UNIX AU TEMPS DU COUVRE-FEU (3)
COMPLÉMENTS ET EXERCICES

par

Thierry Dumont

Table des matières

Dans la suite la syntaxe :

```
cmd>commande arguments
```

désignera une commande (à taper dans le shell) ; le « prompt », c'est à dire le début de la ligne de commande peut varier d'une configuration à l'autre ; `cmd>` *représente ce prompt et n'est pas à taper.*

1. Chemins dans l'arbre des fichiers.

Dans la suite j'emploie le mot directory ou le mot répertoire pour désigner la même chose.

Il y a plusieurs moyens de définir des chemins. On rappelle que la racine de l'arbre est désignée par `cmd>/`.

1. Chemin absolu en partant de la racine ; exemple : `cmd>/etc/firefox/pref`

   Ce qui peut se lire de droite à gauche : `cmd>pref` est dans le répertoire `cmd>firefox` et `cmd>firefox` est dans le répertoire `cmd>etc`, qui est à la racine [1], ou bien de gauche à droite : `cmd>etc` est à la racine, `cmd>firefox` est dans `cmd>etc` et `cmd>pref` est dans le répertoire `cmd>firefox`.
2. Chemins relatifs : le point `cmd>.` désignant le répertoire courant, les deux points `cmd>..` désignant le répertoire père et `cmd>~` (tilde) désignant le *home directory*, on peut définir des chemins *relatifs* (au-dessus, à coté, etc.), comme par exemple `cmd>../truc`.

On rappelle les deux commandes `cmd>pwd` et `cmd>cd` :

— `cmd>pwd` (print working directory) : vous dit où vous êtes.

— `cmd>cd` pour changer de répertoire (change directory) :

`cmd>cd` sans argument vous ramène à votre home-directory. Sinon, la commande doit être

`>cd` chemin

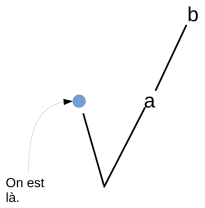
où `>` chemin est un chemin, relatif ou absolu ; exemple :

`cmd>cd /usr/local/bin` (chemin absolu : on décrit le chemin depuis la racine),

ou bien

`cmd>cd ../truc/toto` (c'est à dire aller dans le répertoire père puis là, aller dans `truc` puis dans `toto` –qui doivent exister pour que ça fonctionne–) : c'est un chemin relatif (on décrit le chemin depuis là où on est).

Illustration (déplacement avec un chemin relatif) :



Après `cmd>cd ../a/b`, on est en b.

Exercices **:**— à faire depuis un terminal [2].

1. Déplacements absolus et relatifs :
   - — aller directement dans `cmd>/etc/network`
   - — de là, aller dans `cmd>/etc`
   - — revenir dans le home directory.

   (Note : vous avez certainement du faire un déplacement relatif).
2. À quoi correspond : `cmd>cd ../..` ?

1. Noter que dans `cmd>/etc/firefox/pref`, `cmd>pref` peut être un répertoire ou un fichier, mais si je tape `cmd>/etc/firefox/pref/`, alors `cmd>pref` est obligatoirement un répertoire.
2. on dira plutôt « shell » que terminal, c'est plus chic.

2. La commande `cmd>ls`

Regarder et jouer avec la commande `cmd>ls` permet de se familiariser avec les autres commandes qui fonctionnent toutes sur le même principe :

1. `cmd>ls`

   sans option (**tester**).

2. `>ls` chemin

   par exemple :

   `cmd>ls /usr/bin`

   Exercice : jouer avec différents types de chemins utilisés ci-dessus.

3. `cmd>ls` avec options. Les options commencent par `cmd>-` (c'est le tiret, c'est à dire le signe "*moins*" du 6).

   — Placez vous d'abord dans le home directory, puis tapez :

   `cmd>ls -a`

   Avec l'option `cmd>-a ls` montre les fichiers et répertoires cachés (installés par vos applications, leur nom commence par un point) **(tester).**

   — Avec l'option `cmd>-l` :

   `cmd>ls -l`

   on obtient plein de renseignements (on verra ça plus tard) **(tester).**

   — Combiner plusieurs options

   `cmd>ls -al`

   et puis aussi `cmd>ls -alt`

   (résultats triés par dates) et on peut aussi inverser l'ordre de tri :

   `cmd>ls -altr` **(tester).**

   — Appliquer cette commande avec des options à un chemin :

   `cmd>ls -alt /var/log`

   etc, etc. **(tester).**

   Noter que la syntaxe des options est assez souple : elles peuvent être mises dans n'importe quel ordre ; de plus au lieu de `cmd>ls -alt /var/log` on aurait pu écrire

   `cmd>ls -l -a -t /var/log`

   mais en ne voit pas trop quel en aurait été l'intérêt.

3. Quelques autres commandes

— Les systèmes Unix contiennent leur documentation (manual). La commande est `cmd>man`.

Exercice : jouer avec `cmd>man` par exemple :

`cmd>man ls`

ou bien : `cmd>man man`

Le résultat peut être indigeste !

— `cmd>whoami`

Vous donne votre *login*. En principe votre home-directory est `/home/`+le résultat de cette commande. **Le vérifier**.

— `cmd>less` permet de voir un fichier page par page (appuyer sur espace pour aller à la page suivante). On peut aussi sauter à la prochaine occurrence d'un motif (taper /motif quand on est dans less). (q pour quitter less). Un exemple **(tester)** :

```
cmd>less /etc/services
```

— `>rm` chemin

Efface un fichier (celui qui est au bout du chemin).  **Attention, on travaille sans filet ! Quand un fichier est effacé, on n'a plus qu'à aller chercher la sauvegarde (si on en a une).**

Exemples :

```
cmd>rm toto
cmd>rm ~/truc/machin/toto
```

— `>mkdir` chemin

Crée un répertoire (il ne doit pas déjà exister). Exemples :

```
cmd>mkdir MonDir
cmd>mkdir /truc/machin/chose
```

Ça fonctionnera seulement si /truc/machin/ existe déjà. Sinon faire :

```
cmd>mkdir -p /truc/machin/chose
```

et on crée comme ça tous les répertoires.

— `>rmdir` chemin

Efface un répertoire vide. Exemple :

```
cmd>rmdir ~/truc/machin/chose
```

Et si le répertoire n'est pas vide ? On a un message d'erreur. Alors il faut utiliser la commande `cmd>rm` avec l'option `cmd>-r` (récursive) qui efface le répertoire et tout ce qu'il contient :

```
cmd>rm -r ~/truc/machin/chose
```

Si on a peur, on peut ajouter `cmd>-i`, qui vous demandera de confirmer chaque effacement :

```
cmd>rm -ri ~/truc/machin/chose
```

— Filtre : `cmd>grep`

`cmd>grep udp /etc/services` va lister toutes les lignes qui contiennent `udp` dans le fichier `/etc/services`.

`cmd>grep -v udp /etc/services` va lister toutes les lignes qui ne contiennent **pas** `udp`.

Si on rajoute `cmd>-i` (par exemple `cmd>grep -v nagios /etc/services`) la recherche ne tient plus compte de la casse (minuscules ou majuscules).

Exercice :

La commande :

```
cmd>touch toto
```

crée un fichier vide de nom `cmd>toto`.

Créer des répertoires (emboîtés), créer des fichiers vides dedans et tout effacer [3].

4. Plomberie

Deux définitions :

1. **Entrée standard :** ce que les commandes lisent. Par défaut, c'est votre clavier.
2. **Sortie standard :** là où les commandes écrivent : la fenêtre courante par défaut.

Alors :

3. Mettez donc l'option `cmd>-i` quand vous utilisez `cmd>rm`.

— On peut rediriger la **sortie standard** vers un fichier :

`cmd>ls /var/log >toto`

Les résultats vont dans `cmd>toto` plutôt que sur l'écran. Regarder ensuite le fichier `cmd>toto` avec :

`cmd>less toto`

— Avec `cmd><` c'est l'entrée standard qu'on redirige depuis un fichier (on verra ça plus tard).

— Le « pipe » :

Le principe est le suivant : étant donné des commandes `cmd>A` et `cmd>B`, on fait en sorte que la sortie standard de `cmd>A` soit l'entrée standard de `cmd>B` :

`cmd>A|B`

(le caractère `cmd>|` est obtenu par `Alt Gr` et la touche "6").

Exemple (à tester) :

— On commence par `cmd>grep tcp /etc/services`

qui liste les lignes qui contiennent `tcp` dans `/etc/services`. Évidemment, le résultat sort sur l'écran (la sortie standard).

— La commande `cmd>grep`, s'il n'y a pas de chemin défini, lit sur l'entrée standard.

Donc

`cmd>grep tcp /etc/services | grep Protocol`

va :

1. Filtrer les lignes de `cmd>/etc/services` qui contiennent `tcp`.
2. Ensuite, `cmd>grep Protocol` récupère ce qui normalement va sur l'écran et filtre les lignes qui contiennent `Protocol`.

et voilà. On peut chaîner autant de commandes qu'on veut. Par exemple, la commande :

`>nl` chemin vers un fichier

numérote les lignes d'un fichier (regarder par exemple ce que donne `cmd>nl /etc/services`) ; en l'absence d'argument, la commande `cmd>nl` lit l'entrée standard.

On peut donc chaîner 3 commandes (`cmd>grep`, `cmd>grep` et `cmd>nl`) :

`cmd>grep tcp /etc/services | grep Protocol|nl`

et **compter** ainsi le nombre de lignes du fichier `cmd>/etc/services` qui contiennent `tcp` **et** `Protocol`.

Et puis on peut finir par la commande `cmd>tail` ; ainsi

`cmd>tail /etc/services`

lit le fichier `/etc/services` et en montre la fin, mais `cmd>tail` sans chemin vers un fichier lit sur l'entrée standard. Donc, pour savoir combien de lignes dans `/etc/services` contiennent `tcp` et `Protocol`, et pour limiter le nombre de lignes qui, à la fin, sortent sur l'écran, on peut taper (4 commandes chaînées) :

`cmd>grep tcp /etc/services|grep Protocol|nl|tail` [4]

4. ou `cmd>grep tcp /etc/services|grep Protocol|nl|tail -1` cf. `cmd>man tail`

5. Quelques petits trucs à savoir à propos du shell

Le shell, c'est le programme qui interprète vos commandes. En fait, il existe plusieurs shells disponibles, mais le plus populaire est celui de Steve Bourne, qui a connu plusieurs évolutions, et qui s'appelle maintenant « Bourne Again Shell [5] », soit `cmd>bash`. Si vous tapez :

`cmd>which bash`

vous verrez où il est installé (`cmd>which` vous donne le chemin vers une commande).

Il existe d'autres shells, comme `cmd>zsh`, mais bon...

Des trucs bien pratiques à savoir :

— La complétion automatique :

C'est plutôt puissant. On utilise le caractère `Tab`.

Essayez :

a`Tab` (deux caractères : le **a** et le `Tab`),

puis :

ap`Tab` (note : il faut parfois taper 2 fois `Tab`).

On voit toutes les commandes accessibles qui commencent par `ap`.

De plus, certains logiciels introduisent des règles de complétion. Par exemple :

`cmd>evince toto.Tab`

proposera de visualiser le fichier `toto.pdf` s'il existe.

— L'historique :

— à titre d'exemple, `cmd>!evin` relance la dernière commande qui commençait par `cmd>evin` (attention, tout de même : pensez aux dégats possible de `cmd>!rm`).

— `cmd>history` donne l'historique des commandes, numéroté. On peut par exemple relancer la 135^{e}commande en tapant :

`cmd>!135`

cet historique persiste, même si vous redémarrez la machine.

On peut aussi se servir des flèches pour naviguer dans l'historique.

Un bon conseil : **testez, retestez !**

6. Utilisateurs et groupes

6.1. Les utilisateurs. — Les utilisateurs sont définis dans le fichier `/etc/passwd`

Regardons [6] une ligne du ficher, correspondant à la description de l'utilisateur de nom `moi` :

`moi:x:1000:1000:moi„,:/home/moi:/bin/bash`

On y voit des champs séparés par « `:` » qui sont :

1. Le nom de connexion de l'utilisateur (« login ») (`moi`, ici).
2. `x` : qui contenait jadis le mot de passe.
3. `1000` : en pratique, c'est un nombre qui identifie l'utilisateur, et pas son nom (plus rapide, plus simple). Ici, `1000` et `moi` sont associés et identifient le même utilisateur. `1000` est l'`uid` de `moi`. Et l'`uid` est évidemment unique, des utilisateurs différents ont des `uid` différents.
4. `1000` : identifiant du groupe de l'utilisateur.
5. `moi„,` : le « vrai »nom (j'aurais pu mettre `Thierry Dumont`), suivi de commentaires optionnels.

5. ah, ah, l'astuce !
6. `cmd>less /etc/passwd` (par exemple).

6. Le home directory (`/home/moi`).
7. Le chemin vers le shell utilisé, ici c'est `bash`.

Les utilisateurs appartiennent à des groupes (un ou plusieurs). Dans le fichier `/etc/passwd` on décrit le groupe *principal* auquel l'utilisateur appartient. *Dans beaucoup de distributions Linux, le groupe principal a le même nom que l'utilisateur (moi, ici) et ne contient qu'un seul utilisateur ! Mais rien n'empêche de regrouper les utilisateurs dans différents groupes, ce qui, on va le voir permettrait de définir des ensembles d'utilisateurs plus ou moins privilégies par exemple.*

Remarques :

— En parcourant le fichier `passwd`[6], on remarque que pour certains utilisateurs, le shell est `/usr/sbin/nologin` ou `false` : on ne peut pas se connecter au système sous ses noms d'utilisateur, mais on en verra l'intérêt quand on regardera les *processus*.

— Commandes :

`cmd>adduser` permet d'ajouter un utilisateur,

`cmd>deluser` d'en supprimer un.

Vous ne pourrez pas utiliser ces commandes sous votre login habituel : l'explication viendra par la suite.

6.1.0.1. Un utilisateur pas comme les autres : root. —

```
root:x:0:0:root:/root:/bin/bash
```

root a tous les droits ! même de faire

```
cmd>rm -f /
```

Il convient donc *d'être très prudent* quand on prend cette identité, ce qui est absolument nécessaire pour accéder à certaines parties du système, configurer, installer des logiciels etc. On regardera ça par la suite.

6.2. Les groupes (d'utilisateurs). — Ils sont décrits dans le fichier `/etc/group` qui pour chaque groupe donne :

— Le nom du groupe.

— x : (champ désactivé).

— Le group-id (gid), un nombre unique.

— La liste des utilisateurs qui appartiennent à ce groupe.

Par exemple, su ma machine, le fichier `/etc/group` contient la ligne :
`lpadmin:x:121:moi`.

Ici, l'utilisateur "moi" appartient au groupe lpadmin. On peut conjecturer que "lpadmin" a à voir avec les imprimantes (*line printer admininistration*) et qu'être membre de ce groupe va donner des droits d'administration particuliers.

Exercice : À quels groupes est ce que j'appartiens en tant qu'utilisateur ?
(utiliser `cmd>grep` et `cmd>/etc/group`[7]).

7. Droits

7.1. Commençons par un petit essai. — Dans le fichier `/etc/passwd`, le champ du mot de passe est remplacé par un `x`, le mot de passe effectif étant dans `/etc/shadow`.

7. Ou plus simplement la commande `cmd>groups` (sans argument).

Exercice : Essayez de regarder le fichier /etc/shadow (`cmd>less /etc/shadow`). Que se passe-t-il ?

Il y a visiblement des problèmes de permission (droits) : c'est normal, car on a affaire à un fichier *sensible*, mais comment est-ce que ça marche ? Il faut s'intéresser aux *droits*.

7.2. Droits concernant les fichiers, les répertoires etc.—

Résultat de :

```
cmd>ls -l /etc/passwd
```

```
-rw-r--r-- 1 root root 3444 déc. 6 11:42 /etc/passwd
```

Le premier tiret de

```
-rw-r--r-- 1 root root 3444 déc. 6 11:42 /etc/passwd
```

indique qu'on a affaire à un fichier ; à la place du `-` on aurait **d** pour un répertoire.

Ensuite, il y a 3 triplets de 3 caractères (ici `rw-`, `r--` et `r--`). Dans chaque triplet le premier caractère peut être `r` ou `-`, le second `w` ou `-` et le troisième `x` ou `-`.

- `r` indique le droit de lire, `-` l'impossibilité de lire.
- `w` indique le droit d'écrire, `-` l'impossibilité d'écrire. Attention, écrire doit être pris au sens le plus large : modifier, déplacer dans l'arborescence, renommer ou effacer, tout ce qui peut modifier.
- `x` indique :
  - pour un fichier, le droit d'exécuter, c'est à dire que le fichier est une commande, un programme exécutable.
  - pour un répertoire, c'est le droit de traverser le répertoire, ce qui n'implique pas de lire son contenu !

  Et `-` indique l'absence de ce droit.

Que signifient chacun de ces 3 triplets ?

1. Le premier, les droits de l'utilisateur (root dans notre cas). Qui, ici, peut donc lire le fichier et écrire dessus (modifier, effacer, déplacer, etc !!!).
2. Le second, les droits des autres membres de son groupe (en pratique, dans ce cas précis, il n'y en a sûrement pas) : seulement lire dans le cas présent.
3. Le troisième : les droits du reste du monde, c'est à dire de tous les autres utilisateurs : ici, c'est seulement le droit de lire.

Peut-on changer les droits d'un fichier ou d'un répertoire ? Oui, à condition qu'il vous appartienne, dans ce cas la commande est `cmd>chmod` ; on va en voir un exemple plus loin.

7.3. Autres exemples. —

- Pour `/etc/shadow`, les droits sont :

  ```
  -rw-r----- 1 root shadow 1763 déc. 6 11:42 /etc/shadow
  ```

  et donc le fichier appartient à root et au groupe shadow. Vu les droits, on voit qu'un utilisateur qui n'est pas "root" et qui n'appartient pas à "shadow" ne peut pas lire ce fichier. Notons que si on ajoutait un utilisateur comme "moi" au groupe "shadow", alors il pourrait lire ce fichier.

— `bash` : le shell. Pour savoir où il est, la commande est :

`cmd>which bash`

Le résultat est probablement `/usr/bin/bash`. Ensuite

`cmd>ls -l /usr/bin/bash`

donne :

`-rwxr-xr-x 1 root root 1183448 juin 18 2020 /usr/bin/bash`

Remarquez les "x" pour tout le monde : c'est normal, car il s'agit d'un programme utilisé par tout le monde.

Exercice : Étudiez la racine du système de fichiers en tapant : `cmd>ls -l/`

Exercice :

1. `cmd>cd /tmp` (pour être dans un coin tranquille).
2. Créez des répertoires emboîtés :

   `cmd>mkdir toto`

   `cmd>mkdir toto/tutu` [8]

   Créez un fichier vide dans tutu de nom "file" (ou ce que vous voulez) :

   `cmd>touch toto/tutu/file`
3. Pour voir les droits affectés à toto : `cmd>ls -dl toto`. Cela devrait donner (en remplaçant moi par votre nom d'utilisateur) :

   `drwxrwxr-x 3 moi moi 4096 déc. 24 15:23 toto`
4. Vérifiez que les commandes :

   `cmd>ls toto` et `cmd>ls toto/tutu` fonctionnent.
5. Maintenant changez les droits de toto :

   `cmd>chmod chmod ugo-rw toto`

   (ce qui veut dire "enlever les droits de lire et d'écrire à l'utilisateur (u), au groupe (g) et aux autres (o)).

   `cmd>ls -dl toto` vous indique les nouveaux droits :

   `d--x--x--x 3 moi moi 4096 déc. 24 15:23 toto`

   On ne peut plus lire ni écrire.
6. Regardez maintenant ce que donnent les commandes :
   — `cmd>cd toto` puis `cmd>ls toto`.
   — Maintenant que vous êtes dans `toto`, tapez `cmd>cd tutu` ou `cmd>cd /tmp/toto/tutu`. Vérifiez que vous pouvez écrire dans tutu (`cmd>touch xx` par exemple).
7. Retour dans `/tmp` : `cmd>cd /tmp`. Essayez d'effacer `toto`. Comme il n'est pas vide, la commande `cmd>rmdir` ne peut pas fonctionner. Il faut utiliser `cmd>rm -r toto`. Que se passe-t-il ?
8. Redonnons nous le droit d'écrire sur et dans toto : `cmd>chmod u+w toto`. Essayons à nouveau `cmd>rm -r toto`.

   Pourquoi est ce que ça ne marche pas ? Parce que `cmd>rm -r` a besoin de lire le contenu de `toto` pour l'effacer. On tape `cmd>chmod u+r toto` et là, la commande `cmd>rm -r toto` efface tout. Subtil !

8. On peut faire ça en une seule commande `cmd>mkdir -p toto/tutu`

8. Processus

Dans un guide Unix/Linux on peut lire : *« Un processus est une instance d'un programme en train de s'exécuter, une tâche. Le shell crée un nouveau processus pour exécuter chaque commande »*. Le terme *instance* faisant partie du sabir informatique (le mot n'est pas français), on va tenter d'illustrer ça :

`cmd>which firefox`
Réponse :
`/usr/bin/firefox`.
Le programme firefox est donc stocké dans le fichier `/usr/bin/firefox`.

Que se passe-t-il si je tape `cmd>firefox` en ligne de commande ? Le shell (bash) va chercher [9] un fichier de nom firefox [10] et, après l'avoir trouvé, et de manière un peu simplifiée, demander au noyau linux de le copier en mémoire et de lancer son exécution.

À l'exécution, ce n'est pas seulement une copie qui réside en mémoire, mais aussi des données [11], une description des fichiers ouverts et un « état programme », qui pointe vers l'instruction en cours dans le programme.

À partir de ce moment là, cette "copie" (= instance) devient pendant toute la durée de son exécution un processus.

Le processus ne peut pas accéder en dehors des zones de mémoire que le système lui a alloué.
À chaque processus sont associés :

1. un numéro (PID, *process id*) unique et chaque processus appartient à un utilisateur et hérite des droits (de lire, d'écrire,...) de l'utilisateur *et* des groupes auxquels il appartient. D'autre part, un utilisateur n'a aucun droit sur les processus des autres utilisateurs.
2. un processus *père*, repéré par son PPID (*parent process identifier*) : *tout processus* a un père, à une exception près, car il faut bien un ancêtre commun : c'est le processus `init` qui a le PID 1. Tous les autres en descendent et ont des PID > 1.

Pour pouvoir faire quelques expériences, il faut d'abord apprendre à maîtriser l'outil qui permet de voir quels sont les processus présents et leurs caractéristiques : la commande `cmd>ps` suivie d'options.

8.1. La commande `cmd>ps`. — La commande `cmd>ps` est un peu bizarre.

— Si vous tapez juste `cmd>ps`, vous ne verrez que les processus qui dépendent de votre shell (terminal) soit en général juste `bash` et `ps` (puisque justement, vous êtes en train de faire tourner...`ps`) :

```
    PID TTY          TIME CMD
 338023 pts/5    00:00:00 bash
 371762 pts/5    00:00:00 ps
```

mais vous voyez quand même les `PID`, le terminal `TTY` auquel ils sont attachés, le temps calcul dépensé `TIME` et le nom de la commande `CMD`.

— Si vous tapez `cmd>ps -e`, vous voyez tous les processus en cours. Je recommande de taper

```
cmd>ps -e | less
```

car il y a beaucoup de processus en cours ! Mais ça ne dit quand même pas grand chose sur chaque processus.

9. dans une liste de chemins bien définis – on verra ça plus tard –
10. il y a forcément plein de `x` dans les droits du fichier `/usr/bin/firefox` : voir plus haut !
11. Pensez par exemple à libreoffice : les données, c'est le texte que vous écrivez.

— `cmd>ps -F -U moi` (remplacez `cmd>moi` par votre nom d'utilisateur) : vous montre tous *vos* processus en vous disant beaucoup de choses.

`cmd>ps -F -U moi|less`

ou

`cmd>ps -F -U moi|head` [12]

vont vous permettre de voir le début de cette liste. La première ligne est :

```
UID   PID    PPID  C    SZ   RSS PSR STIME TTY   TIME CMD
```

Détaillons un peu :

— `UID PID PPID` : on a vu ça plus haut.

— `C` : pourcentage d'utilisation du processeur.

— `SZ RSS` : la taille mémoire réservée par le processus et la taille réellement utilisée.

— `STIME` : date de démarrage du processus

— `TTY` : le terminal depuis lequel la commande a été lancée. Mais si la commande n'a pas été lancée depuis un terminal, on a un " ?".

— `TIME` : le temps UC consommé par le processus depuis son démarrage.

— `CMD` : la commande.

— Avec

`cmd>ps -aef`

on voit de manière étendue les processus de tous les utilisateurs.

Exercice : Commencez par agrandir au maximum votre fenêtre de ligne de commande (en général la touche F1 le fait (F1 aussi pour revenir à un terminal de taille normale)).

Ensuite :

`cmd>ps -aef | less`

Vous devriez voir quelque chose qui ressemble à ça :

```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 déc.13 ?      00:00:12 /lib/systemd/systemd --system --deserialize 16
root           2       0  0 déc.13 ?      00:00:00 [kthreadd]
root           3       2  0 déc.13 ?      00:00:00 [rcu_gp]
root           4       2  0 déc.13 ?      00:00:00 [rcu_par_gp]
root           6       2  0 déc.13 ?      00:00:00 [kworker/0:0H-kblockd]
root           9       2  0 déc.13 ?      00:00:00 [mm_percpu_wq]
root          10       2  0 déc.13 ?      00:00:07 [ksoftirqd/0]
root          11       2  0 déc.13 ?      00:07:10 [rcu_sched]
root          12       2  0 déc.13 ?      00:00:02 [migration/0]
```

Ce qui s'analyse ainsi :

— Le process 1 est `systemd` qui va lancer... beaucoup de choses. Sur d'autres distribution (la mienne est une Ubuntu 20/10) cela peut être `init` (c'est très « Linux canal historique »).

— Si on regarde les processus suivants, on voit qu'ils ont des PID croissants, mais qu'ils sont lancés (PPID) par le processus de PID = 2.

— Supposons que firefox est lancé (sinon, il n'y a qu'à le lancer !).

`cmd>ps -aef |grep firefox`

On obtient à peu près (la sortie est tronquée à droite [13]) :

12. `head` montre tes premières lignes d'un fichier. De même, il ya `tail`...
13. et chez vous, ça va forcément être un peu différent.

```
moi       373641  337755  6 16:58 ?        00:00:13 /usr/lib/firefox/firefox
moi       373755  373641  2 16:58 ?        00:00:06 /usr/lib/firefox/firefox ...
moi       373801  373641  0 16:58 ?        00:00:00 /usr/lib/firefox/firefox ..
moi       373956  373641  1 16:59 ?        00:00:02 /usr/lib/firefox/firefox ...
moi       374054  373641  3 17:00 ?        00:00:04 /usr/lib/firefox/firefox ...
moi       374100  373641  2 17:00 ?        00:00:03 /usr/lib/firefox/firefox ...
```

Ce qui s'analyse ainsi (faire la même analyse chez vous) :

1. Il y a 6 processus. les deuxième et les suivants ont un PPID = 373641, et donc ils ont été lancés par le processus de PID = 373641, qui est le premier de la liste [14].
2. Mais si on regarde la première ligne, on voit aussi que le processus de PID = 373641 au PPID = 337755. Qui donc a lancé ce "premier" firefox ?
   `cmd>ps -aef |grep 337755`
   et vous aurez son "père" [15].

8.2. D'autres commandes associées aux processus. —

8.2.1. Voir les processus tourner :— La commande `cmd>top` vous montre les processus actifs en tête. On peut jouer sur le taux de rafraîchissement, et tuer un processus (`k`).

8.2.2. Arrêter un processus : la commande `cmd>kill`. — Cette commande permet d'envoyer un signal à un processus (en pratique, on tue le processus). La syntaxe est : `kill` suivi d'au moins un espace et le PID du process à tuer [16].

Exercice : Chercher à nouveau les processus firefox comme ci-dessus, et tuer le premier. Chez moi cela donne :

```
cmd>kill 373641
```

En procédant ainsi tue le père... et ses descendants => plus aucun firefox ne tourne. On peut aussi tuer un des ses fils.

8.3. Autres notions sur les processus. —

1. Espace utilisateur et espace noyau : la mémoire est coupée en deux parties, l'espace utilisateur et l'espace noyau. Tous vos processus tournent dans l'espace utilisateur, et ceux du noyau dans l'espace noyau. Vous ne pouvez pas intervenir sur les processus qui tournent dans l'espace noyau : c'est une sécurité supplémentaire. Avec les CPUs Intel, ces protections sont mêmes assurées par le processeurs (les *rings*).
2. Un processus peut créer des processus fils. On l'a déjà vu avec `initd`. Une notion un peu différente est le *fork* ou un processus crée une copie de lui même.
3. Les processus peuvent communiquer entre eux. Il existe pour cela différents mécanismes (exemples : une zone de mémoire partagée, les sockets (une sorte de réseau interne à la machine), le *pipe* qu'on a vu avec la commande `|` etc.
4. Un processus peut créer des *processus légers* appelés *threads*. La différence avec la création de processus classiques, c'est principalement que tous les *threads* et leur père partagent la même zone mémoire. Si la commande `cmd>top` vous montre un programme qui consomme plus de 100% d'UC, c'est que plusieurs threads tournent en parallèle. Vous pouvez installer le package `htop` qui permet de bien visualiser ça. L'intérêt des *processus légers* est la rapidité

14. Pourquoi firefox lance-t-il autant de processus ? probablement pour des questions de performances et/ou de sécurité.
15. chez moi, c'est un processus de xfce4, car j'utilise xfce, pas gnome.
16. Pour les processus récalcitrants, on peut faire `cmd>kill -9` PID.

de la communication entre eux puisqu'il n'y a aucun mécanisme de partage de données à mettre en jeu [17]. Le défaut, c'est que leur programmation est délicate (partage de mémoire => pas de protection d'accès mémoire entre threads).

5. **Les démons (daemons) :** Ce sont des programmes résidents qui sont à priori chargés au démarrage. Quelques exemples : `cupsd` : gestionnaire d'impression, `rsyslogd` : gestion des "logs", `ntpd` : synchronisation de l'horloge avec une horloge distante, etc.

9. Forme (presque générale) des commandes Unix

À titre d'exmple, quelques unes des différentes formes que peut prendre la commande `ls` :

1. `cmd>ls`
2. `cmd>ls /tmp`
3. `cmd>ls /tmp /usr/ /bin`
   (ou `cmd>ls /var/log/*.log`, on verra ça plus loin).
4. `cmd>ls -l /tmp /usr/ /bin`
5. `cmd>ls -l -t -r /tmp`
   mais en pratique on tapera plutôt :
   `cmd>ls -ltr /tmp`

On voit la structure d'une commande (les crochets `{ }` indiquent quelque chose de facultatif) :

`commande {-options} {-options} {arguments}`

Une description plus exacte de la syntaxe nécessiterait d'utiliser la forme de Backus–Naur (voir `https://fr.wikipedia.org/wiki/Forme_de_Backus-Naur` par exemple), ce qui n'est pas vraiment prévu dans ce cours.

On remarque qu'il existe en général une forme simple, sans arguments (1), une forme avec des arguments (2)(3) et qu'on modifie le comportement de la commande (on le complexifie) en ajoutant une ou plusieurs options, le "-" précédant chaque option ou chaque groupe d'options.

Dans certains cas les options peuvent avoir elles aussi des arguments. Par exemple, la commande utilisée pour imprimer un fichier :

— `cmd>lpr chemin_vers_le_fichier`
  imprimera le fichier sur l'imprimante standard.

— `cmd>lpr -P imp2 chemin_vers_le_fichier`
  imprimera le fichier sur l'imprimante de nom `imp2`.

10. Noms de fichiers, méta-caractères, et expressions régulières

Un exemple pour commencer (à tester ; évidemment le choix de `/var/log` n'est là 0 que comme exemple) :

`cmd>ls /var/log`

on voit qu'il y a plein de fichiers dont le nom se termine par `log` (par exemple `boot.log`). Comment ne lister que ceux ci ?

Pour cela, on utilise le méta-caractère "`*`" :

`cmd>ls /var/log/*.log`

17. C'est en général là-dessus que repose la parallélisme en mémoire partagée : si vous utilisez des logiciels de vidéo par exemple, ils sont en général multithreadés, tant le genre de calcul qu'ils font est coûteux. Avec n cœurs, on peut en théorie calculer n fois plus vite qu'avec un seul.

et on ne voit plus qu'eux.
Explication :

— "`*`" correspond à n'importe quelle chaîne de caractères (formée de caractères alphabétiques, numériques et autres). La commande

```
cmd>ls /var/log/*.log
```

va lister tous les fichiers qui sont dans `/var/log` dont le nom commence par n'importe quelle chaîne de caractère, mais finit par `.log`.

— Mais ce n'est qu'une particularisation de quelque chose de plus vaste : les *expressions régulières*. Avec les « bons » outils, on peut, par exemple, trouver dans un texte (dans une chaîne de caractères) tous les *mots* qui se terminent par `.log`.

☛ Quelques autres exemples d'utilisation du jocker "`*`" :

— On peut lister tous les fichiers dont le nom commence par exemple par `syslog` :

```
cmd>ls /var/log/syslog*
```

— Dans ce cas précis, on voit que certains fichiers dont le nom commence par `syslog` ont une extension `.gz` (fichiers compressés).

Pour ne lister qu'eux, je peux faire :
`cmd>ls /var/log/syslog*gz` ou dans notre cas `cmd>ls /var/log/syslog.*.gz`.

> — *Le méta-caractère "`*`" correspond à n'importe quelle chaîne de caractères, de longueur quelconque.*
> — *Le méta-caractère "`?`", lui, correspond à un seul caractère, quelconque.*

Exemple (à tester) :
`cmd>ls /var/log/syslog.?.gz`
ou bien
`cmd>ls /var/log/syslog???gz`
dans ce cas, les 3 caractères "matchés" par `???` peuvent être chacun quelconques et différents.

10.0.0.1. ☛ D'autres manières de filtrer :—

— Supposons que je veuille filtrer sur un seul caractère, numérique. Je me sers alors d'intervalle de recherche `[0-9]`, puis :

```
cmd>ls /var/log/syslog.[0-9].gz
```

`[0-9]` va *matcher* tous les caractères de l'intervalle, c'est à dire les chiffres de 0 à 9.

— Si je veux filtrer sur les noms qui commencent par `a`, `b` ou `c`, je peux faire :

```
cmd>ls /var/log/[a-c]*
```

— Bien sûr, on peut tout mixer :

```
cmd>ls /var/log/[a-c]*.[0-9].gz
```

et aussi être plus restrictif :

```
cmd>ls /var/log/[a-c]*.[1-3].gz
```

Attention : l'ordre des caractères alphabétiques est `ABC....Zabc...z` et par conséquent `[a-B]` est un intervalle vide, comme le serait `[3-1]`. En revanche `[A-z]` est l'ensemble de tous les caractères alphabétiques [18].

On peut évidemment utiliser cette technique avec à peu près toutes les commandes. Exemples :

18. C'est vrai tant qu'on se cantonne à l'utilisation des caractères ASCII ; aujourd'hui on peut représenter « tous » les caractères (Unicode).

```
cmd>rm *.log
cmd>mv toto.* /tmp
```
(19)

> Ceci n'est qu'une infime introduction à un monde qu'il serait bon de connaître pour traiter des chaînes de caractères : *les expressions régulières.*

Mais comment est-ce que ça marche ? Il faut en dire un peu plus sur le mécanisme du shell. Que se passe-t-il quand on tape une commande ?
Réponse : la commande est interprétée par le shell :

1. Ce qu'on tape est une chaîne de caractères (se terminant par le caractère "retour chariot").
2. Le shell (bash ou autre) analyse cette chaîne. Il peut y avoir des erreurs lexicographiques (emploi de caractères pas reconnus) ou syntaxiques.

   Essayez par exemple :

   ```
   cmd>&ls
   ```
3. Une fois cette étape passée, shell extrait le premier mot de la ligne, considérant que ce mot se termine au premier caractère "espace", et considère que c'est une commande. Il cherche alors cette commande (voir plus loin, page 18) ; s'il ne la trouve pas, il vous le dit, sinon il lance la commande en lui passant le reste de la ligne comme *argument*. Exemple :

   ```
   cmd>ls -l /tmp
   ```

   la commande `ls` reçoit "`-l /tmp`" (20).
4. Mais si la ligne de commande contient une expression régulière (au lieu de simplement `/tmp`), celle-ci est évaluée et c'est la chaîne résultante qui est passée à la commande : par exemple, si dans mon répertoire courant il existe les fichiers `toto.txt` et `toto.pdf`, la commande

   ```
   cmd>ls -l toto.*
   ```

   est évaluée en :

   ```
   cmd>ls -l toto.pdf toto.txt
   ```

   et c'est la chaîne de caractères "`-l toto.pdf toto.txt`" qui est passée à `ls`.
5. Là s'arrête le travail du shell proprement dit : il passe la main à la commande `ls`, créant ainsi un processus dont il est le père, et dont il garde le contrôle ; `ls` analyse cette ligne : est elle *bien formée ?* (la syntaxe des options est-elle correcte, etc.) : le processus `ls` va faire ce qu'il a à faire, et finalement renvoyer au shell un message "tout s'est bien passé", ou bien un message d'erreur, et le shell reprend complètement la main, et attend de vous une autre commande.♦

   Derrière tout ça il y a la notion importante, mais hors de l'horizon de ce cours, de *grammaire formelle* et d'*analyse syntaxique*. Le shell est un interpréteur de commande sophistiqué qui permet d'écrire des programmes dans un langage bien défini, qui suit une *grammaire* bien précise.

19. `mv` : déplace, mais aussi renomme.
20. Notons que les espaces en trop sont supprimés : plusieurs espaces successifs se comportent comme un seul.

11. Deux manières d'acquérir des pouvoirs exceptionnels

Comme on l'a vu, les utilisateurs « standard » n'ont pas tous les pouvoirs [21], mais il est nécessaire ne serait-ce que pour administrer la machine (ce qui inclut par exemple les mises à jour) d'acquérir des pouvoirs exceptionnels.

Les distributions Linux actuelles proposent deux méthodes :

— *Le superutilisateur :* C'est un utilisateur de nom `root`. Il a, par définition, tous les droits. Donc, il est dangereux d'être `root` [22].

— *La commande* ***`sudo`*** *:* elle permet d'exécuter une commande en étant root, le temps que dure la commande :

```
cmd>sudo rm quelque-chose
```

par exemple.

L'idée, avec `sudo` est de ne donner que temporairement les privilèges de root, en contrôlant le plus possible ce qui est fait, car :

— Il faut avoir le droit de faire "sudo'' ; pour cela il faut être membre du *groupe sudo :*

```
cmd>grep sudo /etc/group
```

pour voir ça.

— Il faut donner son propre mot de passe.

— Le mot de passe vous donne le bien droit de faire "sudo", mais ce droit est limité dans le temps (il faut redonner le mot de passe au bout d'un certain temps).

Regardez ce que donnent :
`cmd>whoami`
et
`cmd>sudo whoami`
Évidemment, la sécurité de "sudo" est assez illusoire.

L'utilisateur "root " est toujours présent : les utilisateurs d'Ubuntu –par exemple– pourront le vérifier en regardant le fichier `/etc/shadow` [23] où un '' !" désactive la possibilité de se connecter.

En fait, c'est assez dépendant des distributions Linux. Sous les Debian, sudo n'est pas installé par défaut, et root est un utilisateur comme les autres, avec un mot de passe. Sous Ubuntu on installe sudo par défaut [24], mais il suffit de donner un mot de passe [25] à l'utilisateur root pour pouvoir aussi se connecter en tant que root (depuis l'écran de connexion par exemple).

Notez enfin que `cmd>sudo -i`, c'est exactement comme s'être connecté en tant que root.

Quelques avantages de sudo :

— On contrôle (via le groupe sudo) qui peut acquérir les privilèges de `root`.

— Il faut déjà être connecté pour pouvoir faire sudo.

— On peut configurer assez finement ce qui est autorisé pour les utilisateurs qui peuvent faire sudo (voir les fichiers et répertoires dont le nom commence par sudo dans `/etc`).

---

21. Entre autres, ils n'ont pas accès à certains fichiers ou répertoires, ou bien des accès restreints (lecture seule par exemple).
22. ne jamais être root en état d'ivresse.
23. `cmd>sudo grep root /etc/shadow`, bien sûr !
24. `cmd>apt install sudo` doit l'installer ; il faudra ensuite ajouter le ou les utilisateurs ad hoc au groupe `sudo`.
25. Question : comment faire ? La commande pour changer *votre* mot de passe est `cmd>passwd`. Pour changer ou définir celui de `root`, il faut... être `root`.

Comment devenir `root` quand la machine n'utilise pas `sudo` ? On peut évidemment se déconnecter et se reconnecter `root` qui est un utilisateur presque comme les autres, ou bien sans se déconnecter, utiliser la commande `su` :

```
man su
                run a command with substitute user and group ID
```

Dans un terminal, on tape :

— `cmd>su -`

Il faut donner le mot de passe de root. Le "-" permet de tout faire comme dans un vrai login (initialiser le shell (cf. page 17).

— `cmd>su - commande`

comme pour `sudo` (mais `su` ne se souvient pas de votre mot de passe).


Encore une fois : ATTENTION !

12. Le shell : configuration, environnement

1. Dans le home directory, quelques fichiers cachés [26] sont importants.

Commande	⇒	Résultat (chez moi)
`cmd>ls ~/.*bash*`		/home/moi/.bash_history /home/moi/.bashrc /home/moi/.bash_logout

Et il existe aussi, toujours dans le home directory, un fichier de nom `.profile`

2. Dans `/etc`

Commande	⇒	Résultat (chez moi)
`cmd>ls /etc/bash*`		/etc/bash.bashrc /etc/bash_completion /etc/bash_completion.d : apport_completion git-prompt gmic

et

Commande	⇒	Résultat (chez moi –abrégé–)
`cmd>ls /etc/profi*`		/etc/profile /etc/profile.d : 01-locale-fix.sh gawk.sh vte-2.91.sh bash_completion.sh

À quoi tout cela sert-il ?

— À définir des *variables d'environnement.*

— À définir ou à modifier des commandes.

— Ce qui a un nom contenant `completion` sert à definir les règles de complétion automatique (quand on tape un TAB).

Les variables d'environnement sont globales et peuvent être utilisées par les programmes pour leur permettre d'adapter leur comportement. Un exemple typique : la langue utilisée.

26. Les fichiers et les répertoires cachés ont un nom qui commence par ".". La commande `cmd>ls` ne les montre pas, il faut ajouter l'option `a` : `cmd>ls -a` pour les voir.

— les fichiers `profile` sont indépendants du shell utilisé.
— L'ordre de parcours est :
  — d'abord le(s) fichiers `profile` de `/etc`,
  — puis `/etc/bash.bashrc`,
  — puis le fichier `~/.profile`
  — puis `~/.bashrc`.

On peut donc, dans les fichiers `~/.profile` et `~/.bashrc` compléter ce qui est fait dans `/etc` ou redéfinir des variables.

Qu'est-ce qui est défini ?— La commande
`cmd>printenv`
permet de voir tout ça. Il y a beaucoup de variables définies !
Remarquons entre autres :

— `SHELL=/bin/bash`
  Ici, la variable est `SHELL` et elle contient `/bin/bash`. C'est le shell qu'on utilise bien sûr.
— Des variables dont le nom commence par `LC` : définissent la langue utilisée et les jeux de caractères utilisés.
— etc.

On affecte des valeurs dans le shell en faisant par exemple :
`cmd>x="coucou"`
mais pour voir ce que contient la variable, il faut faire :
`cmd>echo $x`

Une variable importante : le `path`. — Essayez :
`cmd>echo $PATH`
ou
`cmd>printenv|grep PATH`
On obtient une liste de chemins dans le système de fichiers, ou les chemins sont séparés par `:`. Exemple (chez moi) :
`/home/moi/.local/bin:/usr/local/sbin:/usr/local/bin:`
`/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin`

— Quand on tape une commande, le shell va la chercher successivement dans chacun des chemins de `PATH`, dans l'ordre, et exécute la première trouvée.
— On peut modifier le `PATH`. Pour rajouter un chemin, on peut faire (ici je rajoute `/opt/`) :
  `cmd>PATH=/opt/:$PATH` (ajout au début)
  ou :
  `cmd>PATH=$PATH:/opt/` (ajout à la fin).
  `cmd>echo $PATH` pour voir le résultat.
  C'est typiquement le genre de commande qu'on peut rajouter dans `~/.bashrc`.

13. Quelques fichiers et répertoires cachés importants

Certains logiciels créent des répertoires et des fichiers cachés précieux. Attention à ne pas les effacer ! Entre autres :

— `.mozilla/` : répertoire de firefox. Toute votre vie avec firefox !
— `.mozilla-thunderbird/` : le répertoire de thunderbird.
— `.ssh/` si vous utilisez ssh.
— `.VirtualBox/` si vous utilisez Virtualbox.

Le répertoire `.config/` : contient dans des répertoires les configurations de la plupart des logiciels que vous installez. Si vous configurez des logiciels avec les menus (Préférences), vos modifications vont là-dedans. Attention, donc ! Mais il peut être utile d'effacer un répertoire pour « repartir à zéro » avec une application.

14. Une liste de commandes Unix

Commande	Objet	Exemple	Options importantes, remarques
diff	Différences entre fichiers	`cmd>diff a b`	utiliser entre fichiers texte
file	Devine le type de fichier	`cmd>file *.pdf`	
mv	Déplace, renomme (voir plus bas)		
sort	Trier un fichier	`cmd>sort fich`	ajouter `-n` pour un tri numérique
touch	modifie la date ou crée un fichier vide	`cmd>touch toto`	
chmod	change les droits (voir plus bas)		
which	trouver une commande dans le PATH	`cmd>which uname`	
whoami	votre login		
uptime	temps d'activité (depuis le reboot)		
date	date et heure		plein d'options (man date)
df	espace disque utilisé et libre		`cmd>df -h`, sortie « humaine »

14.0.0.1. Quelques détails :—

— `mv` : déplacer ou renommer. C'est assimilé à une écriture, et donc il faut avoir le droit d'*écrire* sur l'objet qui peut être un fichier ou un répertoire.

Exemples :

— `cmd>mv toto /tmp` déplace le fichier ou le répertoire `toto` dans `/tmp` (il faut aussi avoir le droit d'écrire dans le répetoire but (ici, avec `/tmp`, c'est le cas).

— `cmd>mv toto /tmp/tutu`, on déplace `toto` dans `/tmp` où il s'appelera `tutu`.

— `cmd>mv toto machin`. Le déplacement se réduit à un changement de nom.

— `chmod` : change les droits d'un fichier ou d'un répertoire. Il y a plusieurs façons de procéder, une assez facile, l'autre moins !

— Pour changer les droits du propriétaire, l'option est `u`, c'est `g` pour celle du groupe et `o` pour le reste du monde. Quelques exemples :

1. `cmd>chmod u+w toto`,
2. `cmd>chmod u+x g+x o-w toto`,
3. On peut « factoriser » les ordres : `cmd>chmod ug+x o-w toto` par exemple.
4. La méthode « dure » : on considère les 9 digits possibles `xxxyyyzzz` ; on met un 1 quand on veut que le droit soit ouvert, 0 sinon ; ainsi, pour obtenir `rw-r---`, ceci correspond à : `110 100 000`. Bien maintenant, il s'agit de 3 nombres binaires ; il faut les calculer en base 10 :

   (a) `110`$= 1 \times 4 + 1 \times 2 + 0 = 6$,

   (b) `100`$= 1 \times 4 + 0 \times 2 + 0 = 4$

   (c) `000`$= 0$.

   Et la commande est `cmd>chmod 640 toto`.

   Amusant, non ? (excellent exercice de calcul mental).

15. Bibliothèques (libraries)

Une bibliothèque est un ensemble de pièces de programmes qui peuvent être réutilisées dans un programme. L'intérêt est qu'elles fournissent des fonctions, des structures qui peuvent être utilisées par différents programmes (voir la figure 1 et, par exemple, la référence [4]).

Exemple : la bibliothèque `libc` (plus exactement `glibc`) fournit un grand nombre de fonctions plutôt basiques ; une fonction comme `time` qui permet de récupérer dans un programme l'heure, telle qu'elle est connue par la machine. Un comprend donc l'intérêt a avoir une seule version de cette méthode.

15.1. Bibliothèques statiques et bibliothèques dynamiques. —

— Avec les bibliothèques statiques, les fonctions dont un programme a besoin sont prises dans les bibliothèques au moment de la construction du programme (édition des liens). Le fichier du programme (exemple `/usr/bin/firefox`) contient à peu près tout ce qu'il faut pour faire tourner le programme.

— avec les bibliothèques dynamiques, on va charger en mémoire des fonctions dont on a besoin au moment de leur utilisation.

Quelles sont les bibliothèques dynamiques utilisées par un programme ?— La commande est `ldd`. Exemples :

— `cmd>ldd /usr/bin/firefox`. Réponse :

`n'est pas un exécutable dynamique`. Donc on utilise des bibliothèques statiques.

— `cmd>ldd /bin/ls`

```
   linux-vdso.so.1 (0x00007fff9af8b000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007ff135523000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff13551d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff135333000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff1356b9000)
moi@kepler:/2/home/moi/CoursLinux/Cours/Cours4$ man ldd
moi@kepler:/2/home/moi/CoursLinux/Cours/Cours4$ ldd /bin/ls
linux-vdso.so.1 (0x00007ffd3351f000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fdeb6035000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fdeb5e4b000)
libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007fdeb5dbb000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fdeb5db5000)
/lib64/ld-linux-x86-64.so.2 (0x00007fdeb60c0000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fdeb5d93000)
```

On voit entre autres, que l'on utilise la `libc` (`libc.so.6`), ce qui est normal, car cette bibliothèque contient beaucoup d'utilités de base.

Les intérêts des bibliothèques sont nombreux : ne pas redévelopper ce qui existe, et utiliser des "services" partagés entre le plus grand nombre de programmes améliore la fiabilité de ces services.

On va voir d'autres bibliothèques quand on va étudier les interfaces graphiques.

Un autre exemple : résoudre le système d'équations linéaires ci-contre.
On peut évidemment coder ça à *la main*, mais le mieux est d'utiliser une bibliothèque libre (en pratique : `lapack`) : plus de 30 ans d'expérience, fiabilité et performances imbattables
Intéressé(e) par cet exemple ? voir en appendice page 31.

$$\begin{aligned} 2x + \ y + 3z &= 10 \\ x + \ y + \ z &= \ 6 \\ x + 3y + 2z &= 13 \end{aligned}$$

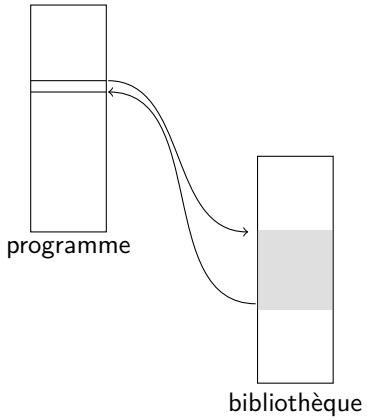


FIGURE 1. Programme et bibliothèque

Nommage des bibliothèques dynamiques. — Pour la `libc` qui est un bon exemple, on voit : `/usr/lib32/libc.so.6`

— L'extension `so` est celle des bibliothèques dynamiques.

— Le `.6` est un numéro de version.

16. Interface graphique, bureau etc.

Comment fonctionnent les systèmes de bureau, d'interfaces graphiques que nous utilisons ? C'est assez complexe, ne serait-ce que parce qu'on hérite d'un passé assez lointain (les années 80). La complexité de ces outils outrepasse celle du noyau Linux. Et puis depuis les débuts de ces développements l'environnement matériel a beaucoup changé : les GPU n'implantaient pas les fonctionalités actuelles.

Une remarque préalable : vous pouvez utiliser une machine Linux sans interface graphique. Tapez `CNTRL+Alt F1` et les plus anciens se retrouveront face à quelque chose qui leur rappellera le DOS.

Linux offre plusieurs environnements de bureau permettant tous de gérer des fenêtres des menus, des éléments graphiques etc. Citons : Gnome, Xfce, Mate, Ukui, KDE, mais il en existe d'autres.

Schématiquement, un système de bureau comporte au moins 3 couches, chacune incarnée par une ou plusieurs bibliothèques :

1. Un niveau *bas* qui sait créer des fenêtres, interagir avec le clavier et la souris du coté de l'utilisateur et, du coté machine avec des dispositifs gérant directement l'affichage. actuellement ce niveau peut être incarné par deux logiciels :
   (a) X11, *The X-Window system.*
   (b) Wayland.
2. Un niveau intermédiaire, capable de gérer des boutons, des menus déroulants, le couper-coller, etc. C'est le plus souvent la bibliothèque GTK, développée à l'origine pour le logiciel de manipulation d'images GIMP.
3. Le niveau supérieur : c'est gnome, Xfce etc, etc. mais en fait c'est un peu plus compliqué : Xfce par exemple (et certainement les autres aussi s'interfacent avec X11 et GTK.

16.1. X11 ou Wayland ?— X11 est ancien (le développement a commencé en 1984), lourd et réputé plus lent que des outils équivalents sous Windows et MacOs. D'où l'idée de développer quelque chose de plus léger et de plus moderne. Il y a eu plusieurs propositions et il semble que Wayland l'emporte.

Certaines distributions utilisent maintenant Wayland (Debian), d'autres garde X11 (Ubuntu). Il faut savoir que pas mal d'applications s'interfacent directement avec X11 et que, pour les faire fonctionner sous

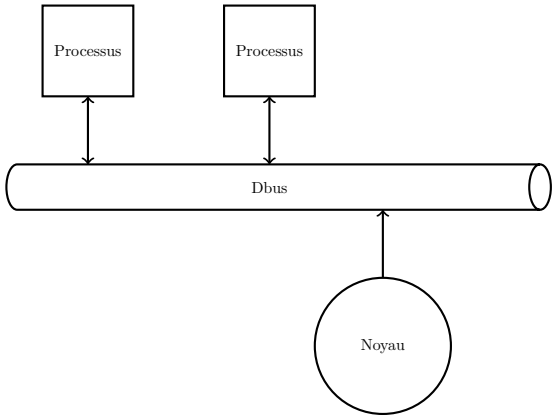


FIGURE 2. Dbus

Wayland, il a fallu développer... un serveur X11 sous Wayland, au moins pour une période transitoire. X11 permet aussi l'affichage déporté (un programme tourne sur une machine, l'affichage est déporté sur une autre) : l'intéret pour cette fonctionnalité diminue : on a longtemps utilisé des terminaux X, désormais obsolètes, et un outil comme x2go permet un bien meilleur affichage à distance (il est basé sur les techniques de compression d'images utilisées pour la télévision numérique).

Regardons un peu le cas Xfce (les autres sont à coup sûr peu différents) :
La commande :
`cmd>ps aux|grep -i xfce`
liste plusieurs processus. C'est donc que :

— Xfce n'est pas monolithique.

— Il va obligatoirement que ces processus communiquent entre eux.

Parmi ces processus, on peut regarder `xfce4-session` qui est sûrement important. La commande
`cmd>ldd /usr/bin/xfce4-session`
liste 82 bibliothèques ! Explicitons en quelques unes :

— `libX11` : c'est bien sûr le fameux X11. Notons aussi d'autres bibliothèques qui ensemblent forment X11 : libXrender, libXext etc. (tout ce dont le nom commence par libX).

— `libgtk` : définit principalement des « widgets » (boutons, menus etc.).

— `libgdk` : « en dessous » de `gtk`. C'est l'intermédiaire entre `gtk` et le système de fenêtrage (X11 ou Wayland ou...).

— `libpango` : gestion des polices de caractères.

— `libcairo` : dessins bi-dimensionnels, vectoriels. Utilise l'accélération graphique, si c'est possible.

— `libwayland` : la bibliothèque est présente bien que la machine utilise x11.

— `libdbus` : c'est une bibliothèque de communication. On en parle ci-dessous.

Dbus. — Dbus est un *bus logiciel*. Citons Wikipédia : « *D-Bus permet à des programmes clients de s'enregistrer auprès de lui, afin d'offrir leurs services aux autres programmes. Il leur permet également de savoir quels services sont disponibles. Les programmes peuvent aussi s'enregistrer afin d'être informés d'événements signalés (parce que gérés) par le noyau, comme le branchement d'un nouveau périphérique* » (voir figure 2).

Il faut mentionner le projet freedesktop.org[6] qui vise à l'interopérabilité des différents environnements de bureau, en définissant des standards et en développant des bibliothèques logicielles (Cairo, par exemple).

17. Les liens

Les liens sont des types d'objets définis sous Unix. Ils permettent d'obtenir une vue d'un fichier ou d'un répertoire « comme s'il était là ». Un lien utilise très peu de ressources.

Sur ma machine, la commande

`cmd>ls -l /`

donne (extrait) :

```
lrwxrwxrwx   1 root root     7 avril 27  2020 lib -> usr/lib
lrwxrwxrwx   1 root root     9 avril 27  2020 lib32 -> usr/lib32
lrwxrwxrwx   1 root root     9 avril 27  2020 lib64 -> usr/lib64
lrwxrwxrwx   1 root root    10 avril 27  2020 libx32 -> usr/libx32
```

`lib` est un *lien symbolique* vers le répertoire `usr/lib`.

On peut vérifier ensuite que les commandes `cmd>ls /lib` et `cmd>ls /usr/lib` donnent le même résultat.

Un lien symbolique :

— S'installe avec la commande `cmd>ln -s`. Exemple :

`cmd>ln -s /usr/lib lienVersLib`

`lienVersLib` pointe alors vers `/usr/lib`.

— Peut pointer vers n'importe quoi : fichier ou répertoire.

— S'efface avec la commande `rm`. Exemple :

`cmd>rm lienVersLib`.

—  Effacer le lien est sans problème (ici `lienVersLib`), mais si on efface l'objet pointé, le lien persiste, mais ne pointe plus sur rien ! Exemple :

`cmd>touch toto`

`cmd>ln -s toto tutu` (`tutu` est un lien vers `toto`).

`cmd>rm toto` (`tutu` ne pointe plus sur rien).

`cmd>less tutu`

`tutu: Aucun fichier ou dossier de ce type`

Il existe d'autres sortes de liens, les liens « hard », assez peu intéressants pour les utilisateurs :

— Ils ne peuvent pointer que vers des fichiers.

— Ils équipent le fichier pointé d'un compteur de référence ; ainsi :

`cmd>ln toto tutu` (tutu est un lien « hard » vers `toto` (notez l'absence de l'option `-s`). La commande `ls` montre 2 fichiers, alors que les données n'existent qu'une seule fois.

`cmd>rm toto`

`cmd>less tutu` : tout se passe bien.

18. Systèmes de fichiers

Définition approximative : une organisation hiérarchiques de fichiers et de répertoires.

C'est une définition à peu près indépendante du support matériel de ces systèmes (voir l'article de Wikipédia [5]).

En pratique sous Unix, à chaque fichier ou répertoire est associé un inode, qui contient tous les renseignements le concernant. Chaque inode a un numéro, unique. La commande :

`cmd>ls -i objet`

permet d'obtenir le numéro d'inode de `objet` (ce qui ne sert pas à grand chose).

Mais les systèmes de fichiers, s'ils montrent au système le même type de structure (arborescence de fichiers et répertoires), ne stockent pas tous en interne les données de la même façon (une bibliothèque est associée à chaque type de système de fichiers), d'où des fonctionalités et des performances différentes. Citons, pour Linux :

— La famille `ext` : `ext1`, `ext2`, `ext3` et `ext4`. Les premières versions (1 et 2) ont été réalisées par Remy Card, intervenant fréquent aux premières Journées du Logiciel Libre à Lyon [3]. C'est le système de fichiers standard sous Linux.

— `Jfs` : (IBM ; peu gourmand en ressources).

— `ReiserFS` : semble abandonné.

— `Xfs` : très grande capacité de stockage, et rapide pour les très grands systèmes de fichiers.

— `Btrfs` : selon les auteurs, c'est le *B-Tree file-system* ou le *Better file system*. Génial sur le papier, mais ne semble pas percer pour l'instant (standard cependant chez Suse).

— `Zfs` : (Open-Zfs). Origine SUN, assez proche de `Btrfs` qui semble en dériver, en tout cas conceptuellement.

— Les systèmes Linux peuvent aussi manipuler (et créer) des systèmes de fichiers propriétaires (`ntfs`, `vfat`, `fat` etc.).

— Systèmes de fichiers en réseau : une machine *exporte* une partie de son arborescence vers d'autres machines ; sous Unix (et donc Linux), cela s'appelle `nfs` (network file system) ; `samba` fait ça aussi.

— Il existe aussi des *clustered file-systems* pour la lecture et l'écriture en parallèle (Lustre, BeeGFS, GPFS, Xtremefs, GlusterFS, MooseFS, XrootD, EOS...) ; ça c'est pour Youtube ou le Centre de Calcul des Physiciens Nucléaires à La Doua [27].

Quelques détails :—

— `Btrfs` et `Zfs` permettent de faire des *snapshots*, soit en français des *instantanés* : obtenir une *vue* du système de fichiers tel qu'il est à un instant donné, en le laissant évoluer par ailleurs. On trouvera en annexe une description (à coup sûr simplifiée) de la technique utilisée (le COW).

— `Btrfs` et `ReiserFS` utilisent tous les deux une représentation des données sous forme d'*arbre binaire équilibré* (*B-Tree*), décrite aussi en annexe, qui assure une très grande vitesse de lecture.

Un des problèmes majeurs est la fiabilité des systèmes de fichiers : outre qu'un bug pourrait avoir des conséquences désastreuses, il faut absolument tenir compte de l'accident qu'un arrêt brutal de la machine (coupure de courant par exemple) pourrait causer. Le risque est d'avoir un système de fichiers incohérent, car l'interruption aura eu lieu alors qu'une écriture était en cours (apparition d'orphelins par exemple). Regardons pour cela l'histoire des systèmes `ext[1-4]` :

— `ext1` était un prototype ; `ext2` (Rémy Card, 1993 [3]) est rapidement devenu le système standard sous Linux. Il limite la fragmentation du support, n'implante pas de sécurité et a fini par apparaître comme limité quant à la taille et au nombre des objets stockés.

— `ext3` (2001) : le principal apport est la *journalisation* qui permet de *rejouer* une E/S qui aurait été interrompue.

— `ext4` (2008) : la limite de la taille est portée à 2^{60} (!) octets. Une autre avantage est de réduire encore la fragmentation. C'est actuellement le système de fichiers sous la quasi totalité des distributions Linux.

18.0.0.1. La journalisation :— On consultera [7] pour une description détaillée. Le principe de la *journalisation* consiste à maintenir une liste (en général dans un fichier) des modifications apportées au système de fichiers, de manière indépendante des modifications elles-mêmes, pour pouvoir rejouer ces modifications en cas de crash, ou au moins de remettre le système dans un état cohérent. C'est ce qui est implanté dans `ext3` (avec différentes options, voir référence ci-dessus).

27. Ou pour le futur télescope à grand champ Vera-C. Rubin : 20 téraoctets par nuit, à conserver au moins 10 ans [2].

18.1. Manipulation des systèmes de fichiers (partitionner,créer, monter, etc.)— Tout, ici, doit être fait en tant que `root`, donc  **Attention, danger !**

On ne va parler que des systèmes de fichiers classiques, physiques (`ext4`, `vfat` etc. Pour `btrfs`, voir par exemple[1]).

1. Un système de fichiers s'installe toujours dans une *partition.* Même si le périphérique sur lequel vous allez installer le système de fichiers ne va en contenir qu'un, vous devez partitionner.
2. On crée ensuite le système de fichiers en utilisant la commande `mkfs` qui installe dans la partition la structure de donnée nécessaire, prête à recevoir vos fichiers et répertoires.
3. Ensuite il faut *monter* le système de fichiers pour qu'il soit incorporé à l'arborescence de la machine (ceci vaut aussi pour les système de fichiers en réseau).

18.1.1. Créer une (des) partitions(s), formater. — En ligne de commande, la commande est `fdisk`. Un outil graphique, plus simple à utiliser, est `gparted`.

La première chose à faire est de connaître le nom du *device* (périphérique à formater). Répétons : il faut être `root`, donc :

— soit vous êtes connecté en tant que `root` (par `cmd>su -` par exemple),

— soit vous préfacez toutes les commandes ci-dessous par `sudo` si votre machine est amie de sudo. Vous pouvez aussi faire une fois pour toutes, dans ce cas : `cmd>sudo -i`.

Dans tous les cas :  !

Pour connaître le périphérique que vous voulez formater vous pouvez par exemple, dans le cas d'un périphérique usb, regarder ce que donne la commande :

`cmd>dmesg`

après avoir introduit le périphérique. Vous verrez par exemple quelque chose comme :

`[sdd] Attached SCSI removable disk`.

Vous pouvez aussi installer et utiliser la commande `lsscsi`, bien pratique. Il faut ensuite taper une commande comme `df` pour voir si le périphérique est monté. Si oui, le démonter (voir plus bas).

On lance la commande

`cmd>fdisk /dev/sdd` (si votre périphérique est bien `/dev/sdd`). Ensuite, la commande est interactive ; il faut :

1. Effacer les partitions existantes (`p` pour les lister, `d` pour en effacer une).
2. `n` pour créer une partition (il y en a différents types, mais utilisons celle par défaut pour linux). On crée une partition primaire, et on en donne la taille (par défaut on utilise tout le périphérique).
3. On peut comme ça créer plusieurs partitions, qui vont s'appeler `/dev/sdd1`, `/dev/sdd2` etc (parce que mon périphérique est `/dev/sdd`).
4. `w` pour écrire les résultats ; En fait jusque là, on n'a rien changé sur le disque. En sautant cette étape, on peut encore tout sauver !
5. `q` pour quitter.

18.1.2. Installer un système de fichiers (formatter). — C'est très simple. Mes partitions s'appellent par exemple `/dev/sdd1`, `/dev/sdd2` etc. Pour installer un système de fichiers `ext4`, il suffit de faire :

`cmd>mkfs -t ext4 /dev/sdd1` (et bien sûr remplacer 1 par ce qu'il faut le cas échéant...).

18.1.3. Monter le système de fichiers. — Monter un système de fichiers consiste à *accrocher* un système de fichiers à un répertoire existant dans l'arborescence. Je peux utiliser un répertoire *existant* (attention, je vais cacher ce qui est éventuellement dedans), ou en créer un. Supposons que j'utilise `/mnt`. La commande est alors :

`cmd>mount -t ext4 /dev/sdd1 /mnt`

Il y a 3 arguments :

1. le type de système de fichiers, ici : `-t ext4`,
2. ce qu'on monte, ici : `/dev/sdd1`,

3. où on le monte, ici : `/mnt`.

Et voilà.

Dans notre cas, `/mnt` est accessible. Ensuite, rien ne prouve que le répertoire ainsi monté sera accessible à tout le monde : à priori, il va appartenir à `root`. Il faut donc en changer les droits, ou le propriétaire. On va voir une autre façon de faire.

18.1.4. Démonter le système de fichiers. — Il faut utiliser `umount`. Dans l'exemple ci-dessus on peut soit faire :

```
cmd>umount /mnt
```

soit :

```
cmd>umount /dev/sdd1
```

18.2. Le fichier `/etc/fstab`. — Si on souhaite automatiser le « mount » de systèmes de fichiers, il faut utiliser le fichier `/etc/fstab`, et donc le modifier [28].

`/etc/fstab` contient la description de tout ce qui doit ou peut être monté, en précisant quoi doit être monté et où.

18.2.1. `fstab`, à l'ancienne. — Exemple :

```
# <file system>  <mount point>  <type>  <options>          <dump>  <pass>
/dev/sda2        /              ext4    errors=remount-ro  0       1
/dev/sda1        /boot/efi      vfat    umask=0077         0       1
```

Le premier champ est ce qu'on monte, le deuxième où on le monte et le troisième le type de système de fichiers. On voit ainsi que la partition `/dev/sda1`, formatée en `vfat` sert pour le boot (c'est uefi). Passons sur le champ "dump", le dernier champ dit l'ordre dans lequel les systèmes de fichiers sont vérifiés au boot. Les options sont nombreuses, ici :

- `errors=remount-ro` dit de remonter sans droits d'écriture en cas d'erreur,
- `umask=0077` indique les droits qu'on enlève : ainsi 077 laisse tous les droits `rwx` au propriétaire (root), et enlève tous les droits (car 7 en binaire est 111) au groupe et aux "autres". Les droits de `/boot/efi` sont donc `rwx------`.

18.2.2. `fstab`, plus moderne. — Citer les noms des partitions comme `/dev/sda1` est source d'erreur. Le système propose un identificateur unique pour chaque partition, qu'on peut obtenir avec la commande `blkid`. Chez moi, cela donne :

```
cmd>blkid /dev/sda1
/dev/sda1: UUID="4454-89BE" BLOCK_SIZE="512" TYPE="vfat" PARTLABEL="EFI System Partition"
PARTUUID="ad32b00b-6810-45f6-93d2-d946eb611524"
```

```
cmd>blkid /dev/sda2
/dev/sda2: UUID="c447c9ae-2491-4526-8fd8-8a56f79b2e0b" BLOCK_SIZE="4096" TYPE="ext4"
PARTUUID="2eb752b3-d027-4383-b811-48f999adf3a2"
```

on récupère les `UID=...` et le fichier `/etc/fstab` devient :

```
# <file system>                             <mount point>  <type>  <options>          <dump>  <pass
UUID="c447c9ae-2491-4526-8fd8-8a56f79b2e0b"  /              ext4    errors=remount-ro  0       1
UUID="4454-89BE"                            /boot/efi      vfat    umask=0077         0       1
```

28. Il faut bien sûr avoir les pouvoirs de `root`. Ensuite, toucher à ce fichier est dangereux. Il faut :
    1. Le copier pour en garder la version actuelle : `cmd>cp /etc/fstab /etc/fstab.save`
       C'est une recommandation qui vaut pour tous les fichiers « sensibles ».
    2. Utiliser un éditeur de texte pour le modifier. Vous en avez plusieurs à disposition : `vim` (pour les masochistes), `gedit` ou `nano`, voir même `emacs`. Pour ce genre de chose, je recommande `nano`.

18.2.3. Autoriser un utilisateur à monter un système de fichiers. — Dans notre cas (monter `/dev/sdd1` sur `/mnt`), la ligne a ajouter à `/etc/fstab` est :

```
/dev/sdd1  /mnt  ext4  noauto,user   Ici :
```

— `user` donne le droit de monter le système de fichiers aux utilisateurs (et donc aussi de démonter).

— `noauto` : le système de fichiers ne sera pas monté au moment du boot.

On peut aussi récupérer l'UUID et écrire quelque chose comme :

```
UUID="6e9d5173-3e30-4e8b-8546-b82a7a311c57"  /mnt  ext4  noauto,user
```

N'importe quel utilisateur peut alors taper :

```
cmd>mount /mnt
```

et

```
cmd>umount /mnt
```

19. Une application complète : sauvegarde d'un répertoire avec rsync

But. — : réaliser une sauvegarde « intelligente » d'un répertoire vers un autre (à priori sur un support externe, connecté en usb).

Rsync. — La commande rsync est très puissante [29] ; à priori c'est une simple copie d'un répertoire vers un autre, par exemple :

```
cmd>rsync /home/ /mnt/home/
```

va copier le répertoire `/home/` vers `/mnt/home/`. Mais attention, *on ne va copier que ce qui n'est pas déjà présent dans* `/mnt/home/`, *ou ce qui est déjà présent dans* `/mnt/home/` mais a été modifié dans `/home/`. Le but est de synchroniser (d'où le nom de la commande) les deux répertoires en faisant le minimum de travail.

Mais `rsync` a plein d'options :

— On peut déjà ajouter `-a` (archive) :

```
cmd>rsync -a /home/ /mnt/home/
```

avec cette option on conserve tous les droits, les propriétaires et les groupes des fichiers et répertoires.

— On peut aussi ajouter `--delete` (oui, il y a bien deux "`-`") :

```
cmd>rsync -a --delete /home/ /mnt/home/
```

alors, si des fichiers existent dans `/mnt/home/` sans exister dans `/home/`, ils sont effacés dans `/mnt/home/`.

Avec ces deux options, on est donc capable de faire une sauvegarde parfaite [30].
Autres options de `rsync`, souvent utiles :

— Verbose : `-v` : vous dit tout ce que fait rsync. C'est rapidement beaucoup trop bavard, mais c'est très utile pour la mise au point !

— `--dry-run` : ne transfère rien, mais fait juste un test.

Avant toute copie, ou la première fois il est vraiment recommandé de faire un test à blanc :
`cmd>rsync -av --delete --dry-run /home/ /mnt/home/`
Bien regarder ce qui se passe ! [31]

---

29. `rsync` n'est pas forcément installé par défaut :
`cmd>sudo apt install rsync`
ou l'équivalent pour l'installer.
30. Le contenu de `/mnt/home/` sera bien identique à celui de `/home/` après.
31. Il peut être judicieux de faire : `cmd>rsync -av --delete --dry-run /home/ /mnt/home/|less`

19.1. La sauvegarde. — On va d'abord faire les choses « à la main », et puis on automatisera la sauvegarde.

Que faut il faire ?

1. Monter la partition sur laquelle on va faire la sauvegarde, si c'est nécessaire.
2. Avec `rsync` synchroniser les deux répertoires : celui à sauvegarder et la sauvegarde.
3. Éventuellement, démonter la partition de sauvegarde.

Ne pas garder la partition de sauvegarde montée en permanence diminue les risques d'écriture intempestifs [32]

Supposons que ma partition de sauvegarde soit toujours `/dev/sdd1`, qu'on la monte sur `/mnt` et qu'on veuille sauvegarder `/home`. Alors, les trois étapes ci-dessus sont :

1. `cmd>mount /mnt` [33]
2. Synchroniser avec `rsync`. La commande est :
   `cmd>rsync -a --delete /home/ /mnt/home/`
   Les options utilisées :
   - `-a` : « archive » : conserver les droits les propriétaires et les groupes : bref faire une sauvegarde exacte.
   - `--delete` : effacer dans le répertoire de backup (ici `/mnt`) tout ce qui n'est pas dans la source.

   Options à considérer, en tout cas pour la mise au point :
   - `-v` : *verbose*, bavard.
   - `--dry-run` : ne rien faire, ne rien copier, ne rien effacer.

   Donc, pour la mise au point, on peut vérifier que tout se passe bien avec :
   `cmd>rsync -av --dry-run --delete /home/ /mnt/home/` [34]

   Ne pas oublier les '"/" À LA FIN pour les répertoires : `/home/`, `/mnt/home/`.

   
3. Éventuellement, démonter la partition de sauvegarde :
   `cmd>umount /mnt`

19.2. Automatiser la sauvegarde. — Le but est de faire une sauvegarde qui se déclenche automatiquement tous les jours.

---

32. Et aussi, certains disques externes s'éteignent une fois démontés.
33. Je suppose qu'il y a la bonne ligne dans `/etc/fstab`. Sinon, on peut toujours faire :
    `cmd>mount -t ext4 /dev/sdd1 /mnt`
    tout aussi efficace.
34. Il n'y a aucun problème à interrompre rsync avec Control+C

19.2.1. Première étape : faire un « script ». — Un script, c'est un programme pour le shell.

Pour cela on va créer un fichier qui contient les commandes vues précédemment (après avoir tout testé à la main !). Voici le fichier, qu'on peut appeler par exemple `sauv.sh`.

```
#!/bin/bash
mount /mnt
rsync -a --delete /home/ /mnt/home/ &>>/var/log/sauv.log
umount /mnt
```

Commentaires :

— La première ligne est là là pour dire que le script doit être exécuté avec le shell *bash*.

— `&»/var/log/sauv.log`. On se souvient de la redirection des entrées-sorties ; En fait il y a plusieurs sorties numérotées 1 (normale), 2 (erreur). Le `&` redirige toutes les sorties ; on les redirige vers `/var/log/sauv.log` et les deux "`>`" (`»`) rallongent le fichier `/var/log/sauv.log` (on n'écrase pas le contenu de `/var/log/sauv.log`).

Où installer ce fichier ? par exemple dans `/etc/cron.daily`. Une fois créé avec l'éditeur de texte, on le rend exécutable (`cmd>chmod +x /etc/cron.daily/sauv.sh`).

Comment l'exécuter automatiquement ? C'est fait ! Il tournera tous les jours où la machine fonctionnera (c'est parce qu'on l'a mis dans `/etc/cron.daily`).

Il reste une chose à faire : le fichier `/var/log/sauv.log` va grossir indéfiniment. Il faut le faire « tourner » avec `logrotate`.

On peut regarder les différents scripts présents dans `/etc/logrotate.d` et en déduire que celui ci devrait convenir (ce n'est pas très critique ; `cmd>man logrotate` en dira plus sur les options du fichier) :

```
/var/log/sauv.log
{
rotate 4
weekly
missingok
notifempty
compress
delaycompress
sharedscripts
endscript
}
```

Installer ce fichier sous le nom `sauve` par exemple dans `/etc/logrotate.d/`.

19.3. Sauvegarder plusieurs répertoires. — On peut assez facilement transformer le *script* ci-dessus page 29 pour sauvegarder plusieurs répertoires. Il peut être intéressant de sauvegarder `/home`, mais aussi `/etc` par exemple, qui contient toute la configuration de la machine. Pour cela, on va un peu programmer le shell (car le shell est un langage de programmation).

19.3.1. Un premier script. — Fabriquons [35] un fichier de nom `essai1.sh` (le nom n'a pas d'importance) :

```
#!/bin/bash
for d in /home/ /etc/;do
echo $d
done
```

Nous avons déjà rencontré la première ligne. À la deuxième ligne nous commencons une boucle ; la variable `d` [36] va contenir successivement les chaînes de caractères `/home/` et `/etc/`. À la troisième ligne, on écrit le contenu de `d` (le shell fait une différence entre une variable (`d` ici) et son contenu qui ici est `$d`). Enfin la dernière ligne termine la boucle `for`. Une fois le fichier écrit, rendons le exécutable :

`cmd>chmod +x essai.sh`

et lançons la commande :

`cmd>./essai.sh`. On voit s'imprimer successivement `/home/` et `/etc`.

19.3.2. Deuxième étape. — On réécrit le script vu page 29 :

```
#!/bin/bash
mount /mnt
for d in /home/ /etc/;do
rsync -a --delete $d /mnt/$d &>>/var/log/sauv.log
umount /mnt
```

Et voilà : `d` contenant succesivement `/home/` et `/etc/`, `/mnt/$d` vaudra succesivement `/mnt/home/` et `/mnt/etc/`, ce qui fera bien ce qu'on cherche [37]. Notons que `/etc/` ne subissant que peu de modifications chaque jour, il n'y aura pratiquement aucun travail à effectuer pour le sauvegarder, *après* la première sauvegarde.

19.4. Synchronisation à distance. — `Rsync` permet des synchronisations de répertoires entre machines distantes, à condition qu'on puisse établir une connexion cryptée (`ssh`) entre les machines.

Supposons que je dispose d'un compte utilisateur `untel` sur la machine distante `mm.xxx.yy`. Je synchroniserai le répertoire distant `/sauv/` (par exemple) avec le répertoire `/home/` de ma machine locale avec la commande :

`cmd>rsync -a --delete -e ssh /home/ untel@mm.xxx.yy:/sauv/`

il peut être bien utile d'ajouter l'option `-z` pour compresser les données transférées et limiter la bande passante sur le réseau :

`cmd>rsync -az --delete -e ssh /home/ untel@mm.xxx.yy:/sauv/`

Une telle sauvegarde est absolument sûre, car la liaison est cryptée [38]

35. Il faut utiliser un éditeur de texte, genre `nano` ou autre.
36. On aurait pu choisir un nom complètement différent.
37. rsync crée automatiquement le répertoire de destination s'il n'existe pas.
38. Sur la machine `mm.xxx.yy`, on peut aussi faire en sorte que le répertoire `/sauv/` soit crypté, mais ce n'est pas du ressort de `rsync`.

Appendices

A. Bibliothèques de calcul

(Retour sur l'exemple de la page 20)

À quelle vitesse peut calculer une machine contemporaine ? Résoudre un système linéaire de n équations à n inconnues est un bon test.

Regardons le programme python suivant (vous pouvez sauter directement à la ligne ☛) :

```
 1 import numpy as np
 2 from scipy.linalg import lu_factor,lu_solve
 3 import time
 4 n=5000
 5 A=np.random.rand(n,n)
 6 B=np.random.rand(n)
 7 #
 8 c1=time.time()
 9 lu, piv = lu_factor(A)
10 X= lu_solve((lu,piv),B)
11 c2=time.time()
12 #
13 n3= 2.*n**3/3.
14 t=c2-c1
15 gflops=n3/t/10**9
16 print("Gigaflops",gflops)
```

Passons sur les lignes 1 à 3 ; aux lignes 4,5 et 6 on fabrique un tableau au hasard de taille 5000 par 5000, et un second membre B de taille 5000. Les deux tableaux définissent le système d'équations. La résolution du système de 5000 équations est effectuée aux lignes 9 et 10. Comme ces lignes sont entourées de deux mesures de "l'heure", la différence (ligne 14) donne le temps calcul t en secondes.

☛ Le jeu commence ici :

1. On peut montrer que la résolution (par la méthode de Gauss, qui est à peu près celle qu'on apprend au collège) « coûte » $\frac{2}{3}n^3$ opérations (additions et multiplications) pour un système de n équations, donc ici $\frac{2}{3} \times 5000^3$ opérations soit $\frac{2}{3} \times 125 \times 10^9$ opérations (environ $83,33...$ milliards d'opérations).
2. Sur ma machine (Intel i5, 3,5 Ghz, 4 cœurs), le temps calcul est de 0.779 secondes.

Donc, ma machine est capable d'effectuer $\frac{2}{3} \times 125 \times 10^9/0.779$ opérations par seconde, soit environ **106 milliards d'opérations par seconde !** (on dit : 106,9 gigaflops).

Diable ! Comment est-ce possible ?

— La machine tourne à 3.5 Gigahertz, ce qui permet à *chaque cœur* d'effectuer $3,5$ milliards d'opérations par seconde.

— Mais, ma machine a 4 cœurs, et le programme est "multithreadé" (parallélisé sur les 4 cœurs), donc on peut atteindre $4 \times 3.5 \times 10^9 = 14$ Gigaflops. Mais ça ne suffit pas à expliquer les 106,9 gigaflops observés.

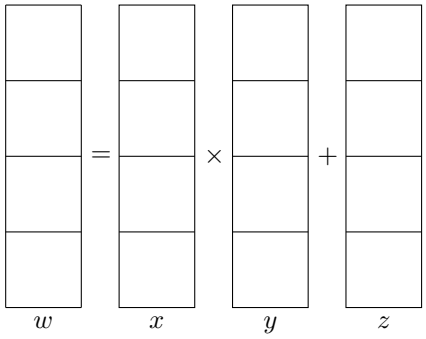


FIGURE 3. AVX (256)

— La réponse vient des instructions "avx" des processeurs Intel. Supposons qu'on ait 4 "vecteurs" w,x,y et z de taille 4 [39] (voir la figure 3) ; alors, les instructions avx permettent de calculer $w = x \times y + z$ en *un seul tour d'horloge* [40], soit 8 opérations (4 additions et 4 multiplications) par tour d'horloge. Chaque cœur, quand il rencontre ce genre d'opération peut donc atteindre la vitesse de $3,5 \times 8 = 28$ gigaflops (milliards d'opérations par seconde). Donc la vitesse maximale de ma machine (il y a 4 cœurs) est de $4 \times 28 = 112$ gigaflops.

Comment est effectué le calcul dans mon programme ? En fait, les lignes 11 et 12 passent la main à la bibliothèque `lapack` (libre et optimisée pour chaque type de machine). Écrire des programmes qui permettent d'approcher d'aussi près les performances théoriques de la machine (106 pour 112) est difficile (c'est quasiment un métier en soi) : **il faut utiliser des bibliothèques !**

Remarque finale : il y a là deux types de parallélisme :

1. Les 4 cœurs font (peuvent faire) des calculs différents sur des données différentes : on parle de parallélisme MIMD (Multiple Instructions Multiple Data).
2. Les instructions "avx" : ont fait en parallèle le même calcul sur des données différentes ($w_i = x_i \times y_i + z_i$). On parle de parallélisme SIMD (Single Instruction Multiple Data). C'est ce que font les GPUs [41] qui ont de très nombreux cœurs de calcul (7000 environ pour les plus puissantes), mais les cœurs font tous la même chose au même moment (idéal pour aditionner deux tableaux, terme à terme) : si vous avez 7000 cœurs cadencés seulement à une vitesse de 1 Ghz (les GPUs ont des vitesses d'horloge relativement faibles), vous atteindrez donc la vitesses de 7000×10^9 opérations par seconde, soit 7 téraflops (7000 milliards d'opérations par seconde).

   Les derniers processeurs Intel ont, pour chaque cœur, *deux* unités avx de 512 bits (au lieu de 256) qui permettent de calculer sur des vecteurs de taille 8 au lieu

39. C'est l'AVX 256 ($256 = 4 \times 64$). Les derniers processeurs ont un AVX 512 (vecteurs de taille 8).
40. Pour être plus précis :
    `pour i= 1 à 4:` $w_i = x_i \times y_i + z_i$
    est effectué en parallèle.
41. Processeurs graphiques.

de 4. La vitesse maximale *théorique* de chaque cœur est donc $2 \times 16 = 32$ fois la fréquence d'horloge.

B. Le COW (Copy On Write)

L'idée de base est : copier c'est lent et ça coûte de la place. On ne va donc copier (un fichier) que quand c'est nécessaire. Par exemple, si deux utilisateurs (deux processus) accèdent à un fichier uniquement en lecture, pourquoi le copier ? En revanche si un des deux processus a envie d'écrire sur le fichier, là il faut faire une copie pour donner à chacun son propre fichier. Comment implanter ça ? En voici une idée [42] :

1. Un premier processus accède au contenu du fichier par un pointeur qui montre au processus le « vrai » contenu du fichier (voir figure 4a). Le pointeur est un objet de petite taille, donc peu coûteux à copier.
2. Un deuxième processus accède aussi en lecture au même fichier (voir figure 4b). On crée un deuxième pointeur qui pointe sur les mêmes données. Mais en plus les données contiennent un compteur de références qui contient le nombre de pointeurs les désignant. Ce compteur qui valait 1 à l'origine vaut maintenant 2.
3. Imaginons que le premier processus veuille écrire sur le fichier ; alors (voir figure 4c) :
   (a) On en fait une copie (c'est là le « copy on write »).
   (b) On a deux versions séparées, qui on toutes les deux un compteur de références égal à 1.

L'effacement d'un fichier partagé en lecture par deux pointeurs est encore plus simple : si le premier processus veut effacer le fichier, on supprime son pointeur et on décrémente le compteur de réferences de 1. Un fichier (ses données) ne peut vraiment être effacé que quand le compteur de références est égal à 1.

Applications :—

— Les systèmes de « snapshots » (instantanés) des file systems Zfs et Btrfs :

Pour faire un snapshot de `/home` (par exemple), on crée un répertoire de nom `home-12h30`, par exemple ; on copie tous les pointeurs de `/home` dans `home-12h30`, ce qui peut être fait très rapidement [43]. Après cela, on a dans le nouveau répertoire une *vue* de `/home` tel qu'il était quand on a fait le snapshot, et cela pour un « prix » très faible. `home` va continuer à évoluer et, si on de touche pas à `home-12h30`, celui-ci va conserver l'état de `home` quand on a fait le *snapshot*.

C. Arbres binaires équilibrés (Balanced Trees, B-Trees)

Les *arbres binaires* sont une structure de données très efficace pour accéder rapidement à des données. Un arbre binaire (voir figure 5) est formé :

42. Ce n'est pas sans rapport avec les « liens hard », cf. page 23.
43. Il faut sûrement être un peu plus astucieux. Par exemple, appliquer la même technique... aux pointeurs ?

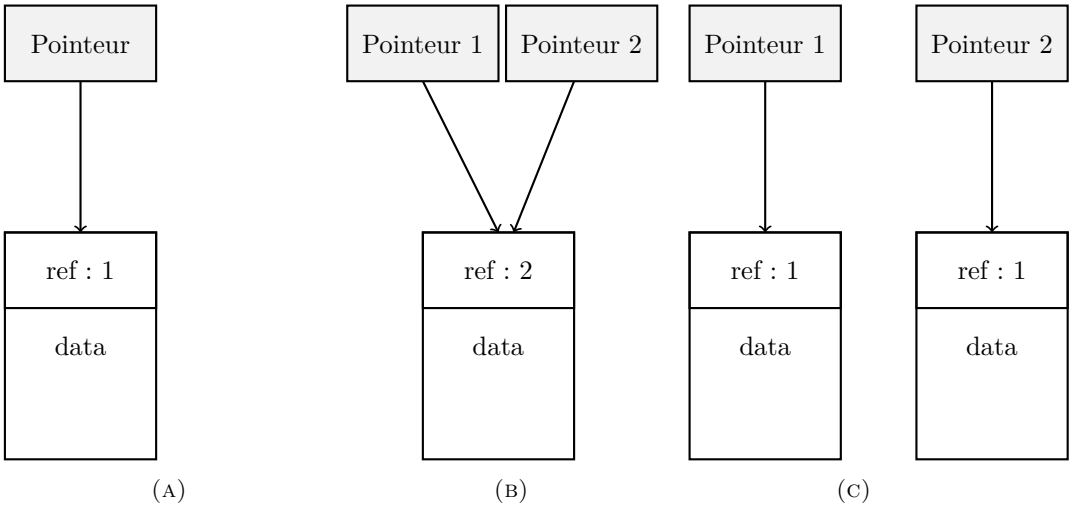


FIGURE 4. Copy On Write

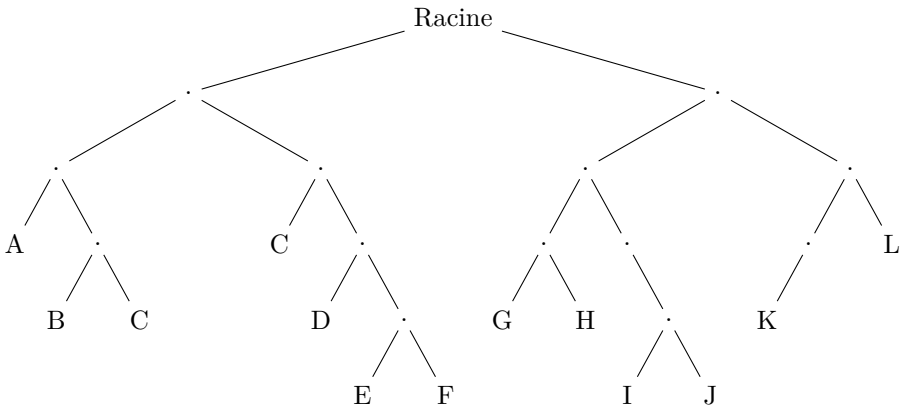


FIGURE 5. Arbre binaire

— de *nœuds* : depuis chaque *nœud* s'échappent zéro, une ou deux *branches*.

— Au bout de chaque branche, il y a un *nœud*. Si le *nœud* n'a pas de branche qui s'en échappe, on dit que c'est une *feuille*.

Le *niveau* d'un nœud ou d'une feuille est le nombre de branches qui faut parcourir pour l'atteindre +1.

Ainsi (figure 5) :

— *Racine* est au niveau 1.

— A, B, C, D, E, F, G, H, I, J, K, L, M sont respectivement aux niveaux : 4, 5, 5, 4, 5, 6, 6, 5, 5, 6, 6, 5, 4.

Supposons maintenant qu'on veuille ranger la liste : 50, 30, 40, 39, 42, 41, 20, 19, 22, 21, 25, 45, 43, 47, 70, 60, 58, 57, 62, 59, 80, 79, 65, 64, 67, 78, 90 dans un arbre binaire. On va :

— Installer 50 à la racine,

— Puis, 30 étant inférieur à 50 au l'installe au bout de la branche de gauche issue de 50.

— $40 < 50$ on va dans la branche de gauche issue de la racine, mais comme $40 > 30$, on l'installe dans la branche de droite issue de 30 (figure 6 (6a)).

— Bref : on part de la racine en allant à gauche ou à droite selon que le nombre est inférieur ou supérieur à la racine. Et chaque fois qu'on rencontre un nœud, on va à gauche ou à droite selon que la valeur à insérer est inférieure ou supérieure à celle du nœud. On insère la valeur quand une branche sélectionnée est vide (figure 6 (6b, 6c et 6d)).

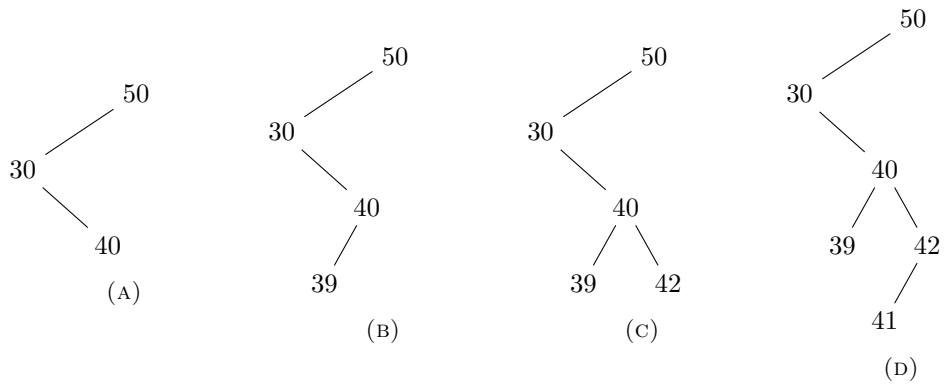


FIGURE 6. Insertions successives dans un arbre binaire

Évidemment, dans l'exemple donné ci-dessus on aurait pu remplacer les nombres par des chaînes de caractères, ou tout type d'objets qu'on peut comparer entre eux.

Considérons maintenant un arbre binaire de n niveaux ($n = 2, 3, ...$).

C'est là que ça devient remarquable ! Pour retrouver un nombre dans l'arbre il faut parcourir au plus n niveaux, soit encore faire n comparaisons et n branchements au plus.

Mais combien y a-t-il de nœuds et donc *objets* (de nombres) rangés dans un arbre de n niveaux ? On suppose que l'arbre est complet (cf. figure 7), c'est à dire que tous les niveaux sont complètement occupés :

— au niveau 1 : 1.

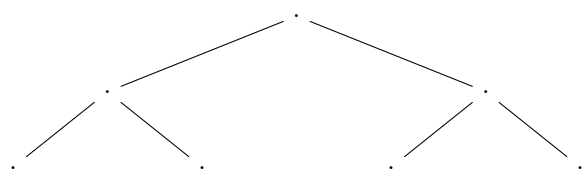


FIGURE 7. arbre binaire complet (3 niveaux)

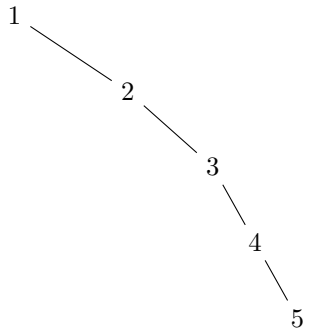


FIGURE 8. Insertion d'une liste ordonnée

— au niveau 2 : 2, soit en tout $2 + 1 = 3$ dans l'arbre.

— au niveau 3 : 4 , soit en tout $1 + 2 + 4 = 7$ dans l'arbre

— ...

— au niveau n : il y a 2^{n-1} nœuds, ce qui fait qu'entre le niveau 1 et le niveau n (compris), on stocke $1 + 2 + 2^2 + 3^2 + \ldots + 2^{n-1}$ soit exactement $2^n - 1$ nœuds.

La recherche est *très rapide* :

— pour 1000 objets rangés dans un arbre binaire complet, comme $2^{10} = 1024$, les recherches se termineront en au plus 10 étapes. Dans un arbre binaire complet contenant un milliard d'objets, comme $10^9 = 1000^3 < 1024^3 = 2^{10^3} = 2^{30}$, les recherches se termineront en au plus 30 étapes.

— Supposons que la population mondiale rangée dans un arbre binaire complet (par ordre de date de naissance ?) : comme il y a environ 6 milliards d'individus (6×10^9) on trouve qu'il faut au maximum 33 étapes pour trouver un individu ! [(44)]

C.1. Oui, mais...— Que se passe t-il quand on insère une liste ordonnée ? (voir la figure 8). On engendre non pas un arbre binaire (toutes les branches gauches sont vides) mais une liste et la recherche à un coût qui n'est pas celui qu'on attend !

Cela vient du fait que le l'arbre n'est pas *équilibré* (*balanced*).

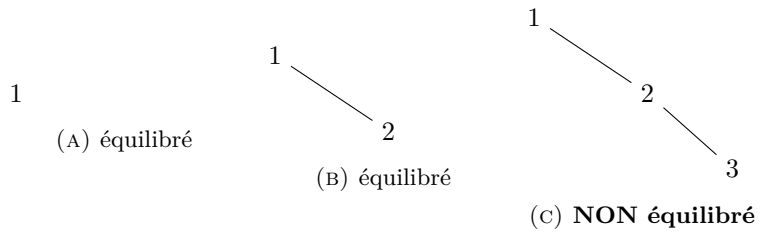
Un arbre binaire est dit équilibré si les niveaux des feuilles diffèrent au plus de 1.

On peut s'arranger pour garder l'arbre équilibré au fur et à mesure des insertions. Reprenons comme exemple l'insertion de la liste (1, 2, 3, 4, 5). Après l'insertion de (1) et (2) l'arbre est équilibré (deux branches, une de longueur 1 –la branche droite–, l'autre de longueur 0 –la branche gauche–). Mais à l'insertion de (3), l'arbre n'est plus équilibré (voir figure 9). Mais on peut rendre l'arbre équilibré par une transformation illustrée à la figure 10 : entre 10a et 10b on a fait remonter (2) d'un niveau, et du coup (1) vient à sa gauche, et l'arbre binaire est équilibré.

Ce qui veut dire que, avec cette technique, il est plus rapide de trouver un objet que dans insérer un.

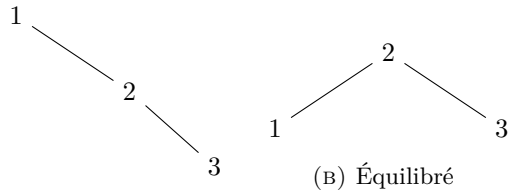
Document composé en LaTeX.

44. Le résultat est qu'une recherche parmi n objets coûte au maximum $\log_2 n$ opérations ($\log_2 2^p$, c'est p).



(A) équilibré (B) équilibré (C) **NON équilibré**

FIGURE 9. Insertion successive de 1, 2, 3



(A) Non équilibré (B) Équilibré

FIGURE 10. Équilibrage