Unix/Linux

Gaetan Bisson

https://gaati.org/bisson/

Introduction

Les systèmes d'exploitation dits « de type Unix » sont omniprésents ; ils dominent notamment les secteurs de l'embarqué et des serveurs. Ils offrent des fonctionnalités logicielles vastes et supportent essentiellement toutes les plateformes matérielles existantes. Ce cours a pour but d'en expliquer le fonctionnement et d'en inculquer l'utilisation par l'interface centrale qu'est l'interpréteur de commandes.

Certains systèmes de type Unix, comme Android ou Ubuntu, sont entièrement configurables par le biais d'interfaces graphiques ergonomiques ; ce cours n'est nullement un prérequis à leur utilisation. Le lecteur est fortement invité à installer un tel système sur son ordinateur personnel afin d'y voir les concepts du cours en action tout en gardant la facilité d'utilisation à laquelle il est habitué.

Table des matières

1 Présentation **4**
- 1.1 Systèmes d'exploitation 4
- 1.2 Systèmes de type Unix 4
- 1.3 Distributions Linux 6
- 1.4 Avantages de Linux 6

2 Concepts fondamentaux **8**
- 2.1 Arborescence de fichiers 8
- 2.2 Comptes utilisateurs 8
- 2.3 Rôle du noyau 10
- 2.4 Processus 10

3 Interpréteur de commandes **12**
- 3.1 Lancer des programmes 12
- 3.2 Déplacements dans l'arborescence 13
- 3.3 Attributs et permissions 14
- 3.4 Exécution série ou parallèle 15

4 Documentation **17**
- 4.1 Le manuel 17
- 4.2 Internet 18
- 4.3 La littérature 18
- 4.4 Les standards 18

5 Concepts avancés des processus **19**
- 5.1 Exécution et code de retour 19
- 5.2 Environnement d'exécution 21
- 5.3 Entrées et sorties 22
- 5.4 Signaux 23

6 Commandes shell avancées **25**
- 6.1 Redirection de flux 25
- 6.2 Exécution conditionnelle 26
- 6.3 Substitution et métacaractères 27
- 6.4 Scripts shell 28

7 Concepts avancés du système **30**
7.1 Partitions et montage 30
7.2 Systèmes de fichiers 31
7.3 Fichiers spéciaux 33
7.4 Autres composants systèmes 34

8 Programmation **35**
8.1 Compilation 35
8.2 Interfaces 35
8.3 Édition de texte 35
8.4 Expressions régulières 35

9 Administration système **37**
9.1 Compte super-utilisateur 37
9.2 Gestion des comptes 37
9.3 Gestion des logiciels 37
9.4 Sécurité 37

10 Problèmes **38**
10.1 Gestion des mots de passes 38
10.2 Courrier électronique 38

Bibliographie **41**

Chapitre 1

Présentation

1.1 Systèmes d'exploitation

Les composants matériels des ordinateurs sont de simples automates : ils reçoivent des instructions et renvoient leurs réponses sur des électrodes sous la forme de signaux électriques. Le comportement d'un composant (les instructions disponibles, les signaux correspondants, etc.) varie de constructeur à constructeur, de série à série, etc. Dans les années cinquante, ces interfaces de bas niveau étaient les seules disponibles aux utilisateurs ; chaque programme était donc écrit pour les composants d'un modèle d'ordinateur spécifique. En outre, on ne savait exécuter qu'un seul programme à la fois.

Depuis les années soixante, les systèmes d'exploitation sont continument développés dans deux buts majeurs :

— présenter aux utilisateurs une interface indépendante du matériel sous-jacent.
— allouer équitablement les ressources matérielles à de multiples utilisateurs.

Un système d'exploitation est ainsi une collection de logiciels qui s'intercale entre les composants matériels et les utilisateurs afin de rendre l'exploitation de ces premiers par ces derniers plus facile. Deux grandes familles subsistent aujourd'hui : DOS et Unix. Leur place dans l'écosystème informatique actuel est esquissée par les diagrammes approximatifs de la figure 1.1.

L'ancêtre de cette première famille est QDOS, un système créé en 1980 que Microsoft a ensuite racheté, modifié, et revendu sous le nom de MS-DOS. Les systèmes Windows étaient à l'origine de simples interfaces graphiques pour MS-DOS et ont depuis évolué en les logiciels que l'on connaît actuellement.

L'ancêtre de la seconde famille est UNIX, un système conçu à Bell Labs en 1969. Rapidement adopté dans le milieu universitaire, les restrictions imposées par sa licence ont par la suite suscité la création d'alternatives libres reproduisant une interface similaire. Cela a donné naissance aux systèmes dits « de type Unix ».

1.2 Systèmes de type Unix

On dit qu'un système d'exploitation est « de type Unix » s'il reproduit des interfaces semblables à celles du UNIX original. La figure 1.2 illustre le développement de ces systèmes en montrant les héritages de code source (traits pleins) et d'idées (traits pointillés). Nous retiendrons que trois branches majeures subsistent :

— les System V (dont AIX et Solaris) ;

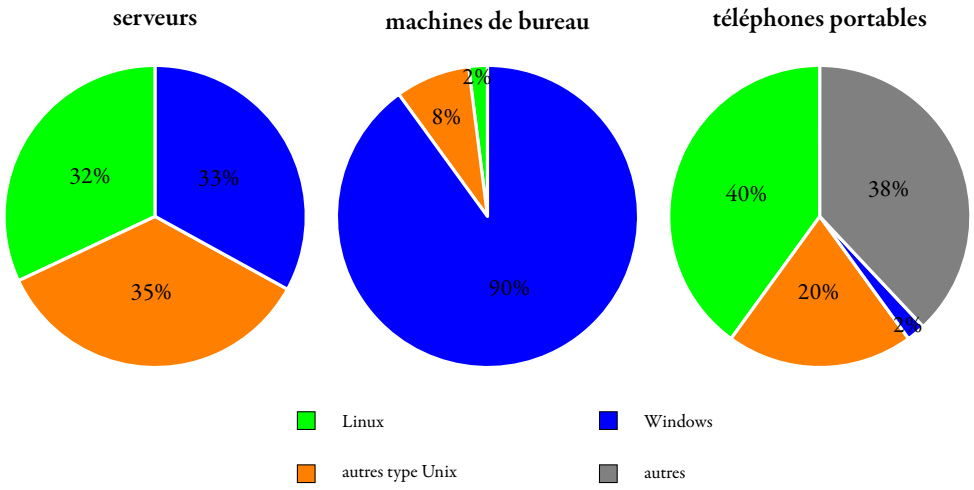

FIGURE 1.1 – Utilisation des systèmes d'exploitation par secteur.

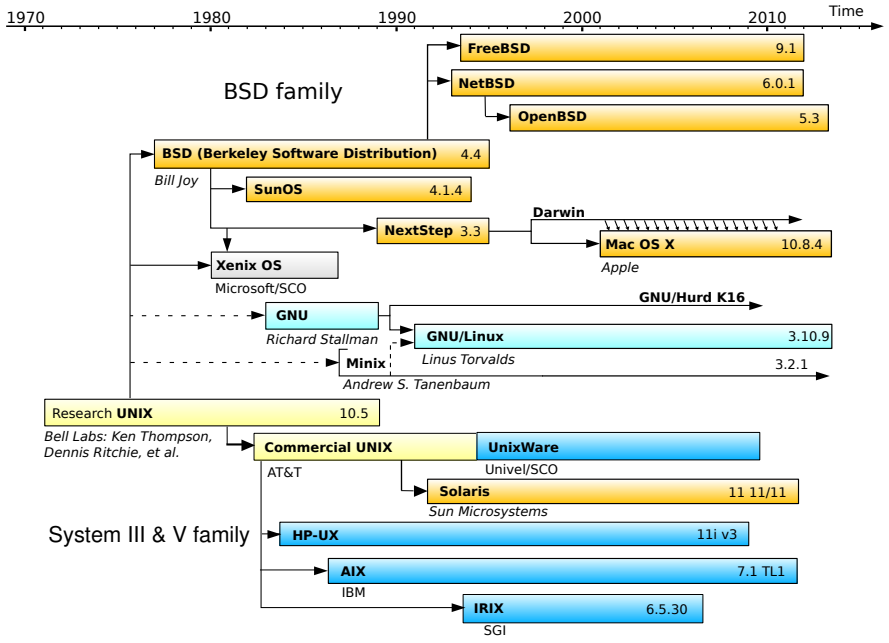

FIGURE 1.2 – Généalogie synthétique des systèmes de types Unix.

— les BSD (dont FreeBSD, NetBSD, OpenBSD et MacOS) ;
— les Linux (par exemple Android, Debian, Ubuntu ou encore Arch Linux).

Même si leur implémentation, leur licence d'utilisation (et leur prix) diffèrent, retenons que tous ces systèmes fournissent des interfaces essentiellement identiques. C'est elles que ce cours a pour vocation d'enseigner.

Pour une présentation historique détaillée des systèmes de type Unix, voir [8]. Dans ce cours, lorsque nous ne pourrons pas en traiter en toute généralité, nous prendrons comme référence les systèmes GNU/Linux, c'est-à-dire alliant le noyau Linux à l'espace utilisateur GNU ; ce sont les systèmes de type Unix les plus courant sur les ordinateurs personnels.

1.3 Distributions Linux

Linux n'est en réalité qu'un noyau, la pièce logicielle qui alloue les ressources matérielles aux utilisateurs et leur présente une interface indépendante du matériel sous-jacent. Par extension, il est aussi devenu d'usage d'appeler « Linux » l'ensemble des systèmes l'ayant comme noyau. Un système d'exploitation contient toutefois bien plus qu'un noyau, notamment :

— un programme d'initialisation, `init`.
— des interfaces utilisateurs, en ligne de commande ou graphique.
— des outils de développement pour le langage de programmation système (C) :
  — une bibliothèque standard, `libc`.
  — un compilateur, `cc`.
  — un déboguer, etc.

Une flottille de pièces logicielles s'articule donc autour du noyau, chacune pouvant trouver plusieurs implémentations et admettre de multiples configurations. L'ensemble de tous ces logiciels s'appelle une distribution Linux.

Il existe des centaines de distributions Linux, chacune ayant ses objectifs spécifiques. Elles ont en commun le noyau Linux mais diffèrent sur le choix des autres logiciels et leur configuration. On compte par exemple :

— Android, conçue pour les terminaux mobiles à écran tactile ;
— Debian, qui vise la solidité et le respect des libertés logicielles ;
— Ubuntu, qui s'efforce d'être accessible aux utilisateurs non avertis ;
— Arch Linux, qui recherche la simplicité du point de vue des développeurs.

1.4 Avantages de Linux

Linux a trois avantages majeurs sur ses compétiteurs : sa qualité, sa modularité, et sa liberté. Sa modularité remonte à Unix et est souvent ce à quoi on fait référence lorsque l'on parle de « philosophie Unix » :

> *Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*
>
> (Doug McIlroy)

Chaque logiciel d'un système de type Unix se borne donc à fournir une fonction bien précise par une interface épurée ; on peut alors réaliser des fonctions plus complexes en assemblant ces programmes élémentaires. En outre, les interfaces étant simples et publiques, on peut remplacer tout logiciel par une implémentation alternative sans aucune incidence sur le système d'exploitation.

La liberté logicielle est l'une des raisons historiques ayant mené à la création de Linux. La majorité des logiciels des distributions Linux sont de nos jours distribués sous la licence GPL qui, essentiellement, autorise :

- toute exécution du logiciel ;
- toute modification du logiciel ;
- la redistribution du logiciel modifié, à condition qu'il :
  - soit lui aussi distribué sous licence GPL.
  - soit accompagné de son code source.

Ce type de licence protège de manière irrévocable deux libertés fondamentales des utilisateurs : celles d'exécution et de modification des logiciels.

Les licences libres permettent aux développeurs de distribuer leurs logiciels en évitant qu'une entité tierce (par exemple, une entreprise) se les approprie sans contribuer en retour. C'est pourquoi des milliers de développeurs, professionnels et amateurs, collaborent chaque jour sur des projets sous licence GPL, par exemple le noyau Linux. Le code source étant librement disponible et modifiable, il est constamment inspecté et amélioré, ce qui produit des logiciels de bien meilleure qualité que les autres modèles de distribution.

Chapitre 2

Concepts fondamentaux

Nous allons maintenant décrire les concepts fondamentaux sous-jacents aux systèmes d'exploitation de type Unix. Pour éviter que cette description reste abstraite, le lecteur est invité à attaquer en parallèle le chapitre suivant qui explique comment manipuler ces concepts grâce à l'interpréteur de commandes.

2.1 Arborescence de fichiers

Unix essaie de rendre la majorité des données accessibles par des fichiers. Pas seulement les données utilisateurs (films, musiques, documents, etc.) mais réellement toutes les données dont il dispose, comme l'état de la mémoire vive ou encore la liste des programmes en cours d'exécution. Il est donc primordial de maîtriser le concept de fichiers et celui, dual, de répertoire. Ces notions sont communes à la grande majorité des systèmes d'exploitation, mais peu les formalisent et les exploitent autant qu'Unix.

> Les fichiers forment les feuilles d'une arborescence de répertoires. On note un fichier ou un répertoire par la suite des répertoires le contenant, séparés par le caractère « `/` », en partant du répertoire racine. On écrira donc :
>
> le répertoire `/nombres/premiers/`
> contient le répertoire `/nombres/premiers/impairs/`
> ainsi que le fichier `/nombres/premiers/deux`

Le contenu de cette arborescence est organisé en les principales branches de la figure 2.1.

2.2 Comptes utilisateurs

Dès l'origine, Unix a été conçu pour être un système multi-utilisateur, c'est-à-dire que plusieurs personnes peuvent utiliser simultanément : il alloue les ressources matérielles (mémoire, processeur, réseau) équitablement et de manière transparente aux utilisateurs tout en assurant la sécurité de leurs données.

Chaque compte utilisateur du système comprend les données suivantes :

— nom de l'utilisateur et numéro associé (UID) ;

`/`	la racine, contient quasi exclusivement ce qui suit.
`/dev/`	renferme des fichiers représentant les périphériques.
`/etc/`	est l'emplacement des fichiers de configuration systèmes.
`/home/`	contient les répertoires des utilisateurs.
`/proc/`	représente la hiérarchie des programmes en cours d'exécution.
`/usr/`	contient les programmes.
`/usr/bin/`	contient les exécutables.
`/usr/lib/`	contient les bibliothèques partagées.
`/usr/include/`	contient les en-têtes des bibliothèques.
`/usr/share/`	contient les autres données associées.
`/sys/`	fournit des informations sur le noyau.
`/tmp/`	héberge les fichiers temporaires à court terme.
`/var/`	renferme les données variables à moyen terme.

FIGURE 2.1 – Principales branches de l'arborescence de répertoires d'un système de type Unix.

— nom du groupe et numéro associé (GID) ;
— (haché du) mot de passe ;
— emplacement du répertoire personnel ;
— emplacement du programme à exécuter en début de session.

Avant de pouvoir utiliser le système, chaque utilisateur doit s'identifier en tapant son nom d'utilisateur et son mot de passe. Si l'authentification est valide, alors le système le place dans son répertoire personnel et lance le programme à exécuter en début de session, typiquement un interpréteur de commandes. Lorsque ce programme se termine, la session se ferme.

Au niveau du système de fichiers, la gestion des multiples utilisateurs se réalise par des métadonnées, aussi appelées attributs, qui accompagnent chaque fichier ou dossier ; elles contiennent notamment :

— l'utilisateur à qui ce fichier appartient (identifié par son UID) ;
— le groupe auquel ce fichier appartient (identifié par son GID) ;
— les droits accès (paramétrables par l'utilisateur) :
  — si l'utilisateur à qui ce fichier appartient a le droit de le lire, modifier ou exécuter ;
  — si les autres utilisateurs du groupe ont le droit de le lire, modifier ou exécuter ;
  — si les autres utilisateurs du système ont le droit de le lire, modifier ou exécuter ;
— les dates de dernière : lecture, modification du fichier, changement de ses attributs.

> Le type d'un fichier en résume les droits d'accès. Il se note sous la forme « `-rwxrwxrwx` » où : le premier caractère est « `-` » pour les fichiers et « `d` » pour les répertoires ; les caractères 2–4 codent les permissions du propriétaire, 5–7 celles du groupe et 8–10 celles des autres. Les symboles « `rwx` » dénotent respectivement les permissions en lecture, écriture et exécution ; le symbole « `-` » dénote quant à lui l'absence de cette permission.
>
> Par exemple, le type « `-rw-r--r--` » est très courant : il dénote un fichier que tout le monde peut lire mais que seul le propriétaire peut écrire.

D'autres mécanismes permettent de paramétrer des droits plus fins mais ceux ci-dessus suffisent dans la grande majorité des cas à séparer ou partager de manière satisfaisante les données entre des utilisateurs multiples.

Naturellement, on ne peut concéder à tous le droit de modifier `/bin/` ou encore `/dev/` (dont l'un des fichiers présente une interface vers le contenu entier de la mémoire vive), mais on souhaiterait tout de même que quelqu'un puisse ne serait-ce que mettre à jour les programmes qui s'y trouvent. Il existe donc un utilisateur tout-puissant, nommé `root` et d'UID `0` ; c'est l'*administrateur système* de la machine. L'utilisateur lambda n'aura quant à lui par défaut que permission de modifier ce qui se trouve dans son répertoire personnel.

L'organisation du serveur utilisé en TD suit des conventions bien établies :
— votre nom d'utilisateur est votre nom de famille tronqué à huit caractères ;
— votre groupe est `math` ou `info` selon votre filière ;
— votre répertoire personnel est `/home/username` ;
— par défaut, vos fichiers sont lisibles par les autres, mais pas modifiables.

2.3 Rôle du noyau

Nous avons déjà mentionné que le noyau est un logiciel qui s'intercale entre la couche matérielle et les autres programmes et jongle avec ces derniers de sorte à : leur présenter une interface indépendante du matériel sous-jacent ; leur allouer les ressources systèmes de manière équitable.

Vu de haut niveau, le noyau donne aux programmes l'illusion de tourner sur une machine idéale possédant :
— une mémoire vive de 32 To utilisable arbitrairement ;
— un système de fichiers dont les limites (nombre et taille des fichiers, profondeur des répertoires, etc.) sont de l'ordre de 2^{64}, c'est-à-dire difficilement atteignables ;
— un processeur toujours disponible (en réalité, il exécute alternativement les instructions des divers programmes lancés, y compris celles du noyau) ;
— une interface réseau (TCP) dont les paquets sont tous transmis, dans l'ordre, etc.

Nous mentionnerons par la suite d'autres tâches incombant au noyau mais son fonctionnement interne ne sera pas abordé dans ce cours ; le lecteur sérieux pourra approfondir ce sujet avec l'ouvrage [1].

2.4 Processus

Un programme n'est qu'une suite d'instructions destinées :
— au processeur : essentiellement pour effectuer du calcul pur ;
— au système d'exploitation : accès aux périphériques, système de fichiers, etc.

Lorsque l'ordinateur est éteint, les programmes, y compris le système d'exploitation, sont généralement stockés sur un disque dur. Lorsqu'on l'allume, une chaîne de programmes de plus en plus complexes se charge (afin de se libérer progressivement des contraintes matérielles), aboutissant au système d'exploitation ; typiquement :

1. Le processeur exécute un micrologiciel situé sur une puce mémoire en lecture seule : BIOS, EFI, etc.
2. Ce micrologiciel exécute la zone d'amorce située sur les 512 premiers octets d'un disque : MBR, GPT, etc.
3. La zone d'amorce exécute le chargeur d'amorçage (bootloader) situé plus loin sur le disque : NTLDR, GRUB, SYSLINUX, etc.
4. Le chargeur d'amorçage exécute le système d'exploitation situé à un emplacement arbitraire du disque : Windows, Linux, BSD, etc.

Ces exécutions se font en série, c'est-à-dire que chaque programme s'exécute *à la place* de celui qui l'a chargé. Lorsque le système d'exploitation prend le contrôle de la machine, en revanche, il perdure jusqu'à l'extinction du système et la majorité des programmes (par exemple, `getty`, `xdm`, etc.) sont lancés *en parallèle*.

Le lancement d'un programme par le système d'exploitation se fait en deux étapes :

1. Le programme est copié dans son intégralité du disque dur vers la mémoire vive.
2. Le programme est exécuté directement depuis la mémoire vive.

On appelle processus tout programme lancé, c'est-à-dire chargé en mémoire.

Cette technique d'exécution est robuste, notamment par ce qu'elle permet de modifier les fichiers sous-jacents sans perturber les processus : pas besoin de redémarrer après une mise à jour. En outre, un même programme peut simultanément être lancé par plusieurs utilisateurs : cela donnera lieu à des processus distincts coexistant en mémoire, chacun n'ayant accès qu'à sa propre zone mémoire.

> Chaque processus est identifié par un entier unique appelé PID. Le premier processus, nommé `init`, à partir duquel tous les autres sont lancés (directement ou indirectement), a pour PID un.

On qualifie de démon tout processus qui s'exécute en arrière plan, c'est-à-dire de manière non directement visible pour l'utilisateur. Les démons sont souvent amenés à s'exécuter continument jusqu'à l'arrêt du système. Par exemple, les programmes `sshd` (service de connexion à distance), `ntpd` (synchroniseur d'horloge système) et `thttpd` (serveur Web) sont des démons courants ; à l'inverse, Firefox et LibreOffice n'en sont pas.

Chapitre 3

Interpréteur de commandes

L'interface utilisateur traditionnelle des systèmes d'exploitation de type Unix est le shell ; elle permet de lancer et de contrôler des processus par des *lignes de commandes*. Deux grandes familles subsistent aujourd'hui :

— Bourne Shell (bash, dash, ksh, zsh)
— C shell (csh, tcsh)

La syntaxe de certaines commandes (typiquement, les boucles `for`) varie légèrement de shell à shell, y compris au sein d'une même famille. Ce cours utilisera exclusivement `bash`, qui est très répandu, mais le lecteur ne rencontrera aucune difficulté majeure à apprendre d'autres shells une fois celui-ci maîtrisé.

Toutes les commandes décrites dans ce cours ne seront présentées que succinctement ; on trouvera des informations détaillées à leur sujet par le biais de la section 4.

3.1 Lancer des programmes

Lorsque la ligne de commande ne comporte qu'un nom de programme, éventuellement suivi d'arguments, le shell se contente de le lancer ; les évènements claviers sont alors envoyés au programme et sa sortie s'affiche à l'écran. Essayer par exemple :

```
$ date
Fri Dec 20 16:59:59 TAHT 2013
$ date -u
Sat Dec 21 02:59:59 UTC 2013
```

Par convention, les arguments optionnels, comme ici « `-u` » qui demande que l'heure soit affichée en temps universel coordonné, commencent généralement par un (voire deux) tirets.

L'une des toutes premières commandes qu'il est fortement recommandé de lancer sur un compte Unix fraîchement obtenu est « `passwd` » ; elle a pour effet de changer le mot de passe du compte :

```
$ passwd
Changing password for bisson.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Remarquez que les mots de passe tapés n'apparaissent pas à l'écran, même pas sous forme d'étoiles. Cela permet d'éviter que quiconque apprenne quoi que ce soit sur votre mot de passe en observant l'écran.

Parmi les programmes standards les plus simples, on a :
— `uptime` donne le temps de fonctionnement et la charge du système.
```
$ uptime
 16:59:59 up 457 days, 08:09, 42 users,  load average: 0.65, 0.68, 0.71
```
— `id` affiche les UID, GID et groupes des utilisateurs.
```
$ id
uid=101(bisson) gid=100(users) groups=100(users),91(video),92(audio)
```
— `who` affiche les utilisateurs actuellement connectés au système.
```
$ who
bisson   pts/1        2013-12-20 16:59 (129.199.129.1)
```
— `ps` donne des informations sur les processus (essayer « `ps -elyFH` »).
— `top` affiche et trie des informations sur les processus en temps réel.
— `echo` se contente de renvoyer la liste des arguments qu'on lui donne.
— `sleep` ne fait strictement rien pendant x secondes puis termine.
— `banner` et `figlet` sont deux programmes de rendu ASCII.
— `history` affiche les dernières commandes interprétées.
— `exit` quitte le shell.

Ces programmes, ainsi que le shell depuis lequel vous les lancez, vous paraissent certainement « spartiates » ; toutefois nous verrons rapidement que c'est une vertu : cela permet d'extraire efficacement l'information pertinente du texte renvoyé et ainsi de combiner aisément ces programmes les uns avec les autres.

Pour finir, mentionnons un éditeur de texte, `nano`, qui est conçu pour être facile à utiliser au sens où il ne requiert aucun apprentissage : à tout moment est affiché en bas du terminal la liste des actions possibles ; cette liste suit la convention usuelle que « `^X` » dénote l'appui simultané des touches contrôle et « `X` ».

3.2 Déplacements dans l'arborescence

La commande « `pwd` » affiche le répertoire courant ; en début de session, il s'agit de votre répertoire personnel. La commande « `cd` » déplace l'utilisateur dans le répertoire de son choix ; il peut en indiquer le chemin :
— absolu : en partant de la racine ;
— relatif : en partant du répertoire courant.

Par exemple, pour aller en `/home/bisson/music/messy`, on pourra taper, au choix, « `cd /home/bisson/music/messy` » ou seulement « `cd music/messy` » si l'on se trouve actuellement en `/home/bisson`.

L'écriture de ces chemins est sujette à quelques conventions :
— « `.` » dénote le répertoire courant.
— « `..` » dénote le répertoire parent.
— « `~` » dénote le répertoire personnel.

Notons que cette dernière convention est interprétée par le shell, contrairement aux deux premières qui le sont par le noyau ; les commandes « `echo .` » et « `echo ~` » n'ont donc pas le même résultat.

Le programme `ls` affiche la liste des fichiers et des sous-répertoires que le répertoire courant contient. Il admet de nombreuses options, notamment :

— `-l` affiche davantage d'informations.
— `-a` affiche tous les fichiers (sinon, ceux commençant par un point sont omis).
— `--color` utilise des couleurs !

Le lecteur est désormais capable de se déplacer dans l'arborescence des répertoires ainsi que de visualiser le contenu d'un fichier (grâce à l'éditeur `nano`) ; nous lui conseillons fortement de se promener et de jeter un œil aux fichiers situés dans les répertoires mentionnés à la section 2.1. Remarquer notamment ce qui démarque Unix de certains autres systèmes :

— Les noms de fichier peuvent posséder 255 caractères.
— Ils n'ont pas nécessairement d'extension « `.xyz` ».
— Beaucoup de fichiers sont en texte pur.

Finissons cette section en présentant les programmes permettant de modifier l'arborescence des répertoires et les fichiers qu'elle contient. Rappelons qu'avec l'éditeur de texte `nano` on peut déjà créer des fichiers à l'emplacement de notre choix ; les répertoires, quant à eux, se créent grâce à la commande `mkdir`.

Les commandes `cp` et `rm` permettent respectivement de copier et supprimer des fichiers. Elles s'appliquent aussi aux répertoires lorsque l'option « `-r` » est donnée. On dispose aussi d'un programme dédié, « `rmdir` » pour supprimer des répertoires.

Le programme `mv` renomme un fichier ou répertoire ; cela revient à copier le fichier puis à supprimer l'original mais est bien plus efficace car aucune copie n'est réalisée : seul ne nom et l'emplacement du fichier sont changés.

Exercice. *Créer un fichier* `toto` *et un répertoire* `bla`*; copier alors* `toto` *dans* `bla`*. Aller en* `bla` *et y déplacer le fichier* `toto` *original. Supprimer enfin le répertoire* `bla`*.*

3.3 Attributs et permissions

La sortie de « `ls -l` » affiche les attributs les plus importants de chaque fichier :

1. les permissions,
2. le nombre de liens,
3. l'utilisateur propriétaire,
4. le groupe propriétaire,
5. la taille en octets,
6. la date de dernière modification,
7. le nom du fichier.

D'autres programmes plus spécifiques permettent de consulter et de modifier les attributs. Pour la consultation, `stat` est probablement le plus utilisé :

```
$ stat .
  File: '.'
  Size: 4096            Blocks: 16         IO Block: 4096   directory
Device: 804h/2052d      Inode: 4096        Links: 18
Access: (0755/drwxr-xr-x)  Uid: (  101/  bisson)   Gid: (  100/   users)
Access: 2013-12-20 16:59:59.123456789 -1000
Modify: 2013-12-19 16:59:59.987654321 -1000
Change: 2013-12-19 16:59:59.987654321 -1000
 Birth: -
```

On retrouve, de manière exhaustive, toutes les informations de la section 2.2 ainsi que des paramètres que nous verrons plus tard dans le cours.

Trois programmes permettent respectivement de changer l'utilisateur propriétaire, le groupe propriétaire et les permissions d'un fichier : `chown`, `chgrp` et `chmod`. Ils s'utilisent tous suivant le même format : « *`programme nouvelle_valeur fichier(s)_à_changer`* ».

Nous avons vu que les permissions pouvaient s'écrire comme un type `rwxrwxrwx` ; elles sont aussi souvent écrites sous forme octale « `xyz` » où x est la somme de 4 (si l'utilisateur peut lire le fichier), 2 (s'il peut l'écrire) et 1 (s'il peut l'exécuter) ; y et z sont les sommes correspondantes pour le groupe et les autres utilisateurs. Les fichiers sont majoritairement en mode `644` (l'utilisateur peut lire et écrire ; les autres peuvent seulement lire) et les répertoires en `755` (l'utilisateur peut lire, écrire, et traverser ; les autres ne peuvent que lire et traverser). C'est ainsi que l'on présente généralement des permissions à `chmod`.

Les permissions par défaut des nouveaux fichiers sont régies par `umask`, le masque de création des modes de fichiers. On peut appeler le programme éponyme pour connaître son masque actuel ou en changer la valeur. Les nouveaux fichiers et répertoires créés se voient attribués le mode suivant par défaut :

— `666 & !umask` pour les fichiers ;
— `777 & !umask` pour les répertoires.

La valeur la plus courante est `022` qui résulte en les permissions par défaut mentionnées ci-dessus : `644` pour les fichiers et `755` pour les répertoires.

Exercice. *Quand le fichier* `/etc/profile` *a-t-il été modifié pour la dernière fois ? Pouvez-vous changer cette date ? Créez un fichier que vous seul pourrez lire ; d'autres pourraient-ils tout de même y écrire ?*

3.4 Exécution série ou parallèle

On peut lancer séquentiellement plusieurs programmes en séparant les commandes correspondantes par des point virgules « ; » : le premier programme se lance alors jusqu'à ce qu'il termine et le shell lance alors le second, etc.

```
$ date; sleep 7; date
Fri Dec 20 16:59:52 TAHT 2013
Fri Dec 20 16:59:59 TAHT 2013
```

Lorsque l'on sépare des commandes d'une esperluette « `&` », toutes (sauf la dernière) sont lancées en arrière-plan. On peut contrôler les processus qui en résultent grâce aux commandes :

— `^C` interrompt l'exécution du programme actuel brutalement.
— `^Z` suspend l'exécution du programme actuel et rend la main au shell.
— `jobs` affiche la liste des programmes actuellement en arrière plan.
— `bg` poursuit l'exécution du dernier programme en arrière plan.
— `fg` poursuit l'exécution du dernier programme en premier plan.

```
$ sleep 9
^Z
[1]+  Stopped                 sleep 9
$ jobs
[1]+  Stopped                 sleep 9
$ bg
[1]+ sleep 9 &
$ fg
sleep 9
```

Bien qu'il soit conçu pour lancer des processus, le shell est un langage de programmation à part entière comportant notamment les branchement conditionnels et boucles de contrôle habituelles. Cela permet d'automatiser toute tâche consistant en une répétition de commandes shell, c'est-à-dire quasiment toutes les tâches répétitives.

```
$ if sleep 1; then date; fi
$ for i in 1 2 3 4 5; do date & sleep 1; done
$ while sleep 1; do date; done
```

Nous verrons plus tard les éléments qui composent ces boucles en détail.

Chapitre 4

Documentation

Certains aspects particuliers des systèmes d'exploitation sont documentés avec beaucoup d'ardeur et de clarté ; d'autres pas. Ce chapitre présente les différentes voies par lesquelles la documentation s'offre au lecteur. Nous recommandons de recourir exclusivement au manuel et aux ressources Web gratuites ; seuls ceux qui souhaitent s'intéresser de près aux systèmes d'exploitation (au delà de ce cours) trouveront un intérêt dans l'achat d'un ouvrage comme [7].

4.1 Le manuel

La première source d'information est *le manuel*, accessible par le programme `man` ; il documente l'interface du système dans son entièreté : l'utilisation du shell, les commandes disponibles, la syntaxe des programmes, les formats des fichiers de configuration, les bibliothèques utilisées, etc. Son format est relativement spartiate, aussi faut-il généralement savoir ce qu'on y cherche.

```
LS(1)                          User Commands                          LS(1)


NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --
sort  is  speci-
       fied.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -a, --all
              do not ignore entries starting with .

       [...]

AUTHOR
       Written by Richard M. Stallman and David MacKenzie.


GNU coreutils 8.22             December 2013                          LS(1)
```

4.2 Internet

Lorsqu'une erreur survient, il est de la plus haute importance de **ne pas ignorer les messages d'erreurs** ; souvent, une requête dans un bon moteur de recherche du Web mènera droit à la solution.

Le fonctionnement de chaque programme est aussi documenté par ses développeurs sous des formats qui peuvent aller d'ouvrages imprimés au simple code source. Quoi qu'il en soit, les développeurs sont évidemment les personnes connaissant le mieux leurs logiciels. Leurs écrits sont une source précieuse d'informations correctes.

4.3 La littérature

Les cours sur les systèmes de type Unix en ligne ne manquent pas ; ils se séparent en cours théoriques sur les fondements d'Unix et pratiques sur l'utilisation d'Unix. Pour les francophones, nous recommandons [2] et [6] dans ces deux catégories respectives, même s'ils ne sont pas nécessairement aussi complet que l'on pourrait vouloir.

De très bons ouvrages papiers existent aussi sur le sujet. Malgré son âge et sa sécheresse, [5] reste une excellente introduction à la pratique des systèmes de type Unix. L'ouvrage [1] traite quant à lui de la conception et de l'organisation interne du système UNIX d'un point de vue relativement technique. Enfin, [7] porte sur les systèmes d'exploitation, en décrit les problématiques et en présente les techniques — c'est l'ouvrage de référence sur le sujet.

4.4 Les standards

Nos lecteurs les plus curieux pourront regarder les standards ; ils visent à normaliser de multiples aspects des différentes distributions Linux voire même des différents systèmes de type Unix. Même si les standards ne sont pas toujours avisés ou respectés par les distributions, ils expliquent beaucoup des choix sous-jacents à l'organisation des systèmes récents.

Le standard POSIX [4] décrit certaines interfaces des systèmes de type Unix, y compris le shell et les programmes standards. Ce standard a été repris dans LSB [3] qui vise à normaliser les systèmes de type Unix de manière complètement exhaustive.

Chapitre 5

Concepts avancés des processus

5.1 Exécution et code de retour

Le noyau offre deux mécanismes de création de nouveaux processus :

— `exec()` remplace le processus qui l'appelle par le programme spécifié en argument ; le PID ne change pas.

— `fork()` effectue une copie conforme du processus appelant ; elle se voit affecter un nouveau PID.

Au démarrage du système, le noyau lance le premier processus, `init`, qui prend pour PID 1 ; c'est alors à lui de lancer tous les autres, exclusivement par combinaisons de `fork()` et `exec()`. Lors de chaque appel à `fork()`, le processus appelant est qualifié de père et sa copie nouvellement créée de fils. Cela donne lieu à la hiérarchie des processus qui forme un arbre enraciné.

```
$ pstree
systemd-+-collectdmon---collectd
        |-crond
        |-dbus-daemon
        |-dhcpcd
        |-dnsmasq
        |-login---startx---xinit-+-X
        |                        `-dwm-+-sh-+-conky
        |                              |    `-sh
        |                              |-xterm---bash-+-mupdf
        |                              |              `-vim
        |                              |-xterm---bash---pstree
        |                              `-xterm---bash---screen
        |-mpd
        |-ntpd
        |-screen---weechat-curses
        |-screen---bash---mpv
        |-sshd
        |-systemd-journal
        |-systemd-logind
        |-systemd-udevd
        |-thttpd
        |-unbound
        `-xplanet
```

Il est courant qu'un processus veuille lancer un programme auxiliaire tout en continuant son exécution propre. Pour cela, il appel `fork()` puis `exec()` pour remplacer le processus fils par le programme voulu. C'est ce que le shell fait par défaut.

En shell, on peut demander l'exécution d'un programme en arrière-plan ; il est alors lancé par `fork()` puis `exec()` mais le shell n'attend pas que ce programme termine pour continuer son exécution — on dit que le processus est détaché du shell. Nous avons déjà vu que la liste des processus lancés depuis le shell peut être consultée par la commande `jobs`, que les commandes `fg` et `bg` permettent de rattacher et détacher un processus et que `^Z` interrompt le processus courant.

```
$ sleep 3 &
[1] 5770
$ jobs
[1]+  Running                 sleep 3 &
$ fg
sleep 3
^Z
[1]+  Stopped                 sleep 3
$ bg
[1]+ sleep 3 &
$ jobs
[1]+  Done                    sleep 3
```

On peut aussi appliquer ces concepts à un bloc de commandes que l'on délimite par des accolades.

```
$ { sleep 3; date; } & date
[1] 421
Fri Dec 20 16:59:57 TAHT 2013
$ Fri Dec 20 17:00:00 TAHT 2013
```

On peut aussi appeler `exec()` mais attention : cela remplace le shell par votre programme.

```
$ exec date
Fri Dec 20 16:59:59 TAHT 2013
[fini le shell]
```

Lorsqu'il termine, tout processus émet un entier appelé code de retour que son père peut récupérer par l'appel système `wait()`. Par convention, un code de retour nul signifie que l'exécution du fils a été un succès ; lorsque le code de retour est non nul, sa valeur exacte peut indiquer la nature de l'erreur.

En shell, le code de retour du dernier processus terminé se note `$?`.

```
$ date; echo $?
Fri Dec 20 16:59:59 TAHT 2013
0
$ date mauvais; echo $?
date: invalid date ‘mauvais’
1
$ ls inexistant; echo $?
ls: cannot access inexistant: No such file or directory
2
```

Exercice. *Compiler puis lancer le programme C ci-dessous.*

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[]) {
        int r;
        for(;;) {
                sleep(2);
                r = fork();
                if (r<0) printf("%i n'a pas pu forker.\n", getpid());
                if (r>0) printf("%i a pour fils %i.\n", getpid(), r);
                if (r==0) printf("%i a pour père %i.\n", getpid(), getppid());
        }
        return 0;
}
```

Se renseigner sur `fork()` *pour comprendre ce qu'il fait. Pour consulter les limites que le système impose, notamment sur le nombre de processus par utilisateur, lancer alors la commande* `ulimit -a`.

5.2 Environnement d'exécution

Chaque processus s'exécute dans un environnement qui s'hérite de père en fils ; il contient notamment :

- `argv`, les arguments du programme
- `pid`, le numéro du processus
- `ppid`, le numéro du processus père
- `umask`, la valeur de `umask`
- `nice`, la priorité du processus
- `env`, les variables d'environnement

En particulier, changer la valeur de `umask` ne la change que dans les descendants du processus. Pour la changer « partout » il faut effectuer ce changement dans le premier processus lancé à la connexion, c'est-à-dire, rajouter la commande à `.profile` ou équivalent. Une fois lancé, l'environnement d'exécution d'un processus peut difficilement être modifié.

Les variables d'environnement sont un dictionnaire de couples (nom, valeur). La commande `env` en affiche la liste.

```
$ env
SHELL=/bin/bash
USER=bisson
PATH=/home/bisson/bin:/usr/local/sbin:/usr/local/bin:/usr/bin
EDITOR=vim
LANG=en_US.UTF-8
PS1=\[\e[0;32m\]\u@\h:\[\e[0;33m\]\w \[\e[0;32m\]$\[\e[0;39m\]
```

Les variables ci-dessus sont créées automatiquement par le système ; elles en configurent certains aspects. Par exemple, la variable `LANG` indique que l'utilisateur souhaite qu'on interagisse avec lui en anglais. L'utilisateur est libre de modifier les variables existantes ou d'en créer des nouvelles ; il peut assigner une valeur à une variable comme le montre l'énumération ci-dessus, et accéder à cette valeur en préfixant le nom de la variable du caractère « `$` ».

```
$ variable=valeur
$ echo $variable
valeur
```

Afin d'exporter une variable dans l'environnement hérité par les processus fils, on l'exporte.

```
$ date
Fri Dec 20 16:59:59 TAHT 2013
$ export LANG=fr_FR.UTF-8 date
vendredi 20 décembre 2013, 16:59:59 (UTC-1000)
```

Remarquer qu'on a jusqu'à présent omis de spécifier l'emplacement du programme à lancer : le shell sélectionne alors le premier un fichier exécutable portant ce nom dans un répertoire de la liste que contient la variable `PATH`.

```
$ echo $PATH
/home/bisson/bin:/usr/local/bin:/usr/bin:/bin
```

Pour `date`, il lancera finalement `/usr/bin/date`.

5.3 Entrées et sorties

Les processus interagissent avec (c'est-à-dire, lisent et écrivent) des fichiers par le biais de descripteurs de fichiers. Au sein de chaque processus, chaque descripteur de fichier est identifié par un entier. Trois sont réservés pour permettre une interactivité minimale du processus avec son environnement :

0. `stdin`, l'entrée standard ;
1. `stdout`, la sortie standard ;
2. `stderr`, l'erreur standard.

Par exemple, en C, pour lire puis afficher les 4096 premiers octets du fichier `/etc/profile` en rapportant les éventuels problèmes sur l'erreur standard, on écrirait :

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
        char *erropen = "L'ouverture du fichier a échouée.\n";
        char *errread = "La lecture du fichier a échouée.\n";
        char buf[4096+1];
        int fd, n;
        fd = open("/etc/profile", O_RDONLY);
        if (fd<0) {
                write(2, erropen, strlen(erropen));
                return 1;
        }
        n = read(fd, buf, 4096);
        if (n<0) {
                write(2, errread, strlen(errread));
                return 1;
        }
        buf[n] = '\0';
        return n==printf("%s\n", buf);
}
```

Lorsqu'on lance un programme depuis un shell, par défaut :
— Les évènements claviers reçus par le shell sont transférés à son entrée standard.
— La sortie standard s'affiche sur le shell.

— L'erreur standard s'affiche sur le shell.

Par exemple, le programme `cat` imprime sur sa sortie standard le contenu des fichiers dont les noms lui sont fournis en argument ou, à défaut d'argument, le contenu de son entrée standard ; la commande `cat /etc/group` affichera donc la liste des groupes d'utilisateurs ainsi que leurs membres.

La liste des descripteurs de fichiers ouverts par un processus est l'une des nombreuses informations grassouilles que l'on peut trouver dans le système de fichier virtuel `/proc` ; par exemple, pour un processus `bash` typique :

```
$ ls -l /proc/916/fd
total 0
lrwx------ 1 bisson users 64 Feb  2 16:40 0 -> /dev/pts/3
lrwx------ 1 bisson users 64 Feb  2 16:40 1 -> /dev/pts/3
lrwx------ 1 bisson users 64 Feb  2 16:04 2 -> /dev/pts/3
lrwx------ 1 bisson users 64 Feb  2 16:40 255 -> /dev/pts/3
```

5.4 Signaux

Les processus peuvent être contrôlés de manière primitive par le biais de signaux. Tout utilisateur peut envoyer des signaux à ses processus par la commande `kill` ; ils réagissent alors au signal d'une façon appropriée.

Les signaux les plus utilisés, `SIGSTOP`, `SIGCONT`, `SIGKILL`, permettent respectivement de mettre en pause, reprendre et arrêter totalement l'exécution un processus. On les retrouve dans la page du manuel `signal(7)` :

```
Signal     Value     Action   Comment
──────────────────────────────────────────────────────────────────────
SIGHUP        1       Term    Hangup or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no readers
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

Nous avons déjà vu qu'en shell on peut envoyer certains signaux au programme courant :

— `^C` envoie le signal `SIGINT`.

— `^Z` envoie le signal `SIGSTOP` et replace le shell en premier plan.

— `bg` envoie le signal `SIGCONT` et laisse le processus en arrière plan.

— `fg` envoie le signal `SIGCONT` et replace le processus en premier plan.

Les signaux sont un moyen de communication inter processus très limité. Il en existe d'autres aussi natifs à Unix comme la mémoire partagée, les sockets, les sémaphores ou encore

les verrous. Aucun n'est réellement satisfaisant et les applications ont ainsi souvent recours à des bibliothèques ou services tiers, notamment `dbus`.

Chapitre 6

Commandes shell avancées

6.1 Redirection de flux

En shell, on peut rediriger l'entrée standard depuis (et la sortie standard vers) un fichier grâce au chevron ouvrant « < » (et au chevron fermant « > »).

```
$ date > fichier
$ ls
fichier
$ cat < fichier
Fri Dec 20 16:59:59 TAHT 2013
```

On peut aussi rediriger des descripteurs de fichiers arbitraires vers des fichiers.

```
$ date mauvais 2> fichier
$ cat fichier
date: invalid date ‘mauvais’
```

Ou encore rediriger des descripteurs de fichiers vers des d'autres descripteurs de fichiers.

```
$ { date; date mauvais; } > fichier 2>&1
$ cat fichier
Fri Dec 20 16:59:59 TAHT 2013
date: invalid date ‘mauvais’
```

Le séparateur de commande le plus puissant en shell est le tube (*pipe*), qui redirige la sortie standard de la commande qui le précède vers l'entrée standard de celle qui le suit ; voir la figure 6.1. Comme la majorité des programmes standards d'Unix sont conçus pour traiter de l'information arrivant sur l'entrée standard et renvoyer le résultat sur la sortie standard, le tube permet de les combiner pour effecteur des opérations à la chaine. C'est un élément essentiel de ce que certains appellent la « philosophie Unix ».

Exercice. *Pour savoir ce que font les commandes suivantes, renseignez vous sur les programmes qu'elles utilisent. Comment ?* ***Grâce à leurs pages du manuel !*** *(En français si vous ne pouvez pas faire autrement.)*

```
$ cat /etc/group | grep $USER
$ last | head
$ ps auxw | cut -d\  -f1 | sort | uniq -c | sort
```

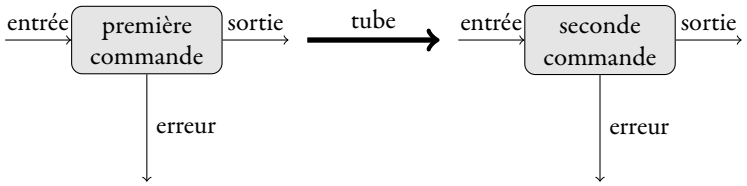

FIGURE 6.1 – Fonctionnement du tube.

Exercice. *Les fichiers situés dans* `/usr/share/dict/` *listent des mots courants ; apprenez à vous servir du programme* `grep` *pour y trouver ceux formés uniquement de voyelles.*

Exercice. *Dans le fichier* `/etc/shadow`*, les mots de passe sont chiffrés comme le fait la commande* `openssl passwd`*. Renseignez vous sur cette commande :*

```
man /usr/share/man/man1/passwd.1ssl.gz
```

Sachant qu'il s'agit bêtement d'un nom commun français, retrouver le mot de passe de l'utilisateur ayant pour entrée :

```
franck:$1$rcWVxfgW$m5TaUuu6oBiQTpAG681OE1:0:0:99999:7:::
```

6.2 Exécution conditionnelle

En shell, le séparateur `&&` permet de lancer la commande suivante si et seulement si le code de retour de la commande précédente est nul ; le séparateur `||` fait exactement le contraire.

```
$ date -u && uname
Sat Dec 21 02:59:59 UTC 2013
Linux
$ date -z && uname
date: invalid option -- 'z'
Try 'date --help' for more information.
$ date z || uname
date: invalid date ‘z’
Linux
```

On peut aussi se servir des programmes triviaux `true` et `false` qui se contentent d'émettre un code de retour respectivement nul et de valeur 1.

```
$ true && date
Fri Dec 20 16:59:59 TAHT 2013
$ true || date
$ false && uname || date
Fri Dec 20 16:59:59 TAHT 2013
```

On peut enfin combiner ces séparateurs avec des blocs de commandes

```
true && { uname; echo great success; }
```

Le plus pratique pour subordonner l'exécution d'un bloc de code à une condition reste toutefois d'utiliser le branchement conditionnel `if` comme il suit.

```
$ if true; then date; else uname; fi
Fri Dec 20 16:59:59 TAHT 2013
$ if false; then date; else uname; fi
Linux
```

On peut aussi inverser (transformer 0 en 1 et toute valeur non-nulle en 0) le code de retour d'une commande en la préfixant du symbole « `!` ».

```
$ if ! ls ~/music; then echo "Pas de répertoire music."; fi
```

Il existe une foule de programmes qui émettent un code de retour nul si et seulement si une certaine condition prescrite est vérifiée ; le plus important s'appelle `test` et peut aussi se lancer sous le nom « `[` ». Voyez sa page de manuel pour toutes ses possibilités d'utilisation.

Finissons en expliquant la syntaxe des boucles `while` qui s'exécutent tant que le programme-condition prescrit émet un code de retour nul. Par exemple, la ligne de commande ci-dessus affiche des statistiques sur le processus de PID 421 tant que celui-ci existe.

```
while [ -d /proc/421 ]; do clear; cat /proc/421/status; sleep 7; done
```

Exercice. *Écrire un programme qui affiche « le processus 421 n'existe pas » toutes les minutes tant que cela est vrai.*

Exercice. *Écrire une commande ouvrant un* `write` *sur le voisin toutes les cinq secondes tant que cela échoue.*

Exercice. *Écrire une commande envoyant par* `write` *à l'utilisateur* `prof` *le message « Monsieur, pourrais tu me rajouter au groupe* `video` *? » toutes les cinq minutes tant que ce n'est pas fait.*

6.3 Substitution et métacaractères

Le shell remplace `$(commande)` par la sortie standard de `commande` avant d'évaluer une ligne. Là où le tube permet de réutiliser un résultat sur l'entrée standard du programme suivant, la substitution permet de le réutiliser comme ses arguments.

```
$ echo "abc$(date)def"
abcSun Feb  9 14:18:45 TAHT 2014def
```

Un exemple relativement simple est de déterminer dans quel(s) fuseau(x) horaire(s) de notre planète il est actuellement dix heures du matin.

```
$ for i in $(seq -14 12); do
        TZ=Etc/GMT$i date
done | grep " 10:"
```

On peut encore vouloir se faire des ennemis grâce à la communication de masse.

```
echo 'Tous chez moi vendredi soir !' > message
for i in $(who | cut -d\  -f1 | sort -u); do
        write $i < message
done
```

Le shell remplace `*` par la liste des fichiers satisfaisant l'expression donnée. Par exemple, sur le serveur, j'ai :

```
$ echo *
bin bomb.c cours
$ echo */*
bin/crypt bin/crypt.c bin/get bin/oom.what bin/out bin/out.auth
bin/poweroff bin/redundancy bin/w3m bin/x cours/users.txt
$ echo */o*
bin/oom.what bin/out bin/out.auth
```

Un exemple serait notamment de supprimer tous les répertoires vides contenus dans le répertoire courant.

```
$ for i in *; do if [ -d $i ]; then rmdir $i; fi; done
```

Exercice. *Que fait la commande suivante ? (Ne la lancez pas bêtement !)*

```
md5sum * | sort | uniq -d -w 32 | cut -c 35- | xargs rm
```

On appelle métacaractère du shell tout caractère qui est interprété autrement que littéralement par ce dernier ; nous avons notamment déjà vu « `;&<>|$*` » mais il en existent bien d'autres : la majorité des caractères non alphanumériques sont actifs. Lorsque l'on veut taper un tel caractère sans qu'il soit interprété par le shell, on dispose de deux moyens principaux :

— le préfixer du symbole « \ », par exemple `echo \<`.
— l'entourer de guillemets anglais simples « ' », par exemple `echo '<'`.

Nous avons à présent couvert une bonne partie des fonctionnalités communes à tous les shell, avec comme syntaxe celle commune aux shell de type Bourne. Lorsque vous approfondirez votre maîtrise du shell plus avant, vous rencontrerez des spécificités de `bash` fréquemment. Certaines de ses spécificités sont des fonctionnalités extrêmement utiles que nous aurons probablement l'occasion de voir en TP, notamment :

— tableaux : `(a b c)`
— test builtin : `[[ a = b ]]`
— manipulation de valeurs : `${x//a/b}`
— arithmétique : `((a++))`

6.4 Scripts shell

Un script shell est un fichier exécutable dont la première ligne est `#!/bin/sh`. Lorsqu'il est lancé, son contenu est interprété comme des lignes de commandes, où la variable `@` dénote les arguments.

```
#!/bin/sh

if [ $HOSTNAME = aji ] || [ $@ = force ]; then
        /sbin/poweroff
else
        echo "Use: /sbin/poweroff"
fi
```

Exercice. *Combien de fois au vingtième siècle le premier janvier était-il un dimanche ?*

Exercice. *Quels sont les utilisateurs occupant plus de 50 ko d'espace disque ?*

Exercice. *Écrire un script shell qui vous délogue si vous restez inactif trop longtemps.*

Exercice. *Écrire un script shell qui identifie les trois processus les plus gourmants en mémoire et demande poliment aux utilisateurs auxquels ils appartiennent de les tuer.*

Exercice. *Écrire un script shell qui, à trois heures du matin chaque jour, lance les commandes du fichier nommés* `a-lancer-le-matin` *dans le répertoire personnel de chaque utilisateur.*

Exercice. *Écrire un script shell qui détermine les trois utilisateurs qui se sont logués le plus de fois cette dernière semaine.*

Chapitre 7

Concepts avancés du système

7.1 Partitions et montage

Un périphérique de stockage fournit essentiellement une suite de bits que l'on peut lire et écrire et qui, contrairement à la mémoire vive, subsiste à l'extinction de la machine ; ils comprennent notamment les disques durs, les cles USB, les cartes SD, les CD-ROM, etc. C'est typiquement grâce à eux que l'on stocke les fichiers des utilisateurs, le système d'exploitation et ses logs.

Afin d'y stocker l'information de manière structurée, chaque périphérique de stockage est décomposé en partitions, chacune contenant un système de fichier. Un système de fichier permet de stocker une arborescence de répertoires et de fichiers comme celle que nous manipulons depuis le début de ce cours. Essentiellement, la partition est l'espace disque alloué et le système de fichier est la manière de stocker les fichiers sur cet espace.

Les périphériques courants sont souvent décomposés ainsi :

- une clef USB n'a qu'une seule partition (système de fichiers `vfat`) ;
- un CD n'a qu'une seule partition (système de fichiers `iso9660`) ;
- un disque dur contient plusieurs partitions :
  - une pour `/boot` (système de fichiers `ext2`) ;
  - une pour `/home` (système de fichiers `jfs`) ;
  - une pour `/` en (système de fichiers `jfs`) ;

Lorsque Linux démarre, il faut lui indiquer une partition à monter à la racine « `/` », c'est-à-dire dont le système de fichier fera office d'arborescence des répertoires pour le système d'exploitation. Par la suite, on peut monter d'autres partitions à d'autres emplacements (n'importe quel répertoire vide fait l'affaire) de l'arborescence.

Pour monter des partitions, on utiliser le programme `mount` comme il suit.

```
$ ls -l /boot
total 0
$ mount /dev/sda1 /boot
$ ls -l /boot
total 23229
-rw-r--r-- 1 root root 16774806 Feb 24 07:11 initramfs-linux-fallback.img
-rw-r--r-- 1 root root  2920869 Feb 24 07:11 initramfs-linux.img
lrwxrwxrwx 1 root root       28 Jul 24  2011 kernel26-fallback.img -> initramfs-
linux-fallback.img
lrwxrwxrwx 1 root root       19 Jul 24  2011 kernel26.img -> initramfs-linux.img
drwxr-xr-x 2 root root     1024 Jan 18 13:08 syslinux
```

```
-rw-r--r-- 1 root root  3978736 Feb 22 13:26 vmlinuz-linux
$ umount /boot
$ ls -l /boot
total 0
```

7.2 Systèmes de fichiers

Un système de fichier gère le stockage des fichiers au sein des partitions. C'est un travail extrêmement difficile : considérez par exemple le scénario suivant.

Exemple. *Supposons qu'un système de fichiers choisisse de stocker les fichiers de manière séquentielle et continue. Sur une partition de* 6 *Go, on écrit un premier fichier* `A` *de* 2 *Go puis un second fichier* `B` *de la même taille. Deux jours plus tard, on efface* `A`*; peut-on alors écrire un fichier* `C` *de* 3 *Go ? Si oui, où ?*

Il existe de nombreux systèmes de fichiers présentant chacun des fonctionnalités propres : certains sont optimisés pour la manipulation de nombreux petits fichiers, d'autres pour celle des gros fichiers, d'autres présentent les fichiers d'une partition distante, etc.

```
$ du /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/
428     /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/btrfs
180     /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/cifs
8       /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/cramfs
44      /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/ecryptfs
248     /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/ext4
52      /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/fat
108     /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/jfs
88      /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/nilfs2
140     /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/reiserfs
400     /usr/lib/modules/3.13.5-1-ARCH/kernel/fs/xfs
```

Un des principes d'Unix est d'exposer le plus de données possibles par le biais de fichiers, en faisant ainsi une interface quasi universelle. Le noyau propose ainsi des systèmes de fichiers spéciaux, notamment :

— `tmpfs` stocke tout en mémoire volatile ;
— `sysfs` fournit des informations sur le noyau ;
— `proc` donne des informations sur les processus.

En pratique, on crée un système de fichiers par la commande `mkfs` et on le répare (lorsque l'on suspecte l'existence d'erreurs, par exemple lorsque la machine s'est éteinte soudainement à cause d'une coupure de courant) on utilise `fsck`. La majorité des systèmes de fichiers Linux n'a pas besoin d'être défragmentée : elle est conçue de sorte à ce que les performances ne soient pas affectées par la fragmentation.

```
$ dd if=/dev/null bs=4k seek=1M of=fs
$ stat fs
  File: `fs'
  Size: 4294967296      Blocks: 0          IO Block: 4096   regular file
Device: 806h/2054d      Inode: 21233696    Links: 1
$ /sbin/mkfs.ext2 fs
mke2fs 1.42.5 (29-Jul-2012)
fs is not a block special device.
Proceed anyway? (y,n) y
$ mkdir mnt
```

```
$ mount fs mnt
$ echo incroyable > mnt/fichier
$ grep fichier fs
Binary file fs matches
```

Sous Linux, les systèmes de fichiers traditionnels séparent le contenu d'un fichier de ses métadonnées. Les métadonnées (permissions, UID, GID, taille, dates de dernière lecture et modification, etc.) se trouvent dans un `inode` qui contient aussi des pointeurs vers un arbre dont les feuilles sont les blocs formant le contenu du fichier. Cette structure d'arbre permet de fragmenter les fichiers tout en gardant un accès rapide au contenu : en un nombre logarithmique d'accès disques.

Il est utile, même si vous n'avez pas encore vu ce concept, de regarder la `struct` correspondante dans `/usr/include/bits/stat.h` :

```
struct stat {
    __dev_t st_dev;             /* Device. */
    __ino_t st_ino;             /* File serial number. */
    __mode_t st_mode;           /* File mode. */
    __nlink_t st_nlink;         /* Link count. */
    __uid_t st_uid;             /* User ID of the file's owner. */
    __gid_t st_gid;             /* Group ID of the file's group. */
    __dev_t st_rdev;            /* Device number, if device. */
    __off_t st_size;            /* Size of file, in bytes. */
    __blksize_t st_blksize;     /* Optimal block size for I/O. */
    __blkcnt_t st_blocks;       /* Number 512-byte blocks allocated. */
    struct timespec st_atim;    /* Time of last access. */
    struct timespec st_mtim;    /* Time of last modification. */
    struct timespec st_ctim;    /* Time of last status change. */
}
```

Un répertoire n'est qu'un fichier spécial dont le contenu n'est autre que la liste des éléments qui s'y trouvent : pour chaque entrée (fichier ou sous-répertoire), on stocke son nom et l'inode correspondant. Il est ainsi possible de faire pointer deux fichiers aux noms et/ou chemins différents vers le même inode. Cela s'appelle un lien dur.

```
$ date > a
$ stat a
  File: ‘a’
  Size: 30              Blocks: 8          IO Block: 4096   regular file
Device: 803h/2051d      Inode: 4203        Links: 1
$ ln a b
$ cat b
Fri Dec 20 16:59:59 TAHT 2013
$ stat b
  File: ‘b’
  Size: 30              Blocks: 8          IO Block: 4096   regular file
Device: 803h/2051d      Inode: 4203        Links: 2
```

Attention, les liens durs admettent deux limitations majeures (pour des raisons évidentes) : un répertoire ne peut pas être un lien dur, et on ne peut pas lier deux fichiers sur des partitions distinctes.

Exemple. *Une application typique des liens durs et la création d'un système de tags. Par exemple, le même fichier vidéo* `film` *peut se trouver à la fois sous les noms* `comédie/film` *et* `tragédie/film`.

Exercice. *Créer un fichier nommé* `one`. *Créer un lien dur* `two` *de* `one`. *Lancer la commande* `stat` *sur ces deux fichiers. Supprimer* `one`. *Lancer la commande* `stat` *sur* `two`.

Exercice. *Identifier tous les fichiers de* `/usr/bin/` *qui sont des liens durs d'autres fichiers.*

D'autres mécanismes de contrôle des fichiers existent et utilisent les attributs étendus. Un exemple important sont les ACL (Access Control List) qui permettent de donner des droits à des utilisateurs spécifiques.

Exercice. *Utiliser les commandes* `setfacl` *et* `getfacl` *pour créer un fichier que seul vous et votre voisin de droite pourrez lire et écrire.*

7.3 Fichiers spéciaux

La majorité des fichiers sont dits « réguliers », c'est-à-dire qu'ils présentent des données dictées par le système de fichiers, que ce soit `/etc/profile` ou `/proc/cpuinfo`. Par opposition, les autre fichiers, qui sont « interprétés » par le noyau, sont qualifiés de spéciaux.

Les fichiers spéciaux comprennent :

- **les répertoires**, dont le noyau interprète le contenu comme une liste de couples (nom, inode).
- **les liens symboliques**, dont le noyau interprète le contenu comme l'emplacement d'un fichier à substituer ; on les crée grâce à la commande « `ln -s` ».
- **les tubes nommés** (***fifo***), qui se comportent comme un tube, c'est-à-dire que leur contenu est volatile et transporté en un seul exemplaire de leur entrée à leur sortie ; on les crée en utilisant le programme « `mkfifo` ».
- les fichiers de périphériques :
  - par caractères :
    - **full**, plein
    - **zero**, infini
    - **null**, trou noir
    - **urandom**, aléatoire
    - **tty***, représentent les terminaux
  - par bloc :
    - **sda***, représentent le disque
    - **mmcblk***, représentent la carte mémoire

Exemple. *L'application phare des liens symboliques est pour fournir des raccourcis sur le système de fichier. Par exemple, vous pouvez créer un lien symbolique* `chez_momo` *dans votre répertoire personnel pointant vers* `/home/momo` *afin de pouvoir aisément taper* `cd chez_momo`.

Dans les exercices ci-dessus, faire particulièrement attention aux différences entre les liens durs et symboliques.

Exercice. *Créer un fichier nommé* `one`. *Créer un lien symbolique* `two` *pointant vers* `one`. *Lancer la commande* `stat` *sur ces deux fichiers. Supprimer* `one`. *Lancer la commande* `stat` *sur* `two`. *Essayer alors de lire le contenu de* `b`.

Exercice. *Créer un répertoire et mettez-y un lien symbolique pointant vers ce répertoire-même. On a alors des répertoires de profondeur infinie. Écrire maintenant un script qui explore une arborescence de répertoire en affichant les noms de tous les sous-répertoires qu'elle contient. Lancer ce script sur le répertoire ci-dessus.*

Exercice. *Utilisez la commande* `mkfifo` *pour dialoguer avec votre voisin de gauche.*

7.4 Autres composants systèmes

init

pam

X

La liste des mots de passe est stockée chiffrée dans le fichier `/etc/shadow` (lisible par `root` seul) et à chaque tentative de logue le système y compare simplement le chiffré du mot de passe tapé.

...

Chapitre 8

Programmation

Pour les programmeurs, pas les bébés : unlink() désaloue un fichier sans prévoir que l'utilisateur puisse ensuite vouloir le récupérer...

On dit que c'est bas niveau.

8.1 Compilation

Typiquement, on télécharge les sources (ASM, C, C++, Ocaml, etc.) puis ./configure `&&` make .

éviter make install ; passer par le gestionnaire de paquet

options notamment passées via : - CC - CFLAGS - LDFLAGS

8.2 Interfaces

code machine Linux glibc bibliothèques procfs/sysfs

appels systèmes classiques...

structures classiques (process, fd, stat, etc.)

8.3 Édition de texte

deux vrais éditeurs :

vim

emacs

8.4 Expressions régulières

grep ne cherche pas que des mots, il cherche des expressions ; exemple `grep '^[aeiouy]*$'` sélectionne les lignes formées uniquement de voyelles

ces expressions dites régulières sont en fait supportés par de nombreux programmes (grep, sed, awk, perl, vim) même si certaines subtiles différences existent dans leur syntaxe. ici, on apprend la syntaxe standard de grep.

Pourquoi des expressions régulières ? Parce qu'un théorème de Kleene nous dit qu'on peut tester une expression régulière grâce à un automate fini, ce qui est extrêmement rapide.

caractères spéciaux : - `^` début de ligne - `$` fin de ligne - `.` n'importe quel caractère - `[...]` n'importe quel caractère entre les crochets - `[^...]` n'importe quel caractère pas entre les crochets - `*` le caractère précédent apparait zéro fois ou plus - `\?` le caractère précédent apparait zéro ou une fois - `\+` le caractère précédent apparait une fois ou plus - `\{n,m\}` le caractère précédent apparait entre `n` et `m` fois. - `\(...\)` un bloc de caractères - `...\|...` un bloc ou l'autre

Le caractère d'échappement `\` permet d'obtenir un caractère litéral : `\$` dénote le symbole dollar.

Entre crochet, on peut utiliser des classes de caractères : - `0-9` - `a-z` - `A-Z` - `[:alnum:]` - `[:print:]` - `[:punct:]` - `[:space:]`

Voir la section `REGULAR EXPRESSIONS` de la page du manuel sur `grep`, elle est très bien faite.

Chapitre 9

Administration système

9.1 Compte super-utilisateur

compte root, souvent le seul créé à l'installation
nécessaire pour toute modification système : - mise à jours - gestion des comptes
root peut se loguer (en local ou à distance), mais ce n'est pas bien, on préfère se loguer en tant qu'utilisateur normal puis passer root quand le besoin se présente.
Pour passer root : élévation de privilège (su, sudo) : - setuid bit - setgid bit - pas de script suid ; il faudrait que ce soit l'interpréteur

Exercice. *Trouver tous les programmes de* `/usr/bin` *qui font usage de l'élévation de privilège.*

9.2 Gestion des comptes

création, modification, suppression
quota, ulimit

9.3 Gestion des logiciels

notion de paquet ; taches : - installation - mise à jour - supression
un paquet contient : - des métadonnées (nom, version, liste des dépendences) - des fichiers à mettre dans /usr - des scripts de pré/post installation
mise à jour sur les distributions classiques

9.4 Sécurité

éthique root : - mettre à jour fréquemment - quand on ne sait pas, on ne fait pas... - on n'espionne pas ses utilisateurs
gestion des mots de passe : lancer john the ripper
lire les release announcements des paquets mis à jours (ou au moins les News de la distro)

Chapitre 10

Problèmes

Il serait judicieux de créer un répertoire dédié à chaque problème.

10.1 Gestion des mots de passes

On souhaite écrire une suite de programmes permettant de gérer les mots de passe de comptes utilisateurs. Toutes les informations nécessaires sont stockées dans un fichier appelé « `shadow` » dont le format est documenté dans la section cinq du manuel. Dans un premier temps, pour simplifier :

— On ignorera les champs relatifs à l'age des mots de passe et leur expiration.
— On supposera que les mots de passes sont tous chiffrés comme le fait la commande : `openssl passwd -1 -salt "alea" "mot de passe"`

1. Écrire un programme nommé `useradd` qui rajoute une ligne au fichier « `shadow` » correspondant au nom d'utilisateur et mot de passe spécifiés.
2. Écrire un programme nommé `login` qui demande à l'utilisateur son nom et son mot de passe et, si l'authentification est un succès, renvoie alors un code de retour nul. Dans le cas inverse, il renverra un code de retour non nul et afficher un message sur l'erreur standard.
3. Écrire un programme nommé `userdel` permettant de supprimer un compte utilisateur donné.
4. Écrire un programme nommé `usermod` permettant de modifier un utilisateur donné.
5. Un utilisateur malavisé a bêtement choisi un nom commun pour mot de passe ; le retrouver sachant que la ligne du fichier « `shadow` » correspondante est la suivante.

   `franck:$1$rcWVxfgW$m5TaUuu6oBiQTpAG681OE1:1:0:99999:7:::`

10.2 Courrier électronique

Le courrier électronique n'est que du texte ; pour envoyer un message à l'adresse `<user@example.com>`, on procède suivant les étapes ci-dessous.

1. On détermine le serveur qui gère le courrier électronique du domaine `example.com` (par une requête DNS de type MX), typiquement `smtp.example.com`.

2. On ouvre une connexion TCP vers le port 25 de `smtp.example.com` ; le serveur nous dit bonjour. On indique alors l'expéditeur et les destinataires du message ; le serveur fait des vérifications minimales (notamment, qu'il a connaissance des destinataires).
3. Si le serveur l'accepte, on lui envoie alors le message proprement dit ; il se compose de :
   (a) en-têtes, typiquement :
      — `Date:`
      — `From:`
      — `To:`
      — `Subject:`
      — `References:`
      — `Content-Type:` (texte brut, HTML ou autre)
   (b) corps du message, c'est-à-dire le contenu réel (formaté suivant `Content-Type:`)
4. La connexion est alors fermée.
5. Le serveur effectue éventuellement des vérifications additionnelles (que le message ne soit pas du spam).
6. Le serveur dépose enfin le message dans la boite de l'utilisateur concerné, un fichier situé dans le répertoire `/var/spool/mail` qui porte le nom de l'utilisateur.

Le logiciel réalisant les étapes ci-dessus s'appelle un « mail transfer agent » (MTA). L'utilisateur peut alors consulter les message déposés dans le fichier `/var/spool/mail/user` quand et comme bon lui semble ; il utilise typiquement un « mail user agent » (MUA) qui affiche les messages sous forme de conversations, propose d'éditer une réponse, etc. Sur le serveur du cours, le MTA utilisé s'appelle `postfix` et deux MUA sont installés, `mailx` et `mutt`.

Voici un exemple de communication entre un MTA (bleu) et un serveur (rouge).

```
220 VM-ServeurUnixL1.adupf.local ESMTP Postfix (Debian/GNU)
HELO localhost
250 VM-ServeurUnixL1.adupf.local
MAIL FROM: prof@localhost
250 2.1.0 Ok
RCPT TO: prof@localhost
250 2.1.0 Ok
RCPT TO: root@localhost
250 2.1.0 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
Date: Tue, 01 Apr 2014 07:08:09 -1000
From: Gaetan Bisson <bisson@vesath.org>
To: Prof Unix <unix@prof.upf.pf>
Subject: Essaie de messagerie

Salut,

Est-ce que ce message arrive bien chez toi ?

--
Gaetan
.
250 2.0.0 Ok: queued as 0C5132130C
QUIT
221 2.0.0 Bye
```

Le serveur dépose alors ce message dans le fichier `/var/spool/mail/prof` qui contient donc :

```
From prof@localhost  Tue Apr  1 08:09:10 2014
Return-Path: <prof@localhost>
X-Original-To: prof@localhost
Delivered-To: prof@localhost
Received: from localhost (localhost [127.0.0.1])
        by VM-ServeurUnixL1.adupf.local (Postfix) with SMTP id B07B22130C;
        Tue,  1 Apr 2014 08:09:10 -1000 (TAHT)
Date: Tue, 01 Apr 2014 07:08:09 -1000
From: Gaetan Bisson <bisson@vesath.org>
To: Prof Unix <unix@prof.upf.pf>
Subject: Essaie de messagerie
Message-Id: <20140401202056.B07B22130C@VM-ServeurUnixL1.adupf.local>

Salut,

Est-ce que ce message arrive bien chez toi ?

--
Gaetan
```

Questions :

1. Envoyer un message à votre voisin en parlant manuellement avec le serveur via la commande « `netcat localhost 25` ».
2. Consulter le message que vous avez reçu par la commande « `cat` » puis « `mutt` ».
3. Écrire un script qui permet d'envoyer un message arbitraire à un utilisateur arbitraire.
4. Écrire un script qui permet de consulter les messages reçu de manière moins spartiate que « `cat` ».

Bibliographie

[1] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986. isbn : 0-132-01799-7.

[2] Jacques Beigbeder. *Programmation système Unix*. Cours de second semestre de l'École normale supérieure. url : `http://www.spi.ens.fr/beig/systeme/`.

[3] The Linux Foundation. *Linux Standard Base. LSB 4.1 Specification*. 2011. url : `http://refspecs.linuxfoundation.org/lsb.shtml`.

[4] IEEE et The Open Group. *POSIX.1-2008. IEEE Std 1003.1-2008 and The Open Group Technical Standard Base Specifications, Issue 7*. 2008. url : `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[5] Brian W. Kernighan et Rob Pike. *The Unix Programming Environment*. Prentice Hall, 1984. isbn : 0-13-937699-2.

[6] *Pages Unix*. Tuteurs informatique de l'École normale supérieure. url : `http://www.tuteurs.ens.fr/unix/`.

[7] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. isbn : 0-135-88187-0.

[8] Wikipedia. *Book on Unix*. 2014. url : `http://en.wikipedia.org/wiki/Book:Unix`.