Shell scripting avec GNU Bash sous Linux

Frédéric Goualard

2022-10-19, v. 0.2

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity. —
Dennis M. Ritchie.

L'un des premiers ordinateurs électroniques, l'ENIAC, mis en service à la fin de l'année 1945, se programmait en raccordant physiquement par des câbles les panneaux contenant les instructions à exécuter (figure 2). Tous les éléments composant l'ordinateur étaient à taille humaine et pouvaient facilement être manipulés par les programmeuses chargées de l'approvisionner continuellement en nouveaux problèmes. Chaque programme était écrit en termes des instructions disponibles sur la machine et utilisait la pleine connaissance de ses capacités, ce qui rendait impossible la réutilisation du code pour une autre machine de conception différente.

La complexité des machines augmentant, on se mit à écrire des morceaux de code chargés de gérer certains des aspects les plus pénibles de l'interaction avec un ordinateur, comme les entrées/sorties, pour finalement obtenir un programme servant d'interface entre l'être humain et toutes les ressources de la machine : le *système d'exploitation* (ou *OS*, pour *Operating System*). Intégré à l'OS, un petit programme interactif a très tôt permis à l'utilisateur d'interagir avec la machine via des instructions simples : le *shell*. Ce petit programme s'est étoffé au fil du temps pour devenir un langage de programmation à part entière.

Nous verrons dans la section 1 les principaux éléments à retenir sur les systèmes d'exploitation, en commençant par un petit historique, et dans les sections 2 et 3 les bases de la programmation avec le langage Bash, le *shell* installé par défaut sur la plupart des distributions de Linux.

Dans ce fascicule, un texte formaté en bleu fait l'objet d'une définition dans un encadré séparé de même couleur. Un texte formaté en vert est le nom d'une commande utilisable dans le terminal Bash et possédant une entrée dans l'index en fin de document.

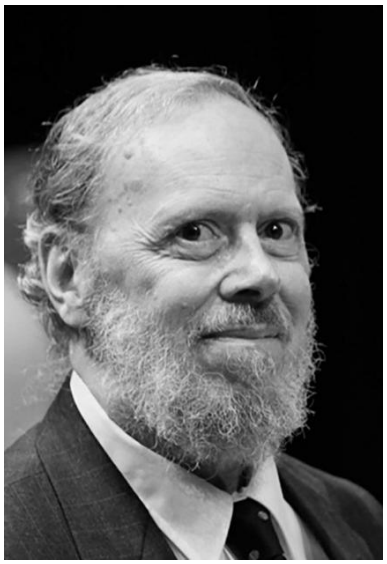


Figure 1 : Dennis M. Ritchie, 1941–2011. Crédit photo : National Inventors Hall of Fame.

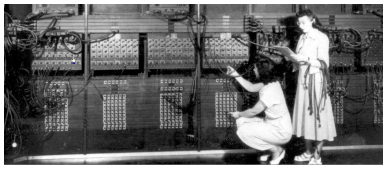


Figure 2 : Marlyn Meltzer (1922–2008) et Ruth Teitelbaum (1924–1986), programmant l'ENIAC. Crédit photo : US Army Research Laboratory.

1 Les systèmes d'exploitation

1.1 Un peu d'histoire

Les premiers ordinateurs commerciaux des années cinquante (construits en série et non spécifiquement pour une tâche et un site précis comme l'ENIAC) étaient des mastodontes gourmands en énergie réalisés à un nombre très réduit d'exemplaires. L'IBM 701, par exemple, tenait dans une pièce de douze mètres de côté et chacun des dix-neuf exemplaires produits était loué pour la « modique somme » de $23 750 par mois (environ $241 000 en 2021)[1].

Ces ordinateurs ne pouvaient exécuter qu'un seul programme à la fois et l'introduction d'un nouveau programme requérait un travail conséquent de la part du programmeur et d'un opérateur : avant même de pouvoir insérer les nombreuses cartes

[1] Robert L. Patrick. *General Motors/North American Monitor for the IBM 704 Computer*. en. Rapp. tech. RAND Corporation, jan. 1987. URL : https://www.rand.org/pubs/papers/P7316.html (visité le 07/09/2021)

Figure 3 : Carte perforée de 80 colonnes. Crédit photo : Wikimedia Commons.

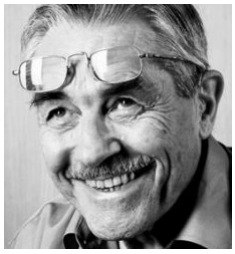


Figure 4 : Louis Pouzin (1931–). Crédit photo : Internet Hall of Fame.

[2] Douglas Ross. "A personal view of the personal work station : some firsts in the Fifties". In : *Proceedings of the ACM Conference on The history of personal workstations*. HPW '86. Association for Computing Machinery, jan. 1986, p. 19-48

[3] Fernando J. Corbató, Marjorie Merwin-Daggett et Robert C. Daley. "An experimental time-sharing system". In : *Proceedings of the May 1-3, 1962, spring joint computer conference*. AIEE-IRE '62 (Spring). Association for Computing Machinery, mai 1962, p. 335-344

perforées (figure 3) encodant le programme dans le lecteur adéquat, il fallait reconfigurer physiquement les différentes parties constituant la machine : système de lecture des cartes, sytème d'impression du résultat, horloge enregistrant le temps de calcul (pour facturer l'utilisation de la machine au bon service, ...). Chaque programmeur venait à son tour reconfigurer la machine en fonction de ses besoins avec l'aide de l'opérateur. Sur de nombreux sites, chacun inscrivait son nom sur une feuille de papier afin de respecter un ordre « premier arrivé, premier servi ». Lorsqu'un programme était fini, ou qu'il échouait, l'opérateur invitait le programmeur suivant sur la liste à rentrer dans la salle pour faire exécuter son programme.

Entre les temps de reconfiguration après chaque programme et les temps d'attente de la machine lorsqu'elle devait lire ou écrire des données sur un périphérique lent (une imprimante, par exemple), le temps réellement utile de ces machines pourtant coûteuses était très réduit.

À partir de 1954, John Frankovich (1928–2018) et Frank Helwig commencèrent à écrire un programme pour exécuter automatiquement les paramétrages nécessaires à l'entrée d'un nouveau programme utilisateur sur l'ordinateur Whirlwind du *Massachusetts Institute of Technology*[2] (MIT) : le *Director Tape Program*. En ayant comme seul but de faciliter l'accès aux ressources d'un ordinateur pour un autre programme, ce programme devint l'ancêtre des systèmes d'exploitation.

La puissance des ordinateurs augmentant, il devint rapidement intéressant de passer d'un modèle purement séquentiel, où un ordinateur n'exécute à tout moment qu'un seul programme et ne sert donc qu'un seul utilisateur, à un modèle de *temps partagé* où un ordinateur partage son temps entre plusieurs programmes qu'il exécute chacun à leur tour pendant un quantum de temps suffisamment court pour que les utilisateurs aient l'illusion d'une exécution en parallèle de tous les programmes. On a alors besoin d'un programme capable de reconfigurer automatiquement le matériel pour le programme utilisateur à exécuter et apte à passer la main d'un programme à un autre en garantissant la sauvegarde de tout ce qui a été calculé durant le quantum de temps. En 1962, Corbató *et al.*[3] présentent CTSS, un programme expérimental capable d'assurer le partage de temps sur un IBM 709 puis sur un IBM 7090. On passe alors d'un système d'exploitation qui permettait seulement un gain de temps par rapport au recours à un opérateur humain à un système d'exploitation indispensable et irremplaçable pour la bonne marche de la machine.

En 1964, *General Electrics*, le *MIT* et les *Bell Labs* commencent le développement du système d'exploitation Multics pour l'ordinateur GE 645. Ce système est à l'origine de nombreuses avancées présentes dans les systèmes d'aujourdhui, dont la notion de droits d'accès sur les fichiers, les liaisons dynamiques de bibliothèques et un langage d'interface entre le système d'exploitation et l'utilisateur.

Ce langage d'interface trouve ses origines dans un programme, RUNCOM, écrit par Louis Pouzin pour interpréter des commandes tapées sur une console d'entrée pour chaîner de manière complexe les requêtes faites à l'ordinateur via Multics. Incidemment, c'est à Pouzin que revient l'honneur d'avoir introduit le terme « *shell* » pour parler d'un tel langage d'interface[4].

Le programme RUNCOM n'est pas encore un vrai langage de programmation mais en utilisant les idées qu'il a introduites, Glenda Schroeder crée en 1965 le premier langage de *shell* pour Multics, ancêtre de tous les *shell* existant actuellement[5].

Le système Multics était une révolution dans l'univers des systèmes d'exploitation par le nombre d'avancées dont il était à l'origine. Le revers de la médaille était sa complexité et sa lourdeur, qui poussèrent les chercheurs des *Bell Labs* à abandonner le projet et à repartir de zéro avec un nouveau système d'exploitation plus simple : *UNIX*. Initialement, *UNIX* n'offrait pas les possibilités *multi-tâches* de Multics, mais il fut progressivement modifié dans les années soixante-dix et réécrit entièrement en C pour devenir un système multi-tâches facilement portable sur divers ordinateurs, ce qui contribua à sa popularité.

[4] Louis Pouzin. *The SHELL : A Global Tool for Calling and Chaining Procedures in the System*. Section IV of the Design Notebook. Massachusetts Institute of Technology, avr. 1965

Figure 5 : Glenda Schroeder. Crédit photo : Tech-heroines.

[5] Louis Pouzin. *The Origin of the Shell*. Nov. 2000. URL : https://www.multicians.org/shell.html (visité le 07/09/2021)

Aujourd'hui, les systèmes Linux, macOS, iOS et Android sont tous des héritiers plus ou moins directs de *UNIX*. Le système MS Windows depuis sa version *Windows NT* dérive, lui, d'un système concurrent plus récent : OS/2.

1.2 Qu'est-ce qu'un système d'exploitation ?

Comme on l'a vu dans l'historique de la section précédente, un système d'exploitation est un programme ou un ensemble de programmes servant d'interface entre le matériel et l'utilisateur, que ce soit l'être humain lui-même ou les programmes qu'il souhaite exécuter.

Un système d'exploitation moderne a deux objectifs :

- Donner une vision uniforme des matériels utilisés : l'utilisateur ne veut pas connaître les détails techniques de l'imprimante connectée à son ordinateur ; il lui suffit de savoir qu'elle peut imprimer en noir et blanc ou en couleur ;
- Exploiter et protéger les ressources de l'ordinateur : un utilisateur ne doit pas pouvoir accéder aux fichiers d'un autre ou utiliser plus de mémoire qu'il ne lui a été alloué, par exemple.

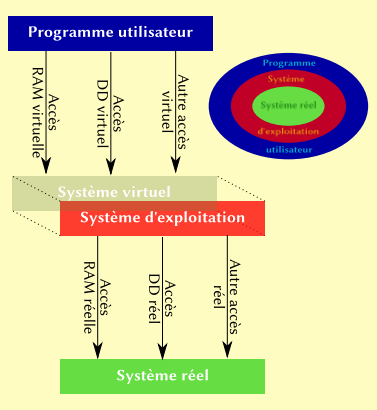


FIGURE 6 : Virtualisation des ressources par l'*OS*.

Les ressources.

Une ressource est un élément réel ou virtuel en quantité limitée à l'intérieur d'un système informatique. Quelques exemples de ressources :

- L'imprimante ;
- Le temps de processeur ;
- Les cases de la mémoire RAM ;
- L'espace du disque dur ;
- ...

Pour atteindre ces deux objectifs, le système va utiliser un même mécanisme : la virtualisation des ressources (figure 6).

Le système d'exploitation, et plus particulièrement sa partie la plus fondamentale, le *noyau*, est le premier programme chargé en mémoire lors du démarrage de l'ordinateur. À ce stade, le processeur exécute le code dans un mode privilégié, le *kernel node*. Dans ce mode, un programme a un accès direct à toutes les ressources. Le noyau gère les exécutions des programmes de l'utilisateur dans un autre mode, n'ayant pas d'accès direct aux ressources : le *user mode*. Toute application qui veut utiliser une ressource doit passer par le système d'exploitation, qui décide d'accorder ou pas l'accès. Cet accès indirect au matériel et aux ressources en général permet au système d'exploitation d'offrir à un programme utilisateur une vision différente de la réalité en substituant une ressource *réelle* par une ressource *virtuelle*.

Grâce à la virtualisation, le système d'exploitation est capable d'exécuter plusieurs programmes, concurremment ou en parallèle, en donnant l'impression à chacun qu'il a accès à l'intégralité de la mémoire, par exemple 4 GiO sur une machine 32 bits, même si la mémoire réellement installée ne dépasse pas 2 GiO : en utilisant le fait qu'un programme ne va jamais utiliser toute la mémoire disponible au même moment, le système d'exploitation peut charger à la demande en mémoire réelle les portions nécessaires à un instant donné ; le reste du temps, les informations sont stockées sur un disque dur, par exemple (figure 7).

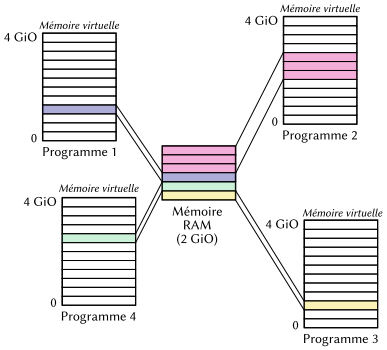


FIGURE 7 : Virtualisation de la mémoire pour supporter le multi-tâches.

Une application chargée en mémoire pour être exécutée est un processus. Un programme sous forme d'un seul fichier exécutable sur le disque dur peut donner lieu, une fois lancé, à plusieurs processus en mémoire. On verra dans la section 6 comment interagir avec les processus grâce au *shell*.

2 L'interpréteur de commandes Bash

2.1 Avant-propos

Le premier langage de *shell* pour UNIX fut écrit par Kenneth THOMPSON en 1971 en réutilisant l'expérience acquise par Glenda SCHROEDER sous Multics. C'était essentiel-

FIGURE 8 : Kenneth THOMPSON (1943–). Crédit photo : Dr. Dobb's.

lement un interpréteur de commandes qui permettait d'interagir avec le système d'exploitation mais pas d'écrire des programmes car il manquait des structures de contrôle pour en faire un vrai langage de programmation.

John Mashey, des *Bell Labs.*, développa un successeur au *Thompson Shell* qui fut rendu public en 1975 : le *Programmer's WorkBench* (PWB) ou *Mashey Shell*. Ce *shell* avait des capacités très étendues par rapport à son prédécesseur et autorisait l'écriture de programmes avec des tests et des répétitives, quoiqu'avec une interface peu intuitive du fait d'un choix de compatibilité de la syntaxe avec le *Thompson Shell*[6].

Afin de s'affranchir des limitations imposées par l'héritage du *Thompson Shell*, Stephen Bourne réécrivit complètement en 1979 un *shell* ayant des fonctionnalités proches du *Mashey Shell* mais avec une syntaxe plus claire et des fonctionnalités de programmation étendues : le *Bourne Shell* (sh). Ce *shell* devint très populaire parmi les programmeurs UNIX et s'imposa comme un standard *de facto*.

En 1989, Brian Fox (figure 11) implémenta une version libre du *Bourne Shell* : *Bourne Again Shell*, ou bash. Ce programme devint le *shell* par défaut dans la plupart des distributions de Linux, le « clone » d'UNIX. Initialement, bash se contentait de copier les fonctionnalités de sh. Au fur et à mesure de son évolution, bash s'est vu rajouter de nouvelles fonctionnalités non présentes dans sh ; par ailleurs, son comportement est différent de sh dans certaines situations. Aussi, lors de l'écriture de code *shell*, il sera important de bien préciser la compatibilité choisie grâce au *shebang* (section 3.1).

L'exécution d'un programme écrit dans un langage de script se fait toujours grâce à un interpréteur : contrairement à des langages comme le C, qui utilise un compilateur pour transformer le texte d'un programme C en du code machine directement exécutable par le processeur, le texte d'un programme en langage de script doit être lu et interprété par un programme spécifique (l'interpréteur) à chaque exécution, ce qui réduit considérablement ses performances.

Il était déjà bien présent dans l'esprit de Mashey[7] lors de la création du *Mashey Shell*, qu'un langage de script devait être minimal et se reposer sur des programmes externes pour toutes les tâches délégables. Une telle organisation rend aisément extensible à l'infini les possibilités de programmation *shell*. Nous présentons dans la suite de cette section les services offerts par bash ainsi que les programmes standards habituellement disponibles sous Linux/MacOS. Le texte est organisé par grandes fonctionnalités du système d'exploitation. La présentation est volontairement succincte et centrée sur ce qui sera utilisée dans le cadre du module X1BI030 « *Langages de scripts* ». Le manuel de référence de BASH est très complet et lisible. On s'y reportera pour connaître les détails des points seulement esquissés dans ce fascicule.

2.2 Le terminal

La programmation avec Bash peut se faire de deux manières :
- Interactivement, en entrant des commandes dans la fenêtre d'un programme appelé un terminal (figure 12) ;
- Programmatiquement, en exécutant un fichier contenant un script composé d'une succession de commandes Bash. On verra cette deuxième forme dans la section 3.1.

Le terminal présente un *prompt* (ou « invite »), qui est un message signalant à l'utilisateur que le système attend une commande[8]. L'utilisateur peut écrire une commande et appuyer sur la touche ↩ pour la faire exécuter. Toutes les commandes précédemment exécutées sont stockées dans un historique. Il est possible de faire défiler les commandes précédentes en utilisant les touches ↑ et ↓. La commande history affiche l'ensemble des commandes avec un identifiant numérique associé. On peut exécuter de nouveau une commande en spécifiant son numéro précédé d'un point d'exclamation :

```
X1BI030@pc1> history
 [...]
```

[6] J. R. Mashey. "Using a command language as a high-level programming language". In : *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. Washington, DC, USA : IEEE Computer Society Press, oct. 1976, p. 169-176

Figure 9 : John Mashey (1946–). Crédit photo : Medium.

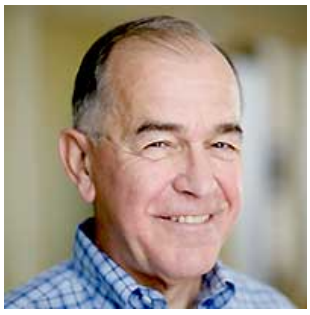


Figure 10 : Stephen Bourne (1944–). Crédit photo : Computerhope.

[7] J. R. Mashey. "Using a command language as a high-level programming language". In : *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. Washington, DC, USA : IEEE Computer Society Press, oct. 1976, p. 169-176

[8] Dans ce fascicule, nous utiliserons l'invite « `X1BI030@pc1>` ». Dans les salles de travaux pratiques, l'invite contient le nom de connection de l'utilisateur, le nom de l'ordinateur utilisé et un rappel du répertoire courant. L'invite peut être personnalisée grâce à quatre variables d'environnement : `PS1`, `PS2`, `PS3` et `PS4`. On se reportera à la documentation officielle de Bash pour les détails.

```
 503  rm bli.mp3
 504  cd rep
 505  ls
 506  history
X1BI030@pc1> !505
bla.txt  toto.mp3
```

La section 9 du manuel de Bash présente plus en détails les possibilités de l'historique de la ligne de commandes.

La suite de ce fascicule présente les commandes essentielles de Bash, ainsi que certains programmes extérieurs, avec plus ou moins de détails. Pour plus d'informations, on se reportera au manuel de chaque commande, accessible avec les programmes man ou help :

```
X1BI030@pc1> man ls
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 .

[...]
```

Il est possible d'effacer l'écran du terminal avec la commande clear.

2.3 Les commandes

L'interpréteur de commandes Bash boucle sur une attente de commande ; il exécute la commande puis se remet en attente jusqu'à ce que l'utilisateur fasse une nouvelle requête. L'exécution est synchrone : aucune autre commande ne peut être écrite tant que la précédente n'a pas fini de s'exécuter.

Il est possible d'exécuter une commande de manière asynchrone en ajoutant une esperluète (« & ») à la fin de la commande :

```
X1BI030@pc1> cp grosfichier.txt ../ailleurs &
[1] 5107
```

Le système rend alors la main sans attendre la fin de la commande, qui continue à s'exécuter en arrière-plan. Le numéro affiché (ici, 5107) est l'identifiant du processus (*Process IDentification*, ou PID) qui peut être utilisé pour manipuler la tâche lancée de manière asynchrone (voir section 6).

Une commande simple[9] s'écrit toujours de la même manière : le premier mot après l'invite est le chemin d'accès à la commande (voir section 2.4.4), ou simplement son nom si la commande se trouve dans la liste des chemins connus du système (voir la variable d'environnement PATH, section 2.6.2) ; chaque paramètre de la commande est ensuite écrit sur la ligne de commandes en le séparant des autres par un ou plusieurs espaces :

```
X1BI030@pc1> ls
1.txt  2.txt  3.txt  tmp
X1BI030@pc1> ls 3.txt
3.txt
```

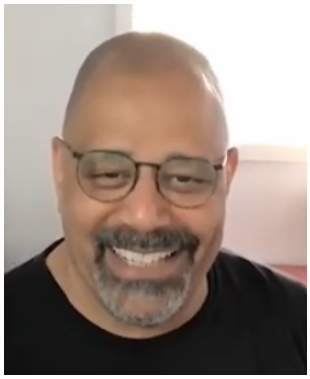


FIGURE 11 : Brian Fox (1959–). Crédit photo : FLOSS Weekly.

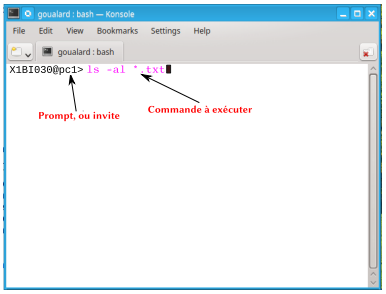


FIGURE 12 : Exemple de terminal BASH

[9] On verra plus loin les commandes composées utilisant, en particulier, les *pipes*.

Une option permet de modifier le comportement d'une commande. Une option peut être courte ou longue. Une option courte est composée d'un tiret, d'une lettre et optionnellement d'un argument — précédé ou non d'espaces ; une option longue est composée de deux tirets, d'un mot et optionnellement d'un signe d'égalité avec un argument :

```
X1BI030@pc1> ls -a
.  ..  1.txt  2.txt  3.txt  tmp
X1BI030@pc1> ls --all
.  ..  1.txt  2.txt  3.txt  tmp
X1BI030@pc1> ls --format=commas
1.txt, 2.txt, 3.txt, tmp
```

Il est possible de combiner des options courtes ensemble et de n'utiliser qu'un seul tiret. La commande « `ls -al` » est alors équivalente à « `ls -a -l` », par exemple.

Une commande peut être une commande interne à l'interpréteur ou un programme externe, auquel cas il existe quelque part un fichier contenant son code. Un programme externe peut avoir été compilé en code machine directement compréhensible par le programme (à partir d'un code C, par exemple) ou être un fichier texte interprétable par un interpréteur (un fichier de code Python ou Bash, par exemple). Le type de la commande exécutée n'influe pas sur la manière de l'appeler ni sur son comportement ; il est donc transparent du point de vue de l'utilisateur. Cependant, il est possible de différencier les différentes commandes avec la commande type si nécessaire :

```
X1BI030@pc1> type ls
ls is aliased to `/bin/ls -N --color'
X1BI030@pc1> type cd
cd is a shell builtin
X1BI030@pc1> type man
man is /usr/bin/man
```

On peut écrire plusieurs commandes sur la même ligne en les séparant par des points-virgules. Chaque commande est exécutée séquentiellement à son tour.

```
X1BI030@pc1> ls; cp toto.txt tmp/
```

2.3.1 Comment Bash trouve les commandes

Lorsqu'une commande est écrite avec un chemin d'accès, Bash recherche le fichier exécutable à l'endroit indiqué :

```
X1BI030@pc1> /usr/local/bin/python -v
Python 3.7.10
X1BI030@pc1> ./monscript.sh
Bonjour !
```

Si la commande est donnée sans aucun chemin d'accès, Bash essaye de la trouver en respectant les étapes suivantes :

1. S'il existe une fonction *shell* (voir section 5) de ce nom, elle est appelée ;
2. S'il existe une fonction interne (exemple : cd) de ce nom, elle est appelée ;
3. Sinon, le *shell* cherche un exécutable de ce nom dans chacun des répertoires listés dans la variable d'environnement PATH (voir section 2.6.2).

2.4 Gestion des fichiers

Tous les systèmes d'exploitation usuels ont adopté une représentation de l'information sous forme de fichiers : une entité d'information (image, texte, ...) possède un nom qui est mis en correspondance par le système de gestion de fichiers avec une localisation sur un périphérique de stockage. Afin de faciliter l'organisation des données, on a aussi la notion de répertoire (ou *directory*, en anglais). On peut faire l'analogie entre un répertoire et une boîte qui contiendrait d'autres fichiers ou répertoires. Cette organisation

hiérarchique donne naturellement lieu à une représentation arborescente des données d'un support de stockage, avec le répertoire racine contenant tous les répertoires et fichiers du support (voir section 2.4.4). Pour des raisons de clarté, on fera parfois dans la suite la différence entre un répertoire et un fichier, même si un répertoire peut être considéré comme un fichier particulier.

2.4.1 Les différents systèmes de gestion de fichiers

Il existe différents systèmes de gestion de fichiers, qui se différencient, entre autres, par les limites qu'ils posent sur le nombre et la taille des fichiers, répertoires et supports de stockage supportés ainsi que sur les droits associés aux fichiers et sur l'ensemble de caractères autorisés pour nommer un fichier ou un répertoire. Le système de gestion de fichiers est parfois intégré au noyau du système d'exploitation (Microsoft Windows, qui impose le système NTFS, par exemple), ou vient comme un module externe qui peut alors être changé suivant les désirs de l'utilisateur (le *Virtual File System* (VFS) de Linux — figure 13 —, par exemple).

La table 1 donne quelques valeurs maximales pour différents systèmes de gestion de fichiers. On voit, par exemple, que le système FAT32 — souvent utilisé pour formater une clé USB ou la carte mémoire d'un appareil photographique numérique — ne permet pas de stocker des fichiers de plus de 4 GiO. De même, le système NTFS, utilisé par Microsoft Windows, ne peut pas gérer des partitions de plus de 256 TiO. Ainsi, un disque dur de 300 TiO devra être découpé en deux partitions pour pouvoir être formaté avec NTFS.

Les multiples de l'octet.

Historiquement, les informaticiens utilisaient les préfixes « kilo », « mega », ... dans un contexte de puissances de 2. Ainsi, un kilo-octet contenait 1024 octets, un mega-octet, $1048576 = 2^{20}$ octets, etc. Cette convention était en violation de la définition standard de ces préfixes utilisant des puissances de 10. Aussi, en 1998, l'*International Electrotechnical Commission* décida de définir de nouveaux préfixes basés sur des puissances de 2 et de réserver les autres préfixes aux puissances de 10, comme il était habituellement l'usage. On a ainsi :

Nom	Notation	Capacité
Kilo-octet	kO	1000 octets
Kibi-octet	KiO	1024 octets
Méga-octet	MO	10^6 octets
Mébi-octet	MiO	2^{20} octets
Gigga-octet	GO	10^9 octets
Gibi-octet	GiO	2^{30} octets
Téra-octet	TO	10^{12} octets
Tébi-octet	TiO	2^{40} octets
Péta-octet	PO	10^{15} octets
Pébi-octet	PiO	2^{50} octets
Exa-octet	EO	10^{18} octets
Exbi-octet	EiO	2^{60} octets

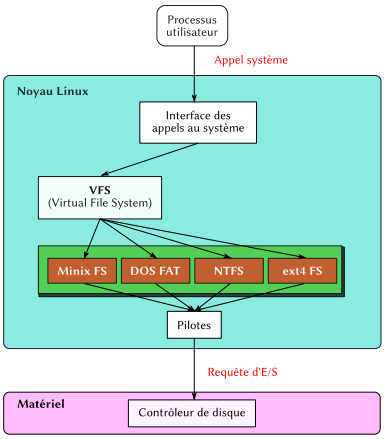


FIGURE 13 : Le *Virtual File System* sous Linux.

TABLE 1 : Caractéristiques des systèmes de gestion de fichiers.

Nom	Taille max. d'un nom	Taille max. d'un fichier	Taille max. d'une partition
Minix FS	30 car.	64 MiO	64 MiO
FAT 16	8+3 car.	2 GiO	2 GiO
FAT 32	255 car.	4 GiO	8 TiO
exFAT	255 car.	128 PiO	128 PiO
NTFS	255 car.	16 TiO	256 TiO
Ext2/Ext3 FS	255 car.	2 TiO	32 TiO
Ext4 FS	255 car.	16 TiO	1 EiO
JFS	255 car.	4 PiO	32 PiO

2.4.2 La gestion de fichiers sous UNIX/Linux

Comme l'indique la citation de RITCHIE au début de ce fascicule, la simplicité était au cœur du design d'UNIX, en partie pour se démarquer de la complexité de son ancêtre Multics. Le modèle adopté par UNIX pour garantir l'uniformité et la simplicité fut celui du fichier : sous UNIX, *tout est fichier* et se manipule en ouvrant, lisant, écrivant et

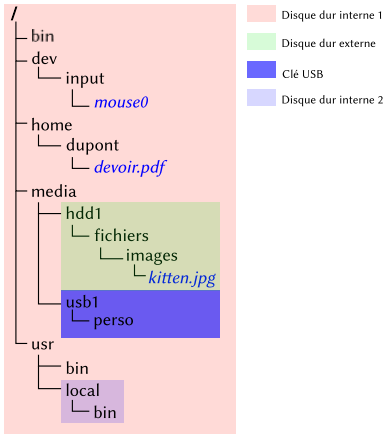


FIGURE 14 : Arborescence du système de fichiers avec plusieurs supports.

fermant le fichier. Tous les fichiers sont rassemblés dans une arborescence unique. On a ainsi, entre autres :

- Des fichiers traditionnels (un fichier PDF, par exemple) ;
- Des fichiers répertoires : un répertoire est un fichier contenant d'autres fichiers ;
- Des fichiers de périphérique : la souris, le clavier, l'écran, ... ont un fichier associé dans l'arborescence (exemple : `/dev/input/mouse0` pour une souris).

L'arborescence de tous les périphériques de stockage présents sur une machine, qu'ils soient internes ou externes, est aussi rattachée à l'arborescence globale (figure 14), à condition qu'ils aient été montés au préalable.

2.4.3 Les attributs des fichiers

Chaque fichier et répertoire possède un ensemble d'attributs dépendant du système de gestion de fichiers utilisés. On considèrera ici essentiellement deux types d'attributs généralement disponibles sous Linux :

- Les étiquettes de dates ;
- Les droits d'accès.

Les étiquettes de dates sont au nombre de trois :

- atime (*access time*) : la date et l'heure de dernier accès à un fichier (par exemple une ouverture en lecture) ;
- mtime (*modify time*) : la date et l'heure de dernière modification d'un fichier ;
- ctime (*change time*) : la date et l'heure de dernière modification des attributs d'un fichier.

Les étiquettes de dates associées à un fichier peuvent être visualisées avec la commande stat et modifiées avec la commande touch. Lorsque le paramètre donné à cette dernière commande ne correspond pas à un fichier existant, elle le crée. On utilisera ce comportement lorsque l'on aura besoin de créer un fichier vide.

Le montage.

Le montage d'un périphérique de stockage (disque dur, clé USB, ...) correspond à l'opération de greffage de l'arborescence de répertoires de ce périphérique à un point donné de l'arborescence globale du système.
Pour monter un périphérique, il faut donner au moins trois informations :

- Le type de gestionnaire de fichiers utilisé par le périphérique (NTFS, FAT32, ...) ;
- Le chemin d'accès au périphérique (exemple : `/dev/sda1`) ;
- Le chemin d'accès où monter l'arborescence du périphérique (exemple : `/mnt/hdd`).

Le montage d'un disque se fait avec la command mount.

```
X1BI030@pc1> stat /bin/bash
  File: /bin/bash
  Size: 1113504         Blocks: 2176       IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 13369351    Links: 1
Access: (0755/-rwxr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2021-09-27 07:39:32.616806179 +0200
Modify: 2019-06-07 00:28:15.000000000 +0200
Change: 2019-09-02 14:49:51.186419773 +0200
 Birth: -
# Création d'un fichier vide toto.txt
X1BI030@pc1> ls toto.txt
/bin/ls: cannot access 'toto.txt': No such file or directory
X1BI030@pc1> touch toto.txt
X1BI030@pc1> ls toto.txt
toto.txt
```

Les systèmes de gestion de fichiers communément utilisés sous Linux associent trois types de droits aux fichiers et répertoires :

- Le droit de lecture (*read*), autorisant la lecture du contenu d'un fichier. Si le fichier est un répertoire, ce droit autorise à lister les fichiers qu'il contient ;
- Le droit d'écriture (*write*), autorisant la modification — y compris la destruction — d'un fichier. Si le fichier est un répertoire, ce droit autorise l'ajout ou l'effacement des fichiers ou répertoires qu'il contient ;

— Le droit d'exécution (*execute*), autorisant à utiliser le fichier comme une commande. Si le fichier est un répertoire, cela autorise à entrer dans le répertoire (par exemple, avec la commande cd).

Un fichier possède systématiquement un propriétaire et un groupe. Il est possible de modifier ces attributs en utilisant la commande chown :

```
X1BI030@pc1> ls -l fichier.txt
-rw-rw-r-- 1 goualard goualard 0 sept. 21 10:17 fichier.txt
X1BI030@pc1> chown dupont:durand fichier.txt
-rw-rw-r-- 1 dupont durand 0 sept. 21 10:17 fichier.txt
```

Les trois droits ci-dessus sont propres à trois types d'utilisateurs :
— Le propriétaire du fichier (*user*) ;
— Le groupe du fichier (*group*) ;
— Le reste du monde (*others*).

La notion de groupe permet d'associer des droits sur un fichier à un ensemble de personnes plutôt qu'à une seule personne identifiée. Il suffit alors d'ajouter tous les utilisateurs requis à un groupe pour qu'ils bénéficient des droits qui y sont associés. L'ajout d'un utilisateur à un groupe peut se faire avec la commande usermod. La commande groups donne la liste des groupes auquel appartient un utilisateur.

L'ensemble des droits associés à un fichier constitue ce que l'on appelle la *matrice de droits*, visible lorsque l'on utilise l'option « -l » de la commande ls (figure 15).

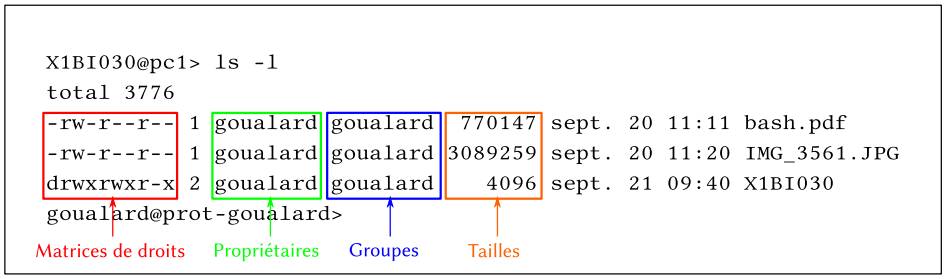
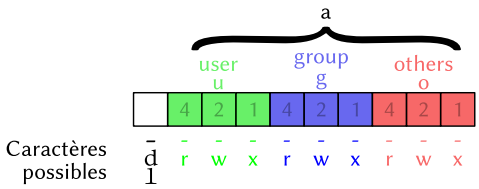


FIGURE 15 : Utilisation de la commande ls avec l'option « -l ».

Une matrice de droits comporte dix caractères :



— Le caractère le plus à gauche peut être un tiret « - » si le fichier est un fichier ordinaire (« *regular file* »), un « d » si le fichier est un répertoire (« *directory* »), ou un « l » si le fichier est un lien vers un autre fichier[10] ;
— Les trois blocs de trois caractères suivants correspondent aux droits du propriétaire, du groupe et du reste du monde. Si le caractère est un tiret « - », le droit correspondant est absent.

D'autres caractères peuvent apparaitre dans la matrice de droits mais nous ne les considérerons pas dans ce fascicule.

La modification des droits d'un fichier ou d'un répertoire se fait avec la commande chmod. On peut préciser les droits à ajouter ou retirer à un fichier :

```
X1BI030@pc1> ls -l
-rw-rw-r-- 1 goualard goualard      0 sept. 21 10:17 fichier.txt
X1BI030@pc1> chmod g+x fichier.txt
-rw-rwxr-- 1 goualard goualard      0 sept. 21 10:17 fichier.txt
```

[10] Un lien a l'apparence d'un fichier mais ne prend pas de place sur le disque dur car il pointe vers un autre fichier physiquement présent. Les liens permettent d'avoir un même fichier dans différents répertoires, éventuellement sous des noms différents, sans dupliquer l'information. On se reportera à la documentation de la commande ln pour la création de liens.

On précise d'abord l'utilisateur concerné (« u », « g », « o », ou « a » pour tous) ; on utilise ensuite le symbole « + » pour ajouter le droit et « - » pour le retirer, puis on indique le ou les droits concernés (r, w, ou x).

Il est aussi possible de modifier les droits numériquement pour tous les utilisateurs à la fois. Pour cela, on ajoute les nombres correspondant aux droits à allouer dans les cases de la figure ci-dessus. Pour obtenir la matrice de droits « -rwxr-xr-x, on écrirait :

```
X1BI030@pc1> chmod 755 fichier.txt
```

2.4.4 L'arborescence des fichiers

Comme indiqué dans la section 2.4, l'ensemble des fichiers d'un système sous Linux est stocké dans une unique arborescence. À l'ouverture d'un terminal, l'utilisateur se trouve dans le répertoire racine de la sous-arborescence contenant ses fichiers. C'est le home directory. Habituellement, le *home directory* d'un utilisateur dupont se trouve dans le répertoire `/home/dupont`. À la Faculté des Sciences et des Techniques de Nantes, il sera dans `/comptes/dupont` car tous les répertoires des utilisateurs sont localisés sur un serveur distant et non pas sur la machine sur laquelle on se connecte dans la salle de travaux pratiques.

Tout fichier ou répertoire peut être manipulé sans ambiguïté en précisant son chemin d'accès dans l'arborescence. Le chemin absolu précise la position du fichier ou du répertoire à partir de la racine, dont le nom est « / ». On sépare chaque composante du chemin d'accès par un « / ». Dans la figure 16, le chemin absolu du répertoire `X1BI030` est `/home/dupont/X1BI030`. Un chemin absolu commence toujours par le symbole « / ».

On peut aussi utiliser un chemin relatif au répertoire dans lequel se trouve actuellement l'utilisateur dans le terminal (le répertoire courant). Ce chemin peut être obtenu avec la commande pwd :

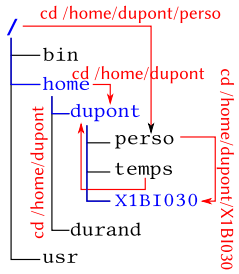


FIGURE 16 : Déplacement avec des chemins absolus.

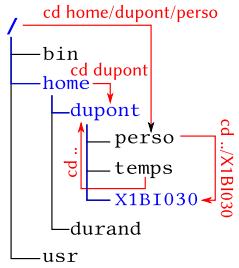


FIGURE 17 : Déplacement avec des chemins relatifs.

```
X1BI030@pc1> pwd
/home/dupont
```

Le chemin relatif pour le répertoire `X1BI030` à partir du répertoire `/home` est `dupont/X1BI030`.

Tous les répertoires possèdent automatiquement deux sous-répertoires :

— « . », qui est le répertoire lui-même ;

— « .. », qui est le chemin relatif du répertoire immédiatement au-dessus.

En utilisant « .. », on peut construire un chemin relatif pour un fichier ou un répertoire ne se trouvant pas dans la même branche que le répertoire courant. Par exemple, si le répertoire courant est `/home/dupont/X1BI30`, le chemin relatif du répertoire `durand` est `../../durand`.

Les figure 16 et 17 donnent des exemples de déplacement dans l'arborescence à l'aide de la commande cd (« *Change Directory* »). Cette commande permet de changer le répertoire courant :

```
X1BI030@pc1> pwd
/home/dupont/X1BI030
X1BI030@pc1> cd ../../durand
X1BI030@pc1> pwd
/home/durand
```

2.4.5 Manipulation des fichiers et répertoires

Les commandes suivantes permettent de manipuler des fichiers ou des répertoires. On se reportera à la documentation disponible avec man pour connaitre leur usage et les options disponibles :

— cp : copie d'un fichier ou d'une liste de fichiers dans un répertoire ; si le dernier argument n'est pas un répertoire, on copie en renommant le premier fichier. Exemple :

```
# On fait une copie de 1.txt nommée 3.txt
X1BI030@pc1> cp 1.txt 3.txt
# On copie les fichiers 1.txt et 2.txt dans le répertoire tmp
X1BI030@pc1> cp 1.txt 2.txt tmp/
```

— ls : Liste les fichiers et répertoires d'un répertoire courant. On peut préciser les informations à afficher avec certaines options (par exemple, « -l »pour avoir les droits, la taille, ... et « -a » pour lister les fichiers et répertoires commençant par un point, qui ne sont normalement pas inclus dans l'affichage). Exemples :

```
X1BI030@pc1> ls
1.txt  2.txt  3.txt  tmp
X1BI030@pc1> ls -a
.  ..  1.txt  2.txt  3.txt  tmp
X1BI030@pc1> ls -l
total 16
-rw-rw-r-- 1 goualard goualard    2 sept. 21 14:49 1.txt
-rw-rw-r-- 1 goualard goualard    2 sept. 21 14:49 2.txt
-rw-rw-r-- 1 goualard goualard    2 sept. 21 14:51 3.txt
drwxrwxr-x 3 goualard goualard 4096 sept. 21 10:17 tmp
```

— mkdir : création de répertoires. Exemples :

```
X1BI030@pc1> mkdir rep
# Création du répertoire d et des répertoires parents s'ils n'existent pas déjà
X1BI030@pc1> mkdir -p a/b/c/d
```

— mv : déplacement de fichiers dans un répertoire ou renommage d'un fichier. Exemples :

```
# On renomme 1.txt en 2.txt
X1BI030@pc1> mv 1.txt 2.txt
# On déplace 1.txt et 2.txt dans le répertoire tmp/
X1BI030@pc1> mv 1.txt 2.txt tmp/
```

— rm : effacement de fichiers ou de répertoires (avec l'option « -r »). Exemples :

```
X1BI030@pc1> rm 1.txt
X1BI030@pc1> rm -r tmp/
```

— rmdir : effacement de répertoires. En pratique, cette commande est rarement utilisée car elle ne fonctionne que pour des répertoires *vides*.

```
X1BI030@pc1> mkdir toto titi
X1BI030@pc1> touch titi/bla.txt
X1BI030@pc1> rmdir toto
X1BI030@pc1> rmdir titi
rmdir: failed to remove 'titi': Directory not empty
```

2.5 Flux et redirections

Tout programme, qu'il soit écrit en Bash, Python, ou le fruit de la compilation d'un code C a la même interface avec l'interpréteur de commandes (figure 18) : il peut recevoir de l'information par les paramètres de la ligne de commandes ou via l'entrée standard (« *stdin* »). En retour, il peut retourner de l'information au système par trois canaux :

— La sortie standard ;
— La sortie d'erreur standard ;
— Le code d'erreur.

L'entrée standard est normalement connectée au clavier et permet donc de lire tout ce qui sera tapé dessus.

Les sorties standard et d'erreur standard sont normalement connectées à l'écran du terminal. Tout ce qui y est envoyé est donc, par défaut, affiché à l'écran et il n'est alors pas possible de différencier ce qui a été envoyé sur un canal ou sur l'autre.

Le code d'erreur est un entier qui vaut 0 si l'exécution s'est bien passée et une valeur non nulle sinon. En cas d'erreur, la valeur retournée dépend du programme. On peut

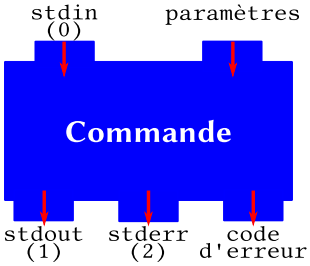


FIGURE 18 : Entrées/Sorties d'une commande.

récupérer cette valeur avec la variable $?. Cette variable contient le code d'erreur de la *dernière commande exécutée.*

```
X1BI030@pc1> rm tutu
rm: cannot remove 'tutu': No such file or directory
X1BI030@pc1> echo $?
1
X1BI030@pc1> rm 2.txt
X1BI030@pc1> echo $?
0
```

La commande echo envoie tous ses paramètres sur la sortie standard.

Il est possible de rediriger le canal d'entrée pour lire dans un fichier. Cela se fait avec la construction :

```
commande < fichier
```

Par exemple, la commande tr permet, entre autres, de remplacer des caractères par d'autres. Contrairement à la plupart des commandes, elle ne lit que l'entrée standard et n'accepte pas de recevoir en paramètre le nom d'un fichier à lire. Une première utilisation correspond à l'entrée directe au clavier du texte à traiter :

```
X1BI030@pc1> tr a A
a
A
tata
tAtA
^C
```

La combinaison de touches Ctrl + c permet de récupérer la main. En utilisant la redirection de l'entrée standard, on peut traiter le contenu d'un fichier :

```
X1BI030@pc1> tr a A < fic.txt
Le chAt mAnge des rAts.
```

La commande cat permet d'afficher le contenu du fichier original :

```
X1BI030@pc1> cat fic.txt
Le chat mange des rats.
```

De la même manière, il est possible de rediriger la sortie standard et la sortie d'erreur standard vers un fichier avec la construction :

```
# Redirection de la sortie standard
commande > fichier.txt
# Redirection de la sortie d'erreur standard
commande 2> fichier.txt
```

Exemples d'utilisation :

```
X1BI030@pc1> ls > resultat.txt
X1BI030@pc1> ls toto.txt > resultat2.txt
/bin/ls: cannot access 'toto.txt': No such file or directory
X1BI030@pc1> ls toto.txt 2> erreurs.log
X1BI030@pc1> cat erreurs.log
/bin/ls: cannot access 'toto.txt': No such file or directory
```

Avec l'opérateur « > », le fichier recevant le résultat de la commande est écrasé s'il existe déjà et créé s'il n'existe pas. On peut utiliser l'opérateur « >> » pour ajouter du contenu dans un fichier :

```
X1BI030@pc1> echo "On marche une certaine nuit de l'année dans la neige," >citadelle.txt
X1BI030@pc1> echo "laquelle est craquante," >>citadelle.txt
X1BI030@pc1> echo "sous les étoiles," >> citadelle.txt
X1BI030@pc1> echo "vers des maisons de bois illuminées." >>citadelle.txt
X1BI030@pc1> cat citadelle.txt
On marche une certaine nuit de l'année dans la neige,
laquelle est craquante,
sous les étoiles,
vers des maisons de bois illuminées.
```

Redirection et trou noir.

On veut parfois se débarrasser du texte provenant de l'exécution d'une commande, par exemple des messages d'avertissements inopportuns. Pour cela, on peut rediriger le flux correspondant vers le fichier « `/dev/null` ». Tout ce qui est écrit dans ce fichier est simplement éliminé :

```
X1BI030@pc1> ls 2.txt
/bin/ls: cannot access '2.txt': No such file or directory
X1BI030@pc1> ls 2.txt 2>/dev/null
X1BI030@pc1> cat /dev/null
```

Il est aussi possible de rediriger un flux vers un autre en utilisant la construction « >& » et en spécifiant à gauche de l'opérateur le numéro de canal redirigé et à droite le numéro de canal destinataire. Ainsi, pour envoyer du texte vers la sortie d'erreur standard, on peut utiliser la commande echo en redirigeant la sortie standard vers la sortie d'erreur standard :

```
echo "Un message d'erreur" 1>&2
```

Enfin, on peut rediriger la sortie standard d'une commande vers l'entrée standard d'une autre avec la construction :

```
commande | commande
```

Cette construction utilise le caractère « | », appelé « *pipe* » (/paɪp/), que l'on obtient en pressant AltGr + 6 sur un clavier de PC, ou Alt + Shift ⇑ + L sur un clavier d'Apple MacBook.

Il est possible de chaîner plus de deux commandes et de réaliser ainsi des tâches complètes en transformant par des commandes successives un texte d'entrée jusqu'à obtenir le résultat souhaité.

Exemple. Les commandes head et tail retournent, respectivement, sur la sortie standard, les k premières lignes et les k dernières lignes du texte reçu en entrée, la valeur de k étant passée en paramètre de l'option « `-n` ». Ainsi, pour obtenir les trois premières lignes du fichier `citadelle.txt`, on peut écrire :

```
X1BI030@pc1> head -n3 citadelle.txt
On marche une certaine nuit de l'année dans la neige,
laquelle est craquante,
sous les étoiles,
```

De la même manière, on peut récupérer les trois dernières lignes :

```
oualard@prot-goualard> tail -n3 citadelle.txt
laquelle est craquante,
sous les étoiles,
vers des maisons de bois illuminées.
```

Pour obtenir seulement la troisième ligne du fichier `citadelle.txt`, on peut d'abord récupérer les trois premières lignes puis garder la dernière ligne de ce résultat (figure 19) :

```
X1BI030@pc1> head -n3 citadelle.txt | tail -n1
sous les étoiles,
```

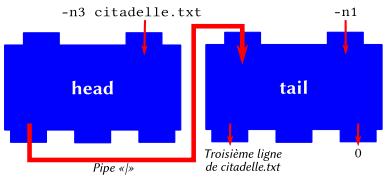


FIGURE 19 : Affichage de la 3e ligne du fichier `citadelle.txt`.

Notez que l'appel de tail ne contient plus de référence à `citadelle.txt` car la commande prend en entrée le résultat de la commande head et non plus le contenu du fichier original.

Le code d'erreur retourné systématiquement par chaque commande peut être utilisé pour chaîner conditionnellement des commandes avec les opérateurs `&&` et `||`. La construction :

```
commande1 && commande2
```

exécute la commande 2 seulement si la commande 1 retourne un code d'erreur nul. À l'inverse, la construction :

```
commande1 || commande2
```

exécute la commande 2 seulement si la commande 1 retourne un code d'erreur non nul. Quelques exemples d'utilisation :

```
# Déplacement d'un fichier dans le nouveau répertoire `tmp` seulement si
# sa création s'est bien passée.
mkdir tmp && mv precieux.txt tmp

# Copie de `toto.txt` en `tutu.txt` et si cela ne marche pas, on
# essaye avec `titi.txt`.
cp toto.txt tutu.txt || cp titi.txt tutu.txt
```

2.6 Les expansions

L'exécution de chaque commande par l'interpréteur Bash se fait toujours en deux grandes étapes :

1. Une phase d'expansion d'éléments substituables ;
2. Une phase d'appel du programme avec les paramètres expansés.

Les éléments substituables sont essentiellement au nombre de cinq et sont remplacés dans cet ordre[11] :

— Expansion des accolades ;
— Substitution des variables ;
— Substitution de commandes ;
— Expansion d'expressions arithmétiques ;
— Expansion du globbing.

[11] Il y a en réalité sept types d'éléments expandables mais nous nous restreignons aux cinq essentiels dans ce fascicule. On se reportera à la section 3.5 du manuel de référence de Bash pour la liste complète.

2.6.1 L'expansion des accolades

Il est possible d'utiliser une paire d'accolades (« {} ») pour générer des chaînes de caractères avec un préfixe et/ou un suffixe commun :

```
mkdir tmp/{toto,titi,tutu}
```

Chaque chaîne de la liste entre accolades est utilisée pour générer une chaîne complète. La commande ci-dessus est donc équivalente à :

```
mkdir tmp/toto tmp/titi tmp/tutu
```

La substitution de « `tmp/{toto,titi,tutu}` » par « `tmp/toto tmp/titi tmp/tutu` » a lieu avant l'appel de la commande mkdir.

2.6.2 Les variables

Le nom d'une variable Bash commence par une lettre ou un blanc souligné (« _ ») et peut contenir ensuite une succession de lettres, de chiffres et de blancs soulignés. Bash fait la différence entre les minuscules et les majuscules : les variables `mavar` et `maVar` sont deux variables différentes. Bash supporte les tableaux unidimensionnels mais nous ne les considérerons pas ici. Reportez-vous à la documentation officielle si vous souhaitez savoir comment les exploiter.

L'affectation d'une valeur à une variable se fait avec le symbole d'égalité « = ». Attention : il ne faut aucun espace à gauche ou à droite du symbole d'égalité, sous peine d'avoir une erreur de syntaxe :

```
# Affectation de la valeur 5 à la variable `mavar`
mavar=5
# Erreurs de syntaxe !
mavar= 6
mavar = 6
mavar =6
```

Pour accéder au contenu d'une variable, il faut précéder son nom d'un dollar (« $ »). Bash substitue alors le nom préfixé par la valeur de la variable :

```
X1BI030@pc1> x=34; echo $x
34
```

Notez que la phase de substitution remplace « echo $x » par « echo 34 » *avant* d'appeler la commande « echo ».

Par défaut, les variables contiennent des chaînes de caractères ; la variable x ci-dessus contient donc la chaîne `"34"` et non l'entier 34. Il faut demander explicitement une interprétation numérique si nécessaire (voir section 2.6.4). Ainsi, on aura :

```
XIBI030@pc1> x=1+1; echo $x
1+1
```

Une variable peut être utilisée même si elle n'a jamais reçu de valeur. Elle est alors initialisée immédiatement avant usage avec la chaîne de caractères vide :

```
X1BI030@pc1> echo +$nouvelle_variable+
++
```

Pour lire une valeur au clavier, on peut utiliser la commande read, qui stocke dans une variable dont le nom est donné en paramètre le texte venant de l'entrée standard :

```
X1BI030@pc1> read -p "Donnez un entier entre 0 et 100? " valeur
Donnez un entier entre 0 et 100? 45
X1BI030@pc1> echo $valeur
45
```

L'option « -p » permet d'ajouter un message avant la lecture.

Le comportement de l'interpréteur Bash peut être modifié par un ensemble de variables prédéfinies appelées variables d'environnement. On peut afficher le contenu de ces variables avec la commande env. On se reportera au Bash reference manual pour une liste exhaustive des variables d'environnement disponibles.

L'une des variables d'environnement les plus utiles est la variable `PATH`, listant les répertoires dans lesquels rechercher une commande à exécuter :

```
X1BI030@pc1> echo $PATH
/usr/bin:/usr/local/bin:/home/dupont/bin
X1BI030@pc1> PATH=$PATH:/home/dupont/local/bin; echo $PATH
/usr/bin:/usr/local/bin:/home/dupont/bin:/home/dupont/local/bin
```

Chaque répertoire est écrit dans l'ordre de la recherche à respecter en le séparant des suivants par un deux-points (« : »).

2.6.3 L'expansion de commandes

On a vu dans la section 2.5 comment transférer la sortie d'une commande vers l'entrée standard d'une autre commande avec le *pipe*. On veut parfois utiliser le résultat d'une commande comme paramètre d'une autre commande. On peut pour cela utiliser l'expansion de commande avec la construction :

```
commande2 $(commande1)
```

La commande 1 est d'abord exécutée et l'on remplace toute l'expression `$(commande1)` par le contenu de la sortie standard ; La commande 2 est alors exécutée avec le ou les paramètres nouvellement générés :

```
# Recherche du chemin du fichier `toto.txt`
X1BI030@pc1> find . -name toto.txt
/home/dupont/X1BI030/tmp/toto.txt
# Effacement du fichier `toto.txt` en utilisant le chemin trouvé par `find`
X1BI030@pc1> rm $(find . -name toto.txt)
```

L'expansion de commandes peut aussi se faire en utilisant les *backquotes* (« ‘ ») mais il est recommandé d'utiliser plutôt la construction présentée ici qui est, elle, imbriquable, contrairement aux *backquotes* :

```
X1BI030@pc1> echo $(echo $(echo Longtemps je me suis couché de bonne heure))
Longtemps je me suis couché de bonne heure
```

2.6.4 L'expansion d'expressions arithmétiques

L'évaluation d'une expression arithmétique doit se faire explicitement, le défaut étant de considérer toute expression comme une chaîne de caractères. Pour cela, on utilise la construction :

`$((expression arithmetique))`

On aura ainsi :

```
X1BI030@pc1> echo $((40+21))
61
X1BI030@pc1> x=1; x=$(($x + 1)); echo $x
2
```

Exceptionnellement, on peut ne pas préfixer par un dollar les variables à l'intérieur des doubles parenthèses car il est clair à partir du contexte que l'on veut récupérer leur valeur. :

```
X1BI030@pc1> x=1; x=$((x + 1)); echo $x
2
```

Je n'encourage pas cette facilité d'écriture, au moins dans un premier temps, de façon à ne pas perdre l'habitude de préfixer les variables par un dollar dans les contextes où c'est indispensable.

2.6.5 Le *globbing*

Les commandes Bash manipulent essentiellement des fichiers et des répertoires. Il est souvent nécessaire de faire référence à plusieurs fichiers ; cela peut devenir pénible de spécifier explicitement tous les fichiers à traiter lorsqu'ils sont nombreux. Une solution apportée par le *globbing* est de spécifier une liste de fichiers en intention en décrivant ce que les noms des fichiers ont en commun. Par exemple, si l'on veut supprimer les fichiers `a1.txt`, `a2.txt`, ..., `a9.txt`, on pourrait écrire :

```
X1BI030@pc1> rm a1.txt a2.txt a3.txt a4.txt a5.txt a6.txt a7.txt a8.txt a9.txt
```

mais une façon plus simple tire parti du fait que tous les noms de fichiers à effacer commencent par un « a », sont suivis d'un chiffre entre « 1 » et « 9 » et se terminent par « .txt », ce qui s'écrit avec l'expression de *globbing* :

```
X1BI030@pc1> rm a[1-9].txt
```

L'expression « `a[1-9].txt` sera expansée par Bash en la suite de paramètres « `a1.txt a2.txt a3.txt a4.txt a5.txt a6.txt a7.txt a8.txt a9.txt` » avant l'appel de la commande rm.

Attention : contrairement à l'expansion des accolades vue dans la section 2.6.1, l'expansion d'une expression de *globbing* ne peut se faire que par rapport à des fichiers existants. Ainsi, si le répertoire courant contient les seuls fichiers `a2.txt` et `a5.txt`, on peut écrire :

```
X1BI030@pc1> ls a[1-9].txt
a2.txt a5.txt
```

Par contre, on ne peut pas utiliser l'expression de *globbing* pour créer des fichiers puisque l'expansion ne se fera pas sur des noms de fichiers inexistants :

```
X1BI030@pc1> touch b[1-9].txt; ls
b[1-9].txt
```

Ici, l'expression de *globbing* ne s'est pas expansée car aucun fichier dans le répertoire courant ne la *matche* ; la commande touch a donc été appelée avec l'expression non expansée et elle a créé un fichier de nom `b[1-9].txt`.

Une expression de *globbing* est construite à partir des éléments suivants et doit *matcher* l'intégralité du nom d'une fonction ou d'un répertoire :

- « **`*`** » : matche n'importe quelle chaîne, y compris la chaîne vide, sauf un point en première position :

  ```
  X1BI030@pc1> ls
  .toto a1.txt a2.txt a3.txt
  X1BI030@pc1> ls a1*
  a1.txt
  X1BI030@pc1> ls *
  a1.txt a2.txt a3.txt
  ```

- « **`?`** » : matche un seul caractère quelconque ;

  ```
  X1BI030@pc1> ls
  .toto a1.txt a2.txt a3.txt
  X1BI030@pc1> rm a?.* ; ls
  .toto
  ```

- « **`[...]`** » : matche l'un des caractères entre les crochets. Deux caractères séparés par un tiret (« - ») dénotent l'intervalle de tous les caractères entre ces deux bornes. Si le premier caractère est un point d'exclamation (« ! ») ou un accent circonflexe (« ^ »), l'expression matche tous les caractères non compris dans l'ensemble :

  ```
  X1BI030@pc1> ls
  a1.txt b.txt c.txt d.txt
  X1BI030@pc1> ls [a-c]*
  a1.txt b.txt c.txt
  X1BI030@pc1> ls [ad]*
  a1.txt d.txt
  X2BI030@pc1> ls [!ac]*
  b.txt d.txt
  ```

Dans une expression de *globbing*, tous les autres caractères matchent eux-mêmes :

```
# Copie vers le répertoire du dessus de tous les fichiers dont le nom ne commence pas par
# une voyelle, est suivi d'un `p`, puis de n'importe quels caractères et se termine par
# tout sauf un chiffre, puis un point et les caractères `pdf`
X1BI030@pc1> cp [!aeiouy]p*[!0-9].pdf ..
```

2.6.6 Le *quoting*

Certains caractères ont un sens spécial, par exemple l'étoile « * » pour le *globbing*. On a parfois besoin d'utiliser ces caractères pour eux-mêmes et non pour leur valeur particulière. Pour cela, il suffit de *quoter* le caractère en le précédant avec un *backslash* (« \ »), ce qui lui fait perdre son sens spécial :

```
# Effacement d'un fichier de nom `[ab]*`
X1BI030@pc1> rm \[ab\]\*
```

Certaines lettres précédées d'un backslash ont un sens spécial. Les plus utiles sont :
— `\n` : retour à la ligne ;
— `\t` : tabulation.
Exemple d'utilisation :

```
X1BI030@pc1> echo -e "Hello\n\tWorld\t!"
Hello
        World   !
```

On notera l'emploi de l'option « -e » de echo pour interpréter correctement ces combinaisons.

On a vu que les données manipulées par Bash étaient des chaînes de caractères par défaut. Il est cependant parfois utile de créer explicitement une chaîne de caractères, par exemple si elle contient des espaces. On peut alors utiliser les guillemets doubles (« " ») ou les apostrophes simples (« ' ») :

```
X1BI030@pc1> rm "Mes Documents/toto.txt"
```

Les guillemets doubles permettent ici de spécifier un nom de fichier avec espace. Sans eux, la commande rm tenterait d'effacer les fichier `Mes` et `Documents/toto.txt`. On obtiendrait le même effet avec des apostrophes simples :

```
X1BI030@pc1> rm 'Mes Documents/toto.txt'
```

Les guillemets et les apostrophes ne sont cependant pas entièrement interchangeables car si les apostrophes préservent toutes les valeurs litérales des caractères, il n'en est pas de même pour les guillemets. En particulier, les guillemets autorisent l'expansion des expressions précédées d'un dollar :

```
X1BI030@pc1> v=10; echo "$v" '$v'
10 $v
X1BI030@pc1> echo "Nous sommes ici: $(pwd)"
Nous sommes ici: /home/dupont/X1BI030
```

3 Programmation avec Bash

Jusqu'à présent, nous avons vu l'appel de programmes sur la ligne de commandes d'un terminal. Il est possible d'utiliser Bash comme un vrai langage de programmation et d'écrire un ensemble de commandes dans un fichier de texte pour exécuter des actions complexes. Il est cependant important de garder à l'esprit que Bash n'est pas fait pour remplacer un langage de programmation comme Python ou C ; sa vraie force réside dans son utilisation comme une *colle* pour faire interagir des programmes différents, ou pour manipuler directement les ressources gérées par le système d'exploitation (processus et fichiers, en particulier).

3.1 Les scripts

Un script est un fichier texte contenant un ensemble de commandes Bash. Pour pouvoir l'utiliser comme n'importe quel autre programme, il faut :

— Donner les droits d'exécution au fichier (voir section 2.4.3) ;
— Indiquer sur la première ligne du fichier le chemin d'accès au programme chargé d'interpréter le reste du fichier (l'interpréteur Bash).

Étant donné le fichier de script `script.sh` le premier point est satisfait avec la commande :

```
X1BI030@pc1> chmod +x script.sh
```

Pour le deuxième point, il faut ajouter en première ligne du fichier le shebang « `#!` » suivi du chemin d'accès à l'interpréteur Bash. On trouve parfois la ligne :

```
#!/bin/sh
```

Cette méthode est absolument à bannir car elle conduit à utiliser le *Bourne shell* à la place de Bash pour interpréter le script. Or, si Bash dérive initialement du *Bourne shell*, il s'en est depuis suffisamment éloigné pour qu'ils ne soient plus interchangeables.

Une deuxième ligne souvent utilisée est :

```
#!/bin/bash
```

L'inconvénient de cette méthode est que le chemin d'accès à l'interpréteur `bash` est défini « en dur » dans le fichier alors que certains systèmes stockent le fichier `bash` dans d'autres répertoires. C'est pourquoi je suggère d'utiliser plutôt la ligne :

```
#!/usr/bin/env bash
```

Le programme env va rechercher la commande bash dans la liste de chemins prédéfinis par le `PATH`, qui peut même être adaptée par l'utilisateur pour appeler sa propre version de Bash plutôt que celle présente par défaut sur le système.

Après la ligne du *shebang*, le fichier de scripts peut contenir une succession de commandes telles que celles que l'on pourrait écrire sur la ligne de commandes du terminal. On peut séparer chaque commande de la suivante par un point-virgule ou par un passage à la ligne indifféremment. Un script peut accéder aux paramètres qui lui ont été passés sur la ligne de commandes grâce aux variables `$0`, `$1`, `$2`, ... contenant les paramètres 0 (nom du programme appelé), 1, 2, ... Le nombre de paramètres est stocké dans la variable `$#`. Au-delà de 9, il faut entourer le nombre indiquant la position d'un paramètre entre accolades : `${10}`, par exemple.

Exemple : contenu du fichier `params.sh` affichant le nombre de paramètres ainsi que les paramètres 0 et 1 :

```
#!/usr/bin/env bash
echo "Nombre de paramètres passés: $#"
echo "Nom du programme appelé: $0"
echo "Valeur du premier paramètre: $1"
```

L'exécution du fichier `params.sh` se trouvant dans le répertoire courant donnera les résultats suivants :

```
X1BI030@pc1> ./params.sh bonjour
Nombre de paramètres passés: 1
Nom du programme appelé: params.sh
Valeur du premier paramètre: bonjour
X1BI030@pc1> ./params.sh
Nombre de paramètres passés: 0
Nom du programme appelé: params.sh
Valeur du premier paramètre:
```

Notez que, comme d'habitude, ce n'est pas une erreur de référencer une variable qui n'a pas de valeur. Dans le deuxième appel, `$1` n'a pas de valeur car aucun paramètre n'a été passé sur la ligne de commandes. Le programme affiche alors la chaîne vide lorsque l'on demande sa valeur.

On peut aussi utiliser `"$*"` pour manipuler l'ensemble des paramètres comme une seule chaîne de caractères ou `"$@"` pour manipuler l'ensemble des paramètres comme une liste de chaînes de caractères sur laquelle on peut itérer. On en verra un exemple dans la section 3.3.

Comme tous les programmes, un script doit retourner un code d'erreur avant de rendre la main au système d'exploitation. Par défaut, le code retourné est celui de la dernière commande exécutée dans le script. On peut retourner explicitement un autre code avec la commande exit suivi du code d'erreur voulu : `exit 1` pour retourner le code d'erreur 1, par exemple .

Le texte qui suit un signe « # » est considéré comme du commentaire jusqu'à la fin de la ligne :

```
#!/usr/bin/env bash
# Ceci est un commentaire
a=4 # Ceci est un autre commentaire
```

3.2 Tests et conditionnelles

Comme les langages de programmation traditionnels, Bash offre la possibilité d'exécuter du code conditionnellement avec deux constructions :

— Une alternative (`if ...; then [else ] fi`), où le cas *sinon* est optionnel ;
— Un choix entre de multiples possibilités (`case ... esac`).

Comme d'habitude, un test est vrai si le code d'erreur associé vaut 0. On peut donc écrire :

```
if ls 2.txt 2>/dev/null 1>&2; then
   echo "2.txt existe"
else
   echo "2.txt n'existe pas"
fi
```

en utilisant le fait que la commande ls retournera un code d'erreur non nul si le fichier `2.txt` n'existe pas. On a aussi dérouté la sortie standard vers la sortie d'erreur standard (`1>&2`) et la sortie d'erreur standard vers le fichier « trou noir » `/dev/null`.

Pour tester l'existence d'un fichier, on peut directement utiliser le test « -e » en l'entourant entre crochets :

```
if [ -e 2.txt ]; then
   echo "2.txt existe"
else
   echo "2.txt n'existe pas"
fi
```

Attention : il est impératif de laisser un espace après le crochet ouvrant et avant le crochet fermant, sinon il y a une erreur de syntaxe.

Vous trouverez une liste des autres tests dans la colonne 15 du Bash Reference Card disponible sur madoc.

Lorsqu'il existe de multiples possibilités, on peut comparer avec `case` la valeur d'une variable avec différentes expressions en utilisant optionnellement une syntaxe similaire au *globbing*.

Avec le script `case.sh` :

```
#!/usr/bin/env bash
case $1 in
    poire|pomme)
        echo "Fruit"
```

```
        ;;
    b*)
        echo "Mot commençant par b"
        ;;
    *)
        echo "Inconnu"
        ;;
esac
```

on a les appels suivants :

```
X1BI030@pc1> ./case.sh pomme
Fruit
X1BI030@pc1> ./case.sh poire
Fruit
X1BI030@pc1> ./case.sh banane
Mot commençant par b
X1BI030@pc1> ./case.sh bison
Mot commençant par b
X1BI030@pc1> ./case.sh abricot
Inconnu
```

3.3 Les répétitives

Bash offre quatre types de répétitives mais on n'en considérera que deux :

— La boucle **for**, pour exécuter un nombre d'itérations connu à l'avance sur les éléments d'une liste ;

— La boucle **while**, pour exécuter un nombre d'itérations a priori inconnu.

La liste d'éléments de la boucle **for** peut être explicite ou provenir de l'expansion d'une expression de *globbing*. La syntaxe est :

```
for nom_variable in liste; do
  Commande 1
  Commande 2
  ...
  Commande k
done
```

On pourrait ainsi avoir :

```
for f in 1.txt 2.txt 3.txt; do
  rm $f
done
```

ce qui serait fonctionnellement équivalent à :

```
rm 1.txt 2.txt 3.txt
```

La commande seq génère une liste d'entiers :

```
X1BI030@pc1> seq 4 7
4
5
6
7
```

En utilisant seq, on peut écrire une boucle itérant sur les entiers de 0 à 9 :

```
for i in $(seq 0 9); do
  echo $i
done
```

L'utilisation du *globbing* permet de traiter facilement une liste de fichiers :

```
# Affichage de la liste des fichiers `.txt` du répertoire parent
# contenant au moins 5 lignes.
for fic in ../*.txt; do
   if [ $(wc -l < "$fic") -ge 5]; then
      echo $fic
   fi
done
```

La commande wc permet de connaître le nombre de lignes d'un fichier avec l'option -l. Notez aussi l'utilisation des guillemets autour de `$fic` pour éviter les problèmes si le nom de fichier courant contient des espaces.

Lorsque l'on ne connait pas le nombre d'itérations a priori, on peut utiliser une boucle « **while** » en testant à chaque itération une condition exprimée de la même manière que pour une conditionnelle « **if** » :

```
# Traitement ligne à ligne d'un fichier texte
cat citadelle.txt | while read ligne; do echo $ligne; done
```

```
# Lecture contrôlée du choix d'un utilisateur
read -p "Oui ou Non (O/N)? " reponse
while [ "$reponse" != "O" -a "$reponse" != "N" ]; do
  echo "Erreur, il faut répondre 'O' ou 'N'" 1>&2
  read -p "Oui ou Non (O/N)? " reponse
done
```

4 Les expressions régulières

Some people, when confronted with a problem, think
"I know, I'll use regular expressions."
Now they have two problems. — Jamie Zawinski.

Les expressions de *globbing* permettent de parler en intention d'un ensemble de fichiers ou répertoires dont les noms partagent certaines caractéristiques. Les expressions régulières ont le même objectif pour du texte quelconque. On réservera donc en général le *globbing* pour les noms de fichiers et les expressions régulières pour leur contenu, même si certaines commandes comme find autorisent l'usage des expressions régulières pour parler des noms de fichiers.

Il existe deux syntaxes pour les expressions régulières :

— Les expressions régulières simples (*Basic Regular Expressions*, ou BRE) ;

— Les expressions régulières étendues (*Extended Regular Expressions*, ou ERE).

Le tableau 2 compare les deux syntaxes avec celle du *globbing*. Le tableau 3 donnent des exemples simples d'utilisation des opérateurs.

Les expressions régulières utilisent les mêmes symboles spéciaux que le *globbing* mais avec un sens différent. Il faut donc faire attention à ne pas confondre les deux.

Les expressions régulières sont disponibles dans la plupart des langages de programmation, soit directement au niveau de la syntaxe du langage (AWK, par exemple), soit grâce à la bibliothèques standard (Python, C, C++, ...). Elles peuvent être utilisées pour rechercher efficacement du texte *matchant* un certain patron, voire pour remplacer le texte matché. Les codes suivants montrent des exemples de recherche avec la commande grep. Cette commande affiche les lignes d'un fichier de texte qui matchent une certaine expression régulière. La commande accepte de nombreuses options pour modifier ce comportement de base (par exemple, pour afficher toutes les lignes qui ne matchent pas une expression régulière).

```
# Recherche dans le fichier `numeros.txt` de l'ensemble des numéros de téléphone
# commençant par `06` et possédant au moins un 7 dans les chiffres suivants:
X1BI030@pc1> grep '^06.*7.*' numeros.txt
```

Table 2 : Comparaison des opérateurs pour les expressions régulières BRE et ERE.

Regexp		Sens	*Globbing*
Basic	Extended		
`[^a-z05]`	`[^a-z05]`	N'importe quel caractère non dans l'ensemble	Idem
.	.	N'importe quel caractère	.
`^$`	`^$`	Ancres de début/fin	`^$`
\?	?	Répéteur (0 ou 1 fois)	1 caractère
*	*	Répéteur (0 ou n fois)	0 ou n caractères
\+	+	Répéteur (1 ou n fois)	+
\(\)	()	Groupement de patrons	—
\{n\}	{n}	Répéteur (exactement n fois)	Expansion d'accolades
\{n,\}	{n,}	Répéteur (n fois ou plus)	Expansion d'accolades
\{n,m\}	{n,m}	Répéteur (entre n et m fois)	Expansion d'accolades
|	|	Alternative (un patron ou l'autre)	—

On prendra soin de quoter l'expression régulière pour qu'elle ne soit pas confondue avec une expression de *globbing*.

Lorsque l'on écrit une expression régulière, il est important de comprendre comment elle matche un texte. Pour cela, il faut se rappeler les trois règles fondamentales du *matching*, qui s'appliquent dans cet ordre :

1. On matche le texte de la gauche vers la droite ;
2. On essaye de matcher le texte le plus long possible sans remettre en cause un matching ultérieur ;
3. On ne fait jamais de retour arrière (un morceau de texte ne peut être matché qu'une fois).

Le site Regex Golf permet de s'amuser à écrire des expressions régulières devant matcher une série de mots (et éviter de matcher une autre série).

Le programme sed est un outil offrant de multiples possibilités de manipulation de texte. Nous ne verrons ici que les possibilités de recherche/remplacement utilisant des expressions régulières. Je vous renvoie à sa documentation pour plus de détails sur ses autres capacités.

Pour chercher/remplacer du texte avec sed, on utilise la commande :

```
sed 's/expression régulière cherchée/remplacement/[g]'
```

L'expression régulière est au format BRE par défaut. L'option « `-E` » permet d'utiliser le format ERE.

Le remplacement de tous les `'a'` par des `'A'` dans un texte qui peut se faire avec la commande tr :

```
X1BI030@pc1> tr a A < fichier.txt
```

peut se faire avec sed de la façon suivante :

```
X1BI030@pc1> sed 's/a/A/g' fichier.txt
```

Sans l'option `'g'` finale, le remplacement ne serait fait que sur la première occurrence de l'expression régulière `'a'`.

Table 3 : Exemples d'utilisation des opérateurs BRE et ERE.

BRE	ERE	Matching
`[0-9]\+`	`[0-9]+`	Entier positif (1 ou plusieurs chiffres)
`[-+]\?[0-9]\+`	`[-+]?[0-9]+`	Entier positif ou négatif (un signe optionnel, suivi d'un entier positif)
`[a-zA-Z_][a-zA-Z0-9]*`	Idem	Identifiant (Une lettre ou un blanc souligné, suivi optionnellement d'une lettre, d'un blanc souligné ou d'un chiffre)

Autres exemples.

```
X1BI030@pc1> echo "pomme poire abricot tomate" | sed 's/pomme\|poire/banane/g'
banane banane abricot tomate
```

Il est possible de réutiliser le texte matché dans le texte de remplacement en mettant entre parenthèses la partie d'expression régulière à réutiliser et en utilisant `\1`, `\2`, ... dans le texte de remplacement ; Chaque `\i` est remplacé par le texte matché par la i^e paire de parenthèses :

```
X1BI030@pc1> echo 1234+654 | sed -E 's/([0-9]+)\+([0-9]+)/\2+\1/'
654+1234
```

5 Les fonctions

Afin de mieux structurer le code d'un script et réutiliser du code à plusieurs endroits, il est possible de définir des fonctions avec la syntaxe :

```
function nom () {
  commande 1
  commande 2
  ...
}
```

Une fonction peut être définie n'importe où dans un script. Elle doit cependant être définie avant sa première utilisation.

Les paramètres de la fonction sont accessibles dans les variables `$1`, `$2`, ..., rendant temporairement inaccessibles les paramètres du script lui-même. La variable `$#` contient le nombre de paramètres :

```
function hello() {
  echo "Bonjour $1 !"
}

hello Jean
```

Si l'on veut définir des variables locales à la fonction, il faut préfixer leur définition avec le mot-clé `local`. Sinon, elles sont considérées comme globales :

```
function local_global() {
  local x=3
  y=5
}

x=1
y=1
local_global
echo $x $y # Affiche 1 5
```

Une fonction peut retourner une valeur numérique sous forme d'un code d'erreur avec le mot-clé **return** :

```
function somme() {
  return $(($1+$2))
}

somme 3 4
echo $? # Affiche 7
```

Le plus souvent, le résultat d'une fonction sera le texte envoyé vers la sortie standard :

```
function miroir() {
  echo $1$(echo $1 |rev)
}
miroir pomme # Affiche pommeemmop
```

6 Gestion des processus

On a vu dans la section 1.2 que le système d'exploitation Linux est capable de gérer l'exécution de plusieurs programmes en même temps, soit concurremment soit en parallèle, en fonction des capacités de l'ordinateur utilisé.

Le *shell* offre plusieurs commandes pour contrôler les différentes tâches (ou processus) en cours d'exécution. La commande ps permet d'afficher la liste des processus suivant différents critères. Par exemple, pour voir tous les processus appartenant à l'utilisateur Dupont, on peut écrire :

```
X1BI030@pc1> ps -u dupont
 PID TTY          TIME CMD
 1499 ?        00:00:00 systemd
 1500 ?        00:00:00 (sd-pam)
 1518 ?        00:00:00 kwalletd
 1520 ?        00:00:00 kwalletd5
 1521 ?        00:00:00 startkde
 1575 ?        00:00:00 dbus-daemon
 1610 ?        00:00:00 ssh-agent
 1646 ?        00:00:00 start_kdeinit
 1647 ?        00:00:00 kdeinit5
 1648 ?        00:00:00 klauncher
 3107 pts/1    00:00:03 okular
 ...
 3181 ?        00:00:54 RDD Process
 4041 ?        00:00:03 Privileged Cont
 4157 ?        00:01:06 Web Content
 5017 ?        00:00:00 gvfsd-metadata
 8503 pts/3    00:00:00 ps
```

La colonne PID indique l'identifiant unique associé à chaque processus. Grâce au PID, on peut envoyer des signaux à un processus avec la commande kill. On peut, par exemple, forcer un processus à s'arrêter, à condition d'avoir les droits suffisants :

```
# Arrêt du processus associé au programme `okular`
X1BI030@pc1> kill -9 3107
```

Un processus peut s'exécuter en arrière-plan (par exemple, s'il a été lancé avec une esperluète) :

```
# Exécution en arrière plan
X1BI030@pc1> mplayer ~/mp3/nocturne.mp3 &
[1] 12034
```

Grâce à la commande fg, on peut utiliser le PID (ici 12034) pour remettre en avant-plan la commande :

```
X1BI030@pc1> fg
mplayer ~/mp3/nocturne.mp3
```

La combinaison de touches Ctrl + z exécutée dans le terminal suspend l'exécution de la commande en cours (ici `mplayer`, revenu en avant-plan) :

```
^Z
[1]+  Stopped                 mplayer ~/mp3/nocturne.mp3
```

On peut alors la relancer en avant-plan avec fg ou l'exécuter en arrière-plan avec bg :

```
X1BI030@pc1> bg
[1]+ mplayer ~/mp3/nocturne.mp3 &
```

La combinaison de touches Ctrl + c envoie un signal d'arrêt définif au programme en avant-plan et permet de récupérer la main dans le terminal.

7 Commandes utiles

L'interpréteur Bash offre une poignée de commandes internes (cd, ...) ; le reste des fonctionnalités du *shell* est offert par des programmes externes. Certains sont présents sur toutes les plateformes UNIX, d'autres ne sont disponibles que dans des environnements particuliers (Linux, par exemple). Je présente très rapidement dans cette section les commandes les plus utiles et susceptibles d'être disponibles en standard dans les distributions Linux les plus communes. On se reportera à leur documentation pour connaitre les détails de leur utilisation.

basename. Retourne le nom d'un fichier débarrassé de son suffixe et de son chemin d'accès :

```
X1BI030@pc1> basename toto.txt .txt
toto
X1BI030@pc1> basename /home/dupont/mp3/nocturne.mp3
nocturne.mp3
```

Notez que pour supprimer le suffixe, il faut l'indiquer explicitement en paramètre (ici, « `.txt` ») ;

cut. Extrait des colonnes d'un texte. Les colonnes peuvent être définies par un nombre de caractères ou par un séparateur de champs :

```
# Extraction de la deuxième colonne d'un texte où les colonnes
# sont séparées par des virgules
X1BI030@pc1> echo a,b12,c0 | cut -d, -f 2
b12
# Extraction des caractères 2,3 et 4
X1BI030@pc1> echo abcdefg | cut -c2-4
bcd
```

date. Affichage ou modification de la date et de l'heure suivant différents formats :

```
# Affichage du jour de la semaine
X1BI030@pc1> date '+%A'
jeudi
```

dirname. Extrait le chemin d'accès d'un fichier :

```
X1BI030@pc1> dirname toto.txt
.
X1BI030@pc1> dirname /home/dupont/mp3/nocturne.mp3
/home/dupont/mp3
```

La commande travaille uniquement sur le texte fourni en paramètre indépendemment de l'existence du fichier en question ;

expr. Évalue une expression arithmétique ou logique. Peut aussi se substituer à sed pour certaines manipulations de chaînes de caractères simples :

```
X1BI030@pc1> expr 45 + 12
57
# Calcul de la longueur d'une chaîne
X1BI030@pc1> expr length "Toto"
4
# Extractions de 4 caractères à partir du 2e
X1BI030@pc1> expr substr "Les sanglots longs" 2 4
es s
# Extraction d'une portion de chaîne définie par une BRE
X1BI030@pc1> expr "Nombre 123 nombre" : '[^0-9]*\([0-9]\+\)'
123
```

Attention : l'usage d'une expression régulière se fait avec une ancre de début implicite. L'expression régulière doit donc matcher le texte à partir du début de la chaîne :

```
# Rien ne sera matché car la chaîne ne commence pas par un entier
X1BI030@pc1> expr "Nombre 123 nombre" : '\([0-9]\+\)'
```

false. Ne fait rien et retourne le code d'erreur 1 :

```
X1BI030@pc1> false; echo $?
1
```

file. Détermine le type d'un fichier en examinant son contenu :

```
X1BI030@pc1> file toto.txt
toto.txt: ASCII text
X1BI030@pc1> file tutu.py
tutu.py: Python script, ASCII text executable
```

find. Recherche de fichiers et répertoires suivant de multiples critères. La recherche peut se faire récursivement dans une arborescence. Il est même possible d'exécuter des actions pour chaque fichier trouvé :

```
# Recherche de tous les fichiers d'extension `.txt` dans l'arborescence
# commençant dans le répertoire courant et modifiés plus récemment que
# le fichier `toto.log`
X1BI030@pc1>  find . -name "*.txt" -newer toto.log -exec cat {} \;
```

L'option `-exec` prend une liste de commandes séparées par des points-virgules qu'elle exécute pour chaque fichier trouvé ; la paire d'accolates « `{}` » est remplacée par le nom de chaque fichier trouvé. La fin de la liste de commandes est indiquée par un point-virgule précédé d'un *backslash* ;

locate. Recherche d'un fichier sur tous les supports de données. Cette commande maintient un fichier contenant les chemins d'accès de tous les fichiers et répertoires. La recherche se fait directement dans ce fichier, ce qui est plus rapide que de faire une recherche effective dans tous les répertoires de l'arborescence. En retour, le résultat de la recherche peut être faussé si le fichier n'est pas régulièrement mis à jour (avec updatedb) :

```
X1BI030@pc1> locate nocturne.mp3
/home/dupont/mp3/nocturne.mp3
/home/durant/musique/01_nocturne.mp3
```

md5sum. Calcule un nombre unique, la somme MD5, identifiant le contenu d'un fichier de manière quasi-unique. Si deux fichiers ont la même somme MD5, on considère qu'ils sont identiques avec une probabilité très grande :

```
X1BI030@pc1> md5sum ~/mp3/nocturne.mp3
54c7cef4b3dae311a67f04bf73b23c9b   /home/dupont/mp3/nocturne.mp3
```

La somme MD5 est exprimée en *hexadécimal*.

mktemp. Retourne un nom unique pour un fichier temporaire. Le fichier n'est pas créé. La commande garantit que le nom retourné n'existe pas déjà :

```
X1BI030@pc1> mktemp
/tmp/tmp.2JPEH9qRT3
```

printf. Affichage de texte formaté. Cette commande fonctionne comme la fonction `printf()` du langage C :

```
X1BI030@pc1> printf "Hello %s\n" Jean
Hello Jean
```

sleep. Ne fait rien pendant un délai exprimé en secondes :

```
# Attend 3 secondes avant de rendre la main à l'utilisateur
X1BI030@pc1> sleep 3
```

sort. Effectue un tri de texte. Le tri peut être alphabétique ou numérique. On peut trier sur une partie seulement du texte :

```
# Tri numérique sur le deuxième champ
X1BI030@pc1> echo -e "1 2 3\n4 -1 6"|sort -n -k2
4 -1 6
1 2 3
```

tee. Copie ce qui est envoyé vers la sortie standard dans un fichier :

```
X1BI030@pc1> ls | tee ls.log
1.txt 2.txt 3.txt
X1BI030@pc1> cat ls.log
1.txt 2.txt 3.txt
```

test. Retourne la valeur de vérité d'une expression logique :

```
# Test de l'existence du fichier 1.txt
X1BI030@pc1> test -e 1.txt; echo $?
0
```

timeout. Lance un programme et lui envoie un signal (arrêt, par défaut) après un temps donné en paramètre :

```
# Arrêt de la commande `sleep` après seulement 2 secondes au lieu de 1000
X1BI030@pc1> timeout 2s sleep 1000
```

true. Ne fait rien et retourne le code d'erreur 0 :

```
X1BI030@pc1> true; echo $?
0
```

uniq. Affiche ou supprime des lignes répétées. Pour fonctionner, les lignes dupliquées doivent être adjacentes :

```
# Suppression des lignes dupliquées
X1BI030@pc1> echo -e "pomme\npoire\npoire"|uniq
pomme
poire
# Comptage du nombre d'occurrences de chaque ligne
X1BI030@pc1> echo -e "pomme\npoire\npoire"|uniq -c
      1 pomme
      2 poire
# Affichage des seules lignes dupliquées
X1BI030@pc1> echo -e "pomme\npoire\npoire"|uniq -d
poire
```

whoami. Retourne le nom de l'utilisateur connecté

```
X1BI030@pc1> whoami
dupont
```

xargs. Passe en paramètres à une commande donnée en paramètre le texte arrivant sur l'entrée standard :

```
# Appel de `echo` en lui passant les paramètres 3 par 3
X1BI030@pc1> echo 1 2 3 4 5 6 7 8 9 | xargs -n3 echo
1 2 3
4 5 6
7 8 9
```

8 Licence de ce document

Les remarques, suggestions et indications d'erreurs peuvent être transmises à l'auteur : frederic.goualard@univ-nantes.fr.

Index des commandes