('.').st_mode)
'drwxrwxr-x'
```

En appelant `stat` sur le répertoire courant, ce dixième symbole se transforme en un `d`. D comme ? D comme di… dir… ? *Directory*, oui ! Ce dixième symbole nous donne le *type* de l'inode. Ainsi, on trouvera les flags suivants dans le module `stat` :

— `S_IFREG` : l'inode désigne un *fichier régulier* (un fichier, quoi), représenté par le préfixe `-`.
— `S_IFDIR` : l'inode désigne un *répertoire*, représenté par le préfixe `d`.

Il existe encore (plein) d'autres flags et d'autres informations encodées dans ce champ `st_mode`, mais nous allons nous contenter de ceux-ci pour l'instant[3]. Nous découvrirons toutes les autres au fur et à mesure que nous toucherons dans ce cours aux notions qu'elles font intervenir, promis !

2.1.5 Modifier les permissions d'un fichier

J'ai mentionné un peu plus haut la commande `chmod`. Voyons comment l'utiliser (ainsi que la fonction Python équivalente, évidemment).

Créons un fichier vide pour faire nos tests.

```
$ touch monfichier
$ stat -c "%A (%a) %U:%G" monfichier
-rw-rw-r-- (664) arnaud:arnaud
```

2. RTFM ! Non, je plaisante, on va en parler juste après. ;)
3. Par contre, si vous êtes curieux et voulez absolument en avoir un aperçu dès maintenant, alors cette fois je vous le dis très sérieusement : RTFM ! `man 2 stat` vous donnera la page du "*fucking manual*" à propos de l'appel système `stat`. Alternativement, la documentation du module Python `stat` vous dira… la même chose, en fait, mais sans vous préciser quels bits sont concernés.

L'option "`-c "%A (%a) %U:%G"`" de la commande `stat` sert à lui dire que je veux afficher uniquement les droits (en version lisible puis en octal), le propriétaire et le groupe du fichier.

La façon la plus frontale de modifier les droits d'un fichier sera de passer à `chmod` la valeur octale du masque de droits. Par exemple, si je souhaite retirer les droits en écriture au groupe, je vais calculer le nouveau masque en prenant l'actuel (0664) auquel je retranche la valeur de `S_IRGRP` (0020), ce qui me donne 0644 :

```
$ chmod 644 monfichier
$ stat -c "%A (%a) %U:%G" monfichier
-rw-r--r-- (644) arnaud:arnaud
```

L'autre syntaxe acceptée par `chmod` consiste à spécifier :

- le(s) type(s) d'utilisateur(s) (`u` pour le propriétaire, `g` pour le groupe et `o` pour les autres),
- l'ajout (`+`) ou le retrait (`-`) du droit,
- la valeur du droit (`r` pour la lecture, `w` pour l'écriture, `x` pour l'exécution)

Par exemple, pour rajouter les droits en exécution au propriétaire et ceux en écriture au groupe et aux autres :

```
$ chmod u+x,go+w monfichier
$ stat -c "%A (%a) %U:%G" monfichier
-rwxrw-rw- (766) arnaud:arnaud
```

Oh et puis non, en fait, je ne veux pas exécuter ce fichier ni que les autres le lisent ni écrivent dedans, finalement :

```
$ chmod u-x,o-rw monfichier
$ stat -c "%A (%a) %U:%G" monfichier
-rw-rw---- (660) arnaud:arnaud
```

En Python, cependant, cette syntaxe *user-friendly* n'existe pas. La fonction `os.chmod()` n'accepte que la valeur du masque, que vous pouvez spécifier soit de façon verbeuse en assemblant celui-ci via les flags du module `stat`.

```
>>> os.chmod("monfichier",
...          stat.S_IRUSR | stat.S_IWUSR | stat.S_IRGRP
... )
>>> stat.filemode(os.stat("monfichier").st_mode)
'-rw-r-----'
```

Soit, ce que personnellement je trouve beaucoup plus simple, en utilisant directement la valeur octale :

```
>>> os.chmod("monfichier", 0o664)
>>> stat.filemode(os.stat("monfichier").st_mode)
'-rw-rw-r--'
```

2.2 La manipulation des flux de données en Python

Maintenant que nous avons une abstraction pour représenter les fichiers sur un périphérique de stockage, voyons un peu comment le noyau va présenter ces fichiers aux programmes qui sont en train de s'exécuter. Nous en profiterons au passage pour découvrir la couche d'abstaction supplémentaire que rajoute Python, pour étendre ce concept à tout ce qui ressemble de près ou de loin à un *flux de données*.

2.2.1 Inodes et descripteurs de fichiers

Bien sûr, manipuler des fichiers est une chose très courante en programmation. Après tout, une fois que l'on a compris le fonctionnement des fonctions de base `open()`, `read()`, `write()` et `close()`, on peut se dire qu'on a fait le tour de la question... La figure 2.3 montre qu'en réalité, ces quelques appels système ne sont que la partie émergée de l'iceberg, et qu'il y a bien plus à en dire que d'expliquer leur signature !

Alors, que se passe-t'il *réellement* lorsque vous ouvrez un fichier dans un programme ?

Pour commencer, le noyau va aller chercher l'inode correspondant dans le système de fichiers. Une fois qu'il a trouvé cet inode (disque), il va le charger en mémoire. Ensuite, il va rajouter une entrée dans sa *table des fichiers ouverts*, qui contient en particulier un pointeur sur cet inode (mémoire). Cette entrée va également contenir d'autres données intéressantes, comme le *mode d'ouverture* du fichier (lecture, écriture, ajout...) et un indicateur de position (pour savoir où le processus [4] en est dans la lecture du fichier).

Cette table des fichiers ouverts est globale à tous les processus qui tournent sur le système. Cela permet, dans certains cas particuliers, que plusieurs processus manipulent la même entrée de la table des fichiers ouverts, mais nous verrons ce genre de choses beaucoup plus loin dans ce cours.

Le processus qui va chercher à manipuler ce fichier, quant à lui, dispose (côté noyau) d'une *table des descripteurs de fichiers*, qui associe, grosso-modo, un indice entier (le **descripteur de fichier**, qu'on abrègera FD dans la suite, pour *file descriptor*) à une entrée de la table des fichiers ouverts. Ce qu'il est intéressant de noter, c'est que cette table est *locale* au processus, tout comme les descripteurs de fichiers (puisque ce sont tout simplement les indices de cette table). Ainsi, comme le schéma nous le montre, on peut très bien avoir un programme ayant ouvert un fichier et que le noyau aura placé dans l'entrée numéro 5 de sa table des FD, et un second programme qui aura ouvert le même fichier mais en l'associant localement au FD 12.

Le fait que la table des descripteurs des fichiers du processus se trouve *côté noyau* n'est pas anodin, puisque ça signifie qu'on ne peut pas la manipuler directement (et donc que l'on ne peut pas faire n'importe quoi avec). Par contre, les descripteurs de fichiers sont exposés côté utilisateur, puisque ce sont eux que le programme passe en argument aux appels système `read`, `write` et `close` (et tout une palanquée d'autres dont nous nous garderons bien d'établir une liste aussi fastidieuse que superflue).

Ainsi, de notre point de vue d'utilisateurs des facilités du noyau, ce sont surtout ces descripteurs de fichiers qui nous intéressent.

4. Nous n'avons pas encore vu la notion de *processus* dans ce cours. Si cela vous perturbe, considérez qu'un processus est une instance d'un programme qui est en train de tourner sur l'ordinateur.

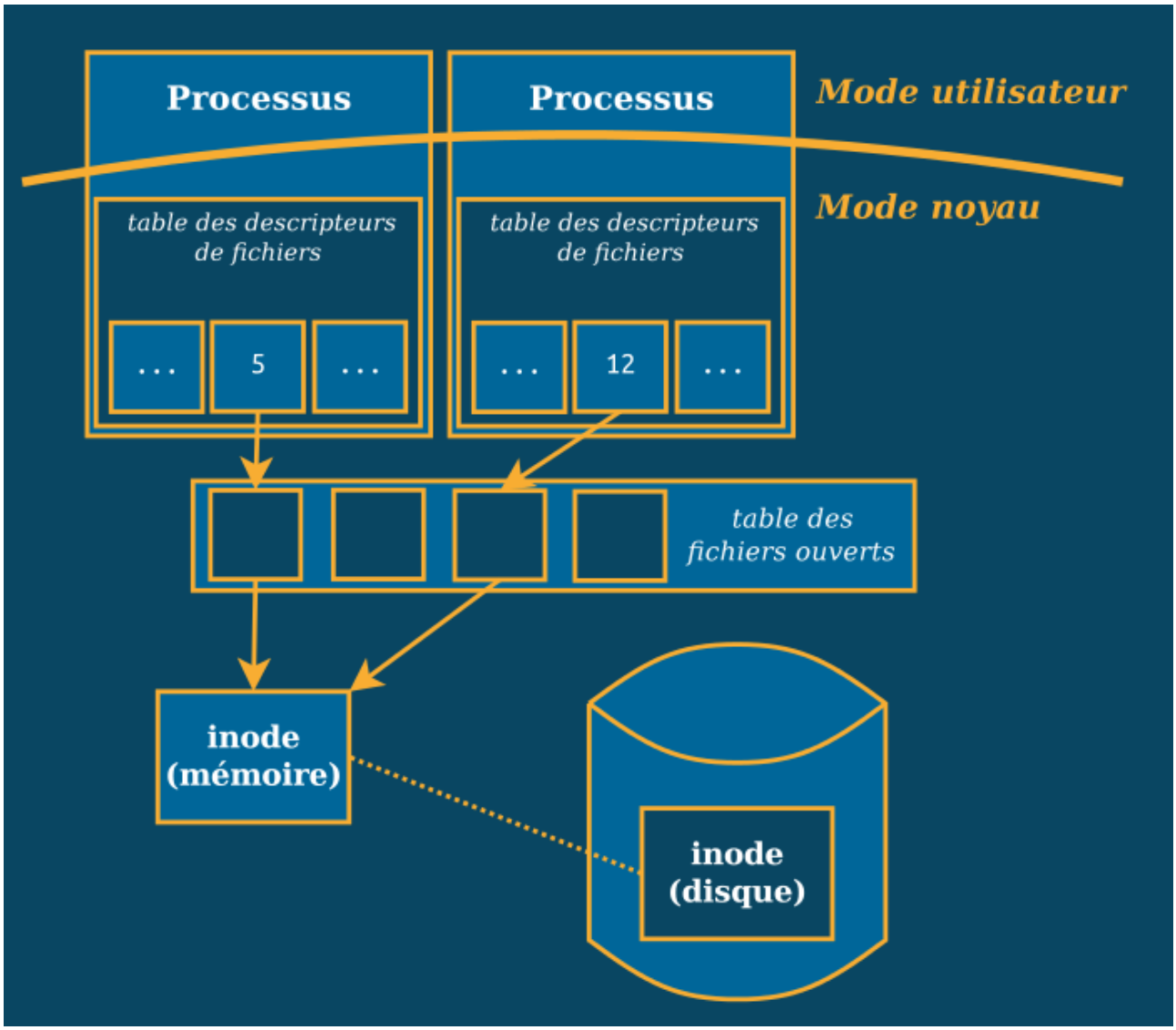

FIGURE 2.3 – Accès aux fichiers en mémoire

2.2.2 L'interaction à bas niveau avec les fichiers

Nous allons travailler sur un exemple simple. Nous allons copier le fichier `menu` dont le contenu est le suivant :

```
$ cat menu
* spam
* eggs
* bacon
* spam
* sausage
* spam
* ham
```

Pour cela, nous allons l'ouvrir, en lire le contenu, puis ouvrir un second fichier (qui n'existe pas encore) pour écrire le contenu à l'intérieur.

Ouvrir un fichier à bas niveau se fait grâce à l'appel système `os.open()`. Cet appel système prend au minimum en argument :

- le chemin vers le fichier,
- un *flag* parmi :
  - `os.O_RDONLY` pour ouvrir le fichier en lecture seule,
  - `os.O_WRONLY` pour ouvrir le fichier en écriture seule,
  - `os.O_RDWR` pour ouvrir le fichier à la fois en lecture et en écriture.

Sa valeur de retour est, bien sûr, le descripteur de fichier qui aura été créé dans la table locale du processus.

```
>>> fd = os.open("menu", os.O_RDONLY)
>>> fd
3
```

Comme vous le constatez, le fichier est associé au descripteur `3` dans la table des descripteurs de fichiers. Les fd `0`, `1` et `2` étant déjà occupés par les trois flux d'entrée/sortie standard du processus, que nous découvrirons plus loin.

Pour lire des données dans un fichier, on appellera la fonction `os.read()` qui réalisera l'appel système du même nom, en lui passant en arguments :

- le descripteur de fichier à lire,
- le nombre d'octets que nous comptons lire. Ce nombre peut être supérieur à la quantité de données restantes à lire dans le fichier, auquel cas la valeur retournée sera la totalité des données du fichier, ou bien inférieur, auquel cas un appel ultérieur nous permettra de continuer à lire les données.

```
>>> data = os.read(fd, 4096)
>>> data
b'* spam\n* eggs\n* bacon\n* spam\n* sausage\n* spam\n* ham\n'
```

Remarquez que la fonction `read` nous a retourné ici un objet de type `bytes` et non une chaîne de caractères Unicode (`str`) : il s'agit de données brutes, non décodées.

Ici, nous avons lu la totalité du contenu de notre fichier. Ainsi, si l'on appelle une nouvelle fois la fonction `read`, nous obtiendrons en retour une donnée vide :

```
>>> os.read(fd, 4096)
b''
```

Fermons notre fichier pour libérer le descripteur avec `os.close()` :

```
>>> os.close(fd)
```

Bien, maintenant, ouvrons le fichier `copie` en écriture :

```
>>> fd = os.open("copie", os.O_WRONLY)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'copie'
```

Python nous crache au visage que ce fichier n'existe pas. Contrairement à l'interface que vous connaissez déjà sûrement, l'appel système `open` ne va pas créer de nouveau fichier lorsque celui-ci n'existe pas. Et si celui-ci existe, il ne saura pas s'il doit l'effacer ou écrire à sa suite. C'est la raison pour laquelle les *flags* de cet appel système, lorsque l'on ouvre le fichier en écriture, doivent être combinés avec une ou plusieurs des valeurs suivantes :

- `os.O_APPEND` : Écrire à la fin du fichier (en mode "ajout") si celui-ci existe déjà,
- `os.O_TRUNC` : Tronquer le fichier (supprimer ses données) si celui-ci existe déjà,
- `os.O_EXCL` : Échouer (en Python, lever une exception) si le fichier existe déjà,
- `os.O_CREAT` : Créer le fichier si celui-ci n'existe pas encore.

Lorsque l'on utilise cette dernière valeur, il est également nécessaire de fournir un argument supplémentaire à `os.open` pour spécifier le `mode` du fichier créé, avec exactement la même sémantique que la fonction `os.chmod`.

Ainsi, nous pouvons ouvrir ce nouveau fichier `copie` en écriture de la façon suivante :

```
>>> fd = os.open("copie", os.O_WRONLY | os.O_CREAT | os.O_TRUNC, mode=0o640)
>>> fd
3
```

Remarquez que `fd` vaut ici une nouvelle fois 3 : le descripteur que nous avons fermé plus haut est réutilisé.

Pour écrire des données dans ce nouveau fichier, rien de plus simple :

```
>>> os.write(fd, data)
52
```

La fonction nous retourne le nombre d'octets effectivement écrits dans le fichier, soit 52 (la taille de la donnée). Nous pouvons maintenant refermer le fichier pour éviter de charger inutilement la table des descripteurs et libérer la ressource.

```
>>> os.close(fd)
```

Nous pouvons remarquer que le fichier `copie` a bien été créé, que j'en suis évidemment propriétaire puisque c'est moi qui l'ai créé, et qu'il s'agit bien d'une copie du `menu` :

```
$ ls -l copie
-rw-r----- 1 arnaud arnaud 52 avril 26 21:57 copie
$ cat copie
* spam
* eggs
* bacon
* spam
* sausage
* spam
* ham
```

Rien de bien compliqué, en somme !

Mais que sont donc ces trois descripteurs de fichier 0, 1 et 2 ? Conventionnellement, le système va attribuer à chaque processus trois *flux* standard :

- `STDIN` (0) : L'entrée standard du processus, qui permet par exemple de lire des données que l'utilisateur saisit au clavier,
- `STDOUT` (1) : La sortie standard, que l'on utilise conventionnellement pour afficher des données dans la console,
- `STDERR` (2) : La sortie d'erreur standard, sur laquelle on écrit les descriptions des erreurs lorsqu'elles se produisent.

Ainsi, les FD 1 et 2 se comporteront comme des fichiers ouverts en écriture :

```
>>> os.write(1, b'Hello, World!\n')
Hello, World!
14
>>> os.write(2, b"Une erreur s'est produite\n")
Une erreur s'est produite
26
```

Quant à l'entrée standard, celle-ci peut être lue avec `read()` : faites l'essai en tapant du texte puis en validant avec la touche `<Entrée>`.

```
>>> os.read(0, 4096)
Ceci est une saisie au clavier.
b'Ceci est une saisie au clavier.\n'
```

Comme vous le constatez, les "fichiers" que le noyau nous expose peuvent également être des abstractions pour le terminal, le clavier, ou un périphérique matériel ou d'autres choses encore... Je suppose que vous comprenez, maintenant, le potentiel des quatres fonctions que nous venons de voir.

2.2.3 L'interface haut niveau : les *file objects*

Dans « la vraie vie », on n'a jamais besoin d'utiliser les quatre appels système que nous venons d'évoquer en Python. Et pour cause ! Celui-ci expose au développeur une interface beaucoup plus confortable pour manipuler des fichiers. Cette interface consiste à envelopper un *file descriptor* dans ce que Python nomme un *file-like object*, et est implémentée dans le module standard `io`.

C'est dans ce module en particulier que réside l'implémentation de la fonction *builtin* `open()`. Contrairement à l'appel système du même nom (qu'elle abstrait pour nous), son comportement est *souple* : nul

besoin de lui spécifier un ensemble de flags complexes pour décider quoi faire, puisqu'elle adopte un comportement par défaut qui répond aux besoins les plus courants. Nous ne détaillerons pas la totalité de son comportement ici, un simple `help(open)` vous fournira sa documentation complète. Cela dit, la différence majeure entre cette fonction à haut niveau et les appels système qu'elle englobe est qu'elle présuppose deux types d'interactions différents avec les fichiers :

— l'interaction en mode **texte**,
— l'interaction en mode **binaire**.

Le mode d'ouverture le plus proche de l'appel système est sans doute le mode binaire. En effet, ce mode permet de lire ou écrire des données brutes (représentées par des objets `bytes`).

Par exemple, utilisons ce mode d'interaction pour lire le fichier `menu` de l'exemple précédent.

```
>>> fobj = open('menu', 'rb')
>>> data = fobj.read()
>>> data
b'* spam\n* eggs\n* bacon\n* spam\n* sausage\n* spam\n* ham\n'
```

Pour rappel, le mode `'rb'` que l'on a passé à cette fonction signifie :

— `'r'` : le fichier est ouvert en lecture (*reading*)
— `'b'` : le fichier est ouvert en mode binaire (*binary*)

En appelant la méthode `read()` de cet objet (sans lui spécifier une taille de buffer), nous avons lu la totalité de son contenu qu'il nous a retourné dans un objet `bytes`.

Si nous souhaitons exploiter cette donnée, il nous faudrait bien sûr la convertir en une chaîne de caractères Unicode, en la décodant.

Ce genre de traitement étant particulièrement courant, Python permet, par défaut, d'ouvrir les fichiers "en mode texte", pour nous épargner la plupart des tâches répétitives.

```
>>> fobj = open('menu', 'r')
>>> data = fobj.read()
>>> data
'* spam\n* eggs\n* bacon\n* spam\n* sausage\n* spam\n* ham\n'
```

Cela peut paraître absolument enfantin (pour l'utilisateur, en tout cas, ça l'est !), mais maintenant que vous savez que Python est obligé de passer par l'appel système `open` lorsqu'il va ouvrir notre fichier en mode texte, *pouvez-vous deviner l'ensemble des opérations supplémentaires qu'il réalise en plus de l'appel système pour avoir ce comportement ?*

3 Les processus

4 Le Parallélisme multi-processus

5 Les threads