Université de Caen Normandie
Département de mathématiques et d'informatique
L2 Informatique, 2017–2018
Unité : INF3B1

Système

L2 Système : Environnement de travail informatique

François Rioult

Table des matières

1. Généralités

Ce cours introduit l'environnement de travail informatique grâce au système Linux, l'interprêteur de commande `bash`*, quelques utilitaires de manipulation de données (expression régulières,* `sed`*,* `awk`*), une introduction à la gestion de processus, de version et la production de documents. En aucun cas il ne s'agit d'un manuel de référence pour Linux ; c'est plutôt la boîte à outils minimale que devrait posséder un informaticien moderne, bien que la majorité des concepts date de plus de trente ans. Cependant, les outils évoqués sont toujours d'actualité et on se limitera ici à l'utilisation des classiques éprouvés, sans tenir compte des derniers développements exotiques.*

Une grande majorité des applications informatiques est fondée sur le principe de la *transformation de documents électroniques*. En particulier, toutes les activités de publication y font appel. Par exemple, on imagine un journaliste qui écrit un article sous forme électronique. Cet article pourra être publié sur un site internet de nouvelles, dans un journal, dans un livre, dans une encyclopédie électronique, etc.

L'informatique actuelle permet d'automatiser ces transformations, en séparant le *fond* (la rédaction du journaliste) et la *forme* choisie pour la présentation. Les traitements font appel à des tâches élémentaires qui sont combinées pour fournir le résultat. Pour cela, une opération complexe est découpée en éléments plus simples jusqu'à pouvoir faire réaliser ces opérations par une machine. On procédera donc généralement au découpage d'un projet complexe selon une *analyse descendante*. La réalisation suit une *analyse montante*.

Un ordinateur est une machine qui ne sait effectuer que des tâches très élémentaires, et en aucun cas les tâches très complexes que peuvent concevoir les humains. Malgré tout, l'ordinateur exécute les ordres extrêmement rapidement et peut considérer des quantités colossales d'information. L'humain et l'ordinateur cohabitent donc généreusement, le premier faisant l'effort d'organiser ses idées pour les rendre accessibles au second.

1.1. Architecture de l'ordinateur

Un ordinateur est constitué de *périphériques d'entrée et de sortie*, dirigés par un *processeur*. L'information circule entre ces éléments matériels sous forme de *flux* unidirectionnel. Pour qualifier précisément les relations les sources et les destinataires de flux, la terminologie *client/serveur* est employée. Par exemple, un utilisateur humain qui demande un service à une machine sera le client, la machine sera le serveur. Lorsque le navigateur demande un document sur site, le navigateur est client, le site serveur.

Périphériques d'entrée

Les périphériques d'entrée sont chargés de recevoir les informations données par l'utilisateur. Par exemple, le clavier transforme un ordre réalisé à l'aide de la pression sur une touche en une série d'impulsions électriques. La souris retranscrit les mouvements qu'elle subit et l'état de ses boutons.

Un micro convertit les sons en signal électrique ; un appareil photo, un scanner ou une (web-)caméra traitent les images.

Périphériques de sortie

L'écran est le principal périphérique de sortie, et compose des images à l'aide de signaux électriques. L'imprimante permet la même chose sur le papier.

Périphériques d'entrée-sortie

Les périphériques qui réalisent à la fois des opérations d'entrée et de sortie sont généralement dévolus au stockage de l'information. En effet, le stockage implique que l'on puisse écrire l'information dans un endroit où elle sera conservée : c'est un périphérique de sortie. Lorsqu'on le désire lire l'information, c'est un périphérique d'entrée. On parle d'accès aléatoire [1] quand on peut indifféremment lire ou écrire. En particulier, la mémoire vive de l'ordinateur est un périphérique de stockage qui conserve les données tant que la machine est allumée : c'est la RAM (random acces memory). Un CD-ROM est un moyen de stockage sur lequel on ne peut que lire (read only memory).

Les périphériques communicants (modem, carte réseau) permettent d'établir une relation physique entre deux machines. Dans ce cas, les échanges d'information auront lieu dans les deux sens et ces périphériques servent à la fois d'entrée et de sortie.

La manette de jeu vibrante est le dernier exemple que nous utiliserons. Si le joueur appuie sur les boutons, c'est un périphérique d'entrée. Lorsque la manette vibre, c'est un périphérique de sortie.

1.2. Vocabulaire

Les signaux traités par un ordinateur sont exclusivement sous forme de signal électrique numérique. Par exemple, en imaginant que chaque touche a un numéro spécifique, le clavier ne transmet pas ce numéro mais une suite de signaux électrique d'une tension de 0 ou 5 Volts.

1. *random* en Anglais

Toute information est donc codée par une suite de 0 ou de 1, appelés *bit*. Pour plus de simplicité, les bits sont groupés par 8 et forment des octets (*byte*). Un octet peut coder $2^8 = 256$ nombres différents.

Le processeur communique avec ses périphériques en utilisant des octets qui respectent le *protocole*. Il ne sait effectuer qu'une tâche électronique basique à la fois sur ces octets : lire un octet sur un périphérique, le manipuler (additions, soustractions, comparaisons avec un autre octet), puis l'écrire sur un périphérique. Cette simplicité montre rapidement ses limites lorsqu'il s'agit de réaliser un programme complexe comme un navigateur.

Cette tâche aurait la même complexité que de réaliser un gâteau au chocolat en partant de protons, et neutrons et d'électrons. Il est quand même plus facile et intéressant de travailler directement avec du sucre, des œufs, du beurre, de la farine et du chocolat. C'est pourquoi on utilisera en informatique des utilitaires qui réalisent ces opérations élémentaires. La combinaison de ces bases fournit des utilitaires plus évolués, qui peuvent à leur tour être combinés.

Le *système d'exploitation* a pour but de proposer ces bases de travail à l'utilisateur et le dispense de la réalisation de tâches fastidieuses. Il fournit directement des outils de travail puissants. Nous utiliserons `Linux` qui est générique des autres systèmes professionnels, mais le vocabulaire développé ici ne lui est pas particulier.

La notion centrale que nous utiliserons est celle de *fichier*. Ce concept désigne toute ressource sur laquelle on peut lire ou écrire des octets. Pour l'utilisateur classique, il s'agit d'une portion du disque dur ou d'un CD-ROM qui contient des informations (texte, image, son) codées sous forme d'octets. Plus généralement, tout périphérique est considéré par le système d'exploitation comme un fichier dans lequel il va lire (entrée) ou écrire (sortie). Cependant, dans la suite de cette présentation, nous utiliserons le terme de fichier pour désigner une portion d'un périphérique de stockage.

Le système d'exploitation est en fait un programme particulier qui exécute *simultanément* les différents programmes demandés par l'utilisateur. Chaque programme est stocké dans un fichier qui contient la suite d'instructions à exécuter. Le système prend en charge l'allocation des ressources du processeur (mémoire, temps de calcul) pour équilibrer la charge de travail entre les différents *processus*. L'impression de simultanéité est obtenue en allouant des créneaux de calcul très courts, de quelques millièmes de secondes, puis en passant d'un tâche à l'autre.

Le vocabulaire courant de l'informatique est le suivant :

source : on dira « le » (fichier) source pour le fichier contenant le code des instructions à exécuter ;

exécutable : programme ou source que le système peut exécuter ;

shell : (coquille) désigne le langage des instructions compréhensibles par le système ;

terminal : c'est un programme particulier qui lit les instructions de l'utilisateur sur le fichier qui correspond au clavier, les exécute et écrit le résultat dans le fichier de l'écran ;

processus : tâche élémentaire réalisée par le système ;

entrée/sortie standard : chaque processus possède par défaut un fichier d'entrée particulier dans lequel il va lire et un fichier de sortie où écrire. Par exemple, l'entrée standard d'un terminal est le clavier, la sortie standard est l'écran ;

sortie standard d'erreur : chaque programme peut potentiellement se retrouver en situation d'erreur (par exemple, l'imprimante n'est pas allumée) et doit signaler les erreurs. Il les écrit dans un fichier, par défaut la sortie standard.

1.3. Codage

Le processeur ne sait gérer que des bits ou des ensembles de bits de taille variable. Les calculateurs actuels sont des machines 32-bits ($2^{32} = 4G$) ou 64-bits ($2^{64} = 16GG$). La mémoire vive contient des octets qui ont une adresse (un nombre appelé *mot*) accessible par le processeur, dont le nombre de bits limite l'espace d'adressage.

L'être humain ne manipule pas que les nombres entiers (*integer*). Il aime aussi les caractères (*char*) notés entre apostrophes (ou quotes) `'...'`, les chaînes de caractères (*string*) notées entre apostrophes ou guillements anglais `"..."`, les nombres flottants (*float*). Tous forment les *types* de base en informatique. Les informaticiens ont donc inventé des codages pour les caractères. Le plus célèbre est l'ASCII, qui attribue un caractère à chaque valeur différente pour un octet. La première partie du code (de 0 à 127, soit en n'utilisant que 7 bits) est internationalement standardisée. La seconde partie concerne les spécificités locales, et sont définies par des normes. Malgré tout, 256 valeurs différentes sont insuffisantes et il faut se tourner vers d'autres standards, comme l'unicode qui permet de s'affranchir de ces contraintes.

Certains caractères ont un fonction particulière, par exemple `$`. Lorsque l'on voudra l'utiliser en tant que caractère et non pas pour sa fonction, il faudra le *quoter* en le précédant du caractère `\`. Inversement, certains caractères précédés de la quote revêtent une signification particulière, par exemple `\n` signifie un passage à la ligne.

1.4. Variable, argument, paramètre, fonction

Une variable est la représentation informatique d'une abstraction humaine. Elle est dotée d'un nom (une chaîne de caractères ne commençant pas par un chiffre) et contenant une valeur. Par exemple, *systeme* = "*linux*" désigne une variable de nom `systeme` et de valeur `"linux"` (une chaîne de caractère). Il existe des variables de tout type, et ce concept sera plus tard étendu à la notion d'*objet*, qui permettent de définir des types complexes génériques : les *classes*.

Une commande informatique est généralement exécutée avec des *arguments* et des *paramètres*. Un argument indique à la commande sur quelles variables elle doit travailler. Un paramètre ou *option* est une quantité fixée maintenue constante dont dépend également le résultat mais spécifie son mode d'obtention. Considérons par exemple le programme *date* qui réalise des opérations sur les dates. Appelé sans argument ni paramètre, il écrit la date actuelle sur la sortie standard au format français : `mercredi 30 août 2017, 12:02:03 (UTC+0200)`. Pour obtenir la date au format standard *linux*, on utilise le paramètre `-R` (le signe `-` indique un paramètre à une lettre, le signe `--` est utilisé pour les paramètres à plusieurs lettres) et on obtient

`Wed, 30 Aug 2017 12:02:49 +0200`. Si maintenant on spécifie un argument, le programme va fixer la date du système à la valeur indiquée : `date --set='Fri, 13 May 2005 15:08:00 +0200'`.

Dans la plupart des langages de programmation, la notation utilisée pour l'utilisation d'une commande complexe est la notation fonctionnelle. On dira qu'on appelle une fonction et que l'on s'intéresse à sa valeur de retour. Par l'exemple précédent, on pourra noter `date('Fri, 13 May 2005 15:08:00 +0200')`.

2. Linux

Linux est une implémentation libre des standards commerciaux Unix des années 70 pour les systèmes d'exploitation. Le « noyau » est le cœur du système, et contient les primitives nécessaire pour la gestion du processeur, en terme d'allocation des ressources et d'ordonnancement des différents processus.

2.1. Le terminal, la ligne de commande, le shell bash

À ce jour et en dépit des interfaces graphiques, une majorité d'informaticien s'accorde sur le fait que l'interaction principale avec la machine passe par le clavier, au travers de commandes à exécuter saisies par l'utilisateur. On utilisera pour cela un programme nommé *terminal*, généralement pourvu d'une interface graphique pour copier/coller et ouvrir d'autre terminaux mais pas plus. Le mot *terminal* date des années lointaines où plusieurs consoles (clavier + écran) étaient branchées sur une même machine avec laquelle elles échangeaient de simples caractères.

En réalité, le terminal fait tourner un interprêteur de commande, en particulier bash qui est le plus répandu sur les systèmes Linux. Cet interprêteur exécute le programme suivant :

1. afficher l'invite de commande ou *prompt*
2. donner le contrôle du clavier à l'utilisateur, jusqu'à ce qu'il presse la touche Entrée
3. analyser le texte fourni par l'utilisateur et le transformer en une commande à exécuter par le système
4. demander au système l'exécution de la commande
5. afficher le résultat de la commande
6. retourner en ligne 1.

```
teraquick:[~]$ pwd
/export/home/rioultf
teraquick:[~]$ ls
bashrc     drush-backups     Modèles  rib.pdf       Téléchargements
bashrc~    examples.desktop  Musique  seaborn-data  these
Bureau     git               PDF      software      tmp
doc        glassfish-4.1.1   perso    Steam         Vidéos
Documents  Images            Public   svn           web
teraquick:[~]$ cd Images/
```

```
teraquick:[Images]$ ls
francois_rioult_2016.jpg
teraquick:[Images]$ cd ..
teraquick:[~]$ ls Images/
francois_rioult_2016.jpg
teraquick:[~]$
```

`bash` peut exécuter des commandes qui sont disponibles dans des fichiers (programmes, scripts `bash`) mais contient également ses propres commandes, dites *builtin* : déplacement dans l'arborescence de fichier, gestion des alias, des variables, de l'interaction avec l'utilisateur.

Le notion de déplacement dans l'arborescence est une abstraction de la gestion du *répertoire courant*), dont vont dépendre le nom des fichiers relatifs (voir section suivante). L'une des tâches de base consiste donc à modifier ce répertoire courant (en *changer*) à l'aide de la commande `cd`. On se repère dans la navigation en contrôlant le répertoire indiqué par le prompt puis visualisant la liste des fichiers du répertoire courant avec la commande `ls`.

Le terminal est très efficace pour manipuler des fichiers, lancer des programmes, automatiser des tâches : c'est la base de l'organisation des tâches. En outre, dans les contextes d'informatique dématérialisée, le seul moyen de piloter une machine à distance est d'utiliser un terminal sous `bash`.

`bash` fournit de puissant outils pour oublier la souris et rendre les interactions claviers performantes en limitant la saisie au strict minimum (entre une ou trois lettres pour chaque mot) :

- gestion de l'historique : on y navigue avec les flèches bas/haut, à l'intérieur d'une ligne, on peut également y effectuer une recherche
- la complétion : lorsque quelques lettres sont saisies, l'appui de la touche `tab` déclenche la recherche de toutes les solutions possibles. Si une seule solution existe, elle vient compléter la commande. Si plusieurs solutions existent, un deuxième appui sur `tab` les affiche et il suffit de taper d'autres lettres pour raffiner.

`bash` est un langage complet avec gestion des variables, boucles, fonctions. Ce n'est pas pour cela qu'il faut programmer avec, car il n'est pas très performant pour faire du calcul. Son usage doit être limité à la gestion de tâches, des itérations, de courts scripts, des enrobeurs (*wrapper*) pour convertir des formats.

En outre, sa syntaxe est assez horrible et exigeante, archaïque par rapport aux langages actuels, mais ce n'est pas une raison suffisante pour faire l'impasse :

- on a besoin de peu d'éléments du langage pour se sortir de la grande majorité des situations ;
- c'est souvent l'un des seuls langages disponibles sur les distributions de base ;
- il permet de faire en très peu de temps des manipulations très puissantes d'architectures informatiques, indispensables en milieu professionnel ;
- il est très performant pour effectuer les tâches pour lesquelles il est conçu, c'est à dire lancer des processus ;
- il faut arrêter de faire des scripts PHP ou Python pour manipuler des fichiers et lancer des tâches, `bash` est là pour cela.

/	racine (root) du système
/bin	fichiers binaires exécutables des principales commandes Unix (ex /bin/bash)
/boot	fichiers nécessaires au démarrage
/dev	fichiers spéciaux utilisés pour la gestion des périphériques (donc des partitions du disque dur)
/proc	point de montage d'un système de pseudo-fichiers qui donnent des informa- tions sur les processus, le matériel, etc.
/etc	fichiers de configuration du système (/etc/passwd, /etc/syslog.conf, /etc/fstab)
/lib	bibliothèques partagées (shared libraries) par les commandes de /bin
/sbin	fichiers binaires pour l'administartion du système (ne pouvant en général être exécutées que par l'utilisateur root)
/tmp	répertoire de fichiers temporaires détruits sans préavis et / ou à intervalles réguliers
/usr	sous-arbre (éventuellement partagé avec NFS) découpé en sous-répertoires comme la racine : /usr/bin, /usr/lib etc.
/usr/include	fichiers pour compilateurs C ou C++
/usr/man	contient les pages du manuel en ligne de la commande man
/usr/doc	contient différentes documentations sur les programmes du système (en particulier les HOWTOS et les FAQ)
/usr/share	fichiers indépendants de l'architecture du système (manuels, sources)
/usr/local	programmes ou logiciels non contenus dans la distribution de Linux qui est utilisée
/home	contient les répertoires utilisateurs (appelé /users au département informatique pour des raisons historiques)

TABLE 2.1. – Répertoires de `Linux`

2.2. Système de fichiers

2.2.1. Architecture

`Linux` travaille sur un système arborescent de fichier dont la racine est / (resp. C : \ sous `windows`), le / étant le séparateur qui permet de marquer les niveaux hiérarchiques (resp. \). La table 2.1 décrit les principaux répertoires d'une architecture `Unix`.

Cette arborescence est virtuelle ou *logique*, car elle ne correspond pas à une hiérarchie physique. Certains de ses éléments sont montés à une adresse précise, grâce au informations du fichier `/etc/fstab`.

```
# <file system>                 <mount point>   <type>      <options>
proc                            /proc           proc        defaults
```

```
# /
/dev/vg/root                    /               xfs         allocsize=64k,relatime,usrquota
# /export
/dev/vg/export                  /export         xfs         relatime,usrquota,prjquota
# /boot
/dev/md10                       /boot           xfs         defaults

# SWAP : activé par /root/clone/scripts/enable_swap
#/dev/vg/swap none swap defaults 0 0

#/dev/scd0                       /media/cdrom0   udf,iso9660 user,noauto,exec,utf8
```

2.2.2. Partitionnement

Pour installer un système Linux, on doit disposer d'une partition qui accueillera la racine /. La taille commune est de 5 à 10 Go. Lorsque l'on dispose de peu de mémoire, il faut également une partition d'échange (*swap*) qui sert à stocker temporairement le contenu de la mémoire lorsqu'elle est épuisée. Sa taille doit être adaptée à la capacité de la mémoire, son type est `linux-swap`. Enfin, il faut une partition pour stocker ses données personnelles, de type `ext3`, montée sur `/home`. On choisit en général une partition indépendante de celle du système, ce qui permet de conserver ses données en cas de remplacement du système.

Si l'on souhaite faire cohabiter un système Windows, il faut commencer par installer celui-ci, ensuite le Linux. Ce dernier se chargera d'installer un système d'amorçage qui permet de choisir le système au démarrage. Si l'on installe Linux avant Windows, Windows efface le système d'amorçage présent dans le MBR (master boot record). Le type d'une partition Windows est souvent `ntfs`, parfois inaccessible en écriture depuis Linux. On pourra préférer le type `fat32`.

Sur un disque dur, il n'est possible de créer que quatre *partitions primaires*. Pour une machine accueillant Linux et Windows, si l'on souhaite avoir des partitions pour ses données personnelles pour chaque système, il faudrait pouvoir disposer de cinq partitions, ce qui est impossible. On réservera donc les partitions primaires pour Windows et Linux. Puis on créera une *partition étendue*, pouvant accueillir autant de *partitions logiques* que souhaité.

2.2.3. Noms et chemins de fichiers

Il existe deux *chemins* pour référencer un fichier :

1. le chemin *absolu*, qui commence à la racine, par exemple `/etc/fstab`
2. le chemin *relatif* où `..` représente le répertoire parent. À partir du répertoire `/users/2006`, le nom relatif de `/etc/fstab` est `../../etc/fstab`.

Le nom d'un fichier (ex. `index.html`) comporte généralement une *extension* (ici `.html`). Elle permet d'indiquer le type de contenu, même si cela n'est pas obligatoire.

Le répertoire d'accueil de l'utilisateur, spécifié dans `/etc/passwd`, est noté `~`.

2.3. Manipulations de fichiers

Le `shell` utilise la notion de répertoire courant, qui spécifie l'adresse dans l'arborescence où les commandes seront exécutées. La table 2.2 décrit les commandes usuelles.

`pwd`	affiche le répertoire courant ;
`ls [adresse]`	affiche les fichiers contenus à l'adresse. L'option `-l` affiche un format long (droits d'accès, propriétaire, taille, date. `-a` affiche tous les fichiers, même ceux qui commencent par `.` ;
`cd`	change le répertoire courant. Sans argument, équivaut à `cd ~`. `cd -` passe au répertoire courant précédent ;
`mkdir repertoire`	crée un répertoire ;
`rmdir repertoire`	détruit un répertoire vide ;
`mv source... cible`	déplace ou renomme un fichier ;
`cp source... cible`	copie un fichier. Avec l'option `-r`, la copie est récursive, c'est-à-dire que c'est l'ensemble de l'arborescence source qui est copiée, avec ses sous répertoires ;
`rm source...`	détruit un fichier ;
`touch fichier`	crée un fichier vide ou le met à l'heure actuelle s'il existe ;
`ln -s source cible`	crée un lien symbolique ;
`locate pattern`	recherche un fichier dont le nom absolu contient `pattern`, en accédant à une base de données ;
`find adresse_depart -name pattern`	parcourt d'arborescence à partir de `adresse_depart` à la recherche d'un fichier de nom `pattern`.

Table 2.2. – Commandes de manipulation de fichiers

2.4. Droits d'accès aux fichiers

Les utilisateurs d'une machine possèdent un numéro d'identifiant et un numéro de groupe indiqués par la commande `id`, le nom est également fourni par `whoami`. Le fichier `/etc/passwd` contient la table qui met en relation le nom d'utilisateur (*user login*), ses identifiant, groupe, nom complet, répertoire de démarrage (`~`), programme de commande.

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

```
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
rioultf:x:16859:2001:Rioult Francois:/export/home/rioultf:/bin/bash
mysql:x:154:164:MySQL Server,,,:/nonexistent:/bin/false
nvidia-persistenced:x:102:103:NVIDIA Persistence Daemon,,,:/:/sbin/nologin
```

Chaque fichier et répertoire exige certains droits d'accès pour être lu (`r`), écrit (`w`) et exécuté (`x`) pour un fichier ou traversé pour un répertoire. Ces trois droits sont définis pour le propriétaire du fichier, son groupe et les autres. Les droits d'accès par défaut lors de la création d'un fichier sont gérés par la commande `umask masque`.

Dans les systèmes `Unix`-like, un utilisateur particulier a tous les privilèges et a l'exclusivité sur certains fichiers. Il s'agit du super utilisateur `root` (racine, $id = 0$), et seules les personnes chargées de l'administration de la machine peuvent l'incarner. En particulier, `root` est le seul utilisateur qui peut lire et modifier le fichier `/etc/shadow` contenant le cryptage des mots de passe.

```
root:$6$g3AU4l/V$JxprXeGDk2FzvmSeGjrQ2vj.7Vg17qIiCsP3XkWGhyj6TlDsm/O7zJpEWEPthZA7msc59yNQSuI
rioultf:$6$/czJgU5l$PeMtg6yNROHExWL5s4/utA5LcAo/ELA/Zs3ZznOdCzprIZaZocEIeczMC8muzySqQS1NS91.
```

2.5. Amorçage - bootstrap

Un ordinateur sous `Linux` est amorcé par les étapes suivantes :

1. un gestionnaire de boot (par exemple `LILO` (`Linux` loader) ou `grub`) permet à l'utilisateur de choisir la *partition* du disque dur qui contient le système à amorcer (ce qui permet de faire cohabiter sur une même machine `Windows` et `Linux`). Pour ce dernier, le gestionnaire indique l'adresse du noyau à exécuter ;
2. le noyau est un programme qui contient l'essentiel pour le système d'exploitation. Il s'agit principalement des méthodes d'allocation des ressources ;
3. une fois opérationnel, le noyau exécute les tâches programmées pour le démarrage dans certains fichiers de configuration. Les différents services sont activés (son, vidéo, réseau, interface graphique, bases de données, web) ;
4. pour finir, la machine se met à la disposition de l'utilisateur en lui proposant de démarrer une interface graphique (type `Windows`) ou de commande au clavier (*shell* – ou coquille – dans un terminal, on parle de *console*).

2.6. Interpréteur de commandes

L'interpréteur de commande est la plus simple interface de dialogue avec la machine. L'utilisateur saisit une commande avec le clavier et la valide avec la touche `return`. La commande est exécuté et produit une trace à l'écran, puis redonne la main à l'utilisateur.

La syntaxe d'une commande est la suivante :
`commande` [*option* ...] [*argument* ...]

Exemple : `wc -l /etc/fstab`

Les options sont généralement précédées d'un tiret lorsqu'elles sont définies par une lettre, et de deux lorsqu'elles utilisent plusieurs lettres. Les options courantes sont :

- **`-i`** interactif, demande des confirmations à l'utilisateur quand nécessaire ;
- **`-f, -force`** force l'exécution ;
- **`-h, -help`** affiche l'aide ;
- **`-v, -verbose`** mode bavard (verbieux).

Une aide est disponible pour les commandes classiques en utilisant `man commande`.

2.6.1. Expressions régulières

Certains caractères ont un rôle particulier et remplacent les autres caractères qui permettent de faire correspondre la commande saisie avec les adresses potentielles. `*` remplace tous les caractères, `?` remplace un caractère. `[a-z]` indique une lettre, `[^abc]` matche un caractère différent de `a`, `b`, `c`. Ainsi, la commande `ls [st]?ell.p*[^f]` matchera avec tout fichier dont le nom commence par `s` ou `t`, dont la troisième lettre est un `e`, les deux suivantes un `l`, l'extension commence par `p` et ne termine pas par `f`. Le nom `shell.ps` convient.

2.6.2. Raccourcis clavier

Certaines combinaisons de touches produisent l'effet particulier indiqué à la table 2.3.

`^D`	termine le shell (équivaut à `exit`
`^C`	interrompt la commande en cours
`tab`	complétion de la commande en cours d'édition
`^L`	efface le terminal
↑, ↓	commande précédente, suivante
`^R`	recherche dans l'historique des commandes
`^S`	gèle l'affichage
`^Q`	dégèle l'affichage
`^Z`	stoppe la commande courante. `bg` envoie cette tâche en arrière plan, `fg` la remet au premier plan

TABLE 2.3. – Combinaisons de touches pour l'interpréteur de commandes.

`c`	création de l'archive
`x`	extraction de l'archive
`f`	précise le nom du fichier de destination (toujours utilisée)
`z`	compresse ou décompresee
`t`	affiche le contenu de l'archive

TABLE 2.4. – Commandes pour l'archivage avec `tar`

D'une façon générale, les touches et commandes d'édition d'`emacs` sont valables sur la ligne de commande.

On dispose de 6 consoles, accessibles avec les touches `Ctrl-Alt-F1` à `Ctrl-Alt-F6`. La dernière console `Ctrl-Alt-F7` est le terminal graphique.

2.6.3. Commandes particulières

`!!` rappelle la dernière commande, `!c` la dernière commande commençant par `c`. `history` liste les 500 dernières commandes. On peut gérer la taille de l'historique en agissant sur les variables suivantes :

```
# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=100000
HISTFILESIZE=2000000
```

2.6.4. Usage de la souris

Dans cet environnement exclusivement textuel, la souris ne permet pas d'indiquer la position du curseur de saisie. En revanche, elle est utilisée pour réaliser des sélections (un *drag* en maintenant le bouton gauche appuyé). Sous `Linux`, le contenu de cette sélection est copié dans le tampon. L'appui sur le bouton du milieu simule la saisie au clavier du contenu du *buffer*. Attention au comportement de ce mécanisme en présence de caractères accentués.

2.7. Archivage

La commande `tar [commande] archive fichier...` permet d'archiver des portions de l'arborescence dans un seul fichier. Les commandes usuelles sont à la table 2.4.

Les conventions veulent qu'un fichier d'archive ait l'extension `.tar`, et `tar.gz` ou `.tgz` lorsqu'il est compressé. En général, on se cantonnera à l'usage `tar cvfz repertoire.tgz repertoire` pour archiver, `tar xvfz repertoire.tgz` pour désarchiver.

On dispose également des utilitaires `zip`, `unzip`, `gzip`, `gunzip`, `unrar`.

2.8. Impressions (obsolète)

La liste des imprimantes est dans le fichier `/etc/printcap`. La commande `lpr fichier` imprime les fichiers au format texte, PostScript (`ps`), PDF ou HTML. `lpq` affiche l'état de la queue et `lprm` permet d'annuler une tâche d'impression.

Dans la pratique, seul le format PostScript donne des résultats corrects. L'utilitaire `a2ps fichier -o fichier.ps` convertit un fichier texte en PostScript selon des feuilles de style. `mpage -4 fichier.ps > fichier4.ps` met 4 pages sur une.

2.9. Configuration

Lorsque le shell démarre, il hérite de l'environnement du processus qui l'exécute. `env` liste cet environnement. En particulier, la variable `PATH` indique les adresses où l'interpréteur de commande va chercher les exécutables.

Puis, il exécute le fichier `~/.bash_profile`, qui lui même appelle `~\.bashrc`. Ce fichier contient des définitions de variables, comme le `PATH` et le *prompt* (invite de commande) dans la variable `PS1`.

Des *alias* permettent de définir des macro-commandes. Elles sont interprétée en ligne de commande, mais pas dans un fichier exécutable. Ex. `alias mv='mv -i'`.

Pour modifier l'environnement d'un terminal, il faut donc éditer le fichier `~/.bashrc`. Pour que les changements soient pris en compte dans le terminal courant, il faudrait copier/coller le contenu du `~/.bashrc` dans la console, ce que les commandes suivantes permettent :

- `source ~/.bashrc`
- `. ~/.bashrc`

3. Shell

Nous abordons maintenant quelques commandes évoluées. Nous définirons également des `scripts`, qui rassemblent dans un fichier une liste de commandes. Celles-ci sont séparées par `;` ou par un saut de ligne. Elles sont enchaînées sur une même ligne par `&&` (et), et conditionnellement avec `||` (ou).

Un fichier shell `script.sh` est exécuté grâce à :
- `. script.sh` ou `sh script.sh` s'il n'a pas les droits d'exécution x
- `./script.sh` s'il possède les droits d'exécution.

Quelques commandes `shell` :

`echo` écrit un texte ou le contenu de variables sur la sortie standard. L'option `-n` supprime le saut de ligne après l'affichage. Les caractères suivants ont un comportement spécial :
- `\c` pas de saut de ligne ;
- `\t` tabulation ;
- `\n` saut de ligne.

`wc` compter les caractères, les mots, les lignes ;

`cat` afficher, concaténer des fichiers ;

`paste` coller deux fichiers l'un à côté de l'autre (`cat` en vertical) ;

`head -n nlignes` affiche les `nlignes` premières lignes d'un fichier ;

`tail -n nlignes` affiche les `nlignes` dernières lignes d'un fichier ;

`type` permet de différencier les commandes shell internes, externes ou alias ;

`history` liste toutes les commandes tapées par l'utilisateur, stockées dans le fichier `~/.bash_history` ;

3.1. Droits sur les fichiers

`chmod <droits> <fichier>` modifie les droits pour le propriétaire (`u`), le groupe (`g`) et les autres (`o`), en les ajoutant (`+`), les enlevant (`-`). L'option `-R` applique les modifications récursivement. Ex. `chmod -R ug+x *`

`chown user:group <fichier>` modifie le propriétaire et le groupe d'un fichier. Seuls `root` et le propriétaire du fichier peuvent effectuer cette opération.

`umask` positionne le masque des droits par défaut. Noter que le droit `x` n'est jamais mis automatiquement pour un fichier.

3.2. Caractères de citation (quote)

Les apostrophes `'...'` traitent tous les éléments de la chaîne indifféremment. `''...''` interprète les caractères `$` et `‘. ‘...‘` ou `$(...)` évaluent leur contenu dans un sous shell. Les commentaires sont introduits par #.

3.3. Redirections

Tout processus possède une entrée standard par défaut et une sortie standard, par défaut le clavier et l'écran. Ces périphériques peuvent être redéfinis à l'aide de redirections. Les identifiants suivant définissent les entrées/sorties classiques :

0 descripteur de l'entrée standard ;
1 sortie standard ;
2 sortie erreur.

Les redirections possibles sont :

— `> fichier` redirige la sortie standard dans `fichier` ;
— `>> fichier` ajoute la sortie standard dans `fichier` ;
— `< fichier` redirige l'entrée standard depuis `fichier` ;
— `cmd1 | cmd2` le tube (pipe) `|` connecte la sortie standard de `cmd1` avec l'entrée standard de `cmd2`. Ce mécanisme permet d'enchaîner des commandes sans stocker de résultat intermédiaire ;
— `tee fichier` duplique l'entrée standard dans `fichier` et la retranscrit sur la sortie standard ;
— `>&descripteur` redirige la sortie standard vers le fichier correspondant à `descripteur`.
— `commande << EOF` redirige les lignes qui suivent la commande vers l'entrée standard. Pour refermer l'entrée, saisir une ligne qui commence par EOF ;
— `exec 0<fichier` permet de rediriger l'entrée standard de toutes les commandes qui suivent ;
— `exec 1>fichier` redirige la sortie standard de toutes les commandes qui suivent.

Exemple d'utilisation :

```
exec 3>&1; exec 1>trace; ls; pwd; exec 1>&3 3>&-
```

```
exec 3<&0; ... exec 0<&3 3<&-
```

(On lie le descripteur de fichier n^o 3 avec celui de la sortie standard, et on redirige vers le fichier `trace`. On effectue les commandes `ls` et `pwd`, dont le résultat est dans `trace`. Pour finir, on restaure l'environnement en redirigeant la sortie standard vers le descripteur n^o 3 qui contient la référence à la sortie standard initiale, et on ferme le descripteur 3.)

3.4. Variables

Pour définir une variable, on utilise la syntaxe `variable=valeur`. La valeur de `variable` est `$variable` ou `${nom}`. `unset variable` annule la définition. `eval` permet d'évaluer le résultat d'un commande comme le contenu d'une variable (ex. `A=toto; B='$A'; echo $B; eval echo $B;`). `set` (ou `env`) indique la liste de toutes les variables définies.

Une variable n'est valable que pour le shell qui l'a définie. En particulier, lorsque la variable est définie dans un script, elle n'existe plus lorsqu'il termine. Pour pallier ce manque, précéder la définition du mot clé `export`.

Deux variables particulières : IFS et OFS définissent les caractères séparateurs pour l'entrée et la sortie. `set|grep IFS` est utilisé pour connaître leur valeur sous un format compréhensible.

3.5. Paramètres des commandes

Dans un script `shell`, certaines variables sont liées aux arguments fournis sur la ligne de commande ou ont un comportement particulier :

- `$0` le nom de la commande
- `$1, $2, ...` le premier argument, le second argument, etc.
- `$*` la concaténation de l'ensemble des paramètres, sous la forme d'un seul argument ;
- `$@` chaque paramètre l'un après l'autre ;
- `$#` nombre de paramètres ;
- `$?` code retour de la dernière commande ;
- `$$` identifiant du shell qui exécute le script ;
- `$!` identifiant du dernier processus lancé en arrière plan.

La différence en `$*` et `$@` ne se fait sentir que lorsqu'ils sont entre guillemets :

```
$ cat test.sh
#!/bin/bash

echo '"$*"'
for i in "$*"; do echo $i; done

echo '"$@"'
for i in "$@"; do echo $i; done

$ bash test.sh 1 2 3 4
"$*"
1 2 3 4
"$@"
1
```

```
2
3
4
```

Cela est utile lorsque l'on souhaite appeler un programmer en préservant l'intégrité des arguments.

L'instruction `shift` permet de décaler la liste des arguments. `set arg1 arg2...` permet d'affecter à la volée `$1`, `$2`, etc.

```
$ set ‘type passwd‘; ll $3
-rwsr-xr-x 1 0 0 28480 2007-02-27 08:53 /usr/bin/passwd
```

`read var` lit une ligne sur l'entrée standard et met le résultat dans `var`. `xargs -l commande` applique `commande` à chaque ligne de l'entrée standard.

3.6. Structures de contrôle

Les structures de contrôle suivantes sont disponibles :

- `if liste; then liste; [elif liste; ...] else liste; fi`
- `while expr; do liste; done`
- `until expr; do liste; done`
- `for var [in liste] do liste; done`
- `case expr in modèle) commandes ;; [modèle) commandes;; esac`

`break` permet de sortir brutalement d'une itération, `continue` passe à l'itération suivante.

Une boucle infinie se réalise de la façon suivante : `while : ; do ... done;`. Pour lire un fichier, on boucle comme suit : `while read line; do echo $line; done < bashrc`.

3.6.1. Commande de test

En `shell`, la commande `test` renvoie 0 lorsque la condition testée est vraie. Avec options, elle vérifie si un fichier (`-f`) ou un répertoire (`-d`) existe. `test fich1 -nt fich2` vérifie si `fichier1` est plus récent que `fichier2`.

Elle permet également de tester des chaînes de caractères : `-z chaine` est vrai si sa longueur est nulle, `-n chaine` teste le contraire, on trouve également `ch1 = ch2`, `ch1 != ch2`.

Les nombres sont testés avec les options `-eq, ne, lt, gt, le, ge`. Les opérateurs logiques sont `!`, -a (et), `-o` (ou).

La commande `expr` effectue des calculs arithmétiques.

3.7. Sous-shell

Il est possible de regrouper des commandes à l'aide de `{...}` et `(...)`. Avec les accolades, il s'agit d'un simple regroupement des variables qui ne change pas l'environnement. Avec les parenthèses, c'est un sous-shell qui est lancé. Les variables utilisées dans un sous shell ne sont pas visibles en dehors du code du sous-shell. Elles ne sont pas utilisables par le processus parent, le shell qui a lancé le sous-shell. Elles sont en réalité des variables locales.

```
$ a=y
$ (a=b;echo $a); echo $a
b
y
$ (export a=b;echo $a); echo $a
b
y
$ { a=b;echo $a;}; echo $a
b
b
```

Dans l'exemple ci-dessus, bien noter que `{` doit être suivi d'un espace et la dernière instruction suivie de `;`.

L'utilisation de ces caractères est obligatoire lorsque l'on souhaite conserver la valeur des variables pour la suite de l'exécution.

```
$ type passwd
passwd is /usr/bin/passwd
$ type passwd | read a b c; echo $c;

$ type passwd | { read a b c; echo $c;}
/usr/bin/passwd
```

3.8. Trace d'exécution

`set -x` visualise les opération effectuées par le shell. `bash -xv <script.sh>` exécute le script en traçant les instructions.

3.9. Fonction

Il est possible d'écrire des fonctions, à l'aide de la syntaxe `function nomfonction { ... }`. Notez la présence d'une espace devant l'accolade ouvrante, absolument nécessaire. Lors de l'ap-

pel, des arguments peuvent être indiqués, la syntaxe est `nomfonction arg1 arg2 ...`. Dans la fonction, on accède à ces arguments à l'aide de `$1, $2, ..., $9, $*, $@`. Lorsqu'il y a plus de 9 arguments, utiliser `shift`.

Les variables locales d'une fonction doivent être déclarées avec la commande `local variable=valeur`. Autrement, les variables et leurs valeurs sont partagées entre la fonction et son appelant.

Une fonction peut retourner une valeur avec l'instruction `return valeur` ($valeur \in [0 - 255]$). Après l'appel de la fonction, ce code de retour est disponible dans `$?`.

4. Filtrage de fichiers et de flux, expressions régulières

Le `shell` doit être vu comme un ensemble d'outils permettant de manipuler des fichiers et des flux, puis d'organiser des traitements. Ces outils sont nombreux, nous faisons ci-après un focus sur deux d'entre : `sed` et `awk`.

La liste des commandes que je n'utilise pas régulièrement (car `sed` et `awk` font le taff parfaitement :

- `tr` : transcode des caractères -> `sed`. 'A réserver à la transformation minuscule, majuscule.
  `cat ... | tr [:upper:] [:lower:] > ...`
- `cut` : filtre le colonnes d'un fichier -> `awk`

4.1. Expression régulières en `shell`

Une expression régulière est une chaîne de caractères qui utilise des caractères particuliers pour désigner des modèles d'expression.

Le `shell` offre la possibilité de traiter ces expressions.

- `expr ''bonjour'' : ''.*''` calcule la taille
- `expr ''il fait beau'' : ''\(..\)''` indique "il"
- `expr ''il fait beau'' : ''.*\(....\)''` indique "beau"

(on indique entre parenthèses ce qu'on veut garder).

Modification de la valeur d'une variable :

- `${#var}` longueur de la chaîne
- `${var##model}` retire la plus grande occurrence au début
- `${var#model}` retire la plus petite occurrence au début
- `${var%%model}` retire la plus grande occurrence à la fin
- `${var%model}` retire la plus petite occurrence à la fin

`basename` et `dirname` indiquent le nom et le répertoire du fichier. En particulier, `basename` permet de supprimer l'extension d'un fichier :

```
$ basename index.html .html
index
```

Personnellement, je n'utilise pas ces expressions régulières du `shell`, préférant investir dans du `sed`, qui fera la même chose en plus puissant, en abusant de la technique du

```
variable=‘echo $valeur | sed ...‘
```

4.2. Expression régulière POSIX

POSIX est une famille de normes techniques définie depuis 1988. En particulier, POSIX définit les expressions régulières. Une expression régulière est une chaîne de caractères qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles.

Nous avons déjà rencontré les expressions régulières acceptées par `bash`, par exemple la commande `ls [st]?ell.p*[^f]` matchera avec tout fichier dont le nom commence par `s` ou `t`, dont la troisième lettre est un `e`, les deux suivantes un `l`, l'extension commence par `p` et ne termine pas par `f`. Le nom `shell.ps` convient.

Dans la norme POSIX, les expressions régulières sont plus riches. On y distinguera :

- les ancres : `^` pour début de ligne, `$` pour fin de ligne
- les classes de caractères :
  - `[aeiouy]` : désigne un caractère parmi
  - `[^aeiouy]` : désigne un caractère sauf
  - `.` : le point désigne un caractère quelconque sauf le passage à la ligne
  - `[:lower:]` : une minuscule
  - `[:upper:]` : une majuscule
- les multiplicateurs :
  - `*` : 0 ou plusieurs
  - `+` : 1 ou plusieurs
  - `?` : 0 ou 1
  - `{n,m}` : entre n et m
- les options :
  - `g` : pour un matching global, qui désigne toutes les portions possibles de la ligne
  - `i` : être insensible à la casse

Voir `https://ecampus.unicaen.fr/mod/resource/view.php?id=870175` pour des détails.

De la même façon qu'une chaîne de caractères se note entre guillemets ou entre quote, une expression régulière se note entre *slashes* : `/.../`.

Exemples (voir `https://uibakery.io/regex-library`) :

numéro de téléphone : `/^\+?[1-9][0-9]{7,14}$/`

date : `/^[0-9]{1,2}\/[0-9]{1,2}\/[0-9]{4}$/`

email : `/^\S+@\S+\.\S+$/`

POSIX distingue les expressions régulières des *expressions régulières étendues*. Par exemple, le `+` fait partie des expressions régulières étendues. Lorsqu'on employera une commande (grep, sed, etc.), il faudra si besoin préciser un paramètre pour spécifier que l'on souhaite utiliser les expressions régulières étendues : c'est souvent l'option `-E`.

4.3. `grep`

La commande `grep regexp fichier` est utilisée pour filtrer les lignes du fichier qui correspondent à une expression régulière (pas besoin de la délimiter par des slashes).

```
$ grep <regexp> <fichier>
# ou
$ cat <fichier> | grep <regexp>
```

Les options classiques sont :
- -v affiche les lignes qui ne contiennent pas la chaîne
- -c compte le nombre de lignes qui matchent
- -i pas de distinction majuscules/minuscules
- -n affiche le numéro des lignes
- -l affiche uniquement les noms de fichier

4.4. `sed`

`sed` permet d'exécuter des commandes d'édition sur un fichier ou un flux (à l'aide d'un pipe). Ces commandes concernent la suppression de ligne, le remplacement de certaines expressions, etc. Les commandes sont séparées par des `;`. `sed` raisonne à l'aide d'un *pattern space*, constitué de la ligne lue en entrée, sur lequel il effectue des transformations. Par défaut, il affiche en sortie les modifications effectuées sur la ligne. Avec l'option `-n`, cette sortie par défaut est désactivée (mode silencieux).

Comme `grep`, `sed` traite des fichiers ou des flux :

```
$ sed <script> <fichier>
# ou
$ cat <fichier> | sed <script>
```

Une commande `sed` est identifiée par une lettre. Par exemple :

d : détruire la ligne

p : afficher la ligne

s/regexp/remplacement/option : substituer une expression régulière par un texte

On peut faire précéder chaque commande d'une indication sur les numéros de ligne où appliquer les commandes, sous la forme `adrbasse[,adrhaute]` : c'est l'adressage par ligne.

La commande principale est celle de substitution 's/regexp/remplacement/'. Ex. `ls -1 | sed -n 's/\.htm$/\` filtre le résultat fourni pas `ls` en n'affichant que les fichiers d'extension `.htm` et en remplaçant l'extension par `.html`.

On peut également précéder une commande d'une expression régulière pour préciser qu'elle ne doit s'appliquer que si la ligne lue contient l'expression régulière : c'est l'adressage par motif.

Exemples :

- `sed 1d` : détruire la première ligne
- `sed '$d'` : détruire la dernière ligne
- `sed 2,5d` : détruire les lignes en 2 et 4
- `sed -n 5000p` : afficher la ligne 5000
- `sed 's/<[^>]*>//g` : remplacer les balises par rien (les détruire)

Les parties d'expression régulière qui sont entourées par `\(...\)` sont mémorisées et rappelées par `\1, \2, ....`

Par exemple :

```
$ cat sample.txt
04/Jan/1997:03:35:22
04/Jan/1997:03:35:47
04/Jan/1997:03:35:55
$ sed 's/^..\/\(...\)\/.*/\1/' sample.txt
Jan
Jan
Jan
$ sed 's/.*:\(..\):\(..\):\(..\)/heure \1 minute \2 seconde \3/' sample.txt
heure 03 minute 35 seconde 22
heure 03 minute 35 seconde 47
heure 03 minute 35 seconde 55
```

Les commandes `sed` peuvent être placées dans un fichier. Ce fichier `.sed` contient une commande par ligne ou des commandes séparées par des `';'`. Il est appelé par `sed -f instructions.sed`.

> The sed command uses two work spaces for holding the line being modified : the pattern space, where the selected line is held ; and the hold space, where a line can be stored temporarily.

Quelques exemples :

— `&` symbolise le pattern matché
— `i\` ajoute une ligne blanche
— `=` affiche le numéro de ligne
— `/3/p` affiche une ligne si elle contient '3'
— `w var` écrit le contenu du fichier dans `w`
— `q` fin du script
— `p` affiche le pattern space
— `d` détruit le pattern space
— `a\debut` ajoute la ligne avant chaque lecture
— `i\debut` idem a mais accepte une chaîne vide
— `w vari` ajoute le contenu du pattern space à vari
— `s/^number//w var` remplacement de la chaîne number et écriture du nouveau pattern space dans w
— `r var` écrit la variable sur le flux de sortie
— `t[label]` si il y a eu substitution, branche à label
— `y/i/o/` remplace tous les i par des o
— `n` affiche le pattern space si l'option -n n'est pas activée et remplace le pattern space avec la prochaine ligne. Permet de grouper les traitements pour plusieurs lignes.
— `N` ajoute la prochaine ligne au pattern space en conservant le `\n`. Permet d'effectuer des recherches sur plusieurs lignes.

```
# exemple d'un script qui rassemble les lignes qui se terminent par \
# label début de ligne
:join
# si \ en fin de ligne, enchaîner une série de commandes commençant par N
/\\$/{N
# retirer le \n
s/\\\n//
#on recommence sur la nouvelle ligne ajoutée par N
b join
}
```

FIGURE 4.1. – Exemple de programme `sed`

4.5. `awk`

`awk` est un véritable langage de programmation évolué, mais sa finalité est de traiter des fichiers de lignes structurées.

Chaque programme peut inclure une section `BEGIN` et une section `END`, qui seront exécutées avant et après le traitement des entrées. Il est ainsi possible d'utiliser `awk` comme un langage de programmation traditionnel, en utilisant uniquement la section BEGIN. C'est d'ailleurs l'une de ses raisons d'être, puisqu'`awk` est le « premier » langage à gérer l'airthmétique flottante.

Ce serait cependant détourner `awk` de son usage que de le réserver à ce genre de tâche. `awk` est

conçu pour traiter de la donnée structurée par ligne et il le fait avec efficacité et simplicité.

On réservera donc la structure BEGIN à l'initialisation des variables. Elle peut accéder au variables suivantes (built-in) :

- `FS` est le séparateur de champs (*field separator*)
- `OFS` séparateur de champ en sortie
- `ARGC` indique le nombre d'arguments de la commande qui exécute le script
- `ARGV` est le tableau qui contient les arguments

L'intérêt principal de `awk` est de traiter chaque ligne sous la forme d'un sélecteur de choix. Un programme `awk` est une succesion de paires « `condition { statement }` :

1. condition : analogue à l'expression qu'on met en parenthèse dans un if, utilise les variables
2. statement : une suite d'instructions valorisant les variable ci-dessous.

Dans les statements, les manipulations de champs sont très aisées : `$0` désigne la ligne complète, `$1...$NF` désignent les champs. Les conditions de déclenchement peuvent faire appel à des expressions régulières de la forme `$3 ~ /fr/` voire en utilisant une variable `$3 ~ variable`. Bien sûr, toutes les expressions utilisant les tests classiques sont autorisées.

Les variables built-in véhiculant de l'information :

- `split($2, tab, " ")`
- `NF` nombre de champs de la ligne lue (#fields)
- `FNR` nombre de lignes traitées dans le fichier courant (#rows)
- `NR` le nombre de lignes traitées depuis le début de l'exécution

`awk` gère les fonctions, mais pas les booléens.

Pour faire des traitements plus orientés caractères que champ, utiliser `perl`, un langage de haut niveau.

```
#!/usr/bin/awk

BEGIN{
    # ARGC
    # ARGV
    # awk considère chaque argument comme un input file
    # pour utiliser des paramètres, les mettre en fin de ligne de commande
    # les lire
    # décrémenter le ARGC, sinon awk cles considèrera
    # comme des fichiers d'entrée

    seuil = ARGV[2]
    ARGC --

    # remplace le paramètre -F, : input Field Separator
    FS = ","

    # output field separator
    OFS = "@"
```

```
    v = 1

    # chaîne
    # par défaut contenation comme dans le shell
    # l = split()
    # sub, substr, RE
    # match : ~

    # tableau : tab associatifs
    # tab[cle] = val
    # for (t in tab)
    # if (key in tab)

    # piper les print dans du shell
    # ex. noms de fichiers dynamiques :
    # filename = "toto" n
    # print ... >> filename
    # print ... | sort

    # jointure
    # dans le begin, lire et hasher l'un des fichiers
    while ((getline line < "countries.csv") > 0){
        split(line, tab, " ")
        countries[tab[1]] = tab[2]
    }
}



# ne s'exécute que sur la première ligne
NR == 1 {
    ...         # faire un truc spécial
    next        # aller à la ligne suivante
                # sans tester les autres cas
}

# si le champ 3 matche la regexp
$3 ~ /.../{
    print
}

# si le champ 1 est égal à une variable
$1 == maVariable {
     ...
}

# à faire tout le temps
```

```
{
    print $0, plus($9, $10)
    # afficher sans retour chariot : printf
    printf("%d ", $6)   # digit
    printf("%s ", $6)   # chaîne
    printf("%f ", $6)   # flottant
}

# cette section sera exécutée une fois que toutes
# les entrées auront été lues
END{
   ...
}

function plus(a, b){
    if (a > b){
        return a + b
    }else{
        return a - b
    }
}
```

Un bon cours R, XML, SQL, AWK, etc. : `http://www.stat.purdue.edu/~mdw/598/`

5. Gestion des processus

Chaque processus dispose de son propre environnement :
- un numéro d'identification PID ;
- l'identifiant de l'utilisateur qui l'a lancé UID (`id`) ;
- l'identifiant du groupe de l'utilisateur GID (`id`) ;
- le répertoire courant (`pwd`) ;
- les fichiers ouverts par le processus (`lsof`) ;
- le masque de création de fichier (`umask`) ;
- la taille maximale des fichiers que le processus peut créer (`ulimit`) ;
- la priorité du processus ;
- les temps d'exécution ;
- le terminal à partir duquel la commande a été lancée.

Tout processus est lancé par son processus père (PPID), le processus `init` lancé au démarrage n'a pas de père et son PID vaut 1. L'ensemble des processus est donc structuré sous la forme d'un arbre.

La commande `ps` permet de visualiser les processus actifs dans le shell courant. Lorsque l'on souhaite examiner tous les processus, on utilisera `ps aux`, voire `f` pour afficher l'arborescence.

5.1. Signaux

Pour contrôler l'exécution d'un processus, on utilise la commande `kill` qui envoie des signaux. `kill -l` affiche tous les signaux :

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      33) SIGRTMIN    34) SIGRTMIN+1
35) SIGRTMIN+2  36) SIGRTMIN+3  37) SIGRTMIN+4  38) SIGRTMIN+5
39) SIGRTMIN+6  40) SIGRTMIN+7  41) SIGRTMIN+8  42) SIGRTMIN+9
43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13
47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
```

```
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Les signaux les plus courants sont :

- 1 HUP (hang up)
- 2 INT (^C) (interrupt)
- 3 QUIT, idem INT mais génère un *core*
- 9 KILL, ne peut être ignoré par le processus, qui s'arrête de force
- 15 TERM, terminaison soft (signal par défaut)
- 19 STOP (^Z)
- 18 CONT

Chaque processus appartient à celui qui l'a lancé, et il ne répondra qu'aux signaux de son propriétaire (ou de `root`). Un exception cependant concerne les exécutables possédant le droit `x` *set user ID* à la place du droit `x` ; dans ce cas, la personne qui exécute ces programmes acquiert momentanément les droits du propriétaires (cf. commande `passwd`).

5.2. Commandes

`time` : indique le temps utilisé pour exécuter une commande (l'option classique est `-p`) ;

`top` : affiche en temps réel les processus actifs ;

`wait` : demande au shell d'attendre la fin des tâches en arrière plan pour continuer. Utile pour synchroniser des processus lancés en parallèle ;

`nice` : affecte une priorité inférieure ;

`nohup` : lance un processus insensible au signal HUP (envoyé par exemple lorsque l'on ferme la fenêtre de shell). Par défaut, la sortie standard est redirigée vers le fichier `nohup.out` ;

`at heure date` : exécute des commandes en différé, à l'heure précisée ;

`watch commande` : effectue indéfiniment la commande ;

`trap` : indique au processus comment gérer le signal `trap ‘‘echo ^C’’ SIGINT`.

5.3. Accès concurrentiel

Les ressources sont généralement restreintes en nombre d'accès. Pour indiquer qu'une ressource est occupée, on utilisera des fichiers temporaires. Ces *verrous* sont crées pendant le temps où la ressource est utilisée, puis effacés. On utilise par exemple des répertoires, créés par `mkdir` qui retourne une erreur si le répertoire existe déjà.

5.4. Communications

Les processus communiquent entre eux par le biais de signaux, détournés dans le récepteur. L'inconvénient des signaux est que la quantité d'information véhiculée est faible.

On pourra préférer transmettre des informations par l'intermédiaire d'un tube nommé (créé par `mknod [nom] p`). L'accès à ce type de fichier doit être restreint à un seul utilisateur simultané.

5.5. Application - 1

Lancer en parallèle un nombre maximum de processus et attendre qu'ils aient fini avant d'en relancer une série (on se base sur la fin du dernier de la série) :

```
#!/bin/bash

MAX_PROC=10
nb_proc=0
last_pid=0
for i in $(find ...)
do
    echo $i
    if [ $nb_proc == $MAX_PROC ]
    then
        wait $last_pid
        nb_proc=0
    fi
    <traitement de $i> &
    last_pid=$!
    nb_proc=$(($nb_proc + 1))
    echo $last_pid $nb_proc
done
```

5.6. Application - 2

On lance en parallèle plusieurs processus et on attend que *chacun* termine :

```
pid=''
for i in 0 1 2 3 4 5; do
    echo "tâche $i $@" 1>&2
    <tâche $i $@> &
    pid="$! $pid"
done

for i in $pid; do
    wait $i
done
```

6. Tips

Installer des clés ssh (marre de taper les mots de passe ?)

Le principe de ssh (ou sftp, scp) permet de s'affranchir de la présentation du mot de passe, à condition d'avoir au préalable déposé votre clef publique sur la machine à laquelle vous souhaitez vous connecter. Pour cela :

- exécutez les commandes :" ssh-keygen -t rsa" , et "ssh-keygen -t dsa"
- des fichiers ont été créés dans votre répertoire (caché) <home>.ssh, en particulier id_rsa.pub et id_dsa.pub. Il s'agit de vos "clefs publiques". Il ne faut pas modifier ces noms de fichiers, car ssh les utilise par défaut.
- créez, dans votre répertoire .ssh, le fichier authorized_keys en y copiant vos clefs publiques : `cat id_rsa.pub id_dsa.pub >authorized_keys`
- Essayez à présent de vous connecter par ssh à une autre machine.

```
# afficher la liste des périphériques de bloc (list block devices)
lsblk

# déclencher la purge des buffers avant de retirer une clé USB
sync

# partitionner ses périphériques
sudo gparted

# graver une ISO
apt-get install unetbootin

# un éditeur de texte disponible en standard dans toutes les distributions
nano

# rendre une variable disponible dans tous les terminaux
# ajouter dans le fichier ~/.bashrc :
export VAR=valeur
# puis recharger le fichier dans l'interpréteur
. ~/.bashrc
# ou
source ~/.bashrc
```

```
# activer la complétion dans l'interpréteur de commande
# il faut interpréter le fichier /etc/bash_completion dans le ~/.bashrc
if [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi

# passer super utilisateur
su
...  # taper le mot de passe de root
su - # permet de conserver l'environnement

# exécuter une commande en étant superutilisateur
sudo <command ...>
#
# Chercher un paquet
apt-get install ... # + tab pour la complétion
ou
# apt-cache search <partie du nom>

# vérifier si un paquet est installé
dpkg -l | grep <paquet>

# installer des clés ssh (marre de taper les mots de passe ?)
Le principe de ssh (ou sftp, scp) permet de s'affranchir de la présentation du mot de passe

    Exécutez les commandes :" ssh-keygen -t rsa" , et "ssh-keygen -t dsa"
    Des fichiers ont été créés dans votre répertoire (caché) <home>.ssh, en particulier id_
    Créez, dans votre répertoire .ssh, le fichier authorized_keys en y copiant vos clefs pub
    cat id_rsa.pub id_dsa.pub >authorized_keys
    Essayez à présent de vous connecter par ssh à une autre machine.




# connaître basename et dirname
$ echo $i
/home/rioultf/svn/m1dnr-res1/systeme/cours/m1dnr-res1.pdf
$ basename $i
m1dnr-res1.pdf
$ basename $i .pdf
m1dnr-res1
$ dirname $i
/home/rioultf/svn/m1dnr-res1/systeme/cours

# compiler/installer un paquet avec make
./configure --prefix=/usr/local
make
```

```
sudo make install # si prefix d'installation inaccessible

# avec cmake
mkdir build && cd build
cmake ..
make

# se passer de souris dans un terminal
# - aller en début et fin de ligne ^A ^E
# - détruire du curseur à la fin de ligne : ^K
# - détruire du curseur au début de ligne : ^U
# - chercher dans l'historique des commandes : ^R
# - rejouer la commande commenaçant par a : !a
# - insérer des caractères non imprimables dans la console ^V tab
# - remplacer des chaînes complexes (RE, CR) -> emacs ou vi

# chercher quel fichier contient une chaîne :
grep -r <mot> .

# utiliser less pour visualiser un fichier
/ pour la recherche, n pour next
q quit
^l affiche la ligne

# écrire dans un fichier :
cat > <fichier>
...
^D

# ajouter à un fichier :
cat >> <fichier>

# faire passer au second plan un programme
^Z       # lance un signal  STOP (19à
bg       # met le process stoppé en BackGround
jobs -l  # affiche la liste des processus en cours dans le terminal
fg       # met en ForeGround la tâche stoppée

# faire une boucle sur un joker de fichiers
for i in *.machin; do echo $i; <traitement> $i > $i.truc; done

# mettre des commandes sur plusieur lignes
java -jar ~/software/clarity-examples/target/info.one-jar.jar $ref\
| awk -f ../filterJson.awk | sed ':a;N;$!ba;s/\n//g'\
|sed 's/{,/{/g;s/,]/],/g;s/,  },},,/}/;s/,    }/}/g'\
| sed 's/\([a-z_]*\):/"\1":/g' \
| sed 's/\\//g'
```

```
# graver une ISO sur une clé USB
unetbootin
```

7. SVN - Subversion

Ce chapitre a été co-écrit avec Bruno Zanuttini et Jean-Luc Lambert.

N.B. Ce document est spécifique à l'utilisation de `subversion` et de `forge` au département d'informatique de l'Université de Caen Basse-Normandie, même si les principes sont généraux. De façon générale, on peut obtenir de l'aide sur les commandes de `svn` avec `svn commit -h` (par exemple).

7.1. Gestion de version

Tout projet est une collaboration. Pour faire collaborer une équipe de développement on utilise des outils de communication et de gestion de projets et des outils de versionning de sources.

A l'université de Caen, l'outil de communication est forge, `https://forge.info.unicaen.fr/`. Forge permet entre autres :

- De faire circuler l'information autour du projet
- De définir les rôles entre les membres de l'équipe projet (manager, développeur, rapporteur)
- De recenser les anomalies, de les attribuer au développeur et de suivre leur résolution
- De disposer d'un dépôt de sources du projet accessible en écriture uniquement aux développeurs et managers et géré par SVN

L'outil de versionning des sources est Subversion (SVN). Les outils de versionning ne sont que des gestionnaires d'arborescence de fichiers dotés de fonctionnalités supplémentaires :

- Ils gardent en mémoire toutes les modifications effectuées
- Ils permettent de revenir à une version antérieure des fichiers ou des arborescences
- Ils permettent d'éviter les conflits d'écriture sur un même fichier par deux développeurs différents

Voici un lien vers la doc complète de SVN : `http://svnbook.red-bean.com/nightly/fr/svn-book.pdf` Pour obtenir de l'aide sur une commande, taper : `svn nom de commande -h`

7.2. Fonctionnement de SVN

SVN, abréviation de Apache SubVersioN, est un gestionnaire de version pour les projets informatique. Son rôle est de fournir aux développeurs un outil où ils peuvent :

1. déposer / sauvegarder des fichiers ;
2. obtenir l'historique des modifications, leur date, auteur, les détails ;
3. revenir en arrière dans l'arbre des versions ;
4. gérer les conflits de modification.

SVN considère la *ligne* comme degré de *modification* d'un fichier : la date et la taille ne sont pas des indicateurs prioritaires. L'énorme avantage de SVN est de permettre, même si c'est déconseillé, à deux personnes d'éditer en même temps le même fichier au format texte, sous réserve qu'ils interviennent à des endroits différents et que les modifications ne soient pas trop importantes. SVN utilise l'utilitaire `diff` qui caractérise très finement les différences entre deux fichiers, et `patch` qui applique les différences à un fichier pour le mettre à jour.

Pour rendre ses services, la partie `svn-server` met à disposition des connexions réseau vers un *dépôt* des fichiers. La partie `svn-client` fournit au développeur une interface de commande vers le serveur. Des interfaces graphiques sont disponibles, par exemple sur la forge.

Même si l'on travaille seul, l'utilisation de SVN est très utile : elle apporte beaucoup de sécurité dans la gestion de l'historique du projet, au détriment d'une certaine souplesse dans l'organisation.

7.2.1. SVN - Fonctionnement

SVN est un couple client - serveur :

— le serveur fournit le service de stockage des fichiers et de leur historique. Il ne fait rien tant que le client n'effectue pas une requête ;
— le client est utilisé par l'humain, au travers d'une interface graphique ou en ligne de commande, pour effectuer des requêtes de modification du dépôt.

Le serveur gère un *dépôt*, sa structure précise n'a pas besoin d'être connue.

Le client possède une copie *locale* d'une version du dépôt. Il y effectue des modifications : ajout / suppression de fichiers, édition, *etc.* Pour reporter les modifications locales sur le serveur, il faut effectuer la commande `svn commit`.

La version locale du dépôt est une arborescence de fichiers, bâtie par l'utilisateur. Le logiciel `svn` effectue l'administration en créant dans chaque répertoire un dossier `.svn`, qui contient lui-même des *meta-données* sous forme de quelques fichiers, dossiers et copies de sauvegarde.

Attention ! `svn` ajoute des meta-données à vos dossiers. Si vous déplacez des dossiers sous SVN dans d'autres dossiers, nettoyez les préalablement.

7.2.2. SVN - Installation

L'installation de SVN nécessite un serveur et un client :

SVN - serveur

Le serveur est souvent fourni par un service, ici la forge du département. On connaît également la forge du libre `http://sourceforge.net/`. Ces services internet permettent de gérer un projet informatique, avec un wiki, des outils de gestion de projet, de signalement de bugs, et fournissent un dépôt SVN.

SVN - Client

Suivant les systèmes, on trouve des clients en ligne de commande ou graphiques :

Linux : installer le paquet `subversion`, une commande `svn` est disponible ;
Windows : on peut installer `Tortoise SVN`, qui ajoute des items au menu contextuel de l'explorateur de fichiers pour lancer les commandes SVN.

SVN - sessions d'exemple

On considère avoir créé un projet nommé `PP` sur la forge. La première étape consiste à récupérer une version *locale* du dépôt.

```
# récupérer le dépôt et créer une copie locale de la version originale
svn checkout https://forge.info.unicaen.fr/svn/PP

# on crée quelques fichiers, répertoires
mkdir branches tags trunk  # répertoires classiques
mkdir tags/projet          # création répertoire local
cp -r ~/monProjet tags/projet  # copie de la maquette dans le dépôt

# on ajoute les répertoires et fichiers créés dans la version
svn add branches tags trunk

# on transmet les modifications du dépôt au serveur
```

```
svn commit -m "version initiale"
```

Ci-dessous, un exemple de session de travail courante : le dépot est créé, on dispose d'une version locale, que l'on souhaite modifier :

```
# on commence par synchroniser la version locale avec le serveur
svn update

# modification diverses....

# transmission des modifications au serveur
svn commit -m "version initiale"
```

7.3. Gestion d'un projet sur la forge

7.3.1. Création d'un dépôt (projet) sur la forge

1. Aller sur la forge à l'URL `https://forge.info.unicaen.fr/` (dans un navigateur).
2. Se connecter en utilisant son `EtuP@ss` (en haut à droite de la page).
3. Aller sur l'onglet `Projets` (en haut à gauche).
4. Cliquer sur `Nouveau projet` (en-dessous de la bannière du site).
5. Remplir les champs (laisser `Sous-projet de` vide).
6. Sauvegarder.

7.3.2. Gestion des droits d'accès au projet

1. Aller sur la forge et se connecter.
2. Cliquer sur le projet concerné dans `Aller à un projet` (en haut à droite de la page).
3. Cliquer sur `Configuration` (en bas à gauche de la bannière) puis sur l'onglet `Membres` (sous la bannière).
4. Utiliser les actions proposées sur la page. Le rôle de `manager` donne les droits de configuration du projet, tandis que le rôle de `développeur` donne seulement le droit d'agir sur le contenu du dépôt.

7.3.3. Création d'une copie locale

Le dépôt centralise les fichiers sur le serveur. On ne peut pas modifier les fichiers directement sur le dépôt. Une copie locale est une image du dépôt, qui a besoin d'être créée une et une seule

fois (sauf cas particuliers) sur chaque machine à partir de laquelle on souhaite travailler sur le projet.

Dans un terminal, taper

```
svn checkout https://forge.info.unicaen.fr/svn/PP -username=LL
```

où `PP` est le nom (identifiant) du projet, et `LL` est votre identifiant `EtuP@ss`. Ceci crée un répertoire nommé `PP` dans le répertoire courant.

7.3.4. Consultation et import des révisions

Une consultation simple peut être effectuée à partir de la forge.

Pour la mise à jour de copies locales, dans un terminal :

- se placer dans la copie locale (répertoire `PP` ci-dessus) et taper `svn update` ; ceci met à jour la copie locale en fonction du dépôt. Les fichiers concernés sont affichés, précédés d'une lettre (A pour ajouté à la copie locale, D pour supprimé, G pour fusionné...) ;
- cette opération doit être faite au minimum au début de chaque séance de travail sur le projet ; un conseil est de le faire avant de commencer à travailler sur un nouveau fichier (pour en récupérer la dernière version) ;
- la commande `svn log` permet de visualiser les commentaires laissés par les développeurs ainsi que les responsables, heures, etc. des modifications. Taper `svn log -r HEAD` pour le dernier commentaire.

7.3.5. Modification de la copie locale

Le travail sur les fichiers se fait dans la copie locale, avec les outils habituels (emacs, vi, IDE...).

- Après chaque modification significative, exécuter

  ```
  svn commit -m "Description de la modification en une ligne"
  ```

  Si l'option `-m` est omise, l'éditeur par défaut s'ouvre. C'est souvent `vi`, il faut alors taper `i` (pour « insertion ») puis un message d'une ligne, puis `Esc`, `:`, `wq<Entrée>`.
- À chaque création ou ajout de fichier ou répertoire (par exemple par copie d'un fichier non versionné), taper `svn add chemin` avec le `commit` pour que les fichiers/répertoires soient pris en compte. Il ne faut pas ajouter ainsi les fichiers qui peuvent être générés à partir d'autres, par exemple ne pas versionner les `.class` si les `.java` le sont.
- Utiliser `svn move fichier destination` et `svn remove fichier` au lieu de `mv` et `rm`. Attention, ceci programme le déplacement ou la suppression dans le dépôt, mais également en local.

— Ainsi, en général des fichiers versionnés et des fichiers non-versionnés coexistent. Taper `svn status` pour connaître le statut de chaque fichier (? pour « non versionné », A pour « ajouté depuis le dernier commit », M pour « modifié depuis le dernier commit », etc.).

7.3.6. Résolution de conflits

Lorsqu'un commit échoue, le conflit se résout en général ainsi :

1. taper `svn update`,
2. éditer le fichier en conflit, par exemple avec emacs,
3. repérer les conflits, signalés par

```
<<<<<<<<<<<<<.mine
Ma version
=============
La version du dépôt
>>>>>>>>>>>>>.r n^o de version
```

4. pour chaque conflit, remplacer les lignes ci-dessus par les lignes à conserver (supprimer notamment les lignes `<<<<<<<<<<<<<`, `=============` et `>>>>>>>>>>>>>`),
5. taper `svn resolve -accept working nomFichier`.

En cas de conflit, la commande update met dans le fichier en conflit les marques de conflit et les deux versions du fichier. Il donne alors la possibilité d'éditer le fichier que l'on peut donc modifier.

La commande resolve permet de déclarer à Subversion que le conflit est résolu et qu'il peut accepter le fichier, la commande resolve a la syntaxe suivante :

```
svn resolve --accept working «~nom du fichier~»
```

Le paramètre –accept permet de spécifier quel fichier accepter. Ici c'est le fichier de travail que l'on vient de modifier mais on peut décider de choisir un autre fichier (voir le détail des options en affichant l'aide).

7.4. Erreurs classiques

— Oublier d'ajouter un fichier avec la commande `svn add` : le fichier est dans le dépôt local, mais n'est pas attaché à la version, ni reporté sur le serveur. Pour prévenir, on peut effectuer dans le répertoire en doute la commande `svn status`.
— Renommer des fichiers *à la main* ou pire, des dossiers. Cela revient à supprimer ces éléments de la version locale. Il faut impérativement utiliser `svn mv`. Attention également au comportement de refactoring d'IDE comme Eclipse ou Netbeans.
— Inclure dans le dépôt des éléments calculés, comme des résultats de compilation, des classes et archives Java, *etc.*. Ces calculs doivent être effectués à la demande par le client,

— Ajouter dans un dépôt une partie d'un autre dépôt : les méta-données dans les répertoires `.svn` vont se mélanger. Il faut préalablement exporter la source de son dépôt avec `svn` export|, qui fournit une copie propre à insérer dans le dépôt destination, puis `avn add`.

7.5. En cas de problème

Même lorsqu'on travaille seul sur un dépôt, on peut arriver à bloquer SVN, qui ne peut plus faire les commit, à cause d'incohérence. C'est le cas lorsque qu'on modifie la structure de l'arborescence sans utiliser les commandes `(svn) mv, rm, cp`.

1. souvent, il suffit de remettre le dépôt local à jour avec `svn update`, car l'incohérence vient de fichiers qui manquent.
2. exporter / sauvegarder la partie de l'arborescence incriminée, mettre à jour, commiter puis rajouter la sauvegarde et commiter.

8. Git - gestion sociale de version

Git prend toute sa saveur en conjonction avec GitHub. Entre autres, la gestion du projet peut se faire directement sur le web, par exemple l'édition des fichiers du dépot.

Git permet de gérer les versions localement, sans mise-à-jour sur le serveur de sauvegarde.

Le commit a un sens différent de SVN : il n'est pas lié au point de l'arborescence considéré mais prend tout le dépot en considération.

1. on crée un *repository* sur GitHub.
2. on récupère une version locale du dépot :
   `git clone https://github.com/frioult/clasik.git`
3. pour ajouter des fichiers ou dossiers existant :
   `git add ...`
4. le commit local se fait comme pour SVN :
   `git commit -a -m "..."`
5. la propagation sur le serveur s'obtient par :
   `git push origin master`
6. mettre à jour le dépot courant
   `git pull origin`

A. Travaux dirigés

A.1. Compléter les réponses de la machine

```
1. Redirections, pipes
$ (
> echo bonjour
> echo bonsoir 1>&2
> ) >toto
----------
$ cat toto
----------

2. Évaluation
Compléter les réponses de la machine: __________
$ X=bonjour
$ XY=toto
$ echo $X
----------
$ echo $XY
----------
$ echo ${X}Y
----------
$
$ X=\$Y
$ Y="Je veux imprimer ceci avec $X"
$ echo $X
----------
$ eval echo $X
----------
$
$ # variables "indicees"
$
$ cat tableau
#!/bin/sh
if test $# -ne 2 ; then
echo "usage: $0 <nom> <indice>" >&2
exit 1
fi
eval echo \$$1$2
```

```
$ T1=1; T2=2; T3=3
$ export T1 T2 T3
$ tableau T 2
----------
$ V=‘tableau T 3‘
$ echo $V
----------
$ tableau T 5
----------
$ tableau T
----------
```

3. Problème de () et de {}

```
$ cd; pwd
/users/ens/frioult
$ cd shell
$ pwd; (cd; pwd); pwd
----------
----------
----------
$ pwd; {
> cd; pwd
> } ; pwd
----------
----------
----------
$ cd shell ; pwd
/users/ens/frioult/shell
$ echo $A
----------
$ A=toto
$ echo $A
toto
$ (
> echo $A
> A=titi
> echo $A
> ) ; echo $A
----------
----------
----------
$ {
> echo $A
> A=titi
> echo $A
> } ; echo $A
----------
```

```
----------
----------
$ A=toto
$ cat titi
echo $A
A=titi
echo $A
$ echo $A; sh titi ; echo $A
----------
----------
----------
----------
$ chmod +x titi
$ echo $A; titi ; echo $A
----------
----------
----------
----------
$ export A
$ echo $A; sh titi ; echo $A
----------
----------
----------
----------
```

A.2. Les variables

Créer une variable `toto` ayant comme valeur `bonjour` et une variable `titi` ayant comme valeur `1234`

Tester les commandes suivantes :

1. `echo $toto`
2. `echo $titi`
3. `echo toto`
4. `echo titi`
5. `echo toto $titi`
6. `echo $tototiti`
7. `echo $toto titi`
8. `echo $$toto`
9. `echo $\$toto`
10. `echo \$$toto`

11. `toto=$titi`
12. `echo $toto`

A.3. Procédures shell

Exécutez les lignes de commandes suivantes. Que font elles ?

1. for i in * ; do echo $i ; done
2. for i in '*' ; do echo $i ; done
3. for i in "*" ; do echo $i ; done
4. for i in '*' ; do echo "$i" ; done
5. for i in "*" ; do echo '$i' ; done
6. for i in 1 2 3 4 ; do echo "$i" ; done >toto
7. echo "1 2 3 4" » toto
8. cat toto
9. for i in cat toto ; do echo "$i" ; done
10. for i in 'cat toto' ; do echo "$i" ; done
11. for i in "cat toto" ; do echo "$i" ; done
12. for i in `cat toto` ; do echo "$i" ; done
13. while read n l ; do echo "$n" ; done < toto
14. while read x n l ; do echo "$n" ; done < toto
15. while read x y n l ; do echo "$n" ; done < toto
16. true ; echo $?
17. false ; echo $?
18. echo $$

A.3.1. Exercice

`yes` permet d'écrire continuellement la chaîne passée en paramètre sur la ligne de commande ou simplement 'y'. `head` permet d'écrire le début d'un fichier. Utiliser ces commandes (et un *pipe*) pour écrire 7 fois :

```
je suis le(a) plus beau(belle)
```

B. Travaux pratiques - Manipulations élémentaires

Notation Dans toute la suite nous utiliserons les polices suivantes :

« `police courrier` »	invite de commandes (prompt) des serveurs ou terme informatique
« **police linéale gras** »	commandes tapées par l'utilisateur
« *police linéale italique* »	réponse du serveur
« police normale »	texte normal

B.1. Ouverture de session

Votre nom de *login* est votre numéro d'étudiant *persopass*. Lorsque vous arriverez devant les machines vous ouvrez une session en donnant votre nom de *login*. La machine vous demandera ensuite votre mot de passe ; vous devrez taper « en aveugle » (c'est-à-dire sans écho visible) le mot de passe qui vous est attribué.

Ouvrez un terminal qui permettra de taper des commandes que l'ordinateur va interpréter.

B.2. Avant de commencer un TP

Avant de commencer le TP, nous allons créer un ensemble de répertoires afin de pouvoir travailler en toute tranquillité. Pour cela tapez la commande suivante : **mkdir -p systeme/tp1** et validez par **[entrée]**. Ceci permet de créer un répertoire `systeme` contenant un répertoire `tp1`.

B.3. Se promener dans l'arborescence

La commande `cd` permet de se déplacer d'un répertoire à l'autre. La commande `pwd` permet de vérifier où on se situe dans l'arborescence.

— Tapez **pwd**. Vous voyez s'afficher le nom complet du répertoire dans lequel vous vous trouvez, lequel est votre répertoire personnel.
— Tapez **cd systeme**, puis **pwd**. Vous êtes maintenant dans le répertoire `systeme`.
— Tapez **cd .**, puis **pwd**. Vous êtes toujours dans le répertoire `systeme`.
— Tapez **cd ..**, puis **pwd**. Vous êtes retourné dans votre répertoire personnel.
— Tapez **cd systeme/tp1**, puis **pwd**. Vous êtes maintenant dans le répertoire `tp1`.
— Tapez **cd .**, puis **pwd**. Vous êtes toujours dans le répertoire `tp1`.
— Tapez **cd** , puis **pwd**. Vous êtes retourné dans votre répertoire personnel.
— Dans toute la suite il faudra impérativement se placer dans votre répertoire `tp1` (en utilisant la commande **cd ~/systeme/tp1**).

Remarque : Une astuce : il peut être fastidieux ou source d'erreur, de taper à la main le nom d'un fichier, d'un répertoire, etc, qui existe déjà. Pour pallier cela, vous pouvez utiliser la `complétion automatique`. Par exemple, pour vous placer dans le répertoire `~/systeme/tp1` en partant du répertoire `~` : tapez **cd s** puis appuyez sur la touche `tab` du clavier ; vous devez maintenant voir *cd systeme/*. Appuyez encore sur la touche `tab`...

B.4. Édition d'un fichier

— Lancez un éditeur de texte, par exemple `gedit` en tapant `gedit &` dans un terminal ; terminer la commande par le caractère `&` permet de la lancer en *arrière plan* : le shell n'attend pas qu'elle soit terminée pour rendre la main à l'utilisateur. Si on oublie le `&`, on peut reprendre la main en stoppant la tâche (tapez `^Z`) puis en la mettant en arrière plan avec la commande `bg` (*background*).
— dans la fenêtre de l'éditeur de texte, tapez un petit texte indiquant qui vous êtes. Par exemple :

```
François Rioult
GREYC CNRS UMR 6072 - Campus II - Université de Caen
Bâtiment Sciences 3 - Bureau S3-353
Téléphone : +33 (0) 231-567-379 (bureau)
            +33 (0) 231-567-330 (fax)
email : Francois.Rioult@unicaen.fr
web   : http://rioultf.users.greyc.fr
---
```

— Enregistrez le fichier.

B.5. Affichage du contenu d'un répertoire ou d'un fichier

— Tapez la commande **ls** dans le terminal. Le nom du fichier que vous venez de saisir est affiché. La commande **ls** permet d'afficher le contenu d'un répertoire ;
— Dans le terminal tapez **cat *fichier*** où *`fichier`* est le nom du fichier que vous venez d'éditer. Le contenu du fichier est affiché. La commande **cat** permet de visualiser le contenu d'un fichier (mais pas de le modifier). `more` est utile pour les fichiers longs en introduisant une pagination mais on lui préfèrera l'ergonomie de `less` (un `alias more='less'` est le

bienvenu dans le fichier de configuration `~/.bashrc`.

B.6. Copie et effacement d'un fichier

— Toujours dans votre terminal tapez la commande **cp *fichier* toto** où *fichier* est le nom du fichier que vous avez créé au B.4, puis tapez **ls**. Un nouveau fichier de nom `toto` apparaît. Vous pouvez regarder son contenu à l'aide de la commande **cat**. La commande **cp** permet de copier des fichiers.

— Toujours dans votre terminal tapez la commande **rm *fichier*** où *fichier* est le nom du fichier que vous venez de taper. On vous demande alors (en anglais) si vous voulez supprimer le fichier en question. Faites lui comprendre que oui (en tapant **y**) ! Puis, pour constater que c'est efficace, tapez **ls**. Le fichier que vous aviez créé a disparu. En effet, la commande **rm** permet de supprimer des fichiers.

B.7. Manipulation des fichiers et des répertoires

Remarque : Dans toute la suite, nous aurons besoin des commandes suivantes : **man**, **ls**, **cp**, **rm**, **cat**, **mkdir**, **rmdir**, **mv**, **emacs**.
Pour obtenir de l'aide sur une commande, il existe un manuel en ligne. Pour l'appeler, il suffit de taper **man *commande***. où *commande* est la commande sur laquelle vous désirez obtenir des informations. Par exemple : **"man ls"**. Pour obtenir de l'aide sur le manuel, il suffit de faire **"man man"**. Il existe un autre moyen d'obtenir des informations sur une commande en faisant **info *commande***. Par exemple : **"info ls"**.
N'hésitez pas à consulter l'aide en ligne.

B.7.1. Création et observations de fichiers

Nous allons maintenant créer de nouveaux fichiers. Il est conseillé de vérifier régulièrement le résultat avec les commandes **'ls'**, **'ls -a'**, **'ls -l'** et **'cat'**.

Créez :
— 5 fichiers vides portant le nom que vous voulez ;
— 5 fichiers vides dont le nom commence par un 'a' *(par exemple : a1 a2 a3 a4 aPartCaCaVaBien)* ;
— 5 fichiers vides dont le nom ne commence pas par un 'a' *(par exemple : min1 min2 B3 E4 Maisaquoicasert)* ;
— 5 fichiers dont le contenu est leur nom ;
— 5 fichiers qui contiennent exactement 5 caractères ;
— 5 fichiers dont le nom commence par un point ('.').

Remarque : Sans doute trouverez-vous fastidieux de taper à la suite 30 commandes très semblables. Pour vous aidez, regardez ce qui se produit lorsque vous appuyez sur les flèches orientées vers le haut et vers le bas...

— Listez tous les fichiers dont le nom commence par un 'a' et seulement ceux-là ;
— Listez tous les fichiers dont le nom commence par un point ('.').

B.7.2. Création de répertoires

Créez les trois répertoires suivants : `A`, `autres`, `.Point`.

B.7.3. Copie et déplacement de fichiers et de répertoires

— Copiez le fichier créé au B.4 dans le répertoire `~` avec le nom `'signature'` ;
— copiez tous les fichiers dont le nom commencent par un `a` dans le répertoire `A` ;
— déplacez tous les fichiers ne commençant pas par un `a` dans le répertoire `autres` ;
— copiez tous les fichiers commençant par un point dans le répertoire `.Point` ;
— déplacez le répertoire `A` dans le répertoire `autres`.
— recopiez le contenu du répertoire `tp1` dans un autre répertoire `tp2`. Le répertoire `tp2` doit être placé dans le répertoire `systeme`.

B.7.4. Effacer les fichiers et les répertoires

— Effacez tous les fichiers qui commencent par `a` dans le répertoire `a` ;
— comment faire pour effacer tous les fichiers du répertoire `autres` sans être obligé de confirmer l'effacement ?
— effacez le répertoire `A` ;
— comment faire pour effacer le répertoire `autres` en une seule opération ?

B.8. Fichiers de configuration

Analysez le contenu des fichiers `.bashrc`, `.bash_profile`, `.bash_history`, `/etc/fstab`, `/etc/passwd`, `/etc/shadow`.

Vérifiez l'espace disponible en Mo sur chaque ressource à l'aide de `df`. Calculez l'espace disque utilisé sur votre compte en Mo. Vérifiez votre quota disque avec `quota` et la taille limite d'un fichier avec `ulimit`.

B.9. Exercices supplémentaires

B.9.1. Exercice 1 : Manipuler les raccourcis claviers sous linux et quelques commandes linux pratiques

Bien organiser son espace de travail, c'est se simplifier la vie par la suite. Il existe des astuces pratiques pour organiser son espace de travail comme celle que nous allons voir pour positionner ses fenêtres sur un quart d'écran / demi-écran. Sélectionner une fenêtre, cliquer sur la barre du haut de la fenêtre avec le clic droit, garder enfoncer le clic droit et déplacer dans les coins la fenêtre.

Que se passe-t-il ?

Pour la suite du TP, organiser votre espace de travail de la façon suivante :
- un terminal sur un quart de l'écran
- le sujet du tp sur une demi-partie de l'écran.

Lancer un terminal avec le raccourci : *CTRL+ALT+T* Pour fermer un terminal : *CTRL+D* Lorsqu'un terminal est ouvert, il est très important de se souvenir de certaines commandes. Parmi ces commandes à ne jamais oublier quand on utilise un terminal :

- *CTRL+C* : arrête le processus en cours
- *CTRL+S* : gèle l'affichage de l'écran =======> *CTRL+Q* : reprendre l'affichage normal

Exécuter la commande : *cat /dev/urandom* Que se passe-t-il ? Vous avez perdu la main sur le terminal. Tester les raccourcis pour geler l'affichage de l'écran, puis reprendre l'affichage normal. Enfin, stopper ce processus. La deuxième partie de cet exercice présente une manière d'afficher le contenu d'un fichier. Les plus utilisées sont less (more), et cat. Exécuter les commandes suivantes et expliquer ce qu'elles font :
- cat /usr/share/dict/words
- cat /usr/share/dict/words | head -10
- cat /usr/share/dict/words | tail -10
- less /usr/share/dict/words

Pour quitter less, il suffit d'appuyer sur la lettre "q". "h" pour obtenir le manuel d'aide. Pour effectuer une recherche par exemple, il suffit de taper le caractère "/". Effectuer la recherche "Ab". Pour parcourir les éléments suivants "n", pour revenir aux précédents "N".

B.9.2. Exercice 2 : Utiliser un éditeur de fichier dans un terminal

Lorsque nous utilisons des terminaux, il est souvent nécessaire d'écrire ou d'éditer des fichiers textes depuis le terminal. Il existe plusieurs éditeurs de textes sous lignes de commandes : jed, nano, vim, ... Remarquons, qu'il est important de bien distinguer les commandes pour afficher un fichier (cat, less), des commandes pour éditer d'un fichier(jed, nano, vim). Ainsi, sous linux, il n'est pas commode d'utiliser une commande comme nano pour afficher le contenu d'un fichier. L'objectif de cet exercice est de découvrir l'un des plus simples à utiliser : nano.

Prise en main de nano

- Ouvrir un terminal et exécuter la commande : *nano*
- Compléter :
  - CTRL + G : Affiche l'aide
  - ? : Permet de sauvegarder votre fichier
  - ? : Permet de quitter nano
  - ? : Permet de faire une recherche
  - CTRL + A : ?
  - CTRL + E : ?
  - CTRL + Y : ?
  - CTRL + V : ?
  - CTRL + _ : ?
  - CTRL + C : ?
  - CTRL + D : ?
  - CTRL + K : ?
- Quitter maintenant nano !

Il existe différentes options d'utilisation de nano. Essayer différentes options et préciser ce que font ces options :

- nano -l
- nano -i
- nano -m

Utiliser le raccourci clavier *CTRL + SHIFT + T* pour ouvrir un nouvel onglet dans votre écran de terminal, pour le fermer, vous pouvez utiliser le raccourci *CTRL + SHIFT + D* pour fermer un onglet ou le terminal. Dans le nouvel écran, créer un fichier texte vide à l'aide de la commande *touch* et nommer le "code.java".

Pour copier en SHELL ou nano : *CTRL + SHIFT + C*

Pour coller en nano : *CTRL + SHIFT + V*

A l'aide de nano éditer le fichier texte et le remplir avec ce code java :

```
// private PropositionalFormula getOneElement() {
// Iterator it = (Iterator) this.iterator();
// if(it.hasNext()) {
// PropositionalFormula prop = ((java.util.Iterator<Set<PropositionalFormula>>)
it).next();
//// return prop;
// } else {
// return null;
// }
// }
```

Enregistrer ce fichier sous le nom "code.java"

Afficher le fichier texte dans le nouvel écran de terminal avec la commande *less* ou *cat* afin de vérifier que vous ayez bien écris dans le fichier.

Pour info, une autre manière pratique pour coller consiste à simplement sélectionner un bout de texte (laisser surligné) puis de cliquer avec la molette de la souris à l'emplacement souhaité.

Ouvrir nano dans l'autre écran et copier/coller le texte affiché du premier écran de terminal dans l'éditeur nano !

Configurer nano

Ces options d'affichage particulières pour nano peuvent être permanentes dans le lancement par défaut de nano. Pour ce faire, il existe un fichier de configuration de nano dans le dossier : /etc/nanorc Ce fichier correspond à la configuration par défaut pour tous les utilisateurs de la machine.

- Afficher le contenu de ce fichier
- Essayer d'éditer ce fichier avec nano. Que se passe-t-il ?
- A quoi correspond le dossier / ?
- copier coller /etc/nanorc dans le dossier sous le nom .nanorc avec la commande destinée à faire ça dans un terminal !
- Pourquoi le "." devant le nom de fichier ?
- Une fois le fichier collé, éditer les lignes en ouvrant avec nano le fichier collé /.nanorc :

```
# set autoindent
...
# set backup
...
# include ...
...
# set linenumbers
...
# set mouse
...
# set matchbrackets "(<[{)>]}"
```

Supprimer les "#" devant.

- Enregistrer les modifications avec nano et relancer nano. Que s'est-il passé ?
- A quoi servaient ces "#" ? Comment appelle-t-on ces lignes "inutiles" dans le jargon de l'informatique ?
- A quoi correspondaient ces lignes ?
- nano peut faire encore plus pour vous, vous pouvez personnaliser la coloration du texte (source : https ://blog.shevarezo.fr/post/2018/01/11/ajouter-coloration-syntaxique-nano-linux).

Nous allons d'abord afficher la coloration des fichiers de code écrits en C, et pour ce faire :

- Créer un dossier /.nano/ avec la bonne commande linux (mkdir)

— Editer le fichier /.nanorc et ajouter la ligne :
— include " /.nano/c.nanorc"
— Que fait cette ligne ajoutée ? ?
— Créer un fichier dans le dossier /.nano/c.nanorc, coller le fichier suivant, puis enregistrer ces modifications :

```
syntax "C" "\.(c(c|pp|xx)?|C)$" "\.(h(h|pp|xx)?|H)$" "\.ii?$" "\.(def)$"
color brightred "\<[A-Z_][0-9A-Z_]+\>"
color green "\<(float|double|bool|char|wchar_t|int|short|long|sizeof|enum|void|static|const
typedef|extern|(un)?signed|inline)\>"
color green "\<((s?size)|(char(16|32))|((u_?)?int(_fast|_least)?(8|16|32|64))|u?int(max|ptr)
color green "\<(class|namespace|template|public|protected|private|typename|this|friend|virtu
mutable|volatile|register|explicit)\>"
color green "\<(for|if|while|do|else|case|default|switch)\>"
color green "\<(try|throw|catch|operator|new|delete)\>"
color green "\<((const|dynamic|reinterpret|static)_cast)\>"
color green "\<(alignas|alignof|asm|auto|compl|concept|constexpr|decltype|export|noexcept|nu
requires|static_assert|thread_local|typeid|override|final)\>"
color green "\<(and|and_eq|bitand|bitor|not|not_eq|or|or_eq|xor|xor_eq)\>"
color brightmagenta "\<(goto|continue|break|return)\>"
color brightcyan "^[[:space:]]*#[[:space:]]*(define|include|(un|ifn?)def|endif|el(if|se)|if
color brightmagenta "'([^'\]|(\\["'abfnrtv\\]))'" "'\\(([0-3]?[0-7]{1,2}))'" "'\\x[0-9A-Fa-f
## GCC builtins
color green "__attribute__[[:space:]]*\(\([^)]*\)\)" "__(aligned|asm|builtin|hidden|inline|p
section|typeof|weak)__"
#Operator Color
color yellow "[.:;,+*|=!\%]" "<" ">" "/" "-" "&"
#Parenthetical Color
color magenta "[(){}]" "\[" "\]"
## String highlighting. You will in general want your comments and
## strings to come last, because syntax highlighting rules will be
## applied in the order they are read in.
color cyan "<[^= ]*>" ""(\\.|[^"])*""
##
## This string is VERY resource intensive!
#color cyan start=""(\\.|[^"])*\\[[:space:]]*$" end="^(\\.|[^"])*""
## Comment highlighting
color brightblue "//.*"
color brightblue start="/\*" end="\*/"
## Trailing whitespace
color ,green "[[:space:]]+$"
```

Effectuer la même procédure pour le fichier /.nano/java.nanorc :

```
## Here is an example for Java.
##
```

```
syntax "Java" "\.java$"
color green "\<(boolean|byte|char|double|float|int|long|new|short|this|transient|void)\>"
color red "\<(break|case|catch|continue|default|do|else|finally|for|if|return|switch|throw|t
color cyan "\<(abstract|class|extends|final|implements|import|instanceof|interface|native|pa
private|protected|public|static|strictfp|super|synchronized|throws|volatile)\>"
color red ""[^"]*""
color yellow "\<(true|false|null)\>"
icolor yellow "\b(([1-9][0-9]+)|0+)\.[0-9]+\b" "\b[1-9][0-9]*\b" "\b0[0-7]*\b" "\b0x[1-9a-f]
color blue "//.*"
color blue start="/\*" end="\*/"
color brightblue start="/\*\*" end="\*/"
color ,green "[[:space:]]+$"
```

Avec la commande de recherche, effectuer une commande de "recherche remplacement" du fichier texte pour rechercher les caractères "//" et les remplacer par "" : CTRL + ALT GR + "//" puis entrer ""

Appuyer sur "o" pour "oui" : Que s'est-il passé ?

En java "//" représente les commentaires. Ce sont des lignes qui ne sont jamais considérées par la machine lorsqu'elles sont compilées.

C. Travaux pratiques - Installation de Linux

On installe Linux sur une machine virtuelle. On utilise pour cela une machine virtuelle déjà partiellement configurée, qui enregistre son disque virtuel dans le répertoire `/tmp`, afin de ne pas entamer le quota disque.

Effectuer les opérations suivantes :

1. lancer la commande `virtualbox-createtp --list`, qui affiche la liste des machines virtuelles préconfigurées
2. créer la machine virtuelle de nom `xubuntu` à partir de la machine virtuelle préparée nommée `2021-09-20_ubuntu-20.04-tmp` :
   `$ virtualbox-createtp -c xubuntu 2021-09-20_ubuntu-20.04-tmp`
3. lancer le programme `virtualbox` qui permet de gérer les machines virtuelles
4. sélectionner votre machine `xubuntu` et vérifier dans les paramètres réseaux que la machine est bien pourvue de deux interfaces :
   a) le NAT qui permettra à la machine virtuelle d'utiliser le réseau de l'hôte, par exemple pour accéder à Internet ou pour installer des paquets logiciels ;
   b) le réseau privé hôte qui permettra de se connecter à la machine virtuelle depuis l'hôte.
5. la section Général/Description explique comment se connecter à la machine virtuelle depuis l'hôte : il faut commencer par déterminer son adresse IP puis utiliser `ssh`.

Il faut maintenant installer le système d'exploitation sur la machine virtuelle. Pour cela :

1. dans la configuration, item Stockage, dans le contrôleur IDE, remplacer les attributs du lecteur optique vide par l'ISO (x)Ubuntu disponible dans
   `/var/local/data/torrents/downloads/xubuntu-20.04.3-desktop-amd64.iso`
2. démarrer la machine virtuelle
3. au moment de choisir la configuration du disque dur, choisir "Autre chose"
4. choisir "Nouvelle table de partition" et valider
5. pour créer une partition, cliquer sur "espace libre" puis sur le bouton "+"
6. effectuer l'installation de XUbuntu avec un disque dur repartitionné avec trois partitions primaires :
   a) 10GB `ext4` monté sur /
   b) 1.5GB `ext4` monté sur /home
   c) le reste de *swap*
7. pour valider ce formulaire, passer en mode plein écran (Ctrl droite + F ou Ctrl + Home et menu Ecran) et déplacer la fenêtre en cliquant dedans et en maintenant le bouton Alt enfoncé, de façon à faire apparaître le bouton "Installer maintenant"

8. terminer l'installation en indiquant votre situation géographique, un login et un mot de passe.

Une fois l'installation effectuée, la machine redémarre sur le nouveau système :

1. connectez vous avec les coordonnées précédemment saisies
2. effectuez les mises-à-jour (vous constatez que la liaison réseau fonctionne au travers de la machine hôte)
3. lancez le navigateur web et constatez que la liaison réseau fonctionne au travers de la machine hôte
4. lancez le terminal puis créez quelques dossiers et fichiers dans votre espace personnel
5. installer le paquet `ssh` :
   ```
   $ sudo -i
   ...
   # aptitude install ssh
   ```
6. sur la machine virtuelle, taper la commande suivante pour déterminer l'adresse IP de la machine virtuelle (celle qui utilise l'interface `vboxnet0`) :
   `ip addr`
7. connectez-vous en ssh à la machine virtuelle depuis la machine hôte :
   `ssh <login>@192.168.56.101`

C.1. Pour passer la machine virtuelle en plein écran :

Dans le menu de la fenêtre de la machine virtuelle, cliquez sur `Périphériques/Insérer l'image CD des addit` on vous propose de télécharger mais cela échoue. Sur la machine hôte, téléchargez donc à la main le fichier dans le dossier `/tmp` :

```
$ wget https://download.virtualbox.org/virtualbox/6.1.34/VBoxGuestAdditions_6.1.34.iso
```

Ensuite, dans le menu `Périphériques/Lecteurs optiques` de la fenêtre de la machine virtuelle, choisissez le fichier récemment téléchargé. Une icône de disque apparaît sur le bureau et une fenêtre s'ouvre avec son contenu. Ouvrez un terminal dans ce contenu (bouton de droite -> ouvrir un terminal ici).

```
# installer gcc make perl
$ sudo apt-get install gcc make perl
# lancer le script d'installation
$ sudo ./autorun.sh
```

Vous devez ensuite redémarrer : la machine virtuelle passe bien en plein écran.

C.2. Reconstruction de la machine

Nous allons maintenant vérifier que les partitions sont indépendantes les unes des autres en réinstallant un système (comme si le système était en panne) et en constatant que la partition `/home` n'est pas impactée :

1. dans la configuration, item Stockage, dans le contrôleur IDE, remplacer les attributs du lecteur optique vide par l'ISO xubuntu disponible dans `/var/local/data/torrents/downloads/xubuntu-20.04.3-desktop-amd64.iso`
2. démarrer la machine virtuelle
3. au moment de choisir la configuration du disque dur, choisir "Autre chose"
4. détruire la partition montée sur /
5. créer une partition de 10GB en ext4 montée sur /
6. indiquer que la partition précédemment utilisée pour /home est de type ext4 et à monter sur /home
7. indiquer les mêmes coordonnées de login que précédemment
8. terminer l'installation

Constatez alors que les dossiers et fichiers précédemment créés n'ont pas été détruits bien que le système ait été réinstallé.

IMPORTANT : avant de quitter la salle de TP, supprimez votre machine virtuelle et vérifiez que le dossier `/tmp/<login>_virtualbox-createtp` est supprimé.

D. Travaux pratiques - `bash`

D.1. Les boucles `for`

D.1.1. Déplacement de fichiers

Créer un ensemble d'une dizaine de fichiers portant l'extension `.gif` et créer un répertoire `Images`.

Faire un petit programme `shell` renommant tous les fichiers portant l'extension `.gif` en `.old`

Faire un petit programme `shell` déplaçant tous les fichiers portant l'extension `.gif` dans le répertoire `Images` en les renommant en `.old`

D.1.2. Exercice

Écrire une procédure shell qui va afficher toutes les lignes du fichier dont le nom est passé en argument. Afficher un message d'erreur (sur la sortie d'erreur) si la ligne de commandes ne contient pas un seul argument.

D.1.3. Réalisation d'un compteur

Le but de cette question est de réaliser une boucle sur un certain nombre d'éléments. Nous voulons par exemple afficher les nombres de 1 à 100. Il est bien évident que l'on ne va pas énumérer tous les nombres compris entre 1 et 100. Pour cela nous aurons besoin d'effectuer quelques calculs sur les entiers.

Faites effectuer par le shell les calculs suivants : *La commande* ***`expr`*** *permet d'effectuer des calculs.*

— 1 + 2
— 4 * 5
— 1 - 1
— 1 - 5

Taper les commandes suivantes (on suppose que `toto=5 titi=3 tata=5`).

- `if [ $toto -eq 5 ]; then echo "toto=5"; fi`
- `if [ $titi -le 5 ]; then echo "titi <= 5"; fi`
- `if [ $toto -eq $tata ]; then echo "toto=tata"; fi`
- `if [ $toto -eq $tata ]; then toto=‘expr $toto + 1‘; fi; echo $toto`

Définir une variable toto=5 et lui ajouter 1

Réfléchir à la manière de réaliser une boucle affichant les nombres de 1 à 100.

D.1.4. Exercice

Écrire un procédure shell qui affiche la table de multiplication (de 0 à 10) du nombre passé en argument.

D.1.5. Exercice

Faire une procédure shell qui permet de répéter le nombre de fois donné en argument la commande spécifiée sur le reste de la ligne de commande. Ex. :

```
$ repeat 3 echo bonjour\\
bonjour
bonjour
bonjour
```

Réécrire la commande `repeat` de l'exercice précédent sans utiliser `while`.

D.2. Les commandes `if` et `test`

```
if test -f ~/.bashrc ; then echo "~/.bashrc est un fichier"; fi

toto=truc; if test -f $toto ; then echo "$toto est un fichier"; else
echo "$toto n'est pas un fichier"; fi

if test -d ~/public_html; then echo "~/public_html est un répertoire"; fi

toto=truc; if test -n "$toto" ; then echo "La variable toto n'est pas vide";
else echo "La variable toto est vide"; fi

if test -n "$titi" ; then echo "La variable titi n'est pas vide";
else echo "La variable titi est vide"; fi
```

D.3. Combinaison de `for` et de `if`

D.3.1. Fichier, répertoire ou autres ?

Afficher pour tous les fichiers et répertoires du répertoire courant si l'élément courant est un fichier ou un répertoire.

Par exemple : Supposons que la commande "`ls`" affiche :
`truc.txt toto.machin public_html/ dossier/`

l'ensemble de commandes demandé affichera :

```
truc.txt est un fichier
toto.machin est un fichier
public_html est un répertoire
dossier est un répertoire
```

Conseils : Faire une boucle (`for`) pour l'ensemble des fichiers du répertoire courant. Puis, pour chaque élément tester (`if`) si l'élément est un fichier ou un répertoire.

D.4. Calculs arithmétiques

Étudiez les possibilités de la commande `expr`.

E. Travaux pratiques - `Awk - Sed - Bash`

E.1. Expressions régulières

Qu 1. Le fichier `/etc/passwd` liste les utilisateurs du système. Chaque ligne a le format suivant `nom_utilisateur:mot_de_passe:uuid:guid:commentaire:home:script`.

En utilisant la commande `grep`, récupérer :

1. la ligne qui correspond à l'utilisateur `man` ;
2. la ligne qui correspond à l'utilisateur `bin` ;
3. les lignes dont le champ `commentaire` est non vide.

On rappelle que la commande `grep` avec l'option `-E` utilise la syntaxe des *expressions régulières étendues*.

Qu 2. Le fichier `/usr/share/dict/words` contient la liste des mots d'un dictionnaire. Taper les commandes suivantes et expliquer leur résultat.

```
cat /usr/share/dict/words | grep -E "u.*a(o|a)"
cat /usr/share/dict/words | grep -E "^p[^aeiu]*n$"
cat /usr/share/dict/words | grep -E "^[pr][^f-z]rs?i..$"
cat /usr/share/dict/words | grep -E "[w-z]{3}"
cat /usr/share/dict/words | grep -E "(.)(.)(.)\3\2\1$"
```

Qu 3. Sélectionner dans le fichier `/usr/share/dict/words` :

1. les mots qui contiennent `ck` ;
2. les mots qui contiennent au moins deux `w` ;
3. les nom propres (c-à-d qui commencent par une majuscule) ;
4. les mots qui commencent par `a` et se terminent par `ing` ;
5. les mots qui ne contiennent que des voyelles ;
6. les mots qui possèdent la lettre `q` non suivie de la lettre `u` ;
7. les mots qui possèdent la suite de lettres `foo` ou `bar` ;
8. les mots qui commencent et terminent par la même lettre.

E.2. Le filtre SED

On rappelle la commande substitution qui a pour format `s/motif/remplacement/flag`. Cette commande remplace `motif` décrit sous forme d'une expression régulière par `remplacement` ; `flag` précise si l'on remplace la première (`flag=1`), la deuxième (`flag=2`)..., ou toutes les occurrences du motif (`flag=g`).

Voici un exemple :

```
$ echo "bla.bla, bla,bla" | sed 's/\([,.]\) \?/\1 /g'
bla. bla, bla, bla
$ echo "bla.bla, bla,bla" | sed -r 's/([,.]) ?/\1 /g'
bla. bla, bla, bla
```

Ici le motif représente un signe de ponctuation – virgule ou point – suivi de 0 ou 1 espace ; chacune de ses occurrences est remplacé par ce signe suivi d'un espace. À noter que l'option `-r` permet d'utiliser les expressions régulières étendues pour décrire le motif.

Qu 4. En procédant par étapes, écrire une commande qui remanie une phrase pour respecter les conventions suivantes :

- pas d'espaces multiples entre les mots mais des espaces simples ;
- pas d'espace avant mais un espace après pour le point, la virgule, les parenthèses fermantes ;
- un espace avant mais pas d'espace après pour les parenthèses ouvrantes ;
- un espace avant et un espace après pour le point virgule, deux-points, point d'interrogation.

```
$ echo "Des espaces   en   trop(ou pas assez  ):c'est  laid;très,très laid  ."
      | sed -r "À définir"
Des espaces en trop (ou pas assez) : c' est laid ; très, très laid.
```

La commande `sed /motif/d` permet de supprimer toute ligne qui contient `motif`, à l'inverse, `sed /motif/!d` supprime toute ligne qui ne contient pas `motif`.

Qu 5. Taper les commandes suivantes et expliquer leur résultat.

```
$ sed -r "/^[a-z]/d; /'s$/d" /usr/share/dict/words
$ sed -r '/^bin/!d' /etc/passwd
```

Qu 6. En partant du fichier `/usr/lib/python3.5/tkinter/colorchooser.py`, écrire

1. une commande qui supprime tous les commentaires du fichier ;

2. une commande qui supprime tous les commentaires du fichier et les lignes vides ;
3. une commande qui extrait du fichier le nom de la classe et ses méthodes.

```
$ sed -r "À définir" /usr/lib/python3.5/tkinter/colorchooser.py
class Chooser(Dialog)
    _fixoptions(self)
    _fixresult(self, widget, result)
```

E.3. L'utilitaire AWK

Copier le fichier `/usr/share/texmf/tex/latex/xcolor/svgnam.def` dans le répertoire courant sous le nom `test.txt`

Qu 7. Taper les commandes suivantes et expliquer leur résultat.

```
awk '/Blue/ { print }' test.txt
awk '/^[A-Z]/ { print }' test.txt | head -10
awk 'BEGIN{FS = "[,;]"} $2 == 0 { print $1 }' test.txt
awk 'BEGIN{FS = "[,;]"} { if ($2 == 0) print $1 }' test.txt
awk 'BEGIN{FS = "[,;]"} $2 > .9 && $3 > .9 && $4 > .9 { print $1 }' test.txt
awk 'BEGIN { nb = 0 } /^[A-Z]/ { nb++ } END { print nb }' test.txt
```

Qu 8. Écrire les commandes qui permettent :

1. d'afficher le nom des couleurs qui ne commence ni par `Dark` ni par `Medium` ;
2. d'afficher le nom des couleurs sans rouge ni vert ;
3. d'afficher le nom des les couleurs foncées ;
4. de compter le nombre de couleurs sans bleu.

Récréation Un jeu en ligne avec des expressions régulières `https://regexcrossword.com/`

E.4. Exercice 2

E.4.1. Première partie

Étapes pour la réalisation d'un script transformant un nom de domaine en nom de pays où est attribuée l'adresse IP :

1. Écrivez un programme `awk` qui prend en paramètre une adresse IPv4 et la transforme en son équivalent en nombre entier 32 bits.
2. Récupérer la base de données `http://software77.net/geo-ip/` (colonne de droite, section "Download") qui associe des intervalles d'adresses à des pays (cf. figure E.1).

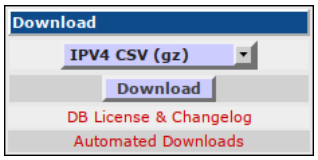

FIGURE E.1. – Fichier à télécharger sur `http://software77.net/geo-ip/`

3. Compléter le programme précédent pour afficher le pays correspondant à une adresse indiquée en paramètre. indiquée.
4. Faire une coquille `shell` qui prend en paramètre un nom de domaine et renvoie son pays d'origine.

Quelques indications supplémentaires

Qu 9. Une adresse IPv4 se code sur 4 octets. Il s'agit de passer de sa notation décimale à point à sa valeur entière. Exemple :

```
$ awk -f qu1.awk
IPv4 : de la notation décimale à point vers le numéro d'identification
255.255.255.255
    4294967295
1.0.0.0
    16777216
0.0.1.0
    256
   (Contrôle-D)
```

Tester et modifier le script suivant à cet effet.

```
function transforme(ip) {
    split(ip,tab,".") ;
    print tab[1], tab[4]
```

```
    return ip
}

BEGIN {
    print "IPv4 : de la notation decimale a point vers le numero d'identificatio
}

{
    print "\t\t"  transforme($0)
}
```

Qu 10. En partant du fichier `IpToCountry.csv`, écrire une commande `sed` pour supprimer toutes les lignes de commentaires ainsi que les guillemets.

```
sed -r 'filtre à définir' IpToCountry.csv | head -3
0,16777215,iana,410227200,ZZ,ZZZ,Reserved
16777216,16777471,apnic,1313020800,AU,AUS,Australia
16777472,16777727,apnic,1302739200,CN,CHN,China
```

Qu 11. Chaque élément de la base IpToCountry.csv a le format suivant
`IP_FROM,IP_TO,REGISTRY,ASSIGNED,CTRY,CNTRY,COUNTRY`
et spécifie le pays correspondant `COUNTRY` à un intervalle de numéros d'identification `[IP_FROM,IP_TO]`. Le but du jeu est de retrouver le pays correspondant à une adresse IPv4 donnée en notation décimale à point :

```
dig +short wikipedia.fr
78.109.84.114
sed -r 'filtre défini qu 2' IpToCountry.csv | awk -f qu3.awk 78.109.84.114
France
```

E.4.2. Deuxième partie

On souhaite analyser un historique de navigation $(user, date, URL)$ comme le montre l'extrait ci-dessous de `/home/etudiants/data/m2-iad/log_anonyme.bz2` :

```
451 [04/Jan/1997:03:35:55 +0100] http://www.netvibes.com
448 [04/Jan/1997:03:36:30 +0100] www.google.com:443
450 [04/Jan/1997:03:36:48 +0100] http://84.55.151.142:8080
452 [04/Jan/1997:03:36:51 +0100] http://127.0.0.1:9010
451 [04/Jan/1997:03:36:55 +0100] http://www.netvibes.com
```

```
453 [04/Jan/1997:03:37:10 +0100] api.del.icio.us:443
453 [04/Jan/1997:03:37:33 +0100] api.del.icio.us:443
448 [04/Jan/1997:03:37:34 +0100] www.google.com:443
```

On commencera par nettoyer à l'aide `sed` le fichier pour ne conserver que les associations $idrootDomain$:

```
451 netvibes.com
448 google.com
451 netvibes.com
448 google.com
```

Puis on générera, à l'aide du script de la première partie, un fichier récapitulant, pour chaque utilisateur, la liste des pays auxquels il a accédé. On utilisera la capacité de Awk à charger, pendant la section BEGIN, un fichier pour construire un tableau associatif. Par exemple, si `countries.csv` contient des associations de type $(domain, country)$, on peut les mémoriser comme suit :

```
BEGIN{
    while ((getline line < "countries.csv") > 0){
        split(line, tab, " ")
        countries[tab[1]] = tab[2]
    }
}
```

F. Travaux pratiques - Parallélisme

F.1. Description

On désire modéliser un système simple d'allocation de ressources :

- un serveur affiche le contenu d'un tube nommé (créé par `mknod [nom] p`). Avant d'écrire dans le tube, le client doit envoyer un message de début et conclure par un message de fin (utiliser les signaux 15 et 16) ;
- plusieurs clients tentent d'écrire dans le tube.

Bien entendu, l'accès au tube est concurrentiel !

F.2. Générateur aléatoire

Extrait de `man urandom` :

> Le générateur de nombres aléatoires regroupe du bruit provenant de son environnement par l'intermédiaire des pilotes de périphériques et d'autres sources, et le stocke dans un réservoir d'entropie. Le générateur mémorise également une estimation du nombre de bits de bruit dans son réservoir d'entropie, et utilise son contenu pour créer des nombres aléatoires.

Expliquer le fonctionnement de la ligne suivante, utilisée pour générer un chiffre aléatoire :

```
cat /dev/urandom | od -d | head -n 1 | sed 's/.*\(.\)$/\1/'
```

Créer un script `random.sh` contenant cette ligne.

F.3. Qualité du générateur aléatoire

Utilisez le script `stat.awk` (code indiqué ci-dessous) pour étudier la qualité de ce générateur aléatoire.

```
for i in $(seq 1000); do ./random.sh ; done | awk -f stat.awk
```

Script awk :

```
#!/bin/awk

{
  if($1 in tab){
    tab[$1] ++;
  }else{
    tab[$1] = 1;
  }
}

END{
  for(i in tab)
    print i " " tab[i];
}
```

F.4. Client

Écrivez un client qui alterne les phases de repos et d'écriture dans le tuyau selon des modes aléatoires.

```
while : ; do
   echo ... > tube
   sleep <seconds>
done
```

F.5. Serveur

Écrivez un serveur qui lit un tube nommé et accuse réception des signaux 15 et 16 (utiliser la commande `trap`), qui doivent débuter et terminer une transmission.

```
trap ... 15
trap ... 16

while : ; do
   read message < tube
   echo $message
done
```

F.6. Accès concurrentiel

Installer un système de verrou pour que plusieurs client puissent être lancés en même temps.

```
if mkdir verrou 2>/dev/null; then
   ...
else
   ... "acces reserve"
fi
```

F.7. Statistiques

Modifier le script `stat.awk` pour qu'il puisse faire des statistiques sur l'allocation des ressources.