Licence 2 CUPGE-UPSSITECH

KINUPI41 - SYSTÈMES

Support de cours/TD intégrés et de TP

Alain CROUZIL, Jean-Denis DUROU

{crouzil,durou}@irit.fr

Avant-propos

Objectif

Ce support est destiné à être utilisé durant les séances de cours/TD intégrés et de TP. Il ne constitue pas un support complet dans la mesure où, en séance, vous seront donnés des explications, des exemples, des illustrations et des compléments indispensables. Toutefois, il permet de préparer à l'avance les séances.

Utilisation

Les trois premiers chapitres concernent plus spécifiquement les cours/TD intégrés et sont ponctués d'exercices. Le chapitre 4 contient les sujets des exercices qui doivent être faits dans les séances de TP . Enfin, le chapitre 5 présente les descriptions des commandes qui sont utilisées dans les exercices. La consultation de ces descriptions étant nécessaire à la réalisation des exercices, la plupart des sujets renvoient le lecteur aux descriptions de certaines commandes. Par ailleurs, l'index situé à la fin du document constitue un outil indispensable pour accéder rapidement à la description d'une commande à partir de son nom.

Signification des pictogrammes

Dans la marge ou près de certains titres, apparaissent les pictogrammes suivants :

✎ désigne un sujet d'exercice de cours/TD intégrés ;

👁 signale les commandes dont il est utile de lire la description pour faire chaque séance de TP ;

☞ indique la description d'une commande ;

➥ précède chaque rubrique de la description d'une commande ;

 signale un point délicat ;

✍ est un repère destiné à l'enseignant.

Bonne lecture !

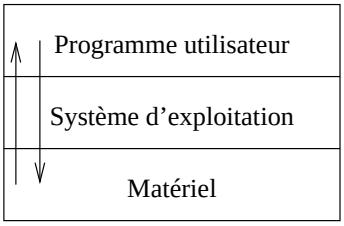
Table des matières

Chapitre 1
Les systèmes d'exploitation

1.1 Définition

Un système d'exploitation (SE) est un logiciel destiné à faciliter et à simplifier l'utilisation d'un ordinateur. Il permet d'exploiter les capacités du matériel en mettant à la disposition de l'utilisateur un ensemble de services.

Le SE est la première couche de logiciel : il prend appui sur les circuits pour mettre à la disposition des programmes d'application les possibilités du matériel.



On distingue différents types de SE :

- Systèmes mono-utilisateur monotâche : un seul utilisateur et un seul programme exécuté à la fois (ex : MS-DOS).
- Systèmes mono-utilisateur multitâche : un seul utilisateur qui peut exécuter plusieurs programmes en même temps (ex : Windows 95/98/Millenium/2000/XP).
- Systèmes multi-utilisateur multitâche : plusieurs utilisateurs peuvent utiliser le même ordinateur en même temps et exécuter plusieurs programmes (ex : Unix, Windows NT serveur / 2000 serveur / 2003 serveur).

1.2 Historique des systèmes d'exploitation

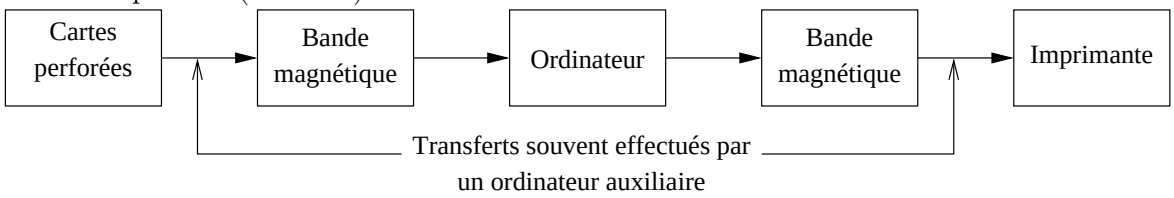
L'évolution des systèmes est étroitement liée à celle du matériel.

1.2.1 Génération zéro : premiers prototypes (44-50)

- Programmation en langage machine uniquement.
- Mise au point « en ligne »(en examinant le contenu binaire de la mémoire).
- Périphériques d'entrées/sorties (E/S) primitifs.
- Pas de système d'exploitation.

1.2.2 Première génération : machines commerciales (51-58)

- Présence d'un superviseur permettant le parallélisme CPU-E/S (CPU : *Central Process Unit*, processeur central). Il permettait de lire le travail suivant sur une bande magnétique d'entrée, lui donner le CPU, reprendre le contrôle, mettre les résultats sur bande.
- Traitement par lots (fournées) :



  Le programmeur n'a aucun accès à la machine :
  - il remet à un opérateur un paquet de cartes ;
  - l'opérateur organise les travaux ;
  - le programmeur récupère le listing des résultats dans un casier.

On peut noter le développement du langage Fortran.

1.2.3 Deuxième génération : machines à transistors (58-64)

- Apparition des disques.
- Le superviseur prend en charge le traitement de toutes les E/S : pour accéder aux périphériques, l'utilisateur doit faire appel à des directives du système.
- Un sous-ensemble du système gère les disques et le placement des fichiers (protections).
- Apparition d'utilitaires : vidage mémoire, gestion de catalogues de fichiers.

1.2.4 Troisième génération : machines à circuits intégrés (64-73)

- Systèmes multi-programmés : exécution « simultanée »de plusieurs programmes (le CPU est ainsi toujours occupé).
- Le système doit gérer le partage des ressoures :
  - allocation du CPU, ordonnancement des tâches ;
  - allocation et partage de la mémoire ;
  - synchronisation entre les tâches ;
  - gestion des files complexes d'E/S.

1.2.5 Quatrième génération : minis et micro-ordinateurs ($\geq$73)

- Apparition des microprocesseurs (CPU sur une seule « puce », 71).
- Les systèmes d'exploitation sont de plus en plus sophistiqués.
- Leur interface utilisateur devient plus simple (« convivialité »).
- Apparition de systèmes comme :
  - VM/CMS (IBM) ;
  - VMS (VAX de Digital) ;
  - CP/M, MS/DOS (IBM-PC et compatibles) et MAC OS (Apple) pour micro-ordinateurs ;
  - Unix pour micros et minis.

1.2.6 Cinquième génération : VLSI, machines dédiées, réseaux (à partir des années 80)

Circuits VLSI (*Very Large Scale Integration*) : une puce peut contenir un très grand nombre de transistors pour un coût faible.

⇒ Construction de machines spéciales dédiées à des applications spécifiques (traitement d'images, du signal, etc.).

⇒ Construction de matériel pour les réseaux : réseaux locaux permettant de répartir la puissance de calcul et de partager des périphériques.

1.3 Motivations du développement des systèmes d'exploitation

— Réduire et dominer la complexité des machines :
  — L'utilisateur doit pouvoir lancer des opérations complexes avec des intructions simples.
  — Les programmes doivent pouvoir utiliser les services fournis par le SE.
— Rentabiliser l'utilisation des ordinateurs :
  — en faisant travailler le CPU de manière optimale ;
  — en facilitant la programmation.
— Préserver les investissements : les SE doivent permettre de changer de matériel en modifiant le moins possible les programmes des utilisateurs (compatibilité).
— Faciliter la vie des utilisateurs : les SE doivent fournir des interfaces conviviales pour travailler efficacement et confortablement.

1.4 Principales tâches réalisées par un SE

1.4.1 Gestion des processus

Un processus est un programme en cours d'exécution. Cette partie concerne donc l'exécution des programmes : création, gestion de l'exécution simultanée (multitâche), synchronisation, communication, etc.

On parle de multiprogrammation, de temps partagé et de parallélisme. Les programmes doivent partager le processeur central (CPU : *Central Process Unit*) pour permettre une exécution « simultanée ». Pour cela, le système doit :

- pouvoir reprendre le contrôle du CPU à tout moment pour répondre à un événement extérieur (ex : `Ctrl-c`) ;
- posséder un algorithme d'ordonnancement pour décider, à un instant donné, lequel, parmi les programmes présents, doit recevoir le contrôle du CPU ;
- répartir la mémoire physique entre les différents programmes en concurrence (gestion des conflits) ;
- fournir des directives permettant à l'utilisateur de contrôler les programmes en cours (destruction, changement de priorités, mise en sommeil, réveil, etc.) ;
- permettre à plusieurs programmes de partager du code ou des données en mémoire ;
- disposer d'un système de protection des utilisateurs entre eux et du SE contre les utilisateurs (protection des données et des programmes en mémoire et sur les disques) ;
- gérer le parallélisme si la machine dispose de plusieurs processeurs.

Le SE gère les communication entre les processus et fournit des outils permettant à plusieurs processus de se synchroniser.

1.4.2 Gestion de la mémoire

Cette partie concerne :
- le transfert des programmes et des données nécessaires à la création des processus depuis un support secondaire (disque) vers un support central (RAM : *Random Access Memory*) ;
- l'allocation de la mémoire ;
- le transfert depuis la mémoire principale vers la mémoire secondaire.

1.4.3 Gestion des fichiers

Il s'agit de gérer les mémoires secondaires : l'utilisateur d'une machine doit pouvoir stocker les programmes et les données dans des fichiers confiés au SE (*file system*, *network file system*, etc.).

Cette partie concerne la création, la destruction, la correspondance avec les dispositifs physiques, les protections, le catalogage, le partage, etc.

1.4.4 Gestion des entrées-sorties

Cette partie concerne les mécanismes qu'utilisent les processus pour communiquer avec l'extérieur (clavier, écran, disques, etc.). Les entrées-sorties sont lentes par rapport à la vitesse du CPU.

1.4.5 Gestion des communications dans les réseaux

Cette partie concerne la gestion des connexions à un réseau de communication, les systèmes répartis et les architectures clients/serveurs.

1.4.6 Interfaces avec l'utilisateur

Le SE doit fournir une interface de dialogue entre le système et ses utilisateurs. Pour cela, il doit :
- fournir un sous-système de gestion des fichiers (voir paragraphe 1.4.3) ;
- permettre le chargement de programmes exécutables en mémoire (voir paragraphe 1.4.2) ;
- posséder un système de traitement des erreurs ;
- fournir des outils permettant à un utilisateur d'intervenir dans le déroulement de son programme (interruptions programmées, arrêt, mise au point, etc.) ;
- fournir des utilitaires : éditeur de texte, compilateur, outils de mise au point, système de file d'attente pour l'imprimante, envoi de messages sur une console distante, etc. ;
- fournir des outils pour l'administration du système permettant de :
  - gérer les utilisateurs :
    - en attribuant à chaque utilisateur un nom (*login*) et un mot de passe ;
    - en attribuant à chaque utilisateur un certain nombre de ressources (espace disque pour les fichiers, privilèges pour l'exécution de commandes spéciales, nombre de fichiers ouverts en même temps, etc.) ;
    - en enregistrant les ressources effectivement utilisées ;
  - gérer les ressources du système :
    - en permettant de régler et de mettre à jour le système en fonction du matériel ;
    - en permettant d'effectuer des sauvegardes régulières de tous les fichiers ;
  - contrôler les performances en effectuant des mesures et en détectant les points d'embouteillage et les saturations (utilisation du CPU, de la mémoire, taux de remplissage des disques, temps de connexion, etc.) ;
- fournir un interpréteur de commandes (*shell*) et/ou une interface graphique : c'est la partie la plus « visible »du système.

1.5 Possibilités de communication avec un SE

L'utilisateur peut communiquer avec le SE, c'est-à-dire faire appel à ses services, à travers : ✍ 1

- les « appels systèmes »à partir d'un langage de programmation (API : *Application Programming Interface*) comme par exemple en C sous Unix ;
- un interpréteur de commandes (*shell*) sous la forme de :
  - lignes de commandes,
  - programmes (*scripts*) ;
- une interface graphique (Aqua, Windows, X-Window, KDE, etc.).

Chapitre 2
Les systèmes Unix

2.1 Généralités

- Début du développement : 1969, laboratoire Bell ATT.
- Première version diffusée : version 6 en 1975, écrite en langage C qui a été développé en même temps qu'Unix pour en assurer la portabilité.

Aujourd'hui Unix fait référence à une famille de SE qui ont des fonctionnalités similaires. En particulier, ils sont multi-utilisateur et multitâche.

Exemples de systèmes Unix :

AIX	IBM
HP/UX	Hewlett-Packard
Solaris	Sun Microsystems
SunOs	Sun Microsystems
Mac OS	Apple
Linux	PC

Quelques raisons du succès d'Unix :

- Écrit dans un langage de haut niveau (portabilité).
- Interface utilisateur simple.
- Appels système réutilisables pour l'écriture de nouvelles commandes.
- Système de gestion de fichiers performant.
- Multi-utilisateur et multitâche.

2.2 Structure générale d'Unix

Un schéma et des explications vont seront donnés durant la séance. ✍ 2

2.3 Interpréteurs de commandes

Un *shell* est un interpréteur de commandes, c'est-à-dire un programme qui lit une commande, l'exécute et attend la commande suivante. Sous Unix, c'est l'interface que l'on utilise couramment pour communiquer avec le système. Il est possible de regrouper plusieurs commandes dans un fichier et d'utiliser des structures de contrôle afin d'écrire des *scripts shell*.

Il existe plusieurs shells :

- la famille des *Bourne shells* : sh, ksh, bash, zsh ;

- la famille des *C-shells* : csh, tcsh.

2.4 Format général d'une commande Unix

`nom_commande options arguments`

Les options se situent juste après le nom de la commande ; elles se caractérisent par le signe - suivi d'une lettre. Les arguments se situent après les options (s'il y en a). Les options et les arguments forment les paramètres de la commande.
Exemple de commande avec 2 options et 3 arguments :
`ls -a -l fichier1 fichier2 fichier3`
Plusieurs options peuvent être regroupées (dans n'importe quel ordre) : `ls -a -l` $\Longleftrightarrow$ `ls -al`
Certaines options peuvent être suivies d'un argument associé à cette option.

2.5 Syntaxe de description d'une commande Unix

La commande `man` (page 59) permet d'obtenir la description d'une commande en tapant :

`man nom_commande`

Les règles qui définissent la syntaxe d'une commande sont les suivantes :
- Les termes entre crochets sont facultatifs.
- Les termes qui ne sont pas entre crochets sont obligatoires.
- Tout ce qui est affiché en **gras** (ou en caractères normaux) doit être tapé tel quel.
- Tout ce qui est affiché en *italique* (ou souligné) doit être remplacé par la valeur adéquate.
- Le caractère `|` sépare 2 éléments qui ne peuvent pas être utilisés simultanément. Par exemple, `[-a|-b]` signifie que l'on peut n'utiliser ni `a` ni `b`, que l'on peut utiliser `-a`, que l'on peut utiliser `-b`, mais que l'on ne peut pas utiliser `-ab`.
- Tout argument suivi de `...` peut être répété autant de fois que nécessaire. Ainsi, `nom_fichier` [...] désigne une liste non vide de fichiers alors que [`nom_fichier` ...] désigne une liste éventuellement vide de fichiers.

Les différentes valeurs que peut prendre le **code de retour** (*EXIT STATUS*) d'une commande sont précisées dans sa description. Ce code est par exemple utilisé avec les structures de contrôle dans les scripts. En général, lorsque l'exécution d'une commande s'est déroulée normalement, elle retourne le code `0`. Dans les autres cas, elle retourne un entier strictement positif.

✎ Exercice 1

Si la syntaxe de la commande `cut` est décrite par les deux lignes suivantes :
`cut -c list [file ...]`
`cut -f list [-d delim] [-s] [file ...]`
les appels suivants sont-ils corrects :

1. `cut -d : -f 1,7 fichier`
2. `cut -c 1-16,26-38`
3. `cut -s fich1 fich2`
4. `cut -f 2 -c 1,3,5 fichier`

2.6 Système de gestion de fichier (SGF) d'Unix

2.6.1 Différents types de fichiers

Sous Unix, un fichier est une source de données qui peuvent être lues ou une destination dans laquelle des données peuvent être écrites. C'est donc, non seulement un fichier sur disque, mais aussi tout dispositif physique : le clavier est un fichier (source), l'écran est un fichier (destination), l'imprimante est un fichier (destination).

- **Fichiers ordinaires** (`-`) : ils contiennent des données (dont les programmes) et sont sur le disque.
- **Liens symboliques** (`l`) : ils sont des « raccourcis » vers d'autres fichiers.
- **Fichiers spéciaux** (`b` **ou** `c`) : ils représentent un dispositif physique (périphérique). Par exemple, on peut accéder au clavier ou à l'écran comme à des fichiers.
  - Fichiers spéciaux de type blocs (`b`) : les échanges de données se font par blocs.
  - Fichiers spéciaux de type caractères (`c`) : les échanges de données se font par caractères.
- **Tubes nommés** (`p`) : appelés aussi des fichiers FIFO, ce sont des fichiers sur disque gérés comme un tube (*pipe*) ; un processus écrit des données, un autre les lit.
- **Répertoires** (`d`) : ils sont stockés sur disque et contiennent des informations utilisées pour organiser l'accès à d'autres fichiers ; du point de vue conceptuel, un répertoire (*directory*) « contient » d'autres fichiers ; un répertoire peut donc contenir d'autres répertoires : cela permet d'obtenir une structure hiérarchique arborescente.

2.6.2 Modèle hiérarchique

Un schéma utilisé par la suite ainsi que des explications vont seront donnés durant la séance. ✍ 3

2.6.3 Désignation (adressage) d'un fichier

- Chaque répertoire contient deux répertoires particuliers :
  `.` : désigne le répertoire lui-même
  `..` : désigne le répertoire supérieur (le « père »).
- Chaque utilisateur dispose d'un **répertoire d'accueil** (*home directory*), appelé parfois « répertoire principal ».

  Exemple : `/users/linfg/l2inf201`
- À chaque instant, un utilisateur se trouve dans l'un des répertoires du système de fichiers : c'est le **répertoire courant** (*working directory*).
- **Désignation relative** : on désigne un fichier en donnant le chemin (suite des noms des répertoires séparés par le caractère `'/'`) qui mène à ce fichier en partant du répertoire **courant**.
  Exemple : si l'on suppose que le répertoire courant est `/users/linfg/l2inf201/REP1`, alors la désignation relative du fichier `f3` est :
  `../REP2/f3`
- **Désignation absolue** : on désigne un fichier en donnant le chemin qui mène à ce fichier en partant du répertoire **racine** (elle commence toujours par `'/'`).
  Exemple : `/users/linfg/l2inf201/REP2/f3`

✎ **Exercice 2**
Dans une désignation, tous les caractères `'/'` ont-ils la même signification ?

✎ **Exercice 3**

1. La désignation relative et la désignation absolue d'un fichier sont-elles uniques ?
2. On définit la longueur d'une désignation comme le nombre de répertoires qu'elle contient. Pour désigner un même élément, quelle est, parmi la désignation relative et la désignation absolue, celle qui est la plus courte ?

✎ **Exercice 4**

Si le répertoire courant est : `/users/linfg/l2inf201/REP1`
donner la désignation relative du fichier : `/users/linfg/l2inf201/REP1/f1`

2.6.4 Droits d'accès

Tout utilisateur est repéré par son nom (*login*) et par son groupe (par exemple, l'utilisateur `l2inf201` fait partie du groupe `licence`). À chaque fichier sont associées trois classes d'utilisateurs :

- le propriétaire (*user* : `u`) ;
- les membres du groupe du propriétaire (*group* : `g`) ;
- les autres utilisateurs (*others* : `o`).

Chaque classe peut avoir, sur un fichier, les droits de :

- lecture (`r`) ;
- écriture (`w`) ;
- exécution (`x`).

▶ **Pour les répertoires :**
- `r` signifie le droit d'afficher le contenu du répertoire ;
- `w` signifie le droit d'ajouter ou de supprimer des fichiers dans le répertoire ;
- `x` signifie le droit de « traverser » le répertoire.

▶ **Pour les autres types de fichiers :**
- `r` signifie le droit de lire le fichier ;
- `w` signifie le droit de modifier le fichier ;
- `x` signifie le droit d'exécuter le fichier (programme binaire ou script shell).

✍ 4 La commande `ls -l` permet d'afficher des informations, dont les droits d'accès, sur un ou plusieurs fichiers.

2.6.5 Liens

2.6.5.1 Liens physiques

On peut accéder à un même fichier physique par plusieurs chemins, éventuellement avec un nom différent, dans un même système de fichiers. Les liens physiques peuvent être créés grâce à la commande
✍ 5 `ln`.

2.6.5.2 Liens symboliques

Il s'agit de raccourcis pour désigner un fichier. Ils peuvent exister entre plusieurs systèmes de fichiers (partitions qui peuvent être « montées » dans la hiérarchie des fichiers). Les liens symboliques peuvent
✍ 6 être créés grâce à la commande `ln -s`.

2.6.6 Occupation disque

▶ La commande `df -k` donne des informations sur l'occupation en Ko (utilisés, disponibles) des systèmes de fichiers accessibles.

- La commande `du -k répertoire` donne le nombre de Ko occupés par chaque répertoire de la sous-arborescence de racine `répertoire`.

Chapitre 3
Éléments de shell de Bourne

3.1 Métacaractères du shell

Les métacaractères du shell (caractères génériques ou jokers) permettent de :

1. **Construire des listes de noms de fichiers**, séparés par des espaces, correspondant à un certain modèle (si aucun nom de fichier ne correspond au modèle, le modèle n'est pas remplacé) : ✍ 7
   - `*` désigne une chaîne de caractères quelconque (éventuellement vide) ne contenant pas de caractère `/`, et ne commençant pas par un caractère `.` si `*` est placé en début de modèle.
   - `?` désigne un caractère quelconque, à l'exception d'un caractère `/`, ou d'un caractère `.` si `?` est situé en début de modèle.
   - `[liste_caractères]` désigne un caractère dans la liste placée entre crochets, définie par énumération, par intervalle (avec le caractère `-`) ou par négation (avec le caractère `!` placé juste après le crochet ouvrant :
     - `[Aa]` désigne le caractère `A` ou le caractère `a`.
     - `[0-9a-zA-Z]` désigne un caractère alphanumérique quelconque.
     - `[!0-9]` désigne n'importe quel caractère sauf un chiffre.

   Le shell commence par interpréter ces métacaractères, c'est-à-dire forme les listes des noms de fichiers qui correspondent au modèle, puis il transmet ces noms de fichiers aux commandes qui sont alors exécutées. Les commandes reçoivent donc en paramètre le résultat de cette interprétation des métacaractères.

✎ **Exercice 5**

(a) Décrire le modèle de noms de fichiers suivant : `REP/*.???`

(b) Écrire les modèles suivants :
- le caractère `a` ou le caractère `b` ou le caractère `c` ou le caractère `d` ou le caractère `e` ;
- le caractère `a` suivi d'un chiffre ou du caractère `b` ;
- une minuscule ou une majuscule ;
- une minuscule suivie d'un chiffre.

2. **Modifier l'interprétation des commandes** :
   - `;` sépare deux commandes (ou plus) situées sur une même ligne ;
   - `'` délimite une chaîne de caractères contenant des espaces (à l'intérieur, tous les métacaractères perdent leur signification) ;
   - `"` délimite une chaîne de caractères contenant des espaces (à l'intérieur, tous les métacaractères perdent leur signification, à l'exception des métacaractères `` ` ``, `$` et `\`) ;
   - `` ` `` « capture »la sortie standard pour former un nouveau paramètre ou une nouvelle commande (cf. paragraphe 3.2.4.5) ;
   - `<` permet de réaliser des redirections de l'entrée standard des commandes (cf. paragraphe 3.2.3) ;

- `>` permet de réaliser des redirections des sorties standard des commandes (cf. paragraphes 3.2.4 et 3.2.5) ;
- `&` suivi d'un chiffre permet de désigner les unités standard lors des redirections (cf. paragraphe 3.2) ; `&` placé après une commande permet de lancer la commande en arrière plan ;
- `|` permet de réaliser un branchement de commandes (cf. paragraphe 3.2.4.4) ;
- `$` retourne la valeur de la variable qui suit (il ne doit pas y avoir d'espace entre le métacaractère `$` et le nom de la variable) et joue également un rôle très particulier vis-à-vis des « paramètres d'un shell »(cf. paragraphes 3.5 et 3.6) ;
- `\` protège le caractère qui suit, que ce soit un caractère normal ou un métacaractère du shell (sauf à l'intérieur d'une chaîne délimitée par des `'`) ; à l'intérieur d'une chaîne de caractères délimitée par des `"`, le caractère `\` n'est interprété comme un métacaractère que s'il est suivi de l'un des quatre métacaractères `"`, `‘`, `$` ou `\` ; dans tous les autres cas, il n'est pas interprété comme un métacaractère, c'est-à-dire que, par exemple, la chaîne `"\*"` est interprétée de la même façon que la chaîne `'\*'` ;
- `(` et `)` permettent de regrouper un ensemble de commandes et de les exécuter dans un « sous-shell »(cf. paragraphe 3.8.1).

✎ Exercice 6

Commencer par lire la description de la commande `ls`, page 60.

1. Écrire la commande permettant d'afficher les noms de fichiers non cachés dont le nom contient le caractère `a`
2. Écrire la commande permettant d'afficher les noms de fichiers non cachés dont le nom comporte quatre caractères exactement.
3. Écrire la commande permettant d'afficher les noms de fichiers non cachés dont le nom comporte au moins cinq caractères.
4. Écrire la commande permettant d'afficher les noms de fichiers non cachés dont le nom commence par une majuscule.

3.2 Redirections

3.2.1 Unités standard

✍ 8

Chaque processus a accès à trois unités logiques :

- l'entrée standard (`stdin`) : par défaut, le clavier ;
- la sortie standard (`stdout`) : par défaut, l'écran ;
- la sortie standard des erreurs (`stderr`) : par défaut, l'écran.

Les redirections permettent de changer les « connexions »par défaut entre les unités logiques et les unités physiques : `stdin` ← clavier, `stdout` → écran et `stderr` → écran.

Outre les paramètres (options et arguments), certaines commandes peuvent donc recevoir des données par l'intermédiaire de l'entrée standard. Par ailleurs, outre l'effet produit par leur exécution, certaines commandes peuvent fournir des résultats à la sortie standard (résultats « normaux » destinés initialement à être affichés à l'écran) et à la sortie standard des erreurs (messages en cas d'erreur). Une commande qui peut recevoir des données par l'intermédiaire de `stdin` et fournir des résultats à `stdout` est appelée un **filtre**. Les filtres présentent le gros avantage de pouvoir être « branchés »les uns à la suite des autres : en redirigeant la sortie standard du filtre `filtre1` sur l'entrée standard du filtre `filtre2`, on peut réaliser un nouveau filtre, dont l'entrée standard est l'entrée standard de `filtre1` et la sortie standard, la sortie standard de `filtre2`. Enfin, certaines commandes, bien que ne pouvant pas fournir de résultats à `stdout`, peuvent recevoir des données par l'intermédiaire de `stdin`.

Une telle commande ne peut apparaître qu'en fin de branchement (sortie d'un filtre). Inversement, certaines commandes, bien que ne pouvant pas recevoir de données par l'intermédiaire de `stdin`, peuvent fournir des résultats à `stdout`. Une telle commande ne peut apparaître qu'en début de branchement (entrée d'un filtre).

3.2.2 Quelques commandes

Afin d'illustrer les notions qui vont être présentées plus loin, nous introduisons les commandes suivantes (ces commandes sont décrites de manière plus détaillée au chapitre 5) :

`echo` [chaîne...]

affiche sur `stdout` les chaînes de caractères qui sont passées en paramètres.

`cat` [désignation_fichier...]

affiche sur `stdout` le contenu des fichiers dont les désignations sont passées en paramètres. Sans paramètre, `cat` affiche sur `stdout` le contenu de `stdin`.

`grep` chaîne [désignation_fichier...]

affiche sur `stdout` les lignes des fichiers dont les désignations sont passées en paramètres (ou de `stdin` s'il n'y a aucun nom) qui contiennent au moins une occurrence de chaîne.

`wc -l` [désignation_fichier...]

affiche sur `stdout` le nombre de lignes des fichiers dont les désignations sont passées en paramètres (ou de `stdin` s'il n'y a aucun nom).

3.2.3 Redirection de l'entrée standard d'une commande

3.2.3.1 Entrée par défaut

Quand une commande attend des lignes sur `stdin`, on peut les taper au clavier en utilisant l'une des deux syntaxes suivantes :

commande	commande << séparateur
ligne$_1$	ligne$_1$
ligne$_2$	ligne$_2$
...	...
<*Ctrl*>D	séparateur

Exemples (les caractères tapés par l'utilisateur sont les caractères gras) :

$ cat	**$ cat << FIN**
bonjour	**> bonjour**
bonjour	**> monsieur**
monsieur	**> FIN**
monsieur	bonjour
<*Ctrl*>D	monsieur
$	$

Remarques sur la première syntaxe :

- Pour terminer l'exécution de la commande et récupérer le prompt, il est nécessaire de taper <*Ctrl*>D, caractère non éditable de code ASCII égal à 4 qui simule la fin d'un fichier en mode interactif (en fait, il faut maintenir appuyée la touche <Ctrl> du clavier, et taper les caractères D ou d).
- Avec cette syntaxe, le texte tapé au clavier est envoyé à la commande ligne par ligne, donc c'est une syntaxe qui est inadaptée pour certaines commandes (par exemple, pour une commande dont le rôle serait de retourner les deux dernières lignes provenant de l'entrée standard).

Remarques sur la seconde syntaxe :

- Les deux caractères << doivent être collés.
- Cette syntaxe peut être utilisée dans les scripts.

✎ Exercice 7

Que se produit-il si l'utilisateur tape les deux séquences suivantes :

1. `grep 'e' << sep`
   `bonjour,`
   `ceci est`
   `un exemple.`
   `sep`
2. `grep 'e'`
   `bonjour,`
   `ceci est`
   `un exemple.`
   `<Ctrl>D`

3.2.3.2 Redirection de l'entrée standard

✍ 9 `commande < désignation_fichier`

L'entrée standard de `commande` est connectée au fichier `désignation_fichier`.

Exemple : `grep 'e' < fich_entree.txt`
affiche les lignes du fichier `fich_entree.txt` qui contiennent au moins un `e`.

Remarque : une telle redirection occasionne l'ouverture du fichier `désignation_fichier` en mode « lecture ».

3.2.4 Redirections de la sortie standard d'une commande

3.2.4.1 Redirection simple de la sortie standard d'une commande

✍ 10 `commande > désignation_fichier`

La sortie standard de `commande` est connectée au fichier `désignation_fichier`. S'il n'existe pas, il est créé. Sinon, son contenu est écrasé par la sortie de `commande`.

Exemple : `grep 'e' > fich_sortie.txt`
écrit, dans le fichier `fich_sortie.txt`, les lignes tapées au clavier qui contiennent au moins un caractère `e`.

Remarque : une telle redirection occasionne l'ouverture du fichier `désignation_fichier` en mode « écriture ».

3.2.4.2 Redirection de la sortie standard d'une commande avec concaténation

`commande » désignation_fichier`

Si le fichier `désignation_fichier` n'existe pas, il est créé. Sinon, la sortie de `commande` est ajoutée à la fin de `désignation_fichier`.

Exemple : `grep 'e' >> fich_sortie.txt`
rajoute, à la fin du fichier `fich_sortie.txt`, les lignes tapées au clavier qui contiennent au moins un caractère `e`.

Remarque : une telle redirection occasionne l'ouverture du fichier désignation_fichier en mode « écriture en fin de fichier ».

3.2.4.3 Redirection de la sortie standard d'une commande vers sa sortie standard des erreurs

commande >&2 ✍ 11

La sortie standard de commande est connectée à sa sortie standard des erreurs. Cette syntaxe est très utile pour afficher des messages d'erreurs dans les scripts.

Exemple : `echo "Fichier introuvable !" >&2`
affiche la chaîne de caractères `Fichier introuvable !` sur `stderr`.

Remarque : les caractères `>&2` doivent être collés.

3.2.4.4 Branchement de commandes

commande_1 | commande_2 ✍ 12

La sortie standard de commande_1 est connectée à l'entrée standard de commande_2. Le métacaractère | est appelé *pipe* (« tube »). Ce branchement permet de réaliser des enchaînements de commandes.

Exemple : `cat fich.txt | grep 'e' | grep '2' | wc -l`
affiche à l'écran le nombre de lignes du fichier `fich.txt` qui contiennent au moins un caractère `e` et au moins un caractère `2`.

3.2.4.5 Capture de la sortie standard d'une commande

`commande` ✍ 13

La sortie standard de commande est transformée en une chaîne de caractères dans laquelle les éventuels sauts de lignes sont transformés en espaces. Cette chaîne de caractères peut constituer une autre commande mais surtout elle peut être utilisée pour former les paramètres d'une autre commande. Cette capture est réalisée en entourant la commande (ou le branchement de commandes) de métacaractères ` (« apostrophe à gauche »ou *back quote*, à ne pas confondre avec l'apostrophe ').

Exemple : si la commande `ls` envoie les deux lignes suivantes sur sa sortie standard :

```
fich1.txt
fich2.txt
```

alors la commande suivante :

```
grep 'e' `ls`
```

affichera les lignes du fichier `fich1.txt` contenant au moins un caractère `e`, suivies des lignes du fichier `fich2.txt` contenant au moins un caractère `e`.

Remarques :

- La commande qui vient d'être donnée en exemple n'est absolument pas équivalente à la commande `ls | grep 'e'`, puisque cette dernière commande affiche, parmi les noms de fichiers retournés par la commande `ls`, ceux qui contiennent au moins un caractère `e` (dans leur nom).
- Rien n'empêche, *a priori*, que les caractères `<`, `>` ou `>>` soient suivis d'une capture, comme dans :
  `grep 'e' < ‘ls‘`
  mais de telles écritures seront incorrectes si la commande capturée (ici, `ls`) retourne autre chose qu'un nom unique de fichier. Dans la pratique, on évitera ce genre d'écriture.
- Il existe une différence importante entre les deux commandes suivantes :
  `ls > fich.txt`
  `echo ‘ls‘ > fich.txt`
  Dans les deux cas, le résultat de la commande `ls` est écrit dans le fichier `fich.txt`, mais cette écriture se fait de deux manières différentes. Dans le premier cas, un seul nom de fichier ou de sous-répertoire est écrit par ligne. Dans le deuxième cas, tous les noms de fichiers et de sous-répertoires sont écrits sur une même ligne, séparés par des espaces, car la capture transforme les sauts de ligne en espaces.

✎ Exercice 8

Si le fichier `liste.txt` contient une liste de noms de fichiers (un par ligne), écrire la commande permettant d'afficher les lignes des fichiers qui contiennent au moins un caractère `e`.

3.2.5 Redirections de la sortie standard des erreurs d'une commande

3.2.5.1 Redirection simple de la sortie standard des erreurs d'une commande

✍ 14

`commande` `2>` `désignation_fichier`

La sortie standard des erreurs de `commande` est connectée au fichier `désignation_fichier`.

Exemple : `ls fichier.txt 2> /dev/null`
écrit, dans le fichier « poubelle », les éventuels messages d'erreur provenant de la commande `ls`. Ce fichier poubelle fonctionne comme un trou noir : ce qui y est écrit est définitivement perdu. Il est par exemple utilisé lorsque l'on ne souhaite pas voir à l'écran s'afficher d'éventuels messages d'erreurs de certaines commandes.

3.2.5.2 Redirection de la sortie standard des erreurs d'une commande avec concaténation

`commande` `2»` `désignation_fichier`

Si le fichier `désignation_fichier` n'existe pas, il est créé. Sinon, la sortie des erreurs de `commande` est ajoutée à la fin de `désignation_fichier`.

3.2.5.3 Redirection de la sortie standard des erreurs d'une commande vers sa sortie standard

✍ 15

`commande` `2>&1`

La sortie standard des erreurs de `commande` est connectée à sa sortie standard.

Exemple :
`ls -l fichier.txt > informations.txt 2>&1`
redirige à la fois la sortie standard et la sortie standard des erreurs vers le fichier `informations.txt` alors que :

`ls -l fichier.txt 2>&1 > informations.txt`
ne redirige que la sortie standard vers le fichier `informations.txt`, car la sortie standard des erreurs a été connectée à la sortie standard avant que celle-ci ne soit redirigée vers `informations.txt`.

Remarques :

- Les caractères des séquences `2>`, `2>>` ou `2>&1` doivent être collés.
- Ces syntaxes varient d'un shell à l'autre.
- La ligne de commande suivante :
  `echo "Problème" 2> erreurs.txt >&2`
  provoque l'écriture de la chaîne de caractères `Problème` dans le fichier `erreurs.txt`, donc est équivalente à la commande :
  `echo "Problème" > erreurs.txt`
  Mais la ligne de commande suivante :
  `echo "Problème" >&2 2> erreurs.txt`
  provoque l'affichage de la chaîne de caractères `Problème` à l'écran car la redirection de la sortie standard des erreurs vers le fichier `erreurs.txt` a été faite après la redirection de la sortie standard vers la sortie standard des erreurs.

✎ Exercice 9

Commencer par lire la description de la commande `mkdir`, page 63.

1. En une seule ligne de commande, créer le sous-répertoire `REP1` et le sous-sous-répertoire `REP1/SREP1`
2. En une seule ligne de commande, créer deux sous-répertoires `REP2` et `REP3` dans le répertoire courant, puis afficher le message `termine`
3. Même question, mais en écrivant ce message dans le fichier `liste.txt`

✎ Exercice 10

Commencer par lire la description de la commande `mail`, page 77.

1. Que réalisent les deux commandes suivantes :

   ```
   echo "bonjour" > cat
   echo "bonjour" | cat
   ```

2. En une seule ligne de commande, envoyer à l'utilisateur `l2inf250`, par courrier électronique, la liste détaillée des répertoires et fichiers contenus dans le répertoire courant.
3. Soit un fichier `liste.txt`, contenant une liste d'adresses électroniques enregistrées ligne par ligne, et soit un fichier `lettre.txt` contenant un message à envoyer. Écrire la ligne de commande permettant d'envoyer ce message aux adresses de la liste.

Attention :
Il est fortement déconseillé de rediriger l'entrée standard ou la sortie standard d'un branchement, pour ne pas écrire des lignes de commande incompréhensibles et pas toujours interprétables par le shell. Par exemple, au lieu de rediriger l'entrée standard, dans la ligne de commande suivante :

```
grep 'e' | grep '2' < fich_entree.txt
```

pour laquelle le shell indique qu'il y a une ambiguïté sur la redirection en entrée, on conseille d'utiliser la commande `cat` de la manière suivante :

```
cat fich_entree.txt | grep 'e' | grep '2'
```

L'écriture suivante est également correcte :

```
( grep 'e' | grep '2' ) < fich_entree.txt
```

mais sa signification est légèrement différente de la ligne de commande précédente, puisque les commandes situées entre parenthèses sont lancées dans un « sous-shell »(cf. paragraphe 3.8.1).

Remarque : bien que syntaxiquement correcte, la séquence `...|cat|...` est toujours inutile.

3.3 Introduction aux expressions régulières

Les expressions régulières fournissent un moyen de définir des modèles de lignes de texte. Une expression régulière est une chaîne de caractères composée de caractères normaux et de caractères spéciaux, appelés « métacaractères des expressions régulières »(à ne pas confondre avec les métacaractères du shell, décrits au paragraphe 3.1). Une expression régulière désigne donc une séquence type de caractères (*pattern*) à laquelle une ligne de texte (dans un fichier par exemple) peut correspondre ou ne pas correspondre.

Elles sont par exemple utilisées pour extraire certaines lignes d'un fichier, pour extraire certains caractères de chaînes de caractères, pour effectuer des substitutions de texte, pour vérifier la validité syntaxique de chaînes de caractères (dans des formulaires sur des pages Web par exemple).

Même si elles ne sont pas spécifiques au shell de Bourne, les expressions régulières sont très utilisées pour l'écriture de scripts shell au travers de commandes comme `expr` (page 82), `sed` (page 74) et surtout `grep` (page 70) .

Comme toutes les chaînes de caractères, on peut écrire une expression régulière entre apostrophes `'`, entre guillemets `"`, ou seule. Sauf dans les cas de nécessité contraire, on conseille d'utiliser la notation entre apostrophes, car alors les caractères `$`, `*`, `\`, etc. ne seront pas interprétés comme des métacaractères du shell. Mais il arrive parfois que l'on souhaite que le shell interprète certains métacaractères avant de transmettre l'expression régulière à la commande qui, à son tour, interprétera les éventuels métacaractères. Dans ce cas, on utilisera des guillemets `"` ou aucun délimiteur.

Les principaux métacaractères des expressions régulières sont les suivants :

- `.` désigne un caractère quelconque.
- `*` désigne une répétition du caractère qui précède, de longueur quelconque éventuellement vide.
- `^` désigne le début d'une ligne.
- `$` désigne la fin d'une ligne.
- `\` protège le caractère suivant, qu'il soit caractère normal ou métacaractère des expressions régulières.
- `\{`*n*`\}` désigne la répétition de *n* fois le caractère précédent, qu'il soit caractère normal ou métacaractère des expressions régulières.
- `[`*liste_caractères*`]` désigne les caractères entre crochets, définis par énumération ou par intervalle.

✎ Exercice 11

Écrire les expressions régulières correspondant à :

- Une ligne quelconque.
- Une ligne vide.
- Une ligne commençant par `debut`.
- Une ligne contenant un caractère `*` suivi d'un caractère quelconque suivi d'un caractère `\`
- Une ligne commençant par une lettre majuscule.
- Une ligne contenant au moins dix lettres successives : la première doit être une majuscule et les neuf suivantes doivent être des minuscules.

3.4 Scripts

Un *script* est un ensemble de commandes Unix rassemblées dans un fichier dont le nom se termine souvent par l'extension `.sh`. Pour lancer l'exécution d'un script contenu dans le fichier `nom_script`, on peut :

- soit taper la commande (cf. la description de la commande `sh`, page 78) :
  `$ sh nom_script [paramètre_script ...]`
- soit le rendre exécutable (à ne faire qu'une seule fois, cf. la description de la commande `chmod`, page 65) :
  `$ chmod u+x nom_script`
  puis l'appeler directement :
  `$ nom_script [paramètre_script ...]`

Lorsqu'on lance l'exécution d'un script, on lance en fait un nouveau shell (on rappelle qu'un shell est un programme exécutable, dont le rôle est d'interpréter des commandes Unix), dit « shell fils », qui porte le nom du script, et c'est dans ce nouveau shell que les commandes du script seront interprétées. Le script, lui, est considéré comme une commande lancée dans le shell de départ (« shell père »).

Remarques :

- Dans un script, les commentaires apparaissent après un caractère `#` situé sur la même ligne.
- Le shell fils qui exécute le script est du même type (`sh`, `csh`, etc.) que celui depuis lequel on lance cette exécution. Si l'on souhaite que ce soit `sh` qui soit invoqué quel que soit le type du shell depuis lequel on lance l'exécution, il suffit de placer, en début de script, la ligne suivante :
  `#! /bin/sh`
  Le répertoire `/bin` n'existe pas : il s'agit d'un lien symbolique vers le répertoire `/usr/bin`, qui contient toutes les commandes externes (cf. paragraphe 5.2).

3.5 Paramètres

3.5.1 Paramètres positionnels d'un shell

Chaque shell dispose de dix *paramètres positionnels*, désignés par `$0`, `$1`, ..., `$8` et `$9`. Le paramètre positionnel `$0` a comme valeur *a priori* le nom du shell. Quant aux neuf autres paramètres positionnels, ils n'ont pas de valeur *a priori* (ils valent en fait la chaîne vide). Il y a deux façons de leur attribuer des valeurs :

- On peut utiliser la commande `set`, page 79.
- Dans le cas où on lance un script, les valeurs des paramètres du script sont affectées aux paramètres positionnels du nouveau shell qui est lancé. En particulier, le paramètre positionnel `$0` reçoit comme valeur le nom du script.

✎ **Exercice 12**
Écrire un script qui après avoir affiché à l'écran son nom, affiche le contenu des deux fichiers dont les noms lui ont été passés en paramètres.

3.5.2 Paramètres spéciaux

Tout shell possède également un certain nombre de paramètres autres que les paramètres positionnels, dont les noms commencent par le métacaractère `$`, et dont les valeurs dépendent exclusivement du contexte du shell :

- Le paramètre `$#` a pour valeur le nombre de paramètres.

- Les paramètres `$*` et `$@` ont pour valeur une chaîne de caractères composée par la liste des paramètres, séparés par des espaces.
  Mais `"$*"` est remplacé par un seul mot, alors que `"$@"` est remplacé par la liste des paramètres dans laquelle chaque paramètre est un mot différent. Une illustration de cette nuance lors de l'utilisation de la boucle `for` est fournie à la page 38.
- Le paramètre `$?` a pour valeur le code de retour de la dernière commande exécutée dans le shell. En particulier, si on lance un script, la valeur du paramètre `$?`, à la fin de l'exécution du script, est le code de retour de la dernière commande du script ayant été exécutée.
- Le paramètre `$$` a pour valeur le numéro du processus du shell.

Exemple : soit le script suivant contenu dans le fichier `exemple` :

```
#! /bin/sh
echo "\$0=$0, \$1=$1, \$2=$2, \$3=$3, \$4=$4, \$5=$5, \$6=$6, \$7=$7, \$8=$8, \$9=$9"
echo "\$#=$#, \$*=$*, \$?=$? \$\$=$$"
```

L'exécution de la commande `exemple toto titi tata` va produire l'affichage suivant :

```
$0=./exemple, $1=toto, $2=titi, $3=tata, $4=, $5=, $6=, $7=, $8=, $9=
$#=3, $*=toto titi tata, $?=0 $$=19681
```

✎ Exercice 13

Commencer par lire la description de la commande `grep`, page 70.

1. On suppose que l'on dispose d'un fichier `calepin.txt`, contenant des noms et des numéros de téléphone rangés selon le modèle suivant :

   ```
   DUPONT Jean 05.61.75.18.47
   MARTIN Yvonne 02.23.34.45.56
   ```

   Écrire le script `calepin.sh` qui recherche dans `calepin.txt` les personnes ayant comme nom le premier paramètre passé à ce script et dont le numéro de téléphone commence par les deux chiffres passés en deuxième paramètre, et qui affiche les informations relatives à ces personnes.
2. Écrire une ligne de commande permettant d'afficher les noms des fichiers du répertoire courant qui contiennent le mot `main` (dans le fichier, et non pas dans le nom du fichier).

3.6 Variables et expressions

Tout shell permet de gérer des variables. Le nom d'une variable peut comporter des lettres, des chiffres et le caractère `_` mais le premier caractère ne peut pas être un chiffre. La valeur d'une variable s'obtient en faisant précéder son nom du caractère `$`. Une variable d'un shell ne peut avoir comme valeur qu'une chaîne de caractères. Avec le shell de Bourne, l'affectation d'une valeur à une variable s'effectue soit à l'aide de la commande `read` (page 80), soit en utilisant la syntaxe suivante :

`variable`=`expression`

où `expression` est une expression du shell ayant comme valeur une chaîne de caractères.

Attention :

- Il ne doit pas y avoir d'espace ni avant ni après le caractère `=`
- La syntaxe de l'affectation d'une valeur à une variable dépend beaucoup du shell utilisé. En particulier, en « C shell », cette syntaxe est très différente de celle qui vient d'être décrite.

En shell, l'expression la plus générale est obtenue par la concaténation d'une ou de plusieurs expressions, qui peuvent être :

- une chaîne de caractères (exemple : `toto`),
- la valeur d'un paramètre positionnel (exemple : `$2`),
- la valeur d'une variable (exemple : `$var`),
- la capture de la sortie standard d'une commande (exemple : `` `pwd` ``).

La concaténation d'expressions s'écrit en juxtaposant ces expressions et en utilisant, si besoin est, les délimiteurs `"` et `'`, sachant que cela a une influence sur l'interprétation des métacaractères du shell.

✎ Exercice 14

Si le répertoire courant est la racine `/`, que produisent les séquences suivantes :

1. ```
   a=1
   b=$a`pwd`
   echo $b
   ```
2. ```
   a=1
   b=$a `pwd`
   echo $b
   ```
3. ```
   a=1
   b="$a `pwd`"
   echo $b
   ```
4. ```
   a=1
   b='$a `pwd`'
   echo $b
   ```

Remarques :

- Il existe, dans tout shell, un certain nombre de variables prédéfinies, comme la variable `PATH` de `sh` dont la valeur est une liste de répertoires.
- Il ne faut pas confondre les variables d'un shell avec ses paramètres positionnels. Pour modifier la valeur d'un paramètre positionnel, il faut utiliser la commande `set` (page 79).
- Lors de l'accès à la valeur d'une variable, des accolades autour de son identificateur sont nécessaires quand cet identificateur est suivi d'un caractère qui peut être interprété comme une partie de l'identificateur.

✍ 16

3.7 Structures de contrôle

3.7.1 Conditions

Les structures de contrôle font appel à des *conditions* qui sont des appels à une commande dont le code de retour est considéré. Si le code de retour de la commande vaut 0, la condition est considérée comme vraie. Sinon, elle est considérée comme fausse (convention inverse de celle du langage C).

Même si la commande utilisée comme condition peut être quelconque, c'est souvent la commande `test` qui est utilisée. Son rôle dans les structures de contrôle explique son importance dans l'écriture des scripts. La description de la commande `test` se trouve à la page 83.

3.7.2 Opérateurs de contrôle

Les opérateurs de contrôle `&&` et `||` indiquent respectivement une liste de commandes liées par un ET et une liste de commandes liées par un OU.

$$\underline{\texttt{commande}_1}\ \texttt{\&\&}\ \underline{\texttt{commande}_2}$$

La $\underline{\texttt{commande}_2}$ est exécutée si, et seulement si, la $\underline{\texttt{commande}_1}$ renvoie un code de retour nul.

commande$_1$ || commande$_2$

La commande$_2$ est exécutée si, et seulement si, la commande$_1$ renvoie un code de retour non nul.

La valeur de retour des listes ET et OU est celle de la dernière commande exécutée dans la liste.

Remarque : la commande `test -f nom -a -r nom` est équivalente à `test -f nom && test -r nom`

3.7.3 Choix

Choix

```
if condition1
then
    séquence_commandes1
elif condition2
then
    séquence_commandes2
else
    séquence_commandes3
fi
```

Choix multiple

```
case expression in
    cas1)
        séquence_commandes1
        ;;
    cas2|cas3)
        séquence_commandes2
        ;;
    *)
        séquence_commandes3
esac
```

Remarques :

- cas$_1$, cas$_2$ et cas$_3$ sont des chaînes de caractères, utilisant éventuellement les métacaractères du shell, ainsi que le caractère `|`, qui signifie OU.
- Le caractère `*`, en tant que métacaractère du shell, signifie « n'importe quelle chaîne de caractères ne commençant pas par `.` »(c'est donc l'équivalent du `default` dans le `switch` du langage C).
- La commande `break` (similaire à l'instruction `break;` du langage C) existe, mais elle n'est pas nécessaire dans la structure de contrôle d'aiguillage à choix multiple, car le double point-virgule `;;` est équivalent à cette commande.

✎ Exercice 15

Commencer par lire la description de la commande `exit`, page 77.

Lorsqu'on passe en paramètre à la commande `file` le nom d'un fichier « binaire exécutable », on obtient l'affichage d'une ligne comportant la chaîne de caractères `executable`. Écrire un script qui affiche `OK` si le fichier dont le nom est passé en paramètre est un binaire exécutable.

3.7.4 Répétitions

```
while condition
do
    séquence_commandes
done
```

```
until condition
do
    séquence_commandes
done
```

```
for variable in liste_cas
do
    séquence_commandes
done
```

Remarques :

- Dans la structure `for`, liste_cas est une liste de chaînes de caractères, utilisant éventuellement les métacaractères du shell.
- Dans les structures de contrôle de boucles, la commande `break` peut être utilisée.
- La syntaxe suivante :
  ```
  for i
  do
  ...
  ```

est équivalente à :
```
for i in "$@"
do
...
```
c'est-à-dire que la variable `i` prend successivement la valeur de chacun des paramètres positionnels ayant reçu une valeur.

✎ Exercice 16

Commencer par lire les descriptions des commandes `expr`, page 82, `head`, page 69, et `tail`, page 69.

Écrire un script `affiche.sh` qui lit soit un fichier dont le nom est passé en paramètre, soit le fichier `/users/linfg/linfg0/omar` si le script est lancé sans paramètre. Le fichier concerné sera lu et affiché ligne par ligne, de deux manières différentes :

- d'abord, en partant du début ;
- ensuite, en partant de la fin.

Voici un exemple d'exécution de ce script :

$ affiche.sh fich.txt
`Affichage en partant du début :`
<première ligne de fich.txt>
...
`Affichage en partant de la fin :`
<dernière ligne de fich.txt>
...
$

3.8 Sous-shell et shell fils

3.8.1 Sous-shell

Une séquence de commandes placée entre parenthèses est exécutée dans un « sous-shell »du shell courant. De la même manière, chaque commande d'un branchement est exécutée dans un sous-shell du shell courant. Dans un sous-shell, on peut accéder aux valeurs des variables et des paramètres du shell courant. On peut également modifier ces valeurs, mais les modifications apportées ne seront pas répercutées à la fin de l'exécution du sous-shell. De manière équivalente, la valeur du répertoire courant du shell courant peut être modifiée, sans que cette modification perdure à la fin de l'exécution du sous-shell. ✍ 17

3.8.2 Shell fils

Lorsqu'on lance l'exécution d'un script ou qu'on tape la commande `sh`, on lance en fait un nouveau shell qui est appelé un « shell fils »du shell courant, qui de ce fait est appelé « shell père ». Un shell fils diffère d'un sous-shell, dans la mesure où on ne peut pas accéder aux valeurs des variables et des paramètres du shell père. Pour une variable `x` du shell père, il est néanmoins possible de lancer la commande `export x` préalablement au lancement du shell fils, auquel cas la valeur de `x` sera connue dans le shell fils. En revanche, comme pour un sous-shell, les modifications éventuelles de la variable `x` ne seront pas répercutées dans le shell père (pas plus qu'une modification de la valeur du répertoire courant). ✍ 18

Remarque :
L'entrée et les sorties standard d'un shell fils peuvent être redirigées au moment de son lancement. Ceci redirige alors implicitement toutes les entrées et sorties standard des commandes lancées dans le shell fils. Ceci est donc vrai, en particulier, lors du lancement de l'exécution d'un script.

3.9 Double évaluation

Le shell permet d'effectuer une double évaluation grâce à la commande `eval`. Cette possibilité est utilisée pour lancer une commande dont le texte n'est construit qu'au moment de l'exécution. C'est par exemple le cas lorsqu'on veut manipuler des variables dont le nom ne sera connu que lors de l'exécution du script.

`eval` [paramètre ...]

Cette commande procède en deux temps :

- Dans un premier temps, la commande est remplacée par la liste de ses paramètres, avec interprétation des métacaractères du shell.
- Dans un deuxième temps, cette liste de paramètres est considérée comme une commande Unix, et exécutée en tant que telle, donc en particulier avec, à nouveau, interprétation des métacaractères du shell.

Exemple : la séquence de commandes :

```
a=1
c='$a'
eval echo $c
```

✍ 19 produit l'affichage de la valeur 1.

✎ Exercice 17

Écrire un script, de nom `variables.sh`, qui reçoit en paramètres une chaîne de caractères et un entier `n`. Ce script doit initialiser des variables, dont le nom est composé de la chaîne de caractères, suivie d'une valeur entière comprise entre `1` et `n`. La valeur affectée à chaque variable sera celle utilisée pour construire son nom. Par exemple, si on lance la commande `variables.sh var 4`, le script devra effectuer les initialisations suivantes :

```
var1=1
var2=2
var3=3
var4=4
```

Remarque : cet exercice ne peut pas être fait sans l'aide de la commande `eval`.

3.10 Éléments pour l'écriture des scripts

3.10.1 Grandes étapes d'un script

Un script est généralement constitué des parties suivantes :

- définitions éventuelles de fonctions ;
- vérification du nombre de paramètres ;
- vérification de la validité des paramètres ;
- traitement.

3.10.2 Fonctions

Dans un script, il est possible de définir et d'utiliser des fonctions. Une fonction ressemble à un script (la syntaxe d'appel est la même), à l'exception :

- de sa définition qui est faite au début d'un script selon la syntaxe suivante :

```
nom_fonction()
{
    séquence_commandes
}
```

- du paramètre `$0` qui vaut, à l'intérieur de la fonction, le nom du shell courant, donc du script, et pas le nom de la fonction ;
- de son exécution qui est réalisée dans le shell courant, c'est-à-dire ni dans un sous-shell, ni dans un shell fils (les variables du shell courant sont donc accessibles à l'intérieur de la fonction) ;
- de la sortie de la fonction et du retour d'un code qui sont réalisés grâce à la commande `return` :

```
return code
```

✍ 20

3.10.3 Gestion des erreurs

- ▶ Généralement, quand le nombre de paramètres est incorrect ou quand un paramètre n'est pas valide :
  - on affiche sur `stderr` un message d'erreur ;
    Exemple : `echo "nombre de paramètres incorrect" >&2`
  - on affiche sur `stderr` une description de la syntaxe d'appel du script ;
    Exemple : `echo "Usage : $0 fichier [...]" >&2`
  - on quitte le script en retournant un code strictement positif.
    Exemple : `exit 1`
- ▶ Durant le traitement, une erreur peut survenir ; on affiche alors un message d'erreur sur `stderr` et, selon les cas, on poursuit le traitement ou on sort du script en retournant un code d'erreur.

3.10.4 Vérification du nombre de paramètres

On utilise la commande `test` (page 83) avec les opérateurs de comparaison `-eq`, `-ne`, `-gt`, `-lt`, `-ge`, `-le` et le paramètre `$#`.
Exemple :

```
if [ $# -ne 1 ]
then
  echo "Nombre de paramètre sincorrect" >&2
  echo "Usage : $0 fichier" >&2
  exit 1
fi
```

3.10.5 Vérification de la validité des paramètres

3.10.5.1 Pour un nom de fichier

- fichier à lire :
  - existence (`test -f`)
  - droit de lecture (`test -r`)
- fichier à modifier :

- existence (`test -f`)
- droit de lecture (`test -r`)
- droit d'écriture (`test -w`)

3.10.5.2 Pour un nom de répertoire

- répertoire à consulter :
  - existence (`test -d`)
  - droit de lecture (`test -r`)
  - droit d'exécution (`test -x`)
- répertoire à modifier :
  - existence (`test -d`)
  - droit de lecture (`test -r`)
  - droit d'exécution (`test -x`)
  - droit d'écriture (`test -w`)

3.10.5.3 Pour un entier

Une solution consiste à effectuer une opération arithmétique avec la commande `expr` (page 82) et à vérifier que son code de retour est strictement inférieur à 2.
Exemple :

```
expr $1 + 0 > /dev/null 2> /dev/null
if [ $? -ge 2 ]
then
  echo "Erreur : $1 n'est pas un entier" >&2
  echo "Usage : $0 entier" >&2
  exit 2
fi
```

✎ Exercice 18

Commencer par lire les descriptions des commandes `cmp`, page 73, et `rm`, page 62. Écrire un script qui supprime toutes les copies d'un fichier dont le nom est passé en premier paramètre, se trouvant dans un répertoire dont le nom est passé en deuxième paramètre.
Remarques :

- Ce script ne doit pas être récursif.
- On supposera que les noms du fichier et du répertoire qui sont passés en paramètres sont écrits de manière relative.
- Le répertoire qui est passé en paramètre peut être le répertoire courant.
- Si le fichier passé en paramètre est contenu dans le répertoire passé en paramètre, alors il ne faut pas le supprimer.

3.10.6 Schémas classiques

3.10.6.1 Affichage d'un texte à l'écran

Il est possible d'utiliser la commande `cat` (page 67) à la place de la commande `echo` (page 81).

Exemple :

```
cat << FIN
Ceci est un texte
écrit sur plusieurs lignes
que je veux afficher à l'écran.
FIN
```

3.10.6.2 Ajout d'un texte à la fin d'un fichier

De la même manière, on peut utiliser la commande `cat` (page 67) et une redirection.
Exemple :

```
cat >> fichier.txt << FIN
Ceci est un texte
écrit sur plusieurs lignes
que je veux ajouter à la fin
du fichier fichier.txt.
FIN
```

3.10.6.3 Parcours d'un fichier ligne par ligne

Soit le fichier `fichier.txt` contenant :

```
Première ligne
Deuxième ligne
Troisième ligne
```

La séquence classique suivante :

```
cat fichier.txt | while read ligne
do
  # Traitement de $ligne
  echo "\$ligne=$ligne"
done
```

produit l'affichage de :

```
$ligne=Première ligne
$ligne=Deuxième ligne
$ligne=Troisième ligne
```

Attention :
Les commandes exécutées dans un branchement sont exécutées dans des sous-shells. Donc, toute modification des valeurs de variables effectuée entre `do` et `done` ne seront pas connues après `done`. Il faut donc que tout le travail soit fait entre `do` et `done` ou bien il faut utiliser un fichier temporaire.
Exemple : la séquence suivante :

```
n=0
cat fichier.txt | while read ligne
do
  n=‘expr $n + 1‘
done
echo $n
```

produit l'affichage de `0` quel que soit le contenu de `fichier.txt` !

Mais il est possible d'éviter la création d'un sous-shell en effectuant une redirection de l'entrée standard de la commande `while` vers le fichier à lire. Par exemple, la séquence suivante :

```
n=0
while read ligne
do
  n=‘expr $n + 1‘
done < fichier.txt
echo $n
```

produit l'affichage de 3.

3.10.6.4 Parcours d'une chaîne de caractères mot par mot

Utilisation de la boucle `for`

Exemple :

Séquence	**Affichage produit**

`phrase="Bientôt les vacances"`
`for mot in $phrase`
`do`
`  # Traitement de $mot`
`  echo "\$mot=$mot"`
`done`
	`$mot=Bientôt`
	`$mot=les`
	`$mot=vacances`

`phrase="Bientôt les vacances"`
`for mot in "$phrase"`
`do`
`  # Traitement de $mot`
`  echo "\$mot=$mot"`
`done`
	`$mot=Bientôt les vacances`

`for mot in Bientôt "les vacances"`
`do`
`  # Traitement de $mot`
`  echo "\$mot=$mot"`
`done`
	`$mot=Bientôt`
	`$mot=les vacances`

Traitement des paramètres passés au script

Pour accéder à chacun des paramètres passés à un script, deux syntaxes sont possible :

- **Syntaxe 1 :** avec une boucle `for`

  ```
  for param in $*
  do
  ```

```
  # Traitement de $param
  echo "Paramètre courant=$param"
done
```

- **Syntaxe 2 :** avec une boucle `while` et la commande `shift` (page 80)

```
while [ $1 ]
do
  # Traitement de $1
  echo "Paramètre courant=$1"
  shift
done
```

Si l'on suppose que le script s'appelle `script.sh`, la commande :
`script.sh p1 p2 p3`
provoque l'affichage de :

```
Paramètre courant=p1
Paramètre courant=p2
Paramètre courant=p3
```

Attention :
Dans le cas particulier où l'un des paramètres peut contenir le caractère espace ces deux syntaxes doivent être modifiées. En effet, si on lance :
`script.sh ab "cd ef"`
alors :

- **avec la syntaxe 1**

en utilisant	on obtient

`$*`	`Paramètre courant=ab`
	`Paramètre courant=cd`
	`Paramètre courant=ef`

`"$*"`	`Paramètre courant=ab cd ef`

`$@`	`Paramètre courant=ab`
	`Paramètre courant=cd`
	`Paramètre courant=ef`

`"$@"`	`Paramètre courant=ab` `Paramètre courant=cd ef`

De manière générale, `$*` et `$@` sont remplacés par les valeurs des paramètres positionnels. Mais `"$*"` est remplacé par un seul mot, c'est-à-dire par `"$1 $2 ..."`, alors qu'avec `"$@"`, chaque paramètre est remplacé par un mot séparé, c'est-à-dire que `"$@"` est remplacé par `"$1" "$2" ...`
Il est donc conseillé d'utiliser la syntaxe suivante :

```
for param
do
...
```

qui est équivalente à :

```
for param in "$@"
```

```
do
...
```

- **avec la syntaxe 2**, on obtient :

```
Paramètre courant=ab
script.sh: test: argument expected
```

car `$1` vaut :

```
ab
```

au premier tour de boucle, mais :

```
cd ef
```

au second, ce qui revient à écrire au niveau de la condition du `while` :

```
[ cd ef ]
```

qui est incorrect. Il faut donc remplacer :

```
[ $1 ]
```

par :

```
[ "$1" ]
```

qui donne bien :

```
Paramètre courant=ab
Paramètre courant=cd ef
```

Traitement des fichiers du répertoire courant

Le traitement des fichiers contenus dans le répertoire courant peut se faire en utilisant le modèle suivant :

```
for fich in *
do
  if [ -f "$fich" ]
  then
    # Traitement de $fich
    echo "\$fich=$fich"
  fi
done
```

Pour traiter les répertoires, il suffit de remplacer `-f` par `-d`.
Attention :
Il est également possible de remplacer `*` par `` `ls` ``. Mais cela ne fonctionne pas avec des noms de fichiers contenant le caractère espace et `"`ls`"` ne fonctionne pas non plus car, dans ce cas, les sauts de ligne (`\n`) sont conservés.

Traitement d'une arborescence

On suppose ici que le nom du répertoire racine de l'arborescence à traiter est passé en premier paramètre du script. Dans ce cas, il est possible d'utiliser le modèle suivant :

```
for fichourep in "$1"/*
do
  if [ -d "$fichourep" ]
  then
    # Appel récursif permettant de traiter le répertoire $fichourep
    "$0" "$fichourep"
  else
```

```
    # Traitement du fichier $fichourep
    echo "Fichier courant : $fichourep"
  fi
done
```


Attention :
Il est également possible de remplacer `"$1"/*` par `` `ls $1` ``. Mais cela ne fonctionne pas avec des noms de fichiers ou de répertoire contenant le caractère espace. En outre, même si l'on suppose que ce n'est pas le cas, il faut également remplacer `$fichourep"` par `$1/$fichourep` afin de construire une désignation correcte.
Remarques :

- Dans les modèles précédents, la vérification des droits des fichiers et des répertoires a été omise. Dans un script complet, elle serait nécessaire.
- Lors du traitement d'une arborescence, on ne se déplace pas. En d'autres termes, le répertoire courant reste inchangé. L'accès aux différent éléments de l'arborescence se fait grâce à l'obtention de leur désignation relative à la racine de l'arborescence.

3.10.6.5 Construction incrémentale d'une chaîne de caractères

Il est souvent utile de construire une chaîne de caractère en la complétant petit à petit de manière itérative. Pour cela, il est possible de s'inspirer de l'exemple suivant :

```
chaine=""
for i in 1 2 3 4
do
  chaine="$chaine $i"
  echo "\$chaine=$chaine"
done
echo "Résultat=$chaine"
```

qui produit l'afffichage suivant :

```
$chaine= 1
$chaine= 1 2
$chaine= 1 2 3
$chaine= 1 2 3 4
Résultat= 1 2 3 4
```

✎ Exercice 19

Écrire le script `afficher_chemins.sh` permettant d'afficher les chemins d'accès de tous les fichiers d'un répertoire donné (à quelque niveau de la sous-arborescence que ce soit).
Remarque :
Ce script est nécessairement récursif. Le nom du script ne doit pas apparaître explicitement dans un script récursif, car il ne serait pas mis à jour si on renommait le fichier contenant le script. Au lieu de cela, il faut utiliser le paramètre positionnel `$0`.

Chapitre 4
Sujets des travaux pratiques

SÉANCE 1

Consignes valables pour l'ensemble des TP

Moodle

Toutes les informations relatives à l'UE « KINUPI41 Systèmes »peuvent être consultées sur moodle. Il est demandé de consulter ce moodle régulièrement.

Choix du shell

Ouvrir un terminal, le shell lancé à la connexion devrait être un `bash`.

On commencera donc, lors de cette première séance de travaux pratiques, par lancer un shell de Bourne, en tapant la commande `sh` dans la fenêtre racine. Le nouveau prompt sera alors un caractère `$`. Par rapport aux autres shells, le shell de Bourne présente deux inconvénients : il ne possède ni historique de commandes (à l'aide des quatre touches du clavier correspondant aux flèches de déplacement), ni complétion automatique des noms de fichiers et d'exécutables (à l'aide de la touche de tabulation). Pour cette raison, on ne le lancera explicitement que lors de la première séance de TDM. Pour les autres séances, on profitera de l'historique de commandes et de la complétion de `bash` ou d'un autre shell, et on verra comment faire quand même interpréter les commandes par un shell de Bourne, qui sera donc lancé de manière implicite.

Si vous utilisez l'interpréteur de commandes `bash`, ce dernier présente quelques différences avec le shell de Bourne. En particulier, pour que la commande `echo` interprète les caractères spéciaux, il faut utiliser l'option `-e` qui n'existe pas avec le shell de Bourne.

👁 **Commandes utiles pour cette séance :** `cat` (page 67), `cd` (page 64), `chmod` (page 65), `cp` (page 61), `echo` (page 81), `grep` (page 70), `head` (page 69), `ls` (page 60), `mkdir` (page 63), `more` (page 68), `mv` (page 61), `pwd` (page 64), `rm` (page 62), `rmdir` (page 63), `sort` (page 71), `tail` (page 69).

Exercice 1

1. Télécharger et extraire le fichier `users.tar` dans le répertoire principal. Déplacez vous dans le répertoire principal. En utilisant la commande `ls`, afficher la liste des fichiers du répertoire `users/linfg/linfg0/S3` dont le nom se termine par `.txt`
2. Afficher intégralement les trois chaînes de caractères suivantes, à l'aide de trois appels à la commande `echo` :

```
Le renard dit "Oh, Monsieur du Corbeau"
J'ai gagne $100 a l'Alcazar
Commentaire : \* Initialisation *\
```

Remarques :

- Le métacaractère \ annihile l'évaluation du caractère immédiatement suivant, ce qui n'est intéressant que dans le cas où ce caractère est un métacaractère du shell.
- Si un délimiteur ', " ou ` est « célibataire »sur sa ligne de commande, alors l'interpréteur de commandes affiche un prompt > qui invite l'utilisateur à terminer la commande en cours de frappe. La commande n'est réellement interprétée que lorsque les délimiteurs forment des paires.
- En cas de problème, l'exécution d'une commande peut être interrompue en tapant la combinaison de touches <*Ctrl*>C

Exercice 2

Taper les commandes nécessaires à la réalisation de chacune des tâches suivantes :

1. Se déplacer dans le répertoire principal, et afficher le nom du répertoire courant.
2. Créer le répertoire `REP`, et son sous-répertoire `SOUSREP`, en une seule ligne de commande.
3. Copier le fichier `users/linfg/linfg0/S3/liste.txt` du répertoire principal dans `SOUSREP`
4. Se positionner dans le répertoire `SOUSREP`, et en afficher le contenu.
5. Déplacer le fichier `liste.txt` de `SOUSREP` vers `REP`
6. Supprimer le répertoire `REP`, ainsi que son contenu.

Exercice 3

Effectuer les tâches suivantes :

1. Copier le fichier `users/linfg/linfg0/S3/liste.txt` dans le répertoire principal, sous le même nom, sans se déplacer.
2. Consulter le contenu de cette copie, sans utiliser d'éditeur de texte (utiliser la commande `more`).
3. Contrôler les droits d'accès au fichier `liste.txt`, en tapant `ls -l`
4. S'en interdire l'accès en écriture.
5. Contrôler la modification des droits d'accès effectuée.
6. Essayer de changer un prénom dans le fichier `liste.txt`, à l'aide de l'éditeur de texte `nedit` ou `gedit` ou `kate`. Sauvegarder. Quitter `nedit` ou `gedit` ou `kate`.
7. Rétablir l'accès en écriture au fichier `liste.txt` pour le propriétaire.
8. Essayer à nouveau d'effectuer la modification précédente, à l'aide de `nedit` ou `gedit` ou `kate` . Sauvegarder.

Exercice 4

La commande `head` permet de récupérer les premières lignes d'un fichier. La commande `tail` permet de récupérer les lignes situées à la fin d'un fichier. Après avoir lancé la commande `man` pour chacune de ces commandes, essayer de trouver *a priori* ce que font les quatre commandes suivantes, puis les taper pour vérifier :

```
tail -n -5 liste.txt
tail -n +5 liste.txt
head -5 liste.txt
head liste.txt
```

Exercice 5

Les lignes du fichier `liste.txt` sont structurées sous la forme suivante (deux champs successifs sont séparés par un caractère espace) :

```
EMMANUEL-EMILE CYRIL linfg151
EMRITTE MEHDI linfg22
```

1. Sachant que tout étudiant a un login du type `linfg<numéro>`, trier le fichier `liste.txt` de manière à afficher la liste des étudiants par ordre de numéros croissants. Pour éviter que le numéro `2` n'apparaisse après le numéro `19`, on conseille de lancer plusieurs commandes successivement, en utilisant l'algorithme suivant :
   - Trier les étudiants dont le numéro ne comporte qu'un chiffre et écrire le résultat dans un fichier `temp`
   - Trier les étudiants dont le numéro comporte deux chiffres exactement, et les rajouter au fichier précédent.
   - Trier les étudiants dont le numéro comporte trois chiffres exactement, et les rajouter au fichier précédent.
   - Afficher le contenu de `temp`, puis détruire ce fichier.
2. Trier ce fichier de manière à afficher la liste des étudiants par ordre alphabétique des prénoms.

Exercice 6

1. Afficher la liste des étudiants ayant `LAURENT` comme prénom.
2. Afficher la liste des étudiants ayant un numéro inférieur à `100` (on aura intérêt à utiliser la commande `grep` avec l'option `-v`).
3. Afficher la liste des étudiants ayant un numéro contenant un chiffre `1` au moins.

Exercice 7

1. À l'aide uniquement de la commande `echo`, et sans utilisation du caractère `\n`, créer le fichier `tdm01.txt` contenant les lignes suivantes :

   ```
   Est-ce que j'arrive
   a rediriger la
   sortie standard ?
   ```

2. Déterminer *a priori* quel doit être le résultat de chacune des quatre commandes ci-dessous, puis contrôler :
   - `cat liste.txt`
   - `cat liste.txt > liste2.txt`
   - `rm liste2.txt`
   - `cat liste.txt | tail -5`
3. Écrire une commande permettant d'afficher les lignes `5` à `10` du fichier `liste.txt`

SÉANCE 2

Commandes utiles pour cette séance : `cp` (page 61), `cut` (page 72), `echo` (page 81), `grep` (page 70), `head` (page 69), `ls` (page 60), `mkdir` (page 63), `rm` (page 62), `rmdir` (page 63), `sort` (page 71), `tail` (page 69), `tr` (page 74), `wc` (page 72).

Exercice 8

Lancer un éditeur de texte, par exemple `nedit` (en tapant la commande `nedit &`). Écrire les scripts permettant de réaliser les trois séquences de tâches suivantes, puis tester ces scripts :

1. `script1.sh`
   - Créer, dans le répertoire courant, le répertoire dont le nom est passé en paramètre.
   - Recopier un fichier du répertoire courant dans le répertoire créé. Le nom du fichier sera passé en paramètre également.
2. `script2.sh`
   - Effacer les fichiers contenus dans un répertoire passé en paramètre.
   - Supprimer ce répertoire.

   On appellera ce script sur des répertoires créés à l'aide de `script1.sh`
3. `script3.sh`
   - Créer un répertoire temporaire `REP`
   - Créer dans ce répertoire le fichier `liste.txt`, contenant la liste des noms des fichiers du répertoire courant dont le nom se termine par l'extension `.txt` ou `.sh`
   - Recopier le fichier correspondant au dernier nom de cette liste dans `REP`
   - Effacer le fichier `liste.txt`

   Vérifier que ce script fonctionne bien, en contrôlant l'existence du répertoire `REP` et d'un fichier dans ce répertoire, dont le nom comporte l'extension `.txt` ou `.sh`

Exercice 9

Écrire un script permettant de dénombrer les fichiers du répertoire courant (les sous-répertoires du répertoire courant ne doivent donc pas être comptés) dans les quatre catégories suivantes :

- Fichiers (cachés ou non) dont le nom se termine par `.txt`
- Fichiers (cachés ou non) dont le nom se termine par `.sh`
- Fichiers cachés.
- Autres fichiers.

Chaque résultat devra être affiché sous la forme suivante :

```
Il y a 2 fichiers dont le nom se termine par .txt
```

Exercice 10

On souhaite effectuer quelques manipulations sur le fichier `liste.txt` qui se trouve dans le répertoire `/users/linfg/linfg0/S3/`. Ce fichier avait déjà été utilisé lors de la première séance. Faire une copie de ce fichier dans le répertoire principal et répondre aux questions suivantes :

1. Afficher le nom (et uniquement le nom) de l'étudiant de `liste.txt` qui a le plus petit numéro.
2. Afficher le nom (et uniquement le nom) de l'étudiant de `liste.txt` qui a le plus grand numéro.
3. Écrire un script permettant d'afficher le nombre d'étudiants de `liste.txt` dont le nom comporte un nombre de lettres supérieur ou égal à la première valeur et strictement inférieur à la deuxième valeur passées en paramètres.

SÉANCE 3

Commandes utiles pour cette séance : `cat` (page 67), `cd` (page 64), `cut` (page 72), `date` (page 85), `echo` (page 81), `exit` (page 77), `expr` (page 82), `grep` (page 70), `ls` (page 60), `test` (page 83), `date` (page 85), `pwd` (page 64), `read` (page 80), `rm` (page 62), `set` (page 79), `test` (page 83).

Exercice 11

Écrire un script permettant de convertir en français la date affichée par la commande « `date -R` », sous le format suivant :

```
Lundi 16 Decembre 1996 15:31:37
```

Pour l'écriture de ce script, il est conseillé d'utiliser un « aiguillage à choix multiple »(cf. paragraphe 3.7.3).

Exercice 12

Écrire un script qui effectue la traduction mot à mot d'un texte anglais en français. Le fichier `f_anglais.txt` contient le texte écrit en anglais. Le fichier `f_français.txt` contiendra la traduction. On dispose d'un fichier `dico.txt`, dont chaque ligne contient un mot en anglais, suivi d'un espace, suivi de la traduction de ce mot en français. On suppose que `dico.txt` permet la traduction de l'ensemble du vocabulaire utilisé dans `f_anglais.txt`

Exemple :
Si le fichier `f_anglais.txt` contient la ligne :

```
my tailor is rich
```

et le fichier `dico.txt` contient les lignes :

```
is est
my mon
rich riche
tailor tailleur
```

alors la ligne suivante doit être écrite dans le fichier `f_français.txt` :

```
mon tailleur est riche
```

Remarque :
Un exemple de fichier `f_anglais.txt` et un fichier `dico.txt` suffisant pour le traduire peuvent être récupérés en tapant :

```
cp /users/linfg/linfg0/S3/f_anglais.txt .
cp /users/linfg/linfg0/S3/dico.txt .
```

Exercice 13

Écrire un script interactif, permettant de naviguer dans une arborescence de fichiers et de répertoires, selon les spécifications suivantes :

- Le script affiche le répertoire courant et la liste numérotée des sous-répertoires. Les fichiers du répertoire courant sont ignorés.
- L'utilisateur tape le numéro du sous-répertoire dans lequel il souhaite se déplacer, avec les cas particuliers suivants :
  - S'il tape `0` : il reste dans le répertoire courant.
  - S'il tape `-1` : il monte d'un répertoire.
  - S'il tape `-2` : il termine l'exécution du script.

SÉANCE 4

Commandes utiles pour cette séance : `cat` (page 67), `cut` (page 72), `cp` (page 61), `echo` (page 81), `eval` (page 78), `exit` (page 77), `expr` (page 82), `grep` (page 70), `head` (page 69), `pwd` (page 64), `read` (page 80), `return` (page 35), `rm` (page 62), `shift` (page 80), `tail` (page 69), `tee` (page 68), `test` (page 83), `tr` (page 74), `wc` (page 72).

Exercice 14

Écrire un script permettant d'afficher à l'écran le contenu d'une série de fichiers dont les noms sont passés en paramètres, et n'utilisant pas de boucle `for`. Le message `<nom fichier> inaccessible` sera affiché si le fichier `<nom fichier>` est inexistant ou ne peut être lu.

L'écriture de ce script nécessite l'utilisation de la commande `shift` et d'une boucle `while`. Il n'est pas récursif.

Exercice 15

On se propose d'écrire un script `inverse.sh` qui réécrit un fichier en inversant l'ordre de ses lignes. Ce script reçoit en paramètre le nom du fichier à inverser. Ce paramètre peut être éventuellement suivi du nom d'un autre fichier. Dans ce cas, l'inversion du fichier initial sera effectuée dans le second fichier (le fichier initial n'étant pas modifié). On pourra suivre, étape par étape, les indications suivantes pour parvenir à l'écriture finale du script :

1. Écrire le script `inverse.sh` qui teste si le nombre de paramètres reçus est bien égal à `1` ou à `2`. Dans le cas contraire, afficher le message suivant :

   ```
   Erreur : inverse.sh nom_fichier1 [nom_fichier2]
   ```

   Produire un code de retour en rapport avec un appel correct ou erroné de la commande.

2. Si le nombre de paramètres est égal à `2`, recopier le fichier correspondant au premier paramètre dans le second et relancer le script `inverse.sh`, avec seulement le second paramètre en paramètre (appel récursif). Afin de tester le bon fonctionnement du script, on affichera le message `Un seul paramètre`, dans le cas où le script n'est lancé qu'avec un seul paramètre. Dans le cas d'un appel avec deux paramètres, ce message ne devra apparaître qu'une fois (après l'appel récursif).

3. Dans le cas où il n'y a qu'un seul nom de fichier en paramètre, récupérer dans la variable `nblignes` le nombre de lignes de ce fichier. Tester le script, en affichant la valeur de `nblignes`.

4. Mettre en place la boucle `while` qui permettra de parcourir le fichier. Pour la tester, on affichera la valeur décrémentée de `nblignes` à chaque parcours de boucle.

5. À chaque parcours de boucle, envoyer les `nblignes` premières lignes du fichier passé en paramètre dans le fichier temporaire `.inv1`, puis concaténer la dernière ligne du fichier `.inv1` à `.inv2`. Vérifier, en consultant le contenu de `.inv2` après exécution du script. Contrôler qu'il ne manque pas de lignes (ni la première ni la dernière). Effacer `.inv2` après chaque test.

6. Recopier `.inv2` dans le fichier passé en paramètre. Effacer les fichiers `.inv1` et `.inv2` Tester le script avec un, puis deux fichiers passés en paramètres.

Exercice 16

Écrire un script réalisant les tâches suivantes :

1. Affichage du menu suivant, en boucle :

```
Vous pouvez taper une commande Unix, ou bien :
fin si vous souhaitez quitter l'application
relancer n si vous souhaitez relancer les n dernières commandes
Votre choix :
```

2. Dans le cas où l'utilisateur tape une commande Unix, lecture de la commande tapée, exécution de cette commande, et affichage à l'écran des résultats produits (si la commande n'est pas tapée correctement, le « résultat »est le message d'erreur).
3. Dans le cas où l'utilisateur tape par exemple `relancer 3`, nouvelle exécution des `3` dernières commandes exécutées et affichage à l'écran de ces commandes et des résultats produits correspondants. Les commandes déjà exécutées seront écrites (en tant que chaînes de caractères) dans le fichier `temp`, de la manière décrite dans l'étape suivante :
4. Écriture, à la fin de `temp`, de la commande exécutée (ou des commandes exécutées, si l'utilisateur a tapé par exemple `relancer 3`), ainsi que des résultats produits, sous la forme :

```
$ <commande n>
<résultats produits par la commande n>
$ <commande n+1>
<résultats produits par la commande n+1>
...
```

5. Enfin, retour au début.

On pourra suivre les indications suivantes :

1. Écrire la fonction `affichage_menu`, qui affiche le menu décrit plus haut.
2. Structurer le script à l'aide d'une boucle `until`.
3. Dans le cas où l'utilisateur tape par exemple `relancer 3`, on écrira dans `temp` les `3` dernières commandes qui y ont été déjà écrites, ainsi que les résultats produits, mais on n'écrira pas dans ce fichier la commande `relancer 3` (c'est-à-dire que `temp` ne doit contenir que des commandes Unix).
4. Écrire la fonction `relancer_n` qui, dans le cas où l'utilisateur tape par exemple `relancer 3`, relance les `3` dernières commandes déjà tapées, réalise les affichages correspondants à l'écran, et met le fichier `temp` à jour.

Remarques :

- On supposera qu'aucune ligne de résultats ne commence par le caractère `$` suivi d'un espace.
- On n'oubliera pas de détruire le fichier `temp` à la fin de l'exécution du script.

SÉANCE 5

Commandes utiles pour cette séance : `basename` (page 67), `cut` (page 72), `echo` (page 81), `exit` (page 77), `expr` (page 82), `test` (page 83), `tr` (page 74).

Exercice 17

Écrire le script `chemins.sh` permettant d'afficher les chemins d'accès aux fichiers contenant une chaîne de caractères donnée, dans un répertoire donné (à quelque niveau de la sous-arborescence que ce soit).
Remarque :
Ce script est nécessairement récursif. Le nom du script ne doit pas apparaître explicitement dans un script récursif, car il ne serait pas mis à jour si on renommait le fichier contenant le script. Au lieu de cela, il faut utiliser le paramètre positionnel `$0`.

Exercice 18

Commencer par lire les descriptions des commandes `cut`, page 72, et `tr`, page 74. Écrire le script `taille.sh` qui calcule la taille moyenne (en nombre d'octets) des fichiers contenus dans un répertoire passé en paramètre, ou dans le répertoire courant si aucun paramètre n'est passé. On prendra garde à ne pas diviser par `0`.
Rappel : la commande `ls -l nom_rep` affiche les informations complètes relatives aux fichiers et aux sous-répertoires contenus dans le répertoire `nom_rep`, sous la forme de neuf champs par ligne, séparés deux à deux par une séquence d'espaces de longueur variable, dans l'ordre suivant : droits d'accès, nombre de liens, nom du propriétaire, nom du groupe du propriétaire, taille en nombre d'octets, date de dernière modification (sur trois champs), nom.

Exercice 19

Écrire le script `rep.sh` qui donne diverses indications sur le contenu d'un répertoire. Appelé sans paramètre, ce script affiche la taille cumulée des fichiers contenus dans le répertoire courant. Appelé avec l'option `-r`, il affiche le nombre de sous-répertoires contenus dans le répertoire courant. Appelé avec l'option `-f`, il affiche le nombre de fichiers contenus dans le répertoire courant. Appelé avec un nom de répertoire passé en paramètre, éventuellement après une des deux options déjà mentionnées, les valeurs affichées concernent le contenu de ce répertoire. Ce script ne doit être lancé qu'avec une seule option au plus. On pourra suivre, étape par étape, les indications suivantes pour le réaliser :

1. Écrire la fonction `sortie` qui affiche le message d'erreur suivant sur `stderr` :

   ```
   Usage : rep.sh [-r|-f] [nom_répertoire]
   ```

   et qui termine l'exécution du script en renvoyant le code qui lui a été passé en paramètre.
2. Tester si le nombre de paramètres est compris entre `0` et `2` et sinon, appeler la fonction `sortie` avec le code `1`.
3. Si le nombre de paramètres est égal à `2`, vérifier que le premier paramètre commence bien par un caractère `-` et que le deuxième paramètre ne commence pas par un caractère `-` (on suppose qu'un nom de répertoire ne peut commencer par un tel caractère), et sinon appeler la fonction `sortie` avec le code `2`.
4. Créer les deux variables suivantes : `option` prend comme valeur la chaîne de caractères correspondant à l'option (caractère `-` compris) si une option a été passée en paramètre, ou la chaîne `aucune` sinon ; `rep` prend comme valeur la chaîne de caractères correspondant au nom de répertoire qui a été passé en paramètre, ou `.` sinon.
5. Si la valeur de `rep` n'est pas un nom de répertoire accessible, alors appeler la fonction `sortie` avec le code `3`.

6. Afficher l'information correspondant à la valeur de `option`. Si la valeur de `option` n'est ni `aucune`, ni `-r`, ni `-f`, alors : appeler la fonction `sortie` avec le code `4` si cette valeur est la chaîne `-` et avec le code `5` sinon.

SÉANCE 6

Commandes utiles pour cette séance : `basename` (page 67), `echo` (page 81), `exit` (page 77), `expr` (page 82), `test` (page 83).

Exercice 20

Écrire le script `arbre_base.sh` permettant d'afficher tous les sous-répertoires d'un répertoire donné en paramètre (quel que soit le niveau des sous-répertoires dans l'arborescence). Ce script de base devra afficher le résultat sous la forme suivante :

```
$ arbre_base.sh TEST
FILS1
S_FILS11
S_FILS12
SS_FILS121
S_FILS13
SS_FILS131
SS_FILS132
FILS2
S_FILS21
FILS3
S_FILS31
$
```

Remarques :

- Pour récupérer cette arborescence de test, taper :
  `cp -r /users/linfg/linfg0/S3/TEST .`
- Les fichiers de la sous-arborescence ne doivent pas être affichés par ce script (dans l'arborescence à récupérer, le répertoire `FILS3` contient un fichier `toto` qui est un frère de `S_FILS31`).

Exercice 21

Attention : cet exercice est une version un peu plus compliquée de l'exercice précédent. Il est donc conseillé de commencer par terminer l'exercice 1.

Écrire le script `arbre.sh` permettant de dessiner à l'écran la sous-arborescence d'un répertoire passé en paramètre, et ce sous la forme suivante :

```
$ arbre.sh TEST
TEST
+- FILS1
|  +- S_FILS11
|  +- S_FILS12
|  |  +- SS_FILS121
|  +- S_FILS13
|     +- SS_FILS131
|     +- SS_FILS132
+- FILS2
|  +- S_FILS21
+- FILS3
   +- S_FILS31
$
```

Remarque : par rapport à l'exercice 1, il est demandé ici que le nom du répertoire passé en paramètre soit affiché en premier.

On pourra commencer par le premier résultat intermédiaire suivant :

```
$ arbre.sh TEST
TEST
FILS1
S_FILS11
S_FILS12
SS_FILS121
S_FILS13
SS_FILS131
SS_FILS132
FILS2
S_FILS21
FILS3
S_FILS31
$
```

puis continuer avec le deuxième résultat intermédiaire suivant :

```
$ arbre.sh TEST
TEST
+- FILS1
|  +- S_FILS11
|  +- S_FILS12
|  |  +- SS_FILS121
|  +- S_FILS13
|  |  +- SS_FILS131
|  |  +- SS_FILS132
+- FILS2
|  +- S_FILS21
+- FILS3
|  +- S_FILS31
$
```

On conseille de tenir compte des indications suivantes :

- Considérer que le script `arbre.sh` est une version modifiée de `arbre_base.sh` qui peut être appelée soit avec un seul paramètre, soit avec deux paramètres :
  - Lorsque l'utilisateur lance `arbre.sh`, il ne passe qu'un seul paramètre, qui est le nom d'un répertoire.
  - Lors des appels récursifs de `arbre.sh`, deux paramètres lui sont passés. Le second paramètre est une chaîne de caractères (placée entre guillemets) qui doit être affichée devant le nom d'un sous-répertoire. Cette chaîne de caractères est composée d'espaces, de caractères `|` et de caractères `+`
- On a intérêt à gérer un compteur de sous-répertoires. Lorsque la valeur de ce compteur est égale à `1` (le sous-répertoire à afficher est le dernier), une gestion spécifique de l'affichage doit être effectuée.

Attention :

- Tester si un répertoire est accessible en lecture et en exécution, avant d'éventuellement lancer un appel récursif.

- Alors que la commande :
  `echo a  b`
  affiche la chaîne suivante :
  `a b`
  qui ne contient qu'un seul espace entre les caractères `a` et `b`, chacune des commandes suivantes :
  - `echo "a  b"`
  - `echo 'a  b'`

  affiche la chaîne :
  `a  b`
  qui contient bien six espaces entre les caractères `a` et `b`.

SÉANCE 7

Processus Unix : terminaux, processus et signaux en shell de Bourne

Introduction

Gestion des terminaux

Un terminal correspond à une fenêtre dans laquelle s'exécute un interpréteur de commandes. Chaque terminal est représenté par un fichier spécial situé sous le répertoire `/dev`. La commande `tty` donne le chemin d'accès à ce fichier. Un terminal a un certain nombre de caractéristiques qui peuvent être affichées grâce à la commande `stty -a`. Parmi ces caractéristiques, les caractères de contrôle permettent d'associer une action à une combinaison de touches du type Ctrl-`touche` (c'est-à-dire qu'il faut maintenir la touche Ctrl enfoncée et appuyer sur la touche `touche`). Par exemple, par défaut, la combinaison de touches Ctrl-`c` permet d'interrompre l'exécution d'un programme. En fait, cela provoque l'envoi du signal `SIGINT` au processus qui s'exécute au premier plan (voir plus loin). Pour pouvoir utiliser une autre combinaison de touches pour réaliser la même action, on peut faire appel à la commande :

`stty action '^touche'`

L'`action` correspondant à Ctrl-`c` est `intr`.

Gestion des processus

Un processus est un programme en cours d'exécution. La commande `ps` permet d'obtenir des informations sur les processus (par défaut, uniquement ceux qui sont contrôlés par le terminal). Elle accepte, entre autres, les options suivantes :

- `-e` : affiche les informations sur tous les processus ;
- `-a` : affiche les informations sur tous les processus sauf ceux qui ne sont pas associés à un terminal ;
- `-f` : affiche des informations détaillées ;
- `-l` : affiche des informations très détaillées ;
- `-u "liste d'utilisateurs"` : affiche les informations sur les processus appartenant aux utilisateurs mentionnés.

Les informations très détaillées affichées peuvent être les suivantes :

- `F` : obsolète ;
- `S` : état des processus
  - `O` : en train de s'exécuter sur le processeur ;
  - `S` : endormi ou bloqué (attend un événement pour continuer) ;
  - `R` : prêt ;
  - `Z` : zombie (terminé alors que son père n'a pas encore pris connaissance de sa terminaison) ;
  - `T` : stoppé (par le signal `SIGSTOP` par exemple) ;
- `UID` : numéro ou nom du propriétaire du processus ;
- `PID` : numéro du processus ;
- `PPID` : numéro du processus père ;
- `C` : obsolète ;
- `PRI` : priorité (plus le nombre est grand, plus la priorité est basse) ;

`NI` : gentillesse (plus le nombre est grand, moins l'accès du processus au processeur est fréquent) ;

`ADDR` : adresse mémoire du processus ;

`SZ` : taille du processus en Ko (code + données + pile) ;

`WCHAN` : nom de la fonction du noyau du système dans laquelle le processus est endormi (bloqué) ;

`STIME` : heure de lancement du processus ;

`TTY` : terminal de contrôle du processus ;

`TIME` : temps d'exécution cumulé du processus ;

`CMD` : commande qui a donné naissance au processus.

La commande `top` permet d'afficher des informations concernant les processus qui utilisent le plus le processeur. Ces informations sont mises à jour régulièrement. La commande `top -Uutilisateur` permet de n'afficher des informations que sur les processus appartenant à `utilisateur`.

La commande `sleep sec` suspend l'exécution pendant `sec` secondes. Par exemple, `(sleep 30; echo "C'est fini !")&` affiche au bout de 30 secondes le message `C'est fini !`

Gestion des jobs

La plupart des interpréteurs de commandes intègrent un « contrôleur des jobs ». Une commande ou un enchaînement de commandes (*pipeline*) constituent un *job*. Un job peut être dans l'un des états suivants :

- *Au premier plan :* il peut effectuer des lectures et des écritures sur son terminal de contrôle. à un instant donné, il peut y avoir au plus un job au premier plan.
- *En arrière-plan :* il ne peut pas effectuer de lecture sur le terminal mais il peut effectuer des écritures si les caractéristiques du terminal le permettent. à un instant donné, il peut y avoir plusieurs jobs en arrière-plan. Lancer un job en arrière-plan laisse l'interpréteur disponible pour lancer une autre commande ; ceci n'est pas le cas quand on lance un job au premier plan.
- *Stoppé :* il est dans un état suspendu.

Chaque job lancé dans un shell reçoit un numéro (entier positif) qui peut servir à l'identifier. L'identification d'un job peut se faire grâce au caractère `%` immédiatement suivi du numéro du job.

La commande `jobs` permet d'afficher le numéro, l'état et la commande correspondant aux jobs du shell courant.

Le caractère `&` en fin de commande permet de créer un job en arrière-plan.

La frappe de la combinaison de touches Ctrl-`z` permet de stopper le job au premier plan en lui envoyant le signal `SIGSTOP`.

La commande `wait %id_job` attend la fin d'exécution d'un job en arrière-plan.

La commande `bg [%id_job]` permet de reprendre l'exécution en arrière-plan d'un job stoppé. Si `bg` est appelée sans argument, c'est le job courant (celui qui vient d'être stoppé) qui est considéré.

La commande `fg [%id_job]` permet de reprendre l'exécution au premier plan d'un job stoppé. Si `fg` est appelée sans argument, c'est le job courant (celui qui vient d'être stoppé) qui est considéré.

Remarque : la commande `kill` accepte un identificateur de job en argument. `kill -s STOP %id_job` permet de stopper le job au premier plan en lui envoyant le signal `STOP`.

Signaux et processus

Le mécanisme des signaux permet de gérer des événements pouvant survenir dans la vie d'un processus. Un signal peut être envoyé à un processus :

- lors d'une erreur interne au processus (erreur arithmétique, violation de protection mémoire, ...) ;

- par la frappe d'une combinaison de touches (par exemple, la combinaison Ctrl-`c` effectuée depuis un terminal envoie le signal `SIGINT` au processus qui a été lancé depuis ce terminal au premier plan) ;
- par un autre processus (commande ou fonction `kill`).

Le mécanisme des signaux permet par exemple de gérer la synchronisation entre processus. L'utilisation classique des signaux vous sera présentée en L3. Ici, nous allons seulement voir un exemple de prise en compte de l'arrivée d'un signal dans un script shell.

Émission d'un signal

Par défaut, les caractéristiques d'un terminal permettent d'envoyer un certain nombre de signaux au processus lancé au premier plan. Par exemple, la combinaison de touches Ctrl-\ envoie le signal `SIGQUIT`.

La commande `kill -signal pid` envoie le signal `signal` au processus d'identificateur `pid`.

La commande `kill -l` affiche la liste des signaux disponibles.

Réception d'un signal

Quand un processus reçoit un signal, il a trois comportements par défaut selon le signal reçu :

- ignorer le signal,
- terminer son exécution,
- terminer son exécution en créant un fichier `core`[1] (c'est par exemple le cas avec la combinaison de touches Ctrl-\ qui envoie le signal `SIGQUIT`).

On peut changer ce comportement par défaut en « déroutant » le signal (ce n'est pas possible pour tous les signaux, comme par exemple `SIGKILL` et `SIGSTOP`). En shell, un tel déroutement est possible grâce à la commande `trap`.

La commande `trap [[commande] signal [...]]` permet au shell courant (en général, à un script) d'exécuter la commande `commande` à l'arrivée des signaux désignés. Cette désignation peut être numérique (à chaque signal est associé un entier positif) ou symbolique (voir `kill -l`). Après l'exécution de la `commande`, l'exécution du shell reprend là où elle avait été interrompue sauf si la commande `exit` est rencontrée. Si la `commande` est la chaîne vide, alors les signaux seront ignorés. Si la `commande` est absente, alors les signaux désignés reprennent leur comportement original (qu'ils avaient au lancement du shell). Si le numéro du signal est `0`, alors la `commande` est exécutée à la sortie du shell (à la fin de l'exécution du script). La commande `trap` sans argument affiche la liste des commandes associées à chaque numéro de signal qui est dérouté.

Attention : le comportement de cette commande peut varier d'un shell à l'autre.

Exercices

Exercice 22

1. Afficher les informations détaillées, puis très détaillées, concernant tous les processus vous appartenant.
2. Afficher le nombre total de processus issus de la commande `firefox` présents.
3. Lancer un interpréteur de shell de Bourne. Lancer la commande `xterm` en arrière-plan et mettre de côté la fenêtre obtenue.

1. Un fichier `core` contient des informations sur le processus au moment de sa terminaison et peut être analysé par certains outils de mise au point.

4. Lancer l'éditeur `nedit` au premier plan, puis stopper son exécution. Enfin, reprendre son exécution en arrière-plan.
5. Vérifier l'état de vos processus en affichant les informations très détaillées concernant les processus du terminal. Vérifier l'état de vos jobs.
6. Stopper l'exécution de l'éditeur et refaire les mêmes vérifications que dans la question précédente.
7. Reprendre l'exécution de l'éditeur au premier plan. Quitter l'éditeur et la fenêtre `xterm`.

Exercice 23

1. Depuis un autre terminal (obtenu par exemple par la commande `xterm&`), lancer la commande `top` afin de vérifier l'état de vos processus tout au long des manipulations de cet exercice (la touche `q` permet de terminer l'exécution de la commande `top`).
2. écrire un script en shell de Bourne qui affiche l'heure en boucle sur `stdout`. Pour cela, on utilisera une boucle infinie (`while true`...) et on fera une pause d'une seconde entre chaque affichage.
3. Vérifier l'effet de l'envoi du signal `SIGINT` (signal numéro 2) sur l'exécution de ce script en utilisant les touches Ctrl-`c`.
4. Associer l'envoi du signal `SIGINT` à la combinaison de touches Ctrl-`a`. Relancer le script précédent et vérifier les effets de cette association.
5. Compléter le script de telle sorte que, lorsqu'il recevra le signal `SIGINT`, un message soit affiché sur `stdout` et le script continue à s'exécuter. Pour arrêter son exécution, on pourra lui envoyer le signal `SIGQUIT` (signal numéro 3) avec la combinaison de touches Ctrl-\.
6. Depuis un autre terminal, vérifier les effets de l'envoi des signaux `SIGINT`, `SIGQUIT` et `SIGKILL` (signal numéro 9) sur le script précédent.

Exercice 24

La commande `trap` peut être utilisée pour gérer l'effacement des fichiers temporaires dans les scripts. En effet, par défaut, si l'utilisateur interrompt l'exécution du script en tapant la combinaison de touches Ctrl-`c`, les éventuels fichiers temporaires créés par le script ne seront pas effacés.

1. écrire un script qui lit un texte tapé au clavier, l'écrit dans un fichier temporaire, l'affiche sur `stdout` et supprime le fichier temporaire. La lecture et l'écriture du texte dans le fichier temporaire se font ligne par ligne. Chaque ligne est lue sur `stdin` et écrite à la fin du fichier temporaire. La lecture doit s'arrêter dès que l'utilisateur tape une ligne vide. Exécuter le script en le terminant normalement, puis en l'interrompant avec Ctrl-`c`. Vérifier dans les deux cas la présence ou l'absence du fichier temporaire à l'issue de l'exécution.
2. Modifier le script précédent de telle sorte :
   - qu'il retourne 0 en cas de fin normale et 1 en cas d'interruption par le signal `INT`,
   - qu'il efface le fichier temporaire en cas de fin normale mais aussi en cas d'interruption par le signal `INT`.

   Depuis l'interpréteur de commandes, vérifiez le code retourné par le script avec la commande `echo $?`

Chapitre 5
Description des commandes

5.1 Introduction

Ce chapitre contient les descriptions des commandes utilisées dans les cours/TD intégrés et dans les TD machine. Toutes les options de toutes les commandes ne sont pas systématiquement décrites. Seules les options les plus courantes sont présentées. Chaque description donne les caractéristiques de la commande (étymologie, descriptif rapide), sa ou ses syntaxes d'appel (en utilisant les conventions décrites au paragraphe 2.5, page 16), une brève description du rôle de ses éventuelles options, des exemples d'utilisation, ses codes de retour et, éventuellement, quelques remarques. Dans les exemples, les noms de répertoires sont généralement formés de lettres majuscules.

Pour une recherche de la description d'une commande à partir de son nom, il est conseillé d'utiliser l'index qui se trouve à la fin de ce document.

5.2 Où sont les commandes ?

Le code exécutable de toutes ces commandes est situé dans le répertoire `/usr/bin`. Néanmoins, un certain nombre d'entre elles (par exemple `echo`) existent également en tant que commandes internes, c'est-à-dire que leur code exécutable est également inclus dans le code de l'interpréteur de commandes lui-même. Dans ce cas, c'est la version interne qui est lancée, sauf si on écrit la désignation du fichier exécutable de la commande externe (par exemple : `/usr/bin/echo`). C'est ce qui explique que les commandes internes puissent différer d'un shell à l'autre, alors que les commandes externes sont les mêmes pour tous les shells.

5.3 À l'aide !

☞ `man`

➥*Caractéristiques :*

1. `man` vient de *manual* (« manuel »).
2. Cette commande permet d'obtenir la description d'une commande Unix à l'écran (généralement en anglais).
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`man` [`-`] [`-adFlrt`] [`-M` arbre] [`-T` macro] [`-s` section] nom_commande [...]

➥*Description :* cf. la description de la commande `man`, en tapant `man man`

➥*Exemple :* la commande `man chmod` affiche la description de la commande `chmod`

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

5.4 Manipulation de l'arborescence des fichiers

☞ `ls`

➥*Caractéristiques :*

1. `ls` vient de *list* (« lister »).
2. Cette commande permet d'afficher le contenu d'un ou de plusieurs répertoires, ou des renseignements concernant un ou plusieurs fichiers. S'il n'y a aucun paramètre, alors la commande affiche le contenu du répertoire courant. Si un nom de répertoire est passé en paramètre, alors la commande affiche son contenu. Si un nom de fichier est passé en paramètre, alors la commande indique si ce fichier est accessible.
3. Elle peut apparaître en début de branchement.

➥*Syntaxes :*
- `ls` [`-RadLCxmlnogrtucpFbqisf1AM`] [`désignation_répertoire` ...]
- `ls` [`-RadLCxmlnogrtucpFbqisf1AM`] [`désignation_fichier` ...]

➥*Description :*
- `-a` : affiche également les fichiers commençant par le caractère `.` (fichiers « cachés »).
- `-l` : affiche une première ligne indiquant la taille totale du répertoire en nombre de « blocs »puis, pour chaque sous-répertoire ou fichier accessible, ses principales caractéristiques sur une ligne : nature (répertoire ou fichier), droits d'accès, nombre de liens, nom du propriétaire, nom du groupe du propriétaire, taille en octets, date et heure de dernière modification, nom (qui commence au $55^{\text{ème}}$ caractère de la ligne). Si cette option est utilisée et qu'un fichier est passé en paramètre, alors ces informations ne sont affichées que pour ce fichier.
- Autres options : cf. la description de la commande `ls`, en tapant `man ls`

➥*Exemple :*

```
$ ls -al
total 1224
drwxr-xr-x   9 durou   tci     512 Dec  3 14:58 .
drwxr-xr-x  13 durou   tci     512 Oct  6 17:21 ..
drwxr-xr-x   2 durou   tci    2048 Dec  4 01:42 TD
drwxr-xr-x   2 durou   tci    1024 Dec  3 18:12 TP
-rw-r--r--   1 durou   tci   12105 Dec  4 00:45 welcome.html
$
```

Dans cet exemple, les répertoires `.` et `..` désignent respectivement le répertoire courant et le répertoire père du répertoire courant. Ce sont des répertoires cachés qui apparaissent systématiquement avec l'option `-a`

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarque :*
La commande `ls` affiche un nom de fichier ou de sous-répertoire par ligne sur sa sortie standard. Néanmoins, lorsque l'affichage s'effectue à l'écran, c'est-à-dire lorsque la sortie standard n'est pas redirigée, les noms de fichiers et de sous-répertoires sont alignés en plusieurs colonnes. Le nombre de ces colonnes est fonction, entre autres, de la taille de la fenêtre dans laquelle la commande `ls` est interprétée.

☞ **cp**

➥*Caractéristiques :*

1. `cp` vient de *copy* (« copier »).
2. Cette commande permet de faire une copie d'un ou de plusieurs fichiers. Si on copie un seul fichier, on peut changer son nom (cf. le premier exemple ci-dessous). En revanche, si on désire copier plusieurs fichiers en une seule commande (cf. le deuxième exemple ci-dessous), on ne peut pas changer leurs noms.

➥*Syntaxes :*

- `cp [-fip] ancienne_désignation_fichier nouvelle_désignation_fichier`
- `cp [-fip] désignation_fichier [...] désignation_répertoire`
- `cp -r|-R [-fip] ancienne_désignation_répertoire [...] nouvelle_désignation_répertoire`

➥*Description :* cf. la description de la commande `cp`, en tapant `man cp`

➥*Exemples :*

- `cp PREPA/TD/td02ex01.c TP/td02ex01.txt`
  Cette commande copie le fichier `td02ex01.c`, situé dans le sous-répertoire `PREPA/TD`, dans le sous-répertoire `TP`, sous le nom `td02ex01.txt`
- `cp PREPA/TD/td02ex*.c TP`
  Cette commande copie tous les fichiers du sous-répertoire `PREPA/TD`, de nom commençant par `td02ex` et comportant l'extension `.c`, dans le sous-répertoire `TP`, sans changer leurs noms.
- `cp -r PREPA/TD/ WIKI`
  Cette commande copie le sous-répertoire `PREPA/TD` et l'ensemble de son contenu dans le sous-répertoire `WIKI`, sans changer les noms.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarque :* on ne peut pas faire une copie d'un fichier dans son répertoire d'origine sans changer son nom, puisque deux fichiers frères ne peuvent pas porter le même nom.

☞ **mv**

➥*Caractéristiques :*

1. `mv` vient de *move* (« déplacer »).
2. Cette commande permet de déplacer un ou plusieurs fichiers ou répertoires dans l'arborescence. Si on déplace un seul fichier ou répertoire, on peut changer son nom (cf. le premier exemple ci-dessous), ce qui permet en particulier de renommer un fichier ou un répertoire sans le déplacer. En revanche, si on désire déplacer plusieurs fichiers ou répertoires en une seule commande, on ne peut pas changer leurs noms (cf. le deuxième exemple ci-dessous).

➥*Syntaxes :*
- `mv` [`-fi`] `ancienne_désignation` `nouvelle_désignation`
- `mv` [`-fi`] `désignation` [...] `désignation_répertoire`

➥*Description :* cf. la description de la commande `mv`, en tapant `man mv`

➥*Exemples :*
- `mv TP ../TP_2005`
  Cette commande permet de renommer `TP` en `TP_2005`, et de le remonter d'un étage dans l'arborescence.
- `mv TD/td02ex* TD/td03ex* PREPA`
  Cette commande permet de déplacer dans le sous-répertoire `PREPA`, les fichiers du sous-répertoire `TD` dont le nom commence par `td02ex` ou `td03ex`, sans changer leurs noms.

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ `rm`

➥*Caractéristiques :*
1. `rm` vient de *remove* (« supprimer »).
2. Cette commande permet de supprimer un ou plusieurs fichiers passés en paramètres.

➥*Syntaxes :*
- `rm` [`-f`] [`-i`] `désignation_fichier` [...]
- `rm -rR` [`-f`] [`-i`] `désignation_répertoire` [...]

➥*Description :*
- `-f` : ne supprime les fichiers désignés que s'ils existent (cela évite d'éventuels messages d'erreur).
- `-i` : demande confirmation avant effacement.
- Autres options : cf. la description de la commande `rm`, en tapant `man rm`

➥*Exemple :*

```
rm *
```

Cette commande doit être évitée en général, car elle efface tous les fichiers non cachés du répertoire courant !

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

Attention :
L'expérience montre que beaucoup d'utilisateurs sont tentés de placer la commande `rm` en fin de branchement, comme dans l'exemple suivant :

```
ls *.bak | rm
```

Cette syntaxe est incorrecte, car la commande `rm` n'est pas un filtre. Pour supprimer les fichiers du répertoire courant comprenant l'extension `.bak`, on peut bien sûr utiliser l'une des deux syntaxes suivantes :

```
rm *.bak
rm `ls *.bak`
```

➥*Remarque :*
Pour supprimer un fichier dont le nom commence par un caractère `-`, comme par exemple `-fichier`, on peut taper la commande :

```
rm -- -fichier
```

La commande `rm -fichier` provoquerait l'affichage d'un message d'erreur, car le shell interpréterait les caractères `-fichier` comme une liste d'options de la commande `rm`. On peut également taper :

```
rm ./-fichier
```

☞ mkdir

➥*Caractéristiques :*

1. `mkdir` vient de *make directory* (« créer répertoire »).
2. Cette commande permet de créer un ou plusieurs répertoires, dont les noms sont passés en paramètres.

➥*Syntaxe :*
`mkdir` [`-m` `mode`] [`-p`] `désignation_répertoire` [...]

➥*Description :*
- `-m mode` : indique les droits d'accès au répertoire à créer.
- `-p` : crée tous les répertoires intermédiaires, si nécessaire.

➥*Exemples :*
- `mkdir -m go-rw REP1`
  Cette commande crée le sous-répertoire `REP1` dans le répertoire courant, en enlevant les droits en lecture et en écriture (`r` pour *read* et `w` pour *write*) à toute personne non propriétaire du compte (`g` pour *group* et `o` pour *others*).
- `mkdir -p REP2/SREP2/SSREP2`
  Cette commande crée le sous-répertoire `REP2` du répertoire courant, le sous-répertoire `SREP2` de `REP2` et le sous-répertoire `SSREP2` de `SREP2`.

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ rmdir

➥*Caractéristiques :*

1. `rmdir` vient de *remove directory* (« supprimer répertoire »).
2. Cette commande permet d'effacer un ou plusieurs répertoires passés en paramètres. Ces répertoires doivent être vides.

➥*Syntaxe :*
`rmdir` [`-ps`] `désignation_répertoire` [...]

➥*Description :* cf. la description de la commande `rmdir`, en tapant `man rmdir`

➥*Exemple :*

```
rmdir ..
```

Cette commande provoque systématiquement une erreur, car le répertoire père du répertoire courant n'est jamais vide. De même, la commande `rmdir .` provoquerait l'affichage d'un message d'erreur, car on ne peut pas supprimer le répertoire dans lequel on se trouve.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ **pwd**

➥*Caractéristiques :*

1. `pwd` vient de *print working directory* (« afficher le répertoire courant »).
2. Cette commande affiche la désignation absolue du répertoire courant.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`pwd`

➥*Description :* Pas d'option.

Exemple :

```
$ pwd
/users/linfg/linfg0/S3
$
```

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ **cd**

➥*Caractéristiques :*

1. `cd` vient de *change directory* (« changer de répertoire »).
2. Cette commande permet de changer de répertoire courant, de manière relative ou absolue.
3. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`cd` [`désignation_répertoire`]

Lorsque la commande est tapée sans paramètre, on se retrouve dans le répertoire d'accueil (*home directory*).

➥*Description :* pas d'option.

➥*Exemple :*

```
cd ../TD
```

Cette commande permet de se déplacer dans l'arborescence vers le répertoire frère du répertoire courant, de nom `TD`

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ chmod

➥*Caractéristiques :*

1. `chmod` vient de *change mode* (« changer de mode »).
2. Cette commande permet de modifier les droits d'accès à un ou à plusieurs fichiers ou répertoires dont on est le propriétaire.

➥*Syntaxe :*

`chmod` [`-fR`] [`personnes`...]`+`|`-`[`permissions`...] `désignation` [...]

- Les `personnes` (classes d'utilisateurs) sont `u` (*user*) s'il s'agit du propriétaire du fichier ou du répertoire, `g` (*group*) s'il s'agit des membres du groupe du propriétaire, `o` (*others*) s'il s'agit des autres utilisateurs ou `a` (*all*) s'il s'agit de tout le monde.
- Le signe `+` signifie qu'on ajoute des droits d'accès, alors que `-` signifie qu'on en enlève.
- Les `permissions` sont `r` (*read*) pour la lecture, `w` (*write*) pour l'écriture et `x` (*execute*) pour l'exécution.

➥*Description :* cf. la description de la commande `chmod`, en tapant `man chmod`

➥*Exemple :*

```
chmod g+rx /users/linfg/linfg0/omar
```

Cette commande rajoute aux membres du groupe les droits en lecture et en exécution sur le fichier spécifié.

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarques :*
- Pour connaître les droits d'accès à un fichier, il suffit de taper la commande `ls -l`
- Pour lancer l'exécution d'un script contenu dans le fichier `nom_script.sh`, on peut aussi le rendre exécutable, puis l'appeler directement, de la manière suivante :
  `$ chmod u+x nom_script.sh`
  `$ nom_script.sh` [`paramètre_script` ...]

☞ find

➥*Caractéristiques :*

1. `find` signifie « trouver ».
2. Cette commande permet de rechercher les fichiers et les répertoires qui se trouvent dans les sous-arborescences dont les racines sont les répertoires dont les désignations sont passées en paramètres. Les derniers paramètres permettent de choisir les fichiers et les répertoires qui nous intéressent ainsi que de préciser ce que l'on souhaite en faire.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

Parmi toutes les possibilités, voici deux utilisations très classiques de la commande `find` :

`find` désignation_répertoire [...] `-name` modèle_nom [`-exec` commande `\;`]

➥*Description :*

- `-name` modèle_nom : sélectionne les fichiers et les répertoires dont les noms respectent le modèle modèle_nom. En général, ce modèle contient des métacaractères du shell qui doivent être interprétés par la commande `find`. Pour empêcher que le shell ne les interprète, il est nécessaire d'entourer modèle_nom de guillemets ou d'apostrophes. Si l'option `-exec` n'est pas utilisée, alors les désignations (relatives aux racines des sous-arborescences) des fichiers et des répertoires sélectionnés sont affichées (une par ligne) sur la sortie standard.
- `-exec` commande : les désignations des fichiers et des répertoires concernés ne sont pas affichées sur la sortie standard, mais la commande commande est appliquée à chacun d'entre eux. Dans cette commande, les caractères `{}` représentent la désignation de chacun des fichiers et les caractères `\;` terminent la commande.
- autre options : cf. la description de la commande `find`, en tapant `man find`

➥*Exemples :*

- `find . -name "??????"`
  Cette commande affiche les désignations relatives des sous-répertoires et des fichiers qui sont des « descendants »du répertoire courant, et dont les noms comportent six caractères exactement.
- `find $HOME -name 'core' -exec rm {} \;`
  Cette commande efface tous les fichiers de nom `core` du compte de l'utilisateur.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ dirname

➥*Caractéristiques :*

1. `dirname` vient de *directory name* (« nom du répertoire »).
2. Cette commande permet d'afficher sur la sortie standard la chaîne de caractères constituée de la désignation d'un fichier ou d'un répertoire privée du dernier caractère `/` et des caractères suivants. Si la désignation ne contient que le nom d'un fichier ou d'un répertoire, alors le caractère `.` qui désigne le répertoire courant est affiché.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`dirname` désignation

➥*Description :* pas d'option.

➥*Exemples :*

- ```
  $ dirname /users/linfg/l2inf201/REP/fich.txt
  /users/linfg/l2inf201/REP
  $
  ```

- ```
  $ dirname fich.txt
  .
  $
  ```

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ `basename`

➥*Caractéristiques :*

1. `basename` vient de *base name* (« nom de base »).
2. Cette commande permet d'afficher sur la sortie standard la chaîne de caractères constituée de la désignation d'un fichier ou d'un répertoire privée des caractères situés avant le dernier caractère `/` ainsi que d'un éventuel suffixe.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`basename` désignation [suffixe]

➥*Description :* pas d'option.

➥*Exemples :*

- ```
  $ basename /users/linfg/l2inf201/REP/fich.txt
  fich.txt
  $
  ```

- ```
  $ basename /users/linfg/l2inf201/REP/fich.txt .txt
  fich
  $
  ```

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

5.5 Accès au contenu des fichiers

☞ `cat`

➥*Caractéristiques :*

1. `cat` vient de *concatenate* (« concaténer »).
2. Cette commande permet d'afficher le contenu de l'entrée standard ou le contenu du ou des fichiers passés en paramètres, en les concaténant.
3. Cette commande est un filtre.

➥*Syntaxe :*

`cat` [`-nbsuvet`] [désignation_fichier ...]

➥*Description :*
- `-n` : numérote les lignes, avant de les envoyer sur la sortie standard.
- `-b` : identique à `-n`, sans numérotation des lignes vides.
- Autres options : cf. la description de la commande `cat`, en tapant `man cat`

➥*Exemples :*

- `cat fich1.txt fich2.txt > fich3.txt`
  Cette commande recopie `fich1.txt` puis, à la suite, `fich2.txt`, le tout dans `fich3.txt`
- Dans l'exemple suivant, ce sont les caractères tapés sur l'entrée standard qui sont écrits dans le fichier `fich.txt` :

```
$ cat > fich.txt
ligne1
ligne2
<Ctrl>D
$
```

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ `more`

➥*Caractéristiques :*

1. `more` signifie « encore ».
2. Cette commande permet d'afficher le contenu de l'entrée standard (ou, s'il y a lieu, le contenu du ou des fichiers passés en paramètres, en les concaténant), page par page. Elle est une version plus interactive de la commande `cat` :
   - Pour afficher la page suivante, il faut taper sur la barre d'espace.
   - Pour afficher seulement une nouvelle ligne, il faut taper sur la touche « entrée ».
   - Pour interrompre l'affichage, il suffit de taper le caractère `q`.
3. C'est un filtre (cf. tout de même la remarque ci-dessous).

➥*Syntaxe :*

`more` [`-cdflrsuw`] [`-`*lignes*] [`+`*numéro_ligne*] [`+/`*modèle*] [*désignation_fichier* ...]

➥*Description :* cf. la description de la commande `more`, en tapant `man more`

➥*Exemples :*

La commande `more 'ls'` affiche le contenu de tous les fichiers du répertoire courant (les sous-répertoires du répertoire courant sont ignorés). Le branchement de commandes `ls -l | more` affiche page par page les informations détaillées concernant les fichiers et les répertoires contenus dans le répertoire courant.

➥*Codes de retour :* cf. la description de la commande `more`, en tapant `man more`

➥*Remarque :* en fait, la commande `more` n'est pas un vrai filtre, dans la mesure où elle n'accepte de données par l'intermédiaire de l'entrée standard que si l'entrée standard a été redirigée (sur ce point, la commande `more` diffère de la commande `cat`).

☞ `tee`

➥*Caractéristiques :*

- `tee` désigne un « embranchement en forme de T ».
- Cette commande permet d'afficher les données reçues sur l'entrée standard simultanément sur la sortie standard et sur la liste (éventuellement vide) de fichiers passés en paramètres.
- C'est un filtre.

➥*Syntaxe :*

`tee` [`-ai`] [*désignation_fichier* ...]

➥*Description :*

- `-a` : les données reçues sur l'entrée standard sont concaténées aux fichiers passés en paramètres.
- Autres options : cf. la description de la commande `tee`, en tapant `man tee`

➥*Exemple :*

```
ls | tee liste_fichiers.txt
```

Cette commande affiche, d'une part, le contenu du répertoire courant à l'écran et écrit, d'autre part, ce contenu dans le fichier `liste_fichiers.txt`

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ head

➥*Caractéristiques :*

1. `head` signifie « tête ».
2. Cette commande affiche l'entrée standard, ou le ou les fichiers passés en paramètres, jusqu'à une position désignée. Sans option, elle affiche les `10` premières lignes.
3. C'est un filtre.

➥*Syntaxe :*

`head` [`-n` position] [désignation_fichier ...]

➥*Description :*

- `-n` position : affiche les position premières lignes du fichier ou de l'entrée standard.
- Autres options : cf. la description de la commande `head`, en tapant `man head`

➥*Exemple :*

```
head -n 5 fichier.c
```

Cette commande affiche les `5` premières lignes de `fichier.c`

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ tail

➥*Caractéristiques :*

1. `tail` signifie « queue ».
2. Cette commande affiche l'entrée standard, ou le fichier passé en paramètre, à partir de la position désignée. Sans option, elle affiche les `10` dernières lignes.
3. C'est un filtre.

➥*Syntaxe :*

`tail` [`-n` [+] position|`-c` [+] position] [désignation_fichier]

➥*Description :*

- `-n position` : affichage des `position` dernières lignes de `désignation_fichier`
- `-n +position` : affichage de `désignation_fichier` à partir de la ligne numéro `position` (cette ligne étant comprise).
- `-c position` : affichage des `position` derniers caractères de `désignation_fichier`
- `-c +position` : affichage de `désignation_fichier` à partir du caractère numéro `position` (ce caractère étant compris).
- Autres options : cf. la description de la commande `tail`, en tapant `man tail`

➥*Exemples :*

- `tail -n +2 fic.txt`
  Cette commande affiche toutes les lignes de `fic.txt`, sauf la première.
- `tail -c 15 fichier.c`
  Cette commande affiche les `15` derniers caractères du fichier `fichier.c`

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ grep

➥*Caractéristiques :*

1. `grep` vient de *global regular expressions parsing* (« analyse d'expressions régulières globales »).
2. Cette commande permet de rechercher dans un ou plusieurs fichiers passés en paramètres (ou dans chaque ligne de l'entrée standard) les lignes qui correspondent à une expression régulière (cf. paragraphe 3.3, page 28) et affiche ces lignes sur la sortie standard. Si plusieurs fichiers sont passés en paramètres, chaque ligne correspondant à l'expression régulière est précédée du nom du fichier la contenant, suivi d'un caractère `:`
3. Cette commande est un filtre.

➥*Syntaxe :*

`grep` [`-bchilnsvw`] `expression_régulière` [`désignation_fichier` ...]

➥*Description :*

- `-l` : la commande n'affiche que le nom de chaque fichier où une ligne correspond (au lieu d'afficher les lignes qui correspondent), ou `<stdin>` si une ligne qui correspond est trouvée sur l'entrée standard.
- `-c` : affiche uniquement, pour chaque fichier, le nombre de lignes qui correspondent.
- `-v` : affiche les lignes qui ne correspondent pas à l'expression régulière.
- `-n` : chaque ligne qui correspond est précédée de son numéro (la première ligne a le numéro `1`), suivi d'un caractère `:`
- Autres options : cf. la description de la commande `grep`, en tapant `man grep`

➥*Exemple :* la commande `grep -c '.*' fich` affiche le nombre de lignes du fichier `fich`

➥*Codes de retour :*

- Une ligne au moins correspond : `0`
- Aucune ligne ne correspond : `1`
- Opération interrompue pour cause d'erreur : `2`

➥*Remarque :*

L'utilisation d'expressions régulières contenant des caractères de contrôle, à l'aide de la commande `grep`, pose quelques problèmes. Par exemple, la commande `grep '\tabc'` recherche, non pas les lignes de l'entrée standard contenant un caractère de tabulation suivi des trois lettres `abc`, mais les lignes

contenant la séquence de quatre lettres `tabc`. Il en serait de même si l'expression régulière était écrite `"\tabc"` ou `\tabc`. Le seul moyen de rendre correcte cette recherche est de taper réellement sur la touche de tabulation du clavier, pour signifier que le premier caractère de l'expression régulière doit être un caractère de tabulation.

☞ sort

➥*Caractéristiques :*

1. `sort` signifie « trier ».
2. Cette commande lit un ou plusieurs fichiers ou l'entrée standard, et affiche les différentes lignes, triées sur un ou plusieurs champs (par défaut, sur un seul champ qui est la ligne), selon l'ordre lexicographique. Les séparateurs de deux champs sont l'espace et la tabulation.
3. C'est un filtre.

➥*Syntaxe :*

`sort` [`-cmu`] [`-o` sortie] [`-T` répertoire] [`-dfiMnr`] [`-b`] [`-t` caractère] [`-k` clé] [`+`pos_1 [`-`pos_2]] [désignation_fichier ...]

➥*Description :*

- `-k` clé : permet de préciser les champs de la ligne à prendre en compte pour le tri. L'argument clé doit être écrit sous la forme numéro_premier_champ[,numéro_dernier_champ]. La partie entre crochets est optionnelle : si elle est absente, cela signifie que le tri prend en compte tous les champs à partir du champ de numéro numéro_premier_champ
- `-u` : affichage d'un seul exemplaire des lignes identiques.
- `-c` : si le fichier à trier est déjà trié, aucun affichage. Sinon, affichage des lignes triées, et retour d'un code égal à `1`.
- Autres options : cf. la description de la commande `sort`, en tapant `man sort`

➥*Exemple :*

Supposons que le fichier de nom `etat_civil.txt` soit structuré de la manière suivante :

```
Anne DUPONT 26 ans
Jean DUPONT 22 ans
Mathieu BARDON 23 ans
```

Voici deux exemples d'exécution de la commande `sort` :

```
$ sort -k 2,2 etat_civil.txt
Mathieu BARDON 23 ans
Anne DUPONT 26 ans
Jean DUPONT 22 ans
$ sort -k 2 etat_civil.txt
Mathieu BARDON 23 ans
Jean DUPONT 22 ans
Anne DUPONT 26 ans
$
```

➥*Codes de retour :*

- Pas d'erreur ou option `-c` sur fichier déjà trié : `0`
- Option `-c` sur fichier non trié : `1`
- Opération interrompue pour cause d'erreur : `> 1`

☞ cut

➥*Caractéristiques :*

1. `cut` signifie « couper ».
2. Cette commande permet de supprimer une partie des lignes d'un ou de plusieurs fichiers, ou de l'entrée standard.
3. Cette commande est un filtre.

➥*Syntaxes :*

- `cut -c liste [désignation_fichier ...]`
- `cut -f liste [-d délimiteur] [-s] [désignation_fichier ...]`

➥*Description :*

- `-c` : ne retient sur chaque ligne que les caractères situés à une position donnée par `liste`. Par exemple, l'option `-c1-50` ne garde que les `50` premiers caractères de chaque ligne.
- `-f` : ne retient qu'une liste de champs (séparés soit par des tabulations, soit par un `délimiteur` spécifié par l'option `-d`). Par exemple, l'option `-f1,3-5` conduira la commande à ne garder que les champs numéros `1`, `3`, `4` et `5` de chaque ligne.
- Autres options : cf. la description de la commande `cut`, en tapant `man cut`

➥*Exemples :*

- Soit un fichier `calepin.txt`, représentant un annuaire téléphonique, dans lequel chaque ligne apparaît sous la forme :
  `DUPONT Jean 05.61.75.18.47`
  La commande suivante :
  `cat calepin.txt | cut -f3 -d' ' | cut -f1 -d'.' | grep '05' | wc -l`
  affiche le nombre de personnes figurant dans ce calepin et résidant dans le Sud-Ouest (le numéro de téléphone de ces personnes commence par les chiffres `05`).
- Le format d'affichage de la commande `wc désignation_fichier` est en fait le suivant :
  `<espaces>nb_l<espaces>nb_m<espaces>nb_c<espaces>désignation_fichier`
  où `nb_l` désigne le nombre de lignes, `nb_m` le nombre de mots et `nb_c` le nombre de caractères de `désignation_fichier`. Quant au symbole `<espaces>`, il désigne une chaîne non vide, de longueur variable, composée du seul caractère espace. Par conséquent, si on désire afficher le nombre de mots du fichier `fichier.txt`, on peut taper l'une des trois commandes suivantes :

```
wc fichier.txt | tr -s ' ' | cut -f3 -d' '
wc -w fichier.txt | tr -s ' ' | cut -f2 -d' '
cat fichier.txt | wc -w | tr -d ' '
```

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarque :* on peut également écrire `-c liste`, `-f liste` et `-d délimiteur` en collant `liste` ou `délimiteur` aux options `-c`, `-f` ou `-d`

☞ wc

➥*Caractéristiques :*

1. `wc` vient de *word count* (« compter les mots »).
2. Cette commande permet de compter le nombre de lignes, de mots ou de caractères dans le ou les fichiers passés en paramètres, ou sur l'entrée standard.
3. Cette commande est un filtre.

➥*Syntaxe :*

`wc [-c|-m|-C] [-lw] [`*`désignation_fichier`*` ...]`

➥*Description :*

- Appelée sans option, la commande `wc` affiche, sur chaque ligne, le nombre de lignes, le nombre de mots, le nombre de caractères et le nom des fichiers passés en paramètres. Ces différentes valeurs sont précédées et séparées entre elles par des espaces, dont le nombre est difficile à prévoir *a priori*.
- `-l`, `-w`, `-c` : permettent de retourner, respectivement, les nombres de lignes, de mots et de caractères.
- Autres options : cf. la description de la commande `wc`, en tapant `man wc`

➥*Exemple :*

```
wc -l exam_S3_2007_2008.txt
```

Cette commande affiche le nombre de lignes du fichier `exam_S3_2007_2008.txt` suivi du nom du fichier.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ **cmp**

➥*Caractéristiques :*

- `cmp` vient de *compare* (« comparer »).
- Cette commande effectue la comparaison des contenus de deux fichiers, ou du contenu d'un fichier et de l'entrée standard. Appelée sans option, elle affiche sur `stdout` le numéro du premier octet qui diffère. Elle n'affiche rien si les deux fichiers sont identiques.
- C'est un filtre (avec la deuxième syntaxe uniquement).

➥*Syntaxes :*

- `cmp [-l] [-s]` *`désignation_fichier`*$_1$ *`désignation_fichier`*$_2$ [*`décalage`*$_1$] [*`décalage`*$_2$]
- `cmp [-l] [-s] -` *`désignation_fichier`*$_2$ [*`décalage`*$_1$] [*`décalage`*$_2$]
  Dans la deuxième syntaxe, le caractère `-` (qui doit précéder *`désignation_fichier`*$_2$) désigne l'entrée standard. Il ne faut donc pas oublier le caractère espace entre `-` et *`désignation_fichier`*$_2$.

➥*Description :*

- `-l` : affiche les numéros de tous les octets qui diffèrent et la valeur de la différence.
- `-s` : n'affiche aucun message, mais gère les codes de retour.

➥*Exemples :*

- `cmp ‘ls | head -1‘ ‘ls | tail -1‘`
  Cette commande compare les contenus du premier et du dernier fichiers contenus dans le répertoire courant.
- `cat fich1.txt | cmp - fich2.txt > fich3.txt`
  Cette commande a le même effet que la commande suivante :
  `cmp fich1.txt fich2.txt > fich3.txt`
  Elles écrivent toutes deux, dans le fichier `fich3.txt`, le résultat de la comparaison des contenus des fichiers `fich1.txt` et `fich2.txt`

➥*Codes de retour :*

- Fichiers identiques : `0`
- Fichiers différents : `1`
- Opération interrompue pour cause d'erreur : `> 1`

➥*Remarque :* les arguments optionnels décalage$_1$ et décalage$_2$ permettent de comparer le contenu du premier fichier, à partir de l'octet numéro décalage$_1$, au contenu du deuxième fichier, à partir de l'octet numéro décalage$_2$.

☞ tr

➥*Caractéristiques :*

1. `tr` vient de *translate* (« traduire »).
2. Cette commande permet de remplacer ou de supprimer des caractères provenant de l'entrée standard. Elle fonctionne également avec les caractères de contrôle présentés dans la description de la commande `echo`, page 81.
3. Cette commande est un filtre.

➥*Syntaxes :*

- `tr -s|-d [-c]` chaîne
- `tr [-cs]` chaîne$_1$ chaîne$_2$
- `tr -ds [-c]` chaîne$_1$ chaîne$_2$

➥*Description :*

- `-s` : élimine les répétitions des caractères composant la chaîne de caractères chaîne, en n'en laissant qu'un à chaque fois.
- `-d` : supprime toutes les occurrences des caractères composant la chaîne de caractères chaîne (ou chaîne$_1$)
- Autres options : cf. la description de la commande `tr`, en tapant `man tr`

✍ 23 ➥*Exemples :*

- `tr 'abc' 'ABC'`
  Cette commande attend l'entrée de données au clavier. Si on tape `calebasse` chaque occurrence d'un `a` est remplacé par un `A`, chaque occurrence d'un `b` est remplacé par un `B` et chaque occurrence d'un `c` est remplacé par un `C`, ce qui donne dans le cas présent :
  `CAleBAsse`
- La commande `tr '\n' ' '` remplace chaque saut de ligne par un espace, parmi les caractères provenant de l'entrée standard.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarques :*

- La commande `tr 'ab' 'A'` n'effectue aucun remplacement pour le caractère `b`, et la commande `tr 'a' 'AB'` ne remplace le caractère `a` que par le caractère `A`.
- La commande `tr` ne permet pas de remplacer des chaînes de caractères. Pour cela, il est possible d'utiliser la commande `sed`, page 74.

☞ sed

➥*Caractéristiques :*

1. `tr` vient de *stream editor* (« éditeur de flux »).

2. Cette commande permet d'effectuer des manipulations complexes sur des fichiers de texte qui peuvent être de grande taille. Les noms des fichiers peuvent être passés en arguments de la commande `sed`. Sans nom de fichier, `sed` travaille sur le texte provenant de l'entrée standard. Les manipulations sont décrites par des « commande d'édition ». Une commande d'édition peut être transmise à la commande `sed` sous la forme d'une chaîne de caractères placée après l'option `-e`. Plusieurs commandes peuvent être regroupées dans un fichier dont le nom doit être placé après l'option `-f`.
3. Cette commande est un filtre.

➥*Syntaxes :*

- `sed [ -n ] commande_edition [ désignation_fichier ... ]`
- `sed [ -n ] [ -e commande_edition ] ... [ -f désignation_fichier_commandes ] ... [ désignation_fichier ... ]`

➥*Description :*

- `-e commande_edition` : demande à `sed` d'appliquer la commande d'édition `commande_edition`.
- `-f désignation_fichier_commandes` : demande à `sed` d'appliquer les commandes d'édition qui se trouvent dans le fichier `désignation_fichier_commandes`.
- Parmi les différentes commandes d'édition disponibles, la fonction de substitution est très utile. Parmi plusieurs possibilités, la syntaxe la plus employée est la suivante :
  `s/exp_reg1/exp_reg2/[g]`
  où `exp_reg1` est l'expression régulière qui décrit ce qui doit être remplacé et `exp_reg2` décrit par quoi cela doit être remplacé. Le caractère optionnel `g` à la fin de la commande signifie que toutes les occurrences doivent être remplacées. En son absence, seule la première occurrence de chaque ligne est remplacée.
- Autres options et autres commandes d'édition : cf. la description de la commande `sed`, en tapant `man sed`

➥*Exemple :* `sed -e 's/TP/TD Machine/g' sujets.txt` ✍ 24

Cette commande lit le fichier `sujets.txt` et affiche son contenu sur la sortie standard en remplaçant toutes les occurrences de la chaîne de caractères `TP` par la chaîne de caractères `TD Machine`.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

5.6 Gestion des processus

☞ ps

➥*Caractéristiques :*

1. `ps` vient de *process status* (« situation des processus »).
2. Cette commande permet d'afficher la liste des processus en cours d'activité. Lancée sans option, elle n'affiche que les processus actifs de l'utilisateur associés à la fenêtre d'où la commande est lancée.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`ps [-aAcdefjlLPy] [-g nom] [-n nom] [[-o format] ...] [-p nom] [-s nom] [-t terminaux] [-u utilisateurs] [-U nom] [-G nom]`

Les processus sont affichés, à raison d'un par ligne, suivant un format variable. Le format par défaut comprend les quatre champs suivants :

- `PID` : numéro d'identification du processus.
- `TTY` : terminal associé au processus.
- `TIME` : temps d'exécution cumulé du processus.
- `CMD` : nom de la commande associée au processus.

➥*Description :*

- `-e` : affichage des informations sur tous les processus.
- `-f` : affichage de la description complète des processus, comportant quatre champs de plus que l'affichage par défaut, et en particulier le nom du propriétaire du processus (champ `UID`) et l'heure de lancement du processus (champ `STIME`).
- `-u` `utilisateurs` : affichage des informations sur tous les processus dont le propriétaire est contenu dans la liste `utilisateurs`.
- Autres options : cf. la description de la commande `ps`, en tapant `man ps`

➥*Exemples :*

- `ps -e | grep 'nedit' | wc -l`
  Cette commande affiche le nombre de processus en cours d'exécution associés à l'éditeur de texte `nedit`
- `ps -u "crouzil durou"`
  Cette commande affiche tous les processus en cours d'exécution sur `marine`, associés à un terminal quelconque, dont le propriétaire est `crouzil` ou `durou` (quatre champs affichés par processus).
- `ps -u crouzil -f`
  Cette commande affiche tous les processus en cours d'exécution sur `marine`, associés à un terminal quelconque, dont le propriétaire est `crouzil` (huit champs affichés par processus).

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ kill

➥*Caractéristiques :*

1. `kill` signifie « tuer ».
2. Cette commande permet d'envoyer un signal à un ou à plusieurs processus dont on connaît les identificateurs.
3. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxes :*

- `kill` [`-signal`] `identificateur_processus` [...]
- `kill -l`

➥*Description :*

- `-signal` : envoi du signal `signal` au processus.
- `-l` : affichage de la liste des signaux qui peuvent être envoyés à un processus.

➥*Exemple :*

```
kill -9 1345
```

Cette commande envoie le signal `9` au processus d'identificateur `1345`. Le signal `9`, qui permet de « tuer »un processus, est le plus souvent utilisé. Le signal envoyé par défaut est `15`.

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ `exit`

➥*Caractéristiques :*

1. `exit` signifie « sortie ».
2. Cette commande permet de sortir d'un shell, en précisant ou non la valeur du code de retour.
3. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`exit` [`code_retour`]

➥*Description :* pas d'option.

➥*Exemple :* Si la commande `exit` est tapée dans la fenêtre « racine », sans qu'aucun shell fils n'ait été lancé, cela provoque une déconnexion.

➥*Codes de retour :*
- Si l'argument `code_retour` existe : la valeur (nécessairement entière) de `code_retour`.
- Sinon : le code de retour de la dernière commande précédemment exécutée.

5.7 Communication avec les autres utilisateurs

☞ `mail`

➥*Caractéristiques :*

1. `mail` signifie « courrier ».
2. Cette commande permet d'envoyer un message par l'intermédiaire de l'entrée standard, ou de lire des messages de manière interactive.
3. Elle peut apparaître en fin de branchement.

➥*Syntaxe :*

- Envoi d'un message : `mail` [`-tw`] [`-m` `type_du_message`] `adresse_destinataire` [...]
- Lecture du courrier : `mail` [`-ehpPqr`] [`-f` `désignation_fichier`]

➥*Description :* cf. la description de la commande `mail`, en tapant `man mail`

➥*Exemple :*

```
mail l2inf250@marine.edu.ups-tlse.fr < message.txt
```

Cette commande envoie le contenu du fichier `message.txt` à l'adresse indiquée.

➥*Codes de retour :*
- Pas d'erreur : `0`
- Pas de courrier, ou opération interrompue pour cause d'erreur : `> 0`

5.8 Interaction avec l'interpréteur de commandes

☞ sh

➥*Caractéristiques :*

- `sh` est l'abréviation de *shell*.
- Cette commande permet de lancer un shell de Bourne. Elle permet aussi d'exécuter un script.

➥*Syntaxes :*

- `sh` [`-acefhiknprstuvx`]
  Le shell qui est lancé lit et interprète les commandes tapées sur l'entrée standard. Pour terminer l'exécution de ce shell, il faut taper la commande `exit`
- `sh` [`-acefhiknprstuvx`] `nom_script` [`paramètre_script` ...]
  Les paramètres de `sh` sont le nom d'un script, suivi des paramètres éventuels de ce script.

➥*Description :* cf. la description de la commande `sh`, en tapant `man sh`

➥*Exemples :*

- ```
  > sh
  $ pwd
  /users/linfg/linfg0
  $ exit
  >
  ```
  Dans cet exemple, on remarque que le nouveau shell se distingue du shell initial (celui qui est lancé dès la connexion à `marine`) par un prompt différent.
- `sh nom_script.sh` [`paramètre_script` ...]
  Cette commande lance l'exécution du script contenu dans le fichier `nom_script.sh`

➥*Codes de retour :* le code de retour de la commande `sh` est le code de retour de la dernière commande interprétée par le nouveau shell (dans le premier exemple ci-dessus, c'est le code de la commande `pwd`, c'est-à-dire `0`).

☞ eval

➥*Caractéristiques :*

1. `eval` vient de *evaluate* (« évaluer »).
2. Cette commande procède en deux temps :
   - Dans un premier temps, la commande est remplacée par la liste de ses paramètres, avec interprétation des métacaractères du shell.
   - Dans un deuxième temps, cette liste de paramètres est considérée comme une commande Unix, et exécutée en tant que telle, donc en particulier avec, à nouveau, interprétation des métacaractères du shell.
3. Elle peut apparaître en début de branchement.
4. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`eval` [$\underline{\texttt{argument}_1}$ ...]

➥*Description :* pas d'option.

➥*Exemple :*

```
a=1
c='$a'
eval echo $c
```
```

Cette séquence de commandes (tapées les unes à la suite des autres, ou à l'intérieur d'un script) affichera la valeur `1`, car la première évaluation de la commande `eval` produit la commande suivante :

```
echo $a
```

En revanche, la séquence suivante afficherait `$a` :

```
a=1
c='$a'
echo $c
```

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

☞ **set**

➥*Caractéristiques :*

1. `set` signifie « placer ».
2. Cette commande permet d'affecter des valeurs aux paramètres positionnels `$1`, `$2`, `$3`, ..., `$9` du shell courant. Appelée sans paramètre, elle permet aussi d'afficher les valeurs de toutes les variables du shell courant :
   - les variables prédéfinies ;
   - les variables définies par l'utilisateur.
3. Elle peut apparaître en début de branchement.
4. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`set` [`-aefhkntuvx`] [valeur] [...]

➥*Description :* cf. la description de la commande `set`, en tapant `man set`

➥*Exemples :*

- `set alpha beta`
  Cette commande affecte la valeur `alpha` au paramètre positionnel `$1` et la valeur `beta` au paramètre positionnel `$2`
- `x=1`
  `set`
  produira un affichage du type suivant :
  ```
  DISPLAY=141.115.12.137:0
  ...
  HOME=/users/linfg/linfg0
  ...
  x=1
  ```
  Dans cet affichage, la dernière ligne concerne une variable définie par l'utilisateur, alors que toutes les lignes précédentes concernent les variables prédéfinies du shell courant.

➥*Codes de retour :* cf. la description de la commande `set`, en tapant `man set`

➥*Remarque :* la commande `set` met automatiquement à jour les paramètres `$#`, `$*` et `$@`.

☞ `shift`

➥*Caractéristiques :*
- `shift` signifie « déplacer ».
- Cette commande permet de réaliser un décalage vers la gauche (d'un rang ou plus) des valeurs des paramètres positionnels du shell courant.
- Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`shift` [n]

➥*Description :* pas d'option.

➥*Exemples :*
- Séquence utilisant la commande `shift` :
```
$ set ab
$ echo $1
ab
$ shift
$ echo $1

$
```
- Script `deca.sh` utilisant la commande `shift` :
```
echo "Avant décalage : $1 $2 $3 $4 $5 $6 $7 $8 $9"
shift
echo "Après décalage : $1 $2 $3 $4 $5 $6 $7 $8 $9"
```
Voici deux exemples d'exécution de ce script, avec des nombres de paramètres différents :
```
$ deca.sh 1 2 3 4 5 6 7 8 9 10
Avant décalage : 1 2 3 4 5 6 7 8 9
Après décalage : 2 3 4 5 6 7 8 9 10
$ deca.sh 1 2 3 4 5
Avant décalage : 1 2 3 4 5
Après décalage : 2 3 4 5
$
```
Donc `$1` reçoit la valeur de `$2`, `$2` reçoit la valeur de `$3`, etc.

➥*Codes de retour :*
- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarques :*
- Lors de l'appel de la commande `shift`, l'absence d'argument n équivaut à une valeur de cet argument égale à `1`.
- De même que pour la commande `set`, les valeurs de `$#`, `$*` et `$@` sont mises à jour automatiquement par la commande `shift` (le paramètre `$#` est décrémenté).

5.9 Entrées et sorties

☞ `read`

➥*Caractéristiques :*

1. `read` signifie « lire ».

2. Cette commande permet d'affecter des valeurs à des variables à partir de l'entrée standard. Elle est souvent utilisée à l'intérieur d'une boucle `while`.
3. Elle peut apparaître en fin de branchement.
4. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`read` nom_variable [...]

➥*Description :* pas d'option.

➥*Exemples :*

- L'exemple suivant montre que si le nombre de mots (séparés par des espaces ou des tabulations) est supérieur au nombre d'arguments de la commande `read`, alors la variable correspondant au dernier argument reçoit toute la fin de la ligne :

```
$ read a b
c'est un exemple
$ echo $a
c'est
$ echo $b
un exemple
$
```

- ```
  cat fichier | while read ligne
  do
          ...
  done
  ```
  Une séquence de ce type permet de « passer en revue »toutes les lignes du fichier `fichier`, de la première à la dernière.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

➥*Remarque :* les caractères tapés sur l'entrée standard ne peuvent pas contenir de caractère saut de ligne, car le premier saut de ligne tapé a pour effet de renvoyer le prompt.

☞ echo

➥*Caractéristiques :*

1. *echo* signifie tout simplement « écho ».
2. Cette commande permet d'afficher une ou plusieurs chaînes de caractères sur la sortie standard.
3. Elle peut apparaître en début de branchement.
4. Il s'agit d'une commande interne, qui peut donc fortement varier d'un shell à l'autre.

➥*Syntaxe :*

`echo` [chaîne ...]

➥*Description :* pas d'option.

➥*Exemple :* la commande `echo *` affiche la liste des noms de fichiers et de sous-répertoires du répertoire courant (fichiers cachés non compris), séparés par des espaces.

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`
  ```

➥*Remarques :*

- La commande `echo` reconnaît également certains caractères spéciaux commençant par \ et appelés « caractères de contrôle » : `\n` pour le saut de ligne, `\t` pour la tabulation, `\c` pour supprimer le retour à la ligne, etc.
- Pour qu'un caractère de contrôle soit interprété en tant que tel, il faut absolument qu'il apparaisse à l'intérieur d'une chaîne de caractères délimitée par des guillemets `"` ou par des apostrophes `'`. Par exemple, les commandes `echo "\n"` ou `echo '\n'` provoqueront bien l'affichage d'un saut de ligne, mais la commande `echo \n` affichera le caractère `n` (en raison de l'interprétation du métacaractère \).
- La commande `echo` provoque toujours un saut de ligne, à la fin de l'affichage, sauf si le dernier caractère est le caractère de contrôle `\c`.

5.10 Opérations arithmétiques

☞ `expr`

➥*Caractéristiques :*

1. `expr` vient de « expression ».
2. Cette commande permet d'effectuer des opérations arithmétiques et logiques, ainsi que des opérations sur les chaînes de caractères plus sophistiquées que la simple concaténation, et affiche le résultat de ces opérations sur la sortie standard.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`expr` $\underline{\texttt{argument}_1}$ $\underline{\texttt{opérateur}_1}$ $\underline{\texttt{argument}_2}$ [$\underline{\texttt{opérateur}_2}$ $\underline{\texttt{argument}_3}$ ...]

où les arguments $\underline{\texttt{argument}_1}$, $\underline{\texttt{argument}_2}$, $\underline{\texttt{argument}_3}$ sont des expressions du shell courant, et où les opérateurs $\underline{\texttt{opérateur}_1}$, $\underline{\texttt{opérateur}_2}$ appartiennent à la liste non exhaustive suivante :

- `+` (addition), `-` (soustraction), `\*` (multiplication), `/` (quotient de la division entière), `%` (reste de la division entière), si $\underline{\texttt{argument}_1}$ et $\underline{\texttt{argument}_2}$ ont comme valeurs des chaînes de caractères constituées uniquement de chiffres (et donc interprétables en tant qu'entiers).
- `=` (égalité), `!=` (différence), `\>`, `\>=`, `\<`, `\<=` (comparaisons), si $\underline{\texttt{argument}_1}$ et $\underline{\texttt{argument}_2}$ ont comme valeurs des chaînes de caractères quelconques. Le résultat de l'opération vaut `1` si la comparaison vaut VRAI et `0` si elle vaut FAUX. Attention : c'est exactement l'inverse pour le code de retour de la commande (cf. ci-dessous la rubrique « codes de retour »).
- `:` (extraction de sous-chaînes de caractères). Avec cet opérateur, l'argument $\underline{\texttt{argument}_1}$ peut avoir comme valeur une chaîne de caractères quelconque, mais l'argument $\underline{\texttt{argument}_2}$ doit être une expression régulière (cf. paragraphe 3.3, page 28). Même s'il n'est pas présent, `expr` suppose que $\underline{\texttt{argument}_2}$ commence par le métacaractère des expressions régulières `^` (début de ligne). On peut spécifier l'ensemble des caractères devant être retirés de la chaîne $\underline{\texttt{argument}_1}$ par des caractères et des métacaractères des expressions régulières, et on doit indiquer la séquence de caractères à extraire de $\underline{\texttt{argument}_1}$ grâce à l'expression régulière `\(.*\)`
  Dans le cas où la correspondance est impossible, le résultat de l'opération est la chaîne vide. Dans le cas où plusieurs correspondances sont possibles, le résultat de l'opération est la chaîne de caractères la plus longue.

➥*Description :* pas d'option.

➥*Exemples :*

- `a=2`
  `b=7`
  `a=‘expr $b % $a‘`
  `echo $a`
  Cette séquence produit l'affichage de `1`
- `chaine="taratata"`
  `sschaine=‘expr $chaine : ’\(.*\)at’‘`
  `echo $sschaine`
  Cette séquence produit l'affichage suivant :
  `tarat`
  On aurait pu écrire l'expression régulière sous d'autres formes, comme par exemple :
  `’\(.*\)at.*’`
  `’\(.*\)ata’`

➥*Codes de retour :*

- Résultat de l'opération différent de `0` et de la chaîne vide : `0`
- Autre résultat : `1`
- Syntaxe incorrecte : `2`
- Opération interrompue pour cause d'erreur : `> 2`

➥*Remarques :*

- L'utilisation des cinq opérateurs `*`, `>`, `>=`, `<` et `<=`, parmi tous ceux qui ont été décrits ci-dessus, nécessite une vigilance particulière, car `*`, `>` et `<`, étant des métacaractères du shell, doivent être protégés. Pour ce faire, on peut soit les faire précéder d'un caractère `\`, soit les entourer de délimiteurs `"` ou `’`.
- Dans le cas où la commande `expr` est lancée avec plusieurs opérateurs, il faut être vigilant, car l'ordre des opérations ne se fait ni de gauche à droite, ni de droite à gauche, mais suivant un ordre prédéfini de priorité des opérateurs.

5.11 Évaluation de conditions

☞ `test`

➥*Caractéristiques :*

1. `test` a une signification explicite.
2. Cette commande permet d'effectuer des tests de comparaison, en renvoyant le code de retour `0` lorsque le résultat est vrai, et une autre valeur sinon. Cette commande est surtout utile pour les structures de contrôle, comme la structure de contrôle de condition. Elle ne fournit pas de résultat sur sa sortie standard, mais elle renvoie un code de retour qui est égal à 0 si le test est VRAI ou à 1 si le test est FAUX.

➥*Syntaxes et description :*

- `test` comparaison ou `[` comparaison `]`
  Il faut vraiment taper les crochets et respecter les espaces, avec cette deuxième écriture.
  La comparaison comparaison peut être la combinaison booléenne de plusieurs comparaisons élémentaires :
  - comparaison$_1$ `-a` comparaison$_2$ pour le `ET` logique.
  - comparaison$_1$ `-o` comparaison$_2$ pour le `OU` logique.
  - `!` comparaison$_3$ pour le `NON` logique.

Des parenthèses précédées de caractères \ (pour protéger les parenthèses, qui sont des métacaractères du shell) peuvent être nécessaires en cas de combinaisons un peu plus compliquées, pour préciser les priorités, comme dans : `! \( comparaison1 -o comparaison2 \)`
Chaque comparaison élémentaire doit être écrite avec la syntaxe suivante :
`expr1 comparateur expr2`
où `expr1` et `expr2` désignent des « expressions »du shell courant (cf. paragraphe 3.6).
Le comparateur `comparateur` s'écrit :

- si les valeurs de `expr1` et `expr2` sont des chaînes de caractères constituées de chiffres :
  - `-eq` désigne l'égalité (abréviation de `eq`ual)
  - `-ne` désigne la non égalité (abréviation de `n`ot `e`qual).
  - `-gt` signifie « strictement supérieur »(abréviation de `g`reater `t`han).
  - `-ge` signifie « supérieur ou égal »(abréviation de `g`reater or `e`qual).
  - `-lt` signifie « strictement inférieur »(abréviation de `l`ess `t`han).
  - `-le` signifie « inférieur ou égal »(abréviation de `l`ess or `e`qual).
- si les valeurs de `expr1` et `expr2` sont des chaînes de caractères quelconques :
  - `=` désigne l'égalité
  - `!=` désigne la non égalité.

Il faut bien prendre garde à ne pas confondre les comparateurs `=` et `-eq`. Alors que `test 2 = 02` retourne FAUX, que `test 2 -eq 02` retourne VRAI et que `test baba = bobo` retourne FAUX, ce qui semble tout à fait cohérent, on sera surpris de constater que `test baba -eq bobo` retourne VRAI. Il ne faut donc pas utiliser le comparateur `-eq` pour comparer des chaînes de caractères.
Pour tester si la variable `chaine` a comme valeur la chaîne vide, on ne peut pas écrire la commande `test $chaine = ""` car, si cette chaîne était vraiment vide, après évaluation des métacaractères, cette commande serait évaluée sous la forme `test = ""`, écriture qui est incorrecte. Pour obtenir une écriture correcte, il est alors nécessaire d'entourer l'expression contenant le nom de la variable de guillemets : `test "$chaine" = ""`
Mais il est plus pratique d'utiliser la deuxième syntaxe de la commande `test` :

- `test $variable`
  Dans cette écriture, la commande `test` teste si la variable `variable` n'a pas pour valeur la chaîne vide. Si tel est le cas, la commande `test` retourne `0` (c'est-à-dire VRAI).
  Avec cette deuxième syntaxe, les opérateurs logiques `-a`, `-o` et `!` nécessitent l'utilisation de guillemets partout où peut apparaître la chaîne vide.
- `test -f|-d|-r|-w|-x désignation` [...]
  Avec cette syntaxe, la commande `test` vérifie si `désignation` est le nom d'un fichier (option `-f`) ou d'un répertoire (option `-d`), et si `désignation` est accessible en lecture (option `-r`), en écriture (option `-w`) ou en exécution (option `-x`) par l'utilisateur. Les opérateurs logiques `-a`, `-o` et `!` s'appliquent à cette troisième syntaxe.

➥*Exemples :*

- `test $# -eq 2 -a $1 -gt 0`
  Cette commande teste si le nombre de paramètres positionnels du shell courant ayant reçu une valeur est égal à `2`, et si le paramètre `$1` a une valeur entière strictement supérieure à `0`.
- ```
  test $chaine
  if [ $? -ne 0 ]
  then
          ...
  ```
  Cette séquence permet de tester si la variable `chaine` a pour valeur la chaîne vide. Le même test peut être effectué plus simplement en utilisant la séquence suivante :
  `if [ ! "$chaine" ]`
  ```

```
then
        ...
```

- `test -f nom -a -r nom`
  Cette commande teste si `nom` est le nom d'un fichier accessible en lecture.

➥*Codes de retour :*

- `comparaison` évaluée à VRAI : `0`
- `comparaison` évaluée à FAUX : `1`
- Opération interrompue pour cause d'erreur : `> 1`

5.12 Accès à la date et à l'heure

☞ `date`

➥*Caractéristiques :*

1. `date` a une signification explicite.
2. Cette commande affiche la date.
3. Elle peut apparaître en début de branchement.

➥*Syntaxe :*

`date` [`-u`] [`+format`]

➥*Description :* cf. la description de la commande `date`, en tapant `man date`

➥*Exemple :*

La commande `date` affiche la date par exemple sous la forme suivante :

```
Mon Dec 8 21:05:47 MET 1997
```

➥*Codes de retour :*

- Pas d'erreur : `0`
- Opération interrompue pour cause d'erreur : `> 0`

Index